Identifying Future Field Accesses in Exhaustive State Space Traversal

Pavel Parízek, Ondřej Lhoták
David R. Cheriton School of Computer Science, University of Waterloo
{pparizek,olhotak}@uwaterloo.ca

Abstract—**One popular approach to detect errors in multi-threaded programs is to systematically explore all possible interleavings. A common algorithmic strategy is to construct the program state space on-the-fly and perform thread scheduling choices at any instruction that could have effects visible to other threads. Existing tools do not look ahead in the code to be executed, and thus their decisions are too conservative. They create unnecessary thread scheduling choices at instructions that do not actually influence other threads, which implies exploring exponentially greater numbers of interleavings.**

In this paper we describe how information about field accesses that may occur in the future can be used to identify and eliminate unnecessary thread choices. This reduces the number of states that must be processed to explore all possible behaviors and therefore improves the performance of exhaustive state space traversal. We have applied this technique to Java PathFinder, using the WALA library for static analysis. Experiments on several Java programs show big performance gains. In particular, it is now possible to check with Java PathFinder more complex programs than before in reasonable time.

Index Terms—**exhaustive state space traversal, field accesses, state explosion, concurrency, Java PathFinder, WALA**

I. Introduction

One group of techniques for detecting errors in multi-threaded programs is based on exhaustive state space traversal. All interleavings of program threads are checked for property violations. Well-known tools that belong to this category are Java PathFinder (JPF) [15] and CHESS [21]. Although such tools implement many optimizations, checking all thread interleavings is still time-consuming, so these tools can currently be applied only to relatively small programs.

A popular strategy, used in JPF, is to construct the program state space on-the-fly and create thread scheduling choices only at instructions that either change or depend on the global state that is visible to other threads. These instructions include reads and writes of shared data and synchronization instructions. Instructions with only thread-local effects are considered as independent, so no thread choices at these instructions are explored [10]. This technique of ignoring the order of thread-local operations is called partial order reduction.

One limitation of this strategy is that the state space traversal algorithm does not see ahead on the execution path and does not know what actions each thread may perform in future. The decision whether an instruction may have global effects visible to other threads is performed using only information in the current state and execution history, and thread scheduling choices must be made at all instructions that may potentially have global effects. This strategy is too conservative and leads to inefficient state space traversal for two reasons: many unnecessary thread scheduling choices may be created in the state space, and the set of explored thread interleavings may also include ones that differ only in the ordering of actions that actually do not influence behavior of other threads. An important category of globally visible actions are accesses to shared objects. Due to inability to see ahead, a common solution is to consider an object shared if it is reachable via pointers from multiple threads (even though the threads may not actually ever follow those pointers to access the object).

Contribution. We introduce a static analysis that determines, for each field access in the program, whether the same field may be accessed again in the code yet to be executed by other threads. Using the static analysis results, it is possible to soundly eliminate many unnecessary thread scheduling choices that would otherwise be created, and thus reduce the number of states and transitions that must be processed to cover all possible behaviors of the given program. The static analysis enables more aggressive partial order reduction.

Our approach is a hybrid between static analysis and exhaustive state space traversal: static analysis is used to reduce the state space size and, at the same time, information available in the dynamic program state provided by JPF is combined with the static analysis results to improve precision.

We have implemented our static analysis using the WALA library [30] and applied it to JPF. Experiments performed on several non-trivial Java applications show large reductions in the number of states 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 8 hours, but our static analysis made it possible to verify them in 12 and 55 minutes.

II. 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).

```java
class Employee {
  String name;
  Integer salary;
  Employee(String name, Integer salary) {
    this.name = name;
    this.salary = salary;
  }
}
class Company {
  List employees = new ArrayList();
  void addEmployee(String name, Integer salary) {
    Employee emp = new Employee(name, salary);
    employees.add(emp);
  }
  void addEmployee(Employee emp) {
    employees.add(emp);
  }
  void printEmployeeNames() {
    for (Employee e : employees) out.println(e.name);
  }
  void setEmployeeSalary(String eName, Integer nSalary) {
    for (Employee e : employees) {
      if (e.name.equals(eName)) e.salary = nSalary;
    }
  }
  Integer getEmployeeSalary(String eName) {
    for (Employee e : employees) {
      if (e.name.equals(eName)) return e.salary;
    }
  }
}
class FirstWorker extends Thread {
  private Company cmp;
  FirstWorker(Company cmp) {
    this.cmp = cmp;
  }
  public void run() {
    cmp.printEmployeeNames();
    Integer r = cmp.getEmployeeSalary("john");
    cmp.setEmployeeSalary("john", 150);
    cmp.printEmployeeNames();
  }
}
class SecondWorker extends Thread {
  private Company cmp;
  SecondWorker(Company cmp) {
    this.cmp = cmp;
  }
  public void run() {
    Integer r = cmp.getEmployeeSalary("paul");
    cmp.addEmployee("mark", 80);
    cmp.printEmployeeNames();
  }
}
public static void main(String[] args) {
  Company cmp = new Company();
  cmp.addEmployee("john", 100);
  Employee emp = new Employee("paul", 120);
  cmp.addEmployee(emp);
  FirstWorker w1 = new FirstWorker(cmp);
  SecondWorker w2 = new SecondWorker(cmp);
  w1.start(); w2.start(); w1.join(); w2.join();
}
```

Fig. 1. Java program used as a running example

III. Java PathFinder

Java PathFinder (JPF) [15] 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. JPF constructs the program state space on-the-fly and at the end of each transition, it checks all configured properties (which may be built-in properties such as race conditions and deadlocks, or user-specified application-specific properties). A transition is a sequence of bytecode instructions executed by a single thread; only the first instruction in the sequence represents the non-deterministic choice. At every transition boundary, JPF saves the current JVM state in a serialized form for the purpose of backtracking and state matching. The complete JVM 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 JVM state. Native methods are either completely re-implemented in Java, or their execution is delegated to the underlying JVM on which JPF is executing and state changes are done using a JPF API. Currently, JPF supports only a subset of the Java library classes that contain native methods.

Scheduling choices are created by JPF before execution of each instruction with globally visible effects. This includes instructions that change the global state (e.g., write a new value to a field of a shared heap object), thus influencing behavior of other threads, and instructions that depend on the global state (e.g., read the current value of some field of a shared heap object). The next thread to run from a JVM state that corresponds to a scheduling choice point is selected from all threads that are runnable in the state. Technically, a scheduling choice at an instruction i has two outcomes with respect to execution order on the current path:

1) the current thread t continues and executes the instruction i at the beginning of the next transition;
2) another thread is scheduled and executes its code; later when the thread t is scheduled again, it executes the instruction i from the state updated by other threads.

This allows JPF to systematically explore all thread interleavings that correspond to different sequences of globally-visible actions. If the only difference between two thread interleavings is the order of purely thread-local instructions, only one of the thread interleavings is explored. Note that JPF does not support the current Java Memory Model - it models only sequential consistency of memory accesses.

Due to the on-the-fly construction of the program state space, JPF does not see ahead in program execution — it does not know what actions each thread may perform in the rest of the program's lifetime — and therefore it must decide whether

an instruction has global effects using only the information in the current state and execution history. JPF works in a conservative way and creates a thread scheduling choice at every instruction that could influence the behavior of any other thread. Consequently, JPF may explicitly traverse two interleavings that differ only in the ordering of instructions that could have global effects, but actually do not.

An important category of globally visible actions are accesses to fields of shared objects. JPF conservatively considers an object shared if it is reachable via a chain of references from a static field, or from local variables of multiple threads. A thread scheduling choice is created at a field access instruction if all the following conditions hold: (1) the target object is reachable from multiple threads, (2) multiple threads are runnable, and (3) access to the field is not controlled by the same lock in all threads. When executing an instruction that reads or writes a field f of an object o, JPF does not know whether any thread may access $o.f$ in the future; thus it conservatively assumes that another thread may access the field if the object is reachable from another thread. An object reachable from multiple threads is determined to be shared even if its fields are accessed only by a single thread.

In the running example (Figure 1), JPF will create thread scheduling choices at instructions accessing all fields of classes defined in the program, because all objects are reachable from each thread, and there is no synchronization. In theory, JPF would create scheduling choices also at instructions accessing internal fields of collection classes, but the default configuration of JPF does not consider field accesses inside the Java core library as transition boundaries. Besides scheduling choices at field access instructions, JPF will also create choices at the calls of the Thread.start and Thread.join methods (in the example), since they change thread status.

IV. Field Access Analysis

The key idea of our approach is to use static analysis to more precisely determine which fields of which objects may be accessed in the future by executing threads, rather than assuming that all fields of all reachable objects will be accessed. We use this information to eliminate some of the unnecessary scheduling choices that JPF creates during state space traversal. The static analysis is performed before JPF begins, and its results are used during JPF's exploration of the program state space.

The most general question that the static analysis must answer is: given a (dynamic) program state and an object o and field f, will any thread other than the one currently executing read or write $o.f$ in the future? The static analysis can only give a conservative over-approximation of the answer to this question. However, we have designed an analysis that gives more precise answers than the original JPF, which always answers "yes" if o is reachable from another thread.

We present several variations of the static analysis of increasing complexity and precision. First, in Section IV-A, we present the basic field access transfer functions within a simple 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. The second analysis, described in Section IV-B, 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, described in Section IV-C, 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 call stack containing the currently executing instruction in each stack frame, the analysis considers only control flow paths that start with that specific call stack. In particular, the analysis is more precise than the second analysis for field accesses occurring after return from the method containing the designated program point p, since the 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 IV-D, uses points-to information to also determine the objects whose fields are to be accessed. Finally, in Section IV-E, we describe an additional analysis, detection of immutable fields, which we have found to improve precision on several common idioms. All variants of the static analysis operate on one thread at a time, calculating the set of fields that may be accessed in only that static thread.

JPF uses the analysis results during state space exploration as follows. When the Java program is in a given state and the next instruction i to be executed is a read (write) field access on $o.f$, JPF queries the analysis results separately for each thread instance (dynamic thread) that is executing in the given dynamic state of the Java program and combines the results to determine whether any dynamic thread may write (read) $o.f$ in future. If yes, JPF creates a thread scheduling choice before the current field access, in order to consider the interleaving in which that "future" access occurs before the current access at i. If no, then the effect of the current access at i is not observable by any other thread, and no thread scheduling choice is needed. We distinguish reads and writes because only write-read and read-write dependencies require a thread scheduling choice. The order of accesses does not affect program execution in other cases. A write-write dependency must be considered only if its result is read later, but in that case there will be a write-read dependence too.

Note that JPF combined with the static field access analysis still explores all execution paths that correspond to different interleavings of globally visible actions and therefore finds the same set of errors as the original JPF. For each field access instruction, a thread scheduling choice is created at the instruction if another thread may access the same field in future — this way it is ensured that both orderings of the field accesses in different threads are explored by JPF.

A. Context-insensitive Field Access Analysis

The field access analysis is an inter-procedural backward flow-sensitive analysis which operates upon the inter-procedural control-flow graph (ICFG) of a given Java program. The analysis computes, for each point p in the code of a thread t, the set of fields that may be accessed by the thread t in the fragment of its execution path from p until the end of its lifetime. 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. 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.

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

Fig. 2. Transfer functions for the context-insensitive analysis

The transfer functions for the context-insensitive field access analysis are shown in Figure 2. These transfer functions are applied in a Kildall-style fixpoint dataflow analysis on the ICFG. The analysis is a backward analysis similar to upward-exposed uses analysis [20, Section 8.3], which computes at each program point the future uses that may read the current value of each variable. The merge operator is set union. 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. Whenever the analysis encounters a read or write instruction, the transfer function adds the field accessed to the set of reads or writes. The transfer functions for method calls and returns are the identity. In this version of the analysis, calls and returns are treated context-insensitively like any other branch instruction.

B. Callee-context-sensitive Field Access Analysis

The simple analysis presented thus far is very imprecise due to two kinds of context insensitivity, which we illustrate using the example in Figure 3. 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.

Suppose that JPF queries the analysis at the point 1. The field access sets at this point would include all fields that may be accessed after the point 2 in method c. If method c is called from multiple call sites, the result at 2 (and therefore also at 1) must include all field accesses occurring after *all* of these call

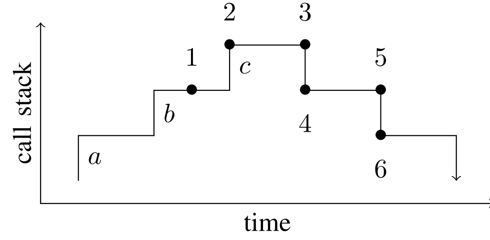

Fig. 3. Example evolution of call stack

sites. Yet since we are querying the analysis at 1, we know that in this case, method c will return to method b, and therefore these field accesses at other call sites of c are superfluous. 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 control returns to method a. However, if we tell the static analysis only that we want results at the static program point 1 in method b, it must conservatively consider *all* of the call sites that control could return to at the end of method b. Therefore, the field access sets at point 5 (and therefore also at point 1) will contain superfluous 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 rest of this section, we modify the field access analysis to be callee-context-sensitive. In the next section, we will show how to use dynamic information to achieve caller-context sensitivity.

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 4. They are similar to the context-insensitive ones from Figure 2, 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, in the example, 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.

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 phase are shown in Figure 5. 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

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

Fig. 4. Transfer functions for the first phase

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.

Instruction	Transfer function
	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]

Fig. 5. Transfer functions for the second phase

Considering the addEmployee(String, Integer) method from the running example (lines 11-14), the result of the first phase for the entry node includes all fields accessed in the addEmployee method and in the constructor of the Employee class. The result of the second phase for the method's exit node already contains field accesses that may occur after the various sites that call of addEmployee(String, Integer), including both read and write access to Employee.name and write access to Employee.salary in the constructor call at line 58, and these are propagated through the method.

C. Context-sensitive Field Access Analysis

Caller-context insensitivity is still a source of imprecision. For example, inside the printEmployeeNames method, the analysis presented so far does not distinguish whether it was called from the first or second call site in FirstWorker.run.

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; thus 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 is computed by JPF simply as the union of the phase one results at all of the points p_i: $\bigcup_{i=0}^{n}$ 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. In the example execution in Figure 3, when execution is at point 1, the call stack contains the 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 printEmployeeNames from FirstWorker.run. The result at any point p inside printEmployeeNames depends on the call site in FirstWorker.run from which it was called. When called from the first site (line 38), the result at p contains `rw` $Employee.salary$ since the calls to getEmployeeSalary and setEmployeeSalary follow on any execution path. When called from the second site (line 41), 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 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 then always creates a thread choice at the write access to the Employee.salary field in the constructor of the Employee class, if the program counter in the FirstWorker thread precedes the call to the getEmployeeSalary method, which contains a read access of the salary field.

D. Using Pointer Analysis

The precision of the field access analysis can be improved by using points-to information to distinguish object instances. A thread scheduling choice is needed at a read (write) of a field of an object only in the presence of 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 thread choice can be eliminated. We have experimented with four variants of points-to analysis: an exhaustive context-insensitive analysis [4], [18], an exhaustive context-sensitive analysis that uses receiver objects as context [19], and context-insensitive and context-sensitive

versions of a demand-driven analysis [29], [28]. The pointer analysis is always performed on the program before JPF starts.

1) Context-insensitive Points-to Analysis: The 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. The field access analysis is extended to make use of points-to information as follows. It 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 $p.f$, 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 future the field f of an object o allocated at the site a. When JPF executes 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 no thread scheduling choice is required.

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.

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.

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 12 and 58. However, no additional thread scheduling choices will be eliminated by JPF when the field access analysis combined with pointer analysis is used, because all local variables of the type Employee in all methods except addEmployee may point to instances allocated at both sites.

3) Demand-Driven Points-to Analysis: Exhaustive points-to analysis is expensive because it computes points-to sets for all variables, even though only the points-to sets of a small subset of variables may actually be needed by JPF. 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.

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. For illustration, in the running example the local variables Company.addEmployee.emp and Company.printEmployeeNames.e (and many others) refer to instances of the Employee class. When JPF 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. 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.

E. Immutable Fields

It is possible to further reduce the number of thread scheduling choices made by JPF by detecting the common idiom of immutable fields: after the object containing the field has been initialized, the field is never changed. More precisely, we define a field as immutable if it is accessed only by its creating thread before its constructor completes, and it is never written after its constructor has completed. At an access of a field known to be immutable, no thread scheduling choice is required. If the access occurs before the constructor has finished, the field is not accessible to other threads. If the access occurs after the constructor has finished, the access must be a read, and no subsequent access can be a write.

We have implemented a static analysis that detects a subset of immutable fields. Before deeming a field f declared in class C immutable, the analysis checks the following conditions:

1) All writes to f must occur in a constructor of C, and must be through the this reference.
2) The this reference must not escape any constructor of C by being written to the heap (to a field or an array element), or by being passed as an argument (including the implicit receiver argument) to a called method.

Note that the analysis has limited precision: it cannot detect immutable fields that are written in methods that are called only from the constructor.

On the running example, the immutable field analysis correctly determines that the Employee.name field is written only in the constructor of Employee and therefore JPF does not have to create scheduling choices at accesses to this field.

In one of our benchmarks (Simple JBB), object initialization is performed not only in the constructor, but also in other methods that are called immediately after the constructor. Therefore, on this benchmark, we manually identified fields that were immutable in the sense that their objects did not escape during their initialization phase (which includes the constructor and the additional initialization method), and the fields were not written after this initialization phase had completed. Detection of such fields could be automated using a static analysis similar to the one described in [31].

V. Experiments

We implemented all variants of the field access analysis described in Section IV using the WALA library for static analysis [30] and integrated them with JPF. We used JPF in a standard configuration with all optimizations that are enabled by default — this includes partial order reduction. When using demand-driven pointer analysis, we used WALA's default analysis budget for each query. Our implementation, experimental setup, and a redistributable subset of the benchmarks is publicly available at http://plg.uwaterloo.ca/~pparizek/jpf/ase11/.

We evaluated our approach on seven applications: CoCoME [6], CRE Demo [1], Daisy file system [26], Raytracer benchmark from the Java Grande Forum suite [14], Elevator benchmark from the PJBench suite [25], plain Java version of the jPapaBench benchmark [17], and Simple JBB. All

these applications are multi-threaded and the individual threads interact significantly.

The CoCoME application (3500 lines of code) is a prototype of a trading system for supermarkets. 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. 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. 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. We also fixed several errors, which were created on purpose by the authors.

The Raytracer benchmark (1100 loc) involves two concurrent threads that render a given scene. Each thread computes a part of the image. We used a configuration with fewer objects in the scene and smaller size of the rendered image.

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 shared objects. Some threads represent the environment and external interrupts, and the remaining threads perform tasks related to airplane control and navigation.

Simple JBB (3000 loc) is a simplified version of the SPEC JBB 2005 benchmark [27], 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). 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.

A general problem for static analyses is handling of native methods. We addressed this issue by manually creating a list of fields accessed inside models of native methods in JPF and 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.

We performed experiments with all variants of the field access analysis described in Section IV. They correspond to specific combinations of possible values of these configuration variables: context-sensitivity of field access analysis, usage of pointer analysis, and detection of immutable fields (Table I).

None of the benchmark applications contain any errors so that JPF traverses the whole state space of the application in each case. The purpose of the experiments is to show how much our approach reduces the state space and exploration time, while preserving the full coverage of program behaviors.

TABLE I
POSSIBLE CONFIGURATIONS OF FIELD ACCESS ANALYSIS

Variables	Options
field access analysis	insensitive callee 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 results of the experiments are provided in Tables II-IV. There are separate tables for the number of states explored by JPF, total running time of JPF combined with the static analysis, and memory consumption (in GB). The second row of each table contains results for the original JPF without any field access analysis. Each of the other rows represents JPF with one configuration of the field access analysis.

We performed experiments with the analysis variant that assumes knowledge of manually-identified immutable fields only for Simple JBB. We found many immutable fields in the source code of Simple JBB by hand.

We put a limit of eight hours on the running time of experiments with jPapaBench and Simple JBB. These limits are sufficient to observe the performance improvements gained by using our approach. The recorded numbers of states vary significantly for those experiments with jPapaBench and Simple JBB that did not finish, because the overhead related to usage of the analysis results in JPF differs over the analysis variants and therefore JPF runs out of the time limit after processing a different number of states in each case.

VI. DISCUSSION

The experimental results show that usage of the static field access analysis significantly reduces (1) the number of states that JPF needs to explore for checking all possible behaviors and (2) the time required to explore the whole state space, and therefore makes it feasible to verify more complex programs than with the original JPF. In particular, the jPapaBench and Simple JBB benchmarks, which could not be verified even after 8 hours with the original JPF, can be verified in 12 and

TABLE II
EXPERIMENTAL RESULTS: NUMBER OF STATES EXPLORED BY JPF

field access	**pointer**	**immutable**	CoCoME	CRE Demo	Daisy	Raytracer	Elevator	jPapaBench	Simple JBB
original JPF			532161	1623101	7060292	3209390	38358616	> 176227405	> 33808665
insensitive	none	disabled	367539	1613571	7060292	3209362	38358616	> 103804005	> 25962010
callee	none	disabled	165300	1614335	4882566	3043647	38391582	> 118469868	> 24727654
sensitive	none	disabled	124874	65762	4753239	3043599	10373397	1294	> 24156805
sensitive	none	enabled	124874	65762	4753239	3043599	10373397	1294	1499543
sensitive	ci ex	disabled	122957	65762	4753215	3043599	10373397	1294	> 19474203
sensitive	ci dd	disabled	124874	65762	4753239	3043599	10373397	1294	> 16826484
sensitive	ci ex	enabled	122957	65762	4753215	3043599	10373397	1294	1499543
sensitive	cs ex	enabled	122957	n/a	4753215	3043599	10373397	1294	n/a
sensitive	ci dd	enabled	124874	65762	4753239	3043599	10373397	1294	1499543
sensitive	cs dd	enabled	124874	64370	4753239	3043599	10373397	1294	1499543

TABLE III
EXPERIMENTAL RESULTS: TOTAL RUNNING TIME OF JPF AND FIELD ACCESS ANALYSIS

field access	**pointer**	**immutable**	CoCoME	CRE Demo	Daisy	Raytracer	Elevator	jPapaBench	Simple JBB
original JPF			265 s	800 s	1256 s	703 s	4737 s	> 8 hours	> 8 hours
insensitive	none	disabled	245 s	984 s	1505 s	1211 s	7874 s	> 8 hours	> 8 hours
callee	none	disabled	117 s	937 s	1047 s	958 s	6380 s	> 8 hours	> 8 hours
sensitive	none	disabled	86 s	84 s	1036 s	915 s	1675 s	732 s	> 8 hours
sensitive	none	enabled	87 s	84 s	1103 s	905 s	1411 s	782 s	3283 s
sensitive	ci ex	disabled	98 s	106 s	1287 s	1188 s	2324 s	807 s	> 8 hours
sensitive	ci dd	disabled	105 s	124 s	1125 s	1212 s	2039 s	775 s	> 8 hours
sensitive	ci ex	enabled	102 s	102 s	1104 s	1296 s	2354 s	798 s	4898 s
sensitive	cs ex	enabled	179 s	> 3 hours	1263 s	1496 s	3699 s	802 s	> 8 hours
sensitive	ci dd	enabled	111 s	124 s	1112 s	1241 s	1857 s	768 s	20532 s
sensitive	cs dd	enabled	110 s	164 s	1151 s	1229 s	22692 s	764 s	12839 s

TABLE IV
EXPERIMENTAL RESULTS: MEMORY CONSUMPTION OF JPF TOGETHER WITH FIELD ACCESS ANALYSIS

field access	**pointer**	**immutable**	CoCoME	CRE Demo	Daisy	Raytracer	Elevator	jPapaBench	Simple JBB
original JPF			0.9 GB	1.3 GB	2.0 GB	2.3 GB	0.8 GB	> 8.4 GB	> 0.9 GB
insensitive	none	disabled	1.6 GB	2.8 GB	2.9 GB	3.0 GB	3.8 GB	> 7.9 GB	> 5.4 GB
callee	none	disabled	2.1 GB	3.3 GB	2.9 GB	3.0 GB	3.8 GB	> 6.4 GB	> 5.0 GB
sensitive	none	disabled	1.8 GB	1.9 GB	2.0 GB	2.9 GB	2.9 GB	2.8 GB	> 3.4 GB
sensitive	none	enabled	2.0 GB	1.7 GB	2.8 GB	2.9 GB	2.9 GB	2.8 GB	3.1 GB
sensitive	ci ex	disabled	2.5 GB	1.8 GB	3.6 GB	3.7 GB	2.5 GB	3.5 GB	> 4.9 GB
sensitive	ci dd	disabled	3.5 GB	3.7 GB	3.5 GB	3.7 GB	3.6 GB	3.5 GB	> 4.3 GB
sensitive	ci ex	enabled	2.7 GB	2.1 GB	3.6 GB	3.7 GB	3.2 GB	3.5 GB	4.8 GB
sensitive	cs ex	enabled	3.3 GB	> 10 GB	2.8 GB	3.6 GB	4.1 GB	3.5 GB	> 9.1 GB
sensitive	ci dd	enabled	3.6 GB	3.7 GB	3.5 GB	3.7 GB	3.6 GB	3.5 GB	4.1 GB
sensitive	cs dd	enabled	2.2 GB	2.3 GB	2.2 GB	2.3 GB	2.3 GB	2.2 GB	2.8 GB

55 minutes, respectively, using JPF with the static field access analysis. Static analysis also speeds up verification of the other benchmarks, by up to 9.5 times (on CRE Demo).

To achieve these improvements, the static analysis must be sufficiently precise. However, in addition to precision, it is important to also consider the time spent on the static analysis and on making use of the static information in JPF. In some cases, the speedup of JPF due to more precise static information is smaller than the increase in the cost of computing and using that static information.

Our first observation is that context sensitivity is very important to the effectiveness of field access analysis, and that its benefit outweighs the modest increase in analysis cost. On all but one benchmark, CoCoME, the context-insensitive analysis has no significant effect on the number of states to be explored. Even on CoCoME, adding context sensitivity further reduces the number of states by nearly two thirds. The benefit of callee context sensitivity is mixed. On two benchmarks, Daisy and Raytracer, it achieves almost the same reduction as full context sensitivity; on two other benchmarks, CRE Demo and Elevator, it has almost no effect; finally, on CoCoME, it has some effect, but less than full context sensitivity. The best variation is the fully context sensitive analysis that uses the dynamic call stack from JPF. Only with this level of precision is it feasible to verify jPapaBench. Full context sensitivity also reduces the number of states in CRE Demo by a factor of 25 compared to callee context sensitivity, and enables significant reductions in all but two of the benchmarks. The cost of the fully context sensitive analysis is small compared to its benefits; on every benchmark, the total time required by JPF and the static analysis is lower for the fully context sensitive analysis than for the other two variations.

The immutable fields analysis makes it feasible to verify the most complex benchmark, Simple JBB. Due to its simplicity,

the analysis wastes little time on the other programs for which it is ineffective.

Overall, we found the points-to analyses that we evaluated to have only a small effect on precision that does not justify the cost of the analysis. On the CoCoME benchmark, the exhaustive points-to analysis reduced the number of states to be explored by 1.5%. On the CRE Demo benchmark, the context-sensitive demand-driven points-to analysis reduced the number of states to be explored by 2.1%. We found no cases where the precision of field access analysis improved due to the context-insensitive demand-driven points-to analysis or the object-sensitive exhaustive analysis. The relative costs of exhaustive and demand-driven points-to analyses were mixed. On some benchmarks, the demand-driven analysis was faster than exhaustive analysis because it did not have to analyze irrelevant parts of the program. On other benchmarks, use of the demand-driven analysis turned out to be more expensive, because JPF issued very large numbers of queries and the demand-driven analysis must do some small amount of work for each query.

The static analysis time is generally negligible compared to the time used by the state space traversal. It is less than a minute for Simple JBB and less than 15 seconds for all other benchmarks. The only exception is the exhaustive object-sensitive pointer analysis for CRE Demo and Simple JBB, in which case the static analysis time is 3 hours and 8 hours, respectively.

The memory consumption results show that although the static analyses do require some memory, the increases in memory consumption are modest compared to the original JPF. The only exception is the object-sensitive points-to analysis, which runs out of memory on the CRE Demo benchmark.

Of all the static analysis variations that we evaluated, the most effective overall is the fully context sensitive field access analysis with detection of immutable fields but without points-to analysis.

VII. Related Work

Several approaches exist that combine Java PathFinder with static analysis for the purpose of more efficient verification of multi-threaded Java programs or search for errors.

The key idea proposed by Chen and MacDonald [7] is to address state explosion by exploring just one thread interleaving for each sequence of globally-visible actions. Static analyses are used to determine sequences of actions that are performed by different threads and have global effects, like accesses to fields on shared objects. For each sequence of actions, JPF is used to check the corresponding thread interleaving for concurrency errors. However, more than one thread interleaving may be actually explored for some action sequences due to imprecision of the static analysis. Other limitations include missing support for dynamically created threads and data non-determinism in the given Java program — our approach does not have these limitations. Experimental results published in [7] show a significant gain in performance, but it remains to be seen how much this approach would help on more complex Java programs, such as Simple JBB.

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 have a global effect or depend on the global state. 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 acquired from JPF. Then JPF explores all thread interleavings involving newly identified unsafe statements and updates 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 static analysis — it is then guaranteed that JPF has explored all interleavings of unsafe statements executed by different threads. The published experimental results show that this approach was evaluated on one program of approximately 20 lines of code, for which the state space size was reduced to a half. No results for large programs are available, and the total running time of JPF combined with static analysis is also not provided — 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.

There exist many other techniques for detecting concurrency errors that target different languages and platforms than Java or do not involve Java PathFinder [2][9][11][16][22][23][24]. The technique for efficient detection of data races proposed by Choi et al. [9] uses static analysis to identify statements that may participate in a data race and dynamic analysis to check these statements. The technique proposed by Agarwal et al. [2] uses type discovery to identify potentially unsafe statements that the subsequent dynamic analysis focuses on during checking. The static analysis proposed by Naik and Aiken [22] computes a must not alias relation for reference variables. It could be used in our approach to distinguish objects of the same class instead of the may alias analysis based on points-to analyses that we experimented with.

A completely dynamic approach to partial order reduction was proposed by Flanagan and Godefroid [12]. The key idea is to dynamically collect information about actual thread interaction during exploration of execution paths and use results of this analysis to identify states where new thread scheduling choices must be created to enable exploration of alternative execution paths. The process is repeated until all known execution paths have been explored and no new thread choices have to be added based on the dynamic analysis. This approach is in theory more precise than ours, but it uses a stateless search and therefore works only for terminating programs with an acyclic state space, while our technique targets stateful search. We are not aware of any combination of dynamic POR with stateful search. Moreover, this approach was evaluated on two 20-line programs in [12] — it remains to be seen how well it works for larger programs, such as SimpleJBB or jPapaBench, in which each thread executes a very long sequence of instructions.

Unkel and Lam [31] 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 [31] as one that is never written after it is first read, 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. The stationary field analysis is conservative — 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.

Guyer et al. [13] and Cherem and Rugina [8] suggested using static analysis to free objects early in order to reduce the load on a dynamic garbage collector. Like our approach, these techniques statically detect objects that will never be accessed in the future, yet are still reachable via a dynamic heap traversal. It may be possible to adapt these static analyses for speeding up JPF, or, conversely, apply some of our static analyses to freeing objects early.

VIII. Conclusion

We have proposed an optimization of exhaustive state space traversal of Java programs with JPF, which addresses the inherent limitations of JPF that are a consequence of the on-the-fly state space construction. The main idea is the elimination of some unnecessary thread scheduling choices in the program's state space using the results of a static analysis that identifies field accesses that may occur on any execution path between a specific point in the code of a thread and exit from the thread. Experimental results show that the combination of JPF with the static field access analysis allows to check more complex multi-threaded Java programs in reasonable time.

More specifically, we found (1) that it is essential to use the context-sensitive field access analysis to achieve the biggest performance improvement and (2) that usage of points-to analyses has only a very small effect. The jPapaBench application could be verified only with the context-sensitive analysis. The Simple JBB benchmark could be verified only with the immutability analysis enabled.

More thread choices could be eliminated by applying similar static analyses to synchronized blocks — JPF would create a thread choice at a monitor enter/exit instruction executed by one thread only if some other thread may acquire or release the same monitor in future. A similar approach to [3] may be used. Another possible improvement is to exploit more information available in the dynamic program state during exhaustive state space traversal. We also plan to design and implement an analysis that detects immutable fields written (initialized) in methods other than constructors.

Acknowledgements

We would like to thank Jan Vitek and Tomas Kalibera for suggesting the idea that resulted in this work. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

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] J. Aldrich, C. Chambers, E.G. Sirer, and S.J. Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs, SAS 1999, LNCS, vol. 1694.

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

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

[6] 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.

[7] J. Chen and S. MacDonald. Testing Concurrent Programs using Value Schedules, ASE 2007, ACM.

[8] S. Cherem and R. Rugina. Compile-time Deallocation of Individual Objects. In ISMM 2006, ACM.

[9] 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.

[10] 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.

[11] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks, In Proceedings of SOSP'03, ACM.

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

[13] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-Me: A Static Analysis for Automatic Individual Object Reclamation. In PLDI 2006.

[14] Java Grande Forum Benchmarks, http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html

[15] Java PathFinder, http://babelfish.arc.nasa.gov/trac/jpf/

[16] V. Kahlon, S. Sankaranarayanan, and A. Gupta. Semantic Reduction of Thread Interleavings in Concurrent Programs, In TACAS 2009.

[17] jPapaBench, http://d3s.mff.cuni.cz/~malohlava/projects/jpapabench/

[18] O. Lhoták and L. Hendren. Scaling Java Points-to Analysis Using Spark, In CC 2003.

[19] A. Milanova, A. Rountev, and B. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java, ACM TOSEM, 14(1), 2005.

[20] Steven S. Muchnick. Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers, 1997.

[21] 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.

[22] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection, In Proceedings of POPL 2007, ACM.

[23] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java, In PLDI 2006, ACM.

[24] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection, In Proceedings of ICSE 2009, IEEE.

[25] Parallel Java Benchmarks, http://code.google.com/p/pjbench

[26] S. Qadeer. Daisy File System, Joint CAV/ISSTA special event on specification, verification and testing of concurrent software, 2004.

[27] SPEC JBB 2005 benchmark, http://www.spec.org/jbb2005/

[28] M. Sridharan and R. Bodik. Refinement-based Context-sensitive Points-to Analysis for Java, PLDI 2006, ACM.

[29] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven Points-to Analysis for Java, OOPSLA 2005, ACM.

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

[31] C. Unkel and M.S. Lam. Automatic Inference of Stationary Fields: A Generalization of Java's Final Fields, POPL 2008, ACM.