Computing Approximate Happens-Before Order with Static and Dynamic Analysis

Pavel Parízek, Pavel Jančík

Abstract: All 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 and unnecessary thread choices. 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 Java Pathfinder (JPF), which constructs the program state space on-the-fly, and therefore uses only information available in the current dynamic 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 in the future during program execution, and used it with 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. JPF uses 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 avoids making unnecessary thread choices. The may-happen-before analysis combines static data flow analysis and usage of information available from the dynamic program state.

This report describes the may-happen-before analysis, and provides results of experiments with several Java programs showing that usage of the may-happen-before analysis together with the field access analysis improves the scalability of JPF.

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.
Contents

1 Introduction 2

2 Overview 3

3 Using Happens-Before Order in JPF 5

4 May-Happen-Before Analysis 8
   4.1 May-Wait Analysis 8
   4.2 Must-Lock/Join Analysis 10
   4.3 Must-Wait Analysis 10
   4.4 May-Notify Analysis 10
   4.5 Future Allocations Analysis 10
   4.6 Lock Patterns Analysis 10

5 Evaluation 11

6 Related Work 12

7 Conclusion 13

References 14
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 [7] and CHESS [11]. 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 possible thread interleavings.

Tools employ partial order reduction (POR) [5] 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 different concurrently-running threads. Other statements are called thread-local. The set of globally-relevant statements contains, for example, start of a new thread, acquisition of a lock, 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., [4–6]) 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 exhaustive state space traversal of multi-threaded Java programs. JPF constructs the program state space on-the-fly using its custom virtual machine that interprets statements (bytecode instructions). Therefore, it cannot look ahead in the program execution to find what may happen in the future, and 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 reachable from multiple threads [2]. 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 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 [13], we designed a hybrid field access analysis, which provides information about fields possibly accessed in the future by program threads, 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 [13]. The analysis is hybrid because it combines static analysis with knowledge of the dynamic program state (thread call stacks) in order to improve precision. 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, however, 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.

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 [10] for field access statements and thread synchronization events. The may-happen-before analysis, too, is a specific combination of static analysis with dynamic analysis and information available from the dynamic program state. JPF uses the happens-before ordering to identify pairs of globally-relevant field access statements that cannot be
public class Example {
    public static int x = 0;

    public static void main(String[] args) {
        Object lock1 = new Object();
        Object lock2 = new Object();

        Thread th1 = new Writer(lock1);
        Thread th2 = new Reader(lock1);
        Thread th3 = new Notifier(lock2);

        th1.start(); th2.start(); th3.start();

        th1.join(); th2.join(); th3.join();
    }
}

class Notifier extends Thread {
    private Object lock2;

    public Notifier(Object lock2) {
        this.lock2 = lock2;
    }

    public void run() {
        synchronized (lock2) {
            lock2.notify();
        }
    }
}

class Writer extends Thread {
    private Object lock1;

    public Writer(Object lock1) {
        this.lock1 = lock1;
    }

    public void run() {
        synchronized (lock1) {
            lock1.notify();
        }
    }
}

class Reader extends Thread {
    private Object lock1;

    public Reader(Object lock1) {
        this.lock1 = lock1;
    }

    public void run() {
        synchronized (lock1) {
            lock1.wait();
        }

        int v = Example.x;
    }
}

Figure 1: Example program

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 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 [13] 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 the program counter in each thread refer to the first instruction, i.e. the program counter in the reader thread being at a location that precedes the call of wait (line 54) and the next instruction of the writer thread being the write access to Example.x (line 38). In this case, JPF together with our analyses must decide whether it must create a thread choice before the write access to cover all possible interleavings of the read and write accesses. 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 56). It means that the write access at line 38 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 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 checks 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 26) on a different monitor object. Consequently, the read access cannot be executed before the write access to Example.x in any thread interleaving due to the happens-before ordering, 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 with 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, JPF and the proposed analyses do not support these concurrency-related features of the Java platform: (1) the full Java Memory Model and (2) spurious wake-ups from the calls of wait that may happen on some platforms.

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, wait, notify, thread start, thread join). The approximate happens-before order specifies thread interleavings that cannot happen at runtime because of synchronization between threads.

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. The static phase of the hybrid analyses computes only partial information about the future behavior of individual program threads. We give more details in Section 4.

The second step is done at the dynamic analysis time in JPF, i.e. during the state space traversal. Full results of the hybrid analyses are computed in this step on demand using information from the dynamic program state. In particular, the happens-before ordering between statements from different threads is computed according to the current state and the program counter of each thread.

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. Assuming that the next statement in the currently executing thread $T_i$ 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 $T_j$, $j \neq i$ in the current dynamic program state, JPF 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 $o.f$ in the future after $p_j$. 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. If the results show that no other thread may access $o.f$ in the future, then the field access in $T_i$ 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 $T_i$, JPF checks results of the may-happen-before analysis to determine whether synchronization events impose a happens-before ordering between the access to $o.f$ in the current thread $T_i$ and the future access in the other thread $T_h$. We describe the procedure for checking results of the may-happen-before analysis in Section 4. If there exists such a happens-before ordering then only such thread interleavings, where the access to $o.f$ in $T_i$ precedes the access in $T_h$, are possible starting from the current dynamic state of the program, and JPF does not have to make a thread choice before the field access statement in $T_i$. Note that these checks must be done for each thread other than $T_i$ that may access $o.f$ in the future.
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.

3 Using Happens-Before Order in JPF

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. JPF queries results of the may-happen-before analysis for the state $s$ and both field accesses. There is a happens-before 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. We distinguish four code patterns (scenarios) in which $T_h$ may get blocked. Figure 2 shows simple code examples for all the patterns. Arrows indicate the happens-before ordering. 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 $T_h$ calls `notify` on a different monitor object $L2$.

JPF needs the following information about each thread to check both conditions:

(I1) whether there is a call of `wait` on every control-flow path before the nearest future access to $o.f$, and

(I2) calls of `wait` that may appear before the nearest future access to $o.f$, and

(I3) all future calls of `notify` until the end of thread’s lifetime.

All this information is provided by the may-happen-before analysis (Section 4).

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 happens-before ordering between the lock release statement in $T_c$ (line 3) and the lock acquisition statement in $T_h$ (line 1), if the thread $T_c$ holds the lock in the current state $s$. 

	

\begin{align*}
\begin{array}{cccc}
\text{Pattern 1: wait and notify (Figure 2a).} & \text{Pattern 2: lock acquisition and release (Figure 2b).}
\end{array}
\end{align*}
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 results of the may-happen-before analysis 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 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 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 2a, 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.

JPF needs the following information to detect this pattern:

(I4) the set of lock acquisition statements in \( T_h \) that appear on every control flow path between the current program point \( p_0 \) and the nearest future access to \( o.f \);

(I5) the set of allocation sites at which some dynamic heap objects may be allocated in future during the program execution (after the current program state \( s \));

(I6) locks held by \( T_c \) in the current program state just before the field access;

(I7) the number of dynamic heap objects allocated at each site associated with a lock variable during the program execution so far (up to the current state \( s \)).

Some of this information is provided by the may-happen-before analysis, and the rest is retrieved from the dynamic program state or computed during the state space traversal. In particular, the number of heap objects allocated at a given site is determined by a listener plugin to JPF that tracks object allocations. The information represented by the second and fourth item in the list above is used to determine whether a single object is ever allocated at the given site during program execution.

**Pattern 3: locking patterns that involve** this (Figure 2b). The thread \( T_h \) gets certainly blocked also in the case when both threads (i) access the field \( f \) on the same object \( o \) and (ii) guard the field access by a lock associated with \( o \). More specifically, if the following conditions are satisfied then \( T_h \) will block before the future access to \( o.f \).

- the current thread \( T_c \) 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 \( v \) (current object in the method performing the access);
- every conflicting future access to \( f \) in \( T_h \) via \( v \) outside of any instance constructor is guarded by a synchronized block over this;
- every conflicting future access to \( f \) in \( T_h \) via \( v \) in some instance constructor is performed on a different dynamic heap object than \( o \) accessed by \( T_c \);
- for each access to \( f \) via \( v \) 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 \( v \) 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. Similarly, boundaries of the synchronized blocks must be in the same method as the field access, because the local variable \( v \) may obviously point to different objects during execution of different methods. 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 object has not escaped from $T_h$ yet, and in particular when it is not being 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.

JPF needs the following information to check whether this pattern is satisfied in the current dynamic program state:

(I8) the list of fields accessed in $T_h$ only through the local variable this,

(I9) whether every access to a given field $f$ via this is performed inside a synchronized block over this (with the exception of instance constructors),

(I6) whether the thread $T_c$ holds the lock over $o$ right before the access to $o.f$, and

(I10) the dynamic call stack of $T_h$ to check whether it is executing an instance constructor on a dynamic heap object other than the target of the field access in $T_c$.

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 future access to $o.f$, and therefore gets blocked. There is a happens-before ordering between every statement in $T_c$ and the call of $\text{join}$ 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.

JPF needs the following information to detect this pattern:

(I11) the set of thread join statements in $T_h$ that appear on every control flow path between the current program point $p_h$ and the nearest future access to $o.f$;

(I5) the set of allocation sites at which some dynamic heap objects may be allocated in future during the program execution (after the current program state $s$);

(I12) the number of dynamic heap objects allocated at each site associated with a thread variable during the program execution so far (up to the current state $s$).

As in the previous cases, some of this information is provided by the may-happen-before analysis, and the rest is retrieved from the dynamic program state.

**Summary.** Considering all the patterns described in this section, JPF has to make a new thread choice before the access to $o.f$ in $T_c$ if some other thread $T_h$ may access $o.f$ in the future and one of the following conditions holds:

- there is an execution path in $T_h$ that does not contain any call of $\text{wait}$ before the access to $o.f$, and, in addition, there is an (possibly different) execution path that does not contain any lock acquisition statement before the access to $o.f$;

- there is a call of $\text{wait}$ on every execution path in $T_h$ before the nearest access to $o.f$, but some thread $T_c$ may call $\text{notify}$ on $l$ in the future, where $l$ is the possible target object of some call of $\text{wait}$ before the field access in $T_h$;

- there is a lock acquisition statement on every execution path in $T_h$ before the nearest access to $o.f$, but either $T_c$ does not hold any lock at the field access or it uses a different lock object than $T_h$ to guard the access to $o.f$.

In all other cases, the happens-before ordering between field access statements and thread synchronization events guarantees that the field access in $T_h$ cannot happen before the access in the current thread $T_c$ on any thread interleaving that may be executed from the current dynamic program state $s$, and thus JPF does not have to make a thread choice before the field access in $T_c$. 

---

4 May-Happen-Before Analysis

The may-happen-before analysis computes a safe approximation of the happens-before ordering between field access statements and synchronization events. It is a hybrid analysis that combines static analysis with the usage of information from the dynamic program state. Therefore, results of the analysis are valid only for the current program state \( s \) and the current program points of all threads in \( s \).

For each program point \( p \) in each thread \( T \) and for each target field \( f \), the analysis provides the following information:

- the set of calls of \( \text{wait} \) that \emph{may} appear between \( p \) and the nearest future access to \( f \) after \( p \) on any control flow path (I2 from the Section 3);
- whether there is a call of \( \text{wait} \) on every control-flow path starting in \( p \) before the nearest future access to \( f \) (I1);
- the set of all future calls of \( \text{notify} \) until the end of the thread’s lifetime on any control flow path starting in \( p \) (I3);
- the set of lock acquisition statements and thread join statements that \emph{must} appear on every control flow path starting in \( p \) before the nearest access to \( f \) (I4, I11);
- the set of allocation sites at which some dynamic heap objects may be allocated in the future during program execution (I5);
- the list of fields accessed in \( T \) after \( p \) only through the local variable \( \text{this} \) and inside synchronized blocks over this (I8, I9).

Note that the list contains only a part of the information required by JPF to decide whether it has to make a new thread choice. The remaining information (I6, I7, I10, I12) is retrieved directly from the current dynamic program state by JPF.

The whole may-happen-before analysis consists of several simple analyses — may-wait analysis, must-lock/join analysis, must-wait analysis, may-notify analysis, future allocations analysis, and lock patterns analysis. Each provides some of the information required by JPF. We describe each simple analysis below in the following subsections. We explain the combination of static analysis with information from the dynamic program state in full detail on the may-wait analysis, which is described first. Other analyses use the same general principle.

4.1 May-Wait Analysis

For each 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 \( \text{wait} \) that may appear between \( p \) and the nearest future access to \( f \). More specifically, each element of the set is an abstract target object (allocation site) of a call of \( \text{wait} \) that appears on some control flow path between \( p \) and the nearest access to \( f \) on that path. The set of possible abstract target objects for each call of \( \text{wait} \) is determined using results of the pointer analysis.

We distinguish between read accesses and write accesses to fields. The analysis results are computed in two stages (A and B). Each stage consists of a static phase followed by usage of information from the dynamic program state.

Stage A. This stage of the analysis computes for each program point \( p \) the set of calls of \( \text{wait} \) that may occur between \( p \) and the nearest future field access (to any field).

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 \( p \) only until the return from the method containing \( p \) (including nested method calls transitively). It is done using a backward flow-sensitive context-insensitive data flow analysis over the full inter-procedural control flow graph (ICFG) of the given thread \( T \).

Figure 3 shows transfer functions for the static backward analysis. When the analysis encounters a call of \( \text{wait} \), it adds every possible target monitor object \( o \) into the set. Transfer functions for field access statements produce the empty set. Also the transfer function for the return statement produces the empty set — it ensures that the set of data flow facts for each point contains only the calls of \( \text{wait} \) that may occur before the return from the current procedure. The transfer function for a call statement combines data for entry to the callee method and data for the next statement in the caller. 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.

Full results of this stage are computed at the dynamic analysis time (in JPF) using knowledge of the dynamic call stack of $T$ (which is a part of the dynamic program state). The dynamic call stack of $T$ specifies a sequence $p_0, p_1, . . . , p_n$ 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. We merge results of the static phase for all points $p_i$, $i = 0 . . . n$, to get the full results for $p_0$ and the current dynamic calling context of $p_0$. A consequence of this design is that the analysis 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 proper caller method and to the corresponding return point $p_i$. Note that the same approach is used also in field access analysis. The 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 $p$.

Stage B. The result of this stage for a program point $p$ is the set $\{ f_1, . . . , f_n \}$ of all the possible nearest field accesses that can happen after $p$. For each control flow path starting in $p$, the first (nearest) field access on the path is in the set.

As in the stage A, the static phase consists of a backwards flow-sensitive context-insensitive inter-procedural static data flow analysis and computes only a partial information. The set of data flow facts contains tuples $(o, f, l)$, where $o$ is the abstract target object, $f$ is the field name, and $l$ is the code location (program point). Figure 3 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. Transfer functions for call and return statements are the same as in the stage A. If there is no field access on the path is in the set.

The full analysis results are computed at the dynamic analysis time (in JPF) using the same approach as in the stage A. The only difference is that the special marks indicating the absence of a field access must be processed during the merge operation. 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, then the analysis merges-in the nearest 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 stages must be combined together and post-processed to get the set of calls of $wait$ that may occur between the point $p$ and the nearest access to the field $f$. We use the following approach.

A graph of field accesses is created from the results of the stage B. 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 nearest 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 stage A. For each pair \((f_a_i, f_{a,i+1})\) of field accesses in the sequence, the post-processor queries the results of the stage A for the program point corresponding to \(f_a_i\) to get the set of calls between \(f_a_i\) and \(f_{a,i+1}\). 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 \(p\).

### 4.2 Must-Lock/Join Analysis

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

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 stage A and by the post-processor when traversing the graph of field accesses.

### 4.3 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 nearest access to the field \(f\). We designed this analysis using the same general principle as in the stage A of the may-wait analysis — static backward inter-procedural data flow analysis (i.e., the static phase) combined with merging of results based on the dynamic call stack.

The result of the static phase for a given program point \(p\) is the set of fields accessed only after a call of wait. For every field \(f\) in the set there must be a call of wait between \(p\) and the nearest future access to \(f\) on every control flow path that starts in \(p\) and includes accesses to \(f\). Note that the set contains also fields not accessed after \(p\).

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.

### 4.4 May-Notify Analysis

This analysis too uses the same general principle as the stage A of the may-wait analysis — combining static data flow analysis with information from the dynamic program state. For each program point \(p\) in the thread \(T\), it collects the set of future calls of notify that may occur after \(p\) on any control flow path before the end of \(T\).

Target abstract objects for the calls of notify are 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.

### 4.5 Future Allocations Analysis

Also this analysis uses the same principle as the stage A of the may-wait analysis. Its result 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.

### 4.6 Lock Patterns Analysis

For each program point \(p\), this analysis finds a set \(\{f_1, f_2, \ldots, f_n\}\) of fields where each element \(f_i\) must satisfy the following conditions:

- every access to \(f_i\) on any control flow path after \(p\) is through the local variable this;
- if the access happens outside of an instance constructor, then it is guarded by a lock that is associated with the heap object represented by this.
The last condition means that, in Java, the field access must be performed inside a synchronized block defined over the variable 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 this 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 same principle as the stage A of the may-wait analysis, i.e. it involves flow-sensitive inter-procedural backward static data flow 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 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 \( \ell \) is:

\[
\text{before}[\ell] = \text{after}[\ell] \setminus \{ f \} \quad \text{if} \quad (v \neq \text{this}) \lor (\neg \text{locked}(\ell) \land \neg \text{init}(\ell))
\]

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 \( \ell \), it says whether \( \ell \) is in a region guarded by a lock over this in the respective method containing \( \ell \). The symbol \( \text{init}(\ell) \) is a function expression that says whether \( \ell \) belongs to an instance constructor or not. In all other cases, the transfer function is an identity.

### 5 Evaluation

We implemented static analyses using the WALA library [16]. JPF API is used to retrieve information about the dynamic program state.

**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 small programs from the CTC repository [1] (Alarm Clock, Linked List, Producer-Consumer, RAX Extended, and Replicated Workers). All benchmark programs involve thread synchronization (locking, calls of \textit{wait} and \textit{notify}). A brief description of each benchmark 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 \textit{wait} and \textit{notify}) quite heavily.

**Experiments.** We performed experimental comparison of three configurations: (1) JPF combined with the may-happen-before analysis and field access analysis, (2) JPF combined only with the field access
Table 1: Thread choices created by JPF

<table>
<thead>
<tr>
<th></th>
<th>original JPF</th>
<th>field access analysis</th>
<th>may-happen-before analysis + field access analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRE Demo</td>
<td>47114</td>
<td>3736</td>
<td>3736</td>
</tr>
<tr>
<td>Daisy</td>
<td>28120251</td>
<td>5720296</td>
<td>5179678</td>
</tr>
<tr>
<td>Elevator</td>
<td>10116121</td>
<td>3399440</td>
<td>3399440</td>
</tr>
<tr>
<td>Cache4j</td>
<td>9443577</td>
<td>9443577</td>
<td>6537678</td>
</tr>
<tr>
<td>Alarm Clock</td>
<td>413996</td>
<td>299433</td>
<td>253800</td>
</tr>
<tr>
<td>Linked List</td>
<td>2893</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td>Producer-Consumer</td>
<td>6095</td>
<td>2367</td>
<td>1041</td>
</tr>
<tr>
<td>RAX Extended</td>
<td>19847</td>
<td>11462</td>
<td>9772</td>
</tr>
<tr>
<td>Replicated Workers</td>
<td>8311425</td>
<td>1210149</td>
<td>504454</td>
</tr>
</tbody>
</table>

Table 1 provides the results of experiments. We report the number of thread choices created by JPF during the state space traversal.

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 Replicated Workers, where usage of the may-happen-before analysis reduces the number of thread choices by a factor of 2.4 compared with the standalone field access analysis, for the Producer-Consumer benchmark (factor of 2.27), and for Cache4j (1.44). Note that the standalone field access analysis does not eliminate any unnecessary thread choices for Cache4j, but usage of the may-happen-before analysis reduces the number of thread choices quite significantly.

On the other hand, there are several benchmark programs for which the may-happen-before analysis does not yield any improvement over the field access analysis. For example, 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 patterns defined in Section 3.

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, and (5) static may-happen-in-parallel 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.

Category 1. Kahlon and Wang [9] recently proposed a unified happens-before model for a single execution trace and a correctness property. The model captures all possible interleavings of 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.

Category 2. The method proposed by von Praun and Gross [14] 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. [8] 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 \( s \) in a given program state, if \( T \) may access the same object as \( s \) in the future and it will not block in the meantime. This approach supports locking operations and also signals (calls of \textit{wait} and \textit{notify}), but it uses only static analysis. It does not consider information from the dynamic program state, and therefore our approach is more precise.

**Category 4.** Farzan and Kincaid [3] 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 [15]. 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 behavior 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. [12] 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.

## 7 Conclusion

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 for some benchmark programs. On the other hand, there exist 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 competitive execution times and reduce its memory consumption. Analysis precision could be improved with usage of more precise pointer analysis and more information from the dynamic program state. We will investigate that and also evaluate components of the may-happen-before analysis separately to see which components are the most useful. 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.
References


