POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

TITLE

Supervisors

Prof. Maurizio MORISIO

Candidate

Marco MILANI

September 2023

ACKNOWLEDGMENT

I would like to express my deepest gratitude to Prof Ugo Buy, my supervisor for the thesis. He is an experienced man who led me through the difficulties, carefully monitoring my progress and was always ready to help me with precious suggestions for the research. I had the opportunity to follow his lessons and appreciate his positive attitude and his passion that convey to me also during this period when I had the opportunity to research with him. I want to thank Prof. Maurizio Morisio and Prof. William Mansky for being part of my committee and reviewing my thesis. I could not have undertaken this journey without the support of my family who gave me all the help and love possible even if they are in another country. Their belief in me has kept my spirits and motivation high during this process. Last but not least, I'd like to express my gratitude to my friends who accompanied me through this journey, especially my roommates Alberto, Rebecca and Simone always ready to help each other and keep the morale up.

CONTRIBUTIONS OF AUTHORS

The development of the Deadlock Detector and Solver (DDS) that has been described in chapter 3 is part of the research of PhD. Eman Aldakheel and Prof. Ugo Buy. Part of the content from the chapter 3 have been previously published in the dissertation of PhD. Aldakheel (1). The images of the pseudocode and the architecture of the DDS are extracted from the dissertation of PhD. Aldakheel (1). Part of the content of the sections Tree, Handling the Correct Tree Building, Detecting Harmful Statement, Observer, Detector Graph, Detector, Solver Detector Algorithm and Solver Algorithm describing the components of the DDS is extracted from the dissertation of PhD. Aldakheel (1).

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Deadlocks is a common problem in concurrent programming. It can cause programs blockage unresponsive and hinder the efficient utilization of hardware capabilities. The objective of this research is to show a methodology that can effectively identify and resolve deadlocks occurring within real world applications using Java libraries, thereby improving the reliability and performance of software applications. To achieve this objective, we conducted a comprehensive investigation of Java libraries, leading to the identification of various instances of deadlocks. We developed specialized drivers to recreate these deadlocks, enabling a thorough analysis of their occurrence and behavior within the libraries. Subsequently, we used the Deadlock Detector and Solver (DDS) toolset to automatically detect and resolve deadlocks during run-time. DDS relies on a supervisory controller to monitor program execution and efficiently detect deadlocks caused by hold-and-wait cycles on Java monitor and reentrant locks. This autonomous and automatic deadlock detection and resolution process eliminates the need for manual intervention, making it highly efficient. The technique to resolve a deadlock depends on preempting a lock in the detected deadlock cycle, with the thread holding the lock being called the "victim" thread. The strategy involves returning the lock to the victim thread once another thread has used it, but requires that one of the threads in the deadlock is suitable to be victimized. To determine this, a preprocessing technique is employed to identify harmful statements in potential deadlock locations that alter a locked object or a shared object before a second lock is requested by a specified thread. Experimental results demonstrate the effectiveness and scalability of the DDS methodology. The average run-time overhead introduced by the DDS approach remains below 7%, ensuring its practicality and viability for real-world deployment. DDS is scalable and does not incur a noticeable overhead with an increase in the number of synchronization points and threads and it is a promising solution for detecting and resolving deadlocks in Java real world applications.

CHAPTER 1

INTRODUCTION

1.1 Deadlock Problem

Multithreaded programming has become an essential aspect of modern software development, allowing programs to perform concurrent tasks and make efficient use of available system resources. All modern computers uses multi-core processors based on the chip multithreading architecture, which employs multiple single-thread processor core integrated within a single CPU. Consequently, these processors can execute numerous threads simultaneously across the multiple cores (2).

In order to use the full power of multicore processors, the software needs to utilize concurrency in order to improve the performance of the program. In multi-threaded systems, it is common for multiple threads to access the same resource concurrently. To prevent data inconsistency and race conditions, a lock is used to ensure that only one thread can access the shared data at a time. However, this approach can result in a deadlock scenario where a thread enters a waiting state due to the requested resource being held by another waiting process. This process, at the same time, is waiting for another resource held by another waiting process, leading to a situation where none of the processes can change their state as they are dependent on the resources held by other processes. This state of the system is known as a deadlock, and it can persist indefinitely, leading to a system failure.

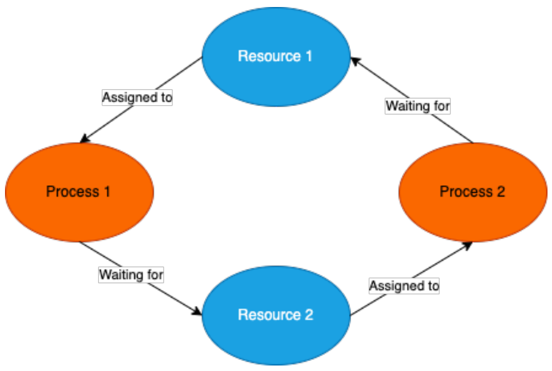


Figure 1: Deadlock illustration.

There are four conditions that are necessary to achieve deadlock (3) :

-Mutual Exclusion: It is necessary to have at least one resource in a mode that cannot be shared. If any other process requests this resource, it must wait until the resource is released.

-Hold and Wait: A process must hold at least one resource while simultaneously waiting for another resource that is currently held by another process.

-No preemption: Once a process holds a resource, it cannot be forcibly taken away from that process until the process willingly releases it.

-Circular Wait: There should be a set of processes P0, P1, P2, . . ., PN where each process P[i] is waiting for process P[(i + 1).

Deadlock is a complex issue that can greatly affect the reliability and consistency of multi-threaded programs with multiple asynchronous threads. Resolving deadlocks typically involves using timeout and rollback mechanisms. If left undetected, deadlocks can result in permanent thread blockage and this can be a significative problem especially for applications that must provide a continuous service. To provide context for this issue, we briefly introduce the research problem, including an overview of multicore hardware and its relationship with Java threads.

1.2 Thesis Objective

The focus of this thesis is the study of the Deadlock Detector and Solver (DDS), a tool designed to identify and resolve deadlocks in Java(1). Developed by Aldakheel and Buy, the DDS underwent extensive testing, primarily based on common multithreaded problems such as the Dining Philosopher and renowned multithreading benchmarks like the Java Grande Benchmark. However, the research lacked an essential component: the evaluation of the tool's performance in real-world applications.

The primary objective of this thesis is to provide experimental evidence demonstrating the effectiveness of the DDS in real-world scenarios. To achieve this goal, we have developed specialized software components that interact with some libraries in a specific way to create a deadlock. With these applications we can replicate the usage of the DDS over an actual software system. This approach enabled us to showcase the practical application and feasibility of the DDS methodology with real-world Java applications. The incorporation of real-world applications into the evaluation process enhances the reliability and applicability of the DDS, thereby establishing its value as a viable solution for detecting and resolving deadlocks in Java.

1.3 State Hypothesis

The focus of this research lies in the detection and resolution of deadlocks in Java, a high-level object-oriented programming language. Java was one of the pioneers in introducing multithreading to enhance program performance. Multithreading enables faster execution, improved scalability, and efficient utilization of multicore CPUs. However, it also brings challenges that developers need to address, especially concerning concurrent access to shared resources. Issues such as data races, deadlocks, and debugging difficulties arise due to the non-deterministic behavior of multithreading. To prevent concurrent accesses in Java, various locking mechanisms are utilized, including intrinsic locks, reentrant locks, and semaphores. In this research we are going to focus on reentrant lock and intrinsic lock deadlock detection and resolution.

Furthermore, Java provides a rich ecosystem of libraries, offering prebuilt code modules that developers can leverage to boost productivity and expedite software development. These libraries encapsulate reusable code, data structures, algorithms, and functionalities that address common programming problems. They are designed to be modular, easy to integrate, and compatible with different Java applications. However, improper use of code from external libraries can introduce complications. Although it is unlikely for a single developer to cause a deadlock on their own, when multiple developers contribute to the codebase, as is often the case with library usage, the occurrence of deadlocks becomes more frequent, as explored in this thesis.

1.4 Summary of Method

In this research, we have successfully identified and replicated various instances of deadlocks occurring in Java libraries. To accomplish this, we employed specialized software components called “drivers” that emulate the usage of these libraries within an actual software system. We developed one driver for each test with the specific aim of recreating a particular deadlock scenario. By simulating the complex structure and challenges present in real-world software systems, these programs provide an ideal platform for analyzing and comprehending deadlock occurrences.

We carefully selected a set of renowned and widely used libraries in Java as the focus of our study. The libraries considered in this research include Log4j, Commons Logging, DBCP, Pool, and Derby. Log4j is a widely recognized logging library for Java applications, providing extensive logging functionalities(4). Commons Logging facilitates integration of logging functionality into applications, supporting various logging libraries and configurations(5). DBCP enables efficient management of multiple connections to databases(6). Pool, on the other hand, is a library designed for the efficient management of connection pools(7). Lastly, Derby is a relational database implementation tailored specifically for Java applications(8). From the aforementioned libraries, we have developed six distinct test drivers that are designed to trigger various types of deadlocks.

1.5 Overview of DDS

This research is about a supervisory controller called the deadlock detector and solver (DDS) and its performance. DDS monitors the running program to detect and resolve deadlocks at run-time. The research methodology comprises three elements: a run-time deadlock detector, a run-time deadlock solver, and a preprocessing module. During run-time, the DDS employs two algorithms: the detector algorithm and the solver algorithm. Their purpose is to oversee, identify, and resolve deadlocks that may occur during run-time. The detector algorithm is responsible for identifying deadlocks by keeping track of how locks are requested, obtained, and released.

The DDS approach is designed to automatically detect and resolve resource deadlocks that may arise during run-time. With our tool we are able to resolve deadlocks from intrinsic locks and reentrant locks. During the preprocessing phase, we conduct interprocedural and alias analyses to identify "harmful statements" that may interfere with the program's consistency if a lock is preempted. Our assumption is that the monitored application does not include semaphores or cyclic barriers, which also could cause deadlocks. Deadlocks caused by semaphores and cyclic barriers are outside the scope of this thesis.

The mechanism used to detect and resolve deadlocks relies on a lock graph, also known as the *lock order graph*, which comprises vertices and directed edges. Each vertex corresponds to a mutex lock that a thread is holding, and each edge represents a request made by a thread for a particular mutex. A cycle in the lock graph signifies the existence of a deadlock. An edge from vertex *v1* to vertex *v2* implies that the thread that is holding the lock associated with *v1* is attempting to acquire the lock related to *v2*, which is concurrently held by another thread and this is the detector algorithm. The Solver algorithm is the other part of the DDS run-time monitoring process. When a cycle is detected in the lock graph, it indicates the presence of a deadlock. The detector identifies the affected thread and informs the solver algorithm about it.

The solver algorithm resolves the deadlock by instructing the affected thread to release its held lock, allowing another thread to acquire it. Once the second thread releases the lock, the solver algorithm returns it to the affected thread, which can then proceed with its processing. The thread whose mutex lock is preempted is referred to as the victim thread.

The third step in the DDS approach is preprocessing, which is crucial for maintaining consistent performance in applications. In order to resolve deadlocks, our solver algorithm preempts one of the locks involved in the deadlock cycle, but the preprocessing phase ensures that lock preemption is safe and does not leave the application in an inconsistent state. However, there is a risk that a thread may have modified an object whose lock is involved in the deadlock, making it unsafe to choose that thread as the victim. This is done by identifying harmful statements in the program code, which are statements that modify shared objects protected by a lock that may be victimized by the solver algorithm. To identify harmful statements, we analyze the program code to locate requests for lock acquisitions and create a tree showing call dependencies among those locations. We examine statements along paths between lock acquisition points to determine if any write operations on shared objects are performed, which are considered harmful statements. If any are found, we cannot preempt the lock that protects that statement. We perceive the occurrence of harmful statements as relatively uncommon, as developers typically acquire all necessary locks before proceeding with operations on locked objects.

1.6 Experimental Evaluation Overview

During the evaluation process of the DDS, we conducted our tests on 6 different drivers using the selected libraries. For each test case, we created two versions: one version with a deadlock intentionally introduced, and another version that was modified to be deadlock-free.

This allowed us to demonstrate the effectiveness of our approach in resolving deadlocks within real-world applications.

Additionally, we utilized the set of test cases that are deadlock-free to measure the performance impact of the DDS. We monitored the CPU time and elapsed time while running the applications both with and without the DDS, and compared the results. The purpose of this comparison was to evaluate the overhead introduced by the DDS.

The test results revealed that DDS adds an average overhead of 9% for the CPU time. This overhead was considered reasonable, considering the benefits gained from the effective resolution of deadlocks. It is important to note that these percentages were obtained under the assumption that no harmful statements were present in the tested applications. Overall, these findings demonstrate that the DDS is capable of effectively resolving deadlocks in real-world applications, while maintaining a reasonable level of overhead.

1.7 Thesis Overview

This thesis is structured into seven chapters, which include an introduction chapter. The subsequent chapters are organized as follows:

Chapter 2 shows the foundation for understanding Java multithreading, the different types of locks examined in this research, the utilization of the Java Virtual Machine Tool Interface, and the significance of libraries in Java.

Chapter 3 illustrate the implementation of the Deadlock Detection and Solution (DDS) methodology, exploring its specific components and how they are interconnected.

Chapter 4 provides a comprehensive analysis of the libraries under consideration in this study, along with detailed explanations of the drivers developed to recreate the specific deadlock scenarios.

Chapter 5 presents the results obtained from our benchmark tests, which showcase the effectiveness of our DDS methodology.

Chapter 6 focuses on the existing related works surrounding deadlock detection, avoidance, and prevention. We explain the differences between these approaches and our proposed methodology.

In Chapter 7, we draw conclusions based on the application of the DDS, our supervisory controller method.

CHAPTER 2

BACKGROUND ON JAVA

Java is a popular and established programming language for developing enterprise applications. Java development has progressed from small programs that run in a Web browser to massive business distributed systems that operate on multiple servers. Some of the most famous applications are developed in Java, mainly for backend programming but also for user interface development. Some example of popular companies that uses Java are Amazon, Spotify, Netflix and Google.

Java is considered one of the first high-level programming languages to introduce multithreading, enabling developers to create multithreaded programs and use the full potential of hardware capabilities (9; 10). According to the TIOBE popularity index, Java has been among the top three programming languages for the past 20 years and now ranks third with a market share of 10.46 % (11). Due to its remarkable scalability, robustness and dynamism, Java proves to be an excellent choice for the development of applications by the Java app development company. With its capabilities, it empowers the company to construct efficient multithreaded applications.

2.1 Locking Mechanisms in Java

The Java programming language provides various object-locking mechanisms such as intrinsic locks, reentrant locks, and semaphores to support thread synchronization. In Java, every thread operates as autonomous path of execution that have the ability to run concurrently with other threads. In order to access a shared object, a thread is required to obtain a mutual exclusion lock specific to that object. The thread will successfully acquire the lock if it is not currently held by any other thread. Otherwise, the thread will wait until the lock is released.

However, concurrent programming can result in complex bugs due to its non-deterministic behavior. One of the primary challenges faced by software developers is the occurrence of deadlock, which happens when a hold-and-wait cycle arises concerning locked objects. Moreover, detecting deadlock through testing is particularly challenging because it is likely to occur non-deterministically. In this section, we will focus on the type of locks analysed by the DDS: intrinsic locks and reentrant locks.

2.1.1 Intrinsic lock

An intrinsic lock is a type of lock associated with every shared object in Java programming. Its primary purpose is to ensure that only one thread at a time can access an object's state through its instance method calls. When a thread invokes a synchronized method on an object, it first needs to obtain the intrinsic lock associated with that object. The lock is automatically released when the method call completes, or if an unhandled exception occurs during the method's execution. While a thread holds an intrinsic lock, no other thread can acquire the same lock. Any other thread that attempts to acquire the lock while it is held will be blocked until the lock is released.

Intrinsic locks are also reentrant, which means that once a thread has acquired the lock for a particular method, it can call other synchronized methods on the same object without having to reacquire the lock. It is not necessary to use the synchronized keyword for every method of an object, only those that require exclusive access to the object's state. It is important to note that constructors cannot be synchronized in Java. Attempting to use the synchronized keyword with a constructor is a syntax error because only the thread that creates an object should be able to access to it while it is being constructed.

There are different ways to use intrinsic lock as shown in 2.1. This example shows a synchronized method and a synchronized block used to manage a bank account. A thread can acquire an intrinsic lock using the synchronized keyword to a non-static method as shown on Line 4 of 2.1. In this case, the thread acquires the intrinsic lock associated with the object receiving the method invocation. The object is unlocked when the synchronized method returns. Alternatively, a synchronized block can be used to acquire an intrinsic lock, as demonstrated on Line 10 of Code 2.1. The synchronized block is executed if the specified object is not already locked. The lock is released when the thread completes the execution of the synchronized block.

2.1.2 Reentrant lock

In Java, a reentrant lock is a specific type of lock that was introduced in Java 1.5. It grants a process the ability to claim the lock multiple times without being blocked by its own actions. This feature proves particularly advantageous in situations where it becomes challenging to keep track of whether a lock has already been acquired. When a lock is not reentrant, a process may grab the lock, then block when attempting to grab it again, causing a deadlock in its own execution. Reentrancy refers to a property exhibited by code that lacks a central mutable state, which could be compromised if the code is invoked while it is already executing. This situation

Listing 2.1: Example of usage of synchronized keyword.

```
public class SynchronizedBankAccount {
    private long balance;
    // synchronized method
    public synchronized long updateBalance(long amount) {
      long newBalance = balance + amount;
      balance = newBalance;
      return balance;
    }

    public void withdraw(double amount){
        // synchronized block
        synchronized(this){
            balance -= amount;
        }
    }
  }
```

may arise from another thread or a recursive execution path that originates within the code itself. If the code depends on shared state that can be modified during its execution, it cannot be considered reentrant, especially if such an update could potentially disrupt its functionality.

Reentrant locks add flexibility to the program. They can be acquired and released in every position of the code. An example of the use of reentrat lock is shown in 2.2. The code snippet shows the use of a reentrant lock to protect a counter that is shared among multiple threads. In the given code, a thread requests and releases the lock by invoking the *lock* and *unlock* functions on a lock object. The reentrant lock ensures that only one thread can hold the lock at a time, allowing exclusive access to the shared counter.

A common use case for a reentrant lock is in situations where a computation involves traversing a graph with cycles or multiple paths to the same node. In such cases, a node could be locked to deal with potential data corruption due to race conditions. However, for performance

Listing 2.2: Example of usage of reentrant locks.

```java
class Counter implements Runnable {
    private String threadName;
    ReentrantLock lock;
    Counter(String threadName, ReentrantLock lock){
        this.threadName = threadName;
        this.lock = lock;
    }
    @Override
    public void run() {
        // acquiring the lock
        lock.lock();
        SharedResource.count++;
        // releasing the lock
        lock.unlock();
    }
}
```

reasons, it may not be desirable to globally lock the entire data structure. Furthermore, the computation may not retain complete information on what nodes it has visited, making it difficult to determine what locks have already been acquired (12). In such situations, a reentrant locking mechanism can alleviate the need to determine whether a node has already been visited. The node can be locked blindly, perhaps unlocking it after it is removed from the queue.

2.1.3 Differences between Reentrant lock and Intrinsic lock

Prior to the introduction of reentrant locks, concurrency was attained through the utilization of synchronized methods and blocks. While both of these mechanisms serve the same purpose of synchronizing access to shared resources, there are several differences between them that developers should be aware of.

An intrinsic lock is the most basic form of locking in Java, and it is implemented using the synchronized keyword. Intrinsic locks provide a simple and straightforward way to synchronize access to shared resources, and they ensure that only one thread can hold the lock at a time. Intrinsic locks have a built-in monitor, which ensures that threads waiting for the lock are notified when the lock becomes available. On the other hand, a reentrant lock is a more advanced form of locking mechanism that provides additional features not available in intrinsic locks. Unlike synchronized constructs, a reentrant lock is unstructured, which means that developers can hold the lock across methods, and they do not need to use a block structure for locking (13).

Reentrant locks have an advantage over intrinsic locks in terms of simplifying the development of concurrent code. In the absence of reentrant locks, if a subclass overrides a synchronized method before calling the superclass method, which is also synchronized on the same object, a deadlock may occur. This situation is a concern with intrinsic locks because they cannot re-enter a lock and will try to acquire the lock they already hold. In contrast, reentrant locks are assigned a hold count, which increases by one every time the lock is acquired by a thread. When a thread releases the lock, the hold count for that lock decreases by one, and the lock becomes free when the hold count reaches zero. However, a thread cannot acquire a lock held by another thread by increasing the count, as the count is associated with the thread for reentrancy purposes.

Reentrant locks also support lock polling and interruptible lock waits that support timeouts, this is possible using the statement *trylock()*. When a lock is requested through *trylock()*, the method checks if the lock is available at that particular time. If the lock is available, the *trylock()* statement returns true, false otherwise. If the *trylock()* method returns true, it is important to unlock the locked object after completing the critical section. A critical section is a part of the code that accesses shared resources and must be executed atomically. Essentially, *trylock()* is a nonblocking lock that locks only if it is available. If the lock is not available, the thread continues executing the next statements. This feature allows better performance since threads are not blocked while checking for the lock availability. On the other hand, with intrinsic locks is not possible to obtain information of whether the lock is available to be acquired. If a thread requires a lock but is not able to acquire it, the thread will be blocked until it can acquire the lock.(14)

Reentrant locks have a configurable fairness policy by passing a boolean value to the constructor method. If it is true, fairness is applied (14). When it comes to acquiring access to shared resources, unfair locks do not guarantee any particular order of threads in obtaining the lock. Conversely, fair locks ensure that locks are acquired in the order in which they were requested. For example, if a thread has been waiting longer in the queue than another, it is guaranteed that once the current thread completes, the longest waiting thread will be able to access the shared resource. This advantage allows for more flexible thread scheduling. However, intrinsic locks can also be more scalable, performing much better under higher contention, as they allow threads to acquire a lock when it becomes available, regardless of their order in the waiting queue.

The benefits of using a reentrant lock come with added complexity. They are generally recommended for advanced users who have identified a specific need for the features provided by reentrant locks. It is essential to note that the vast majority of synchronized blocks hardly ever exhibit any contention. Therefore, it is advisable to develop with intrinsic locks, rather than assuming that the performance will be better if you use a reentrant lock.

2.2 Java libraries

The Java programming language offers users and companies the ability to create libraries of classes, providing programmers with pre-existing code that can be utilized instead of writing everything from scratch. Libraries not only allow the use of interfaces and the creation of subclasses to make small modifications for better program outcomes, but also serve as a solid foundation upon which developers can build their applications. Libraries act as building blocks that simplify complex tasks, introduce abstraction layers, and enable developers to concentrate on solving specific challenges within their domain, rather than reinventing existing functionalities. They are designed to be modular, easy to integrate, and compatible with a wide range of Java applications.

However, utilizing libraries in Java is not always a easy process, as they can encounter compatibility issues with different versions of Java and with other libraries. In fact, they require maintenance to ensure they remain compatible with newer Java versions. During the creation or updating of a library, bugs can be introduced inadvertently, causing problems for users. Our particular focus lies on deadlock bugs, which can be particularly challenging to identify. These bugs can originate from within the library itself or can arise from the usage of multiple threads interacting with libraries while acquiring locks in an improper manner. Detecting and resolving such bugs is even more challenging due to the collaborative nature of library development, involving contributions from multiple developers.

2.3 Java Virtual Machine Tool Interface

The Java Virtual Machine Tool Interface (JVMTI) is a programming interface that development and monitoring tools employ(15). It allows to view the state of programs running in the Java virtual machine (JVM) as well as to control their execution. The JVMTI is intended to provide a VM interface for a variety of tools that require access to JVM state, such as profiling, debugging, monitoring, thread analysis, and coverage analysis tools. The JVMTI is a bidirectional interface, that means that a client, also known as agent, can be notified of occurrences through events. JVMTI may query and control the program using a variety of functions, either in reaction to or independently of events. Agents operate in the same process as the virtual machine that is running the application under examination and communicate with it directly. This communication takes place via the JVMTI. The native in-process interface provides maximum control with little tool intervention. Any native language that supports C language calling conventions and C or C++ definitions can be used to write agents. The agent used in this research define the following callbacks thanks to the JVMTI:

Lock_contended_enter() callback function is called in a situation in which a thread attempts to acquire a reentrant lock currently held by another thread.

Lock_contended_entered() callback function is called when a blocked thread (waiting for a lock) acquires the needed lock.

Monitor_contended_enter() callback function is called when a thread attempts to acquire a monitor lock held by another thread.

Monitor_contended_entered() callback function is called when a waiting thread acquires the monitor for which it was waiting.

The agent uses the function *RawMonitorEnter()* to give the ownership of a monitor to a thread and *RawMonitorExit()* to revoke the ownership of a monitor to a thread. Similarly, it uses *lock()* and *unlock()* functions to assign and release reentrant locks to a thread.

CHAPTER 3

DDS IMPLEMENTATION

The supervisory controller described in this chapter has been developed by Dra. Eman Aldakheel and Prof. Ugo Buy(1). The Deadlock Detector and Solver (DDS) operates by preempting resources at run-time, utilizing a two-part approach consisting of the preprocessing phase and the run-time phase with the detector and solver algorithm. During the preprocessing phase, the primary objective is to maintain consistency throughout the execution of an application. This is achieved by conducting a static analysis of the code, identifying the location of what is referred to as the “harmful statements”. This information is crucial for the solver algorithm, as it ensures that the related resource cannot be preempted. If this happens, we cannot preempt a lock because it would not guarantee the integrity of the program state. The run-time components of the detector and solver are responsible for actively detecting deadlocks during program execution and taking preemptive action by selecting one of the related resources to be preempted. This preemptive action allows for the normal flow of the program to resume, resolving the deadlock situation. By combining the preprocessing phase for maintaining consistency and the run-time phase for deadlock detection and resource preemption, the DDS effectively addresses the challenges posed by deadlocks in Java programs. This comprehensive approach ensures that the program can continue executing without disruptions caused by deadlock scenarios, improving the reliability and efficiency of multithreaded applications.

3.1 Preprocessing

The preprocessing phase of our methodology plays a critical role in ensuring the consistency of the application. It involves an in-depth analysis of the source code, specifically targeting the locations where lock acquisitions are requested. By examining these program locations, we construct a comprehensive tree structure that captures the interdependencies among them. In this tree representation, each vertex corresponds to a program location associated with locking operations, while the edges indicate the existence of a path connecting two locations. This path signifies that the child vertex can be reached from its parent vertex within the program's execution flow. It is important to note that a separate tree is generated for each individual application that is being considered. This tree structure provides valuable insights into the relationships and dependencies among the various program locations where locks are utilized. This knowledge aids us in effectively addressing issues related to concurrency and synchronization. If it is possible to find a victim thread we can ensure the overall consistency and reliability of the application.

3.1.1 Tree

The preprocessing phase of our methodology relies on the analysis of a comprehensive tree structure that encompasses all synchronized blocks and methods, as well as their interconnections. To facilitate this analysis, we utilize the Spoon Java library, which enables us to construct an Abstract Syntax Tree (AST) representing the abstract syntactic structure of the source code. We obtain an AST that serves as a basis for our subsequent analysis. The AST

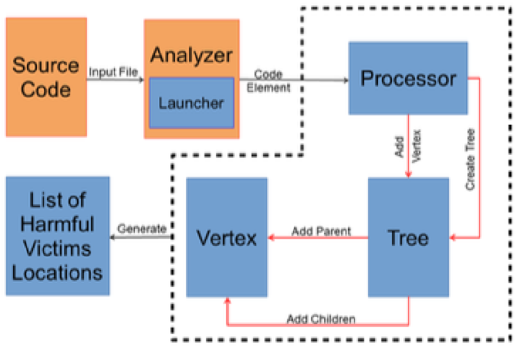


Figure 2: DDS preprocessing architecture.

consists of vertices representing various code elements such as statements, loops, or expressions, while the edges represent the containment relationships between these elements.

Once we have obtained the AST, our preprocessor focuses on extracting relevant information related to synchronized statements and the methods that invoke them to build a reduced three containing only the information needed. This is achieved by examining all statements and expressions within the code and selecting those that correspond to synchronized methods. The main goal of the reduced tree is to show the call dependencies for the synchronized blocks and methods. Every synchronized point is then represented as a vertex in a reduced syntax tree specifically constructed for preprocessing purposes as shown on Figure 3. When it comes to intrinsic locks, the scope is well-defined, making it relatively easier to handle. However, for reentrant locks, we employ a different approach. We need to identify and pair the lock and unlock calls in the code to determine the vertices of the tree that correspond to the lock acquisition and release points. To efficiently manage and access this tree structure, we employ a map data structure with unique keys associated with each vertex. This enables fast retrieval and manipulation of the collected data during subsequent stages of our methodology. The first vertex added to the tree serves as the root, which in our case corresponds to a compilation-time root package generated for each program. Subsequent vertices are connected to this root, forming the hierarchical tree structure. Each method vertex has a parent vertex, which can either be the root or another method vertex. If a vertex represents a synchronization point and has a parent which is a synchronized method, it is considered a direct vertex, indicating the presence of nested synchronization within the code. We also search for methods that invoke any of the synchronized vertices already present in the tree. These invoking methods, which do not contain synchronization points themselves, are referred to as indirect vertices. They serve as connections between the direct vertices, pointing to either a method with synchronization points or another indirect synchronization point. This process of discovering and adding direct and indirect vertices to the preprocessing tree is performed recursively, ensuring that all relevant calls and dependencies are captured within the structure.

3.1.2 Handling the Correct Tree Building

The process of building the tree in our methodology presents certain challenges that need to be addressed. One of these challenges arises from the presence of common programming constructs such as loops and branches, which introduce complexities that can prevent accurate static analysis by disrupting the linearity of the program.

```
// for intrinsic locks
for (each synchronization point) do
    find the parent vertex
    generate a key for the current synchronization point
    create a new vertex for the current synchronization point
    add the vertex to the hashMap using the generated key
    add the vertex to the found parent vertex

// for reentrant locks
for (each critical method invocation) do
    find the closing points for the current invocation
        // not under condition
        if (the closing point is not in a branch)
            // pair the lock() and unlock()
            pair the two invocations
            define the scope to be between the two invocations
                start = lock() invocation
                end = unlock() invocation
        else
            define the desired scope to be from the critical method invocation until the last reachable statement
                start = lock() invocation
                end = last reachable statement to the lock() invocation

// adding indirect vertices
for (each vertex in the tree) do
    // An invocation of a vertex means that the invocation of the method contains the current vertex
    recursively find all invocations of the vertex
    if (an invocation is found)
        // the following operations are for the method containing the invocation
        generate a key for the current method
        create a new indirect vertex for the current method
        add the vertex to the hashMap using the generated key
        add the vertex to the found parent vertex
        if (the current vertex has children )
            // means that this invocation takes place between two synchronization points
            adjust the children vertices\rq{} parent to be the newly added vertex
```

Figure 3: Algorithm used to build the tree.

To face the issue posed by loops, we adopt an approach that involves considering a single iteration and treating it as representative of all subsequent iterations. In essence, we analyze the statements within the loop's body to identify any potentially harmful statements. Previous research has indicated that this method yields promising results in terms of performance and achieving the desired outcomes (16).

When it comes to analyzing branches, we encounter a similar challenge due to the uncertainty surrounding which branch will be executed at run-time. To address this, we assume the execution of both the "if" and "else" statements during the analysis. Consequently, we examine all the statements within both branches to ensure a comprehensive understanding of their potential impact.

Another significant challenge in the tree-building process is determining the scope of reentrant locks. This involves matching the lock and unlock statements for the same object, which can be intricate as it requires meticulous investigation of possible closing points. If an unlock statement is missing or cannot be matched with its corresponding lock, we define the scope as encompassing all the paths reachable from the point of lock acquisition. Similarly, if an unlock statement is located within a branch, we adopt a conservative approach and refrain from matching the lock, instead considering the last reachable point.

During the preprocessing phase, we treat an array of locks as a single lock. This decision stems from the fact that the index for identifying each lock can only be determined at runtime. By treating the array as a whole, we streamline the analysis process and ensure a cohesive approach to handling locks. Overall, addressing these challenges in the tree-building phase of our methodology is crucial to ensure accurate and reliable results in the subsequent stages of the analysis.

3.1.3 Detecting Harmful Statement

The synchronized reduced tree serves as a crucial tool in our methodology for identifying harmful statements within the code. A statement is called harmful if it involves a write operation on a shared object and is executed between two synchronized blocks or lock acquisitions. When a harmful statement is present, we will be unable to preempt the lock that safeguards it at run-time. Therefore, it is essential to accurately identify these statements to ensure proper synchronization and prevent potential inconsistencies. Inside synchronized methods, we must carefully examine write operations on object data members and their compile-time aliasing. This refers to the situation where multiple objects occupy the same shared memory but have different variable names. Determining harmful statements related to objects that are being synchronized becomes challenging when aliases of the synchronized objects exist. To account for unresolved aliasing scenarios, we adopt a conservative approach. We consider all possibilities of aliasing that remain unresolved as potentially harmful statements. This cautious strategy ensures that the program's data integrity is never compromised, even though it may impact the run-time performance.

3.2 DDS Runtime

In this section we explain the architecture of the DDS during run-time, which forms the very essence of this project. At its core, the DDS comprises an observer that monitors key actions performed by the program. When a critical action occurs, the DDS search for the presence of deadlocks and swiftly resolves them. To comprehensively understand the functioning of the DDS during run-time, we first analyze the various components that constitute this framework and examine their interactions. Through this analysis, we aim to elucidate the mechanisms that enable the DDS to effectively monitor and manage deadlocks in a proactive manner. Subsequently, we delve into the process of deadlock detection and resolution employed by the DDS. We explore the methodologies employed to detect the presence of deadlocks. Then, we analyze the strategies implemented to effectively resolve these deadlocks, ensuring the uninterrupted execution of the program.

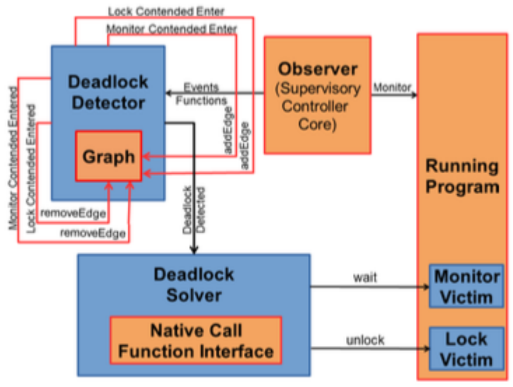


Figure 4: DDS run-time architecture .

3.2.1 Observer

The observer is a key component within the DDS framework, responsible for monitoring a running program, see Figure 4. Serving as the vital connection between the Java Virtual Machine Tool Interface (JVMTI) and the Detector component, the observer filters and selects specific callbacks that are of relevance and interest. This ensures that only pertinent information, such as lock requests, releases, and acquisitions made by threads, is relayed to the Detector for further analysis and processing. The JVMTI provides a rich set of callbacks that furnish the observer with essential data required by the Detector. These callbacks encapsulate crucial information, such as the identity of the thread holding a lock and the thread seeking to acquire it. In this manner, the observer acts as the information conduit, channeling these callbacks to the Detector, enabling a comprehensive understanding of the run-time behavior of locks and monitors within the program. In the dynamic context of a running program, there are specific events that we diligently monitor, which are closely associated with critical callbacks. These events include:

- Monitor Request: Occurring when a thread requests the acquisition of a lock associated with a synchronized block and it remains in a waiting state until another thread releases it. This event triggers the corresponding callback, referred to as *Monitor_Contended_Enter()*.
- Monitor Acquisition: Occurring when a thread successfully acquires a lock pertaining to a synchronized block. This event prompts the corresponding callback, referred to as *Monitor_Contended_Entered()*.
- Reentrant lock request: Occurring when a thread requests the acquisition of a reentrant lock, that is currently owned by another thread. Consequently, it must await the release of this lock by the owning thread. The corresponding callback associated with this event is known as *Lock_Contended_Enter()*.
- Reentrant Lock Acquisition: Occurring when a thread successfully acquires a reentrant lock. This event triggers the corresponding callback, referred to as *Lock_Contended_-Entered()*.

By monitoring and capturing these crucial events through their corresponding callbacks, the observer ensures that the DDS framework knows the essential insights required to detect and resolve potential deadlocks.

3.2.2 Detector Graph

The DDS run-time framework relies on a directed graph to encapsulate and manage crucial knowledge pertaining to threads and the locks they possess. This graph serves as the foundation of the deadlock detection, providing critical insights into the relationships between threads and their lock dependencies. The graph itself is composed of vertices, each of which represents an individual thread within the program. These vertices are interconnected by direct edges that signify the lock request relationships between threads. Consequently, if there exists an edge labeled *e1* from vertex *v1* to vertex *v2*, it signifies that *v1* necessitates a lock currently held by *v2*. Each vertex in the graph stores two key attributes: a list of locks owned by the corresponding thread and a collection of edges connecting it to other vertices. These vertices are uniquely identified by a distinct ID, which serves as a representative marker of the associated thread.

The graph plays a crucial role in the run-time deadlock detection process, offering important functionalities. Firstly, it allows for the addition of edges, which represent the relationships between threads when one thread requests a lock owned by another. Secondly, the graph allows for the removal of edges when a thread successfully acquires a lock that was previously owned by another thread, and for which it had been waiting. This dynamic adjustment effectively updates the graph to reflect the changes in lock ownership. Lastly, the graph serves as a fundamental tool utilized by the DDS detector to detect cycles. When a cycle is identified within the graph, it signifies the presence of a deadlock. In other words, it signifies that a thread *t1* is awaiting a lock owned by thread *t2*, while simultaneously *t2* is awaiting another lock owned by *t1*. This cyclic dependency among threads denotes a deadlock situation that necessitates appropriate resolution strategies.

3.2.3 Detector

The detector is the component of the DDS responsible for utilizing the information provided by the observer to construct and maintain the detector graph. This graph serves as a representation of the run-time state, facilitating deadlock detection and resolution. To efficiently manage and retrieve information, the detector employs a map data structure. This map utilizes unique hash IDs to represent both threads and the locks they have acquired. By using this mapping mechanism, the detector can quickly retrieve and associate relevant data. When a thread acquires a lock, the detector adds a new entry to the map, reflecting this lock acquisition. Conversely, when a thread releases a lock, the corresponding entry is removed from the map. This dynamic mapping process allows for accurate tracking of lock ownership throughout program execution. Upon receiving information about thread actions from the observer, the detector reacts accordingly. It updates the graph representation by incorporating the latest information from the observer as explained in the Section 3.2.2. Additionally, it checks for the presence of cycles within the graph, as it signify the occurrence of a deadlock. Whenever the detector identifies a cycle in the graph, it alerts the deadlock solver. This notification enables the solver to initiate appropriate actions to resolve the deadlock scenario and restore normal program execution.

3.2.4 Solver

The solver is the component of the DDS that resolves the deadlocks that are detected by the detector. When the detector identifies a cycle within the graph, indicating the presence of a deadlock in the program, the solver is activated. The primary objective of the solver is to determine the victim thread and the specific lock or monitor involved in the deadlock. To achieve this, it relies on the information collected and stored within the graph and the map by the detector. By using those information, the solver retrieves the corresponding lock or monitor from the heap. It then proceeds to preempt the lock, effectively interrupting its ownership by the victim thread. This preemptive action allows another thread, which has been waiting for the lock, to acquire it and proceed with its execution. Once the thread that acquired the lock completes its execution, the lock is released and returned to the victim thread. This process of preemption of the locks ensures that the deadlock is resolved, and the program can continue its execution.

3.2.5 Detector Algorithm

The detector is the component of DDS responsible for detecting deadlocks using an algorithm to take action on the lock graph. The presence of a cycle within the graph indicates that the program is in a deadlock state, as explained in Section 3.2.2. In this section, we will specifically discuss how the detector manages the graph and detects cycles within it.

Each vertex in the graph represents a thread and is added to the graph only if it acquires a resource. To optimize performance, the graph remains empty until at least one contention of locks occurs. This is particularly important for large programs with numerous running threads. Whenever a thread requests a lock owned by another thread, we add the corresponding vertices and the related edge to the graph, retrieving them from the map if they are not yet present inside the graph. For every added edge, the detector performs a graph search to check for cycles using a Depth First Search (DFS) algorithm. The algorithm has a complexity of O(V+E), where V is the number of vertices and E is the number of edges in the graph. When a thread successfully acquires the resource it was waiting for, the edge representing that contention is removed from the graph. If the thread no longer owns any resource that other threads are trying to acquire, we also remove the corresponding vertex. The information of the thread is updated in the map by removing the owned lock.

The detector algorithm is triggered when a contention occurs between two threads. In such cases, the detector obtains the necessary information, including the requesting thread and the owning thread. For monitor locks, the detector directly obtains this information from the observer, which provides the requesting thread and the monitor lock object stored within the owning thread. However, for reentrant locks, the necessary information is not readily available within the class, so we retrieve the requester, owner thread, and lock from the heap. At this point, the obtained information is added to both the graph and the map, and a cycle check is performed within the graph. However, it is important to note that not all cycles represent deadlocks. Sometimes, these cycles can simply be a delay in communication between the DDS

```
Function monitor_contended_enter(thr, obj) /* algorithm to detect deadlocks and
 maintain the graph for the monitor.                                          */
begin
    Data: current thread and object
    Result: call DDS solver when a deadlock is detected
    if !owner.find(obj) then
    begin
        thr ⟵ currentowneroftheobject
        owner[obj] ⟵ thr
    end
    else
    begin
        ownerTh ⟵ owner.find(obj)
        graph.addEdge(thr, ownerTh, obj)
        /* checks the graph for a cycle after adding an edge                 */

        if !graph.dag() then
        begin
            wait(≈ 20milliseconds)
            /* rechecks for a cycle in case we have communication delays     */

            if !graph.dag() then
            begin
                deadlockDetected ⟵ true
                if deadlockDetected then
                begin
                    /* call DDS solver algorithm to resolve the detected deadlock */

                    solver(thr, obj)
                end
            end
        end
        else
        begin
            deadlockDetected ⟵ false
        end
    end
end
Function monitor_contended_entered(thr, obj) /* algorithm to delete an edge from
 the graph                                                                    */
begin
    Data: current thread and object.
    Result: deleting edge from the graph.
    owner.find(obj)
    owner.erase(obj)
    graph.removeEdge(thr, obj)
end
```

Figure 5: Run-time DDS detector algorithm for monitor locks.

```
Function lock_contended_enter(thr, obj) /* Algorithm to detect deadlocks by
  building the lock graph when a thread attempts to acquire an unavailable lock.
  */
begin
    Data: current thread and object(lock)
    Result: call DDS solver when a deadlock is detected
    obj ⟵ getBlocker(thr)
    owner ⟵ getOwner(obj)
    if !owner then
    begin
        error ⟵ OwnerNotFound
        exit()
    end
    else
    begin
        ownerTh ⟵ owner
        graph.addEdge(thr, ownerTh, obj)
        /* checks the graph for a cycle after adding an edge                    */

        if !graph.dag() then
        begin
            wait(≈ 20milliseconds)
            /* rechecks for a cycle in the event of communication delays        */

            if !graph.dag() then
            begin
                deadlockDetected ⟵ true
                if deadlockDetected then
                begin
                    /* call DDS solver to resolve the detected deadlock         */

                    solver(thr, obj)
                end
            end
        end
        else
        begin
            deadlockDetected ⟵ false
        end
    end
end
Function lock_contended_entered(thr, obj) /* Algorithm to delete edge from the
  graph when a thread obtains a for which it has been waiting                   */
begin
    Data: current thread and object(lock)
    Result: deleting edge from the graph
    owner ⟵ getOwner(obj)
    owner.erase(obj)
    graph.removeEdge(thr, obj)
end
```

Figure 6: Run-time DDS detector algorithm for reentrant locks.

components since they operate asynchronously with respect to the main program. This delay occurs due to the gap between the execution of the running program and the actual time when the JVMTI callback is invoked. This delay is typically estimated to be between 10 and 20 milliseconds. Thus, we recheck the presence of a cycle after this delay. In case of a deadlock it will persist at run-time, so we can detect it after the delay.

As showed in the code snippet Figure 5, the observer triggers the monitor_contended_enter function when a thread is waiting for a lock. When this occurs, the detector has the necessary input to execute the code and add the corresponding vertices and edge. On the other hand, when the monitor_contended_entered callback is executed, the observer provides the input to remove the related edge between two vertices. Similar to what we did with monitor locks, we add an edge to the graph with each lock_contended_enter call, representing a resource contention, as shown in Figure 6. When a blocked thread successfully acquires a lock for which it was waiting, the lock_contended_entered function is triggered. This signal received from the observer results in the removal of the corresponding edge, as shown in Figure 6.

3.2.6 Solver Algorithm

The solver algorithm is an essential component utilized by the solver to resolve deadlocks within a running program. When the detector detects the presence of a deadlock and alerts the solver, the solver identifies a victim thread to preempt the associated lock. To achieve this, the solver compels the victim thread to release the lock by issuing a wait or unlock statement. This action allows another thread that requires the lock to acquire it. Once the second thread releases the lock, the solver requests a notify or an unlock statement on the victim thread, enabling the victim thread to regain the lost lock and resume processing.

```
1 Function monitor_solver(thr, obj) /* algorithm to detect and resolve
    deadlocks.                                                        */
2 begin
      Data: current thread and object
      Result: resolves the detected deadlock by issuing a wait on the monitor lock
3     $thrsInCy \longleftarrow graph.getVertexsInCycle(thr)$
4     $vicThr \longleftarrow randPickOne(thrsInCy)$
      /* gets object class and class signature                          */
5
6     $mID \longleftarrow GetMethodID(cls, "wait", sig)$
      /* call wait() on the victim thread with passing time out argument
         (tOut)                                                         */
7
8     $CallVoidMethod(obj, mID, tOut)$
9 end
```

Figure 7: Run-time DDS solver algorithm for monitor locks .

The solver can preempt different types of locks, including monitors and reentrant locks, and employs distinct strategies for each case. For monitor locks, the solver directs the victim thread to invoke the wait method and subsequently preempts the lock. This enables the waiting thread to acquire the lock and complete its task. Once the waiting thread releases the lock, the solver invokes the notify method on the victim thread, allowing it to resume execution.

In the case of reentrant locks, we obtain the locked object and utilize the unlock method, as shown in Line 8 of Figure 8. Furthermore, we label the object with the “unlock”status to indicate that it has been preempted from the victim thread. When the deadlock solver executes an unlock operation on the victim thread’s object, the victim thread releases the lock. Subsequently, it blocks itself until the lock is released. Once the current thread completes its execution, we tag the object with “lock” status and relock it on the victim thread by notifying the victim thread of the lock availability.

The code illustrated in Figure 8 and Figure 7 represents scenarios where no harmful statements are identified. If a victim thread cannot be identified, the DDS is unable to preempt a lock, thus leaving the deadlock unresolved in the running program. However, it is worth noting that the occurrence of harmful statements is relatively uncommon in practice. Developers typically acquire all required locks before performing operations on locked objects, minimizing the likelihood of encountering harmful statements and potential deadlocks.

```
Function lock_contended_enter(thr, obj) /* Algorithm to detect and resolve
 deadlocks by building the lock graph when a thread attempts to acquire an
 unavailable lock.                                                        */
begin
    Data: current thread and object(lock)
    Result: resolve the detected deadlock by using unlock
    thrsInCy ⟵ graph.getVertexsInCycle()
    vicThr ⟵ pickVictimthread(thrsInCy)
    /* get object class and class signature to obtain the method id      */

    mID ⟵ GetStaticMethodID(cls, "unlock", sig)
    /* tag the ``clear'' object with ``unlock'' status                   */

    obj.tag(unlock)
    /* call unlock() on the victim thread                                */

    CallStaticBooleanMethod(cls, mID, obj)
    /* tag the ``unlock'' object with ``lock'' status and return it back to
       the victim thread                                                 */

    obj.tag(lock)
    /* get object class and class signature to obtain the method id to
       re-lock() when the tagged object get freed                        */

    mID ⟵ GetStaticMethodID(cls, "lock", sig)
    /* tag the ``lock'' object with ``clear'' status again; so, no further
       action is needed                                                  */

    obj.tag(clear)
    /* call lock() on the victim thread to return the lock back         */

    CallStaticBooleanMethod(cls, mID, obj)
end
```

Figure 8: Run-time DDS solver algorithm for reentrant locks .

CHAPTER 4

DEADLOCK-PRONE JAVA LIBRARIES

4.1 Driver Set Overview

The evaluation of programs and libraries is critical in the aim of increasing software performance and reliability. This chapter examines the numerous programs and libraries used during the assessment process, especially in the context of detecting deadlocks. The programs considered in this research are specifically designed to simulate real-world applications. These simulated programs provide an ideal platform for examining and understanding deadlock occurrences by simulating the complex structure and problems of actual software systems. We have developed specialized software components called "drivers" that emulate the utilization of a library by an an actual software system. It is worth noting that users have played a key role in identifying these deadlocks. When users found a deadlock situation, they promptly reported them on the library owner's web page. This joint effort between users and library owners was critical in detecting and analyzing deadlocks in the examined applications. The libraries under consideration in this study are Log4j(4), Commons Logging(5), DBCP(6), Pool(7), and Derby(8).

4.2 Apache Derby

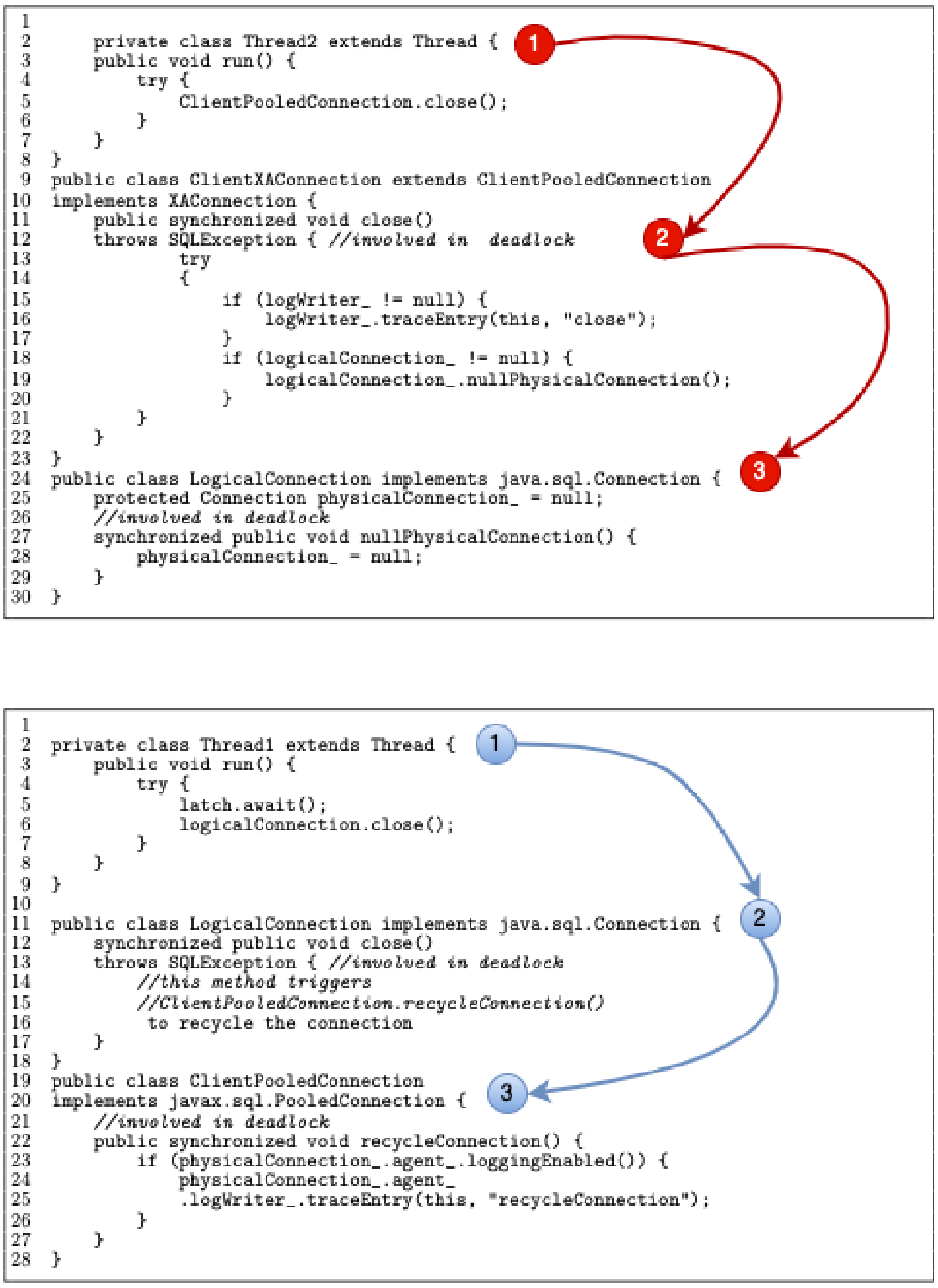
Apache Derby is a relational database management system for online transaction processing that can be easily incorporated in Java programs.(8). Derby is designed to be embedded

TABLE I: TABLE SHOWING THE MAIN USE OF EACH LIBRARY.

Library	Usage
Derby	relational database management system
Log4j	logging library for Java applications
Logging	simple and generic logging abstraction for Java applications
DBCP	manage multiple connection to databases provide by Apache
Pool	manage pools of connections provide by Apache

within an application, meaning that it runs within the same JVM as the application itself. This eliminates the need for a separate database server and simplifies the deployment and administration of the database. Derby is commonly used in scenarios where an application requires a lightweight and portable database solution. It is particularly popular for desktop applications, embedded systems, mobile applications, and small-scale web applications.

This driver simulates the use of a database within an application with multiple connections that are managed by different threads. When a first thread, shown in blue in Figure 9, calls *close()* synchronized method on the LogicalConnection object to close the database connection, the ClientXAConnection object calls *recycleConnection()* synchronized function to notify listeners that the connection can be reused.



```
    private class Thread2 extends Thread {
    public void run() {
        try {
            ClientPooledConnection.close();
        }
    }
}
public class ClientXAConnection extends ClientPooledConnection
implements XAConnection {
    public synchronized void close()
    throws SQLException { //involved in  deadlock
            try
            {
                if (logWriter_ != null) {
                    logWriter_.traceEntry(this, "close");
                }
                if (logicalConnection_ != null) {
                    logicalConnection_.nullPhysicalConnection();
                }
        }
    }
}
public class LogicalConnection implements java.sql.Connection {
    protected Connection physicalConnection_ = null;
    //involved in deadlock
    synchronized public void nullPhysicalConnection() {
        physicalConnection_ = null;
    }
}
```

```
private class Thread1 extends Thread {
    public void run() {
        try {
            latch.await();
            logicalConnection.close();
        }
    }
}

public class LogicalConnection implements java.sql.Connection {
    synchronized public void close()
    throws SQLException { //involved in deadlock
        //this method triggers
        //ClientPooledConnection.recycleConnection()
         to recycle the connection
    }
}
public class ClientPooledConnection
implements javax.sql.PooledConnection {
    //involved in deadlock
    public synchronized void recycleConnection() {
        if (physicalConnection_.agent_.loggingEnabled()) {
            physicalConnection_.agent_
            .logWriter_.traceEntry(this, "recycleConnection");
        }
    }
}
```

Figure 9: Illustration of a deadlock in the library Derby.

Another thread, shown in red in Figure 9, closes the ClientXAConnection connection at the same time with close() synchronized function, that calls logicalConnection.nullPhysicalConnection() synchronized block. This may lead to a deadlock that involves ClientPooledConnection and LogicalConnection objects.We used the JDK 1.6.0.33 and Derby 10.5.1.1 version for this test. This bug was fixed in version 10.9.2.2 of Derby.

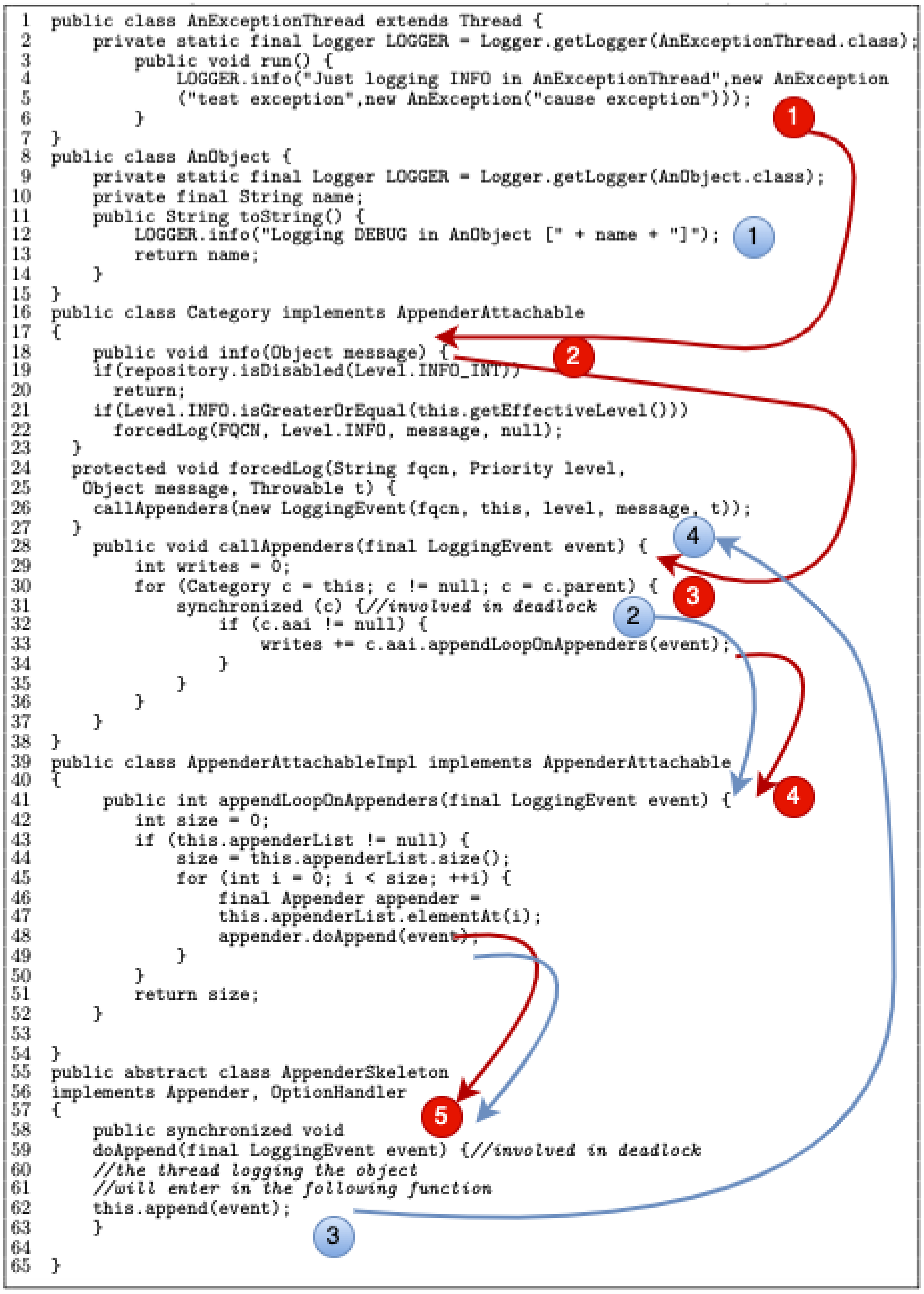
4.3 Log4j

Log4j is a widely used logging library for Java applications (4). It offers a versatile and effective architecture for logging messages at various severity levels, which aids in the tracking and debugging of applications. With Log4j, it is possible to modify logging behavior programmatically or using configuration files. Furthermore it allows to select the output destination for log messages from a variety of options, including console, files, databases, and remote servers. Log4j has strong formatting and filtering tools that let you choose which log messages are displayed and how they are presented. Some well-known apps use this library, including Amazon Web Services, Apple Xcode, Arduino IDE, Netflix, and other programs from Apache and IBM.

The first test case is a multi-threaded application that uses an appender for log. The appender objects are responsible for printing logging messages to various destinations such as consoles, files, sockets, and NT event logs. This deadlock is caused by multiple threads try to log on the same appender. One of those log the info of an object, that have to call the internal method *toString()* to complete the log, while the other log a fixed string. This lead to a deadlock because the two type of log acquire in inverted order the locks on Appender and Category objects. This bug can easily happen when multiple threads are running on the same application and log on the same appender both information about the state of objects and fixed info.

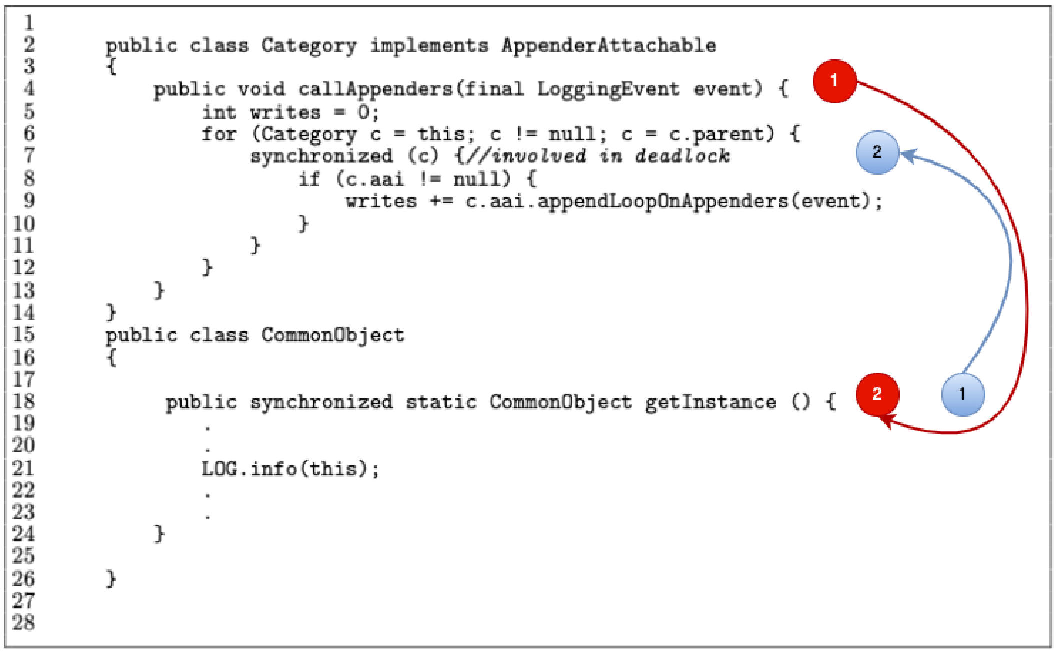
Specifically the first thread, shown in blue in Figure 10, calls *info()* on an object. In this case *appenderSkeleton.doAppen()* synchronized method has been called, then it calls *callAppenders()* which is synchronized on the Category object. The second thread, shown in red in Figure 10, calls the *Category.info()* method as the one before, but just on a fixed string. The deadlock happens because one thread is logging one string while the other is logging the object itself. So the second thread, shown in blue in Figure 10, before it acquires the lock on *callAppenders()*, then the one on *doAppend()*, but then it release the first one and do the conversion of the object to a string and then it reacquires the lock on *callAppenders()*, but in this case the order is inverted and it may cause a deadlock.We used the JDK 18 and log4j 1.2.13 version for this test.

The second driver implements an application scenario where a log statement is called within a synchronized block, which is accessed by multiple threads. This scenario leads to a deadlock, wherein multiple threads simultaneously attempt to log information related to a shared resource inside a synchronized block. The program results in calling in inverted order the synchronized function *getInstance()* and *logger.getLogger()* that calls *Category.callAppenders()*. The flow of the two threads is shown in Figure 11, which has a synchronized statement on the category object. This may lead to a deadlock. This is a problem that existed in log4j 1.x and was fixed in version 2.x.We used the JDK 18 and log4j 1.12.13 version for this test.



```
public class AnExceptionThread extends Thread {
    private static final Logger LOGGER = Logger.getLogger(AnExceptionThread.class);
        public void run() {
            LOGGER.info("Just logging INFO in AnExceptionThread",new AnException
            ("test exception",new AnException("cause exception")));
        }
}
public class AnObject {
    private static final Logger LOGGER = Logger.getLogger(AnObject.class);
    private final String name;
    public String toString() {
        LOGGER.info("Logging DEBUG in AnObject [" + name + "]");
        return name;
    }
}
public class Category implements AppenderAttachable
{
    public void info(Object message) {
    if(repository.isDisabled(Level.INFO_INT))
      return;
    if(Level.INFO.isGreaterOrEqual(this.getEffectiveLevel()))
      forcedLog(FQCN, Level.INFO, message, null);
  }
  protected void forcedLog(String fqcn, Priority level,
   Object message, Throwable t) {
    callAppenders(new LoggingEvent(fqcn, this, level, message, t));
  }
    public void callAppenders(final LoggingEvent event) {
        int writes = 0;
        for (Category c = this; c != null; c = c.parent) {
            synchronized (c) {//involved in deadlock
                if (c.aai != null) {
                    writes += c.aai.appendLoopOnAppenders(event);
                }
            }
        }
    }
}
public class AppenderAttachableImpl implements AppenderAttachable
{
     public int appendLoopOnAppenders(final LoggingEvent event) {
        int size = 0;
        if (this.appenderList != null) {
            size = this.appenderList.size();
            for (int i = 0; i < size; ++i) {
                final Appender appender =
                this.appenderList.elementAt(i);
                appender.doAppend(event);
            }
        }
        return size;
    }

}
public abstract class AppenderSkeleton
implements Appender, OptionHandler
{
    public synchronized void
    doAppend(final LoggingEvent event) {//involved in deadlock
    //the thread logging the object
    //will enter in the following function
    this.append(event);
    }

}
```

Figure 10: Illustration of a deadlock in the library Log4j.



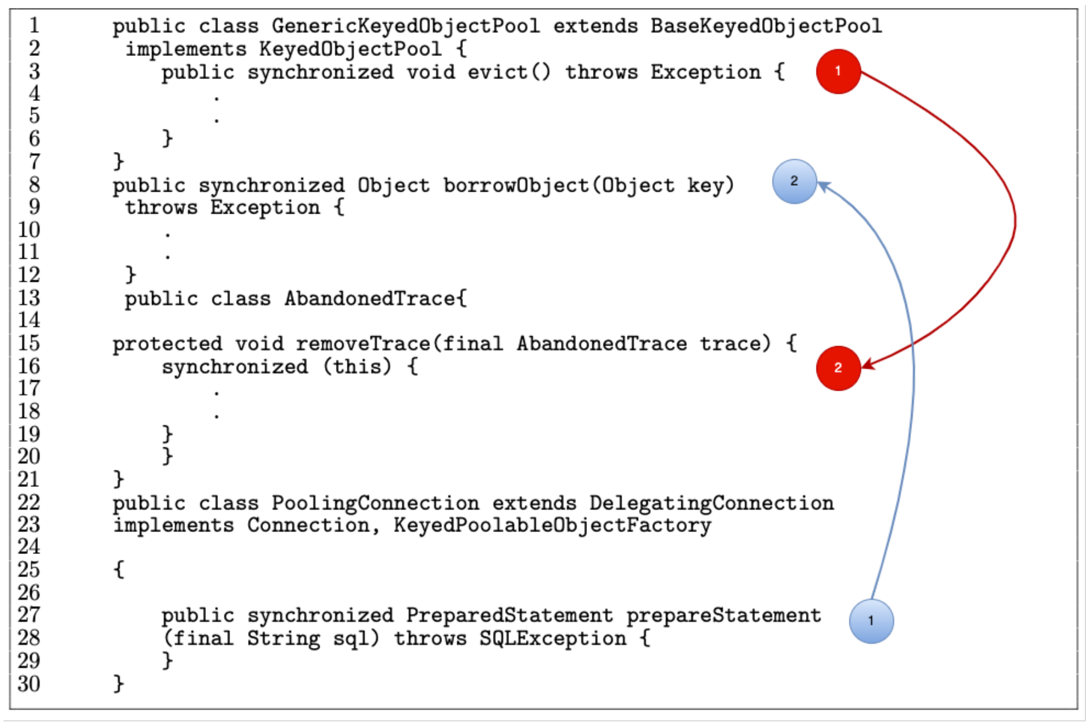
```

public class Category implements AppenderAttachable
{
    public void callAppenders(final LoggingEvent event) {
        int writes = 0;
        for (Category c = this; c != null; c = c.parent) {
            synchronized (c) {//involved in deadlock
                if (c.aai != null) {
                    writes += c.aai.appendLoopOnAppenders(event);
                }
            }
        }
    }
}
public class CommonObject
{

     public synchronized static CommonObject getInstance () {
        .
        .
        LOG.info(this);
        .
        .
    }

}

```

Figure 11: Illustration of a deadlock in the library Log4j.

4.4 DBCP and Pool

DBCP (Database Connection Pooling) is a Java library that provides a standardized and effective method to manage database connections in applications. It is commonly used in Java web applications and other database-intensive applications (6). DBCP permit to construct a pool of previously established database connections that can be reused by various threads or processes in your application. As a result, performance is enhanced because there is no longer any overhead associated with creating a new connection for every database operation. DBCP has the ability to control connections to a variety of database providers, including Oracle, MySQL, PostgreSQL, and more. This library allows developers to minimize the overhead of opening and terminating database connections for each request, resulting in enhanced application performance and scalability. The Pool library provided by Apache provides an object-pooling API and a number of object pool implementations (7). This library manage the reuse of pooled resources, improving efficiency and performance.

The first driver simulates the use of a connection pool to a database using a GenericKeyedObjectPool, an object able to create a map of pool connections. This test is based on the use of prepared statements for a database, a feature that permit to reuse the same query multiple times without re-compiling. This bug has been found in a multi-threaded application working with prepared statements and using the same connection to a database. Our test program is try to reproduce it by using two threads: the first shown in blue in Figure 12, calls synchronized method *prepareStatement()* of PoolingConnection that calls *borrowObject()* synchronized method of GenericKeyedObjectPool. At the same time, the evictor thread, shown in red in Figure 12, is in charge of removing a particular object from the pool, calls the GenericKeyedObjectPool object's synchronized method *evict()*, which in turn calls *AbandonedTrace.addTrace()* method that locks PoolingConnection. This may cause a deadlock. We used the JDK 1.6.0.33 and dbcp 1.2 version for this test. This bug was fixed in version 1.3.

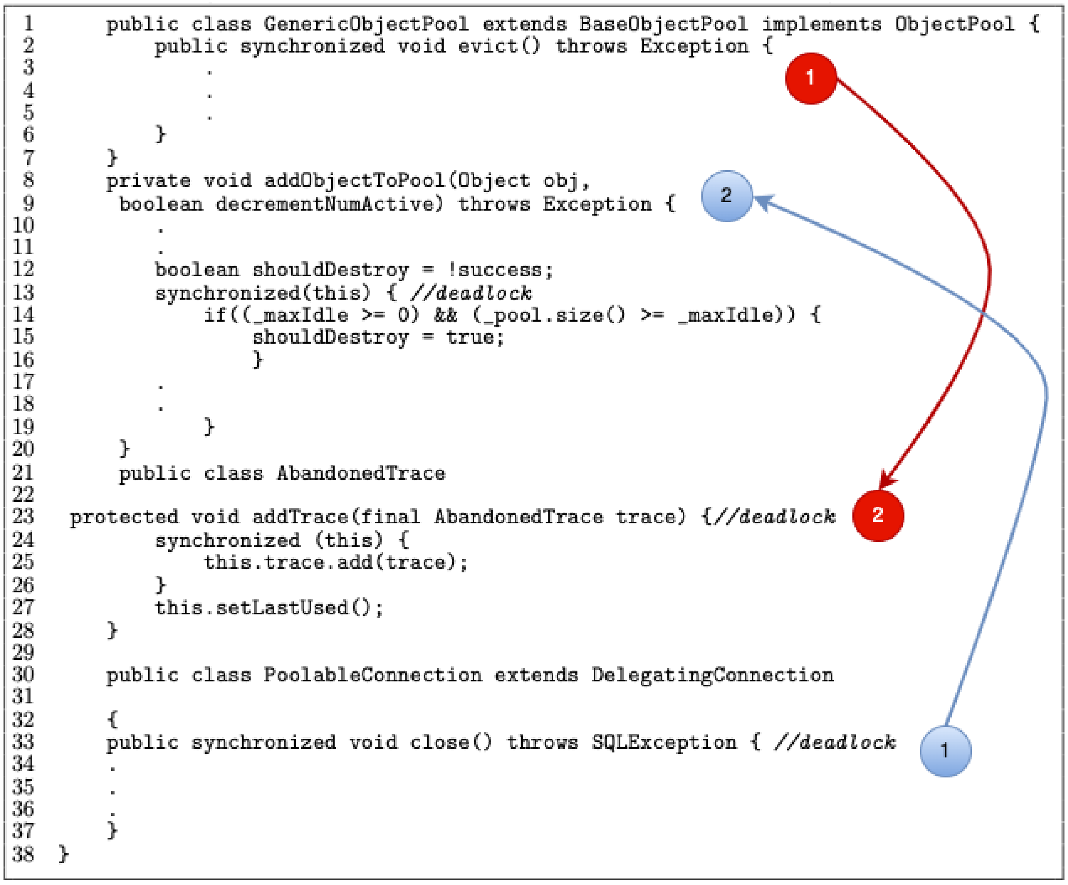


```
public class GenericKeyedObjectPool extends BaseKeyedObjectPool
 implements KeyedObjectPool {
    public synchronized void evict() throws Exception {
        .
        .
    }
}
public synchronized Object borrowObject(Object key)
 throws Exception {
    .
    .
 }
 public class AbandonedTrace{

protected void removeTrace(final AbandonedTrace trace) {
    synchronized (this) {
        .
        .
    }
    }
}
public class PoolingConnection extends DelegatingConnection
implements Connection, KeyedPoolableObjectFactory

{

    public synchronized PreparedStatement prepareStatement
    (final String sql) throws SQLException {
    }
}
```

Figure 12: Illustration of a deadlock in the library DBCP.

The second driver simulates multiple connections to a database by using a PoolableConnectionFactory and a GenericObjectPool. It manages multipe configuration of the pool at the same time that are managed by different threads. While a thread calls *close()* method on PoolableConnectionFactory and an other thread calls the *evict()* method a deadlock may occur. Both methods use synchronized statements on the Trace object, that is in charge of track the connection in use for recovering and reporting abandoned connections, and on GenericObject-

Pool. Specifically, the first Thread, shown in blue in Figure 13, calls close synchronized method on poolableConnection on DBCP library that calls synchronized method *addObjectToPool()* of GenericObjectPool. The second thread, shown in red in Figure 13, calls *evict()* synchronized method of genericObjectPool and then it calls synchronized method *addTrace()* of Abandoned-Trace. The deadlock occurs since poolableConnection extends abandonedTrace. We used the JDK 1.6.0.33 and dbcp 1.2.2 version for this test.This bug was fixed in version 1.3.



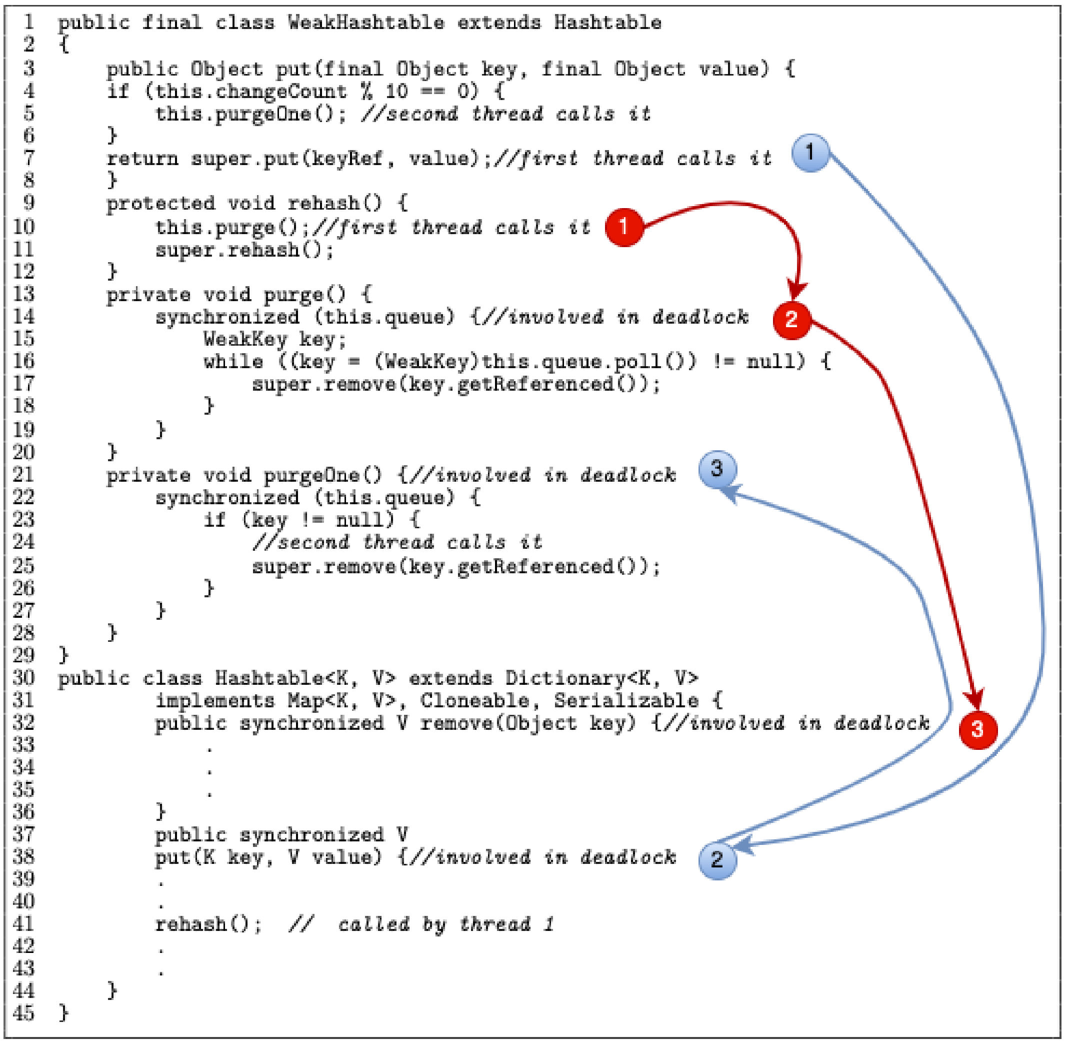
```
    public class GenericObjectPool extends BaseObjectPool implements ObjectPool {
        public synchronized void evict() throws Exception {
            .
            .
            .
        }
    }
    private void addObjectToPool(Object obj,
     boolean decrementNumActive) throws Exception {
        .
        .
        boolean shouldDestroy = !success;
        synchronized(this) { //deadlock
            if((_maxIdle >= 0) && (_pool.size() >= _maxIdle)) {
                shouldDestroy = true;
                }
        .
        .
            }
     }
     public class AbandonedTrace

 protected void addTrace(final AbandonedTrace trace) {//deadlock
        synchronized (this) {
            this.trace.add(trace);
        }
        this.setLastUsed();
    }

    public class PoolableConnection extends DelegatingConnection

    {
    public synchronized void close() throws SQLException { //deadlock
    .
    .
    .
    }
}
```

Figure 13: Illustration of a deadlock in the library DBCP.

4.5 Commons Logging

The Apache Commons Logging library provides a basic and generic logging interface for Java applications(5). It acts as a wrapper for many logging frameworks, including Log4j, allowing developers to write log statements without having to relate their code to a single logging implementation. This library permits flexibility and portability. It allows to switch between different logging systems without changing the code. This adaptability is especially useful when the application must run in a variety of conditions or when it is needed to modify the logging framework utilized in the project.

The driver we wrote consists of multiple threads inserting values inside a hashtable. The program utilizes a WeakHashtable object, which serves the purpose of optimizing the performance of logs generated by a LoggingFactory object. The hashtable plays a crucial role in establishing a connection between the classloader and the logfactory. The utilization of this specific type of hashtable proves to be highly advantageous due to its ability to allow classloaders to be collected by the garbage collector, eliminating the need to invoke LogFactory.release(ClassLoader). However, when a program incorporates different logging configurations, resulting in multiple LoggingFactory objects being added concurrently by various threads, a deadlock situation may arise. This deadlock was initially discovered by a user who encountered the issue while deploying distinct LoggingFactory instances for different web application resources. In our test case we recreate the deadlock with a first thread,shown in blue in Figure 14, that calls *put()* on the WeakHashtable object that calls *put()* synchronized method of Hashtable, that calls *purge()* synchronized method on queue object of WeakHashtable. The second thread, shown in red in Figure 14, calls *put()* on the WeakHashtable object, but this time it calls *purgeOne()* method that it is called every 10 times *put()* function is called. Method *purgeOne()* locks the queue of WeakHashtable and then calls synchronized method *Hashtable.remove()*, this may lead to a deadlock.We used the JDK 18 and Commons Logging 1.1.1 version for this test. This bug was fixed in version 1.2.1.



```
public final class WeakHashtable extends Hashtable
{
    public Object put(final Object key, final Object value) {
    if (this.changeCount % 10 == 0) {
        this.purgeOne(); //second thread calls it
    }
    return super.put(keyRef, value);//first thread calls it
    }
    protected void rehash() {
        this.purge();//first thread calls it
        super.rehash();
    }
    private void purge() {
        synchronized (this.queue) {//involved in deadlock
            WeakKey key;
            while ((key = (WeakKey)this.queue.poll()) != null) {
                super.remove(key.getReferenced());
            }
        }
    }
    private void purgeOne() {//involved in deadlock
        synchronized (this.queue) {
            if (key != null) {
                //second thread calls it
                super.remove(key.getReferenced());
            }
        }
    }
}
public class Hashtable<K, V> extends Dictionary<K, V>
        implements Map<K, V>, Cloneable, Serializable {
        public synchronized V remove(Object key) {//involved in deadlock
            .
            .
            .
        }
        public synchronized V
        put(K key, V value) {//involved in deadlock
        .
        .
        rehash();  //  called by thread 1
        .
        .
    }
}
```

Figure 14: Illustration of a deadlock in the library Commons Logging.

CHAPTER 5

EXPERIMENTAL EVALUATION

In this chapter we discuss the ability of the DDS of detection and resolution of deadlocks run-time. We used a set of the most used libraries of Java that in some version has some vulnerability that causes deadlocks. Calling specific functions of those libraries, we could have evaluate the ability of the DDS in detecting and resolving deadlocks. In addition we run those program using a version of the libraries not affected by deadlocks to evaluate the overhead caused by the DDS in normal conditions. To obtain the non-deadlock version, we employed various approaches. For some of the drivers, we made modifications to the settings to avoid deadlocks, while for others, we used *sleep()* statements strategically to prevent deadlocks from occurring. The test has been runned on a Asus Zenbook with a 1.60GHz intel core I5 8th Gen processor with 4 cores and 8 GB RAM. The computer has installed Ubuntu 22.10 on it.

5.1 Benchmark Set

We ran our set of benchmarks of multithreading applications consisting of drivers that utilize Java well known libraries. The libraries used are Log4j, Commons Logging, DBCP, Pool and Derby. The set is composed of 6 test cases of real bugs found by users in their applications. It comprises the following programs:

- Log4j Driver 1 and driver 2, which perform logging requests by multiple threads.

TABLE II: BENCHMARK DETAILS FOR EACH DRIVER.

Benchmark Name	# Line of Code	# Synchronized statements	# Thread
Log4j Driver 1	1340	11	4
Log4j Driver 2	1425	10	3
Common Logging Driver 1	2080	45	50
DBCP+Pool Driver 1	1870	53	3
DBCP+Pool Driver 2	2340	47	3
Derby Driver	1480	26	3

- Commons Logging driver, which create multiple logging configurations managed by multiple threads.
- DBCP and Pool driver 1 and driver 2, which are the management of pooling connections of multiple databases.
- Derby driver, which manage with multiple threads a database embedded in the Java application.

It is possible to see more details on the single tests in Table II. The table provides insights into various aspects of the tests, such as the number of lines of code (# Line of Code), the number of synchronized statements (# Synchronized statements), and the number of threads (# Thread) running during the execution. All the data of Table II refers to the code executed by the application of both the driver and the library.

5.2 Empirical Results

We conducted our series of benchmark tests, running each benchmark 20 times and recording the average run-time. For the deadlocking benchmarks, we evaluated whether the agent successfully detected and resolved the deadlocks. Additionally, we compared the run-time of the non-deadlocking versions with and without run-time monitoring. During preprocessing, we thoroughly checked for harmful statements to ensure their correct detection. This allowed us to preempt locks during run-time.

For each benchmark, we created two versions: one with deadlocks (DL) and one without (NDL). Similarly, we named with "DDS" the tests in which the DDS approach is used.

The effectiveness of our DDS approach was evaluated by recording the execution time for each benchmark. To minimize differences between the two program versions and ensure accurate run-time measurements, we made minimal modifications, sometimes adding or removing calls to *Thread.sleep()*. However, these modifications may have influenced the execution time to some extent. We recorded the elapsed time and CPU time for each run of the benchmark as shown in Table III.

The agents ran in parallel with the program, and we measured the timing differences and overall performance for each variation. All deadlocks in the deadlocking versions of the benchmarks were successfully detected and resolved.

As shown in Table IV, the non-deadlocking versions exhibited a median increase of 9.5% in CPU time and 5.5% in elapsed time due to monitoring. It is worth noting that the maximum absolute variation in CPU time and elapsed time between using the DDS and not using it was

TABLE III: TIMING FOR THE SELECTED BENCHMARKS.

Benchmark Name	CPU time in S	Elapsed time in S
Log4j Driver 1 DL DDS	0.76	12.41
Log4j Driver 1 NDL DDS	0.89	12.48
Log4j Driver 1 NDL	0.69	12.37
Log4j Driver 2 DL DDS	0.36	0.22
Log4j Driver 2 NDL DDS	0.39	0.34
Log4j Driver 2 NDL	0.36	0.33
Common Logging Driver DL DDS	0.47	0.24
Common Logging Driver NDL DDS	0.43	0.22
Common Logging Driver NDL	0.37	0.19
DBCP+Pool Driver 1 DL DDS	0.68	0.49
DBCP+Pool Driver 1 NDL	0.64	0.45
DBCP+Pool Driver 1 NDL DDS	0.61	0.43
DBCP+Pool Driver 2 DL DDS	0.63	0.56
DBCP+Pool Driver 2 NDL	0.58	0.43
DBCP+Pool Driver 2 NDL DDS	0.59	0.44
Derby Driver DL DDS	0.78	0.57
Derby Driver NDL DDS	0.87	0.58
Derby Driver NDL	0.81	0.53

TABLE IV: PERCENTAGE OF OVERHEAD OF DDS ON DEADLOCK-FREE VERSION OF THE BENCHMARK.

Benchmark Name	% overhead CPU time	% overhead Elapsed time
Log4j Driver 1	22	1
Log4j Driver 2	8	3
Common Logging Driver	13	14
DBCP+Pool Driver 1	5	4
DBCP+Pool Driver 2	2	2
Derby Driver	7	9

130 milliseconds and 110 milliseconds, respectively. These differences are relatively small and can be considered negligible.

It is worth noting that the overhead varies across different tests. There is an expected increase in CPU overhead with a larger number of threads, as observed in the Logging Driver test. This particular test involves 50 threads, each of which contributes to modifications in the DDS graph. On the other hand, tests that create simpler graphs with fewer vertices exhibit an overhead ranging between 2% and 8% in terms of CPU time. Among the tests, the Log4j Driver 1 stands out with the highest recorded CPU overhead. This could potentially be attributed to the randomness introduced by the presence of *sleep()* statements in the code. Remarkably, while the total execution time for this test is around 12 seconds, the actual CPU usage spans between 0.5 and 1 second. This discrepancy suggests that the execution of the application is in a sleep state for the majority of the time. This can potentially lead to the observed higher overhead.

TABLE V: PERCENTAGE OF OVERHEAD OF DDS ON DEADLOCK VERSION AND DEADLOCK-FREE VERSION OF THE BENCHMARK.

Benchmark Name	% overhead CPU time	% overhead Elapsed time
Log4j Driver 1	9	0
Log4j Driver 2	0	-33
Common Logging Driver	21	21
DBCP+Pool Driver 1	5	4
DBCP+Pool Driver 2	10	14
Derby Driver	-4	7

Table V also presents the overhead of the deadlocking versions compared to the deadlock-free versions. Some percentages may be negative due to the presence of *sleep()* statements, which make the two versions not completely comparable. Negative CPU overhead percentages may also occur when the CPU is not performing heavy operations, and threads spend most of the time waiting for others to proceed. This randomness in run-times introduces some variability.

Based on our observations, resolving deadlocks within the deadlocked versions resulted in a mean overhead of 9% in CPU time and 7.6% in elapsed time excluding negative values. These overheads demonstrate the efficiency and effectiveness of our DDS approach in resolving deadlocks in real-world applications.

CHAPTER 6

RELATED WORK

In this chapter we discuss some of the techniques and tools used to detect and handle deadlocks in multicore systems. Deadlocks can be handled using different approaches : detection and resolution, prevention, and avoidance. Potential deadlocks can be detected using dynamic analysis, model checking, run-time monitoring, static analysis and analysis based on lock order graphs or a combination of them.

6.1 Differences between static and dynamic approach

The static approach to deadlock detection involves analyzing the program's source code to identify potential deadlock situations. This is done by examining the program's control flow and resource usage. The goal is to identify patterns in the code that may lead to deadlocks, such as circular dependencies or exclusive resource usage.

On the one hand, one of the main advantages of the static approach is that it can identify potential deadlock situations before the program is executed. This means that developers can take preemptive measures to prevent deadlocks, such as reordering resource acquisition or using different synchronization primitives. However, the static approach has some limitations. It cannot detect all possible deadlock situations, and it may produce false positives or false negatives and often has scalability problems.

On the other hand, the dynamic approach to deadlock detection involves monitoring the program's execution at run-time to identify actual deadlock situations. This is done by tracking the state of the threads and resources in the program and detecting when threads are blocked and waiting for resources that are held by other threads.

The dynamic approach can detect all possible deadlock situations, including those that may not be evident in the program's source code. It can also provide more accurate information about the cause of the deadlock, such as the exact sequence of events that led to the deadlock. However, the dynamic approach has some limitations. It can be computationally expensive, and it may not be able to detect deadlocks that occur in rare or infrequent situations.

In terms of resolution, both approaches have their advantages and disadvantages. The static approach allows developers to take preemptive measures to prevent deadlocks before they occur, while the dynamic approach can provide more accurate information about the cause of the deadlock and help developers to fix the problem after it occurs. Ultimately, the choice between static and dynamic approaches to deadlock detection and resolution depends on the specific needs and requirements of the system being developed.

6.2 Deadlock Prediction and Detection

Deadlock prediction involves analyzing a program's source code or system design to identify potential deadlock situations before they occur. This approach tries to anticipate deadlock by analyzing the sequence of operations and the resources used in the program. Deadlock detection involves monitoring the execution of a program at run-time to identify actual deadlock situations. This approach tracks the state of threads and resources in the program and detects when threads are blocked and waiting for resources that are held by other threads. In this section we examine related works able to predict and detect deadlock occurences, but they are not able to solve it, unlike DDS, which resolve the deadlock after detecting it.

6.2.1 Static analysis

Static detection tools aim to identify deadlocks by analyzing the source code. However, it is worth noting that these tools may generate false positive results and frequently encounter scalability problems when applied to large programs. RacerX (17) is a tool that detects both race situations and deadlocks via flow-sensitive interprocedural analysis, but manual annotatiosn are needed to make it work properly. Extended Static Checking (18) is a compile-time checker used to find common programming errors. One of the responsabilities of the tool is to detect possible deadlocks or race conditions. The tool presented in (19) mixes techniques of cycle graph analysis and a system with tuples to represent deadlock. However, this is able only to detect deadlock between two threads and two locks. Williams et al. (20) presented a tool that uses a flow-sensitive analysis with the task to detect deadlocks in Java libraries, but it does not work with reentrant locks unlike DDS. Similarly the tool presented by Shanbhag et al. (21) is aimed to find deadlocks in large Java libraries using a mixed approach between static and dynamic analysis.

Bogor (22) is a deadlock detector based on model checking. It builds a model of the program by studying all the possible states given the shared and global variables and this permit to detect possible deadlock. However tools that apply model checking are not scalable because of the exponential growth in the number of possible states that a system can reach as the size or complexity of the system increases. Similarly, Gadara (23) is a tool based on model checking; this approach can detect all possible deadlocks, but also some false positives, a type of error where a test result incorrectly indicates the presence of a deadlock, when in fact it is not present.

Jade (24) is a contex-sensitive tool that can predict all two-thread deadlocks, however it can detect also some false positive. Brotherston et al. (25) propose a context-insensitive detector for Android application written in Java; this method analyses application from the changed files and their dependencies, to improve the time needed. It detects all potential deadlock candidates, including false positives. Kamburjan (26) presents an automatic deadlock detector for synchronization on arbitrary boolean conditions. The tool mixes a deductive verification approach with the static analysis. The application can detect only deadlock caused by faulty system design.

Peahen (27) is a deadlock detection method for C and C++ based on a context-insensitive lock-graph analysis that encode only essential information about lock acquisition, that information is enriched by an algorithm that progressively refines the deadlock cycles in the lock graph only for a few interesting calling contexts. This approach allows the tool to be scalable, fast and precise. However it can report false positives. Metcalf and Yavuz (28) propose a tool based on regression analysis to detect deadlocks; their method finds the code changes that involve locks acquisition in an inappropriate order. This approach can report false positives.

6.2.2 Dynamic analysis

Dynamic detection tools detect deadlocks by observing events from real executions; these methods have usually more probability to find real deadlocks but still have problems with false positives. ConLock+ (29) is a tool that monitors and detects possible deadlocks during run-time. MagicLock (30) builds an optimized graph pruning the locks that can not create a cycle, then it uses a DFS algorithm to identify if deadlocks can be present. This tool is efficient and scalable in detecting deadlocks in large programs. AirLock (31) improves the previous detection algorithm thanks to the use of a reachability graph. This allows this tool to determine fast if a cycle is present between two nodes. GoodLock (32) constructs a lock order graph based on locks acquisition order during run-time and then detect cycles inside the graph to identify possible deadlocks. However this detector can only find deadlock between two threads. MagicFuzzer (33) is a very efficient and scalable tool used for C and C++ programs used to detect deadlocks.

In an effort to minimize the time overhead of the detector algorithm, the MulticoreSDK (34) employs a location-based lock order graph. This graph categorizes locks acquired from various threads within the same code location into groups, which are subsequently merged if they share a lock. Another dynamic analyzer, UnHang (35), is utilized for analyzing C and C++ programs. Notably, UnHang employs an optimized lock graph that leverages a per-thread recording approach. This eliminates the need for introducing additional locks when updating generalized dependencies for each thread. UnHang can detect deadlocks created by condition variables and mutex locks, however it can detect some false positives. CheckMate (36) uses a model checking algorithm that is used to analyze Java programs, it needs manual annotations and cannot detect all deadlocks or report false positive.

6.3 Deadlock Detection and Recovery

In this section we analyse some tools that are able to detect a deadlock and solve it. To do this some different approaches have been used; an option is the preemption of a resource, this technique consists in taking one of the resources from the resource owner, in our case a thread, and give it to another process in the hope that it will finish the execution and release the resource sooner. The choice of the resource represent a challenge because not always is possible to solve the deadlock without affecting the program state. ConAir (37) is a tool for save C and C++ programs from failure caused by bugs, it can resolve deadlocks using a timeout associated to each lock, when it is reached, it preempts and re-executes the victim thread.

Another technique is the rollback to a safe state, that consist in the ability of the operating system to restore the state of the program to a previous safe state, when it detect a deadlock. The tools using this techniques has to implement a checkpoint system that records some of the states of the program and all the resources in that moment, in this way when a deadlock occurs it can reverse all the modification and return to a prior safe state. This technique does not garantee that deadlocks will not happens in future runs and it usually produces significant memory and computational overhead. An exaple of tool using this method is Sammati (38), that is built for POSIX threads and use rollback to set the state of a thread involved in a deadlock to the moment of the lock acquisition. In this way it allows the other thread involved to continue its execution. The choice of the thread to rollback is arbitrarly. Similarly Rx (39) is a tool that performs rollback of a thread to a checkpoint and run it again changing the environment based on the failure analysis.

6.4 Deadlock Prevention

In this section we analyse some tools for deadlock prevention, that is a set of strategies used to prevent deadlocks occurences by ensuring that the necessary conditions for a deadlock to occur are never met. Deadlock prevention techniques refer to violating any one of the four necessary conditions : mutual exclusion, hold and wait, no preemption and circular wait.

UnDead (40) is a deadlock detection and prevention dynamic analyzer for C++ with various optimizations. UnDead's detector activate only if the program has more then 1 thread and involves more than one lock per thread to reduce its performance overhead. It has a similar prevention approach as other C/C++ tools like Grace (41) and Click-5 (42), since they both detect deadlocks in current execution and prevent these deadlocks in future executions by breaking the condition of mutual exclusion by simulating a sequential single threaded program execution. UnDead's detection is based on the Depth First Search algorithm. The cycles detected by the aforementioned approaches are predictive, they can detect false positives. Another tool used to prevent deadlocks is the one developed by Botlagunta et al. (43). It uses a resource reservation system that estimate the optimal number of resources required for a deadlock free resource reservation policy. A similar approach is used by Bättig in his research on the Synchronized-By-Default(44) concurrency model, it prevents deadlock by ensuring that only one thread per time can access a shared resource and execute the related code atomically, in this way the mutual exclusion condition is broken and deadlocks are prevented.

6.5 Deadlock Avoidance

Deadlock avoidance is a set of techniques similar to deadlock prevention, but less strict since it is not necessary to break one of the four conditions. It is aimed to ensure that the system has always enough resources to avoid the possibility of a deadlock. This is usually achieved by employing algorithms that dynamically allocate resources to processes in a way that ensures that a cycle of dependencies cannot occur, which is the root cause of deadlocks. Deadlock avoidance is related to the safe state concept, that is a state of the system in which deadlocks cannot occur, since the system knows for each thread the resources allocated for it, the maximum number of resources each thread needs and the number of resources currently available.

If the operating system possesses the capability to allocate or fulfill the maximum resource demands of processes in any sequence, the system is said to be in safe state. However, an unsafe state arises when the operating system fails to prevent processes from requesting resources, potentially resulting in a deadlock. It is important to note that being in an unsafe state does not necessarily guarantee the occurrence of a deadlock. Deadlock avoidance techniques make sure that every thread that allocate a resource does not change the safe state of the system. The majority of the tools uses the banker's algorithm or an improved version, the idea behind it is similar to the regular amount allocation of the bank. To illustrate, consider a scenario within a banking system where a customer submits a request to allocate funds. The bank initiates a verification process to determine if the requested amount is available and can be allocated. Only when this condition is met can the bank proceed to fulfill requests from other customers.

If the condition is satisfied, the requested allocation is processed accordingly. However, if the condition is not met, the customer is required to wait until the funds become available. The algorithm works in the same way, but with the allocation of resources to the different threads.

The tool developed by Martin et. al (45) uses the banker's algorithm with a selection to optimize the allocation process. Similarly, another tool developed by Bemug et al. (46) uses an improved version of the algorithm with a system based on a linked list of threads sorted in increasing order based on the maximum number of resources needed. Therefore the threads with lower resources requirements are executed before the others. Dimmunix (47; 48) is a tool for Java for deadlock avoidance that relies on preventing the reoccurrence of a previous deadlock pattern. It avoid deadlocks by saving inside a database the part of code that cause the deadlock and then it uses a thread suspension mechanism when a deadlock can occur.

CHAPTER 7

CONCLUSION

7.1 Conclusion

This research has explored the problem of deadlocks within Java libraries, presenting a comprehensive evaluation of their occurrence. Through investigation and analysis, we identified a set of deadlocks in widely-used Java libraries. To facilitate the study and the understanding of these deadlocks, we developed specialized drivers, providing a controlled environment for their recreation and examination.

We used the DDS methodology as solution to address these deadlock issues, with the aim of detecting and resolving them during run-time within real-world applications. It was acknowledged that not all deadlocks could be resolved if a victim thread could not be identified. Preempting a lock from a thread could potentially disrupt the application's consistency in these cases. However, within the scope of this research, no harmful statements were found in the considered test cases. It was concluded that such occurrences in real-world applications are rare, as proficient programmers typically ensure the acquisition of all necessary locks before commencing their work.

The DDS approach is different from existing methodologies in its distinctive manner of rectifying and fixing detected deadlocks. Unlike traditional approaches that involve suspending involved threads and rolling back to a safe execution point, DDS relied on resource preemption from a thread that had not modified a shared object. This approach support better performance in terms of overhead. Moreover, the automatic detection and resolution of deadlocks were accomplished without human intervention, demonstrating the efficacy and self-sufficiency of the DDS approach.

Experimental results supported the scalability of DDS and revealed minimal overhead with an average of 9% for CPU time. This research demonstrated that DDS could effectively operate within the realm of libraries and real-world applications. By utilizing this approach, real deadlocks reported by users in their applications were successfully resolved, showcasing the practicality and relevance of DDS in addressing real-life challenges.

In conclusion, this research contributes to show the complexities of deadlocks within Java and his libraries. By identifying and understanding the occurrence of deadlocks, and subsequently applying the DDS methodology, we showed a robust solution for detecting and resolving deadlocks in Java applications. The findings of this research contribute to the advancement of the field, offering developers a reliable tool to tackle deadlock issues in their Java applications, ultimately enhancing their performance, reliability, and user experience.

CITED LITERATURE

1. Aldakheel, E. A.: Deadlock Detector and Solver (DDS). 12 2019.

2. Jie Chen, William Watson, W. M.: Multi-threading performance on commodity multi-core processors. 2007.

3. Scaler: Deadlock in OS. 2022. https://www.scaler.com/topics/operating-system/deadlock-in-os/.

4. Apache: Apache Log4j. 2023. https://logging.apache.org/log4j/2.x/.

5. Apache: Apache Commons Logging. 2014. https://commons.apache.org/proper/commons-logging/.

6. Apache: Apache Commons DBCP. 2021. https://commons.apache.org/proper/commons-dbcp/.

7. Apache: Apache Commons Pool. 2021. https://commons.apache.org/proper/commons-pool/.

8. Apache: Apache Derby. 2022. https://db.apache.org/derby/.

9. Nikita Goel, Vijaya Laxmi, A. S.: Handling multithreading approach using java. International Journal of Computer Science Trends and Technology (IJCST), 3, 2015.

10. Sara Sharifirad, H. H.: A formal framework for specifying concurrent systems. International Journal of Computer Applications, 68(1), 2013.

11. TIOBE Index for May 2023. TIOBE, 2023. https://www.techrepublic.com/article/tiobe-index-language-rankings/.

12. Goetz, B.: Java Concurrency in Practice. Pearson Education, 2010.

13. Maurice Herlihy, N. S.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann, 2012.

14. ReentrantLock (Java Platform SE 8). Oracle, 2023.

15. JVM Tool Interface. Oracle, 2023. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.

16. Madsen, M., Tip, F., and Lhoták, O.: Static analysis of event-driven node.js javascript applications. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, page 505–519, New York, NY, USA, 2015. Association for Computing Machinery.

17. Engler, D. and Ashcraft, K.: Racerx: Effective, static detection of race conditions and deadlocks. volume 37, pages 237–252, 01 2003.

18. Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R.: Extended static checking for java. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02, page 234–245, New York, NY, USA, 2002. Association for Computing Machinery.

19. Naik, M., Park, C.-S., Sen, K., and Gay, D.: Effective static deadlock detection. In 2009 IEEE 31st International Conference on Software Engineering, pages 386–396, 2009.

20. Williams, A., Thies, W., and Ernst, M. D.: Static deadlock detection for java libraries. In ECOOP 2005 - Object-Oriented Programming, ed. A. P. Black, pages 602–629, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

21. Shanbhag, V. K.: Deadlock-detection in java-library using static-analysis. 15th Asia-Pacific Software Engineering Conference, pages 361–368, 2008.

22. Mahdian, F., Rafe, V., and Rafeh, R.: A framework to automatic deadlock detection in concurrent programs. Przegld Elektrotechniczny, 88(1b):182–184, 2012.

23. Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., and Mahlke, S. A.: Gadara: Dynamic deadlock avoidance for multithreaded programs. 8:281–294, 2008.

24. Naik, M., Park, C.-S., Sen, K., and Gay, D.: Effective static deadlock detection. In 2009 IEEE 31st International Conference on Software Engineering, pages 386–396, 2009.

25. Brotherston, J., Brunet, P., Gorogiannis, N., and Kanovich, M.: A compositional deadlock detector for android java. In Proceedings of ASE-36. ACM, 2021.

26. Kamburjan, E.: Detecting deadlocks in formal system models with condition synchronization. Electronic Communications of the EASST, 76, 2019.

27. Cai, Y., Ye, C., Shi, Q., and Zhang, C.: Peahen: Fast and precise static deadlock detection via context reduction. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, pages 784–796, New York, NY, USA, 2022. Association for Computing Machinery.

28. Metcalf, C. A. and Yavuz, T.: Detecting potential deadlocks through change impact analysis. Software Quality Journal, 26(3):1015–1036, Sep 2018.

29. Yan, C. and Quiong, L.: Dynamic testing for deadlocks via constraints. IEEE Transactions on Software Engineering, 42(9):825–842, September 2016.

30. Cai, Y. and Chan, W.: Magiclock: Scalable detection ofpotential deadlocks in large- scale-multithreaded programs. IEEE Transactions on Software Engineering, 40(3):266–281, March 2014.

31. Cai, Y., Meng, R., and Palsberg, J.: Low-overhead deadlock prediction. In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pages 1298–1309. IEEE, 2020.

32. Bensalem, S. and Havelund, K.: Dynamic deadlock analysis of multi-threaded programs. In Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing, HVC'05, pages 208–223. Springer-Verlag, 2005.

33. Cai, Y. and Chan, W. K.: Magicfuzzer: Scalable deadlock detection for large-scale applications. In Proceedings of the 34th International Conference on Software Engineering, pages 606–616. IEEE Press, 2012.

34. Luo, Z. D., Das, R., and Qi, Y.: Multicore sdk: A practical and efficient deadlock detector for real-world applications. 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, pages 309–318, 2011.

35. Zhou, J., Yang, H., Lange, J., and Liu, T.: Deadlock prediction via generalized dependency. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, pages 455–466. Association for Computing Machinery, 2022.

36. Joshi, P., Naik, M., Sen, K., and Gay, D.: An effective dynamic analysis for detecting generalized deadlocks. pages 327–336, 07 2010.

37. Zhang, W., de Kruijf, M., Li, A., Lu, S., and Sankaralingam, K.: Conair: Featherweight concurrency bug recovery via single-threaded idempotent execution. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 113–126, New York, NY, USA, 2013. Association for Computing Machinery.

38. Pyla, H. K. and Varadarajan, S.: Deterministic dynamic deadlock detection and recovery. ACM, page 44, 2012.

39. Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y.: Rx: Treating bugs as allergies - -a safe method to survive software failures. volume 25, pages 235–248, 01 2005.

40. Zhou, J., Silvestro, S., Liu, H., Cai, Y., and Liu, T.: Undead: Detecting and preventing deadlocks in production software. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE '17, pages 729–740. IEEE Press, 2017.

41. Berger, E. D., Yang, T., Liu, T., and Novark, G.: Grace: Safe multithreaded programming for c/c++. SIGPLAN Not., 44(10):81–96, oct 2009.

42. Frigo, M., Leiserson, C. E., and Randall, K. H.: The implementation of the cilk-5 multithreaded language. Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, 1998.

43. Madhavi Devi Botlagunta, S. A. and Rajeswara, R.: A novel resource management technique for deadlock-free systems. PMC, pages 627–635, 2021.

44. Bättig, M.: Efficient Synchronized-by-Default Concurrency. Doctoral thesis, ETH Zurich, Zurich, 2019.

45. Martin, M., Grounds, N., Antonio, J., Crawford, K., and Madden, J.: Banker's deadlock avoidance algorithm for distributed service-oriented architectures. pages 43–50, 01 2010.

46. Begum, M., Faruque, O., Miah, M. W. R., and Das, B.: An improved safety detection algorithm towards deadlock avoidance. pages 73–78, 04 2020.

47. Jula, H., Tralamazza, D., Zamfir, C., and Candea, G.: Deadlock immunity: Enabling systems to defend against deadlocks. 01 2008.

48. Jula, H., Andrica, S., and Candea, G.: Efficiency optimizations for implementations of deadlock immunity. In Runtime Verification, eds. S. Khurshid and K. Sen, pages 78–93, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

VITA

NAME	Marco Milani
EDUCATION	M.S.,Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2023 (expected)
	B.S.,Computer Engeneering, Politecnico of Turin , Turin , Italy, 2021
PUBLICATIONS	