Deadlock Detector and Solver (DDS)

by

Eman Abdullah Aldakheel
B.S., Computer Science, Imam Abdulrahman Bin Faisal University, 2006
M.S., Computer Science, Bowling Green State University, 2011

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2019

Chicago, Illinois

Defense Committee:
Ugo Buy, Chair and Advisor
Ajay Kshemkalyani
Prasad Sistla
Venkat Venkatakrishnan
Jalal Alowibdi, University of Jeddah

To my family and all who supported me in my studies

To my parents, my husband, and my kids (Linda, Abdullah, Lama and Rakan)

To every seeker of knowledge

I dedicate this research

ACKNOWLEDGMENT

I would first like to thank my supervisor, Prof. Ugo A. Buy, who provided his unwavering support, guidance and inspiration to write this dissertation. The door to Prof. Ugo A. Buy was always open whenever I ran into trouble or had a question about the research. He consistently steered me in the right direction and without his constant support and encouragement, the dissertation would not have been completed. Prof. Buy is a man with capability whom you instantly love after meeting even for the first time and never forget. Throughout the duration he was my supervisor for the dissertation, I found him a great coach who always advised me in my favour and interest. His professionalism and positive attitude always inspired me. I look forward to becoming dynamic, passionate, and enthusiastic like him and also able to lead an audience of students as he does. I am extremely thankful to him for his scientific knowledge and advice that he gave during a number of insightful debate and suggestions regarding the research. He has been very resourceful in completing my dissertation. He is my main source of scientific information that assisted me in getting work done in a short time, through responding all my questions and concerns.

I would also like to show my earnest gratitude and appreciation to Prof. Ajay Kshemkalyani, Prof. Prasad Sistla, Prof. Venkat Venkatakrishnan, and Prof. Jalal Alowibdi to be part of my dissertation committee and assisting me with intuitive feedback and inspiring discussion.

I would like to especially express gratitude to the king of the kingdom of Saudi Arabia for providing me with an opportunity of attaining the scholarship offered by the ministry of

higher education and Princess Nourah Bint Abdulrahman University that is organized and represented by the Saudi Arabia Cultural Mission. I am also obliged for the financial support provided by the Department of Computer Science – University of Illinois at Chicago. I am very grateful to my father Abdullah as well as my mother Amira for their utmost support during the dissertation. It became possible due to the big support from the education system in Saudi Arabia, without which I would not be able to defend my thesis today.

Today, with the courage and support provided by my country and my family, I have attained the highest degree with thorough knowledge in computer science. Therefore, I intend to take this knowledge and education back to my homeland so that I can make a contribution in developing and educating the Saudis.

I express my gratitude and my unlimited love to my parents and family who have dedicated their lives for me and provided absolute love and total care. Without their unconditional love and care, I would not have made it this far. My family is always by my side even in a rough time. Today, I am able to stand on my feet due to many experience and pledges that make my life dierent associated with awareness and the quality of education which I believe that I should deliver back to my country. I am always willing to try and lead from my mistake through constant self-learning in computer science so as to keep me knowledgeable and updated. Hence, this degree does not mean to prevent me from acquiring more knowledge to be part of the changing world.

ACKNOWLEDGMENT (Continued)

In the end, I would like to express my love and gratitude to my family, especially my husband Mansour, who accompanied me all the time away from home to get my master's and doctoral degrees. It is because of their love, support, and belief in me that I have succeeded today.

EAA

CONTRIBUTION OF AUTHORS

Chapter 1 introduces the thesis problem, approach, challenges and the contributions of this thesis.

In **Chapter 2**, we present background information on Java mutual exclusion including the differences between intrinsic and reentrant locks. In addition, this chapter introduces the concept of Software Transactional Memory (STM).

Chapter 3 highlights the related work and how previous methods differ from our approach.

Chapter 4 presents the preprocessing algorithm for our supervisory controller methodology; the algorithm identifies synchronization points in a given source code.

Chapter 5 illustrates runtime monitoring, which includes two major phases, the detection phase and the resolution phase.

Chapter 6 explains the implementation details of the Deadlock Detector and Solver (DDS). It also discusses selected benchmarks to evaluate the effectiveness of the DDS. This chapter presents performance results of DDS in terms of execution speed by calculating the imposed runtime overhead of using DDS.

Chapter 7 illustrates how DDS methodology can be applied to any programming language other than Java. We focus on DDS application in C++ language. In addition, we present how the deadlocks are handled in distributed systems and if we can apply our methodology in such systems.

Finally, **Chapter 8** provides my conclusions and the future research work.

CONTRIBUTION OF AUTHORS (Continued)

The following publications are based on dissertation chapters:

- (Eman Aldakheel, and Ugo Buy. "Efficient Run-time Method for Detecting and Resolving Deadlocks in Java Programs." *In 33rd European Conference on Object-Oriented Programming Workshops (ECOOP),* July 2019). I was the primary author and contributor. My advisor, Ugo Buy, contributed to the writing of the manuscript as well as the planning of the work.
- (Eman Aldakheel. "Deadlock Detector and Solver (DDS)." *In ICSE 18 Companion: 40th International Conference on Software Engineering Companion,* New York, NY, USA: ACM, pp. 512–514, May 2018.). I was the only author under the supervision of my advisor, Ugo Buy.
- (Eman Aldakheel, Ugo Buy, and Simran Kaur. "DDS: Deadlock Detector and Solver." *In 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW),* IEEE, pp. 216-223, Oct 2018, doi:10.1109/ISSREW.2018.00009.). I was the primary author and contributor. Simran Kaur contributed to structuring the work in the work early stages. My advisor, Ugo Buy, contributed to the writing of the manuscript as well as the planning and structure of the work.

TABLE OF CONTENTS

CHAPTER		PAGE
1	**INTRODUCTION**	1
1.1	The Deadlock Problem	2
1.1.1	Background Information	2
1.1.2	Illustrative Examples of the Problem	4
1.1.3	Significance of the Problem	6
1.2	Overview of the Approach	7
1.2.1	Our Supervisory Controller	8
1.2.1.1	Detector Algorithm	8
1.2.1.2	Solver Algorithm	9
1.2.1.3	Preprocessing	10
1.2.2	Existing Approaches	12
1.3	Challenges	13
1.4	Contributions	14
2	**BACKGROUND**	17
2.1	Java Mutual Exclusion	17
2.1.1	Intrinsic Locks	17
2.1.2	Reentrant Locks	18
2.1.3	Difference between Intrinsic and Reentrant Locks	20
2.1.3.1	Test for Availability	21
2.1.3.2	Non-Block Structured Locking	22
2.1.3.3	Fairness	22
2.1.3.4	Reentrancy	23
2.2	Software Transactional Memory	25
3	**RELATED WORK**	27
3.1	Deadlock Prediction and Detection	28
3.1.1	Static Approaches	28
3.1.2	Dynamic Approaches	32
3.2	Deadlock Recovery, Prevention, and Avoidance	35
3.2.1	Deadlock Recovery	35
3.2.2	Deadlock Prevention	36
3.2.3	Deadlock Avoidance	37
4	**DDS PREPROCESSING**	40
4.1	Architecture	41
4.1.1	Architecture Components	42

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

4.1.2 Building the Tree . . . 44
4.2 Identifying Harmful Statements . . . 45
4.2.1 Handling Loops and Branches . . . 47
4.2.2 Illustrative Examples . . . 50
4.2.2.1 Intrinsic Locks . . . 50
4.2.2.2 Reentrant Locks . . . 50
4.3 Guarding Harmful Statements . . . 53
4.3.1 Conceptual Overview of DeuceSTM . . . 53
4.3.2 Applying DeuceSTM to the Presented Example . . . 55

5 DDS RUNTIME . . . 57
5.1 Runtime Architecture Components . . . 58
5.1.1 Observer . . . 59
5.1.2 Detector . . . 60
5.1.2.1 DDS Graph . . . 61
5.1.3 Solver . . . 61
5.2 The Detector Algorithm . . . 62
5.3 The Solver Algorithm . . . 67

6 EXPERIMENTAL EVALUATION . . . 70
6.1 Implementation Details . . . 71
6.1.1 Runtime Monitoring . . . 72
6.1.2 Preprocessor . . . 76
6.2 Benchmark Sets . . . 77
6.3 Empirical Results . . . 80
6.3.1 DDS Preprocessing Evaluation . . . 80
6.3.2 DDS Runtime Evaluation . . . 82
6.3.2.1 Dining Philosophers Examples . . . 82
6.3.2.2 Real-World Applications . . . 84
6.3.2.3 Comparison with Other Deadlock Detectors . . . 92
6.3.3 Communication Cost for Runtime Monitoring . . . 94
6.3.4 Limitation . . . 95

7 APPLICATION OF DDS . . . 96
7.1 Application of DDS in a Multicore Environment to Any Other Language . . . 96
7.1.1 Applying DDS to C++ . . . 98
7.2 Application of the DDS to a Distributed System . . . 99
7.2.1 Deadlock-Handling Strategies in Distributed Systems . . . 101
7.2.2 Applying DDS to Distributed Systems . . . 103

8 CONCLUSION AND FUTURE WORK . . . 107

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

8.1 Conclusion 107
8.2 Future Work 109

CITED LITERATURE 111

VITA 120

LIST OF TABLES

TABLE		PAGE
II	STATISTIC DETAILS OF THE CHOSEN BENCHMARK.	78
III	*DDS* PREPROCESSOR TIMINGS FOR THE CHOSEN BENCHMARKS.	81
IV	DINING PHILOSOPHERS' PROGRAMS TIMING MEASUREMENT.	83
V	REAL-WORLD APPLICATIONS TIMING MEASUREMENT. .	86
VI	COMPARISON OF TIMING FOR DIFFERENT DEADLOCK DETECTOR TOOLS.	93
VII	COMMUNICATION COST FOR USING CONTENDED CALLBACKS.	95

LIST OF FIGURES

FIGURE		PAGE
1	Deadlock scenario for the dining philosophers' problem.	5
2	A simple banking transaction.	6
3	Abstract overview of DDS runtime monitoring.	7
4	Overview of DDS approach including runtime components and preprocessing stage.	9
5	Intrinsic locks example in Java.	19
6	Reentrant locks example in Java.	20
7	An example of using trylock() in reentrant locks in Java.	21
8	Reentrancy issue using intrinsic locks.	24
9	Faulty system design example in Java.	30
10	Implementation bug example in Java.	31
11	DDS preprocessing architecture.	41
12	DDS flowchart for building the tree during preprocessing stage.	43
13	Algorithm for building preprocessing's tree.	45
14	Tree example with harmful statements.	49
15	A complete example on reentrant locks in Java.	51
16	DDS runtime architecture.	58
17	DDS high level layered architecture overview.	71
18	DDS architecture to monitor program written in Java Language.	74

LIST OF FIGURES (Continued)

FIGURE		PAGE
19	Extended DDS preprocessing architecture.	75
20	DDS average percentages of overheads with varying number of threads along with two input size sets using STM.	91

LIST OF ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control Flow Graph
DDS	Deadlock Detector and Solver
DFS	Depth First Search
ETH	Eidgenössische Technische Hochschule
JNI	Java Native Interface
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Toolset Interface
pthread	POSIX Thread
QVM	Quality Virtual Machine
SBD	Synchronized-By-Default
STM	Software Transactional Memory
TJ	Transitive Joins
TMTS	Transactional Memory Technical Specification
WFG	Wait for Graph

SUMMARY

Deadlock is found to be one of the most complex problems that can negatively impact the reliability of programs. If deadlocks are not detected and resolved, this can cause permanent thread blockage. Typically, current deadlocks' resolver are based on computation rollback and timeout, which leads to significant delays and inefficient utilization of resources. Therefore, this research presents Deadlock Detector and Solver (DDS), which has the capability to effectively detect and resolve deadlocks in Java programs without requiring code annotations and with a modest performance overhead. DDS depends on a supervisory controller that aims at monitoring program execution. DDS efficiently detects deadlocks caused by hold-and wait cycles on monitor locks. Unlike existing deadlock detectors and solvers, upon detecting a deadlock, DDS employs a preemptive approach to break up the deadlock. Hence, the aim of this empirical research is to detect and resolve deadlocks at runtime. The research aims to assess the effectiveness of DDS on benchmarks of deadlocking and non-deadlocking programs. It will further examine the effects and causes of deadlocks within Java programs.

The functionality adopted for detecting and resolving deadlocks is based on the lock graph consisting of vertices and directed edges. The size of the graph is further optimized for efficiency. If the running application does not have any contention for locks and if no thread requests a resource that is not available at that moment, then the lock graph will be empty. For each lock contention, a vertex is added for each the requester and owner threads. With each edge

addition, the graph is checked for the presence of a cycle. When a thread acquires the resource it was waiting for, the corresponding edge from the graph is removed, along with incident vertices.

The technique to resolve a deadlock depends on preempting a lock in the detected deadlock cycle. The thread holding the lock is called the "victim" thread. In this strategy, the lock is returned to the victim thread once another thread has used it. Nevertheless, in order to make this strategy function, it was crucial that one of the threads in a deadlock is suitable to be victimized. In this regard, a preprocessing technique is employed, which is aimed at detecting threads that can be victimized without affecting the program state. In order to achieve this, the location of source code is identified in which a thread acquires a lock. Consider the following scenario: The thread changes the object secured by the lock or a shared object positioned in the path between the two synchronized points, and the thread attempts to acquire a second lock prior to releasing the first lock. In this scenario, a harmful statement is defined as a source code statement that alters a locked object or a shared object before a second lock is requested by a specified thread. Here, the preprocessor stores synchronized points. The preprocessor also locates the statements that request and release the locks. At last, the preprocessor observes and assesses the statements for possibly harmful statements in potential deadlock locations. As a result, a so-called synchronization tree was created, which captures all locking data from the Java application including both reentrant locks and intrinsic (monitor) locks. The tree shows all flows of calls among locking statements in the source code.

In addition, the preprocessor generates a list of harmful statements along with their locations. The preprocessor annotates the locations of the harmful statements, which is included

in the generated list. The annotation process is done using Software Transactional Memories STMs—a nonblocking methodology to ensure that any write operation is made atomically. The annotation process ensures the consistency of the program state in the runtime monitoring when a deadlock occurs and we have no candidate "victim" thread.

Our empirical findings show that DDS has the ability to solve deadlocks in Java programs without considerable runtime overhead. The probability of deadlocks is however lower than the samples of the experiment. The research found an average overhead performance of about 5.93% with respect to regular performance—when running the benchmarks without injecting harmful statements— of the deadlock detector and solver. This shows that STM has no significant effect. In the absence of deadlocks, there will be no effects on the flow of the program. Hence, DDS can effectively function without affecting the semantics of the program under monitoring. It involves a modest runtime overhead. The findings of the research showed that the DDS method is scalable and does not incur a noticeable overhead with the increase in the number of synchronization points and the number of threads. It is also found that the combination of lock preemption and software transactional memory can be effectively used to resolve deadlocks at runtime.

CHAPTER 1

INTRODUCTION

The introduction of this chapter have been previously published in:

- E. Aldakheel, U. Buy and S. Kaur, "DDS: Deadlock Detector and Solver," *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Memphis, TN, 2018, pp. 216-223. doi: 10.1109/ISSREW.2018.00009
- E. Aldakheel, "Deadlock Detector and Solver (DDS)," *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, Gothenburg, 2018, pp. 512-514.

It is observed that multicore hardware development is consistent with the increasing tendency towards the use of concurrent software. Software developers are increasingly found to be making use of language constructs like *Java threads* so as to leverage the abilities of multicore hardware (1).

In cases where data structures are shared by more than one thread, successful avoidance of data races is heavily dependent on synchronization of object. The occurrence of data races takes place if the data structure is concurrently accessed by several threads, with a minimum of one instance of this access changing the shared data structure. However, object locking can lead to deadlocks, which can be very challenging to trace via testing given their tendency of self-manifestation.

1.1 The Deadlock Problem

Deadlock is a highly complicated challenge in that it can cause serious impact on the consistency and reliability of multithreaded programs with multiple asynchronous threads. Deadlocks take place when a number of threads are mutually blocked thereby preventing a resource from being accessed by another thread (2). The current deadlock resolving methods are typically comprised of timeout and rollback mechanisms. To recover from deadlocks, a common practice is to rollback computations, but this is likely to cause considerable delays. If deadlocks are left undetected, they may lead to permanent thread blockage (1).

The following discussion begins with a brief background of the research problem. Accordingly, background information encompasses a preview of multicore hardware as well as its relationship with Java threads. Later, a discussion of this research problem continues with various scenarios of deadlock. The first situation entails the well-known problem of dining philosophers', whereas the second one is a banking transaction that clearly shows the mechanism in which aliasing can result in deadlocks. Here, we also highlight the problem statement. In addition, this chapter presents the discussion of the research problem, with a focus on demonstrating its significance as an important research area. This is expounded in detail in Section 1.1.3 below.

1.1.1 Background Information

The concept of multicore computing platforms has evolved over the past decade, and its use has increased. The aim of the platforms is to exploit the computational power of multicore hardware through the ability to execute multiple threads simultaneously (1). In order to exploit multicore hardware, software developers need to utilize concurrency within their software pro-

grams to maximize the capabilities of hardware (2). The benefit of the resultant concurrency is faster program execution. The inclusion of object synchronization is known to be important for avoiding data races, which refers to a situation when the data structure gets concurrently shared by several threads, wherein one such access makes changes in the shared structure.

The high-level programming language - Java is known as one of the first ones that added threads for supporting concurrency in execution (3). In addition, Java supports thread synchronization with object-locking mechanisms like intrinsic locks, reentrant locks, and semaphores. In Java programming, a thread functions as an autonomous path of execution. In the programming, every thread has the ability to execute a synchronously with respect to other threads. Prior to accessing a shared object, a thread is first required to acquire a mutual exclusion lock on the object. In the case of the lock not being owned by any thread, the thread trying to obtain the lock will succeed in acquiring it. In the opposite case, the thread needs to wait while the lock is not available. However, concurrency brings about certain complex bugs due to its non-deterministic behavior. Software developers experience the deadlock problem as one of the main challenges affecting program reliability. Deadlock occurs when a hold-and-wait cycle takes place concerning locked objects. In addition, it is particularly difficult to detect deadlock using testing because it is inclined to occur non-deterministically.

We found that DDS has the ability to resolve deadlocks by effectively preempting an intrinsic lock or reentrant lock from a thread introduced in a deadlock. Each instance of a Java class has an intrinsic lock associated with it. A thread has the ability to obtain the lock on an object with either a synchronized block or a synchronized method. When the lock on a particular object

is achievable, a synchronized block can be executed. In this scenario, the thread executing the synchronized block obtains the lock and runs the block. Once the thread completes the function of the block, the lock is released. No other threads are allowed to obtain the lock. A synchronized method works by locking the object receiving a method invocation; when the method returns, the lock is released. In either case, a thread is blocked and required to pause if the requested lock is not available (4).

Reentrant locks provide additional flexibility, because they are obtained and released at arbitrary locations within the program code. Hence, the scope of the locking is not limited to a single block similar to intrinsic locks. A thread sends a request and releases a lock by calling the *lock* and *unlock* functions on a lock object. In the event a thread requests a lock which is already under the hold of another thread, the requester's thread will be blocked until the release of the resource.

1.1.2 Illustrative Examples of the Problem

The deadlocks can be best explained and illustrated through the example of the dining philosophers' problem in which five philosophers are dining at a table, as shown in Figure 1. On the table, there are five bowls and five forks, with one bowl and one fork for each philosopher. In order to eat philosophers, require two forks located adjacent to their respective bowls. After finishing eating, they are able to continue to think after placing the fork over the table. In this example, it is reasonable to view philosophers as threads, whereas forks could be regarded to be resources. In the event that all philosophers pick up their right fork first, deadlock takes place because each philosopher cannot acquire the left fork, which another philosopher is holding.

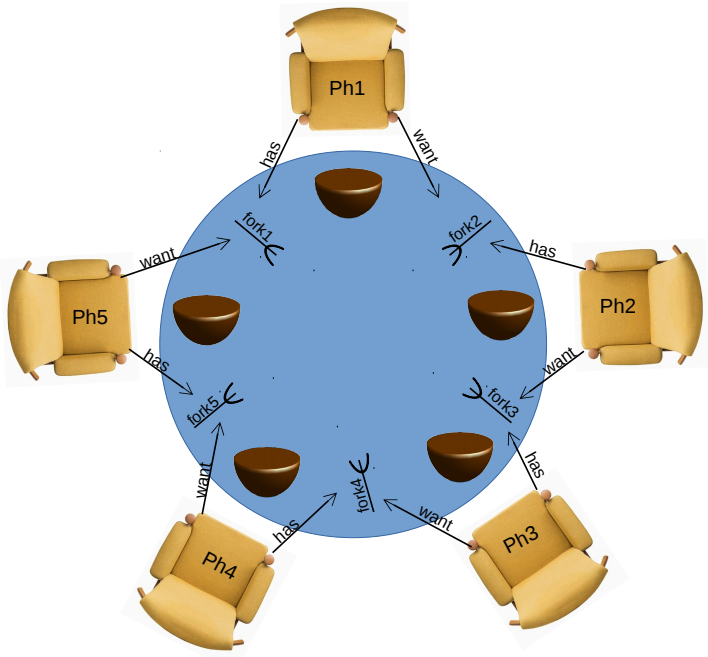

Figure 1: Deadlocking scenario for the dining philosophers' problem with five philosophers—named Ph#—where each philosopher holds the fork to the right—named fork#—and blocked to their left fork—named fork# mod 5.

```
public class Account{
   double balance;
   int id;
   public void withdraw(double amount){
         balance -= amount;
   }
   public void deposit(double amount){
         balance += amount;
   }
   public void transfer(Account from, Account to, double amount){
         synchronized(from){
            from.withdraw(amount);
            synchronized(to){
               to.deposit(amount);
            } //release(to);
         } //release(from);
   }
}
```

Figure 2: Deadlocking scenario in banking transaction when running $\mathtt{transfer}(x, y)$ and $\mathtt{transfer}(y, x)$ simultaneously by two different threads. (Adapted from (5)).

Locking mechanisms tend to interact with other language features such as aliasing. This can be explained in the following way: Consider a simple banking transaction, as shown in Figure 2. Two threads try to execute money transfer from account x to account y and vice versa. Two threads that make attempts to gain access (control) over the resource (account) in the opposite order, thus causing deadlocks.

1.1.3 Significance of the Problem

In Java programming, deadlock is one of the most complex yet critical issues that can considerably affect the reliability of concurrent programs. The occurrence of deadlocks is attributed to the blockage of multiple threads, while simultaneously accessing a shared resource demanded by a different thread. The conventional methods to resolve deadlocks mainly depend on rollback mechanisms and timeout. However, recovery from deadlocks mostly entails the rollback

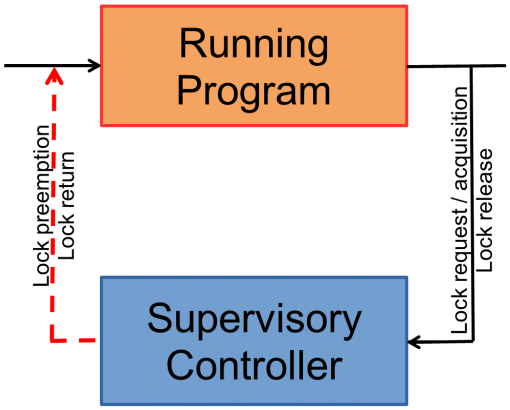

Figure 3: Overview of DDS approach contains monitoring a running program and preemption action when the deadlock occurs represented by dashed arrow.

of computations, causing considerable delays. Therefore, this study is significant because it evaluates DDS and provides an assessment of the efficiency of DDS in resolving deadlocks.

The aim of the thesis is to resolve deadlock with the help of DDS while executing a program without decreasing concurrency and with uncertain time and space overheads. In this regard, the method of DDS mainly comprises three elements: a preprocessor, a detector and a resolver algorithm.

1.2 Overview of the Approach

This research introduces a complete solution based on a supervisory controller called the *deadlock detector and solver (DDS)*. DDS continuously monitors a running program to detect and resolve deadlocks at runtime, as shown in Figure 3.

1.2.1 Our Supervisory Controller

The approach adopted to carry out the research includes three components: runtime deadlock detector, runtime deadlock solver, and preprocessing. At the runtime, the DDS employs two components: the detector algorithm and the solver algorithm. The purpose of this detector and solver is to monitor, detect, and resolve deadlocks at runtime. The *detector algorithm* performs the detection of deadlocks through monitoring the way locks are requested, obtained, and released.

Before we explain the components of our approach, we must define our scope and assumptions. The DDS approach detects and resolves resource deadlock automatically at the runtime. The detection and resolution of resource deadlock involves intrinsic locks and reentrant locks. We performed an interprocedural analysis and an alias analysis offline during the preprocessing phase. The purpose of these analyses was to identify "harmful statements" within the source code, which represents places in the program where preempting a lock could affect the program's consistency. We assume that the monitored application does not contain any semaphores, which could contribute to the deadlock. In addition, we assume that the application does not have I/O statements when STM is used. If an I/O statement is presented in a method that is protected using STM, then STM cannot rollback the effects of those statements.

1.2.1.1 Detector Algorithm

The functionality for detecting and resolving deadlocks is based on the lock order graph, referred to here as the *lock graph*, which consists of vertices and directed edges. Each vertex represents a mutex lock held by a thread, and each edge stands for a request relationship

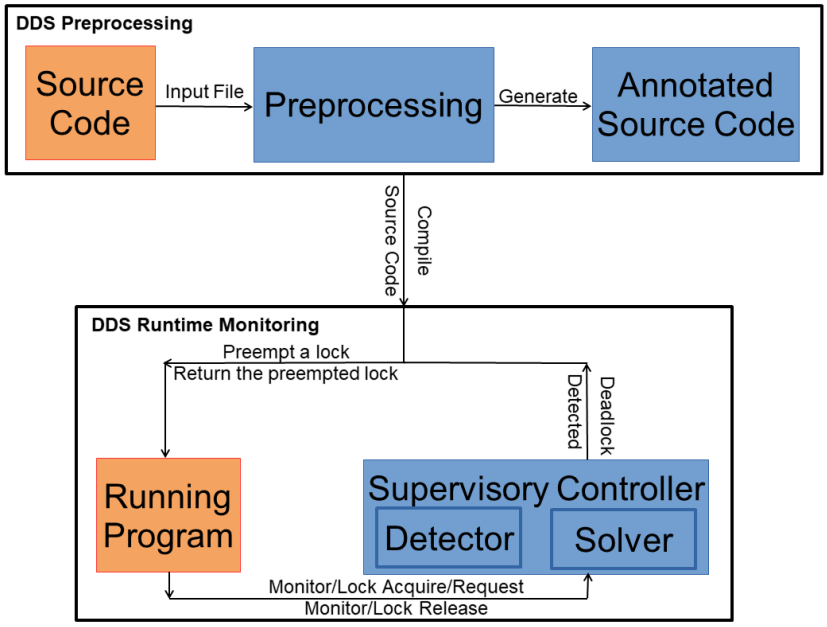

Figure 4: Overview of DDS approach including both runtime monitoring and offline preprocessing.

between two threads for a specific mutex. The presence of a cycle in the lock graph indicates the existence of a deadlock. An edge from v_1 to v_2 means that the thread holding the lock associated with v_1 is trying to acquire the lock associated with v_2, which is concurrently held by another thread.

1.2.1.2 Solver Algorithm

Another component in the runtime monitoring is the *Solver algorithm*. If the lock graph contains a cycle, a deadlock has taken place. In this scenario, the *detector* locates an affected thread and notifies the *solver* of the affected thread. The solver algorithm makes the affected thread release the held lock. Thereafter, another thread that requires the lock can obtain it.

Once the lock is released by the second thread, the solver algorithm returns the lock to the affected thread, which can then continue processing. We denote the thread whose mutex lock is preempted as the *victim* thread.

1.2.1.3 Preprocessing

The third component of the DDS approach is the preprocessing. Preprocessing is an important part of supporting the performance of an application to ensure it remains consistent. Our *solver algorithm* for resolving deadlocks is based on preempting one of the locks involved in the deadlock cycle. The preprocessing phase guarantees that the lock preemption is safe, that is, it does not leave the application in an inconsistent state. We want to ensure that lock preemption is safe because we do not rollback computations to recover from an inconsistent state.

Underlying this method is an inherent risk that a thread $\mathtt{t_l}$ may have modified the object $\mathtt{o}$ whose lock is involved in the deadlock. In this case, $\mathtt{t_l}$ should not be chosen as the *victim* thread because $\mathtt{o}$ could be in an inconsistent state when another thread is given $\mathtt{o's}$ lock. When we find that lock preemption is not safe, we guard shared objects using STM.

If the *solver algorithm* victimizes a lock, and we have modified a shared object protected by that lock, the victimized lock is considered a harmful statement. In other words, we cannot victimize that lock because that will leave the program in an inconsistent state. Thus, harmful statements need to be protected to maintain the application in a consistent state. Therefore, we are using the *STM*—a nonblocking mechanism to ensure that any written update is performed with atomicity as one thread at a time— to protect the harmful statements. In this

preprocessing phase, we identify locations where the *STM* is needed to prevent the preemption of the lock from leaving a shared object in an inconsistent state.

Under this approach, at first, the source code is analyzed to identify the program locations where a request for lock acquisitions is made. See Figure 4. The generated set of program locations is then used to create a tree showing call dependencies among those locations. Here, a tree vertex signifies the location of a program that entails locking, and an edge symbolizes the presence of a control flow path between two locations. For each application, there is a single tree. We specifically consider paths between lock acquisition points. The statements along such paths are examined to determine whether any write operation on a shared object is performed. If such a statement is found, then it is considered a *harmful statement* (i.e., mutex lock). In this case, we cannot preempt the mutex lock that protects that statement. We examine the entire tree for the presence of any *harmful statements*.

The preprocessing algorithm identifies situations in which there may not be any candidate victim threads. In these cases, we apply a method based on a STM to resolve the deadlock by automatically annotating the source code. We annotate the method that contains the harmful statement using the Java annotation *@Atomic*. This annotation results in considering that block as a transaction. A transaction in STM contains a sequence of read and writes on shared variables that are executed atomically. In STM, a read operation accesses the value of a shared variable. A write operation stores a new value to a shared variable. Therefore, using STM guarantees that the access to the shared variables is atomic. The execution time overhead introduced using STM is negligible because we are only using STM methodology with the

methods that contain harmful statements. The other original methods without annotation do not incur a performance penalty when we use STM. In addition, based on our benchmarks, having a harmful statement is rare, and we were required to inject harmful statements in our benchmarks to measure the execution time overhead and memory consumption overhead of using STM.

1.2.2 Existing Approaches

The current methods available for detecting deadlocks are often impractical for detecting them in real-world situations (6; 7; 8; 9; 10) because their algorithms are based on finding a cycle in the lock graph, which has never been optimized. This is far from optimal, particularly for real-world application where we could have a large number of threads. This results in a noticeable slowdown when detecting or avoiding deadlocks. In some cases, they experience significant performance degradation (11; 12; 13; 14). Therefore, it is important to have a runtime monitoring methodology to detect and resolve deadlocks efficiently without manual code annotations as well as with reasonable performance overhead. In this way, it is essential to assess DDS, which is a runtime approach to detect and resolve deadlocks. Unlike Grace (14), Cilk (15), Cilk-5 (16), and UnDead (17), DDS does not force a deterministic code execution. Many other methods need to have manual code annotations (13; 18; 19; 20). When there are no deadlocks, the typical runtime overhead of running the DDS is normally lower than 5% of the original program's runtime. To detect deadlocks, other researchers, (3; 21; 13; 22; 23; 24; 25), have also utilized lock graphs. The approach presented in the current study differs from the

existing methods because our lock graph has been used not only to detect deadlocks but also to resolve them by choosing a "victim" from the threads involved in the deadlock.

1.3 Challenges

A number of challenges are related to the inclusion of reentrant locks in the preprocessing phase. One of the challenges is related to the scope of locking. The scope of intrinsic locks in synchronized blocks and synchronized methods is clear. The scope of a block is the block of code between the curly braces. The scope of the synchronized methods is the body of the method. Consequently, the intrinsic locks provide clear guidelines on where to start and where to stop the analysis. Reentrant locks do not show the scope of the lock based on the source code. Finding the scope requires an extensive search for all paths between "lock" and "unlock" statements.

Another challenge is concerned with the ability of reentrant locks to be performed on a lock object instance that can be locked and unlocked a number of times. For the analysis, the pairs must be corresponding exactly, which means that extraordinary care is required to determine the correct pairings. One lock can have multiple unlocks based on the control flow graph. This challenge becomes more complex in the scenario where there is no unlocking statement because a program with locking could be run even if there is no unlocking.

Object aliasing also creates elusive issues related to determining the harmful statements. Hence, it is not easy to identify which object is synchronized in the presence of the alias of the synchronized object. If one object is synchronized, whereas another object is changed in the path of the synchronized point in the tree, it cannot be declared that one object is different

from the other object since both objects can be the same because of aliasing. However, there is no feasibility of totally resolving the aliasing issue. Thus, the third significant challenge in the study is related to aliasing for DDS preprocessing.

1.4 Contributions

This study makes significant contributions, to the state of art in resolving deadlocks, by discussing and analyzing DDS, a supervisory control methodology for detecting and resolving deadlocks in Java programs. It also contributes by assessing the effectiveness of the DDS in detecting and resolving deadlocks at runtime. The empirical assessment contributes by providing an evaluation of the efficiency of the preprocessing component of the DDS employed to identify situations where no lock can be safely preempted.

This study makes the following specific contributions:

1. It discusses the DDS, a supervisory control methodology for detecting and resolving deadlocks in Java programs.
2. It empirically evaluates the effectiveness of the DDS in detecting and resolving deadlocks at runtime.
3. It empirically assesses the efficiency of the preprocessing component of the DDS, which is used to identify situations where no lock can be safely preempted.
4. It empirically evaluates the efficiency of STM in DDS by measuring the overhead of using a STM methodology.

The study also reports empirical results obtained from the application of DDS to 11 open-source deadlock benchmarks. These benchmarks had a changing number of input sizes. To achieve the findings, each benchmark was run 60 times to increase the coverage of deadlock detection and its resolution. The average CPU time overhead of supervision never exceeded 7.1% of the original program's runtime. The DDS showed a linear runtime increase with the increase in the input size and thread number. It resolved deadlocks effectively and efficiently. Thus, the adopted approach is found to be significantly scalable.

This dissertation is organized into eight chapters including the introduction chapter. The remainder of this dissertation has been organized in the following manner:

Chapter 2 provides background information on mutual exclusion of Java and the concept of software transactional memory.

Chapter 3 highlights the related work on deadlock detection, avoidance, and prevention. We explain how they are different from our proposed work.

Chapter 4 presents the preprocessing algorithm of DDS. The preprocessing phase identifies and guards the harmful statements in the program under control.

Chapter 5 illustrates runtime monitoring, which includes two major components: the detector and solver algorithms.

Chapter 6 explains the implementation details of DDS to apply it to Java language. It also displays selected benchmarks to evaluate the DDS effectiveness with some experiment results.

Chapter 7 illustrates how the DDS methodology can be applied to any programming language other than Java. We focus on DDS application in C++ language. In addition, we

present how the deadlocks are handled in distributed systems and whether we can apply our methodology in such systems.

Finally, **Chapter 8** provides conclusions on our supervisory controller method DDS and some of the future research work.

CHAPTER 2

BACKGROUND

Parts of this chapter have been previously published in:

- E. Aldakheel, U. Buy and S. Kaur, "DDS: Deadlock Detector and Solver," *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Memphis, TN, 2018, pp. 216-223. doi: 10.1109/ISSREW.2018.00009

In this chapter we provide some necessary background information on Java mutual exclusion for intrinsic locks and reentrant locks, and software transactional memory. This chapter will help the reader understand some of the concepts that will be presented in the following chapters of this dissertation.

2.1 Java Mutual Exclusion

Java has multiple constructs for inter-thre ad synchronization and communication. In this thesis, we will focus on intrinsic locks and reentrant locks. Java programming adequately manages and supports multithreading. In order to perform multithreading or concurrent programming, it is important to have effective synchronization and concurrency mechanisms for threads.

2.1.1 Intrinsic Locks

Every class instance in Java has an intrinsic lock associated with it. A thread can acquire a lock on an object using either a *synchronized block* or a *synchronized method* as shown

in Figure 5. A *synchronized block* can be executed only if the lock on the specified object is available. In this case, the thread executing the synchronized block statement acquires the lock and executes the block. The lock is relinquished when the thread completes the execution of the block. A *synchronized method* works similarly, except that it locks the object receiving a method invocation. The lock is relinquished when the synchronized method returns. In either case, if the requested lock is not available, the thread is blocked and forced to wait. In Figure 5, we protect incrementing and decrementing the integer `i` using intrinsic locks. The use of intrinsic locks in the presented example avoids data races, ensuring that only one thread at a time can modify the value of `i`. All synchronized statements in Figure 5 lock the same object (this).

2.1.2 Reentrant Locks

Reentrant locks in Java provide synchronization and increased flexibility compared to intrinsic locks. The reentrant locks apply the interface of class "Lock" to synchronize access to shared resources. In this way, the code that is involved in the operation of the shared resource has a request to acquire and relinquish a lock.

Reentrant locks supply additional flexibility with respect to intrinsic locks because they are acquired and relinquished at arbitrary locations in the program code. Thus, the scope of the locking is not confined to a single block or method as it is with intrinsic locks. A thread requests and relinquishes a lock by calling the *lock* and *unlock* functions on a lock object. See Figure 6. As with intrinsic locks, attempting to acquire a reentrant lock will block the requesting thread if the lock is held by another thread. Reentrant locks are effective locking mechanisms that do

```
public class Counter_Example{
   // shared among multiple threads
   private int i;
   private int readCount;
   ...
   // synchronized method
   public synchronized void increment()
   {
      i++;   // critical section
   }
   public void decrement()
   {
      // synchronized block
      synchronized(this)
      {
          i--;   // critical section
       }
   }
   public int value()
   {
      // synchronized block
      synchronized(this)
      {
          readCount++;  // critical section
         return i;
       }
   }
   ...
}
```

Figure 5: An example showing intrinsic locks using “synchronized” keyword.

```
public class DiningPhilosophers{
   static Lock[] chopsticks; // shared among multiple threads
   ...
   public void pickupChopsticks()
   {
       // starting point of the critical section A
        chopsticks[left].lock(); //pickup left fork
       // starting point of the critical section B
        chopsticks[right].lock(); //pickup right fork
   }
   public void putdownChopsticks()
   {
        chopsticks[left].unlock(); //put down left fork
        // ending point of the critical section A
        chopsticks[right].unlock(); //put down right fork
        // ending point of the critical section B
   }
   public void round()
   {
        pickupChopsticks(); //locking the forks
        eat();
        putdownChopsticks(); //releasing the forks
        think();
   }
}
```

Figure 6: An example showing reentrant locks using "lock" and "unlock" keywords.

not cause problems related to lock reentry. Reentry happens when a thread tries to acquire a lock that the thread already holds (26).

2.1.3 Difference between Intrinsic and Reentrant Locks

Reentrant locks were introduced in Java 1.5. Before reentrant locks existed, concurrency was achieved with the help of *synchronized* methods and blocks. Reentrant locks are aimed at improving the functionality of intrinsic locks. Intrinsic locks are effective in most situations; however, they have certain functional limitations (27). Whereas both intrinsic and reentrant locks can be used to achieve concurrency (28), reentrant locks have the following features:

1. Arbitrary scope as shown in our `DiningPhilosophers` class presented in Figure 6.

```
public class trylock_method{
   static Lock l; // shared among multiple threads
   ...
   public boolean done()
   {
       boolean flag = false;
       if (l.tryLock())
       {
            try
            {
               flag = true;
               // critical section
            } finally {
               l.unlock();
            }
        } else {
            // if the lock is not available perform alternative actions
            System.out.println("The requested lock is not available!");
        }
        return flag;
   }
   ...
}
```

Figure 7: An example showing reentrant locks using "trylock" and "unlock" keywords.

2. Only work on lock objects (i.e., chopsticks in Figure 6).

2.1.3.1 Test for Availability

Intrinsic locks play an important role in various aspects of synchronization. For example, they allow limited access to objects and create important happens-before relationships for visibility (29). One limitation associated with intrinsic locks is that there is no way to perform a test 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.

Reentrant locks provide a convenient way to check the lock availability before acquiring the lock using the statement "trylock". Using *trylock* on a lock inquires weather the lock is

available at that particular time—time of lock request. Statement *trylock* returns `true` if the lock is available and `false` if the lock was not available at the time when the method was invoked. See Figure 7. Thus, if *trylock* returns `true`, we must *unlock* the locked object after completing the critical section, which is a segment of code that accesses shared resources and must be executed atomically. Essentially, *trylock* is a nonblocking *lock* that locks the lock if it is available; otherwise, the thread will continue executing the next statements. This flexibility to try for lock, without causing the thread to block if the lock is not available, results in better performance because threads are not blocked while checking for the lock availability.

2.1.3.2 Non-Block Structured Locking

It is necessary for intrinsic locks to get released within the same block of code where their original acquisition took place. This, in turn, leads to simple coding that is able to interact easily with exception handling. However, the block-structured locking lacks flexibility because its scope is limited—it is defined by method or block scope only. Nonetheless, this is not the only reason for not using intrinsic locks. Some cases require a locking mechanism that is more flexible for better performance (30; 31).

Reentrant locks have an arbitrary scope that is defined dynamically. A thread uses *lock* to get hold of the lock, or it waits until the lock becomes available. The thread that holds the lock must issue *unlock* to release the lock and become available for other waiting threads.

2.1.3.3 Fairness

Compared to intrinsic locks, reentrant locks are considerably more flexible and offer a choice of two fairness policies—an unfair lock or a fair lock. With regard to acquiring access to the

shared resources, unfair locks do not guarantee any particular order of threads to get hold of the lock. Fair locks allow acquiring locks in the order in which they were requested. In other words, if one thread is in the waiting queue for a considerably longer period of time than another thread, it is certain that once the present thread is complete, the thread that has been waiting the longest will be able to access the shared resource. Thus, the advantage of the unfair lock is that it helps reduce wait time and increases an application's overall throughput because it allows the thread to acquire the lock when it becomes available regardless of its order in the waiting queue (32).

2.1.3.4 Reentrancy

Another difference between intrinsic and reentrant locks is related to the ability of reentrant locks to simplify the development of object-oriented concurrent code. On the one hand, during the absence of reentrant locks, when it is observed that a subclass ends up overriding a synchronized method before calling the superclass method—which gets synchronized on the same object —lead to a deadlock (30). Intrinsic locks are not capable of reentering a lock—they will attempt to acquire the lock they already hold.

On the other hand, a hold count is assigned to every reentrant lock. Once a reentrant lock is acquired, the hold count for that lock is increased by one. Once a thread releases a reentrant lock, the hold count for that lock is decreased by one. The reentrant lock is free once the hold count reaches zero. The maximum number of recursive reentrant locks a thread can reach is 2,147,483,647. If a thread exceeds this limit, the locking method shows an error. A thread

```
public class Widget
{
    public synchronized void doSomething()
    {
      ...
    }
}
public class LoggingWidget extends Widget
{
   public synchronized void doSomething()
   {
      System.out.println(toString() + ": calling doSomething");
      super.doSomething();
   }
}
```

Figure 8: Reentrancy issue using intrinsic causing a deadlock situation (27).

cannot acquire a lock held by another thread by increasing the count because the count is associated with the thread for reentrancy purpose.

We use Figure 8 to explain the reentrancy issue associated with intrinsic locks. The methods of doSomething in Widget and LoggingWidget are synchronized; therefore, both methods attempt to acquire the intrinsic lock prior to continuing. Both synchronized statements in Figure 8 lock the same object, this. The call to super.doSomething cannot, in this instance, succeed in acquiring the lock on this object because the current thread is already holding the intrinsic lock on this object. As a result, the thread would always be waiting for a lock that it would not be able to acquire, leading the thread to end up in a deadlock with itself. Reentrant locks have the ability to avoid this kinds of deadlock situations (27).

2.2 Software Transactional Memory

Software Transactional Memories (STMs) (33) apply the transaction concept to mainstream programming, largely as a means of simplifying concurrency programming. The central concept behind STMs is that programmers specify the operations that should be implemented atomically, tasking an underlying transactional framework with the implementation of the desired atomicity while retaining significant parallelism. From the viewpoint of STMs, operations implemented during a transaction have no special meaning associated with them. They merely constitute sequences of writes and reads to shared locations of memory. STMs intercept accesses to shared locations to determine when two simultaneous transactions interfere with one another, causing the STM to restart, abort, or stall at least one such transaction. Several STM methodologies exist, and they differ significantly in the manners in which they ensure operation atomicity. For our work, we chose an existing STM framework named Deuce (34).

Deuce is an open-source transactional memory framework specifically aimed at Java multithreading. An advantage of Deuce is that it supports a wide set of features without altering existing Java libraries or the Java compiler.

Deuce is considered nonintrusive because language extensions or modifications to the Java virtual machine (JVM) are not necessary. It utilizes, by default, an initial locking design that identifies individual field-level conflicts without significantly increasing memory footprints (no additional data are included in the classes). As a result, the garbage collection overhead is minimal.

Deuce has undergone extensive optimization for efficiency, and although there is room for improvement, performance assessments on numerous high-end machines show that it has excellent scalability. Existing benchmarks indicate that its performance surpasses that of alternative JVM-independent Java STMs, such as the DSTM2 framework (35), typically by two orders of magnitude. Deuce generally scales appropriately on multiple workloads and shows that a high-performance Java STM can be implemented even in the absence of ad-hoc compilation. In summary, we chose Deuce because of its high level of performance and usability with respect to competing STMs, including DSTM2 (35), JVSTM (36), and AtomJava (37).

CHAPTER 3

RELATED WORK

Parts of this chapter have been previously published in:

- E. Aldakheel, U. Buy, "Efficient Run-time Method for Detecting and Resolving Deadlocks in Java Programs," *In 33rd European Conference on Object-Oriented Programming Workshops (ECOOP)*, London, UK, 2019.
- E. Aldakheel, U. Buy and S. Kaur, "DDS: Deadlock Detector and Solver," *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Memphis, TN, 2018, pp. 216-223. doi: 10.1109/ISSREW.2018.00009

Because of its nondeterministic behavior, concurrent software is significantly vulnerable to defects (i.e., deadlocks). Therefore, overall, deadlock is the most complex issue that is characterized by the inability to be diagnosed and debugged (22). For a deadlock to take place, four necessary conditions need to be illustrated.

1. **Mutual Exclusion**: No more than one thread can simultaneously access the resource. Put simply, it is not possible to share resources.
2. **Hold and Wait**: A thread has the ability to hold a resource (i.e., lock) and simultaneously wait for another resource possibly held by another thread to be available.
3. **No Preemption**: Once a thread holds a resource, the resource cannot be taken away from that thread until the thread voluntarily releases it.

4. **Circular Wait**: A set of threads each one of them is holding a resource and waiting for another thread in the set to release a resource. Thus, we must have a set of threads $\{t_1, t_2, ..., t_n\}$ where every t_i is waiting for $t_{(i+1)\%n}$.

In this chapter, some of the previously presented techniques and tools aimed at detecting and handling deadlocks in multicore systems are highlighted. Deadlocks can be handled using three different strategies: detection and resolution, prevention, and avoidance. In previous work, both static and dynamic approaches were employed to detect and handle deadlocks, including runtime monitoring and model checking. To detect deadlocks, the majority of these techniques rely on finding a cycle in the lock order graphs or cyclic lock dependencies.

3.1 Deadlock Prediction and Detection

The ultimate goal of the work presented in this section is to predict and detect deadlock occurrences. The discussed methodologies do not have resolution strategies, unlike our method DDS, which resolves the detected deadlock.

3.1.1 Static Approaches

Static approaches seek to detect all deadlocks that may be present in a program under analysis. Williams et al. (38) presented a static analysis tool that used a flow-sensitive and context-sensitive analysis to detect deadlocks in Java libraries. Unlike DDS, this approach does not address reentrant locks. Similarly, RacerX (39), which is a tool for C based systems, uses a flow-sensitive analysis and context-sensitive analysis to detect deadlocks and data races, and requires that source code annotations be done manually. A static approach to detect a deadlock for data-centric synchronization programs was presented in (20). However, in this approach,

code is manually annotated to determine the ordering between various instances of atomic sets; each atomic set is a group of shared memory locations. Unfortunately, such static approaches report false positives. By contrast, DDS is completely automatic, does not require manual annotation, and focuses exclusively on inferring the runtime information without creating any false positives.

Although model checkers are not scalable because of the state space explosion problem, they do demonstrate high coverage of deadlocks. Bogor (40) is an automatic deadlock detector that transforms shared resources and global variables implemented in the Java program into an input for its model checkers. Afterward, Bogor creates an automaton that represents all the possible states of the Java program, enabling it to assess the occurrence of a deadlock. Gadara (13) is another tool that detects deadlocks using model checking. It detects all potential deadlock candidates, including false positives, unlike the proposed approach, which does not introduce false positives; the absence of false positives can be attributed to the use of runtime monitoring, which only records actual deadlocks.

The use of reentrant locks in the deadlock analysis of multithreaded programs requires a more sophisticated detection approach. The infinite states of these multithreaded programs require a detection approach that promises visible access to objects. To address such complexity, Laneve defined a simple calculus featuring a recursion, threads, and synchronizations approach (42). This approach has efficiently detected deadlocks by creating a link between an abstract model and the program. The static semantics of this object model can help in detecting deadlocks in programming languages such as Java. The approach is based on verifying that the

```
public void m1()
{
    lock.lock();
    a = 1;
    aIsOne.signalAll();
    // missing ‘‘unlock’’
}

public void m2()
{
    lock.lock();
    while(a != 1)
        aIsOne.await();
    a = 0;
    lock.unlock();
}
```

Figure 9: An example showing faulty design of reentrant locks by missing the “unlock” in m1 function (41).

created model—based on its extraction process for dependencies between threads and locks by means of acquire (own) and request (want)—does not contain any cycle. It is possible to extend this approach to core calculus that features access to shared objects, creation and execution of threads, as well as Java-like synchronization primitives (42). Nevertheless, DDS is more precise in detecting deadlock and does not report any false positives in contrast to Laneve (42).

Kamburjan (41) proposes a deadlock detection system based on conditional synchronization. The approach integrates a heavyweight deductive-verification tool with lightweight static analysis, and utilizes a theorem to analyze side effects. Yet the incorrect application of the synchronization primitives can result in an erroneous design. Additionally, the application of this detection system is limited to deadlocks caused by faulty system design, as shown in Figure 9, and excludes those caused by implementation bugs, as shown in Figure 10. The two examples contain condition variable (i.e., aIsOne). Condition variables are conditions that are

```
public void m1()
{
    lock.lock();
    a = 2; // bug
    aIsOne.signalAll();
    lock.unlock();
}

public void m2()
{
    lock.lock();
    while(a != 1)
        aIsOne.await();
    a = 0;
    lock.unlock();
}
```

Figure 10: An example showing an implementation bug in the use of reentrant locks. m2 waits for m1 to proceed by changing "a" state (41).

defined relative to lock objects. They provide inter-thread communication similar to the ones provided using wait, notify, and notifyAll. Calling await on a condition causes the thread to wait (block) until anther thread calls signal or signalAll, which causes the blocked thread on await to activate and proceed with its execution. Condition variables can cause a communication deadlock, which is outside the scope of this work. DDS detects deadlocks regardless of the cause.

Metcalf and Yavuz (43) present regression analysis as another static analysis-based detection approach to detect deadlocks in multithreaded Java applications. The approach is based on filtering code changes that employ locks in an inappropriate order. This detection approach maintains a watch list of lock-type pairs, which can later be used by software developers in detecting deadlocks. However, this approach introduces false positives, unlike DDS.

3.1.2 Dynamic Approaches

Dynamic analysis of a model helps us understand the workflow and behavior of the program in various situations. It can assist in identifying problematic areas and eliminating loopholes in the system. MagicFuzzer (21) has presented a deadlock detection method for large-scale C and C++ applications. Furthermore, as an improvement over MagicFuzzer, MagicLock (25) is presented as being more efficient and scalable at detecting deadlocks in large scale programs. The MagicLock scalability comes from the lock reduction process. The lock reduction process is intended to reduce the size of the lock graph, which contains vertices and edges, by removing those vertices and edges that will not participate in a deadlock (e.g., vertices with no incoming edges or outgoing edges). Still, MagicLock reports false positives, whereas DDS does not report any false positives and creates an optimized lock graph, eliminating the need to preprocess a graph to make it scalable before proceeding to find a cycle.

Another tool, called Sherlock (7), detects deadlocks based on concolic execution (44), which uses both concrete and symbolic values as inputs to execute a program both normally and symbolically. Concolic execution collects symbolic constraints with every branch point in the normal (concrete) execution path and schedules various permutations to execute threads, thereby navigating the execution toward a deadlock scenario. Dirk (45) is a deadlock prediction tool that can map a single run of a program to predict bugs in an exponential number of runs. Dirk stands as the first sound deadlock-prediction technique for Java programs. Here, "sound" means that all the predicted bugs can manifest. Unlike DDS, however, MagicFuzzer (21), Sherlock (7) and Dirk (45) do not resolve deadlocks, report on false positives, and introduce noticeable

overhead. Separately, in order to reduce false positives, ASN (46) proposed to confirm—if the detected deadlock is a real deadlock and it is not a false positive—a detected deadlock for large scale multithreaded programs.

ConLock (47) is another dynamic analyzer, whose main purpose is to confirm that a deadlock has occurred. ConLock$^+$ (48), which has a higher confirmation rate, was subsequently designed as an extension. ConLock (47) and ConLock$^+$ (48) can also be used in combination with other tools that report false positive deadlock cycles.

Yet another runtime monitoring algorithm is the Quality Virtual Machine (QVM) (49), which continuously monitors an execution of a deployed Java program and potentially detects existing defects. QVM can detect a deadlock when enabling heap probes, a dynamic checking method for various heap operations that allows the user to specify the overhead budget. QVM collects as much information as the budget allows during the program runtime. In contrast to DDS, QVM simply reports on deadlocks and does not resolve them.

CLAP is a runtime-based tool for C/C++, created by Huang et al. (50), that can reproduce a concurrency bug by logging a thread's local execution paths. In a second step, CLAP explores memory dependencies that can reproduce logged bugs such as deadlocks. The GoodLock algorithm (51) uses runtime analysis to guide a model checker that can detect potential Java program deadlocks. This algorithm cannot detect deadlocks involving more than two threads, however. Thus, it was later generalized to detect deadlocks involving any number of threads using static analysis (52).

Deadlocks usually appear in Java language wherever multiple threads are interacting. Pulse has been identified as a novel operating system mechanism that executes as a system daemon and can help in the dynamic detection of various types of deadlocks (6), including in I/O events. It discovers the dependencies and constructs a general resource graph. The effectiveness of Pulse can be attributed to its ability to peek into the future. Even though Pulse detects multiple types of deadlocks in applications, it still introduces noticeable runtime overhead, false negatives (miss to report a real deadlock), and false positives, unlike DDS. Pulse takes three seconds on average for each deadlock detection, whereas DDS detects a deadlock in one-thousand of a second. Thus, DDS outperforms Pulse in terms of introduced runtime overhead (efficiency).

Avoidance is yet another method used to manage deadlocks (53). Armus (53) is a verification tool that dynamically detects and avoids barrier deadlocks. A barrier is a point in the execution path of a multithreaded program at which each thread is blocked until all the threads have reached the barrier.

The methods of addressing deadlocks that allow a user to work without modifying the source code of the program significantly help the user to determine the location of a deadlock. Li et al. (54) presented an intelligent deadlock locating scheme for multithreaded programs written in C/C++, Qt, and Java that utilizes a deadlock location scheme for these languages, which works through the modification of three resource functions: mutex, lock, and semaphore. DDS does not modify any kernel or operating system implementation, unlike the Li et al. (54) approach, which is based on modifying the POSIX Thread (pthread) kernel.

For further information in distributed systems' deadlock detection, see Section 7.2.1.

3.2 Deadlock Recovery, Prevention, and Avoidance

3.2.1 Deadlock Recovery

Dfixer (11) fixes deadlock by lock pre-acquisitions, an approach that entails disrupting the hold-and-wait condition, whereas our tool works on the preemption of one of the threads involved in a deadlock cycle. Dfixer selects one of the threads that potentially may be introduced in a deadlock and allows the selected thread to hold the previously held lock and pre-acquire the other needed locks. DDS detects deadlocks at runtime and then eliminates them. In contrast, Dfixer (11) is merely a fixing tool for C++ programs, which reduces concurrency by holding too many locks in advance.

Another tool, Sammati (55) is a runtime tool for POSIX multithreaded program that detects and eliminates deadlocks by rolling back to the location of the lock acquisition that caused the deadlock. Subsequently, Sammati performs the required memory updates. Likewise, Rx [14] detects and resolves deadlocks by rolling back computations. However, the victim thread is not arbitrarily chosen, as it is in the case of Sammati (55)—instead, the chosen thread requires light recovery. Furthermore, ConAir (56) can detect a deadlock by turning locks into locks with timeouts. When a lock reaches timeout, ConAir (56) recovers the issue by preempting and re-executing the thread that asked for the timed-out lock. Similarly, Rx (12) recovers from the deadlock by re-executing the last part of code based on the last recorded checkpoint within a new environment. Rx, however, suffers from false positives. DDS does not roll back computations unless a harmful statement is presented in the path between two synchronization points. DDS has the advantage of minimizing memory and processing power consumption. In

addition, ConAir, Sammati, and Rx are based on thread preemption, unlike DDS, which is based on resource preemption.

3.2.2 Deadlock Prevention

Deadlocks can be prevented by ensuring that at least one of the necessary conditions listed in the introduction of Chapter 3 does not hold. The Synchronized-By-Default (SBD) concurrency model is STM-based approach that prevents deadlocks (57) by breaking the first of the four required conditions for a deadlock to occur. Breaking the mutual exclusion condition results in making the nonshareable resource accessible simultaneously by any number of threads. The model considers that all program code must be run in atomic sections unless the programmer identifies nonatomic segments of the code. SBD carries the STM concept by dividing the program into atomic sections. The rule of SBD is to detect when a nonshareable resource has been modified simultaneously by more than one thread. When the SBD detects the modification, it will commit only one of the access and re-execute the other aborted threads again.

Another runtime tool for C/C++, called Grace (14), eliminates multithreading errors, including deadlock. It essentially makes the program execution deterministic through its behavior aligned with sequential single-threaded programs, by converting every thread spawn to a function invocation and all of the presented locks to a no-ops. Similarly, more tools, like Cilk (15) and Cilk-5 (16) are based on the idea of deterministic execution. Cilk expresses the idea of speculative parallelism and is very simple to use. The user has to add the three keywords to the C-based source code. The first key word in *Cilk* is incorporated into the function header. Then, the programmer introduces *spawn* in the place where he or she wants it to be run in

parallel with the parent thread. The last key word to be used is *sync*, which identifies the join points for the thread to be combined again. Cilk and Cilk-5 operate at C language, which is a procedural language, whereas UnDead (17) works on C++, which is an object-oriented language. DDS does not force deterministic execution. DDS uses the supervisory approach, which performs the task of coordination without adversely affecting the performance of the program because it runs in parallel with the Java program.

3.2.3 Deadlock Avoidance

Deadlock avoidance is more flexible than deadlock prevention because avoidance does not require one of the four conditions to be completely broken. Instead, the programmer must guarantee that the system is always in a safe state, which means it is not in a deadlock state and can allocate resources up to the maximum number of resources available in the system. To check the system's state before each allocation of a resource to a thread, the programmer may need to know the following in advance:

- The maximum number of resources each thread needs.
- The current allocated resources in each thread in the system (owned resources).
- The maximum number of available resources in the system.

The request will be granted if the system's state remains safe after the thread acquires the needed resource. In other words, the request is approved only if the system's resulting state is a safe state.

While DDS resolves deadlocks, Gadara (13) avoids the existing deadlocks in programs by avoiding potentially unsafe lock acquisitions based on the static and dynamic analyses conducted in the offline stage, not during the runtime.

Communix (58), Dimmunix (58), and the optimized Dimmunix (59) are runtime tools that build an immune system against deadlocks by preventing the reoccurrence of a previous deadlock pattern. When a deadlock occurs during runtime, Dimmunix stores the deadlock signatures to avoid that flow of execution in the future. All signatures of deadlock flow that have been captured are stored in a database so they may be avoided in the subsequent runs. Communix is responsible for the distribution of the signatures in the online application. Thus, any user who uses the same application over the Internet will receive the same immunity for every single application under the Communix system. Communix is scalable and effective in protecting Java applications against any problematic signatures.

Voss, Cogumbreiro, and Sarker (60) introduced Transitive Joins (TJ) as a policy to address programs with dynamic task parallelism and arbitrary join operations. They proposed an online deadlock avoidance policy that handles illegal joins—which can cause a deadlock—as an exception. The TJ policy guarantees that any join operation will not cause a deadlock if it was permitted under TJ rules.

In this way, DDS with STM-support—our method for the automatic detection and resolution of deadlocks at runtime—is effective in detecting the deadlock by using a supervisory controller that monitors program execution and has the ability to automatically detect the deadlocks due to the hold-and-wait cycles in Java locks. Hence, after detecting a deadlock, a preemptive

strategy is used by the DDS to break it. We have designed a preprocessor that can identify a mutex lock that can be safely removed whenever a deadlock occurs. If such a case does not exist—which is very rare based on our experiment results—we can use STM methodology to guard any harmful statement that contains locks that cannot be safely preempted because they will leave the program in an inconsistent state. DDS combines a lock preemption strategy and STM to resolve deadlocks at runtime. DDS imposes modest computational overhead at runtime, even when STM is used, relative to the runtime of the original program. To the best of our knowledge, we are the first approach to use the lock preemption strategy with STM support to resolve the detected deadlocks. In addition, we did not force any deterministic execution as in Grace (14), Cilk (15), Cilk-5 (16), and UnDead (17). DDS does not report any false positives as in (20; 42; 43), RacerX (39), Bogor (40), Gadara (13), MagicFuzzer (21), MagicLock (25), Sherlock (7), Dirk (45), and Pulse (6).

For further discussion about distributed systems' deadlock handling, see Section 7.2.1.

CHAPTER 4

DDS PREPROCESSING

Parts of this chapter have been previously published in:

- E. Aldakheel, U. Buy, "Efficient Run-time Method for Detecting and Resolving Deadlocks in Java Programs," *In 33rd European Conference on Object-Oriented Programming Workshops (ECOOP)*, London, UK, 2019.
- E. Aldakheel, U. Buy and S. Kaur, "DDS: Deadlock Detector and Solver," *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Memphis, TN, 2018, pp. 216-223. doi: 10.1109/ISSREW.2018.00009

Our approach for resolving a deadlock is based on preempting a resource (i.e., an intrinsic lock or a reentrant lock) in a detected deadlock cycle. The resource is held by a "victim" thread. When another thread is finished using the resource, it is returned to the victimized thread. In the preprocessing, we inspect the source code for the threads that cannot be victimized. Our approach applies STM methodology when all threads involved in a deadlock cannot be victimized because they may leave the program in an inconsistent state. In this chapter, we first discuss the preprocessing architecture components and how they interact. Then, we explain how to identify a harmful statement using the proposed architecture in which we apply interprocedural flow analysis and alias analysis. Finally, we explain how we guard the harmful statements using STM.

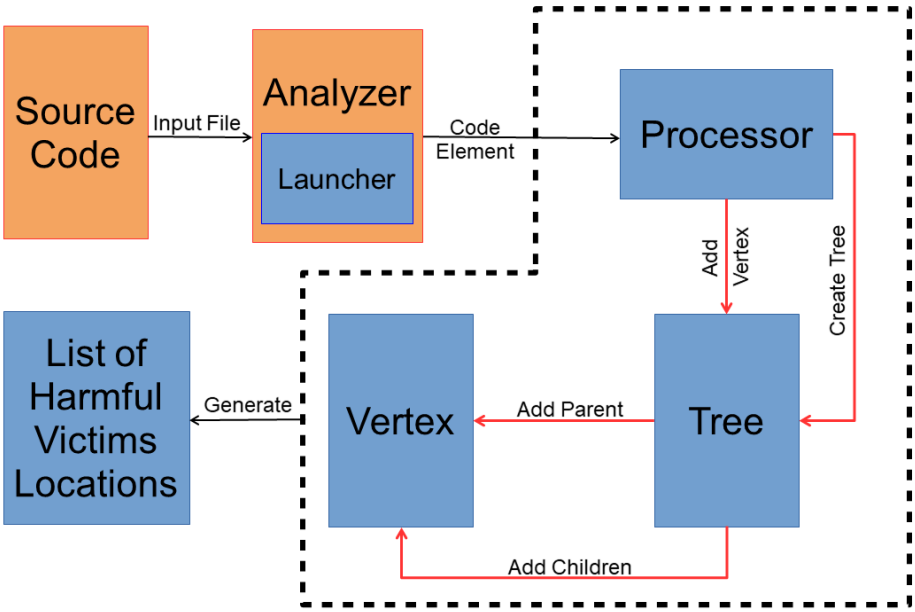

Figure 11: DDS preprocessing architecture.

4.1 Architecture

The preprocessing aspect of our methodology is essential to maintain the application consistency. We first analyze the source code to identify program locations where lock acquisitions are requested. From the resulting set of program locations, we create a tree showing the dependencies among those locations. A tree's vertex represents a program location involving locking, and an edge represents the existence of a path—meaning the child vertex is reachable through its parent—between two locations. A single tree exists for each application under consideration.

4.1.1 Architecture Components

Figure 11 shows different preprocessing components and how they interact. The compiler at the end of the syntax analysis phase creates the Abstract Syntax Tree (AST). This results in a tree representation of the source code's abstract syntactic structure. AST includes vertices that represent code elements, such as statements or loops, and edges that represent containment relationships. Initially, the DDS *launcher* passes a "code element" extracted from the source code based on AST to the DDS *processor*. This code element represents statements and expressions within the statements. Afterward, the DDS *processor* identifies synchronized points from the code elements (e.g., synchronized blocks and synchronized methods). In addition, the DDS *processor* determines the invocations that call the synchronized points. The DDS *processor* creates and maintains the preprocessing *tree* using AST and interprocedural flow analysis to determine all reachable statements to the tree's vertices. Moreover, the DDS *processor* examines statements that could produce harmful effects when their protective monitors are forced to release in a deadlock condition. Potentially *harmful statements* are classified as statements that write or modify either objects or variables that the monitors would otherwise protect. The DDS *processor* uses alias analysis to identify may-alias relationships, meaning these memory locations may contain a pointer to another memory location. We are performing a points-to analysis to identify may-alias relationships for the objects and variables under consideration.

Another essential component is the DDS *annotator*. This component annotates the methods that have a synchronization point and contain harmful statements before the second lock re-

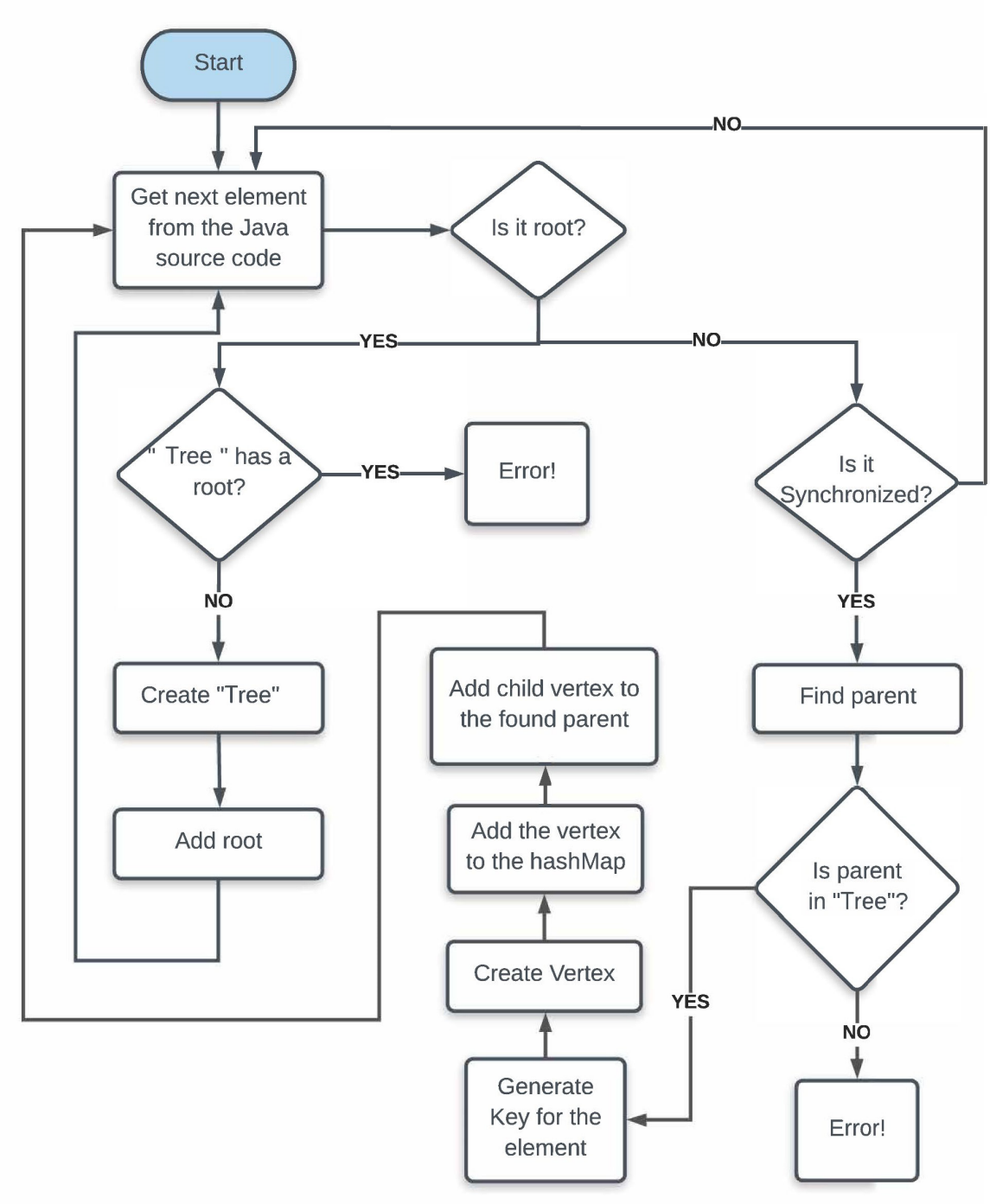

Figure 12: Flowchart for building DDS preprocessing tree during preprocessing stage.

quest. In this step, we introduce STM to the source code by adding *@Atomic* annotation to the methods—represented as vertices in the preprocessing's tree—that contain harmful statements.

4.1.2 Building the Tree

To build the tree, the analysis iterates over the source code, as shown in Figure 12. It builds a vertex for each synchronization point (e.g., synchronized block, synchronized method, lock request, synchronized method invocation). Additionally, we store all vertices on a map for lookup purposes. We create a unique key for each vertex in the source code. The root element—the first vertex to be added to the tree—is a compilation time root package (CtRootPackage). Each vertex has a parent vertex, which is either the root vertex or another synchronization point represented by a vertex in the tree as illustrated in Figure 13. If a vertex is another synchronization point vertex's parent, that means a nested synchronization is in the code. All vertices that have been added to the tree until this point are called direct vertices, referring to their direct relationship to a synchronization point (e.g., the synchronized method A and the method C which contains a synchronized block in Figure 14).

When we add all the direct synchronization points to the preprocessing's tree as vertices, we start to look for all the methods that invoke one of the synchronized vertices in our tree, which we call indirect vertices (e.g., method B, which invokes synchronized method A in Figure 14). We call it indirect because it does not have a synchronization point inside these methods and instead points to either a method that has a synchronization point inside it or to an indirect synchronization point. This process is done recursively until all the calls for a direct or indirect vertex have been discovered and added to the tree, as seen in Figure 14.

```
// 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 13: Algorithm for building preprocessing's tree.

Only one root is in any set of application source codes. Figure 14 shows an example of a DDS preprocessing tree.

4.2 Identifying Harmful Statements

The tree construction is based on the program's AST. We specifically consider paths between lock acquisition points. The statements along these paths are examined to determine if any

write operation is being performed on a shared object. If such a statement is found, then it is considered a *harmful statement*. In this case, we cannot preempt the mutex lock that protects that statement. We examine the entire tree for the presence of any *harmful statements*.

In a deadlock situation, a parent and a child monitor lock exist. The edge between these two vertices is the set of statements the parent monitor protects. The difference between the methods used to analyze potentially harmful statements is determined by the parent monitor type (e.g., synchronized methods and lock statements). Potentially harmful statements in a synchronized method must examine the write operations to object data members and object data members' compile-time aliasing. Unlike lock statements, potentially harmful statements must inspect any write operation to the object data members, their compile-time aliasing, and the lock object aliasing.

However, numerous challenges are related to the inclusion of reentrant locks in the preprocessing step, as discussed in Chapter 1, Section 1.3. The first challenge concerns the scope of the locking. Reentrant locks do not clearly define where a lock's scope ends, necessitating an extensive search for all possible closing points (i.e., *unlock* statements) to be included in the lock vertex. The pairing process of lock and unlock invocations is conservative. In way similar to Gerakios et al. (61), if the unlock invocation appears in a branch (not the same as where the lock invocation executed), we do not pair the lock and the unlock invocations.

The second challenge is that reentrant locks are performed on a lock object instance that can be locked and unlocked multiple times. Because our analysis requires the pairs to be matched exactly, additional precautions must be taken to identify the correct pairing. One lock can have

multiple unlocks based on the control flow graph, and we must consider all of these paths. This challenge is further complicated in cases in which unlocking statements are missing because a program without unlocking could be running. In this case, we inspect all the paths reachable from that lock's acquisition point.

Object aliasing in Java programs also causes subtle issues in determining *harmful statements*. It cannot be easily determined which object is synchronized if an alias of the synchronized object exists. For example, if object `p` is synchronized and object `q` is updated in the path of a synchronized point in the tree, we cannot assume `p` differs from `q` because `q` and `p` could be the same object due to aliasing. Unfortunately, it is impossible to completely resolve the aliasing issue without the runtime support for arbitrary programs. Therefore, aliasing is the third critical challenge for preprocessing.

To address aliasing, the preprocessing phase of *DDS* requires additional functionality to evaluate aliasing occurrences resolvable at compile time. Unfortunately, not all occurrences of aliasing can be resolved during compilation. We use a conservative approach by considering all aliasing possibilities not resolved at compile time to be a *harmful statement*. This strategy ensures that the program data are never left in an inconsistent state even though such data may affect the runtime performance of *DDS*.

4.2.1 Handling Loops and Branches

In program analysis, loops and recursions are considered to be challenging structures, yet they are crucial components in the development of algorithms and software. Both techniques help the programmer execute a set of instructions for repeating all or part of the codes (62).

Due to the complexities and time involved in the resolution of loops, researchers assume execution of a single iteration as feasible for static analysis (63; 64). In proposing a framework for a static loop analysis, Lokuciejewski et al. (63) explained the strength of a single execution loop. They indicated that each time a loop is executed in static analysis, it must meet specific constraints assuming the structure of the loop and the type of statements within the loop body. With multiple iterations or recursions, it becomes quite challenging for the programmer to satisfy these conditions. Single-step evaluation allows researchers to scan the loop headers and bodies to ensure all conditions are satisfied before the evaluation. Similarly, in the JavaScript applications, Madsen et al. (64) reported that appropriate unfolding of the event loop is essential. They considered it important for dealing with the constraints of a single nondeterministic loop. In this regard, we assumed single loop execution for investigating Deadlock Detector and Solver (DDS) in this research. Figure 15 indicates an example showing reentrant locks in Java. Loops (in `LoopNestedLock` class) are used to illustrate the DDS preprocessing phase in identifying harmful statements. We examine each statement in the loop body statements. In case of the assignment statements (i.e., $=$, $+=$ or $-=$), we perform aliasing analysis to find all potential aliasing.

Second, like us, past researchers have assumed the execution of both the if and else branches. The code in Figure 15 illustrates how we used conditional construct, i.e.,

$$\texttt{if(i\%2} > \texttt{0)...else...}$$

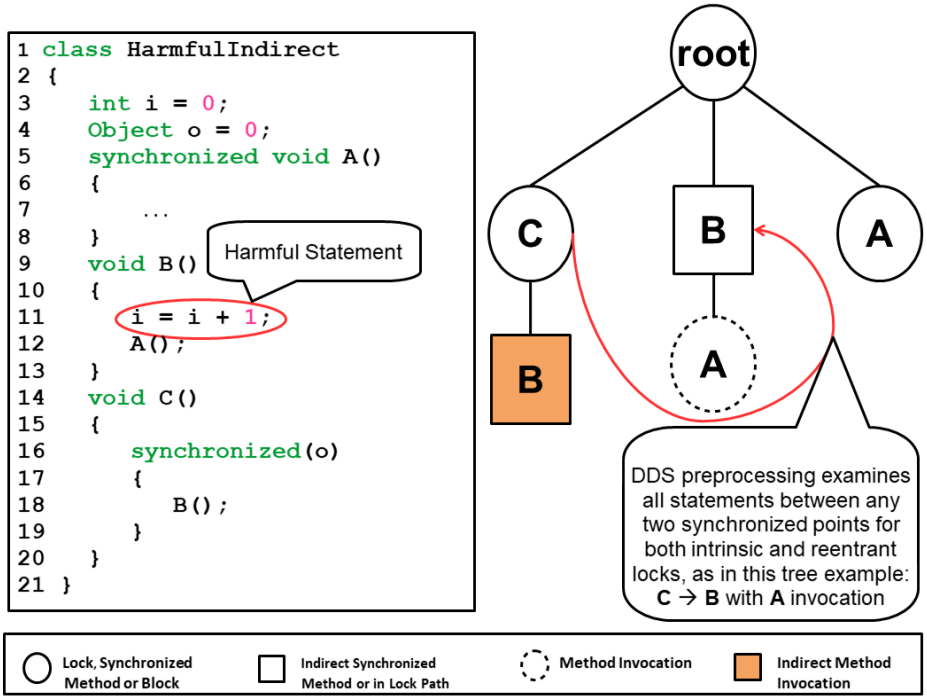

Figure 14: Example of a tree being built during the preprocessing stage of the DDS.

As per Lange et al. (65), the conditional constructs such as if-then-else allow programmers to carry out conditional branching of the program. Conditional constructs exhibit the capacity to yield either branch of a conditional branch through a nondeterministic step, which is extremely important in static analysis. Therefore, we ensured that statements in all the branches are executed.

Third, in this static analysis, we assumed array as a single lock. The same assumption has been used by the other researchers (61). We mitigated the issue of determining the index of such an array that is only determinable at runtime.

4.2.2 Illustrative Examples

4.2.2.1 Intrinsic Locks

Looking at the example in Figure 14, we can see one synchronized method A and one synchronized block in method C, which represent two vertices in the tree. In addition, we have an invocation of the method A in the scope of method B. B is not a direct synchronized method, but it contains an invocation for a direct method A; thus, we add B and the invocation of A to the tree of synchronized points. As a final step, we add all the invocations of B—the indirect method—to the tree to examine the tree paths for the presence of any *harmful statements*. Based on the example presented, if we encounter a deadlock that involves an invocation of method C, then we cannot preempt the monitor of the block in the method C because C invokes B in the synchronized block that contains a *harmful statement*.

To solve this problem and ensure program consistency, we guard the method C using STM. In addition, we consider aliasing, which could be resolved at compile time as part of the *harmful statement* analysis. In our example, we must inspect the integer i and the object o. This inspection guarantees that we do not modify an alias object between the two synchronization points. If the aliasing cannot be resolved, we mark that statement as a *harmful statement* in line with our conservative strategy.

4.2.2.2 Reentrant Locks

When we begin parsing the code, we create the root vertex. The code is parsed and searched for methods that lock an object. These methods are considered as critical methods that can potentially contain harmful statements. Our tree comprises critical methods that are repre-

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Lock;

class LoopNestedLock
{
    public final static int TOTAL_LOCKS = 4;
    public Lock m_locks[];
    public int m_ivar;

    public static void main(String argv[])
    {
      LoopNestedLock lnl = new LoopNestedLock();
      lnl.loop(false);
    }

    public LoopNestedLock()
    {
      m_locks = new ReentrantLock[TOTAL_LOCKS];
      for (int j = 0; j < TOTAL_LOCKS; j++)
          m_locks[j] = new ReentrantLock();
      m_ivar = 0;
    }

    public void harmful_method()
    {
      m_ivar++;   // harmful statement
    }

    public void loop(boolean include)
    {
      int j = 0;
      for (int i = 0; i<TOTAL_LOCKS*2; i++)
      {
          if (i%2>0)
          {
            // statement B'
            m_locks[(j+1)%TOTAL_LOCKS].unlock();

            // statement A'
            m_locks[j%TOTAL_LOCKS].unlock();

            System.out.println("unlocking: " + ((j+1)%TOTAL_LOCKS) + "," + (j%TOTAL_LOCKS));
            j += 2;
          }
          else
          {
            // statement A
            m_locks[j%TOTAL_LOCKS].lock();

            // method contains a harmful statement
            harmful_method();

            // statement B
            m_locks[(j+1)%TOTAL_LOCKS].lock();

            System.out.println("locking: " + (j%TOTAL_LOCKS) + "," + ((j+1)%TOTAL_LOCKS));
          }
       }
    }
}
```

Figure 15: An example showing reentrant locks in Java.

sented in the tree by their position in any given call stack. We create a critical method for each unique lock in a method.

Looking at the example presented in Figure 15, both Statement A and Statement B are critical methods. Each critical method is represented as a unique vertex in the tree.

Static code analysis of loops assumes a single execution of loops. This is justified by allowing the analysis to take the most aggressive approach to finding harmful statements.

As for if-statements, we always assure both the if and else branch are executed. The conditional state is based on runtime and cannot be always determined during static code analysis; thus, we use the aggressive approach of considering all branches as taken when finding harmful statements.

In static analysis of an array of locks, we assume the array is a single lock. This is done to mitigate the issue of determining the index of such array that is only determinable at runtime.

In Figure 15, a single method contains two different locks. This will be represented in our tree as a parent relationship between the two locks in the same method. These are considered as two different critical methods in our tree as distinct vertices.

We traverse the tree to discover harmful statements that are identified by statements between vertices. In the example presented in Figure 15, the `harmful_method` contains a harmful statement and is in between two critical method vertices. This statement modifies a data member, resulting in it qualifying as a harmful statement.

In addition, we search for reference aliasing to handle assignments of lack objects in the static code analysis. Each alias must be followed for harmful statements.

4.3 Guarding Harmful Statements

We start by providing a conceptual overview of DeuceSTM, followed by applying the tool annotation directive to our previously explained examples.

4.3.1 Conceptual Overview of DeuceSTM

One of the biggest challenges in applying the DDS approach is the "missing potential victims" problem. In this regard, software transactional memory (STM) methodology, which is a nonblocking methodology, was used in the annotation process. DeuceSTM, a Java-based STM, was applied. It ensured that any write operation was created atomically. Annotation process was crucial for ascertaining the program state in the runtime monitoring at the time of deadlock occurrence. The STM approach provided us with a powerful tool (i.e., the atomic block) that would ease multithreaded programming and allow for additional parallelism. Notably, the statements stated in an atomic block tend to perform as a singular atomic unit. All these statements are either executed collectively or none take effect at all. By using the STM approach in this investigation, we enclosed the method on the top call stack that contains the lock acquisition—that protects the shared memory—via an atomic block. DeuceSTM helped prevent redundant memory accesses and redundant write-set operations, specifically record keeping (66).

Owing to the homogeneity of DeuceSTM's library, Afek et al. (66) augmented all its STM functions to receive an additional inward parameter, `advice`, which is a plain bit set and signifies precalculated information. Because this parameter was used to fine-tune the instrumentation, it was transmitted to the STM write function during the process of writing fields that will remain unread. One bit in particular corresponded to "no-read-after-write." Subsequently, another

pass was used for implementing static analysis methods to identify optimization opportunities specific to redacted STM read and write operations (67). By augmenting the underlying STM compiler's interface, DeuceSTM allowed the acceptance of information , which is necessary to optimize the STM library method call.

We use DeuceSTM to ensure that when the solver algorithm victimizes a lock, the program remains in a consistent state. For this purpose, STM was used to protect the harmful statements and maintain the application in a consistent state. To do so, we first analyzed the source code, which allowed us to identify the program locations where a request for lock acquisitions was made. Subsequently, a tree was created using the AST of the program. The trees main purpose was to exhibit call dependencies among different locations. A single tree was created for each application. Furthermore, paths between lock acquisition points were considered. The paths between lock acquisition points were analyzed specifically to determine whether any write to a shared object was performed. We investigated the entire tree for the presence of any harmful statements to be guarded using STM. DeuceSTM is used to annotate the top method in the call stack, which protects the harmful statement by locking.

Conceptually, with the help of DeuceSTM, it was possible to avoid any addition to the language or changes to the JVM. With the implementation of atomic blocks, we did not need to consider the variables as part of the transaction and avoided the problem of missing victim threads. At the time of execution, STM requested a Java agent that permits DeuceSTM to interrupt every class loaded and deploy it before JVM loads it. Using *@Atomic* directive gets STM to work while program is running even if we do not encounter a deadlock. If a deadlock

occurs, the shared memory is covered by the *@Atomic* directive, such that if we preempt a lock and two threads interleave in the atomic block of code, STM will abort the interleaved transactions and only commit the one that executed the block autonomously.

4.3.2 Applying DeuceSTM to the Presented Example

In the *DDS annotator*, we guard the top method containing a synchronization point in the call stack; if we find a harmful statement between the two nested synchronization points, we still have the lock on that transaction. This way, we can guarantee whether STM takes the action of rollback, which is only possible in the case of a deadlock. If we do not encounter a deadlock, the shared memory is protected by the parent synchronization point. In other words, thanks to the lock, we always have only one thread at a time accessing the shared memory. The only case wherein we may encounter such an alternate scenario is when a deadlock occurs and we preempt a lock from the victimized thread. The preemption results in only one other thread acquiring the preempted lock, thus accessing the shared memory.

Placing the *@Atomic* annotation on the method containing the lock guarantees the transaction is committed while the thread holds the appropriate lock. Generally, we guard the top method in the call stack that contains a synchronization point.

In case of the dining philosophers' example presented in Figure 6, the `unlock` method, which is `putdownChopsticks`, is higher in the call stack; thus, we guard the method containing `unlock` invocation. If the `lock` invocation is higher or at the same level as the call stack where the `unlock` invocation takes place, then we guard the method containing `lock` invocation.

Subsequently, from the collection of identified *harmful statements*, we pass the top methods in the call stack containing the synchronization statements to the *DDS annotator*. This process annotates the methods with the *@Atomic* declarations to guard the statements from preemption, helping maintain consistency. Adding *@Atomic* annotations to a method allows it to perform autonomously. This step results in adding *@Atomic* annotation to method `C` in Figure 14 example. In Figure 15, the *DDS processor* identifies `loop` method to be passed to the *DDS annotator* because it is the top method in the call stack containing the synchronization point (Statement A). Thus, `loop` method is annotated with *@Atomic* to execute autonomously.

CHAPTER 5

DDS RUNTIME

Parts of this chapter have been previously published in:

- E. Aldakheel, U. Buy, "Efficient Run-time Method for Detecting and Resolving Deadlocks in Java Programs," *In 33rd European Conference on Object-Oriented Programming Workshops (ECOOP)*, London, United Kingdom, 2019.
- E. Aldakheel, U. Buy and S. Kaur, "DDS: Deadlock Detector and Solver," *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Memphis, TN, 2018, pp. 216-223. doi: 10.1109/ISSREW.2018.00009
- E. Aldakheel, "Deadlock Detector and Solver (DDS)," *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, Gothenburg, 2018, pp. 512-514.

Our methodology consists of two steps. In the previous chapter, we explained the first stage, which is DDS preprocessing. In this chapter, we explain the core of our methodology, which is the runtime monitoring. In this step, we continuously monitor specific actions (e.g., acquiring and releasing mutex locks) and detect and resolve deadlocks at runtime. We first discuss the runtime architecture components and how they interact and then illustrate how to detect a deadlock using our architecture. Finally, we explain how we resolve the detected deadlocks.

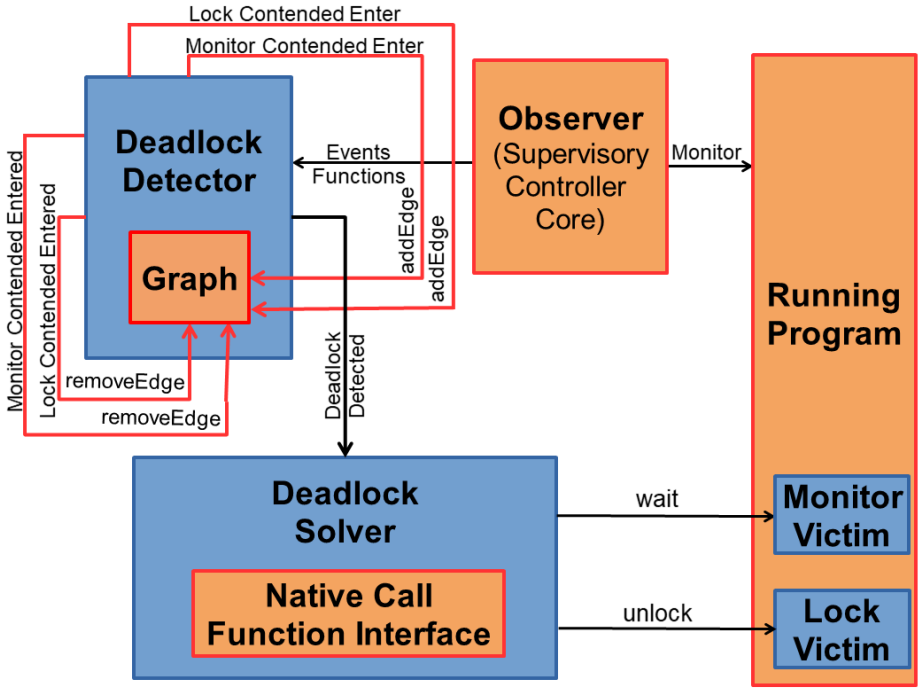

Figure 16: Runtime architecture of Deadlock Detector and Solver (DDS).

5.1 Runtime Architecture Components

The framework for detecting and resolving deadlocks is based on the lock order graph, referred to here as the lock graph, which consists of vertices and directed edges. Each vertex represents a mutex lock held by a thread, and each edge represents a request relationship between two threads for a specific mutex. The presence of a cycle in the lock graph indicates a deadlock.

5.1.1 Observer

At this stage, we monitor a running program on a specific operating system (e.g., Linux and Apple's macOS). The DDS *observer* monitors the running program, as shown in Figure 16. The *observer* is notified of relevant events when they pertain to mutex lock objects, then the *deadlock detector* get notified about relevant events by the *observer*. We record the events and the functions of interest in as a callback. The events occurring in a running program that we monitor are as follows:

- Reentrant lock request, which represents a thread requesting a lock. This request blocks the thread until the lock is free.
- Reentrant lock acquire means that a thread requesting a lock has obtained the lock (the lock currently locked by the requester thread).
- Reentrant lock release occurs when a thread unlocks the locked object (the acquired lock).
- Monitor request results in a thread being in wait status until the lock becomes available or the lock being acquired if the monitor in question is available.
- Monitor lock acquisition happens when the requesting thread gets access to the monitor.
- Monitor lock release occurs when the thread holding the lock releases the lock; thus, the monitor is available for the next requester thread.

The *JVMTI* toolset allows us to define the following callbacks:

- *Lock contended enter* callback function represents a situation in which a thread attempts to acquire a reentrant lock currently held by another thread.

- *Lock contended entered* callback function occurs when a blocked thread (waiting for a lock) acquires the needed lock.
- *Monitor contended enter* callback function occurs when a thread attempts to acquire a monitor lock held by another thread.
- *Monitor contended entered* callback function occurs when a waiting thread acquires the monitor for which it was waiting.

5.1.2 Detector

The detector uses information passed by the *observer* to build and maintain the *lock graph*. Weather a vertex or an edge is added to the *lock graph*, the *deadlock detector* checks for cycles, indicating the existence of a deadlock. Based on the listed events and functions, the *observer* notifies the *deadlock detector* of the need to take the required action. We keep track of the objects and threads. The *deadlock detector* stores the obtained information in maps for lookup and retrieval. For each object, the *deadlock detector* generates a unique hash ID. In each map, it saves objects (e.g., locks and threads) using their hash IDs. When a thread acquires a lock, the *deadlock detector* adds the lock associated with the thread to the owning map. When a thread releases a lock, the *deadlock detector* removes the lock entry from the owning map. In addition, the *deadlock detector* maintains a map of vertices in cycle, which is essential in detecting and resolving deadlocks. The primary functions of the *deadlock detector* are to build the *graph* and call the *deadlock solver* if a cycle is detected, which means a deadlock has been found.

5.1.2.1 DDS Graph

This graph is the core of the *deadlock detector*. The main components of the graph are the vertices and the edges that connect the vertices. The *DDS graph* has directed edges to represent the needs between the vertices—which represent the threads. Thus, an edge e_1 from vertex v_1 to vertex v_2 means that v_1 needs a lock held by v_2.

Each vertex in the graph has a name, its thread name. In addition, each vertex includes a list of edges. Each edge points at an owner vertex. Each vertex has a list of owned locks. The graph has three main functions. These functionalities form the core that detects deadlocks at the runtime. The following list explains each:

- Add edge enables the addition of an edge to the current graph. The addition of an edge represents the need relationship between the two vertices.
- Remove edge occurs when a blocking thread— represented as a vertex in the *DDS graph*— acquires the lock for which the thread was waiting.
- Finding a cycle in the *DDS graph* means that a deadlock exists if the current graph has a cycle.

5.1.3 Solver

The *deadlock solver* is activated when the *deadlock detector* detects a deadlock. The *deadlock solver* is responsible for resolving the detected deadlock. In this component, the *deadlock solver* identifies the victim thread. Then, the *deadlock solver* identifies the participating object (e.g., lock or monitor) that belongs to the victim thread. It obtains the object's class name and the

signature that the *deadlock solver* is going to preempt from the victim thread. Based on the information provided, the *deadlock solver* retrieves the object from the heap. It tags the object based on our tag classification. The *deadlock solver* has the following three tag states:

- "Clear" means that each object is initially in a clear state. The object is also tagged as clear when the preemption process is done—in other words, when the detected deadlock has been resolved.
- The "Unlock" means that an object in the heap has been preempted from the victim thread. It is the object to be locked by a thread other than the victim thread.
- The "Lock" state occurs when the preempted object is again returned to the victim thread. This means that the victim thread again holds the lock object.

5.2 The Detector Algorithm

DDS detects deadlocks by monitoring requests, acquisitions, and releases of mutex locks. The key structure used to detect deadlocks is the lock graph, as illustrated in Section 5.1.2.1. A vertex v_1 represents the lock held by a thread t_1. Suppose that t_1 also requests an unavailable lock modeled by another vertex v_2. In this case, we add an edge from v_1 to v_2 to indicate that the thread holding the lock corresponding to v_1 is waiting for the lock associated with v_2 to be released. Therefore, an edge e_1 from vertex v_1 to vertex v_2 means that thread t_1 needs a resource (i.e., data structure) locked by another thread t_2.

The existence of a cycle in this lock graph represents the presence of a deadlock. Any thread that does not hold a resource does not have any vertices or edges in the lock graph.

Algorithm 1: Runtime DDS detector algorithm for monitor locks.

```
1  Function monitor_contended_enter(thr, obj) /* algorithm to detect deadlocks and
     maintain the graph for the monitor.                                          */
2  begin
       Data: current thread and object
       Result: call DDS solver when a deadlock is detected
3      if !owner.find(obj) then
4      begin
5          thr ⟵ currentowneroftheobject
6          owner[obj] ⟵ thr
7      end
8      else
9      begin
10         ownerTh ⟵ owner.find(obj)
11         graph.addEdge(thr, ownerTh, obj)
           /* checks the graph for a cycle after adding an edge                 */
12
13         if !graph.dag() then
14         begin
15             wait(≈ 20milliseconds)
               /* rechecks for a cycle in case we have communication delays     */
16
17             if !graph.dag() then
18             begin
19                 deadlockDetected ⟵ true
20                 if deadlockDetected then
21                 begin
                       /* call DDS solver algorithm to resolve the detected deadlock */
22
23                     solver(thr, obj)
24                 end
25             end
26         end
27         else
28         begin
29             deadlockDetected ⟵ false
30         end
31     end
32 end
33 Function monitor_contended_entered(thr, obj) /* algorithm to delete an edge from
     the graph                                                                    */
34 begin
       Data: current thread and object.
       Result: deleting edge from the graph.
35     owner.find(obj)
36     owner.erase(obj)
37     graph.removeEdge(thr, obj)
38 end
```

The size of the graph is further optimized as follows: If the running application does not have any contention—that is, if no thread requests a resource that is unavailable at that moment —the lock graph will be empty. For each contention, we add a vertex for each the requesting and owner threads. With each edge addition, we check the graph for the presence of a cycle. For cycle detection, we used a variant of Depth First Search (DFS) in order to search the graph. The algorithm complexity is $O(V + E)$—wherein V refers to the vertexes in totality, while E indicates edges (numerically speaking) in *lock graph*. As soon as a thread acquires the resource for which it has been waiting, we remove the corresponding edge from the graph. In this case, we also update the information associated with v_n to indicate that t_m now holds the corresponding lock. When the thread releases a resource, we remove the corresponding vertex from the graph.

The detector algorithm activates when a resource contention occurs. For the monitor locks, we need to obtain information regarding the requester thread and the owning thread. All the information is contained within the class boundary—the class that uses monitor locks (i.e., synchronized block and synchronized method)—; thus, there is no need to obtain external information. The thread requester t_r, the thread owner t_o, and the resource l comprise the needed information to be hashed and added to the maps. In addition, we need to add the information to the graph. As soon as we add an edge e from t_r to t_o to the graph, as shown in line (11) of Algorithm 1, we check whether the result indicates the formation of a cycle, which would indicate a deadlock exists. However, the formation of a cycle does not always indicate a deadlock. In some cases, the communication delays among the DDS components (i.e.,

observer, detector, and graph) mean that a deadlock has not occurred. For instance, a delay between an event of interest happening in the running program and the corresponding callback being invoked in the DDS imply that the DDS may still have outdated information about the running program. Experiments using Java programs indicate that this delay is usually 10 to 20 milliseconds; however, this delay does not affect the validity of our analysis because deadlocks are persistent. Once a deadlock occurs, it is sustained until the DDS resolves it. For this reason, we recheck the conditions of a circular hold-and-wait pattern after this short delay.

The monitor_contended_entered function is activated when a thread enters the monitor after waiting for another thread to free the monitor. This signal prompts the elimination of edges from the lock graph. Edge removal is achieved by starting from the monitor_contended_entered event that the *observer* detects when monitoring the running program. The *observer* then sends the activation of the monitor_contended_entered event to the *deadlock detector*. Based on the previous signal, the *deadlock detector* then sends a remove-edge request to the graph to remove the designated edge from the graph, as shown in line (37) of Algorithm 1.

We also handle reentrant locks in our DDS methodology. The primary difference between the monitor locks and the reentrant locks is that the reentrant locks are objects and do not have any associations with a specific class or scope. This difference means that the information is not contained in a specific place; thus, we have to search in the heap for the lock object as well as the requester and owner threads. Afterward, we add them to the maps and lock graph, just as we did for the monitor locks. Similar to what we did with the monitor locks, we add an edge to the graph with every lock_contended_enter call, which represents a resource contention, as

Algorithm 2: Runtime DDS detector algorithm for reentrant locks.

```
1  Function lock_contended_enter(thr, obj) /* Algorithm to detect deadlocks by
     building the lock graph when a thread attempts to acquire an unavailable lock.
     */
2  begin
       Data: current thread and object(lock)
       Result: call DDS solver when a deadlock is detected
3      obj ⟵ getBlocker(thr)
4      owner ⟵ getOwner(obj)
5      if !owner then
6      begin
7          error ⟵ OwnerNotFound
8          exit()
9      end
10     else
11     begin
12         ownerTh ⟵ owner
13         graph.addEdge(thr, ownerTh, obj)
           /* checks the graph for a cycle after adding an edge                    */
14
15         if !graph.dag() then
16         begin
17             wait(≈ 20milliseconds)
               /* rechecks for a cycle in the event of communication delays        */
18
19             if !graph.dag() then
20             begin
21                 deadlockDetected ⟵ true
22                 if deadlockDetected then
23                 begin
                       /* call DDS solver to resolve the detected deadlock        */
24
25                     solver(thr, obj)
26                 end
27             end
28         end
29         else
30         begin
31             deadlockDetected ⟵ false
32         end
33     end
34 end
35 Function lock_contended_entered(thr, obj) /* Algorithm to delete edge from the
     graph when a thread obtains a for which it has been waiting                  */
36 begin
       Data: current thread and object(lock)
       Result: deleting edge from the graph
37     owner ⟵ getOwner(obj)
38     owner.erase(obj)
39     graph.removeEdge(thr, obj)
40 end
```

shown in line (13) of Algorithm 2. The lock_contended_enter is applied when a thread tries to lock a reentrant lock that another thread has already acquired.

A blocked thread acquiring a lock for which it has been waiting activates the lock_contended_entered function. The lock_contended_entered signal received from the *observer* causes an edge removal, as shown in line (39) of Algorithm 2.

5.3 The Solver Algorithm

A deadlock has occurred only if a cycle exists in the lock graph. In this case, the *detector* notifies the *solver* of the need to choose a victim thread. The *solver* forces the victim thread to release the lock by issuing a *wait* or *unlock* statement on the victim thread; then, another thread that needs that lock can acquire it. After the second thread releases the lock, the solver requests a *notify* or *lock* statement on the victim thread, allowing the victim thread to obtain the lost lock and continue processing.

The *solver* resolves the deadlock by preempting a mutex lock from one of the threads involved in creating the deadlock. When the deadlock is detected, the *solver* forcibly releases a lock in the chain, resolving the reported deadlock. The *solver* selects the object and the type of method to be performed on the victim thread. In a monitor lock case, we utilize the wait method, as presented in line (8) of Algorithm 3. Attributing *solver* to the deadlock issuing a wait to the victim thread, the victim thread releases the monitor that the same thread held. Subsequently, it blocks itself until the monitor is available (is released).

In the reentrant lock case, we obtain the locked object then employ the unlock method, as presented in line (10) of Algorithm 4. In addition, we tag the object with “unlock” status to

Algorithm 3: Runtime DDS solver algorithm for monitor locks.

```
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 ⟵ graph.getVertexsInCycle(thr)
4     vicThr ⟵ randPickOne(thrsInCy)
      /* gets object class and class signature                        */
5
6     mID ⟵ GetMethodID(cls, "wait", sig)
      /* call wait() on the victim thread with passing time out argument
         (tOut)                                                       */
7
8     CallVoidMethod(obj, mID, tOut)
9 end
```

indicate that this object has been preempted from the victim thread. When the deadlock *solver* issues an unlock on the victim thread's object, the victim thread releases the lock. Subsequently, it blocks itself until the lock is released (available again). As soon as the current thread—not the victim thread—releases the lock, we tag the object with "lock" and relock it on the victim thread by notifying the victim thread of the lock availability.

Algorithm 4: Runtime DDS solver algorithm for reentrant locks.

```
1  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.                                                    */
2  begin
       Data: current thread and object(lock)
       Result: resolve the detected deadlock by using unlock
3      thrsInCy ⟵ graph.getVertexsInCycle()
4      vicThr ⟵ pickVictimthread(thrsInCy)
       /* get object class and class signature to obtain the method id   */
5
6      mID ⟵ GetStaticMethodID(cls,"unlock",sig)
       /* tag the ''clear'' object with ''unlock'' status                */
7
8      obj.tag(unlock)
       /* call unlock() on the victim thread                             */
9
10     CallStaticBooleanMethod(cls, mID, obj)
       /* tag the ''unlock'' object with ''lock'' status and return it back to
          the victim thread                                               */
11
12     obj.tag(lock)
       /* get object class and class signature to obtain the method id to
          re-lock() when the tagged object get freed                     */
13
14     mID ⟵ GetStaticMethodID(cls,"lock",sig)
       /* tag the ''lock'' object with ''clear'' status again; so, no further
          action is needed                                                */
15
16     obj.tag(clear)
       /* call lock() on the victim thread to return the lock back       */
17
18     CallStaticBooleanMethod(cls, mID, obj)
19 end
```

CHAPTER 6

EXPERIMENTAL EVALUATION

Parts of this chapter have been previously published in:

- E. Aldakheel, U. Buy, "Efficient Run-time Method for Detecting and Resolving Deadlocks in Java Programs," *In 33rd European Conference on Object-Oriented Programming Workshops (ECOOP)*, London, UK, 2019.
- E. Aldakheel, U. Buy and S. Kaur, "DDS: Deadlock Detector and Solver," *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Memphis, TN, 2018, pp. 216-223. doi: 10.1109/ISSREW.2018.00009
- E. Aldakheel, "Deadlock Detector and Solver (DDS)," *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, Gothenburg, 2018, pp. 512-514.

We empirically evaluated the effectiveness of the *DDS* in detecting and resolving deadlocks at runtime. We specifically measured the overhead that the *DDS* imposes in detecting and resolving a deadlock. In addition, we empirically evaluated the efficiency of preprocessing in identifying the locks that should not be preempted by DDS when resolving a detected deadlock. We report the preprocessing timing, even though this step is performed once offline before compiling and running the program. The purpose was to ensure that the preprocessing step is performed sufficiently quickly not to incur a long wait time (e.g., hours). We evaluated the

DDS on a suite of multithreaded Java programs. In addition, we compared *DDS* runtime performance with two other deadlock detectors. We ran all experiments on a Linux Ubuntu 16.04 LTS machine with a 1.70 GHz Intel Core i5 processor and 6 GB RAM. This chapter starts with *DDS* implementation details followed by benchmark sets details. Finally, we display our empirical results.

6.1 Implementation Details

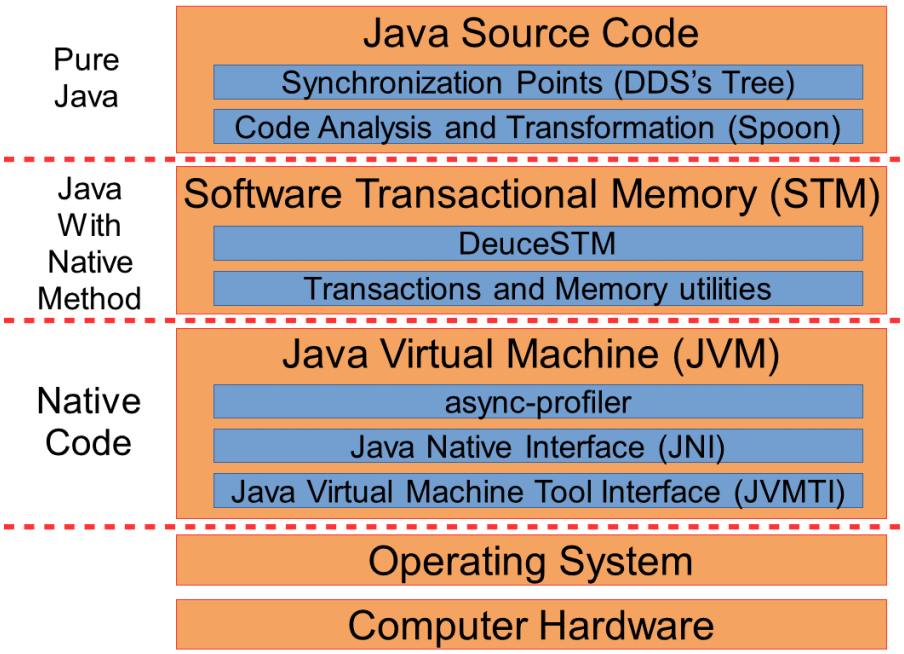

Figure 17: DDS high level layered architecture overview.

The implementation of *DDS* consists of three layers on top of the operating system, as shown in Figure 17. At the bottom is the runtime monitoring layer, which is based on the Java Virtual Machine (JVM). The JVM is a virtual machine to enable the Java program to run on various operating systems. The JVM layer contains a tool to access native code contained in the Java Native Interface (JNI) and the Java Virtual Machine Toolset Interface (JVMTI). The JNI is an interface to enable a native method—written in a language other than Java— to be called in or called back during program execution. We use the JNI to handle object retrieval and lock preemption. The JVMTI is another component for implementing the DDS methodology for the Java program. In particular, the JVMTI is responsible for the verious callbacks that we use to implement our methods (e.g., `monitor_contended_enter` ... etc). The JVMTI is a two-way communication tool interface that monitors and controls the running Java program. The two-way communication is between an "agent" written using the JVMTI and the running program in the JVM. The "agent" is notified of events of interest and can use the JNI to call out and control the execution of the running program. Using the JVMTI, we can inspect the state of a program running in the JVM, thus controlling the execution of the running program. The next layer is the STM layer. Here we use DeuceSTM (34) to access memory utilities and create transactions to guard the *harmful statements*. This layer uses Java enhanced with native methods. The top layer performs the preprocessing step in Java using the Spoon toolkit (68).

6.1.1 Runtime Monitoring

A Java program is executed in the *JVM*, which is then monitored by the *DDS* via the *JVMTI*. The *DDS* runtime implementation consists of two parts: the *detector agent* and the

solver agent. The *JVMTI* allows the two *DDS* agents to communicate with the Java program, which runs in the JVM.

The *JVMTI* provides third parties, such as our agents, with access to the environment of the running Java program. The *JVMTI* also allows our agents to register callback functions that are invoked whenever certain events occur in the *JVM*. We register various callbacks. Function `monitor_contended_enter` is called whenever a thread in the running program requests an intrinsic lock currently held by another thread. Likewise, `lock_contended_enter` is called whenever a thread requests a reentrant lock. The *JVMTI* invokes callbacks `monitor_contended_entered` and `lock_contended_entered` when a thread actually acquires a lock it has requested. We use these callbacks to keep the lock graph up to date.

Figure 18 shows how the deadlock *detector* and *solver* agents interact. The *JVMTI* monitors the *JVM* and notifies the deadlock *detector* agent of any events. The *JVMTI* notifies the deadlock *detector* agent whenever a resource is acquired or is waiting to be relinquished based on the *JVM* callbacks. Finally, the *solver* agent invokes either function *wait* or function *unlock* on the victim thread through the *JNI*, depending on whether an intrinsic lock or reentrant lock is preempted. We use the *async-profiler* tool (69) to obtain access to the reentrant lock object involved in the deadlock.

The registered callbacks allow us to define and maintain the *lock graph*. As described in Chapter 5, vertices in the graph represent locks held by threads. Edges represent threads waiting for a lock held by another thread. Edges are typically added to a graph as part of `monitor_contended_enter` and `lock_contended_enter` functionality. See Figure 18. Edges

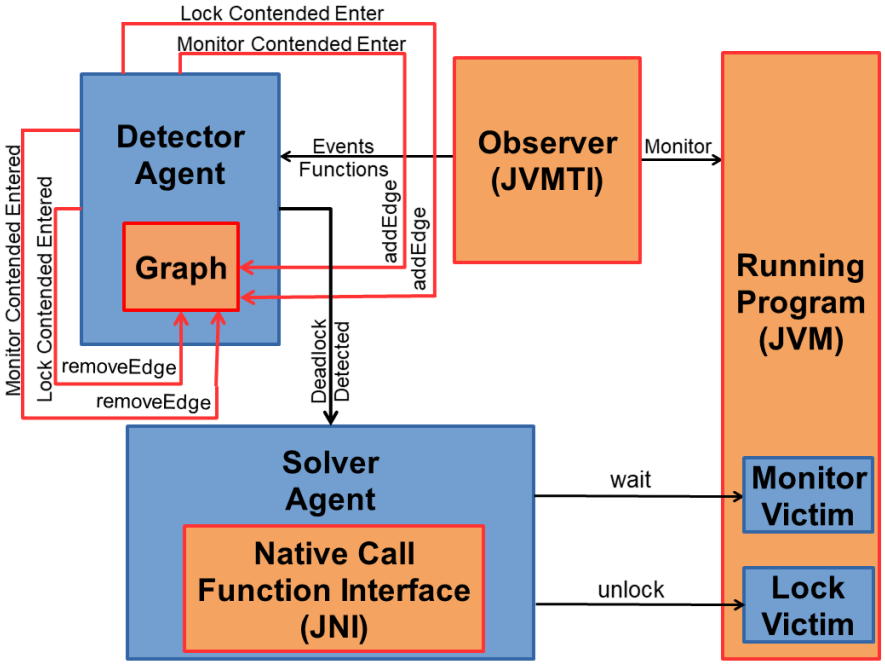

Figure 18: Architecture of Deadlock Detector and Solver (DDS) to monitor program written in Java Language.

are removed from the graph as part of `monitor_contended_entered` and `lock_contended_entered` functionality. After adding an edge to the graph, we check whether a cycle or, in this case, a deadlock was formed as a result. The formation of a cycle does not always mean that a deadlock occurred. In some cases, the communication delays among the JVM, JVMTI, and DDS's detector agents mean that a deadlock did not in fact occur. For this reason, we recheck the conditions of a circular hold-and-wait pattern after a short delay (≈20 milliseconds).

Once we know that a deadlock has in fact occurred, we use the *solver* agent to remove the deadlock. The *solver* agent uses the *JNI* to preempt a lock on the victim thread and resolve

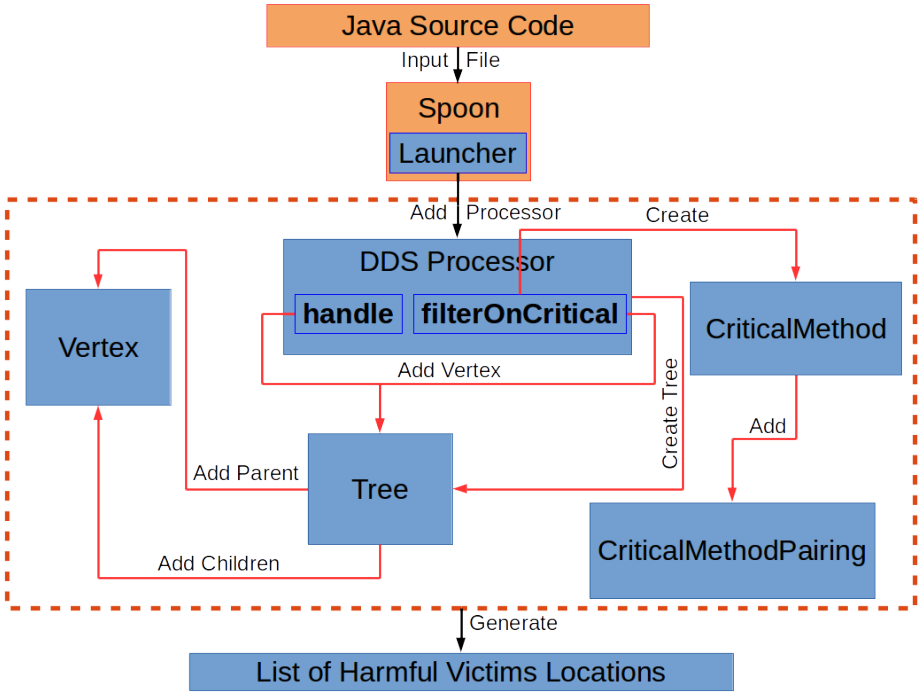

Figure 19: DDS preprocessing architecture to handle different type of locking.

the deadlock. The JNI allows Java code running in the JVM to "call out" external functions. The JNI also allows external functions to "call in" Java functions contained in the JVM. We use the latter capability of the JNI to make a call from our agent to a thread holding a lock involved in a circular hold-and-wait pattern. We specifically call the *wait* method in order to release an intrinsic lock in the hold-and-wait cycle. The thread receiving such a call is awakened and given back the lock after the latter becomes available again. We call *unlock* on a reentrant lock involved in the deadlock cycle to resolve the deadlock.

6.1.2 Preprocessor

We perform the preprocessor step on Java source code using the Spoon toolkit (68). Figure 19 shows the flow of control in the preprocessor. Using Spoon's launcher, we implement our processor. The processor builds a tree whose vertices are statements acquiring either intrinsic or reentrant locks. To handle locking and unlocking of reentrant locks, we trace through the flow of calls, starting from each lock statement. In particular, the preprocessor allows us to identify the program locations evidencing the acquisition of a second lock by a thread after the structures involved in a first lock acquisition have been accessed and modified. In that case, the thread is not a good candidate to be victimized. In this manner, we determine whether a shared object is modified before a second lock is acquired. When this happens, we treat the locking statement as harmful, and we annotate the top method—that contains the lock acquisition— in the call stack, which protects the harmful statement with the label `@Atomic`. DeuceSTM (34) uses this annotation to provide transactional atomicity to subsequent memory-writing operations in that block of code which contains the harmful statement.

As the code is parsed, we create a unique key for each element. Using the key, we create a hashmap. For each synchronized block, synchronized method, or invocation, we add a *vertex* to the preprocessing *tree*. For lock invocations, we set the `filterOnCritical`—which is responsible for detecting reentrant lock function calls to add them to the preprocessing *tree*—to lock, assuming a lock method was executed first before the unlock method. Then, we search for the lock `CriticalMethod`—the data structure for storing critical methods (e.g., lock() and unlock())—structure in the hashmap. We parse the invocation into a return, class/interface type, and class

signature to compare those attributes to the hashmap, which contains the CriticalMethod. Then, we search in the hashmap. If the hashmap is not populated with an entry for this key, we provide a method to convert the key from a search to a new entry. Afterward, we search for an unlock invocation for the same instance (that locked by the lock() method, such as m_locks in Figure 15) to add the pair (of lock() and unlock()) to the CriticalMethodPairing—which is responsible for pairing the beginning (i.e., lock()) and the ending (i.e., unlock()) of a critical section. For the purpose of pairing, any unlock invocation in the condition statement is not the final stop in our search for a *harmful statement* because, at the time of compilation, we do not know whether that condition will be taken. For the lock invocation, we examine all the reachable statements for that lock point. If there are no condition statements between the invocation of a lock for a specific instance and the invocation of the unlock for the same instance, we can pair the two invocations using CriticalMethodPairing.

6.2 Benchmark Sets

We ran two sets of experiments. The first set of benchmarks is from the Java Grande suite (70). It comprises the following programs:

- *barrierbench*, which simulates cyclic barriers in Java,
- *crypt*, which encrypts and decrypts an array of size N,
- *moldyn*, which is a molecular dynamics simulation,
- *lufact*, which performs a Linpack Lower-Upper factorization, preceded by a triangular solve,

TABLE II: BENCHMARK DETAILS, INCLUDING PROGRAM LENGTH IN LINES OF JAVA SOURCE CODE, NUMBER OF *SYNCHRONIZED* STATEMENTS CONTAINED THEREIN, AND SIZE OF THE INPUT SETS.

Benchmark Name	Line Count	Num. of Syn. Stat.	First Input Size Set	Second Input Size Set
elevator	587	8	8 floors	50 floors
tsp	706	6	20 cities	25 cities
barrierbench	710	15	10,000 times	5,000 times
sor	783	14	1,000,000 elements	2,250,000 elements
crypt	1210	14	20,000,000 elements	50,000,000 elements
moldyn	1,451	14	2,048 particles	6,656 particles
lufact	1,563	14	1,000 elements	2,000 elements
raytracer	2,004	15	150 × 150 pixels	500 × 500 pixels
montecarlo	3,702	14	10,000 samples	60,000 samples
hedc	26,577	467	566 bytes	N/A
fop	624,432	43	45.8 kB	102.6 kB

- *raytracer*, which is used to generate an image from different objects by tracing the path of the light and simulating its effect with virtual objects, and
- *montecarlo*, a computational algorithm that relies on repetitive random sampling to produce numerical results.

We also added the following four benchmarks from Eidgenössische Technische Hochschule (ETH) Zurich (71), in English the Swiss Federal Institute of Technology in Zurich, and the *fop* program from the Apache Software Foundation (72) to the first set:

- *elevator*, which simulates elevator operations,
- *tsp*, a solver for the traveling salesman problem,
- *sor*, or successive over-relaxation, and

- *hedc*, a web-crawler kernel application.

The *fop* benchmark renders an XML file, which includes XSL formatting objects in a page layout.

More details on the selected benchmarks are presented in Table II.

The second benchmark set comprises deadlocking and nondeadlocking versions of the popular dining philosophers' problem example. Our dining philosophers benchmark set has four different versions of the dining philosophers' program. There are two control versions, which are deadlock free. The main purpose of these two control experiments is to determine CPU times, elapsed times, and *DDS* overhead when no deadlocks can occur. The other two dining philosophers programs are not deadlock free. For the latter versions, our intent was to measure the overhead for detecting and resolving deadlocks. The four dining philosophers programs are described below.

1. A nondeadlocking dining philosophers program, which uses intrinsic locks (named "NDD Synch" in Table IV).
2. A nondeadlocking dining philosophers program, which uses reentrant locks (named "NDD Lock" in Table IV).
3. A deadlocking dining philosophers program, which uses intrinsic locks (named "DD Synch" in Table IV).
4. A deadlocking dining philosophers program, which uses reentrant locks (named "DD Lock" in Table IV).

Each philosopher alternates between eating and thinking by locking two forks shared with neighboring philosophers. The number of forks is equal to the number of philosophers. The program terminates after each philosopher has eaten 100 times. We used the intrinsic lock and reentrant lock to synchronize the forks. The number of philosophers ranged from 10 to 100 to evaluate the scalability of the agents.

6.3 Empirical Results

We ran the benchmarks 60 times and recorded the average runtime to compute the overhead of both the preprocessor and runtime monitoring. For the deadlocking benchmarks, we checked whether the agent detected and resolved the deadlocks. We also ran the nondeadlocking versions with and without runtime monitoring and compared the runtime of the two versions. We also measured the *DDS* overhead with the selected benchmarks in Table II. For each run, the elapsed time and CPU time were recorded for each benchmark.

As of this writing *DDS* resolves deadlocks involving an arbitrary number of intrinsic locks and up to two reentrant locks because of an issue related to obtain the lock object from the heap and maintain it. We are currently expanding *DDS* to resolve deadlocks involving multiple reentrant locks.

6.3.1 DDS Preprocessing Evaluation

We evaluated the preprocessor based on three criteria. First, we checked that the preprocessor detects *harmful statements* correctly while also taking into account aliasing, which can be resolved at time of compilation. Second, we checked that methods containing harmful

statements were guarded with the "@Atomic" annotation. Last, we measured the overhead of preprocessing applications in our benchmark set.

To test the accuracy of the preprocessor in detecting *harmful statements*, we covered different scenarios depending on the presence or absence of *harmful statements*. There were cases in which a *harmful statement* was present between two successive lock requests. Additionally, we introduced *harmful statements* distant by several function calls from the original lock acquisition to test the preprocessor. Furthermore, we introduced synchronized statements in class constructors. We also added a test set for aliasing, including test cases that could be resolved during compilation and cases that had to be considered *harmful statements*. On the whole, we had over 50 test cases to cover different types and combinations of the needed scenarios.

TABLE III: *DDS* PREPROCESSOR TIMINGS FOR THE CHOSEN BENCHMARKS.

Benchmark Name	CPU Time in Seconds	Elapsed Time in Seconds	Number of Vertices
elevator	11.05	3.95	8
tsp	8.03	2.43	11
barrierbench	11.19	3.01	30
sor	10.29	3.08	22
crypt	10.59	3.06	22
moldyn	14.43	4.31	26
lufact	10.24	3.3	22
raytracer	14.59	4.58	32
montecarlo	15.64	5.23	26
hedc	256.77	223.57	745

For all our test cases, our preprocessor successfully detected the injected *harmful statements* and any aliasing that could occur at time of compilation. The elapsed times for all test cases were under 2 seconds, and the CPU time never exceeded 8 seconds (adding the CPU time for all four cores of our hardware platform).

All our benchmark applications performed the preprocessor step successfully. We artificially injected *harmful statements* to evaluate both the preprocessor and the runtime monitoring under different conditions. As for time measurement, the elapsed and CPU time grew linearly as the number of the vertices increased, as shown in Table III. Note that elapsed times were generally lower than CPU times because we ran our experiments on quad-core hardware.

6.3.2 DDS Runtime Evaluation

In this section, we discuss the overhead introduced using *DDS* for both sets of benchmarks. We analyzed the dining philosophers programs in the first subsection to compare the overhead introduced by resolving deadlocks in the deadlocked version of the dining philosophers problem. Subsequently, we showed how *DDS* affects the runtime of selected real-world applications. Finally, we measured the overhead of using DeuceSTM by artificially injecting *harmful statement* in selected benchmarks.

6.3.2.1 Dining Philosophers Examples

We sought to evaluate the effectiveness of the *DDS* in detecting and resolving deadlocks. We recorded the execution time for the four dining philosophers programs—illustrated in Section 6.2 —to evaluate the efficiency of the agents.

TABLE IV: COMPARISON BETWEEN DD (DEADLOCKING DINING PHILOSOPHER VERSIONS) AND NDD (NON-DEADLOCKING DINING PHILOSOPHER VERSIONS), INCLUDING THE AVERAGE TIME MEASUREMENT IN SECONDS FOR BOTH CPU TIME AND ELAPSED TIME WITH THE AGENT SUPERVISION FOR ALL FOUR VERSIONS; CPU TIME AND ELAPSED TIME OVERHEAD PERCENTAGES; AND THE MINIMUM AND MAXIMUM TIME MEASUREMENTS IN SECONDS FOR CPU TIME.

DP Program		Elapsed Overhead %	CPU Overhead %	Average Elapsed Time	Average CPU Time
#of Ph.	DP Version				
10	NDD Synch	1.53	1.25	2.53	9.26
	DD Synch	—	—	4.74	10.17
	NDD Lock	1.02	1.31	1.54	9.26
	DD Lock	—	—	2.7	9.22
20	NDD Synch	2	1.3	4.75	18.18
	DD Synch	—	—	7.63	19.7
	NDD Lock	1.91	1.27	4.76	18.19
	DD Lock	—	—	5.07	18.09
40	NDD Synch	1.08	0.81	9.15	35.63
	DD Synch	—	—	10	38.18
	NDD Lock	1.07	0.78	9.14	35.63
	DD Lock	—	—	9.27	35.71
80	NDD Synch	1.75	1.55	18.05	70.62
	DD Synch	—	—	22.83	78.81
	NDD Lock	1.47	1.02	17.95	70.26
	DD Lock	—	—	17.94	70.32
100	NDD Synch	1.46	1.1	22.34	87.58
	DD Synch	—	—	28.16	99.72
	NDD Lock	1.44	1.06	22.41	87.63
	DD Lock	—	—	23.15	87.99

The agents ran in parallel with the program, and the timing differences between each of the variations and the overall performance were measured as shown in Table IV. A median growth of 1.18% in CPU time was observed for nondeadlocking versions due to runtime monitoring, whereas the median growth was 1.47% for the elapsed time. These numbers represent a linear relationship between the time taken by the agents and the number of philosophers (threads).

DDS was able to identify all deadlocks that occurred. For the "DD Synch" version, *DDS* successfully detected and resolved all deadlocks regardless of the number of locks involved in the deadlock cycle. For the "DD Lock" version, *DDS* always succeeded in detecting all deadlocks regardless of the number of the locks involved in a deadlock. However, on some occasions, *DDS* was not successful in resolving deadlocks that involved more than two reentrant locks due to lock object retrieval. We are currently investigating ways to address this limitation of the *DDS*. It is worth mentioning that the vast majority of the deadlocks in real applications involve only two locks (73).

Based on the observed timing in the deadlocking versions, we can state that resolving a deadlock within the deadlocked versions incurs only a 4.6% overhead for the CPU time and 5% overhead for the elapsed time.

6.3.2.2 Real-World Applications

We here report empirical results from the two benchmark sets described earlier. For most benchmarks, we evaluated the *DDS* with two different input sizes and three different thread numbers, namely 4, 8, and 16 threads. The elapsed time overhead and CPU time overhead for both input sizes and different numbers of threads for all our benchmarks never exceeded 20%

of the original program's runtime. Our empirical results showed a linear relationship between the overhead imposed by the *DDS* and the number of threads. (See Table V.)

Table V includes the columns "Benchmark," which represents the benchmark name, and "Setup," which in turn corresponds to the input size for each run and the number of threads to run the experiment. Furthermore, the table lists the elapsed time, CPU time averages, and overhead percentage for each benchmark.

The "barrierbench" benchmark unexpectedly reported some negative CPU overheads. These negative percentages were less than 2%, which is not significant. The main reason behind these numbers is that "barrierbench" does not perform any heavy CPU operations; instead, it has been designed to mimic the behavior of Java barriers. Therefore, most of the time threads wait for other threads to proceed to the next step. This behavior introduces a degree of randomness in the resulting runtimes.

In general, as we increased the input size, the overheads increased. Some of the benchmarks showed lower overheads when we increased the input size. These benchmarks had a high thread contention for locks with the smaller input size. As we increased the input size of these benchmarks, we lowered the contention rate, which resulted in faster execution and lower overhead percentages.

Based on the recorded timing in Table V for the selected benchmarks, we concluded that using our supervisory controller, on average, incurred only 7.02% overhead for the CPU time and 4.88% overhead for the elapsed time. These percentages showed when no harmful statements were present; we believe that these numbers are quite reasonable.

TABLE V: REAL APPLICATIONS' TIMING MEASUREMENT INCLUDING THE AVERAGE TIME MEASUREMENT IN SECONDS FOR BOTH CPU TIME AND ELAPSED TIME WITH THE AGENT SUPERVISION FOR EACH BENCHMARK; CPU TIME AND ELAPSED TIME OVERHEAD PERCENTAGES; AND THE MINIMUM AND MAXIMUM TIME MEASUREMENTS IN SECONDS FOR CPU TIME.

Benchmark	Setup		Elapsed Overhead %	CPU Overhead %	Average Elapse Time	Average CPU Time
	Input Size	#of Th.				
elevator	First Input	4	0.69	9.1394	21.94	0.69
		8	0.69	8.12	21.93	0.77
		16	0.69	8.24	21.94	0.86
	Second Input	4	0.28	8.61	54.33	0.75
		8	0.3	7.37	51.64	0.81
		16	0.29	10.37	51.64	0.93
tsp	First Input	4	1.27	7.74	0.92	2.56
		8	0.32	7.73	0.92	2.6
		16	1.82	6.99	0.95	2.74
	Second Input	4	3.56	11.22	0.85	2.43
		8	6.83	11.03	0.89	2.54

Continued on next page

Table V – CONTINUED FROM PREVIOUS PAGE

Benchmark	Setup		Elapsed Overhead %	CPU Overhead %	Average Elapse Time	Average CPU Time
	Input Size	#of Th.				
		16	4	9.62	0.96	2.67
barrierbench	First Input	4	5.83	-0.29	26.69	45.1
		8	4.42	0.33	49.83	128.98
		16	0.69	-1.17	54.08	147.72
	Second Input	4	8.22	1.78	31.81	53.28
		8	2.93	-0.53	50.36	129.13
		16	4	-1.45	54.48	147.66
sor	First Input	4	12.49	18.15	1.09	3.17
		8	2.81	3.92	2.76	9.69
		16	1.84	2.87	4.03	14.64
	Second Input	4	14.32	17.56	1.6	4.96
		8	8.64	10.93	3.16	11.17
		16	1.84	2.14	4.16	15.11
moldyn	First Input	4	0.42	1.05	1.2	4.29
		8	3.2	3.48	1.43	5.18

Continued on next page

Table V – CONTINUED FROM PREVIOUS PAGE

Benchmark	Setup		Elapsed Overhead %	CPU Overhead %	Average Elapse Time	Average CPU Time
	Input Size	#of Th.				
moldyn		16	3.88	4.22	2.6	9.89
	Second Input	4	1.23	0.34	7.68	30.02
		8	1.3	0.25	8.93	34.97
		16	1.06	0.68	13.48	52.97
lufact	First Input	4	0.65	6.92	0.39	0.96
		8	3.78	5.95	0.46	1.23
		16	3.36	5.98	0.59	1.81
	Second Input	4	0.71	1.48	0.95	3.02
		8	1.6	2.42	1.06	3.47
		16	1.69	2.16	1.2	4.06
raytracer	First Input	4	12.63	12.56	0.89	2.97
		8	4.18	4.35	0.98	3.32
		16	0.9	0.92	1.12	3.85
	Second Input	4	9.29	9.35	6.05	23.44
		8	6.13	6.07	6.4	24.65

Continued on next page

Table V – CONTINUED FROM PREVIOUS PAGE

Benchmark	Setup		Elapsed Overhead %	CPU Overhead %	Average Elapse Time	Average CPU Time
	Input Size	#of Th.				
		16	7.31	7.16	6.67	25.64
montecarlo	First Input	4	6.55	14.43	3.21	6.43
		8	17.59	11.36	3.3	6.65
		16	8.66	11.29	3.23	6.7
	Second Input	4	14.77	10.98	15.56	30.53
		8	13	18.96	21.23	38.94
		16	20.95	13.08	23.57	42.05
hedc	First Input	4	4.89	14.85	1.17	1.48
		8	1.34	15.5	1.21	1.58
		16	9.32	17.53	1.47	1.8
fop	First Input	-	4.62	4.07	1.9	4.6
	Second Input	-	5.01	3.99	2.38	6.21

For most chosen benchmarks, we evaluated the *DDS* using STM by injecting *harmful statements* in the source code of the benchmarks. Then, we measured the introduced overhead with two different input sizes and three different thread numbers, namely 4, 16, and 32 threads, as shown in Figure 20. DeuceSTM is implemented to work on Java version 7. If we have a Java file that would not compile with Java 7, we can not use DeuceSTM with it.

The average elapsed time overheads for benchmarks running with STM was 5.83%, and the CPU time overhead average was 14.4% with respect to the runtime of the benchmarks without *DDS* agents. When comparing the runtime of the benchmarks running with STM and the runtime of agents not using STM, we saw increases of only 7.4% and 1.4% for the CPU time and elapsed time, respectively, of the agents with STM with respect to those without STM. Additionally, we measured the memory consumption overhead percentage with respect to the runtime without *DDS* agents. The median memory consumption overheads for benchmarks running with STM was 22% of the original memory use.

The "barrierbench" and "moldyn" benchmarks reported negative percentages of overheads. The main reason behind these numbers is that these benchmarks have sleep and barrier statements that introduce a degree of randomness in the runtimes.

In general, as we increased the thread number, the overheads decreased. These benchmarks had a high degree of contention managed by the STM with the smaller number of threads. As we increased the input size of these benchmarks, the contention rate decreased, which resulted in faster execution and lower overhead percentages.

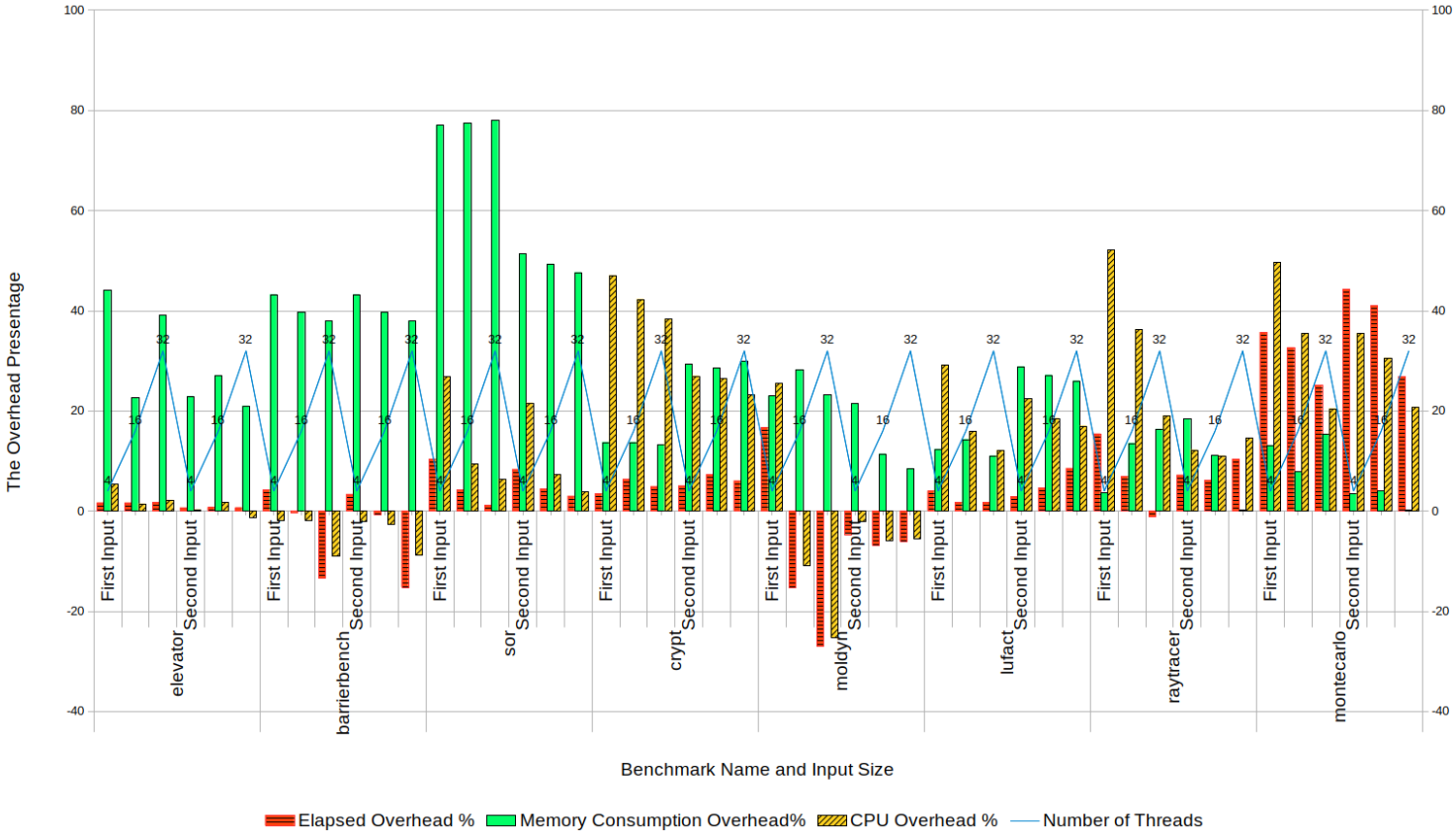

Figure 20: DDS average percentages of overheads with varying number of threads along with two input size sets using STM.

6.3.2.3 Comparison with Other Deadlock Detectors

It was difficult to conduct a one-to-one comparison of *DDS* with prior work because of the absence of tools that resolve deadlock in Java programs to the best of our knowledge DDS is the first runtime tool that detects and resolves deadlocks for Java programs using runtime monitoring. Nevertheless, we found working tools for Java programs that can detect deadlocks and were able to compare DDS with previous works.

We compared *DDS* with one static deadlock detector and one dynamic detector known as "ThreadSafe" (74) and "JCarder" (75). By design, ThreadSafe reports false positives when conducting a static analysis. The aim of ThreadSafe is to detect any possible deadlocks. Because JCarder is a dynamic tool, it only identifies real deadlocks. The two detectors work on Java byte codes. ThreadSafe parses compiled (.class) files to enable an interprocedural dataflow analysis and to identify the control flow in each Java method. For each instruction, ThreadSafe computes a lock set, which is later used to identify problematic or defective code. In the last step, ThreadSafe uses a so-called *Inconsistent Synchronization* checker to report potential deadlocks. JCarder dynamically instruments Java byte code to record information on lock acquisition. Using runtime information, JCarder searches for cycles in the locks acquisition graph.

We executed all our benchmarks in this experiment with two threads; the exception was the "fop" benchmark, which initially started with one thread and then spawned to four. Based on Table VI, we can see that ThreadSafe spent a considerable amount of time, 8844.12% overhead, which is $\approx$ 3 minutes of running compared to $\approx$ 2 seconds total time of the original

running program, analyzing fop's source code. This occurred because ThreadSafe issued four error counts within the source code and used timeout for these errors, causing a timing issue for this benchmark. This result also raises a question about the approach's scalability. On the positive side, ThreadSafe can be used for not only detecting deadlocks but also identifying access violations that can cause data races.

TABLE VI: COMPARISON OF DIFFERENT DEADLOCK DETECTOR TOOLS. THE RECORDED MEASUREMENT INCLUDES THE AVERAGE TIME OVERHEAD MEASUREMENT FOR BOTH CPU TIME AND ELAPSED TIME OVERHEAD PERCENTAGE.

Benchmark	DDS		JCarder		ThreadSafe	
	CPU%	elapsed%	CPU%	elapsed%	CPU%	elapsed%
elevator	3.33	0.29	37.63	1.56	86.02	-99.95
barrierbench	18.18	9.84	26.23	11.29	81.25	72.77
sor	21.49	18.78	45.49	47.86	74.5	69.5
crypt	2.92	13.04	14.19	60.26	46.91	71.56
moldyn	8.12	11.21	16.44	23.39	46.22	53.2
lufact	4	8.89	15.04	28.07	81.75	80.84
raytracer	16.81	15.69	22.46	24.56	64.74	60.55
montecarlo	6.75	13.97	12.65	22	45.69	46.08
hedc	17.65	10	52.94	56	805.88	425
fop	2.38	2.94	2.86	5.88	3711.9	8844.12

The comparison shows that the performance of the *DDS* far exceeds that of JCarder and ThreadSafe when it comes to CPU time and elapsed time, with the exception of the "elevator" benchmark's elapsed time relative to the ThreadSafe elapsed time. The fact that *DDS* is a runtime tool monitoring the running program is the major reason for its elapsed time being

longer compared to that of ThreadSafe. The waiting in the elevator is stimulated by the source code of the elevator through the use of Java sleep statements. These statements led to higher elapsed time for *DDS*. We compared the recorded times in seconds to the average times of the uncontrolled version of the benchmarks and concluded that the maximum elapsed time overhead obtained using *DDS* was 18% for the "sor" benchmark, while JCarder and ThreadSafe had the highest elapsed time overhead at 47% and 69%, respectively.

It is then possible for us to conclude that, in general, *DDS* is more efficient than JCarder and ThreadSafe in terms of added overhead. Finding a resolution for any detected deadlock at runtime is the purpose of our methodology. JCarder and ThreadSafe do not resolve deadlocks.

6.3.3 Communication Cost for Runtime Monitoring

We mainly focused on the time taken by the two callbacks, monitor_contended_enter and monitor_contended_entered, which were used to detect and resolve deadlocks. We analyzed the effects on the performance when the program was run under supervision by recording the average CPU time utilized by the two callbacks mentioned previously for over 100 runs. Table VII showed that the time for calling these two callbacks was negligible compared to the actual CPU time for the dining philosophers. Calling monitor_contended_enter almost always yielded a higher CPU time compared to calling monitor_contended_entered; this was usual when comparing monitor_contended_enter and monitor_contended_entered in Algorithm 1. The main cycle search was in monitor_contended_enter, and monitor_contended_entered was only used for edge and owner deletion. Further, if a deadlock was detected, then call wait was performed through monitor_contended_enter by calling the appropriate JNI functions.

TABLE VII: COMMUNICATION COST (CPU TIME) WHEN USING "MONITOR_CONTENDED_ENTER" COLUMN TITLED "ENTER" AND "MONITOR_CONTENDED_ENTERED" COLUMN TITLED "ENTERED" MEASURED IN MICROSECONDS FOR DIFFERENT NUMBERS OF THREADS BASED ON OVER 100 RUNS.

	10 Threads		40 Threads		100 Threads	
	enter	entered	enter	entered	enter	entered
Average	514.91	191.57	1060.58	330.32	1826.72	397.43
MIN	47	5	56	6	61	7
MAX	2043	907	6602	5337	13437	2563
Median	316	137	447	163	187	321

As we increased the number of the threads, resulting in larger graphs to search in, the time growth increased linearly. The MIN row in Table VII represents the cases of empty graphs before the addition of any vertex or edge. The MAX row shows the case in which almost all the thread vertices and edges have already been added.

6.3.4 Limitation

With the current implementation, *DDS* does not always resolve the reentrant locks deadlock if the deadlock involves more than two such locks. This limitation is due to difficulties in retrieving a lock object using *JVMTI*. However, we are investigating ways to overcome this limitation in future works. It is worth mentioning that according to Lobo and Castor (76) 92% of the deadlocks involve only two resources between two threads.

An additional limitation is that we did not consider certain other locking mechanisms of Java such as semaphores. We will consider semaphores in future extensions to this work.

DeuceSTM is only working with Java program compiled using Java version 7 and bellow. We need to upgrade DeuceSTM to handle new features added after Java version 7 release.

CHAPTER 7

APPLICATION OF DDS

The current methodology discussed in the preceding chapters has been applied to Java. The aim of this chapter is to explain in detail how this methodology can be applied to any other language. We begin this chapter with an analysis of the source code and then move to an examination of runtime monitoring. In this chapter, we try to identify whether any tool exists that can instrument the source code. Previously, we used existing tools (e.g., JVMTI) to monitor Java. Our focus now is on the application of existing tools to the other programming languages such as C++. We endeavor to find a way to preempt a resource (lock) from the victim thread (so for Java, we have used JNI) to issue a "wait" and "notify" functions for the intrinsic locks and a "lock" and "unlock" functions for reentrant locks. This chapter highlights what we need to do to accomplish the same process for any language other than Java (e.g., C++). Finally, we write about how we are planning to apply DDS methodology for distributed systems that use a language supporting the locking mechanism for concurrency.

7.1 Application of DDS in a Multicore Environment to Any Other Language

In the first section we examine the process of applying DDS to any language other than Java. In a multicore system, DDS has great potential to help resolve deadlocks at runtime. The application of DDS using a locking mechanism can be categorized into the following steps.

In the first step, the DDS is required to build a preprocessing tree. Such a tree represents the synchronization point with application of a static analysis tool on the source code. It indicates that DDS helps in checking the program code before testing.

In the second step, we modify the code to protect it from the harmful statements (i.e., protect the program state from the result of victimizing a harmful statement) using the STM methodology. The second step is carried out during the preprocessing stage. However, in case the language does not have any tool to monitor the running program and preempt a resource during runtime, we need to instrument the source code, as described in the third step below.

In the third step, we instrument the source code for tracking the lock request, acquisition, and release during the runtime. Such instrumentation is the most important part of the development of the optimized lock graph.

In the fourth step, in case of a cycle, we need to victimize (i.e., unlock) a lock from a thread that participated in the deadlock cycle.

One of the key components in DDS is the lock graph, which we mentioned in the third step. A lock graph is simply a directed graph, as we described earlier—Chapter 5, Section 5.1.2.1. It captures locks and threads data. In this scenario, it can be used in numerous languages to point to lock information and thread creations. Directed edges symbolize the sequence or pattern for lock acquisition. The source code is required for building the graph at the runtime of every event of interest. Lock graphs are important in deadlock detection because they help in detecting cycles. While checking cycles in lock graphs, deadlocks are checked in two or more threads because every resource request is at risk of being affected by a deadlock. Different

languages can use class-level locks that apply atomic methods using synchronization—intrinsic locks. In addition, reentrant locks can be used as special classes in different languages.

When applying DDS to any other language, the main challenge, which we can phase, relates to the preempting of a resource lock in a multicore system. To address this issue, we need to control and access the thread scheduler and attain information about the execution (e.g., mutex lock and unlock). Preemption of the thread's resource resolves the detected deadlock. The first approach would ensure that in case of a thread being made to wait when it requests new resources, any other resource that was previously held is implicitly released. When resources are requested but not present, the system checks to observe the resources that are held by threads are in a blocked state as they wait for other resources. Upon identification of a cycle in the lock graph, a resource needs to be preempted from a victim thread. Then, the victimized thread is added to the front of the waiting list for the resources that have been taken away.

7.1.1 Applying DDS to C++

As described above, we have four main steps. In this section, we illustrate what tools could be used and what needs to be done for each step.

In the first step, we use *CodeSurfer* (77) to build a preprocessing tree of the DDS. *CodeSurfer* is a static analysis tool that provides a wide range of functionalities (e.g., Control Flow Graph (CFG) and AST) through its Application Programming Interface (API). Using *CodeSurfer*, we can implement our preprocessing algorithm to build the tree. The tool proves useful because it can explore all the execution paths and threads interleaving within the program. It is very helpful in determining the state's reachability.

In the second step, we use the simplified version of *Transactional Memory Technical Specification (TMTS)* (78) to modify the code and protect it from the harmful statements' effect. Based on the harmful statements list we obtained in the previous step, we guard their locations with "transaction_safe" block. Using the proposed keyword, the compiler is able to create and instrument clones for functions reachable to the guard block during the runtime. Therefore, we have a history of memories' contents in case we need to rollback a transaction. In the clones, every memory access (i.e., read and write) is replaced by a function call to a STM implementation.

In the third step, we monitor and control the source code using *Steroids* library (79). *Steroids* is a dynamic analysis library that allows DDS to control the thread scheduler and track the lock request, acquisition, and release during the runtime. This support is granted through the instrumentation of the bitcode for every call to a pthread function—pthread is a c library for thread execution, which includes a set of programming types and function calls. *Steroids* is compiled as a shared static library and provides a C++ interface. In this step, we build the lock graph as we monitor the locks' request and release.

In the fourth step, in the event we have a cycle in our lock graph, we need to victimize a lock from a thread that participated in the detected deadlock cycle. This step is also performed using *Steroids* by calling unlock on the victimized mutex (i.e., lock) during runtime.

7.2 Application of the DDS to a Distributed System

Distributed systems are subject to deadlocks. However, the detection of generalized deadlocks is fairly challenging in a distributed system. A distributed system comprises P processes

(called nodes). All these processes are combined through communication channels. A distributed system does not contain a global shared memory. The nodes communicate with each other by sending messages directly over their channels. These passages are consistently delivered with limited but random delays. Distributed systems are mainly presumed to be fault-free.

The distributed system functions by allowing the node to make requests and blocks. During this process, the node goes into the idle state from the active state. The system continues to reference the blocking or granting of requests from the node by indicating whether they are true or false. Hence, upon having enough true requests granted, the node again assumes the active state from the idle state. Kshemkalyani and Singhal (80) explained how the node unblocks: it removes the outstanding requirements it sent but that are not yet decided or that are decided but not cast off in the assessment of unblocking the condition of requested nodes. Another attribute of the distributed system is related to the type of nodes operating therein. These can be process or resource managers; process nodes disperse a request to another process and get blocked after waiting for a reply. Meanwhile, the resource manager allows the process to access the resource needed to grant the request. However, during a request—that is, communication in progress—within the system, the resource manager is unable to reply to the process nodes, or the process nodes are unable to send requests to the resource manager using a single-request model. In the single-request model, the simplest request model, each process is restricted to requesting only one resource at a time. Different request models have been discussed in the literature (80; 81; 82) (e.g., single-request, AND-request, OR-request, and generalized request). In the AND-request model, each process is allowed to request any number of resources. The

process is active (i.e., unblock) when all the requested resources are granted. The third model is the OR-request model, which allows each process to request any number of resources. The process is active (i.e., unblock) when any one of the requested resources are granted. In the generalized request model, also known as the P-out-of-Q request model, each process is allowed to request any number of resources Q. The process is active (i.e., unblock) when the P number of the requested resources Q is granted. Using the generalized request model, we can represent the AND-request model by having $P = Q$. Additionally, we can represent an OR-request by having $P = 1$. In the single-request and AND-request models, the cycle that is within the WFG is the requirement for a deadlock. For the OR-request model, a knot that is in the WFG is the requirement for the deadlock.

In the following sections, we highlight some of the strategies used for handling deadlocks in distributed systems. In addition, we illustrate whether we can use our methodology in distributed system environments.

7.2.1 Deadlock-Handling Strategies in Distributed Systems

Deadlocks can be handled using three strategies: deadlock prevention, deadlock avoidance, and deadlock detection, as we explained in Chapter 3. In a distributed system, it is rather complex to implement different deadlock-handling strategies given that no one in the system is aware of the condition of other processes in the system. As a result, this hidden information leads to certain irregular delays.

Deadlock prevention is the first and one of the most important strategies for handling deadlock, which is achieved through attaining a process that needs the required resources prior

to the implementation (83). In the traditional methods of deadlock prevention, a request message is sent to sites holding resources through the current requester process. However, this method was criticized for inefficiencies that caused it to decrease the concurrency of the system. Moreover, during the phase of acquiring the resource, a deadlock is likely to take place in different processes. The problem may be solved by pushing processes for the required resources (84).

According to Pyla and Varadarajan (55), a deadlock avoidance strategy entails an approach wherein a resource is allowed to access a process in a situation when the subsequent system condition is safe — when a system can access all resources without entering a deadlock state. However, Zhonghii et al. (85) argued that a deadlock avoidance strategy can be ineffective in a distributed system. They explained that to avoid deadlock, a significant storage dimension is required that has widespread communication facilities for maintaining the information on the global state of the system. In addition, Ali et al. (86) determined that the verification process for a secure global state is required to be reciprocally exclusive so that a number of sites are enabled to concurrently undertake the process of verification. As a result, the process of verification rigorously restrains the system output and the concurrency. Moreover, it is logically more costly to verify for a secure state for a considerably large number of processes and resources.

Deadlock detection, the third deadlock-handling strategy, deals with detecting different deadlocks taking place in the system (87). Deadlock detection requires supervision of the process resource status interaction to detect the presence of any cyclical wait state. According

to Banerjee and Chrysanthis (88), deadlock detection offers two main supportive conditions: when a cycle is detected in the Wait for Graph (WFG) and the state endures while waiting for it to be detected and broken and when the detection of the cycle is able to continue concurrently with regular activities of the system. Hence, running the detection algorithm concurrently with the running system has no negative impact on the throughput of the system, meaning that the literature has been more focused on deadlock detection.

7.2.2 Applying DDS to Distributed Systems

Some of the identified causes behind the deadlock in the distributed system include the absence of a global time concept, algorithms being based on ad hoc methods, and incorrect assumptions about the stability of deadlocks. Because of the absence of the concept of global time, most of the algorithms do not consider the causality relations between the events occurring at different sites, resulting in the enclosure of unpredictable states. Kshemkalyani and Singhal (89) concluded that a distributed deadlock needs to be defined in a more direct manner to understand the causal relationship among the events and processes/sites. According to them, the possibility of the global cycle must be considered in order to understand the existence of different segments at nonoverlapping time intervals.

Kshemkalyani and Singhal (90) proposed one-phase algorithm premised on the two simultaneous sweeps in the distributed system's messages to trace generalized deadlocks efficiently. The algorithm takes "a snapshot of a distributed WFG" into account in the outward sweep whereas it reduces the recorded distributed WFG in the inward sweep to evaluate the presence of a deadlock. One of the significant features of this algorithm is its ability to overlap (in time)

the two sweeps in the process. The researchers drew inferences from the two past investigations into the two-phases algorithm. Both research studies were based on the two-phases algorithm. Bracha and Toeug (91) proposed that the algorithm recorded a snapshot of distributed WFG in the first phase before simulating the granting of requests in the second phase to ascertain the presence of generalized deadlocks. Both phases are nested within each other and subsequently terminated (i.e., the first phase was followed by the second one). In contrast, in the second algorithm proposed by Wang et al. (92), the first phase recorded a snapshot of a distributed WFG, whereas the second phase reduced the static WFG recorded in the first phase for detection of deadlock. Under this proposed approach, the first phase was terminated using a termination detection technique prior to initiating the second phase.

Based on the previous discussion of distributed systems, we can see that DDS methodology can be deployed in a distributed system. However, this would occur under a major assumption-namely, the deadlock's stay persistence —cycle in the graph means deadlock. With the current methodology, we can detect the deadlock caused by the single-request model only. Currently, DDS can be applied to every single node in the distributed system to resolve local deadlock. To expand DDS, we would think of the preprocessing step as the manager to build a centralized list of all the requests. The following is how we are planning to apply DDS to distributed systems.

In the first step, *CodeSurfer* could be used for the preprocessing step to build the tree of the DDS. We can implement our preprocessing algorithm to build the tree. In addition, in this step we can have a log of all the requests to help in managing the communication later. This

step is helpful because each process does not know anything about the other processes in the system. This list will make it feasible to monitor the system activity from a centralized point.

In the second step, we suggest TM^2C (93) as software transactional memory for the distributed systems. Based on the harmful statements generated from the first step, we could guard their locations by instrumenting the source code. The read-and-write memory access of the block of the code, which was run using TM^2C, would be monitored for any write or read conflict. If the system detects a conflict, it will perform a rollback and a reexecute for that block of code.

In the third step, we would monitor and control the source code by instrumenting the resource request and checking whether the current process is blocked because the resource is currently unavailable. If so, we would need to add an edge into our lock graph (called WFG in the distributed systems literature). The main challenge in this step is the communication delay. In distributed systems, there are lots of factors to be taken into account (e.g., network congestion).

In the fourth step, in case we have a cycle in our lock graph, we would need to victimize a resource from a process and then return it after the other waiting process releases it.

Hence, based on the analysis of existing tools that can be used for the application of DDS on other languages and for the application of DDS in a distributed environment, the current methodology can be applied to C++. By following the four steps presented in Section 7.1.1, it would be easier for us to apply the current methodology by following the stages. Conversely, applying the DDS in the distributed system for dealing with the issue of deadlocks is not

completely feasible at this stage. Although we propose applying the DDS in the distributed system, it would only be in the systems that use the single-request model. DDS can be applied on a single node within the distributed system to solve local deadlocks. The DDS methodology is based on finding a cycle in the lock graph; thus DDS cannot be applied to detect and resolve the OR-request or generalized request deadlocks because they are based on finding a knot in the graph. In distributed systems, deadlock handling is challenging because no site possesses what can be referred to as precise information or knowledge regarding the system's state. The lack of precise information occurs due to the inter site communication having a finite and unpredictable delay. This challenge of the process's knowledge of the system needs to be addressed in detail before extending DDS to distributed systems.

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

This study evaluated the causes and effects of deadlocks within the Java program and devised a method of resolving deadlocks. The deadlock problem has been a significant challenge affecting program reliability because of the multithreaded software, whose need has increased over the past decade, which presents complex bugs because of its nondeterministic behavior. Deadlock Detector and Solver (DDS), a runtime methodology that has a preprocessing stage to assist the runtime in resolving the deadlock when there is a harmful statement, has been proposed as an effective solution to the problem. In addition, Software Transactional Memory (STM) is used to guard against any harmful statements and ensure that any written update is performed with atomicity one thread at a time. STM incurs median overhead of 22% for memory and 4% for elapsed time with respect to the runtime without DDS.

DDS is different than other existing approaches to resolving deadlocks in the way it rectifies and fixes detected deadlocks. The current methodologies for resolving deadlocks unravel the issues by suspending the involved threads, rolling back to a safe point in the execution history, and reexecuting from that specific point. However, the DDS approach is based on resource preemption from a thread, which has not been involved in modifying a shared object. We guarantee the program state of our anticipation approach through a preprocessing phase conducted

offline, which detects program positions in which locks can be preempted without disturbing the stability—constancy— of shared objects.

The experiments performed during the study confirm that the DDS's overhead grows linearly with an increase in the number of threads. The findings show that our adopted approach is scalable and does not incur noticeable overhead when the number of synchronized points increases or the size of the program (line of code) increases. The study also succeeded in determining the runtime overhead of the supervisor controller (DDS) relative to the runtime of the controlled program to be approximately 5% on average. This indicated that the DDS quickly detects and resolves the deadlock during runtime. The results also indicated that the DDS could offer a fully automatic approach without the need for manual adjustments. In addition, DDS fixes the deadlocks without undermining the concurrency extent by prompting any part of the controlled program to perform deterministically.

Furthermore, according to this study, deadlocks observed in Java (during runtime) programs can be resolved by the DDS. The tests were undertaken with two levels of getting deadlocked programs unlocked, thus proving the effectiveness of the DDS. The DDS methodology was also tested on the hypothetical cases, although scarce and infrequent in the real world, and it proved to be effective. Conversely, where deadlocks are missing, no effects are manifested in the Java program's flow. Thus, the DDS agent can effectively perform its role without causing any substantial modification to the overhead. Moreover, it is possible to extent this methodology to other languages utilizing locking of objects to achieve their synchronization.

8.2 Future Work

Our future work encompasses performing deeper analyses to estimate the rates of deadlock and harmful statements by measuring performance metrics such as duration of deadlock and mean waiting times of blocked processes. Additionally, in our future work, we suggest extending the methodology to:

1. Handle semaphores, which are synchronization objects that regulate access by several processes to a shared resource in a multithreaded programming environment.
2. Resolve deadlocks caused by reentrant locks that encompass more than two reentrant locks because the current method does not always succeed in resolving reentrant lock deadlocks that comprise more than two locks.
3. Find a solution for STM because it does not handle I/O operations; thus, if we roll back and reexecute, it will exclude I/O operations. Additionally, STM does not permit the manipulation of a nontransactional mutable state or I/O operations. In our future work, we should find an alternative to STM that permits the handling of I/O operations. Consequently, this will enable rollback and reexecution without excluding I/O operations.
4. Apply our DDS methodology to other programming languages. Because our study has focused on using DDS to solve deadlocks within Java programs, we should consider researching the use of DDS to resolve deadlocks in other programming languages, such as C++ or C#, in our future work.

We also need more benchmarks to expose our methodology to real-world deadlock systems with the use of STM. We need to find a benchmark that has enough harmful statements instead of rarely having one because we injected a harmful statement in the benchmark to test our methodology's overhead.

CITED LITERATURE

1. Xiong, W., Park, S., Zhang, J., Zhou, Y., and Ma, Z.: Ad hoc synchronization considered harmful. In OSDI, volume 10, pages 163–176, 2010.

2. Engelmann, C. and Naughton, T.: Toward a performance/resilience tool for hardware/-software co-design of high-performance computing systems. In Parallel Processing (ICPP), 2013 42nd International Conference on, pages 960–969. IEEE, 2013.

3. Sharifirad, S. and Haghighi, H.: A formal framework for specifying concurrent systems. International Journal of Computer Applications, 68(3), 2013.

4. Bensalem, S. and Havelund, K.: Dynamic deadlock analysis of multi-threaded programs. In Hardware and Software, Verification and Testing, eds. S. Ur, E. Bin, and Y. Wolf-sthal, pages 208–223, Berlin, Heidelberg, 2006. Springer, Springer Berlin Heidelberg.

5. Java deadlock, livelock and lock starvation examples.

6. Li, T., Ellis, C. S., Lebeck, A. R., and Sorin, D. J.: Pulse: A dynamic deadlock detection mechanism using speculative execution. In USENIX Annual Technical Conference, General Track, volume 44, 2005.

7. Eslamimehr, M. and Palsberg, J.: Sherlock: Scalable deadlock detection for concurrent programs. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 353–365, New York, NY, USA, 2014. ACM.

8. Joshi, P., Park, C.-S., Sen, K., and Naik, M.: A randomized dynamic program analysis technique for detecting real deadlocks. In ACM Sigplan Notices, volume 44, pages 110–120. ACM, 2009.

9. Yue, H. and Xing, K.: Robust supervisory control for avoiding deadlocks in automated manufacturing systems with one specified unreliable resource. Transactions of the Institute of Measurement and Control, 36(4):435–444, 2014.

10. Li, C., Huang, L., Chen, L., Luo, W., and Li, X.: Pbddr: Probe-based deadlock detection and recovery strategy for component-based systems. In Software Engineering

Conference (APSEC), 2012 19th Asia-Pacific, volume 1, pages 790–795. IEEE, 2012.

11. Cai, Y. and Cao, L.: Fixing deadlocks via lock pre-acquisitions. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pages 1109–1120, New York, NY, USA, 2016. ACM.

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

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

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

15. Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y.: Cilk: An efficient multithreaded runtime system. SIGPLAN Not., 30(8):207–216, Aug 1995.

16. Frigo, M., Leiserson, C. E., and Randall, K. H.: The implementation of the cilk-5 multithreaded language. In Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

17. Zhou, J., Silvestro, S., Liu, H., Cai, Y., and Liu, T.: Undead: detecting and preventing deadlocks in production software. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 729–740. IEEE, October 2017.

18. Tan, L., Zhou, Y., and Padioleau, Y.: acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In Software Engineering (ICSE), 2011 33rd International Conference on, pages 11–20. IEEE, 2011.

19. Joshi, P., Naik, M., Sen, K., and Gay, D.: An effective dynamic analysis for detecting generalized deadlocks. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 327–336, New York, NY, USA, 2010. ACM.

20. Marino, D., Hammer, C., Dolby, J., Vaziri, M., Tip, F., and Vitek, J.: Detecting deadlock in programs with data-centric synchronization. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 322–331, Piscataway, NJ, USA, 2013. IEEE Press.

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

22. Shousha, M., Briand, L., and Labiche, Y.: A uml/marte model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems. IEEE Transactions on Software Engineering, 38(2):354–374, 2012.

23. Liao, H., Zhou, H., and Lafortune, S.: Simulation analysis of multithreaded programs under deadlock-avoidance control. In Simulation Conference (WSC), Proceedings of the 2011 Winter, pages 703–715. IEEE, 2011.

24. Kini, D.: Deadlock code generator, gas station deadlock program and runtime deadlock detector and solver v2.1. Master project, Dept. of Computer Science, University of Illinois at Chicago, 2013. Master's project report.

25. Cai, Y. and Chan, W.: Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. IEEE Transactions on Software Engineering, 40(3):266–281, 2014.

26. Friesen, J. and Pal, S.: Java Threads and the Concurrency Utilities. Springer, 2015.

27. Goetz, B., Peierls, T., Lea, D., Bloch, J., Bowbeer, J., and Holmes, D.: Java concurrency in practice. Pearson Education, 2006.

28. Horstmann, C. S.: Core Java for the Impatient. Pearson Education, 2015.

29. Gallardo, R., Hommel, S., Kannan, S., Gordon, J., and Zakhour, S. B.: The Java Tutorial: A Short Course on the Basics. Addison-Wesley Professional, 2014.

30. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., and Holmes, D.: Java Concurrency in Practice: JAVA CONCURRENCY PRACT _p1. Pearson Education, 2006.

31. Schafer, M., Sridharan, M., Dolby, J., and Tip, F.: Refactoring java programs for flexible locking. In 2011 33rd International Conference on Software Engineering (ICSE), pages 71–80. IEEE, May 2011.

32. Reentrantlock (java platform se 7), Oct 2018.

33. Shavit, N. and Touitou, D.: Software transactional memory. Distributed Computing, 10(2):99–116, 1997.

34. Korland, G., Shavit, N., and Felber, P.: Noninvasive concurrency with java stm. In Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG), 2010.

35. Ziarek, L. and Jagannathan, S.: Memoizing multi-threaded transactions. In Workshop on Declarative Aspects of Multicore Programming, 2008.

36. Cachopo, J. and Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science of Computer Programming, 63(2):172–185, 2006.

37. Hindman, B. and Grossman, D.: Atomicity via source-to-source translation. In Proceedings of the 2006 Workshop on Memory System Performance and Correctness, MSPC '06, pages 82–91, New York, NY, USA, 2006. ACM.

38. Williams, A., Thies, W., and Ernst, M. D.: Static deadlock detection for java libraries. In ECOOP, volume 3586, pages 602–629. Springer, 2005.

39. Engler, D. and Ashcraft, K.: Racerx: Effective, static detection of race conditions and deadlocks. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

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

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

42. Laneve, C.: A lightweight deadlock analysis for programs with threads and reentrant locks. Science of Computer Programming, 181:64–81, 2019.

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

44. Sen, K.: Concolic testing. In Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM.

45. Kalhauge, C. G. and Palsberg, J.: Sound deadlock prediction. Proc. ACM Program. Lang., 2(OOPSLA):146:1–146:29, October 2018.

46. Cai, Y., Jia, C., Wu, S., Zhai, K., and Chan, W. K.: Asn: A dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs. IEEE Transactions on Parallel and Distributed Systems, 26(1):13–23, January 2015.

47. Cai, Y., Wu, S., and Chan, W. K.: Conlock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 491–502, New York, NY, USA, 2014. ACM.

48. Cai, Y. and Lu, Q.: Dynamic testing for deadlocks via constraints. IEEE Transactions on Software Engineering, 42(9):825–842, September 2016.

49. Arnold, M., Vechev, M., and Yahav, E.: Qvm: An efficient runtime for detecting defects in deployed systems. ACM Trans. Softw. Eng. Methodol., 21(1):2:1–2:35, Dec 2011.

50. Huang, J., Zhang, C., and Dolby, J.: Clap: Recording local executions to reproduce concurrency failures. SIGPLAN Not., 48(6):141–152, June 2013.

51. Havelund, K.: Using Runtime Analysis to Guide Model Checking of Java Programs, volume 1885, pages 245–264. Berlin, Heidelberg, Springer Berlin Heidelberg, 2000.

52. Agarwal, R., Wang, L., and Stoller, S. D.: Detecting potential deadlocks with static analysis and run-time monitoring. In Haifa Verification Conference, volume 3875, pages 191–207. Springer, Springer, Berlin, Heidelberg, 2006.

53. Cogumbreiro, T., Hu, R., Martins, F., and Yoshida, N.: Dynamic deadlock verification for general barrier synchronisation. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pages 150–160, New York, NY, USA, 2015. ACM.

54. Li, J., Liu, X., Jiang, L., Liu, B., Yang, Z., and Hu, X.: An intelligent deadlock locating scheme for multithreaded programs. In Proceedings of the 2019 3rd International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence, ISMSI 2019, pages 14–18, New York, NY, USA, 2019. ACM.

55. Pyla, H. K. and Varadarajan, S.: Avoiding deadlock avoidance. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pages 75–86, New York, NY, USA, 2010. ACM.

56. Zhang, W., de Kruijf, M., Li, A., Lu, S., and Sankaralingam, K.: Conair: Featherweight concurrency bug recovery via single-threaded idempotent execution. SIGPLAN Not., 48(4):113–126, Mar 2013.

57. Bättig, M.: Efficient Synchronized-by-Default Concurrency. Doctoral dissertation, ETH Zurich, 2019.

58. Jula, H., Tzn, P., and Candea, G.: Communix: A framework for collaborative deadlock immunity. In 2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN), pages 181–188, June 2011.

59. Jula, H., Andrica, S., and Candea, G.: Efficiency Optimizations for Implementations of Deadlock Immunity, pages 78–93. Berlin, Heidelberg, Springer Berlin Heidelberg, 2012.

60. Voss, C., Cogumbreiro, T., and Sarkar, V.: Transitive joins: a sound and efficient online deadlock-avoidance policy. In PPoPP, pages 378–390, 2019.

61. Gerakios, P., Papaspyrou, N., Sagonas, K., and Vekris, P.: Dynamic deadlock avoidance in systems code using statically inferred effects. In Proceedings of the 6th Workshop on Programming Languages and Operating Systems, PLOS '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

62. Kahsai, T., Rümmer, P., and Schäf, M.: Jayhorn: A java model checker. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, eds. D. Beyer, M. Huisman, F. Kordon, and B. Steffen, pages 214–218, Cham, 2019. Springer, Springer International Publishing.

63. Lokuciejewski, P., Cordes, D., Falk, H., and Marwedel, P.: A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In

2009 International Symposium on Code Generation and Optimization, pages 136–146. IEEE, March 2009.

64. Madsen, M., Tip, F., and Lhoták, O.: Static analysis of event-driven node.js javascript applications. SIGPLAN Not., 50(10):505–519, October 2015.

65. Spalazzi, L., Spegni, F., Liva, G., and Pinzger, M.: Towards model checking security of real time java software. In 2018 International Conference on High Performance Computing Simulation (HPCS), pages 642–649, July 2018.

66. Afek, Y., Korland, G., and Zilberstein, A.: Lowering stm overhead with static analysis. In International Workshop on Languages and Compilers for Parallel Computing, pages 31–45. Springer, 2010.

67. Carvalho, F. M. and Cachopo, J.: Optimizing memory transactions for large-scale programs. Journal of Parallel and Distributed Computing, 89:13–24, 2016.

68. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., and Seinturier, L.: Spoon: A library for implementing analyses and transformations of java source code. Software: Practice and Experience, 46:1155–1179, 2015.

69. Jvm-Profiling-Tools: jvm-profiling-tools/async-profiler, February 2019.

70. Java grande benchmark suite — epcc at the university of edinburgh.

71. Areas of research in computer science.

72. Apache fop.

73. Naik, M., Park, C.-S., Sen, K., and Gay, D.: Effective static deadlock detection. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 386–396, Washington, DC, USA, 2009. IEEE Computer Society.

74. Atkey, R. and Sannella, D.: Threadsafe: Static analysis for java concurrency. Electronic Communications of the EASST, 72, 2015.

75. Jcarder – dynamic deadlock finder for java.

76. Lobo, R. and Castor, F.: Deadlocks as runtime exceptions. In Brazilian Symposium on Programming Languages, eds. A. Pardo and S. D. Swierstra, pages 96–111, Cham, 2015. Springer, Springer International Publishing.

77. Anderson, P. and Teitelbaum, T.: Software inspection using codesurfer. In Proceedings of the First Workshop on Inspection in Software Engineering. Citeseer, 2001.

78. Zardoshti, P., Zhou, T., Balaji, P., Scott, M. L., and Spear, M.: Simplifying transactional memory support in c++. ACM Trans. Archit. Code Optim., 16(3):25:1–25:24, July 2019.

79. Rodríguez, C.: Steroids, 2016.

80. Kshemkalyani, A. D. and Singhal, M.: On characterization and correctness of distributed deadlock detection. Journal of Parallel and Distributed Computing, 22(1):44–59, 1994.

81. Brzezinski, J., Helary, J.-M., Raynal, M., and Singhal, M.: Deadlock models and a general algorithm for distributed deadlock detection. Journal of parallel and distributed computing, 31(2):112–125, 1995.

82. Chen, S., Deng, Y., Attie, P., and Sun, W.: Optimal deadlock detection in distributed systems based on locally constructed wait-for graphs. In Proceedings of 16th International Conference on Distributed Computing Systems, pages 613–619. IEEE, 1996.

83. Bandyopadhyay, A. K.: Weakest precondition based verification tool that models spatial ordering. SIGSOFT Softw. Eng. Notes, 33(2):2:1–2:5, March 2008.

84. Silberschatz, A., Galvin, P. B., and Gagne, G.: Operating system principles. John Wiley & Sons, 2006.

85. Zhonghu, Y., Wendong, X., and Xinhe, X.: Deadlock avoidance in manufacturing systems of parallel mutual exclusion. In Proceedings of 1994 American Control Conference - ACC '94, volume 1, pages 722–723 vol.1, June 1994.

86. Ali, H., El Dnaf, T., and Salah, M.: A proposed algorithm for solving deadlock detection in distributed database systems. In International Conference on Electrical, Electronic and Computer Engineering, 2004. ICEEC '04., pages 144–148. IEEE, Sep. 2004.

87. Farajzadeh, N., Hashemzadeh, M., Mousakhani, M., and Haghighat, A. T.: An efficient generalized deadlock detection and resolution algorithm in distributed systems. In The Fifth International Conference on Computer and Information Technology (CIT'05), pages 303–309. IEEE, Sep. 2005.

88. Banerjee, S. and Chrysanthis, P. K.: A new token passing distributed mutual exclusion algorithm. In Proceedings of 16th International Conference on Distributed Computing Systems, pages 717–724. IEEE, May 1996.

89. Kshemkalyani, A. D. and Singhal, M.: Efficient detection and resolution of generalized distributed deadlocks. IEEE Transactions on Software Engineering, 20(1):43–54, Januray 1994.

90. Kshemkalyani, A. D. and Singhal, M.: A one-phase algorithm to detect distributed deadlocks in replicated databases. IEEE Transactions on Knowledge and Data Engineering, 11(6):880–895, November 1999.

91. Bracha, G. and Toueg, S.: Distributed deadlock detection. Distributed Computing, 2(3):127–138, 1987.

92. Wang, J., Huang, S., and Chen, N.: A distributed algorithm for detecting generalized deadlocks. Tech. Rep., 1990.

93. Gramoli, V., Guerraoui, R., and Trigonakis, V.: Tm^2c: A software transactional memory for many-cores. In Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, pages 351–364, New York, NY, USA, 2012. ACM.

VITA

NAME Eman Abdullah Aldakheel

EDUCATION Ph.D., Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2019 (expected)

M.S., Computer Science, Bowling Green State University, Bowling Green, Ohio, 2011

B.S., Computer Science, Imam Abdulrahman Bin Faisal University, Dammam, Eastern Province, 2006

TEACHING IC3 certification lessons (New Horizons computer training Centres, Fall 2006)

Intro to Computer Science lessons (Imam Abdulrahman Bin Faisal University, Spring 2007)

Computer Science lessons (Riyadh Alislam school, Fall 2007)

Microsoft Office lessons (New Horizons computer training Centres, Spring 2007)

PUBLICATIONS Eman Aldakheel, Ugo Buy. "Efficient Run-time Method for Detecting and Resolving Deadlocks in Java Programs" In 33rd European Conference on Object-Oriented Programming Workshops (ECOOP, 2019).

Eman Aldakheel. "Deadlock Detector and Solver (DDS)" In Proceedings 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (ISSRE, 2018).

Eman Aldakheel, Ugo Buy, Simran Kaur. "DDS: Deadlock Detector and Solver" In Proceedings of 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion) (ICSE, 2018).

Hassan Rajaei, Eman Aldakheel. “Cloud Computing in Computer Science and Engineering Education” In Proceedings of The ASEE Computers in Education (CoED) (ASEE, 2013).

Eman Aldakheel. “A cloud computing framework for computer science education” OhioLINK Electronic Theses and Dissertations Center (2011).

Copyright Permission Statements

ACM Copyright and Audio/Video Release

Title of the Work: Deadlock Detector and Solver (DDS)

Author/Presenter(s): Eman Aldakheel:University of Illinois at Chicago
Type of material: Short Paper

Publication and/or Conference Name: 40th International Conference on Software Engineering Companion Proceedings

I. Copyright Transfer, Reserved Rights and Permitted Uses

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work.

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new ACM Consolidated TeX template Version 1.3 and above automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\copyrightyear{2018}
\acmYear{2018}
\setcopyright{acmcopyright}
\acmConference[ICSE '18 Companion]{40th International Conference on Software Engineering Companion}{May 27-June 3, 2018}{Gothenburg, Sweden}
\acmBooktitle{ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden}
\acmPrice{15.00}
\acmDOI{10.1145/3183440.3190331}
\acmISBN{978-1-4503-5663-3/18/05}
```

ACM TeX template .cls version 2.8, automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference. *Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.*

```
\CopyrightYear{2018}
\setcopyright{acmcopyright}
\conferenceinfo{ICSE '18 Companion,}{May 27-June 3, 2018, Gothenburg, Sweden}
\isbn{978-1-4503-5663-3/18/05}\acmPrice{$15.00}
\doi{https://doi.org/10.1145/3183440.3190331}
```

If you are using the ACM Microsoft Word template, or still using an older version of the ACM TeX template, or the current versions of the ACM SIGCHI, SIGGRAPH, or SIGPLAN TeX templates, you must copy and paste the following text block into your document as per the instructions provided with the templates you are using:

permissions from Permissions@acm.org.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05...$15.00
https://doi.org/10.1145/3183440.3190331

NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

II. Permission For Conference Recording and Distribution

* Your Audio/Video Release is conditional upon you agreeing to the terms set out below.

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ◉ Yes ○ No

III. Auxiliary Material

Do you have any Auxiliary Materials? ○ Yes ◉ No

IV. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

V. Artistic Images
If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part V and be sure to include a notice of copyright with each such image in the paper.
○ We/I do not have any artistic images.
◉ We/I have any artistic images.

Artistic Images: Figure 1 ©Eman Aldakheel
Image Credits: Figure 1: Architecture of Deadlock Detector and Solver

---

VI. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants

DATE: **02/21/2018** sent to ealdak2@uic.edu at **10:02:23**

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

DDS: Deadlock Detector and Solver
Eman Aldakheel, Ugo Buy, Simran Kaur
2018 IEEE International Symposium on Software Reliability Engineering Workshops

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Eman Aldakheel — **Signature**

28-08-2018 — **Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the

IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to the final, published article in IEEE Xplore.

Questions about the submission of the form or manuscript must be sent to the publication's editor.
Please direct all questions about IEEE copyright policy to:
IEEE Intellectual Property Rights Office, copyrights@ieee.org, +1-732-562-3966