View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

A Scalable, Sound, Eventually-Complete Algorithm for Deadlock Immunity

Horatiu Jula and George Candea

EPFL – Swiss Federal Institute of Technology, Lausanne, Switzerland

Abstract. We introduce the concept of *deadlock immunity*—a program's ability to avoid all deadlocks that match patterns of deadlocks experienced in the past. We present here an algorithm for enabling large software systems to automatically acquire such immunity without any programmer assistance. We prove that the algorithm is sound and complete with respect to the immunity property. We implemented the algorithm as a tool for Java programs, and measurements show it introduces only modest performance overhead in real, large applications like JBoss. Deadlock immunity is as useful as complete freedom from deadlocks in many practical cases, so we see the present algorithm as a pragmatic step toward ridding complex concurrent programs of their deadlocks.

1 Introduction

Writing concurrent software is a challenging task, because it requires careful reasoning about complex interactions between concurrently-running threads. Programmers consider concurrency bugs to be some of the most insidious. An important category of such bugs result in deadlocks—situations in which a set of threads cannot make forward progress because each thread is waiting to acquire a lock held by another thread in that set. Avoiding the introduction of deadlock bugs during development is challenging, because large software systems are developed by multiple teams totaling hundreds to thousands of programmers. Testing is not a panacea either, because exercising all possible execution paths and thread interleavings is still infeasible for large programs; the result is that deadlock bugs do slip into most large production software. Unfortunately, debugging deadlocks is tedious, because they are hard to reproduce and diagnose.

We expect deadlocks to become more frequent, as multi-core CPUs lead to higher degrees of concurrency and encourage new software systems to be increasingly more parallel. There have been proposals for making concurrent programming easier, such as transactional memory [6], but issues concerning I/O and long-running operations still make it difficult to provide atomicity transparently (ironically, several transactional memory implementations resort to locking for implementing efficient transactions and can thus lead to application deadlocks). We believe that locks will continue being a primary vehicle for synchronization in multi-threaded applications.

Several approaches detect and prevent the introduction of deadlocks before a program runs, by using various forms of static analysis [4, 3, 16, 7]. These

approaches typically aim to find deadlock bugs in the source code and either let the programmer fix them, or automatically instrument the application with new locks that introduce serialization in the deadlock-prone code. The challenge faced by static approaches is that they either generate many false positives (i.e., wrongly identify deadlock bugs) and burden programmers with sifting through the reports to pick out the true bugs, or they do not scale to large applications due to resource consumption that is exponential in the size of the analyzed program. In fact, false positives vs. scalability appears to be an essential tradeoff in static techniques for finding deadlocks.

Dynamic approaches [5, 2, 17, 15, 12] often face a different challenge: false negatives. Since they rely exclusively on runtime information from the present execution (e.g., a lock trace), deadlocks may still occur, because they cannot be predicted. In fact, the pure version of the deadlock avoidance problem is generally undecidable, because it can be reduced to the halting problem [9][1]. One way to simplify the problem and circumvent undecidability is to save deadlock information that persists across executions, and leverage this knowledge to avoid solely the already-encountered deadlocks.

Our proposed approach detects deadlocks at runtime and saves the *contexts* in which they occurred, in order to avoid the contexts in future runs. This constitutes achieving "immunity" against the corresponding deadlocks. To avoid previously-seen deadlocks we employ program steering [10] and automatically change the scheduling of threads. A program with deadlock immunity will progressively eliminate the manifestations of its deadlocks bugs, by automatically avoiding a monotonically increasing set of deadlock contexts. We expect this approach to result in fewer false positives, because it relies on deadlock patterns that actually manifested, not on inferred deadlocks that may occur in the future. However, if a deadlock does not have a pattern similar to an already encountered one, our approach will not avoid it (false negative). To be precise, the false negative rate of our approach is exactly one per deadlock context, because all runs after the first occurrence will be free of the corresponding deadlock pattern.

Fortunately, deadlock immunity is often as useful as complete deadlock avoidance in practice, since the only difference is that one occurrence per deadlock pattern. Thus, software users now have the option of employing a tool based on our approach, instead of waiting for the manifest deadlock bugs to be fixed by software vendors. In fact, deadlock immunity must not be only an interim solution, but could also provide permanent immunity against those deadlocks, without having to risk the system destabilization often associated with patching.

This paper makes three main contributions: (a) An algorithm for developing deadlock immunity with no assistance from programmers or users; (b) Proof that the algorithm is sound (i.e., avoids deadlocks while preserving liveness) and eventually complete (i.e., avoids all deadlocks after a finite number of steps); and (c) Preliminary evidence that the algorithm can scale to large programs (over 350,000 lines of code) and large degrees of concurrency (up to 280 threads).

[1] In the limited space here we cannot do justice to all the prior work that has provided us with inspiration; we therefore include a more extensive survey in [9].

In the rest of this paper we describe the deadlock immunity algorithm (§2), describe a proof of its soundness and completeness (§3) and analyze its complexity (§4). We present an implementation for Java programs and a preliminary evaluation of effectiveness and performance in real systems (§5), after which the paper concludes (§6).

2 Algorithm for Deadlock Immunity

In this section we present the algorithm per se. After an overview and necessary definitions (§2.1), we describe in detail the instrumentation needed to intercept lock/unlock requests (§2.2). Afterward, we present the two main parts of our approach: the avoidance algorithm (§2.3) and the detection algorithm (§2.4).

2.1 Overview and Definitions

The deadlock immunity algorithm applies to the following abstract model of a multi-threaded program: there is a finite number of threads performing synchronization operations (i.e., lock and unlock) on a finite number of shared mutex locks. When a thread t performs a *lock(l)* on mutex l, it follows 3 steps: (1) t requests lock l; then (2) t waits until l becomes free, i.e., not held by any other thread; and finally (3) t acquires l. A thread can request only one lock at time, and a lock can be held by only one thread at a time. When t performs *unlock(l)*, it releases l, which becomes available to other threads for acquisition. A code region protected by a lock l (i.e., situated between a *lock(l)* and *unlock(l)*) is called a *critical section*. When a thread performs *lock(l')* within the critical section of lock l, we say the thread is doing a *nested lock* (if $l' \neq l$) or a *reentrant lock* (if $l' = l$). Both reentrant and nested locking is supported. We identify the *program position* p at which a thread requests or acquires a lock as the offset of the corresponding instruction within the program source code or binary.

A set of threads is deadlocked iff every thread from that set is waiting (step 2 above) for a lock held by another thread in that set. The immediate cause of a deadlock (or, alternatively, its context) is a given sequence of lock acquisitions that have led to the situation described above. A deadlock avoidance mechanism generally tries to predict impending deadlocks and dynamically reorder the lock acquisitions in order to avoid the predicted deadlocks.

The deadlock immunity algorithm consists of two parts, one that detects deadlocks and another that avoids them by forcing threads to yield (in step 1 above) when they are approaching a previously-seen deadlock. In order to detect deadlocks, we maintain a standard *resource allocation graph* RAG=$[V, E]$. The vertices $v \in V$ can be threads or locks, and the edges $e \in E$ can be request, hold, grant, or yield edges.

A *request edge* $t \rightarrow_r l$ represents thread t requesting permission to wait for lock l (step 1 above). A *grant edge* $t \xrightarrow{p}_g l$ represents the fact that thread t was granted permission by the algorithm to wait for lock l at position p in the program (i.e., to enter step 2). A *hold edge* $t \xleftarrow{p}_h l$ indicates that t has acquired lock l at position

p (step 3). A group of *yield edges* $t \xrightarrow{p_1}_y t_1$, ..., $t \xrightarrow{p_n}_y t_n$ indicates that thread t was forced to yield because threads $t_1, ..., t_n$ had acquired (or were granted) locks at positions $p_1, ..., p_n$; the usefulness of yield edges will become clear later on. In summary, a request edge is the manifestation in the RAG of a lock request and a hold edge the manifestation of a lock acquisition. A grant edge reflects the algorithm's decision to allow a thread to do a blocking wait for a lock, while a yield edge captures the immunity algorithm's decision to pause a thread in order to avoid a potential deadlock. In terms of notation, when the value of an edge label or an endpoint is irrelevant, we mark it with $*$ (as in $v_1 \xrightarrow{*}_y v_2$, or $v_1 \xrightarrow{*}_y *$).

A deadlock appears as a cycle in the RAG, involving exclusively request, grant, and hold edges (i.e., no yield edges). When avoiding deadlocks using thread yields, livelocks can arise; e.g., when a thread t_1 is forced to yield because of thread t_2, while thread t_2 waits for a lock held by thread t_1. We call such livelocks *avoidance-induced livelocks*.

Avoidance-induced livelocks appear as *yield cycles* in the RAG—a cycle is a yield cycle iff all yield edges emerging from its nodes belong to yield cycles. One can think of avoidance-induced livelocks as a group (conjunction) of yield cycles that intersect in a vertex v of the RAG *as well as* in all yield edges that emerge from v. The yield cycle construct enables the algorithm to detect and avoid all avoidance-induced livelocks the same way it detects and avoids deadlocks.

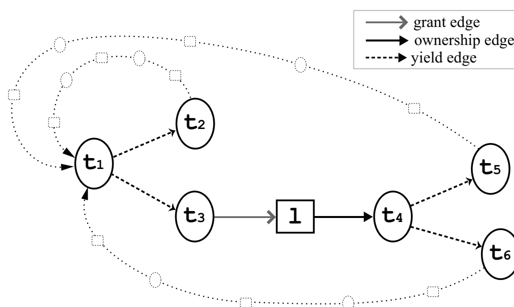

Fig. 1. Livelocked threads and yield cycles.

To illustrate the concept, consider Figure 1: for thread t_1 to be livelocked, all of its yield edges must be part of cycles, as well as all of t_4's yield edges, since t_4 is in one of t_1's yield cycles. If the RAG had solely the $(t_1, t_2, \ldots, t_1)$ and $(t_1, t_3, l, t_4, t_6, \ldots, t_1)$ cycles, then there would be no livelock, because t_4 could "evade" livelock through t_5, allowing t_1 to "evade" through t_3. If, as in Figure 1, cycle $(t_1, t_3, l, t_4, t_5, \ldots, t_1)$ is also present, then the threads have no way to make forward progress and are thus livelocked.

We use instruction location information to capture and save templates of deadlocks and induced livelocks. Remember, the program position p is an abstraction that denotes the location of an instruction in the source code or binary. A *template* is the set of program positions (edge labels) corresponding to the edges of a cycle in the RAG; remember that only grant, hold, and yield edges carry labels. For example, the template of the cycle $t_1 \rightarrow_r l_1 \xrightarrow{p_1}_h t_2 \rightarrow_r ...l_n \xrightarrow{p_n}_h t_1$ is $\{p_1, ..., p_n\}$. Templates capture the "contexts" in which deadlocks occur. A *template instance* is an instantiation of a template in a program execution, i.e., a set of *(thread, position)* tuples, representing distinct threads that are currently holding or have been granted locks at positions corresponding to the template,

that cover all positions in the template; e.g., the instantiation of $\{p_1, ..., p_n\}$ in the current state of a program would take the form $\{(t_1, p_1), ..., (t_n, p_n)\}$.

Templates are analogous to "antibodies"—the algorithm saves them to persistent storage and avoids their re-instantion in all future executions. Since both deadlocks and avoidance-induced livelocks have their templates saved to the same template history, we will refer to all of them as simply *templates*, the unified deadlock and induced livelock history simply *history*, and the deadlocks and avoidance-induced livelocks simply *cycles* when no distinction needs to be made.

An immunizing tool based on the proposed algorithm instruments programs such that all lock and unlock operations are intercepted and relayed to the immunity algorithm's *avoidance module*. This module shares an event queue and the history with the *detection module*, as illustrated in in Figure 2.

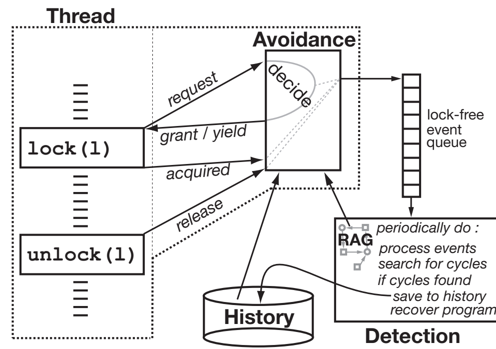

Fig. 2. Architecture of an immunizing tool.

The *avoidance module* runs synchronously with the application, in that it is invoked on every lock request, acquisition or release; avoidance decisions are made only for lock requests. This module is responsible for avoiding cycles, based on the templates stored in history. The avoidance module notifies asynchronously the detection component about events (lock request/acquisition/release) and decisions (grant/yield) using an event queue. On every lock request, this module checks whether any templates in the history would be instantiated by granting the requested lock. If not, the thread is allowed to proceed with locking; otherwise, the thread must yield.

The *detection module* runs asynchronously, in parallel with the program's threads. It periodically updates the RAG based on notifications received from the avoidance module, detects cycles in the RAG, and saves these cycles' templates to the history. It then restarts the deadlocked application's threads.

In summary, the proposed approach has three key features: (1) it captures execution-independent templates of previously-encountered deadlocks and avoids future instantiations of these templates; (2) detects and avoids livelocks induced by avoidance in exactly the same way as deadlocks (yield cycles allow us to cast a liveness property into an easily detectable safety property); (3) detects cycles asynchronously in a separate thread, in order to remove this expensive computation from the critical path. Given that a deadlocked application is not making any progress anyway, the only drawback of asynchrony is a potentially longer recovery time; the latter can easily be tuned by selecting a suitable period.

2.2 Instrumentation

The code or binary of the original program needs to be instrumented such that all lock and unlock operations are intercepted. The instrumentation replaces each call to a native lock/unlock with corresponding code that relays events to the avoidance module and exercises control over the scheduling of the calling thread, as shown in Figure 3. We discuss the instrumentation here as much as necessary for understanding the mechanics of the runtime system; the details appear in the next section.

When an immunized thread t wants a lock l at position p, it asks for permission from the avoidance module, which could ask the thread to yield, in which case the thread will sleep until permitted to proceed. When the avoidance module allows the thread to proceed, t uses the native locking mechanism to acquire l and then notifies the avoidance module of the acquisition event. An unlock operation is analogous, but simpler. This form of instrumentation is just one possible design choice—it could also be implemented inside the runtime, the operating system kernel, etc.

2.3 Avoidance

The avoidance module is responsible for controlling the schedule of threads to avoid previously-encountered deadlocks and avoidance-induced livelocks. The interface offered by this module to the instrumented threads consists of three operations: *Request*, *Acquired* and *Release*, that process lock requests, acquisitions and releases, respectively. We developed both a synchronized version of the algorithm, which uses a global lock to ensure atomicity of *Request* and *Release*, as well as a lock-free version, which eliminates this global lock, but requires additional checks in *Request*, *Release* and in the instrumentation of lock operations. We present here the lock-free version of the algorithm; the synchronized version appears together with the lock-free one in [9].

The *Request*, *Acquired* and *Release* operations are described in Figure 5. The core avoidance occurs in *Request*, whose return value determines whether a thread is paused or allowed to proceed. But, before discussing the algorithms in any more detail, we describe the data structures they employ.

Since avoidance is performed synchronously with calling threads, we aim to minimize the amount of work performed in the critical path; for this we choose fine-grain, efficient data structures in the avoidance module. The avoidance module shares only two data structures with the cycle detector: an event queue (updated by the avoidance module and read by the detector) and the template history (updated by the detector and read by the avoidance module). Cycle detection requires a consistent view of the RAG, thus requiring exclusive access to it, but is also a complex operation, thus holding the RAG locked for extended periods of time. We therefore opted to have all RAG updates be performed in the detection module, based on the events received through the queue. This provides optimal decoupling between the two modules.

The avoidance module uses the following data structures:

• *lockGrantees*[p] is a multiset[2] containing the threads that hold (or are granted) locks at position p; it is initially empty. This is the data structure used in searching for template instantiations.

• *history* is the set of templates of previously-encountered cycles. It is persistent, in that all updates are saved on disk, to be available in subsequent executions. At program startup, *history* is loaded in memory.

• *yieldCause*[t] is the cause of thread t's yield; *yieldCause*[t] has the same structure as a template instance, because it is in effect a subset of a real template instance. It is initially null.

• *yielders*[t, p] is the set of threads that are currently paused and have (t, p) in their *yieldCause*; the *yielders* map is initially empty.

• *yielders*[t, p] is the set of threads that are currently paused and have (t, p) in their *yieldCause*; the *yielders* map is initially empty.

• *owner*[l] is the thread currently holding lock l; it is initially null.

• *acqPos*[l] is the program position where lock l was acquired by its current owner; it is initially null.

• *nLockings*[l] is the number of times lock l was reentrantly locked; it is initially 0.

• *yieldLock*[t] is the lock (condition variable) used for pausing or waking up thread t.

The instrumentation for the lock operation (Figure 3) performs avoidance iff the current thread t does not already hold the lock l it is currently requesting (line 3). To perform the avoidance, the cycle detector is first notified of the request (line 4). Then

```
lock_wrapper( l )
 1  t := current thread ID
 2  p := current program position
 3  if owner[l] ≠ t then
 4     events := events + [request(t, l, p)]
 5     yCause := Request(t,l,p)
 6     if yCause ≠ null then
 7        foreach (t', p') ∈ yCause do
 8           native_lock_for_read( (t',p') )
 9        if ∀(t', p') ∈ yCause :
                1_{lockGrantees[p']}(t') > 0 then
10           native_lock( yieldLock[t] )
11           foreach (t', p') ∈ yCause do
12              yielders[t',p'] :=
                      yielders[t',p'] ∪ {t}
13              native_unlock_for_read((t',p'))
14           yieldCause[t] := yCause
15           events := events +
                 [yield(t, yieldCause[t])]
16           yieldLock[t].wait()
17           native_unlock( yieldLock[t] )
18        else
19           foreach (t', p') ∈ yCause do
20              native_unlock_for_read((t',p'))
21           goto 5
22  native_lock( l )
23  Acquired( t, l, p )

unlock_wrapper( l )
 1  t := current thread ID
 2  Release( t,l )
 3  native_unlock( l )
```

Fig. 3. Lock/Unlock Instrumentation.

[2] A multiset is a set whose elements can be present more than once. An element x is added to a multiset M using the $\uplus$ operator and removed from M using the $\setminus$ operator. For a multiset M and an element x, $1_M(x)$ represents the number of times x was added to M ($M := M \uplus \{x\}$) less the number of times it was removed ($M := M \setminus \{x\}$). x is deleted from M only when $1_M(x)$ reaches zero.

lines 5-6 check if it is safe for t to proceed with locking l. If yes, t uses the native locking mechanism to acquire l (line 22), notifies the avoidance module of the acquisition event (lines 23) and is done. If unsafe, i.e., the avoidance module returned a non-null *yield cause* on line 5, we must check if the yield cause is still current (lines 7-9). If yes, a yield is required: register t in all (thread, position) pairs from the yield cause (lines 11-12), store the yield cause (line 14), notify the detector about the yield decision (line 15), and wait until a thread from the yield cause releases all required locks and wakes t up (line 16). If the yield cause is no longer valid (line 18), we need to re-check whether it is safe to proceed (line 21). The immunity algorithm influences the thread schedule via a simple wait mechanism that relies on the condition variable $yieldLock[t]$; an alternative choice would have been to call yield in a loop, but that is more CPU-intensive.

When a thread requests a lock, the avoidance module checks whether granting that lock would instantiate any of the templates currently in *history*. An *instantiation* of template $T = \{p_1, ..., p_n\}$ is a set of $(thread, position)$ tuples representing distinct threads t that hold (or are granted) locks at positions p from T (i.e., $t \in lockGrantees[p]$), with all positions being covered (i.e., $\forall p \in T : lockGrantees[p] \neq \emptyset$). Thus, a template T would be instantiated by thread t being granted a lock at position p iff $p \in T$ and $T - \{p\}$ is already covered by threads different from t (i.e., $instance(T - \{p\}, \{t\}) \neq null$).

The *templateInstance(t,p)* helper, shown in Figure 4, returns a template instantiation that would occur, if thread t granted a lock at position p (lines 2-4).

If no potential template instantiations are found, *templateInstance(t,p)* returns null. The *instance(T,exclThreads)* helper returns an instantiation of template T that does not involve any thread from $exclThreads$ (line 8), or *null* if such an instantiation does not exist (line 9). If *templateInstance*(t,p) returns an instantiation $\{(t_1,p_1)\ldots,(t_n,p_n)\}$, then it means that $yieldCause[t] = \{(t_1,p_1)\ldots,(t_n,p_n)\}$, i.e., thread t has to wait until t_1 releases all the locks it acquired at p_1, or $\ldots$, or t_n releases all the locks it acquired at p_n. Whenever a thread t releases all locks acquired at position p, it wakes up all yielding threads t_i for which $(t,p) \in yieldCause[t_i]$ by performing a notify on the corresponding condition variable (i.e., on $yieldLock[t_i]$).

```
templateInstance( t,p )
1 foreach T ∈ history where p ∈ T do
2    templInstance := instance( T − {p}, {t} )
3    if templInstance ≠ null then
4       return templInstance
5 return null

instance( T, exclThreads )
1 if T = ∅ then
2    return ∅
3 else
4    pos := choose p ∈ T
5    foreach t ∈ lockGrantees[pos]
              where t ∉ exclThreads do
6       match :=
          instance(T \ {pos}, exclThreads ∪ {t})
7       if match ≠ null then
8          return {(t, pos)} ∪ match
9    return null
```

Fig. 4. Helpers for matching templates.

Figure 5 presents the core operations of the avoidance module.

When *Request(t,l,p)* is invoked, thread t is intially granted lock l (line 1). Then, one checks if thread t can safely proceed, i.e., if no template can be instantiated (line 2). If it is unsafe, the lock grant is canceled (line 6) and thread t must be forced to wait and the yield cause is returned (line 7). If it is safe, one lets t execute the lock by returning a null yield cause (line 7) and notifies the cycle detector about this decision (line 4).

However, a group of threads may still simultaneously instantiate a template after being granted on line 1 the locks they required. If this occurs, at least one thread in the group will notice the instantiation during the check from line 2.

In *Acquired(t,l,p)*, if thread t does not own l, then l is marked as acquired by t (line 2), the position p where l was acquired is saved (line 3), and the detector is notified (line 4). If t already holds l, a counter for reentrant locking of l is incremented (line 5).

In *Release(t,l)*, we decrement the counter associated with t (line 1). If l can be released (line 2), the owner and acquisition position for l must be reset (lines 4-5), the cycle detector notified (line 6), and the lock grant given to t removed by deregistering t from $acqPos[l]$ (line 7).

```
Request( t,l,p )
 1 lockGrantees[p] :=
                 lockGrantees[p] ⊎ {t}
 2 yieldCause := templateInstance(t,p)
 3 if yieldCause = null then
 4    events := events + [grant(t, l, p)]
 5 else
 6    RemoveGrant(t,p)
 7 return yieldCause

Acquired( t,l,p )
 1 if owner[l] ≠ t then
 2    owner[l] := t
 3    acqPos[l] := p
 4    events := events + [acquired(t, l, p)]
 5 nLockings[l] := nLockings[l] + 1

Release( t,l )
 1 nLockings[l] := nLockings[l] - 1
 2 if nLockings[l] = 0 then
 3    p := acqPos[l]
 4    owner[l] := null
 5    acqPos[l] := null
 6    events := events + [release(t, l)]
 7    RemoveGrant(t,p)

RemoveGrant( t,p )
 1 if 1_lockGrantees[p](t) = 1 then
 2    native_lock_for_write( (t,p) )
 3 lockGrantees[p] :=
                 lockGrantees[p] \ {t}
 4 if 1_lockGrantees[p](t) = 0 then
 5    foreach t' ∈ yielders[t, p] do
 6       native_lock(yieldLock[t'])
 7       yieldLock[t'].notify()
 8       native_unlock(yieldLock[t'])
 9    yielders[t,p] := ∅
10    native_unlock_for_write( (t,p) )
```

Fig. 5. The avoidance module.

In *RemoveGrant(t,p)*, if t is about to release all locks from p (line 1), lock (t, p) in write mode (line 2) to ensure consistency of the operations performed in the instrumentation, and remove the grant given to t for position p (line 3). If t released all locks from p (line 4), then wake up (notify) all yielders, i.e., threads that have (t, p) in their yield cause, and finally unlock (t, p) on line 12.

2.4 Detection

The detection module finds cycles in the RAG—deadlocks and avoidance-induced livelocks—and saves their templates to history. As illustrated in Figure 7, it periodically fetches and processes the notifications—RAG events and avoidance decisions—sent by the avoidance module, updates the RAG, and looks for cycles in the RAG. If cycles are found, their templates are computed and saved, after which the threads (or a subset thereof) are restarted. The detection module looks only for RAG cycles containing threads with pending lock requests, because only *request* events can introduce new cycles in the RAG (see proof in §3).

These actions are performed with a period of τ (e.g., 1 second). In principle, the value of τ does not affect correctness, given that it merely introduces a delay between the moment the program becomes deadlocked/livelocked and when this condition is detected. In practice, however, τ is a "knob" for tuning the tradeoff between computation overhead and recovery time: a higher τ reduces the CPU time consumed on updating the RAG and detecting cycles, while a lower τ leads to more prompt detection and, thus, faster recovery from deadlock/livelock, which improves the availability of the program.

waitCycles(v)
1 **foreach** $x \in rag$ **do**
2 x.color := white
3 endings := $\emptyset$
4 hasCycles(v, endings)
5 **return** $\bigcup_{x \in endings} waitChains(x, x)$

hasCycles($v, endings$)
1 **if** $v.color = black$ **then**
2 **return false**
3 **if** $v.color = grey$ **then**
4 endings := endings $\cup \{v\}$
5 **return true**
6 v.color := grey
7 **if** $\exists\ v \overset{*}{\rightarrow}_{r/g/h} v' \in rag$ s.t. $hasCycles(v', endings)$
 $\vee\ \forall\ v \overset{*}{\rightarrow}_y v_i \in rag\ :\ hasCycles(v_i, endings)$ **then**
8 **return true**
9 **else**
10 v.color := black
11 **return false**

template(C)
1 **return** $\{e.pos \mid e \in C \wedge (e = * \overset{*}{\leftarrow}_h * \vee e = * \overset{*}{\rightarrow}_y *)\}$

waitChains (v_1, v_2)
1 cycles := $\emptyset$
2 **if** $\exists e = v_1 \overset{*}{\rightarrow}_{r/g/h} v \in rag$ s.t. $v.color = grey$ **then**
3 **if** $v = v_2$ **then**
4   cycles := cycles $\cup \{\{e\}\}$
5 **else**
6   cycles := cycles $\cup\ (\bigcup_{c \in waitChains(v, v_2)} \{e\} \cup c)$
7 **if** $\forall v_1 \overset{*}{\rightarrow}_y v_i \in rag\ :\ v_i.color = grey$ **then**
8 **choose** $e = v_1 \overset{*}{\rightarrow}_y v_i \in rag$
9 **if** $v_i = v_2$ **then**
10   cycles := cycles $\cup \{\{e\}\}$
11 **else**
12   cycles := cycles $\cup\ (\bigcup_{c \in waitChains(v_i, v_2)} \{e\} \cup c)$
13 **return** cycles

Fig. 6. Helpers for the detection module.

The detection module uses two data structures, *rag* (the resource allocation graph) and *events* (the event queue used to receive RAG events and avoidance decisions from the avoidance module), which is initially empty. A RAG event can be *request(t, l, p)*, *yield(t, yieldCause)*, *grant(t, l, p)*, *acquired(t, l, p)* or *release(t, l)*, corresponding to adding a request edge, adding a yield edge, converting a request edge into a grant edge, converting a grant edge into a hold edge, and removing a hold edge, respectively. *requestingThreads* is the set of threads having pending lock requests.

As in the case of the avoidance module, we make use of helpers, defined in Figure 6. We only give a high-level description of these helpers, because the underlying algorithms are well-known. The *waitCycles, waitChains*, and *hasCycles* helpers are used for cycle detection, and *template(C)* is used to extract the template of a cycle C. *hasCycles(v, endings)* is easiest implemented using colored-DFS [11], in which all explored nodes are marked "grey" or "black", depending on whether they are involved in deadlocks/livelocks. *hasCycles(v, endings)* finds out whether v is involved in deadlocks/livelocks and returns the nodes in which the deadlocks/livelocks (if any) end. *waitCycles(v)* retrieves cycles involving v by exploring the "grey" nodes, starting from the endings returned by $hasCycles(v, endings)$.

A RAG node of type thread can have multiple edges emerging from it: up to one request edge and zero or more yield edges. Thus, a node can be involved in more than one cycle, which means $waitCycles(v)$ could return more than one cycle. However, it is not necessary to retrieve, save and avoid the templates of all cycles containing a particular node: to avoid an induced livelock, it is sufficient to avoid one of its correspond-

main_loop

1 **while** $stop = false$
2   sleep τ milliseconds
3   processEvents()
4   foundCycles := $\emptyset$
5   **foreach** $t \in requestingThreads$ **do**
6   foundCycles :=
      foundCycles $\cup$ waitCycles(t)
7   **if** $foundCycles \neq \emptyset$ **then**
8   **foreach** $c \in foundCycles$ **do**
9   history := history $\cup$ template(c)
10   restart program

processEvents()

1 **while** $events \neq \emptyset$ **do**
2   evt := events.head
3   events := events.tail
4   **switch** evt **do**
5   **case** $request(t, l, p)$
6   rag := rag $\cup$ $\{t \rightarrow_r l\}$
7   requestingThreads :=
        requestingThreads $\cup$ {t}
8   **case** $yield(t, yCause)$
9   rag := rag $\setminus$ $\{t \xrightarrow{*}_y t' | t \xrightarrow{*}_y t' \in rag\}$
        $\cup \{t \xrightarrow{p}_y t' |\ (t', p) \in yCause\}$
10   **case** $grant(t, l, p)$
11   rag := rag $\setminus$ $\{t \rightarrow_r l\} \cup \{t \xrightarrow{p}_g l\}$
12   **case** $acquired(t, l, p)$
13   rag := rag $\setminus$ $\{t \xrightarrow{p}_g l\} \cup \{t \xleftarrow{p}_h l\}$
14   requestingThreads :=
        requestingThreads $\setminus$ $\{t\}$
15   **case** $release(t, l, p)$
16   rag := rag $\setminus$ $\{t \xleftarrow{*}_h l\}$

Fig. 7. The detection module.

ing yield cycles. Thus, if a thread t is involved in an avoidance-induced livelock, it is enough for $waitCycles(t)$ to return just one yield cycle of the livelock.

The two core algorithms are shown in Figure 7. As long as it is not asked to stop, the cycle detector periodically (every τ msec) processes the notifications from the avoidance module (line 3), finds all cycles containing threads with pending requests (lines 4-6) and, if cycles found (line 7), adds the templates of the detected cycles to the history (lines 8-9) and recovers the program (line 10).

3 Soundness and Completeness

In this section we outline the proof of the deadlock immunity algorithm's soundness and refer the reader to [9] for the details. The proof shows soundness by demonstrating *safety*, i.e., that the algorithm indeed avoids previously-seen deadlocks, and *liveness*, i.e., that all threads will eventually make progress. The algorithm is also proven to be *eventually complete*, i.e., that it eventually detects and avoids all cycles, i.e., deadlocks and avoidance-induced livelocks.

In proving soundness and completeness of our algorithm, we make the following assumptions:

- All avoidance routines (*Request*, *Acquired*, *Release*) are thread-safe. This depends on implementation: in the synchronized version of the algorithm [9], atomicity (and therefore thread-safety) is ensured by a global lock. In the lock-free version (Figure 5), thread-safety (consistency) is preserved via the additional check performed in the *Request* routine.
- The number of threads in a program and the number of possible program positions (i.e., program size) are finite.
- All existing deadlock bugs in a program and avoidance-induced livelocks eventually manifest.
- All critical sections eventually terminate, except in cases of deadlock or livelock.
- The native thread scheduler is fair.
- All lock/unlock statements performed in the program are instrumented as shown in Figure 3.
- The position within the program of lock operations previously involved in deadlocks or livelocks does not change from one execution to another, i.e., templates are execution-independent. This assumption could be invalidated by a program upgrade or patch.

We first prove completeness of the cycle detection algorithm. The detection module looks only for cycles containing threads with pending requests, so we first prove that, indeed, only the *request* events can introduce new cycles in the RAG; we do this by proving that the remaining RAG events — *acquired* and *release* — cannot introduce new cycles in the RAG. Second, we prove that the detection module detects all cycles required to perform avoidance.

To prove safety (i.e., that we achieve deadlock immunity), we split *history* into its version before the current execution ($history_{old}$) and the additions made

during the current run ($history_{new}$), i.e., $history = history_{old} \cup history_{new}$. We then prove that the following invariant is maintained: $\forall T \in history_{old} : instance(T, \emptyset) = null$, i.e., the algorithm avoids the instantiation of all templates from $history_{old}$. Then, we prove the invariant $\forall C \in waitCycles(t) : template(C) \notin history_{old}$, i.e., no newly-detected cycle has its template in $history_{old}$. Finally, we prove the invariant $history_{new} \cap history_{old} = \emptyset$, i.e., templates do not repeat in different runs.

To prove completeness (i.e., that an application instrumented with our algorithm eventually develops immunity against all possible deadlocks and avoidance-induced livelocks), we first prove that the number of possible templates is finite. Then we prove that every program instrumented according to Figure 3, after a finite number of restarts, eventually reaches a point beyond which all subsequent executions become free of deadlocks and avoidance-induced livelocks. Finally, we prove that the deadlock immunity algorithm preserves liveness, i.e., all lock requests are eventually granted by our algorithm. The detailed proofs, for both synchronized and lock-free implementations, are presented in [9].

4 Complexity Analysis

In this section, we discuss the theoretical complexity of the algorithm, and in the next section we analyze its empirically measured performance. For conciseness, we highlight here the main results of the complexity analysis and direct the reader to [9] for the details.

For the avoidance module, assume an immunized program is running with a history containing N templates, containing N_P positions on average, and for each position p in a template, $|lockGrantees[p]| = N_G$, on average. Let N_W be the average number of yields (waits) performed by a thread before being granted a lock, and C_{wait} the cost of a *wait()* system call. Let $|yielders[t, p]| = N_Y$ on average, and C_{notify} the cost of a *notify()* call. The complexity of the *Request* operation, which is the most expensive in the avoidance module, is $O(N_W \cdot (C_{wait} + N \cdot (N_P + (N_P - 1)! \cdot N_G^{N_P - 1})) + N_Y \cdot C_{notify})$.

Note that we expect N_P to be small in practice, on the order of $N_P = 2...4$, since deadlocks normally involve no more than 4 threads. This is justified by the fact that, for a group of threads to deadlock, first they have to simultaneously perform nested locks, and then to perform the inner locks in such a way that a circular wait occurs. As far as induced livelocks are concerned, note that an avoidance-induced livelock is a conjunction of yield cycles (§2.4) and a yield cycle will mirror one of the already-encountered deadlocks. Thus, since N_P is small, we expect that the exponential term in N_P to not be dominant in practice.

For the detection module, consider $RAG = [V, E]$, with $|requestingThreads| = N_R$, on average; say we have on average N_E events in the event queue. The complexity of the detection module is $O(N_E + N_R \cdot (|V| + |E|))$, because every event is processed in constant time, and we use the optimal colored DFS algorithm (§2.4) for detecting RAG cycles starting from each thread in *requestingThreads*.

For the complete derivation of the two modules' complexities, please see [9].

5 Evaluation

In order to verify the practicality of deadlock immunity, we built a prototype of the deadlock immunity algorithm for Java programs. After describing the implementation and experimental setup, we evaluate effectiveness (§5.1), performance overhead in a real application server (§5.2), as well as discuss the effect of false positives (§5.3). The interested reader can additionally find in [9] the evaluation of performance overhead using a lock-intensive microbenchmark.

In our implementation, we rely on AspectJ [1], an aspect-oriented compiler, to instrument Java programs. We instrument the bytecode-level calls to *monitorenter* (corresponding to the start of Java `synchronized` blocks) and to *monitorexit* (corresponding to the end of `synchronized` blocks). The instrumentation is embodied by advices that capture lock requests (*before-monitorenter* advice), lock acquisitions (*after-monitorenter* advice), and lock releases (*before-monitorexit* advice). Lock positions are represented as *file:line* strings corresponding to the line of code containing the statement in the source file.

The experiments reported here were run on computers with 2 x 4-core Intel Xeon E5310 1.6GHz CPUs, 4GB RAM, WD-1500 hard disk, 2 NetXtreme II GbE interfaces, interconnected by a dedicated GbE switch, running Linux Fedora Core 7 with kernel 2.6.20, Java HotSpot Server VM 1.6.0, and Java SE 1.6.0.

5.1 Effectiveness

The first question we wanted to answer was whether the proposed approach will avoid deadlocks in real applications. We scoured bug reports for the MySQL database system [13] and, of the various reports of deadlocks or hangs, we were able to reproduce four (#21427, #14972, #31126, and #17709). They all occur in the connector that allows Java programs to interact with the database engine. We wrote test programs that deterministically reproduce these bugs. The immunized test programs detected each deadlock the first time it occurred, saved the template to history, and successfully avoided it in subsequent executions. Since extensive repeated runs never deadlocked, we cite this as an empirical proof point that the immunized programs had developed immunity against the deadlock bugs. MySQL users encountering these bugs face the option of waiting for the MySQL team to fix them, or to use an immunization tool right away.

While in some cases deadlocks can be eliminated by fixing the root cause, in other cases this is not a reasonable option. For example, a number of synchronized classes in the Java runtime environment can cause deadlocks in the applications that call them. Consider two vectors v_1, v_2 in a multithreaded program—since `Vector` is a synchronized class, programmers allegedly need not be concerned by concurrent access to vectors. However, if one thread wants to add all elements of v_2 to v_1 via $v_1.addAll(v_2)$, while another thread concurrently does the reverse via $v_2.addAll(v_1)$, the program can deadlock, because underneath the covers, the JDK locks v_1 then v_2 in one thread, and v_2 then v_1 in the other thread. This is a general problem for all synchronized `Collection` classes

in the JDK, of which there are dozens. It is difficult for developers to knowingly steer clear of deadlocks resulting from the implementation of an opaque interface, and deadlocks hidden underneath the runtime interface are some of the most insidious. At the same time, it is tenuous to precisely document in this interface all possible usage scenarios that could lead to deadlock.

We wrote test cases for six such deadlock traps and immunized them. After encountering the respective deadlocks for the first time, all subsequent executions were free of deadlocks. This resolution requires no programmer intervention and no JDK modifications.

5.2 Performance Overhead in Real Applications

We applied our immunization tool to JBoss [8], a J2EE application server. JBoss is a piece of middleware that allows enterprise and Web applications to be written in Java, with all complexities of transactions, persistence, group communication, replication, etc. being handled transparently on the applications' behalf. Behind virtually every e-commerce Web site today lies an application server. JBoss is one of the most widely used J2EE servers and, at over 350,000 lines of code (excluding comments), it is likely one of the largest systems written in Java.

To benchmark performance on the immunized JBoss, we used the RUBiS benchmark [14], an online auction application modeled after eBay. In our measurements, we used the servlet version of RUBiS with the browse workload.

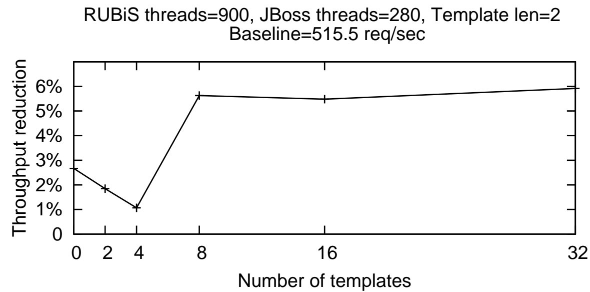

Fig. 8. JBoss throughput drop at 280 threads.

We ran JBoss+RUBiS, the corresponding MySQL database tier, and the RUBiS clients on separate nodes. We operated the auction site just below its saturation point (900 RUBiS client threads); below and above this level we found the impact of our algorithm to be virtually unmeasurable. The JBoss console reported that 280 threads were running inside JBoss. In Figure 8, we show the measured reduction in throughput introduced by our immunization tool for history sizes ranging from 0 to 32 templates of length 2. The templates are random combinations of program locations at which JBoss performs synchronization.

The conclusion is that the cost of immunity against up to 32 templates is a penalty of $< 6\%$ in request throughput on an e-commerce workload, which suggests that our approach offers an efficient deadlock avoidance solution even for the largest production systems.

5.3 Effects of False Positives

An important question is whether, by pruning executions that might lead to deadlock, the immunity algorithm is not being too conservative. Said otherwise, paths that once led to deadlock may not deterministically lead to deadlock. For example, if wrapper methods are used to perform locking (instead of direct calls to *native_lock*), all the lock positions may end up being the same (the location of the lock statement in the wrapper) resulting in overly aggressive serialization of the threads. Exaggerated conservativeness might lead to performance degradation by reducing parallelism, or even to the elimination of some functionality through the persistent avoidance of deadlock-prone executions paths.

While we do not know yet how to quantify the true effect of false positives, or how to measure them directly, our initial experimentation indicates that functionality does not get eliminated for some of the largest programs. Moreover, the low performance overhead suggests that, for the systems we measured, even if loss of parallelism influences negatively the performance, it does so to a small degree. Nevertheless, further experimentation is required, as well as further theoretical analysis of the false positives introduced by deadlock immunity.

The way to reduce false positives is by improving the precision of avoidance. We are currently experimenting with storing more contextual information in the template, such as a suffix of the call path that led to the deadlock-forming lock statements. The added information defines more precisely the particular execution that led to the observed deadlock, thus allowing the algorithm to better distinguish an execution heading for a similar deadlock from one that will not deadlock. Such increase in precision (reduction in false positive rate) will hurt, however, the convergence rate—the less general the avoided templates are, the longer it takes to develop immunity against all the existing deadlock bugs. Said differently, increased precision makes the "eventual" in eventual completeness longer.

We are also exploring techniques for dynamically adjusting (learning) the ideal length of call path suffixes, in order to achieve an optimal precision vs. generality tradeoff. We are also considering saving the sequence of lock positions traversed along the call paths.

Another area we wish to explore further is the use of static analysis and symbolic execution as a way to complement deadlock immunity. In particular, we want to use look-ahead static analysis to help predict deadlocks that are similar to ones we have already seen. Using bounded symbolic execution, on the order of a few instructions ahead of the current state, we can identify unsafe states without having to actually reach them. Performing such analyses at runtime could harness the "free parallelism" made available by the advent of multi-core CPUs. We could also use static analysis for detecting lock statements that would never lead to deadlocks. We could avoid instrumenting these lock statements, thus reducing the intrusiveness and therefore the overhead of our algorithm.

Our empirical evaluation indicates that the deadlock immunity approach is effective at developing immunity against deadlocks, scales to real programs with

hundreds of thousands of lines of code, and introduces low overheads on a real e-commerce workload. There are still some questions to be answered with respect to the effect of false positives, and this is the subject of future work.

6 Conclusion

We described an algorithm for imparting deadlock immunity to software systems, that helps avoid deadlocks with no assistance from programmers or users. We showed that, once an immunized program encounters a deadlock, it avoids in all future executions all deadlocks with that same template. We proved the algorithm's soundness and eventual completeness; we also showed empirically that the algorithm is effective against deadlocks in real software like MySQL JDBC and Java JDK. Preliminary results indicate that the algorithm scales gracefully to large software systems: in JBoss, a $>$350 KLOC application server with 280 threads, worst-case overhead introduced by immunization was a drop of $<$6% in request throughput while avoiding up to 32 deadlock templates.

While pure deadlock avoidance is undecidable, deadlock immunity is decidable, to the extent imposed by the definition of similarity between deadlocks. Our technique builds upon and complements prior work by addressing the challenges of real systems: code size scalability, correctness in the face of program I/O, and performance overhead. We believe deadlock immunity is a practical way to eventually run production systems deadlock-free despite the deadlock bugs that lurk within.

References

1. Aspectj. `http://www.eclipse.org/aspectj`, 2007.
2. P. Boronat and V. Cholvi. A transformation to provide deadlock-free programs. In *Intl. Conf. on Computational Science*, 2003.
3. D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *19th ACM Symp. on Operating Systems Principles*, 2003.
4. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conf. on Programming Language Design and Implementation*, 2002.
5. A. N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7), 1969.
6. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *20th Intl. Symposium on Computer Architecture*, 1993.
7. Java pathfinder. http://javapathfinder.sourceforge.net/doc/What_can_be_checked_with_JPF.html, 2007.
8. JBoss. `http://jboss.org`.
9. H. Jula and G. Candea. A scalable, sound, eventually-complete algorithm for deadlock immunity. Technical Report EPFL-DSLAB-2007-002, EPFL, Lausanne, Switzerland, 2007. `http://dslab.epfl.ch/pubs/dimmunix-algo`.
10. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for java programs. In *Formal Methods in System Design*, 2004.

11. D. E. Knuth. *The Art of Computer Programming*, volume III: Sorting and Searching. Addison-Wesley, 1998.
12. T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *USENIX Annual Technical Conference*, 2005.
13. MySQL bug database. http://bugs.mysql.com/.
14. RUBiS. http://rubis.objectweb.org, 2007.
15. M. Singhal. Deadlock detection in distributed systems. *IEEE Computer*, 22(11), 1989.
16. A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *19th European Conference on Object-Oriented Programming*, 2005.
17. F. Zeng and R. P. Martin. Ghost locks: Deadlock prevention for Java. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.