Appears in Proc. Intl. Conference on Runtime Verification (RV), San Francisco, CA, September 2011

Efficiency Optimizations for Implementations of Deadlock Immunity

Horatiu Jula, Silviu Andrica, and George Candea

School of Computer and Communication Science
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. Deadlock immunity is a property by which programs, once afflicted by a deadlock, develop resistance against future occurrences of that deadlock. Our deadlock immunity system, called Dimmunix, provides transparent immunization against deadlocks involving mutex locks.
In this paper, we focus on efficiently protecting systems against deadlocks regardless of the rate of synchronization operations performed. We describe five optimizations that reduce the runtime overhead imposed by Dimmunix on the host system: (1) offline deadlock detection and signature extraction, which avoids runtime tracking of lock-to-thread allocations; (2) selective program instrumentation, whereby only vulnerable synchronization statements are monitored; (3) inline matching of deadlock signatures, which avoids expensive call stack retrieval; (4) false positive reduction, which avoids unnecessary thread serialization; and (5) safe early resumption of threads, allowing suspended threads to resume their execution more quickly than in the original Dimmunix. Our optimizations enable Dimmunix to achieve a reduction of 2.8x-5.2x in the runtime overhead it introduces for real-world systems like Eclipse, Vuze, and MySQL JDBC.

1 Introduction

When threads do not coordinate correctly in their use of locks, a deadlock can occur—a situation whereby a group of threads cannot make progress (i.e., they hang), because each thread is waiting for another thread to release a lock. Although deadlocks involving other types of synchronization mechanisms exist (e.g., deadlocks caused by condition variables) deadlocks involving locks are prevalent [4, 2, 7].

Deadlocks are an important cause of system failures, as revealed by multiple surveys [4, 2, 7], yet avoiding their introduction during development is challenging. Large software systems are developed by many programmers, which makes it hard to maintain the coding discipline needed to avoid deadlocks. Exercising all possible execution paths and thread interleavings during testing is infeasible in practice for large programs, and even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field, due to dependencies on deadlock-prone third party libraries and plugins.

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, because a fix would entail drastic redesign.

Patches, too, are error-prone: many concurrency bug fixes either introduce new bugs or, instead of fixing the underlying bug, merely decrease the probability of occurrence [4].

To address these problems, we developed a technique called deadlock immunity [3]. It helps applications defend against deadlocks by enabling them, once afflicted by a given deadlock, to automatically develop resistance against future occurrences of that deadlock. We implemented this technique in a system called Dimmunix, which has two modules running simultaneously: (1) a detector that dynamically detects deadlocks and extracts their signatures, and (2) an avoidance module that uses the signatures as antibodies to avoid future occurrences of these deadlocks. A signature is an approximation of the execution flow that led to a deadlock. To avoid a previously encountered deadlock, Dimmunix temporarily suspends the threads whose executions are about to match the signature of that deadlock.

The challenge of efficiently scaling Dimmunix to synchronization-intensive systems resides in its necessity to process *every* lock operation. A high synchronization rate creates a large amount of work for Dimmunix, causing it to induce a high runtime overhead. In this paper, we describe optimizations that are generally applicable to all runtimes implementing deadlock immunity. These optimizations enable the runtimes to protect systems against deadlocks regardless of the rate of synchronization operations.

We present five main optimizations for runtimes implementing deadlock immunity: (1) offline deadlock detection, i.e., deadlocks are detected and their signatures extracted only when the program terminates, instead of performing these tasks whenever a thread requests a lock; (2) selective program instrumentation, whereby only vulnerable synchronization statements are monitored; (3) inline matching of deadlock signatures, which avoids expensive call stack retrieval; (4) false positive reduction, which avoids unnecessary thread serialization; and (5) safe early resumption of threads, allowing suspended threads to resume their execution more quickly than in the original Dimmunix.

We implemented these optimizations in Dimmunix and achieved a reduction of 2.8x-5.2x in the runtime overhead Dimmunix introduces for real world systems like Eclipse, Vuze, and MySQL JDBC.

In the rest of the paper we present background information about Dimmunix (§2), then describe the optimizations (§3), and assess their effectiveness (§4). We then review related work (§5) and conclude (§6).

2 Background

Deadlock immunity is a property by which programs, once afflicted by a deadlock, develop resistance against future occurrences of that deadlock. Dimmunix [3] provides immunization against deadlocks involving mutex locks, with no assistance from programmers or users. Dimmunix can be used by customers to defend themselves against deadlocks while waiting for a fix, and by software vendors as a safety net. Its architecture consists of two parts: (1) a module that detects deadlocks and adds their signatures to a persistent deadlock history, and (2) an avoidance module that prevents occurrences of previously encountered deadlocks, by avoiding execution flows matching signatures from history: whenever the execution of a thread may lead to a previously encountered deadlock, Dimmunix suspends that thread until the deadlock danger passes.

Dimmunix detects and avoids deadlocks by taking control of the program whenever a thread requests a lock, after it acquires that lock, and before it releases that lock. Dimmunix uses these events to update the synchronization state of the program, represented as a resource allocation graph (RAG) whose nodes are threads and locks. The RAG edges are of three types: an edge from a thread t to a lock l denotes that t is waiting to acquire l; an edge from l to t denotes that t is holding l; an edge from a thread t_1 to a thread t_2 means that Dimmunix suspended t_1 because of t_2, in order to avoid a deadlock. Edges are annotated with the call stack a thread had when the edge was created.

Dimmunix detects deadlocks by looking for cycles in the RAG every time a thread requests a lock, and if a cycle is found, it saves the deadlock's signature to a persistent history, to prevent future occurrences of this deadlock. A signature characterizes the deadlocking execution via the program positions of the nested synchronization statements it involves. A program position represents a location in the source code or an offset in the program binary; for Java programs, Dimmunix uses source code locations. We call the nested synchronization statements involved in the deadlock "outer" and "inner" lock statements. The outer lock statements correspond to the acquisitions of the locks involved in the deadlock. The inner lock statements correspond to the places where the threads deadlocked. Since these statements can be reached by a multitude of program executions, of which only a few deadlock, Dimmunix additionally saves in the deadlock signature the call stacks the deadlocked threads had when they acquired the locks involved in the deadlock (called "outer call stacks") and the call stacks the deadlocked threads had at the time of the deadlock (called "inner call stacks"). Each frame of an outer/inner call stack is a program position; the top frame points to a synchronization statement. The top frame of an outer (inner) call stack points to an outer (respectively inner) lock statement.

Imagine a deadlock involving threads t_1 and t_2 that have acquired locks l_1 and l_2, respectively, and now wait to acquire the other lock. This deadlock appears in the RAG as the cycle $l_1 \xrightarrow{CS_1^{out}} t_1 \xrightarrow{CS_1^{in}} l_2 \xrightarrow{CS_2^{out}} t_2 \xrightarrow{CS_2^{in}} l_1$, and the signature of the deadlock consists of the pairs of outer and inner call stacks, i.e., $\{(CS_1^{out}, CS_1^{in}), (CS_2^{out}, CS_2^{in})\}$. The signature is saved to a history file that persists across multiple executions of the program.

An instantiation of a signature S with outer call stacks $CS_1^{out}, ..., CS_n^{out}$ is a situation where threads $t_1, ..., t_n$ hold (or are allowed to acquire) locks $l_1, ..., l_n$ while having call stacks $CS_1^{out}, ..., CS_n^{out}$. Each outer call stack CS_i^{out} is matched up to a predefined depth, called matching depth, defined as the number of consecutive frames in CS_i^{out} to be matched against a thread's call stack, starting from the top frame. We formally represent the instantiation as the set $I = \{(t_1, l_1, CS_1^{out}), ..., (t_n, l_n, CS_n^{out})\}$.

Avoiding previously seen deadlocks consists of avoiding instantiations of signatures from the deadlock history. Consider that thread t_1 requests lock l_1 while having call stack CS_1^{out}. To avoid instantiations of signature S, Dimmunix first "pretends" to allow t_1 to acquire l_1, i.e., it does not allow t_1 to proceed, but it updates the RAG as if it did. Then, Dimmunix checks if instantiations of S are possible; if yes, Dimmunix suspends t_1 until no instantiations of S are possible. When Dimmunix suspends a thread, we say that the thread "yields."

Dimmunix automatically handles avoidance-induced deadlocks: when one occurs, Dimmunix saves its signature (so it will avoid its reoccurrence, just like for a normal deadlock) and resumes the suspended threads. More details appear in [3].

3 Optimizations

In this section, we present the optimizations we performed to Dimmunix to achieve low runtime overhead for Java programs regardless of the synchronization operation throughput. There are three improvements that we achieve: first, we reduce the number of synchronization operations that are intercepted by Dimmunix. Second, we optimize Dimmunix's performance-critical computations. Third, we reduce the amount of time Dimmunix suspends threads to avoid deadlocks.

For Dimmunix to intercept fewer synchronization operations, we needed to implement two optimizations: first, Dimmunix detects deadlocks and extracts their signatures offline, when the program is forcefully terminated. To detect deadlocks, Dimmunix invokes the JVM's deadlock detection method. Previously, Dimmunix used online deadlock detection, i.e., the RAG was updated upon each synchronization operation and the deadlock detection was performed periodically [3]; therefore, Dimmunix needed to intercept every synchronization operation.

We implemented the offline signature extraction for deadlocks involving synchronized blocks/methods. Upon a deadlock, Dimmunix automatically infers the outer call stacks of a deadlock signature from the inner call stacks, which are available at the time of the deadlock. Previously, Dimmunix retrieved and stored upon each lock acquisition the call stack of the thread that requested the lock [3]. Section 3.1 presents details about the inference of outer call stacks.

Second, Dimmunix instruments only the synchronized blocks/methods previously involved in deadlocks. This optimization is effective because most of the mutex synchronization statements (i.e., lock/unlock statements) are synchronized blocks/methods (e.g., more than 96% in Vuze, ActiveMQ, Limewire, and JBoss, and 58.3% in Eclipse) and only the ones previously involved in deadlocks are instrumented. Section 3.2 describes the selective program instrumentation.

We identified one performance-critical computation in Dimmunix: the matching of deadlock signatures' call stacks. Most of the computations are involved in checking whether a previously encountered deadlock is about to reoccur. Previously, Dimmunix used standard call stack retrieval methods (e.g., Java's *Thread.getStackTrace()* method) to obtain the call stack of a thread upon a lock acquisition and compared it to the call stacks of the signatures in the history [3]. Dimmunix spent most of the execution time in the call stack retrieval. We optimized it by inlining the call stack matching (§3.3).

Finally, we reduce the amount of time Dimmunix suspends threads to avoid deadlocks (i.e., the thread serialization), by performing two optimizations: first, Dimmunix automatically detects a posteriori if the decisions to suspend threads to avoid deadlocks were false positives (FPs), and increases the signature matching accuracy whenever a FP is encountered (§3.4). Increasing the matching depth for a signature reduces the probability of matching the signature at runtime, which means that thread yields are less frequent; therefore, the amount of thread serialization decreases. We already introduced in [3] the FP detection mechanism; however, this paper is the first one explaining it in depth. Second, Dimmunix resumes the threads suspended to avoid deadlocks as soon as the program execution reaches a point from which the deadlock situation becomes unreachable, i.e., the inner lock statements involved in the deadlock become unreachable (§3.5). Resuming threads earlier reduces the duration of the yields, which means less thread serialization. Previously, Dimmunix resumed the suspended threads only when at least one lock involved in an avoided signature instantiation was released [3].

3.1 Inferring the Outer Call Stacks of a Signature

To be able to selectively instrument only the synchronized blocks/methods previously involved in deadlocks, Dimmunix needs to automatically infer the outer call stacks of a deadlock signature from the inner call stacks, which are available at the time of the deadlock. Otherwise, Dimmunix would have to retrieve and store the call stack of the caller thread upon each lock acquisition.

To deterministically infer the outer call stack of a signature, we require usage of properly nested synchronization statements, i.e., locks are released in the reverse order of their acquisitions. For example, this is the case for Java's synchronized blocks/methods, but the technique is not limited to these only. Properly nested synchronization statements enable us to deterministically obtain the outer call stacks of a signature from inner call stacks by removing frames from the top of the latter, because the outer call stacks are prefixes of the inner call stacks.

Inferring outer call stacks works as follows: first, Dimmunix finds the threads that acquired the locks involved in a deadlock, as described in Algorithm 1. This requires access to each thread's lock stack, which contains the locks that thread acquired (and still holds) and the requested one on top. For Java programs, these stacks are obtained from the JVM. For each lock l_i involved in the deadlock, Dimmunix finds the program position where lock l_i was acquired. It is not possible to infer this program position based solely on the information provided by the CFG, because the CFG does not contain lock identities. Therefore, Dimmunix needs to find the index k_j of lock l_i in the lock stack LS_j of thread t_j owning l_i (lines 1–4). Next, Dimmunix finds where thread t_j acquired l_i (i.e., its k_j-th lock), by exploring backward the CFG the application and popping call frames from the inner call stack, as shown in Algorithm 2 (lines 8–12). Every time a lock (respectively unlock) statement is encountered, the counter $k_{nesting}$ storing the nesting level is incremented (respectively decremented); initially, $k_{nesting} = 0$ (lines 1–7). The outer lock statement is reached when $k_{nesting} = k_j$, and the algorithm returns the current call stack with the top frame replaced by the current lock statement (lines 4–5). The algorithm is deterministic because exploring backward any execution path leads to the same outer lock statement.

3.2 Selective Program Instrumentation

Dimmunix instruments only the synchronized blocks/methods previously involved in deadlocks. To avoid previously encountered deadlocks, it is sufficient for Dimmunix to instrument only these statements.

```
Input: Deadlocked threads $t_1,...,t_n$ with call stacks $CS_1^{inner},...,CS_n^{inner}$, lock stacks
       $LS_1,...,LS_n$, and requested locks $l_1,...,l_n$.
Output: Signature $S = \{(CS_1^{outer}, CS_1^{inner}),...,(CS_n^{outer}, CS_n^{inner})\}$, where $CS_i^{outer}$ are the
        inferred outer call stacks.
1 foreach $i \in [1,n]$ do
2     Let $t_j$ be the thread holding $l_i$
3     Find the index $k_j$ of $l_i$ in $LS_j$, corresponding to $l_i$'s acquisition
4 foreach $i \in [1,n]$ do
5     $CS_i^{outer} := getOuterCallStack(CS_i^{inner}, k_i, CS_i^{inner}.top, 0)$
6 return $\{(CS_1^{outer}, CS_1^{inner}),...,(CS_n^{outer}, CS_n^{inner})\}$
```

Algorithm 1: *getSignature*: building the signature of a deadlock.

```
Input: Inner call stack $CS^{inner}$; Lock stack index $k$; Current statement $s$, initially the
       statement corresponding to $CS^{inner}$'s top frame; Current nesting level $k_{nesting}$,
       initially 0.
Data: Control flow graph (CFG) of the method containing $s$.
Output: Inferred outer call stack $CS^{outer}$.
1  if there exists an unexplored predecessor $s'$ of $s$ in the CFG then
2      if $s'$ is lock acquisition then
3          $k_{nesting} := k_{nesting} + 1$
4          if $k_{nesting} = k$ then
5              return $CS^{inner}.pop().push(s')$ // replace the top frame with $s'$
6      if $s'$ is lock release then
7          $k_{nesting} := k_{nesting} - 1$
8      return $getOuterCallStack(CS^{inner}, k, s', k_{nesting})$
9  else
10     $CS^{inner} := CS^{inner}.pop()$ // remove the top frame
11     return $getOuterCallStack(CS^{inner}, k, CS^{inner}.top, k_{nesting})$
```

Algorithm 2: $getOuterCallStack(CS^{inner}, k, s, k_{nesting})$: recursively computes the outer call stack corresponding to an inner call stack CS^{inner} and a lock stack index k.

Dimmunix instruments the outer and inner lock statements involved in a deadlock (i.e., the program positions referenced by the top frames of the deadlock's signature), and the corresponding unlock statements. Since synchronized blocks are properly nested, the unlock statements corresponding to a lock statement s_l are easily found by exploring forward the CFG and keeping track of the nesting level. Matching lock and unlock statements have the same nesting level.

The outer call stacks of a deadlock signature cannot be inferred deterministically (in the general case) for explicit lock acquisition statements, like Java's *ReentrantLock.lock()* Therefore, Dimmunix needs to intercept each explicit lock acquisition and store the call stack of the caller thread, in order to obtain the outer call stacks. However, since Java programs mostly use synchronized blocks/methods to acquire locks, the amount of instrumentation is substantially reduced for Java programs.

```
Input: Outer call stack $CS$; Depth $d$. The call frame at depth $d$ in $CS$ is currently matched
       by thread $t$'s execution.
Data: Counter $matches[CS, t]$, initialized to $CS.depth$.
Output: True if $CS$ is matched up to its matching depth $CS.depth$, False otherwise.
1 if $d > CS.depth$ then
2     return False
3 if $d = CS.depth$ then
4     $matches[CS,t] := d - 1$
5 else
6     if $d = matches[CS,t]$ then
7         $matches[CS,t] := matches[CS,t] - 1$
8 return $matches[CS,t] = 0$
```

Algorithm 3: *inlineMatch(CS, d, t)*: checks whether thread t's execution, currently matching the frame at depth d in call stack CS, matches CS up to its matching depth *CS.depth*, i.e., matches the top *CS.depth* frames of CS.

3.3 Inline Call Stack Matching

A straightforward way to match a signature S is to retrieve the current call stack of a thread upon a lock request and compare it to the outer call stacks of S up to their matching depth. If the outer call stacks end in lock statements that execute often, this matching mechanism becomes a bottleneck, because retrieving call stacks is expensive for platforms like the JVM. Inlining the call stack matching considerably reduces the performance overhead incurred by Dimmunix, as we show in §4.

In inline matching, the outer call stacks of a signature are incrementally matched, as the program executes; we present this mechanism in detail in Algorithm 3. For each outer call stack CS of a signature in the deadlock history, Dimmunix automatically instruments the program bytecode before the statements referenced by the frames in CS with code that works as follows: before a thread t executes such a statement, the matching code decrements the counter *matches[CS, t]* (lines 3–7). The counter represents the number of frames in CS that are yet unmatched by thread t, starting from CS's matching depth, i.e., *CS.depth*; the counter is initialized to *CS.depth*. The matching depth *CS.depth* is initialized and updated by Dimmunix, as shown in §3.4. The matching is successful only if the depth d of the currently matched frame in CS is equal to *matches[CS, t]* (line 6). If $d = CS.depth$, the matching restarts, i.e., the counter is reset to $d - 1$ (lines 3–4). Thread t's execution matches CS up to *CS.depth* if and only if *matches[CS, t] = 0* (line 9).

Inline matching means accepting non-contiguous matches, i.e., extraneous frames in a thread's call stack are allowed, as long as the frames referenced by a signature are in the correct order. Dimmunix is oblivious to method calls outside an outer call stack, because they are not instrumented for matching. Since this matching mechanism is less accurate than the straightforward one, the number of false positives may increase; however, the inline matching may protect the application against deadlock manifestations that are not yet captured by the signature.

3.4 Reducing the Number of False Positives

Approaches that try to predict the future with the purpose of avoiding bad outcomes may suffer from false positives (FP), i.e., wrongly predict that the bad outcome will occur. In Dimmunix, FPs can arise when the outer call stacks of a signature are matched too shallowly, or when the lock order depends on inputs, program state, etc.

When a FP occurs, Dimmunix serializes threads in order to avoid an apparent impending deadlock that would actually not have occurred; this can have negative effects on performance, due to a loss in parallelism. Dimmunix "needlessly" serializes a portion of the program execution, causing the program to run slower.

Dimmunix reduces the number of FPs as follows: whenever a deadlock signature S is avoided, Dimmunix checks if the avoidance of S's instantiation was a FP. If it was, then Dimmunix recalibrates the matching accuracy for S.

Detecting False Positives Dimmunix determines whether forcing a thread to yield indeed avoided a deadlock or not, by looking for lock inversions after the yield. A false positive (FP) is a situation where the deadlock could not have happened, under any thread interleaving, for the current program inputs, even if Dimmunix had not avoided the deadlock. Since a yield represents the avoidance of a signature instantiation, Dimmunix associates the notion of false positive with a signature instantiation. Dimmunix classifies an instantiation I as a FP when no lock inversion occurred after avoiding I.

The following data structures are used for FP detection: in a signature instantiation $I = \{(t_1, l_1, CS_1), ..., (t_n, l_n, CS_n)\}$, Dimmunix keeps for each lock l_i the set $I.locksAcq[l_i]$ of locks acquired while holding l_i; for each thread t and lock l, Dimmunix stores the set *instances[t, l]* of signature instantiations involving t and l that Dimmunix avoided since t acquired l last time. Dimmunix initializes with *null* each set $I.locksAcq[l_i]$ when I is constructed, and updates the set only when l_i is released. If a lock l_i is reacquired before I is analyzed, Dimmunix does not change the set $I.locksAcq[l_i]$, i.e., it freezes the set as soon as l_i is released. We denote by *I.sig* the signature instantiated by I.

When a thread t is about to release a lock l, Dimmunix analyzes every signature instantiation $I = \{(t_1, l_1, CS_1)$, ..., $(t_n, l_n, CS_n)\}$ from the *instances[t, l]* set to determine whether it was a false positive (FP), as illustrated in Algorithm 4. When all the sets $I.locksAcq[l_i]$ are non-null, it means that all the locks l_i have been released, and Dimmunix analyzes I (line 4). Dimmunix classifies I as a FP if and only if there is no lock inversion in I, i.e., $l_1 \notin I.locksAcq[l_2]$, ..., or $l_n \notin I.locksAcq[l_1]$ (lines 5–6).

Classifying an instantiation I as an FP if no lock inversion occurred is sound, under the assumption that the thread scheduling does not affect lock identities and expressions controlling the inner lock statements (e.g., through data races).

Calibrating the Signature Matching Accuracy A signature S captures all the possible manifestations of a deadlock bug if and only if all the possible signatures of the same deadlock match S up to the matching depths of its outer call stacks. Choosing too large matching depths can cause Dimmunix to miss manifestations of the deadlock, while choosing too shallow ones can lead to mispredicting a runtime call flow as being headed for deadlock (i.e., a FP).

```
    Input: Thread $t$ releasing $l$; Set $instances[t, l]$; Sets $I.locksAcq[l_i]$ for each instantiation
           $I = \{(t_1, l_1, CS_1), ..., (t_n, l_n, CS_n)\}$ in $instances[t, l]$.
    Output: Number of FPs $numFPs[S]$ corresponding to each signature $S$.
1   // before releasing $l$, check if the instantiations avoided by $t$ before $l$'s acquisition were FPs
2   foreach $I = \{(t_1, l_1, CS_1), ..., (t_n, l_n, CS_n)\} \in instances[t, l]$ do
3       // if all the locks involved in $I$ were released
4       if $\forall i \in [1, n]\ : I.locksAcq[l_i] \neq null$ then
5           if $\exists i \in [1, n]\ s.t.\ l_i \notin I.locksAcq[l_{(i+1)\%n}]$ then
6               $numFPs[I.sig] := numFPs[I.sig] + 1$
7   unlock(l)
```
Algorithm 4: *detectFPs(t, l)*: checks if the signature instantiations that t avoided last time it requested l were FPs.

We now describe how Dimmunix calibrates the matching depths at runtime to reduce FPs while maintaining effectiveness. When a signature S is created, the matching depths of its outer call stacks are set to 1. Hence, S initially captures all the possible manifestations of the deadlock bug. Every time a FP is encountered when avoiding an instantiation of S, the matching depths of S's outer call stacks are incremented.

A scenario where dynamically increasing the matching precision helps is one when an application uses synchronization wrappers, and the lock acquisitions always execute at the same program position. Keeping the matching depth at 1 serializes all the critical sections, which is not desirable. Increasing the matching depth dynamically when FPs are encountered solves this problem.

When the matching depth becomes too large, a signature may not capture all the possible manifestations of the deadlock bug, because there may exist other signatures of the same deadlock bug ending in call stack suffixes that no longer match S. To prevent this situation, Dimmunix merges signatures.

Dimmunix merges the signature S' of a new manifestation of a deadlock bug with the existing signature S of the same deadlock as follows: first, it finds the common suffix of maximum length of the outer call stacks of S and S'. Then, Dimmunix decrements the matching depths for S's outer call stacks to this length, and freezes them. Finally, Dimmunix discards signature S', to keep the deadlock history at a minimal size. If the deadlock reoccurs, with another signature S'', Dimmunix merges S and S'', and so on. This way, Dimmunix finds the deepest matching depth for S, while preserving the ability to avoid all the possible manifestations of the deadlock bug.

From our experience, the number of signatures corresponding to a deadlock bug is low, the maximum being two. If a deadlock bug has few signatures, it takes few occurrences of the deadlock to converge to an optimal matching precision. However, if a deadlock bug has many manifestations with different outer call stack suffixes, Dimmunix will most likely need to encounter only a couple of them to fully protect the application against the deadlock bug, because with each newly discovered signature the calibration algorithm decreases the matching depth of the original signature.

```
    **Input**: Signature $S$, with outer lock statements $s_1^{out},...,s_n^{out}$ and inner lock statements
            $s_1^{in},...,s_n^{in}$.
    **Data**: Control flow graph (CFG).
    **Output**: The set of escape branches.
1   $escape := \emptyset$
2   **foreach** $i \in [1,n]$ **do**
3       **foreach** branch statement $s \in CFG\ s.t.\ s_i^{out} \rightsquigarrow s$ **do**
4           Let $B$ be the set of branches of $s$
5           **if** $\exists b \in B\ s.t.\ b \rightsquigarrow s_i^{in}$ **then**
6               $escape := escape \cup \{b' \in B \mid \neg b' \rightsquigarrow s_i^{in}\}$
7   **return** $escape$
```

Algorithm 5: *findEscapeBranches(S)*: finds the escape branches for a signature S.

3.5 Reducing the Yielding Time

To reduce the duration of a yield, Dimmunix exploits the branches that escape the deadlock, i.e., branches that lead the program away from acquiring the inner lock that triggers the deadlock. When such a branch is taken, Dimmunix stops the avoidance process by canceling the active yields and preventing future ones, until the lock whose acquisition triggered the avoidance is released.

Given the outer and inner call stacks of a deadlock signature, Dimmunix statically detects in the CFG of the application bytecode the “escape branches” that bypass the deadlock; we illustrate this mechanism in detail in Figure 5. To determine these branches, Dimmunix first finds the “critical branches” that need to be taken in order to reach the inner lock statements from the outer lock statements (lines 3–5). If a conditional statement has one or more critical branches (line 5), the remaining branches (if any) are escape branches (line 6). We use the notation $x \rightsquigarrow y$ to denote the fact that statement y is reachable from statement x in the CFG.

Dimmunix inserts code to stop the avoidance process at the escape branches and right after the inner lock statement, if that statement is not in a loop. Since the deadlock situation cannot be reached from these positions, the yielding threads can be safely resumed. If a deadlock occurs due to stopping the avoidance, that deadlock will have different inner lock statements, and therefore it is a new deadlock bug. A new signature is constructed for this deadlock.

4 Evaluation

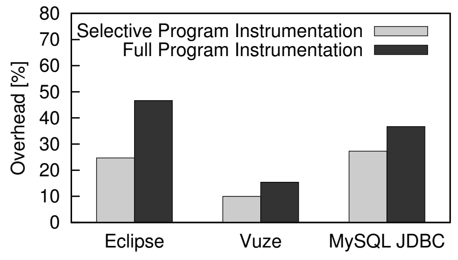
The goal of this section is to assess the performance improvements that result from employing the five optimizations described above.

First, we evaluate the benefits brought by the optimizations in synchronization-intensive scenarios on three real-world applications: Eclipse IDE, Vuze BitTorrent client, and MySQL JDBC. We found that Eclipse and Vuze are synchronization-intensive at startup: they perform 78,536 and respectively 28,872 synchronization operations per second. For MySQL JDBC, we used the JDBCBench benchmark, which performs 100,855 synchronization operations per second.

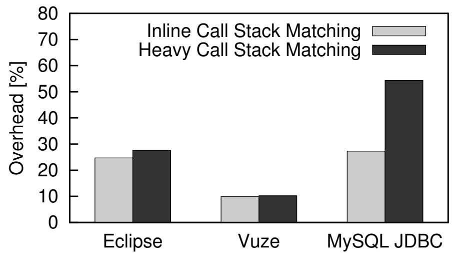
The measurements for the three applications are end-to-end: for Eclipse and Vuze, we compute the runtime overheads introduced by various Dimmunix configurations by comparing the time it takes for the application to start and immediately shut down when Dimmunix is running to the time it takes without Dimmunix; for MySQL JDBC, we compare the number of transactions performed when Dimmunix is running to the number of transactions when Dimmunix is not running.

Our experiments explore Dimmunix's behavior in worst-case scenarios, even though they are unlikely to manifest during steady state operation. In the original article [3] we focus instead on realistic steady state scenarios. A worst-case scenario is one with a high rate of synchronization operations and with deadlock signatures that cover frequently executed nested synchronization statements. For our experiments, we manually generate 20 such signatures for deadlocks involving two threads. By default, Dimmunix is configured to use selective instrumentation, FP detection and matching depth calibration, inline call stack matching, and an initial matching depth of 5.

To measure the benefit of selective instrumentation, we compare the overhead introduced by Dimmunix with and without selective instrumentation. Figure 1a shows that this optimization reduces the overhead caused by Dimmunix by a factor of 1.3x–1.9x.



(a) Selective program instrumentation.



(b) Inline call stack matching.

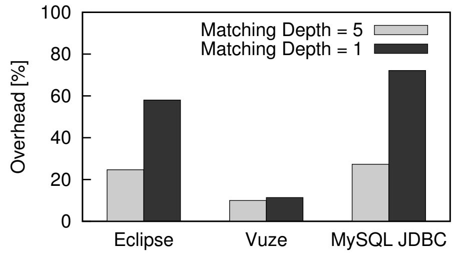
Fig. 1: Benefit of selective instrumentation and inline call stack matching.

To measure the benefit of the inline call stack matching, we compare the use of Java's *getStackTrace()* method for call stack matching against the default configuration. As Figure 1b shows, this optimization reduces the runtime overhead by up to 2x.

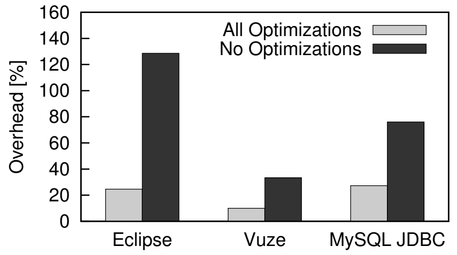
To measure the usefulness of a high signature matching accuracy, we change the initial matching depth to 1 in the default configuration. As Figure 2a shows, increasing the matching depth from 1 to 5 reduces the overhead by a factor of up to 2.6x.

We evaluate the benefit of enabling selective instrumentation and inline call stack matching together, by comparing the runtime overhead introduced by Dimmunix when both optimizations are missing against the default configuration. Figure 2b shows that the effects of the two optimizations compound: together, they reduce Dimmunix's overhead by a factor of 2.8x–5.2x. The performance improvement is higher compared to the sum of the improvements brought by the individual optimizations. The explanation is that using heavy call stack matching for all the synchronization statements is worse than using it for only several synchronization statements.

Although the signatures we generated end in program positions where most synchronization operations execute, they are not instantiated often because the applications' threads seldom synchronize as described in our signatures. Therefore, the benefit of automatically detecting FPs and increasing the matching accuracy, and exploiting the



(a) High matching accuracy.



(b) All optimizations.

Fig. 2: Benefit of high matching accuracy and enabling all the optimizations at once.

escape branches is marginal. We evaluate the effectiveness of these two optimizations on a separate microbenchmark in which signatures are instantiated often.

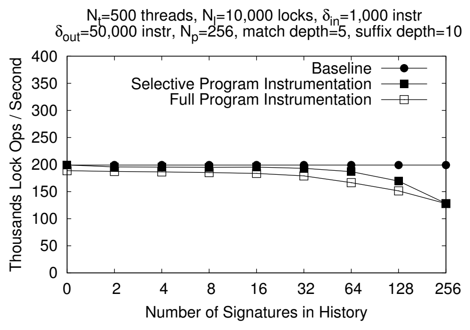
To dissect Dimmunix's performance behavior and understand how it varies with various parameters, we wrote a synchronization-intensive microbenchmark that creates N_t threads that synchronize on locks from a total of N_l shared locks. A thread acquires a lock by executing one of N_p lock acquisition statements, then executes δ_{in} statements, then releases the lock, then executes δ_{out} statements, then acquires another lock. The δ_{in} and δ_{out} delays are implemented as busy loops that execute incrementation statements, to simulate computation. The threads call multiple functions within the microbenchmark so as to build up different call stacks; which function is called is chosen randomly, generating a uniformly distributed selection of call stacks.

We also wrote a tool that generates synthetic deadlock history files containing H signatures of size 2 (the usual number of threads involved in a deadlock [4]). The H signatures cover H lock acquisition statements. If $H = N_p$, then all the lock acquisition statements in the microbenchmark are instrumented. Each signature has two identical call stacks that consist of combinations of the microbenchmark's methods—not signatures of real deadlocks, but avoided as if they were.

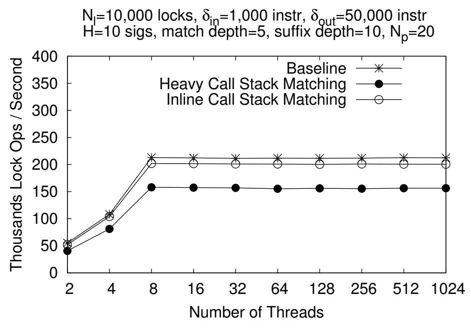
Figure 3a shows that the selective program instrumentation is effective for up to 64 signatures. The overhead of Dimmunix with selective instrumentation is 0–6.1% compared to 5.2–16.4% for full instrumentation. With an empty history, there is no overhead if Dimmunix uses selective instrumentation, while with full instrumentation, the overhead is already 5.2%, comparable to selective instrumentation with 64 signatures, because with full instrumentation Dimmunix performs signature matching at program positions where it is not needed. For more signatures, the overhead increases rapidly.

Figure 3b shows that inline call stack matching considerably reduces performance overhead: if Dimmunix uses the JVM's call stack retrieval, the overhead is 26–27%; if the call stack matching is inlined, the overhead goes down to 4–5%.

To measure the effect of detecting false positives and calibrating the signature matching precision, we first show the effect of false positives (FPs) on performance. A FP causes a thread to needlessly yield, decreasing the rate of synchronization operations. Since our microbenchmark has no real deadlocks, all yields are unnecessary. We compute the overhead caused by FPs by comparing the rate of synchronization operations



(a) Selective instrumentation.

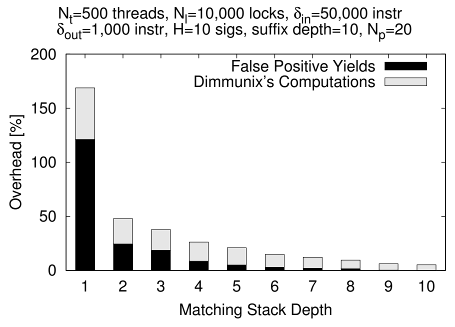


(b) Inline call stack matching.

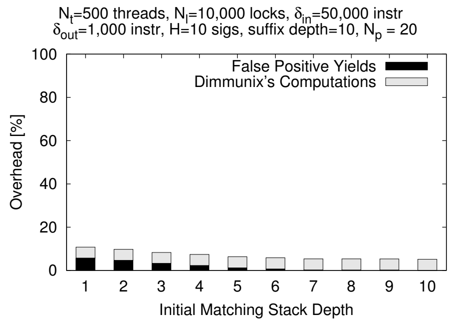
Fig. 3: Results for selective instrumentation and inline call stack matching.

performed when Dimmunix detects that a deadlock will manifest, but takes no avoidance actions, to the same rate when Dimmunix suspends threads to avoid deadlocks.

Figure 4a shows the results: as the matching depth increases, the overhead induced by FPs decreases. For a matching depth of 1, the overhead due to FP yields is 121%. For a matching depth of 4, the overhead due to FP yields drops by 14x, to 8.56%.



(a) Overhead due to false positives.



(b) Matching depth calibration.

Fig. 4: Detecting false positives and calibrating the signature matching precision.

Figure 4b shows the benefit of increasing the matching accuracy as FPs are encountered. Compared to the configuration used for Figure 4a, here Dimmunix is configured to dynamically calibrate the matching depths. If we compare the two figures, the benefit of dynamically increasing the matching accuracy is evident: the overhead becomes acceptable even for an initial matching depth of 1 (i.e., 5.7%).

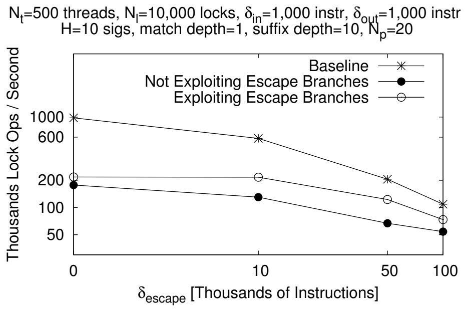


Fig. 5: Exploiting escape branches.

Figure 5 shows the benefit of exploiting escape branches to reduce yielding time. The benefit is substantial if (a) the number of instructions on the escape paths that bypass the deadlock (δ_{escape}) is substantially larger than the number of instructions in the critical section preceding the escape branches, i.e., δ_{in}; (b) signatures are instantiated very often, i.e., the matching depth is low and δ_{out} is small; and (c) there are no FPs, even for shallow matching depths. We disable the matching depth calibration, in order to simulate the scenario in which there are no FPs. For $\delta_{escape} = 0$, there is no benefit in exploiting escape branches. For $\delta_{escape} = 10,000$ (respectively 50,000 and 100,000), the overhead is 77% (respectively 67% and 50%) when escape branches are not exploited, compared to 63% (respectively 40% and 32%) when exploiting escape branches.

5 Related Work

There is a spectrum of runtime techniques for avoiding or preventing deadlocks, i.e., techniques that (1) statically detect potential deadlocks and avoid them at runtime; (2) dynamically prevent deadlocks; (3) transparently recover from deadlocks; and (4) techniques that provide deadlock immunity.

Approaches like [1] and Gadara [8] detect potential deadlocks statically and avoid them at runtime. In [1], authors propose to use new locks, while Gadara uses Petri nets. If the static analysis has false positives, these approaches make the applications avoid false deadlock bugs. This is not the case for Dimmunix, because it avoids only previously detected at runtime. Unlike the two approaches, Dimmunix requires no source code. Unlike Dimmunix, Gadara needs source code annotations from the developers to filter out the false positives, yet this is difficult.

A dynamic deadlock prevention technique is [10], which modifies the JVM to serialize threads' accesses to sets of locks acquired in a nested fashion. There are a couple of shortcomings of this approach: First, the lock acquired at a particular program location can change during the same execution; if it changes often (e.g., it may correspond to an array element), then [10] is not effective. Every time new locks are used, [10] has to update the lock sets. Whenever the lock sets are not up to date, the program is vulnerable to deadlocks. Second, the lock sets are not reusable in future runs: in each run, [10] will have to restart the learning process from scratch. Dimmunix eliminates these shortcomings by abstracting the locks involved in a deadlock to call stacks.

Sammati [6] dynamically detects deadlocks and transparently recovers the applications from deadlocks by executing critical sections in isolation from other threads. If a deadlock happens during the execution of a critical section, the updates performed within the scope of that section up to the deadlock are discarded, in effect rolling back the critical section. Since Sammati is essentially a TM customized for deadlock recovery, the TM challenges (e.g., large critical sections, I/O) apply to Sammati as well.

Deadlock immunity approaches include [5, 9]. These approaches dynamically detect deadlocks, then avoid future occurrences of the same deadlocks. If a deadlock involving threads t_1 and t_2 and locks l_1 and l_2 occurs, the two approaches save the program positions p_1 and p_2 (where l_1 and l_2 were acquired) into the signature of the deadlock; [9] saves, in addition, the positions p'_1 and p'_2 where t_1 and t_2 deadlocked. In future runs, [5, 9] prevent the deadlock from reoccurring by acquiring a "gate lock" every time the lock statement at p_1 or p_2 is about to execute. If the lock at p'_1 (or p'_2) can be soundly inferred at runtime from p_1 (respectively p_2), [9] swaps the lock acquisitions at p_1 and p'_1 (respectively p_2 and p'_2), instead of acquiring a gate lock. The latter avoidance mechanism is difficult in the general case, because predicting which lock objects will be used is undecidable. Therefore, speculatively acquiring the lock at p'_1 (or p'_2) does not guarantee that the deadlock will be avoided.

Dimmunix shares ideas with [5, 9], but uses a more accurate avoidance mechanism. Like in these approaches, Dimmunix's deadlock avoidance mechanism relies on temporarily suspending threads. Dimmunix has fewer false positives, compared to these techniques, thus alleviating the problem of lost parallelism. Finally, the efficiency of Dimmunix's critical-path computations is comparable to acquiring a gate lock.

6 Conclusion

In this paper, we presented the optimizations we brought to Dimmunix, a system that enables applications to defend themselves against deadlocks.

We reduce the overhead introduced by Dimmunix's deadlock detection by performing it offline, when the program terminates. We optimize the deadlock avoidance by (1) performing selective program instrumentation to confine monitoring to only lock statements previously involved in deadlocks, (2) inlining the matching of signatures, (3) reducing the number of false positives, and (4) aborting deadlock avoidance when the deadlock situation becomes unreachable.

We implemented these optimizations for the Dimmunix prototype targeting Java applications. Our evaluation shows that these optimizations significantly reduce the overhead Dimmunix incurs on Java applications.

References

[1] Boronat, P., Cholvi, V.: A transformation to provide deadlock-free programs. In: Intl. Conf. on Computational Science (2003)

[2] Fonseca, P., Li, C., Singhal, V., Rodrigues, R.: A study of the internal and external effects of concurrency bugs. In: Intl. Conf. on Dependable Systems and Networks (2010)

[3] Jula, H., Tralamazza, D., Zamfir, C., Candea, G.: Deadlock immunity: Enabling systems to defend against deadlocks. In: Symp. on Operating Sys. Design and Implem. (2008)

[4] Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In: Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (2008)

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

[6] Pyla, H.K., Varadarajan, S.: Avoiding deadlock avoidance. In: Proceedings of the 19th international conference on Parallel Architectures and Compilation Techniques (PACT) (2010)

[7] Song, X., Chen, H., Zang, B.: Why software hangs and what can be done with it. In: Intl. Conf. on Dependable Systems and Networks (2010)

[8] Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., Mahlke, S.A.: Gadara: Dynamic deadlock avoidance for multithreaded programs. In: Symp. on Operating Sys. Design and Implem. (2008)

[9] Zeng, F.: Pattern-driven deadlock avoidance. In: Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD) (2009)

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