Randomized Backtracking in State Space Traversal

Pavel Parízek and Ondřej Lhoták

David R. Cheriton School of Computer Science, University of Waterloo

Abstract. While exhaustive state space traversal is not feasible in reasonable time for complex concurrent programs, many techniques for efficient detection of concurrency errors and testing of concurrent programs have been introduced in recent years, such as directed search and context-bounded model checking.
We propose to use depth-first traversal with randomized backtracking, where it is possible to backtrack from a state before all outgoing transitions have been explored, and the whole process is driven by random number choices. Experiments with a prototype implementation in JPF on several Java programs show that, in most cases, fewer states must be explored to find an error with our approach than using the existing techniques.

1 Introduction

Exhaustive state space traversal is a means of software verification that is suitable especially for concurrent systems and detection of concurrency errors like deadlocks and race conditions. It is, however, not feasible for large and complex software systems with many threads, because the number of possible thread interleavings that must be explored is too high. A common practice is to employ state space traversal for testing of concurrent programs and detection of concurrency errors. Many techniques that aim to find errors in a reasonable time have recently been proposed. Techniques in one group are based on directed (guided) search that uses heuristics for navigation of the search towards the error state [5–7, 11, 21], so that 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. Some other techniques use randomization or parallel state space traversal with the goal of finding errors in less time [2, 3, 9, 11, 18]. Approaches from another category perform incomplete search with the assumption that many important errors can be found in a particular small part of the system's state space and searching for the other errors is not tractable because of state explosion. This category includes bounding of the number of thread context switches [16] in explicit-state model checking [13] and SAT-based model checking [17], and a random partial order sampling algorithm [20].

In this paper we propose to use *randomized backtracking* in explicit state depth-first traversal for the purpose of efficient detection of concurrency errors. The key ideas are (1) to allow backtracking also from states that still have some unexplored outgoing transitions and (2) to use the results of random number choice in the decision whether to backtrack from such a state (and thus prune a part of the state space) or continue forward along some unexplored transition.

We implemented our approach in Java PathFinder [10] and evaluated it on seven multi-threaded Java programs that contain various concurrency errors. State space traversal with randomized backtracking has better performance than existing techniques supported by JPF on six of the seven benchmark programs, i.e. fewer states are explored by JPF before it detects an error. For the last benchmark, our approach has comparable performance with the best existing technique. 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 backtracking from a state with unexplored transitions, so it cannot be guaranteed that all errors will be discovered.

2 Background and Related Work

We consider only explicit state space traversal, where each state is a snapshot of program variables and threads at one point on one execution path and each transition is a sequence of instructions executed by one thread — each transition is associated with a specific thread. We further assume that each transition in the state space is bounded by non-deterministic choices (e.g., thread scheduling choices). Figure 1 shows the basic algorithm for depth-first state space traversal in this context. The function enabled returns a set of transitions enabled in the state s that must be explored.

```
DFS ( ) :
  visited = {}
  stack = [ ]
  push( stack , s0 )
  explore( s0 )

procedure explore( s )
  if error( s ) then
    counterexample = stack
    terminate
  end if
  transitions = order( enabled( s ) )
  for tr ∈ transitions do
    s' = execute( tr )
    if s' ∉ visited then
      visited = visited ∪ s'
      push( stack , s' )
      explore( s' )
      pop( stack )
    end if
  end for
end proc
```

Fig. 1. Algorithm for depth-first state space traversal

An important parameter of the algorithm for depth-first traversal is the *search order* that determines the sequence in which the transitions leading from a state are explored. The search order is implemented by the function order, which creates a list of transitions with a specific ordering from the given set.

In the rest of this section, we describe in more detail selected existing approaches to more efficient detection of concurrency errors that are based on state space traversal.

Each tool for state space traversal of concurrent programs uses a specific default search order. The basic implementation of the function enabled in most of the tools returns a set that contains one transition for each thread runnable in the given state s. Many optimizations aiming at efficient detection of errors can be expressed through different implementations of the functions enabled and order.

Techniques based on directed (guided) search typically use custom implementations of the order function, which sort transitions according to some heuristic function over the end states of the transitions or over the transitions themselves. A useful heuristic for detection of concurrency errors is to prefer thread context switches (thread interleavings) [7] — when the state s being processed 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 [22]. Two transitions can interfere, for example, if they are associated with different threads and contain instructions that access the same shared variables.

It is also possible to use a random search order, in which the transitions leading from a state s are explored in a random order and the order is selected separately for each state (i.e., there is no common random order for all states). The technique described in [18] combines guided search with a random order such that transitions with the same value of the heuristic function are ordered randomly. Another possible use of randomization, which was proposed in [2], is to explore the complete execution paths in a random order during stateless search.

All techniques that use a custom implementation only for the order function still explore the whole state space if they detect no error during traversal. For an incomplete state space traversal, it is necessary to use a custom implementation of the enabled function that prunes the state space according to some criteria. A very popular approach is to bound the number of thread preemptions (context switches) on each explored state space path [16], where preemption means that the thread associated with a previous transition is still runnable but a transition associated with some other thread is selected. The set returned by the customized function enabled does not contain a transition if its selection would exceed the number of allowed preemptions. The value of the bound determines how many errors are found and the time cost of the search, i.e. a higher bound means that more errors are found and the search takes more time, but it was shown that many errors are found with only two context switches [13].

3 Randomized Backtracking

We alter the standard algorithm for depth-first state space traversal with the following two modifications.

- Rather than backtracking only from fully processed states (whose outgoing transitions have all been explored) and already visited states, the modified algorithm may also backtrack from states still containing unexplored outgoing transitions.

– When the state s that is currently being processed has some unexplored outgoing transitions, random number choice is used to decide whether (a) to move forward and explore one of the outgoing transitions or (b) backtrack already and ignore the remaining unexplored transitions.

The process of state space traversal with randomized backtracking is controlled by three parameters, which will be explained next: (1) *threshold* to enable the randomized backtracking at some search depth, (2) *strategy* to determine the length of backtrack jumps, and (3) *ratio* to decide between going forward and backtracking from a state with unexplored transitions. The parameters are used together to decide whether to backtrack from a state with unexplored transitions and how the results of random number choices are used. Specific values of the parameters make a *configuration* of randomized backtracking.

Algorithm. Figure 2 shows the algorithm for depth-first state space traversal with randomized backtracking. Differences from the basic algorithm in Figure 1 are highlighted by underlining. For each unexplored transition tr from the state s, the algorithm decides whether to execute the transition or backtrack to some earlier state on the current path. The function call rnd(0,1) returns a random number from the interval $\langle 0, 1 \rangle$. After each backtracking step, i.e. after the recursive call to explore returns, the backtrackAgain procedure is used to determine whether, according to the selected strategy, the algorithm should backtrack further. Note that randomized backtracking does not involve custom implementations of the functions order and enabled, and therefore our algorithm can be combined with any existing technique that uses special versions of these functions.

Threshold. Backtracking from a state with some unexplored outgoing transitions is enabled only if the current search depth (i.e., the number of transitions between the initial state and the current state) is greater than the value of the threshold parameter. The algorithm will not backtrack from the state s whose depth is smaller than the threshold until all transitions outgoing from s are explored. Setting this parameter to a specific non-zero value is useful, for example, when the prefix of each state space path represents an initialization phase and the algorithm should not backtrack too early, pruning the state space, before it reaches the interesting part of the state space with respect to the presence of concurrency errors.

Strategy. When the traversal algorithm decides to backtrack from the state s, either because it is fully processed or using the results of random choice, it uses the given strategy to determine the number of transitions that it backtracks over (the length of the backtrack jump). If the strategy defines a backtrack jump of a length greater than the current search depth, the algorithm backtracks through all transitions on the current path and the state space traversal finishes. We consider the following three strategies: fixed, random, and Luby.

The *fixed strategy* corresponds to the behavior of the standard algorithm for depth-first traversal. It means that the algorithm backtracks over a single transition and then decides whether to go forward along some unexplored transition or backtrack again.

```
DFS_RB ( threshold, strategy, ratio ) :
  visited = {}
  stack = [ ]
  push ( stack , s0 )
  explore ( s0 )

procedure explore ( s )
  if error ( s ) then
    counterexample = stack
    terminate
  end if
  transitions = order ( enabled ( s ) )
  for tr ∈ transitions do
    depth = size(stack)
    if depth >= threshold then
      if rnd(0, 1) > ratio(depth) return
    end if
    s' = execute ( tr )
    if s' ∉ visited then
      visited = visited ∪ s'
      push ( stack , s' )
      explore ( s' )
      pop ( stack )
      if backtrackAgain(strategy) return
    end if
  end for
end proc
```

Fig. 2. Algorithm for depth-first state space traversal with randomized backtracking

When the *random strategy* is used, the algorithm backtracks over a random number of transitions at each occasion, i.e. each backtrack jump has a random length. This strategy imposes no bound on the length of the backtrack jumps.

Usage of the *Luby strategy* [12] requires the algorithm to record the total count of backtracking jumps already performed from the start of the state space traversal. The length of a backtrack jump N is equal to the number 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 first few elements of the sequence are: 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8. For example, the third backtrack jump will step over two transitions. For any integer $n > 0$, 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 is bounded by the number of already performed backtrack jumps.

Ratio. The ratio parameter allows to express the degree of general preference for going forward along some unexplored transition over backtracking for decisions based on

random number choice. Assuming that the value of this parameter is R, when the state s that is currently being processed has some unexplored transitions, then the algorithm backtracks from s (instead of exploring some outgoing transition) only if the randomly selected number from the interval $\langle 0, 1\rangle$ is greater than R. The ratio can be defined as a constant number or as a function of the search depth that is represented by an expression $R = 1 - d/c$, where d is the current search depth and c is an integer constant. In the latter case, backtracking becomes more likely from states with a greater depth.

State space pruning. A consequence of the use of randomized backtracking is that an incomplete traversal is performed, because parts of the state space are pruned by backtracking from a state with some unexplored transitions, and therefore errors may be discovered in less time than with existing techniques (e.g., with the exhaustive search). On the other hand, it is not guaranteed that an error state is reached when the randomized backtracking is used, and therefore no error may be detected for some configurations and randomly chosen numbers.

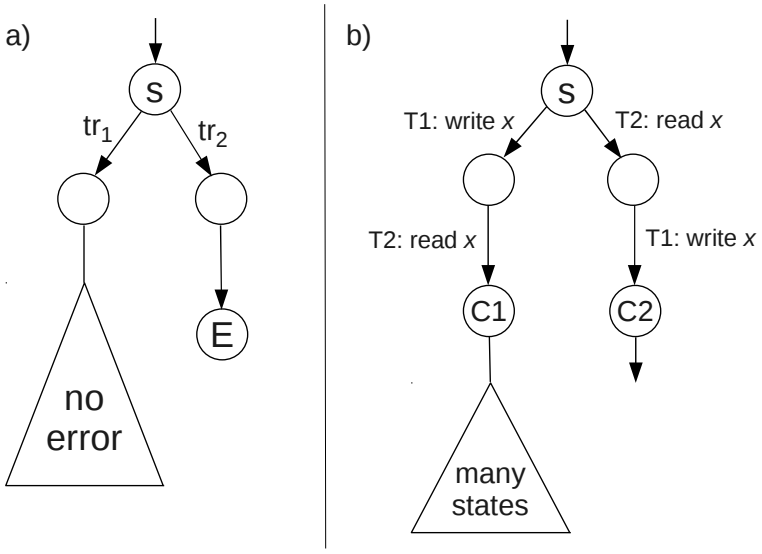

Fig. 3. State space fragments that illustrate pruning by randomized backtracking

Consider the state space fragment in Figure 3a. Assuming that the transition tr_1 from the state s is explored first, randomized backtracking may prune the possibly large part of the state space that does not contain any error state and thus avoid spending a lot of time in traversing the error-free part. The standard algorithm would exhaustively traverse the whole error-free part before exploring the transition tr_2.

Figure 3b shows how randomized backtracking can reduce the time needed to discover a race condition, which involves a pair of unsynchronized accesses to the same

variable x in two threads $T1$ and $T2$ such that at least one of the accesses is a write and the accesses are performed in different orders in different thread schedules. After exploring the state c_1 on the state space path that corresponds to the schedule $T1; T2$, in which the first sequence of conflicting accesses is detected, the large subtree below c_1 may be pruned and a different state space path that corresponds to the schedule $T2; T1$ may be explored instead. In this case, the race condition is recognized when the state c_2 is reached and the second sequence of conflicting accesses with reverse order is detected.

For each state s, the algorithm decides whether to backtrack (based on random number choice) separately for each transition outgoing from s. A consequence of this behavior is that each outgoing transition from s has a different probability (chance) that it will be explored and not pruned — for the first transition in the list returned by the order function, the probability that it will be explored is equal to the ratio R, while a transition with the index i in the list has the probability R^i that it will be explored.

4 Evaluation

We implemented the proposed algorithm for depth-first state space traversal with randomized backtracking in Java PathFinder (JPF) [10] and evaluated its performance on seven multi-threaded Java programs: the Daisy file system [15], the Elevator benchmark from the PJBench suite [14], and five small programs used in a recent comparison of tools for detection of concurrency errors [19] that are publicly available in the CTC repository [1] — the programs are Alarm Clock, Linked List, Producer Consumer, RAX Extended, and Replicated Workers.

Basic characteristics of the programs are provided in Table 1 — total number of source code lines and maximal number of concurrently running threads.

Our implementation and the complete set of experimental results are available at the web site `http://plg.uwaterloo.ca/~pparizek/jpf/spin11/`. The benchmark programs can be downloaded from web sites listed in the references.

Program	Source code lines	Number of threads
Daisy file system	1150	2
Elevator	320	4
Alarm Clock	180	3
Linked List	185	2
Producer Consumer	135	7
RAX Extended	150	5
Replicated Workers	430	6

Table 1. Information about benchmark programs

We designed experiments on the seven benchmark programs to answer the following three questions about the performance of the state space traversal with randomized backtracking. In the context of this paper, performance corresponds to the number of

states processed before an error is found — a lower number of processed states implies better performance.

Q1) For each benchmark, which configuration of randomized backtracking has the best performance (i.e., processes the least number of states before finding an error), and does it have better performance than existing techniques?
Q2) How much different is the performance of different configurations of randomized backtracking for each individual benchmark, i.e. what is the variability of performance over all configurations?
Q3) Is there a configuration that has reasonably good performance (better than existing techniques) for all programs?

The rest of this section contains a description of the experimental setup and then answers to the three questions. In the experimental setup and presentation of results, we followed the general recommendations for evaluating path-sensitive error detection techniques that are described in [4]. We provide values of system-independent metrics, like the number of processed states and depth of the error state.

Experimental setup. For the purpose of the experiments, we manually injected concurrency errors into those benchmark programs that did not already contain any — we created race conditions in all benchmarks except Linked List and Daisy by modifying the scope of synchronized blocks, and we inserted assertions into Daisy that are violated as a consequence of complex race conditions that already existed in the code but JPF cannot detect them directly. The Linked List benchmark already contained a race condition that JPF can detect. We tried to inject such errors as to get benchmarks with low density of error paths (as recommended in [4]), i.e., to inject hard-to-find errors, but this was not possible in some cases without changing the benchmark design and code very significantly. The following benchmarks have a low number of error paths: Daisy file system (0.03 % of paths lead to the error state), Elevator (0.006 %), and RAX Extended (3 %). Other benchmarks contain easy-to-find bugs, for which a large percentage of state space paths lead to the error.

We consider the following existing techniques that are implemented in JPF: exhaustive traversal with default search order, traversal with random search order, directed search with a heuristic that prefers thread interleavings, and context-bounded search. The default search order of JPF means that transitions in the list are ordered by the indices of the associated threads. For the context-bounded search, we considered the following bounds on the number of thread preemptions: 2, 5, 10.

With randomized backtracking, we performed experiments with configurations that correspond to all combinations of threshold values from the set $\{5, 10, 20, 50, 100\}$, values of the ratio parameter from the set $\{0.5, 0.75, 0.9, 0.99, 1 - d/20, 1 - d/50, 1 - d/100, 1 - d/1000\}$, where d is the current search depth, and all three strategies for the length of backtrack jump (fixed, random, and Luby). Names of configurations have the form of tuples (threshold, strategy, ratio).

For experiments that involve randomization, we repeated JPF runs with the same configuration until either 10 runs found an error or 100 runs were performed. The context-bounded traversal is not complete, but it does not involve any randomization, and thus the percentage of JPF runs that find an error is always either 100% or 0%.

Answer to question Q1 (best configuration for each benchmark). Table 2 provides for each benchmark program the experimental results for: (1) traversal with the default search order, (2) the existing technique with the best performance, and (3) the configuration of randomized backtracking with the best average performance from those where 100% of JPF runs found an error. For some benchmarks (Daisy file system, Alarm Clock, Linked List, and RAX Extended), better performance can be achieved by requiring only 50% of JPF runs to find an error. For these benchmarks, the table contains a fourth row showing the best performing such configuration. Note that for the Elevator benchmark, there was no configuration for which the number of JPF runs that found an error is in the interval $[50, 100)$. Each row contains values of the following metrics: the number of states processed before an error was found (mean μ, minimum, maximum, and standard deviation σ), and the percentage of JPF runs for the given configuration that found an error. The running times of JPF are less than one minute for Elevator and Replicated Workers, and a few seconds for all other benchmarks.

The results show that state space traversal with randomized backtracking has much better performance than all of the existing techniques for six of the seven benchmarks. For each of these benchmarks, there is a configuration of randomized backtracking with which 100% of JPF runs find an error, yet the number of explored states is a factor of 1.1 to 44 times lower than for the best existing technique. The exception is the Producer Consumer benchmark, for which both complete search with random search order and randomized backtracking have similar performance; they both explore fewer states than the JPF default search order by a factor of over 160. Although the state space traversal with random search order yields a lower minimal number of explored states than randomized backtracking for the Alarm Clock and Linked List benchmarks, randomized backtracking yields a lower average and maximum than random search order, and therefore we claim that randomized backtracking has better performance for these two benchmarks. Usage of randomized backtracking has significantly better performance than existing techniques in particular for benchmark programs with deep errors (such that a large number of states must be explored to find the error), like the Elevator and Replicated Workers benchmarks, for which the mean number of states explored is reduced by a factor of 8.9 and 44, respectively.

For some benchmark programs, significantly better performance can be achieved by the appropriate configurations when it is not required that all JPF runs find an error — see data for Daisy file system and RAX Extended, which show improvement by a factor of 2.1 and 12, respectively, over the best configuration where 100% of JPF runs found an error. If the percentage of JPF runs that find an error is greater than 50% and each run finishes quickly, then an error would be found with a very high probability (close to 100%) by performing a sequence of JPF runs with the given configuration. The total running time of this sequence of JPF runs might be smaller than the running time of some existing technique and also than the running time of a single JPF run with a different configuration (for which every JPF run that we performed found an error).

Answer to question Q2 (variability of performance by configuration). Table 3 provides for each benchmark program the experimental results for configurations of randomized backtracking that yield the following extremes over the set of all JPF runs for

Configuration	Processed states				Error found
	μ	min	max	σ	
Daisy file system					
default search order	282			0	100 %
thread interleavings	139			0	100 %
(10, random, $1 - d/1000$)	108	102	116	5	100 %
(5, fixed, 0.75)	51	22	88	19	56 %
Elevator					
default search order	143373			0	100 %
random search order	2399	1062	3833	787	100 %
(50, random, $1 - d/100$)	270	255	293	12	100 %
Alarm Clock					
default search order	188			0	100 %
random search order	192	12	380	111	100 %
(10, Luby, $1 - d/1000$)	44	20	165	41	100 %
(5, fixed, $1 - d/20$)	37	15	93	22	91 %
Linked List					
default search order	328			0	100 %
random search order	186	15	234	70	100 %
(10, fixed, $1 - d/50$)	112	51	215	59	100 %
(10, Luby, 0.99)	58	50	71	7	59 %
Producer Consumer					
default search order	9299			0	100 %
random search order	48	25	73	13	100 %
(10, fixed, $1 - d/100$)	57	23	157	41	100 %
RAX Extended					
default search order	1617			0	100 %
thread interleavings	104			0	100 %
(10, Luby, 0.99)	97	86	113	8	100 %
(5, Luby, 0.9)	8	8	9	0	59 %
Replicated Workers					
default search order	9881			0	100 %
context bound (10)	6585			0	100 %
(50, fixed, 0.9)	148	95	278	55	100 %

Table 2. Configurations with the best average performance

all configurations: minimal number of states processed by some JPF run and maximal number of states processed by some JPF run. We consider only configurations where 50% or more of JPF runs detected some error. Moreover, the table provides also the configuration for which the lowest percentage of JPF runs discovered an error.

Each row of the table contains values of the following metrics: number of states processed before an error was found (mean μ, minimum, maximum, and standard deviation σ), and percentage of JPF runs for the given configuration that found some error. If no error was found by any JPF run for some configuration, then columns for all met-

Configuration	Processed states				Error found
	μ	min	max	σ	
Daisy file system					
(5, fixed, 0.9)	135	19	467	165	67 %
(20, fixed, 0.5)	282	282	282	0	100 %
(5, random, $1 - d/20$)	-	-	-	-	0 %
Elevator					
(50, fixed, 0.99)	358	253	424	49	100 %
(100, fixed, $1 - d/50$)	2263	2227	2338	38	100 %
(5, fixed, 0.5)	-	-	-	-	0 %
Alarm Clock					
(5, fixed, 0.5)	85	13	251	74	100 %
(5, fixed, 0.9)	94	13	447	120	100 %
(5, random, $1 - d/100$)	57	19	112	30	19 %
Linked List					
(5, fixed, $1 - d/50$)	74	38	133	27	56 %
(20, fixed, 0.9)	235	170	408	66	100 %
(5, Luby, 0.5)	-	-	-	-	0 %
Producer Consumer					
(10, fixed, $1 - d/50$)	196	18	1251	355	100 %
(50, fixed, 0.5)	9299	9299	9299	0	100 %
(5, Luby, $1 - d/1000$)	242	242	242	0	1 %
RAX Extended					
(5, fixed, 0.5)	60	8	313	88	67 %
(50, fixed, 0.5)	1617	1617	1617	0	100 %
(5, random, $1 - d/50$)	10	8	15	2	26 %
Replicated Workers					
(50, fixed, 0.5)	339	71	1282	385	100 %
(50, Luby, $1 - d/20$)	6258	139	19190	5673	100 %
(5, random, 0.5)	-	-	-	-	0 %

Table 3. Configurations that yield performance extremes

rics related to the number of processed states and search depth in the table contain the character "-".

The results show that there is great variability in performance yielded by different configurations and different outcomes of random number choices on each benchmark program. In particular, the minimal and maximal numbers of states processed by a JPF run that were recorded over all configurations and JPF runs differ by an order of magnitude for some benchmarks (e.g., for the Producer Consumer and Replicated Workers benchmarks). Note that the worst configurations of randomized backtracking (that yield maximum numbers) still have better or the same performance as the default search order, but they have worse performance than other existing techniques, such as the random search order for Producer Consumer, Linked List and RAX Extended.

The numbers of states processed before an error is found also differ significantly among JPF runs with a single configuration for some benchmarks. Consider for example

the Replicated Workers benchmark and the configuration (50, Luby, $1-d/20$), in which case (i) the standard deviation of the number of processed states is approximately equal to the mean value and (ii) the maximum number of states processed by some JPF run with that configuration is 138 times bigger than the minimum number of processed states. Also the percentage of JPF runs that find an error varies to a great degree among different configurations. For some benchmark programs and configurations, a small percentage of JPF runs (or none at all) found an error (e.g. Linked List).

Answer to question Q3 (a generally good configuration). Although different configurations yield the best performance for each benchmark program, reasonably good performance for all of them is achieved by configurations with the ratio 0.9 and the random strategy. Table 4 shows for each benchmark program the results for: (1) the existing technique with the best performance, (2) the configuration of randomized backtracking with the best average performance, and (3) the configuration (H, random, 0.9) with the benchmark-specific threshold value H that achieves the best performance (for the ratio 0.9 and random strategy).

The configuration (H, random, 0.9) yields better or the same performance as the best existing technique for all benchmarks except Producer Consumer, for which the random search order has better performance. Note also that for some benchmarks, such as the Daisy file system and Elevator, the performance of randomized backtracking with the configuration (H, random, 0.9) is very close to the performance of the best configuration for the given benchmark.

The threshold value H must be selected with regard to the given benchmark program, since it influences (i) the chance that a JPF run will find an error and (ii) whether randomized backtracking will have any effect on the number of states traversed. If the threshold is too low, the error will not be found by most of the JPF runs. If the threshold is too high, randomized backtracking will never occur and therefore exhaustive state space traversal with the default search order will be performed by JPF. Table 5 illustrates how the threshold value H influences the performance of randomized backtracking with the configuration (H, random, 0.9) and the percentage of JPF runs that detect an error. For each benchmark and threshold value, the table provides values of the following metrics: the search depth at which the error was detected (mean μ and standard deviation σ) and the percentage of JPF runs that found some error.

Different values of the threshold parameter work for different benchmarks in general. However, the results imply the following three general properties of the performance of randomized backtracking based on the threshold value:

- very few JPF runs may find an error in a given program if the threshold value is too small (significantly smaller than the depth of the error state), because JPF may often backtrack too early before reaching the error state;
- even though the usage of randomized backtracking does not guarantee that an error is discovered, the results show that an error is discovered by all JPF runs for a given threshold (or by a very high percentage of JPF runs), as long as the threshold value is not too small (compared to the depth of the error state);
- if the threshold is too big, randomized backtracking does not have any effect (i.e., it does not have better performance than the existing techniques), because JPF finds

Configuration	Processed states				Error found
	μ	min	max	σ	
Daisy file system					
thread interleavings	139			0	100 %
(10, random, $1 - d/1000$)	108	102	116	5	100 %
(10, random, 0.9)	110	103	119	5.4	100 %
Elevator					
random search order	2399	1062	3833	787	100 %
(50, random, $1 - d/100$)	270	255	293	12	100 %
(50, random, 0.9)	290	261	312	15	100 %
Alarm Clock					
random search order	192	12	380	111	100 %
(5, fixed, $1 - d/20$)	37	15	93	22	91 %
(10, random, 0.9)	90	23	220	64	100 %
Linked List					
random search order	186	15	234	70	100 %
(10, Luby, 0.99)	58	50	71	7	59 %
(20, random, 0.9)	197	171	266	29	100 %
Producer Consumer					
random search order	48	25	73	13	100 %
(10, fixed, $1 - d/100$)	57	23	157	41	100 %
(10, random, 0.9)	189	22	435	133	100 %
RAX Extended					
thread interleavings	104			0	100 %
(5, Luby, 0.9)	8	8	9	0	59 %
(10, random, 0.9)	100	79	119	11	100 %
Replicated Workers					
context bound (10)	6585			0	100 %
(50, fixed, 0.9)	148	95	278	55	100 %
(100, random, 0.9)	522	263	905	203	100 %

Table 4. Configuration with good performance for all benchmarks

an error at a search depth lower than the threshold value and therefore it never backtracks from a state with unexplored outgoing transitions.

For example, the threshold value 5 is too small for Daisy file system and the threshold value 10 is too small for the Elevator benchmark. On the other hand, threshold values 50 or higher are too big for the RAX Extended benchmark.

5 Conclusion

We introduced the idea of using randomized backtracking in state space traversal for the purpose of fast error detection. Experiments with our implementation in JPF on several multi-threaded Java programs show that randomized backtracking has better performance than existing techniques in most cases.

Threshold:		5	10	20	50	100
Daisy file system						
States	μ	30	110	282	282	282
Depth	μ	6	7	7	7	7
	σ	0	0	0	0	0
Error found		1 %	100 %	100 %	100 %	100 %
Elevator						
States	μ	-	-	-	290	2185
Depth	μ	-	-	-	45	45
	σ	-	-	-	0	0
Error found		0 %	0 %	0 %	100 %	100 %
Alarm Clock						
States	μ	59	90	188	188	188
Depth	μ	13	13	13	13	13
	σ	1.7	4.5	0	0	0
Error found		25.6 %	100 %	100 %	100 %	100 %
Linked List						
States	μ	-	75	197	276	328
Depth	μ	-	12	19	48	48
	σ	-	1	1.4	0	0
Error found		0 %	40 %	100 %	100 %	100 %
Producer Consumer						
States	μ	127	189	204	9299	9299
Depth	μ	29	19	25	23	23
	σ	6.4	3.6	0	0	0
Error found		10.3 %	100 %	100 %	100 %	100 %
RAX Extended						
States	μ	12	100	441	1617	1617
Depth	μ	6	6	20	38	38
	σ	0	0	1	0	0
Error found		35.7 %	100 %	100 %	100 %	100 %
Replicated Workers						
States	μ	-	-	-	1774	522
Depth	μ	-	-	-	62	106
	σ	-	-	-	3.2	6.2
Error found		0 %	0 %	0 %	100 %	100 %

Table 5. Performance of the configuration (H, random, 0.9) for different threshold values

In particular, randomized backtracking has better performance than existing techniques in search for hard-to-find errors that are triggered only by a few paths (e.g., Elevator) and also in search for easy-to-find errors (e.g., Replicated Workers).

There is no single best configuration of randomized backtracking that would have the best performance for any benchmark program. However, we recommend to use the configuration (H, random, 0.9) with a specific threshold H, because it performs reasonably well for all benchmarks and, in particular, has significantly better performance

than existing techniques in most cases. Since the optimal threshold value is specific to the benchmark program, a viable approach is to run several instances of JPF with the configuration (H, random, 0.9) and different threshold values in parallel, and stop all of them when one finds an error. The "embarrassingly parallel" approach to search for errors, proposed in [8, 9], could be used.

In the future, we would like to evaluate randomized backtracking on more complex Java programs and to investigate possible approaches to determining reasonable threshold values (e.g., using heuristics). There might be some relation between good threshold values and bounds on the number of preemptions in context-bounded model checking. Another possible application of randomized backtracking is the search for errors in programs with infinite state spaces or infinite paths (e.g., programs that involve some ever increasing counter).

Acknowledgements. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. Concurrency Tool Comparison repository, `https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison`
2. Coons, K.E., Burckhardt, S., Musuvathi, M.: GAMBIT: Effective Unit Testing for Concurrency Libraries. In: PPoPP 2010, ACM.
3. Dwyer, M.B., Elbaum, S.G., Person, S., Purandare, R.: Parallel Randomized State-Space Search. In: ICSE 2007, IEEE CS.
4. Dwyer, M.B., Person, S., Elbaum, S.G.: Controlling Factors in Evaluating Path-Sensitive Error Detection Techniques. In: SIGSOFT FSE 2006, ACM.
5. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed Explicit-State Model Checking in the Validation of Communication Protocols. International Journal on Software Tools for Technology Transfer, 5(2-3), 2004.
6. Edelkamp, S., Schuppan, V., Bosnacki, D., Wijs, A., Fehnker, A., Aljazzar, H.: Survey on Directed Model Checking. In: 5th International Workshop on Model Checking and Artificial Intelligence, LNCS, vol. 5348, 2008.
7. Groce, A., Visser, W.: Heuristics for Model Checking Java Programs. International Journal on Software Tools for Technology Transfer, 6(4), 2004.
8. Holzmann, G.J., Joshi, R., Groce, A.: Tackling Large Verification Problems with the Swarm Tool. In: SPIN 2008, LNCS, vol. 5156.
9. Holzmann, G.J., Joshi, R., Groce, A.: Swarm Verification, In: ASE 2008, IEEE CS.
10. Java PathFinder, `http://babelfish.arc.nasa.gov/trac/jpf/`
11. Jones, M., Mercer, E.: Explicit State Model Checking with Hopper. In: SPIN 2004, LNCS, vol. 2989.
12. Luby, M., Sinclair, A., Zuckerman, D.: Optimal Speedup of Las Vegas Algorithms. Information Processing Letters, 47(4), 1993.
13. Musuvathi, M., Qadeer, S.: Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In: PLDI 2007, ACM.
14. Parallel Java Benchmarks, `http://code.google.com/p/pjbench`
15. Qadeer, S.: Daisy File System. Joint CAV/ISSTA special event on specification, verification and testing of concurrent software, 2004.

16. Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: TACAS 2005, LNCS, vol. 3440.
17. Rabinovitz, I., Grumberg, O.: Bounded Model Checking of Concurrent Programs. In: CAV 2005, LNCS, vol. 3576.
18. Rungta, N., Mercer, E.: Generating Counter-Examples Through Randomized Guided Search. In: SPIN 2007, LNCS, vol. 4595.
19. Rungta, N., Mercer, E.: Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs. In: PADTAD, ACM, 2009.
20. Sen, K.: Effective Random Testing of Concurrent Programs. In: ASE 2007, ACM.
21. Seppi, K., Jones, M., Lamborn, P.: Guided Model Checking with a Bayesian Meta-Heuristic. Fundamenta Informaticae, 70(1-2), 2006.
22. Wehrle, M., Kupferschmid, S.: Context-Enhanced Directed Model Checking. In: SPIN 2010, LNCS, vol. 6349.