Approximating Happens-Before Order: Interplay between Static Analysis and State Space Traversal

Pavel Parízek
Charles University in Prague
parizek@d3s.mff.cuni.cz

Pavel Jančík
Charles University in Prague
jancik@d3s.mff.cuni.cz

ABSTRACT

Techniques and tools for verification of multi-threaded programs must cope with the huge number of possible thread interleavings. Tools based on systematic exploration of a program state space employ partial order reduction to avoid redundant thread interleavings. The key idea is to make non-deterministic thread choices only at statements that read or modify the global state shared by multiple threads. We focus on the approach to partial order reduction used in tools such as Java Pathfinder (JPF), which construct the program state space on-the-fly, and therefore can use only information available in the current program state and execution history to identify statements that may be globally-relevant.

In our previous work, we developed a field access analysis that provides information about fields that may be accessed during program execution, and used it in JPF for more precise identification of globally-relevant statements. We build upon that and propose a hybrid may-happen-before analysis that computes a safe approximation of the happens-before ordering. Partial order reduction techniques can use the happens-before ordering to detect pairs of globally-relevant field access statements that cannot be interleaved arbitrarily (due to synchronization between threads), and based on that avoid making unnecessary thread choices. The may-happen-before analysis combines static analysis with knowledge of information available from the dynamic program state. Results of experiments with several Java programs show that usage of the may-happen-before analysis further improves the scalability of JPF.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

static analysis, state space traversal, dynamic analysis, concurrency, happens-before, Java Pathfinder

1. INTRODUCTION

Many tools for testing and verification of multi-threaded programs are based on systematic traversal of a program state space. Two well-known tools are Java Pathfinder [10] and CHESS [14]. They use the state space traversal to check the program behavior under all possible thread interleavings. Each thread interleaving corresponds to a sequence of thread scheduling decisions, and also to a particular sequence of statements executed by program threads. The main challenge faced by the tools and verification techniques is the need to cope with the huge number of thread interleavings.

Tools employ partial order reduction (POR) [8] to avoid exploration of redundant thread interleavings. We consider a thread interleaving to be redundant if it corresponds to the same sequence of globally-relevant statements as some other thread interleaving that has been already explored during the state space traversal. A *globally-relevant* statement reads or modifies the global state shared by multiple threads, and thus represents interaction between concurrently-running threads. Other statements are called *thread-local*. The set of globally-relevant statements contains, for example, acquisition of a lock, wakeup of a thread waiting on a monitor, and field accesses on heap objects shared by multiple threads.

State space traversal with POR works as follows. The key idea behind it is to consider thread scheduling choices only at globally-relevant statements. Let i be the next statement (instruction) to be executed on the currently running thread. If i is a globally-relevant statement then the verification tool must explore the interleavings where i is really executed next, and also the interleavings where actions of other threads occur before i. The tool will create a non-deterministic thread choice just before i to achieve this and cover all the possible thread interleavings regarding i. If i is a thread-local statement then it cannot influence the execution of other threads and vice versa, and therefore it is sufficient to explore only the interleavings in which i is executed next from the current state. No thread choice is created before any thread-local statement.

Existing approaches to POR (e.g., [7–9]) conservatively over-approximate the set of globally-relevant statements to yield sound exploration of the program state space that covers all distinct thread interleavings of globally-relevant statements. The principal challenge is to determine precisely which statements are globally-relevant. The number of redundant thread interleavings explored during the state space traversal depends on the number of statements that are actually thread-local but were imprecisely identified as globally-relevant.

We focus on the approach to POR used in Java Pathfinder (JPF), which is a framework for state space traversal of multi-threaded Java programs, and describe important concepts in the context of JPF. However, the concepts apply also to other tools for state space traversal that work in a similar way to JPF, such as CHESS. In the

rest of this paper, for simplicity of presentation we use the term "JPF" also when referring just to the POR technique inside JPF, unless specified otherwise.

JPF constructs the program state space on-the-fly using its custom virtual machine that interprets statements (bytecode instructions). Therefore, JPF cannot look ahead in the program execution to find what may happen in the future, and its POR technique uses only information available in the current program state and execution history to determine whether a given statement is globally-relevant and whether it has to make a thread choice before execution of the given statement. An important category of statements considered by JPF to be globally-relevant are field accesses on heap objects that are reachable from multiple threads according to a dynamic escape analysis [3]. JPF conservatively assumes that each thread may in the future really access every field of every heap object that it can reach in the current state. This approach is safe but not precise — a particular object may be reachable from multiple threads but really accessed only by a single thread during the program execution, or the threads may access different fields of a given heap object. As a consequence, JPF explores many redundant thread interleavings because it determines imprecisely that some field access statements may be globally-relevant when they are actually thread-local.

In prior work [16], we designed a hybrid *field access analysis*, which provides information about fields possibly accessed during the program execution, and used its results in JPF for more precise identification of globally-relevant and thread-local statements. Many redundant thread interleavings were avoided in this way, as shown by the experimental results published in [16]. The field access analysis combines static analysis with knowledge of the dynamic program state (thread call stacks) that is maintained by JPF during the state space traversal. For each program point p in each thread T, the hybrid analysis computes the set of fields possibly accessed by the given thread T after the point p on any execution path. The static phase is a backward flow-sensitive context-insensitive data flow analysis performed over the full inter-procedural control flow graph (ICFG) of a given thread. Knowledge of the dynamic call stack of each thread is used to improve precision of the analysis results — the hybrid analysis considers only those return edges in the ICFG that can be actually taken during the program execution. Nevertheless, a limitation of the field access analysis is that it considers the whole lifetime of each thread from the current program state to its end. In practice, threads are usually synchronized to disable certain interleavings (execution paths), and therefore particular sequences of globally-relevant field access statements will not happen during the actual program execution. Thread choices and interleavings that correspond to such sequences of field access statements are still redundant, and could be safely eliminated.

We propose to address this limitation by using a new hybrid *may-happen-before* (MHB) analysis that computes safe approximation of the happens-before ordering [13] for field access statements and thread synchronization events. The may-happen-before analysis, too, is a specific combination of (1) static analysis with (2) the information available from the dynamic program state maintained by JPF. This combination designed to compute the happens-before ordering is the main new aspect of the proposed approach. Static analysis provides information about possible future behavior of the program after a particular code location, and information taken from the dynamic program state is used to improve precision of the results. Many existing program verification techniques involve the happens-before ordering (see, e.g., [7, 12]), but all techniques that we know about compute it using only dynamic analysis that covers a small subset of possible execution paths.

```
1  public class Example {
2    public static int x = 0;
3
4    public static void main(String[] args) {
5      Object lock1 = new Object();
6      Object lock2 = new Object();
7
8      Thread th1 = new Writer(lock1);
9      Thread th2 = new Reader(lock1);
10     Thread th3 = new Notifier(lock2);
11
12     th1.start(); th2.start(); th3.start();
13     th1.join(); th2.join(); th3.join();
14   }
15 }
16
17 class Notifier extends Thread {
18   private Object lock2;
19
20   public Notifier(Object l2) { this.lock2 = l2; }
21
22   public void run() {
23     synchronized (lock2) { lock2.notify(); }
24   }
25 }
26
27 class Writer extends Thread {
28   private Object lock1;
29
30   public Writer(Object l1) { this.lock1 = l1; }
31
32   public void run() {
33     Example.x = 1;
34     synchronized (lock1) { lock1.notify(); }
35   }
36 }
37
38 class Reader extends Thread {
39   private Object lock1;
40
41   public Reader(Object l1) { this.lock1 = l1; }
42
43   public void run() {
44     synchronized (lock1) { lock1.wait(); }
45     int v = Example.x;
46   }
47 }
```

Figure 1: Example program

The POR technique in JPF uses the happens-before ordering to identify pairs of globally-relevant field access statements that cannot be interleaved arbitrarily during the actual program execution, and based on that it avoids creating redundant thread scheduling choices at such field access statements. Results of our experiments show that combination of the field access analysis with the may-happen-before analysis yields a significant improvement over the standalone field access analysis [16] in terms of the number of eliminated redundant interleavings and thread choices.

2. OVERVIEW

We illustrate the whole approach on the example Java program in Figure 1, which involves three threads — reader, writer, and notifier. The reader and writer threads communicate via calls of wait and notify on the same monitor object (lock1) and also via possibly concurrent accesses to the static field Example.x. The notifier thread calls notify on a different monitor object (lock2).

Let s be the current program state in which (i) the program counter

of each thread refers to the first instruction of its method run and (ii) the writer thread is the active one. In this case, JPF together with our analyses must decide whether it must create a thread choice before the write access to Example.x (line 33) to cover all possible interleavings of the read and write accesses that may occur during the rest of the program execution from the state s. This is done in two steps:

1. The field access analysis identifies the possible future read access to the field Example.x by the reader thread (line 45). It means that the write access at line 33 is not a thread-local statement. Now the question is whether the read access may be executed before the write access in some thread interleaving. The answer to this question is computed automatically by the may-happen-before analysis (step 2).

2. The analysis determines that there is a call of wait (line 44) before the read access to Example.x on every execution path in the reader thread. For every call of wait before the read access, the may-happen-before analysis determines whether some other thread may wake up the waiting thread (via a call of notify) before the writer thread executes the write access to Example.x. This cannot happen in case of the example program, because the notifier thread executes the call of notify (line 23) on a different monitor object. Consequently, the read access cannot be executed before the write access to Example.x in any thread interleaving, and thus JPF does not have to make a thread choice.

Note that the static field Example.x is reachable from all program threads, and thus plain JPF would imprecisely create a thread choice before every access to the field.

A very similar process is performed for programs where locks are used to guard accesses to shared fields. In that case, JPF and our analyses have to consider the happens-before ordering between the lock acquisition and release statements in different threads.

The rest of this section gives more details about the whole process — input programs, the field access analysis, main steps of the may-happen-before analysis, and usage of the analysis results in JPF to decide about thread choices.

Our approach targets Java programs with multiple threads that use locks, signals (wait and notify), and thread join statements for mutual synchronization. It supports arbitrarily nested locking operations (acquisition, release) and all possible locking patterns — most notably, also locking patterns other than nested synchronized blocks that are used typically in Java programs, because we take *acquire lock* and *release lock* as completely independent events. On the other hand, both JPF itself and the proposed analyses do not support these concurrency-related features of the Java platform: the full Java Memory Model (JMM), and spurious wakeups from the calls of wait that may happen on some platforms. JPF does not analyze those thread interleaving possible under JMM, in which the effects of some writes in one thread are observed out-of-order by other concurrent threads, and therefore it could miss some errors.

We propose a hybrid may-happen-before analysis that computes safe approximation of the happens-before order for field access statements and thread synchronization events (acquire lock, release lock, calls of wait and notify, thread join). The approximate happens-before order determines which thread interleavings cannot happen at runtime because of thread synchronization.

Both hybrid analyses, i.e. the may-happen-before analysis and the field access analysis, are computed in two steps. The first step involves static analyses performed in advance before a JPF run — pointer analysis, the static phase of the hybrid field access analysis, and the static phase of the hybrid may-happen-before analysis. We use an exhaustive flow-insensitive context-insensitive pointer analysis to identify abstract heap objects and to determine possibly aliased variables. For each program point, the static phase of the hybrid analyses computes only partial information about the future behavior of individual threads. We give more details in Section 4.

Full results of the hybrid analyses are computed on demand in the second step, using information from the dynamic program state. The second step is therefore performed in JPF during the state space traversal. In particular, the happens-before ordering between statements from different threads is computed using (i) results of the static analysis performed in the first step and (ii) specific information from the current dynamic state, including the dynamic call stack (program counter) of each thread.

The design of the may-happen-before analysis as a hybrid one has been motivated by the need to get the most precise results possible at a reasonable cost. For example, the dynamic call stack represents the full and precise calling context in each state associated with a program point p, while an efficient static analysis can only approximate the calling context for p. We provide a list of all the information taken from the dynamic program state at the end of Section 3 and discuss usage of this information to get precise analysis results in Section 4.

JPF uses results of the may-happen-before analysis together with results of the field access analysis to decide whether it must create a thread choice before a field access statement. Let s denote the current dynamic program state. Assuming that the next statement in the currently executing thread T_c is an access to the field f of a heap object o reachable from multiple threads, JPF performs the following steps. For every other thread $T_j, j \neq c$ in the current state s, it queries the results of the field access analysis for the current point p_j in T_j to see whether T_j may execute a possibly conflicting access to $o.f$ on any execution path starting in p_j. We consider only read-write pairs of accesses to the same field as possibly conflicting. The order of two write accesses to the same field may affect the program execution only if the field is eventually read, and in that case both possible sequences of the write accesses will be explored.

If the results show that no other thread may access $o.f$ in the rest of the program execution from the state s, then the field access in T_c is thread-local and JPF does not have to make a thread choice before it. Otherwise, if there is a possible future access to $o.f$ in some thread T_h other than T_c, JPF checks whether the access in the other thread T_h may happen before the access to $o.f$ in the current thread T_c on some execution path (thread interleaving). The analysis determines whether the synchronization events impose a strict execution ordering between the field accesses in T_c and T_h. In that case only such thread interleavings, in which the access to $o.f$ in T_c precedes the access in T_h, are possible starting from the current dynamic state of the program, and thus JPF does not have to make a thread choice before the field access in T_c. Note that all these checks must be done for each thread other than T_c that may access $o.f$ on some execution path starting in s.

3. HAPPENS-BEFORE PATTERNS

Now we describe scenarios in which certain sequences of field access statements are not possible due to synchronization between threads, and define a list of information that the POR technique needs in these cases from the hybrid analysis to decide whether it has to make a thread choice before a field access.

Let the program be in a dynamic state s, the next statement in the current thread T_c be an access to $o.f$, and let there exist a future access to $o.f$ in some other thread T_h. Without loss of generality, we assume that T_c performs a write access to $o.f$ and that T_h may perform a read access in the future. There is a strict execution ordering

between the write access to $o.f$ in T_c and the future read access in T_h, if the other thread T_h is blocked for some reason before the future read access in every possible thread interleaving from s. We distinguish four happens-before patterns (scenarios) in which T_h may get blocked. Figure 2 shows simple code examples for all the patterns. Arrows indicate the ordering between events. We discuss each pattern separately in the rest of this section.

Pattern 1: wait and notify (Figure 2a). The thread T_h gets certainly blocked before the future read access to $o.f$ if the following conditions hold:

- there is a call of wait on every control-flow path between the current point p_h in T_h and the future read access to $o.f$, and
- no thread other than T_c and T_h may call notify on the same monitor object as some call of wait in T_h (and possibly wake up T_h in this way).

In Figure 2a, assuming that the program counter of each thread refers to line 1, T_h gets blocked at the call of wait on the monitor object $L1$. The write access to $o.f$ in T_c is always executed before the read access because the thread T_n calls notify on a different monitor object $L2$.

Pattern 2: lock acquisition and release (Figure 2b). If the current thread T_c holds a lock in the current dynamic state s just before the field access to $o.f$, and there is an acquisition statement on the same lock in the thread T_h on every control flow path before the future access to $o.f$, then T_h gets blocked at the lock acquisition statement that precedes the future field access. There is a strict execution ordering between the lock release statement in T_c (line 3) and the lock acquisition statement in T_h (line 1) in that case.

However, both threads must use the same lock object, i.e. the same dynamic heap object, to guard accesses to $o.f$ in every thread interleaving and on every control flow path. JPF queries the analysis results and the current dynamic program state to determine whether the respective lock variables used in T_c and T_h are guaranteed to point at the same heap object upon execution of the lock acquisition statement. The following conditions must hold:

- the points-to set for the lock variable used in T_h has only a single element,
- the allocation site as for the lock variable in T_h is equal to the allocation site of the currently held lock in T_c, and
- a single object is ever allocated at the site as during the whole program execution.

Note that many dynamic heap objects can be allocated at a given site in general during program execution, and therefore just comparing allocation sites in the points-to set is not a safe approach to determine equality of lock objects. In Figure 2b, both threads use the same lock object L. The benchmark programs that we use for experiments (Section 5), and which are quite representative of typical Java programs, contain many field access statements guarded by locks that satisfy the conditions given above. Global lock objects are typically allocated during program initialization at a site that is executed only once.

Pattern 3: locking patterns that involve this (Figure 2c). This pattern covers a special case of the previous one. It captures the common scenario (for Java programs) of accessing fields through the local variable this inside a synchronized block over this (including synchronized methods).

More specifically, the thread T_h gets certainly blocked also in the case when both threads (i) access the field f on the same dynamic object o and (ii) guard the field access by a lock associated with o. If the following conditions are satisfied, then T_h will block before the future access to the field $o.f$.

- T_c accesses the field f through the local variable v that points to the object o in the current program state;
- T_c holds a lock over the object o (due to the synchronized block over the variable v around the field access);
- every possible conflicting future access to f in T_h is performed through the local variable this (current object in the method performing the access);
- every conflicting future access to f in T_h via this outside of any instance constructor is guarded by a synchronized block over this, i.e. the field access is performed on the object used also as the lock;
- for every conflicting future access to f in T_h via this in some instance constructor, the target object of the field access must be reachable only from T_h at the time of the field access;
- for each access to f via this in T_h outside of instance constructors, the boundaries of the respective synchronized block (i.e., the locked region) around the field access are in the same method.

The conditions permit access to f only through the local variable this in T_h because the value of this cannot be modified inside a given method, and thus we have the guarantee that the field access is performed on the same object that is used as the lock. Any other local variable can be modified inside the method, and thus at the time of the field access the variable may point to a different object than at the time of lock acquisition. Boundaries of the synchronized blocks must be in the same method as the field access, because the local variable this may obviously point to different objects during execution of different methods.

An important difference from the pattern 2, which considers results of the pointer analysis, is that conditions specifying this pattern refer to the syntactical names of local variables (v and this). Consequently, this pattern captures also some cases when T_h gets blocked that are not covered by the pattern 2, and thus improves the precision of our approach. Every possible future access to f in T_h must be inspected (unlike in the other patterns) to check that it is performed through this.

Note also that the conditions permit unsynchronized accesses to f inside constructors, which is a typical scenario (code pattern) in Java programs. It is a safe scenario when the newly created dynamic heap object has not escaped from T_h yet before the field access, because then the object cannot be accessed concurrently in T_c. To check that, we compare the allocation site of the dynamic heap object o to be accessed next in T_c (we get the allocation site from the dynamic program state) and abstract heap objects in the points-to set of the local variable this in T_h. The fifth condition is violated when the newly created object escapes from T_h.

If both the threads T_c and T_h access the field f on the same object o and the conditions are satisfied, then required usage of this in T_h guarantees that the same object is also used as the lock guarding the field accesses.

We described this pattern only for accesses to instance fields via this, but we use the same approach also for static fields accessed inside static synchronized methods.

Pattern 4: thread join (Figure 2d). The last case that we consider here is when the thread T_h calls the join method on T_c before the

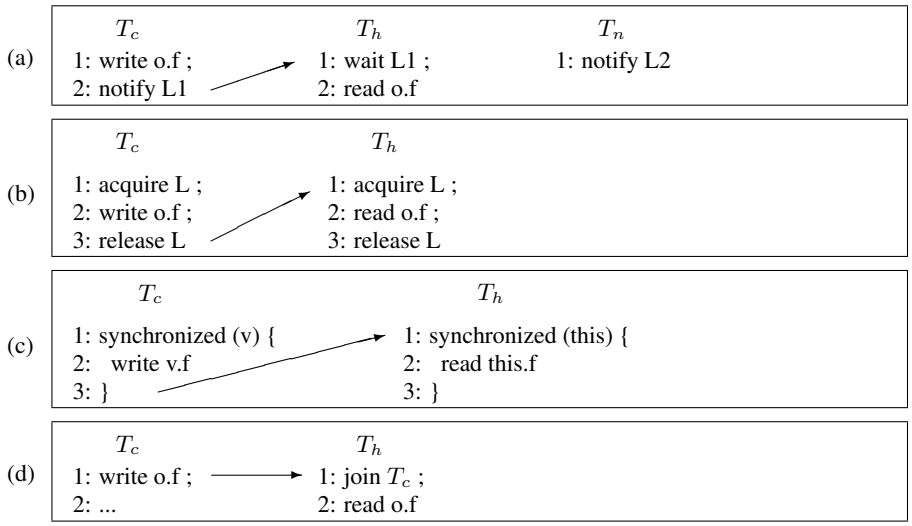

Figure 2: Four code patterns in which the thread T_h gets blocked before the future field access — (a) call of wait, (b) lock acquisition, (c) locking patterns that involve this, and (d) thread join

future access to $o.f$, and therefore gets blocked. There is a strict ordering between every statement in T_c and the call of join in T_h, which guarantees that the access to $o.f$ in T_h cannot occur before the access by T_c in any thread interleaving executed from the current state s. JPF checks whether T_h really executes a thread join on the dynamic heap object representing T_c in the same way as for locks (pattern 2), using results of the may-happen-before analysis and information from the dynamic program state.

Remarks. Note that we do not have to consider any pattern involving calls of the start method on some thread object, because it is not a possibly blocking operation.

Detection. Considering all the happens-before patterns described above in this section, JPF needs information both from the may-happen-before analysis and directly from the dynamic program state to properly detect their occurrences.

More specifically, it needs the following information about each program point p in each thread T from the hybrid may-happen-before analysis (Section 4).

(I1) For each field f, the set of calls of wait such that each element appears on some control flow path starting in p before the first future access to f.

(I2) For each field f, a boolean value saying whether on every control-flow path starting in p there is a call of wait before the first access to f on the path.

(I3) The set of all future calls of notify until the end of the thread's lifetime on any control flow path starting in p.

(I4) For each field f, the set of lock acquisition statements and thread join statements that occur before the first access to f on every control flow path starting in p.

(I5) The set of allocation sites on any control flow path of T that starts in the point p.

(I6) The list of fields accessed in T on any control flow path starting in p only through the local variable this and inside synchronized blocks over this (with the exception of instance constructors).

The following information is retrieved directly from the dynamic program state or computed during the state space traversal by JPF.

(I7) The set of locks held by T in the current dynamic program state just before the field access.

(I8) The number of dynamic heap objects allocated at each site associated with a lock variable or a thread variable during the program execution so far (up to the current state s).

(I9) The dynamic call stack of T, which is used to check whether T is executing an instance constructor on a dynamic heap object other than the target of a specific field access.

The number of dynamic heap objects allocated at a given site is computed by a listener plugin for JPF that tracks object allocations. Note that information represented by the items I5 and I8 is used to determine whether a single object is ever allocated at the given site during program execution.

Using all this data, JPF can determine whether it has to make a thread choice before the access to $o.f$ in T_c, assuming that some other thread T_h may access $o.f$ in the rest of the program execution from the current state. There must be a thread choice before a particular field access statement when none of the patterns apply.

4. MAY-HAPPEN-BEFORE ANALYSIS

We have designed a hybrid analysis that computes the happens-before information required by JPF. It combines static analysis with the knowledge of information from the dynamic program state. Static analysis is performed before the JPF run, and gives only partial results. Full results are computed at the state space exploration time (in JPF) using information from the dynamic program state. Therefore, results are valid only for the particular dynamic state of the given program, and, more specifically, for the current program point of each thread. Analysis results must be computed separately for each thread T (i.e., for its current program point p) in the dynamic program state s.

The whole may-happen-before analysis consists of several components — may-wait analysis, must-lock/join analysis, must-wait analysis, may-notify analysis, future allocations analysis, and the lock patterns analysis. Each component provides some of the information required by JPF (items I1-I6 from the list in the previous section). All the six component analyses are processed one by one.

First we explain our general approach to the combination of a static analysis with information from the dynamic program state, which is used by all the components except the lock patterns analysis, and then we describe each individual component in the rest of this section. We cover only the may-wait analysis and the lock patterns analysis in full detail, because other analyses follow the same principle as the may-wait analysis.

4.1 Combining Static Analysis with Dynamic Program State

Our approach to the design of hybrid analyses, described here, was originally proposed in [16] for the field access analysis.

The static analysis phase, which is performed in advance before the JPF run, gives only partial information that covers behavior of the thread T from the point p until the return from the method containing p (including nested method calls transitively). We use a backward flow-sensitive context-insensitive inter-procedural static data flow analysis, which has specific transfer functions for the call and return statements. The transfer function for a call statement merges data for entry to the callee method M and data for the next statement in the caller. It is formally expressed as follows: before[call M] = before[M.entry] $\cup$ after[call M]. The transfer function for the return statement, defined as before[return] = $\emptyset$, produces the empty set — it ensures that the set of data flow facts for each point captures only the events that may occur before the return from the current method.

A complete result of the given hybrid analysis for the point p in the thread T is computed at the state space exploration time using knowledge of the dynamic call stack of T (which is taken from the dynamic program state). The dynamic call stack of T specifies a sequence $p_0, p_1, \ldots, p_n$ of program points, where p_0 is the current program counter of T (in the top stack frame) and p_i for $i > 0$ is a return point from which the execution of the thread would continue after the return from the previous stack frame. Having this sequence, one just needs to merge partial results of the static analysis phase for all the points $p_i, i = 0...n$, to get the full result for p_0 and the current dynamic calling context of p_0.

The full result for a program point p in the thread T captures the future behavior of T from p until the end of T, and also the behavior of all threads started by T after the point p.

The main benefit of this design is that the hybrid analysis is fully calling-context-sensitive, because it considers only those return edges in the ICFG that can be actually taken during the program execution and ignores return edges that do not lead to the corresponding return point p_i (in the proper caller method).

4.2 May-Wait Analysis

We start our description of the individual components of the may-happen-before analysis with the may-wait analysis, which we cover in full detail, as indicated above.

For the program point p in the given thread T and for each field f that may be accessed after p, the analysis identifies the set of calls of wait that may appear between p and the future access to f. More specifically, each element of the set is an abstract target object (allocation site) of a call of wait that appears on some control flow path between p and the first access to f on that path. The set of possible abstract target objects for each call of wait is determined by the pointer analysis. We distinguish between read accesses and write accesses to fields. This analysis is further decomposed into two independent partial analyses, which are computed separately, and their results are combined by a post-processor at the state space exploration time. Each of the partial analyses involves a static phase and usage of information from the dynamic program state.

Part 1: Calls to wait before the first field access. This part of the may-wait analysis computes for the program point p the set of calls of wait such that each element occurs on some control flow path starting in p before the first future field access (to any field).

Figure 3 shows transfer functions for the static analysis phase. When the analysis encounters a call of wait, it adds every possible target abstract monitor object o (allocation site) into the set. Transfer functions for field access statements produce the empty set. The transfer functions for the call and return statements are defined according to the principle described in Section 4.1. The merge operator is a set union, as shown in the first line. All the sets of data flow facts for program statements are initially empty.

Instruction	Transfer function
	after[ℓ] = $\bigcup_{\ell' \in \text{succ}(\ell)}$ before[ℓ']
ℓ: wait(o)	before[ℓ] = after[ℓ] $\cup$ {o}
ℓ: v = o.f	before[ℓ] = $\emptyset$
ℓ: o.f = v	before[ℓ] = $\emptyset$
ℓ: return	before[ℓ] = $\emptyset$
ℓ: call M	before[ℓ] = before[M.entry] $\cup$ after[ℓ]
ℓ: other instr.	before[ℓ] = after[ℓ]

Figure 3: Transfer functions for the static phase of the first part of the may-wait analysis

Part 2: All possible first field accesses. The result of this second part of the may-wait analysis for a program point p is the set $\{fa_1, \ldots, fa_n\}$ of all the field accesses that may occur as the first after p. For each control flow path starting in p, the first field access on the path is in the set.

Instruction	Transfer function
	after[ℓ] = $\bigcup_{\ell' \in \text{succ}(\ell)}$ before[ℓ']
ℓ: v = o.f	before[ℓ] = { $\langle o, f, \ell \rangle$ }
ℓ: o.f = v	before[ℓ] = { $\langle o, f, \ell \rangle$ }
ℓ: return	before[ℓ] = { #mark# }
ℓ: call M	before[ℓ] = before[M.entry] $\cup$ after[ℓ]
ℓ: other instr.	before[ℓ] = after[ℓ]

Figure 4: Transfer functions for the static phase of the second part of the may-wait analysis

The set of data flow facts contains tuples $\langle o, f, l \rangle$, where o is the abstract target object, f is the field name, and l is the code location (program point). Figure 4 shows the transfer functions. When the analysis encounters a field access statement, it creates a set that contains only a single tuple capturing the respective field access. The transfer function for a call statement is the same as in the first part. If there is no field access between p and the return from the method containing p on some control flow path, the analysis puts a special mark into the resulting set to indicate this.

The special marks indicating the absence of a field access must be processed during the merge operation, when the full analysis results are computed at the state space traversal time using knowledge of the dynamic thread call stack. If the result of the static phase for a point p_i in the method m_i on the dynamic call stack contains the mark, indicating that there may not be a field access between p_i

and the return from m_i on some control flow path, then the analysis merges-in the first field access after the return point p_{i+1} in the method m_{i+1} that called m_i.

Post-processing. Data collected in both parts must be combined together and post-processed to get the set of calls of wait that may occur between the point p and the first access to the field f on some control flow path. We use the following approach.

A graph of field accesses is created from results of the second part. For each control flow path starting in the point p, the graph captures the sequence of field accesses on the path. Systematic traversal of the graph yields all possible sequences of field accesses between p and the first access to f (over all control flow paths), where each sequence contains only accesses to fields other than f. Then the set of calls of wait for each sequence is computed using results of the first part. For each pair (fa_i, fa_{i+1}) of field accesses in the sequence, the post-processor queries the results of the first part for the program point corresponding to fa_i to get the set of calls between fa_i and fa_{i+1}. Data for all the pairs make a set that corresponds to the given sequence. At the end, sets for all the field access sequences are merged using the union operator to get the full set of calls of wait over all control flow paths starting in p.

4.3 Must-Lock/Join Analysis

This analysis identifies lock acquisition statements or thread join statements, depending on the particular configuration, that appear before the first future access to the field f on every control flow path starting in the program point p.

We designed the must-lock/join analysis in a very similar way to the may-wait analysis. There are two differences: (1) data flow facts are target abstract objects of the lock acquisition statements, respectively thread join statements, and (2) set intersection is used as the merge operator in the first part and by the post-processor when traversing the graph of field accesses.

4.4 Must-Wait Analysis

The must-wait analysis determines whether there is a call of wait on every control flow path starting in p before the first access to the field f. The result of the static phase for a point p is the set of fields accessed only after a call of wait on every control flow path.

Field names are the data flow facts. The analysis uses set intersection as the merge operator. All the sets of data flow facts are initially full. Whenever the analysis encounters a field access statement, it removes the corresponding field name from the set. The transfer function for a call of wait produces the full set.

4.5 May-Notify Analysis

This analysis collects the set of future calls of notify that may occur after the point p on any control flow path before the end of T. Target abstract objects for the calls of notify represent the data flow facts in the static phase. The transfer function for a call of notify simply adds the target object into the set. As in the other may-analyses, the merge operator is a set union.

4.6 Future Allocations Analysis

The result of this analysis for a program point p in the thread T is the set of allocation sites at which some dynamic heap objects may be allocated after p on any control flow path. The sets of data flow facts represent allocation sites in the code of the thread T. When the analysis processes an object allocation (the new statement), it adds the site (code location) into the set.

4.7 Lock Patterns Analysis

For each program point p, this analysis finds a set $\{f_1, \ldots, f_n\}$ of fields where each f_i must satisfy the following conditions:

- every access to f_i on any control flow path starting in p is through the local variable this;
- if the access happens outside of an instance constructor, then it must be performed inside a synchronized block over this, and boundaries of the synchronized block must be in the same method as the field access.

The analysis computes the set of fields in two stages (A and B).

Stage A. This stage involves only static data flow analysis and does not use any information from the dynamic program state. For each method m in the program, the static analysis identifies a set of points in m that are inside a region guarded by a lock over the variable this associated with m. We achieve that using an intra-procedural flow-sensitive forward static analysis. The data flow fact is a boolean value saying whether a lock over this is currently held or not. We designed transfer functions that toggle the boolean value when the analysis hits a boundary of a locked region. The merge operator for this analysis is a set intersection.

Stage B. This stage is designed according to the principle described in Section 4.1, i.e. it involves flow-sensitive inter-procedural backward static analysis and queries information from the dynamic program state. It also uses data from the stage A, in addition to the knowledge of dynamic thread call stacks, to compute the full result for each program point p. We describe only the static phase here.

Data flow facts are field names. All the sets of data flow facts are initially full (with all the bits set), and the transfer function for a return statement also produces a full set. Set intersection is used again as the merge operator. The transfer function for a field access to $v.f$ at the location ℓ is:

$$\text{before}[\ell] = \text{after}[\ell] \setminus \{ f \} \text{ if } (v \neq \texttt{this}) \vee (\neg \text{locked}(\ell) \wedge \neg \text{init}(\ell))$$

It says that the field f is removed from the set when it is accessed through some local variable other than this or when the analysis encounters an unsynchronized access via this outside of an instance constructor. The symbol $\text{locked}(\ell)$ is a function expression that captures results of the stage A — for the given location ℓ, it says whether ℓ is in a region guarded by a lock over this in the respective method containing ℓ. The symbol $\text{init}(\ell)$ is a function expression that says whether ℓ belongs to an instance constructor or not. In all other cases, the transfer function is an identity.

The resulting set for a program point p contains also fields not accessed after p, but these are never queried by JPF.

5. EVALUATION

We implemented static analyses using the WALA library [19]. JPF API is used to retrieve information about the dynamic program state. Our implementation, experimental setup, and a redistributable subset of the benchmark programs is publicly available at `http://d3s.mff.cuni.cz/projects/formal_methods/jpf-static/spin14.html`.

Benchmarks. We evaluated the proposed approach on 9 multi-threaded Java programs: CRE Demo, the Daisy file system, the Elevator benchmark from the PJBench suite, Cache4j, and five benchmark programs from the CTC repository [1] (Alarm Clock, Linked List, Producer-Consumer, RAX Extended, and Replicated Workers) that involve non-trivial program logic and concurrency behavior. A brief description of each program follows.

CRE Demo is a high-level prototype of a software system for providing WiFi internet access at airports. The program 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 clients.

Daisy is a simple file system developed as a challenge problem for verification tools. We used it with a manually created test driver that runs two concurrent threads that perform various operations on files and directories.

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

Cache4j is a simple cache framework for Java objects that can be safely used in a multi-threaded environment. We configured the framework to use a blocking cache that prevents concurrent modification of the internal data structures and the LRU eviction algorithm. A part of the application is a test driver that runs two concurrent threads, which perform several operations with the cache (storing and retrieval of objects).

All five benchmark programs from the CTC repository involve multiple threads, which use synchronization operations (locking, calls of wait and notify) quite heavily.

Table 1 shows basic quantitative characteristics of all the benchmark programs — the total number of source code lines (Java LoC) and the maximal number of concurrently running threads.

Benchmark	Java LoC	Threads
CRE Demo	1,300	2
Daisy	800	2
Elevator	300	3
Cache4j	550	2
Alarm Clock	200	3
Linked List	180	2
Producer-Consumer	130	2
RAX Extended	150	3
Replicated Workers	400	2

Table 1: Benchmark programs

Experiments. The purpose of our experiments was to determine how many redundant thread choices can be eliminated using the may-happen-before analysis, and how much it improves the running time of JPF. We performed experimental comparison of the following three configurations:

1. original JPF without any hybrid analysis,
2. JPF combined only with the field access analysis, and
3. JPF combined with the may-happen-before analysis and the field access analysis.

We use the acronym JPF-FA for the second configuration, and the acronym JPF-MHB for the third configuration. In the experiments, we use the fully context-sensitive field access analysis, which is the most precise variant proposed in [16]. JPF traverses the whole state space in the case of every benchmark program, as none of the programs contain any error that JPF can detect.

Table 2 provides the results of our experiments. We report the number of thread choices created by JPF during the state space traversal, and the running time of JPF with the proposed hybrid analyses. The number of thread choices is equivalent to the number of program states that are explicitly saved by JPF for the purpose of backtracking and state matching.

The results show that usage of the may-happen-before analysis allows JPF to avoid many additional redundant thread choices and thread interleavings during the state space traversal. More specifically, the number of thread choices has been reduced for the following benchmarks: Daisy, Cache4j, Alarm Clock, Producer-Consumer, RAX Extended, and Replicated Workers. The biggest improvement has been achieved for Cache4j, where usage of the may-happen-before analysis reduces the number of thread choices by a factor of 8.57 compared with the standalone field access analysis, for the Producer-Consumer benchmark (factor of 4.07), and for Replicated Workers (2.39).

On the other hand, the may-happen-before analysis does not yield any improvement over the field access analysis for the remaining benchmarks. In the case of CRE Demo, the field access analysis itself eliminates all the redundant thread choices before field access statements. The Elevator benchmark contains synchronized accesses to array elements, but our analysis does not support accesses to array elements yet and therefore cannot eliminate any redundant thread choices in that case. Linked List is a program where usage of synchronization does not match the happens-before patterns defined in Section 3.

Running times show that the cost of the may-happen-before analysis is significantly greater than the cost of the field access analysis. This is evident on the results for those benchmark programs, in which case the may-happen-before analysis does not eliminate any additional thread choices over the field access analysis, such as CRE Demo and Elevator. However, for some other benchmarks, the speedup of JPF achieved due to the elimination of many additional redundant thread choices compensates for the increased cost of computing the hybrid analysis. The results for Daisy, Cache4j, and Replicated Workers show that the may-happen-before analysis is practically useful for more complex benchmark programs with large state spaces, as in that case it can eliminate a sufficiently high number of redundant thread choices to warrant its cost.

Precision of the may-happen-before analysis and the field access analysis depends very much on the underlying static pointer analysis. If two dynamic heap objects o_1 and o_2 have the same allocation site then the pointer analysis cannot distinguish them. JPF will make an unnecessary thread choice when, for example, one thread calls wait on o_1 and another thread may call notify on o_2. Additional unnecessary thread choices could be soundly eliminated for most of the benchmark programs with a more precise pointer analysis (e.g., flow-sensitive). We found that by manual inspection of the analysis results and thread choices made in JPF during the state space traversal.

6. RELATED WORK

There exist several categories of related approaches: (1) using dynamic analysis to compute the happens-before order for a particular execution path, (2) static analyses that detect conflicting accesses to heap objects, (3) static analysis-based techniques to eliminate redundant thread interleavings, (4) static data-flow analyses that operate on data structures that capture behavior of multiple threads, (5) static may-happen-in-parallel analyses, and (6) various combinations of static and dynamic analyses. We are not aware of any method to computing the happens-before order that uses only static analysis, and also not aware of any technique combining static analysis with information from the dynamic program state like we do. In the rest of this section, we describe selected approaches from each category and compare them with our approach proposed in this paper.

Category 1. Kahlon and Wang [12] recently proposed a unified happens-before model for a single execution trace and a correctness property. The model captures all possible interleavings of

benchmark	original JPF		JPF-FA		JPF-MHB	
	choices	time	choices	time	choices	time
CRE Demo	47,114	70 s	3,736	13 s	3,736	18 s
Daisy	28,120,251	18326 s	5,438,591	5153 s	4,907,351	4822 s
Elevator	10,116,121	6742 s	2,707,528	2163 s	2,707,528	2343 s
Cache4j	9,443,577	4966 s	9,417,206	6032 s	1,099,347	1064 s
Alarm Clock	413,996	298 s	145,594	108 s	114,472	102 s
Linked List	2,893	2 s	213	3 s	213	6 s
Producer-Consumer	6,095	3 s	1,499	4 s	368	7 s
RAX Extended	19,847	12 s	5,974	6 s	3,810	10 s
Replicated Workers	8,311,425	5172 s	1,202,710	771 s	503,039	531 s

Table 2: Experimental results

events from the given execution trace that are feasible with respect to happens-before constraints imposed by synchronization primitives. In particular, the model preserves the ordering between calls of wait and notify, and the ordering between the lock release statement followed by the acquisition statement on the same lock. The execution trace is acquired using dynamic analysis of the program, and then the happens-before model is inferred using an iterative algorithm. A limitation of this approach is that the model is sound and complete only when data values do not influence the control flow of program threads and their interaction.

The happens-before ordering is computed and used also by dynamic approaches to partial order reduction. In the approach proposed by Flanagan and Godefroid [7], the ordering is created dynamically for the set of all accesses to shared objects that occur on a given execution path. A heap object is marked as shared only when it is truly accessed by multiple threads on the execution path. Two field accesses executed by different threads are in the happens-before relation if they access the same dynamic heap object. Based on the happens-before ordering, the dynamic POR algorithm decides whether to make a thread choice at a field access or not. An advantage of the dynamic POR with the happens-before ordering is that it can precisely distinguish individual (dynamic) heap objects, and therefore it avoids creating some unnecessary thread choices that JPF would make. On the other hand, the approach described in [7] does not take into account whether a given object is reachable from multiple threads at the time of a particular access. For example, it makes a thread choice at a write access to some field in the object's constructor before the object escapes to the heap (multiple threads). Another limitation of this approach is that it does not use state matching, and thus it would need much more time than JPF to explore all distinct program behaviors for most benchmarks.

Specialized dynamic detectors of race conditions also construct the happens-before relation between events on a given execution trace (see, e.g., [5]). It is used to decide whether two memory accesses form a possible data race or not.

Category 2. The method proposed by von Praun and Gross [17] uses static analysis to detect shared heap objects and conflicts between field accesses on the shared objects. For each field access statement, the analysis finds the set of lock objects held by a thread performing the field access. Two field accesses to a heap object by different threads are considered as conflicting if the threads do not hold a common lock.

Category 3. The verification framework proposed by Kahlon et al. [11] uses static analysis together with abstract interpretation to eliminate redundant thread interleavings. As the first step, the framework creates a transaction graph for a given program using a simple approach to partial order reduction. The graph captures the control-flow of all threads, possible interaction between threads, and constraints imposed by synchronization primitives. Nodes of the graph represent program statements at which there must be a thread scheduling choice, and edges represent sequences of instructions that can be executed atomically. Static pointer analysis identifies shared heap objects through which threads may interact. An iterative algorithm based on static analysis is then used to remove nodes that represent statements that are provably not conflicting with other threads. Some thread T is possibly conflicting with the statement st in a given program state, if T may access the same object as st in the future and it will not block in the meantime. This approach supports locking operations and also signals (calls of wait and notify), but it uses only static analysis. It does not consider information from the dynamic program state, and therefore our approach proposed in this paper is more precise.

Category 4. Farzan and Kincaid [4] proposed a compositional static data-flow analysis for programs with nested locking. The analysis computes pairwise reachability of code locations from different threads, i.e. it uses a data structure that represents combined behavior of two threads.

Another technique in this category was proposed by Sinha and Wang [18]. It is a staged static analysis that operates also on a concurrent CFG that captures interactions of all program threads (field accesses on shared objects and thread synchronization). The limiting factor is the size and complexity of the concurrent CFG for large programs.

In our approach, we perform static analysis of individual program threads, and thus we do not have to cope with the size of data structures representing interactions of multiple threads. The happens-before ordering between statements in different threads is computed on demand when JPF needs the information to decide about thread choices.

Category 5. Naumovich et al. [15] designed and evaluated a static data-flow analysis that computes the may-happen-in-parallel information for program statements in different threads. Such analyses provide similar information as our may-happen-before analysis. However, they cannot be used as a direct replacement, because the happens-before ordering applies also to statements that actually cannot happen in parallel during the program execution due to synchronization between threads.

Category 6. Some tools for detecting races combine static analysis together with dynamic analysis [2, 6]. The basic approach consists of three steps: (1) use static analysis to identify memory accesses that must be checked for races, (2) instrument selected memory accesses in the given program, and (3) run the dynamic analysis that monitors all the instrumented accesses. A happens-before ordering between memory accesses and thread synchronization events

is typically constructed during the dynamic analysis step, and later used to detect the actual races.

7. CONCLUSION

The main contribution of this paper is the hybrid may-happen-before analysis, whose results are used by JPF to avoid creating unnecessary thread choices at field accesses. We found that usage of the may-happen-before analysis together with the field access analysis in JPF improves performance and scalability of the state space traversal over the previous work for several benchmark programs. On the other hand, there exist also multi-threaded programs for which the may-happen-before analysis does not yield any improvement over the field access analysis (when used in JPF).

In the near future, we would like to optimize our prototype implementation of the may-happen-before analysis to get faster execution times and reduce its memory consumption. We will investigate how much the analysis precision could be improved with usage of a flow-sensitive pointer analysis and more information from the dynamic program state (including execution history). Our long-term plans include (1) design and evaluation of an analysis that would identify globally-relevant accesses to array elements more precisely than the current JPF, and (2) extending the may-happen-before analysis with support for array elements.

Another possible line of future work is to explore applications of the hybrid analysis beyond JPF. This might include, for example, efficient detectors of data races and other concurrency errors.

8. ACKNOWLEDGMENTS

This work was partially supported by the Grant Agency of the Czech Republic project 13-12121P and the EU project ASCENS 257414. The work was also partially supported by the Grant Agency of the Charles University project 203-10/253297.

9. REFERENCES

[1] Concurrency Tool Comparison repository, `https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison`

[2] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. PLDI 2002.

[3] M. Dwyer, J. Hatcliff, Robby, and V. Ranganath. Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs. Formal Methods in System Design, 25, 2004.

[4] A. Farzan and Z. Kincaid. Compositional Bitvector Analysis for Concurrent Programs with Nested Locks. SAS 2010, LNCS, vol. 6337.

[5] C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. PLDI 2009.

[6] C. Flanagan and S. Freund. RedCard: Redundant Check Elimination for Dynamic Race Detectors. ECOOP 2013, LNCS, vol. 7920.

[7] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. POPL 2005.

[8] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032, 1996.

[9] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian Partial-Order Reduction. SPIN 2007, LNCS, vol. 4595.

[10] Java Pathfinder: a system for verification of Java programs, `http://babelfish.arc.nasa.gov/trac/jpf/`

[11] V. Kahlon, S. Sankaranarayanan and A. Gupta. Semantic Reduction of Thread Interleavings in Concurrent Programs. TACAS 2009, LNCS, vol. 5505.

[12] V. Kahlon and C. Wang. Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. CAV 2010, LNCS, vol. 6174.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Commun. of the ACM, 21(7), 1978.

[14] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI 2008.

[15] G. Naumovich, G.S. Avrunin, and L.A. Clarke. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. ESEC / FSE 1999, LNCS, vol. 1687.

[16] P. Parizek and O. Lhotak. Identifying Future Field Accesses in Exhaustive State Space Traversal. ASE 2011.

[17] C. von Praun and T. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. PLDI 2003.

[18] N. Sinha and C. Wang. Staged Concurrent Program Analysis. FSE 2010.

[19] WALA: T.J. Watson Libraries for Analysis, `http://wala.sourceforge.net/`