Model Checking of Concurrent Programs with Static Analysis of Field Accesses

Pavel Parízek[1]

Department of Distributed and Dependable Systems, Charles University in Prague

Ondřej Lhoták

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

Abstract

Systematic exploration of all possible thread interleavings is a popular approach to detect errors in multi-threaded programs. A common strategy is to use a partial order reduction technique and perform a non-deterministic thread scheduling choice only when the next instruction to be executed may possibly read or modify the global state. However, some verification frameworks and software model checkers, including Java Pathfinder (JPF), construct the program state space on-the-fly during traversal. The partial order reduction technique built into such a tool can use only the information available in the current state to determine whether the execution of a given instruction is globally-relevant. For example, the reduction technique has to make a thread choice at every field access on a heap object reachable from multiple threads, even in the case of fields that are really accessed only by a single thread during program execution, because it does not have any information about what may happen in the future after a particular state. These conservative decisions cause many redundant thread choices.

We propose static analyses that identify globally-relevant field accesses more precisely. For each program state, the analyses give information about field accesses that may occur in the future after the given state. The state space traversal algorithm can use this information to soundly avoid creating unnecessary thread choices, and thus to reduce the number of thread interleavings that must be explored to cover all distinct behaviors of the given program. We implemented the proposed analyses using WALA and integrated them with JPF. Results of experiments on several Java programs show that the static analyses greatly improve the performance and scalability of JPF. In particular, it is now possible to check more complex programs than before in reasonable time.

Keywords: software model checking, exhaustive state space traversal, state explosion, concurrency, field accesses, immutability, static analysis, Java Pathfinder

Email addresses: parizek@d3s.mff.cuni.cz (Pavel Parízek), olhotak@uwaterloo.ca (Ondřej Lhoták)

[1]Part of this work was done while the author was at the University of Waterloo.

1. Introduction

Automated techniques for detecting concurrency errors in multi-threaded programs are becoming very important with the proliferation of multi-core architectures. The biggest challenge is to design techniques that have good performance and scale to large programs, so that they are practically useful.

One group of techniques is based on systematic state space traversal. Their goal is to explore all reachable states of the program under all possible thread interleavings to find property violations (errors). The number of possible thread interleavings is huge even for small programs with a few threads, but many optimizations have been developed. The most prominent of them is partial order reduction (POR) [16].

The key idea of POR is to create a thread scheduling choice (thread context switch) at a point in the program execution only when the instruction to be executed next may read or modify the global state shared by multiple threads. We call the execution of the instruction *globally-relevant* in this case, and *thread-local* otherwise. We distinguish between a static instruction in the program code and its execution in a particular program state. A single static instruction can have both globally-relevant and thread-local executions in different program states. Globally-relevant instruction executions represent interaction between threads. Instructions whose execution may be globally-relevant include thread synchronization operations and accesses to objects shared between multiple threads. The effects of a thread-local instruction execution are invisible to and independent of the concurrent behavior of other threads. Thread interleavings that differ only in the scheduling of thread-local instruction executions yield the same observable behavior. Therefore, no thread scheduling choice is needed when the next instruction execution on the current thread is thread-local. To cover every possible observable behavior of the given program, it is sufficient to explore all thread interleavings that differ in the order of globally-relevant instruction executions.

More specifically, the state space traversal with partial order reduction works as follows. At any point in the program's execution, one thread is running, and some subset of other threads are ready to run. Let i be the next instruction to be executed on the currently running thread. In order to explore all thread interleavings, a verification tool would have to explore the interleaving where i is executed next, and also the interleavings in which actions of other threads occur before i. Typically, the tool starts with the state space path in which the current thread continues and executes i, and later backtracks to the current state and switches to each of the other ready threads in turn to allow them to run before executing i. However, suppose that the execution of i is thread-local. In that case, the program behavior is independent of whether i is executed before or after other threads have been allowed to run, and therefore it is sufficient to execute i first and not consider switching to other threads at this point. As a result, the sequences of executed instructions can be divided into blocks such that execution of all but the first instruction in each block is thread-local. The state space is then explored by treating each block as an atomic step. That is, all interleavings of such blocks are explored, rather than interleavings at the granularity of individual instructions. This way, if the verification tool runs to completion, it is guaranteed that the behaviors of all interleavings of executed instructions have been explored, because the observable program behavior depends only on thread switches immediately before the execution of the first instruction of each block.

Nevertheless, despite partial order reduction and many other optimizations, sys-

tematic checking of all thread interleavings with distinct behaviors is still very time-consuming and is thus currently applicable only to relatively small programs. A specific challenge is to determine, for each instruction used in the program code, which of its executions are globally-relevant and which are thread-local. There exist both static and dynamic approaches (see, for example, [13, 15, 16]) that provide conservative approximations of different precision. The efficiency of state space traversal depends crucially on precisely identifying as many thread-local instruction executions as possible, so that the blocks are as long as possible. Performance suffers if the blocks are shorter than necessary, because shorter blocks imply more thread scheduling choices and therefore exponentially more thread interleavings to explore.

Two well-known tools that implement state space traversal and some form of POR are Java Pathfinder [20] and CHESS [29]. In this paper we focus on the partial order reduction technique used in Java Pathfinder (JPF). We describe partial order reduction, some other important concepts, and especially the proposed techniques in the context of model checking of multi-threaded Java programs with JPF.

JPF constructs the program state space on-the-fly, i.e., it is not created in advance before the state space traversal, and therefore the partial order reduction technique in JPF can use only information in the current state to determine whether the execution of a given instruction may be globally-relevant. This strategy is too conservative because it does not look ahead in the program execution and does not consider any information about what may happen in the future. As a consequence, many unnecessary thread scheduling choices may be created during the state space traversal, if the POR technique determines imprecisely that the execution of some instruction may be globally-relevant when it is actually thread local.

In addition to many more thread interleavings that must be explored, there is another reason why performance of JPF suffers due to unnecessary scheduling choices created during the state space traversal. At every thread scheduling choice, JPF performs garbage collection, serializes the current program state, and performs state matching to determine whether the state has already been encountered. These are expensive operations — they may take up to half of the running time of JPF [34]. Thus excessive thread scheduling choices significantly increase the running time of JPF.

Considering only accesses to fields of heap objects, a solution used by the POR technique in JPF is to consider a field access on a heap object as possibly globally-relevant if the object is reachable from multiple threads via a chain of references (pointers) in the given program state. It performs the dynamic escape analysis proposed in [13] to determine whether a heap object is reachable from multiple threads. Note that some of the threads may not actually access the object during the program execution. For example, execution of an instruction may be considered globally relevant because it writes to a field of an object reachable from other threads, but if no other thread will ever access that field, then the thread scheduling choice at the field write is not necessary. This strategy used to identify thread-local field accesses is imprecise for two reasons. First, the strategy conservatively assumes that every object that is reachable from multiple threads in the current dynamic heap will actually be accessed by those threads in the future. Second, the strategy does not distinguish individual fields of an object; instead, it conservatively assumes that if a thread accesses a given object, then it will access all of the fields of that object.

In this paper, we propose to improve the precision using a hybrid analysis that com-

bines several static analyses and dynamic state space traversal. It provides information about the future behavior of program threads regarding field accesses. For each field access in the program, the proposed analysis determines whether the same field may be accessed again by some other thread in the rest of the program's execution. By identifying more thread-local executions of program instructions, the proposed analysis enables JPF to soundly avoid creating many redundant thread scheduling choices during the state space traversal, and allows to reduce the number of thread interleavings that must be processed to cover all possible distinct program behaviors.

Our approach is hybrid in that static analyses are used to speed up the dynamic state space traversal, and, at the same time, information about the dynamic program state is used to improve the precision of the static analysis results.

We have implemented the static analyses using the WALA library [49]. Experiments performed on several non-trivial Java applications show large reductions in the number of thread scheduling choices that JPF must explore, and therefore in the state space exploration time. It is now possible to check with JPF in a reasonable time much larger programs than before. Of the benchmark programs that we evaluated, two could not be verified with the original JPF even in 12 hours, but our hybrid analysis made it possible to verify them in 5 seconds and 5.2 hours, respectively.

However, all these improvements in precision must be balanced with the cost (in terms of time and memory) of the hybrid analysis. We have experimented with several variants of the analysis to explore the tradeoff between its cost and precision, and to find the variant that yields the biggest improvement in performance over the original JPF.

Contribution. We presented an earlier version of some of this work at ASE 2011 [37]. That paper introduced a static *field access analysis* that detects for each point in the code of each thread the set of fields that may be accessed after that point (in the future). This paper provides a more complete description of the field access analysis and presents experimental results on the current, most recent version of JPF (v7 released in July 2013). In addition, this paper makes two main new contributions. First, this paper defines a new static analysis that identifies fields written only during object initialization (the *immutable fields analysis*, Section 6). The results of this analysis are used to eliminate additional unnecessary thread scheduling choices during the state space traversal, and thus further improve the performance of JPF. Second, this paper reports on an experimental comparison with a fully dynamic approach to partial order reduction [15], which precisely identifies heap objects accessed by multiple threads during the program execution. We provide more details about dynamic POR in Section 8.5. This paper also investigates how our approach can be combined with dynamic POR and experimentally evaluates the performance of this combination.

2. Running Example

We will illustrate the key concepts of our approach on the simple Java program displayed in Figure 1. The program involves the Employee class with several data fields, the Company class that encapsulates a list of employees, and two worker threads that operate on different instances of Employee through methods of the Company class. The main method creates two employees and then starts two concurrent threads. In the example, each worker thread has distinct code (though this is not a requirement for our approach).

```
1  class Employee {
2    String name;
3    Integer salary;
4    Long index;
5
6    Employee(String nm, Integer slr) {
7      this.name = nm;
8      this.salary = slr;
9    }
10 }
11
12 class Company {
13   List employees = new ArrayList();
14
15   void add(String nm, Integer slr) {
16     Employee e = new Employee(nm, slr);
17     e.index = employees.size();
18     employees.add(e);
19   }
20
21   void add(Employee e) {
22     e.index = employees.size();
23     employees.add(e);
24   }
25
26   void print() {
27     for (Employee e : employees)
28       out.println(e.index + ":" + e.name);
29   }
30
31   void setSalary(Long id, Integer slr) {
32     for (Employee e : employees) {
33       if (e.index.equals(id)) e.salary = slr;
34     }
35   }
36
37   Integer getSalary(String nm) {
38     for (Employee e : employees) {
39       if (e.name.equals(nm)) return e.salary;
40     }
41   }
42 }
```

```
43 class FirstWorker extends Thread {
44   private Company cmp;
45
46   FirstWorker(Company c) {
47     this.cmp = c;
48   }
49
50   public void run() {
51     cmp.add("david", 115);
52     cmp.print();
53     cmp.getSalary("john");
54     cmp.setSalary(1, 150);
55     cmp.print();
56   }
57 }
58
59 class SecondWorker extends Thread {
60   private Company cmp;
61
62   SecondWorker(Company c) {
63     this.cmp = c;
64   }
65
66   public void run() {
67     cmp.getSalary("paul");
68     cmp.add("mark", 80);
69     cmp.print();
70   }
71 }
72
73 public static void main(String[] args) {
74   Company c = new Company();
75   c.add("john", 100);
76   c.add(new Employee("paul", 120));
77   Thread w1 = new FirstWorker(c);
78   Thread w2 = new SecondWorker(c);
79   w1.start(); w2.start();
80   w1.join(); w2.join();
81 }
```

Figure 1: Java program used as a running example

The most important characteristics of the program with respect to field accesses follow. The name field of the Employee class is written only in the constructor. The index field of the Employee class is written outside the constructor but only during the object initialization phase, before the object is added into the list and becomes visible to all

```
1   visited = {}
2   exploreState(s0, ch0)
3
4   procedure exploreState(s, ch)
5     if s ∈ visited then return
6     visited = visited ∪ s
7     for th ∈ getRunnableThreads(ch) do
8       s' = executeTransition(s, th)
9       if hasPropertyViolation(s') then terminate
10      ch' = createThreadChoice(s', getAllRunnableThreads(s'))
11      exploreState(s', ch')
12    end for
13  end proc
14
15  procedure executeTransition(s, th)
16    nextInsn = getNextInstruction(th)
17    s = executeInstruction(s, nextInsn)
18    return s
19  end proc
```

Figure 2: Basic algorithm for state space traversal

threads. Each worker thread contains code that performs both read and write accesses on each field of the Employee class, although only some kinds of accesses are performed on specific Employee objects. Note also that each Employee object is reachable from both worker threads via the employees collection, but each thread accesses only some of the objects.

3. Background

In this section, we first describe the basic algorithm for state space traversal of multi-threaded programs and then we provide formal definitions of well-known concepts related to partial order reduction. We also give specific technical details about JPF.

3.1. State Space Traversal

Figure 2 shows the algorithm that performs depth-first traversal of the state space of a multi-threaded program without partial order reduction of any kind. We use a recursive definition of the traversal algorithm because it allows us to describe the key aspects in a simple and clear way. Note, however, that model checking tools such as JPF actually implement the depth-first traversal using an iterative approach with an explicit stack.

The symbols s and s' in Figure 2 represent program states, the symbol ch represents a thread scheduling choice, and th denotes a thread runnable in a particular state. A program state is a snapshot of all variables and threads at some point on some execution path. A transition between two states corresponds to the execution of a single instruction (program statement) by a specific thread. After every transition there is a non-deterministic choice. In this paper we consider only thread scheduling choices and ignore the data non-determinism. Therefore, every state s is associated with a thread

scheduling choice ch that defines a list of threads that can execute from the given state. Each thread th in this list is associated with one transition leading out from the state s to some new state. We say that a transition tr is *enabled* in the state s iff tr represents execution of the next instruction of a thread th that is runnable in s.

The state space traversal algorithm maintains the set *visited* of states that have been already visited for the purpose of state matching. If the algorithm enters a state that has already been visited, it backtracks immediately by returning from the current level of recursion to the previous one (line 5). In the other case when the state s has been reached for the first time, the algorithm marks the state as visited (line 6) and begins processing transitions leading out from s. It processes transitions associated with runnable threads in the list defined by ch one by one. For each thread th, it performs the following steps: executes the next transition which corresponds to executing a single instruction (lines 8 and 15-19), checks all properties that it has been configured to verify (line 9), creates a scheduling choice ch' over all threads runnable in s' (line 10), and recursively explores the state s' and the new thread choice ch' associated with s' (line 11). The properties to be checked may be low-level properties such as absence of data races and deadlocks, or higher-level application-specific properties implemented by the user. Once all the outgoing transitions from s have been explored, the whole segment of the state space rooted at s has been fully processed and the traversal algorithm backtracks from s by returning to the previous level of recursion.

Using the algorithm shown in Figure 2, a model checker could systematically explore all possible thread interleavings and check all reachable program states for property violations. Nevertheless, some partial order reduction technique has to be used in practice to avoid creating redundant thread scheduling choices when the execution of the next instruction is thread-local and to mitigate the well-known state explosion problem [12].

3.2. Partial Order Reduction

State space traversal with partial order reduction, as described informally in Section 1, has been in the past formalized using the concepts of independent transitions, persistent sets [16], and ample sets [12]. Here we provide all the necessary definitions expressed in the context of the state space traversal algorithm in Figure 2 (Section 3.1).

The basic component of partial order reduction (POR) is the independence relation. Two transitions are *independent* if their execution in any order leads to the same program state and they cannot disable each other by making the associated threads not runnable.

Definition 1. Let tr_a and tr_b be transitions representing executions of instructions i_a and i_b by threads th_a and th_b, respectively. Transitions tr_a and tr_b are independent in state s if they preserve enabledness and commute. More precisely,

- if at least one of the transitions, say tr_a, is enabled in s, then the other transition tr_b is enabled in $tr_a(s)$ if and only if it is enabled in s, and
- if both transitions are enabled in s, then $tr_a(tr_b(s)) = tr_b(tr_a(s)) = s'$.

The set of pairs of independent transitions forms a binary, symmetric, and anti-reflexive *independence relation*.

Figure 3 illustrates the concept of independent transitions. Both execution paths captured by the figure are equivalent, as they end in the same state s'.

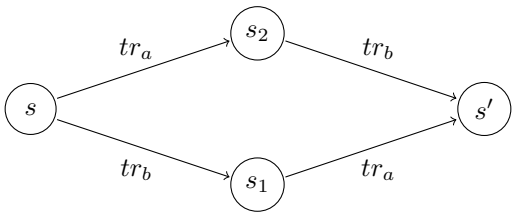

Figure 3: State space fragment containing two independent transitions

Note that the independence relation is sometimes defined as global over all states: two transitions are deemed independent only if they are independent in all possible program states. In practice, two transitions can be independent in some program states and dependent in other states. We consider only the *state-specific independence relation* in the rest of this paper, unless explicitly indicated otherwise.

The independence relation for a specific program is usually derived from the results of some analysis (static or dynamic). For example, a basic approach is to consider two transitions as dependent if they correspond to execution of the respective instructions by different threads and they interfere by accessing the same memory location (e.g., a heap object shared by multiple threads). For any pair of instructions that can be executed by different threads from the state s, if the execution of at least one of the instructions is thread-local in s, then the transitions corresponding to the execution of the instructions are independent.

Partial order reduction techniques use the independence relation to compute, for each state s reached during the traversal, the set of transitions that must be explored from s. The full set of transitions enabled in the state s contains one transition for each thread runnable in s, but in many cases it is provably sufficient to explore only a specific subset of the full set to cover all execution paths (thread interleavings) that start in s. This idea is formally captured by the concept of *persistent sets* [16].

Definition 2. Let $TR_F(s)$ be the set of all transitions enabled in the state s, and let $TR_P(s) \subseteq TR_F(s)$ be a subset. For any sequence $tr_1, \ldots, tr_n$ of transitions from $TR_F(s) \setminus TR_P(s)$, define the sequence of states $s_0, \ldots, s_n$ as $s_0 = s$ and $s_i = tr_i(s_{i-1})$. If for every such sequence, every transition tr_i is independent in state s_{i-1} from every transition in $TR_P(s)$, then $TR_P(s)$ is a persistent set.

A consequence of this definition is that every transition tr_p in a persistent set commutes with every sequence of non-persistent transitions: $tr_n(tr_{n-1}(\ldots(tr_1(tr_p(s)))\ldots)) = tr_p(tr_n(tr_{n-1}(\ldots(tr_1(s))\ldots))) = s'$. Therefore, in order to explore the full state space of all thread interleavings, it is sufficient from each state s to explore the transitions in the persistent set $TR_P(s)$, since every sequence of transitions is equivalent to some reordered sequence that begins with a transition from the persistent set. Figure 4 illustrates this fact on the example of a single transition tr_p in the persistent set and non-persistent transitions $tr_1, \ldots, tr_n$.

Suppose that the model checking algorithm has reached state s by executing an instruction in thread th_{cur}. If th_{cur} is runnable in s and the next instruction execution i on th_{cur} is thread-local, then all other instruction executions on other threads are independent of i. The singleton $\{i\}$ is therefore a persistent set, and execution can

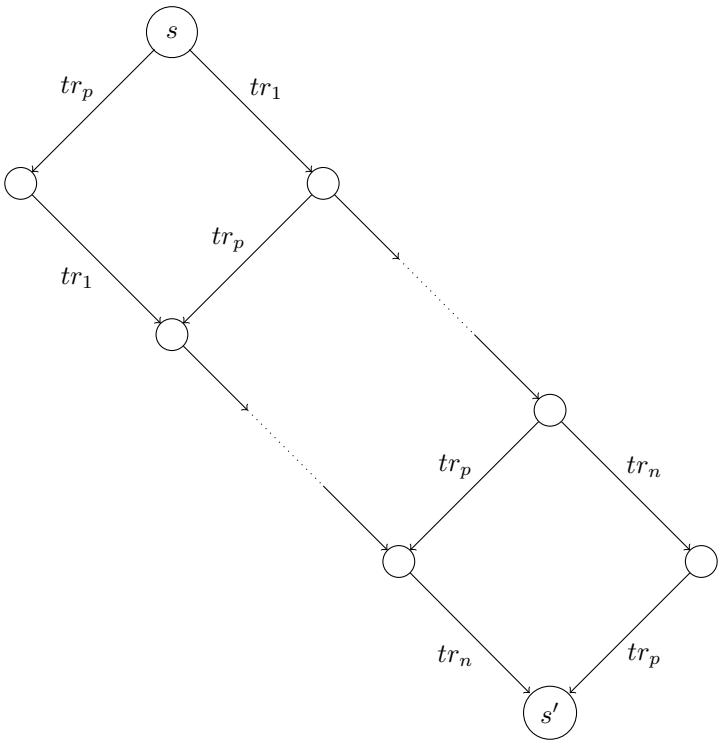

Figure 4: State space fragment illustrating the commutativity between transitions in the persistent set and other transitions (non-persistent)

simply continue on thread th_{cur} with no thread scheduling choice. If the next instruction execution i is globally-relevant, then the set of all threads runnable in s is a persistent set, and the algorithm must create a thread scheduling choice before it can execute i.

Note also that some existing partial order reduction techniques (e.g., [13]) have been defined using the concept of ample sets [12], a special case of persistent sets. An ample set is a persistent set that satisfies additional conditions necessary to ensure fairness with respect to all threads. Techniques based on ample sets must be used for checking liveness properties such as those defined using LTL_{-X}, but we do not discuss them here because our work is restricted to checking safety properties. Details on how to define partial order reduction based on ample sets can be found, e.g., in [13].

3.3. Java Pathfinder and On-the-Fly Partial Order Reduction

We begin this section with a high-level overview of Java Pathfinder (JPF) [20]. We then provide specific details about the state space traversal procedure and the partial order reduction technique used by JPF.

Java Pathfinder is a framework for exhaustive state space traversal of multi-threaded Java programs. The core of JPF is a special Java virtual machine that supports backtracking, state matching, and non-determinism of both data and scheduling decisions. Using these mechanisms, JPF runs a given Java program and explores its state space.

```
1   procedure executeTransition(s, th)
2     // execution of the first instruction is always globally relevant
3     nextInsn = getNextInstruction(th)
4     while nextInsn ≠ null do // while not at the end of the thread
5       s = executeInstruction(s, nextInsn)
6       nextInsn = getNextInstruction(th)
7       if isGloballyRelevant(s, nextInsn) then break
8     end while
9     return s
10  end proc
```

Figure 5: Procedure for executing transitions

A state of the program is a full snapshot of the virtual machine. The complete state includes all heap objects, stacks of all threads and all static data. The implementation of each bytecode instruction in the JPF interpreter includes appropriate code for tracking changes of the program state. Native methods have to be modeled in Java. Currently, JPF contains models only for a subset of the Java library classes with native methods. Note also that JPF does not support the current Java Memory Model — it models only sequential consistency of memory accesses.

JPF constructs the program state space on-the-fly during the traversal, i.e., it is not created in advance. The partial order reduction technique used by JPF follows the concepts described in Section 3.2. Given the current program state s in which the current thread is about to execute the instruction i, the content of the persistent set $TR_P(s)$ depends on i. If the execution of i is globally-relevant then $TR_P(s)$ contains transitions for all threads runnable in s, and otherwise it contains only the transition associated with i. Therefore, JPF has to make a non-deterministic thread scheduling choice only when the next instruction execution is globally-relevant. An atomic transition in the state space created by JPF is a sequence of instructions that consists of an instruction whose execution is globally-relevant in the given program state, followed by any number of instructions whose execution is thread-local, and it ends with a non-deterministic thread scheduling choice. All instructions in a transition are executed by the same thread.

The procedure for executing the next transition of a given thread from Figure 2 is extended with POR in Figure 5 by distinguishing thread-local and globally-relevant executions of program instructions. For each thread th, JPF executes bytecode instructions until it encounters an instruction i whose execution it considers to be globally-relevant. Just before the globally-relevant execution of i on thread th, JPF ends the current transition (line 7). The instruction i will be executed as the first step of the next transition associated with th.

The state space traversal procedure used in JPF explicitly saves the current program state only when it makes a new thread scheduling choice (at a transition boundary). Following this, in the rest of this paper, we consider program states only at transition boundaries, and ignore the intermediate states of the JPF virtual machine in the middle of a sequence of thread-local actions.

To decide whether execution of a given instruction from the current state may be globally-relevant, the partial order reduction technique in JPF uses only information in the current state. In particular, it does not analyze the code that will be executed in the

future.

In the context of this paper, an important category of instructions are accesses to fields of heap objects. When JPF is about to execute an instruction that reads or writes a field f of a heap object o in a thread t, it does not know whether any thread other than t will access $o.f$ in the rest of the program's execution. A solution used by the POR mechanism in JPF is to conservatively assume that an access of the field f may happen in the future if the object o is reachable from multiple threads via a chain of references (pointers) in the given program state. This reachability information is computed by a dynamic escape analysis [13] that systematically traverses the reachable heap from the local variables of all threads and from all static fields.

A thread scheduling choice is therefore created before the execution of an instruction accessing $o.f$ if multiple threads are runnable and the target object o is reachable from multiple threads in the current execution state. No thread choices are ever created before the execution of instructions that access fields of thread-local heap objects that are reachable only from a single thread according to the escape analysis. If the object o is thread-local right before the access to $o.f$, then there cannot be any concurrent accesses to $o.f$ from multiple threads, and therefore the access to $o.f$ must be thread-local.

Even though the escape analysis is dynamic and specific to the current execution state, it is still conservative because it considers a field access $o.f$ globally-relevant whenever the object o is reachable from multiple threads. Even when o is reachable from other threads, the access $o.f$ can still be thread-local if those other threads never actually access o in the future, or if they access only fields of o other than f. In these cases, the POR mechanism based on the dynamic escape analysis imprecisely considers the access to $o.f$ as globally relevant. Many redundant thread scheduling choices are created in such cases, forcing JPF to explore many redundant thread interleavings.

In the running example (Figure 1), JPF will create a thread scheduling choice at every access to fields of the Company class and fields of the worker thread classes, and also at every access to fields of the Employee objects stored in the employees list. This is because in this example, all those field accesses are on objects that are reachable by multiple threads. Note that there are many such accesses, and some are implicit: for example, the expression cmp.print() includes a read access of this.cmp before the call of the method print. In theory, JPF should also create scheduling choices before the execution of instructions accessing internal fields of the Java collection classes (e.g., accessed while retrieving Employee objects from the employees list), but the default configuration of JPF does not consider accesses to fields of classes in the Java core library as transition boundaries. Besides scheduling choices at the execution of field access instructions, JPF will also create scheduling choices when the example code calls Thread.start and Thread.join, since these methods change thread status.

4. Overview of Our Approach

We propose two analyses that identify globally-relevant field accesses more precisely than the POR mechanism in JPF does with the dynamic escape analysis. Given a program state s in which the running thread is about to access a field of a heap object, the analyses determine whether any other thread will read or write the same field in the future, starting from the state s. This question is undecidable, so each analysis can only give a conservative over-approximation of the answer. However, the analyses are more

precise than the approximation used by POR in JPF, which assumes that every thread may access every field of every object reachable from multiple threads. Our two analyses are independent and improve precision in different cases, so both analyses can be used together.

The *field access analysis* determines, for a given field f of a heap object o, program state s, and thread th, whether the thread th will ever access the field $o.f$ in the future if execution continues from the given program state s. The analysis considers the code of th and also the code of all child threads (transitively) started by th in the program execution following s, because any such child thread of th may execute in parallel with any other thread that is ready to execute in the state s.

The *immutable fields analysis* determines whether a given field f is ever modified after a reference to the object containing it has become visible from multiple threads. All writes to a field determined to be immutable must happen only before the object containing the field is reachable from the heap, i.e., before the object's reference has been written anywhere in the heap. The heap is the only way in which a reference to an object can be communicated from one thread to another. Therefore, if no reference to the object exists yet in the heap, then only the thread that created the object can access it.

To be precise, "a field f" refers to a syntactic declaration of a field, which is uniquely identified by the field's name and its declaring class. Two fields with the same name but declared in different classes are considered distinct. On the other hand, if a subclass inherits a field from its superclass, objects of both classes are considered to contain the same field. These details, though standard, are important because they make it possible to identify the field accessed by a given Java bytecode instruction using only the information specified in the instruction. We say "a field f of a heap object o" if we mean the given field in the given object.

The information computed by the proposed analyses is similar to interference dependence [24, 40] between field accesses, which is defined precisely as follows: executions of two instructions i_1 and i_2 are interference dependent if there exists a field f of a heap object o and an execution path p such that i_1 is a write access to $o.f$ in a thread t_1 and i_2 is a read access executed by a different thread t_2, execution of the instruction i_1 precedes execution of i_2 on execution path p, and there is no other field write to $o.f$ between i_1 and i_2 on execution path p. However, existing approaches to computing interference dependence [39, 40] use a flow-insensitive approximation that only checks whether the respective field access instructions are executed by different threads and does not take into account the fact whether i_2 can really be executed after i_1 on some execution path. Contrary to that, our field access analysis is flow-sensitive and it provides information about future behavior of program threads separately for each program state (code location). Another existing technique that computes similar information is dependence analysis [26], which identifies data-flow dependencies of field reads on previous field writes, but it applies only to single-threaded programs. We give more details about the existing approaches in Section 9.

We now describe how the proposed analyses are combined with the POR technique used in JPF. Both analyses are precomputed before JPF begins traversing the state space of the program. The analysis results are used to eliminate some unnecessary thread scheduling choices. In the rest of this paper, we use the acronym JPF-SA to denote the combination of JPF with the field access analysis and the immutable field analysis.

```
1  procedure isGloballyRelevant(s, insn)
2    if isFieldAccess(insn) and isImmutableField(insn.field) then return false
3    if isObjectReachableFromMultipleThreads(s, insn.object) then
4      if isFieldRead(insn) and existsFutureWrite(s, insn) then return true
5      if isFieldWrite(insn) and existsFutureRead(s, insn) then return true
6    end if
7    // other kinds of bytecode instructions
8    return false // default
9  end proc
```

Figure 6: Procedure that identifies instructions whose execution is globally relevant

Assuming that the currently executing thread is t and the next instruction i to be executed by t is a field access on $o.f$, where o is reachable from multiple threads, JPF-SA must decide whether to create a new thread scheduling choice before the execution of i.

First it queries the results of the immutable fields analysis. If the field f is determined to be immutable, then no thread choice needs to be created at any access to the field. This is because if a field is immutable, then only the thread t can write to it, and any read of the field by a different thread can only occur after the object has been stored into the heap, which is only possible after the last write to the field by t. Therefore, every write access to a field determined to be immutable is thread-local. Read accesses to the immutable field f are also thread-local because, after the object o containing f has become visible to other threads, no read access to f by any thread will depend on the behavior of other threads or influence them. More specifically, read accesses to such a field are never interleaved with writes of the same field by any other thread.

If the field f is mutable, then JPF-SA queries the results of the field access analysis separately for each thread that is ready to execute in the current dynamic program state. If the analysis determines that no thread other than t may access $o.f$ in the rest of the program's execution, then the current field access is thread local and no scheduling choice is necessary. The results of the field access analysis for a thread t include the fields accessed by child threads started from t, as indicated above. We distinguish reads and writes, so that JPF-SA would mark the execution of the field access instruction i as globally-relevant only when i would perform a write access and the future access is a read, or vice versa. Reordering a pair of reads has no effect on the program execution. In the case of two writes, their order affects the execution only if the field is eventually read. That case implies an additional write-read dependence that will force scheduling choices at the write accesses.

Instructions other than field accesses are unaffected by our analysis. JPF-SA makes scheduling choices at the execution of those instructions as the original JPF normally would.

The procedure isGloballyRelevant in Figure 6 summarizes how the analysis results are used by JPF-SA during state space traversal with partial order reduction (Figure 5).

Note that JPF combined with the proposed analyses still explores all interleavings of globally-relevant actions and therefore finds the same set of errors as the original JPF.

In the next two sections, we describe the two analyses in more detail.

5. Field Access Analysis

We present several variations of the field access analysis of increasing complexity and precision. First, in Section 5.1, we present the basic field access transfer functions within a simple static dataflow analysis, which is interprocedural but context-insensitive and therefore very imprecise. Given a (static) program point p, the analysis accumulates sets of all fields read and written on all possible paths from that program point, treating method calls and returns as simple branches. In Section 5.2, we illustrate on a specific example how context insensitivity can affect precision of the field access analysis. The second analysis, described in Section 5.3, improves precision using partial context sensitivity. When the control flow path starting at point p contains a method call at a given call site, the analysis takes advantage of the knowledge that the corresponding return will transfer control back to the same call site. The third analysis, a hybrid one, described in Section 5.4, further improves precision using dynamic context, taking advantage of the fact that for each thread, JPF knows not only the currently executing program point p, but the entire call stack. Given a dynamic call stack containing the currently executing instruction in each stack frame, the static analysis considers only control flow paths that start with that specific call stack. In particular, this hybrid analysis improves precision for field accesses occurring after the return from the method containing the designated program point p, since the dynamic call stack specifies precisely where execution will continue after each return. Whereas the first three analyses determine only the fields that are accessed, the fourth analysis, described in Section 5.5, uses points-to information to also determine the objects whose fields are to be accessed.

All variants of the field access analysis operate on one thread at a time, calculating the set of fields that may be accessed in only that static thread and its child threads (transitively). If there are multiple dynamic threads executing the same code, the code is statically analyzed only once, but the results are queried multiple times, once for each dynamic thread that executes the code (using the dynamic runtime context of that thread).

5.1. Context-Insensitive Field Access Analysis

The field access analysis is an inter-procedural backward flow-sensitive analysis that operates upon the inter-procedural control-flow graph (ICFG) of a given Java program.

We use the ICFG constructed by WALA, which includes explicit edges for exceptional control flow. WALA's call graph construction implementation includes facilities to model dynamic features used in the Java libraries such as dynamic class loading and reflection.

The analysis computes, for each point p in the code of a thread t, the set of fields that may be accessed either by the thread t in the fragment of its execution path from p until its end, or by some (transitive) child thread of t started at some point after p. The analysis distinguishes between read and write field access instructions, i.e., it computes separate sets of fields that may be read and that may be written.

The basic versions of the field access analysis consider only fields and do not distinguish different objects. For each field f, the analysis determines whether f may be accessed (for read, write, or both) on any object.

The context-insensitive field access analysis is an instance of a backward Kildall-style fixpoint dataflow analysis on the ICFG, and is similar to upward-exposed uses analysis [28, Section 8.3], which computes at each program point the future uses that

Instruction	Transfer function
$\star$	after0[ℓ] = $\bigcup_{\ell' \in \text{succ}(\ell)}$ before0[ℓ']
ℓ: v = p.f	before0[ℓ] = after0[ℓ] $\cup$ {r f}
ℓ: p.f = v	before0[ℓ] = after0[ℓ] $\cup$ {w f}
ℓ: return	before0[ℓ] = $\bigcup_{r \in \text{succ}(\ell)}$ before0[r]
ℓ: call M	before0[ℓ] = before0[M.entry]
ℓ: other instr.	before0[ℓ] = after0[ℓ]

Figure 7: Transfer functions for the context-insensitive analysis

may read the current value of each variable. The merge operator is set union, as shown in the first line of Figure 7, where the symbol $\star$ represents all instructions. As a consequence, the most optimistic value of the lattice (i.e., $\top$ in the traditional conventions of dataflow analysis, and $\bot$ in the conventions of abstract interpretation) is the empty set. All the sets are initially empty. The dataflow value at the end of the program is also the empty set, since after the program completes, no further fields will be read or written. The transfer functions for each kind of instruction are shown in the remaining lines of Figure 7. The transfer functions for read and write instructions, shown in the second and third lines, add the accessed field to the sets of reads and writes, respectively. The transfer functions for a call into a method and for a return from a method are the identity. In this version of the analysis, calls and returns are treated context-insensitively like any other branch instruction. Therefore, the transfer functions simply copy the read and write sets from just after a call site to the end of the called method, and from the beginning of the called method to just before the call site.

5.2. Context Sensitivity

The simple analysis presented in the previous section is very imprecise due to two kinds of context insensitivity, which we illustrate using the example in Figure 8. The figure represents the evolution of the (dynamic) call stack over the lifetime of a thread. Each method call is shown as a step upward (the call stack grows), and each return is shown as a step downward. In the example, method a calls method b, which in turn calls method c.

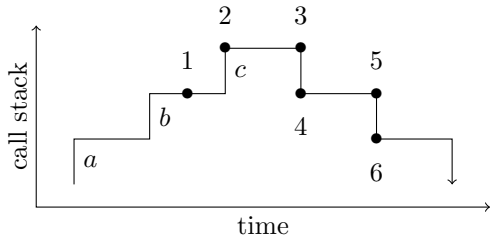

Figure 8: Example evolution of call stack

Suppose that JPF-SA queries the results of the context-insensitive analysis at point 1. The field access sets at this point include all fields that may be accessed after point 2, which is the entry point of method c, because the transfer function for a method call

instruction just propagates sets from the entry point of the callee method to just before the call site. The result at point 2 (and therefore also at 1) includes all field accesses inside method c and all field accesses after point 3, the end of method c. This result at point 3 includes all field accesses occurring after *every* call site of c. Yet, since we are querying the analysis results at 1, we know that method c will return to method b, and therefore that field accesses after any call site other than b are spurious. We call this kind of imprecision *callee-context insensitivity*, since the analysis of method b obtains imprecise information about its callee method c.

After the point labelled 5, we know that, in this particular execution, control returns to method a. However, when computing the results for program point 1 in method b, the static context-insensitive analysis conservatively considers *all* of the call sites that control could return to after method b. Therefore, the field access sets at point 5 (and therefore also at point 1) contain spurious fields that will not be accessed in this execution. We call this kind of imprecision *caller-context insensitivity*, since the analysis of method b does not know that its caller in this execution is precisely the method a.

In the next section, we modify the field access analysis to be callee-context-sensitive. Then, in Section 5.4, we will show how to use dynamic information to achieve caller-context sensitivity.

5.3. Callee-Context-Sensitive Field Access Analysis

The callee-context-sensitive field access analysis comprises two phases. The first phase considers the execution path from a program point p only until the return from the method containing p, rather than all the way to the end of the current thread. The transfer functions for the first phase are shown in Figure 9. They are similar to the context-insensitive ones from Figure 7, except for the return and call instructions. The transfer function for a return instruction returns the empty set, so that the analysis for each program point accumulates only those fields accessed before the current procedure returns. The transfer function for a call instruction combines the fields accessed during the execution of the callee method and the fields accessed in the caller after the call. Therefore, considering the example in Figure 8, the analysis at program point 1 would include the fields accessed inside method c and those accessed between points 4 and 5. However, it would *not* include fields accessed after other call sites of c, since the result for point 2 only includes field accesses up to the return from c; thus the analysis is callee-context-sensitive.

Instruction	Transfer function
$\star$	$\text{after1}[\ell] = \bigcup_{\ell' \in \text{succ}(\ell)} \text{before1}[\ell']$
ℓ: v = p.f	$\text{before1}[\ell] = \text{after1}[\ell] \cup \{\text{r f}\}$
ℓ: p.f = v	$\text{before1}[\ell] = \text{after1}[\ell] \cup \{\text{w f}\}$
ℓ: return	$\text{before1}[\ell] = \{\ \}$
ℓ: call M	$\text{before1}[\ell] = \text{before1}[\text{M.entry}] \cup \text{after1}[\ell]$
ℓ: other instr.	$\text{before1}[\ell] = \text{after1}[\ell]$

Figure 9: Transfer functions for the first phase of the callee-context-sensitive analysis

For correctness at point 1, the analysis must also consider the field accesses after point 5, and this is handled by the second phase. The transfer functions for the second

Instruction	Transfer function
$\star$	after2[ℓ] = $\bigcup_{\ell' \in \text{succ}(\ell)}$ before2[ℓ']
ℓ: v = p.f	before2[ℓ] = after2[ℓ] $\cup$ {r f}
ℓ: p.f = v	before2[ℓ] = after2[ℓ] $\cup$ {w f}
ℓ: return	before2[ℓ] = $\bigcup_{r \in \text{succ}(\ell)}$ before2[r]
ℓ: call M	before2[ℓ] = before1[M.entry] $\cup$ after2[ℓ]
ℓ: other instr.	before2[ℓ] = after2[ℓ]

Figure 10: Transfer functions for the second phase of the callee-context-sensitive analysis

phase are shown in Figure 10. The transfer function for a return instruction merges the fields accessed in all successors of the return instruction (i.e., the successors of all call sites calling the method that contains the return). Thus, the analysis is caller-context-insensitive. The transfer function for the call instruction is the key to achieving callee-context sensitivity. It merges the result for the callee *from the first phase* with the fields accessed after the call (from the second phase). Since the first phase only records fields accessed before the return from the callee method, the resulting set does not include spurious fields from other return sites to which the callee could return. In the example, the transfer function at point 1 would combine the first phase results for point 2, which cover the interval from point 2 to point 3, with the second phase results for point 4, which cover the rest of the execution, including the part of the execution after the return to method a at point 5.

< entry >
    {r *Company.employees*, w *Employee.index*, w *Employee.name*,
        w *Employee.salary*}
Employee e = new Employee(nm, slr);
    {r *Company.employees*, w *Employee.index*}
e.index = employees.size();
    {r *Company.employees*}
employees.add(e);
        {}
< exit >

Figure 11: Results of the first phase of callee-context-sensitive analysis

The results of the first phase for the body of the method add(String, Integer) from the running example (lines 15-19) are shown in Figure 11. Each of the sets in the figure contains only field accesses that may occur between the corresponding point in the code and exit (return) from the method. Here we discuss the content of the individual sets one-by-one, starting at the end, in order to illustrate the process of computing the analysis results. The last set, associated with the point right before the exit from the method, is empty because nothing happens after that point inside the add method and field accesses possibly occurring after the return from the method to its caller are not considered by the first phase. The previous set contains only the element r *Company.employees* because only the read access to the employees field of some instance of the Company class is

< entry >
{r *Company.employees*, rw *Employee.name*, rw *Employee.salary*,
rw *Employee.index*, rw *FirstWorker.cmp*, rw *SecondWorker.cmp*}
Employee e = new Employee(nm, slr);
{r *Company.employees*, rw *Employee.name*, rw *Employee.salary*,
rw *Employee.index*, rw *FirstWorker.cmp*, rw *SecondWorker.cmp*}
e.index = employees.size();
{r *Company.employees*, rw *Employee.name*, rw *Employee.salary*,
rw *Employee.index*, rw *FirstWorker.cmp*, rw *SecondWorker.cmp*}
employees.add(e);
{r *Company.employees*, rw *Employee.name*, rw *Employee.salary*,
rw *Employee.index*, rw *FirstWorker.cmp*, rw *SecondWorker.cmp*}
< exit >

Figure 12: Results of the second phase of callee-context-sensitive analysis

performed between the corresponding point in the method code and the exit. Similarly, the set associated with the point right before the statement e.index = employees.size() contains also the element w *Employee.index* because the next statement writes to the field index of some Employee object pointed to by the local variable e. The first set in the figure, associated with the point right after the entry to the method, contains also field accesses that may be performed during execution of the constructor of the Employee class, whose call is specified at the first line of the method's source code.

The results of the second phase are shown in Figure 12. The set computed for the method's exit node already contains field accesses that may occur after the various call sites of add(String, Integer), and these are propagated through the method. All kinds of accesses to fields defined in the program may occur after the call sites of add with the exception of a write access to Company.employees.

5.4. Context-Sensitive Field Access Analysis

Caller-context insensitivity is still a source of imprecision. For example, inside the print method, the analysis presented so far does not distinguish whether the method was called from the first or second call site in FirstWorker.run. It determines that the Employee.salary field can be read and written, even though it is only read and written after the first call to print, and not after the second call.

The precision of the analysis can be improved by taking advantage of the fact that the dynamic state in JPF includes not only the instruction to be executed, but also the entire call stack, which specifies the program points to which control will return at the end of each currently active method.

For this approach, only phase one of the static analysis described in the previous section is performed. Recall that it computes all fields that may be accessed from the current program point p until the method m containing p returns, including fields accessed in methods (transitively) called from m. This information is independent of the calling context. When a method on the call stack returns, the next frame on the call stack specifies the program point at which execution continues; the static analysis can be queried again for that point. Thus, given a dynamic call stack containing return program

points $(p_1, p_2, \ldots, p_n)$, which is a part of the dynamic program state available to JPF, and the current program point p_0, the set of fields that may be accessed from the current state until the end of the thread (when all methods on the stack have returned) is computed by JPF-SA at the dynamic analysis time simply as the union of the phase one results at all of the points p_i: $\bigcup_{i=0}^{n} \text{before1}[p_i]$. The result is the complete set of future field accesses for the current calling context of the given thread. This result is caller-context-sensitive in that it considers only the return control flow edges that will actually be taken as the stack unwinds, rather than all return edges from the relevant methods. The resulting set of fields reflects the current program counter in each method on the thread call stack (in each stack frame) and therefore distinguishes calls of a method m at different sites (code locations). In the example execution in Figure 8, when execution is at point 1, the call stack contains point 6, so the analysis returns the union of the result at point 1 (which covers the execution from point 1 to point 5) and the result at point 6 (which covers the rest of the execution).

In the running example from Figure 1, this analysis distinguishes the two calls of print from FirstWorker.run. The result at any point p inside print depends on the call site in FirstWorker.run from which it was called. When called from the first site (line 52), the result at p contains `rw` *Employee.salary* since the calls to getSalary and setSalary follow on any execution path. When called from the second site (line 55), the result at p does not contain any access to Employee.salary.

A remaining imprecision in this version of the field access analysis is that it only considers fields, and does not distinguish different object instances. In the running example, it does not distinguish between the different instances of Employee with respect to the salary field; it determines that the salary field of every employee may be written even if the salary of only one employee is changed. JPF-SA then always creates a thread choice at the write access to the Employee.salary field in the setSalary method (called from the FirstWorker thread) if the current program counter in the SecondWorker thread precedes the call to the getSalary method, which contains a read access of the salary field. This follows from the inability of the analysis to distinguish between different instances of the Employee class, and the fact that a possible read access of the salary field on some instance of Employee by the second worker thread might follow an invocation of the setSalary method in such a case.

5.5. Using Pointer Analysis

The precision of the field access analysis can be improved by using points-to information to distinguish object instances. A currently processed read (write) access to a field of an object is globally-relevant only if there is a future write (read) of the same field of the *same* object in another thread. If the analysis can show that the future access occurs on a definitely different object, the currently processed field access is thread-local. We have experimented with four variants of points-to analysis: an exhaustive context-insensitive analysis [3, 25], an exhaustive context-sensitive analysis that uses receiver objects as context [27], and context-insensitive and context-sensitive versions of a demand-driven analysis [46, 45]. The pointer analysis is always performed on the program before JPF starts.

5.5.1. Context-Insensitive Points-to Analysis

The points-to analysis computes, for each reference variable p in the program, a points-to set of allocation sites. If p may refer to a given object, then the site at which the object was allocated appears in the points-to set of p. Therefore, if two variables p and q have disjoint points-to sets, then they can never refer to the same object. The field access analysis is extended to make use of points-to information as follows. Instead of computing only a set of fields that may be accessed, the extended analysis keeps track of pairs $a.f$, where a is an allocation site and f is a field. When the extended analysis encounters an instruction accessing $p.f$ (i.e., field f of the object referred to by p), it queries the points-to set of p, and adds a pair $a.f$ to the set of field accesses for each allocation site a appearing in the points-to set. The result of the field access analysis contains $a.f$ if a given thread may access, in the future, the field f of an object o allocated at the site a. When JPF-SA is about to execute an instruction that reads (writes) $q.f$, it checks the analysis results to determine whether another thread may, in the future, write (read) the field f of an object allocated at one of the sites in points-to set of q. If not, then the execution of the instruction reading (writing) $q.f$ is thread-local and no thread scheduling choice is required.

5.5.2. Object-Sensitive Points-to Analysis

Object sensitivity improves the precision of points-to analysis by identifying objects with their allocation site and an abstraction of the current receiver object (i.e., this) at the time of the allocation.

For example, suppose a collection class allocates an array to hold the objects stored in the collection. Whereas a context-insensitive analysis would conflate the arrays of all instances of the collection class, an object-sensitive analysis would distinguish the arrays of different collection instances, if those collection instances were themselves allocated in different places.

The field access analysis can be modified to make use of object-sensitive points-to analysis in the same way as for context-insensitive analysis, except that allocation sites are replaced with pairs of allocation site and receiver context.

Since object sensitive analysis is computationally expensive, the default settings in WALA apply it only to a heuristic subset of allocation sites. A context is attached to an allocation site only if either the actual allocation site or the context involves some collection objects (instances of Collection or Map). The context extends only to one outer level with respect to the innermost collection object, even if the full receiver object string contains multiple collection objects.

Given the running example, pointer analysis distinguishes instances of the Employee class, because the program contains two allocation sites for the Employee objects at lines 16 and 76. However, no additional field accesses will be precisely identified as thread-local by JPF-SA when the field access analysis combined with pointer analysis is used, because all local variables of the type Employee in all methods except add may point to instances allocated at both sites.

5.5.3. Demand-Driven Points-to Analysis

Exhaustive points-to analysis is expensive because it computes points-to sets for all variables and considers all objects allocated during a program run, even though only the

points-to sets of a small subset of variables may actually be needed. More specifically, JPF-SA needs information about possible field accesses for an object only if the object is reachable from multiple threads when a field access instruction on the object is executed. A demand-driven points-to analysis evaluates explicit queries for individual points-to sets at a modest cost per query, and therefore is more efficient when the points-to sets of only a small proportion of all variables are needed.

In particular, with a demand-driven analysis, it becomes computationally feasible to perform a context-sensitive points-to analysis, for a significant improvement in precision. We have experimented with both the context-insensitive [46] and context-sensitive [45] variants of demand-driven points-to analysis that are implemented in WALA. Note that the context-sensitive demand-driven analysis is not object-sensitive.

To take advantage of demand-driven points-to analysis, the field access analysis is adapted as follows. When the analysis encounters a field read or write on $p.f$, it records both the field f and the variable p that it is accessed through. Thus the analysis accumulates a set of pairs $p.f$, such that the field f of an object pointed-to by variable p may be read (written) in the future. For illustration, in the running example the local variables Company.add().e and Company.print().e (and many others) refer to instances of the Employee class. When JPF-SA encounters a read (write) to $q.f$, it queries the points-to set of q, as well as the points-to sets of all variables p such that the field f may be written (read) through p in the future by another thread. The read (write) access to $q.f$ is marked as globally-relevant (i.e., a thread scheduling choice is needed) if the points-to set of q has a non-empty intersection with any of these other points-to sets.

To reduce the number of queries performed during a run of JPF-SA, the points-to set for each local variable is stored in a cache when it is needed for the first time and computed. At subsequent requests for the same variable, the points-to set is retrieved from the cache instead of querying the demand-driven pointer analysis each time.

6. Immutable Fields Analysis

Immutable fields are a commonly occurring design pattern. Intuitively, a given field is immutable if, for every object containing the field, there is no write access to the field after the object's initialization has completed. We assume that the initialization of a new object consists mainly of write accesses to the object's fields, which are all performed by the single thread that allocated the object. To make this definition of immutability precise, it is necessary to specify the extent of the initialization phase. Several approaches have been proposed at different points in this design space. For example, the approach proposed in [47] defines the end of the object's initialization phase as the moment when the object is accessed from another thread for the first time.

Our approach must be tailored explicitly for fields whose accesses can be safely considered thread-local by JPF-SA. The design of our immutable fields analysis must distinguish the initialization phase of the enclosing object in a balanced way: the analysis must be aggressive enough to identify immutable field patterns actually occurring in typical programs, yet conservative enough to ensure that it marks only fields that are written by no more than one thread.

Therefore, as already discussed in Section 4, we define the end of the object's initialization phase as the moment when the object escapes to the heap (and therefore

becomes potentially reachable from multiple threads). An object escapes to the heap when a reference to it is stored to some field of another object or to some array element. Since threads can communicate only through the heap and not through local variables, no other thread can access the object before it escapes. Now we can formally define the concept of immutable fields.

Definition 3. A given field f is *immutable* if, for every object o containing the field f, there is no write access to $o.f$ after a reference to the object o has escaped to the heap.

The running example (Figure 1) contains three immutable fields: Employee.name, Employee.index, and Company.employees. The field Employee.name is written only in the constructor (category 1). The field Employee.index is written twice. The write access at line 17 belongs to category 5, and the write access at line 22 belongs to category 2. The field Company.employees is also written only in the constructor.

The static immutable fields analysis consists of three stages: computing method summaries, performing escape analysis, and detecting immutable fields. The goal of the first two stages is to compute at each program location which local variables point to objects that may have escaped to the heap. This escape information is used in the third stage to identify immutable fields.

We describe all three analysis stages in the following sections.

6.1. Method Summaries

The first stage of the analysis summarizes how each method affects references to objects. This information will be used in the second stage to determine which objects may escape to the heap due to a method call. Each summary models the effect of a method m and of all methods that may be transitively called from m.

The summary for each method m identifies all parameters of m (including the implicit this parameter) that may escape to the heap inside m, identifies all parameters of m that may be returned from m, and specifies whether m may return an object that has already escaped to the heap.

The algorithm for computing method summaries is similar to the analysis proposed in [30], which generates more complete summaries of method behavior. The algorithm uses a worklist that at any moment contains all methods whose summaries are not yet sound. The worklist is initialized with all methods of the input program and all libraries that it uses. The initial summary for every method says that none of its parameters escape, none of its parameters are returned, and the returned value is not accessible from the heap. In each iteration, the algorithm removes a method m from the head of the worklist and updates its summary using a flow-insensitive intra-procedural analysis. If the newly computed summary for m is different than the previous existing summary for m, then all the callers of m are added to the worklist because their summaries depend on the summary of m. The algorithm terminates because the summaries grow monotonically, and the number of possible summaries for each method is finite. When the algorithm terminates (i.e., the worklist is empty), the summary of each method soundly approximates the behavior of that method under the assumption that every other summary soundly approximates its respective method.

The intra-procedural analysis of the method m consists of several steps whose description follows.

1. For each value v_r returned from m, reaching definitions are traversed iteratively to determine the set R of all possible sources of the value v_r. If one of the possible sources of v_r is a read of a field or array element, then m may return a value that has already escaped to the heap, so this information is recorded in the summary for m. If one of the sources is the value returned from some call to some method m', then the currently available summary for m' is queried. If m' may return an already escaped object, then m can also. In addition, the summary of m' specifies which of its parameters it may return, so the corresponding arguments are added to the set R, and the traversal of the reaching definitions continues further.
2. All formal parameters of m that are possible sources of the value v_r returned by m are recorded in the summary of m.
3. All local variables of m that may point to an object that escapes inside m are identified, and the set E of such variables is created. The set E includes local variables of m whose value is written into a field or an array element. If a local variable v of m is an actual parameter of a call to some method m' and the currently available summary for m' specifies that the corresponding formal parameter may escape inside m', then v is also added into the set E. All possible sources of the variables in E are identified using an iterative traversal of reaching definitions, and are added into the set E.
4. The intersection of the sets R and E is checked for emptiness. If the intersection is not empty then m may return a value that escapes inside m, and this is also recorded in the summary for m.
5. All parameters of m that are in the set E are added to the summary of m as parameters that may escape inside m.

Since the analysis cannot analyze native methods, it handles them conservatively by assuming that all of their parameters can escape and that they return an already escaped object.

6.2. Escape Analysis

The second stage computes at each code location which local variables may point to objects that have escaped to the heap. We use the short name *escaped variables* in this section. The analysis is flow sensitive and inter-procedural, and computes a set of escaped variables for each node and edge in the ICFG.

The analysis associates with each code location l a set of variables that may have escaped before l. The merge operator is set union. At the beginning of the analysis, all of the sets are initialized to an empty set, indicating that no objects have escaped. The analysis iterates until it reaches a fixed point.

Figure 13 shows the transfer functions for instructions that change the set of escaped variables. This includes field accesses, array element accesses, and object allocation. Method calls and returns are discussed below. The symbol "aliases(v)" represents the set of all local variables of m that may be aliased with v. We use type-based aliasing that is defined as follows. If the static type of one local variable is a subtype of the static type of another, then the variables are considered possibly aliased. In addition, if the static types of both variables are interfaces and there is a class that implements both interface types, then the variables are also considered possibly aliased.

Instruction	Transfer function
$\star$	before[ℓ] = $\bigcup_{\ell' \in \text{pred}(\ell)}$ after[ℓ']
ℓ: v = new C	after[ℓ] = before[ℓ] \ { v }
ℓ: v = p.f	after[ℓ] = before[ℓ] ∪ { v }
ℓ: p.f = v	after[ℓ] = before[ℓ] ∪ { v ∪ aliases(v) }
ℓ: v = a[i]	after[ℓ] = before[ℓ] ∪ { v }
ℓ: a[i] = v	after[ℓ] = before[ℓ] ∪ { v ∪ aliases(v) }
ℓ: other instr.	after[ℓ] = before[ℓ]

Figure 13: Transfer functions for the second stage of the immutable fields analysis

Whenever the analysis encounters an instruction that stores a newly allocated object into the variable v, the transfer function removes v from the set of escaped variables, as shown in the second line. For instructions that read a field or an array element into the variable v, the transfer functions add v to the set of escaped variables, because v now points to an object loaded from the heap. In the case of instructions that write into a field or an array element, the transfer functions add v and all variables possibly aliased with v to the set, because every object that may be pointed to by v (and its aliases) is now reachable from the heap.

When the analysis encounters a method call, it propagates the escaping variable information into the callee, but it uses the method summaries computed in the first phase to model the effect of the callee on the caller.

Consider a method M that has formal parameters $v_0, v_1, \ldots, v_n$ and returns the value v_r, and the instruction r = call M($e_0, e_1, \ldots, e_n$) that calls M with arguments $e_0, e_1, \ldots, e_n$ (expressions in the caller) and stores the returned value into the variable r. The method M represents the callee in this scenario. The symbol "summ(M)" represents the summary of M — it is a set that contains all parameters v_i that may escape in M, and, if the returned value v_r escapes inside M, also the value v_r. For every argument e_i that is known to have possibly escaped before the call, the corresponding parameter v_i is added to the set of escaped variables at the beginning of the callee M. The transfer function is:

$$\text{before}[\text{M}(v_0, v_1, \ldots, v_n).\text{entry}] = \{v_i \mid e_i \in \text{before}[\text{call M}(e_0, e_1, \ldots, e_n)]\}$$

A set union is performed at the beginning of the callee to merge information from different call sites.

When processing the return from M (the callee) to the caller, all information about the local variables in M is ignored and only the method summary is used to update the sets in the caller. This makes the analysis context-sensitive and therefore more precise. Specifically, for every parameter v_i that escapes inside M according to its summary, the corresponding argument e_i and all its aliases are considered to have escaped at the call site, and the transfer function for the call site adds them to the set of escaped variables. The transfer function for the return is:

$$\begin{aligned}\text{after}[\text{r = call M}(e_0, \ldots, e_n)] = \text{before}[\text{call M}(e_0, \ldots, e_n)] \cup \{r \mid v_r \in \text{summ(M)}\} \cup \\ \cup \{e_i \cup \text{aliases}(e_i) \mid v_i \in \text{summ(M)}\}\end{aligned}$$

It is necessary to consider aliasing at the return from M because if an argument e_i, i.e., a local variable of the caller, points to an object that escapes inside M, then any other

variables in the caller that also point to the same object must be marked as escaping too. If the method summary specifies that the value v_r returned from M may have escaped, then the variable r in the caller to which the return value is assigned is added to the set of escaped variables.

We split the escape analysis into two stages to achieve most of the precision of context sensitivity at low cost. The analysis as presented does not have to reanalyze each method multiple times in different calling contexts. The method summaries enable the second stage to model method calls precisely without confounding analysis information from multiple call sites at the return to each caller. Without method summaries, the dataflow facts from all call sites of a given method that are merged at its beginning would be propagated back to every call site when the method returns. Thus, the facts from every call site would be spuriously propagated to every other call site that calls the same target method. Thanks to the method summaries, the merged information inside the callee is not propagated back to the caller; instead, the method summary is used to adjust the information in the caller to account for the effect of the callee. The ICFG contains many edges leading to and from library methods, such as HashMap.put, and constructors of application classes. The results of a completely context insensitive analysis without the use of method summaries would therefore be very imprecise.

6.3. Detecting Immutable Fields

The third stage is a simple intra-procedural analysis that uses the results of the escape analysis to determine which fields are immutable. For each field write access to $p.f$, the analysis determines from the escape analysis results whether the object that p points to may have already escaped to the heap. A field f is marked immutable if no write to f occurs on a variable that may point to an object that has escaped to the heap.

We perform this stage only for fields of application classes defined in the program. JPF does not make thread choices in the library methods in the default configuration, so we can safely ignore fields of classes from the Java core libraries — we do not need to know the immutability of fields defined in the library classes.

7. Running Example Revisited

We summarize the benefits of using both the immutable fields analysis and the context-sensitive field access analysis with JPF on the running example (Figure 1).

As already indicated in Section 3.3, the original JPF would make a thread choice at every field access on objects reachable from multiple threads. This includes every read access to the field Company.employees once the worker threads are started, read accesses to the cmp field in the run methods of the worker thread classes, and every field access on Employee objects stored in the list employees.

The static analysis successfully identifies all immutable fields in the program (Employee.name, Employee.index, Company.employees). The fields FirstWorker.cmp and SecondWorker.cmp are not immutable, even though they are written only in the constructors, because a reference to a worker thread object escapes to the heap in the constructor of the superclass (Thread) before the cmp field is initialized. A reference to a Thread object is stored into an array inside a ThreadGroup object.

The remaining field is Employee.salary. It is written in the constructor (line 8) and in the setSalary method (line 33), and read only in the getSalary method (line 39). The

setSalary method is called only by the FirstWorker thread at line 54. JPF will not make a thread choice before the write access in the constructor, because the Employee object is not yet reachable from multiple threads. When considering the results of the context-sensitive field access analysis and dynamic program counters of the worker threads, JPF-SA will create a thread choice at the write access to Employee.salary in the setSalary method (line 33) only if the current position of the program counter of the SecondWorker thread precedes the read access in getSalary. Similarly, in the case of the read access to the field (line 39), JPF-SA will make a thread choice only if the read access is to be performed by the SecondWorker thread and the program counter of the FirstWorker thread lies before the write access at line 33.

Note, however, that some thread choices created by JPF-SA at accesses to Employee.salary are unnecessary because worker threads may perform the accesses on different Employee objects. Knowing the results of the pointer analysis would not help in these cases. JPF-SA would still create unnecessary thread choices at the respective accesses to the field Employee.salary, because the employees list contains objects from both allocation sites defined in the program for instances of the Employee class (lines 16 and 76).

8. Evaluation

We start this section with a short description of our implementation and benchmark programs. Then we present and discuss experimental results on the proposed static analyses. Our main findings and key results are summarized in Section 8.4. In the last subsection, we compare the JPF-SA approach with fully dynamic partial order reduction.

8.1. Implementation

We implemented all variants of the field access analysis (Section 5) and the immutable fields analysis (Section 6) using the WALA library [49], and integrated them with JPF.

Relevant details about the static analyses follow. We use a Killdall-style fixed-point data flow solver over the inter-procedural control-flow graph of a given Java program. We analyze each program together with the Java 7 core libraries. The demand-driven pointer analysis uses WALA's default analysis budget for each query.

In JPF, the decision whether to make a thread choice right before a field access is done by the interpreter of bytecode instructions. We have implemented a custom interpreter of the field access instructions that considers the results of the static analyses. The behavior of JPF-SA also depends on the configuration of the static analyses. When the field access analysis is configured to use the results of pointer analysis, a plug-in for JPF stores the allocation site (and context if needed) for each (dynamic) object allocated during a program run.

Our implementation, experimental setup, and a redistributable subset of the benchmarks is publicly available at `http://d3s.mff.cuni.cz/projects/formal_methods/jpf-static/scp-immfields.html`.

A general problem for static analyses of Java programs is handling of native methods, many of which appear in the Java library classes. We addressed this issue by manually inspecting the models of native methods, which are a part of the JPF distribution, and creating a list of fields accessed inside them. We designed the field access analysis such that it considers these fields as always accessed after any program point. This is a coarse

over-approximation, but a sound approach — no possible field accesses are omitted by the analysis. However, it has no practical consequence with respect to the number of thread choices created by JPF-SA during the state space traversal, because JPF does not make choices inside library classes.

8.2. Benchmarks

We evaluated our approach on eight benchmark programs: CoCoME [7], CRE Demo [1], the Daisy file system [38], the Raytracer benchmark from the Java Grande Forum suite [19], the Elevator benchmark from the PJBench suite [36], a plain Java version of the jPapaBench benchmark [21], a plain Java version of the CDx benchmark [23], and Simple JBB. All of these programs are multi-threaded and the individual threads interact significantly.

The CoCoME application (3500 lines of code) is a prototype of a trading system for department stores. Its architecture has two parts: an inventory management system responsible for a product database and a cash desk line formed by a set of cash desks. Each cash desk is represented by several components that control cash desk hardware (e.g., a bar code scanner). For our experiments, we used a configuration with two cash desks. A part of the application is a simulator (test driver) that runs two threads representing clients.

The CRE Demo application (1700 loc) is a high-level prototype of a software system for providing WiFi internet access at airports. It was developed as a demo application for the Component Reliability Extension project (CRE) [1], whose purpose was to add support for formal behavior specifications to the Fractal component platform [8]. The application consists of modules for user authentication and management of network addresses, and it models operations like payment with a credit card. A part of the application is a simulator that runs two threads representing distinct clients.

Daisy is a simple file system developed as a challenge problem for verification tools (1100 loc). We used it with a manually created test driver that runs two concurrent threads that perform various operations on files and directories. Before our experiments, we also fixed several errors that were created intentionally by the authors, so that JPF explores the whole state space and does not stop prematurely upon hitting an error.

The Raytracer benchmark (1100 loc) involves two worker threads that render a given scene. The scene contains various graphical objects such as lights and spheres. Each thread computes a part of the image. We used a configuration with few objects in the scene and a small size of the rendered image for the purpose of our experiments.

The Elevator benchmark (400 loc) is a simulator of elevators running in a building. Each elevator is modeled by one thread, and one additional thread represents people requesting elevators. We used a configuration with two elevators and four operations performed by each elevator.

The jPapaBench benchmark (4600 loc) is a Java model of on-board control software for a UAV (unmanned aerial vehicle). It contains several concurrent threads that highly interact via fields of heap objects. Some threads represent the environment and external interrupts, and the remaining threads perform tasks related to airplane control and navigation.

The CDx benchmark (3900 loc) simulates an aircraft collision detector. It contains two threads: a simulator and a detector. The simulator thread represents a radar that

provides information about current positions of several aircraft. The detector thread periodically receives a radar frame and detects possible collisions. Both threads communicate using a shared circular buffer. We use a configuration with a pregenerated simulator of two aircraft. We simplified the original code of CDx for the purpose of our experiments. In particular, we removed the parser and evaluator of input files that specify aircraft behavior, as it is not needed with the pregenerated simulator. We also eliminated the code for recording performance data (e.g., various counters), and dependencies of the program behavior on the current system time.

Simple JBB (3000 loc) is a simplified version of the SPEC JBB 2005 benchmark [43], which is a model of an enterprise information system that allows concurrent processing of requests from clients. It models several databases (e.g., orders and stock) and transactions that operate upon these databases. Clients are represented by concurrently running threads. Checking the original SPEC JBB benchmark with JPF is not possible because SPEC JBB uses parts of the Java library that are not supported by JPF (e.g., complex file I/O and time measurement). Moreover, the original SPEC JBB uses large data structures with many elements and each thread executes many transactions (from the point of view of state space size) — while large input data and high load are useful for performance testing, it is sufficient to use small inputs and loads for verification of all possible interleavings of instruction executions that are globally-relevant. We derived Simple JBB from the original SPEC JBB to eliminate these issues. Specifically, we removed the use of library calls not supported by JPF, and we reduced the size of some data structures and the number of transactions performed by each thread. We used a configuration with two threads. Despite these simplifications, Simple JBB has the same concurrency behavior as the original SPEC JBB, so verifying Simple JBB with JPF gives some assurance of the correctness of the original SPEC JBB.

8.3. Experiments with Static Analyses

We performed experiments with all variants of the field access analysis described in Section 5 and the immutable fields analysis described in Section 6. More precisely, we evaluated specific configurations of the static analysis that represent combinations of possible values of the following parameters: the context-sensitivity of the field access analysis, the kind of pointer analysis that is used in the field access analysis, and whether the immutable fields analysis is enabled or disabled. Table 1 shows for each parameter value both its full name and a short name that will be used in the tables of results.

Table 1: Possible configurations of static analysis

Parameters	Values: full name	Short name
field access analysis	context-insensitive	insensitive
	callee-context-sensitive	callee
	context-sensitive	sensitive
pointer analysis	context-insensitive exhaustive	ci ex
	context-sensitive exhaustive	cs ex
	context-insensitive demand-driven	ci dd
	context-sensitive demand-driven	cs dd
immutable fields analysis	disabled	
	enabled	

The purpose of our experiments is to determine how much our approach can reduce the number of thread choices explored by JPF and the running time of JPF, while preserving the full coverage of program behaviors. In order to make this possible, we configured JPF-SA such that it does not search for any errors and traverses the whole state space of the given program in each case.

All the results of our experiments are provided in Tables 2-9. We report values of the following metrics: the number of thread choices created by JPF-SA (Table 2 and 6), the running time of the static analysis (Table 3 and 7), the total running time of JPF combined with the static analysis (Table 4 and 8), and the memory consumption (Table 5 and 9). Note that the number of thread choices created by JPF is equal to the number of explored states. We limited the time of each run of JPF-SA to 12 hours. This limit was reached only on the jPapaBench and CDx benchmarks, and only in some configurations of the analysis.

The experimental results show that the proposed static analyses significantly reduce the number of thread choices that JPF-SA needs to create to check all possible behaviors of the given program and the time required to explore the whole state space of the program. The static analysis must be sufficiently precise to achieve these improvements, but it is also important to consider the cost of the static analysis and of its usage by JPF. In some cases, the speedup of JPF due to more precise static analysis is smaller than the cost of computing and using that static information. We discuss the usefulness of individual configurations and results for specific benchmarks in the next two subsections, and carefully evaluate the tradeoff between the cost and precision of the static analysis.

8.3.1. Configurations without Pointer Analysis

The first four tables 2-5 provide results for the original JPF, and for configurations of the static analysis that do not include pointer analysis. The data for the original JPF (in the second column of each table) represent the baseline against which we measure the benefits of our approach.

We first look at the field access analysis only, with the immutable fields analysis disabled. The results indicate that context sensitivity is in general very important to the effectiveness of the field access analysis. It enables significant reductions of the number of thread choices compared to the original JPF. The context-insensitive field access analysis achieves only a very modest reduction of thread choices, and only for CoCoME and Simple JBB. Callee-context sensitivity improves on the context-insensitive variant only for Daisy and Simple JBB, and has almost the same results for other benchmarks. Therefore, the best variant of the field access analysis is the fully context-sensitive analysis that uses the dynamic call stack from JPF. It achieves significant reductions compared to callee-context sensitivity for all benchmarks except CDx, up to a factor of 12 for CRE Demo. The cost of the fully context-sensitive analysis is small compared to its benefits; on every benchmark except Raytracer, the total time required by JPF with the static analysis is much lower than for the original JPF.

The experimental results also show that the immutable fields analysis can eliminate many additional unnecessary thread choices, and can therefore improve the performance of JPF to a large degree. In particular, the most complex benchmark, CDx, can only be verified when the immutable fields analysis is enabled. On the other hand, for some benchmarks, the immutable fields analysis has no effect on the number of thread choices (e.g., jPapaBench), or the number of eliminated thread choices does not justify the cost of

Table 2: Thread choices created by JPF: configurations without pointer analysis

		field access			immutable
	orig JPF	insensitive	callee	sensitive	
CoCoME	64580	64030	64026	9188	disabled
		27394	27394	8652	enabled
CRE Demo	47114	47114	47114	3736	disabled
		6833	6833	2919	enabled
Daisy	28120251	28120251	23648987	6484559	disabled
		6292903	6292903	6292903	enabled
Raytracer	555126	555126	555124	487735	disabled
		351344	351344	351317	enabled
Elevator	10116121	10116121	10116041	2707528	disabled
		4486872	4486872	2707528	enabled
jPapaBench	n/a	n/a	n/a	332	disabled
		n/a	n/a	332	enabled
CDx	n/a	n/a	n/a	n/a	disabled
		1161949	1161949	890872	enabled
Simple JBB	498837	489845	467797	207174	disabled
		67921	67895	54606	enabled

the immutable fields analysis (e.g., for CRE Demo). The benchmark programs CoCoME, CDx, and Simple JBB contain particularly many immutable fields. The fields are mostly in classes whose instances are used as data transfer objects (e.g., objects representing stock items in Simple JBB). The fields are written only in the constructor before the object escapes to the heap. After the object has escaped, the fields are possibly read by multiple threads. The field access analysis alone cannot eliminate thread choices at accesses to such fields, because for each write access to a given field during the program execution, there may be read accesses to the same field which happen in the future (e.g., in the `equals` method when the object is stored into some collection or in the getter methods), and vice versa.

The general trends in the effectiveness of the variations of the field access analysis remain similar when the immutable fields analysis is also enabled. When both analyses are combined, the callee-context-sensitive variations yield the same results as the context-insensitive variations for all benchmarks, while context-sensitive field access analysis enables significant additional reductions for most benchmarks.

A limitation of our immutable fields analysis is that it does not mark a given field f as immutable when the following conditions hold for f: (i) the last write to f occurs after the enclosing object o escapes to the heap, (ii) only the thread that created the object o writes to the field f, and (iii) other threads do not access the field f before the last write. This can happen either when other threads do not read the field at all, or when other threads read the field f but thread synchronization guarantees that all of the read accesses from other threads happen only after the last write by the creating thread. However, these cases are covered by the context-sensitive field access analysis. If the results of the field access analysis for the current dynamic point say that only read accesses may happen in the future (i.e., the program has already executed the last

Table 3: Running time of static analysis: configurations without pointer analysis

		field access			**immutable**
	orig JPF	insensitive	callee	sensitive	
CoCoME	n/a	6 s	6 s	3 s	disabled
		15 s	13 s	11 s	enabled
CRE Demo	n/a	8 s	8 s	5 s	disabled
		16 s	16 s	14 s	enabled
Daisy	n/a	4 s	5 s	3 s	disabled
		11 s	12 s	11 s	enabled
Raytracer	n/a	4 s	5 s	3 s	disabled
		12 s	12 s	11 s	enabled
Elevator	n/a	6 s	6 s	3 s	disabled
		16 s	15 s	14 s	enabled
jPapaBench	n/a	4 s	4 s	3 s	disabled
		6 s	7 s	6 s	enabled
CDx	n/a	115 s	198 s	97 s	disabled
		953 s	1013 s	933 s	enabled
Simple JBB	n/a	21 s	15 s	6 s	disabled
		23 s	23 s	17 s	enabled

Table 4: Total running time of JPF and static analysis: configurations without pointer analysis

		field access			**immutable**
	orig JPF	insensitive	callee	sensitive	
CoCoME	118 s	119 s	119 s	23 s	disabled
		64 s	63 s	29 s	enabled
CRE Demo	70 s	80 s	80 s	13 s	disabled
		31 s	30 s	20 s	enabled
Daisy	18439 s	18248 s	15383 s	5288 s	disabled
		4579 s	4697 s	4785 s	enabled
Raytracer	351 s	390 s	390 s	367 s	disabled
		263 s	266 s	275 s	enabled
Elevator	6958 s	6850 s	6748 s	1852 s	disabled
		3075 s	3020 s	1831 s	enabled
jPapaBench	$>$ 12 hours	$>$ 12 hours	$>$ 12 hours	5 s	disabled
		$>$ 12 hours	$>$ 12 hours	7 s	enabled
CDx	$>$ 12 hours	$>$ 12 hours	$>$ 12 hours	$>$ 12 hours	disabled
		23958 s	23233 s	18722 s	enabled
Simple JBB	1547 s	1556 s	1415 s	689 s	disabled
		238 s	234 s	191 s	enabled

write in the creating thread), then all the future accesses to the field will be marked as thread-local and JPF-SA will not make any thread choice at those accesses. We found several fields in this category in the code of Simple JBB by manual inspection. JPF with the field access analysis makes thread choices at accesses to such fields only when the

Table 5: Memory consumption of JPF and static analysis: configurations without pointer analysis

		field access			**immutable**
	orig JPF	insensitive	callee	sensitive	
CoCoME	0.8 GB	1.9 GB	3.8 GB	3.6 GB	disabled
		3.5 GB	4.8 GB	3.6 GB	enabled
CRE Demo	0.8 GB	2.7 GB	4.2 GB	2.4 GB	disabled
		3.5 GB	3.4 GB	2.4 GB	enabled
Daisy	2.3 GB	5.9 GB	4.5 GB	2.4 GB	disabled
		5.6 GB	5.5 GB	3.6 GB	enabled
Raytracer	3.3 GB	2.4 GB	3.8 GB	3.6 GB	disabled
		3.7 GB	5.2 GB	3.4 GB	enabled
Elevator	0.8 GB	4.4 GB	4.3 GB	3.6 GB	disabled
		3.9 GB	5.2 GB	2.8 GB	enabled
jPapaBench	n/a	n/a	n/a	0.9 GB	disabled
		n/a	n/a	1.4 GB	enabled
CDx	n/a	n/a	n/a	n/a	disabled
		17.1 GB	16.3 GB	14.2 GB	enabled
Simple JBB	1.4 GB	4.8 GB	5.1 GB	3.7 GB	disabled
		3.8 GB	4.2 GB	4.1 GB	enabled

last write access may happen in the future after the current dynamic program state.

The running time of field access analysis (Table 3) is generally negligible compared to the running time of JPF (state space traversal). It is at most 200 seconds for the CDx benchmark and less than 20 seconds for all other benchmarks. The running time of the immutable fields analysis is much higher than the running time of the field access analysis — especially for CDx — but it is still worthwhile because it has a much lower running time than JPF and its benefits warrant the higher cost, as indicated by data in tables 3 and 4. The memory consumption results show that although the static analyses do require some memory, the increases in memory consumption are feasible.

8.3.2. Comparing Different Pointer Analyses

Tables 6-9 quantify the effects of different pointer analyses. Here, we consider only configurations that include the context-sensitive field access analysis and the immutable fields analysis. The second column of each table again represents the baseline configuration without any pointer analysis.

The points-to analyses that we evaluated (in combination with the field access analysis) have only a very small effect on the precision that does not justify their cost. The exhaustive points-to analyses reduced the number of thread choices by 6% for the CoCoME benchmark, by 20% for Daisy, and by 4% for CDx. On the other benchmarks, none of the exhaustive points-to analyses reduced the number of thread choices. The demand-driven points-to analyses did not reduce the number of thread choices on any of the benchmarks.

In addition to yielding no reduction in the number of thread choices, the demand-driven points-to analyses had a significant runtime overhead when JPF-SA issued very large numbers of queries, because the demand-driven analysis must do some small amount

Table 6: Thread choices created by JPF: configurations involving different pointer analyses with the context-sensitive field access analysis and the immutable fields analysis

	pointer analysis				
	none	ci ex	ci dd	cs ex	cs dd
CoCoME	8652	8112	8652	8112	8652
CRE Demo	2919	2919	2919	n/a	2919
Daisy	6292903	5258535	6292903	5258535	6292903
Raytracer	351317	351317	351317	351317	351317
Elevator	2707528	2707528	2707528	2707528	2707528
jPapaBench	332	332	332	332	332
CDx	890872	851867	890872	851867	890872
Simple JBB	54606	54606	54606	54606	54606

Table 7: Running time of static analysis: configurations involving different pointer analyses with the context-sensitive field access analysis and the immutable fields analysis

	pointer analysis				
	none	ci ex	ci dd	cs ex	cs dd
CoCoME	11 s	12 s	14 s	50 s	14 s
CRE Demo	14 s	14 s	20 s	n/a	20 s
Daisy	11 s	11 s	13 s	16 s	13 s
Raytracer	11 s	11 s	13 s	25 s	13 s
Elevator	14 s	14 s	17 s	33 s	18 s
jPapaBench	6 s	6 s	8 s	11 s	8 s
CDx	933 s	946 s	957 s	1327 s	934 s
Simple JBB	17 s	18 s	30 s	34 s	30 s

Table 8: Total running time of JPF and static analysis: configurations involving different pointer analyses with the context-sensitive field access analysis and the immutable fields analysis

	pointer analysis				
	none	ci ex	ci dd	cs ex	cs dd
CoCoME	29 s	32 s	35 s	79 s	34 s
CRE Demo	20 s	22 s	26 s	n/a	27 s
Daisy	4785 s	4304 s	4860 s	4350 s	27363 s
Raytracer	275 s	313 s	306 s	380 s	305 s
Elevator	1831 s	2192 s	1901 s	3080 s	1895 s
jPapaBench	7 s	8 s	10 s	13 s	10 s
CDx	18722 s	19285 s	20307 s	24164 s	37855 s
Simple JBB	191 s	216 s	223 s	252 s	548 s

of work for each query (see, in particular, the results for Daisy and CDx).

Object-sensitive pointer analysis is theoretically more precise than context-insensitive pointer analysis for programs that use collections, but also more expensive. On the CRE Demo benchmark, the object-sensitive points-to analysis ran out of memory (20 GB) after a few hours. On the other benchmarks, the field access analysis gave the same results

Table 9: Memory consumption of JPF and static analysis: configurations involving different pointer analyses with the context-sensitive field access analysis and the immutable fields analysis

	pointer analysis				
	none	ci ex	ci dd	cs ex	cs dd
CoCoME	3.6 GB	3.6 GB	5.0 GB	5.4 GB	5.0 GB
CRE Demo	2.4 GB	2.6 GB	5.1 GB	> 20 GB	5.1 GB
Daisy	3.6 GB	3.6 GB	5.0 GB	3.5 GB	5.0 GB
Raytracer	3.4 GB	4.2 GB	4.4 GB	5.2 GB	5.0 GB
Elevator	2.8 GB	5.3 GB	5.0 GB	5.5 GB	5.0 GB
jPapaBench	1.4 GB	1.4 GB	2.4 GB	2.4 GB	2.4 GB
CDx	14.2 GB	14.1 GB	15.3 GB	17.6 GB	15.0 GB
Simple JBB	4.1 GB	4.7 GB	5.3 GB	5.8 GB	5.3 GB

with object-sensitive points-to analysis as with context-insensitive points-to analysis, but the object-sensitive points-to analysis lead to a higher total running time.

The running time of the pointer analysis (Table 7) is generally small relative to the running time of JPF, as shown by the difference between the configurations with no pointer analysis and the configurations with pointer analysis. Most of the running time of the static analysis is spent by the immutable fields analysis. The only exception is the exhaustive object-sensitive pointer analysis, which significantly increases the running time of static analysis for most benchmarks. The memory consumption increases to a certain degree with the pointer analysis, but it is still feasible. The only exception is the object-sensitive points-to analysis, which runs out of memory on the CRE Demo benchmark.

8.4. Summary: Key Results and Findings

Figures 14 and 15 show the relative performance of supported analysis configurations, without pointer analysis and with different pointer analyses, respectively. For each configuration, the corresponding value in the graph represents the geometric mean of the total running times (in seconds) of JPF over the benchmark programs. The graph in Figure 14 averages all of the benchmarks except jPapaBench and CDx, on which some configurations failed to complete in the time limit of 12 hours. Similarly, the graph in Figure 15 averages all of the benchmarks except jPapaBench, CDx, and CRE Demo.

The values in the graphs highlight our main findings that we have identified based on the experimental results and the discussion in the previous section.

1. The combination of the context-sensitive field access analysis and the immutable fields analysis is the most effective analysis configuration that achieves the biggest performance improvements, and therefore we recommend this combination to be used with JPF.
2. Nevertheless, the combination of the immutable fields analysis with any field access analysis, including the context-insensitive one, achieves a very large reduction of the number of thread choices and the running time compared to the original JPF.
3. On the other hand, we do not recommend the use of points-to analysis techniques in the field access analysis, because they have a very small effect.

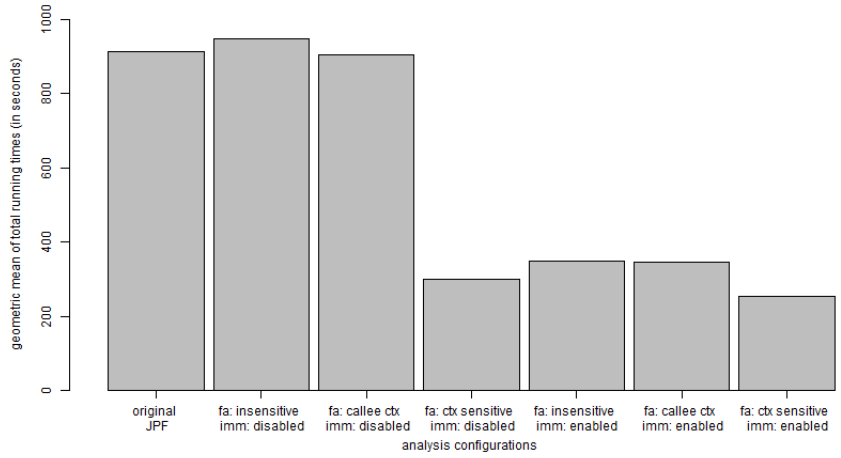

Figure 14: Geometric means of total running times for configurations without pointer analysis

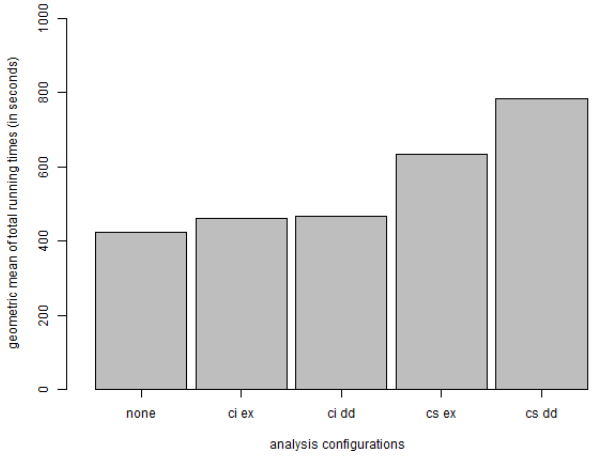

Figure 15: Geometric means of total running times for configurations involving different pointer analyses with the context-sensitive field access analysis and the immutable fields analysis

The static analyses enable JPF to verify more complex programs than it previously could. In particular, the benchmarks jPapaBench and CDx, which could not be verified even after 12 hours with the original JPF, can be verified in 5 seconds and 5.2 hours, respectively, using JPF with the static analysis. The static analysis also speeds up verification of the other benchmarks, by up to a factor of 8 (on Simple JBB).

The cost of the static analyses is small compared to their benefits. The total time

required by JPF with the static analysis is much lower than for the original JPF, and the memory consumption of static analyses is feasible.

8.5. Experiments with Dynamic Partial Order Reduction

We have also compared the benefits of our static analyses with a completely dynamic approach to partial order reduction (dynamic POR) that was proposed by Flanagan and Godefroid [15]. This approach determines instruction executions to be globally relevant in a different way than the original JPF and our static analyses. The key idea behind dynamic POR is to consider the execution of an instruction that accesses a particular heap object as globally-relevant only if some other thread actually accesses the same dynamic object in an observed dynamic execution. More specifically, the state space traversal procedure with dynamic POR consists of the following seven steps.

1. The algorithm completely explores an arbitrary single execution path e. During this process, the algorithm collects information about accesses to heap objects by program threads.
2. Using the information collected in the first step, the algorithm identifies a set O of objects that are truly accessed by multiple threads during execution of the path e.
3. For the path e and the set O, the algorithm computes the happens-before relation for accesses to objects in O that reflects their order in e. Executions of two instructions by different threads are in the happens-before relation if they access the same object.
4. The algorithm inspects all pairs of instructions that are executed by different threads in e and access the same object. For every such a pair (i, j) of instructions, where the execution of j precedes the execution of i in e, the algorithm uses the happens-before relation between the executions of i and j to decide whether to make a thread choice. A thread choice is created at the execution of j only if there is no instruction k such that k is executed by the same thread as i, the execution of k precedes i on e, and the execution of j happens before the execution of k. The motivation behind these conditions is to avoid creating redundant thread choices — the state space traversal procedure would create a thread choice at the execution of j when processing the pair (k, j), so it is not necessary to create another thread choice when processing the pair (i, j).
5. The algorithm finds an alternative execution path e_{alt} that contains some unexplored transitions leading from thread choices and fully explores the path, again collecting information about accesses to heap objects by program threads.
6. Based on that collected information, the algorithm creates additional necessary thread choices on the path e_{alt} using the same approach as in the steps 3 and 4.
7. The steps 5 and 6 are repeated until there are no more unexplored transitions leading from the existing thread choices, and no new thread choices have to be added based on the information collected for e_{alt}.

We have implemented the dynamic POR approach in JPF for the purpose of comparing it experimentally with the original JPF and our JPF-SA. Our implementation uses dynamic POR to make decisions about thread choices at field accesses, and preserves thread choices created by JPF at the execution of all other instructions (e.g., calls of the wait method and entries to synchronized blocks). As in the static field access analysis, we consider only the read-write dependencies between accesses to the same field.

A difference from [15], where the authors used dynamic POR together with a stateless search, is that we keep the state matching enabled in JPF. We adapted the approach to combining dynamic POR with stateful exploration that was proposed by Yang et al. [50]. In our implementation, when a program state s is visited for the first time, JPF explores the whole sub-graph of the state space that starts in s and records a set of all accesses to heap objects that can happen after s. When the state s is reached again during the search, JPF uses the set of recorded accesses to heap objects instead of exploring again the sub-graph of s.

Table 10: Thread choices created by JPF: comparing (1) the original JPF without any static analysis, (2) JPF-SA configured to use the context-sensitive field access analysis and immutable fields analysis, (3) dynamic POR, and (4) dynamic POR with context-sensitive field access analysis and immutable fields analysis

	original JPF	**JPF-SA**	**dynamic POR**	**dyn POR + static**
CoCoME	64580	8652	87	74
CRE Demo	47114	2919	2919	2919
Daisy	28120251	6292903	5447279	4819969
Raytracer	555126	351317	4	4
Elevator	10116121	2707528	414444	312416
jPapaBench	n/a	332	n/a	332
CDx	n/a	890872	n/a	21636
Simple JBB	498837	54606	599	599

The only goal of our experiments with the dynamic POR is to compare its precision against our static analyses, and therefore we report only the numbers of thread choices created by JPF. We do not provide the running times and memory consumption, because our implementation of the dynamic POR algorithm is not optimized for speed and low memory usage due to constraints imposed by the JPF API. Table 10 shows the results for combinations of the basic search procedures (original JPF and dynamic POR) with selected configurations of the static analysis. In particular, we evaluate four configurations: the original JPF, the original JPF combined with our static analyses (JPF-SA), the dynamic POR approach, and the combination of dynamic POR with our static analyses. In this last, combined configuration, the dynamic POR approach can avoid creating some thread choices thanks to the information provided by the static analyses. This applies mostly to instructions that access immutable fields.

The results are mixed: JPF combined with our static analyses works better than dynamic POR in some cases and vice versa. Of course, the combination of dynamic POR with our static analyses always performs best.

The dynamic approach to POR distinguishes individual (dynamic) heap objects, so it handles more precisely the cases where a given field of a specific heap object is accessed only by a single thread during the program execution. This occurs especially for fields that belong to objects that are stored as different elements of some array. In the case of the Raytracer benchmark, the number of thread choices is so small because each worker thread renders a part of the image and has its own (private) set of all relevant heap objects (scene elements, lights, rays) that is accessed exclusively by this worker thread during the program execution. Our static analyses cannot distinguish heap objects in

sets owned by different worker threads — even with a pointer analysis — and therefore they cannot eliminate thread choices in these cases.

On the other hand, a limitation of the dynamic POR algorithm is that it does not consider immutable fields. A write access to some field in the object's constructor is considered as globally-relevant by dynamic POR (assuming that read accesses by different threads happen later) even if the object is not reachable from the heap at the time of the write access. This occurs in our benchmark programs especially in the case of some fields of classes representing data transfer objects.

We also observed that state matching is critical to the good performance of dynamic POR. Our preliminary experiments showed that dynamic POR without state matching (as proposed in [15]) would be infeasible for our benchmarks.

The best results are achieved by the combination of dynamic POR with our static analyses. Our immutable fields analysis eliminates the unnecessary thread choices that dynamic POR cannot detect. Without the static analyses, dynamic POR on its own cannot verify the jPapaBench or CDx benchmarks in the 12 hour time limit.

9. Related Work

We focus our discussion of related work on the existing approaches to systematic exploration of the state space of multi-threaded programs. In particular, we consider techniques that are based on some variant of partial order reduction, or on the combination of static and dynamic analysis.

Table 11 compares the original JPF and the proposed approach (JPF-SA) with selected existing techniques in terms of their features and supported analyses. Each column represents a particular feature, and each row corresponds to a specific technique or a tool. For example, a tick symbol in the second column for a specific technique indicates that it performs some kind of static analysis. Columns 4-7 represent abilities that are important with respect to avoiding unnecessary thread choices during the state space traversal. This includes, for example, identifying thread-local field accesses. More details about the individual existing techniques are provided below in the following subsections.

Table 11: Comparing features supported by JPF, the proposed approach (JPF-SA), and selected existing techniques. The list of features considered here contains static analysis (SA), dynamic analysis (DA), identifying thread-local objects (TLO), identifying thread-local field accesses (TLF), ability to detect immutable fields (IMF), reductions based on locking disciplines (LD), and stateful exploration (ST).

	SA	DA	TLO	TLF	IMF	LD	ST
original JPF [20]	-	✓	✓	-	-	-	✓
Dwyer et al. [13]	✓	✓	✓	-	-	✓	✓
Flanagan and Godefroid [15]	-	✓	✓	✓	-	-	-
Gueta et al. [17]	-	✓	✓	✓	-	-	✓
Yang et al. [50]	-	✓	✓	✓	-	-	✓
Chen and MacDonald [9]	✓	✓	✓	✓	-	-	✓
Brat and Visser [5]	✓	✓	✓	✓	-	-	✓
Choi et al. [11]	✓	✓	✓	✓	-	-	-
Agarwal et al. [2]	✓	✓	✓	✓	-	✓	-
proposed approach (JPF-SA)	✓	✓	✓	✓	✓	-	✓

There also exist many other techniques for detecting concurrency errors [14, 22, 31, 32, 33] that are based mostly on static analysis (e.g., aliasing and may-happen-in parallel analysis). In addition, we describe program analysis techniques that have goals similar to the static analyses presented in this paper.

9.1. Partial Order Reduction Strategies

Several techniques that improve partial order reduction strategies have been proposed in recent years. Here we discuss only techniques that use a dynamic approach to POR and that are designed for the systematic traversal of the explicit state space of multi-threaded programs.

Dwyer et al. [13] proposed multiple strategies for partial order reduction that can be used in model checking of concurrent object-oriented programs. The strategies form two groups. Those in the first group use escape analysis to detect thread-local heap objects, which are reachable only from a single thread, and based on that identify thread-local actions (e.g., field accesses). The second group contains reduction techniques that exploit locking disciplines. Here we discuss only strategies that involve escape analysis, as the other are not specifically related to our approach.

Two variants of escape analysis were proposed in [13]: static and dynamic. The static escape analysis conservatively identifies objects that are thread-local during the whole program execution. The dynamic escape analysis is performed on-the-fly during the state space traversal. After every transition, it analyzes the heap to identify objects that are thread-local in the current dynamic state. Reachability information is acquired through a systematic traversal of all possible chains of object references from local variables of all threads and from static fields. The dynamic escape analysis is more precise than the static escape analysis because it identifies a heap object as thread-local in the specific heap that arises during the dynamic traversal, rather than in all possible heaps. For example, it can determine that an object is thread-local during initialization and becomes shared later during the program execution.

The original JPF already uses the more precise dynamic escape analysis.

A limitation of the dynamic escape analysis is that it recognizes thread locality only at the granularity of whole objects. The JPF-SA approach is more precise because it distinguishes individual fields of an object. Specifically, an access to the field $o.f$ can be precisely identified as thread-local by the field access analysis even when the target object o is reachable by multiple threads, as long as only a single thread will ever access the field f of o in the future.

The immutable fields analysis from Section 6 also uses (static) escape analysis, but for a different purpose. A field is immutable if it is never written after its containing object has escaped to the heap. The flow-sensitive static escape analysis that we presented in Section 6.2 computes, for each code location, the set of variables pointing to objects that may have escaped to the heap. In contrast, the static escape analysis described in [13] only identifies a flow-insensitive set of objects that are thread-local throughout the whole program execution. The more precise dynamic escape analysis could not be used for identifying immutable fields because the list of immutable fields must be available before JPF-SA starts exploring the program state space. Note also that objects reachable from multiple threads can have fields that are immutable, and vice versa.

We describe the approach to dynamic POR by Flanagan and Godefroid [15] in detail, including the experimental comparison with our work, in Section 8.5. Here we only highlight the most important features. The search procedure explores individual execution paths one-by-one using only dynamic analysis. The procedure is stateless, i.e., it does not remember already visited states, and therefore it has to explore each path fully up to its end state. Information about objects and fields truly accessed by multiple threads is collected during the analysis of each path. Note that field accesses on shared objects collected in this way are globally relevant. When an execution path has been explored, then the search procedure computes the happens-before relation for events on the given path (including all the globally-relevant field accesses) and uses the relation to identify additional points where it has to create a thread scheduling choice. This process stops when there are no thread choices with unexplored outgoing transitions. If there were such a thread choice remaining, then that particular unexplored transition would correspond to an alternative execution path that has not been analyzed yet.

Gueta et al. [17] proposed dynamic cartesian POR, which behaves in a similar way as the state space traversal procedure used in JPF. In each state that is associated with a thread scheduling choice, the cartesian POR algorithm generates a vector of finite instruction sequences that contains one sequence for each thread runnable at that state. Each sequence consists of any number of instructions with a guaranteed thread-local execution from the given state followed by a single instruction whose execution will be globally-relevant. Given a vector of instruction sequences, one of them is selected nondeterministically, then all instructions forming the sequence are executed, and a new thread choice is created at its end. State space exploration continues from the state associated with the new thread choice. Note that the whole instruction sequence is computed by the cartesian POR algorithm before it is executed. In contrast, JPF-SA determines whether an instruction execution is globally-relevant on-the-fly immediately before it executes the instruction. The main difference between cartesian POR [17] and the original approach to dynamic POR [15] is that cartesian POR generates vectors of instruction sequences while moving forward along a specific execution path, but dynamic POR identifies points where to add thread choices retroactively after it completely explores a specific execution path and computes the happens-before relation of that path. Other differences are that cartesian POR handles cycles in the state space, and that it stores already visited states to avoid exploring them repeatedly.

Several existing techniques combine state space traversal based on dynamic POR with state matching. In the approach by Yang et al. [50] that we adapted to compare JPF-SA with DPOR, the search procedure dynamically computes the happens-before relation for globally-relevant operations, and also records globally-relevant actions that can happen after a given state. The happens-before relation is maintained using a graph representation. Backward propagation of recorded globally-relevant actions is used to properly update the happens-before relation when a previously visited program state is reached. Another stateful variant of dynamic POR, where all transitions reachable from each state are stored to correctly identify dependent transitions, has been proposed by Ranganath [39]. He also considers an extension in which statically computed dependence information is used together with the dynamic state space traversal algorithm.

9.2. Combining Static and Dynamic Analysis

We are aware of several approaches that combine JPF with static analysis for the purpose of efficient verification of multi-threaded Java programs.

The key idea proposed by Chen and MacDonald [9] is to address state explosion by exploring just one thread interleaving for each sequence of globally-relevant actions. The set of globally-relevant actions considered by this approach includes read and write accesses to shared variables (e.g., fields of objects reachable from multiple threads) and thread synchronization operations. Static analysis is used to determine possible sequences of globally-relevant actions from the code of program threads. For each sequence of actions, JPF attempts to find a thread interleaving that corresponds to the sequence by systematic exploration of the program state space, and checks the interleaving for concurrency errors. If there is no such thread interleaving, then JPF reports the sequence as infeasible. Note that the static analysis may, due to the inherent imprecision, generate many infeasible action sequences. Experimental results published in [9] show a significant gain in performance for small benchmark programs, but it remains to be seen how much this approach would help on more complex Java programs, such as Simple JBB. Another limitation is the missing support for dynamically created threads.

A method described by Brat and Visser [5] combines JPF and static alias analysis in an iterative fashion. The key concept is that of unsafe statements, which correspond to globally-relevant actions. The process starts with all statements optimistically identified as safe and with empty aliasing information. In each step, static analysis computes a more precise set of unsafe statements based on more precise aliasing information that is acquired from the dynamic program states. A statement is considered unsafe if its execution possibly accesses a shared variable (e.g., an object field). JPF then explores all thread interleavings involving executions of the newly identified unsafe statements and updates the aliasing information on-the-fly. Iteration converges to the point where the aliasing information is precise and complete, and all unsafe statements are recognized by the static analysis — then it is guaranteed that JPF has explored all interleavings of executions of unsafe statements. The published experimental results show that this approach was evaluated on one small benchmark program of 20 lines of code, for which the state space size was reduced by half compared to the original JPF. No results for larger programs are available, and the total running time of JPF combined with static analysis is also not provided. Therefore, it is not clear how efficient this approach would be for more complex Java programs, for which static analysis takes more time and significantly more memory.

Other techniques that combine static and dynamic analysis target languages other than Java or they do not involve JPF. For example, the technique for efficient detection of data races proposed by Choi et al. [11] uses static analysis to identify statements that may participate in a data race and dynamic analysis to check executions of these statements. The technique proposed by Agarwal et al. [2] uses type discovery to identify potentially unsafe statements whose executions are precisely checked by the subsequent dynamic analysis.

9.3. Static Analysis

As already discussed in Section 4, the static analyses proposed in this paper — the field access analysis and immutable fields analysis — compute information that is similar

to interference dependence [24, 39, 40] and data-flow dependency for object fields [26]. Given a pair of field access instructions in the program, a dependence analysis determines whether they may access the same field instance at run time. In constrast, given only the first field access, our field access analysis determines whether *any* future field access may access the same field instance. Therefore, in theory, the information computed by our field access analysis could be derived from a complicated dependence analysis that finds dependences between multiple threads. However, field access analysis is much simpler and more efficient than even a single-threaded dependence analysis, let alone a multi-threaded one. This makes field access analysis much more amenable to context sensitivity, especially the sensitivity to the dynamic run-time context that we have used in the JPF-SA approach. Here we provide additional details about the existing approaches to computing dependency information.

Ranganath and Hatcliff [40] proposed an efficient method for precise slicing of concurrent programs that is based on escape analysis and interference dependence. The method consists of three steps: computing an escape analysis, building an interference dependency graph, and the actual program slicing. The static escape analysis was designed as an extension of [42] that provides more precise information about sharing of objects among threads. The results of the escape analysis are used to identify heap objects reachable from multiple threads. The interference dependency graph built in step 2 then contains only edges between statements (nodes) that are executed by different threads and access the same object. Although the results of our field access analysis represent information similar to the interference dependence, the analysis itself is very different from the algorithm used to build a dependency graph. The approach to building the interference dependence graph, described in [40], statically checks for each pair of field access statements only whether the given two statements are executed by different threads, and whether they access the same field of an object shared by both threads according to the escape analysis.

Program slicing techniques may also use data-flow dependencies among program variables, including fields of heap objects. A technique designed by Liang and Harrold [26] targets object-oriented programs. It works in two steps: it builds a graph that captures the dependencies among variables (e.g., object fields and methods call parameters) and other entities that appear in object-oriented programs, and it then performs reachability analysis over the graph to create a program slice. Specifically related to our work, this technique can identify the data-flow dependency of a read access to a particular field f on a previous write access to the field. However, the technique does not consider multiple threads (concurrency), i.e., its results apply only to single-threaded programs. It does not consider dependencies of write accesses on read accesses that would be needed to capture the interference between concurrent threads.

The rest of this subsection describes other static analyses related to our field access analysis and immutable fields analysis.

A pointer analysis that proceeds in two distinct passes, like the callee-context-sensitive field access analysis, was proposed by Nystrom et al. [35]. To define the analysis, they took advantage of the dual way in which context sensitivity improves the precision of a given static analysis: the effects of specific callers can be distinguished in callees, and the effect of a given callee can be specialized at each call site.

The static analysis proposed by Naik and Aiken [31] computes a must not alias

relation for reference variables. The relation could be used instead of points-to analysis to distinguish objects of the same class, i.e., to show that different threads access the field f of different objects.

Guyer et al. [18] and Cherem and Rugina [10] designed static analyses that detect objects that will never be accessed in the future, yet are still reachable via a dynamic heap traversal in a particular dynamic program state. The idea is similar to our field access analysis. However, these analyses were applied to improve memory management rather than verification. In particular, the static information was used to identify objects that can be freed early in order to reduce the load on a dynamic garbage collector.

Unkel and Lam [48] define stationary fields, a notion similar to immutable fields, and present a static analysis that detects a conservative subset of stationary fields. A stationary field is defined in [48] as one that is never written after it is read by any thread, while our definition of immutable fields is independent of reads. Although the definitions are similar, it is theoretically possible for a field to be stationary but not immutable according to our definition, or vice versa. An example of a stationary but not immutable field is one that is never read, but escapes the current thread and is written. An example of an immutable field that is not stationary is one that is first read and then written, without ever having escaped the current thread. Like our immutable fields analysis, the stationary field analysis considers the point where an object escapes to the heap as the end of the initialization phase. The stationary fields analysis is conservative and tracks field accesses precisely only during the initialization phase — a field of an object is always identified as non-stationary if the field is written after the object becomes reachable from the heap, even if there is actually no read of the field before the aforementioned write access.

Several analyses have been designed to detect pure methods [44, 4, 41]. A pure method does not modify fields of objects existing before the method is called. Unlike our analysis that detects immutable fields, purity analyses find areas of the program in which mutable fields are not mutated.

Many variants of escape analysis have been designed and used in the past. For example, Ruf [42] uses the results of a static escape analysis to remove unnecessary thread synchronization. If a given heap object is certainly thread-local during the whole program execution, then field accesses on the object do not have to be guarded with a synchronization operation. Nevertheless, the static escape analysis by Ruf computes information that holds for the entire object lifetime (program execution), and therefore has limited precision. We use a more precise variant of escape analysis as the first and second stage of the immutable fields analysis (Section 6). It can compute object-escape information specifically for each code location, but it is also more complex.

Zhang and Ryder [51] proposed a precise data reachability analysis that is related to steps 1 and 2 of our immutable fields analysis (i.e., computing method summaries and escape analysis). The data reachability analysis computes, for each method, the set of objects that are created in the method and become reachable outside of the method (e.g., in another thread) before the method ends.

10. Conclusion

We have proposed an optimization of the partial order reduction technique used by JPF. The optimization is based on static analyses and particularly targets the verifica-

tion of multi-threaded Java programs in which threads interact via field accesses on heap objects. The key challenge in this context is to precisely identify globally-relevant field accesses, at which there must be a thread scheduling choice in order to guarantee exploration of all thread interleavings with distinct behavior. We designed two analyses that identify globally-relevant field accesses more precisely than the dynamic escape analysis used in JPF. Specifically, the partial order reduction technique in JPF uses only the information in the current program state due to the on-the-fly state space construction, and therefore it must use a conservative approach based on heap reachability to decide whether to make a thread choice. Our analyses provide information about field accesses that may occur on any execution path between a particular state and the end of the program lifetime, which is then used to eliminate many unnecessary thread choices.

Experimental results show that the combination of JPF with the proposed analyses improves the performance and scalability of verification quite significantly. In particular, the benchmarks jPapaBench and CDx could be verified only with our analyses. The total running time is much lower also for all the other benchmark programs, up to a factor of 8 for Simple JBB. We found that the biggest performance improvements were achieved by the context-sensitive field access analysis together with the immutable fields analysis and that usage of points-to analyses with field access analysis has a very small effect. We also showed that the proposed analyses can improve the performance of the state space traversal procedure based on dynamic POR [15].

Although we focused mainly on the approach to POR used by JPF in this paper, we believe that the proposed optimizations could also be applied in other tools for state space traversal of multi-threaded programs written in object-oriented languages, such as MoonWalker [6] and CHESS [29], even if they perform a stateless search.

Acknowledgements

This work was partially supported by the Grant Agency of the Czech Republic project 13-12121P, and partially supported by the Natural Sciences and Engineering Research Council of Canada. We would also like to thank Jan Vitek and Tomas Kalibera for suggesting the idea that we realized in the field access analysis, and Peter Mehlitz for his timely help with configuring and extending Java Pathfinder.

References

[1] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil. Component Reliability Extensions for Fractal Component Model. `http://d3s.mff.cuni.cz/projects/formal_methods/ft/`, 2006.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S.D. Stoller. Optimized Run-time Race Detection and Atomicity Checking Using Partial Discovered Types. In Proceedings of ASE 2005, ACM.

[3] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, 1994.

[4] S. Artzi, A. Kiezun, D. Glasser, and M. D. Ernst. Combined Static and Dynamic Mutability Analysis. In ASE 2007, ACM.

[5] G. Brat and W. Visser. Combining Static Analysis and Model Checking for Software Analysis. ASE 2001, IEEE CS.

[6] N.H.M. Aan de Brugh, V.Y. Nguyen, and T.C. Ruys. MoonWalker: Verification of .NET Programs. In TACAS 2009, LNCS, vol. 5505.

[7] L. Bulej, T. Bures, T. Coupaye, M. Decky, P. Jezek, P. Parizek, F. Plasil, T. Poch, N. Rivierre, O. Sery, and P. Tuma. CoCoME in Fractal. In The Common Component Modeling Example, LNCS, vol. 5153, 2008.
[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema and J.-B. Stefani, The FRACTAL Component Model and its Support in Java. Software: Practice and Experience, 36 (11-12), 2006.
[9] J. Chen and S. MacDonald. Testing Concurrent Programs using Value Schedules. ASE 2007, ACM.
[10] S. Cherem and R. Rugina. Compile-time Deallocation of Individual Objects. In ISMM 2006, ACM.
[11] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In Proceedings of PLDI 2002, ACM.
[12] E. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 2000.
[13] 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.
[14] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In Proceedings of SOSP'03, ACM.
[15] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Proceedings of POPL 2005, ACM.
[16] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, 1996.
[17] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian Partial-Order Reduction. In SPIN 2007, LNCS, vol. 4595.
[18] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-Me: A Static Analysis for Automatic Individual Object Reclamation. In PLDI 2006, ACM.
[19] Java Grande Forum Benchmarks, `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html`
[20] Java Pathfinder, `http://babelfish.arc.nasa.gov/trac/jpf/`
[21] jPapaBench, `http://d3s.mff.cuni.cz/~malohlava/projects/jpapabench/`
[22] V. Kahlon, S. Sankaranarayanan, and A. Gupta. Semantic Reduction of Thread Interleavings in Concurrent Programs. In TACAS 2009, LNCS, vol. 5505.
[23] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CDx: A Family of Real-Time Java Benchmarks. In JTRES 2009, ACM.
[24] J. Krinke. Static Slicing of Threaded Programs. In PASTE 1998, ACM.
[25] O. Lhoták and L. Hendren. Scaling Java Points-to Analysis Using Spark. In CC 2003, LNCS, vol. 2622.
[26] D. Liang and M. J. Harrold. Slicing Objects Using System Dependence Graphs. In ICSM 1998, IEEE CS.
[27] A. Milanova, A. Rountev, and B. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. ACM TOSEM, 14(1), 2005.
[28] Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.
[29] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In OSDI 2008, USENIX.
[30] N.A. Naeem and O. Lhoták. Faster Alias Set Analysis Using Summaries. In CC 2011, LNCS, vol. 6601.
[31] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In Proceedings of POPL 2007, ACM.
[32] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In PLDI 2006, ACM.
[33] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In Proceedings of ICSE 2009, IEEE CS.
[34] V.Y. Nguyen and T.C. Ruys. Memoised Garbage Collection for Software Model Checking. In TACAS 2009, LNCS, vol. 5505.
[35] E. M. Nystrom, H.-S. Kim, and W. W. Hwu. Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. SAS 2004, LNCS, vol. 3148.
[36] Parallel Java Benchmarks, `http://code.google.com/p/pjbench`
[37] P. Parízek and O. Lhoták. Identifying Future Field Accesses in Exhaustive State Space Traversal. In ASE 2011, IEEE CS.
[38] S. Qadeer. Daisy File System. Joint CAV/ISSTA special event on specification, verification and testing of concurrent software, 2004.

[39] V.P. Ranganath. Scalable and Accurate Approaches for Program Dependence Analysis, Slicing, and Verification of Concurrent Object Oriented Programs. PhD thesis, Kansas State University, 2006.
[40] V.P. Ranganath and J. Hatcliff. Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs. CC 2004, LNCS, vol. 2985.
[41] A. Rountev. Precise Identification of Side-Effect-Free Methods in Java. ICSM 2004, IEEE CS.
[42] E. Ruf. Effective Synchronization Removal for Java. In PLDI 2000, ACM.
[43] SPEC JBB 2005 benchmark, `http://www.spec.org/jbb2005/`
[44] A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. VMCAI 2005, LNCS, vol. 3385.
[45] M. Sridharan and R. Bodik. Refinement-based Context-sensitive Points-to Analysis for Java. PLDI 2006, ACM.
[46] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven Points-to Analysis for Java. OOPSLA 2005, ACM.
[47] S.D. Stoller. Model-checking multi-threaded distributed Java programs. STTT, 4(1), 2002.
[48] C. Unkel and M.S. Lam. Automatic Inference of Stationary Fields: A Generalization of Java's Final Fields. POPL 2008, ACM.
[49] WALA, `http://wala.sourceforge.net/`
[50] Y. Yang, X. Chen, G. Gopalakrishnan, and R.M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. SPIN 2008, LNCS, vol. 5156.
[51] W. Zhang and B.G. Ryder. Automatic Construction of Accurate Application Call Graph with Library Call Abstraction for Java. Journal of Software Maintenance and Evolution, 19(4), 2007.