Deadlock Immunity: Enabling General-Purpose Software to Defend Itself against Deadlocks

THÈSE N^{O} 5146 (2011)

PRÉSENTÉE LE 18 AOÛT 2011

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DES SYSTEMES FIABLES

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Horatiu JULA

acceptée sur proposition du jury:

Prof. C. Petitpierre, président du jury
Prof. G. Candea, directeur de thèse
Prof. R. Bianchini, rapporteur
Prof. P. Felber, rapporteur
Prof. V. Kuncak, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2011

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Prof. George Candea) Principal Adviser

Résumé

L'immunité contre les interblocages est une propriété par laquelle les programmes, après être affectés par un interblocage une première fois, développent une résistance contre les interblocages futurs. Nous donnons la possibilité aux programmes d'être immunisés contre les interblocages impliquant des exclusions mutuelles, des sémaphores, des verrous de lecture-écriture, des initialisations de classes, et des synchronisations externes, le tout sans aucune intervention de la part de l'utilisateur et sans modifier la sémantique du programme. En d'autres termes, nous proposons une immunisation transparente et non-intrusive contre les interblocages.

Notre système d'immunisation, appelé Dimmunix, est disponible pour les applications Java, les applications C/C++ qui utilisent des threads POSIX, et le système d'exploitation Android. Dimmunix pour Java offre une immunité au niveau de l'application. Dimmunix pour les threads POSIX offre une immunité à toutes les applications C/C++ qui utilisent la bibliothèque de threads POSIX. Enfin, Dimmunix pour Android offre une immunité à toutes les applications fonctionnant sur le système Android. Nous avons aussi développé Communix, une plateforme d'immunité distribuée grâce à laquelle les machines connectées à Internet coopérent les unes avec les autres pour s'immuniser mutuellement contre les interblocages.

Puisque les verrous d'exclusion mutuelle sont vraisemblablement les primitives de synchronisation les plus utilisées, nous nous intéressons principalement aux interblocages impliquant les exclusions mutuelles. Le prototype de Dimmunix pour Java est une implémentation complète de Dimmunix. Les autres prototypes ne supportent que les interblocages dus aux verrous d'exclusion mutuelle. Nous avons optimisé la gestion des exclusions mutuelles dans Dimmunix pour Java afin de gérer efficacement les applications qui utilisent

de manière intensive ces primitives de synchronisation.

Nos prototypes de Dimmunix sont efficaces contre les interblocages rapportés dans des applications réelles, telles que Limewire, MySQL JDBC, ActiveMQ, MySQL Server, et SQLite. Le prototype de Dimmunix pour Java fonctionne efficacement sur les applications à synchronisation intensive, comme JBoss et Eclipse, qui sont constituées de millions de lignes de code et de centaines de threads. Nous montrons aussi que les prototypes de Dimmunix pour Java pour les autres types d'interblocages, Dimmunix pour les threads POSIX, Dimmunix pour Android, et Communix sont efficaces. De plus, le prototype de Communix arrive à contenir les attaques par déni de service qui essayent d'exploiter la plateforme d'immunité collaborative.

Mots-clés: interblocages, détection, évitement, immunité, collaborative

Preface

Deadlock immunity is a property by which programs, once afflicted by a deadlock, develop resistance against future occurrences of that deadlock. We enable real applications to automatically achieve immunity against deadlock bugs involving mutex locks, semaphores, read-write locks, class initialization, and external synchronization, with no user intervention, and without changing the semantics of the applications. In other words, we provide transparent, non-intrusive immunization against deadlocks.

Our deadlock immunity system, called Dimmunix, is available for Java applications, C/C++ applications using POSIX Threads, and Android OS. Java Dimmunix provides application-level immunity, i.e., deadlock immunity to individual Java applications. POSIX Threads Dimmunix provides library-level immunity, i.e., deadlock immunity to all the C/C++ applications using the POSIX Threads library. Android Dimmunix provides platform-level immunity, i.e., deadlock immunity to all the applications running within the Android OS. We also developed, on top of Java Dimmunix, a collaborative deadlock immunity framework called Communix, in which machines connected to the Internet collaborate to immunize each other against deadlocks.

Since mutex locks are likely the most widely used synchronization construct, we focus on mutex deadlocks, i.e., deadlocks involving mutexes. The Java Dimmunix prototype is a complete implementation of Dimmunix; the other prototypes handle only mutex deadlocks. We heavily optimized the part of the Java Dimmunix prototype responsible for mutex deadlocks, in order to efficiently handle synchronization-intensive applications.

Our prototype implementations are effective against deadlocks reported in real applications, like Limewire, MySQL JDBC, ActiveMQ, MySQL Server, and SQLite. The Java

Dimmunix prototype for mutex deadlocks runs efficiently on synchronization-intensive applications, like JBoss and Eclipse, with millions of lines of code and hundreds of threads. We also show that Java Dimmunix for non-mutex deadlocks, POSIX Threads Dimmunix, Android Dimmunix, and Communix are efficient. Moreover, the Communix prototype manages to contain denial of service attacks that attempt to exploit the collaborative immunization framework.

Keywords: deadlocks, detection, avoidance, immunity, collaborative

Contents

List of Tables

List of Figures

Chapter 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. Programmers consider concurrency bugs to be some of the most insidious and, not surprisingly, a large number of bugs are related to concurrency [Lu et al., 2008].

To ensure the consistency of the computations performed on the state shared by different threads, programmers use synchronization mechanisms like locks (mutexes), condition variables, semaphores, or busy waiting (i.e., loops that periodically inspect/update the shared state). We describe these synchronization mechanisms in §3.1.

When threads do not coordinate correctly in their use of synchronization, a deadlock can occur—a situation whereby a group of threads cannot make progress (i.e., hang), because each thread is waiting for other threads in the group to perform some action, i.e., release a lock/semaphore, finish some computation, or update some shared state. In other words, a deadlock involves circular waits for resources or events. A deadlock can also occur between separate processes (i.e., threads that do not implicitly share state), if the synchronization is performed on resources (events) that are external to the running process, e.g., file locks.

Surveys Lu et al. [2008], Fonseca et al. [2010], Song et al. [2010] show that an important fraction of concurrency failures (i.e., failures caused by concurrency bugs) are deadlocks,

i.e., 38–55%. Lu et al. [2008] studied 105 concurrency bugs from four representative applications: the MySQL database server, the Apache web server, the Mozilla web browser, and the OpenOffice office suite. Song et al. [2010] analyzed 233 hangs from the MySQL and PostgreSQL database servers, the Apache web server, and the Firefox web browser. Fonseca et al. [2010] analyzed 80 concurrency bugs from the MySQL database server. We summarize in Table 1.1 the results of these surveys concerning deadlocks. According to the surveys, most of the hangs are deadlocks, i.e., 55–84%. The surveys also show that deadlock hangs represent an important fraction of the concurrency failures, i.e., 38–55%. The non-deadlock hangs are different: they cannot be represented as circular waits (§3.3.10). For instance, scenarios where a client does not receive a reply for a request, or a thread waits indefinitely for a message/event that it missed, are not deadlocks. The surveys also study the impact of non-deadlock concurrency bugs. Lu et al. [2008] found 34 crashes and 6 non-deadlock hangs caused by atomicity and order violations, as we show in Table 1.1. Song et al. [2010] found 26 non-deadlock hangs, out of which 13 are caused by data races, and 13 have other causes (e.g., missed messages/notifications). In the MySQL database server, Fonseca et al. [2010] found 22 crashes, 6 hangs, 19 byzantine failures, and 5 performance degradations that are due to non-deadlock concurrency bugs.

Survey	Deadlock hangs	Non-deadlock hangs	Other concurrency failures	Deadlocks (%)
Lu et al. [2008]	31	6	34	44%
Song et al. [2010]	32	26		55%
Fonseca et al. [2010]	32	6	46	38%

Table 1.1: Deadlocks are an important fraction of the concurrency failures in real-world software.

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 verifying large programs.

More importantly, 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. Furthermore, modern systems are increasingly designed to accommodate extensions written by third parties, which can introduce new deadlocks into the target systems (e.g., Web browser plugins, applications based on 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, because that would 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 [Lu et al., 2008]. Although progress has been made in testing and debugging concurrent programs [Zamfir and Candea, 2010, Musuvathi et al., 2008], it may take months to properly fix concurrency bugs [Fonseca et al., 2010].

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. At the same time, existing applications achieve higher degrees of runtime concurrency. There exist proposals for making concurrent programming easier, such as transactional memory [Herlihy and Moss, 1993], but issues surrounding I/O and long-running operations make it difficult to provide atomicity transparently. The Scala programming language proposes actors [Sca], as the primary concurrency construct. Actors are basically concurrent processes that communicate by exchanging messages. However, message passing systems are also subject to deadlocks, as any concurrency construct that involves waiting for resources or events.

Our goal is to help applications defend themselves against deadlocks by providing them with *deadlock immunity*—a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of that deadlock, with no assistance from programmers or users. In other words, deadlock immunity enables applications to

avoid deadlocks that they previously encountered.

We address five types of deadlocks: *mutex deadlocks* involving synchronization on mutex locks, *semaphore deadlocks* involving synchronization on semaphores, *read-write deadlocks* involving synchronization using read-write locks, *initialization deadlocks* involving mutexes together with class initialization code, and *external deadlocks* involving synchronization on external objects (e.g., file locks). We define these deadlock patterns in §3.3. There exist other types of deadlocks, such as self-deadlocks and wait-notify deadlocks (§3.3), but they cannot be avoided without altering the semantics of the applications (§6.1). There exist also starvation situations which may lead to deadlocks, like blocked notifications, ad-hoc deadlocks, and join deadlocks (§3.3). These bugs are at the frontier between starvation and deadlock; avoiding them automatically is hard (§6.2).

We propose a technique that provides deadlock immunity to applications without any assistance from users or programmers. We implemented this technique in a system called Dimmunix, which is available for Java applications, C/C++ applications using POSIX Threads, and Android OS. Java Dimmunix provides application-level immunity, i.e., deadlock immunity to individual Java applications. POSIX Threads Dimmunix provides library-level immunity, i.e., deadlock immunity to all the C/C++ applications using the POSIX Threads library. Android Dimmunix provides platform-level immunity, i.e., deadlock immunity to all the applications running within the Android OS. Java Dimmunix provides immunity against all the five deadlock types. The other two implementations provide immunity against mutex deadlocks. We also developed, on top of Java Dimmunix, a collaborative deadlock immunity framework called Communix, in which machines connected to the Internet collaborate to immunize each other against deadlocks.

Dimmunix has two modules running simultaneously: a detection module dynamically detects deadlocks and extracts their "fingerprints"; an avoidance module uses the fingerprints as antibodies to avoid future occurrences of these deadlocks. The fingerprint is an approximation of the execution flow that led to deadlock. To avoid a previously discovered deadlock, Dimmunix temporarily suspends threads whenever their execution is about to match the fingerprint of that deadlock.

Dimmunix is purely software-based (i.e., does not require any cooperation from the

hardware) and does not require any changes to the application's source code. Java Dimmunix automatically instruments the synchronization statements; POSIX Threads Dimmunix and Android Dimmunix intercept the synchronization operations within the synchronization libraries (primitives).

Dimmunix is effective against deadlock bugs in real applications, like the JBoss application server, the Limewire peer-to-peer file sharing system, and the MySQL database server, while incurring a low performance overhead (§10). In the absence of deadlocks, the performance overhead is negligible. In the presence of deadlocks, the performance overhead is highly dependent on the location of the deadlock-prone code within the target application (e.g., if the synchronization statements involved in deadlocks are on the critical path, they are executed often, and Dimmunix may incur a noticeable performance overhead).

Dimmunix avoids deadlocks without altering the semantics of non-real-time applications, because the deadlock avoidance consists merely of altering the thread schedule. However, there exist deadlock types that cannot be avoided without changing the semantics of the application (§6.1); Dimmunix only detects such deadlocks, but does not avoid them. Dimmunix does not handle distributed deadlocks. Dimmunix handles only deadlocks occurring on one machine, among different threads of the same process or among different processes.

For purposes of illustration, consider the use of Dimmunix in the following two realistic scenarios. In the first scenario, the Firefox web browser deadlocks because of a plugin (e.g., the Flash plugin) every time it renders a particular web page. With Dimmunix, the browser will be able to render the page after the first time this happens and the user kills the deadlocked Firefox process and restarts it. In the second scenario, the Eclipse IDE deadlocks every time at startup due to a deadlock bug in a plugin. Without Dimmunix, Eclipse is unusable, unless the plugin is manually removed. If Eclipse runs with Dimmunix, it will deadlock only the first time it starts; after Dimmunix obtains the fingerprint of the deadlock, Eclipse will not deadlock anymore due to that bug.

We recommend Dimmunix for general-purpose systems, such as desktop and enterprise applications, or server software. For general-purpose applications, we consider deadlock immunity to be almost as useful as preventing all deadlocks. However, Dimmunix can

cause undue interference in real-time systems. Safety-critical 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 to run deadlock-free. Normally, Dimmunix does not affect functionality in general-purpose applications; however, interference with wait-notify schemes or features that rely on code that executes concurrently in separate threads can impede functionality (§6).

Dimmunix can be used by both software vendors and end users alike. On the one hand, vendors faced with the current impossibility of shipping large software that is bug-free could instrument their ready-to-ship software with Dimmunix and get an extra safety net: Dimmunix can keep users happy by allowing them to use the deadlock-prone system while the developers try to fix the bugs. On the other hand, 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 low-cost "band-aid" with many practical benefits.

The thesis is structured as follows: In Chapter 2, we summarize the related work. In Chapter 3, we provide the necessary background information. In Chapter 4, we explain how Dimmunix detects deadlocks and extracts their fingerprints. In Chapter 5, we explain how Dimmunix avoids deadlocks based on these fingerprints. In Chapter 6, we describe the properties and limitations of Dimmunix. In Chapter 7, we present Communix, our collaborative deadlock immunity framework. In Chapter 8, we describe optimizations that reduce the performance overhead incurred by applications using Dimmunix. In Chapter 9, we provide implementation details of Java Dimmunix, POSIX Threads Dimmunix, Android Dimmunix, and Communix. In Chapter 10, we evaluate these prototype implementations, and in Chapter 11 we conclude.

Chapter 2

Related Work

There is a spectrum of approaches for avoiding (preventing) deadlocks: language-level deadlock prevention (§2.1), static deadlock detection (§2.2), hybrid techniques that statically detect potential deadlocks and avoid them dynamically (§2.3), approaches that dynamically detect potential deadlocks and avoid them, or transparently recover from deadlocks (§2.4), and finally techniques that provide deadlock immunity (§2.5). We also present techniques that target other bugs, like buffer overruns, data races, and atomicity violations (§2.6). 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.

2.1 Language-level Deadlock Prevention

Language-level approaches [Boyapati et al., 2002, Lampson and Redell, 1980] use powerful type systems to simplify the writing of lock-based concurrent programs and thus avoid synchronization problems altogether. For instance, Boyapati et al. [2002] uses a variant of ownership types to prevent data races and deadlocks; the programmers partition all the locks into a fixed number of lock levels and specify a partial order among the lock levels. The type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the decreasing order of their lock levels. Such approaches avoid the runtime performance overhead and prevent 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) [Herlihy and Moss, 1993] 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. TMs allow programmers to define atomic sections (i.e., code regions that execute atomically). The purpose of the TMs is to guarantee that there are no data races among atomic sections. According to the TM semantics, the execution of an atomic section is implicitly deadlock-free, because there is no explicit synchronization among atomic sections. TM optimistically executes the atomic sections in isolation. If no data races (conflicts) are found, the updates performed in the atomic sections are committed, i.e., made visible to the other threads; if there is a data race, the TM rolls back one of the atomic sections involved in the data race and restarts it. 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 Koskinen and Herlihy [2008] shows that many modern TM implementations use lock-based techniques to improve performance and are thus subject to deadlock. So far, TM implementations have not been proven to work out-of-the-box for real applications, i.e., by enabling the replacing all critical sections with transactions. Thus, we believe that, while TM is powerful, it cannot address all the concurrency problems in real systems.

2.2 Static Deadlock Detection

Static analysis tools look for deadlocks at compile time and help programmers remove them. ESC [Flanagan et al., 2002] uses a theorem prover and relies on annotations to provide knowledge to the analysis; Houdini [Flanagan and Leino, 2001] helps generate some of these annotations automatically. RacerX [Engler and Ashcraft, 2003] and Williams et al. [2005] use flow-sensitive analyses to find deadlocks. In Java JDK 1.4, the tool described in [Williams et al., 2005] 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 statically finding deadlocks is to use model checkers, which systematically explore all the possible executions 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 (JPF) [Java PathFinder], one of the most successful model checkers, offers limited support for Java I/O libraries and no support for Java network and graphical libraries. In [Java PathFinder], it was reported that JPF was able to find bugs in an internal 1 MLOC Fujitsu application. The application needed to be converted first into a fully executable stand-alone Java model, which was then checked by JPF; JPF explored 125,000 program states and millions of execution paths. However, it is not clear whether all the possible execution paths were explored. Real-world applications are large (e.g., MySQL has >1 MLOC) and use I/O, network, and/or graphical libraries; this restricts the use of model checking in the development of general-purpose systems. To the best of our knowledge, there are no model checkers that can explore all the possible execution paths in large applications.

2.3 Static Detection and Dynamic Avoidance of Deadlocks

There exist hybrid approaches, like [Boronat and Cholvi, 2003] and Gadara [Wang et al., 2008], which detect deadlock potentials (i.e., code that might be deadlock-prone) statically, then avoid at runtime the potential deadlocks. Boronat and Cholvi [2003] avoids the potential deadlocks by statically introducing new locks in the code. Gadara uses Petri nets to build a mechanism that avoids deadlocks at runtime. These approaches make the applications avoid false deadlock potentials if the static analysis has false positives; since Dimmunix avoids only deadlocks that it previously detected at runtime, Dimmunix does not have this kind of false positives.

Moreover, the two approaches use the application's source code to statically find the deadlock potentials, while Dimmunix requires no source code. Gadara needs source code

annotations from the developers to filter out false positives. Annotating deadlock potentials as true/false positives requires deep understanding of the source code: the developer has to conceptually analyze all the possible sequences of lock acquisitions for the code reported as deadlock prone, as well as all the possible interleavings of these sequences. Since humans make mistakes, these annotations may be wrong. Moreover, they may be rendered invalid by plugins added later to the code base. Dimmunix requires no source code annotations and can handle dynamically added code.

2.4 Dynamic Deadlock Avoidance (Recovery)

A notable dynamic deadlock avoidance technique is [Zeng and Martin, 2004]. This technique modifies the JVM to serialize threads' accesses to sets of locks acquired in a nested fashion. For instance, if the program acquires lock l_2 while holding lock l_1, the two locks will belong to the same lock set; Zeng and Martin [2004] forces the acquisition of a new lock l (called ghost lock) before the acquisition of any lock from this lock set, for the rest of the execution. There are a couple of shortcomings of this approach: First, it relies on the fact that deadlocks occur later in the execution, when the lock sets are already obtained. Second, 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 [Zeng and Martin, 2004] is not effective. Every time new locks are used, [Zeng and Martin, 2004] has to update the lock sets. Whenever the lock sets are not up to date, the program is vulnerable to deadlocks. Third, the lock sets are not reusable in future runs: in each run, [Zeng and Martin, 2004] will have to restart the learning process from scratch. Dimmunix eliminates the three shortcomings by abstracting the locks involved in a deadlock to program locations (or call stacks); based on these, Dimmunix constructs a fingerprint of the deadlock. Then, Dimmunix uses the fingerprint to avoid any occurrence of that deadlock in future runs, at any point in the execution. However, there is a compromise that Dimmunix has to make: to provide protection against a deadlock bug, Dimmunix needs to first witness the deadlock.

Sammati [Pyla and Varadarajan, 2010], a framework that post-dates our work, dynamically detects deadlocks and transparently recovers the applications from deadlocks using

TM techniques. Sammati is a preloadable library that intercepts the POSIX Threads synchronization operations to dynamically detect and recover from deadlocks. Sammati makes the memory accesses performed by a thread during a critical section thread-local; this is called "privatizing" the shared memory pages. When the critical section ends, the updates are saved into the shared memory. 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. Sammati is intrusive and heavyweight; its efficiency and correctness were not tested on real applications. 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. In contrast, Dimmunix is not intrusive: to avoid deadlocks, it only alters the thread schedule underneath the application.

2.5 Deadlock Immunity

Other deadlock immunity approaches include [Nir-Buchbinder et al., 2008, Zeng, 2009]. 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 fingerprint of the deadlock; Zeng [2009] saves, in addition, the positions p'_1 and p'_2 where t_1 and t_2 deadlocked. In future runs, Nir-Buchbinder et al. [2008], Zeng [2009] 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), Zeng [2009] 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 [Nir-Buchbinder et al., 2008, Zeng, 2009], but uses a more accurate avoidance mechanism (§5). Like in these approaches (and any other deadlock avoidance technique), Dimmunix's deadlock avoidance mechanism relies on temporarily suspending threads, i.e., serializing concurrent code. Dimmunix has fewer false positives,

compared to these techniques (§10.1.2), i.e., concurrent code is less often serialized, thus alleviating the problem of lost parallelism. The efficiency of Dimmunix's critical-path computations is comparable to acquiring a gate lock. Therefore, the overall performance overhead incurred by Dimmunix is smaller (§10.1.2). Dimmunix handles five types of deadlock bugs, while Nir-Buchbinder et al. [2008], Zeng [2009] handle only mutex deadlocks.

2.6 Protection Against Other Bugs

There exist techniques that protect applications against data races and atomicity violations. Kivati [Chew and Lie, 2010] uses static analysis to detect accesses to shared memory that may be involved in atomicity violations; pairs of such accesses that should execute atomically delimit atomic regions. Each time a thread t executes an atomic region, Kivati uses a debug register to monitor the memory address x accessed by the delimiting instructions. Hence, Kivati is able to intercept accesses to x that would break the atomicity. When a thread t' is executing such an access, t' is suspended until t exits the atomic region. Thanks to using the hardware, Kivati incurs low performance overhead, but sacrifices completeness. However, there exist some important challenges for Kivati: (1) the same atomic region may be executed by different threads simultaneously, with different shared memory addresses, and (2) different atomic regions may execute in parallel. Kivati can handle few such simultaneous executions, since the number of debug registers is small (e.g., 4). Therefore, Kivati does not guarantee that the atomicity violations are always avoided.

Techniques like AtomRace [Letko et al., 2008] and [Krena et al., 2007] provide immunity against data races and atomicity violations. These approaches are purely software based; they dynamically detect races and atomicity violations, then prevent them from reoccurring by introducing additional synchronization. These techniques incur high performance overhead while detecting the data races and atomicity violations.

There exist also immunity techniques that protect applications against buffer overruns. Bouncer [Costa et al., 2007] provides immunity against buffer overruns; it automatically detects buffer overruns, then automatically infers filters that prevent malicious inputs from exploiting the same vulnerability. There is also a framework, Vigilante [Costa et al., 2005],

that distributes these filters to other nodes, to protect them against Internet worms. Similarly, our collaborative deadlock immunity framework, Communix, distributes deadlock signatures to other nodes to protect them against deadlocks they have not experienced yet. Vigilante relies on collaborative worm detection at end hosts and does not require hosts to trust each other. Communix differs from Vigilante in the validation of the received bug signatures: Vigilante uses replay to validate a new signature, while Communix efficiently checks deadlock signatures statically.

LOOM [Wu et al., 2010] bypasses data races at runtime using execution filters. It provides a language for developers to write execution filters that explicitly synchronize code. LOOM requires the developers to manually patch the code with execution filters. In contrast, Dimmunix performs automatic patching.

Similar to Dimmunix, ClearView [Perkins et al., 2010] performs automatic patching of errors in deployed software. ClearView monitors programs to detect buffer overruns and illegal control flow transfers, and generates repair patches that avoid the previously detected bugs by changing the state or the control flow of the program. Therefore, ClearView needs to be more intrusive than Dimmunix, because it targets memory and control flow errors. In order to avoid previously encountered deadlocks, Dimmunix only needs to alter the thread schedule. The runtime overhead incurred by the monitors in Firefox is 47–200%. In contrast, the runtime overhead incurred by Dimmunix's deadlock detection is negligible (§10).

Chapter 3

Background

In this chapter, we present the synchronization primitives that can lead to deadlocks (§3.1), and the terminology that we use to describe the deadlock immunity techniques (§3.2). Then, we define the deadlocks that are fundamentally different from each other, or require fundamentally different deadlock immunity techniques (§3.3). Finally, we give an overview of Dimmunix's design (§3.4).

3.1 Synchronization Primitives

In this thesis, we consider only thread synchronization mechanisms which involve explicit waits for resources or events. More precisely, a thread waits for a resource to become available (e.g., waits to acquire a lock) or waits for an event (e.g., waits for a message to arrive). A deadlock is a circular chain of such waits.

The waits can be synchronous or asynchronous. In a synchronous wait, a thread is suspended until the resource is available or the event happens. In an asynchronous wait, a thread periodically checks if the resource became available or the event occurred; in the meanwhile, the thread can perform some computation, rather than just polling the resource/event in a busy loop.

TM is a form of synchronization which does not involve explicit waits for resources or events. TMs optimistically execute atomic sections; if data races are detected among atomic sections, some of these sections are rolled back and restarted. However, there is an

implicit wait: a transaction (atomic section) terminates and the updates performed within it are committed only when the TM detects no conflicts involving that transaction.

A synchronization mechanism can be explicit, i.e., specified by the program, or implicit, i.e., hidden in the runtime. An example of an implicit (hidden) synchronization is the class initialization in Java—a class' static initializer is called by the JVM when the class is used for the first time. The JVM needs to perform some synchronization to make sure that the initializer executes only once.

A synchronization mechanism can use well-defined synchronization primitives or ad-hoc synchronization. Ad-hoc synchronization does not use well-defined synchronization primitives. Ad-hoc synchronization usually involves loops that periodically inspect some shared state, and terminate when a particular condition on the shared state holds [Xiong et al., 2010].

The deadlock-prone synchronization primitives handled by Dimmunix are: mutex lock acquisitions, read-write lock acquisitions, semaphore acquisitions, external lock acquisitions, and waits on condition variables. Other deadlock-prone synchronization mechanisms are: ad-hoc synchronization, thread joins, barriers, and synchronous waits for messages. Mutex locks, read-write locks, semaphores, condition variables, thread joins, and barriers are well known. An external lock is a lock associated with an object external to the running process, e.g., a file lock. Message passing is not a synchronization mechanism by itself. However, waiting to receive a message from another process (host) is a form of synchronization. An example of a message passing framework is the Java Messaging Service [jms, 2004].

In Table 3.1, we report the number of times the synchronization primitives handled by Dimmunix appear in the source code of real Java and C/C++ applications. Most of the synchronization operations are mutex lock acquisitions. SyncFinder [Xiong et al., 2010] counted the ad-hoc synchronizations in C/C++ applications; there are 6–33 ad-hoc synchronizations in the studied C/C++ applications. We did not find any ad-hoc synchronization in the Java programs we studied.

In the remainder of this section, we present the primitives used for synchronization: acquisition and release of mutex locks (§3.1.1), read-write locks (§3.1.2), semaphores (§3.1.3), and external locks (§3.1.4), wait, signal, and broadcast performed on condition

Application	LOC	Mutex lock acqs	R/W Lock acqs	Cond var waits	Semaphore acqs	File lock acqs
Eclipse	1,888,825	5,202	0	25	0	3
JBoss	636,895	869	56	77	12	0
Limewire	595,623	3,358	171	86	0	0
Vuze	476,702	1,429	13	4	5	0
ActiveMQ	96,172	330	0	21	5	0
Firefox	2,362,610	62	7	9	6	14
VLC	302,265	21	0	5	0	0

Table 3.1: The synchronization operations in several Java and C/C++ applications.

variables (§3.1.5), ad-hoc synchronization (§3.1.6), thread joins (§3.1.7), barriers (§3.1.8), and message passing primitives (§3.1.9).

3.1.1 Mutex Locks

Mutex locks are perhaps the most well-known and widely used synchronization mechanism. As we show in Table 3.1, one is entitled to expect mutex lock acquisitions to be the most frequent among synchronization operations.

We abstract by *lock(x)/unlock(x)* the acquisition/release of mutex lock *x*. Since mutexes are widely used in concurrent programs, we assume that the reader is familiar with the semantics of a mutex acquisition/release. We explain only the reentrancy aspect of mutex acquisitions.

A lock acquisition primitive can be reentrant or non-reentrant. If it is non-reentrant, a *lock(x)* call hangs if lock *x* is already held by the caller thread. If it is reentrant, *lock(x)* proceeds and increments a counter indicating the number of times lock *x* was (re)acquired; each *unlock(x)* call decrements the counter; lock *x* is free when the counter gets back to 0. For the remainder of the thesis, we assume in our explanations that lock acquisitions are reentrant.

We present now the standard mutex lock implementations for Java and C/C++. For Java programs, the standard lock implementations are the synchronized blocks (methods) [Jav, h] and the *java.util.concurrent.ReentrantLock* class [Jav, e]. Synchronized methods can

be trivially reduced to synchronized blocks; therefore we will only refer to synchronized blocks for the remainder of the thesis. For C/C++ programs, the standard lock implementations are provided by the POSIX Threads library [Pth, c].

Java Synchronized Blocks

First, we remind the reader that any Java object can be used as a lock by synchronized blocks.

A synchronized block has the syntax below:

```
synchronized (x) {
  //critical section
}
```

The above synchronized block can be viewed as:

```
lock(x)
//critical section
unlock(x)
```

In a synchronized block, the lock acquisition/release operations are automatically generated by the Java compiler. The Java compiler compiles the above synchronized block as follows: it inserts at the beginning of the block a *monitorenter(x)* statement which acquires the lock on object x, and inserts before all the exit points of the synchronized block a *monitorexit(x)* statement which releases the lock on x. Since the acquisition and release operations corresponding to a synchronized block are automatically generated by the compiler, we distinguish them from explicit lock/unlock operations, like Java *ReentrantLock.lock/unlock* and *pthreads_mutex_lock/unlock*.

Nested synchronized blocks acquire and release locks in a FIFO order: if a synchronized block B_1 contains another synchronized block B_2, the lock acquired by B_2 is always released before the lock acquired by B_1. Consider the nested synchronized blocks below:

```
synchronized (x) {
  synchronized (y) {
    ...
  }
}
```

The Java compiler makes sure that object *y* is always released before object *x*. Thanks to this property, the static analysis we perform in §8.1 is decidable. More precisely, Dimmunix needs to find the program positions where the locks involved in a deadlock were acquired. The synchronization pattern implemented by the synchronized blocks greatly simplifies this analysis.

Synchronized methods are equivalent to synchronized blocks. Consider the synchronized method below:

```
public synchronized void m() {
  //body of m
}
```

Using the "synchronized" keyword is equivalent to wrapping the body of *m* in a *synchronized (this) { ... }* block, where "this" is the reference to the caller object:

```
public void m() {
  synchronized (this) {
    //body of m
  }
}
```

Java ReentrantLock

The ReentrantLock class provides explicit lock/unlock primitives. The programmer must take care of using them in a safe manner: a ReentrantLock must be released on all the execution paths that exit the critical section.

The safe pattern of using a ReentrantLock *x* is illustrated in the Java code below:

```
try {
  x.lock();
  //critical section
}
finally {
  x.unlock();
}
```

The above code is conceptually equivalent to a *synchronized(x){...}* block. The "finally" block automatically releases *x* every time the execution exits the critical section. If programmers do not use this safe pattern, they have to take care of manually inserting *x.unlock()* calls before all the exit points of the critical section.

POSIX Threads Locks

The lock/unlock primitives provided by the POSIX Threads library are *pthread_mutex_lock* and *pthread_mutex_unlock*. Unfortunately, neither C nor C++ provide try-finally blocks; therefore, the programmer must remember to release the lock on every execution path that exits the critical section.

The POSIX Threads locks are non-reentrant by default. If a C/C++ programmer wants to simulate Java-like synchronized methods, he/she must remember to make the lock for the *this* object reentrant.

3.1.2 Read-Write Locks

A read-write lock is formed of a read lock and a write lock. The read lock can be held simultaneously by any number of threads, while the write lock can be held by only one thread at a time. While a thread holds the write lock, the read lock cannot be held by any other thread, and vice versa. In other words, read locks are mutually exclusive with write locks.

Given a read-write lock *x*, we denote by *lockr(x)/lockw(x)* the acquisition of *x*'s read/write lock. For the sake of generality, in this thesis we assume that read-write lock acquisitions

are reentrant. By *unlockr(x)/unlockw(x)*, we denote the release of *x*'s read/write lock. By acquiring *x* in read/write mode, we mean acquiring *x*'s read/write lock.

The standard Java implementation of read-write locks is the ReentrantReadWriteLock class [Jav, f]. The POSIX Threads library provides the routines *pthread_rwlock_rdlock*, *pthread_rwlock_wrlock*, and *pthread_rwlock_unlock* [PTh], for acquiring and releasing read-write locks.

In this thesis, we use the semantics of Java ReentrantReadWriteLock [Jav, f] for the read-write locks. The most important aspects of these semantics are:

- A thread holding only the read lock cannot acquire the write lock without releasing the read lock first.
- A thread holding the write lock can also acquire the read lock. In this situation, an *unlockr* (respectively *unlockw*) call releases the read (respectively write) lock, but keeps the write (respectively read) lock.

We illustrate these semantics in the code in Figure 3.1. The code represents a singleton pattern.

3.1.3 Semaphores

A semaphore is a lock that maintains a set of acquisition permits. A semaphore with more than 1 permit can be acquired simultaneously by multiple threads. A semaphore has a value associated with it, which stores the number of available permits; the value can be incremented/decremented by any thread. Therefore, there is no notion of ownership for a semaphore.

We denote by *acquire(s, n)* the acquisition of *n* permits for semaphore *s*; by *release(s, n)*, we denote the release of *n* permits for *s*. If the second parameter (i.e., *n*) is omitted, then $n = 1$. By $s.v$, we denote the value of *s*, i.e., the number of available permits.

The standard implementation of a semaphore for Java is the Semaphore class [Jav, g]. The standard C/C++ implementation of semaphore acquisition and release operations is provided by the POSIX *sem_wait* and *sem_post* routines [Pos, b].

The semantics we use for the two operations are consistent with the Java Semaphore class [Jav, g]. In brief, the semantics of an *acquire(s, n)* operation are:

```
//retrieve a unique object for each key value k
//x is a read-write lock that synchronizes the accesses
//to a cache stored in hash map M
lockr(x)
obj = M[k]
if obj == null {
  //release the read lock, then upgrade to write lock
  unlockr(x)
  lockw(x)
  lockr(x)
  //the read and write locks are both held at this point
  obj = M[k]
  //recheck, M[k] might have been filled by another thread
  if obj == null {
    obj = new object
    M[k] = obj
  }
  unlockw(x)
  //downgraded to read lock; only the read lock is held here
}
unlockr(x)
return obj
```

Figure 3.1: Code illustrating the semantics of read-write locks.

- If $s.v \geq n$, $s.v$ is decremented by n, and the acquisition routine returns immediately.
- If $s.v < n$, the caller thread is suspended until $s.v \geq n$; then, $s.v$ is decremented by n, and the acquisition routine returns.

A *release(s, n)* performs the following steps:

1. The value $s.v$ is incremented by n.
2. While $s.v > 0$ and there is a thread waiting for $k \leq s.v$ permits, resume that thread. If $s.v = 0$ or there is no thread waiting for $k \leq s.v$ permits, the release routine returns.

POSIX semaphores [Pos, b] do not accept the acquisition/release of more than 1 permit at a time. For $n = 1$, the above semantics are consistent with the POSIX semaphores.

We illustrate the semantics of the *acquire* and *release* operations in the code below, which implements a pattern where a thread waits for n notifications from other threads:

```
//s is a semaphore; initially s.v = 0
Thread 1:                                Threads 2, ..., n+1:
acquire(s, n)//wait for n permits        release(s) //give a permit
```

3.1.4 External Locks

External locks are shared among processes and synchronize accesses to resources that are external to the running processes. For instance, a file lock synchronizes the accesses to a file. The Dimmunix prototype handles only file locks; however, it can be easily extended to handle other types of external locks.

A file lock functions like a mutex, i.e., it can be held by only one process at a time. Therefore, for locking and unlocking a file f, we use the same abstractions as for mutex locks, i.e., *lock(f)* and *unlock(f)*. By locking (respectively unlocking) a file f, we mean acquiring (respectively releasing) the file lock associated with f.

Java provides primitives for locking/unlocking files in the FileChannel class [Jav, c]. The standard C/C++ libraries provide the *flock* and *funlock* primitives for locking and unlocking files [CFi].

3.1.5 Condition Variables

Condition variables implement a wait-notify synchronization pattern, where a thread waits to be notified by another thread about a particular event (e.g., the update of a shared data structure). To synchronize the accesses to the shared state, condition variables have a mutex lock attached to them.

There are three types of operations performed on condition variables: wait, signal, and broadcast. We denote by *wait(c)* a wait operation performed on condition variable c, and by *signal(c)* (respectively *broadcast(c)*) a signal (respectively broadcast) operation performed on c. By notification we mean either a signal or a broadcast. By "waiter thread" we mean a thread blocked on a wait call; by "notifier thread" we mean a thread that is supposed to notify a waiter thread. By *lock(c)* (respectively *unlock(c)*) we denote the acquisition (respectively release) of c's mutex lock. By acquiring (respectively releasing) c we mean acquiring (respectively releasing) c's mutex lock.

There are two semantics for condition variables: Hoare semantics [Hoare, 1974] and Mesa semantics [Lampson and Redell, 1980]. The latter are more relaxed than the former and were developed as a way to improve the performance of the operations on condition variables; Mesa semantics are used in the Java and POSIX Threads condition variable implementations. Therefore, we use the Mesa semantics in this thesis. More precisely, we use the Java variation of the Mesa semantics, in which a notifier thread has to hold c when executing *signal(c)* or *broadcast(c)* on condition variable c.

The semantics of the wait, signal, and broadcast operations performed on a condition variable c are:

- A *wait(c)* call atomically releases c and suspends the caller thread (which now becomes a waiter thread). The caller thread can be resumed by a *signal(c)* or *broadcast(c)* call executed by a notifier thread. Right before returning from the wait call, the caller thread automatically reacquires c.
- A *signal(c)* call resumes a thread waiting on c.
- A *broadcast(c)* call resumes all the threads waiting on c.
- A notifier thread must hold c when calling *signal(c)* or *broadcast(c)*.
- A *signal(c)* (or *broadcast(c)*) call does not resume a *wait(c)* call that executes later; only currently waiting threads are resumed by signal (respectively broadcast) calls.

The difference between the Mesa and Hoare semantics is related to the notification mechanism. Let t_n be a notifier thread performing a notification on condition variable c, which resumes a thread t_w waiting on c. In the Hoare semantics, t_n passes c's ownership to t_w together with sending the notification, in order to guarantee that no change in the shared state occurs before t_w returns from the wait call. When releasing c, t_w returns c's ownership back to t_n. In Mesa semantics, the notification only resumes t_w, without giving up the ownership of c.

In Mesa semantics, the shared state may change between the moment of the notification and the moment when the wait call returns. Therefore, the wait condition has to be rechecked, as we illustrate in the pseudocode below:

```
//c is a condition variable; x is a shared variable
Waiter thread:                                  Notifier thread:
lock(c)                                         lock(c)
while condition on x does not hold {            update x
  wait(c)                                       signal(c)
}                                               unlock(c)
//condition holds here
unlock(c)
```

Java provides condition variable primitives in the Object class [Jav, i] and the Condition interface [Jav, b]. The POSIX Threads library implements the wait, signal, and broadcast primitives in the *pthread_cond_wait*, *pthread_cond_signal*, and *pthread_cond_broadcast* routines [Pth, a].

3.1.6 Ad-Hoc Synchronization

Ad-hoc synchronization is a form of synchronization that does not use well-defined synchronization primitives. Ad-hoc synchronization usually involves loops that periodically inspect some shared state, and terminate when a particular condition on the shared state holds [Xiong et al., 2010]. We illustrate such a synchronization loop in the pseudocode below:

```
//x is a shared variable
Waiter thread:                                  Notifier thread:
while condition on x does not hold {            update x
  sleep or do some computation
}
//condition holds here
```

Ad-hoc synchronization is conceptually equivalent to wait-notify schemes that use condition variables. For instance, the above code is equivalent to the code at the end of §3.1.5. Because of this, ad-hoc synchronization does not introduce any new research challenges, w.r.t. deadlock immunity. Therefore, the Dimmunix prototype does not specifically handle deadlocks involving ad-hoc synchronization.

We envision that tools like SyncFinder [Xiong et al., 2010] can be used to automatically annotate ad-hoc synchronizations in programs, thus enabling the Dimmunix prototype to handle deadlocks involving ad-hoc synchronization. For instance, SyncFinder associates an id *L* to the above busy loop, then inserts the *Sync_Loop_Begin(L)* and *Sync_Loop_End(L)* calls to stub routines at the beginning and the end of the loop; it also inserts *Sync_Write(&x, L)* calls in the places where *x* is updated and *Sync_Read(&x, L)* calls wherever *x* is read in the busy loop. Dimmunix can associate a condition variable *c* to id *L*, and handle a *Sync_Loop_Begin(L)* (respectively *Sync_Loop_End(L)*) call as a *lock(c)* (respectively *unlock(c)*) operation. Dimmunix can handle the *Sync_Read(&x, L)* (respectively *Sync_Write(&x, L)*) calls as *wait(c)* (respectively *broadcast(c)*) operations.

3.1.7 Thread Joins

A thread join is a synchronization mechanism where a thread waits for another thread to finish its execution. We denote by *join(t)* the wait for a thread *t* to finish.

Just like ad-hoc synchronization, a thread join can be reduced to a wait-notify scheme using condition variables. We can associate a fresh condition variable $c[t]$ and a flag *done[t]* with every thread *t*; a *join(t)* call can be viewed as:

```
//initially done[t] = false
Thread calling join(t):                  Thread t:
//code equivalent to join(t)             //finish the execution
lock(c[t])                               lock(c[t])
if !done[t]                              done[t] = true
  wait(c[t])                             broadcast(c[t])
unlock(c[t])                             unlock(c[t])
```

Since thread joins can be reduced to wait-notify schemes using condition variables, Dimmunix does not specifically handle deadlocks involving thread joins.

Java provides a thread join primitive in the Thread class [Jav, d]; the POSIX Threads library implements it in the *pthread_join* routine [Pth, b].

3.1.8 Barriers

A barrier implements a synchronization mechanism in which a group of n threads synchronize to start performing some computation at the same time. In other words, the n threads coordinate in order to "cross" the barrier at the same time.

A barrier b with n slots is implemented by suspending the first $n-1$ threads until the last thread arrives at the barrier. When the last thread arrives, the other threads are resumed, and all the n threads proceed with their execution. This can be implemented using mutexes and semaphores, as we illustrate in the code below:

```
//mutex b.mutex
//queue size b.q; initially b.q = 0
//semaphore b.sem; initially b.sem.v = 0
lock(b.mutex)
b.q++
cross = (b.q == n)? true: false
unlock(b.mutex)
//if it is the last thread to reach the barrier
if cross {
  //wake up the other threads and proceed
  release(b.sem, n-1)
}
else {
  //wait until all the threads reach the barrier
  acquire(b.sem)
}
```

Since a barrier can be implemented with mutexes and semaphores, it does not offer any interesting research challenges, w.r.t. deadlock immunity. Therefore, the Dimmunix prototype does not specifically handle barriers.

Java implements barriers in the CyclicBarrier class [Jav, a]. The POSIX Threads library implements the wait on a barrier in the *pthread_barrier_wait* routine [Pos, a].

3.1.9 Message Passing

Message passing frameworks enable different processes on the same machine or different machines to communicate by exchanging messages. An example of a message passing framework is the Java Message Service (JMS) [jms, 2004]. JMS provides message queues that implement the producer-consumer design pattern. A message can be added to a queue by a producer process and extracted (consumed) from the queue by a consumer process. By *receive(q)* we denote that the caller process is synchronously waiting for a message on queue q; this routine returns the message consumed from q. By *send(q, m)* we denote that the caller process sends a message m by appending it to queue q.

Message passing can be viewed a distributed wait-notify scheme. Synchronously receiving a message is similar to waiting on a condition variable. Sending a message is similar to notifying a thread. Therefore, deadlocks involving message passing are similar to deadlocks involving waits and notifications on condition variables; the only difference is that the former can involve multiple processes (possibly running on multiple machines), while the latter involve multiple threads of the same process. We describe these wait-notify deadlocks in §3.3.8.

3.2 Terminology

In this section we explain the terminology we use to describe the concepts related to deadlock immunity.

We explain first the terminology related to the notion of deadlock. A deadlock is a circular chain of waits for resources or events. The entities that are waiting for resources or events are threads; the threads can belong to different processes, in the case of external locks. For simplicity, we call a mutex lock, read-write lock, or external lock simply "lock" when its type is not relevant. The resources are of two types: locks or semaphores. We introduced previously the notion of acquiring (or holding) a condition variable; this is a shorthand for saying that the mutex lock associated with the condition variable is acquired (respectively held). By size of a deadlock, we mean the number of threads involved in the deadlock.

By lock inversion, we mean that a lock y is requested while holding a lock x by a thread t_1, and x is requested while holding y by another thread t_2. A mutex deadlock always involves a lock inversion. However, a lock inversion does not necessarily lead to a deadlock.

We explain now the notions of program position and call stack. By program position (or program location), we refer to any information that identifies a statement in a program, i.e., source code location or offset in the binary. Program locations have the same meaning in every execution of the application, unless the code is changed and recompiled, or the program binary is modified.

To represent the synchronization state of a program, one often uses a resource allocation graph (RAG). A RAG is a directed graph that connects thread and resource nodes; a resource node represents a lock or a semaphore. The edges are of three types, as illustrated in Table 3.2: request edges, hold edges, and wait edges. A request edge is from a thread node t to a resource node x, i.e., $t \longrightarrow x$; this means that thread t is waiting to acquire resource x. A hold edge is from a resource node x to a thread node t, i.e., $x \longrightarrow t$; this means that thread t is holding resource x. A wait edge is from a thread node t_1 to another thread node t_2 and it is annotated with a condition variable, i.e., $t_1 \xrightarrow{c} t_2$; this means that thread t_1 is waiting for a notification on condition variable c from thread t_2. An edge representing the request (or acquisition) of a read-write lock in read/write mode is annotated with the mode, i.e., $t \xrightarrow{rd/wr} l$ ($l \xrightarrow{rd/wr} t$); this means that thread t is waiting for (or is holding) the lock l in read/write mode. To simplify the explanations, we assume that a read-write lock can be held only in one mode at a time, i.e., either read or write mode; however, Dimmunix also handles the case when the lock is held in both modes simultaneously.

The RAG edges can also be labeled with program positions or call stacks. For instance, a request edge $t \xrightarrow{p} x$ (or $t \xrightarrow{CS} x$) denotes that thread t is waiting for resource x at program position p (respectively with call stack CS). A wait edge $t_1 \xrightarrow{c@p} t_2$ means that thread t_1 is waiting at program position p for a notification on condition variable c from thread t_2. In general, the presence of "$@p$" (or "$@CS$") in an edge label means that the event captured by the label occurred at program position p (respectively with call stack CS).

The semantics of request and hold edges for semaphores are: a request (hold) edge connecting thread node t and semaphore node s means that thread t waits to acquire (holds) one permit of semaphore s. To simplify the explanations, we assume that only one permit

Edge	Type	Meaning
$t \longrightarrow x$	request	thread t is waiting to acquire resource x
$t \xrightarrow{rd} l$	request	thread t is waiting to acquire read-write lock l in read mode
$t \xrightarrow{wr} l$	request	thread t is waiting to acquire read-write lock l in write mode
$x \longrightarrow t$	hold	thread t is holding resource x
$l \xrightarrow{rd} t$	hold	thread t is holding read-write lock l in read mode
$l \xrightarrow{wr} t$	hold	thread t is holding read-write lock l in write mode
$t_1 \xrightarrow{c} t_2$	wait	thread t_1 is waiting for a notification on condition variable c from thread t_2

Table 3.2: The edges of a resource allocation graph (RAG).

is requested (or held) at a time by one thread; however, Dimmunix also handles the case when multiple permits are requested (respectively held) simultaneously by one thread. The semaphore nodes are annotated with the number of available permits, e.g., $s^{(0)}$ denotes a semaphore s with no available permits.

By $x \in RAG$ we denote that node x belongs to the RAG's nodes; by $x \longrightarrow y \in RAG$ we denote that the edge $x \longrightarrow y$ belongs to the RAG's edges. When the type of nodes x and y is not specified, $x \longrightarrow y$ represents an arbitrary edge.

For a RAG, we define the notion of *multicycle*. A multicycle is a set of nodes M in the RAG with the property that for every node $x \in M$ the set of nodes reachable from x is M. We use the notation $x \rightsquigarrow y$ to denote the fact that node y is reachable from node x, i.e., there is a path from x to y; $x \rightsquigarrow x$ only holds when x is part of a cycle involving at least two distinct nodes. Given a subgraph $G \subseteq RAG$, $x \overset{G}{\rightsquigarrow} y$ denotes that there is a path from x to y whose intermediate nodes all belong to G.

A call stack is an approximation of the running thread's call flow. A call stack of size n consists of n call frames (or simply frames) capturing the last n method/function calls that the running thread is currently executing. The frames contain the program locations where the calls were initiated. The frames are nested: the method/function from frame $k+1$ is executed within the method/function from frame k. In C/C++, the call stack can be computed using the *backtrace()* routine; in Java, the call stack is returned by the *Thread.getStackTrace()* method. The top frame returned by *Thread.getStackTrace()* is the

program location of the instruction that the running thread is currently executing.

We explain now the notion of nested acquisitions of resources (e.g., lock, semaphore); deadlocks involving only resource acquisitions can only happen when the acquisitions are nested. Two acquisitions of resources r_1 and r_2 are nested if and only if one of them is performed within the scope of the other, i.e., a thread acquires r_2 (or r_1) while holding r_1 (or r_2). For instance, the code below illustrates two nested lock acquisitions:

```
lock(x)
lock(y)
unlock(y)
unlock(x)
```

By nesting depth (or nesting level) we denote the number of resources held at a particular program location. For instance, in the above code, the nesting depth is 2 right after *lock(y)* executes, 1 right after *unlock(y)* executes, and 0 at the end. Given two nested resource acquisitions, we call "inner" the one at the deeper nesting level, and "outer" the one at the shallower nesting level. Similarly, we call "inner (outer) call stack" the call stack that the running thread has when it executes the inner (outer) resource acquisition.

3.3 Deadlocks

In this chapter, we define the deadlocks that are fundamentally different from each other, or at least require fundamentally different deadlock immunity techniques.

There are five types of deadlocks handled by Dimmunix. The first is the mutex deadlock (§3.3.1), which involves mutex locks. This type of deadlock is likely to be encountered most often in real-world software [Lu et al., 2008, Fonseca et al., 2010, Song et al., 2010]. The second is the read-write deadlock (§3.3.2), which involves read-write locks. The third is the semaphore deadlock (§3.3.3), which involves semaphores. There also exist hybrid deadlocks, involving mutex locks, read-write locks, and/or semaphores (§3.3.4); Dimmunix handles hybrid deadlocks, too. The fourth is the initialization deadlock (§3.3.5), which involves class initialization and mutex locks. The fifth is the external deadlock (§3.3.6), which involves external locks.

There are also blocked notifications (§3.3.7), which involve condition variables and mutex locks. A blocked notification is a starvation situation which may lead to deadlock. Ad-hoc synchronization and thread joins can be reduced to wait-notify schemes using condition variables (§3.1.6, §3.1.7). Therefore, the ad-hoc deadlocks and join deadlocks are equivalent to blocked notifications, i.e., starvation that may lead to deadlock.

Since blocked notifications are at the frontier between starvation and deadlock, it is not possible (in the general case) to offer a feasible immunization solution for them. However, we provide a tentative immunization technique and present its limitations in §6.2.

We also describe the deadlocks that are not handled by the Dimmunix prototype: wait-notify deadlocks (§3.3.8), self-deadlocks (§3.3.9), and non-deadlock hangs (§3.3.10). A wait-notify deadlock involves only wait and signal/broadcast calls; this deadlock can be detected, but it cannot be avoided without changing the semantics of the application (§6.1). A self-deadlock is not a deadlock per se, because it involves only one thread; it is trivial to detect, but is not avoidable without changing the semantics of the program (§6.1). A non-deadlock hang does not involve circular waits for resources or events. Non-deadlock hangs are hard to detect, and hard to avoid without changing the semantics of the application. Therefore, the Dimmunix prototype does not handle wait-notify deadlocks, self-deadlocks, and non-deadlock hangs.

3.3.1 Mutex Deadlocks

A mutex deadlock is a situation where every thread in a group of threads is waiting for a mutex lock held by other threads in the group. A mutex deadlock is likely the most frequently encountered type of deadlock, because mutexes are arguably the most widely used synchronization constructs, as suggested in Table 3.1.

A mutex deadlock leads to a permanent hang of the threads involved if no hang recovery mechanism is in place (e.g., microreboot [Candea et al., 2004], forced abort). This is because only the owner of a mutex lock can release the lock.

We illustrate how a mutex deadlock may occur using the code below:

```
Thread t1:                          Thread t2:
lock(l1)

                                    lock(l2)
lock(l2) //blocked here              lock(l1) //blocked here
unlock(l2)                          unlock(l1)
unlock(l1)                          unlock(l2)
```

The deadlock involves threads t_1 and t_2 and mutex locks l_1 and l_2; it is graphically illustrated by the RAG in Figure 3.2. The circular wait chain is the following: t_1 holds l_1 and waits for l_2; t_2 holds l_2 and waits for l_1.

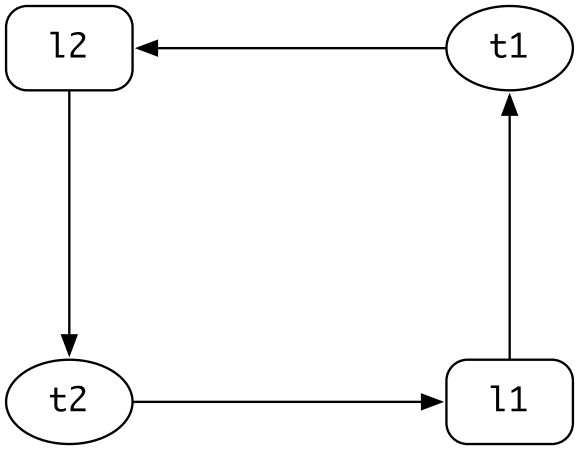

Figure 3.2: A mutex deadlock involving threads t_1, t_2 and mutex locks l_1, l_2.

A deadlock involving threads $t_1,...,t_n$ and mutex locks $l_1,...,l_n$ can be represented as a simple cycle in the RAG, formed of request and hold edges:

$$l_1 \longrightarrow t_1 \longrightarrow l_2 \longrightarrow ... \; t_n \longrightarrow l_1$$

3.3.2 Read-Write Deadlocks

A read-write deadlock is a situation where every thread in a group of threads is waiting for a read-write lock held by other threads in the group; if a thread in this group is waiting for a lock in read mode, then that lock must be held in write mode by some other thread in the group, in order to have a deadlock.

A read-write deadlock leads to a permanent hang of the threads involved. This is because only the owner of a read-write lock can release the lock.

We illustrate two simultaneous read-write deadlocks in the code below:

```
Thread t1:              Thread t2:              Thread t3:
lockr(l1)

                        lockw(l2)

                                                lockr(l1)
lockw(l2) //blocked     lockw(l1) //blocked     lockr(l2) //blocked
unlockw(l2)             unlockw(l1)             unlockr(l2)
unlockr(l1)             unlockw(l2)             unlockr(l1)
```

The deadlocks involve threads t_1, t_2, and t_3 and read-write locks l_1 and l_2; they are graphically illustrated by the RAG in Figure 3.3. The circular waits are the following: In the first deadlock, t_1 holds l_1 in read mode and waits to acquire l_2 in write mode; t_2 holds l_2 in write mode and waits to acquire l_1 in write mode. In the second deadlock, t_3 holds l_1 in read mode and waits to acquire l_2 in read mode; t_2 holds l_2 in write mode and waits to acquire l_1 in write mode.

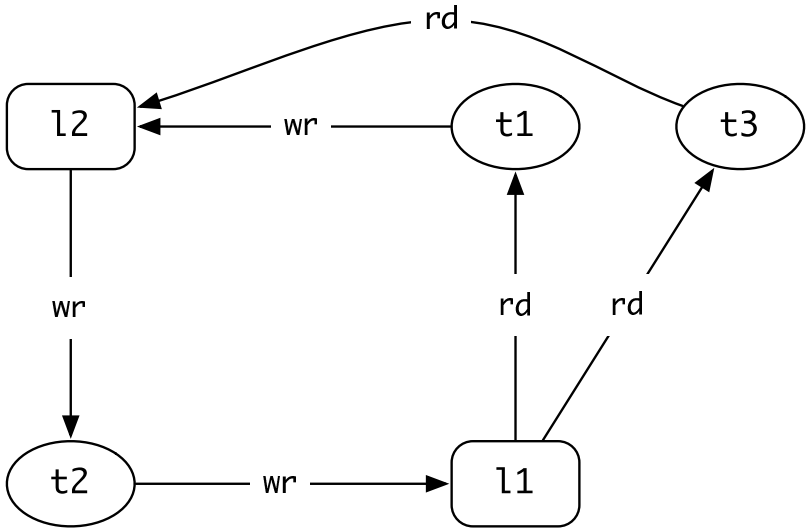

Figure 3.3: A multicycle representing two read-write deadlocks involving threads t_1, t_2, t_3 and read-write locks l_1, l_2.

A deadlock involving threads $t_1, ..., t_n$ and read-write locks $l_1, ..., l_n$ can be represented as a simple cycle in the RAG, formed of request and hold edges

$$l_1 \xrightarrow{rd/wr} t_1 \xrightarrow{rd/wr} l_2 \xrightarrow{rd/wr} ... \; t_n \xrightarrow{rd/wr} l_1$$

with the property that, for each read mode request edge, the next edge must be a write mode hold edge:

$$\forall i \in \overline{1,n} : t_i \xrightarrow{rd} l_{(i+1)\%n} \Rightarrow l_{(i+1)\%n} \xrightarrow{wr} t_{(i+1)\%n}$$

By "%" we denote the modulo operator.

3.3.3 Semaphore Deadlocks

A semaphore deadlock is a situation where every thread in a group of threads is waiting to acquire permits of a semaphore s, and the semaphore does not have enough permits; all the threads holding permits of s belong to this group.

Since any active thread can release permits for a semaphore s, the following invariant must hold for s to be involved in a deadlock: a release operation on s is always performed by threads holding permits of s. In other words, s has to be used as a lock. If the invariant does not hold, there is no deadlock situation.

We illustrate a semaphore deadlock in the code in Figure 3.4. The deadlock involves threads t_1, t_2, and t_3 and semaphores s_1 and s_2; it is graphically illustrated by the RAG in Figure 3.5. The circular wait chain is the following: t_1 holds a permit of s_1 and waits for a permit of s_2; t_2 holds the only permit of s_2 and waits for a permit of s_1; t_3 holds the other permit of s_1 and waits for a permit of s_2.

A deadlock involving threads $t_1, ..., t_n$ and semaphores $s_1, ..., s_m$ appears as a multicycle in a RAG, formed of request and hold edges:

$$\forall x \in M = \{t_1, ..., t_n, s_1, ..., s_m\} \; : \; (\{y \in RAG \mid x \rightsquigarrow y\} = M)$$

The above property states what it means for M to be a multicycle: every node y reachable from a node $x \in M$ belongs to M. For the semaphore nodes, we omitted the labels indicating the number of available permits; all these labels are 0, i.e., semaphore s_i appears as $s_i^{(0)}$ in

```
//semaphore s1 has 2 permits; semaphore s2 has 1 permit
Thread t1:              Thread t2:              Thread t3:
acquire(s1)
                        acquire(s2)
                                                acquire(s1)
acquire(s2)//blocked    acquire(s1)//blocked    acquire(s2)//blocked
release(s2)             release(s1)             release(s2)
release(s1)             release(s2)             release(s1)
```

Figure 3.4: Code illustrating a semaphore deadlock.

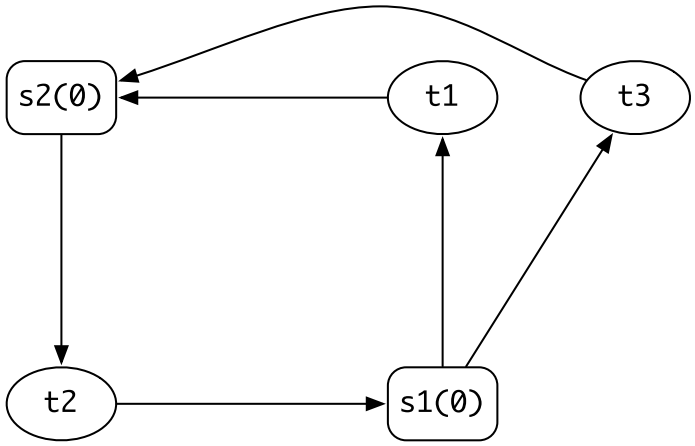

Figure 3.5: A semaphore deadlock involving threads t_1, t_2, t_3 and semaphores s_1, s_2.

the RAG.

3.3.4 Hybrid Deadlocks

A hybrid deadlock is a deadlock involving mutex locks, read-write locks, and/or semaphores. In other words, it generalizes mutex deadlocks, read-write deadlocks, and semaphore deadlocks.

In the RAG, a hybrid deadlock is represented as a "relaxed" form of a multicycle, where escape paths (i.e., paths that do not belong to the multicycle) are allowed for read-write lock nodes. For semaphore nodes, no escape paths are allowed. We illustrate a hybrid deadlock in the RAG in Figure 3.6, having one escape path (i.e., $l_2 \xrightarrow{rd} t_4$).

More precisely, a hybrid deadlock is a subgraph $M \subseteq RAG$ having the following properties:

1. M is a cycle.
2. Within M, read mode request edges must be followed by write mode hold edges.
3. No escape paths are allowed for semaphore nodes.

Formally, the three properties can be expressed as:

1. $\forall t \in M : t \overset{M}{\rightsquigarrow} t$
2. $\forall t \xrightarrow{rd} l_{rw} \in M : (\exists l_{rw} \xrightarrow{wr} t' \in M)$
3. $\forall t \longrightarrow s^{(k)} \in M : k = 0 \wedge (\forall s \longrightarrow t' \in RAG : t' \in M)$

where t and t' are thread nodes, l_{rw} is a read-write lock node, and $s^{(k)}$ is a semaphore node with k available permits. For a semaphore node, we omit the number of available permits when it is not relevant.

Additionally, for a semaphore s to be involved in a hybrid deadlock, the invariant mentioned in §3.3.3 must hold, i.e., s is always released by threads holding permits of s.

3.3.5 Initialization Deadlocks

Initialization deadlocks involve class initialization and mutex locks. They can occur only in languages like Java and C#, that allow programmers to define static initializers for classes. A static initializer of a class A is a piece of code that executes right after A is loaded, and before an object of type A is created (or a static field/method of A is used) for the first time. Dimmunix does not handle initialization deadlocks involving read-write locks or semaphores, because such deadlocks do not offer any new insights, w.r.t. deadlock immunity.

We illustrate an initialization deadlock in the Java code in Figure 4.4. Thread t_1 holds $B.x$ and is waiting for class A to initialize right before creating an object of type A. Thread t_2 executes A's static initializer right before creating an object of type A and is waiting to acquire $B.x$ inside the initializer.

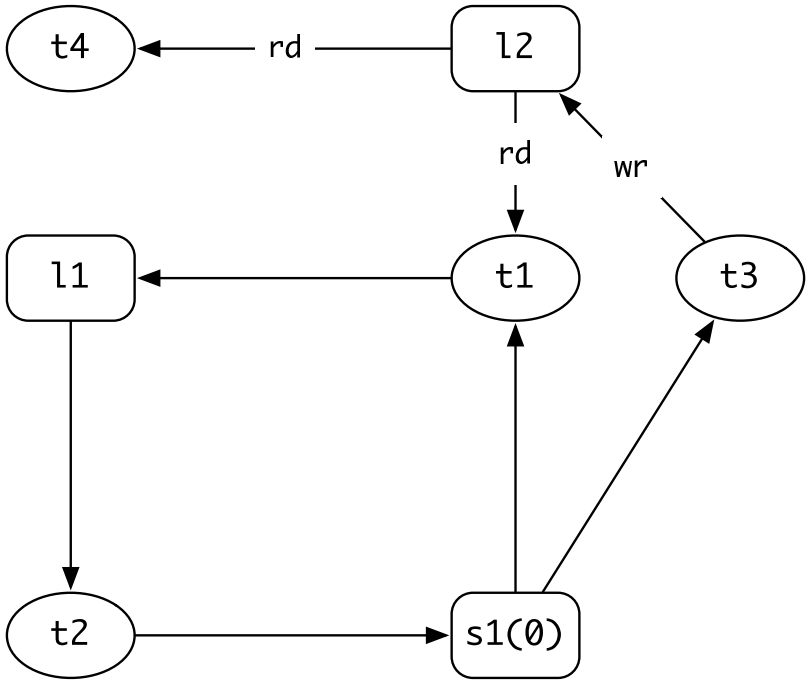

Figure 3.6: A hybrid deadlock involving threads t_1, t_2, t_3, t_4, mutex lock l_1, read-write lock l_2, and semaphore s_1.

```
class A {
  static { //static initializer
    synchronized (B.x) { //blocked here (t1 holds B.x)
    }
  }
}

Thread t1:                          Thread t2:
synchronized (B.x) {
                                    A a2 = new A(); //initializing A
  A a1 = new A(); //waiting for A to initialize
}
```

Figure 3.7: Code illustrating an initialization deadlock.

The invocation of the static initializer of a class A is conceptually equivalent to inserting the code in Figure 3.8 before each statement that may trigger A's initialization (e.g., a "new A()" statement).

```
//initially initializedA = false
synchronized (initLockA) {
  if (!initializedA) {
    Class.forName("A"); //load and initialize class A
    initializedA = true;
  }
}
A a = new A();
```

Figure 3.8: Code equivalent to initializing class *A* before *A* is used for the first time.

The above transformation shows that initialization deadlocks can be represented as mutex deadlocks. Therefore, initialization deadlocks are equivalent to mutex deadlocks.

However, waiting for a class to initialize cannot be considered an explicit synchronization mechanism, because class initialization is (usually) not explicitly invoked. Class initialization is triggered automatically by the runtime (e.g., JVM, .NET CLR) when it invokes a constructor/method or uses a field of a class that is not yet loaded. The synchronization involved in class initialization is hidden in the runtime, i.e., not exposed to the programmer. Since initialization deadlocks are caused by such hidden synchronization, they require a different immunization technique from that used for mutex deadlocks.

Initialization deadlocks are representative of deadlocks involving synchronization hidden in the runtime. The challenge specific to these deadlocks is intercepting the hidden synchronization performed by the runtime. Dimmunix intercepts the hidden synchronization involved in class initialization without changing the runtime (§9.1). However, this may not be possible for all types of "hidden" deadlocks.

3.3.6 External Deadlocks

External deadlocks involve synchronization on external locks, e.g., file locks. Therefore, these deadlocks may occur among threads from different processes. The Dimmunix prototype only handles file locks, but other types of external locks can be easily plugged into Dimmunix.

Threads can deadlock when acquiring file locks, as shown in the code below:

```
//f1 and f2 are files
Thread t1:                          Thread t2:
lock(f1) //acquire lock on f1

                                    lock(f2) //acquire lock on f2
lock(f2) //blocked here             lock(f1) //blocked here
unlock(f2)                          unlock(f1)
unlock(f1)                          unlock(f2)
```

External deadlocks are equivalent to mutex deadlocks, but the fact that external locks are shared among processes calls for a fundamentally different immunization technique: the synchronization state (i.e., the RAG) has to be shared among processes.

3.3.7 Blocked Notifications

A blocked notification is a situation where a notifier thread t_n that is supposed to resume a waiter thread t_w cannot perform the notification, because it needs to acquire a mutex lock l held by t_w, before signaling t_w. A deadlock situation occurs if thread t_n is the only notifier or all the notifier threads are waiting to acquire lock l; otherwise, the blocked notification represents just a starvation situation, i.e., thread t_w starves thread t_n by holding lock l while waiting on a condition variable.

In general, having a thread t_w hold a lock l while waiting on a condition variable is a bad design; t_w may starve threads that need to acquire lock l, because waits for events (or updates of a shared state) can last indefinitely. Therefore, a blocked notification is not a deadlock per se; it is a starvation situation that may lead to deadlock. Determining whether a blocked notification represents a deadlock is undecidable.

Blocked notifications involving a waiter thread t_w, a notifier thread t_n, a mutex lock l, and a condition variable c can have 8 different patterns, corresponding to all the ways in which the critical sections of l and c can be arranged to have a blocked notification. The first pattern is:

```
Thread Tw:                    Thread Tn:
lock(l)
lock(c)
wait(c)                       lock(l)
unlock(c)                     unlock(l)
unlock(l)                     lock(c)
                              signal/broadcast(c)
                              unlock(c)
```

We illustrate graphically the above pattern in the RAG in Figure 3.9. Thread t_w executes the *wait(c)* call while holding l, and thread t_n needs to acquire l before executing *signal/broadcast(c)*.

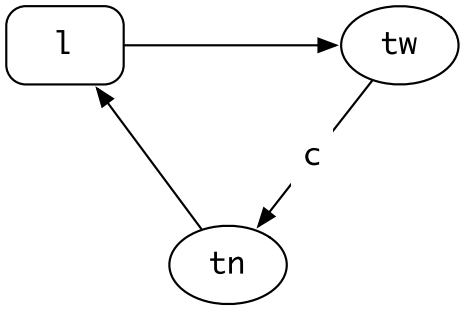

Figure 3.9: A blocked notification involving threads t_w, t_n, mutex lock l, and condition variable c.

The second blocked notification pattern is:

```
Thread Tw:                    Thread Tn:
lock(l)
lock(c)
wait(c)                       lock(c)
unlock(c)                     lock(l)
unlock(l)                     unlock(l)
                              signal/broadcast(c)
                              unlock(c)
```

The third pattern is:

```
Thread Tw:                  Thread Tn:
lock(l)
lock(c)
wait(c)                     lock(l)
unlock(c)                   lock(c)
unlock(l)                   signal/broadcast(c)
                            unlock(c)
                            unlock(l)
```

The fourth pattern is:

```
Thread Tw:                  Thread Tn:
lock(l)
lock(c)
wait(c)                     lock(c)
unlock(c)                   lock(l)
unlock(l)                   signal/broadcast(c)
                            unlock(l)
                            unlock(c)
```

The other 4 patterns are similar to the 4 above, with the difference that in thread t_w the critical section of c wraps the critical section of l, as in:

```
Thread Tw:
lock(c)
lock(l)
wait(c)
unlock(l)
unlock(c)
```

We propose a technique that can detect all the 8 patterns, but can avoid only the patterns where the critical section of l appears before the critical section of c in thread t_n (§6.2).

A blocked notification involving threads t_w and t_n, mutex lock l, and condition variable c can be defined as a cycle in a RAG formed of request, hold, and wait edges:

$$l \longrightarrow t_w \xrightarrow{c} t_n \longrightarrow l$$

Figure 3.9 graphically illustrates this cycle.

3.3.8 Wait-Notify Deadlocks

A wait-notify deadlock is a situation where a group of threads (or processes) wait to be notified by (respectively receive messages from) one or more threads (respectively processes) from the same group. A wait-notify deadlock involves condition variables or message passing.

We illustrate in Figure 3.10 a wait-notify deadlock involving condition variables: Threads t_1 and t_2 wait on condition variables c_1 and c_2. If the only thread that can resume threads waiting on c_1 (respectively c_2) is t_2 (respectively t_1), the two threads are deadlocked. We graphically illustrate this deadlock in Figure 3.11.

```
//condition variables c1, c2
Thread t1:                          Thread t2:
lock(c1)
wait(c1) //deadlocked here
unlock(c1)
                                    lock(c2)
                                    wait(c2) //deadlocked here
                                    unlock(c2)
lock(c2)                            lock(c1)
signal(c2)                          signal(c1)
unlock(c2)                          unlock(c1)
```

Figure 3.10: Code illustrating a wait-notify deadlock.

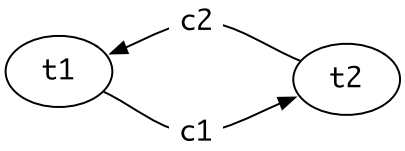

Figure 3.11: A wait-notify deadlock involving threads t_1, t_2 and condition variables c_1, c_2.

Processes exchanging messages can deadlock as illustrated in Figure 3.12. The processes p_1 and p_2 wait for a message on the empty queues q_1 and q_2. If the only process

that can add messages to q_1 (respectively q_2) is p_2 (respectively p_1), the two processes are deadlocked.

```
//message queues q1, q2; both are empty
Process p1:                          Process p2:
m1 = receive(q1)                     m1 = receive(q2)
//new message m2                     //new message m2
send(q2, m2)                         send(q1, m2)
```

Figure 3.12: Wait-notify deadlock involving message passing.

Dimmunix can detect wait-notify deadlocks, but cannot avoid them. Dimmunix avoids deadlocks by only altering the thread schedule, without changing the program's semantics. Wait-notify deadlocks cannot be avoided by just altering the thread schedule, as we explain in §6.1.

3.3.9 Self-Deadlocks

Self-deadlocks can occur when using non-reentrant synchronization constructs. If a resource acquisition is not reentrant, the caller thread can self-deadlock if it attempts to re-acquire a resource that it already holds. For instance, a POSIX Threads mutex is non-reentrant by default; therefore, the application will self-deadlock if a mutex is not set as reentrant and is used in a reentrant way.

Dimmunix can detect self-deadlocks, but cannot avoid them. Avoiding self-deadlocks requires altering the semantics of the application (§6.1).

3.3.10 Non-Deadlock Hangs

Non-deadlock hangs do not involve circular waits; they are caused by infinite loops or missed notifications/events.

Notifications can be missed due to data races, as we show in the code below:

```
Thread Tw:                          Thread Tn:
while running {                     //terminate application
                                    running = false
                                    lock(c)
                                    signal(c)
                                    unlock(c)
  lock(c)
  wait(c)
  unlock(c)
  //process event
}
```

Since the shared variable *running* is not protected by any lock, the following data race can occur: thread t_n sets *running* = *false* to stop the event processing thread t_w; if the threads interleave as shown above, thread t_w still "believes" that *running* = *true*, and the notification is missed, because it occurs prior to the wait call. Since the application was already shut down, no events are going to happen. Therefore, t_w will hang indefinitely.

Dimmunix does not handle non-deadlock hangs. To accurately detect these hangs, and soundly avoid them, semantic knowledge about the application is needed.

3.4 Dimmunix Overview

Programs augmented with a deadlock immunity system develop "antibodies" matching previously encountered deadlocks, and rely on the antibodies to avoid reoccurrences of these deadlocks. With every new deadlock encountered by the program, its resistance to deadlocks is improved, because there is one more deadlock.

The notion of *deadlock signature* is essential to Dimmunix. A deadlock signature is what we called before a deadlock fingerprint—an abstraction of the execution flow that led to deadlock. A deadlock signature must be formed of elements that have the same meaning in any execution of the program. Such elements are the program positions and the call stacks. By instantiation of a deadlock signature, we refer to an execution flow that matches the signature.

To avoid deadlocks, Dimmunix alters the thread schedule by temporarily suspending threads, i.e., making them yield. The effect of yielding is code serialization, i.e., making some code execute sequentially, rather than concurrently.

Dimmunix has two modules, one for deadlock detection and one for deadlock avoidance, as illustrated in Figure 3.13. The detection module updates the RAG upon each synchronization operation; the RAG captures the application's synchronization state. We call the RAG maintained by the detection module "detection RAG" (i.e., RAG_{det}). When a deadlock occurs, we assume that the program is forcefully terminated, either manually by the user or automatically by a watchdog process/thread. Right before the program terminates, the detection module inspects the RAG for deadlocks, i.e., checks for cycles in the RAG. If it finds a deadlock, the detection code extracts the signature of the deadlock and stores it in a persistent *deadlock history*.

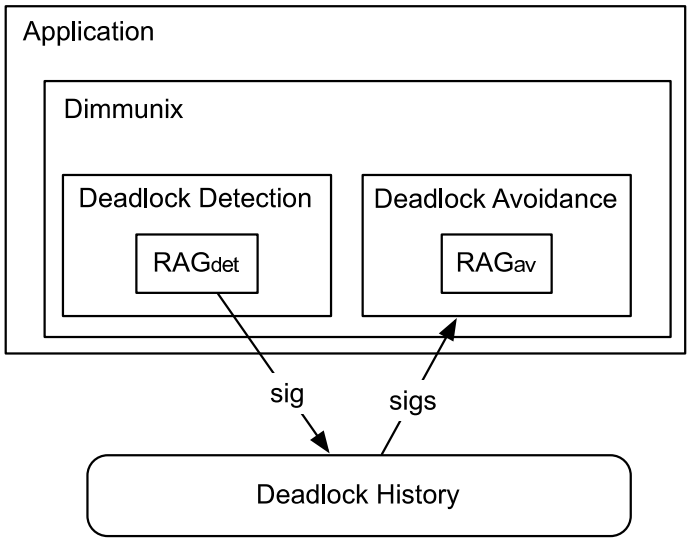

Figure 3.13: The architecture of Dimmunix.

The avoidance module prevents reoccurrences of previously encountered deadlocks, using the signatures from the history. More precisely, Dimmunix avoids instantiations of these signatures, i.e., it prevents execution flows that match these signatures. For a deadlock to occur, its signature must be instantiated. Therefore, by avoiding instantiations of a deadlock's signature, Dimmunix avoids occurrences of that deadlock.

Deadlock avoidance is performed before a resource is acquired; whenever a thread

would instantiate a deadlock signature if allowed to proceed with the resource acquisition, Dimmunix suspends the thread. For instance, the code that avoids mutex deadlocks executes before mutex acquisition statements. If a mutex acquisition would lead to the instantiation of a deadlock signature, Dimmunix delays the acquisition; otherwise, it allows the program to proceed.

Dimmunix keeps the synchronization state relevant to deadlock avoidance in a separate RAG; we call it "avoidance RAG" (i.e., RAG_{av}). The avoidance RAG is decoupled from the detection RAG, and enables Dimmunix to deterministically avoid previously encountered deadlocks.

Dimmunix must update the avoidance RAG upon each synchronization operation (i.e., online), whereas the detection RAG can be updated offline (i.e., when the program terminates) or periodically. The reason is that the deadlock detection can be performed offline or periodically, while the avoidance must be done online.

The calls into Dimmunix can be directly instrumented into the target binary or can reside in the synchronization library. We discuss various layers where Dimmunix can be implemented in §6.5.

Dimmunix runs in the user space, within the address space of the target program, as we illustrate in Figure 3.13. There is a different instance of Dimmunix running withing each process. For all the synchronization constructs (except external locks), the synchronization state is visible only within the currently running process. The synchronization state corresponding to external locks is shared by all the processes.

Chapter 4

Observing and Learning Deadlock Fingerprints

In this chapter, we describe one proposed technique for deadlock detection and for the extraction of deadlock signatures. First, we describe the general technique used to detect deadlocks and generate their signatures (§4.1). Then, we describe how Dimmunix detects mutex, read-write, and semaphore deadlocks and extracts their signatures (§4.2). Finally, we explain the deadlock detection and signature generation for initialization deadlocks (§4.3) and external deadlocks (§4.4).

4.1 Overview

In order to detect deadlocks, Dimmunix needs to maintain the synchronization state in a resource allocation graph (RAG). Dimmunix intercepts each synchronization operation and updates the RAG based on that operation. There are some synchronization operations that need multiple interception points; for instance, a resource acquisition operation needs to be intercepted right before its execution and right after its completion.

The deadlock detection consists of finding (multi)cycles in the RAG. To be able to generate the deadlock signatures, Dimmunix retrieves at runtime the program position (or call stack up to a predefined depth, if more accuracy is needed) of each synchronization operation.

Right before the program is forcefully terminated, Dimmunix detects deadlocks, extracts their signatures, and saves them in the persistent history. Optionally, Dimmunix could detect deadlocks periodically, and automatically terminate the program when it detects a deadlock.

The deadlock recovery mechanisms is orthogonal to Dimmunix; therefore, we chose the simple solution of detecting deadlocks upon program termination. This is good enough for desktop applications, because the user terminates the application when he/she notices that the application hangs. Automated deadlock recovery is needed only for self-managing applications, like servers. Dimmunix could provide a hook in the deadlock detection code, for such 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 [Qin et al., 2007] into this application-specific deadlock resolution hook could provide application-transparent deadlock recovery.

4.2 Detection of Mutex, Read-Write, and Semaphore Deadlocks

In this section, we explain how Dimmunix detects mutex, read-write, and semaphore deadlocks, and generates their signatures (§4.2.1); then, we illustrate the detection and signature generation with an example (§4.2.2); finally, we discuss the correctness of our approach (§4.2.3).

4.2.1 Deadlock Detection and Signature Generation

To be able to detect mutex, read-write, and semaphore deadlocks, Dimmunix intercepts the resource acquisition/release operations and updates the RAG accordingly. The interception is illustrated in Figure 4.1. Right before a mutex lock l, read-write lock l_{rw}, or semaphore s is requested by a thread t, with call stack CS, Dimmunix adds a request edge annotated with CS to the RAG, i.e., $t \xrightarrow{CS} l$, $t \xrightarrow{CS} l_{rw}$, or $t \xrightarrow{CS} s$. Right after the lock/semaphore is acquired, Dimmunix turns the request edge into a hold edge, i.e., $l \xrightarrow{CS} t$, $l_{rw} \xrightarrow{CS} t$, or $s \xrightarrow{CS} t$. Right before a resource is released, the corresponding hold edge is removed from the RAG.

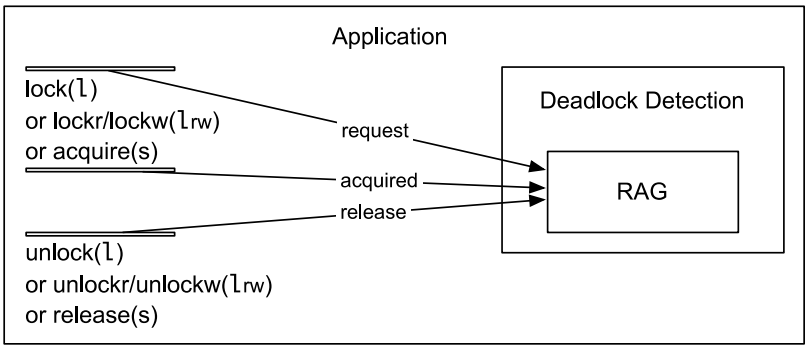

Figure 4.1: Dimmunix intercepts resource acquisition/release operations and updates the RAG.

Dimmunix detects deadlocks involving mutex locks, read-write locks, or semaphores with a unified detection algorithm, that can detect also hybrid deadlocks. Hybrid deadlocks are a generalization of mutex, read-write, and semaphore deadlocks.

We describe the deadlock detection in the *getDeadlock* routine from Algorithm 1. To check if a thread node t is in a deadlock, Dimmunix calls *getDeadlock(t, t)*; the routine explores paths in the RAG recursively, starting from t. If a deadlock is found, the routine returns a cycle. Essentially, the algorithm checks at each point if there exist paths in the RAG that return to t; if they exist, it returns one of these paths (lines 1–12). The following properties must hold along all the paths of a multicycle: (1) for read-write locks, a read-mode request edge $t \xrightarrow{rd} l_{rw}$ must be followed by a write-mode hold edge $l_{rw} \xrightarrow{wr} t'$ (line 4); (2) for a semaphore request edge $t \longrightarrow s$, semaphore s must have no available permits, and thread t must be reachable from all the threads holding permits of s (line 10). We omit the edge labels, as they are not relevant for deadlock detection.

The signature of a deadlock consists of the outer and inner call stacks of the resource acquisition operations involved in the deadlock, i.e., the edge labels along the cycle returned by the *getDeadlock* routine. Assume *getDeadlock* returns the cycle $\{t_1 \xrightarrow{CS_1^{in}} r_2, ..., t_n \xrightarrow{CS_n^{in}} r_1, r_1 \xrightarrow{CS_1^{out}} t_1, ..., r_n \xrightarrow{CS_n^{out}} t_n\}$ representing a deadlock involving threads $t_1, ..., t_n$ and resources $r_1, ..., r_n$; $CS_1^{in}, ..., CS_n^{in}$ are the inner call stacks, i.e., the call stacks threads $t_1, ..., t_{n-1}, t_n$ had at the time of the deadlock, when waiting for resources $r_2, ..., r_n, r_1$; $CS_1^{out}, ..., CS_n^{out}$ are the outer call stacks, i.e., the call stacks that threads $t_1, ..., t_n$ had when they acquired resources

Input: Current thread node t; Destination thread node t_d. Initially, $t = t_d$.
Data: The RAG; threads t', t'', mutex lock l, read-write lock l_{rw}, semaphore s with k permits (i.e., $s^{(k)}$) are nodes in the RAG.
Output: A cycle containing t_d; if no cycle is found, return $\emptyset$.

1 **if** $\exists t \longrightarrow l, l \longrightarrow t' \in RAG\ s.t.\ t' \rightsquigarrow t_d \vee t' = t_d$ **then**
2 **return** $\{t \longrightarrow l, l \longrightarrow t'\} \cup getDeadlock(t', t_d)$
3 **end**
4 **if** $\exists t \xrightarrow{rd} l_{rw}, l_{rw} \xrightarrow{wr} t' \in RAG\ s.t.\ t' \rightsquigarrow t_d \vee t' = t_d$ **then**
5 **return** $\{t \xrightarrow{rd} l_{rw}, l_{rw} \xrightarrow{wr} t'\} \cup getDeadlock(t', t_d)$
6 **end**
7 **if** $\exists t \xrightarrow{wr} l_{rw}, l_{rw} \xrightarrow{rd/wr} t' \in RAG\ s.t.\ t' \rightsquigarrow t_d \vee t' = t_d$ **then**
8 **return** $\{t \xrightarrow{wr} l_{rw}, l_{rw} \xrightarrow{rd/wr} t'\} \cup getDeadlock(t', t_d)$
9 **end**
10 **if** $\exists t \longrightarrow s^{(k)}, s^{(k)} \longrightarrow t' \in RAG\ s.t.\ k = 0 \wedge (\forall s \longrightarrow t'' \in RAG : t'' \rightsquigarrow t_d \vee t'' = t_d)$ **then**
11 **return** $\{t \longrightarrow s, s \longrightarrow t'\} \cup getDeadlock(t', t_d)$
12 **end**
13 **return** $\emptyset$

Algorithm 1: $getDeadlock(t, t_d)$: finds a cycle in the RAG containing thread node t_d; initially, $t = t_d$.

$r_1, ..., r_n$. The signature of the deadlock is:

$$\{(CS_1^{out}, CS_1^{in}), ..., (CS_n^{out}, CS_n^{in})\}$$

For mutex deadlocks, Dimmunix uses call stack suffixes in the signature; for non-mutex deadlocks, it uses only program positions, i.e., call stack suffixes of depth 1. As we explain in §6.3, call stacks of depth > 1 are needed only for mutex deadlocks, to have higher accuracy in the deadlock avoidance.

The top frames of the outer and inner call stacks of a deadlock signature are the lock statements involved in the deadlock. We assume that these statements uniquely delimit the deadlock bug, i.e., if a deadlock involves different outer and/or inner lock statements, then it represents a different deadlock bug. In other words, we assume that all the manifestations of a deadlock bug involve the same outer and inner lock statements.

For a semaphore acquisition at program position p to be considered as part of a deadlock

bug, Dimmunix checks if the following invariant holds: every semaphore s acquired at p is released by a thread that owns permits of s. This invariant is stronger than the one introduced in §3.3.3. The advantage of using this invariant is that it can be evaluated across multiple program executions, because the meaning of a program position is the same in any execution. If the program breaks the invariant, the deadlock signatures containing position p are removed from the history.

4.2.2 An Example

We illustrate the deadlock detection and signature generation on a simple example. Consider the code below:

```
Thread t1:                              Thread t2:
void run () {                           void run() {
  lock(l1) //position p1
                                          lock(l2) //position p2
  f(l2)                                   f(l1)
  unlock(l1)                              unlock(l2)
}                                       }

void f(arg) {
  lock(arg) //position p3, both threads are deadlocked here
  unlock(arg)
}
```

If thread t_1 holds mutex l_1 and thread t_2 simultaneously holds mutex l_2, the two threads deadlock. The outer lock statements are at program positions p_1 and p_2; the inner lock statements are at position p_3. The outer call stacks are $CS_1^{out} = [p_1]$ and $CS_2^{out} = [p_2]$. The inner call stacks are $CS_1^{in} = [p_1, p_3]$ and $CS_2^{in} = [p_2, p_3]$, where p_3 is the top frame.

The RAG is updated as follows. Right before thread t_1 (or t_2) executes *lock(l1)* (respectively *lock(l2)*), Dimmunix adds the request edge $t_1 \xrightarrow{[p_1]} l_1$ (respectively $t_2 \xrightarrow{[p_2]} l_2$) to the RAG. Right after t_1 (or t_2) acquires l_1 (respectively l_2), Dimmunix turns the request edge into the hold edge $l_1 \xrightarrow{[p_1]} t_1$ (respectively $l_2 \xrightarrow{[p_2]} t_2$). Right before t_1 (or t_2) releases l_1 (respectively

l_2), Dimmunix removes the hold edge from the RAG. The RAG updates triggered by the lock/unlock statements within function f are similar.

The signature of this mutex deadlock is formed of the outer and inner call stacks:

$$\{([p_1],[p_1,p_3]),([p_2],[p_2,p_3])\}$$

4.2.3 Correctness Argument

The non-trivial aspects regarding the correctness of the deadlock detection are the ones related to the consistency of the RAG. The RAG updates do not execute atomically with the synchronization operations. Inconsistencies could happen if a hold edge $l \longrightarrow t$ would be removed after mutex l is released. If l is acquired in the meanwhile by another thread t', the new hold edge $l \longrightarrow t'$ may appear in the RAG together with the edge $l \longrightarrow t$. This means that mutex l may appear in the RAG with two owners, i.e., threads t and t'.

The RAG updates have to be consistent with the partial order defined by the semantics of mutex acquisition/release primitives. The release of a mutex always precedes a new acquisition of the mutex; the same order has to be ensured by the RAG updates, i.e., a hold edge $l \longrightarrow t$ has to be removed from the RAG before a new hold edge $l \longrightarrow t'$ is introduced into the RAG. Dimmunix preserves this partial order by removing the hold edge $l \longrightarrow t$ before mutex l is released.

The deadlock detection algorithm that we presented does not have false positives (FPs), even if the user kills the application abruptly (e.g., by using "kill -9"). A FP means that a deadlock is detected even if the program does not deadlock. If no lock inversion happens, the detection algorithm cannot find any deadlock. Without loss of generality, consider the lock inversion in Figure 4.2. No matter when the user kills the application, edges $l_1 \longrightarrow t_1$, $t_1 \longrightarrow l_2$, $l_2 \longrightarrow t_2$, and $t_2 \longrightarrow l_1$ cannot exist simultaneously in the RAG, because the RAG updates are synchronous (i.e., they are not executed in a separate thread).

However, there may be false negatives if the user kills the application when it is about to deadlock. More precisely, this happens when the inner lock statements that would lead to deadlock are not yet executed when the application is killed. We believe this is a minor inconvenience, because it is unlikely that a user terminates an application exactly when a deadlock is about to occur.

```
Thread t1:                          Thread t2:
lock(l1)
lock(l2)
unlock(l2)
unlock(l1)
                                    lock(l2)
                                    lock(l1)
                                    unlock(l1)
                                    unlock(l2)
```

Figure 4.2: Code illustrating a lock inversion.

Remember that a hybrid deadlock is a multicycle. In order to avoid a multicycle, it is enough to break only one cycle from the multicycle. Therefore, it is sufficient for the deadlock detection routine to return one cycle.

4.3 Detection of Initialization Deadlocks

Initialization deadlocks involve class initialization and mutex locks; to detect such deadlocks, Dimmunix needs to monitor class initialization and the acquisitions/releases of mutex locks. The RAG updates performed upon acquisitions/releases of locks were already explained in §4.2.

The RAG updates performed upon class initialization are illustrated in Figure 4.3. The invocation of class C's initializer is conceptually equivalent to synchronizing on a mutex l_C associated with C (§3.3.5). When class C is being loaded on behalf of a thread t, and C's initializer is about to start, Dimmunix adds the request edge $t \xrightarrow{C} l_C$ to the RAG. The edge is annotated with the name of the class. Right after C's initializer starts, Dimmunix turns the request edge into the hold edge $l_C \xrightarrow{C} t$. Right after C's initializer returns, Dimmunix removes the hold edge from the RAG. Even if the hold edge is removed after the initialization, the RAG is in a consistent state, because the initializer is invoked only once.

Consider an initialization deadlock involving threads t_1 and t_2, mutex l, and class C's initializer, as illustrated in the code in Figure 4.4. Thread t_1 acquired mutex l at program position p, and is waiting for C to initialize. Thread t_2 runs C's initializer, and is waiting

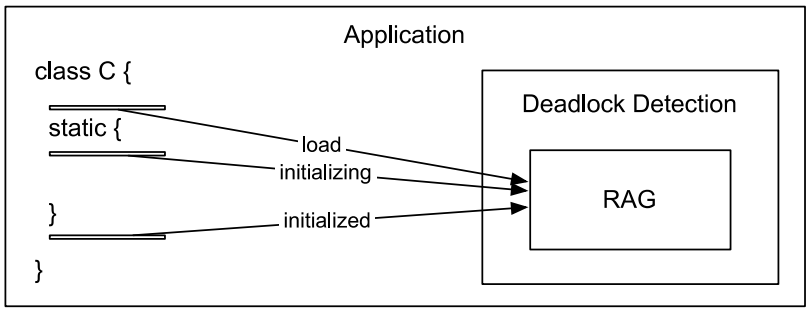

Figure 4.3: Updating the RAG to detect initialization deadlocks.

for l inside the initializer. The signature of this deadlock is:

$$(p, C)$$

The presence of class C's name in the signature means that the deadlock involved class C's initialization. Conceptually, using class C's name in the signature is similar to using the program position corresponding to the first statement of C's initializer. Dimmunix uses program positions instead of call stacks for this type of deadlock, because it is not necessary to have an accurate deadlock avoidance (§6.3).

```
class C {
  static { //lock(lC)
    lock(l) //t2 deadlocked here
    unlock(l)
  } //unlock(lC)
}

Thread t1:                                Thread t2:
lock(l) //position p
                                          new C() //initializing C
new C() //deadlocked here, waiting for C to initialize
unlock(l)
```

Figure 4.4: Code illustrating an initialization deadlock.

4.4 Detection of External Deadlocks

Conceptually, external deadlocks are a form of mutex deadlocks; however, there is a fundamental difference between the two: external deadlocks can happen between multiple processes, because the external locks are shared by all the processes.

Therefore, the detection and avoidance mechanisms are fundamentally different, compared to mutex deadlocks—a process-shared synchronization state is needed. To share the synchronization state among processes, Dimmunix uses a database backed by a SQL server.

Dimmunix handles external deadlocks involving file locks; each time a file f is locked (unlocked), Dimmunix updates a process-shared RAG, as depicted in Figure 4.5. Conceptually, the RAG updates, the deadlock detection, and the signature extraction are similar to the ones described in §4.2 for mutex deadlocks. However, there is a fundamental difference: to update the RAG and detect deadlocks, Dimmunix needs to access the process-shared synchronization state.

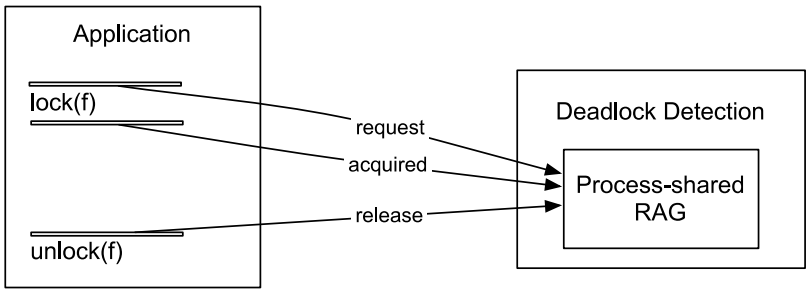

Figure 4.5: Updating the process-shared RAG to detect external deadlocks.

Dimmunix stores the synchronization state in a database backed by a SQL server, as follows: There is a table *Edges* storing the RAG edges, with columns *tid*, *pid*, *state*, *file*, and *pos*, encoding the attributes of an edge: thread id, process id, synchronization state (i.e., request, hold), the name of the file whose lock is requested (held), and the program position where the file is locked. The RAG updates are simple SQL queries that add, modify, or delete records in the Edges table. Also the deadlock history is shared among processes. The SQL server stores the shared history in the *History* table.

Dimmunix handles external deadlocks involving two threads, from possibly different processes. Consider a deadlock involving threads t_1, t_2 and files f_1, f_2, i.e., $f_1 \longrightarrow t_1 \longrightarrow$

$f_2 \longrightarrow t_2 \longrightarrow f_1$. To detect the deadlock and extract its signature, Dimmunix sends the following query to the SQL server:

```
insert into History select e1.pos, e2.pos, e3.pos, e4.pos
  from Edges e1, Edges e2, Edges e3, Edges e4 where
    e1.state = 'hold' and e2.state = 'request' and
    e3.state = 'hold' and e4.state = 'request' and
    e1.tid = e2.tid and e1.pid = e2.pid and e2.file = e3.file and
    e3.tid = e4.tid and e3.pid = e4.pid and e4.file = e1.file and
    e1.file != e3.file and (e1.tid != e3.tid or e1.pid != e3.pid)
```

In the above SQL query, $e_1, ..., e_4$ denote the four edges of the cycle, i.e. $f_1 \xrightarrow{p_1^{out}} t_1$, $t_1 \xrightarrow{p_1^{in}} f_2$, $f_2 \xrightarrow{p_2^{out}} t_2$, and $t_2 \xrightarrow{p_2^{in}} f_1$, where p_1^{in} and p_2^{in} are the positions of the inner lock statements, and p_1^{out} and p_2^{out} are the positions of the outer lock statements. The signature of the deadlock is:

$$\{(p_1^{out}, p_1^{in}), (p_2^{out}, p_2^{in})\}$$

For the signatures, program positions (rather than call stacks) are sufficient, since accuracy is not necessary for the deadlock avoidance (§6.3). Detecting external deadlocks involving a given number $n > 2$ of threads is similar. Since Lu et al. [2008] showed that a deadlock usually involves 2 threads, the Dimmunix prototype handles only external deadlocks of size 2 (i.e., involving 2 threads). For the same reason, handling external deadlocks of an arbitrary size is not necessary.

The deadlock detection approach illustrated above works also for distributed deadlocks, because the process-shared RAG is updated online, i.e., upon each file lock acquisition/release. The SQL server can reside on a dedicated machine; applications running with Dimmunix on different machines can update the shared synchronization state and detect deadlocks by querying the remote SQL server.

Updating the process-shared RAG online is expensive, as we show in §10.2. Therefore, we use the offline detection mechanism described below:

- Dimmunix maintains locally (within a Java process) for each thread t two per-thread queues storing the synchronization state relevant to t: (1) *t.reqs* stores the file locks requested and not acquired yet by t, and (2) *t.acqs* stores the file locks held by t.

An item of a queue is a tuple (f, p), where f is the file whose lock is requested (or held) and p is the program position where f was requested; the position of the acquisition is the same as the request position. A release operation on file f's lock cancels the corresponding acquisition of f's lock, i.e., removes the tuple *(t, f.acqPos)* from *t.acqs*, where *f.acqPos* is the program position where f's lock was acquired.

- When the application terminates, Dimmunix uploads for each thread t the queues *t.reqs* and *t.acqs* to the SQL server. More precisely, for each tuple (f, p) from *t.reqs* (or *t.acqs*) Dimmunix sends a query to the SQL server to add the request edge $t \xrightarrow{p} f$ (respectively the hold edge $f \xrightarrow{p} t$) to the RAG. Then, Dimmunix sends the deadlock detection query shown above to the SQL server.

The offline deadlock detection substantially improves Dimmunix's performance (compared to the online detection), because maintaining the two queues is cheap (§10.2).

Chapter 5

Avoiding Known Deadlocks at Runtime

In this chapter, we explain how Dimmunix modifies dynamically the thread schedule to deterministically avoid previously encountered deadlocks. First, we give an overview of the avoidance mechanism (§5.1). Then, we explain the avoidance of previously encountered deadlocks that involve mutexes, read-write locks, and/or semaphores (§5.2). Finally, we explain how Dimmunix avoids previously encountered initialization deadlocks (§5.3) and external deadlocks (§5.4).

5.1 Overview

Remember from §4.1 that the signature of a deadlock approximates execution flows leading to that deadlock. The signature of a mutex deadlock consists of call stacks. The signatures of non-mutex deadlocks consist of program positions.

We define the notions of signature match and signature instantiation. A signature match is an execution flow matching a signature from history. A signature instantiation is a signature match where all the threads hold (or are committed to wait for) the resources they requested when matching the signature.

More precisely, a program instantiates a deadlock signature S if there is a group of threads that simultaneously hold (or wait for) resources at program positions (or with call stacks) that belong to S. If at least one thread is not yet committed to wait for the resource it requested (i.e., it just called or is about to call the resource acquisition routine), S is only

matched, but not instantiated. For some deadlock types (e.g., mutex deadlocks), not all the elements of a signature are involved in signature matching. For instance, only the outer call stacks of a mutex deadlock signature are matched against the program execution.

To avoid deadlocks that were previously encountered, Dimmunix avoids instantiations of their signatures. To avoid instantiations of a signature S, Dimmunix prevents the execution flows matching S from turning into instantiations of S.

Dimmunix avoids deadlocks by delaying resource acquisitions. The avoidance code executes right before the resource acquisition/release operations. When a thread t is about to acquire a resource r at a program position (or with a call stack) that was previously involved in a deadlock, i.e., it is part of a signature from the history, Dimmunix decides to either allow t to proceed with the acquisition, or suspend t. If thread t would instantiate a deadlock signature if allowed to proceed, Dimmunix suspends t. If thread t would not instantiate any signature, Dimmunix allows t to proceed with the acquisition. When thread t is about to release resource r, Dimmunix resumes the threads that are yielding (i.e., are suspended by Dimmunix) due to signature instantiations involving t.

5.2 Avoidance of Deadlocks Involving Mutexes, Read-Write Locks, and/or Semaphores

In this section, we describe the algorithm Dimmunix uses to avoid mutex, read-write, and semaphore deadlocks (§5.2.1); then, we illustrate the avoidance on an example (§5.2.2); finally, we discuss the correctness of the avoidance algorithm (§5.2.3).

5.2.1 Deadlock Avoidance

In this section, we describe the avoidance mechanism for hybrid deadlocks (i.e., deadlocks involving mutexes, read-write locks, and/or semaphores), because they are a generalization of mutex, read-write, and semaphore deadlocks.

To avoid hybrid deadlocks, Dimmunix intercepts resource acquisition/release operations, as we illustrate in Figure 5.1. When a thread t requests a resource x, Dimmunix

decides to allow t to proceed if the acquisition of x would not instantiate any deadlock signature from the history; otherwise, Dimmunix makes t yield until the acquisition of x no longer risks instantiating any signature from the history. If the decision is "allow", t can proceed with the acquisition of x, and Dimmunix adds an *allow* edge $t \xrightarrow{A} x$ to the avoidance RAG; the allow edge is labeled with the call stack CS that t currently has, i.e., $t \xrightarrow{A@CS} x$. When thread t is about to release x, Dimmunix removes the allow edge from the avoidance RAG; therefore, the meaning of the allow edge is that t was allowed to acquire or holds x. If the decision is "yield", it means that thread t would be involved in a signature instantiation if allowed to proceed; therefore, t yields, and Dimmunix adds a *yield* edge $t \xrightarrow{Y} t'$ to the avoidance RAG for each thread t' ($\neq t$) involved in the signature instantiation. The labeling of yield edges is described later in this section.

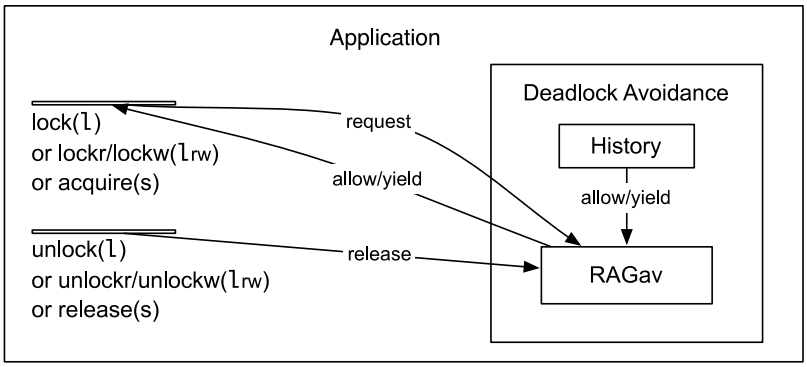

Figure 5.1: Interception of resource acquisition and release operations enables the avoidance of deadlocks.

The avoidance RAG is completely decoupled from the detection RAG. More precisely, the avoidance code maintains the allow and yield edges independently from the detection code, which maintains the request, hold, and wait edges. We denote by RAG_{av} the avoidance RAG.

Dimmunix avoids deadlocks that it previously detected by avoiding instantiations of their signatures stored in the history. To find instantiations of a deadlock signature S, Dimmunix matches the program execution against the outer call stacks of S, up to a predefined matching depth (e.g., 10). A matching depth of 1 would imply that only the top frames of S's outer call stacks are compared; the top frames of S's call stacks are the outer and inner resource acquisition statements involved in the deadlock. Since a deadlock bug is

uniquely delimited by the top frames of its signature (i.e., a signature with different top frames represents a different deadlock), a matching depth of 1 enables Dimmunix to avoid any manifestation of the deadlock that S represents, albeit with too much generality (i.e., even deadlock-free executions might appear as being matches). We discuss the matching accuracy in more detail in §6.3.

We define now in detail what it means for a signature to be instantiated. The instantiation I of a deadlock signature $S = \{(CS_1^{out}, CS_1^{in}), ..., (CS_n^{out}, CS_n^{in})\}$ involving threads $t_1, ..., t_n$ and resources $r_1, ..., r_n$, is represented as

$$I = \{(t_1, r_1, CS_1^{out}), ..., (t_n, r_n, CS_n^{out})\}$$

Signature S is instantiated if and only if threads t_i are allowed to acquire (or already hold) r_i, and had call stacks CS_i^{out} when they requested r_i. This can be expressed as:

$$\forall i \in \overline{1,n} : t_i \overset{A@CS_i^{out}}{\longrightarrow} r_i \in RAG_{av}$$

To check if a thread t would instantiate S if allowed to acquire resource r, Dimmunix tentatively adds the allow edge $t \overset{A}{\longrightarrow} r$ to RAG_{av}, then checks if t and r can be part of a group of threads $t_1, ..., t_n$ and resources $r_1, ..., r_n$ that satisfies the instantiation property described above. If yes, allowing thread t to proceed would lead to the instantiation of S.

We explain now the construction of the yield edges. Consider a signature instantiation $\{(t_1, r_1, CS_1^{out}), ..., (t_n, r_n, CS_n^{out})\}$, and assume with no loss of generality that thread t_1 is the one that would cause the instantiation if allowed to acquire resource r_1. Dimmunix inserts into RAG_{av} the yield edges $t_1 \overset{Y@CS_2^{out}}{\longrightarrow} t_2, ..., t_1 \overset{Y@CS_n^{out}}{\longrightarrow} t_n$, annotated with the call stacks $CS_2^{out}, ..., CS_n^{out}$ that threads $t_2, ..., t_n$ had when they were allowed to acquire resources $r_2, ..., r_n$.

The avoidance mechanism is described in detail in Algorithm 2. Thread t acquires resource r with call stack CS (line 14), then releases r (line 23). Right before the acquisition operation, Dimmunix avoids deadlocks (lines 1–12) as follows: The allow edge $t \overset{A@CS}{\longrightarrow} r$ is tentatively added to RAG_{av} (line 1). If there is a signature match, i.e., the allow edge would cause an instantiation $I = \{(t, r, CS), (t_2, r_2, CS_2), ..., (t_n, r_n, CS_n)\}$ of a signature S from the

history (line 2), Dimmunix then removes the tentative allow edge (line 3), adds I to the set *instances[S]* storing the current instantiations of S (line 6), and adds the yield edges $t \xrightarrow{Y@CS_2} t_2$, ..., $t \xrightarrow{Y@CS_n} t_n$ to RAG_{av} (line 7). Then, thread t yields by waiting until Dimmunix removes I from *instances[S]*, i.e., until t_2 releases resource r_2, ..., or t_n releases resource r_n (line 8). When t resumes, Dimmunix removes the yield edges from RAG_{av} (line 9), reintroduces the allow edge into RAG_{av} (line 11), and checks again for signature matches (line 2). If there are no more signature matches involving thread t, Dimmunix allows thread t to proceed with the acquisition of r (line 13). When t is about to release r, Dimmunix performs the following steps: removes the allow edge $t \xrightarrow{A@CS} r$ from RAG_{av} (line 15); for each avoided instantiation I of a signature S that contains the triple (t, r, CS), Dimmunix resumes all the threads waiting for I to be removed from *instances[S]* (lines 16–21).

Just as in any deadlock avoidance technique based on altering the thread schedule, Dimmunix's avoidance mechanism may cause starvation: yield edges may introduce so called *yield cycles*, i.e., deadlock situations caused by the threads' yielding. Dimmunix saves the signature of a yield cycle in the history, exactly as it does for a normal deadlock signature.

Dimmunix avoids yield cycles the same way it avoids normal deadlocks. We illustrate such a cycle in Figure 5.2 corresponding to the code in Figure 5.3. The yield cycle involves thread t_1 and t_2, and mutexes l_1, l_2, and l_3. Thread t_1 is avoiding an instantiation of the deadlock signature $S = \{(CS_1^{out}, CS_1^{in}), (CS_2^{out}, CS_2^{in})\}$. More precisely, thread t_1 acquired (and still holds) l_1 with call stack $CS_1'^{out}$ and requests l_2 with call stack CS_1^{out}; thread t_2 acquired (and still holds) l_3 with call stack CS_2^{out} and waits for l_1 with call stack $CS_2'^{in}$. Thread t_1 yields until t_2 releases l_3; allowing t_1 to proceed would cause an instantiation of S. At the same time, t_2 waits for lock l_1 held by t_1. This is a deadlock situation caused by Dimmunix's avoidance mechanism. The signature of the avoidance-induced deadlock is $\{(CS_1'^{out}, CS_1^{out}), (CS_2^{out}, CS_2'^{in})\}$.

The yield cycle and the avoided deadlock are nested, as shown in Figure 5.3. The outer (respectively inner) lock statements of the deadlock are the top frames of the call stacks CS_1^{out} and CS_2^{out} (respectively CS_1^{in} and CS_2^{in}). The outer (respectively inner) lock statements of the yield cycle are the top frames of the call stacks $CS_1'^{out}$ and CS_2^{out} (respectively CS_1^{in} and $CS_2'^{in}$). In thread t_1's code, the outer critical section of the yield cycle (i.e., the one

Input: Thread t, resource r, call stack CS; t acquires r, then releases r.
Data: RAG_{av}; Deadlock history *Hist*; The set *instances[S]* for each signature S in history, storing the current instantiations of S.

1 $RAG_{av} := RAG_{av} \cup \{t \xrightarrow{A@CS} r\}$
2 **while** *Edge* $t \xrightarrow{A@CS} r$ *causes an instantiation of a signature* $S \in Hist$ **do**
3 $\quad RAG_{av} := RAG_{av} \setminus \{t \xrightarrow{A@CS} r\}$
4 $\quad$ Let $I = \{(t,r,CS),(t_2,r_2,CS_2),...,(t_n,r_n,CS_n)\}$ be the instantiation of S
5 $\quad$ *lock(S.condVar)* //*S.condVar* is the condition variable associated to S
6 $\quad instances[S] := instances[S] \cup \{I\}$
7 $\quad RAG_{av} := RAG_{av} \cup \{t \xrightarrow{Y@CS_2} t_2,...,t \xrightarrow{Y@CS_n} t_n\}$
8 $\quad$ *wait(S.condVar)* //wait until I is removed from $instances[S]$
9 $\quad RAG_{av} := RAG_{av} \setminus \{t \xrightarrow{Y@CS_2} t_2,...,t \xrightarrow{Y@CS_n} t_n\}$
10 $\quad$ *unlock(S.condVar)*
11 $\quad RAG_{av} := RAG_{av} \cup \{t \xrightarrow{A@CS} r\}$
12 **end**
13 **acquire** r
14 //critical section
15 $RAG_{av} := RAG_{av} \setminus \{t \xrightarrow{A@CS} r\}$
16 **foreach** $S \in Hist, I \in instances[S]$ *where* $(t,r,CS) \in I$ **do**
17 $\quad$ *lock(S.condVar)*
18 $\quad instances[S] := instances[S] \setminus \{I\}$
19 $\quad$ *broadcast(S.condVar)* //resume all threads waiting for I to be removed from $instances[S]$
20 $\quad$ *unlock(S.condVar)*
21 **end**
22 **release** r

Algorithm 2: Avoidance of hybrid deadlocks.

delimited by the *lock*(l_1) and *unlock*(l_1) statements) wraps the outer critical section of the deadlock (i.e., the one delimited by the *lock*(l_2) and *unlock*(l_2) statements).

For mutex deadlocks involving at least 3 threads, there exist at least 2 yield edges emerging from a thread node. Therefore, the yield cycles caused by avoiding such deadlocks are multicycles. However, since deadlocks involving more than two threads are not common [Lu et al., 2008], we have chosen to illustrate a simple yield cycle, involving only one yield edge.

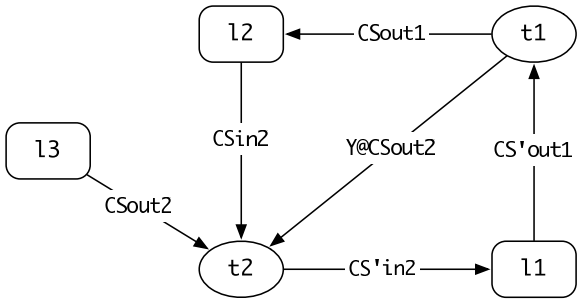

Figure 5.2: Yield cycle.

```
Thread t1:                              Thread t2:
lock(l1) //CS'out1
lock(l2) //CSout1, yield here
                                        lock(l3) //CSout2
lock(l3) //CSin1, deadlock here         lock(l2) //CSin2, deadlock here
unlock(l3)                              lock(l1) //CS'in2, starved here
unlock(l2)                              unlock(l1)
unlock(l1)                              unlock(l2)
                                        unlock(l3)
```

Figure 5.3: Code illustrating an avoidance-induced deadlock.

Right after Dimmunix detects a yield cycle and saves its signature, Dimmunix breaks the yield cycle by resuming one of the yielding threads. In other words, Dimmunix stops the deadlock avoidance. Since a deadlock usually involves two threads, there is usually one yielding thread. The starvation situation will not reoccur, because the signature of the avoidance-induced deadlock is saved in the history.

Having the threads starved is as bad as reencountering the deadlock. However, since deadlocks can be non-deterministic, resuming the yielding threads may not result in a reoccurrence of the avoided deadlock. Therefore, the design choice of breaking the starvation is the best one.

We also implemented the avoidance technique described in [Nir-Buchbinder et al., 2008, Boronat and Cholvi, 2003], in order to compare its performance and accuracy to Dimmunix's deadlock avoidance. To avoid deadlocks, [Nir-Buchbinder et al., 2008, Boronat and Cholvi, 2003] use a "gate lock" for each discovered deadlock bug; the gate lock is acquired each time an outer lock statement involved in the deadlock is about to execute. We

implemented in the Java Dimmunix prototype this avoidance technique; we call it *simple avoidance*. We also implemented a variation of this technique, called *hybrid avoidance*, which uses call stack information—the gate lock is acquired only when an outer call stack of the deadlock signature is matched. We implemented the simple and hybrid avoidance techniques only for mutex deadlocks. We compare the accuracy and efficiency of Dimmunix's avoidance mechanism to the simple and hybrid avoidance in §10.1.2.

More precisely, the simple and hybrid avoidance techniques work as follows: Dimmunix associates to each signature S in history a semaphore $S.sem$; initially $S.sem.v = 1$. Whenever a thread t requests a lock l with call stack CS, Dimmunix checks if CS matches call stacks of signatures in the history. Consider that CS matches call stacks in the signatures $S_1, ..., S_n$. Before allowing thread t to proceed with the acquisition of l, Dimmunix acquires $S_1.sem, ..., S_n.sem$, in increasing order of the signature ids. Before allowing t to release l, Dimmunix releases the semaphores $S_1.sem, ..., S_n.sem$; the order in which the semaphores are released is not important. The only difference between the simple and hybrid avoidance is that the simple avoidance uses a call stack matching depth of 1, while the hybrid avoidance can use an arbitrary matching depth. Dimmunix uses semaphores to be able to bypass avoidance-induced deadlocks. Bypassing a starvation situation caused by deadlock avoidance consists of releasing the semaphores acquired in the avoidance process.

5.2.2 An Example

We illustrate the avoidance mechanism on a simple example of a mutex deadlock, depicted in Figure 5.4. We reuse the code example from §4.2.2.

Assume that the program deadlocks; the deadlock signature is:

$$S = \{([p_1], [p_1, p_3]), ([p_2], [p_2, p_3])\}$$

In the next program execution, Dimmunix avoids the deadlock as follows: Say thread t_2 acquires mutex l_2, then thread t_1 requests mutex l_1. Therefore, there are two allow edges in RAG_{av}: $t_2 \xrightarrow{A@[p2]} l_2$ and $t_1 \xrightarrow{A@[p1]} l_1$. Edge $t_1 \xrightarrow{A@[p1]} l_1$ is only tentatively added, i.e., t_1 is not

```
Thread t1:                                Thread t2:
void run () {                             void run() {
  lock(l1) //position p1
                                            lock(l2) //position p2
  f(l2)                                     f(l1)
  unlock(l1)                                unlock(l2)
}                                         }

void f(arg) {
  lock(arg) //position p3, both threads are deadlocked here
  unlock(arg)
}
```

Figure 5.4: Code example to illustrate the deadlock avoidance.

allowed to proceed yet. Dimmunix detects the instantiation of signature S:

$$\{(t_1, l_1, [p_1]), (t_2, l_2, [p_2])\}$$

Thread t_1 yields until the allow edge $t_2 \xrightarrow{A@[p2]} l_2$ disappears from RAG_{av}, i.e., until t_2 releases l_2. Once this happens, there are no more instantiations possible for S, and Dimmunix allows t_1 to proceed. In this way, the deadlock is avoided, and will be avoided in every future execution. Therefore, the deadlock has been eradicated without modifying the program.

5.2.3 Correctness Argument

Dimmunix guarantees that, as long as a thread t can cause an instantiation of a signature from history, t yields, i.e., does not proceed with the resource acquisition. Hence, Dimmunix prevents execution flows that instantiate signatures from history. For simplicity, we assume that the matching depth is 1. For matching depths > 1, the reasoning is similar.

For a deadlock to occur, its signature must be instantiated. Since Dimmunix avoids any instantiation of a signature from the history, and all deadlocks that Dimmunix previously detected have their signatures in the history, Dimmunix avoids any reoccurrence of a deadlock that it previously encountered.

Dimmunix's avoidance mechanism is thread-safe, i.e., threads cannot instantiate signatures. The most important property that ensures the thread-safety is: the tentative introduction of allow edges works like a barrier; if two threads are simultaneously crossing the barrier, one of them will "see" the other thread's allow edge, after crossing the barrier. The thread observing the other allow edge will yield. Therefore, no global lock is needed for the avoidance mechanism. We also needed fine-grained synchronization to ensure thread-safety; we chose not to include that synchronization in Algorithm 2, for a better readability.

The avoidance mechanism is eventually starvation-free. We know that a yield cycle wraps the avoided deadlock (§5.2.1). Therefore, the sum of the nesting levels of the outer lock statements is smaller for the yield cycle, compared to the deadlock. This sum is finite for the original deadlock and it decreases with each yield cycle. If the sum is 0, there can be no yield cycle (i.e., no starvation), because the nesting level of any lock statement is ≥ 0. Therefore, the maximum number of yield cycles is equal to the value of this sum for the original deadlock.

In practice, we have not encountered any starvation situation caused by the deadlock avoidance, for any real deadlock that Dimmunix avoided. Therefore, we believe that such starvation situations are not common.

5.3 Avoidance of Initialization Deadlocks

Conceptually, the avoidance of initialization deadlocks (i.e., deadlocks involving mutex locks and class initialization) can be implemented exactly as the avoidance of hybrid deadlocks, because initialization deadlocks can be reduced to mutex deadlocks (§3.3.5). However, since a class initializer executes only once, and waiting for a class to initialize is not a synchronization operation per se, we need a different avoidance technique.

The avoidance of initialization deadlocks is asymmetric, because a signature is asymmetric: Consider the signature (p, C) from §4.3, where p is the program location of the "offending" mutex acquisition, and C is the class whose initializer deadlocked. Right before a thread t requests mutex l at position p, Dimmunix avoids instantiating the signature (p, C) as shown in Algorithm 3. First, Dimmunix adds the allow edge $t \xrightarrow{A@p} l$ to RAG_{av} (line 1). Then, if class C is initializing, thread t waits until C initializes (lines 2–4). Finally,

when thread t is about to release l, Dimmunix removes the allow edge from RAG_{av} (line 7). Right before class C's initializer is about to start, Dimmunix avoids instantiations of (p,C) as shown in Algorithm 4. First, Dimmunix sets the *initializing[C]* flag to *true* (line 1). The thread loading class C waits until there are no threads allowed to acquire (or holding) a mutex at position p, i.e., there are no allow edges labeled with p (lines 2–4). Finally, after class C's initializer returns, Dimmunix announces that class C is initialized (line 6).

Input: Thread t; Mutex l; Program position p.
Data: The deadlock history *Hist*.
1 $RAG_{av} := RAG_{av} \cup \{t \xrightarrow{A@p} l\}$
2 **while** $\exists S = (p,C) \in Hist$ *where initializing[C]* **do**
3 wait until class C is initialized
4 **end**
5 **lock(x)**
6 //critical section
7 $RAG_{av} := RAG_{av} \setminus \{t \xrightarrow{A@p} l\}$
8 notify threads waiting for edge $t \xrightarrow{A@p} l$ to be removed from RAG_{av}
9 **unlock(x)**
Algorithm 3: Avoiding initialization deadlocks before the offending mutex acquisition.

Input: Thread t, class C
Data: The deadlock history *Hist*
1 *initializing[C] := true*
2 **while** $\exists (p,C) \in Hist, t' \xrightarrow{A@p} l \in RAG_{av}$ **do**
3 wait until the allow edge $t' \xrightarrow{A@p} l$ is removed from RAG_{av}
4 **end**
5 **call C's initializer**
6 *initializing[C] := false*
7 notify threads waiting for class C to initialize
Algorithm 4: Avoiding initialization deadlocks when a class initializer is about to start.

Dimmunix does not handle initialization deadlocks involving more than two threads, or other types of resources (e.g., read-write locks, semaphores), and does not deal with yield cycles for this type of deadlock. Since mutex deadlocks are likely the most frequent among all deadlock types (§3.3.1), we chose to offer a complete immunity system only for mutex

deadlocks.

5.4 Avoidance of External Deadlocks

To avoid external deadlocks (i.e., deadlocks involving synchronization on external locks, like file locks), Dimmunix needs to maintain the allow edges in a process-shared avoidance RAG. In order to avoid signature instantiations, Dimmunix inspects the process-shared avoidance RAG upon each request of an external lock. Dimmunix avoids only external deadlocks involving file locks and two threads from possibly different processes.

As we explained in §4.4, Dimmunix keeps the shared synchronization state in a SQL database. Dimmunix stores the allow edges in the *Edges* table that we introduced in §4.4, using the *state* attribute "allow". Dimmunix stores the deadlock history in the *History* table, with attributes *"Pout1"*, *"Pin1"*, *"Pout2"*, and *"Pin2"*, denoting the outer and inner positions of a deadlock signature.

Right before a thread with id *tid* from process with id *pid* attempts to lock file f at program position p, Dimmunix first adds the allow edge to the shared RAG_{av}, using the SQL query below:

```
insert into Edges values (tid, pid, 'allow', f, p)
```

Then, Dimmunix checks for instantiations of signatures from the deadlock history, i.e., two allow edges involving (1) two different threads, one of which being thread *tid* from process *pid*, (2) two different files, one of which being f, and (3) both positions of a signature, one of them being p. The SQL query which checks for signature instantiations is:

```
select * from Edges e1, Edges e2, History h where
 e1.state = 'allow' and e2.state = 'allow' and
 e1.tid = tid and e1.pid = pid and e1.file = f and e1.pos = p and
 (e1.tid != e2.tid or e1.pid != e2.pid) and e1.file != e2.file and
 e1.pos = h.Pout1 and e2.pos = h.Pout2
```

Right before file f is unlocked, Dimmunix removes the allow edge using the following SQL query:

```
delete from Edges e where
   e.tid = tid and e.pid = pid and
   e.state = 'allow' and e.file = f and e.pos = p
```

The deadlock avoidance approach illustrated above works for distributed deadlocks as well. The SQL server can reside on a dedicated machine; applications running with Dimmunix on different machines can update/inspect the shared synchronization state and avoid deadlocks by querying the remote SQL server.

Since it is expensive to update the shared avoidance RAG online (i.e., upon each acquisition/release of a file lock), as we show in §10.2, we chose to implement in the Dimmunix prototype the simplified avoidance mechanism described below:

- Dimmunix associates a file lock $S.l$ with each signature S in the history of external deadlocks.
- When a thread t attempts to lock a file f at a program position p, Dimmunix checks if p belongs to signatures in the history. Let $S_1, ..., S_n$ be the signatures containing p, in the increasing order of their ids (every signature has a unique id). Before allowing t to proceed with the acquisition of f's lock, Dimmunix acquires $S_1.l, ..., S_n.l$.
- When a thread t is about to unlock a file f, Dimmunix checks if $f.acqPos$ (the program position where f's lock was acquired) belongs to a signature in the history. Let $S_1, ..., S_n$ be the signatures containing $f.acqPos$. Before allowing t to release f's lock, Dimmunix releases $S_1.l, ..., S_n.l$.

This avoidance technique is cheap, as we show in §10.2. However, it has a shortcoming: it cannot be distributed, because the file locks associated with deadlock signatures are visible only from processes running on the same machine. Therefore, this avoidance mechanism cannot be used to avoid deadlocks among processes running on different machines.

Dimmunix does not handle external deadlocks involving more than two threads or other types of external locks, and does not handle yield cycles for this type of deadlock. Since mutex deadlocks are likely the most frequent among all deadlock types (§3.3.1), we chose to offer a complete immunity system only for mutex deadlocks.

Chapter 6

Discussion

In this chapter, we discuss various aspects related to Dimmunix. First, we describe the unavoidable deadlock patterns (§6.1). We present a tentative immunization technique for blocked notifications and describe its limitations in §6.2. Then, we discuss the false positives and false negatives in the context of Dimmunix (§6.3), and describe how Dimmunix may disable functionality (§6.4). Finally, we compare the two layers where Dimmunix can be implemented: platform-wide, i.e., within the synchronization library, and application-level, i.e., attached to a specific application (§6.5).

6.1 Unavoidable Deadlocks

Remember that Dimmunix avoids deadlocks by only altering thread schedules. Any intervention that changes the semantics of the running program is unacceptable. In this section, we present the deadlock patterns that are unavoidable without changing the semantics of the program.

The unavoidable deadlock patterns are the wait-notify deadlocks, self-deadlocks, and 6 patterns of blocked notifications. Dimmunix can only detect these deadlocks. We explain the 6 unavoidable blocked notification patterns in §6.2.3.

A wait-notify deadlock involves circular waits for notifications, as we illustrate in the code in Figure 6.1. Thread t_1 waits for thread t_2 to send a notification, while t_2 waits for t_1 to send a notification. Therefore, the two threads are deadlocked.

```
//condition variables c1, c2
Thread t1:                              Thread t2:
lock(c1)
wait(c1) //deadlocked here
unlock(c1)
                                        lock(c2)
                                        wait(c2) //deadlocked here
                                        unlock(c2)
lock(c2)                                lock(c1)
signal(c2)                              signal(c1)
unlock(c2)                              unlock(c1)
```

Figure 6.1: Code illustrating a wait-notify deadlock.

A wait-notify deadlock cannot be avoided by just altering the thread schedules. No matter how threads t_1 and t_2 interleave, they are going to deadlock when they will execute the wait calls.

A self-deadlock is a situation where a thread is executing a non-reentrant synchronization operation on a resource that it already holds (§3.3.9). Consider the C code below:

```
Thread t:
//x is a mutex, not configured to be reentrant
pthread_mutex_lock(x);
pthread_mutex_lock(x); //blocked here
```

Mutex x is non-reentrant, therefore the caller thread t will self-deadlock at the second lock statement, and hang indefinitely.

A self-deadlock cannot be avoided, unless the program semantics are changed. The above self-deadlock can be avoided only if Dimmunix automatically turns x into a reentrant mutex, or simply ignores the second lock statement, which is conceptually the same as making x reentrant. Both these interventions change the semantics of the application. The programmer may rely on the fact that the mutex acquisitions are non-reentrant. For instance, assume that a piece of code which is not supposed to execute reentrantly; making mutex x reentrant would break this protection mechanism, and uncover another bug in the code, which may be worse than the self-deadlock.

6.2 Blocked Notifications: At the Frontier Between Starvation and Deadlock

In this section, we describe a tentative immunization approach for blocked notifications. First, we propose a technique for detecting and extracting signatures of blocked notifications (§6.2.1). Then, we propose a technique for avoiding previously encountered blocked notifications (§6.2.2). Finally, we describe the limitations of these techniques (§6.2.3). We implemented these techniques in the Java Dimmunix prototype and evaluated their impact on performance in §10.2.

6.2.1 Detection of Blocked Notifications

To detect blocked notifications (i.e., deadlocks involving condition variables and mutex locks), Dimmunix needs to intercept operations on mutexes (i.e., lock, unlock) and condition variables (i.e., wait, signal, broadcast); Dimmunix updates the RAG each time such an operation executes. The RAG updates triggered by mutex operations are the ones explained in §4.2.

Dimmunix updates the RAG upon each wait, signal, or broadcast operation, as depicted in Figure 6.2. Right before a *wait(c)* call performed on a condition variable c by a thread t, at program position p, Dimmunix adds to the RAG a wait edge $t \xrightarrow{c@p} t'$ for each potential notifier thread t', i.e., a thread that may later execute *signal(c)* or *broadcast(c)*. The wait edge is annotated with position p and condition variable c. Right after the wait call returns, Dimmunix removes the wait edge from the RAG.

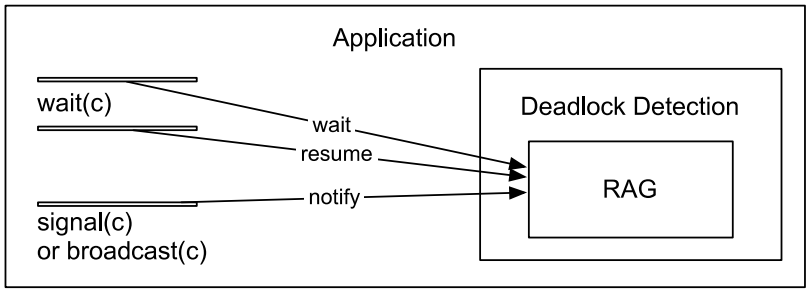

Figure 6.2: Updating the RAG to detect blocked notifications.

A blocked notification occurs when a waiter thread t_w holds a mutex l while waiting on a condition variable c, and a notifier thread t_n is waiting to acquire l and may later execute *signal(c)* or *broadcast(c)* if it was not blocked by l. Predicting what thread t_n will do if it was not blocked by mutex l is undecidable; therefore, a blocked notification does not necessarily represent a deadlock. If thread t_n does not "intend" to later execute *signal/broadcast(c)*, there is no deadlock—we have only a starvation situation where thread t_w starves thread t_n by holding lock l. Another reason for which a blocked notification is not a definite deadlock is the fact that there may be other threads than t_n that may notify thread t_w. Moreover, the set of potential notifier threads (i.e., that may notify t_w) may dynamically change, and predicting this set at runtime is undecidable.

To detect blocked notifications, Dimmunix needs to estimate the set *PN[c]* of potential notifiers for each condition variable c. In the context of blocked notification detection, a wait edge $t_w \xrightarrow{c} t_n$ means that thread t_w is blocked on a *wait(c)* call and thread t_n is a potential notifier, i.e., $t_n \in PN[c]$. There is a blocked notification involving a waiter thread t_w, a condition variable c, and a mutex lock l if and only if the formula below holds:

$$\exists l \longrightarrow t_w, t_w \xrightarrow{c} t_n, t_n \longrightarrow l \in RAG$$

The blocked notification can be formally represented as a RAG cycle $l \xrightarrow{p^l_w} t_w \xrightarrow{c@p^c_w} t_n \xrightarrow{p^l_n} l$. The edges are labeled with the program position p^l_w where thread t_w acquired l, the position p^c_w where t_w is waiting on c, and the position p^l_n where thread t_n is waiting for l. The signature of the blocked notification is a triple formed of the three program positions:

$$(p^l_w, p^c_w, p^l_n)$$

Dimmunix uses program positions in the signature, instead of call stacks, because accuracy is not needed for the deadlock avoidance (§6.3).

To compute the set $PN[c]$ for a condition variable c, Dimmunix uses the following heuristic: a thread is a potential notifier if it previously executed *signal(c)* or *broadcast(c)* in the current execution. This heuristic misses blocked notifications that occur at the first *signal/broadcast(c)* call. In other words, if the set $PN[c]$ is incomplete, our detection technique has false negatives (FNs). However, we believe that a blocked notification involving

c usually does not occur at the first execution of a *wait(c)* call.

6.2.2 Avoidance of Blocked Notifications

Dimmunix avoids blocked notifications by delaying wait calls previously involved in blocked notifications. Consider a blocked notification involving waiter thread t_w, notifier thread t_n, mutex l, and condition variable c. We illustrate it in the code below:

```
Thread Tw:                          Thread Tn:
lock(l) //position Pwl
lock(c)
wait(c) //position Pwc              lock(l) //position Pnl
unlock(c)                           unlock(l)
unlock(l)                           lock(c) //position Pnc
                                    signal/broadcast(c)
                                    unlock(c)
```

Thread t_w holds mutex l, which it acquired at program position p_w^l; t_w is waiting on condition variable c, at position p_w^c. Thread t_n requests mutex l at position p_n^l, and "intends" to notify t_w (by calling *signal(c)* or *broadcast(c)*), after acquiring l. The signature of the blocked notification is (p_w^l, p_w^c, p_n^l).

To avoid reoccurrences of the same blocked notification, Dimmunix enforces the following thread interleaving:

```
Thread Tw:                          Thread Tn:
                                    lock(l) //position Pnl
                                    unlock(l)
lock(l) //position Pwl
lock(c)
wait(c) //position Pwc              lock(c) //position Pnc
unlock(c)                           signal/broadcast(c)
unlock(l)                           unlock(c)
```

Right before thread t_w executes *lock(l)*, Dimmunix suspends t_w until thread t_n is about to execute *lock(c)*. Then, Dimmunix suspends t_n until t_w is about to wait on c. For the case

where the critical section of l wraps the critical section of c in thread t_w, Dimmunix uses the same avoidance technique. The other 6 patterns of blocked notifications are not avoidable (§6.2.3); Dimmunix can only detect and report them.

For avoiding blocked notifications, Dimmunix needs to compute for each wait call at a program position p_w the positions $LN[p_w]$ of the matching signal (broadcast) calls in the program, i.e., the notifications performed on the same condition variable as the one used at p_w. More precisely, the set $LN[p_w]$ contains the positions of the lock statements corresponding to these notifications. For instance, in the above code example, $LN[p_w] = \{p_n^c\}$, where p_n^c is the position of the *lock(c)* statement in thread t_n's code. The set $LN[p_w]$ is computed dynamically, to avoid code inspection involving an expensive may-alias static analysis.

We describe now in detail the steps of the avoidance mechanism. Assume thread t_w is about to execute a *lock(l)* statement on mutex l, at program position p_w^l. Assume p_w^l is part of a blocked notification signature (p_w^l, p_w^c, p_n^l). Dimmunix performs the following steps to avoid reoccurrences of the same blocked notification:

1. Suspends t_w until a lock statement at a position from $LN[p_w^c]$ is about to execute.

2. A thread t_n is about to execute a lock statement at position p_n^c; if $p_n^c \in LN[p_w^c]$ or p_n^c was not encountered yet at runtime, Dimmunix resumes t_w and suspends t_n until t_w executes a wait call at position p_w^c or releases l; afterwards, thread t_n is resumed. If the condition variable used at position p_n^c is the same as the one used at p_w^c and $p_n^c \notin LN[p_w^c]$, Dimmunix adds p_n^c to $LN[p_w^c]$.

The above avoidance technique usually does not suspend a waiter thread indefinitely. If the set $LN[p_w]$ corresponding to a wait position p_w is incomplete, Dimmunix conservatively resumes the waiter thread t_w, each time a new critical section corresponding to a notification is encountered at runtime. Eventually, $LN[p_w]$ will be complete. However, the waiter thread is suspended indefinitely if the condition variables used at positions from $LN[p_w]$ are no longer used at p_w. We believe that these remappings of notifications to other wait calls are not likely to be encountered in real programs. If there exist such programs, this limitation can be addressed by an accurate may-alias static analysis.

If all the decisions to execute a wait (or signal/broadcast) call on a condition variable c are taken within c's critical section, Dimmunix preserves the partial order between the *wait(c)* calls and the *signal/broadcast(c)* calls; the semantics of condition variables specify that a notification has no effect on a wait call that executes later (§3.1.5). In other words, Dimmunix does not cause inversions of wait and signal/broadcast calls. A *wait(c)* call is delayed only until a notifier thread is about to acquire c and send a notification; then, Dimmunix suspends the notifier thread until the *wait(c)* call is about to execute. Therefore, the partial order between the waits and notifications is preserved.

6.2.3 Limitations

Among the 8 patterns of blocked notifications, 6 are unavoidable. One unavoidable pattern is:

```
Thread Tw:                    Thread Tn:
lock(l)
lock(c)
wait(c)                       lock(c)
unlock(c)                     lock(l)
unlock(l)                     unlock(l)
                              signal/broadcast(c)
                              unlock(c)
```

Dimmunix's avoidance mechanism described in §6.2.2 does not work on this pattern. Dimmunix would delay thread t_w right before it executes *lock(l)*, until thread t_n is about to execute *lock(c)*. That does not prevent the statement *lock(l)* from t_n's code from blocking the notification, because *lock(l)* is not yet reached by t_n. Dimmunix would have to suspend t_w until after t_n executes *lock(l)*. At that point, thread t_n already holds c. Therefore, thread t_w can reach the wait call only if thread t_n executes the signal/broadcast call first. Hence, the notification would always have to happen before the wait call. In that case, thread t_w always misses the notification; t_w may even hang indefinitely, if there is no further notification. This breaks the semantics of the program.

Deterministically missing a notification breaks the partial order implied by the semantics of condition variables. These semantics state that the wait call must precede the signal/broadcast call, for the notification to be effective (§3.1.5). Hence, a transformation which renders a notification useless, after previously being effective, changes the semantics of the application. The other 5 patterns are unavoidable for the same reasons, i.e., any avoidance mechanism would either lead to an indefinite hang, or break the partial oder implied by the semantics of condition variables.

We say that a blocked notification reported by Dimmunix is a false positive (FP) if and only if it does not represent a deadlock. More precisely, the presence of a thread t_n in the set $PN[c]$ does not guarantee that t_n always "intends" to execute *signal(c)* or *broadcast(c)* after acquiring mutex l at position p_n^l. We illustrate a FP in the code below:

```
Thread Tw:                    Thread Tn:
lock(l)
lock(c)
wait(c)                       if cond1 {
unlock(c)                       lock(l) //Pnl
unlock(l)                       unlock(l)
                              }
                              lock(c)
                              if cond2
                                signal(c)
                              unlock(c)
```

The two threads are deadlocked when $cond_1$ and $cond_2$ both hold. If only $cond_1$ holds, there is no deadlock, since thread t_n does not "intend" to execute *signal(c)*.

Avoiding FPs is not possible for the general case, because knowing exactly what branches a thread is going to take is undecidable. For instance, in the above code, it is not possible (in the general case) to know at program position p_n^l whether thread t_n is going to execute *signal(c)* if it would be able to proceed with the acquisition of mutex l; predicting whether $cond_2$ will hold is undecidable.

Dimmunix can make a waiter thread yield indefinitely in the following wait-notify pattern: (1) some shared state is updated by the waiter right before executing the wait call,

(2) the notifier thread decides to resume the waiter based on the shared state, and (3) the notification decision is taken outside c's critical section. Consider the code in Figure 6.3, where a notifier thread wakes up a waiter thread that registered to a wait queue q. Dimmunix makes thread t_w wait at position p_w^l until thread t_n reaches the *lock(c)* statement at position p_n^c. Assume that (1) queue q is initially empty, (2) t_n is the only notifier thread, and (3) t_w is the only thread that adds elements to q. Since t_w did not execute statement *q.add(c)* yet, *q.pop()* returns *null*. This means thread t_n is not going to send any notification, i.e., will not reach the *lock(c)* statement at p_n^c. Therefore, thread t_w waits indefinitely. Without Dimmunix, the program hangs only when the condition *cond* holds; with Dimmunix, the program always hangs. Dimmunix would need to statically analyze the program to identify such wait-notify patterns, and disable the avoidance of blocked notifications involving these patterns.

```
Thread Tw:                              Thread Tn:
                                        c = q.pop()
                                        if (cond) {
                                          lock(l) //position Pnl
                                          unlock(l)
                                        }
                                        if (c != null) {
lock(l) //position Pwl
lock(c)
q.add(c)                                  lock(c) //position Pnc
wait(c) //position Pwc                    signal/broadcast(c)
unlock(c)                                 unlock(c)
unlock(l)                               }
```

Figure 6.3: Code illustrating a wait-notify pattern which makes the avoidance mechanism introduce a hang.

Dimmunix does not guarantee that a blocked notification is always avoided, i.e., the avoidance mechanism may have FNs. Consider the code below, involving 1 waiter thread and 2 notifier threads:

```
Thread Tw:                          Threads Tn, Tn':
                                    lock(l) //position Pnl
                                    unlock(l)

lock(l) //position Pwl
lock(c)
wait(c) //position Pwc              lock(c') //position Pnc
unlock(c)                           signal/broadcast(c')
unlock(l)                           unlock(c')
```

To avoid the blocked notification, Dimmunix makes thread t_w wait until thread t_n (or t'_n) executes *lock(c')* at position p_n^c. Assume t_n executes *lock(c')* first. If $c' \neq c$, the *signal/broadcast(c')* call will not resume the *wait(c)* call. If later thread t'_n needs to acquire l and c' becomes equal to c, the program runs into the same blocked notification that Dimmunix is attempting to avoid.

Dimmunix does not handle blocked notifications involving more than two threads, or other types of resources (e.g., read-write locks, semaphores), and does not deal with yield cycles caused by avoiding blocked notifications. Since mutex deadlocks are likely the most frequent among all deadlock types (§3.3.1), we chose to offer a complete immunity system only for mutex deadlocks.

6.3 False Positives and False Negatives

A false positive (FP) in the context of deadlock avoidance means avoiding a false deadlock situation, i.e., the execution is conservatively serialized, even though a deadlock could not have occurred. A false negative (FN) means missing to avoid a potential deadlock situation.

In the context of Dimmunix's deadlock avoidance, a FP has the same meaning, i.e., serializing concurrent code more than necessary; a FN means that the deadlock signature does not capture all the possible manifestations of the deadlock. In absolute terms, Dimmunix has only FNs and no FPs, as long as no deadlock is encountered. If all the deadlock bugs in a program have been experienced, and their signatures capture all the possible manifestations of the deadlocks, Dimmunix has no FNs, but may have FPs; in other words, the application is deadlock-free, but may be over-serialized.

If the call stack matching depth is 1, a deadlock signature captures all the possible manifestations of the deadlock bug it represents. In this case, Dimmunix has no FNs, w.r.t., avoiding the deadlock bug. However, there may be FPs, i.e., avoidances of execution flows that are not deadlock prone. Remember that the matching depth is relevant only for the outer call stacks of a signature; only the outer call stacks are matched in the deadlock avoidance.

The higher the matching depth, the more likely is that Dimmunix will have FNs, i.e., miss manifestations of the deadlock bug captured by the signature; however, a high matching depth implies fewer FPs, i.e., better accuracy for the avoidance mechanism. We illustrate this principle in Figure 6.4, which illustrates 6 different call flow suffixes (i.e., call stacks) of depth 3 ending in the same outer lock statement. The two outer call stacks in bold are the ones that are deadlock prone; the one with the dotted arrow is not discovered yet by Dimmunix, i.e., it did not manifest yet. Therefore, there are 4 outer call stacks which would lead to FPs, if matched. If the matching depth is 1, all the 4 FP call stacks are matched, but no true positive (TP) call stack is missed, i.e., there are no FNs. If the matching depth is 2, both TP call stacks are matched, and no FP call stack is matched. If the matching depth is 3, only 1 TP call stack is matched, i.e., there is 1 FN, and no FP call stack is matched.

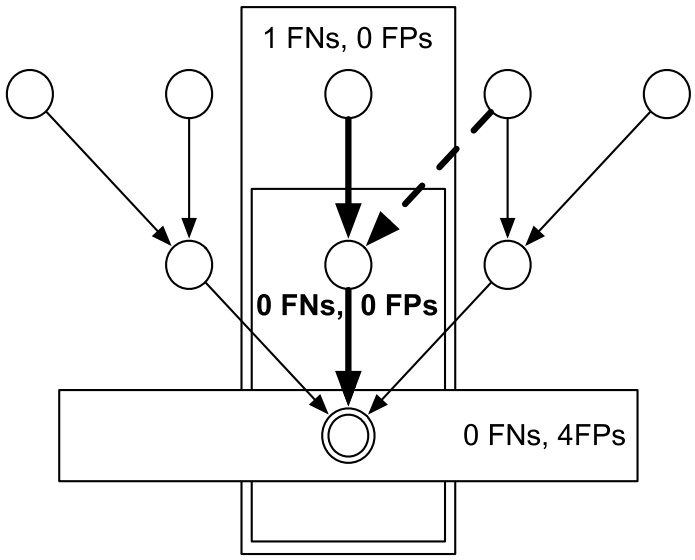

Figure 6.4: The matching depth influences the number of FPs and FNs.

Having high accuracy (i.e., few FPs) for the deadlock avoidance is important for reducing the loss of parallelism (i.e., number of yields). The higher the accuracy of the signature matching, the less frequent the program execution matches the outer call stacks of a signature; therefore the probability to instantiate the signature is lower, and the yields are fewer. This is important for mutex deadlocks, because mutexes are heavily used, compared to other synchronization primitives, as shown in Table 3.1. Moreover, the throughput of synchronization operations performed on mutexes (§10.1) is much higher than the throughput of non-mutex synchronization primitives (§10.2) in the Java applications that we studied. Therefore, for non-mutex deadlocks it is not essential to have an accurate avoidance. For these deadlocks we use program positions (instead of call stacks) in the signatures; program positions are essentially call stacks of depth 1.

Using outer call stacks of depth 1 can be harmful, if a program uses mostly custom synchronization implemented with explicit lock/unlock operations; Dimmunix would serialize most of the critical sections, as soon as the first deadlock occurs. Consider a Java program that uses the following wrapper of the *RentrantLock* class:

```
public class MyLock {
  private ReentrantLock l;
  public void lock() {
    l.lock();
  }
  public void unlock() {
    l.unlock();
  }
}
```

If any deadlock happens in the program, the outer call stacks will indicate the position p of the *l.lock()* statement within the *MyLock* class. Dimmunix would serialize all the critical sections that use objects of type *MyLock* for synchronization, because the synchronizations are all performed at position p.

For synchronized blocks, it is safe to use outer call stacks of depth 1, because the synchronized blocks that appear in wrappers cannot be outer lock statements in a deadlock signature. Synchronized methods are essentially synchronized blocks, therefore we only

discuss about synchronized blocks. Synchronized blocks are intra-procedural, i.e., a *monitorexit(l)* statement has to execute in the same method as the corresponding *monitorenter(l)* statement, like in the wrapper below:

```
public class MyLock {
  private Object l;
  public void lock() {
    synchronized(l) { //monitorenter(l)
      //update state
    } //monitorexit(l)
  }
  public void unlock() {
    synchronized(l) { //monitorenter(l)
      //update state
    } //monitorexit(l)
  }
}
```

Typically, the synchronized blocks from synchronization wrappers are not nested; therefore, they cannot be the outer positions of a deadlock signature, because the program cannot deadlock inside them. If somehow these synchronized blocks are nested, they are normally deadlock-free, otherwise a program that heavily uses the wrapper is likely to deadlock often; if such a program exists, it "deserves" to be entirely serialized.

In fact, Dimmunix uses a separate matching depth for each outer call stack of a signature. So far, we used a global matching depth, to simplify the explanations.

Dimmunix attempts to achieve an optimal matching depth for each outer call stack of a signature. An optimal matching depth is the maximum depth for which there are no FNs. For instance, for the call stacks depicted in Figure 6.4, the optimal matching depth is 2, because there are 0 FNs and 0 FPs for that depth. Dimmunix has a mechanism for learning the optimal matching depths (§8). Communix uses a collaborative learning of the optimal matching depths, by aggregating information about signatures discovered by different users (§7).

6.4 Impact on Functionality

In this section, we explain how Dimmunix may disable program functionality. Assume the pseudocode below:

```
//two cars should move simultaneously on the screen
Thread t1:                              Thread t2:
lock(l1) //position Pout1               lock(l2) //position Pout2
move car 1                              move car 2
if some condition holds {               if some condition holds {
  lock(l2) //position Pin1                lock(l1) //position Pin2
  unlock(l2)                              unlock(l1)
}                                       }
unlock(l1)
```

Assume the program deadlocks; the signature of the deadlock is $\{([p_1^{out}],[p_1^{in}]),([p_2^{out}],[p_2^{in}])\}$. Dimmunix would not allow the two cars to move simultaneously on the screen, because it needs to serialize the executions of the two threads at positions p_1^{out} and p_2^{out}.

To cope with loss of functionality, Dimmunix (optionally) offers the following simple solution: it automatically disables a signature that is instantiated often (e.g., several times in a second), and most of the instantiations (e.g., 99%) are FPs, for several minutes. Moreover, as we mentioned in §6.3, having accurate signature matching helps to reduce the number of yields, which will also reduce the loss of functionality.

Dimmunix may cause starvation if it interferes with wait-notify synchronization. Consider the code below, involving a waiter thread t_w and a notifier thread t_n:

```
Thread Tw:                              Thread Tn:
lock(l) //position Pout1                lock(c) //position Pout2
lock(c) //position Pin1                 if (cond) {
wait(c) //position Pw                     lock(l) //position Pin2
unlock(c)                                 unlock(l)
unlock(l)                               }
                                        signal/broadcast(c)
                                        unlock(c)
```

Two kinds of deadlocks can happen in the above code: a mutex deadlock and a blocked notification. The mutex deadlock involves the lock statements at positions p_1^{out}, p_2^{out}, p_1^{in}, and p_2^{in}. The blocked notification involves the programs positions p_1^{out}, p_w, and p_2^{in}. Assume that the mutex deadlock happened first, thread t_n is the only notifier thread, and condition *cond* does not hold. Whenever thread t_n executes *lock(c)* after thread t_w executes *lock(l)*, Dimmunix suspends t_n until t_w releases lock l, in order to avoid the deadlock. Since t_w waits for t_n to execute *signal/broadcast(c)* and holds l, we have a starvation situation. If *wait(c)* does not execute in a loop, Dimmunix could safely resume t_n right after the *wait(c)* call releases c; however, it is often the case that the wait call executes in a loop. Since *cond* does not hold, the program would not run into blocked notification and t_n could resume t_w if allowed to proceed. Therefore, we can consider that this starvation situation is a loss of functionality. To prevent such unfortunate interferences with wait-notify synchronization, Dimmunix ignores the signatures of mutex (or initialization) deadlocks that involve critical sections containing wait, signal, or broadcast calls.

Although it is possible for Dimmunix to disable functionality, we have not yet encountered loss of functionality due to Dimmunix in the applications we studied (§10). Even with synthetic (i.e., fake) deadlock signatures, there was no loss of functionality.

6.5 Platform-wide vs. Application-level Immunity

We first explain the notions of platform-wide and application-level deadlock immunity, from the user's perspective. Platform-wide immunity means that all applications are immunized against deadlocks by default, without having to be launched in a special way. Application-level immunity means that the applications are not immunized by default against deadlocks, and have to be executed in a special way to run with Dimmunix.

We discuss three aspects concerning platform-wide deadlock immunity: First, we show that it cannot be implemented in the kernel space; it has to be implemented in the user space, i.e., in the synchronization library. Second, we compare application-level to platform-wide deadlock immunity. Third, we explain what needs to be done to have an efficient platform-wide Dimmunix.

Platform-wide deadlock immunity has to be implemented in the user space, i.e., in the

synchronization library, as we illustrate in Figure 6.5. All the modern platforms/libraries providing synchronization routines (e.g., JVM, POSIX threads) first attempt to acquire a lock in the user space. Dimmunix must intercept all the lock acquisitions. Since a lock may be acquired in the user space, the interception must be done in the user space.

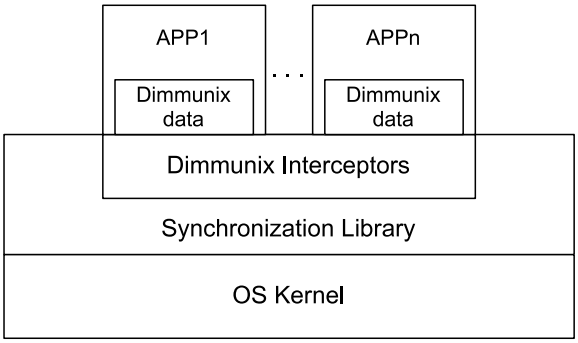

Figure 6.5: Platform-wide Dimmunix.

Since platform-wide Dimmunix has to run in user-space, there is a different instance of Dimmunix running within each process. Dimmunix's avoidance and detection mechanisms are application-local, i.e., deadlocks are detected and avoided locally in each application, in isolation from the other applications.

We compare now application-level to platform-wide deadlock immunity systems. Application-level Dimmunix can be instrumentation-based or interception-based, i.e., it can instrument the synchronization statements in the program binary (e.g., by using AspectJ bytecode instrumentation framework [asp]), or it can intercept the synchronization operations through a preloadable library (e.g., by using LD_PPRELOAD). An instrumentation-based implementation has the possibility to instrument only the synchronization statements previously involved in deadlocks (e.g., Java Dimmunix), in order to minimize the performance overhead and the intrusiveness. An interception-based implementation (e.g., POSIX Threads Dimmunix) does not have this possibility, because intercepting the synchronization operations involves overriding the synchronization routines. There is also library-level deadlock immunity, i.e., protection against deadlocks for all the applications using a particular synchronization library. For instance, changing the POSIX Threads library to provide deadlock immunity would be a library-level implementation of Dimmunix; we provide such an implementation for the FreeBSD POSIX Threads library. However, a preloadable

Dimmunix library which can be attached to a C/C++ application using LD_PRELOAD is an application-level implementation, because the POSIX Threads library would not provide deadlock immunity by default; the users have to use LD_PRELOAD to run a C/C++ application with Dimmunix.

For platform-wide deadlock immunity, an interception-based implementation is the most natural choice; Android Dimmunix is interception-based. The only drawback of an interception-based implementation is that it cannot selectively instrument synchronization statements, while an instrumentation-based implementation can. However, an instrumentation-based implementation is more complicated, because it would have to dynamically modify the binary of every application, before executing it. Moreover, the binary instrumentation/analysis frameworks are not mature enough to allow a robust implementation. In future however, such an implementation may be feasible.

An efficient platform-wide Dimmunix needs small CPU and memory consumptions, because all the applications run with Dimmunix. It is important to satisfy these requirements, especially for mobile platforms, which are designed to optimize the CPU and memory consumptions. To achieve computational efficiency, the code on the critical path must be optimized, i.e., the look-up of RAG nodes and the call stack retrieval. For memory efficiency, we need to dynamically allocate and reuse memory. In §9.3, we give more details about how we achieved an efficient implementation of Android Dimmunix, our platform-wide deadlock immunity system for Android OS.

Chapter 7

Collaborative Immunity

Failure immunization techniques protect the programs against a specific bug or vulnerability exploit by learning from its past manifestations. We use the term "failure" to denote a manifestation of the bug. An immunization system detects the failure, extracts its fingerprint, and uses it to avoid reoccurrences of the same bug. We call this fingerprint "bug signature". A bug signature is an approximation of the execution flow that led to the failure. An accurate signature captures only one or several manifestations of the bug. A general signature captures all or most of the manifestations of the bug.

Failure immunization systems avoid only manifestations of previously encountered bugs; a bug must manifest at least once for the application to be protected against it. Therefore, the false positives rate is low, because only manifestations of real bugs are avoided. A false positive is the situation where a failure is avoided with no reason, i.e., the failure could not have occurred, even without any avoidance. However, there are false negatives (i.e., bugs against which the application is unprotected) until all the bugs manifest.

One possible solution to address the aforementioned drawback is collaborative immunization via distribution of bug signatures. More specifically, once a user encounters a bug, the bug's signature is automatically generated and distributed to other users through the Internet. Therefore, each bug needs to be encountered once, by any user, then the other users get protected against the bug, without having to experience it.

We present Communix, a collaborative immunization framework that enables Java programs running on different machines to distribute deadlock signatures, in order to immunize each other against deadlock bugs. The signatures of a deadlock can protect against the deadlock any user connected to the Internet and running the same application, even if he/she did not experience the deadlock yet. Besides signature distribution, Communix provides signature validation and generalization. Signature validation ensures that the incoming signatures match the target applications, and protects the users against malicious signatures. Signature generalization keeps the repository of deadlock signatures compact, by merging multiple deadlock signatures into one signature.

To distribute deadlock signatures, Communix uses an immunity server; client machines upload signatures discovered by Dimmunix to the server, and periodically retrieve the new signatures from the server. Each time a Java application starts on a client machine, Communix selects from these signatures the ones that are valid for that application and, if possible, it generalizes existing deadlock signatures.

Communix is efficient and scalable. In §10.5, we show that the server can process efficiently 30,000 simultaneous requests, at a rate of 63,000 requests per second. The agent can analyze 1,000 new deadlock signatures in 2–3 seconds (§10.5).

We present two scenarios that illustrate the benefit of frameworks like Communix. In the first scenario, the user opens a web page, and the browser deadlocks while rendering the content of the page, due to a Java applet. The user shuts down the browser, then restarts it and opens the same page. If the browser is equipped with Dimmunix, it will successfully open the page; if not, it might deadlock again. However, it may be undesirable to have the browser deadlocking in the first place. Even the first occurrence of the deadlock may have severe consequences: the browser might be in the middle of some important operation, like purchasing an expensive product, or booking a flight. Therefore, a framework like Communix that prevents other users from encountering the deadlock in the first place is beneficial. In the second scenario, a deadlock-prone version of a plugin is released for the Eclipse IDE, which makes Eclipse hang. If the plugin has multiple deadlock bugs, each user has to encounter all these deadlocks for Dimmunix to be able to avoid them. Sharing the signatures of the deadlocks with users who just installed the plugin is useful; these users will not experience any deadlocks while using the plugin if all deadlocks have already been

encountered by some users.

The rest of this chapter is organized as follows: We provide an overview of Communix in §7.1. We describe the design of Communix in §7.2.

7.1 Concept and Design

In collaborative deadlock immunity, different machines connected to the Internet work together to achieve immunity against deadlocks by sharing their deadlock signatures.

An important benefit of sharing the deadlock signatures is that any application can use the collective knowledge of the other nodes to generalize deadlock signatures from its history. The generalization consists of merging different signatures of the same deadlock bug. The role of signature generalization is to keep few signatures per deadlock bug, in order to have a small size of the deadlock history for each application. If all possible manifestations of a deadlock bug D were experienced by some nodes in the Internet, the current signatures of D are the most accurate signatures that enable Dimmunix to avoid all the manifestations of D.

Upon receiving a signature from other nodes, Communix checks whether the signature can be used by the running application. This first validation step assumes the node that sent the signature is honest.

Attackers may send fake deadlock signatures that do not represent real deadlock bugs; these signatures may cause denial of service (DoS) in applications instrumented with Dimmunix. Such signatures may exploit Dimmunix to increase the runtime overhead of signature matching and reduce the parallelism due to suspending threads. The validation process should prevent such signatures from harming the performance or the functionality of the applications. Therefore, additional checks are performed (§7.2.3).

7.2 Design

In this section, we describe the architecture of the Communix framework (§7.2.1), explain the signature distribution (§7.2.2), and describe in detail the signature validation (§7.2.3) and signature generalization (§7.2.4).

7.2.1 Communix Framework

Communix has five components, as we illustrate in Figure 7.1: Dimmunix (i.e., the deadlock immunization component of Communix), Communix plugin, Communix server, Communix client, and Communix agent. Dimmunix is in charge of (1) detecting deadlocks, (2) saving their corresponding signatures into the running application's deadlock history, and (3) preventing the application from encountering the same deadlocks again.

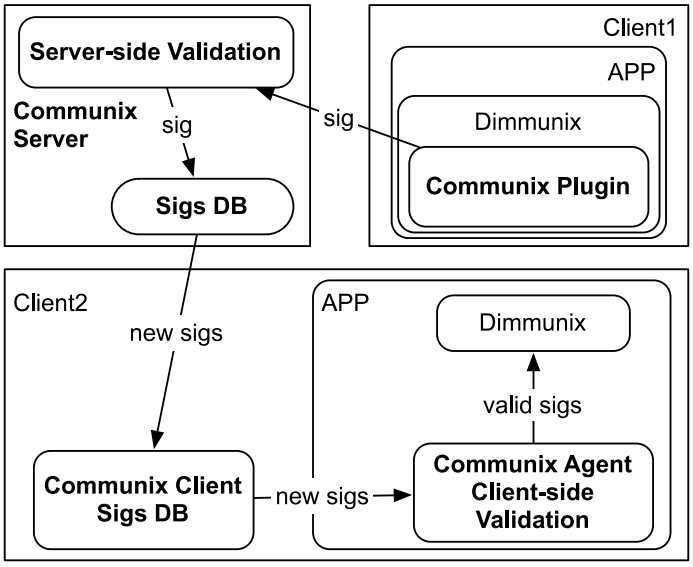

Figure 7.1: Communix architecture.

Communix uses a centralized signature distribution framework. The Communix plugin, implemented on top of Dimmunix, sends the deadlock signatures to the Communix server, right after Dimmunix produces the signatures. In order to obtain new deadlock signatures from the server, a machine must have the Communix client installed; the client periodically downloads the new deadlock signatures from the server into a local repository. Any Java application running with Dimmunix can use these signatures to improve its protection against deadlocks.

A centralized signature distribution improves the protection against deadlocks for all the machines connected to the Internet that are equipped with Communix. Each newly discovered signature *S* becomes available to any machine connected to the Internet; as soon as other nodes download *S* from the Communix server and validate it, they are protected

against deadlock manifestations matching S, without having to encounter S.

The client-side signature validation and signature generalization are performed by the Communix agent. The agent runs together with Dimmunix, in a Java application's address space. When the application starts, the agent selects from the local repository the new signatures that are valid, i.e., that can be used by the application. If a new signature S is found valid, the agent attempts to merge S with an existing signature from the running application's deadlock history. If S cannot be merged with any existing signature, then it represents a new deadlock bug; the agent adds S to the history, in order to prevent future occurrences of deadlocks matching S.

To validate a new deadlock signature, the Communix agent checks whether the signature matches the running application. In addition, Communix protects the users against DoS attacks based on distributing malicious signatures. To generalize deadlock signatures, Communix merges signatures representing the same deadlock bug into one signature.

Communix is application agnostic; no application specific information (like application name, or version) is required in addition to the deadlock signatures. This makes Communix a practical approach that works for any Java application.

7.2.2 Signature Distribution

Communix allows different users running the same application (or different applications sharing some deadlock-prone library) to share signatures. The more users run some deadlock-prone code, the more likely it is that all possible manifestations of the deadlock bug are experienced in a short period of time, and all users get fully immunized against the deadlock.

Once Dimmunix detects a deadlock, the Communix plugin sends the corresponding signature to the Communix server. The Communix server collects in a database all the deadlock signatures discovered by Java applications running with Dimmunix on arbitrary machines connected to the Internet. To decide whether to add an incoming signature to the database, the server performs a simple signature validation, described in §7.2.3.

The Communix client, running on an arbitrary machine in the Internet, periodically downloads the new deadlock signatures from the server into a local repository. The local repository is updated once a day; a high frequency (e.g., once a minute) would overload

the Communix server. The updates are incremental, i.e., the client requests from the server only the signatures that are not present in the local repository. When a Java application A starts, the Communix agent inspects the new signatures from the local repository: the agent checks the validity of each new signature S (§7.2.3); if S is valid, the agent adds S to A's deadlock history. The inspection of the local repository is incremental, i.e., every signature is analyzed only once.

The Communix client runs as a background process, decoupled from the agent. Without this decoupling, the Communix agent would have to connect to the server and retrieve new deadlock signatures every time a Java application starts. This would introduce an unnecessary overhead.

Note that Communix does not require users to provide any application specific information (like name or version) with the signatures they share. Communix only needs hash values of class bytecodes, in order to distinguish different versions of the same class or different classes having the same name. The hash values are automatically computed by the Communix plugin, when Dimmunix produces the signatures. This makes Communix application agnostic.

7.2.3 Signature Validation

Before sending a signature to the server, the Communix plugin attaches to each call stack frame of the signature the hash of the class bytecode containing that frame.

Each time a Java application running Dimmunix starts, the Communix agent selects from the local repository the signatures that match the running application. The agent checks whether the hashes of an incoming signature match the bytecode hashes of the running application. If the hashes do not match for all the top frames, the signature is rejected; otherwise, the agent keeps from the signature the longest call stack suffixes with hashes matching the application.

If all nodes were honest, the above check would have been sufficient; unfortunately, there are attackers that may try to exploit Dimmunix by sending fake signatures, therefore additional checks are needed.

Preventing (Containing) DoS Attacks

An attacker may attempt to perform a performance DoS attack based on signature flooding, to put pressure on Dimmunix's signature matching mechanism. Such an attack consists of sending many fake signatures that manage to pass the validation and get accepted into the deadlock history of an application. This would put pressure on Dimmunix, because all these signatures have to be matched at runtime.

Communix manages to prevent such attacks by performing three additional checks. The first two checks are performed by the server, and the third one is performed by the Communix agent, on the client side. If any of these checks fails, the signature is rejected.

First, the server requires each incoming signature to be accompanied by an encrypted id of the sender. The encrypted id is provided once by the Communix server. The server uses the sender ids to bind each incoming signature to the user who sent the signature. Since an attacker can fake many IP addresses, they cannot be used to identify the senders; it must be hard for an attacker to obtain multiple ids.

Second, the server makes sure that every two distinct signatures sent by the same user (i.e., having the same sender id) have no common top frames. This restriction should not affect honest users, because it is not likely that a user would experience such "adjacent" deadlocks. However, if he/she does experience such situations, the signatures wrongly rejected due to this restriction can be provided by other users.

Third, the agent checks whether the outer call stacks of a new signature end in nested synchronized blocks/methods. Checking whether a synchronized block/method is nested is straightforward, due to the disciplined way the Java compiler nests these constructs. We describe the algorithm in §7.2.3. Communix does not handle explicit lock/unlock operations (e.g., calls to *ReentrantLock.lock/unlock()*). However, this is a minor deficiency, since Java programs use mostly synchronized blocks/methods (§10.5).

Thanks to the above three checks, the possibility to flood Dimmunix with fake signatures is limited. If there are N nested synchronized blocks/methods in a Java application A, an attacker cannot "provide" more than N signatures that get accepted into A's deadlock history. Typically, in a Java application there are a few hundred nested synchronized

blocks/methods (§10.5). Therefore, an attacker cannot force more than a few hundred signatures into the deadlock history of an application.

Another type of performance DoS attack that an attacker may attempt is to send fake signatures that slow down an application. These attacks force Dimmunix to avoid instantiations of fake signatures or signatures that are too general. The more general a signature is, the more often Dimmunix has to avoid instantiations of the signature. This means Dimmunix suspends threads more often then needed, which may considerably slow down the application. The attacker may exploit the generalization mechanism to retain only the top frames of the outer call stacks, or send directly signatures with outer call stacks of depth 1.

The Communix agent prevents the attackers from sending signatures with outer call stacks of depth < 5. For the applications we studied, the outer call stacks have large depths (usually > 10); therefore, we believe that this restriction does not affect the honest users. Signatures with outer call stacks of depth 5 incur an acceptable performance overhead; for depth 1, the overhead is considerable (§10.5,§10.1.2). Therefore, the outer call stacks must have the depth ≥ 5. To prevent an attacker from exploiting the signature generalization mechanism to obtain outer call stacks of depth 1, the agent does not merge signatures below depth 5, for the outer call stacks. Alternatively, one could compute the minimal depth d that outer call stacks corresponding to a nested synchronized block/method can have; the threshold would be $min(d,5)$, rather than 5, in this case.

The third check ensures that the worst damage an attacker can do is to force into the deadlock history of an application signatures with outer call stacks of depth 5, that cover all the nested synchronized blocks/methods. We show in §10.5 and §10.1.2 that such a scenario causes only 7–29% performance overhead in the Java applications we studied.

An attacker may also attempt a functionality DoS attack that disables features of an application. If a certain feature needs some code to execute concurrently, that feature would no longer be available, if Dimmunix makes the code execute sequentially. This undesired effect can be caused also by real deadlock signatures; some concurrent code may be deadlock-prone, and execute most of the time without deadlocking. To prevent such situations, we use Dimmunix' false positive detection mechanism. If after 100 instantiations of a signature S there was no true positive, and there was at least one interval of 1 second having more than 10 instantiations of S, Dimmunix decides to warn the user about signature S;

the user can decide to keep S, if he/she notices no change in the behavior of the application.

Attackers may attempt to put pressure on the Communix server by sending bursts of fake signatures to the server. The server processes only up to 10 signatures per day from one user; beyond this threshold, the signatures from that user are ignored by the server. This restriction usually does not affect honest users, since it is unlikely that a user would experience so many different deadlocks (or different manifestations of a deadlock) in 1 day. However, the wrongly rejected signatures can be provided by other users.

In the remainder of this section, we describe in detail the server-side and client-side signature validation.

Server-side Signature Validation

The Communix server requires each user to accompany the signatures he/she sends with an encrypted user id that the server provides. The server provides a unique id to each user; the id is encrypted, in order to prevent users from manufacturing their own ids. To be able to share its signatures, each user has to previously obtain the encrypted id from the Communix server. The server uses AES encryption, with a predefined 128-bit key, to produce the encrypted user ids. We did not implement the service for issuing the encrypted user ids; such a service exceeds the scope of this work. The problem of preventing attackers from impersonating multiple users has been extensively studied.

Upon receiving a signature S accompanied by an encrypted id I, the Communix server decrypts I to obtain the id of the user that sent the signature. After retrieving the sender id, the server checks whether the same user already sent a signature S' which is adjacent to S, i.e., S and S' have some (but not all) top frames in common. If the user already sent a signature adjacent to S, the server refuses to add S to its database.

Rejecting adjacent signatures from the same user considerably reduces the capability of an attacker to provide fake signatures. Assume there are N synchronized blocks/methods in an application, and there are N_d possible call stack suffixes of depth d, for each synchronized block/method. Without this restriction, the attacker can manufacture $(N \cdot N_d)^4$ signatures of two-thread deadlocks that pass the validation, for each depth $d \geq 5$; this gives a total of $N^4 \cdot \sum_{d=5}^{\infty} N_d^4$ possible signatures. With this restriction, the attacker can provide only N signatures.

Client-side Signature Validation

For each new signature S, the Communix agent checks whether S matches the running Java application, then it checks whether the outer call stacks of S end in nested synchronized blocks/methods.

For each call stack of signature S, the Communix agent checks whether the hashes it carries match the running application A. Each call stack of signature S is encoded as a sequence of frames $[c_1.m_1 : l_1 : h_1, ..., c_n.m_n : l_n : h_n]$, where c_i are class names, m_i are method names, l_i are line numbers, and h_i is the hash of class c_i's bytecode. The hashes are attached by the Communix plugin when Dimmunix produces signature S. The hash value h_k matches application A if and only if class c_k's bytecode from application A has the hash h_k. The hash check starts from the top frame, i.e., frame n; if h_n does not match A, signature S is rejected. If h_k ($1 \leq k < n$) is the first hash value that does not match A, the frames $1, ..., k$ are removed from the call stack; if all hashes match, the call stack remains unchanged. For efficiency, the Communix agent computes the hash of a class first time the class is loaded, then it reuses the computed hash value.

The hash checking covers also the inner call stacks, even though they are not used by Dimmunix for deadlock avoidance. The signature may correspond to an earlier version of the application, where the code between the outer and inner lock statements was deadlock-prone. That code might have been fixed in a newer version of the application. If the Communix agent would check only the hashes of the outer call stacks, these code changes would be missed, and the false positive signature would pass the validation.

We describe now the algorithm for checking whether a synchronized block B is nested; we also provide a detailed description in Algorithm 5. Given the control flow graph (CFG) of an application binary, and the *monitorenter* statement corresponding to a synchronized block, the Communix agent inspects the CFG, starting from the successor of s. As soon as a *monitorenter* (respectively *monitorexit*) statement is encountered, the algorithm returns that B is nested (respectively non-nested); see lines 1–2. If a method call statement s_{call} is met, the algorithm returns that B is nested, if any method that may be called (directly on indirectly) by s_{call} is either synchronized or contains a synchronized block (lines 3–8). The exploration continues recursively with the unexplored successors of s

Input: Current statement s, initially the successor of the *monitorenter* statement corresponding to a synchronized block.
Data: Control flow graph.
Output: True/False

```
1  if s is monitorenter or monitorexit then
2      return (s is monitorenter)? True: False
3  if s is a method call then
4      foreach method m that may be called by s do
5          if m is synchronized then return True
6          Let s_0 be the first statement of m
7          if isNested(s_0) then return True
8      end
9  foreach unexplored successor s' of s do
10     if isNested(s') then return True
11 end
12 return False
```

Algorithm 5: *isNested(s)*: recursively computes whether a synchronized block is nested.

Since a synchronized method is semantically equivalent to a *synchronized(this)* block that wraps the method body, the algorithm for checking whether synchronized methods are nested is similar. In fact, the AspectJ instrumentation framework [asp] that Dimmunix uses transforms the synchronized methods into synchronized blocks.

For efficiency, the Communix agent precomputes the locations of all the nested synchronized blocks/methods, when the application runs for the first time. Checking if the outer call stacks of a signature end in nested synchronized blocks/methods consists of determining if the top frames belong to this precomputed set of locations. To inspect the application bytecode, the Communix agent uses the Soot bytecode analysis framework [Vallée-Rai et al., 1999].

Each time new classes are loaded, in addition to the ones loaded in the previous runs, the Communix agent repeats the nesting check for all the signatures from the local repository that passed the hash check and failed the nesting check. There is no need to recheck the nesting for the rest of the signatures, because adding new classes to the CFG can only uncover new nested synchronized blocks/methods.

7.2.4 Signature Generalization

The signature generalization consists of merging different signatures corresponding to the same deadlock bug, i.e., that end in the same inner and outer lock statements. The resulting signature consists of the longest common suffixes of the call stacks forming these signatures.

It is important to generalize signatures for the following reason. If the outer call stack suffixes are long, the signature may not be able to always avoid the deadlock. In other words, there may be false negatives, i.e., other signatures of the same deadlock ending in different outer call stack suffixes. If there are multiple manifestations of the deadlock having different outer call stack suffixes, it may take a long time until a single user experiences all these manifestations.

A trivial solution to avoid all the possible manifestations of a deadlock would be to match only the top frames of the signature's outer call stacks. However, there is an important drawback to this solution: having the outer call stacks matched too shallowly introduces false positives (§6.3) and therefore reduces the parallelism, which may have a negative impact on performance (§10.1.2, §10.1.3).

The generalization process is the following: When a Java application starts, the Communix agent checks if new signatures that passed the validation could be merged with existing signatures from the deadlock history of the running application. The signatures that cannot be merged are added to the history.

Two signatures S and S' can be merged if and only if they represent the same deadlock bug (i.e., the top frames of S have to be identical to the top frames of S'), and either (1) S and S' were produced on the local machine, or (2) S/S' is a remote signature and the resulting signature has the outer call stacks of depths ≥ 5.

Merging two signatures consists of finding the longest common call stack suffixes of the two signatures. Given two signatures $S = \{(CS_{1,out}, CS_{1,in}), ...\}$ and $S' = \{(CS'_{1,out}, CS'_{1,in}), ...\}$, their generalization is the signature $S^g = \{(CS^g_{1,out}, CS^g_{1,in}), ...\}$, where CS^g is the longest common suffix of call stacks CS and CS'.

Chapter 8

Optimizations

In this chapter, we present the optimizations we performed in Java Dimmunix, to achieve low performance overhead and high accuracy (i.e., few FPs). The optimizations related to high accuracy are implemented only for mutex deadlocks. We do not attempt to obtain high accuracy when avoiding non-mutex deadlocks, because they involve synchronization primitives which are not often executed (§10.1).

To achieve efficient deadlock immunity for lock-intensive applications, two goals have to be satisfied. First, the deadlock detection and avoidance mechanisms must have low overhead. Second, the deadlock avoidance mechanism has to be accurate, i.e., have few false positives (FPs).

Retrieving the call stack of a thread and matching it against the history plays a central role in Dimmunix's efficiency. If a thread's call stack matches an outer call stack of a signature from history (up to the matching depth), deadlock avoidance is performed and the thread may yield, otherwise the thread is allowed to proceed. Whenever a thread t acquires a lock l, t's current call stack is used to label the hold edge $l \longrightarrow t$ in the RAG; the deadlock signatures are formed of such labels. Therefore, to achieve low overhead, an efficient call stack retrieval routine is needed (e.g., the *backtrace()* C routine).

If the call stack retrieval routine is expensive (e.g., Java's *Thread.getStackTrace()* method), Dimmunix has to be invoked as rarely as possible, in order to substantially reduce the overhead. Therefore, intercepting all the lock operations is not suitable—a selective interception of lock operations is necessary. This is possible only through a selective program

instrumentation. To achieve a minimal amount of program instrumentation, the following steps have to be performed: First, the outer call stacks of a deadlock signature must be inferred from the inner call stacks, without keeping track of the call stack of each lock acquisition operation. Second, the avoidance code has to be instrumented only at program sites where deadlocks were previously detected. We describe the selective instrumentation of Java programs in §8.1.

If a signature is on the critical path (i.e., at least 1 outer call stack ends in a lock statement that is executed often), the call stack matching has to be optimized, since it is performed often by the avoidance code. If retrieving the call stacks is expensive, inlining the call stack matching reduces the overhead of the deadlock avoidance. We describe the inline call matching for Java programs in §8.2.

Achieving high accuracy for the deadlock avoidance is important for reducing the number of yields and the duration of each yield. Reducing the frequency and the duration of the yields has a positive impact on the performance of the applications (§10.1). To reduce the number of yields (§8.3), Java Dimmunix detects false positives (§8.3.1), then it calibrates the matching precision to reduce the number of FPs (§8.3.2), while preserving the effectiveness of the avoidance mechanism. The goal is to have zero FNs, with as few FPs as possible. To reduce the duration of a yield, Java Dimmunix exploits the branches that escape the deadlock (i.e., the deadlock situation is not reachable if such a branch is taken): when such a branch is taken, Dimmunix stops the avoidance process by canceling the active yields and preventing the future ones, until the lock whose acquisition triggered the avoidance is released (§8.4).

Even though non-mutex deadlocks involve synchronization primitives which are not often executed in programs, we want to achieve an efficient immunity framework for these operations. Since accuracy is not needed for these deadlocks, Dimmunix uses only program positions in the deadlock signatures. The position retrieval is equivalent to retrieving a call stack of depth 1, which is still expensive for Java programs. Therefore, we optimize the retrieval of program positions in Java programs, by inserting right before a synchronization instruction at a program position p code that stores p as the current execution point for the running thread (§8.2).

8.1 Selective Program Instrumentation

To minimize the amount of program instrumentation, the following steps have to be performed. First, the outer call stacks of a deadlock signature must be inferred from the inner call stacks, without intercepting the lock acquisition operations. Second, the avoidance code has to be instrumented only at program locations where deadlocks were previously detected.

Dimmunix infers the outer call stacks of a deadlock signature from the inner call stacks, by traversing backward the control flow graph (CFG) of the application bytecode. The outer call stacks can be easily inferred if the deadlock involves only synchronized blocks. For synchronized blocks, Dimmunix only needs to keep at runtime the lock stack (i.e., nested lock acquisitions) of each thread and the RAG. Since the JVM already does these operations, Dimmunix does not need to intercept the synchronized blocks.

Since yield cycles are avoidance-induced deadlocks, Dimmunix uses the same mechanism to infer the outer call stacks for signatures of yield cycles. Dimmunix only needs to augment the JVM's RAG with yield edges, in order to detect yield cycles; this operation is performed by the avoidance code, which is inserted only at program locations where Dimmunix previously detected deadlocks.

The outer call stacks are inferred using the lock stacks maintained by the JVM and the CFG of the program bytecode; we illustrate the inference procedure in detail in Algorithms 6 and 7. Having the lock stack LS_i, inner call stack CS_i, and the currently requested lock l_i of each deadlocked thread t_i, Dimmunix infers the outer call stacks as follows: First, for each thread t_i requesting lock l_i, Dimmunix finds the index k_j of lock l_i in the lock stack LS_j of the thread t_j that holds l_i (lines 1–4 in Algorithm 6). For each call stack CS_i, Dimmunix explores the CFG of the application bytecode backward, starting from CS_i's top frame, and following the call stack (lines 5–7 in Algorithm 6, lines 1,8,10,11 in Algorithm 7). Every time a lock (or unlock) statement corresponding to a synchronized block is encountered, the counter $k_{nesting}$ keeping the nesting level is incremented (respectively decremented); the counter is initially zero (lines 2–7 in Algorithm 7). Once $k_{nesting} = k_i$, the algorithm stops and returns the call stack CS'_i that results from removing the explored frames from CS_i and adding the frame representing the current program position (lines 4–5 in Algorithm 7). The

deadlock signature has the outer call stacks CS'_i (line 8 in Algorithm 6).

Input: Deadlocked threads $t_1, ..., t_n$ with call stacks $CS_1, ..., CS_n$, lock stacks $LS_1, ..., LS_n$, and requested locks $l_1, ..., l_n$.
Data: Control flow graph (CFG).
Output: Signature $S = \{(CS'_1, CS_1), ..., (CS'_n, CS_n)\}$, where CS'_i are the inferred outer call stacks.

1 **foreach** $i \in \overline{1,n}$ **do**
2 Let t_j be the thread holding l_i
3 Find the last index k_j of l_i in LS_j, corresponding to l_i's acquisition
4 **end**
5 **foreach** $i \in \overline{1,n}$ **do**
6 $CS'_i := getOuterCallStack(CS_i, k_i, CS_i.top, 0)$
7 **end**
8 **return** $\{(CS'_1, CS_1), ..., (CS'_n, CS_n)\}$

Algorithm 6: Building the signature of a deadlock.

Input: Inner call stack CS; Lock stack index k; Current statement s, initially the statement corresponding to CS's top frame; Current nesting level $k_{nesting}$, initially 0.
Data: Control flow graph (CFG).
Output: Outer call stack CS'.

1 **if** there exists a predecessor s' of s **then**
2 **if** s' is *monitorenter* **then**
3 $k_{nesting}++$
4 **if** $k_{nesting} = k$ **then**
5 **return** $CS.pop().push(s')$ //replace the top frame of CS with s'
6 **if** s' is *monitorexit* **then**
7 $k_{nesting}--$
8 **return** $getOuterCallStack(CS, k, s', k_{nesting})$
9 **else**
10 $CS := CS.pop()$ //remove the top frame
11 **return** $getOuterCallStack(CS, k, CS.top, k_{nesting})$
12 **end**

Algorithm 7: $getOuterCallStack(CS, k, s, k_{nesting})$: recursively computes the outer call stack corresponding to an inner call stack CS and a lock stack index k.

The above algorithm is sound for synchronized blocks, because nested synchronized blocks work like stacks (§3.1.1). Thanks to this property, the algorithm deterministically

stops at one program position. Therefore, it is enough to explore only one execution path in the CFG, to find the outer call stack.

Dimmunix inserts the avoidance code only at program positions involved in previously detected deadlocks. Deadlock avoidance needs to be done only at these positions, because Dimmunix avoids only previously detected deadlocks. More precisely, Dimmunix instruments the outer and inner lock statements, and the corresponding unlock statements. Since the synchronized blocks are properly nested (§3.1.1), the unlock statements corresponding to a lock statement s_l are easily found by exploring the CFG forward and keeping track of the nesting level. The unlock statements corresponding to s_l are the ones having the same nesting level as s_l.

Selective program instrumentation is needed also when dealing with blocked notifications. To avoid blocked notifications, Java Dimmunix needs to intercept synchronized blocks. Intercepting all the synchronized blocks would cancel the benefits of the selective instrumentation described above. For dealing with blocked notifications, Dimmunix intercepts only the synchronized blocks containing *notify(All)* calls.

Java Dimmunix also needs to intercept the synchronized blocks involved in blocked notifications, i.e., the synchronized block wrapping the wait call, and the one blocking the notification. Consider a blocked notification with signature (p_w^l, p_w^c, p_n^l), where p_w^l is the program position where the mutex l causing the deadlock was acquired by the waiter thread t_w, p_w^c is the position where t_w deadlocked (i.e., the location of the wait call), and p_n^l is the position where the notifier thread t_n requested l and deadlocked. Positions p_w^c and p_n^l are available immediately; they are the top frames of threads t_w and t_n's call stacks, at the moment of the deadlock. Position p_w^l is not available at runtime; the synchronized block at p_w^l is not intercepted, therefore its position is not available when the deadlock happens. To infer the position p_w^l, Dimmunix employs the same mechanism (described above) used to infer outer call stacks from inner call stacks.

Outer call stacks cannot be inferred deterministically if the deadlocks involve explicit resource acquisition/release operations (e.g., explicit lock/unlock operations). The reason is that the explicit synchronizations do not have a deterministic nesting, because the acquisition and release operations are decoupled. Therefore, whenever Dimmunix needs to

retrieve at runtime the call stacks (or program positions) of explicit synchronization operations, it has to intercept these operations and retrieve their call stacks (respectively program positions).

To be able to construct the signatures of deadlocks involving explicit synchronization operations, Java Dimmunix intercepts all the explicit synchronizations and keeps the program position of each resource acquisition. Java programs heavily use synchronized blocks/methods for synchronization. More than 90% of the synchronizations in the Java programs we studied are synchronized blocks/methods, as shown in Table 3.1. Therefore, since synchronized blocks/methods do not need to be instrumented (except the ones previously involved in deadlocks), the amount of instrumentation is drastically reduced for Java programs.

8.2 Inlining Call Stack Matching and Position Retrieval

If the deadlock signatures are on the critical path, the call stack matching becomes a bottleneck. If retrieving the call stacks is expensive, inlining the call stack matching considerably reduces the performance overhead (§10.1).

To match outer call stacks of deadlock signatures on-the-fly, Java Dimmunix inserts code that keeps track of the number of matched frames for each thread; we present this mechanism in detail in Algorithm 8. Before a thread t executes a method call (or lock statement) that matches a frame in an outer call stack CS of a deadlock signature, the counter *matches[CS, t]* is decremented (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 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 (lines 3–4). Thread t's execution matches CS up to $CS.depth$ if and only if *matches[CS, t] = 0* (line 9).

To efficiently obtain at runtime the program positions of explicit resource acquisition statements, Java Dimmunix statically finds these statements in the program bytecode, and instruments them as follows: right before each acquisition statement s, Dimmunix inserts code that stores the position of s in a thread-local variable. Therefore, s's program position

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] − −
8 end
9 return matches[CS,t] = 0
```

Algorithm 8: *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*.

can be retrieved (almost) instantly when s executes.

8.3 Reducing the Number of False Positives

Java Dimmunix for mutex deadlocks 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 (§8.3.1). If it was a FP, Dimmunix calibrates the matching precision for S (§8.3.2).

8.3.1 Detecting False Positives

Dimmunix is able to determine whether forcing a thread to yield indeed avoided a mutex deadlock or not. For each lock l, Dimmunix keeps the set *locksAcq[l]* of locks acquired and still held since l was acquired last time. For each thread t, Dimmunix keeps the set *locksHeld[t]* of locks held by thread t. For each thread t and lock l, Dimmunix keeps the set *instances[t, l]* of signature instantiations involving t and l that Dimmunix avoided since t acquired l last time. When thread t is about to release l, Dimmunix analyzes every instantiation from the *instances[t, l]* set, to determine whether it was a false positive (FP). Dimmunix classifies an instantiation I as a FP if and only if it finds no lock inversions in

the *locksAcq* sets of I.

We illustrate the FP detection mechanism in detail in Algorithm 9. Whenever Dimmunix avoids a signature instantiation $I = \{(t_1, l_1, CS_1), ..., (t_n, l_n, CS_n)\}$, it adds I to the sets $instances[t_1, l_1], ..., instances[t_n, l_n]$ (lines 1–6). When a lock l is about to be released by a thread t, Dimmunix performs the following steps for each instantiation I from the *instances[t, l]* set (lines 14–23):

1. Copies the *locksAcq[l]* set into I, at the position where l appears in I (line 15).

2. Checks if the other locks from I have been released, i.e., the *locksAcq* copies are present at all the positions in I (line 17).

3. If yes, Dimmunix finally checks if there is a lock inversion in instantiation I (line 18); if there is, I is classified as a TP, otherwise it is flagged as a FP (lines 19–21). There is a lock inversion in a signature instantiation $\{(t_1, l_1, CS_1), (t_2, l_2, CS_2), ..., (t_n, l_n, CS_n)\}$ if and only if l_1 belongs to the *locksAcq* copy corresponding to l_2, ..., and l_n belongs to the *locksAcq* copy corresponding to l_1 (line 18).

If no lock inversion is found in a signature instantiation I, then I is definitely a FP, i.e., the FP detection algorithm is sound. If a lock inversion is found in I, then I is likely to be a TP. However, some logic that prevents deadlocks even in the presence of lock inversions might have been activated; therefore, the FP detection algorithm is not complete. The explanation is simple: every mutex deadlock involves a lock inversion, but not all lock inversions represent mutex deadlocks.

The above algorithm works also for determining whether the avoidance of a starvation signature was a FP. As we showed in §5.2, yield cycles are in fact deadlocks.

8.3.2 Calibrating the Signature Matching Precision

Java Dimmunix calibrates the signature matching precision to reduce the number of FPs, while preserving the effectiveness of the avoidance mechanism. The goal is to have zero FNs, with as few FPs as possible.

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

Input: Thread t acquiring lock l, then releasing l; Signature S being avoided by t.
Data: Set *instances[t, l]*, initially empty; Set *locksHeld[t]*, initially empty; Sets *locksAcq*, initially empty.
Output: Number of FPs $nFP[S]$ and number of TPs $nTP[S]$ corresponding to signature S.

```
1  while Dimmunix finds an instantiation I = {(t_1, l_1, CS_1), ..., (t_n, l_n, CS_n)} of S do
2      foreach i ∈ 1,n do
3          instances[t_i, l_i] := instances[t_i, l_i] ∪ {I}
4      end
5      yield to avoid I
6  end
7  lock(l)
8  foreach l' ∈ locksHeld[t] do
9      locksAcq[l'] := locksAcq[l'] ∪ {l}
10 end
11 locksHeld[t] := locksHeld[t] ∪ {l}
12 //critical section
13 //before releasing l, check if the avoided instantiations were FPs
14 foreach I = {(t_1, l_1, CS_1), ..., (t_n, l_n, CS_n)} ∈ instances[t, l] do
15     I.locksAcq[l] := locksAcq[l] //initially, I.locksAcq[l] = null
16     //if all the other locks involved in I were released
17     if ∀i ∈ 1,n : I.locksAcq[l_i] ≠ null then
18         if ∀i ∈ 1,n : l_i ∈ I.locksAcq[l_(i+1)%n] then
19             nTPs[S]++
20         else
21             nFPs[S]--
22         end
23 end
24 locksHeld[t] := locksHeld[t] \ {l}
25 instances[t, l] := ∅
26 locksAcq[l] := ∅
27 unlock(l)
```

Algorithm 9: Extending the deadlock avoidance algorithm to detect false positives.

its outer call stacks. Choosing too large matching depths can cause Dimmunix to miss manifestations of the deadlock (i.e., have FNs), while choosing too shallow ones can lead to mispredicting a runtime call flow as being headed for deadlock (i.e., this is a FP).

We describe now how Dimmunix calibrates the matching depths at runtime. 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 the one illustrated in §6.3. If an application uses synchronization wrappers, the lock acquisitions always execute at the same program position. Keeping the matching depths at 1 would serialize all the critical sections, which is not desirable. Increasing the matching depths dynamically when FPs are encountered solves this problem.

When the matching depths become too large, a signature may not capture all the possible manifestations of the deadlock bug. More precisely, when the matching depths of a signature S's outer call stacks are > 1, there may be other signatures of the same deadlock bug ending in call stack suffixes that do not match S any longer. Once the deadlock bug manifests again, it means it has a signature that does not match S, otherwise Dimmunix would have avoided manifestations matching that signature.

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 lengths of the common suffixes of the outer call stacks of S and S'. Then, Dimmunix decrements the matching depths for S's outer call stacks to these lengths, and freezes them, i.e., they cannot be incremented further. Since the matching depths are frozen, Dimmunix discards signature S', to keep the deadlock history at a minimal size; for each deadlock bug, only the first encountered signature is kept in the history. If the deadlock reoccurs, S's matching depths are decremented again. This way, Dimmunix finds the highest matching depths for S's outer call stacks that preserve the ability to avoid all the possible manifestations of the deadlock bug. This merging algorithm is similar to the signature generalization algorithm used by Communix (§7.2.4).

From our experience, the number of signatures corresponding to a deadlock bug is low; we encountered up to 2 signatures (with outer call stack suffixes of size 10) for the deadlock bugs we reproduced §10.1.1. 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 signatures, i.e., 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. The reason is that, with each newly discovered signature, the calibration algorithm decreases the matching depths of the original signature's outer call stacks, and does not allow them to increase again. In other words, the avoidance becomes more conservative with each newly discovered signature of a deadlock bug.

8.4 Exploiting Escape Branches to Reduce Yielding Time

To reduce the duration of a yield, Java Dimmunix exploits the branches that escape the deadlock: when such a branch is taken, Dimmunix stops the avoidance process: it cancels the active yields and prevents the future ones, until the lock whose acquisition triggered the avoidance is released.

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 := ∅
2 foreach i ∈ 1,n do
3     foreach branch statement s ∈ CFG s.t. s_i^out ⇝ s do
4         Let B be the set of branches of s
5         if ∃b ∈ B s.t. b⇝s_i^in then
6             escape := escape ∪ {b' ∈ B | ¬b'⇝s_i^in}
7     end
8 end
9 return escape
```

Algorithm 10: *findEscapeBranches(S)*: finds the escape branches corresponding to a signature S.

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 Algorithm 10. 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.

Chapter 9

Prototype Implementations

In this chapter, we provide details about our Dimmunix implementations: Dimmunix for Java 9.1, Dimmunix for POSIX Threads 9.2, Dimmunix for Android 9.3, and Communix for Java 9.4. Java Dimmunix is an application-level implementation—it relies on automatically instrumenting the application bytecode at class load-time. POSIX Threads Dimmunix is a library-level implementation—it intercepts all the *pthread_mutex_lock/unlock()* calls. Android Dimmunix is a platform-wide implementation—it intercepts all the *monitorenter/monitorexit* operations within the Android OS. Communix is a collaborative deadlock immunity framework (§7); we implemented Communix on top of Java Dimmunix.

9.1 Dimmunix for Java: Application-level Instrumentation

Java Dimmunix immunizes programs against the following types of deadlocks:

- mutex deadlocks involving synchronized blocks and *ReentrantLock* objects
- hybrid deadlocks involving *ReentrantLock* objects, semaphores (i.e., *Semaphore* objects), and/or read-write locks (i.e., *ReentrantReadWriteLock* objects)
- initialization deadlocks involving synchronized blocks and class initialization
- external deadlocks involving file locks (i.e., *FileLock* objects)

- blocked notifications involving synchronized blocks, *Object.wait*, *Object.notify*, and *Object.notifyAll* calls

Dimmunix is implemented as a Java agent. When it boots, Dimmunix creates a monitor thread that is responsible for detecting yield cycles and non-mutex deadlocks. The agent defines a shutdown hook, which executes when the application shuts down. The shutdown hook invokes Dimmunix to perform the deadlock detection.

At class load-time, Dimmunix instruments the program bytecode with the avoidance code, using the AspectJ instrumentation framework [asp]. To selectively instrument synchronization instructions, we customized the implementation of AspectJ to instrument only the synchronization positions (i.e., program locations where synchronization is performed) indicated in a configuration file. To instrument (at class load-time) instructions situated at arbitrary locations (i.e., the code that stops the avoidance and the code for the inline call stack matching), we use the ASM bytecode instrumentation framework [Bruneton, 2007].

We chose a load-time instrumentation for two reasons. First, we wanted to ease the instrumentation of an application. The instrumentation is performed by the Dimmunix agent and AspectJ instrumentation agent; the only operation a user has to perform is to specify these two agents in the command line arguments when running a Java application. If we chose a static instrumentation, the user would have had to perform the tedious task of using the AspectJ compiler to instrument all ".class" files with Dimmunix. Second, we do not want to alter the persistent application bytecode. The user does not have to back up the uninstrumented version of the application; he/she can just remove the two agents from the command line arguments, if he/she wants to disable Dimmunix.

To detect mutex deadlocks, Dimmunix simply invokes the JVM's deadlock detection by calling *ThreadMXBean.findDeadlockedThreads* method. This method returns the list of deadlocked threads. Dimmunix obtains all the information it needs about the deadlocked threads (i.e., their call stacks, lock stacks and requested locks) from the *ThreadInfo* object associated with each thread. Starting from this information, Dimmunix infers the deadlock signature by statically analyzing the bytecode, using the Soot static analysis framework [Vallée-Rai et al., 1999].

To detect yield cycles, Dimmunix keeps the yield edges in its own RAG and periodically augments it with the necessary request and hold edges from the JVM's RAG, starting from

the yield edges. Starting from a yield edge $t_1 \longrightarrow t_2$, Dimmunix retrieves the lock l_2 that thread t_2 is waiting for, and the thread t_3 that holds l_2 from t_2's *ThreadInfo* object. Then, Dimmunix adds the request edge $t_2 \longrightarrow t_2$ and the hold edge $t_3 \longrightarrow l_2$ to its RAG. The process stops when the current thread does not wait for any lock or the current lock is not held by any thread.

The JVM does not detect non-mutex deadlocks; therefore Dimmunix needs to implement the detection of these deadlocks. To detect initialization deadlocks, Dimmunix only needs to instrument the static class initializers; the rest of the information is provided by the JVM through the *ThreadInfo* class. To detect blocked notifications, Dimmunix intercepts the wait and notify(All) calls and the synchronized blocks corresponding to notify(All) calls; Dimmunix also uses information provided by the *ThreadInfo* class.

To avoid initialization deadlocks, the Dimmunix agent intercepts the class loading and performs the avoidance, before the static class initializer is invoked. To avoid blocked notifications, Dimmunix needs to precompute the positions of all synchronized blocks corresponding to notifications (i.e., wrapping notify(All) calls); to do this, Dimmunix uses the Soot static analysis framework [Vallée-Rai et al., 1999].

To detect and avoid read-write (respectively semaphore) deadlocks, Dimmunix intercepts all the synchronization operations executed on read-write locks (respectively semaphores). To efficiently retrieve the program positions of these operations, Dimmunix inspects the application bytecode using Soot, to detect the positions of the acquisition and release statements performed on read-write locks or semaphores. Then, Dimmunix uses ASM to instrument the program at these positions with code that retrieves them (almost) instantly at runtime; we chose this technique to avoid expensive calls to *Thread.getStackTrace()*.

To detect and avoid external deadlocks, Dimmunix uses MySQL server for the database storing the process-shared synchronization state. Dimmunix intercepts all the calls to *FileChannel.lock()* and *FileChannel.unlock()* methods.

9.2 Dimmunix for POSIX Threads: Library-level Interception

POSIX Threads Dimmunix offers immunity against mutex deadlocks. We have two POSIX Threads Dimmunix implementations: one targets Linux distributions and uses LD_PRELOAD to override the mutex synchronization routines, and the other one targets FreeBSD distributions and changes the POSIX Threads library to provide deadlock immunity. Both implementations intercept the calls to *pthread_mutex_lock* and *pthread_mutex_unlock*, in order to invoke the Dimmunix core upon each mutex synchronization operation.

The most difficult challenge we faced was related to custom memory allocators that were calling *pthread_mutex_lock* at link-time. In Firefox, for instance, such allocators are invoked when libraries are loaded, i.e., before the address of the real *pthread_mutex_lock* is available to Dimmunix. This causes a segmentation fault when Dimmunix attempts to invoke the real *pthread_mutex_lock* routine, because its address is still *null*. We chose to simply skip the calls to the real *pthread_mutex_lock* and *pthread_mutex_unlock* routines, as long as they are not loaded, i.e., *dlsym* returns *null*. This fix did not cause any problems (e.g., crashes), in any of the C/C++ applications that we ran with Dimmunix (e.g., Firefox, VLC).

Since C/C++ applications are not compiled with source code information by default, we had to use instruction offsets in the binary for the call frames in the deadlock signatures. The challenge with the binary offsets is the fact the addresses returned by a *backtrace* call change in each execution, even though the frames point to the same instructions. Fortunately for Dimmunix, the offset of an instruction within its binary is the same in each program execution. Therefore, we compute offsets relative to the beginning of the binary, for each return address r provided by a *backtrace* call; for that, we first obtain the address b of the beginning of the binary, then the offset we need is $r - b$.

If we would keep only binary offsets in the deadlock signatures, we would not be able to reconstruct the address of frame in the next execution of the program, when loading the signatures into the memory. Therefore, Dimmunix keeps in a signature the name of the corresponding binary next to each offset. When Dimmunix loads a signature, the frame addresses are reconstructed as follows: First, Dimmunix obtains the start address b of the

binary using *dlopen*. Then, it uses from the signature the offset o within the binary, to reconstruct the frame address; this address is simply $b+o$.

We also implemented an efficient Dimmunix prototype within the FreeBSD POSIX Threads library. This implementation achieves zero overhead look-ups for thread and lock nodes in the RAG, because pointers to these nodes are stored as fields in the POSIX thread and mutex data structures. We evaluate this implementation in §10.3.

9.3 Dimmunix for Android OS: Platform-level Interception

In this section, we explain the design choices we took in the implementation of Android Dimmunix. First, we explain our choice of implementing Dimmunix within Android's Dalvik VM. Second, we explain our decision to store only the top frames in the outer call stacks.

We implemented Dimmunix in Android OS 2.2, within the Dalvik VM. We modified Dalvik VM, to call Dimmunix upon each monitor acquisition/release. More precisely, we changed the code of Dalvik VM's synchronization routines to call into Dimmunix, as we illustrate in Figure 9.1. Dalvik VM is a customized JVM, in which Android OS runs all the applications. Since platform-wide deadlock immunity cannot be implemented in the kernel space (§6.5), an implementation within the Dalvik VM is the natural choice.

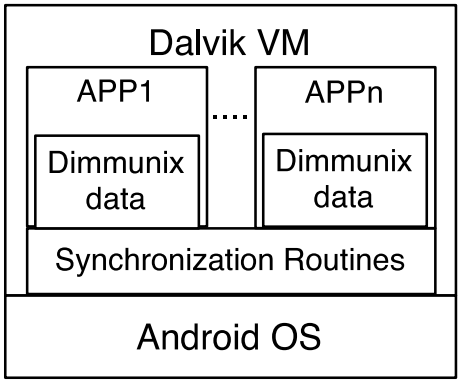

Figure 9.1: The Architecture of Android Dimmunix.

Such an implementation allows Android Dimmunix to detect and avoid deadlocks caused by lock inversions due to *wait()* calls. We show how such an inversion can lead to a deadlock, in the Java code below:

```
Thread t1:                          Thread t2:
synchronized(x) {                   synchronized(x) {
 synchronized(y) {                    synchronized(y) {
  x.wait();
 }                                    }
}                                   }
```

If thread t_1 is executing *x.wait()* and thread t_2 just acquired monitor x, the two threads are going to deadlock: when thread t_1 finishes waiting on x, it will attempt to reacquire x, while holding monitor y; thread t_2 waits for y, while holding x. To detect and avoid such deadlocks, we changed the code of the *Object.wait()* native method: for each *x.wait()* call, Dimmunix is called before and after the reacquisition of x, at the end of the *wait* function.

An instrumentation-based Dimmunix for Java cannot handle such deadlocks caused by *wait()* calls. For each *x.wait()* call, the reacquisition of x at the end of the wait call has to be intercepted; therefore, the code of the *Object.wait()* native method has to be changed.

In order to obtain the outer call stacks, Android Dimmunix needs to retrieve, for each lock acquisition, the call stack the owner thread had when it acquired the lock. For each lock l, Dimmunix stores in $l.acqPos$ the call stack corresponding to the last acquisition of l. In the signature of a deadlock involving threads t_1 and t_2, and locks l_1 and l_2, the outer call stacks are $l_1.acqPos$ and $l_2.acqPos$.

Android Dimmunix uses outer call stacks of depth 1 in the signatures. Retrieving the call stack of each lock acquisition is expensive; therefore, we decided to retrieve only 1 frame, at the cost of a higher false positives rate in the deadlock avoidance. As we showed in §6.3, it is safe to use outer call stacks of depth 1 for synchronized blocks/methods.

Android Dimmunix handles only synchronized blocks/methods. However, this is not a major shortcoming; there are 1,050 synchronized blocks/methods and only 15 explicit lock/unlock operations in Android 2.2 essential applications.

Android Dimmunix has two components: the Dimmunix core, which implements the deadlock immunity, and the integration code, which (1) obtains the information needed to

call the Dimmunix core, (2) extends existing Dalvik VM data structures, to provide instant access to per-thread/monitor Dimmunix-related information, and (3) calls the Dimmunix core. The core has 661 lines of code (LOC), and the integration code has 155 LOC.

Android Dimmunix uses the following data structures: The struct *Node* stores a RAG node corresponding to a thread/monitor object. The struct *Position* stores the program location of a *monitorenter* operation and the set of threads that hold (or are allowed by Dimmunix to acquire) locks at that location.

To achieve zero-overhead look-up of the RAG node corresponding to a thread/monitor object, we added a *"node"* field in Dalvik's *Thread* and *Monitor* structs. We also added a *"stackBuffer"* field in struct *Thread*, where Dimmunix retrieves the call stack. We illustrate these changes in the code below:

```
typedef struct Thread {
 ...
 Node node; //RAG node
 char* stackBuffer; //call stack buffer
} Thread;

struct Monitor {
 ...
 Node node; //RAG node
};
```

We describe now how Dimmunix's data structures are initialized. Whenever the Dalvik VM forks a new process, the *initDimmunix* routine is called, to initialize Dimmunix's global data for that process, e.g., the deadlock history, and the *positions* global map that associates a unique *Position* object to each program location. Remember that Android Dimmunix runs in user-space, therefore this global data is per-process. We modified the routines that fork Dalvik processes, i.e., *Dalvik_dalvik_system_Zygote_fork* and *forkAndSpecializeCommon*, to initialize Dimmunix as soon as the child process starts, as we show in Figure 9.2. Each time a thread (or monitor) object *t* (respectively *mon*) is created by Dalvik's *allocThread* (respectively *dvmCreateMonitor(Object* obj)*) function, the integration code initializes the RAG node corresponding to *t* (respectively *mon*). We illustrate

these changes in Figure 9.3.

```
pid_t pid;
...
pid = fork();
if (pid == 0) {
  //child process, initialize Dimmunix
  initDimmunix();
  ...
}
return pid;
```

Figure 9.2: Initializing Dimmunix when a child process starts.

```
static Thread* allocThread(...) {
 Thread* t;
 ...
 initNode(&t->node, t, T_THREAD);
 //initialize the stack buffer
 t->stackBuffer = (char*)malloc(STACKSIZE);
 return t;
}

Monitor* dvmCreateMonitor(Object* obj) {
  Monitor* mon;
  ...
  initNode(&mon->node, obj, T_MONITOR);
  return mon;
}
```

Figure 9.3: Initializing the RAG nodes.

The Dimmunix core is called by three routines, upon each *monitorenter/monitorexit* statement: The *Request()* routine executes before a *monitorenter* statement; it performs deadlock detection and returns whether a deadlock signature would be instantiated if the lock acquisition would be approved. The *Acquired()* routine runs immediately after a *monitorenter* statement, and the *Release()* routine runs right before a *monitorexit* statement; these two routines perform only RAG updates. For thread-safety, Dimmunix uses a global

lock within these methods. As we show in §10.4, Dimmunix is efficient, even in the presence of this global lock, because the calls to the three methods are cheap.

The Dalvik VM implements the *monitorenter*, *monitorexit*, and *Object.wait()* statements in the routines *lockMonitor*, *unlockMonitor*, and *waitMonitor*. We changed *lockMonitor* to invoke the *Request* and *Acquired* Dimmunix routines:

```
void lockMonitor(Thread* t, Monitor* mon) {
 //monitorenter(mon), before acquiring mon
 dvmGetCallStack(t);
 Position* pos = getPosition(t->stackBuffer);
 int sigId; //matched sig in history
 do {
  sigId = Request(&t->node, &mon->node, pos);
  //if instantiation found, yield and retry
  if (sigId >= 0)
   wait(history[sigId]);
 } while (sigId >= 0);
 //t is allowed to acquire mon
 ...
 //after acquiring mon
 Acquired(&t->node, &mon->node);
}
```

We implemented the *dvmGetCallStack* routine that retrieves the top frame of a thread *t*'s call stack into the *t->stackBuffer* buffer. As long as there is a signature *S* in history that is instantiated, Dimmunix makes the caller thread wait on a condition variable associated with *S*.

We changed the *unlockMonitor* and *waitMonitor* routines, to call Dimmunix's *Release* function, right before the monitor is released. If the released monitor was acquired with a call stack in the history, Dimmunix resumes all the threads waiting on signatures containing that call stack, as we illustrate in Figure 9.4.

Dimmunix turns the thin lock associated with an object *x* into a fat lock, i.e., a *Monitor* object, as soon as a *monitorenter(x)* statement is called. The reason is that a RAG lock node

```
//thread t, before releasing mon
Position* pos = mon->node.acqPos;
if (pos->inHistory) {
 int sigId;
 for (sigId=0; sigId < histSize; sigId++) {
  if (history[sigId].contains(pos))
   notifyAll(history[sigId]);
 }
}
Release(&t->node, &mon->node);
//release mon
```

Figure 9.4: Changes in the *unlockMonitor* and *waitMonitor* routines.

is encapsulated in a *Monitor* object; the thin lock is a simple integer field, which cannot accommodate a RAG node. To make sure that each *monitorenter* statement is executed on a fat lock, we added the code below before calls to *lockMonitor*:

```
//if the lock is thin
if (LW_MONITOR(obj->lock) == NULL) {
  pthread_mutex_lock(&globalLock);
  //if still thin, fatten the lock
  if (LW_MONITOR(obj->lock) == NULL) {
    Monitor* mon = dvmCreateMonitor(obj);
    obj->lock = (u4)mon | LW_SHAPE_FAT;
  }
  pthread_mutex_unlock(&globalLock);
}
```

Most of the CPU and memory consumptions are due to the computations related to the call stacks. Dimmunix allocates a unique *Position* object for each call stack of a synchronization operation. Using call stacks of depth 1 minimizes the number of *Position* objects. The *Thread.stackBuffer* field makes the call stack retrieval more efficient; the field is thread-local, and the *dvmGetCallStack* routine does not need to allocate memory for storing the current call stack.

To eliminate the overhead incurred by the call stack retrieval, the compiler could produce a unique id for each synchronization statement, based on the location of that statement. The ids would be constant in all the executions of the application, because every id is bound to a program location. Dimmunix can use the ids instead of the call stacks, to identify synchronization statements. The compiler can pass the id as a parameter to the *lockMonitor*, *unlockMonitor*, and *waitMonitor* routines; this way, retrieving the id would not incur any performance penalty.

If the deadlock signatures are on the critical path, Dimmunix may incur significant performance overhead, due to the computations performed in the avoidance code. More precisely, the *Request* routine looks for signatures in the history that are instantiated. For signature matching, Dimmunix maintains for each *Position* object p a queue that stores the threads holding (or allowed to acquire) locks with position (call stack) p. To reduce the number of memory allocations, Dimmunix uses a second queue, where the elements deleted from the main queue are stored. Whenever a thread t needs to be added to the main queue and the second queue is non-empty, Dimmunix pops an element from the second queue, makes it point to t, and adds it to the main queue.

Android Dimmunix does not handle deadlocks involving native code (i.e., using Android NDK). However, it is possible to handle such deadlocks, by intercepting the synchronization operations within the POSIX Threads library. This must be done carefully, because the Dalvik VM already uses this library to implement the synchronization operations in Java. Therefore, Android OS should allow Dimmunix to intercept the calls to the POSIX Threads synchronization routines only when native code executes.

9.4 Communix for Java: Collaborative Immunity

Communix is implemented on top of Java Dimmunix, and has four components: Communix server, Communix client, Communix agent, and Communix plugin. The Communix server and client use the *java.net.Socket* class for communication. The Communix client is running as a separate Java process; it periodically (once a day) retrieves new deadlock signatures from the server, into a local repository. The Communix plugin is implemented within Dimmunix, in a separate package; Dimmunix invokes the plugin to upload newly

discovered deadlock signatures to the server. At startup, Dimmunix instructs the Communix agent to scan the local repository for new signatures that match the running application; then, the Communix agent validates each signature that matches the application.

Since deadlock signatures are never removed from a repository, checking for new signatures becomes trivial. The Communix plugin keeps a per-application cursor indicating to point until which the local signature repository was scanned last time.

For checking if the outer call stacks of a deadlock signature end in nested synchronized blocks (methods), Communix agent uses the Soot bytecode analysis framework [Vallée-Rai et al., 1999].

Chapter 10

Evaluation

In this chapter, we evaluate the prototype implementations. First, we evaluate the part of Java Dimmunix responsible for mutex deadlocks (§10.1), then the part responsible for non-mutex deadlocks (§10.2). We evaluate POSIX Threads Dimmunix in §10.3, and Android Dimmunix in §10.4. Finally, we evaluate Communix in §10.5.

We explain first the formula we use to compute the performance overhead incurred by Dimmunix. No matter whether we measure execution time or throughput (e.g., synchronizations per second), we compute the performance overhead incurred by Dimmunix (in %) using the formula $\frac{|x_{Dimmunix} - x_{base}|}{x_{base}} \cdot 100$, where $x_{Dimmunix}$ is the execution time (or throughput) of the application running with Dimmunix, and x_{base} is the execution time (respectively throughput) of the vanilla application. If we measure the execution time, an overhead of 90% means that the application is 90% slower; if we measure the throughput, it means a 10x throughput reduction.

10.1 Java Dimmunix—Mutex Deadlocks

In this section, we evaluate the most important part of Java Dimmunix, i.e., the one that deals with mutex deadlocks. We answer the following 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 (§10.1.1)? What performance overhead does Dimmunix introduce, and how does this overhead vary as parameters (e.g., number of threads, number of

signatures) change (§10.1.2, §10.1.3)? What is the impact of false positives on performance (§10.1.4)? How effective are the optimizations we performed (§10.1.5)?

To quantify Dimmunix's impact on system performance, we used as metrics throughput (e.g., requests per second, synchronizations per second) and execution time. We report in §10.1.2 end-to-end measurements on real applications, and in §10.1.3 we use synthetic microbenchmarks to drill deeper into the performance characteristics.

10.1.1 Effectiveness Against Real Deadlocks

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

True Deadlock Bugs

System	Bug #	Deadlock Between ...	TP Yields	FP Yields	Signatures
MySQL 5.0 JDBC	2147	PreparedStatement.getWarnings() and Connection.close()	1	0	1
MySQL 5.0 JDBC	14972	Connection.prepareStatement() and Statement.close()	1	0	1
MySQL 5.0 JDBC	31136	PreparedStatement.executeQuery() and Connection.close()	1	0	1
MySQL 5.0 JDBC	17709	Statement.executeQuery() and Connection.prepareStatement()	1	0	1
Limewire 4.17.9	1449	HsqlDB TaskQueue cancel() and shutdown()	15	0	2
ActiveMQ 3.1	336	ActiveMQMessageConsumer processMessage() and setMessageListener()	8–12	0–1	1
ActiveMQ 4.0	575	Queue.dropEvent() and PrefetchSubscription.add()	8–11	34–48	1

Table 10.1: Reported deadlock bugs avoided by Dimmunix in real Java applications.

To verify the effectiveness against real bugs, we reproduced deadlocks that were reported against real systems. 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 10.1. We include the number of yields recorded during the trials with full Dimmunix as a measure of how often deadlock signatures were encountered and avoided. For most cases, there is one yield, corresponding to the one deadlock reproduced 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-prone execution pattern later. For all but the ActiveMQ tests there were no false positives. In the case of ActiveMQ 3.1 (respectively 4.0), Dimmunix found that 0–1 (respectively 34–48) yields were FPs and 8–12 (respectively 8–11) were TPs.

We configured the maximum length of the call stack suffixes and the matching depth to 10; in 100 runs that deterministically reproduced the deadlocks, Dimmunix did not find more than two signatures with different outer call stacks for a deadlock bug (in most of the cases it found just one). Even if the matching precision was high, the signatures were as effective as with a matching depth of 1, in most of the cases.

However, if a deadlock bug has many deadlock signatures with different outer call stacks of depth 10, Dimmunix will most likely need to encounter only a couple of them to fully protect the application against the deadlock bug. With each newly discovered signature, the calibration algorithm decreases the matching depths of the original signature, and does not allow them to increase again, i.e., the deadlock avoidance becomes more conservative (§8.3.2).

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 Java 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 10.2 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 10.2: Java JDK 1.6 deadlocks avoided by Dimmunix.

10.1.2 Real Applications

We measured the end-to-end overhead in "immunized" JBoss, MySQL JDBC, Limewire, Vuze, and Eclipse. For JBoss we used the RUBiS e-commerce benchmark [RUBiS], and for MySQL JDBC we used the JDBCBench benchmark [JDBCBench]. We are unaware of any benchmarks for Limewire, Vuze, and Eclipse. Therefore, we measured Dimmunix's impact on operations that are lock-intensive and have a constant execution time. For Eclipse (respectively Vuze), we measured the time it takes to start it and immediately shut down. For Limewire, we measured the execution time of an upload test included in Limewire's source code.

We compare the performance of Dimmunix's deadlock avoidance mechanism to the performance of the simple (respectively hybrid) avoidance (§5.2.1). Our goal is to find out which avoidance mechanism is the most efficient. Dimmunix's avoidance mechanism surpasses both the hybrid and simple avoidance techniques.

Table 10.3 presents the characteristics of the performance tests we ran on real applications, i.e., number of threads performing synchronization, number of synchronizations per second, number of objects used as mutex locks during the execution, number of synchronization positions (i.e., program positions where synchronization is performed), percentage of nested synchronization operations covered by the 20 "busiest" nested synchronization positions, the number of lock objects used at these program positions, number of call stacks of depth 10 that appeared when performing nested synchronization, and percentage of nested synchronization operations covered by the 20 "busiest" nested synchronization call stacks. As shown in the table, JBoss and MySQL JDBC tests are the most lock-intensive applications; they perform 100,000–180,000 synchronizations per second. We studied here only the synchronizations on mutexes, i.e., synchronized blocks and calls to *ReentrantLock.lock()*. We call a synchronization operation on lock l nested if and only if another synchronization is performed inside l's critical section. A nested synchronization position is a program position where at least one nested synchronization operation was performed.

We measured the performance overhead incurred by Dimmunix with a history 20 synthetic deadlock signatures ending in the top 20 "busiest" nested synchronization positions, which according to Table 10.3 cover at least 99% of the nested synchronization operations. In other words, we measure the performance overhead in a worst-case scenario where most of the nested synchronization positions are part of deadlock signatures and therefore are instrumented with Dimmunix's avoidance code. Regarding the avoidance overhead, the synthesized signatures have the same effect as real ones. We chose only positions of nested synchronizations, because unnested synchronizations cannot be involved in mutex deadlocks.

The matching depth is set to 5, which provides a good accuracy for Dimmunix's avoidance (i.e., the yields are few) and a good accuracy for the hybrid avoidance, for the applications we studied (see Table 10.4). In practice, Dimmunix initializes the matching depth

Characteristics	System, Test Scenario				
	JBoss, RUBiS	MySQL JDBC, JDBCBench	Eclipse, Startup + Shutdown	Limewire, Upload test	Vuze, Startup + Shutdown
number of threads	284	505	31	214	76
number of sync ops/sec	178,279	100,855	78,536	13,905	28,872
nested sync ops	25%	30%	47%	27%	7%
number of lock objects	12,022	100,009	153,541	41,746	1,958
number of sync positions	399	17	878	247	373
nested sync ops covered by the 20 "busiest" nested sync positions	99%	100% (6 nested sync pos)	99%	99%	99%
number of lock objects used by the 20 "busiest" sync positions	4,767	4,777	11,337	4,893	1,037
number of synchronization call stacks of depth 10	23,254	99	95,875	2,230	12,051
nested sync ops covered by the 20 "busiest" nested sync call stacks	93%	100%	67%	95%	85%

Table 10.3: Mutex synchronization in real Java applications.

to 1, and increases it as FPs are encountered in the deadlock avoidance; therefore, the maximum matching depth is the length of the call stack suffix. We set the maximum length of the call stack suffixes to 10, to accommodate applications that may require higher matching depths. We do not go beyond depth 10, because the call stack matching is an expensive operation. Even if the hybrid avoidance cannot use the full suffixes (because it is unable to detect FPs and increase the matching depth), we use the same suffix size; we want to compare the hybrid avoidance to Dimmunix's avoidance in the same conditions, because the two techniques have identical call stack matching mechanisms. However, we set the suffix sizes to 1 for the simple avoidance, because only the top frames are relevant.

Table 10.4 presents the results of our measurements. The "Vanilla" row shows the performance (request/transaction throughput, or execution time) of the vanilla application. The "0 (respectively 20) sigs Dimmunix" row shows the performance of the application instrumented with Dimmunix, with 0 (respectively 20) signatures in the history. In the "20 sigs simple (respectively hybrid)" rows, the simple (respectively hybrid) avoidance algorithm is used to avoid deadlocks.

When the history is empty, there is only class loading overhead, due to invoking the

System, Test scenario		Performance	Overhead	#yields	Yield cycles
JBoss, RUBiS	Vanilla	525 req/sec			
	0 sigs Dimmunix	523 req/sec			
	20 sigs Dimmunix	373 req/sec	28.7%	50	0
	20 sigs hybrid	351 req/sec	32.9%	89,339	6
	20 sigs simple	198 req/sec	62.1%	8,041,986	10
MySQL JDBC, JDBCBench	Vanilla	2309 txn/sec			
	0 sigs Dimmunix	2264 txn/sec			
	20 sigs Dimmunix	1635 txn/sec	27.8%	0	0
	20 sigs hybrid	1691 txn/sec	25.3%	450,299	0
	20 sigs simple	518 txn/sec	77.1%	800,301	142
Eclipse, Startup+Shutdown	Vanilla	13.77 sec			
	0 sigs Dimmunix	23.01 sec			
	20 sigs Dimmunix	27.50 sec	19.5%	0	0
	20 sigs hybrid	34.72 sec	50.9%	10,786	6
	20 sigs simple	72.70 sec	216%	160,714	15
Limewire, Upload test	Vanilla	20.7 sec			
	0 sigs Dimmunix	31.25 sec			
	20 sigs Dimmunix	33.41 sec	6.9%	0	0
	20 sigs hybrid	33.77 sec	8%	10,781	0
	20 sigs simple	31.92 sec	2.1%	42,970	0
Vuze, Startup+Shutdown	Vanilla	10.13 sec			
	0 sigs Dimmunix	15.67 sec			
	20 sigs Dimmunix	16.95 sec	8.2%	0	0
	20 sigs hybrid	16.83 sec	7.4%	10	0
	20 sigs simple	16.66 sec	6.3%	79	0

Table 10.4: Performance results on real applications.

AspectJ load-time weaver. This is a one-time overhead, that manifests only when a class is used for the first time in the application. Most of this overhead occurs during the application startup, since most of the classes are loaded at startup. As shown in Table 10.4, the load-time overhead is the highest in the Eclipse, Limewire, and Vuze tests. Since the load-time overhead is not due to Dimmunix's computations, we use the "0 sigs" values as a baseline for measuring the performance overhead incurred by Dimmunix.

Once the classes are loaded, the performance overhead is only due to Dimmunix; the worst-case overhead is the highest in the JBoss, MySQL JDBC, and Eclipse tests. Note that these measurements are for a worst-case scenario, i.e., most of the nested synchronization statements are part of deadlock signatures and therefore are intercepted by Dimmunix. Therefore, the worst-case performance overhead can be noticeable in applications like

JBoss, MySQL JDBC, and Eclipse. If no signature has call stacks on the critical path, the overhead due to Dimmunix is negligible (i.e., $< 2\%$) for all the tests, as if the deadlock history was empty. The reason is that only the synchronization statements belonging to deadlock signatures are intercepted by Dimmunix.

In Table 10.4, we also compare the performance of Dimmunix's avoidance to the performance of the simple and hybrid avoidance techniques. Dimmunix's avoidance is by far the most accurate, causing the lowest number of yields and no yield cycles; the other two avoidance techniques cause much more yields and also cause yield cycles. Since the deadlock signatures are synthetic, we can assume that most of the yields are FPs. Therefore, we can deduce that Dimmunix's avoidance mechanism is much more accurate than the simple and hybrid avoidance mechanisms. Dimmunix's avoidance caused no yield cycles, while the other avoidance mechanisms caused yield cycles; this shows that a higher number of yields increases the probability of incurring starvation. We do not claim that Dimmunix's avoidance mechanism cannot cause yield cycles in practice; we just show empirically that Dimmunix's avoidance causes fewer yield cycles than the other two techniques.

Thanks to the higher accuracy, Dimmunix's avoidance incurs a substantially lower performance overhead than the simple avoidance in JBoss, MySQL JDBC, and Eclipse, as Table 10.4 shows. Compared to the hybrid avoidance, Dimmunix's avoidance incurs practically the same performance overhead in 4 out of 5 applications; in Eclipse, Dimmunix's avoidance is significantly more efficient. Dimmunix's avoidance is more robust than the hybrid avoidance, because it causes fewer yield cycles. Even if the simple avoidance causes much more yields, its performance may be better than Dimmunix's avoidance (as shown in Table 10.4), for the following reasons: (1) the time spent in critical sections may be small enough for the yields to not count, (2) the simple avoidance mechanism is cheaper (it only consists of acquiring semaphores associated with signatures), and (3) the simple avoidance needs to match only one call frame.

Even if there are few yields, Dimmunix may incur a noticeable performance overhead, as Table 10.4 shows. If at least one call stack of a signature S is on the critical path, i.e., it ends in a lock statement that is executed often, S can incur a noticeable overhead, although it is instantiated rarely. The reason is that Dimmunix has to match the execution flow against S, in order to avoid instantiating S.

In 3 out of 5 applications the Dimmunix's avoidance and the hybrid avoidance incur a substantially lower performance overhead than the simple avoidance. This means that matching the call stacks accurately greatly improves Dimmunix's performance.

We profiled the JBoss, MySQL JDBC, and Eclipse tests—we found out that most of the overhead is in the call stack matching. This is due to the fact that the deadlock signatures share call frames. If two signatures share a call frame pointing to program position p, the inline matching code is invoked for both signatures, each time the program execution reaches p.

10.1.3 Microbenchmarks

To dissect Dimmunix's performance behavior and understand how it varies with various parameters, we wrote a lock-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 until δ_{in} instructions are executed, before being released and a new lock is requested after δ_{out} instructions. The delays are implemented as busy loops that simply execute incrementation instructions, 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 2, which is the usual number of threads involved in a deadlock. Generated signatures consist of stack combinations for synchronization operations in the benchmark program—not signatures of real deadlocks, but avoided as if they were. The benchmark has a configurable number N_p (typically grater than H) of synchronization positions. The H signatures cover H synchronization positions (each signature has two identical call stacks); only these positions are instrumented by Dimmunix. If $H = N_p$, Dimmunix ends up intercepting all the synchronization operations.

The typical benchmark configuration that we chose is the following: We configure N_p to 20 synchronization positions, and H to 10 signatures. We chose $N_p = 20$ because most of the nested synchronizations occurred in 20 program positions, in all the applications we

studied, as shown in Table 10.3. Since $H = 10$, Dimmunix is invoked for 50% of the nested synchronization operations. The delays δ_{in} and δ_{out} are typically configured to $1,000$ and respectively $50,000$ instructions, which allows a throughput of up to $200,000$ synchronizations per second on our 8-core machines; this is similar to the highest throughput of synchronization operations that we encountered in the real applications we tested. By default, we configure N_t to 500 threads, which is similar to the maximum number of threads in the applications we tested. We configure N_l to $10,000$ locks, which is similar to the largest number of locks used in nested synchronizations that we encountered.

The default Dimmunix configuration that we used is the following: We fix the matching depth to 5 and the length of the call stack suffixes to 10, as in the tests we performed on real applications. We disable the avoidance, i.e., yielding is not performed, in order to measure just the overhead due to Dimmunix's computations. By default, the inline call stack matching and the selective instrumentation are enabled, for efficiency. However, the FP detection and the matching depth calibration are not enabled by default, because no yields are performed.

Overhead as a function of the number of threads: Figure 10.1 shows how synchronization throughput varies with the number of threads. We observe that Dimmunix scales well: for up to 1,024 threads, Dimmunix incurs 4–5% overhead, in a lock-intensive scenario with 200,000 synchronizations per second. Yield decisions are ignored, in order to measure only the computational overhead; the overhead due to yielding is measured in §10.1.4.

According to Figure 10.1, Dimmunix's avoidance and the hybrid avoidance have very similar computational overheads. Therefore, we evaluate from now on only the efficiency of Dimmunix's avoidance. We decided not to evaluate the simple avoidance in the microbenchmark, because the tests we performed on real applications clearly showed that it is less efficient than the other two techniques (§10.1.2).

The hybrid avoidance causes significantly more signature matches than Dimmunix's avoidance. The reason is that every time the execution of a thread t matches a call stack belonging to a signature S, the hybrid avoidance regards S as matched and decides to make thread t yield.

As the microbenchmark 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

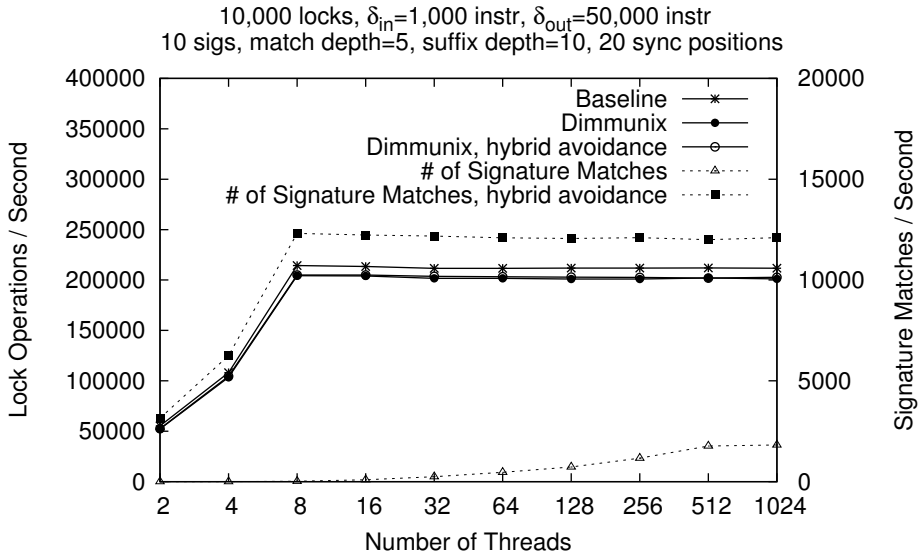

Figure 10.1: Dimmunix microbenchmark lock throughput as a function of number of threads.

lock/unlock operations. To validate this hypothesis, we measured the variation of the synchronization throughput with the values of δ_{in} and δ_{out}—Figure 10.2 shows the results.

The performance overhead introduced by Dimmunix is the highest (31%) when the program does almost nothing but lock and unlock (i.e., δ_{in}=1000, δ_{out}=0). This is not surprising, because Dimmunix intercepts the calls to lock/unlock and performs additional computation in the critical path. However, as the interval between critical sections (δ_{out}) or inside critical sections (δ_{in}) increases, the overhead decreases. For inter-critical-section intervals of $10,000$ instructions or more, the overhead is modest.

Note that a direct performance comparison between Dimmunix and the baseline is somewhat unfair to Dimmunix, because non-immunized programs deadlock and stop running, whereas immunized ones continue running and do 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

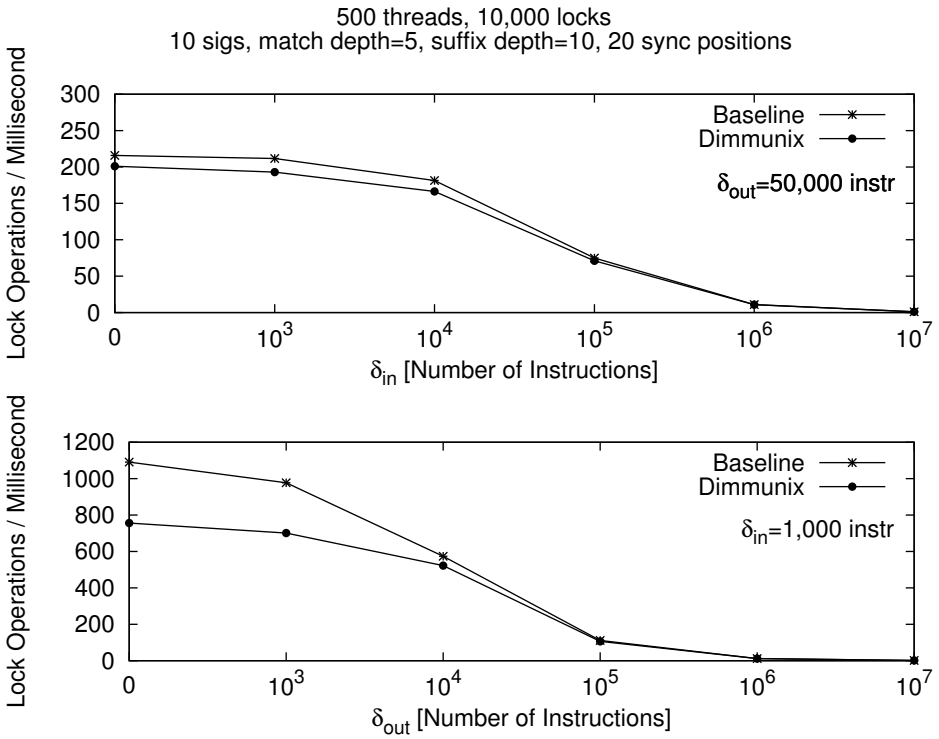

Figure 10.2: Variation of lock throughput as a function of δ_{in} and δ_{out}.

in practice are limited to two threads [Lu et al., 2008], so variation with signature size is not that interesting.

We show in Figure 10.3 the performance overhead introduced by varying history size from 0 to 256 signatures, and the matching depth from 1 to 10. The overhead introduced by the history size and matching depth is insignificant between 2 and 16 signatures, i.e., 1–2.7%. The overhead between 32 and 256 signatures increases significantly (i.e., 2.7–35.8% for matching depth > 1, and 7.2–41% for matching depth 1), because the percentage of synchronization positions instrumented with avoidance code increases rapidly. The overhead corresponding to the matching depth 1 is slightly higher compared to the matching depths 5 and 10 (the difference becomes noticeable for more than 16 signatures), because the number of call stack matches is higher for lower matching depths, and therefore Dimmunix has to check more often whether signatures in history are instantiated.

Breakdown of overhead: Having seen the impact of the number of threads, the history

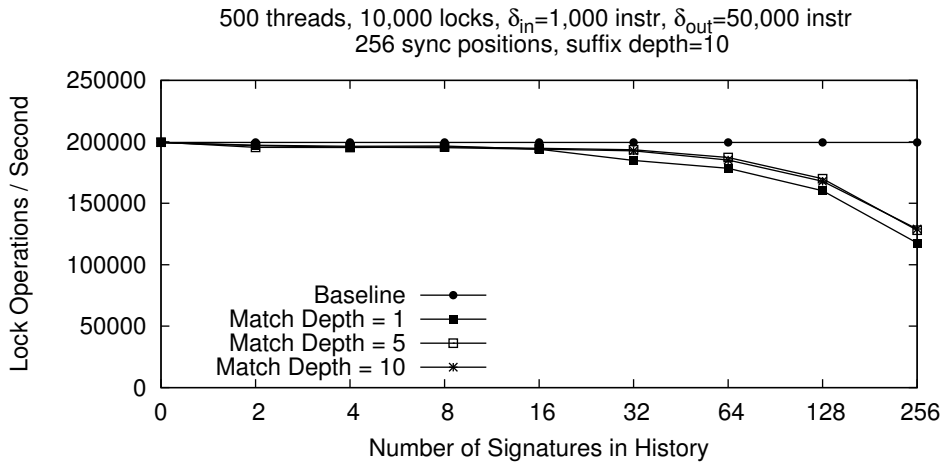

Figure 10.3: Lock throughput as a function of history size and matching depth.

size, and the 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 synchronization throughput. First, we measured the overhead introduced by the instrumentation, which includes the inline call stack matching. Then, we added the data structure lookups and updates performed in the avoidance code. Finally, we ran full Dimmunix, with the signature matching code enabled. Yielding decisions are still ignored, in order to measure only the computational overhead.

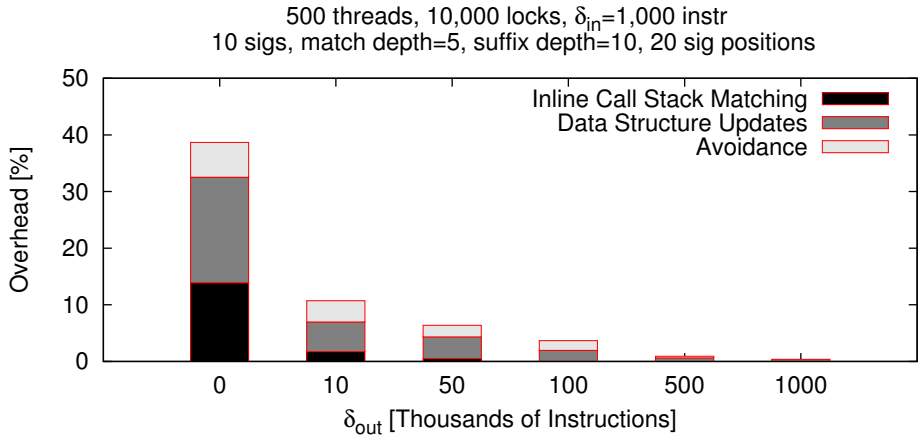

Figure 10.4: Breakdown of overhead.

The results are shown in Figure 10.4. For the most lock-intensive applications, the overhead is almost equally shared between the avoidance code and the data structure updates/look-ups. For less lock-intensive applications, most of the overhead is in the data structure updates/look-ups. The avoidance overhead is higher for small δ_{out} values, because call stacks are matched more often, and therefore signature matching code is invoked more frequently. The inline call stack matching incurs a noticeable overhead for $\delta_{out} = 0$.

The signatures do not share call frames; if they would, the overhead due to inline call stack matching could be substantially higher, as it is for real applications (§10.1.2). We chose to have no frame sharing between signatures, because we wanted to avoid making the inline call stack matching a bottleneck, in order to measure also the overhead due to the other computations, i.e., data structure look-ups/updates and signature matching.

10.1.4 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; FPs can arise when signatures are matched too shallowly, or when the lock order 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.

To measure the effect of false positives (FPs), we used the microbenchmark described in §10.1.3. All deadlock signatures in the microbenchmark are synthetic, i.e., there are no true positives. Therefore, the yielding overhead gives us exactly the overhead induced by FPs. Although it is not realistic to have no true positives (TPs), the number of TPs is not relevant when measuring the effect of FPs. The yielding overhead is the difference between Dimmunix's overhead with the yielding enabled and the overhead with the yielding disabled. Calibration of matching precision is turned off.

Figure 10.5 shows the results—as the matching precision is increased, the overhead induced by FPs decreases. For a matching depth of 1, the overhead due to FPs is 121%. There are hardly any yields for depths greater than 8, because the probability of instantiating signatures is low; therefore, the overhead due to FPs is negligible. However, Dimmunix's overhead is higher for matching depths ≤ 5 (i.e., 13.7–32.3%) compared to matching depths > 5 (i.e., 4.8–10.6%), because the call stacks are matched more often, and therefore the signature matching code is invoked more frequently.

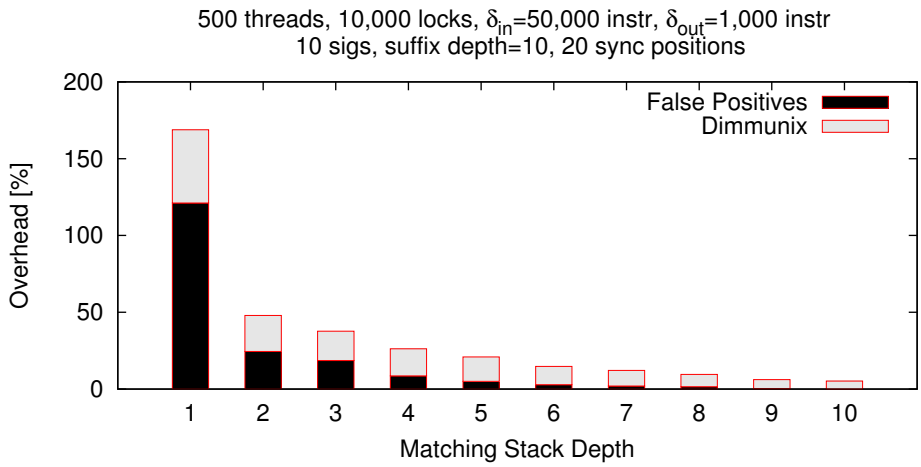

Figure 10.5: Overhead due to false positives.

10.1.5 Optimizations

To evaluate the effectiveness of the optimizations described in §8, we used the microbenchmark described in §10.1.3.

Figure 10.6 shows that the selective program instrumentation (§8.1) is very effective; up to 64 signatures, Dimmunix with selective instrumentation performs much better than with full instrumentation, i.e., 0–6.1% vs. 5.2–16.4%. According to our microbenchmark, having n signatures in the history means instrumenting n synchronization positions out of the total 256 positions. With an empty history, there is no overhead if Dimmunix uses selective instrumentation; with full instrumentation, the overhead is already 5.2%, which is comparable to selective instrumentation with 64 signatures. If Dimmunix uses selective

instrumentation, the overhead is low up to 64 signatures (i.e., 0–6.1%); for more signatures, it increases rapidly until it reaches the overhead of Dimmunix with full instrumentation (i.e., 6.1–35.6%). The conclusion we can draw is that the overhead remains small as long as there are few deadlock signatures on the critical path.

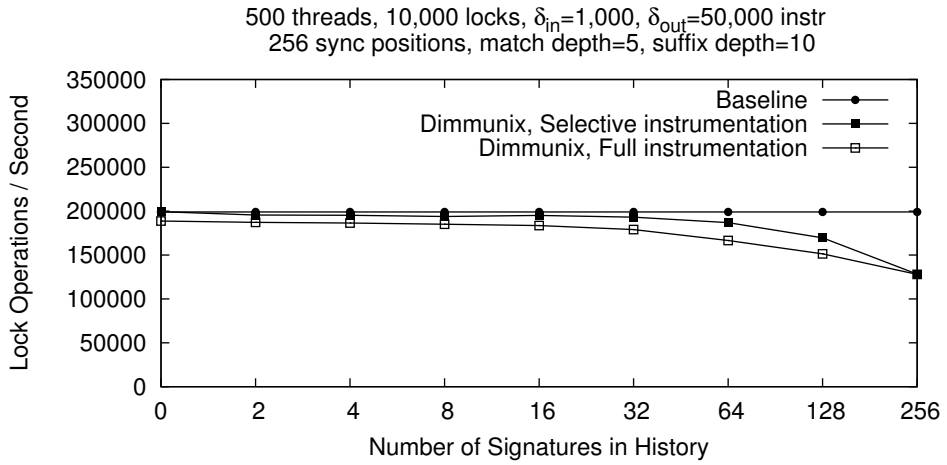

Figure 10.6: Benefit of selective program instrumentation.

Figure 10.7 shows the benefit of increasing the matching precision as FPs are encountered. The experiment is similar to the one depicted in Figure 10.5, with the only difference that now Dimmunix is configured to dynamically calibrate the matching depths. If we compare the two figures, the benefit of dynamically increasing the matching precision is evident; the overhead becomes acceptable even for an initial matching depth of 1, i.e., 5.7%. Without dynamic matching depth calibration, the overhead due to FPs is 121% for matching depth 1.

Figure 10.8 shows the benefit of exploiting the escape branches to stop the avoidance earlier. In this experiment Dimmunix runs with the yielding enabled, to show the benefit of stopping the avoidance earlier. The benefit is substantial if (1) the number of instructions on the escape paths that bypass the deadlock (i.e., δ_{escape}) is substantially larger than the number of instructions in the critical section preceding the escape branches (i.e., δ_{in}), (2) signatures are instantiated very often, i.e., the matching depth is low and δ_{out} is small, and (3) there are no FPs, even for shallow matching depths. Therefore, in this experiment we configure $\delta_{in} = \delta_{out} = 1,000$ and the matching depth to 1; we also disable the matching

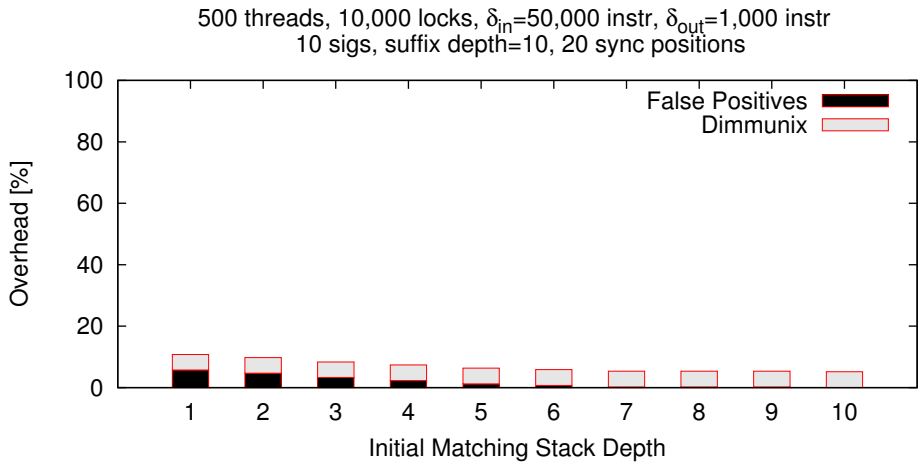

Figure 10.7: Benefit of dynamic matching depth calibration.

depth calibration, in order to simulate the scenario in which there are no FPs. For $\delta_{escape} = 0$, there is no benefit in exploiting the escape branches. For $\delta_{escape} = 10,000$ (respectively 50,000 and 100,000), the overhead is 77% (respectively 67% and 50%) without escaping, compared to 63% (respectively 40% and 32%) with escaping.

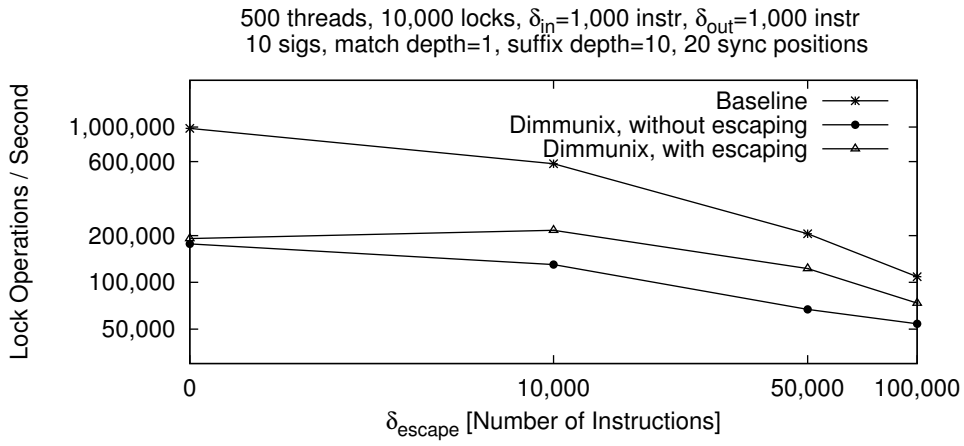

Figure 10.8: Benefit of exploiting the escape branches.

Figure 10.9 shows that inlining the call stack matching considerably reduces the 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%.

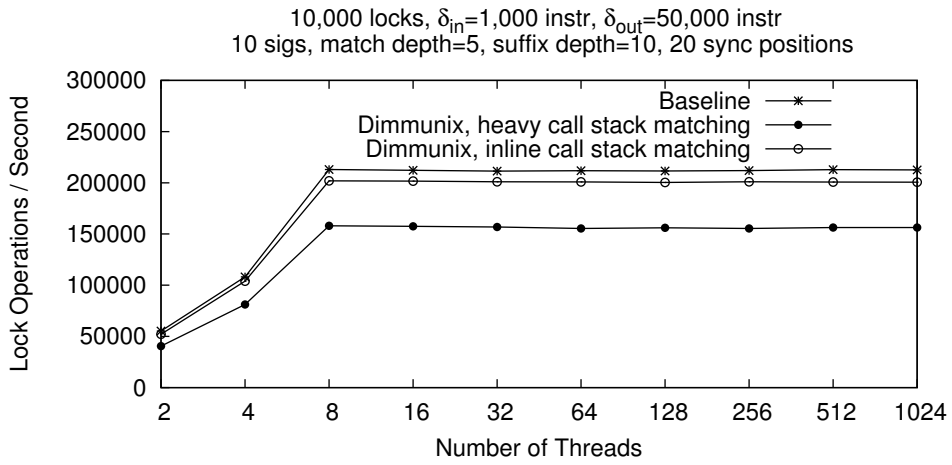

Figure 10.9: Benefit of inlining the call stack matching.

10.2 Java Dimmunix—Non-Mutex Deadlocks

In this section, we evaluate the part of Java Dimmunix that deals with non-mutex deadlocks, i.e., hybrid deadlocks, initialization deadlocks, external deadlocks, and blocked notifications.

We profiled the non-mutex synchronization behavior of real Java applications. More precisely, we measured the throughput of *Object.wait()*, *Object.notify/notifyAll()*, *ReentrantReadWriteLock.lock()*, *Semaphore.acquire()*, and *FileChannel.lock()* calls. We also counted the number of threads executing these calls, the number of objects on which these calls are performed, and the number of program positions where they execute. We did not count the calls to custom synchronization routines; we focused on synchronization primitives provided by the JDK.

We illustrate the profiling results in Table 10.5. We used the same tests as for Table 10.3, to be able to compare the results. There were no calls to *ReentrantReadWriteLock.lock()*, *Semaphore.acquire()*, or *FileChannel.lock()* in any of the tests, i.e., no read-write lock, semaphore, or file lock synchronization primitives provided by the JDK were used. All the tests except JDBCBench used the condition variables provided by the JDK. By comparing the profiling results from Table 10.5 to the results from Table 10.3, we notice that the throughput of *Object.wait()* and *Object.notify/notifyAll()* calls is much smaller than the

throughput of mutex lock acquisitions. This empirically shows once more that the mutex locks are the most encountered synchronization construct in real Java applications.

Characteristics	System, Test Scenario				
	JBoss, RUBiS	MySQL JDBC, JDBCBench	Eclipse, Startup + Shutdown	Limewire, Upload test	Vuze, Startup + Shutdown
Object.wait()					
number of calls/sec	0.1	0	5.3	9.6	57.7
number of threads	13	0	104	53	41
number of objects	12	0	96	152	98
number of call sites	9	0	12	4	4
Object.notify/notifyAll()					
number of calls/sec	57.2	0	33.7	1,058	34.3
number of threads	4	0	68	2	41
number of objects	15	0	629	29,432	418
number of call sites	14	0	31	6	4

Table 10.5: Non-mutex synchronization in real Java applications.

We do not need to evaluate the efficiency of Dimmunix for initialization deadlocks. On the critical path, Dimmunix only needs to check whether the current lock statement belongs to a signature of an initialization deadlock; this is similar to the signature look-up involved in avoiding hybrid deadlocks. The rest of the computations are cheaper than the ones done for mutex deadlocks or hybrid deadlocks.

Dimmunix successfully detected and avoided an initialization deadlock equivalent to a deadlock involving Java's *LogManager* class; the bug id is 4994705 in the Oracle's bug database. More precisely, we ran the test case provided in the report, which simulates the *LogManager* deadlock. Dimmunix successfully detected the deadlock and avoided it in the subsequent runs of the test case.

Figure 10.10 shows that the Dimmunix prototype is efficient when handling hybrid deadlocks, for up to 256 threads. The microbenchmark performs synchronizations on 1,000 *ReentrantLock* objects, 1,000 *ReentrantReadWriteLock* objects, and 1,000 *Semaphore* objects. The microbenchmark has 2–1,024 threads executing synchronizations (in a loop) at 20 program positions. We use a history of 10 synthetic deadlock signatures covering 10 synchronization positions. In a loop iteration, a thread executes δ_{in}=1,000 incrementation instructions inside the critical section and δ_{out}=1,000,000 outside. As shown in Figure 10.10, the performance overhead incurred by Dimmunix stays within acceptable limits

(i.e., 0–7.7%) for up to 128 threads. For 256–1,024 threads, the performance overhead increases rapidly from 14.6% to 80%. The reason is that we use a global lock to ensure the thread safety of Dimmunix's computations, and the synchronization on the global lock becomes a bottleneck. For instance, for 512–1,024 threads the overhead factor is 1.5–5x. The part of Java Dimmunix handling hybrid deadlocks can be significantly improved if we remove the global lock and use lock-free data structures, as we did for mutex deadlocks.

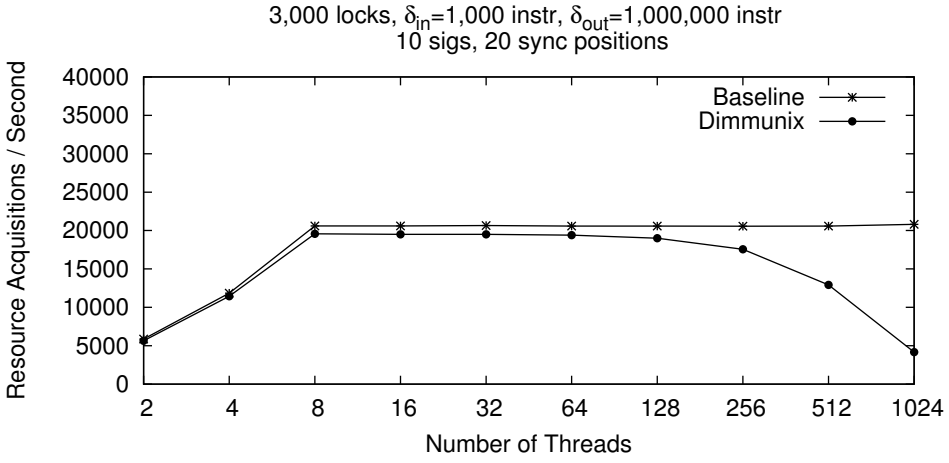

Figure 10.10: Dimmunix's performance for hybrid deadlocks.

Figure 10.11 shows that Dimmunix efficiently handles external deadlocks. The microbenchmark runs 2–1,024 threads, with 1 file lock per thread. Every thread executes synchronizations (in a loop) at 20 program positions. In a loop iteration, a thread executes δ_{in}=1,000 incrementation instructions inside the critical section and δ_{out}=1,000,000 outside. The deadlock history has 10 synthetic signatures covering 10 synchronization positions. When it uses offline deadlock detection, Dimmunix incurs a low performance overhead, i.e., 5.6–10%. When it uses online deadlock detection, the overhead factor is 16–453x, which makes an application unusable. This shows that making the deadlock detection offline substantially improves Dimmunix's performance.

Figure 10.12 shows that Dimmunix efficiently handles blocked notifications if the wait condition always holds. Remember that Dimmunix conservatively makes the waiter thread yield before it executes a critical section containing a wait call previously involved in a

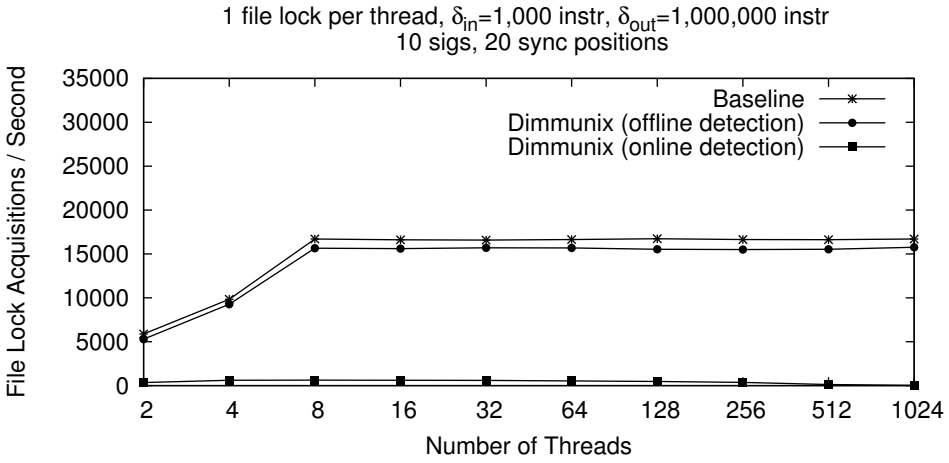

Figure 10.11: Dimmunix's performance for external deadlocks.

blocked notification; the yield lasts until a notification is about to arrive. Therefore, whenever the waiter thread does not "intend" to wait, Dimmunix will needlessly delay the thread. However, if there is no wait condition involved, Dimmunix only postpones the wait call until a notification is about to arrive.

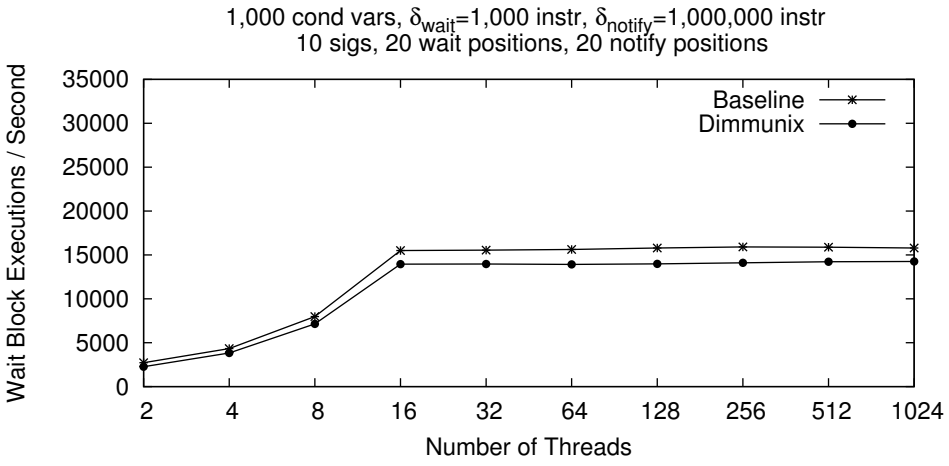

Figure 10.12: Dimmunix's performance for blocked notifications, when the wait condition always holds.

The microbenchmark we used runs 2–1,024 threads, half waiters and half notifiers. The threads execute wait calls (respectively notify calls) in a loop, at random program

positions from 20 positions of wait calls (respectively 20 positions of notify calls). We use a history with 10 synthetic blocked notification signatures covering 10 wait positions. A waiter thread executes in each loop iteration a wait call on a random object from a pool of 1,000 objects; right before waiting on an object x, the waiter thread adds x to a wait queue. A notifier thread pops in each loop iteration an element y from the wait queue and calls *y.notify()*. A waiter (respectively notifier) thread executes $\delta_{wait} = 1,000$ (respectively $\delta_{notify} = 1,000,000$) incrementation instructions between consecutive loop iterations. We measure the performance in terms of number of executions of synchronized blocks containing wait calls. Figure 10.12 shows that the performance overhead incurred by Dimmunix is acceptable, i.e., 9.7–16.4%. Most of the overhead is due to synchronization mechanism used in the blocked notification avoidance.

When the wait condition does not always hold, Dimmunix can slow down the application considerably, as we show in Figure 10.13. We fix the number of threads to 500, and vary the probability for a wait condition to hold from 0.0001 to 1. The lower the wait probability, the more substantial the slowdown is. If the wait probability is 0.0001–0.01, the slowdown incurred by Dimmunix is 2.4–6.8x. If the wait probability is ≥ 0.1, the performance overhead is acceptable, i.e., 9.6–18.4%. This shows that even for relatively low wait probabilities (e.g., 0.1) the performance overhead is acceptable.

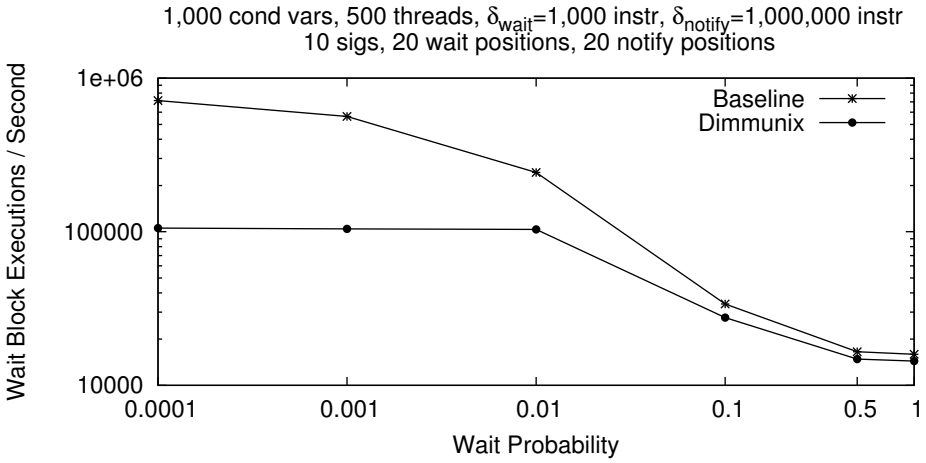

Figure 10.13: Dimmunix's performance for blocked notifications, when the wait condition holds with a given probability.

Note that the overheads incurred by the five Java Dimmunix modules responsible for mutex deadlocks, hybrid deadlocks, initialization deadlocks, external deadlocks, and blocked notifications do not add up if the modules are used simultaneously. The five modules have few interception points in common. The modules responsible for mutex deadlocks and hybrid deadlocks both intercept the calls to *ReentrantLock.lock()*. However, the mechanism for retrieving the program position p where the call executes and searching for a signature containing p is similar in the two modules, and therefore can be shared. Besides, the *ReentrantLock* mutexes are not as heavily used as the synchronized blocks. The module responsible for external deadlocks does not share any interception points with other modules.

The modules handling mutex deadlocks, initialization deadlocks, and blocked notifications need to intercept synchronized blocks. For blocked notifications, Dimmunix needs to intercept synchronized blocks wrapping wait or notify(All) calls and synchronized blocks that were previously involved in blocked notifications. For mutex deadlocks and initialization deadlocks, Dimmunix needs to intercept synchronized blocks previously involved in deadlocks. However, the three modules usually need to intercept disjoint synchronized blocks. Dimmunix does not handle mutex (or initialization) deadlocks that involve synchronized blocks wrapping wait or notify(All) calls (§6.4); therefore, these synchronized blocks are only intercepted by the module responsible for blocked notifications. These modules would need to intercept the same synchronized block B only if B was involved in multiple types of deadlocks. Since Dimmunix ignores signatures of mutex (or initialization) deadlocks that involve synchronized blocks containing wait or notify(All) calls (§6.4), a synchronized block involved in a mutex (respectively initialization) deadlock and a blocked notification is only intercepted by the module handling blocked notifications. Therefore, a synchronized block can be shared only by mutex deadlocks and initialization deadlocks. Since initialization deadlocks are cheap to avoid, the overhead added due to intercepting such synchronized blocks is negligible.

10.3 POSIX Threads Dimmunix

POSIX Threads Dimmunix is effective against real deadlocks encountered in real applications, as we illustrate in Table 10.6. Dimmunix successfully detected and avoided the deadlocks, in these applications. The meanings of the columns are the same as the ones in Table 10.1.

System	Bug #	Deadlock Between ...	Yields	Signatures
MySQL 6.0.4	37080	INSERT and TRUNCATE in two different threads	1	1
SQLite 3.3.0	1672	Deadlock in the custom recursive lock implementation	1	1
HawkNL 1.6b3	n/a	nlShutdown() called concurrently with nlClose()	10	1

Table 10.6: Reported deadlock bugs avoided by Dimmunix in real C/C++ applications.

We measured the performance of FreeBSD pthreads Dimmunix on a microbenchmark. The microbenchmark simulates various throughputs of synchronization operations, number of threads, number of locks, and number of signatures.

Figure 10.14 shows how synchronization throughput (in terms of lock operations) varies with the number of threads. We chose the delays inside and outside the critical sections 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. The microbenchmark performs around 8,300 lock operations per second, which is comparable to the highest synchronization throughput we encountered in the C/C++ applications we studied.

We show in Figure 10.15 the performance overhead introduced by varying history size from 2 to 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.

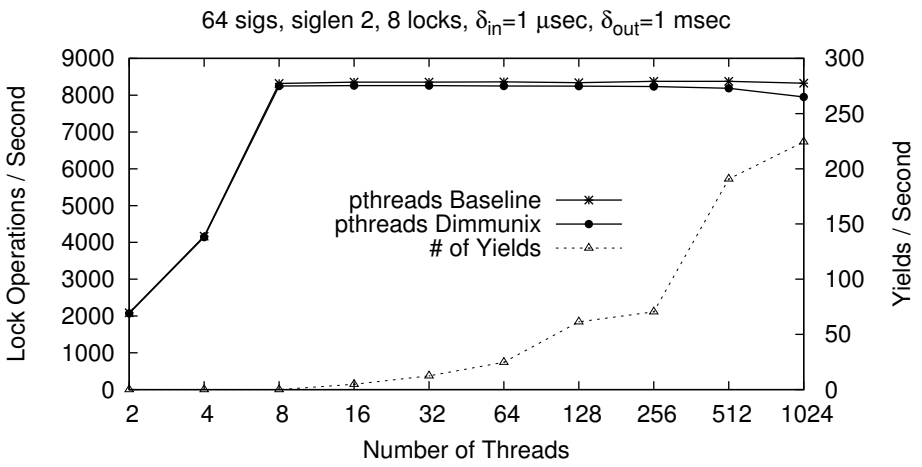

Figure 10.14: Dimmunix microbenchmark lock throughput as a function of number of threads. Overhead is 0.6% to 4.5% for FreeBSD pthreads.

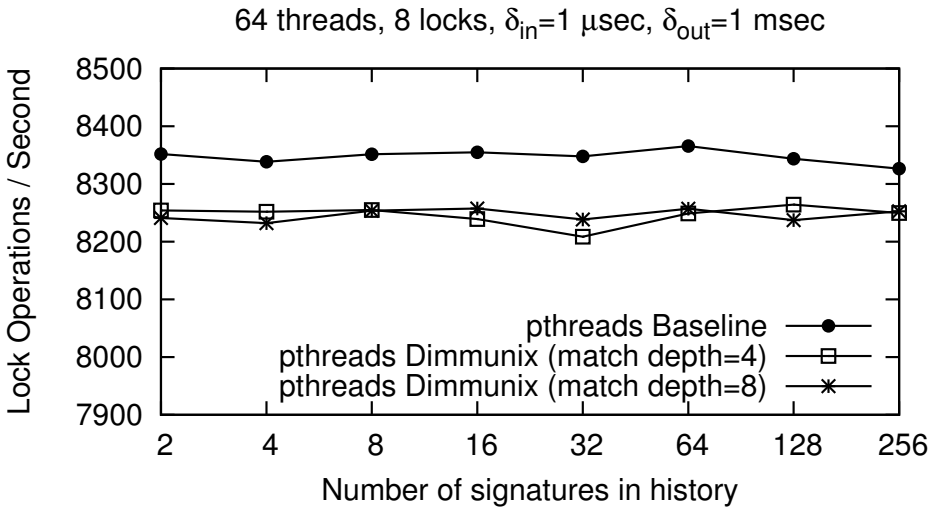

Figure 10.15: Lock throughput as a function of history size and matching depth for pthreads.

10.4 Android Dimmunix

We installed a Dimmunix-enabled Android 2.2 OS on a Nexus One phone, equipped with a 1-core 1GHz CPU, and 512 MB of RAM memory. While using the applications installed on the phone, we noticed no slowdown, compared to the vanilla Android 2.2 OS installation.

We reproduced a real deadlock involving Android's NotificationManagerService and StatusBarService classes (issue id: 7986), which froze the entire phone's interface. We made a small Android application in which one thread issues a notification, and a second thread expands the status bar, in the same time. The two threads called concurrently the methods *NotificationManagerService.enqueueNotificationWithTag* and *StatusBarService$H.handleMessage*, which made the two services deadlock. This deadlock made the whole phone's interface hang. Dimmunix detected the deadlock and saved its signature in the persistent history. After rebooting the phone, Dimmunix successfully avoided any reoccurrence of the deadlock.

We profiled the synchronization behavior of 8 Android applications, with Dimmunix disabled. The results are shown in Table 10.7. For each application, we profiled its synchronization behavior during several minutes of intensive usage; then, we selected the 30 seconds interval with the highest average synchronization throughput. In these time intervals, the 8 applications perform 309–1952 synchronizations per second, using 23–119 threads.

Table 10.7: Statistics about various Android applications.

Application	Threads	Syncs/sec	Memory consumption	
			Dimmunix: 52%	Vanilla: 50%
Email	46	1,952	15.8 MB	15.0 MB
Browser	61	1,411	38.9 MB	37.9 MB
Maps	119	1,143	23.7 MB	22.9 MB
Market	78	891	17.9 MB	17.3 MB
Calendar	26	815	14.4 MB	14.0 MB
Talk	33	527	11.2 MB	10.7 MB
Angry Birds	23	325	29.7 MB	29.3 MB
Camera	26	309	11.8 MB	11.4 MB

To measure the performance overhead, we reproduced in a microbenchmark the most intensive synchronization behavior that we observed in the 8 applications we studied. The microbenchmark runs 2–512 threads, that execute synchronized blocks on random lock objects, to avoid contention; lock contention has the undesired effect of hiding the performance overhead. We do not use sleeps, because they hide the performance overhead; we use busy waits instead, to simulate computation inside and outside the critical sections.

We use a history of 64–256 synthetic signatures, to simulate the scenario in which many synchronization statements are involved in deadlock bugs. The microbenchmark executes 1738–1756 synchronizations per second, with Dimmunix disabled; this is similar to the synchronization throughput of the most lock-intensive applications that we studied (i.e., Email and Browser). On the Dimmunix-enabled Android OS, the microbenchmark runs 1657–1681 synchronizations per second. This means 4–5% performance overhead. Most of the overhead is due to the call stack retrieval, i.e., the calls to *dvmGetCallStack*.

We also measured the power consumption after an intensive usage. With and without Dimmunix, Android OS reported that the Android applications and the OS are responsible for 14% of the power consumption. Therefore, Dimmunix does not increase the power consumption.

We evaluated the memory overhead incurred by Android Dimmunix; the results are shown in Table 10.7. Dimmunix incurs 1.3–5.3% memory overhead in the 8 applications we studied. Overall, for all the running applications, the memory overhead is 4%; the overall memory consumption is 52% for the Dimmunix-enabled Android OS, and 50% for the vanilla Android OS.

10.5 Communix

In this section, we first evaluate the performance of Communix (§10.5.1), then we evaluate the impact DoS attacks can have on Java applications running Dimmunix (§10.5.2). Finally, we estimate the time it takes for an application to achieve full deadlock protection with Communix, compared to using Dimmunix alone (§10.5.3).

The experiments were run on machines with two 4-core Intel Xeon 2GHz processors each, 20 GB of memory, running Ubuntu Linux 10.04.

10.5.1 Performance

In this section, we first evaluate the performance of the Communix server, then the performance of the whole signature distribution in an end-to-end setting. Then, we evaluate the performance of the client-side signature validation plus the signature generalization.

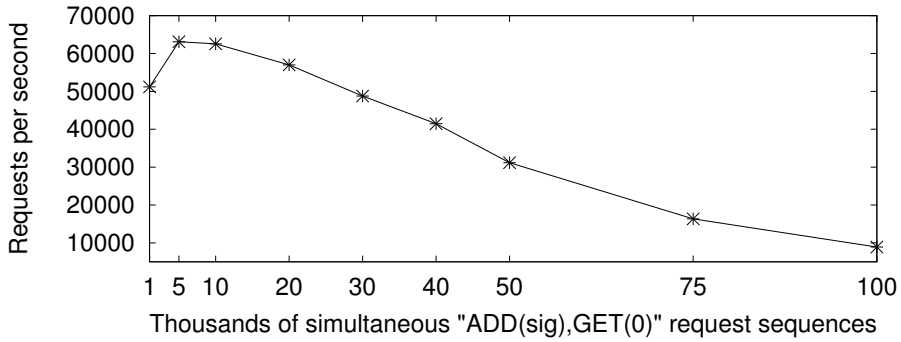

Figure 10.16: The performance of the Communix server.

Since the signature generalization and client-side signature validation are both performed by the Communix agent at application startup, we decided to evaluate them together. Finally, we measure the time it takes for the Communix agent to find the nested synchronized blocks/methods; the time it takes to compute the hashes of the loaded classes is negligible compared to the time it takes to perform the nesting analysis.

The server processes two types of requests: an *ADD(sig)* request that means "add signature *sig* to the database", and a *GET(k)* request that means "send me the signatures from the database starting from index *k*". Normally, a client having a local repository with *n* signatures sends *GET(n+1)* requests to the server to retrieve the new signatures. We wanted to evaluate worst-case scenarios, therefore we use only *GET(0)* requests in our measurements, which means that the server is always asked to send all its signatures.

To evaluate the server's performance, we invoke the request processing routines from 1,000–100,000 simultaneous threads. This test measures the efficiency of the server's computations, i.e., adding new random signatures to the database (including the server-side signature validation) and iterating through the entire database. Figure 10.16 shows that the server scales well up to 30,000 simultaneous "*ADD(sig),GET(0)*" sequences of requests. At its peak, the server processes 63,000 requests per second.

We evaluate the performance of the signature distribution in an end-to-end setting. On one machine we ran the Communix server, and on another machine we ran 10–200 client threads that send 10 "*ADD(sig),GET(0)*" sequences of requests each. Figure 10.17 shows that the signature distribution scales well up to 30 client threads, i.e., 300 simultaneous

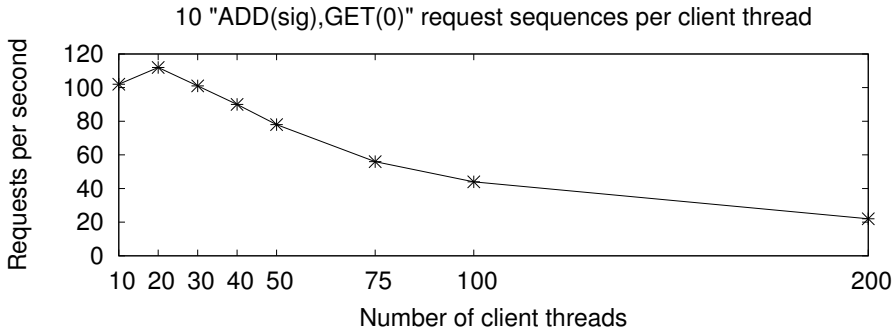

Figure 10.17: The performance of the signature distribution.

"*ADD(sig),GET(0)*" sequences of requests. However, the throughput (i.e., requests served per second) is up to two orders of magnitude lower compared to Figure 10.16. The explanation is that the network communication between the server and the client threads becomes a bottleneck. The size of a signature is 1.7 KB. If there are N client threads and each thread sent on average k "*ADD(sig),GET(0)*" request sequences to the server, the server has to send $(k+1/2) \times N^2 \times 1.7$ KB of data to the N clients, on average, to serve the next round of *GET(0)* requests. If $N = 200$, the server has to send in the 10th round approximately 630 MB of data to the 200 clients. To summarize, a server with one network card cannot distribute signatures fast if multiple clients ask simultaneously for a large number of signatures.

As shown in Figure 10.17, a client thread receives 20–110 replies per second to "*ADD(sig), GET(0)*" request sequences, from the Communix server. Therefore, it takes 9–50 milliseconds to send the two requests to the server and get the replies. However, the latency of the signature distribution is up to 1 day, because the Communix client downloads the new signatures from the Communix server only once a day.

We evaluate the Communix agent on large Java applications, i.e., JBoss, Limewire, and Vuze. JBoss is a well-known Java application server, while Limewire and Vuze are well-known peer-to-peer file sharing applications. For each application, we measure the time it takes to start and immediately shut down. In Figure 10.18, we show the performance of the computations performed at startup by the agent, i.e., client-side signature validation and

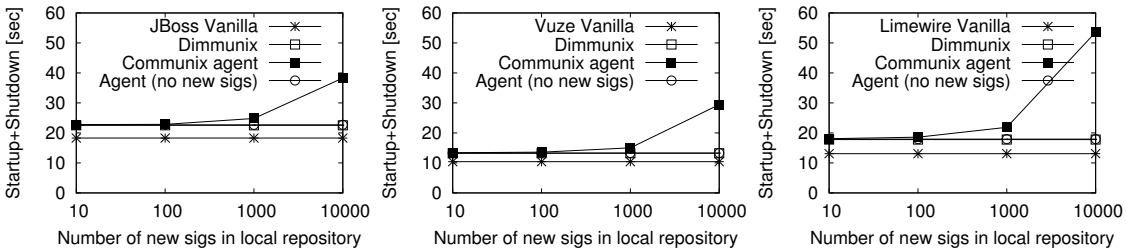

Figure 10.18: The performance of client-side computations, i.e., client-side signature validation and signature generalization.

signature generalization. For up to 1,000 new signatures in the local repository, the Communix agent incurs a startup delay of up to 2–3 seconds, i.e., 11–16% startup slowdown.

Table 10.8: Statistics about various Java applications, and the performance of the nesting analysis.

App	Size (LOC)	Sync bl/meths	Explicit sync ops	Nested (Analyzed)	Nesting check (sec)
JBoss	636,895	1,898	104	249 (844)	114
Limewire	595,623	1,435	189	277 (781)	122
Vuze	476,702	3,653	14	120 (432)	50

In Table 10.8, we show the efficiency of the static detection of nested synchronized blocks/methods and some statistics we collected about the three applications, i.e., size in lines of code (LOC), number of synchronized blocks/methods, number of explicit lock/unlock operations (i.e., calls to *ReentrantLock.lock/unlock()*), and the number of nested synchronized blocks/methods that the nesting analysis reports. The Communix agent could analyze only 11–54% of the synchronized blocks/methods. For the rest of the synchronized blocks/methods, the Soot static analysis framework could not retrieve enough information for the nesting analysis (i.e., it could not retrieve the CFGs of some of the methods). Table 10.8 shows that it takes 50–122 seconds to analyze 432–844 synchronized blocks/methods. The nesting analysis is performed at shutdown, first time the application runs, and each time new classes (w.r.t. the previous run) are loaded. Therefore, the analysis is performed only for the first couple of runs. Moreover, since the analysis is performed at shutdown, the delay is not bothersome for the user, if the user does not intend to restart the application soon.

10.5.2 Impact of Denial of Service Attacks

The attackers have only one way to exploit Dimmunix, to slow down a Java application: they can send signatures with outer call stacks of depth 5 which cover all the nested synchronized blocks/methods that are on the critical path, in order to maximize the amount of thread serialization in applications running Dimmunix. If there is already a signature S in the deadlock history that can be merged with a malicious signature S', signature S' will replace S in the history, by exploiting the generalization mechanism.

Table 10.9 shows that attackers providing malicious deadlock signatures can cause only 7–29% performance overhead in the studied real applications running with Dimmunix. The tests run with 20 deadlock signatures in the history, with outer call stacks of depth 5. These outer calls are on the critical path, i.e., more than 99% of the nested synchronized blocks/methods are executed with these call stacks. In this worst-case scenario, the performance overhead incurred by Dimmunix is 7–29%, which is acceptable for general-purpose applications. If none of the signatures is on the critical path, the performance overhead incurred by Dimmunix is negligible (i.e., $< 2\%$). For outer call stacks of depth 1, the performance overhead is considerable (i.e., $> 100\%$), for some of the applications we studied. However, this situation is avoided, because the Communix agent does not accept incoming signatures with outer call stacks of depth < 5. Therefore, Communix successfully contains DoS attacks.

Table 10.9: Worst-case overhead incurred while under a DoS attack.

Application	**Benchmark/Test**	**Overhead**
JBoss	RUBiS	28.7%
MySQL JDBC	JDBCBench	27.8%
Eclipse	Startup + Shutdown	19.5%
Limewire	Upload test	6.9%
Vuze	Startup + Shutdown	8.2%

Making it hard for a user to obtain multiple encrypted ids from the Communix server, together with restricting the server to process only up to 10 signatures per day for the same user id, protects the server and the clients against flooding with fake signatures. Assuming 100 attackers manage to obtain 5 ids each from the server, and they keep sending fake signatures to the server, the attackers could make the server process and add to its database

only up to $100 * 5 * 10 = 5,000$ signatures in 1 day. Assuming the worst case, i.e., the 5,000 signatures are sent simultaneously by the 100 attackers, the server can process the signatures in 1 second, the Communix client can download them in a few minutes, and the agent can process them in 10–15 seconds.

10.5.3 Time to Achieve Full Protection

As we mentioned in §7.2.4, it may take a long time for a single user to experience all the deadlocks of an application and all the manifestations of these deadlocks. Therefore, it may take a long time until Dimmunix alone can provide full protection against deadlocks.

If there are many users of an application A, Communix can considerably reduce the time it takes for A to be deadlock-free. The time it takes for Communix to provide full protection against deadlocks for application A is inversely proportional to the number N_u of users that run A in different ways. If there are N_d possible deadlock manifestations in A and it takes on average t days for a user to experience one manifestation, A will be deadlock-free in roughly $t * N_d$ days, if Dimmunix alone is used. If Communix is used, all the users of A will have A deadlock-free in roughly $t * N_d/N_u$ days. The larger N_u, the higher the gain that Communix brings.

The estimate we made here is purely theoretical. A real evaluation is possible only if Communix is deployed in the field and statistics are collected after a considerable period of usage (e.g., months) from many (e.g., thousands) users.

Chapter 11

Conclusion

In this thesis, we presented Dimmunix, a system that enables applications to defend themselves against deadlocks. Dimmunix enables real applications to automatically achieve immunity against deadlock bugs involving mutex locks, semaphores, read-write locks, class initialization, and external synchronization, with no user intervention, and without changing the semantics of the applications. Dimmunix is transparent and non-intrusive. Our Dimmunix prototypes provide deadlock immunity to Java programs, C/C++ programs using the POSIX Threads library, and the Android OS. We also implemented a framework called Communix, which enables users connected to the Internet to share deadlock signatures, in order to improve their protection against deadlocks. We evaluated these implementations—they are effective against deadlocks reported in real applications and run efficiently on synchronization-intensive applications with millions of lines of code and hundreds of threads.

Bibliography

C/C++ file locks.
http://linux.die.net/man/2/flock.

Java barriers.
http://download.oracle.com/javase/6/docs/api/java/util/concurrent
/CyclicBarrier.html, a.

Java condition variables.
http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks
/Condition.html, b.

Java file locks.
http://download.oracle.com/javase/1.5.0/docs/api/java/nio/channels
/FileChannel.html, c.

Java thread join.
http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html, d.

Java reentrant locks.
http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks
/ReentrantLock.html, e.

Java reentrant read-write locks.
http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks
/ReentrantReadWriteLock.html, f.

Java semaphores.
http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent
/Semaphore.html, g.

Java synchronized blocks.
http://download.oracle.com/javase/tutorial/essential/concurrency/locksync.html, h.

Java built-in condition variables.
http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html#wait()
http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html#notify()
http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html#notifyAll(), i.

POSIX threads read-write locks.
http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_rwlock_rdlock.html
http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_rwlock_wrlock.html
http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_rwlock_unlock.html.

POSIX threads barriers.
http://pubs.opengroup.org/onlinepubs/009695399/functions
/pthread_barrier_wait.html, a.

POSIX semaphores.
http://pubs.opengroup.org/onlinepubs/007908799/xsh/sem_wait.html
http://pubs.opengroup.org/onlinepubs/007908799/xsh/sem_post.html, b.

POSIX threads condition variables.
http://www.opengroup.org/onlinepubs/007908799/xsh/pthread_cond_wait.html
http://www.opengroup.org/onlinepubs/007908799/xsh/pthread_cond_signal.html, a.

POSIX threads join.
http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_join.html, b.

POSIX threads mutex locks.
http://www.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_lock.html, c.

Scala actors. http://www.scala-lang.org/node/242.

AspectJ. http://www.eclipse.org/aspectj.

Java Messaging Service.
http://java.sun.com/developer/technicalArticles/Ecommerce/jms/index.html, 2004.

Pablo Boronat and Vicent Cholvi. A transformation to provide deadlock-free programs. In *Intl. Conf. on Computational Science*, 2003.

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

Eric Bruneton. *ASM 3.0 A Java bytecode engineering library*. 2007.

George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – a technique for cheap recovery. In *Symp. on Operating Sys. Design and Implem.*, 2004.

Lee Chew and David Lie. Kivati: fast detection and prevention of atomicity violations. In *ACM EuroSys European Conf. on Computer Systems*, 2010.

Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. In *Symp. on Operating Systems Principles*, 2005.

Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *Symp. on Operating Systems Principles*, 2007.

Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Symp. on Operating Systems Principles*, 2003.

Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Intl. Symp. on Formal Methods Europe*, 2001.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Conf. on Programming Language Design and Implem.*, 2002.

Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Intl. Conf. on Dependable Systems and Networks*, 2010.

Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Intl. Symp. on Computer Architecture*, 1993.

C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10), 1974.

Java PathFinder. Java PathFinder. http://javapathfinder.sourceforge.net, 2007.

JDBCBench. JDBCBench. http://mmmysql.sourceforge.net/performance.

Eric Koskinen and Mauric Herlihy. Dreadlocks: Efficient deadlock detection. In *ACM Symp. on Parallelism in Algorithms and Architectures*, 2008.

Bohuslav Krena, Zdenek Letko, Rachel Tzoref, Shmuel Ur, and Tomás Vojnar. Healing data races on-the-fly. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2007.

Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2), 1980.

Zdenek Letko, Tomás Vojnar, and Bohuslav Krena. Atomrace: Data race and atomicity violation detector and healer. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, 2008.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 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.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

Yarden Nir-Buchbinder, Rachel Tzoref, and Shmuel Ur. Deadlocks: From exhibiting to healing. In *Workshop on Runtime Verification*, 2008.

Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Symp. on Operating Sys. Design and Implem.*, 2010.

Hari K. Pyla and Srinidhi Varadarajan. Avoiding deadlock avoidance. In *Proceedings of the 19th international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating bugs as allergies – a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3), 2007.

RUBiS. RUBiS. http://rubis.objectweb.org, 2007.

Xiang Song, Haibo Chen, and Binyu Zang. Why software hangs and what can be done with it. In *Intl. Conf. on Dependable Systems and Networks*, 2010.

Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - A Java optimization framework. In *Conf. of the Centre for Advanced Studies on Collaborative Research*, 1999.

Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for Java libraries. In *European Conf. on Object-Oriented Programming*, 2005.

Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing races in live applications with execution filters. In *Symp. on Operating Sys. Design and Implem.*, 2010.

Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad-hoc synchronization considered harmful. In *Symp. on Operating Sys. Design and Implem.*, 2010.

Cristian Zamfir and George Candea. Execution synthesis: A technique for automated debugging. In *ACM EuroSys European Conf. on Computer Systems*, 2010.

Fancong Zeng. Pattern-driven deadlock avoidance. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2009.

Fancong Zeng and Richard P. Martin. Ghost locks: Deadlock prevention for Java. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.

HORATIU JULA

École Polytechnique Fédérale de Lausanne
EPFL – IC – DSLAB, Station 14
Building INN, Room 321
1015 Lausanne, Switzerland

horatiu.jula@epfl.ch
Tel. +41 (21) 693-7509
http://people.epfl.ch/horatiu.jula

RESEARCH INTERESTS

Improve the dependability of software systems, while increasing programmer's productivity. Practical solutions for enabling highly concurrent large-scale applications to protect themselves against concurrency bugs, i.e., deadlocks, data races and atomicity violations. Language features and compilation schemes that guarantee freedom of concurrency bugs, with minimal loss of parallelism.

EDUCATION

2011 **Ph.D. in Computer Science, expected in August 2011**
EPFL – Swiss Federal Institute of Technology *(Lausanne, Switzerland)*
Dissertation: "Deadlock Immunity: Enabling General-Purpose Software to Defend Itself Against Deadlocks"
Advisor: Prof. George Candea

2004 **Dipl.Eng. in Computer Science, GPA: 10/10**
Technical University of Cluj-Napoca *(Cluj-Napoca, Romania)*
Thesis: "Abstract State Machine Executable C# Specification"
Advisor: Prof. Robert Stärk

WORK EXPERIENCE

Jun. 2007 – present **Research Assistant in Dependable Systems Lab**
EPFL – Swiss Federal Institute of Technology *(Lausanne, Switzerland)*
Advisor: Prof. George Candea

Dec. 2006 – May 2007 **Research Intern in Dependable Systems Lab**
EPFL – Swiss Federal Institute of Technology *(Lausanne, Switzerland)*
Advisor: Prof. George Candea

Sep. 2004 – Aug. 2006 **Research Assistant in Formal Verification group**
ETHZ – Swiss Federal Institute of Technology *(Zurich, Switzerland)*
Advisor: Prof. Daniel Kröning

Feb. – Jun. 2004 **Research Intern in Formal Specification, Verification and Validation of Systems group**
ETHZ – Swiss Federal Institute of Technology *(Zurich, Switzerland)*
Advisor: Prof. Robert Stärk

2001 – Feb. 2004 **Part-time Programmer**
Advansoft S.R.L. *(Cluj-Napoca, Romania)*

HONORS

- *Valedictorian of my class* (May 2004)
  Conferred by Technical University of Cluj-Napoca to the students having the highest GPA
- *Various prizes at regional and national Physics and Mathemathics contests* (1996-1999)

TEACHING ACTIVITY

- Teaching assistant in Prof. George Candea's Software Engineering course (undergraduate level) – Fall 2008, Fall 2009, Fall 2010
- Teaching assistant in Prof. Daniel Kröning's Digitaltechnik (digital circuit design) course (undergraduate level) – Spring 2005
- Teaching assistant in Prof. Daniel Kröning's Formal Verification course (M.S. level) – Fall 2004

STUDENT ADVISING

- Thomas Rensch, semester project (2010)
- Pinar Tozun, semester project (2009)
- Cristina Basescu, diploma project (2009)
- Silviu Andrica, diploma project (2008)
- Pranav Garg, internship (2008)

PEER-REVIEWED PUBLICATIONS

- Efficiency Optimizations for Deadlock Immunity Runtimes
  Horatiu Jula, Silviu Andrica, George Candea
  International Conference on Runtime Verification (RV), San Francisco, CA, 2011
- Platform-wide Deadlock Immunity for Mobile Phones
  Horatiu Jula, Thomas Rensch, George Candea
  Workshop on Hot Topics in System Dependability (HotDep), Hong Kong, China, 2011
- Communix: A Framework for Collaborative Deadlock Immunity
  Horatiu Jula, Pinar Tozun, George Candea
  International Conference on Dependable Systems and Networks (DSN), Hong Kong, China, 2011
- iProve: A Scalable Approach to Consumer-Verifiable Software Guarantees
  Silviu Andrica, Horatiu Jula, George Candea
  International Conference on Dependable Systems and Networks (DSN), Chicago, IL, 2010
- Deadlock Immunity: Enabling Systems To Defend Against Deadlocks
  Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea
  8th Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, 2008
- A Scalable, Sound, Eventually-Complete Algorithm for Deadlock Immunity
  Horatiu Jula and George Candea
  Workshop on Runtime Verification (RV), Budapest, Hungary, 2008
- Alternative Implementation of the C# Iterator Blocks
  Horatiu Jula
  Journal of Object Technology (JOT), Volume 5, Number 7, 2006
- ASM Semantics for C# 2.0
  Horatiu Jula
  International Workshop on Abstract State Machines (ASM), Paris, France, 2005
- An Executable Specification of C#
  Horatiu Jula, Nicu G. Fruja
  International Workshop on Abstract State Machines (ASM), Paris, France, 2005

CURRENT PROJECTS

Dimmunix – http://dimmunix.epfl.ch

Dimmunix enables general-purpose applications to defend themselves against deadlock bugs, i.e., avoid deadlocks that they previously encountered. Dimmunix is implemented for Java, POSIX Threads, and Android OS. POSIX Threads and Android Dimmunix currently provide immunity against deadlocks involving mutex locks. Android Dimmunix is implemented within the Dalvik VM, which runs all the Android applications; therefore, Android Dimmunix provides platform-wide deadlock immunity, to all applications running on an Android phone. We also optimized the Java Dimmunix for synchronization-intensive applications. We extended Java Dimmunix with immunity against non-mutex deadlocks, i.e., deadlocks involving read-write locks, semaphores, condition variables, or external synchronization. We ran Dimmunix with real applications, like JBoss, Limewire, Vuze, Eclipse, Apache ActiveMQ, MySQL server, and SQLite.

We also implemented a collaborative version of Dimmunix, called Communix. Communix enables machines connected to the Internet to immunize each other against deadlocks. Once a node encounters a deadlock, the other nodes get protected against the deadlock, without having to encounter the deadlock.

Dimmunix appeared in the news, in magazines like EPFL Flash (Switzerland), IEEE Computer (USA), Scientific Computing (USA), Science Daily (USA), Swiss IT Magazine (Switzerland), inside-it.ch (Switzerland), Swisster (Switzerland), The Engineer (UK), The Register (UK), Continuity Central (UK), Ecademy (UK), L'Atelier (France), Techno Science (France), Innovations Report (Germany), PC Welt (Germany), Noorderlicht (The Netherlands), Webwereld (The Netherlands), Oggi Scienza (Italy), PC World (Brazil), Inovacao Tecnologica (Brazil), Diario Digital (Portugal), SG.hu (Hungary), telereport.rs (Serbia), Deccan Chronicle (India).

MISCELLANEOUS

LANGUAGES

- English: fluent
- Romanian: fluent
- French: good
- German: fair

EXTERNAL REVIEWS

- EuroSys 2009, 2011
- ASPLOS 2010
- USENIX 2009, 2011