Department of Distributed and Dependable Systems
Technical report no. D3S-TR-2015-03
December 1, 2015

Hybrid Analysis of Future Accesses and Heuristics for Fast Detection of Concurrency Errors

Pavel Parízek

Abstract: Systematic state space traversal is a very popular approach for detecting errors in multithreaded programs. Nevertheless, it is very expensive because any non-trivial program exhibits a huge number of possible interleavings, and therefore some combination of guided search and bounded search is often used to achieve good performance. We present two heuristics that are based on a hybrid static-dynamic analysis that can identify possible accesses to shared objects. One heuristic changes the order in which transitions are explored, and the second heuristic prunes selected transitions. Results of experiments on several Java programs, which we performed using our prototype implementation in Java Pathfinder, show that the hybrid analysis together with heuristics improves the performance of error detection quite significantly.

This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S.

1 Introduction

Systematic traversal of the program state space is a popular approach for detecting errors in systems with multiple concurrently-running threads. It is used, for example, in software model checking [26] and also in concurrency testing [1], where the main goal is to check the program behavior under all possible thread interleavings. Efficient detection of concurrency errors, such as deadlocks and atomicity violations, has become very important especially with the greater proliferation of multithreaded software that exploits widely available multi-core processors. However, full systematic traversal is very expensive both in terms of time and memory, because any non-trivial multithreaded program exhibits a huge number of possible interleavings. Good performance in practice can be achieved through the use of heuristics and bounded search. Many techniques and optimizations have been already proposed, including directed search [6, 10], randomization [18, 23], parallel state space traversal [4, 11], bounding the number of thread preemptions [16, 22], and concolic testing for concurrent systems [7].

Parízek and Lhoták designed a hybrid analysis that identifies possible accesses to shared objects and used it to optimize partial order reduction in the context of exhaustive verification [19, 20]. More specifically, redundant thread scheduling choices in the state space were eliminated based on results of the hybrid analysis. The analysis combines static analysis with dynamic analysis, symbolic interpretation of program statements, and usage of information from dynamic program states on-the-fly during the state space traversal. Here we apply the hybrid analysis in a different context — to accelerate detection of concurrency errors. We present two new heuristics that guide the search based on the hybrid analysis. The first heuristic uses analysis results to change the order in which individual transitions are explored during the state space traversal, and the second heuristic prunes some transitions in each state that corresponds to a non-deterministic thread scheduling choice. Both heuristics use the analysis results on-the-fly together with the knowledge of dynamic execution history (i.e., the currently processed state space path) in order to discover errors faster.

We implemented the heuristics in Java Pathfinder (JPF) [12], and evaluated the hybrid analysis together with heuristics on several multithreaded Java programs. Results of our experiments show that (i) the hybrid analysis alone can reduce the time needed to find errors quite significantly and (ii) the proposed heuristics improve the performance even further in some configurations. A practical benefit of the proposed approach is more efficient detection of concurrency-related errors that involve objects fields and array elements. The hybrid analysis and heuristics enable JPF to find errors faster than with the existing techniques based on state space traversal. We observed big performance improvements especially for the more complex benchmarks from our set, which have large state spaces.

The rest of this paper is structured as follows. In the next section we describe important background concepts, including the hybrid analysis, and we also provide a more detailed overview of related work. Then we present our main contribution: two heuristics for reordering and pruning transitions (Section 3), and experimental evaluation of the error detection performance. We provide the empirical data in Section 4 and discuss general results in Section 5.

2 Background and Related Work

In the first part of this section, we describe the basic procedure for explicit state space traversal. Then we discuss existing work on heuristics and optimizations, which have the common goal of detecting errors as fast as possible. We also provide an overview of the hybrid analysis at the end.

2.1 State Space Traversal

Figure 1 shows the basic algorithm for depth-first traversal of a program state space. We assume that the state space is constructed on-the-fly during traversal and that program statements are interpreted using dynamic concrete execution. The symbol s represents a program state and the symbol tr represents a transition. Each state is a snapshot of all variables and threads at some point during the execution of a program under some thread interleaving. A transition is a sequence of executed instructions, which is associated with a specific thread and bounded by non-deterministic scheduling choices.

Thread scheduling choices are typically created only at instructions that access the global state visible by multiple concurrent threads. The main goal is to avoid redundant exploration of thread interleavings, and in this way to mitigate state explosion. Some technique of partial order reduction (POR) [9] is

```
visited = {}
path = [ ]
explore(s_0)

procedure explore(s)
  if error(s) then terminate
  for tr ∈ order(filter(enabled(s))) do
    s' = execute(s, tr)
    if s' ∉ visited then
      visited = visited ∪ s'
      push(path, ⟨s, tr, s'⟩)
      explore(s')
      pop(path)
    end if
  end for
end proc
```

Figure 1: *Algorithm for depth-first traversal of a state space*

used to determine the set of globally-visible instructions. A lot of work has been done on various approaches to partial order reduction, including POR based on heap reachability [5] and dynamic POR [8], but details are out of the scope of this paper.

The state space traversal procedure maintains the set of states that have been already visited (for the purpose of state matching) and also the current path in the form of a stack of states and transitions. Program states need to be explicitly saved only at transition boundaries.

The function enabled returns a set of transitions that are enabled in a given state and must be explored. Each thread that is runnable in state s is associated with one transition in the set $\mathrm{enabled}(s)$. If there are multiple threads runnable in s, then the state space traversal procedure has to make a nondeterministic choice among all of them. We say that all threads runnable in state s are enabled in the choice ch that is associated with s. The function filter can be used to prune some transitions leading from s, and the function order determines the sequence in which the transitions are explored (i.e., the search order). Most heuristics and optimizations described in the next subsection are based on custom implementations of these two functions.

Many popular tools, including Java Pathfinder [12], implement the approach described here.

2.2 Fast Detection of Concurrency Errors

A very popular approach to fast error detection via state space traversal is guided search [6]. The basic idea is to navigate the search towards error states with the help of various heuristics [10], so that fragments of the state space more likely to contain errors are explored first during the traversal. In each state, transitions are explored in the order determined by some heuristic function.

Closely related to the topic of this paper are the concepts of *useless transitions* and *interference contexts* proposed by Wehrle et al. [28, 29]. Authors introduce a heuristic function that gives preference to transitions that interfere with some of the previous transitions on the current state space path. Two transitions interfere if they access the same variables and one of the accesses is a write. However, the heuristic was applied only to checking models of finite software systems defined as multiple automata, and the respective publications do not discuss approaches to obtain the information about which transitions may interfere.

Kim et al. [15] proposed an approach for detecting race conditions that is also based on heuristics and information about interfering accesses to shared variables. In this case, the heuristic functions consider only the execution trace prefix from the initial state to the current dynamic state, and completely neglect possible future behavior.

Randomized search is also quite effective according to recent studies [18, 23], especially in combination with parallel traversal of different state space fragments [4, 11]. The results of a random number choice can be used, for example, to determine the search order over individual transitions and complete execution paths [3], to identify state space fragments that will be pruned [18], and it can be also combined with the guided search [23].

Many techniques that improve the performance of systematic concurrency testing and model check-

ing use some kind of bounded search. This includes, for example, bounding the number of thread preemptions on each state space path [16, 22] and bounding the depth of state space traversal [25]. A set returned by the customized function $\mathrm{enabled}$ does not contain any transition that would exceed the given bound. The motivation behind all these approaches is that many errors can be found with very few thread preemptions [16] and in small depths. Therefore, the search for errors is restricted to a corresponding region of the program state space, and other fragments are pruned because exhaustive traversal is not tractable due to state explosion. A recent experimental study by Thomson et al. [24] provides a comprehensive overview of techniques in this category and their comparison.

2.3 Hybrid Analysis

The hybrid analysis identifies object fields and array elements that may be accessed by multiple threads during execution of the program from a particular state. It provides over-approximate description of the possible future behavior of program threads, and therefore complements information that can be retrieved from the current dynamic state and the current state space path (execution history). The analysis was defined just for accesses to fields of heap objects in the original publication [19], and only recently we extended it to support also accesses to individual elements of shared array objects [21]. Here we provide a high-level overview of the main aspects of its design.

An input to the hybrid analysis is a program that contains multiple concurrent threads. For each program point p in each thread T, the analysis computes the set of fields and array elements possibly accessed by thread T after the point p on any execution path. It has two phases: static and dynamic.

The static phase gives only partial results that cover the behavior of a thread T only between the point p and return from the method containing p (including nested method calls transitively). For this purpose, we use a backward flow-sensitive context-insensitive static data flow analysis that is performed over the full inter-procedural control flow graph of each thread. Exhaustive flow-insensitive and context-insensitive pointer analysis identifies abstract heap objects and possibly aliased variables.

Complete results of the hybrid analysis are computed on demand at dynamic analysis time, i.e. on-the-fly during the state space traversal. The analysis uses information taken from the dynamic program states, in particular the call stacks of all threads and values of specific variables. Let pc_i be the current program counter of thread T_i (in the top stack frame). In order to get the full result for pc_i, it is sufficient to take the current locations in all the stack frames of T_i and merge partial results of the static phase for the corresponding program points.

Considering the dynamic state s, where $pc_1, \ldots, pc_n$ are the current locations of threads $T_1, \ldots, T_n$, respectively, results for the program points $pc_1, \ldots, pc_n$ capture all possible future accesses that may happen during the program execution from the state s. Results of the hybrid analysis are fully context-sensitive, and therefore very precise, because they reflect the dynamic calling context of each pc_i. On the other hand, the results are always valid only for the given (current) dynamic program state. Note also that, in practice, the analysis considers read and write accesses separately in order to enable detection of read-write conflicts between different threads.

3 Heuristics

The main idea behind our approach is to use the results of hybrid analysis in three ways: (1) to eliminate redundant non-deterministic thread choices during the state space traversal, (2) to determine the order in which individual transitions from a given state are explored, and (3) to prune transitions that likely do not lead to an error state. A method for eliminating redundant thread choices has been already proposed in [19], so here in this paper we focus on the second and third item in the list. More specifically, we propose one heuristic for reordering transitions (Section 3.1) and one heuristic that identifies transitions to be pruned (Section 3.2). Both heuristics are encoded in custom implementations of the functions order and filter (Figure 1).

When designing the heuristics, we have built upon the concepts of useless transitions and interference contexts proposed by Wehrle et al. [28, 29]. Our procedure for identification of useless transitions and possibly interfering transitions combines the hybrid analysis with knowledge of the current dynamic state and execution history. Contrary to the prior work by Wehrle et al., our procedure is applicable to programs written in mainstream object-oriented languages (such as Java), and it can be used when the program state space is created on-the-fly (i.e., when it is not available up front). The heuristics

drive the state space traversal using this procedure. We also introduce several parameters that influence their behavior.

3.1 Reordering Transitions

This heuristic changes the order of transitions at each non-deterministic thread choice based on two pieces of information: (1) a list of accesses to fields and array elements that were performed on the current state space path from the initial state to the given choice, and (2) the results of the hybrid analysis that specify accesses that may occur during the rest of the program execution. In other words, the heuristic combines knowledge of the past accesses with information about possible future events. Our main rationale is to prioritize threads that are more likely to trigger errors caused by race conditions over fields and array elements.

Upon reaching a state that is associated with a non-deterministic thread choice, the state space traversal procedure augmented with this heuristic identifies threads that may in the future perform actions possibly interfering with some of the past accesses. The procedure maintains an up-to-date list of fields and array elements that were accessed on the current state space path. Let s be the currently processed state at some point during the traversal, ch be the thread choice associated with s, and L be the list of past accesses on the current path up to s. For each thread T_i runnable in s, the procedure queries the results of the hybrid analysis for the current program counter pc_i of T_i in order to retrieve a set F_i of possible future accesses, and then it computes the intersection of L and F_i. The set of *interfering threads* contains every T_i for which the intersection is not empty. All outgoing transitions enabled in state s, which are associated with interfering threads, are then moved to the front of the list (order) at the choice ch, and therefore future actions of threads possibly interfering with some past actions are explored in precedence during the state space traversal. Note, however, that we do not enforce any particular order of exploration within the group of interfering threads, and we also preserve the default order for the other threads (transitions).

We have also designed this heuristics as parameterized and configurable by the user. One parameter is the percentage of the length of the current path that is considered for identification of interfering threads. It determines the fragment of the current path from which the past accesses are collected into the set L. A value lower than 100 means that the heuristic takes into account only a subset of past accesses, and purposedly ignores accesses that were performed close to the beginning of the current path. Our motivation behind this parameter is that, in general, usage of a small value might enable faster detection of errors in cases when the possibly racy accesses from concurrent threads are performed close to each other on an execution path. Note that usage of values less than 100 is sound because all transitions and program behaviors are explored eventually — this heuristic influences only the order of exploration of individual transitions.

Another parameter is a flag that says whether the heuristic has to distinguish read and write accesses. If this feature is disabled then all accesses to a particular field or array element are considered as potentially interfering, while in the other case only the read-write pairs are marked as such. Distinguishing between read and write accesses makes the heuristic more precise, but possibly more expensive in terms of running time.

A configuration of the reordering heuristic is a pair (RW, P), where the variable RW represents the boolean flag (values on and off) that says whether read and write accesses should be distinguished, and the value of P is the percentage of the current path that is analyzed in order to collect past accesses into the set L at each choice (state). We use this notation for configurations in the rest of the paper.

3.2 Pruning Transitions

The second heuristic works in a similar way to the first one, and it has exactly the same parameters. Just instead of reordering, at each state it prunes all enabled transitions that are *not* associated with interfering threads. Our rationale behind this heuristic is to neglect threads that are not likely to trigger errors caused by race conditions at fields and array elements.

An exception to the general rule applies in the case of states with only a single enabled transition (i.e., with a single runnable thread), where the respective transition is not pruned. Similarly, if all transitions enabled in a given state would be pruned then one of them is preserved. We designed these two exceptions in order to ensure that at least some execution paths (thread interleavings) are fully explored. However, we would also like to emphasize that pruning a transition associated with a thread T at a particular state s does not mean that future actions of T are never explored. There must exist an

execution path starting in the state s such that (i) either T will belong to the set of interfering threads at some point on the path or (ii) it will be a single runnable thread at some point — unless there is a deadlock that would be reported anyway.

Still, this heuristic is not sound. It may prune transitions that are important, because they represent actions that are not considered by the hybrid analysis (e.g., starting of a new thread or releasing blocked threads) but could trigger possibly errorneous thread interleavings anyway. The state space traversal procedure may miss some errors, when this heuristic is used.

3.3 Implementation

We implemented the hybrid analysis and proposed heuristics using Java Pathfinder (JPF) [12], which is a framework for state space traversal of multithreaded Java programs, and the WALA library for static analysis [27]. In order to support reordering and pruning of enabled transitions at thread choices, we created several custom modules for JPF: listeners that record every access performed by the program during its execution, a custom scheduler factory that produces a list of transitions for each thread choice, and a non-standard interpreter of Java bytecode instructions for accesses to object fields and array elements. JPF API operations were used to retrieve information about the dynamic program state.

During our work on the extension of the hybrid analysis towards array elements, we also implemented several optimizations that improve the analysis precision in practice. For example, the static inter-procedural control flow graph (ICFG) of a given program contains edges that, for every call of Thread.start(), connect the call with the run method of every thread class defined in the program source code — and because of that the static analysis is not very precise. The hybrid analysis with our optimizations takes into account the Java class (type) of the dynamic receiver object when it processes a call of Thread.start(), and in this way avoids some infeasible executions paths that are modeled by the ICFG.

The complete implementation, together with the experimental setup and benchmark programs described in the next section, is publicly available at `http://d3s.mff.cuni.cz/projects/formal_methods/jpf-static/musepat16.html`.

4 Experiments

The main goal of our experimental evaluation was to find how much the hybrid analysis and proposed heuristics improve the error detection performance of existing techniques based on state space traversal. In addition, we wanted to check whether our results confirm the benefits of similar heuristics designed by Wehrle et al. [28, 29].

We measured the performance of hybrid analysis and both heuristics on 9 multithreaded Java programs: the Daisy file system, the Elevator benchmark from the pjbench suite [17], a plain Java version of the jPapaBench benchmark [13], a plain Java version of the CDx benchmark [14], four benchmarks from the CTC repository [2] (Alarm Clock, Linked List, Producer-Consumer, and Replicated Workers), and a Java version of QSortMT from the Inspect benchmark suite [30]. jPapabench, with 4500 lines of code and 7 concurrently running threads, is the most complex benchmark that we use. Some of the benchmark programs already contained errors in the form of possible atomicity violations that are caused by incorrect or missing synchronization of accesses to fields and array elements from multiple threads. For the purpose of experiments, we manually injected similar errors into the other benchmarks. Note that JPF can detect race conditions that trigger such atomicity violations.

Here we provide tables with data for all experiments that we performed. We discuss the main results in the next section.

Table 1 shows the effects of hybrid analysis on the error detection performance. It contains data for these configurations of JPF:

- partial order reduction (POR) based on heap reachability [5] with the default search order in JPF (table column with the label "heap reach POR"),
- POR based on heap reachability with the hybrid analysis that is used only to eliminate redundant thread choices (column "HR + hybrid"),
- our implementation of the dynamic POR algorithm by Flanagan and Godefroid [8] combined with state matching (column "dynamic POR"),

benchmark	heap reach POR states	heap reach POR time	HR + hybrid states	HR + hybrid time	dynamic POR states	dynamic POR time	DPOR + hybrid states	DPOR + hybrid time
Daisy	493645	139 s	297523	95 s	-	-	266291	91 s
Elevator	61465	14 s	16574	7 s	1671602	511 s	485070	143 s
jPapaBench	457139	144 s	94567	41 s	-	-	-	-
CDx	383568	2870 s	48069	456 s	-	-	-	-
Alarm Clock	950	1 s	313	3 s	786	1 s	165	3 s
Linked List	2119	1 s	345	3 s	970	1 s	830	3 s
Prod-Cons	35351	16 s	7535	6 s	5596	3 s	4005	4 s
Rep Workers	12951140	6113 s	441253	178 s	14303	5 s	3909	4 s
QSortMT	4883	2 s	2564	2 s	-	-	-	-

benchmark	random search time	random + hybrid time
Daisy	86 ± 57 s	59 ± 38 s
Elevator	4 ± 4 s	3 ± 2 s
jPapaBench	1 ± 0 s	3 ± 0 s
CDx	162 ± 115 s	63 ± 46 s
Alarm Clock	1 ± 0 s	3 ± 0 s
Linked List	1 ± 0 s	3 ± 0 s
Prod-Cons	1 ± 0 s	3 ± 0 s
Rep Workers	2761 ± 2996 s	3 ± 0 s
QSortMT	1 ± 0 s	2 ± 0 s

Table 1: *Experiments: performance improvement over existing techniques*

- dynamic POR with state matching and hybrid analysis that is used only to eliminate redundant thread choices (column "DPOR + hybrid"),
- random search order with POR based on heap reachability (column "random search"), and
- random search order combined with the hybrid analysis (the column labeled "random + hybrid").

Note that Table 1 has two vertically-placed parts in order to accommodate all the configurations. We report the total number of states processed during the traversal before JPF detects an error, and the total running time of JPF (including all phases of the hybrid analysis, where applicable). The number of states is equivalent to the number of thread choices, because JPF explicitly saves program states only at transition boundaries. In the case of random search, we run each experiment 10 times, and report values in the form $A \pm D$, where A stands for the average and D is the standard deviation. Note that we do not report the number of states for random search in order to save space. We set the time limit of 2 hours for all configurations. The symbol "-" in a table cell represents the situation where JPF run out of the time limit or did not find an error for a given configuration and a benchmark program.

Tables 2 and 3 provide data for selected configurations of both heuristics, when used together with the POR based on heap reachability and with the hybrid analysis. Results for the hybrid analysis alone represent the baseline in this case. For the configuration variable P, which determines the fragment (percentage) of the current path that is analyzed for the purpose of collecting past accesses, we picked the values {10, 25, 50, 75, 90, 100} in order to cover the whole interval evenly. Data for the whole path (i.e., the value $P = 100$) are always shown in the leftmost respective column. We combine every supported value of the variable P with both values of the configuration variable RW. To avoid very large tables, we report the total running time only for configurations that distinguish between read and write accesses ("RW: on"), as it is sufficient to show the general dependency of the running time on the value of the variable P. Trends are very similar for the other configurations.

For the benchmarks Daisy and CDx, in which case the standalone random search needs a relatively long time to find errors, we also combined random search with the hybrid analysis and selected configurations of our heuristics. Table 4 contains results of these experiments. We performed each of the experiments 10 times, and in each iteration we run 4 parallel instances of JPF. Our motivation for the usage of parallelism was to exploit multiple cores also to achieve faster detection of concurrency errors in the case of Daisy and CDx. We report the minimal observed time needed to reach an error state, the maximum observed time, average over all iterations, and standard deviation. The second column (with

benchmark	HR + hybrid	HR + hybrid + reordering heuristic						
			P:	100 %	10 %	25 %	50 %	75 %	90 %
Daisy	states: 297523 time: 95 s	RW: on	states	297523	297523	297523	297523	297523	297523
			time	175 s	124 s	129 s	136 s	148 s	150 s
		RW: off	states	297523	297523	297523	297523	297523	297523
Elevator	states: 16574 time: 7 s	RW: on	states	16574	16574	439	439	439	439
			time	8 s	8 s	3 s	3 s	3 s	3 s
		RW: off	states	16574	16574	16574	16574	16574	16574
jPapaBench	states: 94567 time: 41 s	RW: on	states	94567	94567	94567	94567	94567	94567
			time	88 s	53 s	57 s	66 s	75 s	80 s
		RW: off	states	94567	94567	94567	94567	94567	94567
CDx	states: 48069 time: 456 s	RW: on	states	48069	29531	48069	48069	48069	48069
			time	580 s	329 s	553 s	554 s	573 s	562 s
		RW: off	states	48069	48069	48069	48069	48069	48069
Alarm Clock	states: 313 time: 3 s	RW: on	states	313	313	313	149	10	10
			time	3 s	3 s	3 s	3 s	3 s	3 s
		RW: off	states	313	313	313	10	10	10
Linked List	states: 345 time: 3 s	RW: on	states	345	345	345	345	345	345
			time	4 s	4 s	4 s	4 s	4 s	4 s
		RW: off	states	345	345	345	345	345	345
Prod-Cons	states: 7535 time: 6 s	RW: on	states	7535	5084	7538	7538	7538	7535
			time	7 s	6 s	7 s	7 s	7 s	7 s
		RW: off	states	7535	7535	7535	7535	7535	7535
Rep Workers	states: 441253 time: 178 s	RW: on	states	441253	114	441253	441253	441253	441253
			time	238 s	3 s	209 s	222 s	226 s	226 s
		RW: off	states	441253	441253	441253	441253	441253	441253
QSortMT	states: 2564 time: 2 s	RW: on	states	2564	2455	101	2565	2564	2564
			time	4 s	4 s	3 s	4 s	4 s	4 s
		RW: off	states	2564	2564	2564	2564	2564	2564

Table 2: *Experiments: different configurations of the reordering heuristic*

benchmark	HR + hybrid	HR + hybrid + pruning heuristic						
			P:	100 %	10 %	25 %	50 %	75 %	90 %
Daisy	states: 297523 time: 95 s	RW: on	states	297523	-	296782	296789	296789	296789
			time	170 s	-	125 s	139 s	142 s	151 s
		RW: off	states	297523	-	296789	296789	296789	296789
Elevator	states: 16574 time: 7 s	RW: on	states	16574	16574	-	-	-	-
			time	8 s	8 s	-	-	-	-
		RW: off	states	16574	16574	16574	16574	16574	16574
jPapaBench	states: 94567 time: 41 s	RW: on	states	94567	94567	94567	94567	94567	94567
			time	88 s	53 s	58 s	66 s	75 s	81 s
		RW: off	states	94567	94567	94567	94567	94567	94567
CDx	states: 48069 time: 456 s	RW: on	states	48069	26183	39942	39942	39942	39942
			time	606 s	290 s	457 s	464 s	466 s	479 s
		RW: off	states	48069	41517	47892	48069	48069	48069
Alarm Clock	states: 313 time: 3 s	RW: on	states	313	274	285	149	10	10
			time	4 s	4 s	4 s	4 s	4 s	4 s
		RW: off	states	313	307	313	10	10	10
Linked List	states: 345 time: 3 s	RW: on	states	345	337	345	345	345	345
			time	4 s	4 s	4 s	4 s	4 s	4 s
		RW: off	states	345	345	345	345	345	345
Prod-Cons	states: 7535 time: 6 s	RW: on	states	7535	5084	7538	7538	7538	7535
			time	7 s	6 s	7 s	7 s	7 s	7 s
		RW: off	states	7535	7535	7535	7535	7535	7535
Rep Workers	states: 441253 time: 178 s	RW: on	states	441253	114	441253	441253	441253	441253
			time	235 s	4 s	212 s	229 s	236 s	230 s
		RW: off	states	441253	441253	441253	441253	441253	441253
QSortMT	states: 2564 time: 2 s	RW: on	states	2564	2450	101	2560	2559	2559
			time	4 s	4 s	3 s	4 s	4 s	4 s
		RW: off	states	2564	2564	2564	2564	2564	2564

Table 3: *Experiments: different configurations of the pruning heuristic*

benchmark	hybrid		reordering (RW: on)			pruning (RW: on)		
		P:	10%	25%	50%	10%	25%	50%
Daisy	59 ± 38 s	min	6 s	7 s	7 s	-	7 s	7 s
		max	262 s	262 s	282 s	-	128 s	128 s
		avg	86 s	66 s	69 s	-	81 s	57 s
		dev	70 s	73 s	71 s	-	52 s	50 s
CDx	63 ± 46 s	min	44 s	45 s	44 s	46 s	48 s	46 s
		max	182 s	195 s	208 s	189 s	198 s	188 s
		avg	93 s	102 s	89 s	97 s	93 s	85 s
		dev	38 s	44 s	41 s	41 s	42 s	40 s

Table 4: *Experiments: parallel random search with heuristics*

the label "hybrid") contains data for random search just with the hybrid analysis — they represent the baseline for this set of experiments.

Finally, Table 5 contains additional data for those benchmarks that are parameterizable by the maximal number of concurrently-running threads. We picked only some configurations of JPF with heuristics, and a specific range of the number of concurrent threads for each benchmark. The third group of columns, with the label "best heuristic", shows data for the configuration of proposed heuristics that achieves the best result for the given benchmark program. The lowest number of threads considered for a particular benchmark is the default one, which was used for experiments reported in the other tables.

5 Discussion

The results of experiments show how much the performance of existing approaches is improved by usage of the hybrid analysis and heuristics during the state space traversal. Our main observations, based on data in tables 1-3, are provided in the following list.

- Compared to all the existing approaches that we considered in our evaluation, the corresponding configurations with hybrid analysis and proposed heuristics need to explore much less states to find an error in the case of 8 benchmark programs (out of 9), and they reduce the overall running time by a great margin for 5 benchmarks.
- For the remaining benchmarks, our approach still has a good performance that is comparable with the existing techniques in terms of running time.
- Just the hybrid analysis, when it is used to eliminate redundant thread choices, improves the speed of error detection by up to 35 times (for Rep Workers) over JPF with POR based on heap reachability, and by a factor of 3.6 (for Elevator) with respect to dynamic POR.
- The proposed heuristics achieve even better performance in the case of specific configurations and benchmark programs; for example, improving the speed with respect to POR based on heap reachability by the factor of 1400 for Rep Workers and by the factor of 21 for CDx.

A practical benefit of the hybrid analysis and heuristics is apparent especially in case of the more complex benchmarks from our set, such as CDx, Daisy, jPapaBench, and Rep Workers, which have large state spaces and for which the time needed to find errors was reduced by the biggest factor.

Surprisingly, dynamic POR is much slower than POR based on heap reachability for Elevator (with or without the hybrid analysis and our heuristics), and for 3 benchmarks it even did not find an error before the time limit.

The performance of random search is improved quite significantly by usage of the hybrid analysis in the case of 4 benchmark programs — Daisy, Elevator, CDx, and Rep Workers. For the other 5 benchmarks, results with and without the hybrid analysis are comparable, and the running times are very low in general (just few seconds).

In the rest of this section, we discuss results for specific configurations of both heuristics and for individual benchmark programs, and we also highlight some additional conclusions.

Results that we obtained for the heuristic based on reordering transitions (Table 2) are not very positive compared to the baseline, i.e. the heuristic does not help to reach error states faster in many cases, and therefore it seems that its usage does not have benefits in the case of programs written in

benchmark - #threads	HR + hybrid		best heuristic		reordering (RW: on, P=100%)		pruning (RW: on, P=100%)	
	states	time	states	time	states	time	states	time
Daisy - 2	297523	95 s	297523	124 s	297523	175 s	297523	170 s
Daisy - 3	-	-	-	-	-	-	-	-
Alarm Clock - 3	313	3 s	10	3 s	313	3 s	313	4 s
Alarm Clock - 4	2363	3 s	12	3 s	2363	4 s	2363	4 s
Alarm Clock - 5	17667	8 s	14	3 s	17667	10 s	17667	10 s
Linked List - 2	345	3 s	337	4 s	345	4 s	345	4 s
Linked List - 3	460	3 s	460	4 s	460	4 s	460	4 s
Linked List - 4	481	3 s	481	4 s	481	4 s	481	4 s
Prod-Cons - 7	7535	6 s	5084	6 s	7535	7 s	7535	7 s
Prod-Cons - 9	41142	22 s	41145	23 s	41142	25 s	41142	25 s
Prod-Cons - 12	39130	25 s	39133	27 s	39130	32 s	39130	31 s
Rep Workers - 5	441253	178 s	114	3 s	441253	238 s	441253	235 s
Rep Workers - 6	2114522	983 s	129	3 s	2114522	1248 s	2114522	1285 s
Rep Workers - 8	-	-	162	3 s	-	-	-	-

Table 5: *Experiments: scalability with respect to the maximal number of concurrent threads*

mainstream languages, although the results published in [29] are quite good. The reordering heuristic achieved better performance in the case of four benchmarks (CDx, Elevator, QSortMT, Rep Workers), but only using specific configurations, where the variable RW has the value on and P is either 10 % or 25 %. In general, the reordering heuristic does not improve performance especially for benchmark programs in which (all) threads have the same code, or when multiple threads access the same field in a racy manner, because then either all threads or none of them are interfering and the order of exploration of individual transitions from a thread choice is not actually changed. Such results can be observed for the benchmark Prod-Cons. Reordering is not very useful also when all interfering threads correspond to transitions that are already at the beginning of the default (original) search order. This happens, for example, in the case of benchmarks Daisy and jPapaBench. When inspecting the results for Linked List, we found that transitions moved back in the order very frequently correspond to threads waiting, e.g., in the body of Thread.join, and then reordering has no practical effect. There may be other situations where the reordering heuristic does not improve performance, but those described above are most prevalent in the set of benchmark programs that we used.

On the other hand, results for the heuristic based on pruning transitions (Table 3) show that it improves the error detection performance quite significantly in some configurations and for some benchmarks, but it may also miss some errors. Good performance is achieved, in general, for configurations where only 10 % of the current path is analyzed in every state by the procedure that determines possibly interfering threads. More specifically, data in Table 3 show that the configuration where P = 10 % yields the best performance for 4 benchmarks (CDx, Linked List, Prod-Cons, and Rep Workers). The configuration with P equal to 25 % achieves the best performance for QSortMT.

A notable exception to the patterns described above in the two previous paragraphs are the results for Alarm Clock. In that case, the best performance was achieved by configurations where the value of P is in the range 50 % – 90 %, and there is a large variability between results for individual configurations.

Usage of heuristics together with the random search and hybrid analysis does not improve the performance on average, according to running times reported in Table 4. In fact, heuristics slightly increase the running time for most configurations that we considered.

Data in Table 5 show that, for the purpose of detecting errors, JPF with hybrid analysis and heuristics scales quite well to programs with many concurrently running threads. The increase of the total running time as a function of the number of threads is low for Alarm Clock, Linked List, and Prod-Cons. JPF runs out of the time limit in all configurations only for Daisy with 3 threads.

The cost of the hybrid analysis is very low in general. However, it may be responsible for a slight increase of the total running time when JPF has to explore only a small number of states before detecting an error — see, for example, the data for benchmarks Alarm Clock and Linked List in Table 1. The cost of both heuristics in terms of the running time depends especially on the value of P, i.e. on the fragment of the current path that is analyzed in order to retrieve past accesses. Results of our experiments show that it is more efficient to use small values of P (e.g., 10 % and 25 %), because then JPF spends less time

processing the current path at each state.

We have also made some observations based on manual inspection of the execution logs, which we discuss in the next paragraphs.

For benchmark programs such as jPapaBench, where specific fields and array elements are accessed very often throughout the program execution, the same set of interfering threads would be typically identified at a given state for any value of P — independently of whether past accesses are collected just from last 10 % of the current path or from the whole path. All configurations that differ only in the value of P would yield the same numbers of states explored before reaching an error, but different running times. In such cases, better performance can be in general achieved with rather small values of P (10 % or 25 %).

Behavior of the heuristics also really depends on whether read and write accesses are distinguished or not. We found that, in the latter case when the configuration variable RW has the value off, more threads are typically identified as possibly interfering, and therefore more transitions are reordered or pruned at individual states when JPF uses the respective configurations.

Majority of the benchmark programs have the property that all configurations of JPF and heuristics, when applied to a program from this group, find the same error state (location). However, there are few exceptions. For example, JPF with the dynamic POR finds a different error state in Prod-Cons than all the configurations involving POR based on heap reachability, and JPF with the hybrid analysis finds a different error state in Daisy than plain JPF.

To summarize, the hybrid analysis and selected configurations of heuristics improve the error detection performance of existing approaches (POR based on heap reachability, dynamic POR, and random search) to a great extent. We recommend to use the following two configurations based on our results:

1. JPF with random search and the hybrid analysis used only to eliminate redundant thread choices (i.e., without any of the heuristics), and

2. hybrid analysis together with the heuristic based on pruning in the configuration (RW: on, P: 10 %), where read and write accesses are distinguished and only 10 % of the current path will be analyzed in each state.

Both these configurations should be run in parallel to minimize the time needed to find errors.

We would also like to emphasize that, although we implemented and evaluated the proposed approach only in the context of JPF and Java programs, we believe that similar results could be achieved also for multithreaded programs written in other mainstream object-oriented languages, such as C++, C#, and Scala.

References

[1] T. Ball, S. Burckhardt, P. de Halleux, M. Musuvathi, and S. Qadeer. Predictable and Progressive Testing of Multithreaded Code. IEEE Software, 28(3), 2011.

[2] Concurrency Tool Comparison (CTC) repository,

[3] K.E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: Effective Unit Testing for Concurrency Libraries. Proceedings of PPoPP 2010, ACM.

[4] M.B. Dwyer, S.G. Elbaum, S. Person, and R. Purandare. Parallel Randomized State-Space Search. Proceedings of ICSE 2007, IEEE.

[5] 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, 2004.

[6] S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker, and H. Aljazzar. Survey on Directed Model Checking. Proceedings of MoChArt 2008, LNCS, vol. 5348.

[7] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic Testing. Proceedings of ESEC/FSE 2013, ACM.

[8] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. Proceedings of POPL 2005, ACM.

[9] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, 1996.

[10] A. Groce and W. Visser. Heuristics for Model Checking Java Programs. International Journal on Software Tools for Technology Transfer, 6(4), 2004.

[11] G. Holzmann, R. Joshi, and A. Groce. Swarm Verification. Proceedings of ASE 2008, IEEE.

[12] Java Pathfinder: verification platform for Java programs,

[13] jPapaBench: A Java version of the PapaBench benchmark,

[14] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CDx: A Family of Real-Time Java Benchmarks. Proceedings of JTRES 2009, ACM.

[15] K. Kim, T. Yavuz-Kahveci, and B.A. Sanders. Precise Data Race Detection in a Relaxed Memory Model Using Heuristic-Based Model Checking. Proceedings of ASE 2009, IEEE.

[16] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. Proceedings of PLDI 2007, ACM.

[17] Parallel Java Benchmarks (pjbench suite),

[18] P. Parízek and O. Lhoták. Randomized Backtracking in State Space Traversal. Proc. of SPIN 2011, LNCS, vol. 6823.

[19] P. Parízek and O. Lhoták. Identifying Future Field Accesses in Exhaustive State Space Traversal. Proceedings of ASE 2011, IEEE.

[20] P. Parízek and O. Lhoták. Model Checking of Concurrent Programs with Static Analysis of Field Accesses. Science of Computer Programming, 98, 2015.

[21] P. Parízek. Hybrid Analysis for Partial Order Reduction of Programs with Arrays. Proceedings of VMCAI 2016, to appear.

[22] S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. Proceedings of TACAS 2005, LNCS, vol. 3440.

[23] N. Rungta and E. Mercer. Generating Counter-Examples Through Randomized Guided Search. Proceedings of SPIN 2007, LNCS, vol. 4595.

[24] P. Thomson, A. Donaldson, and A. Betts. Concurrency Testing Using Schedule Bounding: An Empirical Study. Proceedings of PPoPP 2014, ACM.

[25] A. Udupa, A. Desai, and S.K. Rajamani. Depth Bounded Explicit-State Model Checking. Proceedings of SPIN 2011, LNCS, vol. 6823.

[26] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. Automated Software Engineering, 10(2), 2003.

[27] WALA: T.J. Watson Libraries for Analysis,

[28] M. Wehrle, S. Kupferschmid, and A. Podelski. Transition-Based Directed Model Checking. Proceedings of TACAS 2009, LNCS, vol. 5505.

[29] M. Wehrle and S. Kupferschmid. Context-Enhanced Directed Model Checking. Proceedings of SPIN 2010, LNCS, vol. 6349.

[30] 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.