Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Fast Detection of Concurrency Errors by State Space Traversal with Randomization and Early Backtracking

Pavel Parízek[1], **Ondřej Lhoták**[2]
[1]Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
[2]David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada

The date of receipt and acceptance will be inserted by the editor

Abstract. State space traversal is a very popular approach to detect concurrency errors and test concurrent programs. However, it is not practically feasible for complex programs with many thread interleavings and a large state space. Many techniques explore only a part of the state space in order to find errors quickly — building upon the observation that errors can often be found in a particular small part of the state space. Great improvements of performance have been achieved also through randomization.

In the context of this research direction, we present the DFS-RB algorithm that augments the standard algorithm for depth-first traversal with early backtracking. Specifically, it is possible to backtrack early from a state before all outgoing transitions have been explored. The DFS-RB algorithm is non-deterministic — it uses random numbers, together with values of several parameters, to determine when and how early backtracking takes place in the search.

To evaluate DFS-RB, we performed a large experimental study with our prototype implementation in Java Pathfinder on several Java programs. The results show that DFS-RB achieves better performance in terms of speed and error detection than many state-of-the-art techniques for many benchmarks in our set. Nevertheless, it is difficult to find a single configuration of DFS-RB that works well for many different benchmarks. We designed a ranking algorithm whose purpose is to identify configurations that yield overall consistently good performance with a small variation.

Key words: state space traversal – randomization – backtracking – concurrency errors

1 Introduction

Efficient detection of bugs in software systems is important because software is widely used and thus errors are costly. Many techniques and tools for detecting bugs are based on the systematic traversal of the state space. We focus on concurrent systems, i.e., on programs with multiple concurrent threads. For concurrent programs, state space traversal techniques explore the program behavior under all possible thread schedules. In this paper, we focus on non-determinism in thread scheduling, and we do not consider data non-determinism. Although state space traversal is popular and has been used to find real concurrency errors, it does not scale well to large and complex systems with many threads. The main reason behind the performance and scalability issues is the huge number of possible thread schedules (interleavings) that exist for any non-trivial program. This problem is often called *state explosion*.

In the past, various optimizations and heuristics have been developed with two goals: (1) to mitigate state explosion and thus improve scalability, and (2) to quickly detect concurrency errors. We categorize the existing techniques into two groups based on their primary approach: the first group of techniques improve speed while completely exploring the state space, and the second group of techniques explore only part of the state space to save time and find errors quickly. In the first group, some complete techniques use heuristics to quickly guide the search towards error states [12, 13, 21, 27, 49, 57]. Other complete approaches traverse the state space in parallel to reduce the time needed to find errors [2, 9, 23, 25, 30, 32, 52].

When full state space exploration is not tractable, techniques in the second group can be used. These techniques are motivated by the assumption that many important errors can be found in a particular small part of the state space. Incomplete techniques include bounding the number of thread context switches [43] in explicit-state model checking [35] and SAT-based model checking [44], delay-bounded scheduling [15], random walk, and methods based on the beam search [5]. Even concolic methods with bounded depth have been used to search for concurrency bugs [17].

Both complete and incomplete techniques can benefit from randomization in various ways, for example to guide the search [8, 46], in combination with parallel search [9], and in a random partial order sampling algorithm [48]. In this paper, we present the concept of *randomized backtracking* in a depth-first algorithm for state space traversal, and its application to fast detection of concurrency errors in multithreaded programs. The key idea is to allow the search algorithm to occasionally backtrack early (at random) even when the current state still has unexplored outgoing transitions. The behavior of the algorithm is controlled by a configuration that consists of the values of several user-defined parameters. In particular, the parameters are used to influence the probability of early backtracking depending on the position of the current state in the program state space.

We implemented the proposed approach in Java Pathfinder (JPF) [63] and evaluated it on several multithreaded Java programs that contain concurrency errors. An important part of our contribution is a ranking algorithm that, based on the results of experiments with randomized backtracking, identifies configurations of randomized backtracking that achieve good performance. By the word *performance*, we mean the ability to find errors together with the speed of detection.

We also compared randomized backtracking with other state-of-the-art techniques for detecting concurrency errors. The results of our experiments show that randomized backtracking helps to achieve better performance than many state-of-the-art techniques on many benchmark programs, i.e. fewer states are explored by JPF before it detects an error. Details are provided in Section 5 and Section 9.

On the other hand, a consequence of randomized backtracking is that an incomplete search is performed, because parts of the state space are pruned by early backtracking from states with unexplored transitions. Some errors may be always missed, but we can at least provide some guarantees on the state space coverage with respect to parameter values (Section 5.3). We also show in Section 6 and Section 9 that it is quite difficult to choose the right configuration that yields good performance for multiple benchmarks. A viable scenario is to run multiple instances of the search procedure, each with different configuration to control the randomized backtracking, for example in an embarrassingly parallel way.

Contribution. This paper extends our previous work that we presented at SPIN 2011 [38]. In that paper, we introduced the concept of randomized backtracking in state space traversal, defined the core algorithm that is controlled with three parameters, and presented the results of initial experiments. New contributions of this paper include the following:

- the design and implementation of several extensions in the form of additional parameters that influence the behavior of state space traversal with randomized backtracking in new ways (Section 4),
- a broader experimental evaluation of the core algorithm together with all the extensions (Section 5),
- a ranking system for automated identification of useful configurations of randomized backtracking that yield good and predictable error detection performance over all benchmarks (Section 6),
- an experimental comparison with a large number of state-of-the-art techniques for detecting concurrency errors, such as guided search with heuristics, dynamic partial order reduction, and systematic concurrency testing with bounded numbers of thread context switches (Section 9), and
- a thorough discussion of the experimental results which highlights the general observations and conclusions (briefly summarized above).

Outline. A more detailed outline of the rest of this paper is as follows. We provide important background information in Section 2 and an overview of the whole approach in the first part of Section 3. We describe the core algorithm for state space traversal with randomized backtracking in detail in Section 3.1 and the proposed extensions in Section 4. The remaining sections present our experimental and qualitative evaluation. Section 5 reports on the experimental evaluation of the core algorithm and extensions. Within this section, we also describe our implementation and benchmark programs (Section 5.1), including the important characteristics of their state spaces. Section 6 defines the ranking system for configurations. In Section 7, we validate the ranking system and discuss its general applicability. We discuss the related work in Section 8 and present the results of our experimental comparison with other approaches in Section 9. Finally, we discuss threats to validity in Section 10 and conclude in Section 11.

2 Background

In this section, we describe the standard depth-first search algorithm (DFS) for systematic traversal of a program state space, and we define important terminology that we will use throughout the paper.

Figure 1 shows the DFS algorithm for state space traversal of multithreaded programs. We kindly ask the reader to ignore the shaded and underlined parts for now. We present a recursive definition of DFS because it allows us to describe the key aspects in a simple and clear way. However, tools such as JPF actually implement depth-first traversal using an iterative approach that is more efficient.

The symbol s in Figure 1 represents program states, and the symbol tr represents a transition enabled in a given state. We consider only explicit state space traversal, where each state is a snapshot of all variables and threads at some point during the program execution. A transition between two states corresponds to the execution of a sequence of instructions (program statements) that ends with a non-deterministic thread scheduling choice. Each transition is associated with some thread, and all instructions in the transition are executed by that one thread.

Exploration starts from the initial state s_0 (line 4), where only the main thread is runnable. The DFS algorithm maintains two data structures: the set $visited$ of states that have already been reached during the traversal (for the purpose of

state matching), and the current state space $path$ represented by a sequence of transitions from the initial state s_0 to the current state. The function push just adds a given transition to the end of the $path$ (line 19), and the function pop simply removes the last transition (line 21). When the algorithm enters a state s that has already been visited, it backtracks immediately by returning from the current level of recursion to the previous one (line 7). In the other case, when the state s has been reached for the first time, the algorithm marks the state as visited (line 8), checks for errors (lines 9-12), and then begins processing the transitions that lead out from s one by one (lines 13-23). Each transition tr enabled in state s is associated with one thread that is runnable in s. When all the outgoing transitions from s have been explored, the whole segment of the program state space rooted at s has been fully processed and the traversal algorithm backtracks from s by returning to the previous level of recursion.

The function enabled (line 13) returns a set of transitions enabled in the state s that must be explored to cover all program behaviors. A typical default implementation of this function returns just the set that contains one transition for each thread runnable in the given state s. The function filter can be used to prune some transitions leading from s. However, its default version preserves all transitions. An important parameter of the state space traversal algorithm is also the *search order* that determines the sequence in which transitions leading from a state are explored — this is implemented by the function order. Heuristics and optimizations of state space traversal are very often based on custom variants of the functions order and filter.

Using the DFS algorithm, a verification tool can systematically explore the program behavior under all possible thread interleavings, and check all reachable program states for concurrency errors and other property violations. Many popular tools, including Java Pathfinder, use a state space traversal procedure that follows the approach described above.

```
1  DFS_RB( C ):
2     visited := {}
3     path := []
4     explore( s0 , C )
5
6  procedure explore( s , C )
7     if s ∈ visited then return
8     visited := visited ∪ s
9     if error( s ) then
10       counterexample := path
11       terminate
12    end if
13    for tr ∈ order( filter( enabled( s ) ) ) do
14       d := depth(path, C)
15       if d >= threshold(path, C) then
16          if rnd(0, 1) > ratio(path, C) return
17       end if
18       s' := execute( s , tr )
19       push( path , tr )
20       explore( s' , C )
21       pop( path )
22       if backtrackAgain(path, C) return
23    end for
24 end proc
```

Fig. 1. DFS-RB: algorithm for depth-first state space traversal with randomized backtracking

3 Randomized Backtracking: Overview

The standard DFS algorithm for state space traversal backtracks only when a state has already been visited, or when a state has been fully processed, in that all of its outgoing transitions have been explored. In our approach, we modified the algorithm so that it may also backtrack early, even when there are still unexplored outgoing transitions from the current state. Fragments of the state space are pruned in that case. The decision whether to backtrack early is influenced by the random number choice, the values of several parameters of the algorithm, and the shape of the program state space. Throughout this paper, we refer to our algorithm as DFS-RB, which stands for depth-first state space traversal with randomized backtracking.

The core of the DFS-RB algorithm, first proposed in [38], has three parameters that control the usage of randomized backtracking during the state space traversal. The parameters are: (1) *threshold*, which enables early backtracking only at a certain search depth (length of the current path), (2) *strategy*, which determines the length of backtrack jumps, and (3) *ratio*, which expresses the preference for going forward over early backtracking from a state with unexplored transitions. In the original variant of the algorithm [38], we supported just constant values for threshold and strategy. For the ratio parameter, we supported constant values and also dynamic values that depend on the current search depth. Although the search depth is defined as the length of the current path by default, there are other options that we discuss in Section 4.

In this paper, we extend the original DFS-RB algorithm [38] with support for the following:

- constant threshold values that represent the minimal number of thread context switches instead of the path length;
- dynamic threshold values that depend on the length of the first explored path;
- dynamic threshold values that depend on the number of thread context switches on the current path;
- dynamic ratio values that depend on the association between transitions and threads on the current path;
- a multiplicative coefficient for the ratio parameter, which is used only when the last two transitions on the current state space path are associated with different threads;
- and, finally, iterative increasing of threshold values.

For some of the parameters, we also define additional possible values that were not used in our previous work. We provide more details on the core algorithm, new extensions, and supported parameters in this section and in the next one.

Specific dynamic values of all the parameters are determined by a *configuration* of randomized backtracking. The main purpose of the experimental evaluation, whose results we present in sections 5, 6 and 9, is to identify useful configurations that yield consistently good error detection performance of state space traversal on our benchmark programs. As indicated above, we say that performance corresponds to the ability to find errors together with the speed of detection. We define the speed of detection (i.e., error finding speed) more precisely as the number of states processed before an error is found and the running time. In general, a lower number of processed states implies better total running time of a verification tool.

3.1 Core DFS-RB Algorithm

Figure 1 shows the core of the DFS-RB algorithm. The differences from the standard DFS algorithm are highlighted by underlining and shading.

The whole DFS-RB algorithm has one parameter, the configuration C of randomized backtracking, which is used to determine the actual values of the conceptual parameters of the algorithm — namely threshold, strategy and ratio. In this section, we discuss how the values of threshold, strategy, and ratio influence the behavior of the overall DFS-RB algorithm, leaving the corresponding functions unimplemented for now. We describe the individual elements of C and the way values of threshold and ratio are computed in Section 4, and there we also show the bodies of functions depth and $\mathrm{backtrackAgain}$.

For each unexplored transition tr from the state s, the algorithm first checks the current search depth against the value of threshold to see whether early backtracking is enabled or not (line 15). When it is enabled, a random number from the interval $\langle 0, 1\rangle$ is generated by the function call $\mathrm{rnd}(0, 1)$ and compared with the ratio value (line 16) in order to decide whether to (a) backtrack early to some previous state on the current path, thus ignoring the remaining unexplored transitions from s, or (b) move forward and execute the transition tr. After each backtracking step, i.e. after the recursive call to $\mathrm{explore}$ returns, the procedure $\mathrm{backtrackAgain}$ is used to determine whether the algorithm should backtrack further according to the selected strategy (line 22). Note that the DFS-RB algorithm does not depend on any specific properties of the functions order and filter, so it can be easily combined with any existing technique that redefines these functions.

Setting the value of *threshold* to a specific non-zero value prevents the algorithm from backtracking too early (at a small search depth). This is useful, for example, when the prefix of each state space path represents a single-threaded initialization phase; in such a case, the algorithm should not backtrack (and prune) too early, before it reaches the part of the state space that captures interleavings of multiple threads, i.e. the part where some concurrency errors may exist.

The decision whether to backtrack is made separately for each outgoing transition from a particular state s. Since the algorithm may decide (with probability $1 - R$) to backtrack at each outgoing transition, the probability that a given transition will be explored depends on its position in the list returned by the order function: the probability that the i-th transition in the list will be explored is R^i, since the algorithm could backtrack at transition i or at any of the $i - 1$ transitions before it.

When the DFS-RB algorithm decides to backtrack early from state s, based on the results of random choice, it can jump back over multiple previous transitions on the current path according to the selected *strategy*. If the given strategy defines a jump of a length greater than the length of the current path, the algorithm backtracks through all transitions on the current path and the state space traversal finishes. All remaining unexplored transitions from the initial state are pruned in this case.

3.2 Research Goals

Here we define research goals concerning the performance and configurations of DFS-RB that we aim to fulfill in this paper.

Goal G0: Analyze the overall performance and practical benefits of the DFS-RB algorithm with respect to error discovery, and its ease of use. Specifically, we want to find out whether DFS-RB can improve speed of error detection over state-of-the-art techniques.

In addition to overall evaluation of DFS-RB, we want to identify specific useful configurations. We divided this task into three research goals G1-G3.

Goal G1: Evaluate selected individual configurations of randomized backtracking with respect to their performance. Determine which configurations have the best (very good) error detection performance — for each benchmark separately and overall.

Goal G2: Find configurations that yield consistently good performance for many benchmarks, and with a small variability. Usage of such configurations may lead to high predictability with respect to error detection in a reasonable time.

Goal G3: Compare the consistently good configurations of randomized backtracking against state-of-the-art techniques for detecting concurrency bugs, integrate randomized backtracking with some of the existing techniques and look for possible impacts on performance that such integration may have.

3.3 Incomplete Search

A consequence of the use of randomized backtracking is that the state space traversal procedure performs an incomplete search, because parts of the state space may be pruned by early backtracking. Errors may be discovered faster due to early backtracking if the DFS-RB algorithm prunes large state

space fragments that do not contain any error states, and therefore avoids spending a lot of time exploring them. On the other hand, the algorithm may also prune state space fragments that contain error states, but that is not a problem as long as some other errors are reached and reported to the user.

Our assumption is that error states are typically evenly distributed over the whole state space of a given program (i.e., over all paths in the state space). For additional details and concrete numbers, see the results of our state space analysis for benchmark programs in Section 5.1. The state space traversal procedure can backtrack early from a particular state and still reach an error, because there are typically many other state space fragments with error paths that will be explored after early backtracking. In other words, even when some error states are pruned, the procedure may reach different ones.

However, we would like to emphasize that, in general, we cannot guarantee that an error state will be always reached when randomized backtracking is used. For some configurations, it is even very unlikely that an error will be detected — for example, because the configuration allows early backtracking at a very small depth. We discuss the ability of individual configurations to find errors in our evaluation. But first, we provide details on the core DFS-RB algorithm and the newly proposed extensions.

4 Parameter Values and Extensions

In this section, we define the functions that compute the values of the main conceptual parameters of the DFS-RB algorithm (threshold and ratio), the auxiliary functions depth and backtrackAgain, and the new extensions of the original algorithm that were briefly introduced in Section 3. Together, these functions implement the extensions that are one of the new contributions of this paper. The configuration C consists of six variables — thb, thm, thr, rtb, rtc, and stg. The names, semantics, and usage of all these variables are explained in the following subsections. We refer to variables from a configuration C using the dot-notation, i.e. by expressions like $C.thb$.

For each configuration variable, we provide a list of specific concrete values that we use in our experimental evaluation (Section 5 and later). Our general motivation was to choose such values that should work well for a wide range of subject programs with respect to important characteristics, including size and complexity, density of error states (paths), and the length of error paths. The state space characteristics are further discussed in Section 5.1.1. In the process of selecting concrete parameter values, we considered also the design of the DFS-RB algorithm and its extensions, our preliminary experiments, and domain knowledge (experience).

4.1 Computing Threshold

First, we describe how the value of threshold is computed. In the original approach [38], we supported only constant natural numbers. The additional ways of defining the threshold value, introduced by the respective new extensions and described below, are motivated by the need to achieve better performance.

```
1  procedure threshold(path, C)
2    if is_constant(C.thb) then
3      v := C.thb
4    else if C.thb ~ L · f then
5      v := get_first_path_length() · f
6    else if C.thb ~ I : t_1 − ... − t_n then
7      v := extract_sequence(C.thb) [search_count]
8    end if
9    if C.thr = "no context switch" then
10     v := v − num_choices_without_switch(path)
11   end if
12   return v
13 end proc
14
15 procedure depth(path, C)
16   if C.thm = "context switches" then
17     return num_thread_switches(path)
18   else return length(path) // default
19   end if
20 end proc
```

Fig. 2. Procedures for computing threshold values and search depth

Figure 2 shows the body of the function that computes the concrete value of threshold and also the function that returns the current search depth. Both functions accept two arguments — the current path and the configuration C of the DFS-RB algorithm. They use variables $C.thb$, $C.thr$, and $C.thm$ from the configuration.

Value. The *base value of threshold*, represented by the variable $C.thb$, can be specified by the user either as a constant natural number, as a fraction of the length of the first path explored during the traversal, or as a sequence of constant values. The symbol $\sim$ represents pattern matching on the symbolic expressions that are the possible values of $C.thb$. We describe the first two cases here, and the third one (iteration over a sequence) in the next paragraph. The constant number is directly returned as the value of threshold. We use the following constants: 5, 10, 20, 50, and 100.

In the second case, $L \cdot f$, the symbol L represents the length of the first explored path and f is a user-specified real coefficient. The threshold value is computed dynamically during the state space traversal by multiplying L and f. In our experiments, we considered these five possible values of the coefficient f: 0.1, 0.25, 0.33, 0.5, and 0.7. Note that the first path is always run to completion, i.e. it is fully explored up to an end state (or to a visited state in the case of a state space cycle). Our rationale behind this extension is to tune the threshold value to the depth of the state space of a specific program. We use the length of the first explored path to approximate the length of the average path in the state space. The length of paths through the state space varies between programs, so different programs require different threshold

values. If an error state is on the first explored path (a very rare case), then it is detected very fast anyway.

Iteration. Another extension of the original algorithm that we propose is the support for iteratively increasing a threshold value when a run of a verification tool does not find any error within a time limit. The threshold base value in the form of a symbol $\mathrm{I} : \mathrm{t}_1 - \ldots - \mathrm{t}_\mathrm{n}$ defines a sequence $t_1, \ldots, t_n$ of constant natural numbers. In this case, the DFS-RB algorithm is run possibly several times, iterating over the values in the sequence in an ascending order, until it detects an error or reaches the end of the sequence. Each iteration has the same user-defined time bound. When DFS-RB runs out of time in one iteration, the search is completely restarted (i.e., the set of visited states is emptied) with the next threshold value. We reuse all the constant numbers that we picked for threshold base value as elements of the sequence, yielding the symbolic base value $\mathrm{I} : 5 - 10 - 20 - 50 - 100$ that is abbreviated as I in tables with data (Section 5 and Section 9). The variable $search\_count$, used at line 7 in Figure 2, is a global counter that specifies the index of the current run of the state space traversal procedure for the given configuration (within the scope of iteration over $t_1, \ldots, t_n$). This idea was inspired by the iterative context-bounding that is used for the purpose of systematic concurrency testing in CHESS [35, 36]. It should be useful especially when the search skips most of the program state space due to a very small threshold value. The higher values in the sequence guarantee coverage of a larger fragment of the program state space.

Reduction. We also support dynamic reduction of threshold values according to the expression $T - n$, where T stands for the current intermediate threshold value and n is the number of thread scheduling choice points on the current path where a thread context switch did not occur. It is controlled by the variable $C.thr$, standing for *threshold reduction*, that has two possible values: $\mathrm{disabled}$ (abbreviated as d) and "$\mathrm{no\ context\ switch}$" ($\mathrm{ncs}$). This extension permits even earlier backtracking from a particular state space path if many of the thread scheduling decisions on the path choose to continue executing the same thread. Our motivation is that concurrency errors are more likely to occur on execution paths with frequent thread context switches.

Mode. The variable $C.thm$ represents *threshold mode*, which has one of two values: $\mathrm{path\ length}$ (abbreviated as pl) and $\mathrm{context\ switches}$ (cs). In the first case, the search depth and threshold value are defined as the number of transitions on a given path through the state space. In the second case, they are defined as the number of thread context switches on the path.

Remarks. The selection of a threshold base value is especially important, as we indicated in Section 3.1. If the value is too small, the DFS-RB algorithm may backtrack so early that it never explores deep enough in the state space to reach an error state. On the other hand, if the value is too large, the algorithm might never backtrack early, and traverse the same number of states as an exhaustive traversal. That is the main reason why we consider a broad range of constant values and some values based on the length of the first explored path. Note, however, that even though a too small or too large threshold value greatly reduces the likeliness that JPF-RB will reach an error state quickly (or at all), it may certainly still happen — the likeliness depends also on other parameters, including the ratio and strategy.

```
1  procedure ratio(path, C)
2     v := eval(path, C.rtb) // expression
3     t_p := prev_transition_thread(path)
4     t_l := last_transition_thread(path)
5     if t_p ≠ t_l then v := v · C.rtc
6     return v
7  end proc
```

Fig. 3. Procedure for computing the ratio value

4.2 Computing Ratio

The original variant of the DFS-RB algorithm [38] supported two ways of defining the value of ratio — either as a constant number, or as a function of the length of the current path (search depth) represented by the expression $1 - d/c$, where d is the current search depth and c is a constant natural number. The expression $1 - d/c$ makes the likelihood of early backtracking grow with increasing search depth.

Here we propose two new extensions that are related to ratio. Both extensions depend on the association between transitions and threads on the current state space path. Figure 3 shows the body of a function that computes the actual ratio value according to the given configuration.

Base value. A user of the DFS-RB algorithm can choose from several options when defining the *base ratio value*, represented by the variable $C.rtb$. More specifically, the procedure for computing ratio (Figure 3) supports the base value ($C.rtb$) given in one of the following ways:

- as a constant real number from the interval $\langle 0, 1)$,
- the expression $1 - d/c$ over the search depth d and a constant integer number c,
- the expression $1 - r/c$, where r represents the number of consecutive previous transitions on the current path that are associated with the same thread (i.e., the length of a suffix of the current state space path without a thread context switch) and c is a constant natural number,
- and the expression c^r, where r is again the number of consecutive previous transitions associated with the same thread and c is a constant real number smaller than 1.0.

The expressions are evaluated dynamically during the state space traversal by the auxiliary function eval. Our motivation for introducing the new expressions is to increase the likelihood of early backtracking when there is a long sequence

```
1  procedure backtrackAgain(path, C)
2    d := depth(path, C)
3    if d < threshold(path, C) return false
4    if C.stg = "fixed" return false
5    if C.stg = "random" then
6      if rnd(0, 1) > ratio(path, C) return true
7    end if
8    if C.stg = "luby" then
9      if luby_num(total_jump_index) > cur_jump_len then
10       cur_jump_len += 1
11       return true
12     end if
13   end if
14 end proc
```

Fig. 4. Procedure that determines the length of backtrack jumps

of transitions associated with the same thread. When the algorithm backtracks and then starts exploring another path, on which different threads interleave to a greater degree, it increases the chance of hitting a concurrency error of some kind. An advantage of the formula c^r is that it always yields a non-zero chance of going forward, although the chance is very small for long sequences of transitions with the same thread. The expression $1 - r/c$ makes the algorithm always backtrack when $r \geq c$. The list of all concrete base values of ratio that we selected for the purpose of experimental evaluation contains the following expressions: 0.50, 0.75, 0.90, 0.99, $1 - d/20$, $1 - d/50$, $1 - d/100$, $1 - d/1000$, $1 - r/2$, $1 - r/5$, $1 - r/10$, 0.50^r, 0.75^r, 0.90^r, and 0.95^r.

Coefficient. The variable $C.rtc$ represents the multiplicative *coefficient* for the base value that is used only when the thread t_l associated with the last transition on the current path is different from the thread t_p associated with the previous transition. Our motivation behind this feature of the DFS-RB algorithm is to make early backtracking less likely when a thread context switch has occurred just before the last transition. Just after such a context switch, there is a greater chance that the current path may trigger some concurrency error. The default value of the coefficient parameter $C.rtc$ is 1, meaning that the extension is disabled. We use also three other concrete values: 1.1, 1.2, and 1.5.

4.3 Backtracking Strategies

Figure 4 shows the body of backtrackAgain, a function that determines the length of backtrack jumps according to the chosen strategy, which is represented by the symbol $C.stg$. The function is called repeatedly at each level of the state space; as long as it continues to return true, the algorithm continues to backtrack to earlier levels. We support three strategies — fixed, random, and Luby — that are described below in this section. The common property is that a jump must always stop when the current depth becomes smaller than the threshold value.

Under the *fixed strategy*, abbreviated as F in tables with data, the algorithm backtracks over a single transition at a time, so the procedure backtrackAgain always returns false at line 4. Then the whole algorithm decides again whether to go forward along some unexplored transition or backtrack further (lines 14-17 of Figure 1).

When the *random strategy* (R) is used, the result of a random number choice (rnd) is compared against the ratio value, like in the case of all decisions about early backtracking that are based on the random choice. The actual overall effect is that the algorithm backtracks over a random number of transitions at each occasion. We use a random choice with uniform probability distribution of results over the range $[0, 1)$. The only restriction on the length of the backtrack jumps under this strategy is that a jump cannot continue below the threshold.

The *Luby strategy* [33], abbreviated as Lb, is the most complex one. It requires that the algorithm records (i) the total number of backtracking jumps already performed from the start of the state space traversal and (ii) the length of the current jump. For this purpose, we use the global variables $total\_jump\_index$ and cur_jump_len, respectively. The auxiliary function luby_num computes the actual length of a backtrack jump based on the number of already performed jumps. It is equal to the value at the corresponding position in the Luby sequence $l_1, l_2, \ldots$, which is defined by the following expression:

$$l_i = 2^{n-1}, \text{ if } i = 2^n - 1$$
$$l_i = l_{i-2^{n-1}+1} \text{ if } 2^{n-1} \leq i < 2^n - 1$$

The symbol i represents a position in the sequence and n is a natural number. For example, the third backtrack jump will step over two transitions, because the first few elements of the sequence are 1, 1, 2, 1, 1, 2, 4. Given any two natural numbers n and i that satisfy $n \geq i > 0$, there are exactly 2^i elements with the value 2^{n-i} between any pair of elements with the value 2^n, and the element with the value 2^n occurs for the first time at the position $2^{n+1} - 1$. Therefore, the maximal possible length of a backtrack jump under this strategy is also bounded by the number of backtrack jumps that have already been performed.

All three strategies are widely and successfully used in state-of-the-art SAT solvers to control restarts (e.g., [4, 14]) and also in processes of other kinds [34]. In the case of SAT solvers, restarts help when the search process spends lot of time in the part of the state space that does not contain any solution — the goal is to avoid such heavy tails in the probability distribution of the running time [20].

The Luby strategy was originally proposed to speed up randomized algorithms of the Las Vegas type with an unknown probability distribution of the running time. Note that while our application of the Luby strategy in DFS-RB violates some of its requirements on the probability distribution, and therefore certain theoretical properties described in [33] are not preserved, the strategy is still useful for detecting errors as we show later.

5 Evaluation of the DFS-RB Algorithm

The first part of our evaluation consists of experiments with the standalone DFS-RB algorithm, including all extensions. In particular, we compare the bug detection performance of state space traversal under different configurations of randomized backtracking in order to address the research goal G1 that we defined in Section 3.2 — we aim to identify configurations that achieve the best performance, for every individual benchmark separately and overall.

In order to evaluate the performance of state space traversal with randomized backtracking, and to address our research goals, we implemented the DFS-RB algorithm in Java Pathfinder (JPF) [63]. We use the abbreviation JPF-RB for Java Pathfinder combined with the DFS-RB algorithm.

The only change to JPF is the usage of a custom search driver, which performs the algorithm shown in Figure 1 together with all of the extensions described in Section 4. The values of all the search parameters are specified through the configuration mechanism of JPF. The strategies for the length of backtrack jumps are hardwired into the search driver.

All of the experiments described in this section were performed using the release[1] of JPF that was current in June 2012, i.e. at the time when we started collecting data.

5.1 Benchmarks

To experimentally evaluate the DFS-RB algorithm, we selected 9 multithreaded Java programs that were used in many other recent studies on concurrency testing and model checking. The benchmark programs are:

- the Daisy file system [42] originally used as a subject for a verification challenge;
- the Elevator benchmark from the PJBench suite [65];
- five small programs used in a recent comparison of tools for detecting concurrency errors [47] — namely Alarm Clock, Linked List, Producer Consumer, RAX Extended, and Replicated Workers, all of which are publicly available in the CTC repository [62];
- jPapaBench [64] — a plain Java version of the PapaBench benchmark that models an autopilot software for unmanned aerial vehicles;
- and the Monte Carlo benchmark from the Java Grande suite [50].

Table 1 provides basic characteristics of all the benchmark programs — the total number of source code lines (LoC) and the maximal number of concurrently running threads. We manually created artificial race conditions by modifying the scope of synchronized blocks in five of the benchmarks: Alarm Clock, Elevator, Producer Consumer, RAX Extended, and Replicated Workers. The other benchmarks already contained errors (mostly data race conditions) that JPF can detect.

[1] We used the release of JPF corresponding to the commit number 715 in the repository for JPF v6.

Additional information about the benchmark programs is provided in the next section.

5.1.1 State Space Characteristics

Besides the number of source code lines and the number of threads, an important characteristic of programs with respect to evaluation of bug detection techniques is the *error path density*, defined as the percentage of state space paths that contain some error state. A low error path density means that the program contains bugs that are hard to find. The set of benchmarks used for evaluation should contain at least some programs with hard-to-find bugs in order to increase the validity and significance of the experimental results.

We analyzed the state space of every benchmark program to determine the values of the following metrics (state space characteristics): the ratio of error states with respect to the total number of processed states, and the ratio of error paths with respect to the number of all traversed state space paths. The latter metric corresponds to error path density. Our primary motivation behind the analysis was to confirm that some of the benchmarks contain hard-to-find bugs.

The analysis of a program state space was designed in a similar way to the one described in [11]. On every benchmark program, we ran JPF with a random search order and a custom listener that gathered the information necessary to compute the metrics. The main goal of the listener was to collect as much data as possible in a limited time (1 hour) and with practical memory consumption, and therefore it stored only hash values for execution paths and used sampling with a dynamically changing resolution (sampling rate) during the traversal. More specifically, it recorded data samples only in every $x/1000$-th state reached during the traversal, where x represents the current total number of already visited states. Values of $x/1000$ for $x < 1000$ are rounded up to 1. Using this approach to sampling, the analysis starts with a very fine-grained resolution, and gradually decreases the resolution as the explored fraction of the state space gets bigger. We used a random search order instead of the default configuration of JPF in order to increase the chance of reaching error states — we present data that validate this decision in Section 9.

The results of the analysis are inherently approximate due to sampling, but we believe that the time limit of one hour is sufficiently large to allow gathering representative information about the state space of a given program (i.e., to get reasonably precise estimates). Table 1 presents the results for each individual benchmark separately.

We made the following general observations based on the results in Table 1 and a manual inspection of the raw data collected by our listener. For most of the benchmarks, error states are distributed rather uniformly all over the state space and many different paths (thread interleavings) lead to each error state — this validates our assumption given in Section 3.3 and suggests that many error states can be reached even with early backtracking. However, the lengths of error paths exhibit a great variance, as apparent from differences between the minimal and the average length. Due to sampling or low er-

Table 1. Benchmark programs: basic information and state space characteristics

Benchmark	LoC	Threads	Ratio of error states	Ratio of error paths	Minimal length of error path	Average length of error path
Daisy file system	800	2	0.03	0.0000005	326	326
Elevator	300	4	0.0000002	0.00000667	96	113
Alarm Clock	200	3	0.18	0.22	10	141
Linked List	180	2	0.01	0.22	48	317
Producer Consumer	130	7	0.14	0.08	169	221
RAX Extended	150	5	0.05	0.84	11	1477
Replicated Workers	400	6	0.004	0.004	156	514
jPapaBench	4500	7	0.02	1.0	108	19921
Monte Carlo	2500	2	0.0005	0.85	379	7840

ror density, we observed a very small number of error paths in the case of some benchmarks (e.g., Daisy and Elevator). In particular, the minimal length is equal to the average for Daisy, where we recorded just a single error path.

The results of our analysis also indicate that some of our benchmarks, such as Daisy and Elevator, have a very low error path density, and therefore contain hard-to-find bugs. Some other benchmarks — namely Alarm Clock, Linked List, RAX Extended, jPapaBench, and Monte Carlo — have a high percentage of state space paths leading to error states. In the case of jPapaBench, all paths lead to error states, but we observed great variance between their lengths.

5.2 Experiments

When designing our experiments, we tried to follow as closely as possible the general recommendations for evaluating path-sensitive error detection techniques that were proposed by Dwyer et al. [11]. The recommendations most important for us are (i) to report also machine- and system-independent metrics and (ii) to perform experiments on a set of benchmarks that is diverse especially with respect to the density of error paths and states. We already discussed properties of our benchmarks in Section 5.1.1, and enumerate the reported metrics at the end of this section.

Let CFG be the set of all configurations and B be the set of all benchmark programs that we have. The symbol E then denotes the set $E = CFG \times B$ of all possible experiments. For every pair $(cfg, b) \in E$, where $cfg \in CFG$ and $b \in B$, we executed JPF-RB multiple times, using a different random number seed in each run.

The set of configurations contains all possible combinations of the parameter values enumerated in Section 4. The name of each configuration is written as a tuple of actual values in the form: $\langle thb, thm, thr, stg, rtb, rtc \rangle$. In total, we have 7920 configurations that we analyze in our experiments.

All the experiments were performed in two phases. In the first phase, we made only 5 runs of JPF-RB for each experiment $e \in E$, and identified the set $E_V \subseteq E$ of experiments (cfg, b) for which the results have some degree of variability (e.g., different numbers of explored states before reaching an error) and at least some runs of JPF-RB detected an error. Then, as the second phase, we performed 100 runs of JPF-RB only for the experiments in the set E_V to get a larger sample. Executing 100 runs for every experiment in the set E would be too costly, especially in cases where no variability among the runs can be observed. Our assumption is that if none of the initial 5 runs for a given experiment detects an error, then there is a very small chance that an error would be detected in 100 runs. We made a similar assumption regarding the variability of results (numbers of explored states and running times).

We set a time limit of 1 minute for every run of JPF-RB. For configurations involving the threshold base value $\mathrm{I} : 5 - 10 - 20 - 50 - 100$, the time limit was 1 minute for each iteration.

Following the recommendations given in [11], we report the values of machine-independent metrics commonly used for state space exploration, such as the number of states processed before reaching an error and the percentage of JPF-RB runs in which an error was found. The number of explored states approximates the real time quite well according to Jagannath et al. [26]. Note, however, that JPF explicitly saves (and counts) program states only at non-deterministic choices — specifically, at thread scheduling choices in the case of multithreaded programs. Besides the number of states, we also report the total running time in seconds, which is a machine-dependent metric, in order to illustrate the real practical usefulness and cost of the DFS-RB algorithm under different configurations.

5.3 Results and Discussion

Here we present the results of the experiments and discuss the most important observations.

Variability among configurations. First, we observe that there is a great variability of speed between configurations of randomized backtracking.

Table 2 contains data for configurations that produce extremes with respect to the number of explored states — namely, the lowest minimum number of processed states in some run of JPF-RB (marked by the symbol LM in the table), the highest maximum number of processed states (HM), the lo-

west average over all runs of JPF-RB in a given experiment (LA), and the highest average over all runs (HA) — and also data for the configuration where the average number of states represents the median over all configurations for the given benchmark (marked by the symbol MC). Both the lowest minimum and the highest maximum are identified over all configurations and JPF-RB runs, and then we report the configuration for which the particular extreme was observed. Note, however, that we consider only experiments (configurations) where at least 50% of JPF-RB runs detected some error. There are many experiments, some for every benchmark, where the percentage of JPF-RB runs that reached an error is smaller than 50% or even equal to 0%. For some benchmarks, the overall minimum number of processed states was achieved by multiple configurations — in that case, we arbitrarily chose one of these configurations to include in the table. For the metric *number of states*, we report the four basic statistics — average (μ), minimum, maximum, and standard deviation (σ) — while for the *running time* we report just the average and standard deviation.

We made two specific observations based on the results in Table 2.

- For each benchmark, the best average speed, and the other extremes, are achieved by different configurations.
- There is a large difference between all the extremes for each benchmark, which indicates the large variation of speed between configurations.

Figure 5 shows the general trends of average error-detection speed. Each graph presents data for all configurations (x-axis) of the DFS-RB algorithm running on one benchmark. The black line represents the ratio between the average speed (number of explored states) for a given configuration and the median over all configurations (row labeled by "MC" in Table 2). Note, however, that the median is computed only based on data for configurations where at least 50% of JPF runs detected some error (like in the case of Table 2). Configurations are sorted by the value of the ratio in an increasing order. We also do not show ratio values greater than the cut-off point of 10, because otherwise extremes would dominate the shape of the line and obscure the differences among smaller ratio values (at the left side of a graph). In addition, a part of the empty graph segment (without any line) on the right side of a graph represents configurations for which no run of JPF-RB found an error.

For every benchmark, the data in Table 2 and Figure 5 show that error-detection speed is reasonably good for many configurations because the values of our metrics for the median configuration (MC) are quite close to the best recorded values (e.g., the lowest average (LA)). On the other hand, there also exist many configurations that yield bad performance for all of the benchmarks — these correspond to the empty graph segments. The running times of JPF-RB, also shown in Table 2, are proportional to the numbers of explored states.

Variability within a single configuration. The results of our experiments also show that, for many experiments, there is a large variability of error detection speed between JPF-RB runs within a single configuration. The degree of variability is highlighted by the graphs in Figure 6.

To be more specific, Figure 6 contains graphs that show the coefficient of variation in the number of explored states over all JPF-RB runs for a configuration. The coefficient is defined as the ratio between the standard deviation and mean. Each graph presents data for all configurations that are ordered by the respective presented values in an increasing order. To enable easier comparison, all graphs have the same scale in the range $[0, 2]$ on the y-axis. Like in the case of Figure 5, a part of the empty segment (without any line) on the right side of a graph represents configurations for which no run of JPF-RB found an error.

Based on the graphs in Figure 6, we made the following observations regarding the variability in error-detection speed. For some benchmarks, such as Elevator and Replicated Workers, more than half of the configurations have a large internal variability. In particular, we found that for many of those configurations, the coefficient of variation indicates that the standard deviation has a larger absolute value than the mean. On the other hand, for every benchmark, there also exist many configurations (more than 20%) that yield a very small variability. Although there is no clear pattern in the names of such configurations that would be fully applicable to every benchmark, in Section 6 we identify a subset of configurations that produce reasonably small variability for all benchmarks.

Ability to find errors. Our third important observation concerns the ability of individual configurations of the DFS-RB algorithm to find errors. The ability is expressed by the percentage of JPF-RB runs that detected an error (i.e., the percentages of successful JPF-RB runs). Figure 7 provides, for each benchmark, the overall perspective based on the data for all configurations. The configurations are sorted by the percentage of successful runs in a decreasing order, so that configurations with the good result (i.e., that yield a high percentage) are on the left side, in line with the graphs in Figure 5.

The graphs show that the percentage of successful JPF-RB runs varies to a great degree among different configurations; it can be very small or zero. In particular, JPF-RB failed to detect an error in any run for a large number (majority) of configurations for some benchmarks, including Daisy file system, Elevator and jPapaBench.

When inspecting the results of our experiments with randomized backtracking, we also found some correlation between the results and characteristics of individual benchmarks (Section 5.1.1). Very low error path density, which we observed for the benchmarks Daisy and Elevator, corresponds to a rather low number of configurations that enable JPF-RB to detect an error in a given time limit. On the other hand, for benchmarks with high error density, such as Alarm Clock, Linked List, and RAX Extended, we observed that a great majority of configurations yield a similar average performance of JPF-RB. These observations confirm our intuition and expectations with respect to JPF-RB (and bug finders in general).

Table 2. Selected configurations of randomized backtracking

Benchmark	Configuration $\langle thb, thm, thr, stg, rtb, rtc \rangle$	States μ	min	max	σ	Time μ	σ	Found
Daisy file system	LM: $\langle \text{I}, \text{pl}, \text{ncs}, \text{R}, 0.90^r, 1 \rangle$	178061	234	770375	154154.29	26 s	31 s	100 %
	HM: $\langle \text{I}, \text{pl}, \text{d}, \text{F}, 0.75, 1.1 \rangle$	480871	12197	2480220	520660.51	52 s	45 s	99 %
	LA: $\langle 10, \text{pl}, \text{d}, \text{Lb}, 0.90, 1.2 \rangle$	17482	6935	41467	7825.64	1 s	1 s	66 %
	HA: $\langle \text{I}, \text{pl}, \text{d}, \text{F}, 1-d/1000, 1.1 \rangle$	878169	300172	1989673	448758.5	104 s	56 s	55 %
	MC: $\langle L \cdot 0.25, \text{cs}, \text{d}, \text{F}, 1-r/10, 1.2 \rangle$	264395	137254	379313	56908.91	39 s	8 s	69 %
Elevator	LM: $\langle \text{I}, \text{pl}, \text{d}, \text{Lb}, 0.90, 1 \rangle$	277935	56	689888	193714.13	36 s	21 s	65 %
	HM: $\langle \text{I}, \text{pl}, \text{ncs}, \text{F}, 1-d/100, 1.5 \rangle$	559099	232468	1101557	245588.46	69 s	34 s	50 %
	LA: $\langle 5, \text{pl}, \text{d}, \text{Lb}, 1-r/5, 1.5 \rangle$	250	103	738	119.67	1 s	1 s	68 %
	HA: $\langle \text{I}, \text{pl}, \text{d}, \text{F}, 1-d/100, 1.5 \rangle$	640081	334514	1081143	268851.94	61 s	30 s	54 %
	MC: $\langle \text{I}, \text{pl}, \text{d}, \text{R}, 0.50, 1.2 \rangle$	5756	272	13106	2836.11	1 s	1 s	100 %
Alarm Clock	LM: $\langle 5, \text{pl}, \text{d}, \text{F}, 0.50, 1 \rangle$	107	13	912	124.74	1 s	1 s	100 %
	HM: $\langle \text{I}, \text{pl}, \text{ncs}, \text{R}, 1-r/10, 1 \rangle$	68	13	2539	253.59	1 s	1 s	100 %
	LA: $\langle 10, \text{pl}, \text{ncs}, \text{Lb}, 1-r/5, 1.2 \rangle$	21	16	128	14.53	1 s	1 s	100 %
	HA: $\langle \text{I}, \text{cs}, \text{ncs}, \text{R}, 1-r/2, 1 \rangle$	1065	94	1954	658.35	1 s	1 s	100 %
	MC: $\langle 5, \text{cs}, \text{ncs}, \text{R}, 0.90^r, 1.1 \rangle$	138	116	177	12.65	1 s	1 s	100 %
Linked List	LM: $\langle L \cdot 0.25, \text{cs}, \text{d}, \text{F}, 0.75, 1 \rangle$	150	25	712	144.28	1 s	1 s	92 %
	HM: $\langle 5, \text{cs}, \text{ncs}, \text{Lb}, 1-d/50, 1.2 \rangle$	6973	182	28362	10894.76	1 s	2 s	100 %
	LA: $\langle L \cdot 0.5, \text{cs}, \text{d}, \text{R}, 0.75^r, 1.5 \rangle$	52	39	218	19.67	1 s	1 s	100 %
	HA: $\langle \text{I}, \text{cs}, \text{ncs}, \text{Lb}, 1-d/20, 1 \rangle$	25945	166	27909	7069.17	3 s	1 s	100 %
	MC: $\langle L \cdot 0.5, \text{cs}, \text{ncs}, \text{R}, 0.99, 1.1 \rangle$	265	156	528	71.29	1 s	1 s	100 %
Producer Consumer	LM: $\langle 20, \text{pl}, \text{d}, \text{F}, 0.50, 1.2 \rangle$	56	28	197	31.86	1 s	1 s	100 %
	HM: $\langle L \cdot 0.33, \text{pl}, \text{d}, \text{F}, 0.75^r, 1.5 \rangle$	100182	26895	704525	121345.38	8 s	10 s	79 %
	LA: $\langle 20, \text{pl}, \text{d}, \text{R}, 0.75^r, 1.5 \rangle$	37	30	69	7.28	1 s	1 s	100 %
	HA: $\langle L \cdot 0.1, \text{pl}, \text{d}, \text{R}, 0.75^r, 1.2 \rangle$	361490	14553	685924	146641.2	31 s	13 s	75 %
	MC: $\langle 50, \text{pl}, \text{ncs}, \text{R}, 1-r/2, 1 \rangle$	2082	278	2379	348.62	1 s	1 s	100 %
RAX Extended	LM: $\langle \text{I}, \text{pl}, \text{d}, \text{F}, 0.50, 1.1 \rangle$	84	20	403	70.41	1 s	1 s	100 %
	HM: $\langle L \cdot 0.7, \text{pl}, \text{d}, \text{Lb}, 0.50, 1.1 \rangle$	485	68	2521	747.98	1 s	1 s	100 %
	LA: $\langle 20, \text{pl}, \text{d}, \text{F}, 1-r/5, 1.5 \rangle$	22	22	22	0.0	1 s	1 s	100 %
	HA: $\langle L \cdot 0.7, \text{pl}, \text{d}, \text{F}, 1-d/20, 1.5 \rangle$	1911	1911	1911	0.0	1 s	1 s	100 %
	MC: $\langle L \cdot 0.33, \text{pl}, \text{d}, \text{Lb}, 0.95^r, 1 \rangle$	76	68	119	9.33	1 s	1 s	100 %
Replicated Workers	LM: $\langle 10, \text{cs}, \text{d}, \text{F}, 0.50, 1.1 \rangle$	917	48	10283	1596.48	1 s	1 s	90 %
	HM: $\langle \text{I}, \text{pl}, \text{d}, \text{F}, 1-d/1000, 1.2 \rangle$	570410	113	2463250	648277.4	56 s	67 s	95 %
	LA: $\langle 50, \text{pl}, \text{ncs}, \text{F}, 0.95^r, 1.1 \rangle$	128	89	159	16.34	1 s	1 s	54 %
	HA: $\langle \text{I}, \text{pl}, \text{ncs}, \text{F}, 1-d/50, 1 \rangle$	1838504	157	2319496	403219.67	202 s	37 s	100 %
	MC: $\langle L \cdot 0.1, \text{pl}, \text{d}, \text{Lb}, 1-r/5, 1 \rangle$	1569	698	9484	1184.25	1 s	1 s	100 %
jPapaBench	LM: $\langle 20, \text{pl}, \text{ncs}, \text{F}, 1-r/2, 1.1 \rangle$	912	195	2420	399.0	1 s	1 s	100 %
	HM: $\langle \text{I}, \text{pl}, \text{ncs}, \text{R}, 1-d/100, 1.2 \rangle$	962899	33685	1548026	592607.68	91 s	52 s	77 %
	LA: $\langle 10, \text{pl}, \text{ncs}, \text{F}, 1-r/2, 1.5 \rangle$	785	215	2562	307.91	1 s	1 s	100 %
	HA: $\langle \text{I}, \text{pl}, \text{ncs}, \text{R}, 1-d/100, 1.1 \rangle$	1150845	65585	1537950	452529.92	171 s	89 s	77 %
	MC: $\langle L \cdot 0.33, \text{pl}, \text{d}, \text{Lb}, 1-r/10, 1.1 \rangle$	12824	12603	13712	187.21	2 s	1 s	100 %
Monte Carlo	LM: $\langle \text{I}, \text{pl}, \text{d}, \text{R}, 1-d/1000, 1 \rangle$	347	113	1356	183.44	1 s	1 s	100 %
	HM: $\langle L \cdot 0.5, \text{pl}, \text{d}, \text{F}, 0.50^r, 1.2 \rangle$	18451	13206	26009	3223.28	50 s	8 s	55 %
	LA: $\langle 100, \text{pl}, \text{ncs}, \text{R}, 1-r/5, 1.5 \rangle$	144	129	253	29.29	1 s	1 s	100 %
	HA: $\langle L \cdot 0.5, \text{pl}, \text{d}, \text{F}, 1-r/2, 1.5 \rangle$	24536	24432	24655	58.08	56 s	1 s	62 %
	MC: $\langle \text{I}, \text{pl}, \text{d}, \text{Lb}, 1-r/5, 1.1 \rangle$	338	219	390	32.8	1 s	1 s	100 %

Coverage. Here we discuss coverage of the program state space, which is closely associated with the ability to find errors. Although the DFS-RB algorithm may not reach a particular error state because of early backtracking, based on its configuration we can specify fragments of the state space that are guaranteed to be explored during the search for errors. First, the algorithm always explores all states and transitions with depth less than the value of threshold. If the search with a particular threshold value does not report any error, then the state space fragment up to that depth is safe.

Remarks. We also observed that for some benchmarks, the best average speed of error detection is achieved by configurations for which we recorded just over 50-60% of successful JPF-RB runs — for example, see the data for Daisy file system, Elevator, and Replicated Workers. Therefore, we would

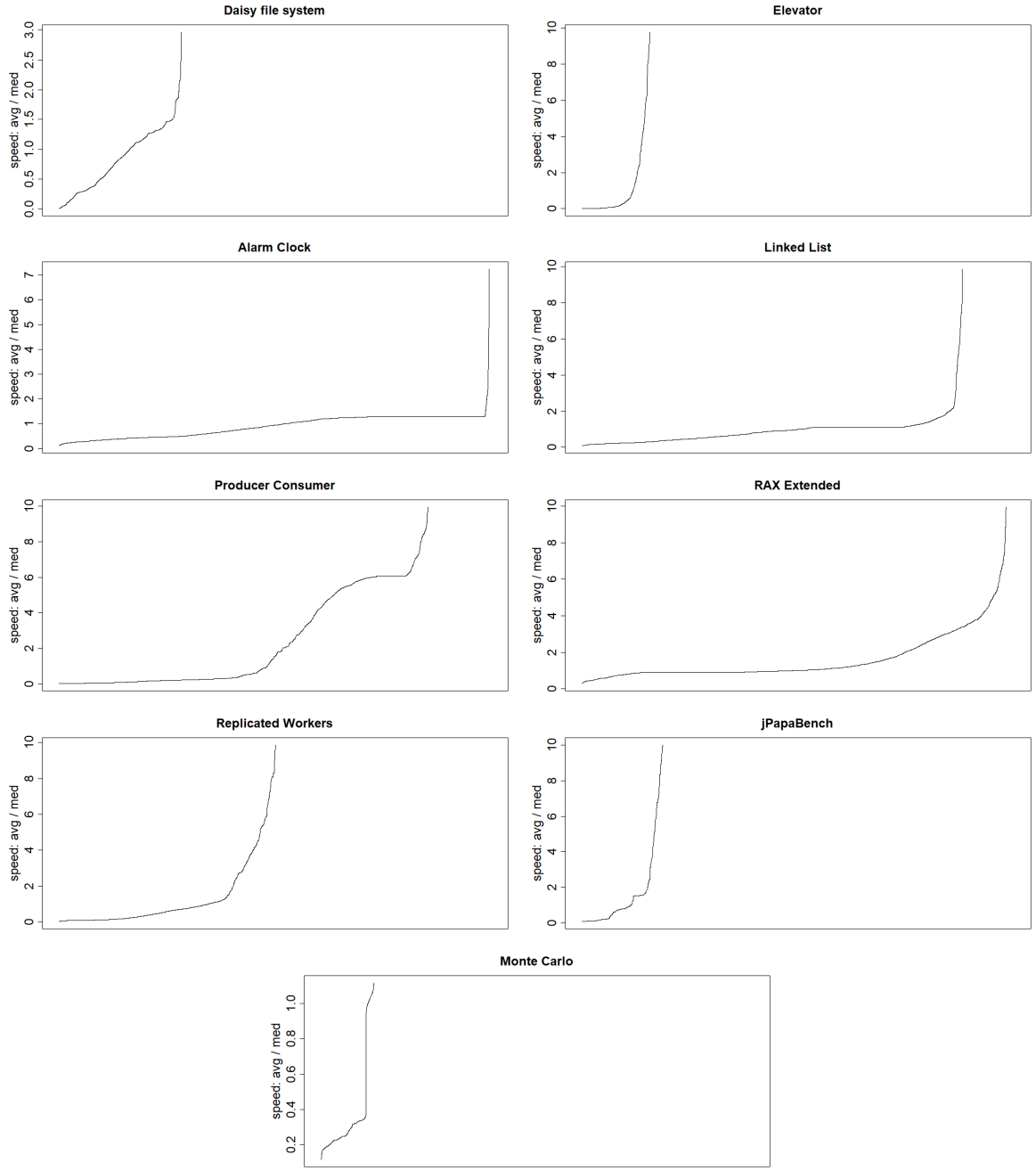

Fig. 5. General trends in error-detection speed over all configurations for individual benchmarks

like to highlight the fact that it is not necessary to consider in practice only those configurations where all (100%) runs of JPF-RB are successful. If for some configuration the percentage of successful JPF-RB runs is greater than 50% and each run finishes quickly, then a sequence of JPF runs with the given configuration would be very likely to find an error quickly.

Summary. To summarize, the raw performance data presented in Table 2 and Figure 5–7, together with the corresponding discussion above in this section, partially fulfill the goal G1 defined in Section 3.2. We pointed out which configuration gives the best performance for each individual benchmark, and we also illustrated the great degree of variability in speed and in the ability to find errors among different configurations and within configurations.

6 Ranking Configurations

In the previous section, we found that the best-performing configuration is different for each benchmark. In this section, we present a ranking system with the goal of systematic identification of useful configurations of DFS-RB that achieve

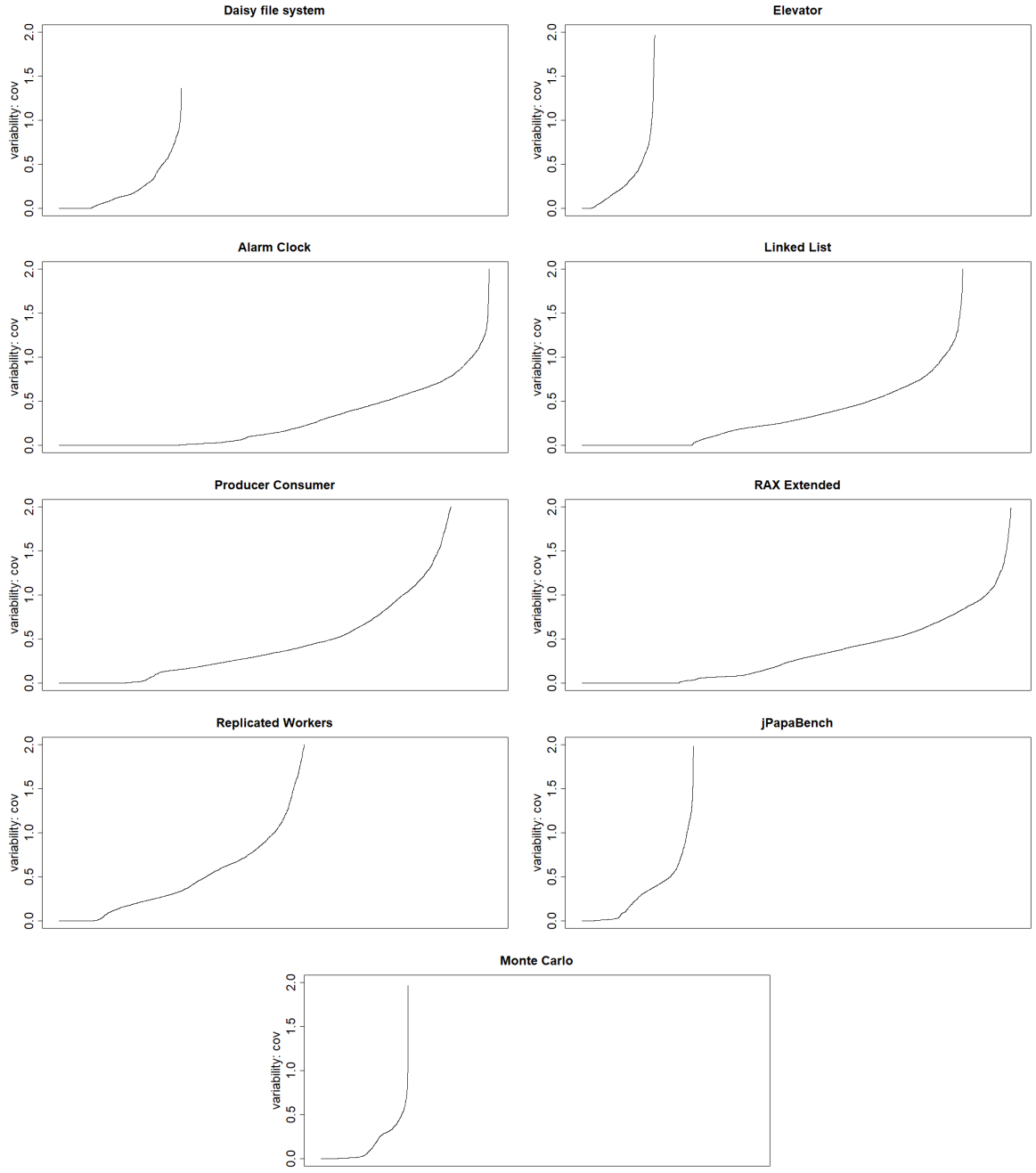

Fig. 6. Variation of error-detection speed within configurations for individual benchmarks

very good performance for multiple benchmarks. In addition, we also identify specific parameters of the DFS-RB algorithm and their values that are the most important for achieving good error detection performance, i.e. parameter values that enable JPF-RB to reach an error state (if present) quite fast. In the next section, we will validate how well our ranking system generalizes to new, unknown bechmarks.

A configuration is deemed to be useful if it satisfies the following properties:

1. JPF-RB detects an error before the time limit in many runs (preferably in all runs) for every benchmark program when using the given configuration.
2. The configuration yields a very high error detection speed on average for all benchmarks, i.e. there is a small average distance from the best recorded speed.
3. There exists a small variation in speed over all benchmark programs and runs of JPF-RB.

The first property represents the ability to find many errors, while the third one implies predictable speed of error detection. We use *distance from the best achieved speed*, a ra-

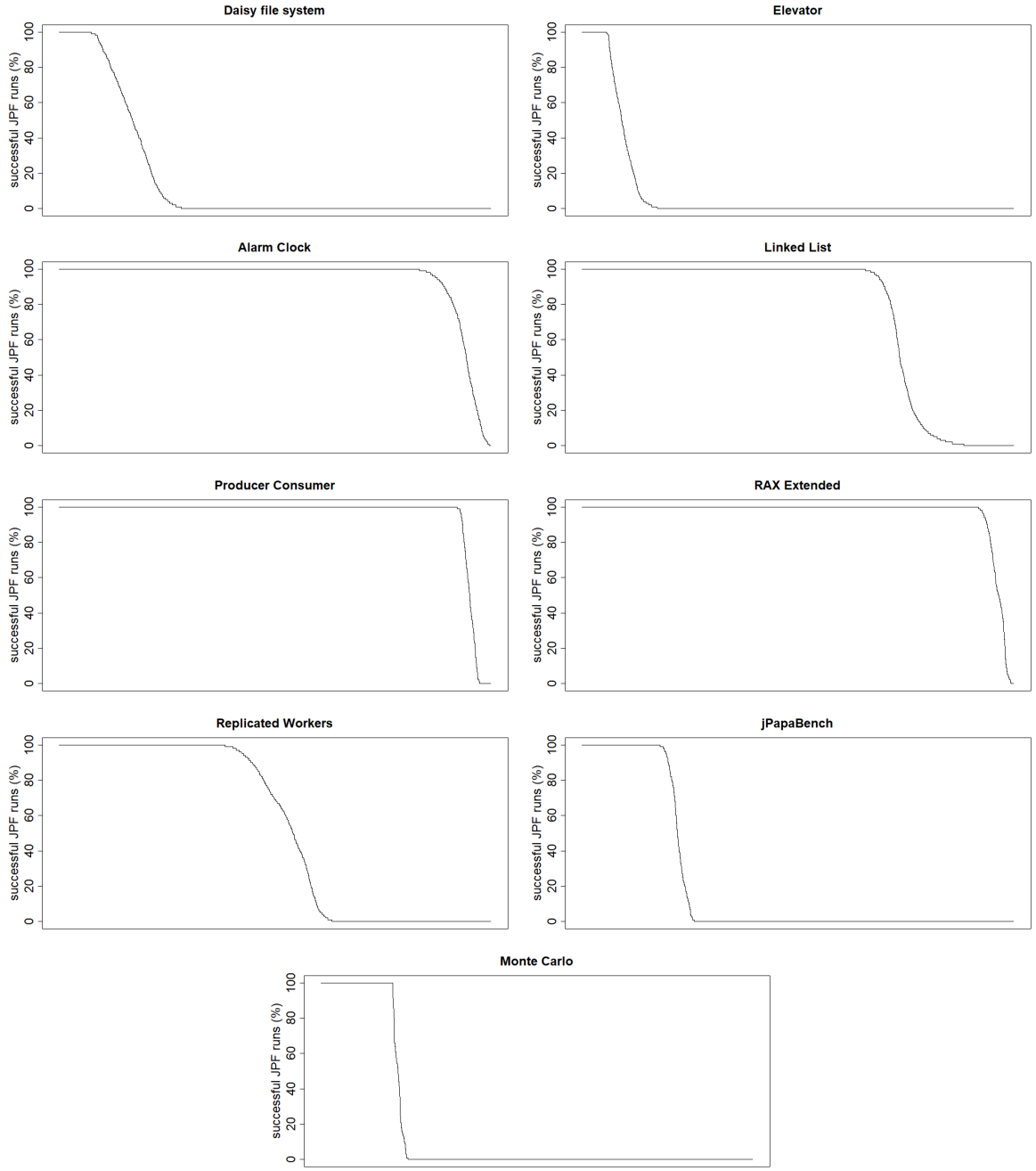

Fig. 7. Ability to find errors for individual benchmarks

tio of the average values for the given configuration and the best one, as a normalized measure (i.e., as the standardized effect size) of error detection speed of individual configurations that is valid over multiple benchmarks. It is necessary to use a normalized distance because the raw data for different benchmarks have a very different scale — for example, 100 states versus 100000.

Our ranking system processes the raw performance data that we used for the evaluation of standalone DFS-RB algorithm in Section 5, and ranks configurations according to the score that is determined based on the criteria that correspond to the properties defined above. More specifically, the criteria are (1) the ability to find errors in many runs of JPF-RB, (2) distance from the configuration that achieves the best speed on a given benchmark, and (3) the degree of variation in speed. An important practical aspect of our ranking system is that useful configurations are identified just once based on the data presented in Section 5.

6.1 Algorithm

We designed an algorithm that computes four ranked lists of individual configurations — one list for each criterion, and a combined list that reflects all of the criteria together.

The algorithm uses the following inputs: the set of all configurations, the set $bench$ of benchmark programs to be considered in a given run of the algorithm, information about the percentage of JPF-RB runs that found some error for a given configuration and benchmark program, the distance from the configuration that yields the best speed for a given benchmark, and data on the variation in speed. The distance and variation are computed over all of the raw data in advance.

A run of the algorithm processes individual configurations one by one in four steps. Each of the first three steps is implemented by a function that encodes the respective criterion and produces a score that reflects how well the given configuration satisfies the criterion. The score determines a rank of the configuration, which corresponds to its position in the given ranking list. We designed the algorithm such that a lower score implies a higher rank.

The result of the first step reflects the percentage of successful JPF-RB runs, i.e. the number of runs that found an error before the time limit, for the given configuration cfg and each relevant benchmark program. We denote the percentage by the symbol $errp(cfg, b)$. The score of a given configuration is the expected number of JPF-RB runs needed to find an error, which is computed by the expression $\prod_{b \in bench} 100/errp(cfg, b)$. This scoring policy gives an infinite penalty to a configuration if, for some benchmark, no run of JPF-RB can successfully detect an error within the time limit.

The second step affects the score of a configuration by its distance from the best speed result for a given benchmark. A smaller distance yields a better score in our system.

Finally, the third step takes into account the degree of variation in speed between different runs of JPF-RB for the given configuration and benchmark. We capture it by the coefficient of variation (cov), i.e. the ratio between the standard deviation and the mean, of the running times of JPF-RB. It is a measure of variance that is by construction normalized over all benchmarks, and therefore can be used to compare the error-detection speed achieved by different configurations. We use the expression $1 + cov$ in order to handle zero variation (or close-to-zero), because the minimal possible (best) score should be 1. Smaller variation implies higher predictability and stability of speed, which are the desired properties and therefore correspond to a better score.

The result of the last step, which is the combined rank of the given configuration, is computed as the sum $r = r_e + r_d + r_v$ of the absolute ranks of the configuration in the three lists that reflect the individual criteria.

We normalized the output scores in the ranking lists that capture the individual criteria such that the best configuration has the score 1 and other scores are modified proportionally.

6.2 Results: Best Configurations

We executed the ranking algorithm on data for all the benchmarks together, and also on data for each individual benchmark separately. However, to save space, we report only the scores and rankings computed over all the benchmarks.

An important fact is that, for 7682 configurations out of the total 7920, there exists a benchmark such that all runs of JPF-RB with the given configuration failed to detect an error within the time limit. We denote such configurations as *failing*, and the remaining 238 as *successful* (for them, on every benchmark at least some runs of JPF-RB detected an error). All failing configurations are placed at the bottom of a ranking list and they have the same score (rank). Note also that the sublists of failing configurations are identical in all ranking lists that are computed over all benchmarks — there can be several such ranking lists, each of them capturing a different subset of the criteria defined above. In the rest of this section and in the next one, we distinguish between (a) the full ranking lists that also include all failing configurations and (b) the sublists that contain only the configurations for which JPF-RB succeeded in some runs on every benchmark.

Table 3 shows the top 10 configurations in each of the three ranking lists that correspond to the individual criteria described in the previous section, together with their respective scores. In the order from the top to the bottom, the fragments of ranking lists shown in the table reflect (1) the ability to find errors, (2) the distance from the best achieved speed, and (3) the variation in speed between JPF-RB runs. For each configuration that appears in some of the ranking list fragments in Table 3, we provide its score according to the respective criterion and also the rank (absolute position) of the configuration in the other two ranking lists.

The data in Table 3 show that all configurations at the top of the ranking lists for (i) the ability to find errors (top segment of the table) and (ii) the distance from the best speed (middle segment) involve the threshold base value $\mathrm{I} : 5 - 10 - 20 - 50 - 100$ and the threshold mode $\mathrm{path\ length}$. Other parameters take many different values in the best configurations. For example, the top segment also shows that a user should take the ratio base value from the set $\{1 - d/100, 1 - r/2, 0.50^r\}$ if his primary criterion is that all (or most) runs of JPF-RB finish before the time limit and report an error (if present). When the user's preference is that at least some runs of JPF-RB achieve speed close to the best recorded, then he should use a ratio base value from the set $\{1 - r/5, 1 - r/10\}$, according to the middle segment.

The bottom segment of Table 3 shows that very low variation is achieved by threshold values in the set $\{L \cdot 0.1, L \cdot 0.25, L \cdot 0.33\}$ together with the threshold mode $\mathrm{path\ length}$, threshold reduction $\mathrm{disabled}$, and the Luby strategy for backtracking jumps.

Table 4 presents the top 10 configurations in the ranking list that is based on the combined score. These are the most useful configurations that achieve overall consistently good and stable error detection performance, i.e. configurations that achieve high speed, detect errors in all (or most) JPF-RB runs

Table 3. Ranking lists of configurations — top 10 by each individual criterion

Finding Errors		
$\langle$I, pl, d, F, 0.50^r, 1$\rangle$	1.00	(168, 118)
$\langle$I, pl, d, F, 0.50^r, 1.1$\rangle$	1.00	(145, 127)
$\langle$I, pl, d, F, $1-d/100$, 1$\rangle$	1.00	(197, 226)
$\langle$I, pl, d, F, $1-d/100$, 1.1$\rangle$	1.00	(166, 157)
$\langle$I, pl, d, F, $1-d/100$, 1.2$\rangle$	1.00	(189, 159)
$\langle$I, pl, d, F, $1-r/2$, 1$\rangle$	1.00	(182, 106)
$\langle$I, pl, d, F, $1-r/2$, 1.1$\rangle$	1.00	(175, 102)
$\langle$I, pl, d, F, $1-r/2$, 1.2$\rangle$	1.00	(160, 111)
$\langle$I, pl, d, Lb, 0.75, 1.2$\rangle$	1.00	(72, 204)
$\langle$I, pl, d, Lb, 0.90^r, 1.1$\rangle$	1.00	(59, 208)
Distance		
$\langle$I, pl, ncs, Lb, 0.95^r, 1.5$\rangle$	1.00	(189.5, 130)
$\langle$I, pl, d, F, $1-r/5$, 1.1$\rangle$	1.76	(87, 169)
$\langle$I, pl, ncs, R, $1-r/10$, 1$\rangle$	1.94	(123.5, 216)
$\langle$I, pl, ncs, F, $1-r/5$, 1.2$\rangle$	2.84	(106, 177)
$\langle$I, pl, d, R, $1-r/5$, 1.1$\rangle$	3.03	(70, 160)
$\langle$I, pl, d, F, $1-r/5$, 1.2$\rangle$	4.01	(104.5, 179)
$\langle$I, pl, ncs, F, 0.75^r, 1$\rangle$	4.81	(91.5, 181)
$\langle$I, pl, d, R, $1-r/10$, 1$\rangle$	4.89	(121, 184)
$\langle$I, pl, d, R, 0.75, 1$\rangle$	6.64	(120, 217)
$\langle$I, pl, d, R, 0.90^r, 1$\rangle$	6.70	(156.5, 183)
Variation		
$\langle L{\cdot}0.25$, pl, d, F, $1-d/100$, 1.2$\rangle$	1.00	(45, 235)
$\langle L{\cdot}0.25$, pl, d, Lb, $1-d/1000$, 1.1$\rangle$	1.09	(215, 218)
$\langle L{\cdot}0.25$, pl, d, F, $1-d/100$, 1.1$\rangle$	1.14	(30, 234)
$\langle L{\cdot}0.25$, pl, d, Lb, $1-d/100$, 1.2$\rangle$	1.20	(208, 210)
$\langle L{\cdot}0.33$, pl, d, Lb, $1-r/5$, 1.2$\rangle$	1.30	(227, 126)
$\langle L{\cdot}0.1$, pl, d, Lb, 0.75^r, 1.2$\rangle$	1.41	(234, 54)
$\langle L{\cdot}0.33$, pl, d, Lb, $1-r/10$, 1.5$\rangle$	1.41	(168, 181)
$\langle L{\cdot}0.25$, pl, d, Lb, $1-r/5$, 1.2$\rangle$	1.42	(201, 109)
$\langle L{\cdot}0.25$, pl, d, F, $1-d/100$, 1$\rangle$	1.43	(31.5, 233)
$\langle L{\cdot}0.1$, pl, d, Lb, $1-r/5$, 1$\rangle$	1.46	(238, 57)

Table 4. Ranking lists of configurations — top 10 by the combined score r

	r	(r_e,r_d,r_v)
$\langle$I, pl, d, Lb, 0.75, 1.5$\rangle$	218	(39, 34, 145)
$\langle$I, pl, d, Lb, 0.90^r, 1.2$\rangle$	230	(15, 48, 167)
$\langle$I, pl, ncs, F, $1-r/5$, 1$\rangle$	232.5	(66.5, 11, 155)
$\langle$I, pl, d, R, $1-r/5$, 1.1$\rangle$	235	(70, 5, 160)
$\langle$I, pl, ncs, Lb, 0.90^r, 1.5$\rangle$	235	(15, 49, 171)
$\langle$I, pl, ncs, Lb, 0.90^r, 1.2$\rangle$	238	(15, 51, 172)
$\langle$I, pl, ncs, Lb, 0.75, 1.5$\rangle$	239	(44, 42, 153)
$\langle$I, pl, d, Lb, 0.95^r, 1$\rangle$	241	(47, 44, 150)
$\langle$I, pl, ncs, R, $1-r/2$, 1.5$\rangle$	245	(15, 139, 91)
$\langle$I, pl, d, F, 0.75^r, 1.5$\rangle$	245.5	(91.5, 21, 133)

and have reasonably small variation. In addition to the combined score, we show also the ranks for individual criteria — i.e., the components r_e, r_d and r_v — in this table. Data in Table 4 indicate that key parameter values, which are the most important for good overall performance, are the threshold base value $\mathrm{I} : 5 - 10 - 20 - 50 - 100$ and the threshold mode $\mathrm{path\ length}$. The values of all other parameters differ among the top configurations, and therefore influence the performance of DFS-RB to a much lesser degree.

Table 5. Similarity between ranking lists computed by our algorithm for individual criteria

Pair of criteria	ρ - just success
finding errors - distance	0.08
finding errors - variation	-0.53
distance - variation	-0.50

The data presented in Table 3 and Table 4 also indicate that a configuration performing extremely well on one criterion very often does not perform well according to the other two criteria. For example, the best configuration with respect to the ability to find errors is at the position 168 in the ranking list for distance and at the position 118 in variation.

Using the ranking lists for individual criteria (the ability to find errors, the distance from the best speed, and the variation in performance), it is also possible to find out whether there exist configurations of DFS-RB that are very good (i.e., achieve high ranks) with respect to all three criteria. Our approach was to use Spearman's rank correlation coefficient (also called the Spearman's ρ method) [51] to measure the correlation (similarity) among the three ranking lists computed by our algorithm for the individual criteria. The coefficient for a pair of ranking lists of length n is computed by the expression

$$\rho = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2 - 1)},$$

where d_i is the difference between the ranks of a configuration i in the two input lists. The values of the expression are normalized in the interval $\langle -1, 1 \rangle$, where 1 represents perfect correlation and -1 is full opposition.

Table 5 shows the resulting similarity coefficients (values of ρ) for all three pairs of these criteria. We included only the successful configurations for the purpose of this calculation, but all of them, not only the 10 best ones that are displayed in Table 3. The values of ρ show that, even though all three corresponding ranking lists are computed over the identical sets of successful configurations, the orderings of configurations in the lists are quite different. Some configurations achieve a very good ranking both in the ability to find errors and in speed (e.g., for the configuration $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$ the ranking is 39 and 34, respectively), while other configurations produce bad results in these two criteria, so that overall it evens out. We can also observe that the difference specifically between the ranking list for the variation and for the other two criteria is very large. It is, therefore, quite difficult to find a configuration that achieves both high speed and low variation at the same time.

To summarize, the preceding analysis completes our answer to the goal G1 defined in Section 3.2, and also represents our full answer to the goal G2. Based on the tables with the top fragments of ranking lists, we identified configurations with consistently good performance over many benchmarks. The complete ranking lists over all of the benchmarks

and the ranking lists computed for individual benchmarks are available at the web site `http://d3s.mff.cuni.cz/~parizek/stttt18/`.

In order to increase our confidence that the proposed ranking algorithm can reliably identify configurations with consistently good performance, we also did pairwise comparisons of individual configurations based on the Vargha and Delaney's $\hat{A}_{12}$ statistical test [55]. Given a pair of configurations and the respective collections of raw performance data, the test can determine which of the two configurations is more likely to achieve better performance. The raw performance data include (i) the numbers of states processed by individual JPF-RB runs, where fewer states corresponds to better speed, and (ii) the identification of runs that failed to find an error, for which we assume that the number of states is infinite. However, the test does not capture the extent of the difference in speed between the two configurations (i.e., the test in general does not capture variation in speed), and therefore we could compare the results of this test only against steps 1 and 2 of our ranking algorithm, which reflect the ability to find errors and the distance from the configuration with the best speed, respectively.

For two configurations c_1 and c_2, and the respective lists L_1 and L_2 of numbers of states processed by JPF-RB runs, the result a_{12} of the pairwise comparison of c_1 with c_2 is computed as follows. Let L_{12} be a list created by concatenating L_1 with L_2, and R_{12} be a list of the ranks of the numbers in L_{12}. Then, a_{12} is computed by the expression $a_{12} = (s_1/m - (m+1)/2)/n$, where s_1 is the sum of ranks of the numbers that belong to the list L_1, m is the length of L_1, and n is the length of L_2. If $a_{12} < 0.5$, then the configuration c_1 is more likely to achieve better performance, otherwise it is c_2.

The results of pairwise comparisons are used to assign a score to each configuration in the following way. The initial score is 0. For each compared pair, the score of the better configuration in the given pair is increased by 1, while the worse configuration has its score reduced by 1. At first, we processed the data for each benchmark separately to get a list of benchmark-specific scores for every configuration, and then we computed the average over all benchmarks. The average score indicates whether a given configuration can yield better performance than other configurations on many benchmarks. Table 6 shows the top 10 configurations according to their score determined by pairwise comparisons. For each configuration in this table, we also show its position in the overall ranking list generated by our algorithm. We provide the complete lists of configurations sorted by their scores — for each individual benchmark and the average over all the benchmarks — at the web site `http://d3s.mff.cuni.cz/~parizek/stttt18/`.

We observed that there is a great overlap between (1) the full list of configurations generated using the score based on the Vargha and Delaney's $\hat{A}_{12}$ statistical test and (2) the full ranking lists generated by our ranking algorithm. However, the lists are based on different principles, and therefore not directly substitutable. Our ranking algorithm (Table 4) orders configurations according to the three metrics that we proposed as predictors of configurations that would perform well. Vargha and Delaney's $\hat{A}_{12}$ test (Table 6) orders configurations according to the frequency of a configuration actually outperforming other configurations. The similarity of the two lists indicates that our metrics are effective in identifying configurations that perform well, and confirms the high relevance of the ranking lists computed by our algorithm.

Table 6. List of configurations — top 10 based on pairwise comparisons using Vargha and Delaney's $\hat{A}_{12}$ test

	score	ranking
$\langle$I, pl, d, F, $1-r/5, 1.2\rangle$	7162	57
$\langle$I, pl, d, F, $1-r/10, 1\rangle$	7148	118
$\langle$I, pl, d, F, $1-r/5, 1.1\rangle$	7123	17
$\langle$I, pl, d, F, $0.75^r, 1.2\rangle$	7118	56
$\langle$I, pl, ncs, F, $1-r/5, 1.2\rangle$	7118	52
$\langle$I, pl, ncs, F, $1-r/10, 1\rangle$	7117	154
$\langle$I, pl, ncs, F, $0.75^r, 1.2\rangle$	7105	49
$\langle$I, pl, ncs, F, $1-r/5, 1.1\rangle$	7092	83
$\langle$I, pl, d, R, $1-r/10, 1\rangle$	7081	77
$\langle$I, pl, d, F, $0.75^r, 1.5\rangle$	7060	10

7 Validation of Ranking System

In the previous section, we applied the ranking system to find configurations that achieve good overall performance on the nine benchmarks. In this section, we validate how well the ranking system generalizes from training benchmarks to new, unknown programs. We do this in two ways. First, we perform leave-one-out cross-validation: we find the best overall configurations for subsets of eight of the nine benchmarks, and evaluate the performance of those configurations on the ninth benchmark not used for training. Second, we select the configurations from the previous section that worked best overall on the nine benchmarks, and evaluate them on a set of twelve benchmarks with a more recent version[2] of JPF that was current in December 2015.

7.1 Cross-validation using leave-one-out.

The key idea of validation based on the leave-one-out principle is to repeat the following two steps for every benchmark b.

1. Execute the ranking algorithm on performance data for all benchmarks except b. (*training phase*)
2. Compare the ranking list created in the first step to the ranking list constructed from performance data only for the benchmark b. (*validation phase*)

We compared the similarity of the ranking lists using Spearman's ρ method again. Table 7 shows the results. Each column corresponds to one benchmark b_c specified in the header. In each data cell, we present the value of the similarity

[2] We used the version defined by the commit number 29 in the repository that contains JPF v8.

measure between the ranking list created by excluding the benchmark b_c (which is specified in the column header) and the ranking list just for b_c. Note again that values fall into the range $\langle -1, 1 \rangle$, where 1 represents perfect correlation between the given two ranking lists and -1 the exact opposite. The suffix "with failures" indicates usage of the full ranking lists that also include failing configurations, while the suffix "just success" means that we take as input the sublists that involve only the successful configurations.

The results presented in Table 7 show that the values of the similarity measure are quite high for pairs of ranking lists that contain failing configurations. In those cases, the lists capture very similar orderings of individual configurations by their score. This is due to the facts that (1) around 97% configurations are failing and (2) the sublists of failing configurations are identical in all the ranking lists. On the other hand, the rather low similarity coefficients between pairs of ranking lists computed over only successful configurations indicate that, for each benchmark, the best performance is achieved by a different subset of configurations — this observation is consistent with Table 2 in Section 5.3. Together, these results indicate (1) that our ranking algorithm helps to identify configurations that always find an error, but (2) there do not exist configurations that consistently achieve very good performance (speed) on every benchmark.

7.2 New benchmarks.

We evaluated the best overall configurations on twelve benchmark programs using the more recent version of JPF. Three of the twelve benchmarks are new: we selected them only after we identified the best overall configurations using the nine training benchmarks, i.e. the programs that we described in Section 5.1. The three new benchmark programs were:

- a plain Java version of the CDx benchmark [29],
- the Cache4j benchmark from the PJBench suite [65],
- and the QSortMT benchmark from the Inspect suite [61].

Table 8 provides basic characteristics of the three new benchmark programs — the number of source code lines (LoC) and the number of concurrent threads. We manually created artificial race conditions in Cache4j by modifying the scope of synchronized blocks. The other two benchmarks already contained errors (also data races).

We analyzed the state space of each new benchmark in the same way as for the original suite (details are provided in Section 5.1.1). Table 8 presents the results. All three benchmarks have a very low error path density and therefore contain hard-to-find bugs.

Experiments with top-ranked configurations. In order to further investigate general applicability of the ranking lists beyond our training set, we performed experiments on all 7920 configurations for each of the 12 benchmarks — including three new benchmarks that were not used as input for the ranking system. Table 9 shows, for each configuration cfg in the top 10 of the overall ranking list (taken from Table 4) and for each benchmark b, how many configurations achieve better average performance than cfg on b. We abbreviated the names of several benchmarks in the table header to reduce its width. The symbol "-" indicates that JPF-RB failed to detect an error within the time limit in the given experiment.

The data in Table 9 show that, as we expected, the top configurations according to the overall ranking do not achieve the best performance for individual benchmarks, but they are still placed quite high in the lists of configurations sorted by average performance — mostly in the top third.

An obvious exception is the CDx benchmark, for which DFS-RB fails to detect an error in 9 out of the 10 top configurations overall, and succeeded just in a single JPF run under the remaining configuration that is ranked as second. The main reason why DFS-RB fails on CDx is that many error states are pruned as a consequence of early backtracking. The bug present in the code of CDx is a race condition triggered by concurrent accesses to a particular field in two specific threads. We showed in Table 8 that the corresponding error states and paths are very rare. Using manual inspection of the execution logs of JPF, we found that one of the following two conditions has to be satisfied in order to increase the likeliness of hitting an error state during the traversal of the state space of CDx:

- either JPF uses a custom search order (e.g., random search or the guided search with heuristics that is described in Section 8),
- or, in the case of the default search order, it explores the first state space path up to the end state and then continues with some other paths.

Both conditions are violated by standalone DFS-RB, which uses the default search order and early backtracking, and therefore fails to explore some interleavings of the racy field accesses. This hypothesis was confirmed by additional experiments that we performed on CDx, using threshold values in the set $\{200, 500, 1000, 1500\}$. Table 10 shows the results of the experiments with these additional threshold values. Other parameters are taken from the overall best configuration of DFS-RB.

8 Related Work

We provide more details about selected existing approaches to quick and scalable detection of concurrency errors that are based on state space traversal. In particular, we describe all techniques that we use in experimental evaluation (Section 9). Here we provide only a qualitative comparison of the other techniques with randomized backtracking. We divided the techniques into six categories based on the main approaches involved: guided search with heuristics, bounded (incomplete) search, partial order reduction, usage of randomization, parallelization, and combination with static analysis. Some techniques belong to multiple categories — in such cases, we picked the category based on the primary distinctive feature of the given technique. At the end of this section, we

Table 7. Validation: similarity between ranking lists created according to the leave-one-out principle

Benchmark b_c:	Daisy file system	Elevator	Alarm Clock	Linked List	Producer Consumer	RAX Extended	Replicated Workers	jPapaBench	Monte Carlo
Only b_c - with failures	0.695	0.701	0.509	0.457	0.509	0.444	0.503	0.713	0.762
Only b_c - just success	0.092	0.119	0.074	-0.045	0.237	-0.101	-0.218	-0.230	-0.137

Table 8. Additional benchmark programs: basic information and state space characteristics

Benchmark	LoC	Threads	Ratio of error states	Ratio of error paths	Minimal length of error path	Average length of error path
CDx	3900	2	0.0004	0.0000000001	375	4177
Cache4j	550	2	0.00096	0.0000000006	253	363
QSortMT	290	2	0.007	0.00000000002	21	40

Table 9. Relative performance of top-ranked configurations

Config rank	Daisy fs	Elevator	Alarm Clock	Linked List	Prod.-Cons.	RAX Ext.	Repl.-Work.	jPapa-Bench	Monte Carlo	CDx	Cache4j	QSortMT
1.	16%	8.2%	7.3%	7.1%	8.5%	10%	15.3%	15.1%	5.6%	-	34.8%	35.5%
2.	15%	5.7%	7.3%	2.3%	5.8%	1.6%	16.2%	26.5%	7.1%	0.1%	36%	39.5%
3.	13.9%	9.8%	30.1%	1.5%	5.4%	6%	23.2%	2%	1.5%	-	-	71.1%
4.	15%	19.7%	46.2%	5.6%	7.1%	3.5%	33%	1.2%	3.5%	-	34.5%	70.7%
5.	13.3%	5.4%	2.2%	1.8%	2.1%	4.7%	13.4%	18.5%	7.3%	-	35.5%	32.6%
6.	13%	5.1%	16%	5.7%	7.7%	4.4%	16.6%	22.8%	7%	-	34.9%	38.2%
7.	15.1%	9.4%	10.1%	7.8%	4.6%	6.9%	12.2%	13.2%	6.5%	-	34%	34%
8.	15%	13.9%	6.5%	8.4%	2.7%	8.6%	14%	14.7%	5.6%	-	19.7%	22.7%
9.	12%	12.5%	18.2%	16.7%	33.5%	43.3%	54%	1.6%	-	-	-	-
10.	13.6%	19%	15.5%	0.6%	6%	1.5%	54.1%	24.5%	6.1%	-	0.1%	9.3%

discuss existing work in the form of experimental studies and compare their conclusions with our results.

Before delving into the individual categories, we want to highlight that most of the techniques can be expressed through custom implementations of the functions enabled and order in the standard DFS algorithm for state space traversal (Figure 1). In general, all techniques that use a custom implementation only for the function order explore the whole state space if they do not detect an error during the search. A custom implementation of the function enabled, which prunes the state space according to some criteria, has to be used in order to perform an incomplete traversal. We provide specific details below. The search order, i.e. the order in which transitions outgoing from a state are explored, also has a great influence on the error detection performance. Each tool for state space traversal of concurrent programs uses its own default search order, which is typically implementation-specific.

Category 1: Guided Search with Heuristics Techniques in this category use heuristics to navigate the search quickly towards error states, while preserving the full coverage of a program state space. The part of the state space that is more likely to contain error states is explored first and the rest is explored afterwards, thus making it possible to discover errors in less time. At each state, the heuristics determine the order in which enabled transitions are explored, and in particular they help to choose the next transition to be explored. More specifically, the list of enabled transitions is sorted according to a heuristic function that is evaluated over the transitions themselves or over their destination states. All of this is typically done in a custom implementation of the function order.

Many different heuristic functions have been proposed over time [12,13,21,27,49,57,58]. A popular heuristic, which is useful especially for fast detection of concurrency errors, maximizes thread preemption on each execution path [21]. When the current state s was reached by execution of thread t, the function order puts transitions associated with threads other than t at the beginning of the list.

Another useful heuristic gives preference to transitions that may interfere with some of the previous transitions on the current state space path [58]. Two transitions can interfere, for example, if they are associated with different threads and contain instructions that access the same heap object.

The main difference of these approaches from our DFS-RB algorithm is that the state space of a program is completely explored if no error state is hit during the search. On the other hand, there are similarities between parameters of the DFS-RB algorithm and some of the heuristics described above. Certain values of the parameters thr and rtc increase the likeliness of early backtracking when an actual thread switch did not occur at a scheduling choice point — in this

Table 10. Additional experiments with CDx

Configuration of DFS-RB	States	Time	Found
$\langle 200, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5\rangle$	error states pruned in all runs		0%
$\langle 500, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5\rangle$	error states pruned in all runs		0%
$\langle 1000, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5\rangle$	error states pruned in all runs		0%
$\langle 1500, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5\rangle$	6777 ± 0	14 ± 0 s	100%

way, DFS-RB also tries to maximize thread preemption in order to detect concurrency errors.

Category 2: Bounded Search. As we already mentioned in the introduction, many techniques explore only a bounded part of the state space because the complete traversal is not tractable for non-trivial programs. They are also motivated by the assumption that many errors can be found in a particular small part of the state space — for example, on execution traces that involve at most two thread preemptions [35].

Limiting the number of thread preemptions (context switches) on each explored state space path [43] is actually the most popular approach to bounded search (state space traversal). It was applied both in explicit-state model checking [35] and in SAT-based model checking [44]. In the context of explicit-state traversal based on DFS (Figure 1), an implementation typically uses a customized function $\mathrm{enabled}$. A transition is excluded from the resulting set if its exploration would exceed the number of allowed thread preemptions. When there are available resources, e.g. time and memory, the bound can be iteratively increased [35] to discover more errors.

Depth bounding [54] is another technique that naturally belongs into this category. In fact, our DFS-RB algorithm corresponds to an exhaustive search for errors up to a given bound (threshold) that is augmented with random walks to greater depths below the threshold. Depending on the value of the configuration variable $C.thm$ (threshold mode), the algorithm can mimic either depth bounding or the bound on the number of thread preemptions.

Category 3: Partial Order Reduction. An important component of many techniques and tools for efficient detection of concurrency errors, especially those based on model checking and systematic concurrency testing, is partial order reduction (POR) [19]. The goal of POR is to identify the subset of all possible interleavings of program threads that must be explored in order to cover all observable behaviors of a given system. In order to achieve this goal, a POR algorithm distinguishes between visible actions, which read or modify the global state, and thread-local actions. Furthermore, a subset of visible actions is responsible for actual communication between concurrently running threads — we call such actions interfering. All other actions are independent. The key idea behind POR is to explore, in an optimal case, only a single interleaving from each set of thread interleavings that differ only in the order of independent actions. The scalability of verification is greatly improved by this reduction in the number of thread interleavings that are actually explored during the state space traversal. From the perspective of implementation, POR is often realized by creating non-deterministic thread scheduling choices only at interfering actions. Many POR algorithms have been developed in the past. The list of most prominent algorithms includes the dynamic POR by Flanagan and Godefroid [18], POR based on heap reachability [10], cartesian POR [22], and the recently developed optimal dynamic POR [1].

JPF performs a custom variant of dynamic POR underneath the level at which DFS-RB operates. The POR in JPF identifies interfering actions on-the-fly during the state space traversal based on the reachability of heap objects from multiple threads. Nevertheless, a significant limitation of the JPF's dynamic POR is that it can miss some observable behaviors (it is not complete) due to the way in which shared heap objects are detected and the amount of information preserved globally during the search.

Category 4: Usage of Randomization. We are aware of multiple testing and verification approaches that use randomization in some way during the search for errors. A brief overview of selected approaches is provided below. Some of them are orthogonal to randomized backtracking and therefore, like in the case of POR, they too can be easily combined with the DFS-RB algorithm.

The most basic approach, but still quite popular, is to use a random search, where the transitions leading from a state s are explored in a random order.

Randomization can be used also, for example, to guide the search [8, 46], in a random partial order sampling algorithm [48], and in combination with restarts of the search process [37]. In general, both complete and incomplete search techniques can benefit from randomization.

The key idea of the technique developed by Coons et al. [8] is to explore complete execution paths in an order determined by priority functions. One of the supported priority functions is random search (walk).

Sen et al. [48] proposed a method based on random sampling of different partial orders, which has a more uniform behavior than the simple randomized search algorithm. Here, partial orders are sets of thread interleavings with different execution sequences of interfering actions.

Probabilistic concurrency testing (PCT) by Burckhardt et al. [6] extends the simple search algorithm driven by random number choice and thread priorities. Thread preemption is allowed and enforced only at certain program code locations, following the assumption that any given concurrency bug can be triggered when just a few instructions are ordered in a spe-

cific way. An important contribution is the definition of expressions that (1) yield explicit probability guarantees that a given run will find a concurrency bug and (2) determine how many runs of the verification tool are needed, on average, to find the given bug.

Parizek and Kalibera [37] combined random search order with frequent restarts of the state space traversal process. This work was inspired by the restarts performed by SAT solvers according to certain strategies. Applying the same idea to concurrency bug detection through state space traversal helped to achieve significant performance improvements.

Jones and Sorber [28] proposed an algorithm for efficient detection of LTL property violations that is based on parallel random walks and early backtracking. Depth-bounded random walks are used to find cycles with accepting states and early backtracking helps to avoid walking too deep (i.e., away from the accepting state). The goal of the whole procedure is to search for accepting cycles at a particular depth and its close vicinity. In order to achieve this goal, the probability of early backtracking is computed using the Butterworth filter that originated in the signal processing domain. Parameters of the filter determine how close the vicinity actually should be. This algorithm is useful for detecting cycles in the state space that involve a given accepting state (which must be already found before). However, our DFS-RB algorithm tries just to find an error state — and then the search procedure immediately terminates. It would make sense to use the Butterworth filter in DFS-RB if we wanted to search further in the close vicinity of the respective discovered error state, but that is not the case in our current setting.

Category 5: Parallelization. The time needed to find errors can be reduced also by running multiple instances of the search process in parallel, for example on different units of a multi-core CPU or different nodes of a distributed system (cluster), such that each instance explores a different part of the state space. When one of the instances detects an error, all others are terminated and the error is reported. Techniques belonging to this category include combination of parallel search with randomization [9], swarm verification [24,25], various approaches to distributed model checking, and also multi-core model checking. Some techniques even utilize modern GPUs. Each individual technique that we are aware of is designed using one of the following two principles — either (i) the state space is partitioned and each region is then explored on one node in the cluster or one CPU core [2, 3, 7, 23, 31, 32, 52], or (ii) multiple independent search jobs (in some cases a very large number) are started where each of them explores the whole state space but using a different search order [24, 25, 30, 45]. The latter is an example of an embarrassingly parallel approach to verification. In addition, the techniques differ in aspects such as whether they use local or shared state storage, whether they use on-the-fly partitioning or static (done in advance before the start of traversal), the manner of communication between nodes (exchanging states that have to be explored), support for load balancing, and supported kinds of properties (safety, LTL). All of these approaches are very useful especially in the case of hard-to-find concurrency errors, where the total cost (running time) is reduced by a factor ranging from 2 up to 1000.

Parallel execution of multiple instances, each with a different configuration, is a practical option also in the case of the DFS-RB algorithm. For example, each instance could use a different threshold value. Most of the approaches to parallel and distributed model checking listed above can be easily combined with DFS-RB, because they use DFS for state space exploration. It is possible to run DFS-RB on each node of a cluster, respectively on each CPU core, to search for errors in each partition of the state space.

However, there exist also some approaches based on BFS (breadth-first search) — for example, model checking algorithms that exploit GPUs (many cores) [59], techniques combining distributed model checking with beam search [60], and combinations of BFS with random search order [16]. Our current definition of the DFS-RB algorithm obviously does not support BFS, so it is not directly applicable to those approaches. The meaning of some configuration parameters would have to be adapted from DFS to BFS.

Category 6: Combination with Static Analysis. The techniques in this category combine state space traversal with hybrid analyses that provide information about the future behavior of program threads. The main purpose of hybrid analyses is more precise identification of interfering actions, where thread choices have to be created during the traversal. Each hybrid analysis has two phases — static and dynamic. The static phase, which is performed in advance, computes only partial information. The full results of the hybrid analysis are computed on-the-fly during the state space traversal, using data collected in the first phase and information taken from dynamic states (including, for example, the dynamic call stack of each thread and the dynamic values of program variables).

Parizek and Lhotak proposed hybrid analyses both for field accesses [39] and shared array elements [40], and applied them in JPF to optimize POR and improve scalability by eliminating redundant thread choices. In addition, Parizek [41] also developed two heuristics for very fast error detection that are based on the hybrid analysis. One heuristic changes the order in which transitions are explored, prioritizing transitions such that actions interfering with some past accesses may be performed in the future on the respective execution traces. The second heuristic prunes transitions for which the hybrid analysis can provably determine that no interfering actions are executed in the future.

Surveys and Experimental Studies. Approaches from different categories have been compared in several recent experimental studies. We discuss their main observations and compare them with our own.

Rungta and Mercer [47] compared several tools (each representing a different category) on a large set of benchmarks, and claimed the following.

1. The ability to discover errors by DFS with a random search order degrades when the number of threads increases.
2. Iterative preemption-bounding with a randomized search order can outperform standard DFS with a random search order when only a few preemptions (one or two) are sufficient to reach an error state.
3. Standalone iterative preemption bounding [35] and dynamic partial order reduction [18], both with default search order, cannot find errors in concurrent programs in a reasonable time limit.

The third claim is in line with our own findings that we present in the next section. We did not perform experiments that might confirm or refute the other two claims.

Thomson et al. [53] experimentally compared different techniques for concurrency testing that are based on reducing the number of thread schedules explored during the search. The authors of this study focused on preemption bounding [35], delay bounding [15] and a random scheduler (i.e., random search). Their most interesting observation is that a random scheduler has comparable performance to schedule-bounding techniques. In addition, they found that (1) delay bounding is faster than preemption bounding and (2) schedule-bounding is in general superior to the standard depth-first search algorithm.

9 Comparison with Other Approaches

Here we report on the second part of our evaluation, in which we compare the performance of the DFS-RB algorithm against selected well-known state-of-the-art approaches for detecting concurrency errors that involve state space traversal. We performed the respective experiments on all 12 benchmarks (including CDx, Cache4j, and QSortMT). The list of techniques is provided below. In the case of randomized backtracking, we considered only the top 10 best configurations overall, as identified by the ranking algorithm (Section 6). The main goal is to find out whether some of the top 10 configurations of randomized backtracking (and which ones) achieve better performance than the state-of-the-art techniques. We also integrated the DFS-RB algorithm with some of the other approaches in order to evaluate the impact on performance that such integration may have, in particular to find which combinations are most efficient.

We selected the following state-of-the-art approaches using state space traversal. All of them are described in the section about related work. Below we discuss especially the specific configurations of the respective approaches that we used in our experiments.

1. Exhaustive state space traversal with the default search order used by JPF. This order reflects the indexes of threads associated with transitions, starting from 0 up to N.
2. Bounded search with a fixed limit on the number of thread preemptions on every state space path (see Category 2 in Section 8). For the standalone bounded search, we used two different configurations with the bounds 5 and 10, respectively. However, in the combination with DFS-RB, we used only the bound of 10.
3. Guided search with the heuristic that maximizes thread preemption on each execution path (Category 1).
4. State space traversal with dynamic POR by Flanagan and Godefroid [18], implemented in JPF (Category 3). Here we must emphasize that randomized backtracking and dynamic POR cannot be combined, because dynamic POR is not compatible with early backtracking due to its need to fully explore each state space path (up to an end state or an already visited state).
5. State space traversal using POR based on heap reachability, which is the default in JPF, combined with the hybrid analyses of future field accesses and shared array elements (Category 6 in Section 8).
6. Search with heuristics based on hybrid analyses of field accesses and shared array elements — namely transition reordering and pruning (Category 6). We used only the best configuration of these heuristics, i.e. the one with very good performance according to [41], for our experiments reported here.
7. State space traversal with a random search order (Category 4 in Section 8 on related work).
8. Restarts of state space traversal combined with random search (Category 4). The frequency of restarts is determined by a basic time unit and one of the following three strategies: fixed, Luby, and Walsh. The Walsh strategy [56] generates a sequence $u, ru, r^2u, r^3u, \ldots$, where u is the basic time unit and r is the ratio. Note that here the variable r denotes a ratio specific to the Walsh strategy, which is completely different (and independent) from the ratio used as a parameter of the DFS-RB algorithm. In our experiments, we used $r = \sqrt{2}$. For standalone runs, we use the basic time unit of 5 seconds to allow multiple restarts of JPF within the time limit for randomized backtracking (one minute). On the other hand, in the combination of restarts with DFS-RB, we use only the fixed strategy and the time unit of 30 seconds to allow one restart of a JPF run within the time limit. The motivation behind this decision is to let JPF run for some time (and give some space to randomized backtracking) before it is restarted.

Only the last two approaches in the list already involve randomization. Note also that JPF already supports the following approaches out-of-the-box: bounded search with a limit on thread preemptions, guided search with the heuristic that maximizes preemption, POR based on heap reachability, and random search. We implemented the remaining ones — in particular, dynamic POR, hybrid analyses of future accesses, heuristics for transition reordering and pruning based on the hybrid analysis, and restarts of the state space traversal.

9.1 Experiments

We organized and ran the experiments in a very similar way to the evaluation of the standalone DFS-RB algorithm. For

each technique and configuration that involves random number choice, we performed 10 runs of JPF. The set of reported metrics contains the number of processed states, total running time in seconds, and the percentage of JPF runs that found an error within the time limit. The values of basic statistics, i.e. average and standard deviation, are again provided in the case of experiments that involve randomization. Here we use the format $\mu \pm \sigma$. In the cases when all JPF runs for a particular experiment failed, we indicate whether that was due to timeout or because all the error states were pruned.

9.2 Results

We provide the results of the experiments in the form of graphs in Figure 8 and tables in Appendix A. Then, in the following sections, we discuss the results and make some general observations.

The graphs in Figure 8 present a relative comparison of the performance of the DFS-RB algorithm, the state-of-the art techniques, and combinations of the two, on individual benchmarks. We consider only selected configurations of DFS-RB from the top 10 in the overall ranking list (which reflects the combined score) — specifically, configurations at positions 1, 2, 3, and 10. Note also that standalone DFS-RB is implemented on top of JPF with the default search order, and therefore we did not explicitly include the combination with the default search order in the tables and graphs, as that would be redundant.

Each point in a graph represents the average number of states processed by JPF-RB with a given configuration, and each error bar represents the standard deviation. In the case of techniques that do not involve random choice, the standard deviation is zero. The symbol "X " is shown near the top line for every approach that timed out or failed to find an error. A logarithmic scale is used on the Y axis in order to get a more fine-grained resolution for small values, and to enable easier differentiation among configurations based on their performance. We distinguish the three groups of techniques by different colors: red is used for the standalone DFS-RB, green is used for the state-of-the-art techniques, and blue is used for combinations of DFS-RB with the state-of-the-art techniques. In addition, the state-of-the art techniques are presented in the graphs in the following order (from left to right): non-randomized techniques that we also combined with DFS-RB, randomized techniques combined with DFS-RB, and others that were not combined. The groups of techniques are separated by vertical dotted lines. For each combination and the corresponding state-of-the-art technique, we put the respective points and error bars (in blue and green) right next to each other on the X axis in order to highlight differences in performance.

The graphs report the numbers of states because they allow us to identify more precisely the configurations with better or worse performance. The running times are proportial to the numbers of states, but for small benchmarks they are all 1 second. Note, however, that the purpose of these graphs is to indicate general trends. All the underlying concrete values are provided in the per-benchmark tables in Appendix A for closer inspection.

We performed manual inspection of the graphs in Figure 8 and concrete data in Table 11-22 (see the Appendix A) to derive conclusions that we discuss below.

9.3 Discussion: Timeouts

The most significant characteristic of every pair of a benchmark program and an individual technique (configuration) in our setting is whether the technique successfully found an error in the benchmark within the time limit or failed due to timing out. In total, when considering all experiments (i.e., all pairs of a technique or configuration with a benchmark program), the individual state-of-the-art techniques time out in 37 cases out of 144. We evaluated 6 of the 12 state-of-the-art techniques both by themselves and in combination with DFS-RB, on the 12 benchmarks. Of these 72 pairs of technique and benchmark, 15 configurations time out when the state-of-the-art technique is used alone without DFS-RB. When the state-of-the-art technique is combined with DFS-RB, only 7 of the 72 configurations time out. These 7 cases include two timeouts on the Monte Carlo benchmark, three on CDx, and one timeout per Linked List and QSortMT benchmarks.

To be more specific, we identified six different patterns:

1. A majority of the state-of-the-art techniques timed out on the Daisy benchmark.
2. On Elevator, jPapaBench and Replicated Workers, all configurations of standalone DFS-RB and combinations with other techniques finished within the time limit, while half of the state-of-the-art techniques timed out. Specifically, when a state-of-the-art technique timed out, its combination with DFS-RB found an error quickly.
3. The results are mixed for the Monte Carlo benchmark. The combination with DFS-RB finishes within the time limit in one of the cases when the corresponding state-of-the-art technique timed out. However, in each group, there are some successful approaches and some failing due to timing out.
4. Just one combination timed out in the case of Linked List, and one configuration of standalone DFS-RB timed out for the Cache4j benchmark.
5. Two approaches failed for the QSortMT benchmark — one combination with DFS-RB and one individual state-of-the-art technique.
6. For the CDx benchmark, the respective graph in Figure 8 shows that combinations of state-of-the-art with DFS-RB failed in three cases, while just one individual state-of-the-art technique timed out.

All techniques succeed for the remaining benchmarks, namely Alarm Clock, Producer Consumer, and RAX Extended.

9.4 Discussion: Performance

When analyzing the performance (speed) of individual techniques and configurations on benchmarks for which they success-

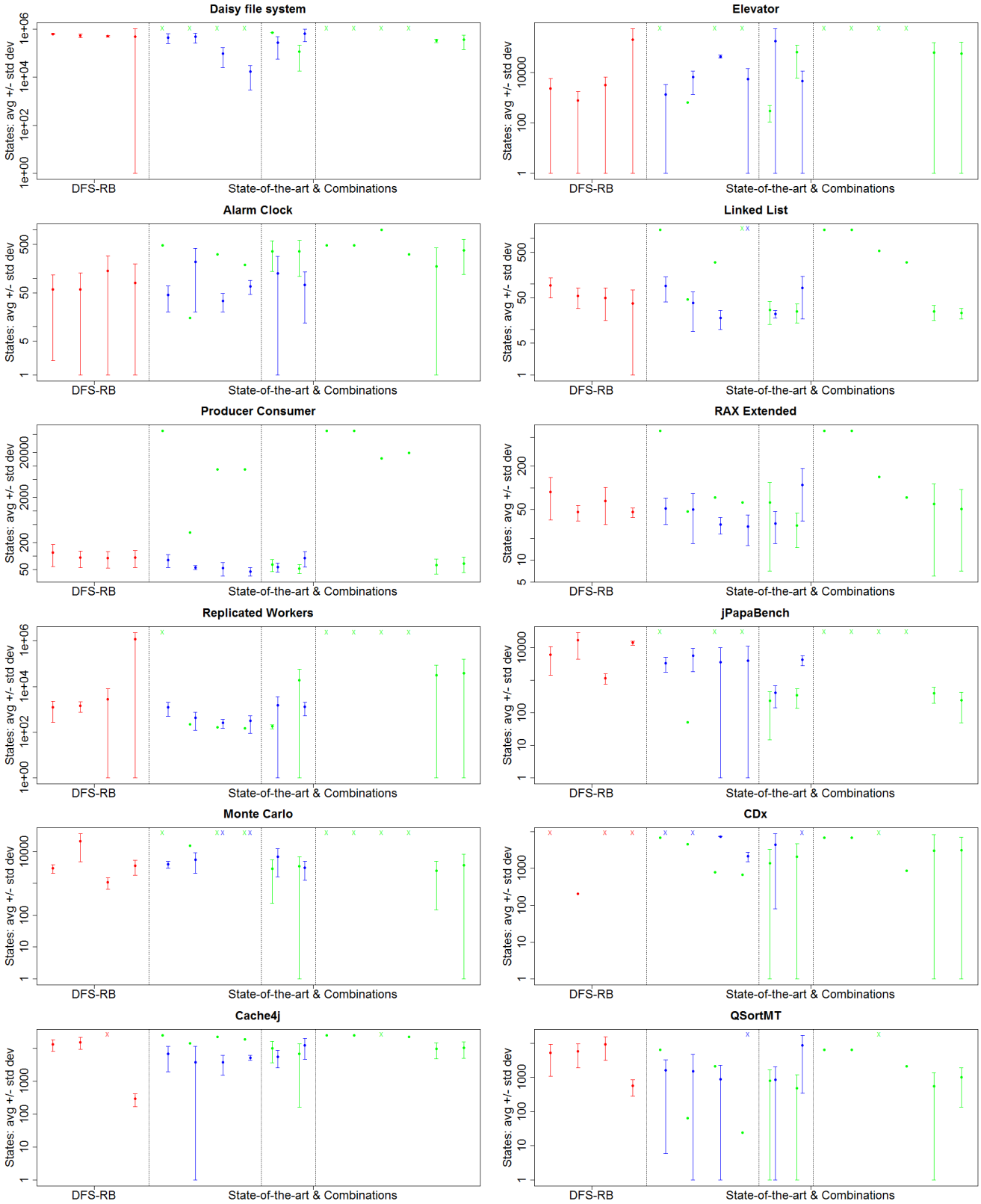

Fig. 8. Overall relative comparison of DFS-RB with state-of-the-art techniques. In each graph, standalone DFS-RB is in red, the state-of-the-art techniques are in green, and the combinations of DFS-RB with the state-of-the-art techniques are in blue. The state-of-the-art techniques are presented in the following order: non-randomized techniques also combined with DFS-RB, randomized techniques combined with DFS-RB, and others that were not combined.

fully reported an error, we identified four cases that quite closely match those discussed in the previous subsection.

1. A combination of a state-of-the-art technique with DFS-RB achieves better performance than the corresponding standalone technique in almost all the relevant cases for 6 benchmarks out of 12: Daisy, Alarm Clock, Linked List, Producer Consumer, RAX Extended, and Cache4j. The exception, i.e. the state-of-the-art technique that is faster than the combination, is typically a restart of the state space traversal or random search, but also guided search for one benchmark. The top configurations of standalone DFS-RB also have better performance than most of the state-of-the-art techniques.
2. For Elevator, Replicated Workers and QSortMT, the results are mixed in the cases when the state-of-the-art technique finishes within the time limit. DFS-RB, either standalone or in combination, has better performance than some of these state-of-the-art techniques and worse performance than others.
3. The results are mixed for the Monte Carlo benchmark even when we consider only the successful techniques and configurations (which found an error). Most of them achieve performance comparable to each other.
4. Finally, in the case of jPapaBench and CDx, the state-of-the-art techniques that finish within the time limit (e.g., random search and restarts) have better performance than DFS-RB.

Out of the 53 cases where both a state-of-the art technique and its combination with DFS-RB finish successfully within the time limit on a given benchmark (i.e., they report an error), the combination achieves better performance 29 times.

9.5 Discussion: Summary

Overall, DFS-RB with configurations identified by the ranking system achieves good performance when compared to state-of-the-art techniques. We also want to highlight the following general observations regarding the ability to find errors and speed:

- Combinations involving DFS-RB almost always find an error within the time limit, while many state-of-the-art techniques failed due to timeout for the complex benchmarks such as Daisy and jPapaBench. In other words, the combinations have a positive effect on the error detection performance, both with respect to the standalone DFS-RB algorithm and the state-of-the-art approaches.
- Almost all state-of-the-art techniques with a better performance than DFS-RB (on specific benchmarks) also involve randomization. This indicates that randomization is important for fast detection of concurrency errors.
- DFS-RB achieves better performance especially for benchmarks that contain deep errors (i.e., such that a large number of states must be explored to find the error), which includes Daisy, Elevator, Producer Consumer, and Replicated Workers. If a particular state-of-the-art technique timed out for a benchmark from this group, then the combination with DFS-RB detected an error within the time limit.
- Regarding the three benchmarks used only for validation of the ranking system and for comparison of DFS-RB with state-of-the-art techniques, standalone DFS-RB and combinations work quite well for Cache4j and QSortMT, while it fails in many cases for CDx (see details at the end of Section 7.2).

The observations discussed here and in the previous two sections also represent our answer to the research goals G0 and G3 defined in Section 3.2.

10 Threats to Validity

We have identified seven possible threats to validity of our experimental evaluation and the derived conclusions, which we discuss in this section. All the threats are related to the decisions that we made when designing the DFS-RB algorithm, ranking system, and experiments. The consequences of those decisions may influence, for example, usage of the DFS-RB algorithm in a slightly different context and the best achievable performance.

Threat 1. The set of benchmark programs that we used to evaluate DFS-RB might not be sufficiently representative of all possible multi-threaded Java programs. As a consequence, general observations and recommendations derived from our state space analysis and experiments may not apply in some other setting or to programs with specific characteristics. We attempted to migitate this threat by using a diverse set of benchmarks. However, results for the CDx benchmark indicate that there exist programs on which DFS-RB will fail under many configurations.

Threat 2. The general applicability of the computed rankings of the DFS-RB configurations may be limited due to possible over-fitting to our benchmark programs. In order to address this issue, we evaluated the performance and ranking of configurations on a separate group of benchmarks (CDx, Cache4j, QSortMT) that were not a part of the training set, and we also performed cross-validation based on the leave-one-out principle (Section 7). The results give us some degree of confidence that the top configurations in the ranking lists will achieve good performance also on many other subject programs, although there will always be some exceptions (such as CDx).

Threat 3. Although our set of benchmarks contains programs of different size and complexity, we did not include any very large Java program. Tools such as JPF cannot handle many programs for three main reasons: (1) the state spaces of the programs are too large, with too many possible thread interleavings, (2) the tools do not support advanced features of the runtime environment (e.g., Java virtual machine) and the standard library, and (3) the programs have a complex

environment (e.g., network and user inputs) that is hard to model in a realistic way. However, we plan to address some of these issues in our future work by designing and implementing tools for automated construction of abstract models from large software systems.

Threat 4. This threat is related to the parameters of DFS-RB and the associated mechanisms. We selected the specific parameter values based on initial experiments and our experience with techniques involving state space traversal. Our choices may not be optimal in the sense that the selected values do not enable the absolutely best possible performance of DFS-RB, especially when comparing it with state-of-the-art techniques. Results for CDx show that, in some cases, different values of certain parameters (i.e., values that we did not choose initially) have to be used in order to enable DFS-RB to find errors. Nevertheless, the values that we used still help to achieve good overall error-detection performance for many of our benchmarks.

Threat 5. For techniques that involve random choice, the results of the experiments are influenced by the size of data samples, i.e. by the number of times the experiments are repeated. We performed 100 runs for each experiment described in Section 5, and 10 runs for experiments described in Section 9. Although the number of runs is quite high, even larger samples would increase confidence in the results of the experiments and the robustness of the general observations derived from the results.

Threat 6. The results of the evaluation may be influenced by the configuration of the state-of-the-art techniques that we compared with. For each technique, besides the changes needed to enable it, we used the same configuration of JPF as in the default search. In particular, we did not try to optimize the JPF configuration for any of the techniques, including the DFS-RB algorithm.

Threat 7. Finally, we had to implement some of the state-of-the-art techniques ourselves in JPF, and it is possible that a different implementation would have different performance. This applies to the following techniques: bounded search with a limit on the number of thread preemptions, guided search with a heuristic that maximizes thread preemption on every path, restarts of state space traversal, dynamic POR, and techniques based on the hybrid analysis of future accesses to fields and array elements. All other techniques were implemented by other researchers and are distributed with JPF. In the case of bounded search and guided search with heuristics, we modified an existing implementation that was already available for JPF. We mitigated this problem by reporting, for each experiment, the number of processed states as the primary performance metric. This metric is less sensitive to differences in low-level implementation details than execution time.

11 Conclusion

We proposed the idea of using randomized backtracking (i.e., the DFS-RB algorithm) in state space traversal with the goal of fast error detection, and conducted a large experimental study in order to evaluate the benefits and limitations of DFS-RB. Results of the experiments that we performed on several multi-threaded Java programs show that usage of DFS-RB helps to achieve better performance than many state-of-the-art techniques for selected benchmarks — especially for benchmarks that contain deep errors. In particular, DFS-RB finds an error within the time limit in a large number of cases where state-of-the-art techniques fail (e.g., due to timeout). A limitation of our approach is the difficulty of finding configurations, e.g. one for each benchmark, that would give the best performance. There is no single configuration of randomized backtracking that performs well for every benchmark. We designed and used the ranking algorithm to identify configurations that achieve overall good performance with reasonably small variation. The ranking algorithm was validated using the leave-one-out principle and through results of experiments with three additional benchmarks. All of this helped to increase our confidence in the general applicability of the proposed DFS-RB algorithm and the ranking system. However, as in the case of many other approaches based on randomization and state space pruning, we still cannot provide any guarantees that DFS-RB finds all errors in a specific program.

Our implementation of the DFS-RB algorithm, together with all of the benchmark programs and scripts necessary to run the experiments, is available at the web site `http://d3s.mff.cuni.cz/~parizek/sttt18/`.

In practice, randomized backtracking could be used especially in the "embarrassingly parallel" approach to fast search for errors, which has been proposed in [24, 25]. Another possible application is the search for errors in programs with infinite state spaces or infinite paths (e.g., programs that involve some ever-increasing counter).

Acknowledgements. The first phase of this work was partially supported by the Czech Science Foundation project 14-11384S and the second phase was partially supported by the Czech Science Foundation project 18-17403S. It was also partially supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal Dynamic Partial Order Reduction. In Proceedings of POPL 2014, ACM.
2. J. Barnat, L. Brim, and P. Rockai. On-the-fly parallel model checking algorithm that is optimal for verification of weak LTL properties. Science of Computer Programming, 77(12), 2012.
3. G. Behrmann, T. Hune, and F. Vaandrager. Distributing Timed Model Checking - How the Search Order Matters. In Proceedings of CAV 2000, LNCS, vol. 1855.
4. A. Biere. PicoSAT Essentials. Journal on Satisfiability, Boolean Modeling and Computation (JSAT), 4, 2008.

5. R. Bisiani. Beam Search. Encyclopedia of Artificial Intelligence, Wiley Interscience Publication, 1992.
6. S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In Proceedings of ASPLOS 2010, ACM.
7. G. Ciardo, J. Gluckman, and D. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. INFORMS Journal on Computing, 10(1), 1998.
8. K.E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: Effective Unit Testing for Concurrency Libraries. In Proceedings of PPoPP 2010, ACM.
9. M.B. Dwyer, S.G. Elbaum, S. Person, and R. Purandare. Parallel Randomized State-Space Search. In Proceedings of ICSE 2007, IEEE CS.
10. M. Dwyer, J. Hatcliff, Robby, and V. Ranganath. Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs. Formal Methods in System Design, 25(2-3), 2004.
11. M.B. Dwyer, S. Person, and S.G. Elbaum. Controlling Factors in Evaluating Path-Sensitive Error Detection Techniques. In Proceedings of SIGSOFT FSE 2006, ACM.
12. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed Explicit-State Model Checking in the Validation of Communication Protocols. International Journal on Software Tools for Technology Transfer, 5(2-3), 2004.
13. S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker, and H. Aljazzar. Survey on Directed Model Checking. In Proceedings of 5th International Workshop on Model Checking and Artificial Intelligence, LNCS, vol. 5348, 2008.
14. N. Een and N. Sorensson. An Extensible SAT-solver. In Proceedings of SAT 2003, LNCS, vol. 2919.
15. M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-Bounded Scheduling. In Proceedings of POPL 2011, ACM.
16. T.A.N. Engels, J.F. Groote, M.J van Weerdenburg, and T.A.C. Willemse. Search algorithms for automated validation. Journal of Logic and Algebraic Programming, 78(4), 2009.
17. A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic Testing. In Proceedings of ESEC/FSE 2013, ACM.
18. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Proceedings of POPL 2005, ACM.
19. P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, 1996.
20. C. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search Through Randomization. In Proceedings of AAAI 1998.
21. A. Groce and W. Visser. Heuristics for Model Checking Java Programs. International Journal on Software Tools for Technology Transfer, 6(4), 2004.
22. G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian Partial-Order Reduction. In Proceedings of SPIN 2007, LNCS, vol. 4595.
23. G.J. Holzmann and D. Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. IEEE Transactions on Software Engineering, 33(10), 2007.
24. G.J. Holzmann, R. Joshi, and A. Groce. Tackling Large Verification Problems with the Swarm Tool. In Proceedings of SPIN 2008, LNCS, vol. 5156.
25. G.J. Holzmann, R. Joshi, and A. Groce. Swarm Verification, In Proceedings of ASE 2008, IEEE CS.
26. V. Jagannath, M. Kirn, Y. Lin, and D. Marinov. Evaluating Machine-Independent Metrics for State-Space Exploration. In Proceedings of ICST 2012, IEEE CS.
27. M. Jones and E. Mercer. Explicit State Model Checking with Hopper. In Proceedings of SPIN 2004, LNCS, vol. 2989.
28. M. Jones and J. Sorber. Parallel Search for LTL Violations. International Journal on Software Tools for Technology Transfer, 7(1), 2005.
29. T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CDx: A Family of Real-Time Java Benchmarks. In Proceedings of JTRES 2009, ACM.
30. A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs. Multi-core Nested Depth-First Search. In Proceedings of ATVA 2011, LNCS, vol. 6996.
31. A. Laarman, J. van de Pol, and M. Weber: Boosting Multi-Core Reachability Performance with Shared Hash Tables. In Proceedings of FMCAD 2010, IEEE.
32. F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In Proceedings of SPIN 1999, LNCS, vol. 1680.
33. M. Luby, A. Sinclair, and D. Zuckerman. Optimal Speedup of Las Vegas Algorithms. Information Processing Letters, 47(4), 1993.
34. A.P.A. van Moorsel and K. Wolter. Analysis of Restart Mechanisms in Software Systems. IEEE Transactions on Software Engineering, 32(8), 2006.
35. M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In Proceedings of PLDI 2007, ACM.
36. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs, In Proceedings of OSDI 2008, USENIX.
37. P. Parízek and T. Kalibera. Efficient Detection of Errors in Java Components Using Random Environment and Restarts. In Proceedings of TACAS 2010, LNCS, vol. 6015.
38. P. Parízek and O. Lhoták. Randomized Backtracking in State Space Traversal. In Proceedings of SPIN 2011, LNCS, vol. 6823.
39. P. Parízek and O. Lhoták. Identifying Future Field Accesses in Exhaustive State Space Traversal. In Proceedings of ASE 2011, IEEE CS.
40. P. Parízek. Hybrid Analysis for Partial Order Reduction of Programs with Arrays. In Proceedings of VMCAI 2016, LNCS, vol. 9583.
41. P. Parízek. Fast Error Detection with Hybrid Analyses of Future Accesses. In Proceedings of SAC 2016, MUSEPAT track, ACM.
42. S. Qadeer. Daisy File System. Joint CAV/ISSTA special event on specification, verification and testing of concurrent software, 2004.
43. S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In Proceedings of TACAS 2005, LNCS, vol. 3440.
44. I. Rabinovitz and O. Grumberg. Bounded Model Checking of Concurrent Programs. In Proceedings of CAV 2005, LNCS, vol. 3576.
45. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Parallel Explicit Model Checking for Generalized Büchi Automata. In Proceedings of TACAS 2015, LNCS, vol. 9035.
46. N. Rungta and E. Mercer. Generating Counter-Examples Through Randomized Guided Search. In Proceedings of SPIN 2007, LNCS, vol. 4595.
47. N. Rungta and E. Mercer. Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs. In Proceedings of PADTAD, ACM, 2009.
48. K. Sen. Effective Random Testing of Concurrent Programs. In Proceedings of ASE 2007, ACM.

49. K. Seppi, M. Jones, and P. Lamborn. Guided Model Checking with a Bayesian Meta-Heuristic. Fundamenta Informaticae, 70(1-2), 2006.
50. L.A. Smith, J.M. Bull, and J. Obdrzalek. A Parallel Java Grande Benchmark Suite. Supercomputing 2001, ACM. https://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html
51. C. Spearman. The Proof and Measurement of Association between Two Things. American Journal of Psychology, 15(1), 1904.
52. U. Stern and D. L. Dill. Parallelizing the Mur*phi* Verifier. In Proceedings of CAV 1997, LNCS, vol. 1254.
53. P. Thomson, A.F. Donaldson, and A. Betts. Concurrency Testing Using Schedule Bounding: an Empirical Study. In Proceedings of PPoPP 2014, ACM.
54. A. Udupa, A. Desai, and S. Rajamani. Depth Bounded Explicit-State Model Checking. In Proceedings of SPIN 2011, LNCS, vol. 6823.
55. A. Vargha and H.D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics, 25(2), 2000.
56. T. Walsh. Search in a Small World. In Proceedings of IJCAI 1999, Morgan Kaufmann.
57. M. Wehrle, S. Kupferschmid, and A. Podelski. Transition-Based Directed Model Checking. In Proceedings of TACAS 2009, LNCS, vol. 5505.
58. M. Wehrle and S. Kupferschmid. Context-Enhanced Directed Model Checking. In Proceedings of SPIN 2010, LNCS, vol. 6349.
59. A. Wijs and D. Bosnacki. Many-core on-the-fly model checking of safety properties using GPUs. International Journal on Software Tools for Technology Transfer, 18(2), 2016.
60. A. Wijs and B. Lisser. Distributed Extended Beam Search for Quantitative Model Checking. In Proceedings of MoChArt 2006, LNCS, vol. 4428.
61. Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008.
62. Concurrency Tool Comparison repository, https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison
63. Java Pathfinder, http://babelfish.arc.nasa.gov/trac/jpf/
64. jPapaBench, http://d3s.mff.cuni.cz/~malohlava/projects/jpapabench/
65. Parallel Java Benchmarks, https://bitbucket.org/psl-lab/pjbench

A Results of Experiments with Twelve Benchmarks

Here we provide Table 11-22 with concrete data on error detection performance that were used to create the graphs in Figure 8. We created one table per benchmark to enable easy comparison of the error detection performance of different configurations and techniques on each benchmark. Each table is divided into three segments. The top segment contains the performance data for selected configurations of DFS-RB from the overall top 10 list (which reflects the combined score). We selected the configurations at positions 1, 2, 3, and 10 in the list in order to show the best ones while also covering the whole range. The middle segment of each table shows the performance data for the state-of-the-art techniques that we included in our experimental comparison. Finally, the bottom segment contains the data for the combination of the overall best configuration of DFS-RB with the state-of-the-art techniques.

Table 11. Results for Daisy file system

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		635290 ± 46187	46 ± 4 s	100%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		539848 ± 86695	38 ± 7 s	100%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		503423 ± 36969	38 ± 3 s	100%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		500124 ± 570299	38 ± 43 s	90%
State-of-the-art techniques	JPF with default search order		timed-out		no
	Bounded thread preemption	5	timed-out		no
		10	timed-out		no
	Guided search with maximized preemption		timed-out		no
	Dynamic POR		timed-out		no
	Heap reach POR and hybrid analyses		timed-out		no
	Heuristics based on hybrid analyses	reordering	timed-out		no
		pruning	timed-out		no
	Random search order		708569 ± 3640	56 ± 0 s	20%
	Restarts of state space traversal	fixed	117695 ± 99225	10 ± 9 s	70%
		Luby	333791 ± 53683	28 ± 5 s	30%
		Walsh	361368 ± 216298	29 ± 17 s	40%
Combinations of DFS-RB with ...	Bounded thread preemption	10	451321 ± 196883	32 ± 15 s	100%
	Guided search with maximized preemption		485374 ± 215229	35 ± 16 s	100%
	Heuristics based on hybrid analyses	reordering	96900 ± 72115	16 ± 12 s	100%
		pruning	17138 ± 14201	3 ± 3 s	50%
	Random search order		279632 ± 222617	19 ± 16 s	100%
	Restarts of state space traversal	fixed	669970 ± 360195	47 ± 27 s	70%

Table 12. Results for Elevator

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		2322 ± 3608	1 ± 0 s	100%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		789 ± 1005	1 ± 0 s	100%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		3251 ± 3522	26 ± 30 s	90%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		204639 ± 354045	15 ± 26 s	80%
State-of-the-art techniques	JPF with default search order		timed-out		no
	Bounded thread preemption	5	timed-out		no
		10	timed-out		no
	Guided search with maximized preemption		663	1 s	yes
	Dynamic POR		timed-out		no
	Heap reach POR and hybrid analyses		timed-out		no
	Heuristics based on hybrid analyses	reordering	timed-out		no
		pruning	timed-out		no
	Random search order		302 ± 193	1 ± 0 s	80%
	Restarts of state space traversal	fixed	66944 ± 60678	5 ± 5 s	100%
		Luby	63507 ± 92710	5 ± 7 s	90%
		Walsh	59169 ± 105894	5 ± 9 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	1372 ± 2052	1 ± 0 s	100%
	Guided search with maximized preemption		6645 ± 5271	1 ± 1 s	100%
	Heuristics based on hybrid analyses	reordering	44668 ± 5374	1 ± 2 s	100%
		pruning	5579 ± 8855	2 ± 3 s	100%
	Random search order		179139 ± 397784	14 ± 31 s	100%
	Restarts of state space traversal	fixed	4766 ± 7007	1 ± 1 s	100%

Table 13. Results for Alarm Clock

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		59 ± 57	1 ± 0 s	100%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		59 ± 70	1 ± 0 s	100%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		140 ± 150	1 ± 0 s	100%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		80 ± 115	1 ± 0 s	100%
State-of-the-art techniques	JPF with default search order		474	1 s	yes
	Bounded thread preemption	5	474	1 s	yes
		10	474	1 s	yes
	Guided search with maximized preemption		15	1 s	yes
	Dynamic POR		1002	1 s	yes
	Heap reach POR and hybrid analyses		309	1 s	yes
	Heuristics based on hybrid analyses	reordering	309	1 s	yes
		pruning	189	1 s	yes
	Random search order		361 ± 222	1 ± 0 s	100%
	Restarts of state space traversal	fixed	356 ± 247	1 ± 0 s	100%
		Luby	175 ± 248	1 ± 0 s	100%
		Walsh	374 ± 254	1 ± 0 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	45 ± 25	1 ± 0 s	100%
	Guided search with maximized preemption		216 ± 196	1 ± 0 s	100%
	Heuristics based on hybrid analyses	reordering	34 ± 14	1 ± 1 s	100%
		pruning	68 ± 22	1 ± 1 s	100%
	Random search order		126 ± 156	1 ± 0 s	100%
	Restarts of state space traversal	fixed	73 ± 61	1 ± 0 s	100%

Table 14. Results for Linked List

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		92 ± 43	1 ± 0 s	100%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		55 ± 26	1 ± 0 s	100%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		49 ± 33	1 ± 0 s	100%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		37 ± 37	1 ± 0 s	100%
State-of-the-art techniques	JPF with default search order		1555	1 s	yes
	Bounded thread preemption	5	1555	1 s	yes
		10	1555	1 s	yes
	Guided search with maximized preemption		46	1 s	yes
	Dynamic POR		534	1 s	yes
	Heap reach POR and hybrid analyses		296	1 s	yes
	Heuristics based on hybrid analyses	reordering	296	2 s	yes
		pruning	error states pruned		no
	Random search order		27 ± 14	1 ± 0 s	100%
	Restarts of state space traversal	fixed	25 ± 11	1 ± 0 s	100%
		Luby	25 ± 9	1 ± 0 s	100%
		Walsh	23 ± 6	1 ± 0 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	91 ± 51	1 ± 0 s	100%
	Guided search with maximized preemption		38 ± 29	1 ± 0 s	100%
	Heuristics based on hybrid analyses	reordering	18 ± 8	2 ± 1 s	100%
		pruning	error states pruned in all runs		0%
	Random search order		22 ± 4	1 ± 0 s	100%
	Restarts of state space traversal	fixed	82 ± 65	1 ± 0 s	100%

Table 15. Results for Producer Consumer

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		119 ± 60	1 ± 0 s	100%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		93 ± 37	1 ± 0 s	100%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		91 ± 36	1 ± 0 s	100%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		94 ± 38	1 ± 0 s	100%
State-of-the-art techniques	JPF with default search order		60521	5 s	yes
	Bounded thread preemption	5	60521	5 s	yes
		10	60521	5 s	yes
	Guided search with maximized preemption		335	1 s	yes
	Dynamic POR		14561	2 s	yes
	Heap reach POR and hybrid analyses		19550	4 s	yes
	Heuristics based on hybrid analyses	reordering	8482	3 s	yes
		pruning	8473	3 s	yes
	Random search order		65 ± 19	1 ± 0 s	100%
	Restarts of state space traversal	fixed	53 ± 12	1 ± 0 s	100%
		Luby	63 ± 23	1 ± 0 s	100%
		Walsh	69 ± 26	1 ± 0 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	82 ± 26	1 ± 0 s	100%
	Guided search with maximized preemption		56 ± 6	1 ± 0 s	100%
	Heuristics based on hybrid analyses	reordering	54 ± 18	2 ± 1 s	100%
		pruning	46 ± 10	2 ± 1 s	100%
	Random search order		57 ± 13	1 ± 0 s	100%
	Restarts of state space traversal	fixed	91 ± 34	1 ± 0 s	100%

Table 16. Results for RAX Extended

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		88 ± 52	1 ± 0 s	100%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		46 ± 11	1 ± 0 s	100%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		66 ± 35	1 ± 0 s	100%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		46 ± 7	1 ± 0 s	100%
State-of-the-art techniques	JPF with default search order		618	1 s	yes
	Bounded thread preemption	5	618	1 s	yes
		10	618	1 s	yes
	Guided search with maximized preemption		47	1 s	yes
	Dynamic POR		141	1 s	yes
	Heap reach POR and hybrid analyses		74	2 s	yes
	Heuristics based on hybrid analyses	reordering	74	2 s	yes
		pruning	63	2 s	yes
	Random search order		63 ± 56	1 ± 0 s	100%
	Restarts of state space traversal	fixed	30 ± 15	1 ± 0 s	100%
		Luby	60 ± 54	1 ± 0 s	100%
		Walsh	51 ± 44	1 ± 0 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	52 ± 21	1 ± 0 s	100%
	Guided search with maximized preemption		50 ± 33	1 ± 0 s	100%
	Heuristics based on hybrid analyses	reordering	31 ± 8	2 ± 1 s	100%
		pruning	29 ± 13	2 ± 1 s	100%
	Random search order		32 ± 15	1 ± 0 s	100%
	Restarts of state space traversal	fixed	110 ± 75	1 ± 0 s	100%

Table 17. Results for Replicated Workers

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle$I, pl, d, Lb, 0.75, 1.5$\rangle$		1259 ± 982	1 ± 0 s	100%
	2. $\langle$I, pl, d, Lb, 0.90^r, 1.2$\rangle$		1428 ± 693	1 ± 0 s	100%
	3. $\langle$I, pl, ncs, F, $1-r/5$, 1$\rangle$		2811 ± 5087	1 ± 0 s	100%
	10. $\langle$I, pl, d, F, 0.75^r, 1.5$\rangle$		1159843 ± 1161713	80 ± 80 s	90%
State-of-the-art techniques	JPF with default search order		timed-out		no
	Bounded thread preemption	5	timed-out		no
		10	timed-out		no
	Guided search with maximized preemption		224	1 s	yes
	Dynamic POR		timed-out		no
	Heap reach POR and hybrid analyses		timed-out		no
	Heuristics based on hybrid analyses	reordering	165	5 s	yes
		pruning	149	5 s	yes
	Random search order		179 ± 38	1 ± 0 s	50%
	Restarts of state space traversal	fixed	18478 ± 40287	1 ± 3 s	100%
		Luby	31663 ± 56124	2 ± 5 s	100%
		Walsh	39187 ± 117130	3 ± 9 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	1258 ± 762	1 ± 0 s	100%
	Guided search with maximized preemption		440 ± 318	1 ± 0 s	100%
	Heuristics based on hybrid analyses	reordering	258 ± 107	2 ± 1 s	100%
		pruning	313 ± 221	2 ± 1 s	100%
	Random search order		1526 ± 2005	1 ± 0 s	100%
	Restarts of state space traversal	fixed	1295 ± 769	1 ± 0 s	100%

Table 18. Results for jPapaBench

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle$I, pl, d, Lb, 0.75, 1.5$\rangle$		6157 ± 4718	1 ± 0 s	100%
	2. $\langle$I, pl, d, Lb, 0.90^r, 1.2$\rangle$		16912 ± 12480	1 ± 1 s	100%
	3. $\langle$I, pl, ncs, F, $1-r/5$, 1$\rangle$		1170 ± 411	1 ± 0 s	100%
	10. $\langle$I, pl, d, F, 0.75^r, 1.5$\rangle$		14098 ± 1974	1 ± 0 s	100%
State-of-the-art techniques	JPF with default search order		timed-out		no
	Bounded thread preemption	5	timed-out		no
		10	timed-out		no
	Guided search with maximized preemption		52	1 s	yes
	Dynamic POR		timed-out		no
	Heap reach POR and hybrid analyses		timed-out		no
	Heuristics based on hybrid analyses	reordering	timed-out		no
		pruning	timed-out		no
	Random search order		232 ± 217	1 ± 0 s	100%
	Restarts of state space traversal	fixed	349 ± 210	1 ± 0 s	100%
		Luby	404 ± 208	1 ± 0 s	100%
		Walsh	241 ± 192	1 ± 0 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	3405 ± 1654	1 ± 0 s	100%
	Guided search with maximized preemption		5780 ± 3921	1 ± 0 s	100%
	Heuristics based on hybrid analyses	reordering	3539 ± 6553	3 ± 5 s	100%
		pruning	3933 ± 7304	5 ± 6 s	100%
	Random search order		412 ± 269	1 ± 0 s	100%
	Restarts of state space traversal	fixed	4295 ± 1438	1 ± 0 s	100%

Table 19. Results for Monte Carlo

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		2933 ± 866	1 ± 0 s	100%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		20657 ± 15910	12 ± 10 s	100%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		1088 ± 433	1 ± 0 s	100%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		3546 ± 1724	1 ± 1 s	100%
State-of-the-art techniques	JPF with default search order		timed-out		no
	Bounded thread preemption	5	timed-out		no
		10	timed-out		no
	Guided search with maximized preemption		14962	16 s	yes
	Dynamic POR		timed-out		no
	Heap reach POR and hybrid analyses		timed-out		no
	Heuristics based on hybrid analyses	reordering	timed-out		no
		pruning	timed-out		no
	Random search order		2911 ± 2671	1 ± 3 s	100%
	Restarts of state space traversal	fixed	3427 ± 3458	2 ± 2 s	100%
		Luby	2503 ± 2357	1 ± 1 s	100%
		Walsh	3727 ± 4601	2 ± 3 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	3942 ± 995	2 ± 0 s	100%
	Guided search with maximized preemption		5593 ± 3537	3 ± 2 s	100%
	Heuristics based on hybrid analyses	reordering	all runs timed-out		0%
		pruning	all runs timed-out		0%
	Random search order		6812 ± 5196	4 ± 3 s	100%
	Restarts of state space traversal	fixed	3109 ± 1845	1 ± 1 s	100%

Table 20. Results for CDx

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		error states pruned in all runs		0%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		205 ± 0	1 ± 0 s	10%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		error states pruned in all runs		0%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		error states pruned in all runs		0%
State-of-the-art techniques	JPF with default search order		6777	16 s	yes
	Bounded thread preemption	5	6777	16 s	yes
		10	6777	16 s	yes
	Guided search with maximized preemption		4601	12 s	yes
	Dynamic POR		timed-out		no
	Heap reach POR and hybrid analyses		860	17 s	yes
	Heuristics based on hybrid analyses	reordering	784	19 s	yes
		pruning	681	19 s	yes
	Random search order		1403 ± 1906	3 ± 5 s	100%
	Restarts of state space traversal	fixed	2103 ± 2634	5 ± 7 s	100%
		Luby	2998 ± 5201	7 ± 14 s	100%
		Walsh	3125 ± 4002	8 ± 10 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	error states pruned in all runs		0%
	Guided search with maximized preemption		error states pruned in all runs		0%
	Heuristics based on hybrid analyses	reordering	7355 ± 128	81 ± 10 s	100%
		pruning	2157 ± 621	67 ± 9 s	100%
	Random search order		4462 ± 4383	4 ± 5 s	90%
	Restarts of state space traversal	fixed	error states pruned in all runs		0%

Table 21. Results for Cache4j

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		13072 ± 4828	1 ± 0 s	100%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		15271 ± 5948	1 ± 0 s	90%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		error states pruned in all runs		0%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		294 ± 127	1 ± 0 s	100%
State-of-the-art techniques	JPF with default search order		24991	5 s	yes
	Bounded thread preemption	5	24991	5 s	yes
		10	24991	6 s	yes
	Guided search with maximized preemption		14201	3 s	yes
	Dynamic POR		timed-out		no
	Heap reach POR and hybrid analyses		22072	7 s	yes
	Heuristics based on hybrid analyses	reordering	22072	10 s	yes
		pruning	19017	8 s	yes
	Random search order		9933 ± 6318	1 ± 0 s	100%
	Restarts of state space traversal	fixed	6920 ± 6755	1 ± 0 s	100%
		Luby	9741 ± 4967	1 ± 1 s	100%
		Walsh	10323 ± 5351	1 ± 1 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	6788 ± 4855	1 ± 0 s	100%
	Guided search with maximized preemption		8318 ± 7541	1 ± 0 s	100%
	Heuristics based on hybrid analyses	reordering	3779 ± 2261	2 ± 1 s	100%
		pruning	5221 ± 961	3 ± 1 s	100%
	Random search order		5580 ± 2988	1 ± 0 s	100%
	Restarts of state space traversal	fixed	12423 ± 7751	1 ± 0 s	100%

Table 22. Results for QSortMT

	Configuration/Technique		States	Time	Found
DFS-RB algorithm (selection from top 10)	1. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.75, 1.5 \rangle$		5172 ± 4081	1 ± 0 s	100%
	2. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{Lb}, 0.90^r, 1.2 \rangle$		5746 ± 3817	1 ± 0 s	100%
	3. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{ncs}, \mathrm{F}, 1-r/5, 1 \rangle$		9206 ± 6091	1 ± 0 s	60%
	10. $\langle \mathrm{I}, \mathrm{pl}, \mathrm{d}, \mathrm{F}, 0.75^r, 1.5 \rangle$		561 ± 284	1 ± 0 s	100%
State-of-the-art techniques	JPF with default search order		6277	1 s	yes
	Bounded thread preemption	5	6277	1 s	yes
		10	6277	1 s	yes
	Guided search with maximized preemption		65	1 s	yes
	Dynamic POR		timed-out		no
	Heap reach POR and hybrid analyses		2094	3 s	yes
	Heuristics based on hybrid analyses	reordering	2094	3 s	yes
		pruning	24	2 s	yes
	Random search order		807 ± 844	1 ± 0 s	100%
	Restarts of state space traversal	fixed	482 ± 705	1 ± 0 s	100%
		Luby	553 ± 807	1 ± 0 s	100%
		Walsh	1009 ± 873	1 ± 0 s	100%
Combinations of DFS-RB with ...	Bounded thread preemption	10	1629 ± 1623	1 ± 0 s	100%
	Guided search with maximized preemption		1504 ± 3144	1 ± 0 s	90%
	Heuristics based on hybrid analyses	reordering	873 ± 1341	1 ± 1 s	100%
		pruning	error states pruned in all runs		0%
	Random search order		843 ± 1212	1 ± 0 s	100%
	Restarts of state space traversal	fixed	8671 ± 8325	1 ± 0 s	100%