Detection of deadlock potentials in multithreaded programs

R. Agarwal
S. Bensalem
E. Farchi
K. Havelund
Y. Nir-Buchbinder
S. D. Stoller
S. Ur
L. Wang

Concurrent programs are well known for containing errors that are difficult to detect, reproduce, and diagnose. Deadlock is a common concurrency error, which occurs when a set of threads are blocked, due to each attempting to acquire a lock held by another. This paper presents a collection of highly scalable static and dynamic techniques for exposing potential deadlocks. The basis is a known algorithm, which, when locks are acquired in a nested fashion, captures the nesting order in a lock graph. A cycle in the graph indicates a deadlock potential. We propose three extensions to this basic algorithm to eliminate, or label as low severity, false warnings of possible deadlocks (false positives). These false positives may be due to cycles within one thread, cycles guarded by a gate lock (an enclosing lock that prevents deadlocks), and cycles involving several code fragments that cannot possibly execute in parallel. We also present a technique that combines information from multiple runs of the program into a single lock graph, to find deadlock potentials that would not be revealed by analyzing one run at a time. Finally, this paper describes the use of static analysis to automatically reduce the overhead of dynamic checking for deadlock potentials.

Introduction

Concurrent programs are well known for containing errors that are difficult to detect, reproduce, and diagnose. Some common programming errors include data races and deadlocks. A data race occurs when two or more threads concurrently access a shared variable, at least one of the accesses is a write, and no mechanism is used to enforce mutual exclusion. Data races can be avoided by proper use of locks. However, the use of locks introduces the potential for deadlocks. Two types of deadlocks, namely, *resource deadlocks* and *communication deadlocks*, are discussed in the literature [1, 2]. In the case of resource deadlocks, a set of threads are deadlocked if each thread in the set is waiting to acquire a lock held by another thread in the set. In the case of communication deadlocks, threads wait for messages or signals that do not occur. In the Java** programming language, resource deadlocks result from the use of synchronized methods and synchronized statements. Communication deadlocks result from the use of the wait and notify primitives. The algorithms presented in this paper address resource deadlocks, from now on referred to as *deadlocks*, illustrated by example programs written in Java.

Deadlocks can be analyzed using a variety of techniques, such as model checking (using algorithms that explore all possible behaviors of a program), dynamic analysis (analyzing only one or just a few executions), and static analysis (analyzing the source code without executing it). Model checking is computationally expensive and often impractical for large software applications. Static analysis can guarantee that all executions of a program are deadlock free but often yields false warnings of possible deadlocks, also called *false positives* or *false alarms*. Dynamic analysis generally produces fewer false alarms, which is a significant practical advantage because diagnosing all of the warnings from static analysis of large code bases may be time consuming. However, dynamic analysis may still yield false positives, as well as false negatives (missed errors), and requires code instrumentation that results in a slow down of the analyzed program. This paper addresses these

Digital Object Identifier: 10.1147/JRD.2010.2060276

0018-8646/10/$5.00 © 2010 IBM

three problems related to dynamic analysis. In particular, we discuss improving a known dynamic analysis algorithm and the use of static analysis to reduce runtime overhead.

The fundamental idea behind dynamic analysis is the known result [3] that deadlock potentials can be exposed by analyzing locking order patterns in an execution trace from a nondeadlocking run of the program. The technique consists of building a lock graph and searching for cycles within the graph. Nodes in the graph are locks. The graph contains an edge (a connection) from lock l_1 to lock l_2 if a thread at some point holds l_1 while acquiring l_2. A cycle in the graph (i.e., a sequence of edges that begins and ends at the same node) indicates a deadlock potential. The algorithm detects deadlock potentials very effectively, independently of whether the program actually deadlocks during the particular run that is analyzed. This is evidenced by the comparative study documented in [4]. However, the algorithm has three classes of shortcomings, all addressed in this paper: false positives, false negatives, and runtime overhead, as outlined in the following.

False positives occur when the basic algorithm reports deadlock potentials in cases where no deadlock is possible. This paper proposes three extensions to the basic algorithm to identify false positives due to 1) cycles within one thread, 2) cycles guarded by a gate lock (an enclosing lock that prevents interleaving of nested locks that could lead to deadlock), and 3) cycles between code fragments that cannot possibly execute in parallel due to the causality relation defined by thread start–join relationships between threads. Note that, in Java, a thread can start a thread `t` by executing `t.start()`, and it can wait for `t` to terminate by executing `t.join()`. In the section "An example," we present an example that illustrates these situations. Such false positives should probably still be reported since lock graph cycles generally are undesirable, but they can now be graded as having lower severity, an important piece of information in those cases where the lock order violation is intended.

False negatives occur when existing deadlock potentials are missed by the algorithm. The basic algorithm is surprisingly effective but is limited by the fact that it analyzes only one execution at a time. A technique is presented that reduces false negatives by combining information from multiple executions. The main challenge in accomplishing this is to identify a correspondence between lock objects in different executions, because the actual lock objects across executions are different.

Runtime overhead is caused by the instrumentation needed to intercept operations on locks and other synchronization operations (e.g., start and join operations on threads) and to either run the analysis algorithm (online analysis) or record information about the operation for subsequent analysis (offline analysis). Static analysis can be used to decrease the runtime overhead. A type system that ensures the absence of races and atomicity violations is extended with the deadlock types of Boyapati et al. [5], which keep track of the locking order and can show that parts of a program are deadlock free. We provide an algorithm that infers deadlock types for a given program and an algorithm that on the basis of the result of type inference determines which lock operations can safely be ignored (i.e., neither intercepted nor analyzed) by the dynamic analysis.

The Visual Threads tool [3] is one of the earliest practical systems for dynamic deadlock detection by analysis of locking orders. The Visual Threads algorithm constructs and analyzes lock graphs, as briefly described above. Our GoodLock algorithm [6] improved upon the Visual Threads algorithm by introducing the concept of gate locks to reduce false positives. The GoodLock algorithm was based on a different data structure, namely lock trees, which also capture locking order but, unlike lock graphs, never contain cycles. However, the GoodLock algorithm only detected deadlock potentials between pairs of threads. In contrast, the algorithm presented below is based on lock graphs, as is the algorithm in [3], and hence can detect deadlock potentials involving any number of threads while still handling gate locks, and in addition using the causality relation defined by start and join operations between threads to further reduce false positives.

In other work, we have approached the problem of false positives by developing techniques for checking whether a deadlock potential can actually lead to a deadlock. For example, in [6], a model checker is used to explore warnings of data race and deadlock potentials produced by dynamic analysis. The method in [7] generates a scheduler (from deadlock potentials) that attempts to drive the application into a deadlock. Finally, the work in [8] involves the informing of a scheduling "noise maker" of deadlock potentials. The noise maker inserts code that influences the scheduler, avoiding the need for a special scheduler.

The false-positive-reducing algorithm presented below first appeared in [9]; a similar algorithm appeared in [10]. The static analysis component was introduced in [10]. The main contributions of this paper are clearer descriptions of the analysis techniques originally described in [9, 10], new experimental results for the static analysis technique, the first published description of the multitrace analysis (which was presented at PADTAD 2005 [11], a workshop on "Parallel and Distributed Systems: Testing, Analysis, and Debugging," but not described in a paper), and new experimental results for it.

This paper is organized as follows: First, we present the dynamic analysis algorithm for reducing false positives, focusing on the graph data structure and how it is produced from a single execution trace and analyzed. Second, we outline how a graph structure can be built from multiple execution traces. Third, we describe how static analysis is

used to reduce runtime monitoring overhead. Fourth, we discuss interactions between the techniques. Finally, we discuss implementation issues and experimental results.

Reducing false positives

An example

We use an example to illustrate the three categories of false positives that are reported by the basic algorithm but not by the improved algorithm. The first category, *single-threaded cycles*, refers to cycles that are created by a single thread. *Guarded cycles* refer to cycles that are guarded by a gate lock acquired by involved threads "above" the cycle. Finally, *thread segmented cycles* refer to cycles—between thread segments separated by start–join relations—that consequently cannot execute in parallel. The following program illustrates these three situations and a true positive.

```
  Main thread:
01 : new T1().start();
02 : new T2().start();

  Thread T1:
03 : synchronized(G) {
04 :   synchronized(L1) {
05 :     synchronized(L2) {}
06 :   }
07 : };
08 : t3 = new T3();
09 : t3.start();
10 : t3.join();
11 : synchronized(L2) {
12 :   synchronized(L1) {}
13 : }

  Thread T2:
14 : synchronized(G) {
15 :   synchronized(L2) {
16 :     synchronized(L1) {}
17 :   }
18 : }

  Thread T3:
19 : synchronized(L1) {
20 :   synchronized(L2) {}
21 : }
```

The main thread starts the two threads T_1 and T_2 that subsequently run in parallel. Thread T_1 acquires the locks G, L_1, and L_2 in a nested manner (using the synchronized block construct of Java) and releases these again (when exiting the corresponding synchronization blocks). The Java statement `synchronized(L){S}` acquires the lock `L`, executes `S`, and then releases `L`. Then, T_1 starts thread T_3 (which now runs in parallel with thread T_2), waits for its termination, and then, upon the termination of T_3, acquires (and releases) locks L_2 and L_1 in a nested manner. Hence, threads T_1 and T_3 do not execute code in parallel. Threads T_2 and T_3 acquire and release locks and terminate.

The actual deadlock potential (true positive) exists between threads T_2 and T_3, corresponding to a cyclic access pattern on locks L_1 and L_2 in lines 15–16 and 19–20: The two threads take the two locks in opposite order. This can lead to the situation where thread T_1 acquires lock L_2, thread T_3 acquires lock L_1, and now none of the two threads can acquire the second lock. The three false positives are as follows: The single-threaded cycle within thread T_1 on locks L_1 and L_2 in lines 04–05 and 11–12 clearly does not represent a deadlock (since the two code sections cannot execute in parallel). The cycle between threads T_1 and T_2 on locks L_1 and L_2 in lines 04–05 and 15–16 cannot lead to a deadlock because both threads acquire the lock G first (lines 03 and 14). G is referred to as a *gate lock*. Finally, the cycle between threads T_1 and T_3 on locks L_1 and L_2 in lines 11–12 and 19–20 cannot lead to a deadlock, because T_3 terminates before T_1 executes lines 11–12. Such a cycle is referred to as a *thread segmented cycle*.

When analyzing a program for deadlock potentials, we are interested in observing all lock acquisitions and releases, and all thread starts and joins. That is, in addition to the acquire and release events $acquire(t, l)$ and $release(t, l)$ for thread t and lock l, the trace also contains events for thread start, i.e., $start(t_1, t_2)$, and thread join, i.e., $join(t_1, t_2)$, meaning, respectively, that t_1 starts or joins t_2. The program can be instrumented to produce a trace (finite sequence) $\sigma = e_1, e_2, \ldots, e_n$ of such events. Here, n is the number of events in the trace or program run. Let T_σ and L_σ denote the sets of threads and locks, respectively, that occur in σ. Java allows recursive acquisitions of locks by a thread: A thread can acquire a lock and then reacquire it again without having released it in between. However, we assume, for convenience, that the trace is reentrant free in the sense that a lock is never recursively acquired by a thread. This is ensured by deleting any recursive acquisitions and the matching releases before analysis. Alternatively, the code can be instrumented so that it does not emit recursive acquisitions by counting the number of acquisitions without matching releases. For the purpose of illustration, we assume a nondeadlocking execution trace σ for this program. It does not matter which trace is used since all nondeadlocking traces will reveal all four cycles in the program using the basic algorithm.

Basic cycle detection algorithm

The basic algorithm sketched in [3] works as follows: The multithreaded program under observation is executed, whereas only *acquire* and *release* events are observed.

```
Input: An execution trace σ
C_L : [T_σ → P(L_σ)] = [ ]; lock context
G_L : P(L_σ × L_σ) = ∅; lock graph
for (i = 1 .. |σ|) do
   case σ[i] of
     acquire(t, l) →
       G_L := G_L ∪ {(l', l) | l' ∈ C_L(t)};
       C_L := C_L † [t → C_L(t) ∪ {l}];
     release(t, l) →
       C_L := C_L † [t → C_L(t) - {l}]
end;
for each c in cycles(G_L) do
   print ("deadlock potential:", c);
```

Figure 1

Basic lock graph algorithm.

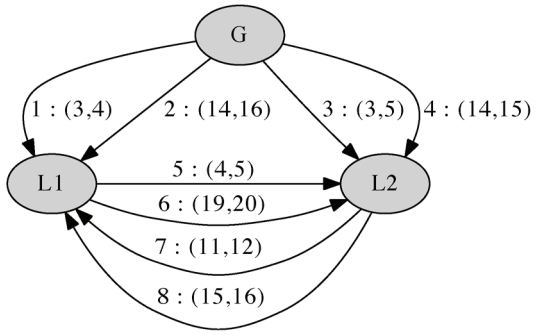

Figure 2

Basic lock graph (adapted from [9], with permission).

A graph is built, where nodes are locks and where directed edges between nodes symbolize locking orders. That is, an edge goes from a lock l_1 to a lock l_2 if a thread at some point during the execution of the program already holds lock l_1 while acquiring lock l_2. Any cycle in the graph signifies potential for a deadlock. **Figure 1** shows the algorithm that constructs the lock graph G_L, and subsequently the set of cycles, denoted by $cycles(G_L)$, representing the potential deadlock situations in the program. The following operators are used to define the algorithms in this paper: Consider two sets A and B. The term $[A \rightarrow B]$ denotes the set of finite maps from A to B. The term [] is the empty map. For a given map M, the term $M \dagger [e_1 \rightarrow e_2]$ denotes the map M', which is equal to M, except that e_1 maps to e_2. That is, $M'(e_1) = e_2$. The term $A \times B$ denotes the Cartesian product of A and B, containing all pairs of the form (a, b), where $a \in A$ and $b \in B$. The term $P(A)$ (power set) denotes the set of subsets of $A : \{s | s \subseteq A\}$). The usual operators are defined for sets, such as the empty set ϕ, set union $A \cup B$, and set comprehension $\{e|p\}$ for an expression e and a predicate p.

The *lock graph* is computed (**Figure 2**) and stored in the second variable in Figure 1 as a directed graph $G_L : P(L_\sigma \times L_\sigma)$. G_L is the minimal graph such that $(l_1, l_2) \in G_L$ if at some point in the trace a thread acquires the lock l_2 while already holding the lock l_1. This lock graph is computed using a *lock context*, the first variable, defined as a mapping $C_L : [T_\sigma \rightarrow P(L_\sigma)]$, from thread IDs to sets of locks. That is, during the trace traversal, a thread ID is mapped to the set of locks held by the thread at that position in the trace. Each lock acquisition event $acquire(t, l)$ (thread t acquires lock l) results in the lock graph G_L to be augmented by an edge from a lock l' to l if thread t already holds l' according to the lock context C_L. Furthermore, the lock context C_L is updated adding l to the set of locks held by t. Each lock release event $release(t, l)$ (thread t releases lock l) results in the lock context C_L being updated by removing l from the set of locks held by t. The lock graph for the code example in the section "An example" is shown in Figure 2. The graph consists of the nodes G, L_1, and L_2, corresponding to the locks in the program, as well as edges between the nodes. The edges are numbered from 1 to 8 for reference. In addition, a pair of numbers (x, y) is associated with each edge. The two numbers indicate in what source code lines the locks were acquired (x for the first, and y for the second). For example, there are two edges from L_1 and L_2, one edge representing the fact that a thread acquired L_1 in line 4 and then L_2 in line 5, and one edge representing the fact that a thread acquired L_1 in line 19 and L_2 in line 20. The graph exposes four cycles corresponding to the four possible deadlock potentials described in the section "An example." That is, the true positive is represented by the cycle consisting of the two edges between L_1 and L_2 numbered 6 and 8. The three false positives are represented by the three cycles 5–7, 5–8, and 6–7, respectively.

Extended cycle detection algorithm

The new algorithm will filter out false positives stemming from *single-threaded cycles*, *guarded cycles*, and *thread segmented cycles*. The extension consists in all three cases of labeling edges with additional information and using this information to filter out false positives. Single-threaded cycles are detected by labeling each edge between two locks with the ID of the thread that acquired both locks. For a cycle to be valid, and hence regarded as a true positive, the threads in the cycle must all differ. Guarded cycles are detected by further labeling each edge between locks with the set of locks held by that thread when the target

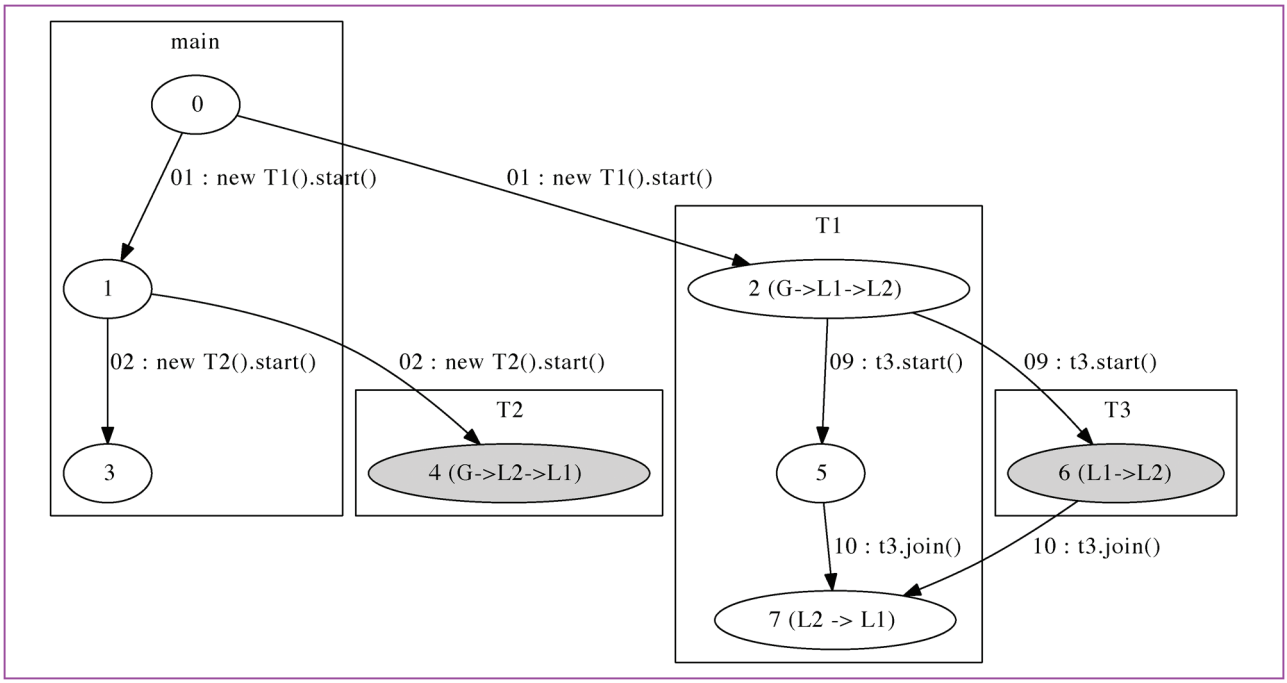

Figure 3

Segmentation graph (adapted from [9], with permission).

(second) lock was acquired. This set is referred to as the *guard set*. For a cycle to be valid, and hence regarded as a true positive, the guard sets in the cycle must have an empty intersection.

Concerning *thread segmented cycles*, the solution requires some additional data structures. Assume that traces now also contain start and join events. A new directed segmentation graph records which code segments execute before others. The lock graph is extended with extra label information that specifies in which segments locks are acquired, and the definition of validity of a cycle is extended to incorporate a check that the lock acquisitions occur in segments that can execute in parallel (required for a deadlock to occur). The idea of using segmentation in runtime analysis was initially suggested in [3] to reduce the number of false positives in data race analysis using the Eraser algorithm [12].

More specifically, during execution, the solution is to associate segment identifiers (natural numbers, starting from 0) with segments of the code that are separated by statements that *start* or *join* other threads. **Figure 3** illustrates the segmentation graph for the example program above. For illustrative purposes, it is augmented with 1) the statements (and their line numbers) that cause the graph to be updated, 2) information inside relevant segments about the order in which locks are taken in the segment, and 3) shading of the two segments that together cause a deadlock potential. It should be interpreted as follows: When a thread t_1 (executing in some segment) starts another thread t_2, two new segments are allocated: one for t_1 to continue in, and one for t_2 to start executing in. The execution order between the segments is recorded as directed edges in the graph: the original segment of t_1 executes before both of the two new segments. Similarly, when a thread t_1 joins another thread t_2 (after waiting for its termination), a new segment is allocated for t_1 to continue in. Again, the execution order of the previous segments of t_1 and t_2 relative to the new segment is recorded: they both execute before this new segment. For example, we see that segment 6 of thread T_3 executes before segment 7 of thread T_1. Segment 6 is the one in which T_3 executes lines 19 and 20, whereas segment 7 is the one in which T_1 executes lines 11 and 12.

Let $R : P(N \times N)$ (N stands for the natural numbers) be such a segmentation graph. The *happens-before* relation $\_ \Rightarrow \_ : P(N \times N)$ is the transitive closure R^* of R. That is, given two segments s_1 and s_2, we say that s_1 *happens before* s_2 if $s_1 \Rightarrow s_2$. Note that for two given segments s_1 and s_2, if neither $s_1 \Rightarrow s_2$ nor $s_2 \Rightarrow s_1$, then we say that s_1 *happens in parallel* with s_2.

Figure 4 presents the algorithm for constructing the segmentation graph and lock graph from an execution trace. The algorithm declares five variables: 1) a segmentation counter n; 2) a segmentation context C_S; 3) a lock context

```
Input: An execution trace σ
n : N = 1; next available segment;
C_S : [T_σ → N] = [ ]; segmentation context
C_L : [T_σ →P(L_σ × N)] = [ ]; lock context
G_S : P(N × N) = ∅; segmentation graph
G_L : P(L_σ × (N × T_σ × P(L_σ) × N) × L_σ) = ∅; lock graph
for (i = 1 .. |σ|) do
        case σ[i] of
         acquire(t,l) →
                G_L := G_L ∪ {(l',(s1,(t,g),s2), l) |
                                (l', s1) ∈ C_L(t) ∧
                                g = {l'' | (l'',s) ∈ C_L(t)} ∧
                                s2 = C_S(t)};
                C_L := C_L † [t → C_L(t) ∪ {(l,C_S(t))}];
         release(t,l) →
                C_L := C_L † [t → C_L(t) - {(l,∗)}];
         start(t1,t2) →
                G_S := G_S ∪ {(C_S(t1),n), (C_S(t1),n+1)};
                C_S := C_S † [t1 → n, t2 → n+1];
                n := n + 2;
         join(t1,t2) →
                G_S := G_S ∪ {(C_S(t1),n), (C_S(t2),n)};
                C_S := C_S † [t1 → n];
                n := n + 1;
end;
for each c in cycles_s(G_L) do
        print ("deadlock potential:",c);
```

Figure 4

Extended lock graph algorithm.

C_L; 4) a segmentation graph G_S; and 5) a lock graph G_L. The segmentation context $C_S : [T_\sigma \rightarrow N]$ maps each thread to the segment in which it is currently executing. The segmentation counter n represents the next available segment. The lock context $C_L : [T_\sigma \rightarrow P(L_\sigma \times N)]$ maps each thread to a set of (*lock, segment*) pairs. In the basic algorithm, it was a mapping from each thread to the set of locks held by that thread at any point during the trace traversal. Now, we add as information the segment in which each lock was acquired. The segmentation graph $G_S : P(N \times N)$ is the set of tuples (s_1, s_2) representing the fact that segment s_1 executes before segment s_2. The *happens-before* relation $\_ \Rightarrow \_$ is the transitive closure of G_S. Finally, the lock graph $G_L : P(L_\sigma \times (N \times T_\sigma \times P(L_\sigma) \times N) \times L_\sigma)$ defines the set of tuples $(l_1, (s_1, t, g, s_2), l_2)$, representing an edge from the lock l_1 to the lock l_2 labeled (s_1, t, g, s_2), and representing the fact that thread t acquired the lock l_2, while holding all the locks in the set g, including the lock l_1. In addition, the edge is labeled with the segments s_1 and s_2 in which the locks l_1 and l_2 were acquired by t.

The body of the algorithm works as follows: Each lock acquisition event $acquire(t, l)$ results in the lock graph G_L being augmented by an edge from every lock l' that t already holds to l, each such edge labeled (s_1, t, g, s_2). The label is to be interpreted as follows: Thread t already holds all the locks in the lock set g, including l', according to the lock context C_L; l' was acquired in segment s_1 according to C_L; and l is acquired in segment s_2 according to C_S. Furthermore, the lock context C_L is updated adding (l, s_2) to the set of locks held by t. Each lock release event $release(t, l)$ (thread t releases lock l) results in the lock context C_L being updated by removing l from the set of locks held by t. A start event $start(t_1, t_2)$, representing that thread t_1 starts thread t_2, "allocates" two new segments n (for t_1 to continue in) and $n + 1$ (for the new t_2) and updates the segmentation graph to record that the current segment of t_1 executes before n and before $n + 1$. The segmentation context C_S is updated to reflect in what segments t_1 and t_2 continue to execute in. A join event $join(t_1, t_2)$, representing that t_1 waits for and joins the termination of t_2, causes the segmentation graph G_S to record that t_1 starts in a new segment n and that t_1's previous segment and t_2's final segment execute before that.

For a cycle to be valid, and hence regarded as a true positive, the threads and guard sets occurring in labels of the cycle must be valid as explained earlier (threads must differ and guard sets must not overlap). In addition, the segments in which locks are acquired must allow for a deadlock to actually happen. For example, consider a cycle between two threads t_1 and t_2 on two locks l_1 and l_2. Assume further that t_1 acquires l_1 in segment x_1 and then l_2 in segment x_2, whereas t_2 acquires them in the opposite order, in segments y_1 and y_2, respectively. Then, it must be possible for t_1 and t_2 to each acquire its first lock before the other attempts to acquire its second lock for a deadlock to occur. In other words, it should not be the case that either $x_2 \Rightarrow y_1$ or $y_2 \Rightarrow x_1$.

The cycle validity checks mentioned above can be formalized as follows: Let there be defined four functions *thread*, *guards*, seg_1, and seg_2 on edges such that, for any edge $\varepsilon = (l_1, (s_1, t, g, s_2), l_2)$ in the lock graph, $thread(\varepsilon) = t$, $guards(\varepsilon) = g$, $seg_1(\varepsilon) = s_1$, and $seg_2(\varepsilon) = s_2$. Then, for any two edges ε_1 and ε_2 in the cycle, 1) the threads must differ, i.e., $thread(\varepsilon_1) \neq thread(\varepsilon_2)$; 2) guard sets must not overlap, i.e., $guards(\varepsilon_1) \cap guards(\varepsilon_2) = \phi$; and 3) segments must not be ordered, i.e., $\neg(seg_2(\varepsilon_1) \rightarrow seg_1(\varepsilon_2))$.

Let us illustrate the algorithm with our example. The segmented and guarded lock graph and the segmentation graph are shown in **Figures 5** and 3, respectively. The lock graph contains the same number of edges as the basic graph in Figure 2, although now labeled with additional information. As an example, edge 5 from lock L_1 to L_2 annotated with line numbers 4 and 5 is now additionally labeled with the tuple: "$\langle 2, (T_1, \{G, L_1\}), 2\rangle$." The interpretation is as follows: during program execution, thread T_1 acquired lock L_1 in line 4, in code segment 2

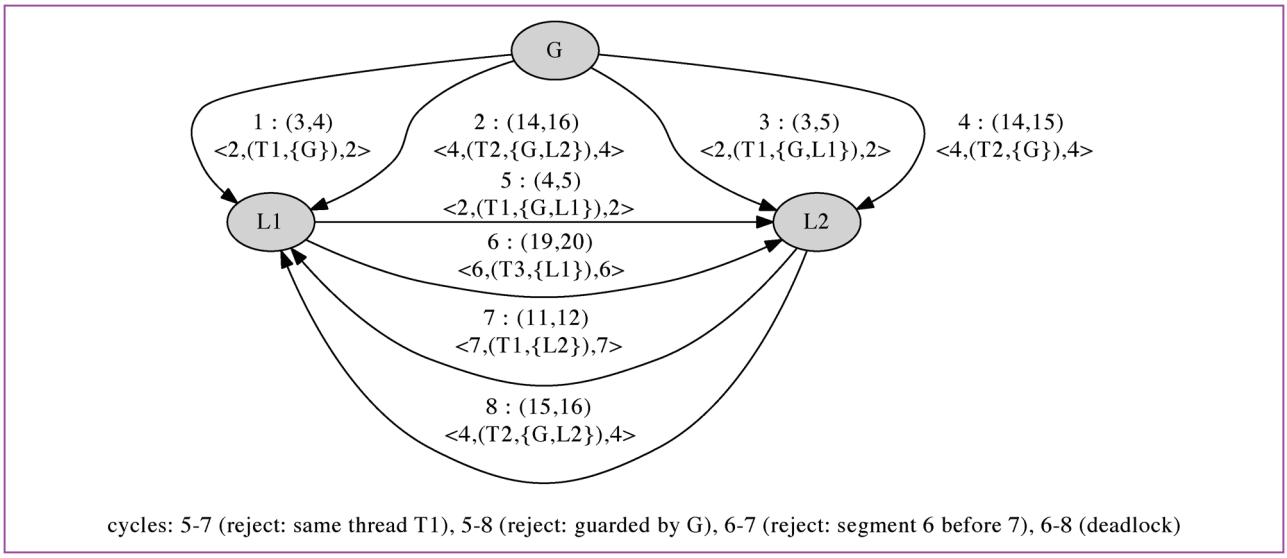

Figure 5

Extended lock graph (adapted from [9], with permission).

(leftmost 2), and subsequently, in a nested manner, acquired lock L_2 in line 5, in code segment 2 (rightmost 2). Furthermore, when thread T_1 acquired lock L_2, T_1 already held the locks in the set $\{G, L_1\}$, the interesting of these being the lock G.

The segmentation graph illustrates the code segments of which each thread consists. The *main* thread executes in segment 0 until it starts thread T_1 in line 01. After that, the *main* thread continues in segment 1, and the newly started thread T_1 executes in segment 2. In segment 1, the *main* thread furthermore starts thread T_2 in line 02 and continues in segment 3 (where it terminates). Thread T_2 starts executing in segment 4. Thread T_1, in turn, while executing in segment 2, starts thread T_3 in line 09 and continues in segment 5, whereas the newly started thread T_3 executes in segment 6. In segment 5, thread T_1 waits for thread T_3 to terminate. When this happens (T_1 successfully executes a join operation on T_3 in line 10), T_1 continues in segment 7. The segmentation graph describes what segments execute before others. For example, segment 6 executes before segment 7.

False positives are now eliminated as follows: First, the cycle with edges numbered 5 and 7 with labels, respectively, "2, $(T_1, \{G, L_1\})$, 2" and "7, $(T_1, \{L_2\})$, 7" is eliminated because it is not thread valid: the same thread T_1 occurs on both edges (single-threaded cycle). Second, the cycle 5–8 with labels "2, $(T_1, \{G, L_1\})$, 2" and "4, $(T_2, \{G, L_2\})$, 4" is eliminated because of the lock G being member of both lock sets $\{G, L_1\}$ and $\{G, L_2\}$ (guarded cycle). Finally, the cycle 6–7 with labels "7, $(T_1, \{L_2\})$, 7" and "6, $(T_3, \{L_1\})$, 6" is eliminated since the target segment 6 of the edge numbered 6 executes before the source segment 7 of the edge numbered 7 (thread segmented cycle).

The algorithm requires that threads have access to a shared memory space where the segment counter and segmentation graph are stored. In our case studies, threads run in the same memory space. If this is not the case, some form of communication between threads and a monitor thread maintaining the segment counter and graph is required. Note that multicore machines typically provide shared memory for the different CPUs. In the case where start–join operations occur in loops, the algorithm still works, but the segmentation graph now linearly grows with the number of start–join calls, causing additional resource consumption by the analysis.

Reducing false negatives

Motivation

It may happen that a test suite runs all of the code segments involved in a deadlock potential, but no single test runs all of those code segments [assuming that each test is executed in a different run of the Java virtual machine (JVM**) and generates a separate trace]. This is particularly likely with the advent of test-driven development techniques, which promote writing small tests, e.g., tests for a single method. In such scenarios, if the dynamic analysis algorithm processes the trace from each test independently, it will miss some deadlock potentials.

The main challenge in adapting the algorithm to process multiple traces together is that the algorithm is based on

```
for each trace T do
        LG′ = [ ];
        for each acquire operation acquire(cl,t,l) in T do
                LG′[l] := LG′ [l] ∪ {cl};
                LG[cl] := LG[cl] ∪ LG′ [l];
                for each location cl′ in LG′ [l] do
                        LG[cl′] := LG[cl];
                end;
        end;
end;
```

Figure 6

Lock grouping algorithm.

lock identity. In Java, lock identity is the reference to the lock object, which is typically the address of the object in memory and has no meaning outside the scope of a single JVM run.

Our approach to identification of locks across runs is based on the observation that locks used in the same code location are likely to be the same lock. This is a heuristic motivated by taking the perspective of a programmer. When a programmer formulates a lock discipline policy for the program—in particular, the order in which nested locks should be acquired—it is normally expressed in terms of the name of the lock variables, with possible aliasing; this is usually equivalent to the set of locations where the lock is used.

Algorithm

We enhance the traces described in the section "An example" with information about the code location of lock acquisition and release events, i.e., $acquire(cl, t, l)$ and $release(cl, t, l)$, where cl is the code location. We make a first pass in which the code locations of lock operations are grouped into equivalence classes called *lock groups*.

Definition 1 (lock group)

Code locations cl_1 and cl_2 of lock acquire operations are equivalent if the same lock is acquired at both locations in some trace, i.e., if there exist a trace σ and a lock object l such that σ contains the entries $acquire(cl_1, t_1, l)$ and $acquire(cl_2, t_2, l)$ for some threads t_1 and t_2. Each equivalence class of this equivalence relation is called a *lock group*.

The algorithm in **Figure 6** takes a set of traces as input and computes a function LG that maps each lock acquire operation (identified by its code location) to its lock group. The algorithm also computes an auxiliary function LG' that maps each lock object to a lock group.

A second pass is made over the traces, running the algorithm from the previous section, except that nodes in the lock graph now represent lock groups rather than lock objects; that is, the node affected by an acquire operation $acquire(cl, t, l)$ is $LG[cl]$. Gate locks are handled as before, except that a gate lock is now represented by a lock group. The identities of the lock objects (the third parameter of acquire and release events in the traces) are ignored in this pass. A single lock graph is created from the entire set of traces.

Example

Consider the following number utility consisting of the two classes, given to clients as thread safe:

```
public class MyFloat {
  private float value;
  private final Object lock = new Object();

  public MyFloat(float initValue) {
    value = initValue;
  }

  public float get() {
CL1: synchronized(lock) {
      return value;
    }
  }

  public void addInt(MyInt anInt) {
CL2: synchronized(lock) {
      value+ = anInt.get();
    }
  }
}

public class MyInt {
  private int value;
  private final Object lock = new Object();

  public MyInt(int initValue) {
    value = initValue;
  }

  public int get() {
CL3: synchronized(lock) {
      return value;
    }
  }

  public void setRound(MyFloat aFloat) {
CL4: synchronized(lock) {
      value = (int)aFloat.get();
    }
  }
}
```

This utility has a deadlock potential, which can be seen in the following usage example:

Main:
```
MyInt i1 = new MyInt(5);
MyFloat f1 = new MyFloat(5.4f);
```

Thread T1:
```
f1.addInt(i1);
```

Thread T2:
```
i1.setRound(f1);
```

Indeed, the single-trace algorithm will reveal the deadlock potential if run on this test. Consider, however, the following two tests:

```
void testAddition() {
  MyFloat f = new MyFloat(5.4f);
  MyInt i = new MyInt(5);
  f.addInt(i);
  assertEqualsUpTo(10.4f, f.get(), 0.01f);
}

void testRounding() {
  MyFloat f = new MyFloat(5.4f);
  MyInt i = new MyInt(3);
  i.setRound(f);
  assertEquals(5, i.get());
}
```

Here, the single-trace algorithm will not reveal the deadlock potential, regardless of whether the two tests are run in two threads in the same JVM, in the same thread one after another, or in two different JVM invocations, simply because the two conflicting methods are used with different lock objects.

The multitrace algorithm, on the other hand, will reveal the deadlock potential, regardless of whether the two test methods are run in two threads in the same JVM, in the same thread one after another, or in two different JVM invocations. It works as follows: From the trace of `testAddition()`, the algorithm will deduce that locations CL_1 and CL_2 are in the same lock group (e.g., lg_1), because `MyFloat.get()` and `MyFloat.addInt()` were called on the same object (variable f in the test), and hence, the two acquire operations were performed on the same object. Similarly, from the trace of `testRounding()`, the algorithm will deduce that CL_3 and CL_4 are in the same lock group (e.g., lg_2). Now, in the lock graph creation phase, the trace of `testAddition()` will show that lg_2 is acquired when lg_1 is held, and the trace of `testRounding()` will show that lg_1 is acquired when lg_2 is held. A cycle is detected, and a warning is given.

Mixtures

The multitrace algorithm can issue warnings not only for cycles among lock groups but also for another situation, called a *mixture*. A *mixture* warning is given if a thread performs nested acquisitions of two different lock objects in code locations belonging to the same lock group. This is a potential deadlock in cases such as the following:

```
class MySet {
  private final Object lock = new Object();
  ...
  public void addElement(...) {
CL1: synchronized(lock) {
       ...
    }
  }

  public void addAll(MySetother) {
CL2: synchronized(this.lock) {
CL3:   synchronized(other.lock) {
         ...
       }
    }
  }
}
```

The `addAll()` method is analogous to `java.util.Set.addAll()`. The deadlock potential is exemplified by the following multithreaded test:

Main thread:
```
MySet s1, s2;
```

Thread T1:
```
s1.addAll(s2);
```

Thread T2:
```
s2.addAll(s1);
```

While the single-trace algorithm can expose the deadlock potential when run on this test, the multitrace analysis can expose it even on a simple test, which is single threaded and calls `addAll()` only once:

```
void testMySet() {
  MySet s1;
  MySet s2;
  ...
CL4: s1.addElement(...);
CL5: s2.addElement(...);
CL6: s1.addAll(s2);
  ...
}
```

The lock grouping algorithm groups *CL*2 together with *CL*1, because both were done on the same object (`s1` in *CL*4 and *CL*6 of the test); it also groups *CL*3 together with *CL*1, because both were done on `s2` in *CL*5 and *CL*6 of the test; hence, all three lock locations are in the same lock group. In *CL*2 and *CL*3, locks on two different objects are acquired nestedly in code locations in the same lock group; thus, a warning is issued. Incidentally, note that a gate lock is the most straightforward way to synchronize `addAll()` correctly:

```
class MySetFixed {
  private final Object lock = new Object();
  private final static Object gateLock =
  new Object();
  ...
  public void addElement(...) {
    synchronized(lock) {
      ...
    }
  }

  public void addAll(MySet other) {
    synchronized(gateLock) {
      synchronized(this.lock) {
        synchronized(other.lock) {
          ...
        }
      }
    }
  }
}
```

Discussion

The three `test...()` methods above show that the multiple-trace technique is particularly powerful in contexts of different tests on small fragments of the code, which is typical of unit tests. The technique is able to combine information from different tests, revealing the connections between parts of the code exercised by each test, connections that give rise to the deadlock potential.

In some rare cases, a program may choose locking objects dynamically, and not by the lock variable name. This may make the heuristic associating locks by location fail, causing the multiple-trace technique to yield false positives. We know of one valid programming pattern doing this: in the `addAll()` example from the section "Mixtures," instead of using a gate lock, the order of locking can be determined by the runtime identity of the lock objects. This pattern is described in [13]. Eliminating this type of false positives is probably possible with static analysis, identifying the valid pattern, but remains as further work.

The effectiveness of the multiple-trace algorithm is dependent on the details of the tests. In the number utility example, if `testAddition()` called `f.addInt(i)` but not `f.get()`, then the lock grouping phase would not "know" to put CL_1 and CL_2 in the same lock group, and a warning would not be given. The warning depends on `f.get()`, although it is not part of the deadlock. Similarly, in the set utility example, if `testMySet()` did not call `addElement()` on both sets in addition to `addAll()`, the algorithm would not "know" to put CL_2 and CL_3 in the same lock group, and a warning would not be given—the warning depends on `addElement()`, which is not part of the deadlock. On a more positive note, the examples represent natural tests of the given code; thus, it is at least reasonably likely that these deadlock potentials will be revealed by the multitrace algorithm.

Reducing overhead with static analysis

Although the dynamic analysis algorithms in the previous sections are efficient and effective, they can miss deadlock potentials, and their runtime overhead is not negligible. This section describes a type-based static analysis that can prove absence of deadlocks in parts of the code and describes how this information can be used to reduce the overhead of dynamic analysis.

Deadlock-type system

Boyapati et al. [5] define a type system for Java that ensures programs are deadlock free. The types, which we call *deadlock types*, associate a *lock level* with each lock. The typing rules ensure that threads perform nested acquires in descending order by lock level; that is, if a thread acquires a lock l_2 (that the thread does not already hold) while holding a lock l_1, then the level of l_2 level is less than the level of l_1. This ensures absence of cyclic waiting and, therefore, absence of deadlock.

A new lock level l is declared by the statement `LockLevel l = new`. A partial order on lock levels is defined by statements of the form `Locklevel` $l_2 < l_1$. Lock levels are associated only with expressions that denote objects possibly used as locks (i.e., as the target object of a `synchronized` method or the argument of a `synchronized` statement). These expressions are identified using Boyapati and Rinard's Parameterized Race-Free Java type system [14]. We omit details of Parameterized Race-Free Java, since it plays a limited role in the deadlock-type system.

We focus on the *basic* deadlock-type system, in which all instances of a class have the same lock level. Extensions to the basic type system—for example, allowing different instances of a class to have different lock levels and allowing lock orderings to depend on the positions of objects in tree-based data structures—would allow greater focusing of the dynamic analysis in some cases but would also

increase the complexity and running time of the static analysis.

To enable methods to be type checked individually, each method m is annotated with a `locks` clause that contains a set S of lock levels. Method m may acquire locks whose level is equal to or less than a level in S; this restriction is enforced by the typing rule for `synchronized` statements. At each call to m, the caller may hold only locks whose levels are greater than all the levels in S; this restriction is enforced by the typing rule for method invocation statements.

It is common in Java for a method to acquire a lock already held by the caller; this occurs, for example, when a synchronized method calls a synchronized method of the same class on the same object. To allow typing of such methods, the `locks` clause of a method m may also contain a lock l. The typing rules allow m to acquire l and locks with level less than the level of l, and they allow callers of m to hold l and locks with level greater than the level of l.

For example, consider the Number Utility classes from the section "Example." Suppose we try to assign lock level `LF` to all instances of `MyFloat` and assign lock level `LI` to all instances of `MyInt`. The declarations `LockLevel LF = new` and `LockLevel LI = new` would be added in `MyFloat` and `MyInt`, respectively. The declaration of method `addInt` would become `public void addInt(MyInt anInt) locks lock, LI`. The inclusion of `LI` in the `locks` clause reflects the call `anInt.get()`, which acquires a lock of level `LI`. The declarations of other methods would also be extended with `locks` clauses. For type checking of `addInt` to succeed, the declaration `LockLevel MyFloat.LF > MyInt.LI` is needed, because `addInt` holds a lock with level `LF` when it calls `anInt.get()`. With this lock-level ordering, the type checker will report that `MyInt.setRound` is untypable, because it acquires nested locks in increasing order with respect to this ordering. This reflects the potential deadlock in the program.

Type inference algorithm

This section presents a type inference algorithm for the basic deadlock-type system. The algorithm assumes the program is already annotated with Parameterized Race-Free Java types [14] (e.g., by using the type inference algorithm in [15] or [16]) to indicate which fields, methods parameters, and local variables may refer to objects used as locks. The algorithm produces correct deadlock typings (including lock-level declarations, lock-level orderings, and `locks` clauses) for all typable programs. It does not explicitly determine whether the given program is typable: it infers the best deadlock types it can, regardless of whether the program is completely typable. This is useful for optimization of dynamic analysis, as discussed below. A type checker for the deadlock-type system is run after type inference to check the inferred types. The algorithm consists of the following steps.

Step 1—Each field, method parameter, and local variable that may refer to an object used as a lock is initially assigned a distinct lock level. This imposes the fewest constraints on the program. However, some expressions must have the same lock level for the program to be typable. Specifically, equality constraints among lock levels are generated based on the assignment statements and method invocations in the program: the two expressions in an assignment statement must have the same lock level (just as they must have the same type), and each argument in a method call must have the same lock level as the corresponding formal parameter of the method. These equality constraints are processed using the standard union-find algorithm. All lock levels that end up in the same set are replaced with a single lock level.
Step 2—This step constructs a static lock graph that captures the locking pattern of the program. The graph contains a *lock node* corresponding to each `synchronized` statement in the program (including the implicit `synchronized` statements enclosing the bodies of `synchronized` methods), a *method node* corresponding to each method m, and a *call node* corresponding to each method call statement. For a call node n, let $called(n)$ be the set of method nodes corresponding to methods possibly called by n.

The graph contains edges that represent possible intra- and interprocedural nesting of lock acquire operations. There is an edge from a lock node n_1 to a lock node or call node n_2 if the statement corresponding to n_2 is syntactically nested within the `synchronized` statement corresponding to n_1, and there are no `synchronized` statements between them. There is an edge from a method node n_m to the lock node for each outermost `synchronized` statement in the body of m. There is an edge from each call node n to each method node in $called(n)$.

We enhance this step to ignore `synchronized` statements that acquire a lock already held by the same thread. We call such `synchronized` statements *redundant*. We conservatively identify them using simple syntactic checks and information from race-free types [10].
Step 3—This step computes `locks` clauses. To do this, it associates a set L_n of lock levels with each node n. These sets are the least solution to the following recursive equations. The solution is obtained from a standard fixed-point computation. For a lock node n, L_n is a singleton set containing the level of the lock acquired by n, as determined in Step 1. For a call node n, $L_n = \bigcup_{n' \in called(n)} L_{n'}$. For a method node n, $L_n = \bigcup_{n' \in succ(n)} L_{n'}$, where $succ(n)$ is the set of successor

nodes of n. For each method m, the lock levels in L_n are included in the `locks` clause of m, where n is the method node corresponding to m.

Next, for each method m, the algorithm determines whether to include a lock in the `locks` clause of m. If m contains a `synchronized` statement that acquires a lock e with level l, and if m may be called with a lock of the same level as e already held (namely, if n_m is reachable in the static lock graph from a lock node with level l in another method), then the algorithm includes e in the `locks` clause of m, because this is the only possibility for making the program typable (if the typing rules can verify that the caller acquires the same lock e, the program is typable with this typing; otherwise, the program is not typable).

Step 4—This step computes orderings among lock levels. For each edge from a lock node n to a lock node or call node n', for each lock level l in L_n and each lock level l' in $L_{n'}$, add the declaration `LockLevel` $l > l'$.

The running time of the type inference algorithm is typically linear in the size of the program; intuitively, this is because the analysis basically labels method declarations and method calls with information about locks held when the method is called, and programs typically hold a small number (not proportional to the program size) of locks at each point.

Focused dynamic analysis

Deadlock types enforce a conservative strategy for preventing deadlocks. Therefore, some deadlock-free programs are not typable in this type system. For example, the type system assigns the same lock level to all objects stored in a collection; thus, a program that performs nested acquisitions on objects from a collection is not typable, although it may be deadlock free. Deadlock types can be used to optimize dynamic analysis of deadlock potentials in programs that are not (completely) typable, by eliminating dynamic checks for parts of the program guaranteed to be deadlock free by the type system. In other words, the dynamic analysis is focused on parts of the program that might have deadlocks.

Focusing of dynamic analysis is achieved as follows: First, we find all lock levels that are in cycles in the inferred lock-level ordering. Second, when instrumenting the program for dynamic analysis, we instrument only `synchronized` statements such that the lock level of the acquired lock is part of a cycle in the lock-level graph. Other `synchronized` statements cannot be involved in deadlock potentials.

Interactions between the techniques

The three refinements from the section "Reducing false positives" can be used in the multitrace analysis to reduce false positives. While the guarded-cycles refinement combines well with the multiple-trace technique, the other two refinements are somewhat contradictory with respect to the reasoning behind it. In the number utility example, the warning is justified by the presumption that `MyFloat.addInt()` and `MyInt.round()` may be invoked by two threads in parallel, although the testing may have run the two respective test cases (`testAddition()` and `testRounding()`) in different JVM runs. We may just as well see them in one thread of one JVM run or in two threads segmented by start–join; this is regarded as just a matter of how the test framework is configured. Thus, to gain maximum benefit from the multitrace analysis, warnings corresponding to single-threaded cycles and segmented cycles should usually be reported.

Without the guarded-cycles refinement, the warnings given by the multitrace analysis (the union of *cycle* and *mixture* warnings) are a superset of those given by the single-trace analysis. This follows from the fact that two lock operations performed on the same object in a given trace are classified by the lock grouping algorithm as being in the same lock group.

With the guarded-cycles refinement, the multitrace analysis might omit some warnings given by the single-trace analysis. For example, suppose `MySetFixed` contained the bug that `gateLock` were an instance member, rather than a static one. In this case, it does not prevent the deadlock, because `gateLock` is different between the two threads that use conflicting lock order. Indeed, the single-trace analysis would not regard this cycle as guarded and will give a warning. However, the multitrace analysis will consider the acquire on `gateLock` to be a valid guard for the cycle, since it is done in the same code location and, hence, on the same lock group.

While in a trace of the multithreaded test it is easy to see that the cycle is not guarded, in a trace of the single-threaded test, the wrong implementation (instance member) is indistinguishable from the correct implementation (static member), because only one gate lock is created and used in the trace, regardless of which implementation is used. This suggests that guarded cycles should be given a higher severity level in multitrace analysis than they are given in single-trace analysis.

Static analysis and focused checking are largely orthogonal to the techniques for reducing false positives and false negatives, except that focused checking may reduce the effectiveness of the gate lock technique, by eliminating instrumentation of locks that might act as gate locks.

Implementation and experiments

Implementation

The methods described in this paper have been implemented in two separate tools, applied to different case studies. The single-trace analysis is implemented as described in [9], as part of the Java PathExplorer (JPaX) system [17]. Focused

dynamic analysis is incorporated in this tool, although the deadlock-type inference algorithm is not implemented and is manually applied. The multitrace analysis is implemented as part of the IBM Concurrent Testing (ConTest) tool [18]. Ideally, the techniques would all be integrated in one system and evaluated on the same case studies (preferably "real-life" applications); this is a direction for future work. The integrated system would consist of four main modules. The *static analysis module* analyzes the program and produces information about which code locations need runtime monitoring. The *instrumentation module* automatically instruments these locations by inserting instructions that, during program execution, invoke methods of the observer module to inform it about synchronization events. The *observer module* generates traces and passes them to the *trace analysis module*, which analyzes the traces using the algorithms described in previous sections. Other testing and performance tools can be integrated into this architecture, for example, trace analysis modules that detect data races [12, 19] and atomicity violations [20–22]. Integration of code coverage measurement tools is useful for identifying synchronization statements that were not exercised by a test suite. Exercising all of the synchronization statements helps reduce false negatives in the dynamic analysis.

Experiments

Dynamic analysis of single traces

The single-trace algorithm has been applied to three National Aeronautics and Space Administration (NASA) case studies: a planetary rover controller for a rover named K9, programmed in 35,000 lines of C++; a 7,500 line Java version of the K9 rover controller used in an evaluation of Java verification tools conducted at NASA Ames Research Center; and a planner named Europa programmed in approximately 10,000 lines of C++. For the case studies in C++, operations in the program were instrumented by hand to update a log file with trace events. The instrumentation was automatically performed for the Java program using bytecode instrumentation. The generated log files were then read and analyzed applying the single-trace algorithm.

The tool found one cyclic deadlock in each of these systems. The deadlocks in the C++ applications were unknown to the programmers. The deadlock in the Java application was seeded as part of a broader experiment to compare analysis tools, as described in [4]. However, in none of these applications was there a need to reduce false positives, and hence, the basic algorithm would have given the same result.

Dynamic analysis of multiple traces

Results of applying the multitrace analysis to a number utility and set utility are described in the sections "Example" and "Mixtures," respectively. The set utility example is similar to the implementation of sets in the standard library of Sun for Java 1.4 (in later versions, the implementation was fixed to prevent this deadlock). The latter is used as a case study in [23], which presents a test case similar to the multithreaded test of MySet in the section "Mixtures," and shows that their analysis framework, like our single-trace algorithm, reveals the deadlock potential from this test case. As described in the section "Mixtures," our multitrace algorithm reveals the deadlock potential even from a simple single-threaded test case, which calls `addAll()` only once. This kind of test is more likely to be written by a programmer, particularly if the programmer is not specifically testing for deadlocks. More generally, one can argue that multiple-trace analysis is particularly useful in unit testing scenarios.

In a larger experiment, the multitrace algorithm was applied at Telefonica, the largest telecommunications company in the Spanish-speaking market, on a Java component consisting of 60,000 lines of code in 246 classes, with 312 synchronization statements. The test suite runs between two to a few dozens of threads concurrently. In the first round of testing, a complex cycle was revealed, resulting from six synchronized methods in different classes all calling each other. This problem would have been found by the basic algorithm as well, with the given test case. Even without analyzing whether a deadlock can actually occur in that code (which would have been a challenging task), the developers acknowledged that this was bad programming and changed the locking policy of the code.

However, a second round of tests revealed that the change was not sufficiently good. Specifically, the multitrace analysis gave a *mixture* warning. This was again acknowledged by the developers as requiring a code change. This problem would not have been detected by the single-trace algorithm.

Reducing overhead with static analysis

The Java version of the K9 rover controller was used to demonstrate the utility of static analysis to reduce overhead of dynamic analysis. The code consists of 72 classes and runs seven threads operating on seven locks. Each lock is acquired multiple times by the different threads via 31 synchronization statements. The type inference algorithm was manually applied to infer deadlock types, including orderings among lock levels. The ordering contained one cycle, involving three classes. In the focused dynamic analysis, lock operations on instances of other classes were not instrumented. A small change to the code was needed for type inference to produce this result: we duplicated the code for one method with a Boolean argument, specialized the copies for the two possible values of that argument, and replaced calls to the original method with calls to the appropriate copy (that argument is a constant at all call sites).

This kind of transformation, which has the effect of making the static analysis partially context sensitive, can be automated.

Numerous runs of the application yielded the same result, namely, that two threads engage in three deadlock potentials. All three are on the same two objects; the deadlock potentials differ in the lines of code at which the locks are acquired.

A total of 80 runs of the program were performed: 40 with unfocused dynamic analysis and 40 with focused dynamic analysis. Focusing reduced the overhead by approximately two thirds; that is, on average, only 32.3% of the synchronizations are monitored by the focused instrumentation. For example, in one run, 44 synchronizations were executed, and only 14 were monitored for deadlock potentials.

Focusing did not have a significant effect on effectiveness of detection, as expected. Without focusing, no deadlock potentials were detected in 3 runs, two deadlock potentials (the same two) were detected in 15 runs, and all three deadlock potentials were detected in 22 runs. With focusing, two deadlocks potentials (again the same two) were detected in 19 runs, and all three deadlock potentials were detected in 21 runs.

In summary, deadlock potentials were detected in 96.25% of the runs (77 out of 80), the reported deadlock potentials all correspond to deadlocks that can occur in some execution of the program (i.e., none are false alarms), and static analysis reduced the cost of the dynamic analysis by approximately two thirds.

Acknowledgments

The work of R. Agarwal, S. D. Stoller, and L. Wang was supported in part by the National Science Foundation under Grants CNS-0509230 and CCF-0613913 and in part by the Office of Naval Research under Grant N00014-07-1-0928. Part of the work of K. Havelund was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1. M. Singhal, "Deadlock detection in distributed systems," *Computer*, vol. 22, no. 11, pp. 37–48, Nov. 1989.
2. E. Knapp, "Deadlock detection in distributed database systems," *ACM Comput. Surv.*, vol. 19, no. 4, pp. 303–328, Dec. 1987.
3. J. Harrow, "Runtime checking of multithreaded applications with Visual Threads," in *Proc. SPIN Model Checking Softw. Verification*, Stanford, CA, Aug. 30–Sep. 1, 2000, pp. 331–342.
4. G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, W. Visser, and R. Washington, "Experimental evaluation of verification and validation tools on Martian rover software," *Formal Methods Syst. Des.*, vol. 25, no. 2/3, pp. 167–198, Sep.–Nov. 2004.
5. C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming: Preventing data races and deadlocks," in *Proc. 17th ACM OOPSLA*, Seattle, WA, Nov. 4–8, 2002, pp. 211–230.
6. K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *Proc. SPIN Model Checking Softw. Verification*, Stanford, CA, Aug. 30–Sep. 1, 2000, pp. 245–264.
7. S. Bensalem, J. C. Fernandez, K. Havelund, and L. Mounier, "Confirmation of deadlock potentials detected by runtime analysis," in *Proc. PADTAD*, Portland, ME, Jul. 17, 2006, pp. 41–50.
8. Y. Nir-Buchbinder, R. Tzoref, and S. Ur, "Deadlocks: From exhibiting to healing," in *Proc. RV*, Budapest, Hungary, Mar. 30, 2008, pp. 104–118.
9. S. Bensalem and K. Havelund, "Dynamic deadlock analysis of multi-threaded programs," in *Proc. PADTAD Track IBM Verification Conf.*, Haifa, Israel, Nov. 13–16, 2005, pp. 208–223.
10. R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *Proc. PADTAD*, Haifa, Israel, Nov. 13–16, 2005, pp. 191–207.
11. E. Farchi, Y. Nir-Buchbinder, and S. Ur, "A cross-run lock discipline checker for Java," presented at the Parallel Distributed Systems: Testing Debugging (PADTAD), Haifa, Israel, Nov. 13–16, 2005. [Online]. Available: http://alphaworks.ibm.com/tech/contest
12. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.
13. B. Goetz, *Java Concurrency in Practice*. Reading, MA: Addison-Wesley, 2006.
14. C. Boyapati and M. C. Rinard, "A parameterized type system for race-free Java programs," in *Proc. 16th ACM Conf. OOPSLA*, Tampa Bay, FL, Oct. 14–18, 2001, pp. 56–69.
15. R. Agarwal and S. D. Stoller, "Type inference for parameterized race-free Java," in *Proc. 5th Int. Conf. Verification, Model Checking Abstr. Interpretation*, Venice, Italy, Jan. 11–13, 2004, pp. 149–160.
16. J. Rose, N. Swamy, and M. Hicks, "Dynamic inference of polymorphic lock types," *Sci. Comput. Program.*, vol. 58, no. 3, pp. 366–383, Dec. 2005.
17. K. Havelund and G. Rosu, "An overview of the runtime verification tool Java PathExplorer," *Formal Methods Syst. Des.*, vol. 24, no. 2, pp. 189–215, Mar. 2004.
18. Y. Nir-Buchbinder and S. Ur, "ConTest listeners: A concurrency-oriented infrastructure for Java test and heal tools," in *Proc. 4th SOQUA*, Dubrovnik, Croatia, Sep. 3/4, 2007, pp. 9–16.
19. E. Bodden and K. Havelund, "RACER: Effective race detection using AspectJ," *IEEE Trans. Softw. Eng.*, Extended version of paper presented at ISSTA'08, vol. 36, no. 4, Jul./Aug. 2010.
20. C. Artho, K. Havelund, and A. Biere, "High-level data races," *Softw. Testing, Verification Reliab. (STVR)*, vol. 13, no. 4, pp. 207–227, Dec. 2003.
21. C. Artho, K. Havelund, and A. Biere, "Using block-local atomicity to detect stale-value concurrency errors," in *Proc. 2nd ATVA*, Taipei, Taiwan, Oct. 31–Nov. 3, 2004, pp. 150–164.
22. L. Wang and S. D. Stoller, "Accurate and efficient runtime detection of atomicity errors in concurrent programs," in *Proc. ACM SIGPLAN PPoPP*, New York, Mar. 29–31, 2006, pp. 137–146.
23. K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Proc. CAV*, Seattle, WA, Aug. 17–20, 2006, pp. 419–423.

Received September 15, 2009; accepted for publication October 15, 2009

Rahul Agarwal *Synopsys, Inc., Mountain View, CA 94043 USA (agarwal@synopsys.com).* Dr. Agarwal received a B.Tech. degree in computer science and engineering from Indian Institute of Technology, Delhi, in 2001, and M.S. and Ph.D. degrees in computer science from State University of New York at Stony Brook in 2003 and 2008, respectively. He is a Senior Research and Development Engineer at Synopsys. Prior to joining Synopsys, he was a postdoctorate fellow at Cadence Research Labs at Berkeley where he worked on software verification. At Synopsys, he currently works on scaling the performance of a chip physical verification tool. His research interests include concurrency, software testing, and verification.

Saddek Bensalem *Verimag and Joseph Fourier University, Grenoble, France (saddek.bensalem@imag.fr).* Dr. Bensalem received M.Sc. and Ph.D. degrees in computer science in 1982 and 1985, respectively, from Institut Polytechnique de Grenoble (INP Grenoble), France. He is a professor at the Joseph Fourier University. He has been a member of more than 50 conference and workshop program committees and is the author or coauthor of 70 refereed research publications. He has spent two years at Stanford Research Institute (SRI International) in California and one year at the University of Stanford as Visiting Scientist. He has broad experience with industry, notably through joint projects with partners such as Astrium, the European Space Agency, and GMV. His current research activities include component-based design, modeling, and analysis of real-time systems with a focus on correct-by-construction techniques.

Eitan Farchi *IBM Research Division, Haifa Research Laboratory, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (farchi@il.ibm.com).* Dr. Farchi received B.Sc. and M.Sc. degrees in computer science and mathematics from the Hebrew University in Jerusalem, Israel, in 1987 and 1990, respectively, and a Ph.D. degree in mathematics and computer science from Haifa University, Israel, in 1999. He is the manager of the Software Testing, Verification, and Review Methodologies in IBM Haifa Research Laboratory. Since 1993, he has been working at the IBM Haifa Research Laboratory. He is the author or coauthor of 32 technical papers and 13 patents, and he has received seven IBM awards. Dr. Farchi is an IBM Senior Technical Staff Member and a cochair of PADTAD, a workshop on testing multithreaded applications.

Klaus Havelund *Jet Propulsion Laboratory (JPL), California Institute of Technology, Pasadena, CA 91109 USA (klaus.havelund@jpl.nasa.gov).* Dr. Havelund received a Master of Science degree and a Ph.D. degree, both in computer science, from the University of Copenhagen, Denmark, in 1986 and 1994, respectively. His doctoral work was mainly conducted at École Normale Supérieure, Paris, France. He is a Senior Research Scientist at the Laboratory for Reliable Software (LaRS). He has been a member of more than 60 conference and workshop program committees and is the author or coauthor of 80 refereed research publications and one book. His primary research interests include verification of software and, in particular, runtime verification techniques such as specification-based monitoring and dynamic concurrency analysis. Dr. Havelund cofounded the Runtime Verification conference.

Yarden Nir-Buchbinder *IBM Research Division, Haifa Research Laboratory, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (yarden@il.ibm.com).* Mr. Nir-Buchbinder received a B.Sc. degree in computer science from the Technion in Israel in 2000 and an M.A. degree in philosophy from Haifa University in 2005. He is a Research Staff Member in the Code Optimization and Quality Technologies department at the IBM Haifa Research Laboratory. He joined IBM in 2000, where he has been working on the research of concurrency quality and test coverage technologies. He is the coauthor of 11 technical papers and two patents and has received three IBM awards.

Scott D. Stoller *Computer Science Department, Stony Brook University, Stony Brook, NY 11794 USA (stoller@cs.stonybrook.edu).* Dr. Stoller received a Bachelor's degree in physics, *summa cum laude*, from Princeton University in 1990 and a Ph.D. degree in computer science from Cornell University in 1997. He received a National Science Foundation CAREER Award in 1999 and an ONR Young Investigator Award in 2002. He is a member of the team that won the NASA Turning Goals into Reality Award for Engineering Innovation in 2003. He is the author or coauthor of more than 70 refereed research publications. His primary research interests are analysis, optimization, testing, and verification of software.

Shmuel Ur *IBM Research Division, Haifa Research Laboratory, Haifa University Campus, Mount Carmel, Haifa 31905, Israel (ur@il.ibm.com).* Dr. Ur received B.Sc. and M.Sc. degrees in Computer Science from the Technion, Israel in 1987 and 1990, respectively, and a Ph.D. degree in Algorithm, Combinatorics, and Optimization from Carnegie Mellon University in Pittsburgh in 1994. Subsequently he joined IBM Research, where he is currently a Research Staff Member in the Code Optimization and Quality Technologies department at the Haifa Research Laboratory. He works in the field of software testing and concentrates on coverage and testing of multithreaded programs. He is the technical leader of the area of coverage in IBM. He is the author or coauthor of 50 technical papers and 20 patents and has received five IBM awards. Dr. Ur cofounded PADTAD, a workshop on testing multithreaded applications, founded the Haifa Verification Conference (HVC), and is an IBM Master Inventor.

Liqiang Wang *Department of Computer Science, University of Wyoming, Laramie, WY 82071 USA (wang@cs.uwyo.edu).* Dr. Wang received a B.S. degree in mathematics from Hebei Normal University of China in 1995, an M.S. degree in computer science from Sichuan University of China in 1998, and M.S. and Ph.D. degrees in computer science from Stony Brook University in 2003 and 2006, respectively. He subsequently joined the University of Wyoming as an assistant professor. His primary research interest is the design and analysis of parallel systems, particularly on emerging platforms including multicore CPUs and GPUs (graphics processing units).