Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables [*]

Rahul Agarwal Scott D. Stoller

ABSTRACT

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose. A common kind of concurrency error is deadlock, which occurs when some threads are permanently blocked. This paper defines a run-time notion of potential deadlock in programs with locks, semaphores, and condition variables. Informally, an execution has potential for a deadlock if some feasible permutation of the execution results in a deadlock. Feasibility of a permutation is determined by ordering constraints amongst events in the execution. Previous work on run-time detection of potential deadlocks are for programs that use locks. This paper presents run-time algorithms to detect potential deadlocks in programs that use locks (block structured as well as non block structured), semaphores, and condition variables.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software/Program Verification**]: Reliability; D.2.5 [**Testing and Debugging**]: Debugging Aids,Testing tools

Keywords

Deadlocks, Concurrent Programs, Testing, Reliability

General Terms

Reliability, Algorithms

1. INTRODUCTION

Multithreaded programs are becoming increasingly common. Such programs are notorious for containing errors that are difficult to reproduce and diagnose at run-time. Common synchronization errors in multithreaded programs include data races, atomicity violations, and deadlocks. This paper focuses on deadlocks. Informally, a *deadlock* occurs when some threads are permanently blocked.

Errors in multithreaded programs are often difficult to find and reproduce because they manifest themselves only in some rare executions, based on underconstrained (effectively non-deterministic) scheduling decisions. Therefore, algorithms that can detect *potential errors* in an observed execution, even if the potential error does not actually occur in that execution, are valuable debugging tools that can greatly increase the probability of detecting insidious scheduling-dependent errors during testing. This has motivated work done on run-time detection of potential races [17, 23, 16], potential atomicity violations [10, 25], and potential deadlocks [12, 3, 2, 11].

Work on run-time detection of potential deadlocks has focused on programs that use locks. The GoodLock algorithm proposed by Havelund detects potential deadlocks involving two threads [12]. This algorithm was later generalized to handle any number of threads, independently by us [2], and by Bensalem and Havelund [3]. Our multi-thread GoodLock algorithm detects potential for deadlocks in programs that use block structured locking, as in Java. This paper extends that algorithm to handle non block structured locking as well. Bensalem and Havelund's algorithm [3] handles non block structured locking, but does not incorporate some of the optimizations in our algorithm [2].

Semaphores are another common synchronization mechanism [20]. Semaphores can be used to provide mutual exclusion or condition synchronization. This dual-use nature of semaphores makes analysis of programs that use them more challenging. This paper presents algorithms to detect potential for deadlocks involving semaphores.

Condition variables are another common synchronization mechanism. For example, the POSIX threads library provides `pthread_cond_wait` and `pthread_cond_signal` routines for condition synchronization. `pthread_cond_wait` blocks the calling thread until the specified condition is signaled using the `pthread_cond_signal` routine. Java has `wait` and `notify` routines for condition synchronization. Lost notifies (or lost signals) are a common cause of blocked threads in programs that use condition variables [8, 13]. A notify is lost if it occurs before the thread it should wake actually

---

[*] Address: Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400. This work was supported in part by NSF under Grants CNS-0509230 and CCR-0205376 and by ONR under Grants N00014-02-1-0363 and N00014-04-1-0722. Email: {ragarwal,stoller}@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~{ragarwal,stoller}

PADTAD-IV, July 17, 2006, Portland, Maine, USA.

calls wait. As a result, the notify has no effect, and when that thread does call wait, it may wait forever. This paper presents algorithms to detect potential for lost notifies.

In summary, the paper makes the following contributions:

- A definition of potential for deadlock and potential for lost notify in an execution for programs that use one or more of the following synchronization primitives: locks (block structured as well as non block structured), condition variables, and semaphores.
- A definition of the feasible permutations of an execution for such programs; this is the basis for the definition of potential for deadlock.
- Algorithms to detect potential for deadlocks in such programs.

Our work has some limitations intrinsic to pure run-time approaches, which look only at executions (sequences of states) not at the program itself. First, we only attempt to analyze the effect of different schedules, not the effect of different inputs to the program. Second, some of the permutations we identify as feasible might not be possible executions of the program. Prior work on run-time detection of potential atomicity violations [25, 24], and potential deadlocks [2, 3] share these limitations and have proven effective in practice nevertheless.

In future work, we plan to evaluate the algorithms presented in this paper by implementing them and testing them on benchmarks. For algorithms presented in the paper, we plan to investigate if more efficient algorithms are possible. In particular, we think that algorithms for detecting potential for deadlock due to semaphores presented later in Section 4, can be improved by identifying some orderings on the semaphores.

The paper is structured as follows. Section 2 defines potential for deadlock and potential for lost signals. Sections 3 and 4 discuss run-time detection of potential deadlocks involving locks, and semaphores respectively. Section 5 discusses detection of potential for lost signals. Section 6 discusses detection of potential deadlocks involving multiple synchronization mechanisms. Section 7 discusses related work.

2. POTENTIAL FOR A DEADLOCK

The definitions and algorithms in this paper apply to programs in any language that use the following synchronization mechanisms. For illustration, we indicate the availability of these mechanisms in Java and C++. For concreteness, we will mainly use the Java terminology in the rest of the paper.

- locks: Block structured locks (*i.e.*, locks whose acquires and releases are nested so that the most recently acquired lock is the next one to be released) are built into Java. The Java 5 concurrency library (`java.util.concurrent`) and the POSIX pthread library for C provide locks that are not necessarily block structured. In pthreads, locks are called mutexes.
- semaphores: Semaphores are provided by the Java 5 concurrency library and by the pthread library for C. We call the operations on semaphores `up` and `down`.
- condition variables: Condition variables are built into Java; the operations are called `wait`, `notify`, and `notifyall`. Condition variables are provided by the POSIX pthread library for C; the operations are called `pthread_cond_wait`, `pthread_cond_signal`, and `pthread_cond_broadcast`. In Java and pthreads, a lock is associated with each condition variable, and it should be held whenever an operation on the condition variable is invoked.
- fork and join: Java provides `start` and `join` constructs to start and join. In the POSIX pthread library, the corresponding operations are `pthread_create` and `pthread_join`.

The programs may use other synchronization mechanisms as well, although this may cause our algorithms to produce some false alarms, unless the definition of feasible permutation is extended to reflect ordering constraints imposed by those other mechanisms.

An *event* is one step in the execution of a program. This paper considers events that perform the following kinds of operations: acquire and release of locks, wait and notify on condition variables, up and down operations on semaphores, accesses to shared variables, and thread start, join, and termination operations.

A *trace tr* is a sequence of events in a given execution. A *feasible permutation* of a trace is a trace that is consistent with the original order of events from each thread and with constraints imposed by synchronization events. The constraint imposed by locks is that no lock is held by multiple threads at the same time. The constraints imposed by other synchronization mechanisms are expressed as happens-before orderings. Here we present the framework; orderings imposed by specific kinds of synchronization events are described later.

Happens-before is a partial order on the events in an execution. If event e_1 happens-before event e_2, then e_1 must occur before e_2 in all feasible permutations of the trace.

For an event e in trace tr, we call e a *blocking event* if one of the following holds:

- e is an acquire of a lock l by thread t, and l is currently held by another thread when e occurs.
- e is a wait on a condition variable.
- e is a down on a semaphore whose value is 0.

We do not currently look for deadlocks involving join, so we do not classify it as a blocking event.

We say that a trace tr deadlocks if a set T of threads in tr exists, such that the last event for each thread in T is a blocking event, and all threads in tr not in T have terminated.

An execution trace has *potential for deadlock*, if some feasible permutation of the trace deadlocks. Informally, we also say that the executed program has potential for deadlock.

The above definition considers all synchronization mechanisms together. We develop separate algorithms for detection of potential deadlocks involving different synchronization mechanisms before combining them. Next we define potential for deadlock due to a single synchronization mechanism. These definitions serve as correctness conditions for those algorithms.

An execution trace has potential for deadlock due to locks / semaphores if some feasible permutation of the trace restricted to operations on the specified synchronization mechanism and operations on threads (*i.e.*, ignoring all other operations) deadlocks.

We do not define "potential for deadlock due to condition variables" here, because it is essentially the same as potential for lost notify, defined below.

For potential deadlock due to locks, we also define a simpler conservative condition that can be checked more efficiently. Because this condition is conservative, checking it can produce false alarms. Our experience so far suggests that such false alarms are rare in practice.

A program has potential for deadlock due to locks ignoring gate locks if there exist distinct threads $t_0, \ldots, t_{m-1}$ and locks $l_0, \ldots, l_{m-1}$ in the given trace tr such that, for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \ mod \ m}$. We call this condition Potential for Deadlock due to Locks Ignoring Gate Locks (PDL-IGL) condition. This condition ignores the effect of *gate locks* [12], which are locks that are held when other locks are acquired and prevent deadlocking interleavings (*i.e.*, permutations) of the acquires of those locks.

An execution trace tr has *potential for lost notify* if it contains a notify or notifyall event e such that there is a feasible permutation of tr in which e wakes up fewer threads than it does in tr. This is possible when the wait event of one of the threads woken in tr is not constrained to happen-before e. Note that a notify that does not wake any threads in tr might lead to some permanently blocked threads in the monitored execution tr, but this is easily detected, and we do not consider such actual lost notifies to be potential lost notifies.

Next we discuss happen-before orderings due to start/join events.

When a thread t_1 calls t_2`.start()` to start another thread t_2, then the thread start event in t_1 happens-before the first event of t_2. Similarly, when a thread t_1 calls t_2`.join()` to wait for thread t_2 to terminate, then the the last event of t_2 happens before the thread join event in t_1.

In addition, we consider orderings due to control dependencies on accesses to shared variables which is useful for condition synchronization:

- A read event e_r (on some shared variable) that occurs in the boolean condition in an if-then statement or while statement happens after the previous write event e_w to that variable (*i.e.*, e_w happens-before e_r) and happens-before the next write event to that variable.

Intuitively, this condition helps make permutations in which the condition would have a different value infeasible. This is important for analysis of condition synchronization, because the synchronization operations are often guarded by the condition in an if-then or while statement. In contrast, such orderings can usually be safely ignored in analysis of locking. To completely ensure infeasibility of such permutations, we would also need to consider flow of values from shared variables into unshared variables used in conditions. We conjecture that in practice this would eliminate too few false alarms to be worthwhile. We plan to evaluate this conjecture experimentally.

The instrumentations needed to detect such orderings can easily be inserted by a source-code transformation but cannot easily be inserted by lower level (*e.g.*, bytecode) transformations.

To keep track of these orderings, one can use vector clocks [15], as in [16], or thread segment identifiers, as in [25].

Orderings due to other events are discussed in later sections.

3. DETECTION OF POTENTIAL DEADLOCKS INVOLVING LOCKS

We review our algorithm [2] for run-time detection of potential deadlocks for programs that use block structured locking, and then describe how to extend it to handle general locking. It constructs a run-time lock tree for each thread, as in Havelund's GoodLock algorithm [12]. The run-time lock tree for a thread represents the nested pattern in which locks are acquired and released by the thread. Each node of the run-time lock tree is labeled with a lock and represents the thread acquiring that lock. There is an edge from a node n_1 to a node n_2 if n_1 represents the most recently acquired lock that the thread holds when it acquires the lock associated with n_2. At each instant, each run-time lock tree has one node designated as the *current node*; the path from the root of the tree to that node represents the nested acquires of locks held by that thread at that instant. If a thread re-acquires a lock that it already holds, its run-time lock tree does not contain a node representing the re-acquire. [1] When a thread acquires a lock that it does not already hold, if there is already a child of the current node labeled with that lock, that child becomes the current node, otherwise a new child labeled with that lock is created and becomes the current node.

At the end of the execution, it constructs a run-time lock graph, which is a directed graph $G = (V, E)$, where V contains all the nodes of all the run-time lock trees, and the set E of directed edges contains (1) *tree edges:* the directed (from parent to child) edges in each of the run-time lock trees, and (2) *inter edges:* bidirectional edges between nodes that are labeled with the same lock and that are in different run-time lock trees.

For a run-time lock graph G, a *valid path* is a path that does not contain consecutive inter edges and such that nodes from each lock tree appear as at most one consecutive subsequence in the path. Similarly, a *valid cycle* is a cycle that does not contain consecutive inter edges and nodes from each thread appear as at most one consecutive subsequence in the cycle. As shown in [1, 2], there is a valid cycle iff the execution has potential for deadlock due to locks ignoring gate locks.

Existence of a valid cycle is detected by traversing all valid paths starting from the root of each lock tree in G using a modified depth-first search (DFS) algorithm, which differs from standard DFS in two ways. First, it traverses only valid paths, because it extends the current path (on the search stack) only with edges satisfying both criteria for validity. Second, a node all of whose neighbors have been explored may be explored multiple times (along incoming inter edges); this is necessary because the set of threads with some lock-tree nodes on the stack might be different on different visits, so the set of valid paths that can be explored

---

[1] This matches the semantics of Java locks. For pthread mutexes, it is an error for a thread to re-acquire a mutex it holds; we assume run-time checking for this is already done.

by continuing the search from that node is different. The above algorithm is optimized by observing that many valid paths share a common suffix. For details, see [1].

To handle general (*i.e.*, not necessarily block structured) locking, the run-time lock tree construction needs to be changed as follows. For each thread, the new lock trees keep track of which locks are held by a thread when it acquires another lock. The root node of each lock tree is labeled with the name t of the thread. The root has one child for each lock acquired by t. Each of those nodes is labeled with the name l of one of those locks and has a child labeled with a lock l' iff t acquired l' while holding l. Thus, the height of each lock tree is at most 2. Nodes at depth one (child nodes of root) that are also leaf nodes are redundant and hence can be removed from the lock tree. After the run-time lock trees are constructed, the run-time lock graph construction remains the same. The algorithm to detect valid cycles remains as before.

As an example, Figure 1 shows the run-time lock graph for the illustrative program in Figure 2 which uses non block structured locks. Note that the lock tree of each thread has height 2. For example, the lock tree for thread T4 has height 2, even though T4 holds 3 locks simultaneously. The graph in Figure 1 contains several cycles including the following three, where $\texttt{l}i^{\mathrm{T}j}$ denotes the node for lock `l`i in the run-time lock tree for thread j: $\texttt{l3}^{\mathrm{T1}} \rightarrow \texttt{l3}^{\mathrm{T2}} \rightarrow \texttt{l3}^{\mathrm{T4}} \rightarrow \texttt{l3}^{\mathrm{T1}}$, $\texttt{l1}^{\mathrm{T1}} \rightarrow \texttt{l2}^{\mathrm{T1}} \rightarrow \texttt{l2}^{\mathrm{T2}} \rightarrow \texttt{l3}^{\mathrm{T2}} \rightarrow \texttt{l3}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T3}} \rightarrow \texttt{l1}^{\mathrm{T3}} \rightarrow \texttt{l1}^{\mathrm{T1}}$, and $\texttt{l3}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T4}} \rightarrow \texttt{l3}^{\mathrm{T4}} \rightarrow \texttt{l3}^{\mathrm{T1}}$.

The first cycle is not valid because it contains two or more consecutive inter edges. The second cycle is not valid because nodes from thread T1 appear in more than one subsequence. The third cycle is valid and hence indicates a potential deadlock. Specifically, it indicates that the program in Figure 2 can deadlock if thread 1 acquires lock `l3` and waits for lock `l4` and thread 4 acquires lock `l4` and waits for lock `l3`.

Now we show that PDL-IGL holds iff the run-time lock graph G contains a valid cycle. Suppose PDL-IGL holds, *i.e.*, there exist distinct threads $t_0, \ldots, t_{m-1}$ and locks $l_0, \ldots, l_{m-1}$ such that for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \ mod \ m}$. Let n_i and n'_i denote the nodes in T_i corresponding to the acquire of l_i and the acquire of $l_{i+1 \ mod \ m}$ nested within it, respectively. Since thread t_i acquires lock l_i and waits for lock $l_{i+1 \ mod \ m}$, there is an edge from n_i to n'_i in run-time lock tree T_i for t_i (by construction). Also, there is an inter edge from n'_i in run-time lock tree T_i to $n_{i+1 \ mod \ m}$ in run-time lock tree $T_{i+1 \ mod \ m}$ in G (by construction). These tree edges and inter edges together form a valid cycle.

Next, we show that existence of a valid cycle C in G implies that the PDL-IGL condition holds. The cycle involves nodes from more than one lock tree, because nodes of a single tree cannot be involved in a cycle. Suppose C had nodes n_i and n'_i in run-time lock tree T_i for thread t_i, $i \in 0..m-1$. Also, nodes n'_i and $n_{i+1 \ mod \ m}$ are labeled with the same lock (they are consecutive nodes from different lock trees and this is only possible through an inter edge which connects two similar labeled locks). Thus, existence of C implies there exist distinct threads $t_0, \ldots, t_{m-1}$ and locks $l_0, \ldots, l_{m-1}$ (node n_i corresponds to lock l_i and node n'_i corresponds to lock $l_{i+1 \ mod \ m}$) such that, for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \ mod \ m}$. Hence, the PDL-IGL condition holds.

However, the algorithm does not consider gate locks and therefore produces false alarms whenever some common lock acquired by at least two threads prevents deadlocks. To eliminate these false alarms, we extend the algorithm to check whether there exist distinct $t_0 \ldots t_{m-1}$ and locks $l_0 \ldots, l_{m-1}$ such that for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \ mod \ m}$ and there do not exist t_i, t_j, and l such that t_i and t_j hold l when acquiring l_i and l_j, respectively. (Such a lock l is called a *gate lock* for the cycle). We call this the Potential for Deadlocks from Locks (PDL) condition.

The above algorithm can be extended to handle gate locks. To account for gate locks, each thread maintains for each edge (l, l') in the lock tree, a set of locks that were held but not released at the time the thread acquired lock l' while also holding lock l. The acquire of lock l' while holding lock l can happen multiple times during the execution of a thread possibly with a different set of locks already held. Each such set of locks is maintained for each edge. After detecting a valid cycle as above, we check if there is a gate lock preventing a deadlock by checking if any two edges e_1 and e_2 in a valid cycle share a common lock by taking an intersection of every set of locks for e_1 with every set of locks for e_2 and checking if any of the intersection resulted in a non-empty set. These checks eliminate false alarms due to gate locks, but makes the algorithm more expensive.

4. DETECTION OF POTENTIAL DEADLOCKS INVOLVING SEMAPHORES

Semaphores can be used to provide mutual exclusion or condition synchronization. This dual-use nature of semaphores makes analysis of programs that use them more challenging. To detect potential for deadlocks involving semaphores, we first use heuristics to determine which semaphores are being used for mutual exclusion. These semaphores are then analyzed exactly as if they were locks, with `down` treated as acquire, and `up` treated as release. The other semaphores are analyzed as described below.

We classify a semaphore *sem* as used for mutual exclusion in a given execution σ if *sem*'s initial and maximum values in σ are 1 and, letting σ' be the restriction of σ onto operations on *sem*, each `down` in σ' either is the last event in σ' or is immediately followed by an `up` by the same thread.

Semaphores not used for mutual exclusion are usually used for condition synchronization. They induce the following happens-before ordering:

- An `up` event e_u that unblocks a thread blocked on a down event e_d happens-before $succ(e_d)$, where $succ(e)$ is the event immediately following e on the same thread.

To detect potential for deadlocks due to semaphores not used for mutual exclusion, we look at all feasible permutations allowed by the ordering constraints, tracking the values of the semaphore. If there is a permutation which can result in a deadlock, a warning of a potential for deadlock is issued.

Consider the program for the cigarette smokers problem [14] shown in Figure 3. It uses 4 semaphores for condition synchronization and has a potential for deadlock involving semaphores. Consider, the following deadlock-free trace of the program. The semaphores `tobacco`, `paper`, and `matches` are initialized to 0, and `order` is initialized to 1. The agent thread does a down on `order`, followed by an up

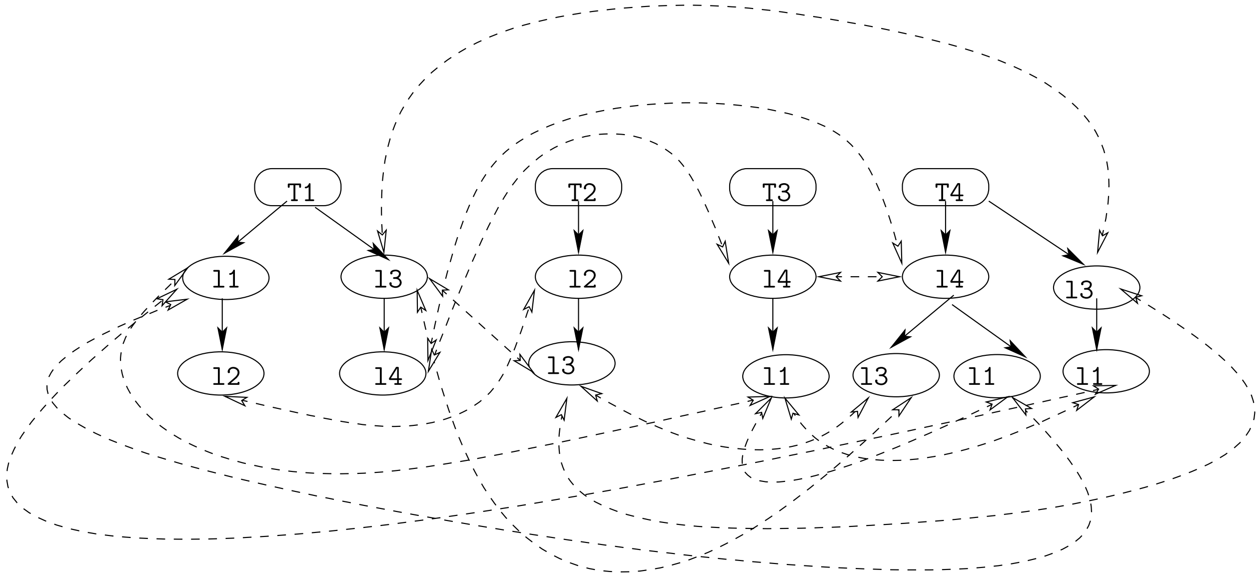

Figure 1: Run-time lock graph

```
Thread 1:
  acquire(l1);
  acquire(l2);
  release(l1);
  release(l2);
  acquire(l3);
  acquire(l4);
  release(l4);
  release(l3);
```

```
Thread 2:
  acquire(l2);
  acquire(l3);
  release(l3);
  release(l2);
```

```
Thread 3:
  acquire(l4);
  acquire(l1);
  release(l1);
  release(l4);
```

```
Thread 4:
  acquire(l4);
  acquire(l3);
  acquire(l1);
  release(l4);
  release(l1);
  release(l3);
```

Figure 2: Synchronization behavior of 4 threads.

on `tobacco` and `paper`, making `tobacco` and `paper` available, and then blocks waiting for `order` semaphore to be signaled. smoker 1 which was initially blocked waiting for `tobacco` uses `tobacco` and `paper` and then performs up on the `order` semaphore, following which the agent thread unblocks and makes `paper` and `matches` available by doing an up on those semaphores. This time smoker 2 unblocks and uses up `paper` and `matches`.

Based on the happens-before ordering introduced above, the above execution trace is drawn as a partial order in Figure 4. In the figure, t,p,m,o stand for tobacco, paper,matches, and order respectively. Each event happens-before the next event on the same thread, and an edge from an event e to an event e' of another thread means that e happens-before e'. There is a feasible permutation where the agent thread does a down on `order`, followed by an up on `tobacco` and an up on `paper`, followed by a down on `tobacco` by smoker 1 and a down on `paper` by smoker 2. The last 2 events have no causal predecessors, so they can occur as above, leaving the system deadlocked. Therefore, the above execution trace has a potential for deadlock.

5. DETECTION OF POTENTIAL FOR LOST NOTIFIES

As described in Section 1, lost notifies are a common cause of blocked threads in programs using condition variables.

Our algorithm to detect potential for lost notifies is based directly on the definition in Section 2: for each notify or notifyall event e_n, for each thread t woken by e_n, there is a potential for lost notify if t's corresponding wait event e_w does not happen-before e_n. Lost notify can result in multiple threads blocked on a wait if the wait in those threads does not happen before the corresponding notify or notifyAll intended for those threads.

Consider the program shown in Figure 5. In this program, the intended behavior is that the `computeThread` object repeatedly waits, waiting for the `EventHandler` object to notify it whenever an `update` event happens. The `computeThread` then unblocks and performs the computation. However, this program has a potential for a lost notify that may cause the desired computation not to occur. That may happen if the `EventHandler` object notifies the `computeThread` of an `update` event before it waits, resulting in a lost notify. Our algorithm warns of potential for lost notifies even if notifies are not lost in the observed execution.

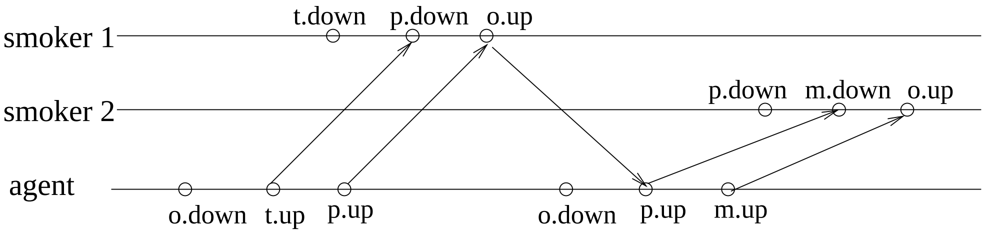

Figure 4: Partial order for an execution trace of cigarette smokers problem.

Happens-before orderings induced by condition synchronization (as well as start/join and other synchronization mechanisms) are considered in this analysis. Specifically, we consider the following ordering constraints due to condition variables.

- For each notify or notifyAll event e_n and each wait event e_w that is notified by e_n, e_n happens-before $succ(e_w)$, where $succ(e)$ is the event immediately after e on the same thread. Since, the same lock l must be held when e_w and e_n occur and only one thread can hold a given lock at a time, this is equivalent to saying that the release of l after e_n happens-before $succ(e_w)$.

This condition is similar to the ordering on down and up on semaphores not used for mutual exclusion in Section 4; this reflects the fact that semaphores not used for mutual exclusion are typically used for condition synchronization.

We continue to use the orderings on reads and writes on shared variables given in Section 4.

Figure 6 shows typical code for producer-consumer synchronization implemented using condition variables. The above code is taken from Java tutorial on wait and notify at Sun's website [21]. Specifically, it shows the `put` and `get` methods for a shared buffer with capacity one. The producer repeatedly invokes the `put` method, while the consumer repeatedly invokes the `get` method. Consider the following execution trace, where `available` is initially false: the consumer executes `get` until it blocks on the `wait`, then the producer executes `put` to completion without blocking, then the awoken consumer executes the rest of `get`. Our ordering constraints imply that the consumer's read of `available` happens-before the producer's write to `available`. This along with locking constraints ensure that the consumer's first acquire of the lock on the shared buffer happens-before the producer's acquire of that lock. Hence, the consumer's `wait` happens-before the producer's `notifyAll`, hence there is no potential for lost notify. The consumer's `notifyAll` does not wake any thread in the observed execution, so our algorithm does not produce any warnings for it.

6. DETECTION OF POTENTIAL DEADLOCKS INVOLVING LOCKS, CONDITION VARIABLES, AND SEMAPHORES

As discussed in Section 2, for programs involving locks, condition variables, and semaphores, a potential for deadlock occurs if some feasible permutation of the execution trace deadlocks. To find feasible permutations, we first determine if the semaphores are used for mutual exclusion using the heuristics presented in Section 4. Semaphores used for mutual exclusion are treated exactly like locks. All the ordering constraints introduced in previous sections and the constraint imposed by locks (no lock is held by multiple threads at the same time) are taken into account.It is unclear how to extend the lock-graph-based algorithm in Section 3 to efficiently consider the effects of condition variables and semaphores. Therefore, when considering all three synchronization mechanisms, we currently use a naive algorithm that checks each feasible permutation of the trace for deadlock.

Consider the program shown in Figure 7. Although the example is a bit contrived, it is interesting as it uses locks, semaphores, and condition variables and has both deadlock-free executions and deadlocking executions. The program has three threads. Each thread invokes a separate method `doWait`, `doNotify`, and `doCompute` on the same shared object o. Threads invoking `doWait` and `doNotify` methods use the same shared object o' as the argument. Consider the following deadlock-free trace. `sem` is initialized to 0. Thread 3 first invokes the `doCompute` method, acquires the lock on o, does an `up` operation on `sem` and releases the lock on o. Thread 1 then invokes the `doWait` method, acquires the lock on o, acquires the lock on o', and waits releasing the lock on o'. Thread 2 then invokes the `doNotify` method, acquires the lock on o', does a notify and releases the lock on o'. Thread 1 then wakes up, releases the lock on o and proceeds to termination.

Given this trace, our algorithm correctly identifies two synchronization problems in the program, corresponding to the following feasible permutations of this trace. One feasible permutation which results in a deadlock is if thread 1 invokes `doWait` method and acquires the lock on o, acquires the lock on o', and waits releasing the lock on o'. Each of the threads is then blocked: Thread 1 on the wait, thread 2 on `sem.down`, and thread 3 trying to acquire a lock on o. Another feasible permutation is if thread 3 invokes the `doCompute` method, followed by thread 2 invoking the `doNotify` method, followed by thread 1 invoking the `doWait` method. This permutation results in a lost notify.

7. RELATED WORK

7.1 Run-time analysis

The GoodLock algorithm [12], multi-thread GoodLock algorithms developed by us [2] and Bensalem and Havelund [3], and the algorithm in Visual Threads[11] detect potential for deadlocks due to locks. These algorithms do not consider semaphores or condition variables.

ConTest [6] detects actual deadlocks, not potential deadlocks, and therefore may miss some potential deadlocks. On

```
Initially, tobacco =0, paper =0,
 matches =0, order =1

 smoker 1
 ---------
  while (1) {
   tobacco.down()
   paper.down()
   order.up()
  }

 smoker 2
 ---------
  while (1) {
   paper.down()
   matches.down()
   order.up()
  }

 smoker 3
 ---------
  while (1) {
   matches.down()
   tobacco.down()
   order.up()
  }

 agent
 ---------
  while (1) {
   order.down()
   up on one of tobaco, paper, matches at random
   up on one of the three at random but not above
  }
```

Figure 3: Program for the cigarette smokers problem.

the other hand, ConTest's scheduling perturbation heuristics make potential deadlocks of all kinds (including deadlocks due to condition synchronization) more likely to manifest themselves as actual deadlocks during testing with ConTest, compared to testing without ConTest. An extension to ConTest implements a run-time deadlock checking algorithm that combines information obtained from multiple executions of the program [9]. Farchi *et al.* [8] present heuristics that increase the probability that lost notification bugs will manifest themselves during testing.

Pulse [14] is an operating system mechanism that uses speculative execution to detect deadlocks involving reusable resources, such as locks, and consumable resources, such as semaphores. Pulse can handle many synchronization mechanisms, including the ones we consider, but it is designed to detect actual deadlocks, not potential for deadlock.

Sen *et al.* [19] define feasible permutations of an execution of a multithreaded program and give algorithms to compute them which are implemented in the JMPaX tool [18]. They generate orderings among all reads and writes of shared variables. They consider lock acquires and releases as writes to shared variables. This is overly conservative as it prevents permuting two synchronized blocks. Condition synchronization is handled by generating writes to shared variables by notified and notifying threads; this is similar to the ordering we consider. Chen and Rosu [5] define a more relaxed causal ordering that takes the program's control dependence into account. These techniques have not yet been applied to detect potential for deadlocks.

```
class EventHandler extends ... {

  public void handleEvent(Event e) {
    switch(e.type) {
      update:
        data.update(e);
        synchronized(computeThread) {
         computeThread.notify();
        }
        break;
        ....
    }
  }
}


class ComputeThread extends Thread {

  public void run {
    while(true) {
      synchronized(this) {
         this.wait();
         compute();
      }
    }
  }
}
```

Figure 5: A program with a potential for lost notify.

7.2 Static Analysis

Boyapati, Lee and Rinard [4] introduce a static type system that ensures Java programs are deadlock-free. The types express a partial order among locks. Deadlocks involving locks and a condition variable are prevented by the simple constraint that a thread can invoke e.`wait` only if the thread holds no locks other than the lock on e. Semaphores and lost notifies are not considered.

Engler *et al.* [7], von Praun [22], and Williams *et al.* [26] developed inter-procedural static analyses that detect possible deadlocks. Engler *et al.*'s analysis detects only possible deadlocks involving synchronization primitives used like locks. They do not handle condition variables or semaphores not used for mutual exclusion. They use heuristics to determine which semaphores are used for mutual exclusion. Williams *et al.*'s analysis detects only possible deadlocks involving only locks. von Praun considers locks, and condition variables, but does not consider lost notifies or semaphores. These static analyses are also based on checking whether locks are acquired in a consistent order by all threads. These static analyses are more sophisticated and more accurate than Boyapati *et al.*'s deadlock types but still produce nu-

```
public synchronized int get() {
    while (available == false) {
        try {
            //Wait for Producer to put value.
            wait();
        } catch (InterruptedException e) { }
    }
    available = false;
    //Notify Producer that value has been retrieved.
    notifyAll();
    return contents;
}

public synchronized void put(int value) {
    while (available == true) {
        try {
            //Wait for Consumer to get value.
            wait();
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    //Notify Consumer that value has been set.
    notifyAll();
}
```

Figure 6: Shared buffer implemented using wait and notify.

```
public synchronized doWait(Object ob) {
  compute();
  try {
    synchronized(ob) {ob.wait();}
  } catch (InterruptedException e) { }
}


public doNotify(Object ob) {
  sem.down();
  synchronized(ob) {ob.notify();}
}

public synchronized doCompute() {
  compute();
  sem.up();
}
```

Figure 7: Program with a potential for deadlock involving multiple synchronization primitives.

merous false alarms. Engler *et al.* and Williams *et al.* partially address this problem by using heuristics to rank or suppress warnings that seem more likely to be false alarms. We expect that run-time detection of potential deadlocks, like run-time detection of races and atomicity violations, will produce fewer false alarms than static analysis, because aliasing and infeasible path elimination are not problems for run-time analysis.

[2] uses the idea of using static analysis to optimize run-time checking to detect potential deadlocks involving locks.

8. REFERENCES

[1] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. Technical Report DAR-05-25, Computer Science Department, SUNY at Stony Brook, Sept. 2005. Available at http://www.cs.sunysb.edu/~ragarwal/deadlock/.

[2] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference*, volume 3875 of *Lecture Notes in Computer Science*, pages 191–207. Springer-Verlag, 2006. Received the conference's Best Paper Award.

[3] S. Bensalem and K. Havelund. Scalable deadlock analysis of multi-threaded programs. In *Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference*. Springer-Verlag, Nov. 2005.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 211–230, Nov. 2002.

[5] F. Chen and G. Roşu. Predicting concurrency errors at runtime using sliced causality. Technical report, Computer Science at UIUC (No. UIUCDCS-R-2005-2660), 2005.

[6] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[7] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. 24th ACM Symposium on Operating System Principles*, pages 237–252. ACM Press, Oct. 2003.

[8] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society.

[9] E. Farchi, Y. Nir-Buchbinder, and S. Ur. Cross-run lock discipline checker for java. Presentation at the 2005 IBM Verification Conference.

[10] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. 31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Jan. 2004.

[11] J. J. Harrow. Runtime checking of multithreaded applications with Visual Threads. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*, pages 331–342. Springer-Verlag, Aug. 2000.

[12] K. Havelund. Using runtime analysis to guide model checking of java programs. In *Proc. 7th Int'l. SPIN*

Workshop on Model Checking of Software, volume 1885 of *Lecture Notes in Computer Science*, pages 245–264. Springer-Verlag, Aug. 2000.

[13] D. Hovemeyer and W. Pugh. Finding concurrency bugs in java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2005.

[14] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. In *Proceedings of the USENIX Annual Technical Conference*, pages 31–44, April 2005.

[15] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al, editor, *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[16] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proc. ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[18] K. Sen, G. Roşu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 337–346. ACM, 2003.

[19] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *International Journal on Software Technology and Tools Transfer (STTT) (To Appear)*, 2005. Previous version appeared in TACAS'04, LNCS volumn 2988, pages 123-138.

[20] W. Stallings. *Operating Systems*. Prentice-Hall, 5th edition edition, 2005.

[21] http://java.sun.com/docs/books/tutorial/essential/threads/waitAndNotify.html.

[22] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, ETH Zürich, 2004.

[23] C. von Praun and T. R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, Oct. 2001.

[24] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 137–146. ACM Press, Mar. 2006.

[25] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, Feb. 2006. A preliminary version appeared in *Proceedings of the Third Workshop on Runtime Verification (RV)*, Electronic Notes in Theoretical Computer Science 89(2), Elsevier, 2003.

[26] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *Proc. 2005 European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, July 2005.