# Hybrid Analysis for Partial Order Reduction of Programs with Arrays

Pavel Parízek

Charles University in Prague, Faculty of Mathematics and Physics,
Department of Distributed and Dependable Systems

**Abstract.** An important component of efficient approaches to software model checking and systematic concurrency testing is partial order reduction, which eliminates redundant non-deterministic thread scheduling choices during the state space traversal. Thread choices have to be created only at the execution of actions that access the global state visible by multiple threads, so the key challenge is to precisely determine the set of such globally-relevant actions. This includes accesses to object fields and array elements, and thread synchronization.
However, some tools completely disable thread choices at actions that access individual array elements in order to avoid state explosion. We show that they can miss concurrency errors in such a case. Then, as the main contribution, we present a new hybrid analysis that identifies globally-relevant actions that access arrays. Our hybrid analysis combines static analysis with dynamic analysis, usage of information from dynamic program states, and symbolic interpretation of program statements. Results of experiments with two popular approaches to partial order reduction show that usage of the hybrid analysis (1) eliminates many additional redundant thread choices and (2) improves the performance of software model checking on programs that use arrays.

## 1 Introduction

Systematic traversal of the program state space is a popular approach for detecting concurrency-related errors. It is used, for example, in software model checking [22], where the goal is to check the program behavior under all possible thread interleavings.

Each interleaving corresponds to a sequence of thread scheduling decisions and also to a particular sequence of actions performed by the program threads. We divide the actions into two sets: globally-relevant and thread-local. A *globally-relevant* action reads or modifies the global state shared by multiple threads. The set of globally-relevant actions contains accesses to fields of heap objects and array elements, and thread synchronization operations (e.g., acquisition of a lock). Other actions are *thread-local*.

Any non-trivial multithreaded program exhibits a huge number of possible interleavings, but many of them differ only in the order of thread-local actions. It is necessary to check just all the possible interleavings of globally-relevant actions, and to explore each of them just once. Techniques based on state space traversal use partial order reduction (POR) [5] to avoid redundant exploration of thread interleavings in order to mitigate state explosion.

```
class Writer extends Thread {
  public void run() {
    int[] buf = SharedData.buffer;
    buf[0] = x;
  }
}
```

```
class Reader extends Thread {
  public void run() {
    int[] buf = SharedData.buffer;
    v = buf[0];
  }
}
```

**Fig. 1.** Example: race condition involving an array element

The key idea behind POR is to consider non-deterministic thread scheduling choices only at globally-relevant actions, while avoiding redundant choices at thread-local actions. A lot of work has been done on POR in the context of software model checking (e.g., [3, 4, 6, 15]). All the existing approaches to POR have to conservatively over-approximate the set of globally-relevant actions in order to ensure coverage of all distinct thread interleavings. On the other hand, they also strive to be as precise as possible, because the number of thread interleavings explored redundantly during the state space traversal depends on the number of actions that are actually thread-local but were imprecisely identified as globally-relevant. For example, dynamic POR [4] uses dynamic analysis to identify (i) heap objects really accessed by multiple threads and (ii) actions performed upon such objects. Another technique [3] uses escape analysis to identify objects that are reachable from multiple threads. Some work has been done also on the combination of static analysis with dynamic analysis for precise identification of globally-relevant field accesses on shared heap objects [15, 16].

An important category of actions that may be globally-relevant are accesses to array objects stored in the heap. However, in the default configuration, POR algorithms in tools like Java Pathfinder [8] do not allow thread scheduling choices at actions that access individual array elements in order to avoid state explosion. For each access to an array element, they make a scheduling choice only at the preceding action that retrieves the array object from the heap (e.g., a field read access).

The problem with this approach to POR is that state space traversal can miss some concurrency errors. Consider the small Java-like program in Figure 1, where two threads access a shared array (buffer). Each thread retrieves a reference to the array object from the heap through a field read, stores the reference into a local variable buf, and then accesses the first element. The field read actions do not have to be synchronized at all, but there is a race condition that involves the array accesses. A verification tool cannot detect this race condition if it uses a POR algorithm with disabled thread choices at accesses to individual array elements. We found similar race conditions also in some of our benchmark programs — we discuss that in more detail at the end of Section 5.

Consequently, the state space traversal procedure with POR has to create thread scheduling choices at array element accesses in order to enable discovery of all such race conditions and other concurrency errors. The basic option for identifying globally-relevant accesses to array elements is to consider heap reachability [3]. When the given array object is not reachable from multiple threads, then every access to elements of the array is a thread-local action and no thread choice is necessary.

We propose a new hybrid analysis that soundly identifies array elements possibly accessed by multiple threads during the program execution. Results of the hybrid analysis can be used by POR to decide more precisely whether a given access to an array element is globally-relevant or thread-local. Then, thread choices at accesses to individual elements can be enabled without a high risk of state explosion. Although the state space size might increase in the worst case, it will stay in reasonable limits because POR avoids many redundant choices at thread-local accesses based on the hybrid analysis.

Our hybrid analysis combines static analysis with dynamic analysis and symbolic interpretation of program statements, and it also uses information from dynamic program states that is available on-the-fly during the state space traversal. We describe key concepts on the examples of multithreaded Java programs, but the analysis is applicable also to programs written in other languages, such as C# and C++. For simplicity of presentation, we consider only arrays with a single dimension in most of the paper and discuss support for multi-dimensional arrays at the end of Section 3.

An important feature of the hybrid analysis is compatibility with all memory models that we are aware of, including relaxed memory models such as JMM [10] and TSO [19]. The only requirements are that the underlying tool, which performs state space traversal, has to simulate the given memory model to a full extent and it must provide correct information about the dynamic program state, in particular taking into account delayed propagation of the effects of writes to shared variables among threads.

Experimental results provided in Section 5 show that our hybrid analysis helps to avoid many redundant thread choices during the state space traversal. It improves the precision and performance of existing approaches to POR on multithreaded programs that use arrays, and therefore enables more efficient detection of concurrency-related errors that involve array elements by software model checking.

In the next section we provide an overview of the whole approach. Then we discuss situations and code patterns where our hybrid analysis can eliminate a redundant thread choice (Section 3), and explain the analysis algorithm in more detail in Section 4. The rest of the paper contains evaluation, description of related work, and a brief summary.

## 2 Overview

Figure 2 shows the basic algorithm for depth-first state space traversal of multithreaded programs with POR. We assume that the program state space is constructed on-the-fly during traversal and that statements are interpreted using dynamic concrete execution. In addition, we consider only thread scheduling choices and ignore the data non-determinism in this paper. The symbol $s$ represents a program state, the symbol $ch$ represents a thread choice, and $T$ denotes a thread runnable in a particular state. Exploration starts from the initial state $s_0$ and the initial choice $ch_0$, where only the main thread is runnable. An atomic transition between two states corresponds to the execution of a sequence of instructions (program statements) that consists of a globally-relevant action, followed by any number of thread-local actions, and it ends with a thread choice. The POR algorithm creates a new thread choice just before execution of an action that it considers to be globally-relevant. All instructions in a transition are executed by the

```
visited = {}
exploreState($s_0$, $ch_0$)

procedure exploreState($s$, $ch$)
   if $s \in visited$ then return
   $visited$ = $visited \cup s$
   for $T \in$ getRunnableThreads($ch$) do
      $s'$ = executeTransition($s$, $T$)
      if isErrorState($s'$) then terminate
      $ch'$ = createThreadChoice($s'$, getRunnableThreads($s'$))
      exploreState($s'$, $ch'$)
   end for
end proc

procedure executeTransition($s$, $T$)
   $i$ = getNextInstruction($T$) // must be globally relevant
   while $i \neq$ null do // while not at the end of the thread
      $s$ = executeInstruction($s$, $i$)
      $i$ = getNextInstruction($T$)
      if isGloballyRelevant($s$, $i$, $T$) then break
   end while
   return $s$
end proc
```

**Fig. 2.** Basic algorithm for state space traversal with POR

same thread. Note that many popular tools, including Java Pathfinder [8], use a state space traversal procedure that follows this approach.

In this setting, the POR algorithm itself can use information only from (i) the current dynamic program state, (ii) the current state space path (execution history), and (iii) the already explored part of the state space to decide whether the action to be executed next is globally-relevant or thread-local, because it does not see ahead in program execution. A popular approach is to identify globally-relevant actions based on heap rechability in the current dynamic state [3]. This approach is safe but not very precise — a particular heap object (an array) may be reachable from multiple threads but really accessed only by a single thread during the program execution, or the individual threads may access different elements of a given array. The POR algorithm has to conservatively assume that each thread may in the future access every object reachable in the current state, and therefore many redundant thread choices are created during the state space traversal.

The proposed hybrid analysis determines more precise information about which array elements may be accessed in the future during the rest of program execution from the current state. We used the general principle introduced for field accesses in [15] and adapted it significantly for accesses to array elements. For each program point $p$ in each thread $T$, the analysis computes the set of array elements (over all array objects that may exist in the heap) possibly accessed by thread $T$ after the point $p$ on any execution path. In other words, the analysis provides over-approximate information about future

```
procedure isGloballyRelevant($s$, $i$, $T_c$)
   $a$ = getTargetArrayObject($i$)
   if $\neg$isArrayReachableFromMultipleThreads($s$, $a$) then return false
   for $T_o \in$ getOtherThreads($s$, $T_c$) do
      if existsFutureConflictingAccess($T_o$, $a$) then
         if possiblyEqualIndexes($s$, $T_c$, $T_o$, $a$) then return true
      end if
   end for
   return false // default
end proc
```

**Fig. 3.** Procedure that identifies globally-relevant accesses to array elements

behavior of $T$ after a specific code location. Array objects are identified by their static allocation sites and individual elements are identified by their symbolic indexes.

Our hybrid analysis has two phases: 1) static analysis that computes partial information, and 2) post-processing on-the-fly during the state space exploration (i.e., at the dynamic analysis time). Full results are generated in the second phase, when data provided by the static analysis are combined with specific information from dynamic program states, including the dynamic call stack of each thread and concrete values of some expressions used as array element indexes. The results are more precise than what would be possible to get with a reasonably expensive static analysis.

Here, in the rest of this section, we describe how the analysis results are used during the state space traversal to avoid redundant thread choices.

When the next action to be executed is an access to some array element, the POR algorithm has to decide whether to make a thread choice or not. Figure 3 captures the procedure at a high level of abstraction. The symbol $s$ represents the current dynamic state, $T_c$ is the currently scheduled thread, and $i$ is the next instruction of $T_c$.

First, the algorithm checks whether the target array object $a$ is reachable from multiple threads in the state $s$. If it is, then the procedure retrieves the results of the hybrid analysis for the current point of every thread $T_o$ other than $T_c$, and inspects the results to find whether some of the other threads may access the array $a$ in a conflicting way (read versus write) on any execution path that starts in $s$.

For the array accesses that may be performed by some other thread, the hybrid analysis inspects also symbolic indexes of array elements. More specifically, it compares (1) the concrete value of the array element index for the next access in $T_c$, which can be easily retrieved from the current dynamic state $s$, and (2) the symbolic index for each of the possible future conflicting accesses to $a$. Under some conditions, the concrete value of the array element index can be soundly determined also for a possible future access — the respective situations and code patterns are discussed in the next section.

A thread choice has to be created in the state $s$ only when some thread $T_o$ may possibly access the same element of $a$ as $T_c$, because otherwise the respective action of $T_c$ is thread-local. In particular, if every possible conflicting future access to the array $a$ in some other thread provably uses a different concrete value of an element index, then the POR algorithm does not have to make a thread choice.

## 3 Array Access Patterns

Here we discuss patterns of concurrent accesses to array elements, for which our hybrid analysis can eliminate a redundant thread choice, and also cases where it cannot eliminate a thread choice due to imprecision. Each code pattern involves two threads:

- the *active thread* whose next action is the array access in question (where a thread choice will be created or not depending on the analysis results), and
- the *conflicting thread*, which may access the same array elements as the active thread in the future on some execution path.

In all the patterns we assume that the array data is reachable from both threads. The various kinds of symbolic expressions that can be used as array element indexes are considered only for the conflicting thread, because for the active thread we can always get the actual concrete index value from the current dynamic program state.

**Constants.** The most basic pattern is the usage of an integer constant as the array element index. We show on this example how to interpret also the other patterns below.

| active thread | conflict thread |
|---|---|
| data[$e$] = x | y = data[1] |

In the code of the active thread, we use the symbol $e$ to denote the concrete value of the index expression. The symbolic index associated with the possible future access by the conflicting thread (i.e., the constant 1 in the code fragment above) is compared with the value $e$. If the values are different then a thread choice would be redundant at the array access in the active thread, because each thread accesses different elements.

**Local variables.** Another common case is when the symbolic index associated with the future access by the conflicting thread is a local variable $v$ of a method $m$. In order to decide soundly about making a new choice, the hybrid analysis can use the current value of $v$ (from the dynamic state) only if the following two conditions are satisfied.

1. The conflicting thread is executing the method $m$ in the current dynamic state $s$.
2. The local variable $v$ is not updated in the rest of the program execution starting from the state $s$.

We consider all methods on the current dynamic call stack of a given thread as currently executing. The concrete value obviously cannot be retrieved for local variables of methods that are not yet on the dynamic call stack of a respective thread. Note also that the local variable $v$ of $m$ may be updated in the future in two ways — either by assignment in the rest of the current execution of $m$, or by a future call of $m$ at any time during the program execution.

Consider the following example, where the variable $v$ is not updated after the access to data and the method run is not called again.

```
active thread        conflict thread
main():              run(args):
  ...                  v = f(args)
  data[e] = x          ...
  ...                  y = data[v]
```

The hybrid analysis can safely eliminate a thread choice only if the concrete dynamic value of $v$ is different from $e$.

A typical situation where the variable $v$ may be updated later during the execution of $m$ is shown in the next example. Here, $v$ is also a control variable of the loop.

```
active thread          conflict thread
main():                run(args):
  ...                    for (v = 0; v < 10; v++)
  data[e] = x              y = data[v]
```

The hybrid analysis cannot determine whether another iteration of the loop might be executed or not, and therefore a future update of $v$ is always possible in this case.

We have to consider also future calls of the method $m$ because every local variable of $m$ has to be initialized (i.e., updated) before it can be used as array index. Although each execution of $m$ has its own instances of local variables, the symbolic name $v$ is common to all of the executions. Therefore, an update of $v$ may occur between the current state and the relevant array access in a future execution of $m$.

**Object fields.** When the symbolic index contains a field access path *fp*, the analysis can use the current dynamic value of *fp* only if the following conditions are satisfied.

1. In the case of instance fields, the access path must contain the local variable this associated with one of the currently executing methods of the conflicting thread.
2. No field in the access path *fp* is updated in the future during the rest of program execution starting from the current dynamic state $s$.

Then, the dynamic value of *fp* can be used to compute the concrete value of the array index expression in the conflicting thread. If the result is not equal to the value of the index expression $e$ used by the active thread, then both threads will always access different elements of the shared array at the respective code locations, and thus the POR algorithm does not have to create a new thread choice.

**Multi-dimensional arrays.** Our hybrid analysis supports multi-dimensional arrays but only with a limited precision. Element indexes are inspected and compared only for the innermost dimension, using the same approach as for single-dimensional arrays. Index expressions for outer dimensions are completely ignored by the hybrid analysis, which therefore assumes (i) that concurrent threads may use the same index values and (ii) that any two elements of an outer array may be aliased. A possible choice can be safely eliminated only when both threads use provably different values of element indexes for the innermost dimension. This case is illustrated by the following example, where $e_1$ might be equal to $e_2$.

```
active thread          conflict thread
data[e1][0] = x        y = data[e2][1]
```

On the other hand, a choice must be preserved when both threads may use the same index value for the innermost dimension, such as $e_1$ and $e_2$ in the example below, even if different values (e.g., 0 and 1) are used at some outer dimension. The expressions data[0] and data[1] may point to the same innermost array because of aliasing.

| active thread | conflict thread |
|---|---|
| data[0][$e_1$] = x | y = data[1][$e_2$] |

Note also that we have to analyze possible read-write conflicts only for the innermost dimension, because only read-read conflicts may happen at outer dimensions and they do not require thread choices.

## 4 Hybrid Analysis

The hybrid analysis computes all the information necessary to decide whether a thread choice must be created — in particular, for each of the scenarios described in the previous section. We designed the analysis in a modular way. Each component provides information about one of the following: (1) accesses to array objects, (2) future accesses to specific array elements, (3) symbolic values of element indexes, (4) local variables possibly updated in the future, (5) updated object fields, and (6) future method calls.

First we describe the general principles and then we provide additional details about the individual components. Every component that is an inter-procedural analysis has two phases: static and dynamic. Both phases are designed and executed using an approach that was proposed in [15]. The static analysis runs first, and then follows the state space traversal with dynamic analysis. Results of the static analysis (phase 1) are combined with information taken from the dynamic program state (phase 2) on-the-fly during the state space traversal, i.e. at the dynamic analysis time.

The static phase involves a backward flow-sensitive and context-insensitive analysis that is performed over the full inter-procedural control flow graph (ICFG) of a given thread. For each program point $p$ in the thread $T$, it provides only information about the behavior of $T$ between the point $p$ and the return from the method $m$ containing $p$. Note that the result for $p$ in $m$ covers also methods called from $m$ (transitively).

Full results are computed at the dynamic analysis time based on the knowledge of the dynamic call stack of each thread, which is a part of the dynamic program state. The dynamic call stack of a given thread specifies a sequence $p_0, p_1, \ldots, p_N$ of program points, where $p_0$ is the current program counter of the thread (in the top stack frame), and $p_i$ is the point from which execution of the thread would continue after return from the method associated with the previous stack frame. When the hybrid analysis is queried for data about the current point $p$ of some thread $T$, it takes the data computed by the static analysis phase for each point $p_i, i = 0, \ldots, N$ on the dynamic call stack of $T$, where $p = p_0$, and merges them all to get the precise and complete results for $p$.

The complete results for a program point $p$ in thread $T$ cover the future behavior of $T$ after the point $p$ (until the end of $T$), and also the behavior of all child threads of $T$ started after $p$. Here, a child thread of $T$ is another thread created and started by $T$.

Note also that the complete results of the hybrid analysis are fully context-sensitive for the following two reasons: (1) they reflect the current dynamic calling context of $p$ in $T$, i.e., the current program counter in each method on the dynamic call stack of $T$, and (2) they precisely match calls with returns. Only those method call and return edges in the ICFG that can be actually taken during the concrete program execution are considered by the hybrid analysis.

**Accesses to array objects.** This component of the hybrid analysis identifies all arrays possibly accessed in the future by a given thread. More specifically, for each program point $p$ in each thread $T$, it computes the set of all array objects that may be accessed on some execution path after $p$. Static allocation sites are used to represent the actual array objects also here. The analysis considers read and write accesses separately in order to enable precise detection of read-write conflicts. It is an inter-procedural analysis, which therefore has two phases — static and dynamic — in our approach.

| Instruction | Transfer function |
|---|---|
| | after[$\ell$] = $\bigcup_{\ell' \in \mathrm{succ}(\ell)}$ before[$\ell'$] |
| $\ell$: v = a[i] | before[$\ell$] = after[$\ell$] $\cup$ $\{r$ a$\}$ |
| $\ell$: a[i] = v | before[$\ell$] = after[$\ell$] $\cup$ $\{w$ a$\}$ |
| $\ell$: return | before[$\ell$] = $\emptyset$ |
| $\ell$: call M | before[$\ell$] = before[M.entry] $\cup$ after[$\ell$] |
| $\ell$: other instr. | before[$\ell$] = after[$\ell$] |

**Fig. 4.** Transfer functions for the static phase of the array objects analysis

Figure 4 shows transfer functions for the static phase. When the analysis encounters a read or write access to an array a, it adds the target array object into the set of data-flow facts. The transfer functions for the call and return statements are defined in this way to ensure that the result of the static phase for a point $p$ in a method $m$ covers only the execution between $p$ and return from $m$. The merge operator is a set union.

**Array elements.** Possible future accesses to individual array elements are identified using an analysis component that works in a very similar way to the one for array objects. This analysis computes, for each program point $p$ in each thread, the set of all possible accesses to array elements that may occur on some execution path after $p$. It gathers the following information about each access: a target array object (allocation site), method signature, and instruction index (bytecode position). Knowledge of the method signature and bytecode position is used by the next component to associate each particular access with symbolic values of array element indexes.

**Symbolic indexes.** This component performs symbolic interpretation of the code in each method to determine symbolic expressions that represent indexes of array elements. A symbolic expression may include local variables, field access paths, nested accesses to array elements, numeric constants, and arithmetic operators.

When processing the code of a method, the analysis maintains a stack of symbolic expressions, which models the concrete dynamic stack containing local variables and operands. The symbolic stack is updated during interpretation to capture the effects of executed program statements. For each statement, all its operands are removed from the stack and then the result is pushed onto it.

The following example illustrates how the symbolic value of an element index is computed for a particular array access. We consider the statement v = a[o.f+2].

| instructions | symbolic stack |
|---|---|
| 1: load a | [a] |
| 2: load o | [a, o] |
| 3: getfield f | [a, o.f] |
| 4: const 2 | [a, o.f, 2] |
| 5: add | [a, o.f+2] |
| 6: arrayload | [$e$] |
| 7: store v | [] |

The left column contains a sequence of instructions that corresponds to the statement, and the right column shows the content of the symbolic stack after each instruction. At line 5, the top value on the stack represents the symbolic array element index.

**Updated local variables.** The sets of possibly updated local variables are computed by an intra-procedural static analysis of each method. For each point $p$ in method $m$, the analysis identifies all future write accesses to local variables of $m$ that may occur on some execution path in $m$. Note that this component of the whole hybrid analysis does not use any information available in the dynamic program state.

Transfer function for the store operation just records the index (name) of the target local variable. For all other statements, the transfer function is identity.

**Updated fields.** We use the field access analysis proposed in [15] to find all fields that may be updated on some execution path in thread $T$ after the point $p$. The analysis is fully inter-procedural and combines the static phase with information taken from the dynamic program state.

However, the field access analysis alone is not sufficient for the following reason: a symbolic value of an array element index may refer to a field of a heap object that does not exist yet in the current dynamic state. It is therefore necessary to consider also possible future allocations of heap objects of the respective class (type). The current dynamic value of a given field may be safely used by the hybrid analysis and POR, as discussed in Section 3, only when the following two conditions hold.

1. The field is provably not updated in the future according to the field access analysis.
2. No heap object of the given type may be allocated later during the program execution starting from the current dynamic state.

We use a simple analysis to find allocation sites at which some dynamic heap object may be possibly allocated in the future (on some execution path starting in $p$).

Although the conditions are quite restrictive, we believe that they will be satisfied in many cases in practice. Based on manual inspection of the source code of our benchmark programs (listed in Section 5), we found that array index expressions quite often refer to fields of heap objects that are allocated early during the program execution. The concrete dynamic value of an object field can be safely used in such cases, helping to eliminate many redundant thread choices.

**Method calls.** The last component of the hybrid analysis identifies methods that may be called in the future after the current state. It is an inter-procedural analysis that represents methods by their signatures. The transfer function for the call statement adds into the set of facts every method that is a possible target according to the call graph.

## 5 Evaluation

We implemented the proposed hybrid analysis in Java Pathfinder (JPF) [8], which is a framework for state space traversal of multithreaded Java programs. JPF uses on-the-fly state space construction, depth-first search, and concrete execution of Java bytecode instructions. In order to support decisions about thread choices based on the results of our hybrid analysis, we created a non-standard interpreter of Java bytecode instructions for array access. We used the WALA library [23] for static analysis and JPF API to retrieve information from the dynamic program state. Symbolic interpretation of Java bytecode, which collects symbolic expressions that represent indexes of array elements, is performed by a custom engine that we also built using WALA.

Our prototype implementation, together with the experimental setup and benchmark programs described below, is publicly available at `http://d3s.mff.cuni.cz/projects/formal_methods/jpf-static/vmcai16.html`.

**Benchmarks.** We evaluated the hybrid analysis on 11 multithreaded Java programs from widely known benchmark suites (Java Grande Forum [7], CTC [2], pjbench [13]), our previous work, and existing studies by other researchers [20]. Table 1 shows the list of benchmark programs and their quantitative characteristics — the total number of source code lines (Java LoC) and the maximal number of concurrently running threads. All the benchmark programs that we use contain array objects reachable from multiple threads and many accesses to array elements in their source code.

**Table 1.** Benchmark programs

| Benchmark | Java LoC | Threads |
|---|---|---|
| CRE Demo | 1,300 | 2 |
| Daisy | 800 | 2 |
| Crypt | 300 | 2 |
| Elevator | 300 | 3 |
| Simple JBB | 2700 | 2 |
| Alarm Clock | 200 | 3 |
| Prod-Cons | 130 | 2 |
| Rep Workers | 400 | 2 |
| SOR | 160 | 2 |
| TSP | 420 | 2 |
| QSort MT | 290 | 2 |

For selected benchmarks, we provide a more detailed characteristic that is relevant for the discussion of experimental results later in this section. The benchmark program Crypt contains three shared arrays, but each thread accesses different elements of the arrays, and therefore all possible thread choices at the accesses to arrays would be redundant. In the case of CRE Demo and Daisy, each array object used directly in the application source code is reachable only from a single thread, which means that accesses to arrays are thread-local, but the programs involve shared collections (e.g., Vector and HashSet) that use arrays internally.

**Table 2.** Configurations of POR

| Description | Short name |
|---|---|
| heap reachability without thread choices at bytecode instructions for array element access | HR + no array ch |
| heap reachability with thread choices enabled at bytecode instructions for array element access | HR + all array ch |
| heap reachability with field access analysis and enabled thread choices at array element accesses | HR + fields + all array ch |
| heap reachability with field access analysis, thread choices at array accesses, and hybrid analysis | HR + fields + hybrid |
| dynamic POR without thread choices at bytecode instructions for array element access | DPOR + no array ch |
| dynamic POR with thread choices enabled at bytecode instructions for array access | DPOR + enabled array ch |
| dynamic POR with field access analysis and enabled choices at array element accesses | DPOR + fields + enabled array ch |
| dynamic POR with field access analysis, enabled choices at array accesses, and hybrid analysis | DPOR + fields + hybrid |

**Experiments.** The goal of our experimental evaluation was to find how many redundant thread choices the hybrid analysis really eliminates during the state space traversal, and how much it improves performance and scalability of different approaches to partial order reduction in the context of software model checking. We performed experiments with the hybrid analysis for shared array elements proposed in this paper, the hybrid field access analysis [15], the POR algorithm based on heap reachability, and our implementation of the dynamic POR algorithm described in [4]. For the purpose of our experiments, we have implemented also the dynamic POR algorithm in JPF and combined it with state matching.

Table 2 shows all configurations of POR that we considered in our experiments. For each configuration, it provides a brief description and a short name used in tables with results. Note that we say ”array access” instead of ”array element access” in some table rows, but with the same intentional meaning, as the table would be too large otherwise.

For each configuration and benchmark program, i.e. for every experiment, we report the following metrics: (1) the total number of thread choices created by JPF at all kinds of bytecode instructions during the state space traversal, and (2) the total running time of JPF combined with all phases of the hybrid analysis. The number of thread choices shows precision, while the running time indicates performance.

In the first set of experiments, we configured JPF to traverse the whole state space of each benchmark program — we had to disable reporting of errors because otherwise JPF would stop upon reaching an error state. We used the time limit of 8 hours and memory limit of 20 GB. The symbol ”-”, when present in some cell of a table with results, indicates that JPF run out of the limit for a given configuration and benchmark.

**Discussion.** The results in Table 3 and Table 4 show that usage of our hybrid analysis together with POR in general reduces the number of thread choices and improves the

**Table 3.** Experimental results: POR algorithm based on heap reachability

| | HR + no array ch | | HR + all array ch | | HR + fields + all array ch | | HR + fields + hybrid | |
|---|---|---|---|---|---|---|---|---|
| benchmark | choices | time | choices | time | choices | time | choices | time |
| CRE Demo | 30942 | 51 s | 103016 | 174 s | 41146 | 79 s | 29737 | 69 s |
| Daisy | 28436002 | 17954 s | 32347254 | 18357 s | 8453587 | 5972 s | 8453587 | 6765 s |
| Crypt | 4993 | 3 s | 682273 | 238 s | 674041 | 237 s | 46105 | 29 s |
| Elevator | 10167560 | 7656 s | 23709139 | 18339 s | 9980240 | 7426 s | 4748393 | 3872 s |
| Simple JBB | 575519 | 1779 s | 836889 | 2583 s | 515312 | 1722 s | 344428 | 1269 s |
| Alarm Clock | 531463 | 432 s | 742027 | 601 s | 344791 | 285 s | 344791 | 289 s |
| Prod-Cons | 6410 | 4 s | 6934 | 4 s | 2792 | 4 s | 2792 | 6 s |
| Rep Workers | 9810966 | 6860 s | 9983423 | 7045 s | 1714694 | 1169 s | 1714694 | 1275 s |
| SOR | 222129 | 123 s | 1565386 | 882 s | 772837 | 451 s | 273693 | 160 s |
| TSP | 35273 | 572 s | 47475 | 779 s | 15386 | 257 s | 13258 | 221 s |

**Table 4.** Experimental results: dynamic POR

| | DPOR + no array ch | | DPOR + enabled array ch | | DPOR + fields + enabled array ch | | DPOR + fields + hybrid | |
|---|---|---|---|---|---|---|---|---|
| benchmark | choices | time | choices | time | choices | time | choices | time |
| CRE Demo | 2015 | 11 s | 2232 | 20 s | 2207 | 18 s | 2197 | 22 s |
| Daisy | - | - | - | - | - | - | - | - |
| Crypt | 9 | 1 s | 9 | 1 s | 9 | 3 s | 9 | 5 s |
| Elevator | 414345 | 913 s | 501732 | 1371 s | 408192 | 886 s | 342817 | 648 s |
| Simple JBB | 602 | 30 s | 608 | 36 s | 608 | 36 s | 608 | 38 s |
| Alarm Clock | 102076 | 147 s | 155974 | 227 s | 103964 | 123 s | 103964 | 125 s |
| Prod-Cons | 429 | 1 s | 444 | 1 s | 407 | 3 s | 407 | 4 s |
| Rep Workers | - | - | - | - | - | - | - | - |
| SOR | 135 | 2 s | 40594 | 208 s | 26503 | 135 s | 19819 | 71 s |
| TSP | 101 | 67 s | 101 | 94 s | 97 | 66 s | 97 | 58 s |

running time for both POR algorithms that we considered. In the next few paragraphs, we discuss the results for individual benchmark programs in more detail and highlight important observations.

For many configurations and benchmark programs, the total number of thread choices created during the state space traversal is much higher when choices are enabled at accesses to array elements. This is evident from the values in columns ”HR + no array ch” and ”HR + all array ch” (Table 3), respectively in the columns ”DPOR + no array ch” and ”DPOR + enabled array ch” (Table 4). We observed an extreme increase of the number of thread choices in two cases — by the factor of 137 for the Crypt benchmark with POR based on heap reachability, and by the factor of 300 for the SOR benchmark when using the dynamic POR. On the other hand, there is a negligible increase for Prod-Cons and Rep Workers, and no increase for the benchmarks Crypt, Simple JBB, and TSP when using the dynamic POR.

Data in tables 3 and 4 also indicate how many redundant choices were eliminated by the hybrid analysis, and how much it improved the performance and scalability of state space traversal. The result for a particular benchmark and POR based on heap reachability corresponds to the difference between values in the columns "HR + fields + all array ch" and "HR + fields + hybrid" of Table 3. Similarly, in the case of dynamic POR one has to consider values in the columns "DPOR + fields + enabled array ch" and "DPOR + fields + hybrid" of Table 4. We observe that our hybrid analysis eliminates many redundant thread choices at array accesses for 6 out of 10 benchmarks, namely the following: CRE Demo, Crypt, Elevator, Simple JBB, SOR, and TSP. In the case of four benchmark programs — CRE Demo, Crypt, Simple JBB, and TSP — the hybrid analysis significantly reduced the total number of thread choices only when it is combined with the POR based on heap reachability. The factor of reduction in the number of thread choices lies in the range from 1.16 (for TSP and POR based on heap reachability) up to 14.62 (Crypt and again POR based on heap reachability).

Our results for the benchmarks Alarm Clock, Daisy, Prod-Cons, and Rep Workers indicate that all redundant thread choices were eliminated by the field access analysis. For example, by manual inspection of the source code of Prod-Cons we have found that all accesses to array elements are properly synchronized, and therefore no thread choices are created at their execution.

Here we compare dynamic POR with the POR algorithm based on heap reachability. A well-known fact is that dynamic POR is very precise and creates much less thread choices [12, 16]. For example, it correctly identifies that all accesses to array elements in the Crypt benchmark are thread-local actions. It analyzes small programs very fast (in few seconds) — see, e.g., the data for Crypt and Prod-Cons in Table 4 — but it has a significantly higher running time and memory consumption for some of the more complex benchmark programs. Specifically, our implementation of dynamic POR run out of memory for Daisy and Rep Workers. Even though dynamic POR itself avoids many redundant thread choices, usage of our hybrid analysis can still improve precision and also the running time — data for the benchmarks Elevator and TSP highlight this case. We discuss reasons for the observed behavior of dynamic POR in Section 6.

The cost of the static phase of the hybrid analysis is negligible, as it runs for few seconds at most. This is apparent especially from the data for benchmarks Crypt and Prod-Cons, where a majority of the total running time is consumed by static analysis. The cost of the dynamic analysis phase, which is performed on-the-fly during the state space traversal, depends heavily on the number of executed actions (program statements) for which JPF queries the hybrid analysis. For every such action, the hybrid analysis must decide whether it is globally-relevant or not. Results for the benchmarks Daisy and Rep Workers in the right-most columns of Table 3 show that the cost of the dynamic analysis phase may be significant if JPF performs many queries — in general, one query for each thread choice created in the configuration "HR + all array ch". Note that for Daisy and Rep Workers, the hybrid analysis for shared array elements does not eliminate any additional thread choices when compared to the configuration "HR + fields + all array ch" that involves just the field access analysis, and therefore hybrid analysis is responsible for the increase of running time. However, despite the relatively

**Table 5.** Experimental results: search for concurrency errors

| | HR + all array ch | | HR + fields + hybrid | | DPOR + enabled array ch | | DPOR + fields + hybrid | |
|---|---|---|---|---|---|---|---|---|
| benchmark | choices | time | choices | time | choices | time | choices | time |
| Daisy | 253336 | 143 s | 173441 | 151 s | - | - | - | - |
| Elevator | 31169 | 14 s | 8494 | 9 s | 178748 | 529 s | 80486 | 165 s |
| Alarm Clock | 428 | 1 s | 161 | 4 s | 179 | 1 s | 71 | 4 s |
| Prod-Cons | 12073 | 17 s | 3030 | 8 s | 1114 | 3 s | 1101 | 6 s |
| Rep Workers | 6708 | 5 s | 1545 | 6 s | 4527 | 6 s | 1699 | 6 s |
| QSort MT | 2635 | 2 s | 1428 | 4 s | - | - | - | - |

high cost, the speedup of JPF achieved due to the elimination of many redundant thread choices makes the proposed hybrid analysis practically useful for many programs.

We also performed experiments with several benchmark programs to find whether our hybrid analysis improves the speed of error detection. For that purpose, we had to manually inject concurrency errors into some of the programs. Table 5 contains results for selected configurations. We have considered both the POR based on heap reachability and the dynamic POR, each with enabled thread choices at accesses to array elements, and then with or without the hybrid analysis.

Usage of the hybrid analysis (i) helped to reduce the number of thread choices created before reaching an error state for all the benchmarks, and (ii) also helped to improve performance by a factor greater than 2 for the benchmark Elevator (with dynamic POR) and for the benchmark Prod-Cons (just with POR based on heap reachability). When the error is detected very quickly in the baseline configurations, then the cost of the hybrid analysis is responsible for slight increase of the total running time — see, e.g., the data for Prod-Cons and the dynamic POR. Interestingly, dynamic POR is much slower than JPF with heap reachability for Elevator, and it did not find any error for Daisy and QSort MT.

Regarding the actual errors, JPF reported a race condition involving a particular array element only for the benchmarks Elevator and QSortMT. They could not be detected if threads choices were disabled at array accesses. Other benchmarks contain also race conditions that involve field accesses, and the corresponding error states are discovered by JPF sooner than the possible races at array element accesses.

## 6 Related Work

We discuss selected approaches to partial order reduction, which are used in software model checking, and also few static analysis-based techniques that can be used to identify shared array elements.

Dwyer et al. [3] proposed to use a heap reachability information that is computed by a static or dynamic escape analysis. If a given heap object is reachable from multiple threads, then all operations upon the object have to be marked as globally-relevant, independently of which threads may really access the object. The dynamic escape analysis is performed on-the-fly during the state space traversal, and therefore it can use knowledge of the dynamic program state to give more precise results than the static escape analysis. An important limitation of this approach is that it works at the granularity of whole objects and arrays. For example, if an array object is reachable from two threads but every element is accessed only by a single thread, then all the accesses are still imprecisely considered as globally-relevant even though they are actually thread-local. Our hybrid analysis is more precise because (i) for each thread $T$ it computes the set of array objects accessed by $T$ and (ii) it can distinguish individual array elements.

The dynamic POR algorithm that was proposed by Flanagan and Godefroid [4] is very precise. It explores each dynamic execution path of the given program separately, and for each path determines the set of array elements that were truly accessed by multiple threads on the path. The main advantage of dynamic POR is that it can distinguish between individual dynamic heap objects, unlike the static pointer analysis whose results we also use in our hybrid analysis. More specifically, dynamic POR can precisely identify every shared memory location, e.g. a dynamic array object with the concrete value of an element index, and creates thread choices retroactively at accesses to such locations. Every added choice corresponds to a new thread interleaving that must be explored later. A limitation of this dynamic POR algorithm performance-wise is that it performs redundant computation because (i) it has to execute each dynamic path until the end state and (ii) it has to track all accesses to object fields and array elements. A given path has to be fully analyzed even if it does not contribute any new thread choices, and this can negatively impact performance in the case of long execution paths. We believe that the redundant computation is the main reason for the surprisingly long running times of the dynamic POR that we reported in Section 5. The need to keep track of many accesses to fields and array elements is the main reason for high memory consumption that we observed with our implementation. Our hybrid analysis improves the performance of dynamic POR, when they are combined together, by identifying thread-local accesses to array elements that the dynamic POR does not have to track. In Section 5, we also reported that the combination of dynamic POR with hybrid analysis improves precision for some benchmarks. The standalone dynamic POR does not consider reachability of heap objects by individual threads, and therefore it may still create some redundant thread choices. More specifically, when processing two instructions $i$ and $j$ that access the same element on the same array object $a$, the dynamic POR does not check whether the array $a$ was reachable by thread $T_j$ (which executes $j$) at the time of the access by instruction $i$.

Other recent approaches to partial order reduction include, for example, the Cartesian POR [6] and the combination of dynamic POR with state matching [24], which address some limitations of the original approach to dynamic POR. Unnecessary thread choices can be eliminated from the state space also by preemption sealing [1], which allows the user to enable thread scheduler only inside specific program modules.

Many techniques that improve the error detection performance of software model checking are based on bounding the number of explored thread interleavings. See the recent experimental study by Thomson et al. [20] for a comprehensive overview. Techniques from this group are orthogonal to our proposed approach, because they limit the search to a particular region of the state space, while preserving all thread choices.

Another group of related techniques includes static and dynamic analyses that can determine whether a given heap object (field) is stationary according to the definition in [21]. Such objects and fields may be updated only during initialization, while they are reachable only from a single thread. Once the object becomes shared, it can be just read in the rest of the program execution. The analyses for detecting stationary objects [9] and fields [21] could be extended towards array elements, and then used to compute a subset of the information that is produced by our hybrid analysis. No thread choice would have to be created at accesses to a stationary array element during the state space traversal, because there cannot occur any conflicting pair of read-write accesses to such an element from different threads.

Shape analysis together with pointer analysis can be also used to identify heap objects and array elements possibly shared between multiple threads. For example, the analysis proposed by Sagiv et al. [17] determines the set of memory locations that are directly reachable from two or more pointer variables. Client analyses can derive various higher-level sharing properties from this information. Our hybrid analysis is different especially in that it determines only whether an array element is possibly accessed by multiple threads — it does not compute the heap reachability information and does not perform any kind of shape analysis.

Marron et al. [11] proposed an analysis that determines whether elements of a given array may be aliased. In that case, threads accessing the respective different array elements would in fact access the same object. Our hybrid analysis does not compute aliasing information of such kind — rather it answers the question whether multiple threads can access the same array element (i.e., whether threads can use the same index when accessing the array), independently of possible aliasing between array elements.

## 7 Conclusion

Our motivation for this work was to optimize the existing popular approaches to partial order reduction in the context of programs that heavily use arrays. We proposed a hybrid static-dynamic analysis that identifies array elements that are possibly accessed by multiple threads during the program execution. Results of experiments that we performed on several benchmark programs show that combination of the hybrid analysis with POR improves performance and scalability of state space traversal. The main benefit of the hybrid analysis is that, in tools like Java Pathfinder, thread choices can be enabled at globally-relevant accesses to individual arrays elements, which is a necessary step for detecting specific race conditions and other kinds of concurrency errors, all that without a high risk of state explosion and at a reasonable cost in terms of the running time.

In the future, we plan to integrate the proposed hybrid analysis for array elements with the may-happen-before analysis [14]. Another possible line of future research work is to design some variant of the dynamic determinacy analysis [18] for multithreaded programs, and use it to improve the precision of our hybrid analyses.

**Acknowledgments.** This work was partially supported by the Grant Agency of the Czech Republic project 13-12121P.

## References

1. T. Ball, S. Burckhardt, K.E. Coons, M. Musuvathi, and S. Qadeer. Preemption Sealing for Efficient Concurrency Testing. In Proceedings of TACAS 2010, LNCS, vol. 6015.
2. Concurrency Tool Comparison repository, `https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison`
3. 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.
4. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Proceedings of POPL 2005, ACM.
5. P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, 1996.
6. G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian Partial-Order Reduction. In Proceedings of SPIN 2007, LNCS, vol. 4595.
7. The Java Grande Forum Benchmark Suite, `https://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html`
8. Java Pathfinder: a system for verification of Java programs, `http://babelfish.arc.nasa.gov/trac/jpf/`
9. D. Li, W. Srisa-an, and M.B. Dwyer. SOS: Saving Time in Dynamic Race Detection with Stationary Analysis. In Proceedings of OOPSLA 2011, ACM.
10. J. Manson, W. Pugh, and S.V. Adve. The Java Memory Model. In Proceedings of POPL 2005, ACM.
11. M. Marron, M. Mendez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing Analysis of Arrays, Collections, and Recursive Structures. In Proceedings of PASTE 2008, ACM.
12. E. Noonan, E. Mercer, and N. Rungta. Vector-Clock Based Partial Order Reduction for JPF. ACM SIGSOFT Software Engineering Notes, 39(1), 2014.
13. pjbench: Parallel Java Benchmarks, `https://bitbucket.org/pag-lab/pjbench`
14. P. Parizek and P. Jancik. Approximating Happens-Before Order: Interplay between Static Analysis and State Space Traversal. In Proceedings of SPIN 2014, ACM.
15. P. Parizek and O. Lhotak. Identifying Future Field Accesses in Exhaustive State Space Traversal. In Proceedings of ASE 2011, IEEE CS.
16. P. Parizek and O. Lhotak. Model Checking of Concurrent Programs with Static Analysis of Field Accesses. Science of Computer Programming, 98(part 4), 2015.
17. M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. ACM Transactions on Programming Languages and Systems, 24(3), 2002.
18. M. Schaefer, M. Sridharan, J. Dolby, and F. Tip. Dynamic Determinacy Analysis. In Proceedings of PLDI 2013, ACM.
19. P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. Myreen. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. Comm. of the ACM, 53(7), 2010.
20. P. Thomson, A. Donaldson, and A. Betts. Concurrency Testing Using Schedule Bounding: An Empirical Study. In Proceedings of PPoPP 2014, ACM.
21. C. Unkel and M.S. Lam. Automatic Inference of Stationary Fields: A Generalization of Java's Final Fields. In Proceedings of POPL 2008, ACM.
22. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. Automated Software Engineering, 10(2), 2003.
23. WALA: T.J. Watson Libraries for Analysis, `http://wala.sourceforge.net/`
24. Y. Yang, X. Chen, G. Gopalakrishnan, and R.M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In Proceedings of SPIN 2008, LNCS, vol. 5156.