Fast Error Detection with Hybrid Analyses of Future Accesses

Pavel Parízek
Charles University in Prague, Faculty of Mathematics and Physics
parizek@d3s.mff.cuni.cz

ABSTRACT

Systematic state space traversal is a 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. Some kind of guided 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 significantly improves the performance of error detection.

CCS Concepts

•Software and its engineering → Automated static analysis; Dynamic analysis; Software testing and debugging;

Keywords

state space traversal, heuristics, static analysis, dynamic analysis

1. INTRODUCTION

Efficient detection of concurrency errors, such as deadlocks and atomicity violations, has become very important especially with the greater proliferation of software that exploits multi-core processors. Systematic traversal of the program state space is one of the popular approaches for detecting errors in systems with multiple threads. Its main goal is to check the program behavior under all possible thread interleavings. Although full traversal is not tractable for non-trivial multithreaded programs that exhibit huge numbers of possible interleavings, good performance in practice can be achieved through the use of heuristics and optimizations.

Parízek and Lhoták designed a hybrid analysis that identifies possible accesses to shared variables and used its results to eliminate redundant thread scheduling choices in the context of exhaustive verification [8]. 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 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 changes the order in which transitions are explored, and the second heuristic prunes some transitions at each non-deterministic thread scheduling choice.

The rest of this paper is structured as follows. First we describe important background concepts, including the hybrid analysis, and we also provide an overview of related work. Then we present both heuristics (Section 3), and experimental evaluation in Section 4.

2. BACKGROUND AND RELATED WORK

State Space Traversal. Figure 1 shows the basic algorithm for depth-first traversal of a program state space. The symbol s represents a program state and ch represents a thread choice. Each transition (tr) is a sequence of executed instructions that is associated with a specific thread and bounded by non-deterministic scheduling choices. Some technique of partial order reduction (POR) [2,3] is used to avoid redundant exploration of thread interleavings — more specifically, it creates thread scheduling choices only before the execution of instructions that may access the global state visible by multiple threads. The algorithm maintains also the set of already visited states for the purpose of state matching.

```
visited = {}
explore(s_0, ch_0)

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

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

The function enabled returns a set of transitions that are enabled in a given choice ch and must be explored. Each transition in the set corresponds to one thread that is runnable in state s associated with the choice ch. The function filter may prune some transitions leading from s, and the function order determines the search order in which the transitions are explored. Heuristics and optimizations typically use custom implementations of these functions.

Fast Detection of Concurrency Errors. A very popular approach is guided search, whose basic idea is to navigate towards error states with the help of various heuristics [4]. Randomized search is

also quite effective, especially in combination with parallel traversal of different state space fragments [1]. The results of a random number choice can be used also to identify fragments that will be pruned [7]. Many techniques also restrict the search for errors to a small part of the state space — their overview and experimental comparison is provided by Thomson et al. in a recent study [11].

Closely related to the topic of this paper are the concepts of *useless transitions* and *interference contexts* proposed by Wehrle et al. [13, 14]. Authors define a heuristic that gives preference to transitions accessing the same variables as some previous transitions on the current state space path. However, the respective publications do not discuss approaches to identify such interfering transitions.

Kim et al. [6] proposed an approach to detect race conditions that is based on heuristics and information about interfering accesses to variables. The heuristics consider only the execution trace up to the current dynamic state, and neglect possible future behavior.

Hybrid Analysis. The hybrid analysis provides over-approximate description of the future behavior of program threads in terms of accesses to shared variables [8, 10]. For each program point p in each thread T, the analysis computes the set of object 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. Complete results are then computed on-the-fly during the state space traversal, using information taken from dynamic program states. It is just needed to retrieve the current locations in all frames on the dynamic call stack of each thread T_i and to merge the corresponding data computed by the static analysis. Considering the dynamic state s, results for the current program points $pc_1, \ldots, pc_n$ of threads $T_1, \ldots, T_n$, respectively, capture all accesses that may happen during the program execution from s.

The complete results of hybrid analysis are very precise, because they reflect the dynamic calling context of each pc_i. On the other hand, they 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 of our approach is to use the results of hybrid analysis in three ways: (1) to eliminate redundant thread choices during the state space traversal, (2) to determine the search order for transitions, and (3) to prune transitions that likely do not lead to an error state. A method for eliminating redundant choices has been already proposed in [8], so here we focus on the second and third item in the list. We describe the heuristic for reordering transitions in Section 3.1, and the heuristic that prunes transitions in Section 3.2. Both heuristics use the hybrid analysis results on-the-fly together with knowledge of the current dynamic state and execution path.

When designing the heuristics, we have built upon the concepts of useless transitions and interference contexts proposed by Wehrle et al. [13, 14]. However, contrary to that prior work, 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.

3.1 Reordering Transitions

Our main rationale behind this heuristic is to prioritize threads that are more likely to trigger errors caused by race conditions over fields and array elements. The heuristic changes the order of transitions at each choice based on two pieces of information: (1) a list of past accesses to fields and array elements that were already performed on the current state space path, and (2) results of the hybrid analysis that specify possible future accesses. Upon reaching a state that is associated with a thread choice, the heuristic identifies threads that may in the future perform actions possibly interfering with some of the past accesses. An up-to-date list of past accesses is maintained by the state space traversal procedure.

Let s be the current 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 heuristic queries the hybrid analysis results in order to retrieve a set F_i of possible future accesses by T_i, and then computes intersection of L and F_i. The set of *interfering threads* contains every T_i for which the intersection is not empty. All transitions associated with interfering threads are then moved to the front of the list (order) at the choice ch, and therefore future actions of interfering threads are explored first. Note, however, that we do not enforce any particular order within the group of interfering threads, and we also preserve the default order for the other threads (transitions).

We have designed this heuristics as configurable by the user. One parameter is the percentage of the length of the current path that determines the fragment from which past accesses are collected into the set L. A small value might enable faster detection of errors when the possibly racy accesses from different threads are performed close to each other on an execution path. Note that usage of any value of this parameter is sound because all transitions are explored eventually — this heuristic influences only the search order.

Another parameter is a boolean flag that says whether the heuristic has to distinguish read and write accesses. It makes the heuristic more precise, but possibly more expensive in terms of running time. Otherwise, if this feature is disabled then all accesses to a particular field or array element are considered as potentially interfering.

A configuration of the reordering heuristic is a pair (RW, P), where RW is the flag that says whether to distinguish read and write accesses, and P is the percentage of the current path that is analyzed at each choice to collect past accesses into the set L.

3.2 Pruning Transitions

The second heuristic works in a similar way to the first one, and it has the same parameters. Just instead of reordering, 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.

An exception to the general rule applies when all transitions enabled in a given state would be pruned. One transition is preserved in such a case. This is necessary to ensure that at least some execution traces are fully explored. Note also that pruning a transition associated with a thread T at a state s does not mean that future actions of T are never explored. There must exist an execution path starting in s such that (i) either T will belong to the set of interfering threads at some point on the path or (ii) T will be a single runnable thread at some point. Still, this heuristic is not sound. Transitions that represent actions not considered by the hybrid analysis (e.g., starting of a new thread or releasing blocked threads) may be pruned even if they could trigger possibly errorneous behavior.

4. EVALUATION

The main goal of our evaluation was to find how much the hybrid analysis and proposed heuristics improve the error detection performance. We also wanted to check whether our results confirm the benefits of similar heuristics designed by Wehrle et al. [13, 14].

Here, in this paper, we provide only selected results and discuss the most significant observations. More details about our imple-

Table 1: Experiments: performance improvement over existing techniques

	heap reach POR		HR + hybrid		dynamic POR		DPOR + hybrid		random search	random + hybrid
benchmark	states	time	states	time	states	time	states	time	time	time
Daisy	493645	139 s	297523	95 s	-	-	266291	91 s	86 ± 57 s	59 ± 38 s
Elevator	61465	14 s	16574	7 s	1671602	511 s	485070	143 s	4 ± 4 s	3 ± 2 s
jPapaBench	457139	144 s	94567	41 s	-	-	-	-	1 ± 0 s	3 ± 0 s
CDx	383568	2870 s	48069	456 s	-	-	-	-	162 ± 115 s	63 ± 46 s
Alarm Clock	950	1 s	313	3 s	786	1 s	165	3 s	1 ± 0 s	3 ± 0 s
Rep Workers	12951140	6113 s	441253	178 s	14303	5 s	3909	4 s	2761 ± 2996 s	3 ± 0 s
QSortMT	4883	2 s	2564	2 s	-	-	-	-	1 ± 0 s	2 ± 0 s

Table 2: Experiments: different configurations of the reordering heuristic

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
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
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
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
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
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
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

mentation, the complete experimental evaluation, and more extensive discussion of all the results can be found in [9].

Implementation. We implemented the hybrid analysis and heuristics using Java Pathfinder (JPF) [5] and the WALA library for static analysis [12]. The complete implementation, together with experimental setup and benchmarks, is publicly available at http://d3s.mff.cuni.cz/projects/formal_methods/jpf-static/musepat16.html.

Benchmarks. We measured the performance of hybrid analysis and heuristics on 7 multithreaded Java programs. jPapabench, with 4500 lines of code and 7 threads, is the most complex benchmark that we used. Some of the benchmarks already contained errors in the form of atomicity violations caused by incorrect synchronization of accesses to fields and array elements. For the purpose of experiments, we injected similar errors into the other benchmarks.

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

- POR based on heap reachability [2] with the default search order (table column with the label "heap reach POR"),
- POR based on heap reachability with the hybrid analysis used to eliminate redundant choices (column "HR + hybrid"),
- dynamic POR algorithm by Flanagan and Godefroid [3] combined with state matching (column "dynamic POR"),
- dynamic POR with state matching and hybrid analysis that eliminates redundant 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").

We report the total number of states processed before JPF detects an error, and the total running time. Note that the number of states is equivalent to the number of thread choices. 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. We omitted the number of states for random search in order to save space. We used 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.

Tables 2 and 3 provide data for selected configurations of both heuristics, used together with POR based on heap reachability. Results for the standalone hybrid analysis represent the baseline in this case. For the configuration variable P, we picked the values {10, 25, 50, 75, 90, 100} in order to cover the whole interval evenly. We report data only for configurations that distinguish between read and write accesses ("RW: on"), as it is sufficient to show the dependency of error-detection performance on the value of P.

Discussion. Results of our experiments show that (i) usage of the hybrid analysis in JPF can reduce the time needed to find errors quite significantly and (ii) the proposed heuristics help to achieve even better performance for some configurations and benchmarks. We observed big improvements especially for the more complex benchmarks from our set, such as CDx, Daisy, jPapaBench, and Rep Workers. In the rest of this section, we highlight our main observations that are based on data in tables 1-3.

Compared to the existing approaches that we considered in our evaluation, JPF with the hybrid analysis and heuristics needs to explore much less states to find an error for all 7 benchmarks, and

Table 3: Experiments: different configurations of the pruning 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
Elevator	states: 16574 time: 7 s	RW: on	states	16574	16574	-	-	-	-
			time	8 s	8 s	-	-	-	-
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
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
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
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
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

the running time is reduced by a great margin for 5 benchmarks.

Just the hybrid analysis alone 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. Heuristics yield further reduction of the running time — for example, by the factor of 60 in the case of Rep Workers and by the factor of 1.5 for CDx. To be more specific, the heuristic based on reordering transitions (Table 2) achieved better performance in the case of four benchmarks (CDx, Elevator, QSortMT, Rep Workers), but only using configurations where the variable P has the value 10 % or 25 %. On the other hand, results for the heuristic based on pruning transitions (Table 3) show that it can find an error much faster in some configurations and for some benchmarks, but it may also miss some errors. Good performance is achieved by the pruning heuristic, in general, for configurations where the value of P is 10 %. An exception to the pattern described above is Alarm Clock, for which the best performance was achieved by configurations with P in the range 50 % – 90 %.

Surprisingly, dynamic POR is much slower than POR based on heap reachability for Elevator, and for 3 benchmarks it even did not find any error before the time limit.

The performance of random search is improved quite significantly by the hybrid analysis in the case of 4 benchmarks — Daisy, Elevator, CDx, and Rep Workers. Results are comparable in all the other cases, and the running times are very low in general.

In the technical report [9], we provide also data on scalability for benchmarks that are parameterizable by the maximal number of threads. Results show that JPF with the hybrid analysis and heuristics scales quite well for the purpose of detecting errors, even though it runs out of the time limit for some of the more complex benchmarks (e.g., Daisy) when a higher number of threads is used.

The cost of the hybrid analysis is very low in general. However, it may be responsible for a slight increase of the running time when JPF has to explore only a small number of states before detecting an error — see, for example, the data for Alarm Clock in Table 1.

The cost of both heuristics depends on the value of P. 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.

5. ACKNOWLEDGEMENTS

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

6. REFERENCES

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

[2] 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.

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

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

[5] Java Pathfinder, http://babelfish.arc.nasa.gov/trac/jpf

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

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

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

[9] P. Parízek. Hybrid Analysis of Future Accesses and Heuristics for Fast Detection of Concurrency Errors. Tech. Report No. D3S-TR-2015-03, Dep. of Distributed and Dependable Systems, Charles University in Prague, December 2015. http://d3s.mff.cuni.cz/publications/download/D3S-TR-2015-03.pdf

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

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

[12] WALA, http://wala.sourceforge.net/

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

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