Hybrid Partial Order Reduction with Under-Approximate Dynamic Points-To and Determinacy Information

Pavel Parízek
Charles University, Faculty of Mathematics and Physics

***Abstract*—Verification techniques for concurrent systems are often based on systematic state space traversal. An important piece of such techniques is partial order reduction (POR). Many algorithms of POR have been already developed, each having specific advantages and drawbacks. For example, fully dynamic POR is very precise but it has to check every pair of visible actions to detect all interferences. Approaches involving static analysis can exploit knowledge about future behavior of program threads, but they have limited precision.**

We present a new hybrid POR algorithm that builds upon (i) dynamic POR and (ii) hybrid field access analysis that combines static analysis with data taken on-the-fly from dynamic program states. The key feature of our algorithm is usage of under-approximate dynamic points-to and determinacy information, which is gradually refined during a run of the state space traversal procedure. Knowledge of dynamic points-to sets for local variables improves precision of the field access analysis. Our experimental results show that the proposed hybrid POR achieves better performance than existing techniques on selected benchmarks, and it enables fast detection of concurrency errors.

I. Introduction

Software systems that involve multiple threads are now ubiquitous, but also prone to errors such as data races. Therefore, efficient methods for automated verification of such systems are important. Verification algorithms based on systematic state space traversal are particularly suited for this purpose, but they do not scale well because of the huge number of possible thread interleavings exhibited by any non-trivial system. An important piece of such algorithms is partial order reduction (POR), which identifies the subset of possible thread interleavings that must be explored to cover all observable behaviors of a given system, and in this way improves the performance and scalability of verification. The goal of POR is: (1) to ensure that, for each set of thread interleavings that differ only in the order of independent actions, at least one interleaving from the set is explored, and (2) to minimize the number of thread interleavings from each set that are explored — that means exactly one thread interleaving in the optimal case. To achieve this goal, POR techniques create non-deterministic thread scheduling choices only at visible actions that represent possible interference between threads.

We distinguish between *visible* actions, which read or modify the global state reachable by multiple threads, and *thread-local* actions. Visible actions are, for example, accesses to fields of heap objects and thread synchronization operations. Only a subset of visible actions is responsible for the actual communication between threads — we call such actions *interfering*. All other actions are *independent*. The state space traveral procedure with POR has to explore all possible interleavings of interfering actions. The main challenge is to identify the interfering actions as precisely as possible.

A prominent example of a POR technique is the dynamic approach by Flanagan and Godefroid [5]. Their algorithm explores individual execution traces (interleavings) one by one using dynamic analysis, and for each trace determines the set of heap objects and fields that were truly accessed by multiple threads. An advantage of dynamic POR is that it recognizes each dynamic heap object, and thus identifies shared memory locations precisely. New thread choices are created retroactively only at accesses to shared locations, and every added choice yields traces that must be explored eventually. On the other hand, a limitation of this approach to dynamic POR is that it has to (i) explore each trace until the end state, (ii) keep track of all accesses to object fields that occurred on the trace, and (iii) check every pair of visible actions to detect all interferences (i.e., to compute the independence relation). This can negatively impact performance especially in the case of programs with large state spaces and long execution traces.

Another viable approach to POR is to use a hybrid analysis of field accesses [12] [13], which consists of two phases — static and dynamic. Static analysis computes only partial data for each program point (i.e., a source code location). Full results are generated and applied on-the-fly during the state space traversal based on the knowledge of dynamic program states. This approach determines an over-approximate set of interfering actions, and it is directly compatible with state matching. On the other hand, it has limited precision because it uses a static pointer analysis that cannot distinguish among dynamic heap objects allocated at the same code location.

We present a new hybrid POR algorithm that builds upon dynamic POR [5] and the hybrid field access analysis [12], combining their advantages and addressing their limitations. The main idea behind the proposed algorithm is the usage of under-approximate dynamic points-to and determinacy information for local variables, which is iteratively refined. We use the definition of determinacy by Schaefer et al. [14], which intuitively says that a given variable is *determinate* at a particular code location if it always has the same value every time program execution reaches the location. An informal

overview of the hybrid POR algorithm follows.

The state space traversal procedure augmented with our hybrid POR works in a similar way to dynamic POR. It starts by exploring an arbitrary execution trace, and on-the-fly inserts new thread choices at field accesses that occur on the trace and are deemed to be interfering. Two accesses to the same field f are *interfering* if (i) they are performed by different threads, (ii) they may target the same dynamic heap object, and (iii) at least one of them is a write. Like in the dynamic POR, every new choice corresponds to additional thread interleavings that must be explored eventually. Another important feature of the proposed algorithm is compatibility with state matching, which is needed to support cyclic state spaces.

Hybrid POR recognizes interfering field accesses based on (i) the results of the hybrid field access analysis and (ii) the dynamic points-to and determinacy information. Execution traces are processed by dynamic analysis that tracks field accesses on heap objects. When processing a field access, our hybrid POR algorithm performs the following three steps:

1) It retrieves the dynamic concrete value of the local variable through which the field access was performed, and updates the points-to and determinacy information to reflect the concrete value (i.e., a dynamic heap object).
2) Then it queries the hybrid field access analysis and the dynamic points-to information to find whether there may be a future access to the same field on the same dynamic heap object — in that case, the current field access is marked as interfering with the future one and a new thread choice is created.
3) Finally, for every previous field access on the current trace, our algorithm checks whether the previous access may be interfering with the current field access according to the updated points-to and determinacy information — additional thread choices may be created in this way.

We illustrate the main steps of hybrid POR using the program in Figure 1. It involves two threads that perform field accesses on shared objects. Let us assume that the execution trace read o.f ; write p.g ; read q.h ; write o.f is explored first. Hybrid POR detects interference between the read access to o.f in thread T_1 and the subsequent write in T_2 just before execution of the write, when the dynamic value of the variable o is added to its points-to set and previous field accesses on the trace are inspected. A new thread choice is added at the read access in T_1. The second explored trace is read q.h ; write o.f ; read o.f ; write p.g. In this case, the interference is discovered already before the write to o.f, which precedes all other accesses to o.f on this trace, because the points-to sets of all variables are already non-empty and thus the hybrid analysis can identify the future interfering read access.

The key feature of our algorithm is that the initial under-approximation of points-to and determinacy information is very coarse — every variable of a reference type is assumed to be determinate and to have an empty points-to set. However, as more and more execution traces are explored during the state space traversal, the under-approximation is gradually refined to cover the possible behavior of a program under different schedules. The dynamically computed points-to set and determinacy status for each reference variable enables the hybrid field access analysis to provide precise information about the future behavior of each thread, and that in turn enables the hybrid POR to detect real interference between field accesses in different threads very precisely.

T_1	T_2
read o.f ;	read q.h ;
write p.g	write o.f

Fig. 1. Example program

A run of the state space traversal procedure terminates when there are no unexplored thread choices and interleavings left. Then, iterative refinement of the points-to and determinacy information must have reached a fixed point, and results of the hybrid analysis together with the dynamic points-to information soundly over-approximate the set of field accesses that may occur during the program execution under any thread schedule. State space traversal with hybrid POR then covers all interleavings of interfering actions. However, in general, termination of the procedure is not guaranteed.

In addition to the limitations of existing approaches mentioned above, our rationale behind the hybrid POR algorithm is based on partially-automated inspection of benchmark programs (Section IV), where we analyzed the determinacy status of variables through which field accesses are performed. We discovered that, for many of the programs, there is a quite high number (over 50%) of cases where such variables are determinate at code locations that correspond to field accesses. Our goal was to exploit this observation to optimize verification of multithreaded programs, and also to enable faster detection of real errors. Results of our experiments show that usage of precise dynamic points-to sets and determinacy information (1) eliminates many redundant thread choices and (2) improves performance especially for large programs.

The rest of the paper is organized as follows. We provide background definitions and an overview of the hybrid field access analysis in Section II. Then, in Section III, we formally define the hybrid POR algorithm and prove its soundness. Section IV contains results of experiments with our implementation and their discussion. We compare the proposed approach with related work in Section V, and then we conclude.

II. Background

Program and state space. A program P consists of threads $t_1, \ldots, t_n$ from the set T, where each thread executes actions from the set A. Each action $a \in A$ corresponds to a program statement. Each state of the program is a snapshot of all variables, heap objects, and threads at some point during its execution. An atomic transition tr between two states is a pair $tr = (t, [a_0, \ldots, a_n])$ of a thread $t \in T$ and a sequence of actions executed by t. The first action a_0 in the sequence is interfering and others must be independent. There can be just one interfering action in any transition because a new thread choice is created when the action to be executed next is interfering. We assume that states are explicitly saved only

at transition boundaries, and therefore each visible state s is associated with a choice ch over all threads runnable in s. An execution trace e is a sequence $(t_{i_0}, a_0), \ldots, (t_{i_n}, a_n)$ of thread-action pairs. Each trace represents one thread interleaving. Such definition of execution traces allows us to identify, for each field access action a, the specific thread that executed the action a — this information is needed to detect interference. We also assume that the only source of nondeterminism in the program state space are thread scheduling choices at interfering actions. Input data must be specified explicitly in the source code.

Determinacy. In this paper, we extend the definition of determinacy from [14], which applies only to sequential programs, towards multiple threads. A variable v is determinate at a program point p in the context of thread t, if v always has the same value every time (1) execution reaches the point p and (2) the thread t is active. Only the values assigned to v in the scope of thread t are considered when the determinacy status of v with respect to p and t has to be updated.

Hybrid field access analysis. The hybrid analysis of field accesses was introduced by Parízek and Lhoták [12]. It combines static analysis with information taken on-the-fly from dynamic program states. For each dynamic state s reached during the traversal, and for each thread t in s, it computes an overapproximation of the set of object fields possibly accessed by t in the future on any execution path starting in s. Results of the hybrid analysis are computed in two phases.

The first phase involves static analysis, which is run before the state space traversal and computes only partial results. We use a backward flow-sensitive and context-insensitive interprocedural data-flow analysis. For each point p in the code of thread t, it provides information that cover the future behavior of t only between the point p and return from the method containing p (including nested method calls transitively).

The second phase is performed on-the-fly during the state space traversal. Full results of the hybrid analysis are generated on-demand, every time POR has to decide whether the current field access is interfering with some other action. It is done based on the knowledge of the dynamic call stack of each thread. Let s be the current dynamic state just before execution of a field access. The dynamic call stack of a thread t specifies a sequence $p_0, p_1, \ldots, p_n$ of program points, where $p = p_0$ is the current program counter of thread t (in the top stack frame), and $p_i, i > 0$ is the point from which execution of t would continue after return from the method associated with the previous stack frame. When the hybrid analysis is queried about the current point p of thread t in state s, it takes data computed by the static analysis for each point $p_i, i = 0, \ldots, n$ on the dynamic call stack of t and merges all the data to get the complete result for p in the context of the state s. The result covers the future behavior of t after the point p, and also the behavior of all child threads of t started after p.

A consequence of the usage of dynamic call stack is that results of the whole hybrid analysis are fully context-sensitive and therefore very precise. On the other hand, the results are always valid only for the current dynamic state.

```
1  init: visited = ∅ ; pointsto = ∅ ; determinacy = ∅
2  exploreState(s0, ch0, [ ], ∅)
3
4  procedure exploreState(s, ch, accs, hbo)
5    if s ∈ visited return
6    visited = visited ∪ s
7    for t ∈ getRunnableThreads(ch) do
8      (s', accs', hbo') = executeTransition(s, t, accs, hbo)
9      ch' = createThreadChoice(s')
10     exploreState(s', ch', accs', hbo')
11   end for
12
13 procedure executeTransition(s, tc, accs, hbo)
14   ac = getNextAction(tc) // must be interfering
15   while ac ≠ null do // not at the end of thread
16     s = executeAction(s, ac, tc)
17     if isErrorState(s) terminate
18     if isFieldAccess(ac) then
19       (vc, oc, fc, pc) = getFieldAccessInfo(ac, s)
20       accs = accs ⊕ (ac, tc)
21       extendDynPointstoSet(vc, oc, tc)
22       inspectPreviousAccesses(s, ac, tc, accs, hbo)
23     end if
24     hbo = updateHappensBeforeOrder(hbo, ac)
25     ac = getNextAction(tc)
26     if isInterferingAction(ac, tc, s) break
27   end while
28   return (s, accs, hbo)
29
30 procedure isInterferingAction(ac, tc, s)
31   if isFieldAccess(ac) then
32     for t ∈ getOtherThreads(s, tc) do
33       if existsFutureInterferingAccess(ac, tc, t, s) then
34         return true
35   // other kinds of actions
36   return false // default
```

Fig. 2. Algorithm for state space traversal with hybrid POR

III. Hybrid POR Algorithm

Figure 2 shows the core of the algorithm for state space traversal combined with hybrid POR. Procedures that detect interfering field accesses are defined in Figure 3. We present a recursive definition of the algorithm because it allows us to explain the key aspects of hybrid POR in a simple and clear way — especially in comparison with an iterative encoding of the algorithm that is more efficient (and therefore used by our implementation) but also more intricate.

Core of the algorithm. The top-level procedure exploreState drives the state space traversal and performs state matching. When this recursive procedure is called for a state s that has not been already visited during the traversal, it retrieves all threads enabled in the choice ch associated with s (line 7) and explores the next transition for each of the threads. Traversal terminates immediately when it reaches an error state.

Three global data structures are used by the algorithm — the set of visited states, the relation *pointsto* that captures dynamic points-to sets for variables, and the relation *determinacy* that maintains the determinacy status of every variable. Information stored in these data structures is preserved across all execution traces. Reading and updating of the relations *pointsto* and *determinacy* are implemented by

```
37 procedure existsFutureInterferingAccess(a_c, t_c, t, s)
38     (v_c, o_c, f_c, p_c) = getFieldAccessInfo(a_c, s)
39     for a_t ∈ getFutureFieldAccesses(t, s) do
40         if ¬(isInterferingAccess(a_c, a_t) ∧ t_c ≠ t) continue
41         (v_t, f_t, p_t) = getFieldAccessInfo(a_t)
42         if isDeterminate(v_t, p_t, t) then
43             o_t = getSingleDynPointstoValue(v_t, p_t, t)
44             if o_c = o_t return true
45         else if // indeterminate variable
46             if o_c ∈ getDynPointstoSet(v_t, p_t, t) return true
47         end if
48     end for
49     return false
50
51 procedure inspectPreviousAccesses(s, a_c, t_c, accs, hbo)
52     (v_c, o_c, f_c, p_c) = getFieldAccessInfo(a_c, s)
53     for (a_t, t) ∈ accs do // previous accesses
54         if ¬(isInterferingAccess(a_c, a_t) ∧ t_c ≠ t) continue
55         (v_t, f_t, p_t) = getFieldAccessInfo(a_t)
56         pt_t = getDynPointstoSet(v_t, p_t, t)
57         if o_c ∈ pt_t ∧ ¬isOrderedStrictly(hbo, a_t, a_c) then
58             markInterferingAction(a_t)
59     end for
```

Fig. 3. Procedures that recognize interfering field accesses

auxiliary procedures isDeterminate, extendDynPointstoSet, getDynPointstoSet, and getSingleDynPointstoValue. Both points-to and determinacy information are always specific to a tuple (v, p, t) of a variable v, a program point p, and a thread t. It means that, like in the case of determinacy, the dynamic points-to set for a variable v is defined only in the context of a specific program point p and thread t.

In addition, our algorithm uses the data structures named $accs$ and hbo, which hold information specific to the currently processed dynamic execution trace. The list $accs$ contains all the field access actions that were performed on the current trace before the current state, and hbo captures the happens-before ordering relation between actions. Both data structures are needed for precise identification of pairs of interfering field accesses, as we explain below in more detail.

The symbol t_c in Figures 2 and 3 represents the currently active thread. Symbols having the subscript c, such as a_c and f_c, refer to information associated with the current thread t_c or with the current field access action. Analogously, symbols having the subscript t refer to information associated with some other thread that is represented by the symbol t.

A run of the algorithm starts with empty determinacy and points-to relations (line 1) in order to satisfy the initial assumption that (i) every variable is determinate and (ii) all possibly concurrent accesses to the same field are performed through variables that have disjoint points-to sets. This initial coarse under-approximation is refined during the state space traversal, and at every moment it reflects all the actions and traces that were explored so far. The hybrid field access analysis depends on the points-to and determinacy information. Both the precision of the analysis and its coverage of possible future behavior of program threads are improved during the run of the algorithm based on the gradually refined under-approximation.

For each executed field access action a_c, the algorithm performs the following four steps (at lines 19-22):

1) Calls the auxiliary procedure getFieldAccessInfo to retrieve information about the field access: the variable v_c through which the access is performed, a dynamic heap object o_c to which v_c points in the current dynamic state s, the field name f_c, and a program point p_c.
2) Updates the list $accs$ of field accesses that were already performed on the current execution trace.
3) Adds the heap object o_c into the points-to set of the variable v_c, and updates the determinacy information for v_c based on the size of its points-to set. The variable v_c remains determinate only if the size is 0 or 1.
4) Inspects all the previous field accesses on the current trace in order to detect additional pairs of interfering actions. We provide more details about this step later.

We use two variants of the function getFieldAccessInfo. The dynamic heap object is returned as an element of the tuple only by the variant that takes the current state s as an argument.

The happens-before ordering relation is updated for each executed action (line 24). It has to reflect also synchronization actions that may block or release some thread.

A transition ends when the next action a_c to be executed in thread t_c is interfering with some other action, because then a new thread choice has to be created. The main part of the corresponding logic is implemented by the procedure isInterferingAction. For every thread other than the current one (t_c), it calls another procedure that looks for interfering future field accesses (line 33). If the action a_c is a field access, and some thread t may in the future access the same field of the same dynamic heap object, then a_c is interfering. Similar checks have to be done for all kinds of actions.

Detection of interfering field accesses. The procedure existsFutureInterferingAccess queries the hybrid field access analysis for the current program point in thread t (line 42), and inspects the results to find whether some of the possible future accesses by thread t may be interfering with the current field access action a_c. For each interfering future access, the algorithm queries the points-to set and determinacy status of the respective variable v_t at the point p_t in thread t. It has to decide whether one of the following two conditions holds.

(A) The variable v_t is determinate, and its points-to set has a single element o_t that is equal to the target object o_c of the currently processed field access a_c.
(B) The variable v_t is not determinate, which means it may point to different objects at distinct execution traces, and some element of the points-to set of v_t is equal to o_c.

If one of the conditions is true then the current field access action a_c in the active thread t_c may really interfere with the future action a_t on some thread interleaving. Note also that the condition A does not hold for all determinate variables, because the points-to set of some variable can be empty as a consequence of the initial under-approximation.

After each update of the dynamic points-to sets (line 21), where the target object o_c of the current access a_c is the

newly added element, it is necessary to check all the previous field accesses on the current trace and compare them with a_c, because some new pairs of interfering actions may be discovered. This is done in the procedure $\mathrm{inspectPreviousAccesses}$, using an approach very similar to the original dynamic POR algorithm [5] (which is based on vector clocks). For each possibly interfering previous access a_t on the current trace, the procedure retrieves the dynamic points-to set pt_t for the respective variable v_t and checks presence of o_c in the set. If o_c is in pt_t and there is not a strict happens-before ordering between the field accesses in question, then the action a_t has to be marked as interfering with a_c. The corresponding thread choice will be added later during the state space traversal.

Our hybrid POR algorithm uses the happens-before ordering relation in the same way (and for the same purpose) as the original approach to dynamic POR [5]. That is, to avoid identifying some pairs of field accesses spuriously as interfering, when only a single interleaving of the actions is possible due to thread synchronization. A pair (a_i, a_j) of interfering field accesses, where $i < j$, meaning that a_i precedes a_j on the current trace, is strictly ordered according to the happens-before relation if the following two conditions hold:

1) There is an action a_k, $i < k < j$, that is in the happens-before relation with a_i or with some action following a_i.
2) Both actions a_k and a_j are executed by the same thread that is different from the thread executing a_i.

If both conditions hold, then a_i must happen strictly before a_j with respect to the ordering relation for the current trace. A thread choice at a_i can be soundly avoided because actions a_i and a_j cannot be interleaved the other way in this context.

The main benefit of the dynamic points-to and determinacy information is that hybrid POR can detect interference between field accesses very precisely with respect to (i) possible future behavior of program threads and (ii) previous accesses on the current trace. Usage of the under-approximate dynamic points-to sets enables the hybrid field access analysis to provide much more precise results than with a static pointer analysis.

A. Soundness and Termination

Theorem 1. *The proposed algorithm for state space traversal with hybrid POR terminates either (A) when it reaches an error state or (B) when all possible distinct interleavings of interfering actions in concurrent threads have been explored.*

Proof. The first condition (A) is trivally satisfied by the call of the procedure $\mathrm{isErrorState}$ at line 17 in Figure 2, so we focus on the second condition (B) in our proof. We need to consider only distinct interleavings of interfering actions, because execution traces that differ only in the order of independent actions yield equivalent observable behavior. To satisfy the condition B, our algorithm has to (1) identify all pairs of interfering field accesses, (2) add a new thread choice to every interfering action, and (3) explore all thread choices in the state space. We show in the next few paragraphs that all three tasks are performed in a sound manner by the algorithm.

For each field access a_c, all interfering actions are detected by combination of the hybrid analysis with the inspection of previous accesses. The hybrid analysis alone may fail to detect some future accesses interfering with a_c because of the under-approximate points-to sets. Let a_t be such a future access. Interference between a_c and a_t will be detected when a_t is executed, because the target dynamic object of a_t becomes known at that moment and a_c will then represent a previous access with respect to a_t. A complete dynamic information about every previous action on the current trace is available.

Regarding thread choices at interfering actions, we distinguish two cases. If the algorithm determines that the current field access action a_c in the active thread t_c is interfering with some possible future accesses, then a new choice is created at the current action (line 9 in the procedure $\mathrm{exploreState}$) and its exploration starts immediately. The process is more complicated when some previous field access a_t on the current trace is newly identified as interfering with a_c. A corresponding thread choice will be created and explored later just when the state space traversal procedure backtracks over the action a_t. We omitted the respective statements — for adding new choices retroactively to previous accesses on the current trace — from the pseudo-code in Figure 2, because this aspect of hybrid POR cannot be encoded in the recursive definition of the algorithm in a simple way.

It follows from the discussion above that a new choice is created at a field access action a iff there is some other access interfering with a. For each pair (a_i, a_j) of interfering accesses, existence of the choice guarantees that both interleavings of a_i and a_j will be explored eventually. The state space traversal procedure explores all transitions enabled at each choice ch, and therefore backtracks from a state s associated with ch only when the whole state space fragment with s as the root has been processed. Consequently, no thread interleaving will be omitted during the traversal. □

IV. Evaluation

We implemented our hybrid POR algorithm in Java Pathfinder (JPF) [21], which is a framework for verification and analysis of Java programs. JPF is responsible for traversal of the program state space and for execution of Java bytecode instructions. In order to support decisions about thread choices, we created a non-standard interpreter of bytecode instructions for accesses to object fields. Our interpreter queries results of the hybrid field access analysis and the data structures maintained by the algorithm. We used the WALA library [23] for static analysis and JPF API to retrieve information from the dynamic program states. One custom listener for JPF collects the dynamic points-to sets, and another listener computes the happens-before ordering relation.

The complete source code of our implementation, together with benchmark programs and scripts needed to run all experiments, is available at http://d3s.mff.cuni.cz/projects/formal_methods/jpf-static/fmcad16.html.

The goal of our experimental evaluation was to compare the performance and scalability of hybrid POR against selected

other approaches to POR. For each approach, we wanted to find how much time it takes to explore the whole state space of individual benchmarks, and how fast it can detect concurrency errors. We consider POR based on heap reachability [4] and the original dynamic POR [5], both combined with stateful traversal of the program state space. While POR based on heap reachability is supported by JPF for a long time, we created our own implementation of dynamic POR. Note that combination of the original dynamic POR with stateful search addresses the first limitation mentioned in Section I, i.e. the need to explore each trace until the end state — details are provided in Section V. We used this variant of dynamic POR in our experiments in order to perform a fair comparison, because other approaches involve state matching too. Moreover, the original dynamic POR [5] (that performs a stateless search) would not scale at all to most of our larger benchmarks, as we found at the initial stage of this research project.

Benchmarks. We performed experiments on 16 multithreaded Java programs, mostly from widely known benchmark suites (Java Grande [19], CTC [18], Inspect [16], and pjbench [22]). Other programs, such as jPapaBench [20] and Simple JBB, were used in our previous work and recent experimental studies. The smallest benchmark in our set is Prod-Cons with 130 lines of source code and 2 threads, while the most complex one is jPapaBench with 4500 lines of code and 7 threads.

Experiments. We evaluated four configurations of POR in our experiments: (1) POR based on heap reachability, (2) POR based on heap reachability together with hybrid analysis of field accesses, (3) dynamic POR with state matching, and (4) hybrid POR. In tables with results, we use a short name "Heap Reach" for the first configuration in the list and a short name "HR + fields" for the second configuration (which corresponds to the technique proposed by Parízek and Lhoták in [12]).

For every experiment, we report (1) the number of thread choices created by JPF during the state space traversal, which we use to assess precision of POR techniques, and (2) the total running time of JPF (with static analysis), which indicates performance and scalability of the techniques.

Table I shows results for the first set of experiments, where we configured JPF to explore the whole state space of each benchmark, i.e. we disabled the check for error states. We used the time limit of 12 hours and memory limit of 20 GB.

Error detection performance of the POR techniques is reported in Table II. For the purpose of these experiments, we selected only those benchmarks from our set that already contained some concurrency errors (e.g., race conditions). We used the time limit of 1 hour only in this case.

Discussion. In the case of complete traversal of a program state space (Table I), the results are mixed. Hybrid POR achieves better precision and performance than other approaches for 5 benchmarks out of 15 — Cache4j, Alarm Clock, RAX Extended, Rep Workers, and TSP. The biggest improvement was achieved for Cache4j, where hybrid POR is faster than the second-best configuration "HR + fields" by the factor of 3.1. Dynamic POR achieves better precision and performance than other techniques for 2 benchmarks, Simple JBB and Linked List. For three benchmarks — CoCoME, Crypt, and SOR — hybrid POR and dynamic POR have the same precision, but dynamic POR yields better performance.

Results are not clearly in favor of one technique in the case of remaining five benchmarks. Hybrid POR is more precise than "HR + fields" for CRE Demo and Daisy, but it has the same or worse performance. Dynamic POR achieves the best precision for CRE Demo and Elevator, but it is slower than hybrid POR in both cases. On the other hand, hybrid POR is the most precise technique for Prod-Cons, while dynamic POR is the fastest. The results for Elevator also highlight the limitations of dynamic POR that we discussed in Section I — it creates less thread choices than hybrid POR, but it is slower by the factor of 2.2.

All the POR techniques failed on jPapaBench because of: (1) a high number of field accesses at execution traces, (2) the length of transitions (JPF must interpret all instructions), and (3) the size of program states which must be processed by the state matching procedure. Over 1.5 million thread choices were created in the state space of jPapaBench until the timeout. In addition, dynamic POR run out of the time limit also for Daisy, Cache4j, and Rep Workers. The memory limit was sufficiently large for all our experiments. However, especially in the case of Daisy and jPapaBench, memory consumption was quite high and therefore a large part of the running time of JPF was spent by garbage collection.

When considering the search for errors (Table II), state space traversal with hybrid POR is faster than competing techniques for 4 benchmarks out of 7 — Elevator, jPapaBench, Rep Workers, and QSort MT. The biggest improvement by the factor of 5.7 was achieved for jPapaBench, which is the most complex benchmark in our set. Although none of the POR techniques can explore the whole state space of jPapaBench, an error state was reached quite fast with hybrid POR.

Dynamic POR detects an error faster in the case of 3 benchmarks out of 7. For one of them, Alarm Clock, dynamic POR is faster than hybrid POR but it creates more thread choices. It failed to detect any error in jPapaBench before the time limit. In the case of LinkedList, hybrid POR creates much more thread choices because of the under-approximate points-to analysis — more program states and field accesses have to be explored before the points-to analysis is refined enough to identify a data race. Results for other benchmarks nevertheless show that such "anomaly" is quite rare.

The hybrid POR algorithm has a certain overhead, when compared to dynamic POR, because it runs the static analysis upfront and performs numerous queries of the hybrid analysis results on-the-fly. This overhead is clearly visible on smaller benchmarks, such as Prod-Cons, for which hybrid POR creates less thread choices but its total running time is higher. Just few seconds are taken by the static analysis for each benchmark.

To summarize, we have made the following two main observations based on our experimental results:

- There is not an obvious winner in the comparison between hybrid POR and dynamic POR, as each is better than the other roughly for a half of the benchmarks.

TABLE I
EXPERIMENTAL RESULTS: COMPLETE STATE SPACE TRAVERSAL

benchmark	Heap Reach		HR + fields		dynamic POR		hybrid POR	
	choices	time	choices	time	choices	time	choices	time
CRE Demo	30942	50 s	2476	9 s	2015	11 s	2086	9 s
CoCoME	81150	160 s	23880	59 s	72	3 s	72	5 s
Daisy	28436002	15405 s	6647236	4574 s	-		6028026	7787 s
Crypt	4993	3 s	9	2 s	9	1 s	9	2 s
Elevator	10167560	7617 s	2731316	1954 s	429466	1288 s	461996	585 s
Cache4j	11716552	7336 s	8615847	5613 s	-		1970110	1785 s
Simple JBB	575519	1768 s	277599	959 s	602	31 s	5648	81 s
jPapaBench	-		-		-		-	
Alarm Clock	531463	432 s	141138	117 s	109018	188 s	48166	47 s
Linked List	5919	3 s	1969	5 s	283	1 s	1422	5 s
Prod-Cons	6410	4 s	2532	6 s	592	1 s	356	4 s
RAX Extended	26346	18 s	13864	13 s	11315	125 s	3519	7 s
Rep Workers	9810966	6850 s	1653037	1264 s	-		739418	584 s
SOR	222129	122 s	86193	72 s	135	2 s	135	4 s
TSP	35273	591 s	9285	154 s	101	65 s	86	37 s

TABLE II
EXPERIMENTAL RESULTS: SEARCH FOR CONCURRENCY ERRORS

benchmark	Heap Reach		HR + fields		dynamic POR		Hybrid POR	
	choices	time	choices	time	choices	time	choices	time
Elevator	27053	12 s	9123	7 s	119797	285 s	1156	5 s
jPapaBench	230709	147 s	48337	40 s	-		262	7 s
Alarm Clock	428	1 s	161	3 s	167	1 s	65	3 s
Linked List	1341	1 s	270	3 s	80	1 s	1290	6 s
RAX Extended	1315	1 s	22	2 s	18	1 s	20	3 s
Rep Workers	6685	6 s	1522	5 s	4516	6 s	1054	4 s
QSort MT	3221	2 s	959	3 s	-		274	2 s

- Hybrid POR achieves better performance than purely dynamic POR on benchmarks that have larger state spaces, such as Cache4j and Daisy, and it can successfully verify 3 out of the 4 benchmarks at which dynamic POR fails.

State space traversal with hybrid POR detects errors very fast, and it can also explore all distinct interleavings of interfering actions in a reasonable time. By manual inspection of the execution logs of JPF, we found that the precision achieved by hybrid POR is largely due to the fact that our algorithm maintains the dynamic points-to sets and determinacy information separately for each program point and each thread. Many redundant thread choices are avoided in this way.

Dynamic POR is less precise than hybrid POR for some benchmarks (e.g., Alarm Clock and Prod-Cons) because it can make a redundant thread choice at instruction i that accesses an object o in the following situation: (1) there is an instruction j in another thread that accesses o, (2) j was executed before i, and (3) the object o is not reachable by multiple threads at the time j was executed. In the case of hybrid POR, the hybrid analysis marks the access by j as thread-local, and therefore enables more precise handling of situations like this.

Now we discuss the performance differences between hybrid POR and dynamic POR in either direction. An advantage of hybrid POR is that it needs to check much less pairs of visible field accesses to detect the interfering ones (i.e., to compute the full independence relation). When the hybrid analysis is queried at a dynamic state, it efficiently identifies all future field accesses that cannot interfere with the current action. For those accesses, hybrid POR can safely omit checks of interference also during the inspection of previous actions on the current trace (line 54 in Figure 3). On the other hand, the purely dynamic POR has to consider all the previous accesses. The difference in the number of pairs of visible accesses that must be checked is quite significant for programs with large state spaces and long execution traces. It is mainly for this reason that hybrid POR achieves better performance on larger benchmarks. On the other hand, for some benchmarks, performance of hybrid POR suffers (i) from imprecision of the underlying static analysis and (ii) from the need to refine the under-approximate information in multiple iterations.

V. RELATED WORK

Many approaches to POR have already been developed in the context of model checking and concurrency testing. Notable examples are the original dynamic POR [5] and Cartesian POR [8]. Furthermore, Abdulla et al. [1] recently proposed an optimal algorithm for dynamic POR.

The goal of all POR techniques is to limit the number of thread choices on every execution trace. In addition, most POR algorithms try to minimize the number of transitions to be explored from each thread choice, using the concepts of persistent sets and sleep sets [6]. Our hybrid POR minimizes just the number of thread choices in the state space, i.e. all enabled transitions are explored at each thread choice. Cartesian POR is the most closely related approach in this respect.

The algorithm for dynamic POR by Flanagan and Godefroid [5] works only with stateless model checking. It does not

handle cyclic state spaces, and performs redundant computation when re-exploring already visited states. Other researchers designed extensions of this algorithm to address its limitations. Yang et al. [17] combined dynamic POR with stateful search, and Thomson et al. [15] proposed the lazy happens-before relation that enables dynamic POR to avoid redundant exploration of some thread interleavings for programs with coarse-grained locking. We adapted the ideas of Yang et al. [17] in our implementation of dynamic POR in JPF. Upon reaching an already visited state, the algorithm just has to consider field accesses that could occur in the rest of the program execution after the state. The necessary information about possible future field accesses is collected at backtracking steps.

Hybrid POR is directly compatible with state matching. Unlike the approach of Yang et al. [17], it does not have to keep track of field accesses that may occur after the given state. The hybrid field access analysis provides the information about future behavior of each thread.

We are aware of several other techniques involving POR that combine static and dynamic analysis [3] [9]. A common pattern behind them is the computation of an approximate dependency relation (a set of interfering actions) by static analysis, followed by (or interleaved with) the usage of dynamic analysis to improve precision based on information taken from dynamic program states and execution traces. For example, Kusano and Wang [9] proposed a framework that combines dynamic POR with a slicing algorithm in order to focus the search on interfering accesses that may cause assertion violations or deadlocks. The slicing algorithm uses static analysis to identify data dependencies and dynamic analysis to compute a precise aliasing information on-the-fly.

Our approach also follows the recent trend of verification algorithms based on iteratively refined under-approximation that captures (prefixes of) feasible execution traces of a given program. This large group of techniques includes, for example, context-bounded search with iterative increase of the maximal number of preemptions [11], and lazy abstraction with refinement based on interpolants [10]. There are even algorithms, such as UFO [2] and SMASH [7], that combine under-approximation with over-approximation and iteratively refine both abstractions until an error is found or the program is proven safe. The motivation behind such techniques is the detection of real errors in a practical time. When there are sufficient resources, use of the iterative refinement enables the algorithms to gradually increase coverage of the program state space, and to eventually explore all the execution traces.

VI. CONCLUSION

Our main contribution presented in this paper is the hybrid POR algorithm and its usage in a state space traversal procedure. Hybrid POR combines static analysis with data taken on-the-fly from dynamic program states, with iteratively refined under-approximate dynamic points-to and determinacy information, and also with the happens-before ordering relation.

Results of our experiments show that, for programs with larger state spaces, hybrid POR outperforms all the other approaches that we considered. The ability to look ahead by querying the hybrid field access analysis is the main reason behind good performance of hybrid POR. On the other hand, there is a certain overhead associated with the hybrid field access analysis. Hybrid POR is slower than dynamic POR for small benchmarks due to the overhead, but it still achieves good running times.

In the future, we would like to adapt the lazy happens-before relation [15] in order to improve the precision and performance of hybrid POR even further. We also plan to investigate possible incremental approaches to POR.

ACKNOWLEDGEMENTS.

This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and Charles University institutional funding SVV-2016-260331.

REFERENCES

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal Dynamic Partial Order Reduction. Proceedings of POPL 2014, ACM.
[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-Approximations to Over-Approximations and Back. Proceedings of TACAS 2012, LNCS, vol. 7214.
[3] G. Brat and W. Visser. Combining Static Analysis and Model Checking for Software Analysis. Proceedings of ASE 2001, IEEE.
[4] M. Dwyer, J. Hatcliff, Robby, and V. Ranganath. Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs. Formal Methods in System Design, 25(2-3), 2004.
[5] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. Proceedings of POPL 2005, ACM.
[6] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, 1996.
[7] P. Godefroid, A. Nori, S.K. Rajamani, and S. Tetali. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. Proceedings of POPL 2010, ACM.
[8] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian Partial-Order Reduction. Proceedings of SPIN 2007, LNCS, vol. 4595.
[9] M. Kusano and C. Wang. Assertion Guided Abstraction: A Cooperative Optimization for Dynamic Partial Order Reduction. Proceedings of ASE 2014, ACM.
[10] K. McMillan. Lazy Abstraction with Interpolants. Proceedings of CAV 2006, LNCS, vol. 4144.
[11] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. Proceedings of PLDI 2007, ACM.
[12] P. Parízek and O. Lhoták. Identifying Future Field Accesses in Exhaustive State Space Traversal. Proceedings of ASE 2011, IEEE.
[13] P. Parízek and O. Lhoták. Model Checking of Concurrent Programs with Static Analysis of Field Accesses. Sci. Comput. Programm., 98, 2015.
[14] M. Schaefer, M. Sridharan, J. Dolby, and F. Tip. Dynamic Determinacy Analysis. Proceedings of PLDI 2013, ACM.
[15] P. Thomson and A. Donaldson. The Lazy Happens-Before Relation: Better Partial-Order Reduction for Systematic Concurrency Testing. Proceedings of PPoPP 2015, ACM.
[16] 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.
[17] Y. Yang, X. Chen, G. Gopalakrishnan, and R.M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. Proc. of SPIN 2008, LNCS, vol. 5156.
[18] Concurrency Tool Comparison repository, https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison
[19] The Java Grande Forum Benchmark Suite, https://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html
[20] jPapaBench, http://d3s.mff.cuni.cz/~malohlava/projects/jpapabench/
[21] Java Pathfinder, http://babelfish.arc.nasa.gov/trac/jpf
[22] pjbench: Parallel Java Benchmarks, https://bitbucket.org/pag-lab/pjbench
[23] WALA: T.J. Watson Libraries for Analysis, http://wala.sourceforge.net/