Deadlock Immunity: Enabling Systems To Defend Against Deadlocks

Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea
School of Computer & Communication Sciences
EPFL (Lausanne, Switzerland)

Abstract

Deadlock immunity is a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of that and similar deadlocks. We describe a technique that enables programs to automatically gain such immunity without assistance from programmers or users. We implemented the technique for both Java and POSIX threads and evaluated it with several real systems, including MySQL, JBoss, SQLite, Apache ActiveMQ, Limewire, and Java JDK. The results demonstrate effectiveness against real deadlock bugs, while incurring modest performance overhead and scaling to 1024 threads. We therefore conclude that deadlock immunity offers programmers and users an attractive tool for coping with elusive deadlocks.

1 Introduction

Writing concurrent software is one of the most challenging endeavors faced by software engineers, because it requires careful reasoning about complex interactions between concurrently running threads. Many programmers consider concurrency bugs to be some of the most insidious and, not surprisingly, a large number of bugs are related to concurrency [16].

The simplest mechanism used for synchronizing concurrent accesses to shared data is the mutex lock. When threads do not coordinate correctly in their use of locks, deadlock can ensue—a situation whereby a group of threads cannot make forward progress, because each one is waiting to acquire a lock held by another thread in that group. Deadlock immunity helps develop resistance against such deadlocks.

Avoiding the introduction of deadlock bugs during development is challenging. Large software systems are developed by multiple teams totaling hundreds to thousands of programmers, which makes it hard to maintain the coding discipline needed to avoid deadlock bugs. Testing, although helpful, is not a panacea, because exercising all possible execution paths and thread interleavings is still infeasible in practice for all but toy programs.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field. Dependencies on deadlock-prone third party libraries or runtimes can deadlock programs that are otherwise correct. Upgrading these libraries or runtimes can introduce new executions that were not covered by prior testing. Furthermore, modern systems accommodate extensions written by third parties, which can introduce new deadlocks into the target systems (e.g., Web browser plugins, enterprise Java beans).

Debugging deadlocks is hard—merely seeing a deadlock happen does not mean the bug is easy to fix. Deadlocks often require complex sequences of low-probability events to manifest (e.g., timing or workload dependencies, presence or absence of debug code, compiler optimization options), making them hard to reproduce and diagnose. Sometimes deadlocks are too costly to fix, as they entail drastic redesign. Patches are error-prone: many concurrency bug fixes either introduce new bugs or, instead of fixing the underlying bug, merely decrease the probability of occurrence [16].

We expect the deadlock challenge to persist and likely become worse over time: On the one hand, software systems continue getting larger and more complex. On the other hand, owing to the advent of multi-core architectures and other forms of parallel hardware, new applications are written using more threads, while existing applications achieve higher degrees of runtime concurrency. There exist proposals for making concurrent programming easier, such as transactional memory [8], but issues surrounding I/O and long-running operations make it difficult to provide atomicity transparently.

In this paper, we introduce the notion of deadlock immunity—a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of similar deadlocks. We describe Dimmunix, a tool for developing deadlock immunity with no assistance from programmers or users. The first time a deadlock pattern manifests, Dimmunix automatically captures its signature and subsequently avoids entering the same pattern. Signatures can be proactively distributed to immunize users who have not yet encountered that deadlock. Dimmunix can be used by customers to defend against deadlocks while waiting for a vendor patch, and by software vendors as a safety net.

In the rest of the paper we survey related work (§2), give an overview of our system (§3-§4), present details of our technique (§5), describe three Dimmunix implementations (§6), evaluate them (§7), discuss how Dimmunix can be used in practice (§8), and conclude (§9).

2 Related Work

There is a spectrum of approaches for avoiding deadlocks, from purely static techniques to purely dynamic ones. Dimmunix targets general-purpose systems, not real-time or safety-critical ones, so we describe this spectrum of solutions keeping our target domain in mind.

Language-level approaches [3, 15] use powerful type systems to simplify the writing of lock-based concurrent programs and thus avoid synchronization problems altogether. This avoids runtime performance overhead and prevents deadlocks outright, but requires programmers to be disciplined, adopt new languages and constructs, or annotate their code. While this is the ideal way to avoid deadlocks, programmers' human limits have motivated a number of complementary approaches.

Transactional memory (TM) [8] holds promise for simplifying the way program concurrency is expressed. TM converts the locking order problem into a thread scheduling problem, thus moving the burden from programmers to the runtime, which we consider a good tradeoff. There are still challenges with TM semantics, such as what happens when programmers use large atomic blocks, or when TM code calls into non-TM code or performs I/O. Performance is still an issue, and [14] shows that many modern TM implementations use lock-based techniques to improve performance and are subject to deadlock. Thus, we believe TM is powerful, but it cannot address all concurrency problems in real systems.

Time-triggered systems [13] and statically scheduled real-time systems [22] perform task synchronization before the program runs, by deciding schedules a priori based on task parameters like mutual-exclusion constraints and request processing time. When such parameters are known a priori, the approach guarantees safety and liveness; however, general-purpose systems rarely have such information ahead of time. Event-triggered real-time systems are more flexible and incorporate a priori constraints in the form of thread priorities; protocols like priority ceiling [20], used to prevent priority inversion, conveniently prevent deadlocks too. In general-purpose systems, though, even merely assigning priorities to the various threads is difficult, as the threads often serve a variety of purposes over their lifetime.

Static analysis tools look for deadlocks at compile time and help programmers remove them. ESC [7] uses a theorem prover and relies on annotations to provide knowledge to the analysis; Houdini [6] helps generate some of these annotations automatically. [5] and [21] use flow-sensitive analyses to find deadlocks. In Java JDK 1.4, the tool described in [21] reported 100,000 potential deadlocks and the authors used unsound filtering to trim this result set down to 70, which were then manually reduced to 7 actual deadlock bugs. Static analyses run fast, avoid runtime overheads, and can help prevent deadlocks, but when they generate false positives, it is ultimately the programmers who have to winnow the results. Developers under pressure to ship production code fast are often reticent to take on this burden.

Another approach to finding deadlocks is to use model checkers, which systematically explore all possible states of the program; in the case of concurrent programs, this includes all thread interleavings. Model checkers achieve high coverage and are sound, but suffer from poor scalability due to the "state-space explosion" problem. Java PathFinder, one of the most successful model checkers, is restricted to applications up to $\sim$10 KLOC [10] and does not support native I/O libraries. Real-world applications are large (e.g., MySQL has $>$1 MLOC) and perform frequent I/O, which restricts the use of model checking in the development of general-purpose systems.

Further toward the dynamic end of the spectrum, [17] discovers deadlocks at runtime, then wraps the corresponding parts of the code in one "gate lock"; in subsequent executions, the gate lock must be acquired prior to entering the code block. This approach is similar to [2], except that the latter detects deadlocks statically, thus exhibiting more false positives than [17]. In a dual approach to these two, [23] modifies the JVM to serialize threads' access to lock sets (instead of program code) that could induce deadlocks. Dimmunix shares ideas with these dynamic approaches, but uses added context information to achieve finer grain avoidance and considerably fewer false positives (as will be seen in §7.3).

Finally, there are purely dynamic approaches, like Rx [18]. Upon deadlock, Rx can roll back a program to a checkpoint and retry the execution in a modified environment; new timing conditions could prevent deadlock reoccurrence. However, Rx does not (and was not meant to) build up resistance against future occurrences of the deadlock, so the system as a whole does not "improve" itself over time. The performance overhead induced by repeated re-executions can be unpredictable (in the extreme case of a deterministic deadlock, Rx cannot go past it) and retried executions cannot safely span I/O. In contrast, Dimmunix actively prevents programs from re-encountering previously seen deadlock patterns.

Deadlock immunity explores a new design point on this spectrum of deadlock avoidance solutions, combining static elements (e.g., control flow signatures) with dynamic approaches (e.g., runtime steering of thread schedules). This combination makes Dimmunix embody new tradeoffs, which we found to be advantageous when avoiding deadlocks in large, real, general-purpose systems.

3 System Overview

Programs augmented with a deadlock immunity system develop "antibodies" matching observed deadlock patterns, store them in a persistent history, and then alter future thread schedules in order to avoid executing patterns like the ones that were previously seen. With every new deadlock pattern encountered by the program, its resistance to deadlocks is improved.

When buggy code runs and deadlocks, we refer to an approximate suffix of the call flow that led to deadlock as a *deadlock pattern*—this is an approximation of the control flow that turned the bug into a deadlock. A runtime instantiation of a deadlock pattern constitutes a *deadlock occurrence*. Thus, a deadlock bug begets a deadlock pattern, which in turn begets a deadlock occurrence. One deadlock pattern can generate a potentially unbounded number of runtime deadlock occurrences, e.g., because lock identities vary across different manifestations of the same deadlock pattern. Dimmunix automatically avoids previously seen deadlock patterns, in order to reduce the number of deadlock occurrences. To recognize repeated deadlock patterns, it saves "fingerprints" of every new pattern; we call these *deadlock signatures*. Runtime conditions can cause a deadlock pattern to not always lead to deadlock, in which case avoiding the pattern results in a false positive (more details in §5.5).

The Dimmunix architecture is illustrated in Figure 1. There are two parts: *avoidance instrumentation code* prevents reoccurrences of previously encountered deadlocks and a *monitor thread* finds and adds deadlock information to the persistent *deadlock history*. Avoidance code can be directly instrumented into the target binary or can reside in a thread library. This instrumentation code intercepts the lock/unlock operations in target programs and transfers control to Dimmunix any time lock or unlock is performed; Dimmunix itself runs within the address space of the target program.

At the beginning of a lock call, a *request* method in the avoidance instrumentation decides whether to allow the lock operation to proceed. This decision can be *GO*, if locking is allowed, or *YIELD*, if not. In the case of a yield, the thread is forced by the instrumentation code to yield the CPU, and the lock attempt is transparently retried later. When the program finally acquires the lock, the instrumentation code invokes *acquired*. Unlock operations are preceded by a call to *release*.

The avoidance code enqueues request, go, yield, acquired, and release events onto a lock-free queue that is drained by the monitor thread. The monitor wakes up periodically and updates a resource allocation graph (RAG) according to received events, searches for deadlock cycles, and saves the cycle signatures to the persistent history. The delay between the occurrence of a deadlock and its detection by the asynchronous monitor has an upper bound determined by the wakeup frequency.

Dimmunix uses the RAG to represent a program's synchronization state. Most edges are labeled with the call stack of the origin thread, representing an approximation of that thread's recent control flow. When a deadlock is found, Dimmunix archives a combination of the involved threads' stacks into a deadlock signature.

Avoiding deadlocks requires anticipating whether the acquisition of a lock would lead to the instantiation of a signature of a previously-encountered deadlock pattern. For a signature with call stacks $\{S_1, S_2, ...\}$ to be instantiated, there must exist threads $T_1, T_2, ...$ that either hold or are allowed to wait for locks $L_1, L_2, ...$ while having call stacks $S_1, S_2, ...$ An instantiation of a signature captures the corresponding thread-lock-stack bindings: $\{(T_1, L_1, S_1), (T_2, L_2, S_2), ...\}$.

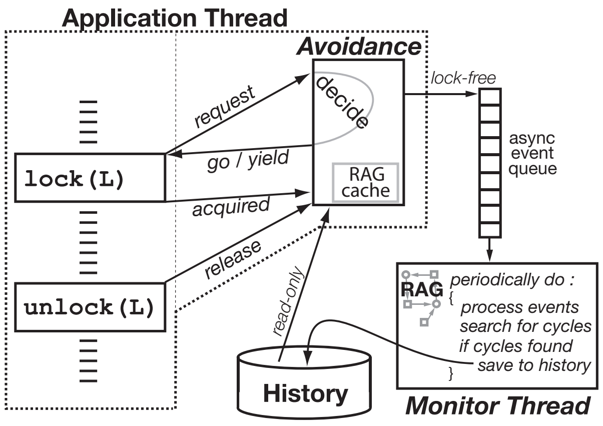


Figure 1: Dimmunix architecture.

The way in which a deadlocked program *recovers* is orthogonal to Dimmunix and, in practice, would most likely be done via restart. Dimmunix can provide a hook in the monitor thread for programs to define more sophisticated deadlock recovery methods; the hook can be invoked right after the deadlock signature is saved. For instance, plugging Rx's checkpoint/rollback facility [18] into this application-specific deadlock resolution hook could provide application-transparent deadlock recovery.

Any scheduling-based approach to deadlock avoidance faces the risk of occasionally reaching starvation states, in which threads are actively yielding, waiting in vain for synchronization conditions to change. In Dimmunix, this is handled automatically: when induced starvation occurs, Dimmunix saves the signature of the starvation state, breaks the starvation by canceling the yield for the starved thread holding most locks, and allows the freed thread to pursue its most recently requested lock. Dimmunix will subsequently be able to avoid entering this same starvation condition again.

We recommend Dimmunix for general-purpose systems, such as desktop and enterprise applications, server software, etc.; in real-time systems or safety-critical systems, Dimmunix can cause undue interference (§5.7). Systems in which even the very first occurrence of a deadlock cannot be tolerated are not good targets for Dimmunix; such systems require more programmer-intensive approaches if they want to run deadlock-free.

Dimmunix can be used by software vendors and end users alike. Faced with the current impossibility of shipping large software that is bug-free, vendors could instrument their ready-to-ship software with Dimmunix and get an extra safety net. Dimmunix will keep users happy by allowing them to use the deadlock-prone system while developers try to fix the bugs. Also, users frustrated with deadlock-prone applications can use Dimmunix on their own to improve their user experience. We do not advocate deadlock immunity as a replacement for correct concurrent programming—ultimately, concurrency bugs need to be fixed in the design and the code—but it does offer a “band-aid” with many practical benefits.

4 An Example

We now illustrate how Dimmunix works with an example of two deadlock-prone threads.

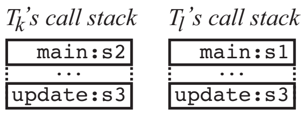
```
     main {
s1:    update(A,B)
        ...
s2:    update(B,A)
     }

     update(x,y) {
s3:    lock(x)
s4:    lock(y)
        ...
       unlock(y)
       unlock(x)
     }
```

The pseudocode on the right accesses two global shared variables *A* and *B*, each protected by its own mutex. s_1, s_2,... are the equivalent of goto labels. For simplicity, we assume there are no pointers.

If two different threads T_i and T_j run the code concurrently, they may attempt to lock *A* and *B* in opposite order, which can lead to deadlock, i.e., if T_i executes statement s_1 and then s_3, while T_j executes s_2 followed by s_3. The call flow can be represented abstractly as $<T_i{:}[s_1,s_3],T_j{:}[s_2,s_3]>$. There exist other execution patterns too, such as $<T_i{:}[s_1,s_3],T_j{:}[s_1,s_3]>$ that do not lead to deadlock.

The first time the code enters a deadlock, Dimmunix will see it as a cycle in the RAG and save its signature based on the threads’ call stacks at the time of their lock acquisitions. When T_i acquires the lock on *A*, the return addresses on its stack are $[s_1,s_3]$, because it called *update()* from s_1 and *lock()* from s_3; similarly, when T_j acquires the lock on *B*, T_j’s call stack is $[s_2,s_3]$. Upon deadlock, the $<T_i{:}[s_1,s_3],T_j{:}[s_2,s_3]>$ call pattern is saved to history as deadlock signature $\{[s_1,s_3],[s_2,s_3]\}$. Signatures do not include thread or lock identities, thus making them portable from one execution to the next.

Consider now a subsequent run of the program, in which some thread T_k executes s_2 followed by s_3, acquires the lock on *B*, and then some other thread T_l executes s_1 and then makes the call to *lock(x)* in statement s_3, in order to acquire *A*. If Dimmunix were to allow this lock operation to proceed, this execution could deadlock. Dimmunix infers the deadlock danger by matching the threads’ call stacks (shown on right) to the signature.



Given that there is a match, Dimmunix decides to force T_l to yield until the lock that T_k acquired at s_3 is released. After *B* is released, T_l is allowed to lock *A* and proceed. In this way, the program became immune to the deadlock pattern $\{[s_1,s_3],[s_2,s_3]\}$.

Note that Dimmunix does not merely serialize code blocks, as would be done by wrapping *update()* in a Java *synchronized{...}* block or as was done in prior work. For instance, on the above example, [17] would add a “gate lock” around the code for *update()* and serialize all calls to it, even in the case of execution patterns that do not lead to deadlock, such as $\{[s_1,s_3],[s_1,s_3]\}$. [23] would add a “ghost lock” for *A* and *B*, that would have to be acquired prior to locking either *A* or *B*.

Dimmunix achieves finer grain avoidance by (a) using call path information to distinguish between executions—of all paths that end up at s_3, Dimmunix avoids only those that executed a call path previously seen to lead to deadlock—and (b) using runtime information about which locks are held by other threads to avoid these paths only when they indeed seem dangerous.

5 Deadlock and Starvation Avoidance

We now present selected details of Dimmunix: the core data structure (§5.1), detection (§5.2), construction of signatures (§5.3), runtime avoidance of archived signatures (§5.4), calibration of signature matching precision (§5.5), auxiliary data structures (§5.6), and a synopsis of Dimmunix’s properties and limitations (§5.7).

5.1 Capturing Synchronization State

Dimmunix conceptually uses a resource allocation graph (RAG) to represent the synchronization state of a program. In practice, the RAG is built on top of several performance-optimized data structures (details in §5.6).

The RAG is a directed graph with two types of vertices: *threads T* and *locks L*. There are three types of edges connecting threads to locks and one type of edges connecting threads to threads. *Request edges* indicate that a thread *T* wants to acquire lock *L*, *allow edges* indicate that thread *T* has been allowed by Dimmunix to block waiting for *L*, and *hold edges* indicate that *T* has acquired and presently holds lock *L*. If the avoidance code decides to not allow a thread *T*’s lock request, it will force *T* to yield. This state is reflected in the RAG by a *yield edge* connecting thread *T* to T', indicating that *T* is currently yielding because of locks that T' acquired or was allowed to wait for. Dimmunix reschedules the paused thread *T* whenever lock conditions change in a way that could enable *T* to acquire the desired lock. Figure 2 illustrates a subgraph of a real RAG.

A hold edge, like $L_7 \xrightarrow{S_y} T_{13}$, always points from a lock to a thread and indicates that the lock is held by that thread; it also carries as label a simplified version S_y of the call stack that the thread had at the time it acquired the lock. A yield edge, like $T_{13} \xrightarrow{S_x} T_{22}$, always points from a thread to another thread; it indicates that T_{13} has been forced to yield because T_{22} acquired a lock with call stack S_x that would cause T_{13} to instantiate a signature if it was allowed to execute lock().

In order to support reentrant locks, as are standard in Java and available in POSIX threads, the RAG is a multi-

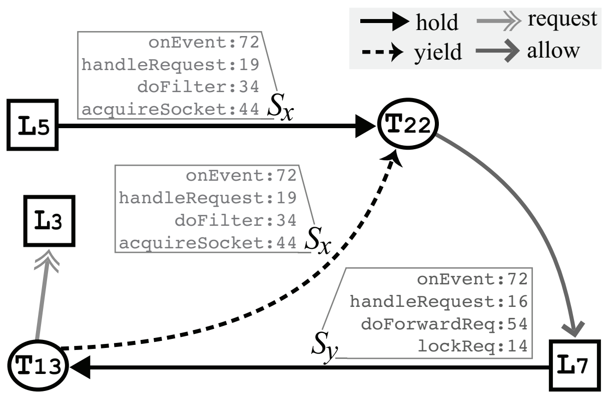


Figure 2: Fragment of a resource allocation graph.

set of edges; it can represent locks that are acquired multiple times by the same holder and, thus, have to be released as many times as acquired before becoming available to other threads.

Finally, the RAG does not always provide an up-to-date view of the program's synchronization state, since it is updated lazily by the monitor. This is acceptable for cycle detection, but the avoidance code needs some information to always be current, such as the mapping from locks to owners. Therefore, the avoidance instrumentation also maintains a simpler "cache" of parts of the RAG (in the form of simplified lock-free data structures) to make correct yield/allow decisions.

5.2 Detecting Deadlocks and Starvation

The monitor thread wakes up every τ milliseconds, drains all events from the lock-free event queue, and updates the RAG according to these events; then it searches for cycles. The monitor only searches for cycles involving edges that were added by the most recently processed batch of events; there cannot be new cycles formed that involve exclusively old edges. The value of τ is configurable, and the right choice depends on the application at hand; e.g., in an interactive program, $\tau = 100$ milliseconds would be reasonable.

Events enqueued by the same thread are correctly ordered with respect to each other. As far as other threads are concerned, we need to ensure a partial ordering that guarantees a *release* event on lock L in thread T_i will appear in the queue prior to any other thread T_j's *acquired* event on L. Given that the runtime (e.g., JVM) completes lock(L) in T_j strictly after it completed unlock(L) in T_i, and the *release* event in T_i precedes the unlock(L), and the *acquired* event in T_j follows the lock(L), the required partial ordering is guaranteed.

There are two cycle types of interest: deadlock cycles and yield cycles. A thread T is in a *deadlock* iff T is part of a cycle made up exclusively of hold, allow, and request edges—this is similar to deadlock cycles in standard wait-for graphs. Yield cycles are used to detect induced starvation. Any yield-based deadlock avoidance technique runs the risk of inducing one or more threads to starve due to yielding while waiting for a thread that is blocked. Thus, any dynamic scheduling-based deadlock avoidance approach must also avoid induced starvation.

Consider Figure 3, which shows a RAG in which a starvation state has been reached (nodes and edges not used in the discussion are smaller and dotted; call stack edge labels are not shown). For T_1 to be starved, both its yield edges $T_1 \rightarrow T_2$ and $T_1 \rightarrow T_3$ must be part of cycles, as well as both of T_4's yield edges. If the RAG had only the $(T_1, T_2, ..., T_1)$ and $(T_1, T_3, L, T_4, T_6, ..., T_1)$ cycles, then this would not be a starvation state, because T_4 could evade starvation through T_5, allowing T_1 to eventually evade through T_3. If, as in Figure 3, cycle $(T_1, T_3, L, T_4, T_5, ..., T_1)$ is also present, then neither thread can make progress.

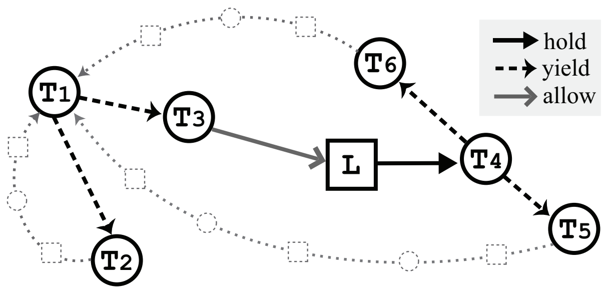


Figure 3: Starved threads in a yield cycle.

We say that a thread T is involved in an *induced starvation* condition iff T is part of a yield cycle. A yield cycle is a subgraph of the RAG in which all nodes reachable from a node T through T's yield edges can in turn reach T. The graph in Figure 3 is a yield cycle.

Dimmunix uses cycle detection as a universal mechanism for detecting both deadlocks and induced starvation: when the monitor encounters a yield cycle in the RAG, it saves its signature to the history, as if it was a deadlock. Dimmunix uses the same logic to avoid both deadlock patterns and induced starvation patterns.

5.3 From Cycles to Signatures

The signature of a cycle is a multiset containing the call stack labels of all hold edges and yield edges in that cycle. The signature must be a multiset because different threads may have acquired different locks while having the same call stack, by virtue of executing the same code.

Figure 2 shows a simple yield cycle $(T_{13}, T_{22}, L_7, T_{13})$, whose signature is $\{S_x, S_y\}$. The signature is archived by the monitor into the history that persists across program restarts.

A signature contains one call stack per thread blocked in the detected deadlock or starvation. The number of threads involved is bounded by the maximum number of threads that can run at any given time, so a signature can have no more than that number of call stacks. A call stack is always of finite size (usually set by the OS or thread library); thus, the size of a signature is finite. Signatures are essentially permutations of "instruction addresses" in the code, and there is a finite number of instructions in an application; given that duplicate signatures are disallowed, the signature history cannot grow indefinitely.

After saving a deadlock signature, the monitor can wait for deadlock recovery to be performed externally, or it can invoke an application-specific deadlock resolution handler. After saving a starvation signature, the monitor can break the starvation as described in §3.

Signatures abstract solely the call flow that led to deadlock or starvation; no program data (such as lock/thread IDs or values of variables) are recorded. This ensures that signatures preserve the generality of a deadlock pattern and are fully portable from one execution to the next. Program state can vary frequently within one run or from one run to the next (due to inputs, load, etc.) and requiring that this state also be matched, in addition to the call flow, would cause Dimmunix to incur many false negatives. The downside of the generality of patterns are occasional false positives, as will be discussed in §5.5.

5.4 Avoiding Previously Seen Patterns

The avoidance code is split from the monitoring code, so that expensive operations (cycle detection, history file I/O, etc.) can be done asynchronously, outside the application's critical path. The deadlock history is loaded from disk into memory at startup time and shared read-only among all threads; the monitor is the only thread mutating the history, both in-memory and on-disk.

As illustrated in Figure 1, Dimmunix intercepts all lock and unlock calls in the target program or intercepts them within a thread library. When the application performs a lock, the instrumentation invokes the *request* method, which returns a *YIELD* or *GO* answer. The *GO* case indicates that Dimmunix considers it safe (w.r.t. the history) for the thread to block waiting for the lock; this does not mean the lock is available. When the lock is actually acquired, the instrumentation invokes an *acquired* method in the avoidance code; when the lock is released, it invokes a *release* method—both methods serve solely to update the RAG, as they do not return any decision.

The *request* method determines whether allowing the current thread T's lock request would take the program into a situation that matches a previously seen deadlock or starvation. For this, it tentatively adds the corresponding allow edge to the RAG cache and searches for an instantiation of a signature from history; this consists of finding a set of (thread, lock, stack) tuples in the RAG cache that provide an exact cover of the signature. All thread-lock-stack tuples in the instance must correspond to distinct threads and locks. Checking for signature instantiation takes into consideration allow edges in addition to hold edges, because an allow edge represents a commitment by a thread to block waiting for a lock.

If a potential deadlock instance is found, then the tentative allow edge is flipped around into a request edge, and a yield edge is inserted into the RAG cache from T to each thread $T_i \neq T$ in the signature instance: these threads T_i are the "causes" of T's yield. Each yield edge gets its label from its yield cause (e.g., in Figure 2, $T_{13} \longrightarrow T_{22}$ gets label S_x from hold edge $L_5 \xrightarrow{S_x} T_{22}$). A *yield* event is sent to the monitor and a *YIELD* decision is returned to the instrumentation.

If no instance is found, then T's allow edge is kept, the corresponding *allow* event is sent to the monitor, and a *GO* decision is returned; any yield edges emerging from the current thread's node are removed.

When the *acquired* method is invoked, the corresponding allow edge in the RAG cache is converted into a hold edge and an *acquired* event is sent to the monitor. When *release* is invoked, the corresponding hold edge is removed and a *release* event is enqueued for the monitor.

Dimmunix provides two levels of immunity, each with its pros and cons; they can be selected via a configuration flag. By default, *weak immunity* is enforced: induced starvation is automatically broken (after saving its signature) and the program continues as if Dimmunix wasn't present—this is the least intrusive, but may lead to reoccurrences of some deadlock patterns. The number of times the initially avoided deadlock pattern can reoccur is bounded by the maximum nesting depth of locks in the program. The intuitive reason behind this upper bound is that avoiding a deadlock or starvation is always done at least one nesting level above the one where the avoided deadlock or starvation occurs. In *strong immunity* mode, the program is restarted every time a starvation is encountered, instead of merely breaking the yield cycle—while more intrusive, this mode guarantees that no deadlock or starvation patterns ever reoccur.

In our experience, one deadlock bug usually has one deadlock pattern (see §7). In the ideal case, if there are n deadlock bugs in the program, after exactly n occurrences of deadlocks the program will have acquired immunity against all n bugs. However, there could also be k induced starvation cases and, in the worst case, each new starvation situation will lead (after breaking) to the deadlock that was being avoided. Thus, it will take $n+k$ occurrences of deadlocks to develop immunity against all n deadlocks in the program. The exact values of n and k depend on the specific program at hand.

5.5 Calibrating the Matching Precision

A signature contains the call stacks from the corresponding RAG cycle, along with a "matching depth," indicating how long a suffix of each call stack should be considered during matching. In the simplest case, this depth is set to a fixed value (4 by default). However, choosing too long a suffix can cause Dimmunix to miss manifestations of a deadlock bug, while choosing too short a suffix can lead to mispredicting a runtime call flow as being headed for deadlock (i.e., this is a false positive). In this section we describe how Dimmunix can optionally calibrate the matching depth at runtime.

First, Dimmunix must be able to heuristically determine whether it did not cause a false positive (FP), i.e., whether forcing a thread to yield indeed avoided a deadlock or not. After deciding to avoid a given signature X, Dimmunix performs a retrospective analysis: All lock operations performed by threads involved in the potential deadlock are logged to the monitor thread, along with lock operations performed by the blocked thread after it was released from the yield. The monitor thread then looks for lock inversions in this log; if none are found, the avoidance was likely a FP, i.e., in the absence of avoidance, there would have likely not been a deadlock.

Using this heuristic, Dimmunix estimates the FP rate for each possible matching depth: when signature X is created, depth starts at 1 and is kept there for the first N_A avoidances of X, then incremented for the next N_A avoidances of X, and so on until maximum depth is reached. The N_A parameter is 20 by default. Then Dimmunix chooses the smallest depth d that exhibited the lowest FP rate $FP_{\min}$ and sets X's matching depth to d.

False positives are not exclusively due to overly general matching, but could also be caused by input or value dependencies; e.g., pattern X may lead to deadlock for some inputs but not for others, so avoiding X can have false positives even at the most precise matching depth. For this reason, $FP_{\min}$ can be non-zero, and multiple depths can have the same $FP_{\min}$ rate; choosing the smallest depth gives us the most general pattern.

The algorithm implemented in Dimmunix is slightly more complex. For instance, to increase calibration speed, when encountering a FP at depth k, Dimmunix analyzes whether it would have performed avoidance had the depth been $k+1$, $k+2$,... and, if yes, increments the FP counts for those depths as well; this allows the calibration to run fewer than N_A iterations at the larger depths. One could associate a per-stack matching depth instead of a per-signature depth; while this would be theoretically more precise, we found the current heuristic to be satisfactory for the systems discussed in §7.

Once X's calibration is complete, Dimmunix stops tracking FPs for X. After X has been avoided N_T times, Dimmunix performs a recalibration, in case program conditions have changed ($N_T = 10^4$ by default).

Dynamic calibration is a way to heuristically choose a deadlock pattern that is more balanced than if we chose a fixed-length suffix of the call stacks. This optional calibration algorithm is orthogonal to the rest of Dimmunix, since avoiding an execution pattern that matches partially a signature will cause all executions that match the signature fully (i.e., the precise deadlock pattern) to be avoided. Calibration merely makes Dimmunix more efficient at avoiding deadlocks similar to the ones already encountered, without incurring undue false positives.

5.6 Auxiliary Data Structures

The RAG is extended with several other data structures, which serve to improve both asymptotic and actual performance. For example, we achieve $O(1)$ lookup of thread and lock nodes, because they are kept in a preallocated vector and a lightly loaded hash table, respectively. In the case of library-based Dimmunix, the RAG nodes are embedded in the library's own thread and mutex data structures. Moreover, data structures necessary for avoidance and detection are themselves embedded in the thread and lock nodes. For example, the set *yieldCause* containing all of a thread T's yield edges is directly accessible from the thread node T.

Dimmunix uses a hash table to map raw call stacks to our own call stack objects. Matching a call stack consists of hashing the raw call stack and finding the corresponding metadata object S, if it exists. From S, one can directly get to, e.g., the *Allowed* set, containing handles to all the threads that are permitted to wait for locks while having call stack S; *Allowed* includes those threads that have acquired and still hold the locks. When checking for signature instantiations, a thread will check the *Allowed* sets for all call stacks S_i from the signature to be matched. In most cases, at least one of these sets is empty, meaning there is no thread holding a lock in that stack configuration, so the signature is not instantiated.

Complexity of the *request* method in the avoidance code is $O(D \cdot H \cdot T! \cdot G^T)$, where D is the maximum depth at which Dimmunix can match a call stack, H is the number of signatures in history, T is the maximum number of threads that can be involved in a deadlock, and G is the maximum number of locks acquired or waited for at the same time by threads with the exact same call stack. In practice, D is a constant and T is almost always two [16], bringing complexity closer to $O(H \cdot G^2)$.

Most accesses to the history and RAG cache are thread-safe, because they mutate allow and hold edges that involve the current thread only, so no other thread could be changing them at the same time. The *request* and *release* methods are the only ones that need to both consult and update the shared *Allowed* set. To do so safely without using locks, we use a variation of Peterson's algorithm for mutual exclusion generalized to n threads [9].

To find cycles in the RAG, we use the Colored DFS algorithm, whose theoretical complexity is $O(N_E + N_T \cdot (|V| + |E|))$, where the RAG is a graph $[V,E]$, the maximum number of threads issuing lock requests at any one time is N_T, and the maximum number of events in the asynchronous lock-free event queue is N_E.

5.7 Dimmunix Properties and Limitations

In this section, we summarize the most important properties of the algorithms presented so far. A formal description of an earlier version of the algorithm and its properties can be found in [12].

Dimmunix can never affect a deadlock-free program's correctness. Dimmunix saves a signature only when a deadlock actually happens, i.e., when there is a cycle in the RAG. A program that never deadlocks will have a perpetually empty history, which means no avoidance will ever be done.

Dimmunix must know about all synchronization mechanisms used in the system. In programs that mix Dimmunix-instrumented synchronization with non-instrumented synchronization, Dimmunix can interfere with the mechanisms it is unaware of (e.g., a program that combines non-instrumented busy-wait loops with instrumented POSIX threads locks could be starved). Thus, Dimmunix requires that the non-instrumented synchronization routines be indicated in a configuration file, similar to the way RacerX [5] does; Dimmunix will then ignore the avoidance decision whenever a call to the foreign synchronization is encountered.

Some deadlock patterns are too risky to avoid. Say there is an operation W that is implemented such that all possible execution paths are deadlock-prone. Dimmunix essentially prunes those paths that have deadlocked in the past, leaving only those that have not deadlocked; for W, this could mean eventually pruning all execution paths, leading to the loss of W's functionality. Although we have never noticed such functionality loss in thousands of executions of several instrumented desktop and server programs, it is possible in principle, so Dimmunix offers two options when running in "weak immunity" mode:

First, Dimmunix allows users to disable signatures. Every time Dimmunix avoids a signature, it logs the avoidance action in a field of the signature in the history. Now consider the following use scenario: a user is in front of their Web browser and, every time a suspected deadlock is avoided, Dimmunix beeps, the way pop-up blockers do. Say the user clicks on a menu item and s/he just hears a beep but nothing happens—the menu has been disabled due to avoidance. The user can now instruct Dimmunix to disable the last avoided signature, the same way s/he would enable pop-ups for a given site. The signature will never be avoided again and the menu is usable again (but it may occasionally deadlock, since the deadlock pattern is not being avoided).

Second, if users cannot be directly involved in detecting such starvation-based loss of functionality, Dimmunix has a configurable system-wide upper bound (e.g., 200 msec) for how long Dimmunix can keep a thread waiting in order to avoid a deadlock pattern; once this maximum is reached, the thread is released from the yield. Once a particular pattern has accumulated a large number of such aborts, it can be automatically disabled, or a warning can be issued instead to the user indicating this deadlock pattern is too risky to avoid.

Dimmunix cannot induce a non-real-time program to produce wrong outputs, even with strong immunity, because Dimmunix works solely by altering thread schedules. Schedulers in general-purpose systems (like a commodity JVM) do not provide strong guarantees, so the correctness of a program's outputs cannot reasonably depend on the scheduler. Starvation, as described above, is a liveness issue in a non-real-time system, so it cannot lead to the generation of incorrect outputs, i.e., it cannot violate safety.

Dimmunix never adds a false deadlock to the history, since it detects and saves only signatures of true deadlock patterns. Without a real deadlock, there cannot be a deadlock cycle in the RAG, hence the signature database cannot contain the signature of a deadlock pattern that never led to deadlock.

6 Dimmunix for Java and POSIX Threads

We currently have three implementations of Dimmunix: one for Java programs and two for programs using POSIX threads (pthreads): one for FreeBSD libthr and the other for Linux NPTL. They can be downloaded from `http://dimmunix.epfl.ch/`. The Java version has ∼1400 lines of Java code. The FreeBSD version has ∼1100 lines of C++ code plus ∼20 lines changed in libthr, while the Linux version has ∼1700 lines of C++ code plus ∼30 lines changed in NPTL. The latter's extra code is to support both 32-bit and 64-bit platforms.

The implementations illustrate two different approaches: the Java version directly instruments the target bytecode, while the pthreads implementations rely on modified pthreads libraries. Neither approach requires access to a program's source code nor does it ask programmers to changes how they write their code.

Java provides three main synchronization primitives: monitors, explicit locks, and wait queues; our implementation currently supports the first two. Monitors are declared using a *synchronized(x) {...}* statement, which translates, at the bytecode level, into *monitorenter(x)*, followed by the code block, followed by *monitorexit(x)*; an additional *monitorexit(x)* is placed on the exception path. If a thread attempts to enter a monitor it is already in, the call returns immediately; the thread will have to exit that monitor the same number of times it entered it before the monitor becomes available to others.

In order to intercept the monitor entry/exit and explicit lock/unlock requests, we use an aspect-oriented compiler, AspectJ [1], to directly instrument target programs at either bytecode or source level. The instrumented Java bytecode can be executed in any standard Java 1.5 VM or later. We implemented the avoidance code as aspects that get woven into the target program before and after every *monitorenter* and *lock* bytecode, as well as before every *monitorexit* and *unlock* bytecode. The aspects intercept the corresponding operations, update all necessary data structures, and decide whether to allow a lock request or to pause the thread instead. Call stacks are vectors of <methodName, file:line#> strings. The monitor thread is started automatically when the program starts up.

Dimmunix pauses a thread by making a Java *wait* call from within the instrumentation code; we do not use *Thread.yield*, because we found *wait* to scale considerably better. There is one synchronization object, *yieldLock[T]*, dedicated to each thread T and, when T is to yield, the instrumentation code calls *yieldLock[t].wait()*. When a thread T' frees a lock L' acquired with call stack S', then all threads T_i for which $(T', L', S') \in$ *yieldCause[T_i]* (see §5.6) have no more reason to yield, so they are woken up via a call to *yieldLock[T_i].notifyAll()*.

For **POSIX threads**, we modified the `libthr` library in FreeBSD and the Native POSIX Threads Library (NPTL) in Linux, which is part of glibc. The modified libraries are 100% compatible drop-in replacements. Porting to other POSIX threads libraries is straightforward. The pthreads-based implementations are similar to the Java implementation, with a few exceptions:

The basic synchronization primitive in POSIX threads is the *mutex*, and there are three types: normal mutex, recursive mutex (equivalent to Java's reentrant lock), and error-checking mutex, which returns EDEADLK if a thread attempts to lock a non-recursive locked mutex and thus self-deadlock. Dimmunix does not watch for self-deadlocks, since pthreads already offers the error-checking mutex option.

We instrumented all the basic mutex management functions. Locks associated with conditional variables are also instrumented. Having direct access to the thread library internals simplifies data access; for example, instead of keeping track of locks externally (as is done in the Java version), we can simply extend the original library data structures. Call stacks are unwound with backtrace(), and Dimmunix computes the byte offset of each return address relative to the beginning of the binary and stores these offsets in execution-independent signatures.

Java guarantees that all native lock() operations are blocking, i.e., after a successful *request* the thread will either *acquire* the lock or become blocked on it. This is not the case for pthreads, which allows a lock acquisition to time out (*pthread_mutex_timedlock*) or to return immediately if there is contention (*pthread_mutex_trylock*). To support trylocks and timedlocks, we introduced a new event in pthreads Dimmunix called *cancel*, which rolls back a previous lock *request* upon a timeout.

7 Evaluation

In this section we answer a number of practical questions. First and foremost, does Dimmunix work for real systems that do I/O, use system libraries, and interact with users and other systems (§7.1)? What performance overhead does Dimmunix introduce, and how does this overhead vary as parameters change (§7.2)? What is the impact of false positives on performance (§7.3)? What overheads does Dimmunix introduce in terms of resource consumption (§7.4)?

We evaluated Dimmunix with several real systems: MySQL (C/C++ open-source database), SQLite (C open-source embedded database), Apache ActiveMQ (Java open-source message broker for enterprise applications), JBoss (Java open-source enterprise application server), Limewire (Java peer-to-peer file sharing application), the Java JDK (provider of all class libraries that implement the Java API), and HawkNL (C library specialized for network games). These are widely-used systems within their category; some are large, such as MySQL, which has over 1 million lines of code excluding comments.

For all experiments reported here, we used strong immunity, with τ=100 msec; in the microbenchmarks we used a fixed call stack matching depth of 4. Measurements were obtained on 8-core computers (2x4-core Intel Xeon E5310 1.6GHz CPUs), 4GB RAM, WD-1500 hard disk, two NetXtreme II GbE interfaces with dedicated GbE switch, running Linux and FreeBSD, Java HotSpot Server VM 1.6.0, and Java SE 1.6.0.

7.1 Effectiveness Against Real Deadlocks

In practice, deadlocks arise from two main sources: bugs in the logic of the program (§7.1.1) and technically permissible, but yet inappropriate uses of third party code (§7.1.2); Dimmunix addresses both.

7.1.1 True Deadlock Bugs

To verify effectiveness against real bugs, we reproduced deadlocks that were reported against the systems described above. We used timing loops to generate "exploits," i.e. test cases that deterministically reproduced the deadlocks. It took on average two programmer-days to successfully reproduce a bug; we abandoned many bugs, because we could not reproduce them reliably. We ran each test 100 times in three different configurations: First, we ran the unmodified program, and the test always deadlocked prior to completion. Second, we ran the program instrumented with full Dimmunix, but ignored all yield decisions, to verify that timing changes introduced by the instrumentation did not affect the deadlock—again, each test case deadlocked in every run. Finally, we ran the program with full Dimmunix, with signatures of previously-encountered deadlocks in the history—in each case, Dimmunix successfully avoided the deadlock and allowed the test to run to completion.

The results are centralized in Table 1. We include the number of yields recorded during the trials with full Dimmunix as a measure of how often deadlock patterns were encountered and avoided. For most cases, there is one yield, corresponding to the one deadlock reproduced

System	Bug #	Deadlock Between ...	# Yields per Trial			Dlk Patterns	
			Min	Avg	Max	#	Depth
MySQL 6.0.4	37080	INSERT and TRUNCATE in two different threads	1	1	4	1	4
SQLite 3.3.0	1672	Deadlock in the custom recursive lock implementation	1	1	1	1	3
HawkNL 1.6b3	n/a	nlShutdown() called concurrently with nlClose()	10	10	10	1	2
MySQL 5.0 JDBC	2147	PreparedStatement.getWarnings() and Connection.close()	1	1	1	1	3
MySQL 5.0 JDBC	14972	Connection.prepareStatement() and Statement.close()	1	1	1	1	4
MySQL 5.0 JDBC	31136	PreparedStatement.executeQuery() and Connection.close()	1	1	1	1	3
MySQL 5.0 JDBC	17709	Statement.executeQuery() and Connection.prepareStatement()	1	1	1	1	3
Limewire 4.17.9	1449	HsqlDB TaskQueue cancel and shutdown()	15	15	15	2	10,10
ActiveMQ 3.1	336	Listener creation and active dispatching of messages to consumer	1	181079	221292	1	2
ActiveMQ 4.0	575	Queue.dropEvent() and PrefetchSubscription.add()	11252	80387	113652	3	2,2,2

Table 1: A few reported deadlock bugs avoided by Dimmunix in popular server and desktop applications.

by the test case. In some cases, however, the number of yields was much higher, because avoiding the initial deadlock enabled the test to continue and re-enter the same deadlock pattern later. For all but the ActiveMQ tests there were no false positives; in the case of ActiveMQ, we could not accurately determine if any of the reported yields were false positives.

We also inspected the code for each bug, to determine how many different deadlock patterns can be generated by the bug. The last two columns in Table 1 indicate the number of deadlock patterns ("#" column) and the size of the pattern ("Depth" column). Depth corresponds to the type of matching depth discussed in §5.5. Dimmunix correctly detected, saved, and avoided all patterns, except in the case of ActiveMQ #575, where we were able to only reproduce one of the three patterns, so Dimmunix only witnessed, saved and avoided that one.

7.1.2 Invitations to Deadlock

When using third party libraries, it is possible to use the offered APIs in ways that lead to deadlock inside the library, despite there being no logic bug in the calling program. For example, several synchronized base classes in the Java runtime environment can lead to deadlocks.

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 inside the JDK, because under 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.

Table 2 shows deadlocks we reproduced in JDK 1.6.0; they were all successfully avoided by Dimmunix. While not bugs per se, these are invitations to deadlock. Ideally, APIs should be documented thoroughly, but there is always a tradeoff between productivity and pedantry in documentation. Moreover, programmers cannot think of every possible way in which their API will be used. Runtime tools like Dimmunix provide an inexpensive alternative to this dilemma: avoid the deadlocks when and if they manifest. This requires no programmer intervention and no JDK modifications.

PrintWriter class: With *w* a PrintWriter, concurrently call w.write() and CharArrayWriter.writeTo(w)
Vector: Concurrently call v_1.addAll(v_2) and v_2.addAll(v_1)
Hashtable: With h_1 a member of h_2 and h_2 a member of h_1, concurrently call h_1.equals(foo) and h_2.equals(bar)
StringBuffer: With StringBuffers s_1 and s_2, concurrently call s_1.append(s_2) and s_2.append(s_1)
BeanContextSupport: concurrent propertyChange() and remove()

Table 2: Java JDK 1.6 deadlocks avoided by Dimmunix.

7.2 Performance Overhead

In this section we systematically quantify Dimmunix's impact on system performance, using request throughput and latency as the main metrics. First, we report in §7.2.1 end-to-end measurements on real systems and then use synthetic microbenchmarks to drill deeper into the performance characteristics (§7.2.2).

7.2.1 Real Applications

To measure end-to-end overhead, we ran standard performance benchmarks on "immunized" JBoss 4.0 and MySQL 5.0 JDBC. For JBoss, we used the RUBiS e-commerce benchmark [19], for MySQL JDBC we used JDBCBench [11]. For HawkNL, Limewire, and ActiveMQ we are unaware of any benchmarks.

We took the measurements for various history sizes, to see how overhead changes as more signatures accumulate. Since we had insufficient real deadlock signatures, we synthesized additional ones as random combinations of real program stacks with which the target system performs synchronization. From the point of view of avoidance overhead, synthesized signatures have the same effect as real ones. Figure 4 presents the results.

The cost of immunity against up to 128 deadlock signatures is modest in large systems with hundreds of threads in realistic settings—e.g., JBoss/RUBiS ran with

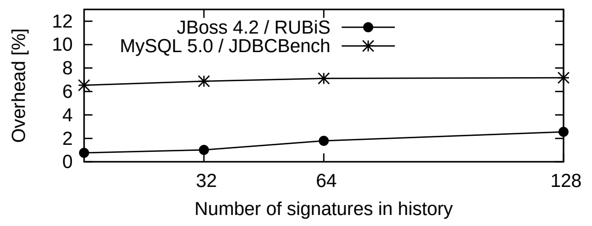


Figure 4: Performance overhead introduced in real systems, computed on the benchmark-specific metric. Maximum overhead is 2.6% for JBoss and 7.17% for MySQL JDBC.

3000 clients, a mixed read/write workload, and performed on average ∼500 lock operations per second while running 280 threads. We did not witness a statistically meaningful drop in response time for either system. In light of these results, it is reasonable to suggest that users encountering deadlocks be offered the option of using Dimmunix right away to cope, while the respective development teams fix the underlying bugs. The development teams themselves could also provide deadlock signatures to customers until fixes for the bugs become available.

7.2.2 Microbenchmarks

To dissect Dimmunix's performance behavior and understand how it varies with various parameters, we wrote a synchronization-intensive microbenchmark that creates N_t threads and has them synchronize on locks from a total of N_l locks shared among the threads; a lock is held for δ_{in} time before being released and a new lock is requested after δ_{out} time; the delays are implemented as busy loops, thus simulating computation done inside and outside the critical sections. The threads call multiple functions within the microbenchmark so as to build up different call stacks; which function is called at each level is chosen randomly, thus generating a uniformly distributed selection of call stacks.

We also wrote a tool that generates synthetic deadlock history files containing H signatures, all of size S; for a real application, H represents the number of deadlock/starvation signatures that have accumulated in the history, and a signature's size indicates the number of threads involved in that deadlock. Generated signatures consist of random stack combinations for synchronization operations in the benchmark program—not signatures of real deadlocks, but avoided as if they were.

Overhead as a function of the number of threads: Figure 5 shows how synchronization throughput (in terms of lock operations) varies with the number of threads in Java and pthreads, respectively. We chose δ_{in}=1 microsecond and δ_{out}=1 millisecond, to simulate a program that grabs a lock, updates some in-memory shared data structures, releases the lock, and then performs computation outside the critical section.

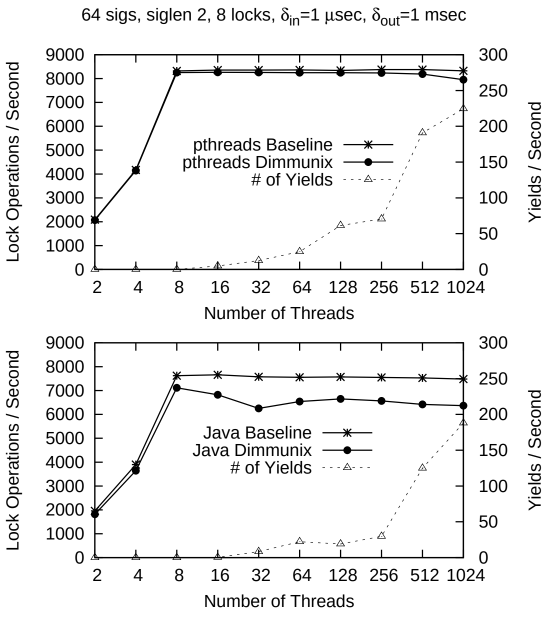


Figure 5: Dimmunix microbenchmark lock throughput as a function of number of threads. Overhead is 0.6% to 4.5% for FreeBSD pthreads and 6.5% to 17.5% for Java.

We observe that Dimmunix scales well: for up to 1024 threads, the pthreads implementation exhibits maximum 4.5% overhead, while the Java implementation maximum 17.5%. The difference between the implementations is, we believe, primarily due to Java-specific overheads (such as returning the call stack as a vector of strings vs. mere pointers in C, or introducing extra memory fences around synchronized{} blocks, that pthreads does not do). As the benchmark approaches the behavior we see in real applications that perform I/O, we would expect the overhead to be further absorbed by the time spent between lock/unlock operations. To validate this hypothesis, we measured the variation of lock throughput with the values of δ_{in} and δ_{out}—Figure 6 shows the results for Java; pthreads results are similar.

The overhead introduced by Dimmunix is highest when the program does nothing but lock and unlock (i.e., δ_{in}=δ_{out}=0). This is not surprising, because Dimmunix intercepts the calls to lock/unlock and performs additional computation in the critical path. lock/unlock are generally fast operations that take a few machine instructions to perform, so adding 10× more instructions in the path will cause the overhead to be 10×. However, as the interval between critical sections (δ_{out}) or inside critical sections (δ_{in}) increases, the throughput difference between the immunized vs. non-immunized microbenchmark decreases correspondingly. For most common scenarios (i.e., inter-critical-section intervals of 1 millisecond or more), overhead is modest.

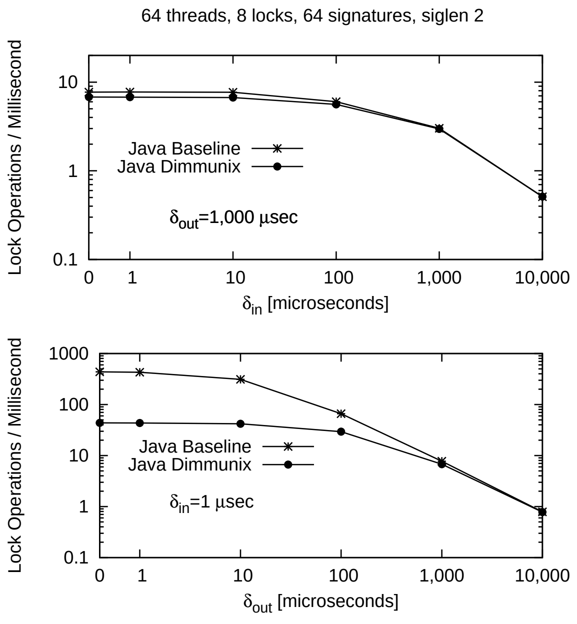


Figure 6: Variation of lock throughput as a function of δ_{in} and δ_{out} for Java; the pthreads version is similar.

Note that a direct comparison of overhead between Dimmunix and the baseline is somewhat unfair to Dimmunix, because non-immunized programs deadlock and stop running, whereas immunized ones continue running and doing useful work.

Impact of history size and matching depth: The performance penalty incurred by matching current executions against signatures from history should increase with the size of the history (i.e., number of signatures) as well as the depth at which signatures are matched with current stacks. Average length of a signature (i.e., average number of threads involved in the captured deadlock) also influences matching time, but the vast majority of deadlocks in practice are limited to two threads [16], so variation with signature size is not that interesting.

In addition to the matching overhead, as more and more deadlocks are discovered in the program, the program must be serialized increasingly more in order to be deadlock-safe (i.e., there are more deadlocks to avoid)—our overhead measurements include both effects.

We show in Figure 7 the performance overhead introduced by varying history size from 2-256 signatures. The overhead introduced by history size and matching depth is relatively constant across this range, which means that searching through history is a negligible component of Dimmunix overhead.

Breakdown of overhead: Having seen the impact of number of threads, history size, and matching depth, we profiled the overhead, to understand which parts of Dimmunix contribute the most. For this, we selectively disabled parts of Dimmunix and measured the lock throughput. First we measured the overhead introduced by the base instrumentation, then we added the data structure lookups and updates performed by *request* in the avoidance code, then we ran full Dimmunix, including avoidance.

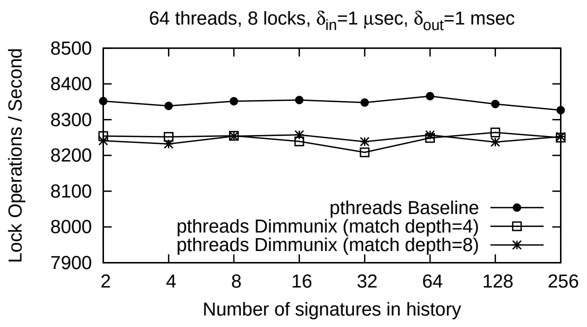


Figure 7: Lock throughput as a function of history size and matching depth for pthreads. Java results are similar.

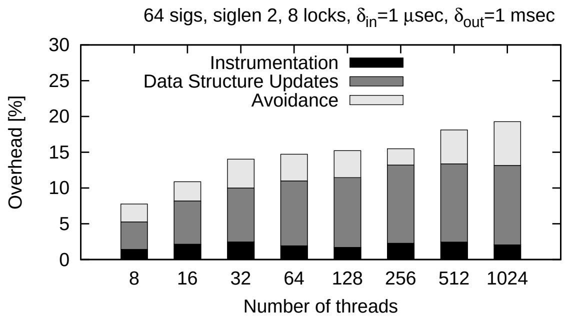


Figure 8: Breakdown of overhead for Java Dimmunix.

The results for Java are shown in Figure 8—the bulk of the overhead is introduced by the data structure lookups and updates. For pthreads, the trend is similar, except that the dominant fraction of overhead is introduced by the instrumentation code. The main reason is that the changes to the pthreads library interfere with the fastpath of the pthreads mutex: it first performs a compare-and-swap (CAS) and only if that is unsuccessful does it make a system call. Our current implementation causes that CAS to be unsuccessful with higher probability.

7.3 False Positives

Any approach that tries to predict the future with the purpose of avoiding bad outcomes suffers from false positives, i.e., wrongly predicting that the bad outcome will occur. Dimmunix is no exception. False positives can arise when signatures are matched too shallowly, or when the lock order in a pattern depends on inputs, program state, etc. Our microbenchmark does not have the latter type of dependencies.

In a false positive, Dimmunix reschedules threads in order to avoid an apparent impending deadlock that would actually not have occurred; this can have negative or positive effects on performance, the latter due to reduced contention. We concern ourselves here with the negative effects, which result from a loss in parallelism: Dimmunix serializes "needlessly" a portion of the program execution, which causes the program to run slower.

In our microbenchmark, let D be the program's maximum stack depth (we set D=10) and let k be the depth at which we match signature stacks in the avoidance code. We consider a true positive to be an avoidance triggered by a match at depth D; a false positive occurs when a signature is matched to depth k but would not match to depth D. If k=D, there are no false positives, because the signatures are matched exactly, but if k<D, then we can get false positives, because several different runtime stacks produce a match on the same signature.

In order to determine the overhead induced by false positives, we compare the lock throughput obtained while matching at depths k<D (i.e., in the presence of false positives) to that obtained while matching at depth D (no false positives)—the difference represents the time wasted due to false positives. To measure the overhead introduced by Dimmunix itself, separate from that introduced by false positives, we measure the overhead of Dimmunix when all its avoidance decisions are ignored (thus, no false positives) and subtract it from the baseline. Calibration of matching precision is turned off. Figure 9 shows the results—as the precision of matching is increased, the overhead induced by false positives decreases. There are hardly any false positives for depths of 8 and 9 because the probability of encountering a stack that matches at that depth and not at depth 10 is very low.

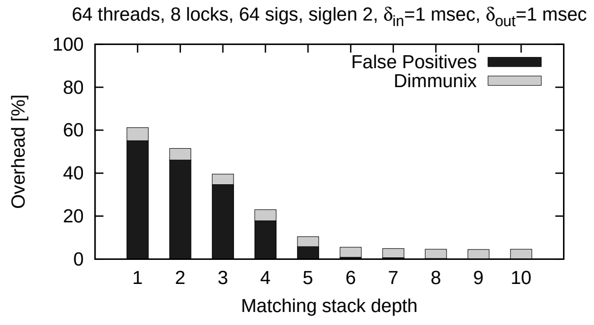


Figure 9: Overhead induced by false positives.

We ran this same experiment using the technique based on gate locks [17], the best hybrid dynamic/static deadlock avoidance we know of. To avoid the 64 deadlocks represented in history, 45 gate locks were required; since [17] does not use call stacks, matching depth is irelevant. Throughput overhead with gate locks was 70%—more than an order of magnitude greater than Dimmunix's 4.6% overhead for stack depth $\geq$8 and close to Dimmunix's 61.2% overhead at stack depth 1. There were 561,627 false positives with gate locks; in contrast, Dimmunix's false positives ranged from 0 (at depth 10) to 573,912 (at depth 1). This is consistent with the fact that, for stack depth 1, the two approaches are similar.

As mentioned in §5.7, false positives can also disable functionality. We did not encounter such loss during any of the thousands of executions of the various server and desktop applications described in this paper, but Dimmunix does provide two resolutions for these situations: manual or automatic disabling of signatures.

7.4 Resource Utilization

The final aspect of Dimmunix we wish to measure is how many additional resources it requires, compared to non-immunized programs. Dimmunix uses CPU time for computation, memory for its data structures, and disk space for the history. The latter is insignificant: on the order of 200-1000 bytes per signature, amounting to tens of KBs for a realistic history. CPU overhead was virtually zero in all our measurements; in fact, delaying some of the threads can even lead to negative overhead, through the reduction of contention.

In measurements ranging from 2-1024 threads, 8-32 shared locks, and a history of 64 two-thread signatures, the pthreads implementations incurred a memory overhead of 6-25 MB, and the Java implementation 79-127 MB. As described in §5.6, we use preallocation to reduce performance overhead, and the data structures themselves have redundancy, to speed up lookups. We expect a Dimmunix version optimized for memory footprint to have considerably less memory overhead.

8 Using Dimmunix in Practice

Dimmunix helps programs develop resistance against deadlocks, without assistance from programmers (i.e., no annotations, no specifications) or from system users. Dimmunix can be used as a band-aid to complement all the other development and deployment tools for software systems, such as static analyzers and model checkers. In systems that have checkpoint facilities [18] or are microrebootable [4], strong immunity can offer strong guarantees at low cost; in other general-purpose systems, weak immunity lends progressively stronger resistance to deadlocks, without incurring additional recoveries.

Aside from achieving immunity, Dimmunix can also be used as an alternative to patching and upgrading: instead of modifying the program code, it can be "patched" against deadlock bugs by simply inserting the corresponding bug's signature into the deadlock history and asking Dimmunix to reload the history; the target program need not even be restarted. Similarly, vendors could ship their software with signatures for known deadlocks, as an alternative to fixing them in the released code when doing so is too expensive or risky.

When upgrading a system, the signature history also needs to be updated. First, code locations captured in the signatures' call stacks may have shifted or disappeared; static analysis can be used to map from old to new code and "port" signatures from one revision to the next, thus accounting for refactoring, addition/removal of code, etc. Second, at a higher semantic level, deadlock behaviors may have been modified by the upgrade (e.g., by fixing a bug), thus rendering some signatures obsolete, regardless of porting. The calibration of matching precision (§5.5) is therefore re-enabled after every upgrade for all signatures, and any signatures that encounter 100% false positive rate after this recalibration can be automatically discarded as obsolete.

Dimmunix currently does not take into account the priorities of threads when making scheduling decisions; the server applications we are familiar with do not use thread priorities, but perhaps other applications do. We believe support for priorities can easily be added.

Although Dimmunix does not introduce new bugs, avoiding deadlocks could trigger latent program bugs that might otherwise not manifest, in much the same way an upgrade of the JVM, system libraries, or kernel could do. While this is not a limitation per se, it is a factor to be considered when using Dimmunix in practice.

9 Conclusion

In this paper we described a technique for augmenting software systems with an "immune system" against deadlocks. Unlike pure deadlock avoidance, deadlock immunity is easier to achieve in practical settings and appears to be almost as useful as ideal pure avoidance. The essence of our approach is to "fingerprint" control flows that lead to deadlock, save them in a persistent history, and avoid them during subsequent runs.

We showed empirically that real systems instrumented with Dimmunix can develop resistance against real deadlocks, without any assistance from programmers or users, while incurring modest overhead. Dimmunix scales gracefully to large software systems with up to 1024 threads and preserves correctness of general-purpose programs. We conclude that deadlock immunity is a practical approach to avoiding deadlocks, that can improve end users' experience with deadlock-prone systems and also keep production systems running deadlock-free despite the bugs that lurk within.

Acknowledgments

We thank our shepherd, Wolfgang Schröder-Preikschat, and the anonymous reviewers for their help in refining our paper. We are indebted to Vinu Rajashekar for the NPTL port, Silviu Andrica for his help with ActiveMQ, and Katerina Argyraki, Emmanuel Cecchet, Steve Hand, and Willy Zwaenepoel for their early feedback and ideas.

References

[1] AspectJ. http://www.eclipse.org/aspectj.

[2] P. Boronat and V. Cholvi. A transformation to provide deadlock-free programs. In *Intl. Conf. on Computational Science*, 2003.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. 2002.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. 2004.

[5] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. 2003.

[6] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Intl. Symp. on Formal Methods Europe*, 2001.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. 2002.

[8] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. 1993.

[9] M. Hofri. Proof of a mutual exclusion algorithm. *ACM SIGOPS Operating Systems Review*, 24(1):18–22, 1990.

[10] Java PathFinder. http://javapathfinder.sourceforge.net, 2007.

[11] JDBCBench. http://mmmysql.sourceforge.net/performance.

[12] H. Jula and G. Candea. A scalable, sound, eventually-complete algorithm for deadlock immunity. In *Workshop on Runtime Verification*, 2008.

[13] H. Kopetz. Fault management in the time triggered protocol. In *EuroMicro Conf. on Real-Time Systems*, 1994.

[14] E. Koskinen and M. Herlihy. Dreadlocks: Efficient deadlock detection. In *ACM Symp. on Parallelism in Algorithms and Architectures*, 2008.

[15] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2), 1980.

[16] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. 2008.

[17] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: From exhibiting to healing. In *Workshop on Runtime Verification*, 2008.

[18] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies – a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[19] RUBiS. http://rubis.objectweb.org, 2007.

[20] L. Sha, R. Rajkumar, and S. Sathaye. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990.

[21] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *European Conf. on Object-Oriented Programming*, 2005.

[22] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering*, 16(3), 1990.

[23] F. Zeng and R. P. Martin. Ghost locks: Deadlock prevention for Java. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.