Effective Static Deadlock Detection

Mayur Naik
Intel Research
mayur.naik@intel.com

Chang-Seo Park and Koushik Sen
UC Berkeley
{parkcs,ksen}@cs.berkeley.edu

David Gay
Intel Research
david.e.gay@intel.com

Abstract

We present an effective static deadlock detection algorithm for Java. Our algorithm uses a novel combination of static analyses each of which approximates a different necessary condition for a deadlock. We have implemented the algorithm and report upon our experience applying it to a suite of multi-threaded Java programs. While neither sound nor complete, our approach is effective in practice, finding all known deadlocks as well as discovering previously unknown ones in our benchmarks with few false alarms.

1 Introduction

A deadlock in a shared-memory multi-threaded program is an unintended condition in which a set of threads blocks forever because each thread in the set is waiting to acquire a lock already held by another thread in the set. Today’s concurrent programs are riddled with deadlocks—a problem further exacerbated by the shift to multicore processors. For instance, roughly 6500/198,000 (∼ 3%) of the bug reports in Sun’s bug database at http://bugs.sun.com involve the keyword “deadlock”. Moreover, fixing other concurrency problems like races often involves introducing new synchronization, which in turn can introduce new deadlocks. Hence, static deadlock detection is valuable for testing and debugging such programs.

Previous static deadlock detection approaches are based on type systems [3,4], dataflow analyses [1,7,12,13,17,19,21], or model checking [5,6,11] (Section 7). The annotation burden for type-based approaches is often significant while model checking approaches currently do not scale to beyond a few thousand lines of code. Approaches based on dataflow analysis, on the other hand, have been applied to large programs but are highly imprecise.

In this paper, we present an effective static deadlock detection algorithm for Java (Section 3). Conceptually, our algorithm considers every tuple \( (t^a, l^1_1, l^1_2, l^2_1, l^2_2) \), where \( t^a, t^b \) denote abstract threads and \( l^1_1, l^1_2, l^2_1, l^2_2 \) denote lock acquisition statements, and checks if any pair of threads abstracted by \( t^a \) and \( t^b \) may deadlock by waiting to acquire a pair of locks \( z_1 \) and \( z_2 \) at \( l^2_1 \) and \( l^2_2 \), while already holding locks \( z_1 \) and \( z_2 \) at \( l^1_1 \) and \( l^1_2 \). Our key idea is to express the complex property of deadlock freedom for a pair of threads/locks—a problem that no existing static analysis can directly solve effectively—in terms of six problems that can be solved effectively using existing static analyses:

- **reachable**: In some execution of the program, can a thread abstracted by \( t^a \) reach \( l^2_1 \) and, after acquiring a lock at \( l^1_1 \), proceed to reach \( l^2_2 \) while still holding the lock (and similarly for \( t^b, l^1_1, l^1_2 \) )?
- **aliasing**: In some execution of the program, can a lock acquired at \( l^2_1 \) be the same as a lock acquired at \( l^2_2 \) (and similarly for \( l^1_1, l^1_2 ) \)?
- **escaping**: In some execution of the program, can a lock acquired at \( l^1_1 \) be accessible from more than one thread (and similarly for each of \( l^2_1, l^1_2, l^2_2) \)?
- **parallel**: In some execution of the program, can different threads abstracted by \( t^a \) and \( t^b \) simultaneously reach \( l^2_1 \) and \( l^2_2 \), respectively?
- **non-reentrant**: In some execution of the program, can a thread abstracted by \( t^a \) acquire a lock at \( l^2_1 \) it does not already hold and, while holding that lock, proceed to acquire a lock at \( l^2_2 \) it does not already hold (and similarly for \( t^b, l^1_1, l^1_2) \)? If the thread acquires the same lock it already holds then the second lock acquisition cannot cause a deadlock as locks are reentrant in Java.
- **non-guarded**: In some execution of the program, can different threads abstracted by \( t^a \) and \( t^b \) reach \( l^2_1 \) and \( l^2_1 \), respectively, without holding a common lock? If the two threads already hold a common lock then we call it a guarding lock (also called a gate lock [10]).

Each of these six necessary conditions is undecidable. Thus, any solution to each of them is necessarily unsound or incomplete. Our algorithm soundly approximates the first four conditions using well-known static analyses, namely, a call-graph analysis, a may-alias analysis, a thread-escape
analysis, and a may-happen-in-parallel analysis, respectively. Soundly approximating the last two conditions, however, requires a must-alias analysis, which is much harder than may-alias analysis. We address this problem using a common unsound solution: we use our may-alias analysis to masquerade as a must-alias analysis—as a result, we may fail to report some real deadlocks.

We may also report false deadlocks, either due to imprecision in our approximation of the six conditions, or because the deadlock is prevented by some condition not considered by our algorithm (Section 6). However, our approach is extensible: additional conditions, perhaps specific to the language or even the application at hand, can easily be added. In fact, the non-guarded and non-reentrant conditions specifically target Java programs. These idioms, if not identified, cause any static deadlock detector for Java to report overwhelmingly many false deadlocks [10, 21].

Our algorithm, while unsound and incomplete, is effective in practice. We have implemented it in a tool JADE (Section 4) and applied it to a suite of multi-threaded Java programs comprising over 1.5 MLOC. Our approach found all known deadlocks as well as discovered previously unknown ones in the suite, with few false alarms (Section 5).

2 Example

We first illustrate our approach on a real-world case: the JDK’s logging facilities from package java.util.logging. These facilities are provided as a library whereas our approach uses whole-program static analyses and thus requires a closed program, i.e., a complete program with a main method. So the first step in applying our approach to an open program such as a library is to build a harness that simulates clients exercising the interface of the program. Currently, we construct harnesses manually. Our algorithm is not path-sensitive and it ignores the values of primitive data. Hence, it neither requires a detailed, fully concrete harness nor test input data.

A snippet of our harness for this example is shown in class Harness in Figure 1. For brevity, we omit access qualifiers on classes, methods, and fields. Also, we label object allocation sites h1–h3, synchronized methods m1–m3, and thread run methods m4 and m5. The harness creates and starts two threads which we identify by their object allocation sites h1 and h2. Thread h1 calls static method LogManager.addLogger, which returns the unique logger having the specified name, creating the logger if it does not already exist in the global logger manager. This manager, allocated at site h3 and stored in static field LogManager.manager, maintains all existing loggers in a hashtable. On the other hand, thread h2 calls instance method LogManager.addLogger, which adds the specified logger to the global logger manager’s hashtable if it

```java
class Harness {
    static void main(String[] args) {
        Thread v1 = new Thread() {
            void run() {
                LogManager.addLogger(...);
            }
        }; v1.start();
        Thread v2 = new Thread() {
            void run() {
                LogManager.manager.addLogger(...);
            }
        }; v2.start();
        LogManager lm = LogManager.manager;
        Logger l = lm.getLogger(name);
        lm.addLogger(l);
        return 1;
    }
}
```

---

Figure 1. Example Java program.

```java
*** Stack trace of thread <Harness.java:11>:
LogManager.addLogger (LogManager.java:280)
- this allocated at <LogManager.java:155>
- waiting to lock <LogManager.java:155>
LogManager.addLogger (LogManager.java:280)
- waiting to lock <LogManager.java:155>
Harness$1.run (Harness.java:13)

*** Stack trace of thread <Harness.java:16>:
LogManager.addLogger (LogManager.java:226)
- this allocated at <LogManager.java:155>
- waiting to lock <LogManager.java:155>
LogManager.addLogger (LogManager.java:314)
- this allocated at <LogManager.java:155>
- waiting to lock <LogManager.java:155>
Harness$2.run (Harness.java:18)
```

---

Figure 2. Example deadlock report.
does not already contain a logger with that name.

Our algorithm reports the counterexample shown in Figure 2 for this program. It is similar to a thread stack dump output by a dynamic tool except that it is produced by a static tool and hence may denote a false deadlock. To improve usability we provide additional details to help users determine whether the counterexample denotes a real deadlock or a false positive. First, although we cannot provide concrete addresses of locks, we can identify their allocation sites. For instance, the counterexample in Figure 2 reports a deadlock between threads $h_1$ and $h_2$, identified by allocation sites $<\text{Harness.java}:11>$ and $<\text{Harness.java}:16>$, respectively. Likewise, instead of providing concrete addresses of locks, we provide abstract locks. An abstract lock is a set of abstract objects where each abstract object is a site at which the lock may be allocated. More generally, abstract objects may be sequences of multiple such sites, allowing different objects allocated at the same site to be distinguished (Section 3.1). In Figure 2, $<\text{LogManager.java}:155>$ denotes the lock on the LogManager object allocated at site $h_3$ at LogManeger.java:155 and stored in static field LogManager.manager while $\{\text{java.lang.Class}\}$ denotes the lock on the implicitly allocated java.lang.Class object stored in the implicit static field Logger.class.

Finally, each instance method called in each stack trace is coupled with an abstract object denoting the site at which the distinguished this variable of that method is allocated. In Figure 2, instance method LogManager.addLogger in either stack trace is called in a context in which this variable is allocated at site $h_3$ at LogManeger.java:155. In more complex programs, the same method may be analyzed in multiple contexts (Section 3.1).

It is easy to see that the above counterexample denotes a real deadlock: thread $h_1$ waits to acquire the lock on LogManager.manager while holding the lock on Logger.class, whereas thread $h_2$ waits to acquire the lock on Logger.class while holding the lock on LogManager.manager.

Our algorithm reports another counterexample for the above program; the only difference is that the topmost call in the stack trace of thread $h_1$ is to method LogManager.getLogger from call site Logger.getLogger (LogManager.java:228) instead of to method LogManager.addLogger from call site Logger.getLogger (LogManager.java:231). Both methods attempt to acquire the same lock, on LogManager.manager, and hence both counterexamples denote the same deadlock. In our experience, our algorithm’s ability to report all possible ways in which the same deadlock may occur helps in determining the best fix for the deadlock. In the above program, for instance, both counterexamples (and many similar ones resulting from parts of the program not shown here) contain the same last call in the stack trace of thread $h_2$, namely, code LogManager.getLogger (pname) at LogManager.java:314. Indeed, the fix for this deadlock is to replace this code by the inlined body of method LogManager.getLogger without its synchronization so that it does not hold the lock on Logger.class.

3 Algorithm

Our algorithm is based on sound and unsound approximations of our six necessary conditions (Section 3.2). Effectively approximating these conditions needs precise call-graph and points-to information—we use a form of combined call-graph and may-alias analysis called k-object-sensitive analysis [14] (Section 3.1). Finally, to improve usability, our algorithm generates and groups counterexamples to explain the deadlocks it detects (Section 3.3).

Before presenting our algorithm, we summarize our notation (Figure 3). Our algorithm takes as input a closed program with a main method denoted $m_{main}$. We use $M$ to denote the set of all method implementations that may be reachable from $m_{main}$. $M$ may be a crude over-approximation, e.g., one computed by Class Hierarchy Analysis (CHA). We use $m_{start} \in M$ to denote the start() method of class java.lang.Thread, the method used to explicitly spawn a thread. We use $V$ to denote the set of all local variables referenced by methods in $M$. We assume that each method $m \in M$ may be synchronized on any one of its arguments, specified by $\text{sync}((o, m))$ ($o$ is irrelevant but simplifies our notation), but does not contain any other synchronized blocks in its body. If method $m$ is not synchronized, then the partial function $\text{sync}$ is not defined at $(o, m)$. It is easy to transform any Java program to satisfy this restriction (Section 4).

Figure 3 also shows the relations produced by our four sound whole-program static analyses: call-graph analysis
method call sites are handled. The analysis begins in Figure 1, concentrating on how object allocation sites and may-happen-in-parallel analysis (mhp). These relations are the ones we need to define our deadlock detector; internally our analyses track information in greater detail (e.g., the may-alias analysis tracks the contents of the heap and static fields). We outline how pt and cg are computed in Section 3.1; we reuse the thread-escape analysis and may-happen-in-parallel analysis from earlier work [15].

### 3.1 k-Object-Sensitive Analysis

k-object-sensitive analysis [14] is an object sensitive, context sensitive, and flow insensitive analysis that computes call-graph and points-to approximations.

The analysis is object sensitive in that it can represent different objects allocated at the same site by potentially different abstract objects. An abstract object \( o \in \mathcal{O} \) is a finite sequence of object allocation sites denoted \([h_1 \ldots : h_n] \). The first allocation site \( h_1 \) is the represented object’s allocation site. The subsequent allocation sites \([h_2 \ldots : h_n] \) represent the object denoted by the distinguished this variable of the method where \( o \) was allocated—thus, that this object was allocated at \( h_2 \), in a method whose this object is represented by \([h_3 \ldots : h_n] \), and so on. For static methods, which lack the this variable, we represent the this object by [] (which represents no objects).

The analysis is also context sensitive in that it can analyze each method implementation in potentially multiple abstract contexts. An abstract context \( c \in \mathcal{C} \) is a pair \((o, m)\) of an abstract object \( o \) and a method \( m \) such that \( o \) abstracts the this object of \( m \); as above, for static methods \( o = [] \).

Finally, the analysis is flow insensitive as it computes global (instead of per program point) points-to information. This, however, does not adversely affect the precision of the analysis on local variables as our implementation operates on a Static Single Assignment (SSA) representation of the program (Section 4).

The analysis produces the following relations:

- \( cg \subseteq (\mathcal{C} \times \mathcal{C}) \), the context-sensitive call graph, contains each tuple \(((o_1, m_1), (o_2, m_2))\) such that method \( m_1 \) may call method \( m_2 \) with its this object abstracted by \( o_2 \) when the this object of \( m_1 \) is abstracted by \( o_1 \).

- \( pt \subseteq (\mathcal{C} \times \mathcal{V} \times \mathcal{O}) \), the points-to information for local variables, contains each tuple \((c, v, o)\) such that local variable \( v \) may point to abstract object \( o \) in abstract context \( c \).

We illustrate how k-object-sensitive analysis computes these relations for our running example from Figure 1, concentrating on how object allocation sites and method call sites are handled. The analysis begins by deeming reachable the contexts of the main method (\( [] \), Harness.main) and of every class initializer method (e.g., (\[] LogManager.<clinit>). As the analysis is flow insensitive, whenever a context \((o, m)\) is reachable, every statement in the body of \( m \) is reachable.

The analysis presumes a positive integer associated with each object allocation site, called the \( k \)-value of that site. Consider any such site \( v = \text{new}^h \ldots \), where \( h \in \mathbb{H} \) and \( v \in \mathcal{V} \), in a method \( m \) that the analysis deems reachable in a context \((o, m)\). Then, the analysis adds tuple \(((o, m), v, h\oplus_k o)\) to relation pt, where \( h\oplus_k o \) is a finite non-empty sequence of object allocation sites whose head is \( h \) and whose tail comprises at most the \( k - 1 \) most significant sites in \( o \) in order. Our deadlock detection algorithm automatically chooses potentially different \( k \)-values for different sites (Section 4). For our running example, however, we presume \( k = 1 \) for all sites. The initially reachable context (\( [] \), Harness.main) contains object allocation statements:

\[
v1 = \text{new}^{h_1} \ldots \text{ and } v2 = \text{new}^{h_2} \ldots
\]

and so the analysis adds the following tuples to pt:

\[
( ( [], Harness.main ), v1, [h1] )
\]
\[
( ( [], Harness.main ), v2, [h2] )
\]

If \( n(\ldots) \) is a static method call in a reachable context \((o, m)\), the analysis adds tuple \(((o, m), ([n]) \to cg\). Also, the analysis henceforth deems context \(([], n)\) reachable.

If \( v.n(\ldots) \) is an instance method call, then the target method depends upon the run-time type of the object denoted by \( v \). Every \(((o, m), v, [h_1 \ldots : h_n]) \in pt\) denotes a target in a potentially different context. The analysis thus adds \(((o, m), ([h_1 \ldots : h_n], n'))\) to cg, where \( n' \) is the target of a call to \( n \) for an object allocated at site \( h_1 \). We must also add \((([h_1 \ldots : h_n], n'), \text{this}, [h_1 \ldots : h_n])\) to pt—this treatment of the this argument is key to precision [14]. Also, the analysis henceforth deems context \(([h_1 \ldots : h_n], n')\) reachable. Furthermore, if \( n'' = m_{\text{start}} \) (a thread is started), the context \(([h_1 \ldots : h_n], n'')\) is also deemed reachable, where \( n'' \) is the run() method of the class allocated at \( h_1 \).

For our running example, since the analysis has deemed context (\([[]\), Harness.main) reachable and Harness.main contains calls v1.start() and v2.start(), the analysis adds the following tuples to cg:

\[
( ( [] , Harness.main ), ([h1], m_{\text{start}}) )
\]
\[
( ( [] , Harness.main ), ([h2], m_{\text{start}}) )
\]

and deems contexts \(((h1, m_4)\) and \((h2, m_5)\) of the respective run() methods reachable.

### 3.2 Deadlock Computation

Our deadlock detection algorithm represents threads (\( t \)) by the abstract context of the thread’s entry method, and
lock acquisitions \(l\) by the abstract context of synchronized methods; the latter suffices as we presume that methods do not contain any synchronized blocks in their body. We represent sets of held locks \(L\) by sets of abstract contexts of synchronized methods that acquire the corresponding locks.

Figure 4 defines some properties of threads, lock acquisitions, and lock sets that we derive from \(pt\) and \(cg\) and use in the rest of our algorithm. We use \(c_1 \rightarrow c_2 \triangleright L\) to denote that context \(c_2\) may be reachable from context \(c_1\) along some path in some thread, and, moreover, a thread executing that path may hold set of locks \(L\) upon reaching \(c_2\) (we elide \(\triangleright\) when the locks are irrelevant). We use \(\text{mayAlias}(l_1, l_2)\) to denote that lock acquisitions at \(l_1\) and \(l_2\) may acquire the same lock. We extend \(\text{mayAlias}\) to lock sets as usual.

We use reachability \(\rightarrow\) to approximate the set of startable threads and reachable lock acquisitions:

\[
\text{threads} = \{ x \mid \exists n : x \in \text{threads}_n \} \quad \text{where} \\
\text{threads}_0 = \{ ([], m_{\text{main}}) \} \\
\text{threads}_{n+1} = \text{threads}_n \cup \\
\quad \{ (o, \text{run}) | c \in \text{threads}_n \land c \rightarrow (o, m_{\text{start}}) \} \\
\text{locks} = \{ c | c' \in \text{threads} \land c' \rightarrow c \land \text{sync}(c) \text{ defined} \}
\]

For our running example, we have:

\[
\text{threads} = \{ ([], m_{\text{main}}), t_1, t_2 \} \\
\text{locks} = \{ l_1, l_2, l_3 \} \\
\quad t_1 \triangleq ([h_1], m_4) \quad t_2 \triangleq ([h_2], m_5) \\
\quad l_1 \triangleq ([], m_1) \quad l_2 \triangleq ([h_3], m_2) \quad l_3 \triangleq ([h_4], m_3)
\]

A deadlock six-tuple \(d = (t^a, l_1^a, l_2^a, l^b, l_1^b, l_2^b)\) denotes a deadlock involving a pair of locks \(z_1\) and \(z_2\) such that thread \(t^a\) holds lock \(z_1\) it acquired at synchronized method \(l_1^a\) and is waiting to acquire lock \(z_2\) at \(l_2^a\) while thread \(t^b\) holds lock \(z_2\) it acquired at synchronized method \(l_1^b\) and is waiting to acquire lock \(z_1\) at \(l_2^b\). Conceptually, our deadlock detection algorithm simply filters all potential deadlocks through our six necessary conditions, computing the final set of potential deadlocks to be reported as:

\[
\text{finalDeadlocks} = \{ d | d = (t^a, l_1^a, l_2^a, l^b, l_1^b, l_2^b) \land \text{reachableDeadlock} d \land \text{aliasedDeadlock} d \land \text{escapingDeadlock} d \land \text{parallelDeadlock} d \land \text{nonReentDeadlock} d \land \text{nonGrdedDeadlock} d \}
\]

Our six necessary conditions are formally defined in Sections 3.2.1–3.2.6 below.

For our running example, we focus on two potential deadlocks:

\[
d_1 \triangleq (t_1, l_1, l_2, t_2, l_2, l_1) \\
d_2 \triangleq (t_1, l_1, l_3, t_2, l_2, l_1)
\]

Each of these tuples denotes a possible deadlock between abstract threads \(t_1\) and \(t_2\). In both tuples, thread \(t_2\) holds a lock at \(l_2\) (context \([h_3], m_2\)) and is waiting to acquire a lock at \(l_1\) (context \([h_4], m_3\)). Also, in both tuples, in thread \(t_1\) holds a lock at \(l_1\), but it is waiting to acquire a lock at \(l_2\) in tuple \(d_1\) and a lock at \(l_3\) in tuple \(d_2\) as we will see, \(d_1\) and \(d_2\) pass all six conditions and are thus contained in \(\text{finalDeadlocks}\).

3.2.1 Computation of reachableDeadlock

For a tuple \((t^a, l_1^a, l_2^a, t^b, l_1^b, l_2^b)\) to be a deadlock our \textit{reachable} condition must be satisfied: Can a thread abstracted by \(t^a\) reach \(l_1^a\) and, after acquiring a lock at \(l_1^a\), proceed to reach \(l_2^a\) while still holding the lock (and similarly for \(t^b, l_1^b, l_2^b\)? Our algorithm uses the reachability property (Figure 4) to approximate this condition:

\[
\text{reachableDeadlock} (t^a, l_1^a, l_2^a, t^b, l_1^b, l_2^b) \quad \text{if} \quad t^a \rightarrow l_1^a \land l_1^a \rightarrow l_2^a \land t^b \rightarrow l_1^b \land l_1^b \rightarrow l_2^b
\]

For our running example, it is easy to see that thread \(t_1\) reaches \(l_1\), then \(l_3\) and subsequently \(l_2\), while \(t_2\) reaches \(l_2\) and then \(l_1\). Thus both tuples \(d_1\) and \(d_2\) satisfy \textit{reachableDeadlock}.

3.2.2 Computation of aliasedDeadlock

For a tuple \((t^a, l_1^a, l_2^a, t^b, l_1^b, l_2^b)\) to be a deadlock our \textit{aliased} condition must be satisfied: Can a lock acquired at \(l_1^a\) be the same as a lock acquired at \(l_2^a\) (and, similarly for \(l_1^b, l_2^b\))? Our algorithm uses the \textit{mayAlias} property (Figure 4) to approximate this condition:

\[
\text{aliasedDeadlock} (t^a, l_1^a, l_2^a, t^b, l_1^b, l_2^b) \quad \text{if} \quad \text{mayAlias}(l_1^a, l_2^a) \land \text{mayAlias}(l_1^b, l_2^b)
\]

For our running example, both tuples \(d_1\) and \(d_2\) satisfy \textit{aliasedDeadlock}: predicates \textit{mayAlias}(l_1^a, l_1^b) and \textit{mayAlias}(l_2^a, l_2^b)
mayAlias(l₂, l₂) hold trivially and hence tuple d₁ satisfies aliasingDeadlock; additionally, mayAlias(l₁, l₂) holds because abstract object [b₂] satisfies the two conjuncts in the definition of mayAlias, and hence tuple d₂ also satisfies aliasingDeadlock.

3.2.3 Computation of escapingDeadlock

The JDK contains many classes (e.g. java.util.Vector) with synchronized methods. When such objects cannot be accessed by more than one thread, they cannot participate in a deadlock. Thus, for a tuple (t², l²₁, l²₂, l²₃) to be a deadlock our escaping condition must be satisfied: Can a lock acquired at l²₁ be accessible from more than one thread (and similarly for each of l²₂, l²₃)?

We approximate this condition using a thread-escape analysis. Our application of this analysis to static deadlock detection appears novel and we quantify the need for it in our experiments (Section 5).

The thread-escape problem is usually defined as follows: “In some execution, is some object allocated at a given site h accessible from more than one thread?” To increase precision, we refine the notion of thread-escape to track when an object escapes. This allows the escaping condition to eliminate some deadlock reports on objects that later escape to other threads. Formally, (c, v) must be in relation esc if argument v of abstract context c may be accessible from more than one thread. Our escaping condition is thus:

\[
\text{escapingDeadlock}(t², l²₁, l²₂, l²₃) \text{ if } \\
(\{l²₁, \text{sync}(l²₁)\} \in \text{esc} \land \{l²₂, \text{sync}(l²₂)\} \in \text{esc} \land \\
(\{l²₃, \text{sync}(l²₃)\} \in \text{esc} \land \{l²₂, \text{sync}(l²₂)\} \in \text{esc} \\
\]

For our running example, LogManager.manager (l₂, l₁) and Logger.class (l₁), being static fields, clearly escape everywhere, and so both tuples d₁ and d₂ satisfy escapingDeadlock.

3.2.4 Computation of parallelDeadlock

For a tuple (t², l²₁, l²₂, l²₃, t³, l³₁, l³₂) to be a deadlock our parallel condition must be satisfied: Can thread t³ abstracted by t³ and t² simultaneously reach l²₂ and l³₂ respectively? The motivation for checking this condition is twofold. First, it eliminates each tuple (t, *, *, t, *, *) where t abstracts at most one thread in any execution. The most common example of such an abstract thread is (l₁, mₘₐᵢₐᵣ), but it also applies to any thread class allocated at most once in every execution. The second motivation is that even if different threads abstracted by t² and t³ may be able to reach l²₂ and l³₂ respectively, the thread structure of the program may forbid them from doing so simultaneously, namely, threads t² and t³ may be in a “parent-child” relation, causing l²₂ to happen before l³₂ in all executions.

We approximate these two conditions using a may-happen-in-parallel analysis that computes relation mhp which contains each tuple (t₁, (o, m), t₂) such that a thread abstracted by t₂ may be running in parallel when a thread abstracted by t₁ reaches method m in context o. Our may-happen-in-parallel analysis is simple and only models the program’s thread structure, ignoring locks and other kinds of synchronization (fork-join, barrier, etc). Our parallel condition is thus:

\[
\text{parallelDeadlock}(t², l²₁, l²₂, t³, l³₁, l³₂) \text{ if } \\
(t², l²₂, t³) \in \text{mhp} \lor (t³, l³₂, t²) \in \text{mhp} \\
\]

For our running example, clearly nothing prevents t₁ and t₂ from running in parallel, so tuples d₁ and d₂ satisfy parallelDeadlock.

3.2.5 Computation of nonReentDeadlock

In Java, a thread can re-acquire a lock it already holds. This reentrant lock acquisition cannot cause a deadlock. Thus, for a tuple (t², l²₁, l²₂, t³, l³₁, l³₂) to be a deadlock our non-reentrant condition must be satisfied: Can a thread abstracted by t² acquire a lock at l²₁ it does not already hold and, while holding that lock, proceed to acquire a lock at l²₂ it does not already hold (and similarly for l³₁, l³₂)?

Soundly identifying reentrant locks requires must-alias analysis. Must-alias analysis, however, is much harder than may-alias analysis. Instead, we use our may-alias analysis itself to unsoundly check that whenever a thread abstracted by t acquires a lock at l₁ and, while holding that lock, proceeds to acquire a lock at l₂, then the lock it acquires at l₁ or l₂ may (soundness requires must) be already held by the thread—a property approximated by reentrant:

\[
\text{reentrant}(t, l₁, l₂) \iff l₁ = l₂ \lor \\
(\forall L₁ : (t \rightarrow l₁ \lor L₁ \Rightarrow \text{mayAlias}(\{l₁, l₂\}, L₁))) \lor \\
(\forall L₂ : (t \rightarrow l₂ \lor L₂ \Rightarrow \text{mayAlias}(\{l₁, l₂\}))) \\
\]

Intuitively, the first conjunct checks that the locks acquired at l₁ and l₂ may be the same. The second conjunct checks that when a thread abstracted by t reaches up to but not including l₁, the set of locks L₁ it holds may contain the lock it will acquire at l₁ or l₂. The third conjunct checks that when the thread proceeds from l₁ and reaches up to but not including l₂, the set of locks L₂ it holds may contain the lock it will acquire at l₂. Next, we use the reentrant predicate to approximate our non-reentrant condition as follows:

\[
\neg\text{reentrant}(t², l²₁, l²₂) \land \neg\text{reentrant}(t³, l³₁, l³₂) \\
\]

The above approximation itself is sound but the approximation performed by the reentrant predicate it uses is unsound; thus, a tuple that does not satisfy nonReentDeadlock is not provably deadlock-free.
For our running example, the two locks acquired by either thread do not alias, and no locks are acquired prior to the first lock or between the first and second lock in either thread, so tuples \( d_1 \) and \( d_2 \) satisfy nonReentDeadlock.

### 3.2.6 Computation of nonGrdedDeadlock

One approach to preventing deadlock is to acquire a common guarding lock in all threads that might deadlock. Thus, for a tuple \( (t^a, l^a_1, l^a_2, t^b, l^b_1, l^b_2) \) to be a deadlock our non-guarded condition must be satisfied: Can threads abstracted by \( t^a \) and \( t^b \) reach \( l^a_1 \) and \( l^b_1 \), respectively, without already holding a common lock?

Soundly identifying guarding locks, like reentrant locks, needs a must-alias analysis. We once again use our may-alias analysis to unsoundly check whether every pair of threads abstracted by \( t^a \) and \( t^b \) (soundness requires must) hold a common lock whenever they reach \( l^a_1 \) and \( l^b_1 \), respectively — a property approximated by guarded:

\[
\text{guarded}(t^a, t^b, l^a_1, l^b_1) \iff \forall L_1, L_2: (t^a \rightarrow t^b \triangleright L_1 \land t^b \rightarrow t^b \triangleright L_2) \implies \text{mayAlias}(L_1, L_2)
\]

Then, we use the guarded predicate to approximate our non-guarded condition as follows:

\[
\text{nonGrdedDeadlock}(t^a, l^a_1, l^a_2, t^b, l^b_1, l^b_2) \iff \neg\text{guarded}(t^a, l^a_1, l^a_2, t^b, l^b_1, l^b_2)
\]

The above approximation itself is sound but the approximation performed by the guarded predicate it uses is unsound; thus, a tuple that does not satisfy nonGrdedDeadlock is not necessarily deadlock-free.

For our running example, as we saw for nonReentDeadlock, no locks are acquired prior to the first lock, so tuples \( d_1 \) and \( d_2 \) satisfy nonGrdedDeadlock.

### 3.3 Post-Processing

Our algorithm reports a counterexample for each tuple in finalDeadlocks. The counterexample reported for a tuple \( (t^a, l^a_1, l^a_2, t^b, l^b_1, l^b_2) \) in finalDeadlocks consists of a pair of paths \( P_1 \) and \( P_2 \) in the context-sensitive call graph denoting possible stack traces of threads abstracted by \( t^a \) and \( t^b \), respectively, at the point of the deadlock. Specifically, \( P_1 \) is the shortest path from \( t^a \) to \( l^a_2 \) via \( l^a_1 \) and, similarly, \( P_2 \) is the shortest path from \( t^b \) to \( l^b_2 \) via \( l^b_1 \). Unlike stack traces reported by a dynamic tool, however, paths \( P_1 \) and \( P_2 \) may be infinite; we aid the user in comprehending them by providing additional details such as the context in which each instance method along each of those paths is called and the set of abstract objects of each lock that is synchronized along each of those paths. For our running example, Figure 2 shows the counterexample reported for tuple \( d_1 \).

Our algorithm also groups together counterexamples likely to be symptoms of the same deadlock. For each tuple \( (t^a, l^a_1, l^a_2, t^b, l^b_1, l^b_2) \) in finalDeadlocks, it computes a pair of lock types \( (\tau_1, \tau_2) \) as the least upper bounds of the types of abstract objects in the points-to sets \( O_1 \) and \( O_2 \) of the two locks involved in the deadlock where:

\[
O_1 = \{ o \mid (l^a_1, \text{sync}(l^a_1), o) \in \text{pt} \land (l^b_1, \text{sync}(l^b_1), o) \in \text{pt} \} \quad O_2 = \{ o \mid (l^b_2, \text{sync}(l^b_2), o) \in \text{pt} \land (l^a_1, \text{sync}(l^a_1), o) \in \text{pt} \}
\]

Then, our algorithm groups together the counterexamples reported for tuples in finalDeadlocks that have the same pair of lock types. For our running example, both tuples \( d_1 \) and \( d_2 \) have the same pair of lock types \( \{\text{java.lang.Class}, \text{java.util.logging.LogManager}\} \). Hence, our algorithm groups their counterexamples together.

### 4 Implementation

We implemented our deadlock detection algorithm in a tool called JADE. JADE takes as input a closed Java program in bytecode form and, optionally, as source code (the latter is used only to report source-level counterexamples). It uses the Soot framework \[18\] to construct a 0-CFA-based call graph to determine the set \( \mathcal{M} \) of all methods that may be reachable from the main method. It rewrites each synchronized block synchronized (\( v \) \{ \( s \} \)) as a call to a fresh static method, synchronized on argument \( v \) with body \( s \). It then converts the program into Static Single Assignment (SSA) form to increase the precision of the flow-insensitive \( k \)-object-sensitive analysis.

JADE then uses the results of \( k \)-object-sensitive analysis to perform the thread-escape and may-happen-in-parallel analyses. All three analyses are expressed in Datalog and solved using bddbddb \[20\], a Binary Decision Diagram (BDD)-based Datalog solver. BDDs compactly represent the input relations, such as those representing basic facts about the program (e.g., function \( \text{sync} \)), as well as the relations output by these analyses (e.g., pt, \( \text{mbp} \), etc.). Finally, JADE runs our deadlock detection algorithm which is also expressed in Datalog and computes relation finalDeadlocks that approximates the set of tuples satisfying our six necessary conditions for a deadlock.

Our implementation of \( k \)-object-sensitive analysis is parameterized by three parameters:

- \( \mathcal{M} \subseteq \mathcal{M} \) containing each method that must be analyzed context-insensitively (i.e., in the lone context \( [] \)).
- \( \forall \subseteq \mathcal{V} \) containing each local variable whose points-to information must be maintained context-insensitively (i.e., in the lone context \( [] \)).
- \( \mathcal{K} : \mathbb{H} \rightarrow \mathbb{N}^+ \) mapping each object allocation site to a positive integer called its \( k \)-value (Section 3.1).
For scalability, our $k$-object-sensitive analysis uses an iterative refinement-based approach: we run all three analyses and the deadlock detection algorithm in each iteration using increasingly refined $M$, $V$, and $K$. In the first iteration, the cheapest possible $k$-object-sensitive analysis is run, using $M = M_0$, $V = V_0$ and $K = \lambda h.1$, which is effectively a 0-CFA-based analysis, and finalDeadlocks is computed. These deadlocks, however, typically contain many false positives due to the imprecision of 0-CFA-based analysis (Section 5). Hence, instead of being reported, they are used to refine parameters $M$, $V$, and $K$ and the $k$-object-sensitive analysis is re-run. The refinement algorithm considers each tuple in finalDeadlocks as an effect of imprecision and finds all its possible causes in terms of $M$, $V$, and $K$ (e.g., a certain method must be analyzed context-sensitively, the $k$-value of a certain site must be increased, etc.). Since the other analyses depend upon $k$-object-sensitive analysis, they are also re-run, and finally the deadlock detection algorithm itself is re-run to compute a new finalDeadlocks. The process terminates either if finalDeadlocks is empty or if its size begins to grow; the latter termination criterion prevents overwhelming the user with too many reports denoting the same deadlock.

5 Experiments

We evaluated JADE on a suite of multi-threaded Java programs comprising over 1.5 MLOC. The suite includes the multi-threaded benchmarks from the Java Grande suite (moldyn, montecarlo, and raytracer); from ETH, a traveling salesman problem implementation (tsap), a successive over-relaxation benchmark (sox) and a web crawler (hedec); a website download and mirror tool (weblech); a web spider engine (jspider); WC3’s web server platform (jigaws); and Apache’s FTP server (ftpd). The suite also includes open programs for which we manually wrote harnesses: Apache's database connection pooling library (dbcp); a fast caching library (cache4j); the JDK4 logging facilities (logging); and JDK4 implementations of lists, sets, and maps wrapped in synchronized collections (collections). All our benchmarks along with JADE’s deadlocks reports are available at http://chord.stanford.edu/deadlocks.html

Table 1 summarizes JADE’s results. The ‘LOC’, ‘classes’, ‘methods’, and ‘syncs’ columns show the numbers of lines of code, classes, methods, and synchronized statements deemed reachable from the main method by Soot’s 0-CFA-based analysis. The ‘time’ column provides the total running time of JADE. The experiments were performed on a 64-bit Linux server with two 2GHz Intel Xeon quad-core processors and 8GB memory. JADE, however, is single-threaded and 32-bit, and hence utilizes only a single core and at most 4GB memory.

The ‘0-CFA’ and ‘$k$-obj.’ columns give the size of finalDeadlocks after one and two iterations of our algorithm (Section 4)—finalDeadlocks is empty or starts to grow, and JADE terminates, after at most two iterations for all our benchmarks. The first iteration uses a $k$-object-sensitive analysis that is essentially a 0-CFA-based analysis (Section 4). The difference between the two columns, most notable for hedec, weblech, jspider, ftp, and dbcp, is the number of extra false positives that would be reported by a 0-CFA-based analysis over a $k$-object-sensitive one. All previous static deadlock detectors we are aware of employ a 0-CFA-based analysis or an even more imprecise CHA-based analysis; moreover, they exclude checking one or more of our six necessary conditions (Section 7).

Figure 5 justifies the need for the escaping, parallel, non-reentrant and non-guarded conditions—we consider the reachable and aliasing conditions fundamental to our deadlock definition. We measure the effectiveness of a particular condition by switching it off and observing the increase in the size of finalDeadlocks. The graphs exclude benchmarks moldyn, raytracer, sox, and cache4j as the size of finalDeadlocks is not noticeably affected for any of them by switching off any single condition—these benchmarks are relatively small and have a relatively simple synchronization structure (indicated by the numbers in the ‘sync’ column in Table 1) and gain no significant benefit from any one condition.

The left graph in Figure 5 shows the effectiveness of the sound escaping and parallel conditions. The bars are normalized to the number of deadlocks obtained by checking only the reachable and aliasing conditions. The ‘soundDeadlocks’ partition of each bar denotes the number of deadlocks obtained by checking all four sound conditions. The ‘only Par.’ (resp. ‘only Esc.’) partition denotes the number of deadlocks that are soundly filtered out exclusively by parallel (resp. escaping). The ‘Esc. or Par.’ partition denotes the number of deadlocks that are filtered out by both parallel and escaping. The right graph in Figure 5 shows the effectiveness of the unsound non-reentrant and non-guarded conditions. The bar for each benchmark in this graph further partitions the ‘soundDeadlocks’ partition of the bar for the corresponding benchmark in the left graph. The ‘finalDeadlocks’ partition denotes the size of finalDeadlocks. The ‘only N.G.’ (resp. ‘only N.R.’) partition denotes the number of deadlocks that are filtered out exclusively by non-guarded (resp. non-reentrant). Finally, the ‘N.R. or N.G.’ partition denotes the number of deadlocks that are filtered out by both non-guarded and non-reentrant. In summary, we see that each condition is important for some benchmark.

Our algorithm generates a counterexample for each tuple in finalDeadlocks reported under column ‘$k$-obj.’ in Table 1. These counterexamples are grouped by the pair of
The ‘total’ column in the table denotes the total number of such groups for each benchmark. The last ‘real’ column denotes the number of groups which contain at least one real deadlock. We confirmed real deadlocks by creating concrete test cases that were able to exhibit them. The deadlocks in hedc, weblech, jigsaw, and ftp were not in application code but in the JDK’s logging facilities implemented in java.util.logging. This was our primary motivation for studying the logging benchmark; all the deadlocks reported for the above benchmarks are also included in those reported for logging. Additionally, this benchmark includes the previously known deadlock that is explained in our running example (Section 2) but is not triggered by any of the other benchmarks.

We found three application-level deadlocks in dbcp of which one was previously known. Finally, all deadlocks reported in benchmark collections are real and previously known. Strictly speaking, these are not bugs in the JDK collections per se but they indicate ways in which clients could erroneously use those collections and trigger deadlocks. We included collections, studied in previous work on deadlock detection [21], to confirm that our unsound approach could find all known deadlocks.

6 Limitations

Our deadlock detection algorithm is unsound. We begin by noting that it only reports deadlocks between two threads and two locks. Deadlocks between more than two threads/locks are possible and it is easy at least in principle to extend our approach to detect such deadlocks. However, empirical evidence from bug databases of popular open-source Java programs, such as http://bugs.sun.com
and http://issues.apache.org, shows that the vast majority of deadlocks involve only two threads/locks (in fact, we did not encounter a single deadlock involving more than two threads/locks in perusing the above databases).

Our algorithm detects reentrant locks and guarding locks unsoundly (Sections 3.2.5 and 3.2.6). Two promising future directions are to check our non-reentrant condition using the form of must-alias analysis used to check finite-state properties [8] and to check our non-guarded condition using the form of must-alias analysis used to check races [16].

The key source of false positives in our experiments is the relatively imprecise thread-escape analysis used by our algorithm. Existing work on this analysis was driven primarily by the need to eliminate redundant synchronization in Java programs and subsided in recent years after modern JVMs diminished the run-time speedups achieved by this optimization. We hope our application of this analysis to static deadlock detection, and in our earlier work to static race detection [15], will renew advances in this analysis.

Our algorithm only detects deadlocks due to lock-based synchronization whereas other kinds of synchronization, notably wait-notify in Java, can cause deadlocks as well which our algorithm does not report.

Finally, our implementation ignores the effects of native methods and reflection in Java though we mitigate this problem by manually providing “stubs” for common native methods and annotations for statically resolving dynamic class loading sites in the JDK library.

7 Related work

Previous work on deadlock detection for shared-memory multi-threaded programs includes static approaches based on type systems, dataflow analysis, or model checking, as well as dynamic approaches.

7.1 Type Systems

Boyapati et al. [3, 4] present an ownership type system for Java that allows programmers to specify a partial order among locks. The type checker statically ensures that well-typed programs are deadlock-free. Our approach is unsound and cannot prove deadlock freedom. On the other hand, it does not require annotations and scales to larger programs.

7.2 Dataflow Analysis

Artho and Biere [1] augment Jlint, a static dataflow analysis based bug-finding tool for Java, with checks for several patterns that could indicate deadlocks. It performs local (per class or per method) analyses and cannot, for instance, infer that syntactically different expressions or synchronized blocks in methods of different classes may hold the same lock. Jlint lies in the category of lightweight tools that are unsound and incomplete but target common bug patterns and scale well; another similar tool is LockLint for C [17].

Von Praun [19] presents an algorithm that performs whole-program 0-CFA-based call-graph and may-alias analysis of Java programs to compute the static lock-order graph and reports cycles in it as possible deadlocks. Unlike our approach, his algorithm can report deadlocks involving more than two threads/locks. Like our approach, however, his algorithm is unsound and incomplete, and it checks necessary conditions for a deadlock that amount to our reachable, aliasing, and non-reentrant conditions, but not our parallel, escaping, and non-guarded conditions.

Williams et al. [21] present an algorithm that traverses the given Java program’s call graph bottom-up and builds a lock-order graph summary per method. It then merges the summaries of thread entry methods into a global lock-order graph by unifying may-aliasing lock nodes together, and reports cycles in it as potential deadlocks. Unlike our approach, their algorithm can report deadlocks involving more than two threads/locks. Also, unlike our unsound checking of the non-reentrant condition, they handle reentrant locks soundly, but only detect them when lock expressions are local variables (as opposed to fields). This coupled with their CHA-based call-graph and may-alias analysis (which is less precise than a 0-CFA-based one) and the lack of checking of the parallel, escaping, and non-guarded conditions leads to significant imprecision which they address by applying several unsound heuristics.

Engler and Ashcraft [7] present RacerX, a static tool that performs flow-sensitive interprocedural analysis of C programs to compute the static lock-order graph and reports cycles in it as possible deadlocks. Their approach scales well but is highly imprecise and employs heuristics for ranking the deadlock reports in decreasing order of likelihood.

Masticola et al. [12, 13] present sound deadlock detection algorithms for various parallelism and synchronization models mainly in the context of Ada. A key aspect of their approach is non-concurrency analysis which may be viewed as the counterpart of our may-happen-in-parallel analysis.

7.3 Model Checking

The SPIN model checker has been used to verify deadlock freedom for Java programs by translating them into Promela, SPIN’s modeling language [6, 11]. Model checking based on counterexample-guided abstraction refinement has also been applied to deadlock detection in message passing based C programs [5]. A general limitation of model checking approaches is that they presume that the input program has a finite and tractable state-space.
7.4 Dynamic Analysis

While deadlocks actually occurring in executions are easy to detect, dynamic approaches such as Visual Threads [9] monitor the order in which locks are held by each thread in an execution and report cycles in the resulting dynamic lock-order graph as potential deadlocks that could occur in a different execution. The Goodlock algorithm [2,10] extends this approach to reduce false positives, namely, it tracks thread fork/join events and guarding locks that render cycles infeasible; this is akin to checking our parallel and non-guarded conditions, respectively. Like any dynamic analysis, these approaches are inherently unsound and cannot be applied to open programs and without test input data.

8 Conclusion

We have presented a novel static deadlock detection algorithm for Java that uses four static analyses to approximate six necessary conditions for a deadlock. We have implemented and applied it to a suite of multi-threaded Java programs comprising over 1.5 MLOC. While unsound and incomplete, our approach is effective in practice, finding all known deadlocks as well as discovering previously unknown ones in our benchmarks with few false alarms.

9 Acknowledgment

We would like to thank Pallavi Joshi, Christos Stergiou, and the anonymous reviewers for their valuable comments. This research was supported in part by a generous gift from Intel, by Microsoft and Intel funding (award #DIG07-10227) and by matching funding by U.C. Discovery (award #DIG07-10227).

References