



Pós-Graduação em Ciência da Computação

Rafael Brandão Lôbo

## **DEADLOCKS AS RUNTIME EXCEPTIONS**

M.Sc. Dissertation



Federal University of Pernambuco  
posgraduacao@cin.ufpe.br  
[www.cin.ufpe.br/~posgraduacao](http://www.cin.ufpe.br/~posgraduacao)

RECIFE

2015





Federal University of Pernambuco  
Center for Informatics  
Graduate in Computer Science

Rafael Brandão Lôbo

## **DEADLOCKS AS RUNTIME EXCEPTIONS**

*A M.Sc. Dissertation presented to the Center for Informatics  
of Federal University of Pernambuco in partial fulfillment  
of the requirements for the degree of Master of Science in  
Computer Science.*

*Advisor: Fernando Jose Castor de Lima Filho*

RECIFE  
2015

---

Rafael Brandão Lôbo

Deadlocks as Runtime Exceptions/ Rafael Brandão Lôbo. – RECIFE, 2015-  
81 p. : il. (algumas color.) ; 30 cm.

Advisor Fernando Jose Castor de Lima Filho

M.Sc. Dissertation – Universidade Federal de Pernambuco, 2015.

1. Palavra-chave1. 2. Palavra-chave2. I. Orientador. II. Universidade xxx. III.  
Faculdade de xxx. IV. Título

CDU 02:141:005.7

---

Defesa de dissertação de mestrado apresentada por **Rafael Brandão Lôbo** ao programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **Deadlocks as Runtime Exceptions**, orientada pelo **Prof. Fernando Jose Castor de Lima Filho** e aprovada pela banca examinadora formada pelos professores:

---

Prof. Fernando Jose Castor de Lima Filho  
Centro de Informática/UFPE

---

Prof. Henrique Emanuel Mostaert Rebêlo  
Centro de Informática/UFPE

---

Prof. Nélio Alessandro Azevedo Cacho  
Departamento de Informática e Matemática Aplicada/UFRN



*I dedicate this thesis to all my family, friends and professors  
who gave me the necessary support to get here.*





# Acknowledgements

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.



*I have yet to see any problem, however complicated, which, when looked at  
in the right way, did not become still more complicated.*

—POUL ANDERSON



# Resumo

Deadlocks são um tipo comum de bug de concorrência. Quando um deadlock acontece, é difícil determinar claramente se houve um deadlock de verdade ou se a aplicação está lenta ou travada por qualquer outro motivo. Também é difícil estabelecer a causa do deadlock. Em geral, desenvolvedores lidam com deadlocks de várias maneiras: utilizando ferramentas analíticas; utilizando mecanismos específicos da aplicação para detectar deadlocks; ou simplesmente usando técnicas para evitar a ocorrência de deadlocks no momento da construção do código. Neste trabalho, propomos uma abordagem diferente. Acreditamos que se deadlocks se manifestarem durante a execução na forma de exceções, programadores serão capazes de identificar esses deadlocks de forma mais precisa e mais rápida. Levamos em consideração alguns aspectos para tornar esta abordagem prática: (i) a maioria dos deadlocks que ocorrem em sistemas reais envolvem apenas duas threads adquirindo dois locks (TTTL deadlocks); e (ii) é possível detectar TTTL deadlocks de forma suficientemente eficiente para uso prático na maioria dos sistemas. Conduzimos um estudo com bugs reportados em softwares de larga escala e descobrimos que mais de 90% dos bugs identificados como deadlocks eram de fato TTTL. Extendemos a classe `ReentrantLock` de Java para detectar TTTL deadlocks e medimos seu overhead na performance com um benchmark bastante conservador. Para aplicações não dominadas por travas, o overhead é estimado em menos de 6%. Realizamos uma avaliação empírica para testar usabilidade através de dois experimentos. Nesta avaliação, mostramos que, em média, estudantes terminam tarefas de 16.87% a 30.7% mais rapidamente usando nossa abordagem, tendo o tipo de abordagem como fator de maior significância, e, em um dos experimentos, número de respostas corretas aumentaram significativamente em 81.25%.

**Palavras-chave:** Deadlock, Concorrência, Tratamento de Exceção, Estudos Empíricos



# Abstract

Deadlocks are a common type of concurrency bug. When a deadlock occurs, it is difficult to clearly determine whether there is an actual deadlock or if the application is slow or hanging due to a different reason. It is also difficult to establish the cause of the deadlock. In general, developers deal with deadlocks by using analysis tools, introducing application-specific deadlock detection mechanisms, or simply by using techniques to avoid the occurrence of deadlocks by construction. In this paper we propose a different approach. We believe that if deadlocks manifest at runtime, as exceptions, programmers will be able to identify these deadlocks in an accurate and timely manner. We leverage two insights to make this practical: (i) most deadlocks occurring in real systems involve only two threads acquiring two locks (TTTL deadlocks); and (ii) it's possible to detect TTTL deadlocks efficiently enough for most practical systems. We conducted a study on bug reports and found that more than 90% of identified deadlocks were indeed TTTL. We extended Java's `ReentrantLock` class to detect TTTL deadlocks and measured the performance overhead of this approach with a conservative benchmark. For applications whose execution time is not dominated by locking, the overhead is estimated as below 6%. Empirical usability evaluation in two experiments showed that students finished tasks 16.87% to 30.7% faster on the average using our approach with the lock being the most significant factor behind it, and, in one of the experiments answers were significantly more accurate (81.25% more correct bugs found).

**Keywords:** Deadlock, Concurrency, Exception Handling, Empirical Studies





# List of Figures

5.1	Regression model. . . . .	53
5.2	First experiment box-plot graphic. . . . .	53
5.3	First experiment: before and after box-cox transformation ( $\lambda = 5$ ). . . . .	54
5.4	Tukey Test of Additivity hypothesis . . . . .	54
5.5	Second experiment box-plot graphic. . . . .	55
5.6	Before and after box-cox transformation ( $\lambda = 1.3636$ ). . . . .	55



# List of Tables

3.1	Labeled Categories and Estimations . . . . .	39
5.1	Criteria to evaluate students answers . . . . .	50
5.2	Latin Square design . . . . .	51
5.3	First experiment's average time spent . . . . .	53
5.4	Undergraduate students experiment ANOVA results. . . . .	54
5.5	Second experiment's average time spent . . . . .	55
5.6	Graduate students ANOVA results. . . . .	56
5.7	Undergraduate students answers accuracy . . . . .	56
5.8	Graduate students answers accuracy . . . . .	56
5.9	Benchmark time measurements (in seconds) . . . . .	60
A.1	Sampled bug reports (part 1) . . . . .	69
A.2	Sampled bug reports (part 2) . . . . .	70
A.3	Sampled bug reports (part 3) . . . . .	71
A.4	Sampled bug reports (part 4) . . . . .	72
A.5	Sampled bug reports (part 5) . . . . .	73
A.6	Bug report study: search queries and results . . . . .	73



# List of Acronyms

<b>IDE</b>	Integrated Development Environment .....	35
<b>TTTL</b>	two-thread, two-lock .....	24



# Contents

<b>1</b>	<b>Introduction</b>	<b>23</b>
1.1	Summary of Goals . . . . .	24
1.2	Outline . . . . .	25
<b>2</b>	<b>Background Chapter</b>	<b>27</b>
2.1	Introduction . . . . .	27
2.2	Static Analysis . . . . .	27
2.3	Dynamic Analysis . . . . .	28
2.4	Mixing Static and Dynamic Analysis . . . . .	30
2.5	Deadlocks as Exceptions . . . . .	32
<b>3</b>	<b>Bug Reports Study</b>	<b>35</b>
3.1	Introduction . . . . .	35
3.2	Sample Collection . . . . .	35
3.3	Data Labeling . . . . .	36
3.3.1	Category . . . . .	36
3.3.2	Number of threads/resources . . . . .	37
3.3.3	Type . . . . .	37
3.3.4	Notes . . . . .	38
3.4	Labeling Guidelines . . . . .	38
3.5	Results Analysis . . . . .	39
3.6	Threats to Validity . . . . .	39
<b>4</b>	<b>Deadlock Detection</b>	<b>41</b>
4.1	Introduction . . . . .	41
4.2	Deadlock Detection Protocol . . . . .	41
4.2.1	Proof . . . . .	43
4.3	Protocol Extension . . . . .	44
4.4	Implementation . . . . .	45
<b>5</b>	<b>Evaluation</b>	<b>49</b>
5.1	Introduction . . . . .	49
5.2	Usability Evaluation . . . . .	49
5.2.1	Time Analysis . . . . .	50
5.2.1.1	Metrics Collection . . . . .	51
5.2.1.2	Experiment Operation . . . . .	52

5.2.1.3	Results . . . . .	52
5.2.2	Accuracy Analysis . . . . .	55
5.2.3	Discussion . . . . .	57
5.2.4	Threats To Internal Validity . . . . .	57
5.2.4.1	Time Measurement . . . . .	57
5.2.4.2	Exercises . . . . .	58
5.2.5	Threats to External Validity . . . . .	59
5.2.5.1	Students . . . . .	59
5.3	Performance Analysis . . . . .	59
<b>6</b>	<b>Concluding Remarks</b>	<b>61</b>
6.1	Research Contributions . . . . .	61
6.2	Future Work . . . . .	62
	<b>References</b>	<b>63</b>
	<b>Appendix</b>	<b>67</b>
<b>A</b>	<b>Bug Reports Study Attachments</b>	<b>69</b>
<b>B</b>	<b>Evaluation Chapter Attachments</b>	<b>75</b>
B.1	Time Analysis . . . . .	75
B.2	Accuracy Analysis . . . . .	79
B.3	Experiment First Assignment . . . . .	80
B.4	Experiment Second Assignment . . . . .	81



# 1

## Introduction

Real-world applications use concurrency to take the more advantage of multicore processors and do computation in parallel using multiple threads/processes. However concurrent code is difficult to write correctly, as it is well documented ([LU et al., 2008](#)). In a concurrent code, for example, a developer must take in consideration all possible interleaves that multiple threads in the running code can take which is usually not feasible, and when multiple threads read and write concurrently a certain value in memory, data races may occur. Identifying parts of code that should not allow concurrent threads to run simultaneously is one step forward solving the problem. These areas are known as critical sections and one way to prevent such data races is to protect them with locks.

Lock is a synchronization mechanism that enforce mutual exclusion policy over threads for any given resource. Threads attempting to acquire a lock that was already acquired by other thread will be blocked until that resource is released. Only the owner of the lock can release that lock and unblock others, so usually a thread releases it when it finishes to execute code in a critical section, allowing other threads to proceed. They are widely used to protect critical zones, thus avoiding data races in concurrent code. However a problem introduced by badly composing locks is known as deadlock ([PYLA, 2013](#)).

Deadlocks are a very common type of error in concurrent systems ([LU et al., 2008](#)). They manifest when threads are waiting each other in a cycle, where each thread is waiting for another thread to release its desired lock, producing a never-ending wait.

There are two well-documented types of deadlocks, resource deadlocks and communication deadlocks ([SINGHAL, 1989](#))([KNAPP, 1987](#)). Resource deadlocks are deadlocks that stem from threads attempting to obtain exclusive access to resources. Communication deadlocks are deadlocks caused by at least one thread waiting for any kind of signal that either never arrives or arrived too early to be detected. As other studies did ([DA LUO; DAS; QI, 2011](#))([CAI; CHAN, 2012](#)), in this work our focus is also on resource deadlocks and whenever the term *deadlock* is used we will implicitly mean resource deadlock.

In practice, developers employ a number of approaches to deal with deadlocks: (i) static program analyses ([MARINO et al., 2013](#))([ENGLER; ASHCRAFT, 2003](#))([SHANBHAG,](#)

2008)(WILLIAMS; THIES; ERNST, 2005); (ii) dynamic program analyses (DA LUO; DAS; QI, 2011)(CAI; CHAN, 2012)(PYLA; VARADARAJAN, 2010)(PYLA; VARADARAJAN, 2012)(PYLA, 2013)(QIN et al., 2005); (iii) application-specific deadlock detection infrastructures (ORDEREDLOCK, 2014); (iv) techniques to guarantee the absence of deadlocks by construction (MARINO et al., 2013); (v) model checking (HAELUND; PRESSBURGER, 2000). The first two approaches are known to be heavyweight. In addition, the former often produces many false positives. The third approach has limited applicability and often imposes a high runtime overhead. The fourth approach has a low cost but cannot be employed in cases where it is not feasible to order lock acquisitions nor use non-blocking locking primitives. Finally, model checking is a powerful solution but has limited scalability when applied in the context of real programs. It also has limited generality, since some programs with side effects simply cannot be model checked.

In this paper we advocate an approach that complements the aforementioned ones. In summary, we believe deadlocks should not fail silently but instead their occurrence should be signaled as exceptions at runtime. To make this vision possible, we leverage two insights: (i) the vast majority of existing deadlocks occur between two threads attempting to acquire two locks (as reported by other authors (LU et al., 2008) and confirmed by us in Section 3); (ii) it is possible to efficiently introduce deadlock detection for these two-thread, two-lock (TTTL) deadlocks within the locking mechanism itself, incurring in an overhead that is low for applications whose execution time is not dominated by locking. We present a new type of lock that automatically checks for TTTL deadlocks at runtime and, if one is found, throws an exception indicating the problem. We have implemented this approach as an extension to Java’s `ReentrantLock` class. Deadlock exceptions are already supported in programming languages such as Haskell (MARLOW, 2013) and Go (AIMONETTI, 2015) but they focus on different types of deadlocks, and runtime exceptions for other concurrency bugs, e.g., for data races, have been proposed in previous studies (BISWAS et al., 2015).

We present data from an empirical study showing that our assumption about the prevalence of TTTL deadlocks holds in practice. It confirms the findings of a previous study that focused on concurrency bugs in general (LU et al., 2008). To evaluate our approach, we conducted two controlled experiments. In both cases, subjects using these new locks were able to detect deadlocks significantly faster than subjects not using them. Furthermore, in one of the studies, this approach helped the subjects to more accurately identify the causes of the deadlock. We also show that our approach has an overhead that, while non-negligible, is low for applications whose execution time is not dominated by locking.

## 1.1 Summary of Goals

In this work we want to accomplish the following goals: (i) verify that TTTL deadlocks are the most common type of deadlock in practice; (ii) present a solution for TTTL deadlock

detection that can be implemented with significantly low overhead; and (iii) evaluate deadlock exceptions' impact on software development productivity, such as finding deadlocks in code and understanding how they happen more accurately.

## 1.2 Outline

In this introduction, we briefly contextualized the research and our motivation behind introducing deadlock runtime exceptions in programming languages and how it is different from previous studies. The following chapters will go in more details about it:

- Chapter 2 highlights previous studies regarding deadlock detection and why we believe it is important to have deadlock exceptions support in programming languages.
- Chapter 3 presents our study on bug reports in open source projects, focusing on deadlock bugs and identifying some of its characteristics.
- Chapter 4 shows our deadlock detection protocol, a sketch of a proof to support it, and explains what changes we did on Java's *ReentrantLock* to support it.
- Chapter 5 discuss the empiric evaluation we did to measure efficiency of deadlock exceptions and a brief performance analysis of runtime overhead.
- Chapter 6 presents the contributions of this work, discusses some related and future work, and presents our main conclusions.
- Appendix follows complementing with all data we used for bug reports study and how we calculated experiments results.



# 2

## Background Chapter

### 2.1 Introduction

In this chapter, previous studies related to deadlock detection topic will be briefly discussed, starting with two most common techniques: static analysis and dynamic analysis. Then we will move into the idea of deadlock exceptions and what other languages already supported some sort of deadlock exception today and why it should be useful.

### 2.2 Static Analysis

Static analysis techniques verify source code from a program and try to identify potential problems present in the code without even executing it. For deadlock detection, they generally attempt to detect cyclic relationships of resource acquisition between threads where each cycle represents a possible deadlock. Many static analysis approaches were proposed over the past decade ([ENGLER; ASHCRAFT, 2003](#))([BOYAPATI; LEE; RINARD, 2002](#))([SHANBHAG, 2008](#))([PRAUN, 2004](#))([FLANAGAN et al., 2013](#))([WILLIAMS; THIES; ERNST, 2005](#)) but in general they suffer of significant amount of false positives being reported as there may exist deadlock cycles that are just impossible to happen during execution. Also, some deadlocks cannot be detected when the language is weakly typed, such as C/C++.

Recently, [MARINO et al. \(2013\)](#) proposed a static analysis technique to detect potential deadlocks in programs that used an extension of Java called AJ that implements atomic sets for class fields as an abstraction of locks to prevent data races and atomicity violations by construction. Its declarative nature allows the algorithm to infer which locks each thread may acquire and compute a partial order for those atomic sets which would also be consistent with lock acquisition order. If such order was detected, the program was guaranteed to be deadlock-free, otherwise possible deadlock would be reported. It was implemented as extension of their existing AJ-to-Java compiler and synchronization annotations were given as special Java comments in the code. These comments would be parsed and given to the type checker to execute the deadlock analysis. AJ source code would be translated to Java and written into a separated project with

the transformed code which would later be compiled into bytecode and executed by JVM. A limitation of this approach is that AJ is a research language and does not have real users, thus obtaining suitable subject programs to do any evaluation was difficult, and their chosen projects to evaluate might not represent concurrent programming styles that occur in practice, but at least for most of the subject programs analysed, deadlock-freedom could be demonstrated without any programmer intervention.

## 2.3 Dynamic Analysis

Dynamic analysis finds potential deadlock cycles from an execution trace of a program which makes them often more scalable and precise than static analysis. However, due to the sizes of large-scale programs, the probability that a given run will show a thread acquiring a lock at the right time to trigger a deadlock for each potential deadlock present in the code is very low, which poses a challenge for dynamic deadlock detection tools. Thus if a given run does not identify potential deadlocks, it does not mean the program is free from deadlocks either. Another disadvantage of dynamic analysis is that they often incur runtime overhead and most techniques can't be applied in real-world applications for lack of scalability.

DA LUO; DAS; QI (2011) present a dynamic deadlock detection tool called MulticoreSDK which consists of two phases. In the first phase, the algorithm works offline by examining a single execution trace obtained after an instrumented program finishes its execution, creating a lightweight lock graph based on program locations of lock events and identifying locks that may be deadlock-related in this graph by checking cycles on it. Second phase consists of examining the execution trace and constructing a filtered lock graph based on the lock id of each lock event, analysing only deadlock-related locks found in the first phase. Each cycle found in this final graph is reported as potential deadlock in the program. It works for java programs and its instrumentation is done by deploying a bytecode instrumentation technique (TANTER et al., 2002) to insert extra bytecode around instructions such as *monitorenter* and *monitorexit* which would record the thread and the id of lock objects, also around methods lock and unlock of *Lock* interface in `java.util.concurrent` package and enter/exit of synchronized methods. Although the algorithm requires to pass over the program trace twice, most of its performance overhead is consumed in the deadlock analysis itself. When compared with a traditional approach (FARCHI; TZOREF, ???), MulticoreSDK has a very small performance gain in analysis time on small applications while memory consumption was reduced by about half; meanwhile, for large applications, about 33% performance gain was reached, consuming about 10 times less memory.

CAI; CHAN (2012) present a dynamic analysis technique known as MagicFuzzer which consists of three phases. First phase consists of critical events such as a thread creation or lock acquisition and release being monitored in a running program to generate a log of series of lock dependencies which can be viewed as a lock dependency relationship. On the second

phase, the algorithm classifies such relations in a cyclic-set which contains all the locks that may occur in any potential deadlock cycles found in a given execution. Then it constructs a set of thread-specific lock-dependency relations based on the locks inside that cyclic-set which is finally traversed to find potential deadlock cycles. On the third phase, all deadlock cycles found in second phase are used as input to the program execution, and it is actively re-executed to observe if any further execution will trigger any of those potential deadlock cycles in the input, reporting the deadlocks whenever they occur, until all deadlocks are found or the limit of times to re-execute is reached. One of its key differences is the presence of an active random scheduler to check against a set of deadlock cycles which is said to improve the likelihood that a match between a cycle and an execution will be found, but its proof is left open to future work. It was implemented using a dynamic instrumentation analysis tool known as Pin 2.9 (LUK et al., 2005) which only works for C/C++ programs that use Pthreads libraries on a Linux System. It was tested on large-scale C/C++ programs such as *MySQL* (MYSQL, 2015), *Firefox* (FIREFOX, 2015) and *Chromium* (CHROMIUM, 2015), and for some of them, the test would simply start the program and then close it when the interface appeared, while other programs used test cases adopted from bug reports in their own repositories. After evaluating the results of tests, it was concluded that it could run efficiently with low memory consumption if compared to similar techniques, like MulticoreSDK (DA LUO; DAS; QI, 2011).

PYLA; VARADARAJAN (2010) presents *Sammati* as a dynamic analysis tool which is capable of detecting deadlocks automatically and recovering for any threaded application that uses the pthreads interface without any code changes, transparently detecting and resolving deadlocks. It is implemented as a library that overloads the standard pthreads interface and turns locks into a deadlock-free operation. By associating memory accesses with locks and specifying ownership of memory updates within a critical section to a set of locks, it makes memory updates to be only visible outside of the critical section when all parent locks are released. When a deadlock is detected, one of the threads involved in the deadlock is chosen as a victim and rolled back to the point of acquisition of the offending lock, discarding memory updates done. This way it would be safe to restart the code that runs into that critical section again because the memory updates never were made visible outside of the critical section. Differently from transactional memory systems, its performance impact is minimized because its runtime privatizes memory at page granularity instead of instrumenting individual load/store operations. Containment for threaded codes was implemented with a technique described in (BERGER et al., 2009): if each thread had its own virtual address space, such as most recent UNIX operating systems, privatization can be implemented efficiently in a runtime environment where threads are converted to *cords* which is a single threaded process. In this algorithm, threads are converted to cords transparently, and since all cords share a global address space, the deadlock detection algorithm has access to the holding and waiting sets of each cord and the deadlock detection can be performed at the time a lock is acquired. Its detection algorithm uses a global hash table that associates locks being held with its owning cord, another hash table to associate a cord with a lock it is waiting on and

a queue per cord of locks acquired. Its deadlock detection complexity is linear and has upper bound of  $\mathcal{O}(n)$  where  $n$  is the number of threads. However, rollback of acquired locks and all its related memory changes adds a high runtime overhead, given the high costs related to its address space protection and privatization logic. Also, there are some limitations in what kind of memory can be recovered in a rollback, like non-idempotent actions (i.e. I/O, semaphores, condition signals, etc.) and shared libraries state (PYLA, 2013). Part of these limitations were addressed with Serenity *serenity*, like serializing some disk I/O operations within critical sessions, and it will be discussed further in the next section.

QIN et al. (2005) proposes a dynamic approach called Rx that rolls back a server application once a bug occurs to a checkpoint, trying to modify the server environment on its re-execution. In that study, bugs are treated as "allergies" and different program executions could either have the "allergen" or not. In the context of concurrency bugs such as deadlocks, as timing is essential for deadlocks to manifest, a retry of the server request that caused a deadlock could be enough to get rid of the deadlock. It works by dynamically changing the execution environment based on the failure symptoms, and then executing again the same code region that contained the bug, but now in the new environment. If the re-execution successfully pass through the problematic period, all environment changes are disabled thus time and space overheads are avoided. Intercept of memory-related library calls (i.e. *malloc()*, *realloc()*, etc.) was implemented on a memory wrapper in order to provide environmental changes. During normal execution, it will simply invoke the corresponding standard memory management library calls, which incurs little overhead. Then at re-execution phase (called *recovery* mode), the memory wrapper would activate a memory-related environmental change instructed by the control unit. Its evaluations shows that it recovered from software failure in about 0.16 seconds which was significantly better compared to what was done before (web application restart).

## 2.4 Mixing Static and Dynamic Analysis

It is also possible to mix both static analysis and dynamic analysis in an hybrid analysis to take advantage of both types and boost performance during runtime. Some techniques deploy static analysis to infer deadlock types and dynamic analysis for the parts of the program that could not be analysed in the first part, reducing the overhead caused by dynamic analysis drastically.

GRECHANIK et al. (2013) proposes *REDACT* that would statically detect all hold-and-wait cycles among resources in SQL statements and prevent database deadlocks dynamically by running a supervisory program to intercept transactions sent by applications to databases. Once a potential deadlock is detected, the conflicting transaction is delayed which results on breaking the deadlock cycle. As database deadlocks may degrade performance significantly, it tries to prevent them of happening. An static analysis is executed at compile time and later used at runtime, allowing its deadlock detection algorithm to just make simple lookup in the hold-and-wait cycles already detected in compile time, improving deadlock detection performance compared to similar



techniques. In the first step, some manual-effort is required on every application to build all database transactions used in the application, which allows static analysis phase to begin. Its results are parsed and used to generate hold-and-wait cycles necessary by the runtime detection algorithm. The dynamic phase is responsible to prevent database deadlocks automatically. It adds interceptors in the application with callbacks associated with particular events and instead of sending SQL statements from application to the database, they divert the statements to the supervisory controller whose goal is to quickly analyse if hold-and-wait cycles are present in the SQL statements that are in the execution queue. If no hold-and-wait cycle is present, then it forwards these statements to the database for execution normally. Otherwise, it holds back one SQL statement while allowing others to proceed, and once these statements are executed and results are sent back to applications, the held back statement is finally executed by the database. Some of its limitations include that not all hold-and-wait cycles detected in the static analysis will actually lead to database deadlocks, so false positives are still possible. Databases are highly optimized against deadlocks, so even though some obvious deadlock should happen in theory, in practice they would be prevented by the database itself, but this approach will handle such cases as if a deadlock would happen anyway and consequently imposes an overhead in addition to some reduction in the level of parallelism in applications. Also, when the static analysis produces too many false positives of hold-and-wait cycles, there is a significant performance overhead overall.

[PYLA; VARADARAJAN \(2012\)](#) presents Serenity as a successor of *Sammati* ([PYLA; VARADARAJAN, 2010](#)) to handle some previous limitations by mixing some runtime analysis with its previous runtime approach, so they share many similarities. Program analysis and compile time instrumentation were used in Serenity to guide runtime and efficiently achieve memory isolation with some static analysis techniques to infer the scope of a lock and using selective compile time instrumentation on those identified scopes. In order to handle some previous I/O limitations, Serenity serializes disk I/O operations performed within critical sections and marks threads that performed some of these operations. When a deadlock is detected, differently from *Sammati*, it will choose as victim a thread that did not perform I/O inside any lock context. It also tracks all disk I/O primitives (i.e. read, write, etc.) and all thread creation/termination primitives (i.e. fork, clone, etc.) as I/O operations to verify whether such operations were performed within a lock context. The new approach also has its own limitations, as it cannot recover from certain deadlocks such as arbitrary I/O within the context of a restartable critical section ([PYLA, 2013](#)). It also requires some programmer input if lock/unlock operations are encapsulated in functions/classes, so they should be explicitly identified by the developer before running its compile time analysis. Furthermore, recompilation is now necessary for application and all its external dependencies (i.e. shared libraries) in order to also instrument them.

## 2.5 Deadlocks as Exceptions

Deadlock detection or recovery algorithms are not always as efficient or reliable as they should be, as we've seen in the previous sections. Some techniques generates false positives as deadlocks candidates, other techniques imposes high runtime overhead. Some also offer recovery from deadlocks automatically, but they don't work if certain operations happen inside critical sections. Finally, some techniques are too specific to the type of application (such as webserver or databases) and they are not easily translated into other application types efficiently.

Building deadlock-free programs is an old challenge which is not yet solved efficiently. Many years ago, Concurrent Pascal ([HANSEN, 1975](#)) was proposed as programming language with a compile-time deadlock detection built-in. It had several limitations such as fixed amount of processes and recursive functions not allowed. In that limited scenario, it could process a partial ordering of process interactions and prove certain system properties such as absence of deadlocks during compilation.

In modern languages and databases systems, deadlocks are nothing but bugs, and they have exceptions in few special cases. Database deadlocks are typically detected within database engines at runtime ([GRECHANIK et al., 2013](#)) raising deadlock exceptions and rolling back transactions involved in the circular wait. Haskell programming language throws deadlock exceptions when garbage collector detects a thread as unreachable<sup>1</sup> and no other thread can wake it up. Another programming language, Go, throws deadlock exception when all *goroutines* are blocked waiting on their message channel<sup>2</sup>.

We believe the presence of deadlock exceptions in programming languages are very useful specially when it's guaranteed to be fired and programmers can rely on them. The main advantage is that they inform there's something unexpectedly wrong in the code which allows programmers to identify the problem easier and potentially fix it faster. Furthermore if the exception is guaranteed to be fired then programmers can have specialized code to run when it happens.

For instance, a deadlock exception could be handled in the code to trigger some deadlock recovery mechanism specialized for that case, if it is indeed necessary. This approach has an advantage on its own because it does not rely on any auto-recovery mechanism as seen in some previous deadlock detection algorithms, because they were in general very inefficient. Thus, automatically recovering from deadlocks could potentially cover real bugs that needs to be fixed immediately and also turn application's performance a lot worse in exchange.

In non-interactive systems, there's a huge benefit of having deadlock exceptions in the programming language. Currently, it is very difficult to find a deadlock when it happens but it's a lot harder when the system is not monitored constantly, as non-interactive systems usually are. [PYE \(2011\)](#) discussed how difficult it was to identify one of the concurrency bugs which was

---

<sup>1</sup><http://goo.gl/v09kqn>

<sup>2</sup><http://guzalexander.com/2013/12/06/golang-channels-tutorial.html>

just a classical deadlock (two threads circularly waiting each other). However, the deadlock did not leave the application completely stuck, but instead an intended behaviour in a specific case did not happen because of this hidden deadlock which made the problem very hard to debug and be identified.

We want to solve this problem by extending programming languages to support deadlock exceptions. We believe they can help developers to find and fix their mistakes in the code when they have such exception. Although we started this work with our focus on Java developers, our idea is not limited to Java. We think other languages could also benefit from deadlock exceptions.



# 3

## Bug Reports Study

### 3.1 Introduction

Attempting to generalize deadlock detection at runtime does not seem feasible from a performance viewpoint, since existing dynamic analyses take considerable time (CAI; CHAN, 2012). But study conducted by LU et al. (2008) found that 30 out of 31 deadlock bug reports involved at most two resources. We suspected TTTL deadlocks were more common in real world systems than more complex deadlocks, so we investigated this further. This chapter presents the results of this investigation.

### 3.2 Sample Collection

We've chosen three open source projects which used Java as main programming language and made use of concurrent programming: Lucene<sup>1</sup>, Eclipse<sup>2</sup> and OpenJDK<sup>3</sup>.

Lucene is a text search engine library that can be used along many applications, where concurrent programming was used to deliver high performance. Eclipse is one of the most popular Integrated Development Environment (IDE) for java developers. OpenJDK is an open-source implementation of the Java Platform. These three projects share a few similarities: they're written in Java; they have vast inventory of bug reports on their repositories and publicly available tools to search for bug reports; and lastly, they share a software development culture of reviews inside bug reports by discussing solutions to fix the problem. In particular, this last characteristic was very important to allows us to analyze bug reports and infer their classification with confidence.

Initially a search for bug reports was conducted in each repository for the keyword *deadlock*, collecting 541 bug reports in total. In reality, to perform a search for *deadlock*, as each project had a different bug repository, we used slightly different query parameters to find relevant

---

<sup>1</sup><http://lucene.apache.org/>

<sup>2</sup><https://eclipse.org/>

<sup>3</sup><http://openjdk.java.net/>

bug reports (more details in Table A.6). In Lucene, the results should match the word "deadlock" anywhere in the bug report (i.e. in summary or in comments), on module "lucene-core" with issue type set to "bug" and status as "closed", which found 27 bugs. In Eclipse, search was set to contain bugs with word "deadlock" in summary, with resolution set to "fixed" and status as "resolved", resulting in 406 bugs. In OpenJDK, bugs should contain word "deadlock" inside the summary, they should be related to its module named "JDK", but they also should have issue type "bug", resolution "fixed" and status "resolved", finding 108 bugs. We then proceeded to calculate the sample size that would allow us to have 95% of confidence level and 5% sampling error (with 50% of response distribution for optimum size) which resulted in 225 bugs. Thus we created a random sample of that size to analyze further (see Appendix A).

### 3.3 Data Labeling

We have merged all bug reports of our random sample in one single table, where each row represented a different bug report and each column represents an attribute that we were interested. The first attribute was the name of the bug. Each name was composed by a prefix that could be either *LUCENE*, *ECLIPSE* or *JDK*, followed by the bug number on their own repository. The second attribute was the category: a character that could be either "A", "B", "C" or "D", added by us after going through that particular bug report with our manual inspections. Other fields such as *type*, *number of threads*, *number of resources* and *notes* will be detailed shortly. Although we collected additional fields such as *time* and *comments* and made available in the final version of this table, we did not use them for the analysis conducted in this work.

#### 3.3.1 Category

This is one of the most important fields, as we want to be able to identify what kind of deadlock this bug represents, or if it's not a real deadlock. We have four different values for this field and they must be one of the following:

- *A*: We are confident this is a resource deadlock. We should be able to provide a short explanation of how the bug occurs, which or how many threads are involved and how many locks are involved in this bug.
- *B*: We are confident this is not a resource deadlock, so it must be a communication deadlock. It might be a lost notify/signal bug. We should be able to identify if this is a lost notify/signal or have clear evidence this is not a resource deadlock (adding a note whenever possible).
- *C*: We are confident this is a false-positive for "deadlock" search. The term was used as a synonym of "hang" or "infinite loop", or to refer to another deadlock bug. In some cases, it is possible that a bug refers to another bug which was fixing a deadlock,

so the initial bug may not be deadlock-related and just fix a regression for another bug (which could be deadlock-related). In other words, this is not a deadlock bug at all.

- *D*: We are not confident whether this is a resource deadlock or a communication deadlock, or even if this is a false-positive for deadlock. There's not enough information in the bug report, or the information is just inconclusive. Since we are not experts on any of these code repositories, it's hard to classify in any other category.

General guideline for classifying bugs in category A was to only assign it when there was a clear comment in the bug explaining what threads and which resources are involved or other evidences can clarify without doubt how many threads and lock resources are involved. In a few cases, the explanation was not fully clear but the attachments on the bug provided a clear thread dump report showing which threads were involved and which locks each one were holding and waiting for. When such evidence was present, we could use it to make the final decision. Similarly, bugs in category B could also be confirmed by looking into source code changes in cases where we were almost clear about its category. For example, if the patch changes areas of the code where a `notifyAll` is added or moved, then it serves as a strong evidence to confirm it should be indeed in category B. In other cases, it is deadlock where the first thread is holding a lock but also it is in an infinite loop waiting for others to finish while other threads are stuck waiting to acquire a lock the first thread already acquired, so we would understand it as a communication deadlock: the "message" or "signal" which the first thread have been waiting is whether the other threads have finished, but it never arrives. Luckily, category C was often easy to classify since there usually was a comment in the bug report referencing another kind of bug, using the term "deadlock" as a synonym for "hang". If a particular bug only has a comment that refers to another bug (e.g. a regression) as a deadlock bug, then the bug being investigated might not be a deadlock by itself, which would also fall into this category. Finally, category D is for all other bugs which could not be classified as either A, B or C.

### 3.3.2 Number of threads/resources

Whenever possible, the reviewer should state the number of threads and resources involved, even if this is in the category B. If it's unknown how many resources but it is clear how many threads are involved, then only one of them should be filled and the other field should remain empty.

### 3.3.3 Type

This field is just an annotation and it should be used to specify what kinds of resources a certain bug use. For example if there were two threads and they were in a circular deadlock, then this field should be *locks/synchronized*. If explicit locks were used on both threads, then

just *locks* should be used, or if only synchronized blocks were involved, then just *synchronized* should be present. The symbol + indicates a separation between threads, so for example "locks + wait" means that one thread holds a lock while the other (probably holding the lock) waits for a signal. Unfortunately, as this nomenclature could be confusing, the field *notes* should be used to clarify and write down what was found about this bug.

### 3.3.4 Notes

This field was encouraged to be used specially to remind other reviewers in the future of how the conclusion was made for cases where it was not straight-forward to choose the category. In these cases, it should contain the evidence found that helped the final decision to be made.

## 3.4 Labeling Guidelines

In order to minimize error/bias on our classification and organized how the review was executed, we've created a set of guidelines that every reviewer should follow which basically describes how data should be analyzed for a certain bug, in what order, and what decisions could be made:

1. Look at bug title and bug main description (usually the first comment). Sometimes the reporter have an idea of how the bug occurs and which threads are involved, so this is a big help.
2. Look at further comments and see if someone understood this bug completely. Someone must have provided a reasonable explanation of how this bug occurs. If the category is already clear, then finish these steps; otherwise proceed.
3. If available, look at the patches (specially the final patch) and what changes have been made. If uncertain about this bug being in category B and the patch either moves or adds a `notifyAll` call, then it most likely is a category B bug. If this is not the case, then proceed.
4. If available, look at the related bugs or duplicates. It's often to find an initial bug that is unclear but which points out to a duplicate that have been largely discussed and is clear. Restart from step 1 for each of those related bugs. If a category was not assigned yet, then proceed.
5. See other attachments if available, like text files with thread dumps or stack traces. If they provide enough information to clarify which category it is, then assign a category to it, otherwise proceed.
6. Classify this bug in the category D.



**Table 3.1:** Labeled Categories and Estimations

Category	Number of Bugs	Estimated
A	101	146
A and TTTL	93	134
B	32	46
C	23	33
D	69	0

## 3.5 Results Analysis

Since we want to find how many resource deadlock bugs were TTTL deadlocks, we discard bugs in B and C category because they can't be resource deadlocks. What we have left are the bugs we could not determine its category with confidence. Thus in the worse case all bugs in category D should be resource deadlocks but none of them should be TTTL deadlocks. Worse case scenario is given by Equation 3.1. In that equation,  $bugs(...)$  returns the number of bug reports that matches all parameters. Now, if we want to look at the best case scenario, then all bugs classified in D category must also be TTTL deadlocks. In worse case, 54.7% resource deadlocks are TTTL deadlocks, while in the best case, 95.29% are TTTL deadlocks. In Table 3.1 (second column), we can see in that from all resource deadlocks we identified, 92.07% of them are indeed TTTL deadlocks. Another interesting finding is that 75.93% of all deadlocks are indeed resource deadlocks.

$$bugs\_ratio = \frac{bugs(A, threads = 2, resources = 2)}{bugs(A) + bugs(D)} . \quad (3.1)$$

However neither the worse case nor the best case scenarios seems realistic. We believe that a more realistic scenario would be to assume that bugs in category D are distributed roughly the same way as those in categories A, B, and C. If that is the case (last column of Table 3.1), out of all resource deadlocks, we estimate that 91.7% of them would also be TTTL deadlocks. Thus TTTL deadlocks are certainly the most popular type of resource deadlocks, amounting to more than 9 out of every 10 resource deadlocks. This result makes it evident that an approach to automatically detect these deadlocks has practical value.

## 3.6 Threats to Validity

Although we've created a set of guidelines that all reviewers should follow expecting to have more than one reviewer available on this research, in reality we couldn't find more than one reviewer to execute the labeling on all bug reports due to constraints on time and resources. There is both an advantage and a disadvantage given by this limitation. The advantage is that it is easier to guarantee that all bug reports were reviewed following the exact same set of rules

as they were all reviewed by the same person. The disadvantage, however, is that there was no second reviewer to double check if the labels were indeed coherent to their respective bug reports.

Futhermore, one factor that might limit generalization of these findings is that we've looked at only three different open-source projects written mostly in Java. In the real world, software is written in many different languages, where each language may have different distributions of deadlock bugs. However, as we've implemented deadlock exceptions in Java's `ReentrantLock`, we focused on investigating what kind of deadlock bugs developers usually face when developing with Java. This way we could understand whether our solution could be useful in practice. For that purpose, we've carefully chosen popular open-source projects that represent high quality software and contain well established open source communities. Their repositories had a rich resource of bug reports with many comments and documents that actually helped us to label each bug individually and they provided online tools to allow us to search into their bug report archives which was essential for this study. We believe that these projects were representative to evaluate the distribution of deadlock bugs for software written in Java. We also suspect our findings are also true for other popular languages but we leave it open for future work.

# 4

## Deadlock Detection

### 4.1 Introduction

In this chapter we present the proposed approach. We extend the notion of lock by making locks responsible for both detecting TTTL deadlocks and raising exceptions whenever such deadlocks occur. In this section we present an algorithm implementing this extended notion of lock and show that our algorithm guarantees that (i) every TTTL deadlock is detected; and (ii) if an exception reporting a deadlock is raised, it must stem from the occurrence of a TTTL deadlock.

### 4.2 Deadlock Detection Protocol

We have modified the default implementation of Java's `ReentrantLock` to allow efficient runtime detection of TTTL deadlocks. It works as follows:

1. Each lock has a pointer to a thread, the owner of the lock, or `null` when no thread owns that lock.
2. Each lock has an integer to represent its current state: 0 means the lock is free and no thread owns it (the *unlocked* state), 1 means there is a thread that owns the lock (the *locked* state). For simplicity, we are only interested on these two states. Nonetheless, in the implementation of `ReentrantLock`, each time a thread owner acquires the same lock, this state would be incremented, and decremented each time the thread releases it.
3. Each thread has a thread-local list of pointers to locks it currently owns.
4. Each lock has a waiting queue of threads that are waiting to acquire it. Whenever a thread tries to obtain a lock when it's already acquired, the thread will add itself to the waiting queue before parking. Upon the event of releasing the lock, the owner of that lock will look for the first thread in the waiting queue and unpark it.

5. When a thread wants to acquire a lock, it will swap the current state to *locked* if the current state is *unlocked* atomically.
  - (a) If the thread fails, it must be because the lock is already owned by some other thread, then it will add itself on the waiting queue for that lock. Finally, the thread will park.
  - (b) Otherwise, the thread will set itself as the current owner of that lock and also add this lock to its thread-local list of pointers of locks it owns.
6. When a thread is about to release a lock, the current owner pointer of that lock is set to `null` and that lock is also removed from the thread-local list of owned locks. Finally, the lock state is changed to *unlocked*.
7. Before parking, a thread will check whether there is a deadlock. When the current thread is unable to acquire its desired lock, it must be because another thread already owns it. It is possible to know who is the owner of any lock, so the current thread identifies the owner of its desired lock as the conflicting thread. Then the current thread will search on each lock of its list of owned locks if the conflicting thread is waiting for it.
  - (a) If positive, then we have a circular dependency (current thread is stuck waiting for its desired lock and the conflicting thread is stuck waiting for a lock the current thread owns) and thus a deadlock exception will be raised.
  - (b) Otherwise, the thread parks.

We take advantage of the current algorithm employed by `ReentrantLock` and some of its guarantees listed below to avoid the need to introduce extra synchronization mechanisms or costly atomic operations during deadlock detection:

1. The operation of swapping the state of a lock from *unlocked* to *locked* must be done atomically by the thread, so only one thread can be successful at a time.
2. A thread will only park when it is guaranteed that some other thread can unpark it. Missing notifications will never happen and concurrent uses of park and unpark on the same thread will be resolved gracefully.
3. Inserts on each lock's waiting queue must be done atomically. If multiple threads concurrently attempt to insert themselves in the waiting queue on the same lock, they will both succeed eventually but the exact order of insertions is not important.
4. Once the last element in the waiting queue of a lock is read, it should be safe to read all threads in the waiting queue that arrived before the last element. Since the thread who reads the waiting queues is also the one who blocks every thread waiting on the

queues, we can guarantee the only updates that could happen concurrently are new insertions at the end of each queue. However insertions in the end of the queue are not important once a last element pointer is obtained.

### 4.2.1 Proof

Now we provide a sketch of a proof to describe why this protocol works.

**Lemma 4.1.** *The proposed protocol can always detect TTTL deadlocks.*

*Proof.* By way of contradiction, suppose not and a TTTL deadlock occurred without it being detected. Lets assume that threads  $A$  and  $B$  have both acquired locks  $a$  and  $b$  respectively, as follows:

$$write_A(state_a = locked) \rightarrow write_A(owner_a = A) \quad (4.1)$$

$$write_B(state_b = locked) \rightarrow write_B(owner_b = B) \quad (4.2)$$

In the above expressions, ' $x \rightarrow y$ ' indicates that event  $x$  happened before event  $y$ . Notation ' $write_B(owner_b = B)$ ' indicates that thread  $B$  wrote to variable  $owner_b$  the value  $B$ . And now each thread will attempt to acquire the opposing lock: thread  $A$  is trying to acquire lock  $b$  and thread  $B$  is trying to acquire lock  $a$ , as follows:

$$read_A(state_b == locked) \rightarrow write_A(waiting\_queue_b.insert(A)) \quad (4.3)$$

$$read_B(state_a == locked) \rightarrow write_B(waiting\_queue_a.insert(B)) \quad (4.4)$$

The notation ' $read_A(state_b == locked)$ ' indicates that thread  $A$  read variable  $state_b$  and obtained value  $locked$ . If a TTTL deadlock happened, then both threads are now parked and all previous equations should be correct. But before parking, each thread must check for deadlock by inspecting each lock it owns if the opposing thread is on its waiting queue. As we initially assumed no deadlock exception has been raised, then both threads are parked and also the following equations must be correct:

$$read_A(owner_b == B) \rightarrow read_A(waiting\_queue_a.contains(B) == false) \quad (4.5)$$

$$read_B(owner_a == A) \rightarrow read_B(waiting\_queue_b.contains(A) == false) \quad (4.6)$$

The problem with the previous equations is that they both cannot be true simultaneously. Before checking for deadlock, each thread must add itself on the waiting queue of its desired lock. If it holds that the opposing thread is not in the waiting queue yet, then it must be because it did not start to check for deadlock yet, thus a contradiction.  $\square$

**Lemma 4.2.** *The proposed protocol never raises a deadlock exception for a non-existent TTTL deadlock.*

*Proof.* By way of contradiction, assume the opposite: a deadlock exception was raised and there is no real TTTL deadlock. Exactly one of the following equations must be true in order to raise a deadlock exception (if both were true at the same time, an actual deadlock would have occurred):

$$read_A(owner_b == B) \rightarrow read_A(waiting\_queue_a.contains(B) == true) \quad (4.7)$$

$$read_B(owner_a == A) \rightarrow read_B(waiting\_queue_b.contains(A) == true) \quad (4.8)$$

Suppose without loss of generality that the first equation is true. It means that thread  $B$  is waiting for lock  $a$  and it is also the owner of lock  $b$ . If it is on the waiting queue, that thread is either parked already or about to park and in both cases thread  $B$  is going to depend on the release of lock  $a$  to proceed. However, as we have seen previously, thread  $A$  at this point is also about to park and is checking for a deadlock. If this condition holds, we have a circular dependency between threads  $A$  and  $B$ , a real TTTL deadlock, thus we have a contradiction.  $\square$

### 4.3 Protocol Extension

The protocol we presented previously guarantees that an exception is raised in at least one of the threads involved in a deadlock. A safer approach, however, would be to have exceptions raised in both threads involved in the deadlock. In this section we describe an extension to the protocol that provides this guarantee. This does not affect how deadlock is detected but what should be done after a deadlock is detected. Thus, does not impact the correctness of the protocol. The proposed extension comprises the following:

1. Each lock has a list of tainted threads. This list should only be read or updated by the owner of that lock, allowing immunity from interference without any extra synchronization cost.
2. Once a deadlock is detected and the current thread is about to raise a deadlock exception, it already knows which thread is conflicting with itself and which lock that thread desires. The current thread (the owner of the desired lock) will add this conflicting thread to the tainted threads list for that lock. After that, the deadlock exception is raised.
3. When the conflicting thread is unparked and finally acquires its desired lock (it becomes the owner of that lock), then it is allowed to read the list of tainted threads. If this thread identifies itself in this list, then it must be because it was part of a deadlock before, so it removes its reference from the list and also raises a deadlock exception.
4. Every operation on the list of tainted threads of any lock (either reading or inserting values) should be followed up by some cleanup on all references to threads that are

no longer running.

That is sufficient to force both threads to raise exceptions when only one of them would raise an exception in the initial protocol. The latter only raises exception on both threads if they simultaneously reach the point where they check for deadlocks. However, for this particular case, this change introduces a different problem: dangling references. If each thread adds their conflicting thread to the lists of tainted threads of the locks they own, but none of them is able to acquire their respective desired locks (as in *item 3*), both threads will leave their references behind for others to cleanup (as in *item 4*). We minimize this issue by cleaning these references as soon as any thread acquires the lock.

## 4.4 Implementation

The full version of the modified OpenJDK `ReentrantLock` that implements this algorithm is available in our code repository ([GITHUB, 2015](#)). In our initial commit, we copied the original `ReentrantLock` source code and related lock classes from Java SE Development Kit 8 and kept it mostly without changes, and then afterwards we landed commits that actually implement the concepts we will discuss in this section.

We will focus on what changes were done on the original implementation presenting pseudo-code to illustrate how they work (more details on our repository's commits history).

### *Keeping ownedLocks list updated*

```
Lock currentLock;
DEFINE_PER_LOCK(Thread, owner);

DEFINE_PER_THREAD(List<Lock>, ownedLocks);

setExclusiveOwner(Thread t) {
    currentLock.owner = t;
    if (currentLock.owner == null) {
        unregisterOwnedLock();
    } else {
        registerOwnedLock();
    }
}

registerOwnedLock() {
    ownedLocks.push(currentLock);
}
```

```
unregisterOwnedLock() {  
    ownedLocks.remove(currentLock);  
}
```

Following the first part of the protocol, we must keep a list of locks each thread owns. This list is thread-local: the main advantage is that we automatically get freedom from interference as each thread will only manage its own list. In original implementation, `setExclusiveOwner` is used whenever a thread successfully acquire a lock or release it. On our version, we modified it to intercept it and also update the list of owned locks accordingly as follows: whenever the call resets the owner it means there was a release, so we unregister that lock and removes it from the list of owned locks of the current thread; furthermore, we do the opposite when the owner is not null, as it means that the thread has owned that lock and it should update its own list of owned locks to add that particular entry.

*Detecting deadlock and throw exception*

```
park() {  
    Thread other = currentLock.owner;  
    if (isAnyOwnedLockDesiredBy(other)) {  
        clearOwnedLocksByCurrentThread();  
        throw new DeadlockException();  
    }  
    LockSupport.park(this);  
}
```

```
isAnyOwnedLockDesiredBy(Thread t) {  
    forEach(lock : ownedLocks) {  
        if (lock.queue.contains(t)) {  
            return True;  
        }  
    }  
    return False;  
}
```

```
clearOwnedLocksByCurrentThread() {  
    ownedLocks.clear();  
}
```

When a thread attempts to acquire a lock and this lock is already owned, we deploy the deadlock check right before the thread get parked. It starts by checking which thread is the owner of this particular lock. Then the current thread checks on its list of owned locks whether any of



them has the owner of the desired lock waiting for it. If positive, then we have a circular wait: this thread wants another lock, but the owner of that lock is waiting for a lock this thread owns. In that case, we must throw a deadlock exception.

The changes described previously were part the first version of the algorithm and we minimally changed `ReentrantLock` to support it. That version was used on our usability evaluation (Section 5.2). The only problem of that algorithm is it couldn't guarantee the exception would be thrown in both threads involved.

Later, we decided to tackle that problem and make it guarantee that both threads would throw the exception. We will present again some functions to illustrate how they changed in this last version.

*Changes on `ReentrantLock`: throw deadlock exception on both threads*

```
DEFINE_PER_LOCK(List<Thread>, taintedThreads);

park() {
    Thread other = currentLock.owner;
    List<Lock> desiredLocks;
    desiredLocks = getMyOwnedLocksDesiredBy(other);
    if (!desiredLocks.empty()) {
        foreach(lock : desiredLocks) {
            lock.taintedThreads.add(other);
        }
        clearOwnedLocksByCurrentThread();
        throw new DeadlockException();
    }
    LockSupport.park(this);
}

getMyOwnedLocksDesiredBy(Thread t) {
    List<Lock> desiredLocks = [];
    foreach(lock : ownedLocks) {
        if (lock.queue.contains(t)) {
            desiredLocks.add(lock);
        }
    }
    return desiredLocks;
}

void setExclusiveOwner(Thread t) {
```

```
currentLock.owner = t;
if (currentLock.owner == null) {
    unregisterOwnedLock();
} else {
    registerOwnedLock();
    if (taintedThreads.contains(t)) {
        clearOwnedLocksByCurrentThread();
        throw new DeadlockException();
    }
}
}
```

We expanded the algorithm by adding a list of threads on each lock object containing threads that just went into a deadlock and should also throw exception as soon as possible. Note although this particular list is not thread-local, we guaranteed freedom from interference, as we applied a policy on that list of tainted threads to only permit read/write operations for the owner of the lock.

When a deadlock was detected on the previous version, the following case was possible: one thread throws an exception and releases all its locks, then the second thread would finally acquire its desired lock which should be free after the first thread released its locks.

However, it is important to note that when the first thread is about to throw an exception, the second thread is already stuck waiting for the first thread.

Also, the first thread was still the owner of the current lock object allowing it to modify the list of tainted threads inside that lock. Then it could update list of tainted threads by adding the second thread on it. Finally, by the time the second thread acquires this lock which should be free now, it should also throw an exception.

The only disadvantage of this final solution was the possibility of leaking `Thread` references inside tainted threads lists for each lock. It can only happen when both threads simultaneously detect the deadlock and both throw exceptions. In that case, they would have added the opposite thread inside their own lock's taintedThreads list, but afterwards both threads would stop and none of them would attempt to acquire the opposite lock again. We've minimized the effect of this leak by adding a policy to remove non-active thread references from tainted threads list every time any update is done on it. In practice we expect other threads would eventually clean up these thread references. We also provide code that implements this last policy on our repository [GITHUB \(2015\)](#).

# 5

## Evaluation

### 5.1 Introduction

In this chapter we present an evaluation of our approach. Our evaluation comprises two parts: (i) a usability evaluation involving two experiments with two groups of students (Section 5.2); and (ii) a preliminary analysis of the performance overhead of our approach (Section 5.3). The exact input, instructions, and any additional document we have used in this section are available in appendices or in our repository ([GITHUB, 2015](#)).

### 5.2 Usability Evaluation

We ran empirical evaluation to measure the efficiency of deadlock exceptions with regard to problem solving speed and accuracy. We defined two research questions for this evaluation:

- **RQ1.** Is the time spent to identify the bug reduced using our implementation?
- **RQ2.** Is the accuracy in the identification of the causes of a deadlock bug improved for developers using our approach?

The metric we watched to answer the first question was the time (in seconds) to finish each question in the test. For the second question, the metric we used was the number of correct answers.

We evaluated students answers under different criteria (see Table 5.1), where each one received a score that was either *0*, *0.5* or *1*. *0* means the criteria was not met at all, *0.5* means it was partially met, and *1* means it was complete. Whenever  $(A - B) + C \geq 1.5$  was true, we defined it as a correct answer. This equation means that an answer was correct when: (i) the bug was described as deadlock; (ii) the answer did not assert there was also another type of bug; (iii) at least one of the methods involved were identified in the answer.

We didn't use criteria *D* and *E* because, in our questions statements, we didn't make it clear that we wanted description of all locks involved in the deadlock. Also, our deadlock

**Table 5.1:** Criteria to evaluate students answers

Criteria	Description
A	Correctly classified problem as deadlock
B	Classified problem as different from deadlock
C	Correctly identified method calls involved in the deadlock
D	Correctly identified locks involved in the deadlock
E	Pointed unrelated methods as part of the deadlock

implementation at that time did not guarantee exception on both threads, thus if the student could identify at least one lock it was good enough.

### 5.2.1 Time Analysis

We defined the following hypothesis to answer **RQ1**:

$$H_0 : \mu_{TimeLockA} \geq \mu_{TimeLockB} \quad (5.1)$$

$$H_1 : \mu_{TimeLockA} < \mu_{TimeLockB} \quad (5.2)$$

In order to prevent *bias*, we needed to control a few factors during the experiment execution. The first factor was the selection of subjects to participate on this experiment, as different background knowledge could potentially influence chosen metrics. The second factor we had to control was the complexity of programs that each subject would have to look into. Complexity was interpreted as a direct relation to the amount of files in the program, number of threads and number of locks to analyze; as we've assumed that easier programs could have little or no benefit from deadlock exceptions, we wanted to have one program that we considered easy to identify the problem and another that was more complex and composed by many files and classes, reflecting a more realistic case. We provided implementations of each program using either *LockA* or *LockB*: the two possible treatments that we wanted to compare.

We decided to use Latin Square Design (BOX; HUNTER; HUNTER, 2005) to control these two factors mentioned earlier: subjects and program complexity factors. Since we had N subjects, 2 programs and 2 possible treatments, we disposed subjects in rows and programs in columns of latin squares, randomly assigning in each cell of the square a treatment that could be *LockA* or *LockB*, but also guaranteeing that for any given row or column in this square, each treatment appears only once (see Table 5.2). Consequently, we have replication, local control and randomization which are the three principles of experiment design (BOX; HUNTER; HUNTER, 2005).

We wrote two programs with different complexity which were presented in the same order for all subjects. The first program, known as *Bank*, contained 4 classes spread over 4 files, 3 threads, 3 explicit locks, and 82 lines of code in average. The second program, known as *Eclipse*

**Table 5.2:** Latin Square design

	Program 1	Program 2
Subject 1	LockA	LockB
Subject 2	LockB	LockA

had 15 classes spread in 11 files, 4 threads, 5 explicit locks, and 40 lines of code in average. We expected the first program to be easier to identify the deadlock because it contained fewer classes and files. Each program could use either *LockA* or *LockB* but we randomly assigned a group to each student so that if they fall into group A, they would start with *LockA* in the first question, but change to *LockB* on the second question; or if they fall in group B, they they would start with *LockB* and switch to *LockA* in the second question. We randomly paired subjects in tuples composed of one subject in group A and another subject of group B, then we created latin squares for each one of these pairs, where any remainders were discarded.

We repeated this experiment twice. The two experiments had similar setups but differed in terms of the subjects. For the first experiment, the subjects comprised a group of third-year undergraduate students who underwent an 18-hour concurrent programming course. The course included a number of programming assignments. The experiment was conducted as a test for the course. The subjects of the second experiment were graduate students enrolled in master's degree or PhD program attending a 40-hour Parallel Programming course with a focus on algorithms and data structures. They had classes about advanced concepts of parallel programming and had practical exercises, including implementing a number of different locking approaches. The participants in the second experiment were all volunteers and were not required to take part in it. Also, for both experiments, the assignments were the same. It asked students to identify any problems they could with the provided programs.

### 5.2.1.1 Metrics Collection

All students started the experiment with program *Bank*. When they finished, they received the second program, *Eclipse*. When a student finished one of the programs, we set a timestamp on it, or in a few cases, we allowed students to do it by themselves but we had to double check the values and fix in case there was any error. The timestamp was written based on a chronometer visible to everyone in the laboratory. For the first group (undergraduate students), we allowed 90 minutes per program. For the second group (graduate students), we allowed 60 minutes per program.

Each one should start the experiment with the first question containing *Program 1* and once they finish to provide an answer, they should request for the second question. At that point, we collect and place a timestamp in their answer. Once they finish the second question containing *Program 2*, then they should again give us a notice so we can leave a new timestamp. Later we used these timestamps to measure how long they took to finish each question. We have started

this experiment with a time limit for each question of 60 minutes each. However, during the first experiment, students complained and we realized it would not be sufficient for all students so we expanded to 90 minutes.

The timestamp was written by students conducting the experiment based on a counter we projected on the laboratory wall in real time. In a few circumstances the subject could write the timestamp when they finished, but we have double checked the value at the time we collected their answer and we fixed the value in case there was a mistake.

### 5.2.1.2 Experiment Operation

We executed this experiment in two different days. In the first day we did it with undergraduate students in replacement of their default exam, thus their participation was obligatory. We disclaimed they could optionally leave a comment in the test saying they did not want to take part in this research and we would not use their data. Fortunately no one chose to not participate. In the second day, we executed it with volunteer students after their Parallel Programming class was over. In total, 31 students participated on the first day and 16 students participated on the second day. We had to discard 2 students from second day experiment data because they both arrived late and they had to leave early.

We had two stacks of tests for each model type (either A or B) to equally distribute them among students in the beginning of the test. The entire test had two questions but they were not in the same document, so they would not see the second question until the first was completed. Students who started with a question of a certain type should receive the next question of the same type when they finish. That way we could easily guarantee if they started with `LOCKA`, they would switch to `LOCKB` on second question and vice-versa.

We distributed the initial question to all students and asked them to not start solving it yet. We read the setup instructions and demonstrated it on the projected screen which was visible to everyone, asking them to repeat the instructions at the same time. In case they found any problem, they should let us know and we would try to solve it immediately. Once everyone had the initial setup done, we projected a chronometer in the screen and asked them to start at that time.

During the test, students should raise a hand to sign when they finish the each question. Then we would write the timestamp we saw at the projected screen on their test and give the next question if available. When subjects were close to time limit for that question, we would let them know 10 minutes in advance and then collect the test once the time limit was reached.

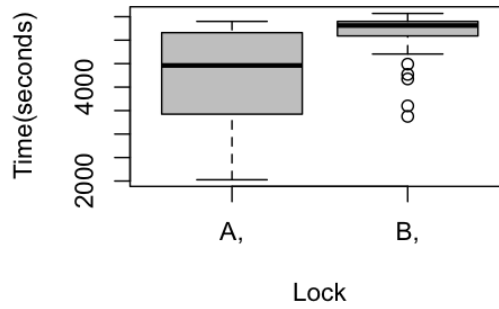
### 5.2.1.3 Results

Time analysis was conducted with R Statistical Software using the inputs extracted from each day. We used the linear model described in Figure 5.1 that considers the effect of different factors on the response variable as proposed by other authors ([ACCIOLY, 2012](#)) ([SANCHEZ,](#)

$$Y_{lijk} = \mu + \tau_l + \tau\alpha_{li} + \beta_j + \gamma_k + \tau\gamma_{lk} + \varepsilon_{lijk}$$

- $Y_{lijk}$  - response of  $l_{th}$  replica,  $i_{th}$  student,  $j_{th}$  program,  $k_{th}$  lock  
 $\tau_l$  - effect of  $l_{th}$  replica  
 $\tau\alpha_{li}$  - effect of interaction between  $l_{th}$  replica and  $i_{th}$  student  
 $\beta_j$  - effect of  $j_{th}$  program  
 $\gamma_k$  - effect of  $k_{th}$  lock  
 $\tau\gamma_{lk}$  - effect of interaction between  $l_{th}$  replica and  $k_{th}$  lock  
 $\varepsilon_{lijk}$  - random error

**Figure 5.1:** Regression model.



**Figure 5.2:** First experiment box-plot graphic.

2004).

Initially, we've plotted box-plot graphic (Figure 5.2) and calculated the average time spent to solve tasks for both groups (Table 5.3). They show that students using `LockB` took more time to complete the tasks. On Figure 5.2, we can see time spent for both groups stop at nearly the same point, and we believe the reason behind it is that we enforced a time limit for each question. If there was no such time limit on each question, we believe that `LockB` time spent range would be significantly wider.

Then we run the Box-Cox transformation - a power transformation - to reduce anomalies such as non-additivity and non-normality. First we verify if the transformation is needed, obtaining the curve in the left of Figure 5.3. Since the value of `lambda` at the maximum point in the curve is not approximately 1, we should apply it. In order to apply the transformation,  $Y_{lijk}$  should be powered to that  $\lambda$  on our regression model. Now, using the transformed model, we obtain the curve shown in the right of Figure 5.3.

In the next step we ran Tukey Test of Additivity to check whether effect model was

	Time Spent (seconds)
LockA	4227.800
LockB	5086.367

**Table 5.3:** First experiment's average time spent



**Figure 5.3:** First experiment: before and after box-cox transformation ( $\lambda = 5$ ).

$H_0$  : The model is additive

$H_1$  :  $H_0$  is *false*

**Figure 5.4:** Tukey Test of Additivity hypothesis

additive. Interaction between factors displayed on rows and columns of each latin square won't affect significantly the response only when the model is additive (BOX; HUNTER; HUNTER, 2005), so we must verify it (Figure 5.4).

When the test was executed, we obtained p-value of 0.514, which means we cannot reject  $H_0$ . Consequently the model was considered to be additive.

Finally, we ran the ANOVA (ANalysis Of VAriance) test which compares the effect of treatments on the response variable, providing an approximated p-value for every associated factor (see Table 5.4). When a variable has p-value  $< 0.05$ , it means that factor was statistically significant to the response. It shows that **lock factor was the most significant to the response**, allowing us to reject our null hypothesis defined in Equation 10.

Now we will show the results collected by the second experiment with graduate students. They were exposed to the same set of problems in a different day, but as explained before, they only had a time limit of 1 hour per question.

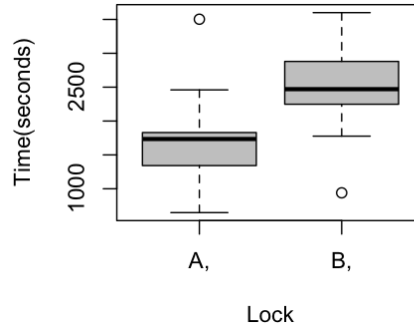
When we analyze the box-plot for the second group (Image 5.5) and their average time spent (Table 5.5), we can see there was a clear improvement on the time for students using *LockA*.

Moving forward with the analysis, we check if a box-cox transformation is needed. Since

**Table 5.4:** Undergraduate students experiment ANOVA results.

	Df	Sum Sq	Mean Sq	F value	<i>p-value</i>	
Replica	14	3.8633e+37	2.7595e+36	1.6553	0.1784197	
Program	1	4.1460e+36	4.1460e+36	2.4869	0.1371197	
Lock	1	3.9489e+37	3.9489e+37	23.6873	0.0002492	***
Replica:Student	15	4.1013e+37	2.7342e+36	1.6401	0.1808595	
Replica:Lock	14	2.4033e+37	1.7166e+36	1.0297	0.4785520	
Residuals	14	2.3340e+37	1.6671e+36			





**Figure 5.5:** Second experiment box-plot graphic.

	Time Spent (seconds)
LockA	1737.714
LockB	2507.714

**Table 5.5:** Second experiment's average time spent

the value is not approximately 1, we apply the power transformation the same way we did with the first experiment, but with the corresponding lambda value.

Finally, running ANOVA, we can see that the type of lock was the most significant factor for the response time, as shown in Table 5.6. Again, we can reject the null hypothesis.

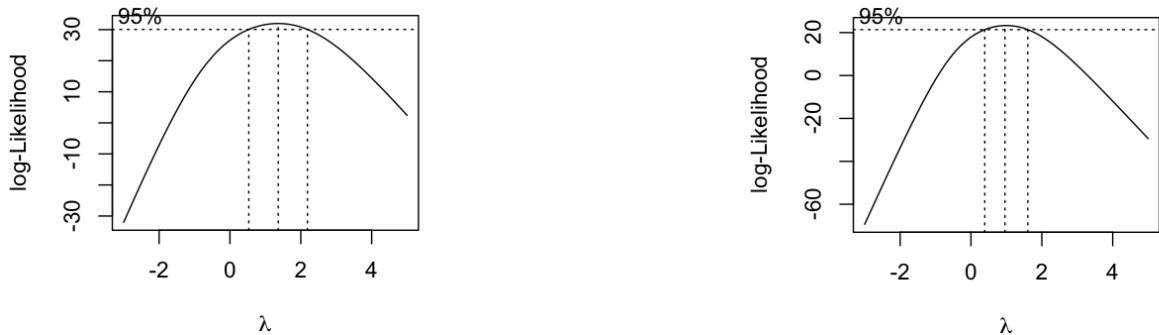
### 5.2.2 Accuracy Analysis

We used the number of correct answers using each lock to measure accuracy, so we defined the following hypothesis to answer **RQ2**.

$$H_0 : \mu_{CorrectAnswersLockA} \leq \mu_{CorrectAnswersLockB} \quad (5.3)$$

$$H_1 : \mu_{CorrectAnswersLockA} > \mu_{CorrectAnswersLockB} \quad (5.4)$$

Once we collected all answers, we manually evaluated each one of them according to the criterias established previously, where each criteria had an associated value between 0 and 1. Then we've ran a script that evaluates the equation we defined before to classify whether an



**Figure 5.6:** Before and after box-cox transformation ( $\lambda = 1.3636$ ).

**Table 5.6:** Graduate students ANOVA results.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
replica	6	2576883250	429480542	14.1891	0.0025793	**
program	1	6875586	6875586	0.2272	0.6505035	
lock	1	1958179433	1958179433	64.6938	0.0001975	***
replica:student	7	2328154077	332593440	10.9881	0.0047601	**
replica:lock	6	823830276	137305046	4.5362	0.0441188	*
Residuals	6	181610625	30268438			

**Table 5.7:** Undergraduate students answers accuracy

	Correct	Incorrect
LockA	29	2
LockB	16	15

answer was correct or not. Grouping the results in tables, we have Table 5.7 and Table 5.8.

To compare the accuracy of the subjects using Java's regular `ReentrantLock` and our modified implementation, we employed a two-tailed Fisher's Exact Test [AGRESTI \(1992\)](#). We could not use ANOVA because the data for accuracy is categorical (Correct vs. Incorrect) instead of numerical. Applying Fisher's exact test on data from Table 5.7 and Table 5.8, we can see that undergraduate students results presented a two-tailed P value equals 0.0004: the association between rows (groups) and columns (outcomes) was considered to be extremely statistically significant; consequently, it suggests that an improvement on accuracy occurred due to the use of the proposed approach, where it had 81.25% more correct answers. However graduate students results only had 30% more correct answers with a two-tailed P value equals 0.3259, which does not represent statistically significant evidence.

Although we cannot draw strong conclusions regarding improved accuracy, we found some interesting behavior. Some students in the second group were greatly experienced on concurrent programming and they knew how to efficiently find a deadlock using the tools available in their IDE of choice, thus being able to finish the tasks really quickly for both problems. This observation allows us to hypothesize that deadlock exceptions are more helpful for less experienced programmers, but we leave investigation of this matter for future work.

**Table 5.8:** Graduate students answers accuracy

	Correct	Incorrect
LockA	13	1
LockB	10	4

### 5.2.3 Discussion

We can see in our results that both groups of students have improved their time to solve the problem when they had the lock with deadlock exception. Also, on the first group, we have found statistically significant evidence that it improved answers accuracy, but not for the second group.

We cannot draw conclusions regarding the improved accuracy on the second group, but we can bring up some relevant aspects we've observed and make a few hypothesis. Some students in the second group were greatly experienced on concurrent programming and they knew how to efficiently find a deadlock using the tools available in Eclipse. Thus, they have finished the exercise really quickly for both problems, knowing exactly which points in the code were involved in the deadlock. So we can see that deadlock exceptions are more helpful for unexperienced programmers in general, and it's possible that even if we had a bigger sample for the second group, we would still not see a significant difference that would indicate deadlock exceptions improved their accuracy.

However, we believe that the benefits of deadlock exceptions are beyond helping unexperienced programmers to find deadlocks more precisely. Experienced programmers would still benefit in many cases where the deadlock is not as obvious as in the exercise we've presented. For example, in a more realistic situation, a deadlock can happen in a background thread that doesn't really affect the program execution overall but make the execution lacking some expected behavior. Furthermore, in non-interactive systems where they are only running in background, is nearly impossible to know when there's a problem unless this software is monitored constantly which is very time consuming or the system produces output constantly that is affected by a potential deadlock. If we have a deadlock exception, we can either prepare and handle this exception on the code level, or just have this signal from output that would help developers to fix it later.

### 5.2.4 Threats To Internal Validity

In this experiment, we've collected evidence on how the presence of deadlock exception affects student ability to identify deadlocks accurately. However, we must raise a few considerations regarding the validity of our results.

#### 5.2.4.1 Time Measurement

Since we wanted to run the experiment in a homogenous environment, we've decided to run it in a laboratory in Federal University of Pernambuco, and we've provided links to download the exercise and a few instructions explaining how to deploy it. We wanted to make it as easy as possible and before we've started the test, we gave a small presentation reproducing step by step the instructions that would be described on each exercise, so everyone could follow up and

make the setup at the same time. Once everyone was done, we've started to count the time and allowed them to run the programs and start debugging. However, this procedure was not enough: there was a few students (approximately 3 in total) who did the setup differently and could not execute the program; therefore, they've lost a few minutes until we've fixed that for them. Since they've lost only a few minutes, we have still counted them as part of the experiment and did not discount the time.

Furthermore, some students arrived at the test more than 10 minutes late. We've allowed them to join, but some of the remaining computers in the laboratory had issues like they were not logging in or the mouse was not working. We've lost a few minutes to make them work or find a new computer and once each of them did the setup, we've started to count their time individually.

Whenever a student finished a given question, if the time was below the time limit they had available, we have marked the current timestamp on each student's name in the whiteboard. Each entry inserted was already sorted by time, so we easily tracked whether each student was close to the second question's time limit. It would have been better to do this automatically rather than doing manually, so we could potentially reduce overhead of these timestamp operations and increase their precision.

Also, we believe that our imposed time limit have limited more drastically the time ranges on the first group because they spent more time on each question. Also the fact it was an exam for them may have delayed the time to answer because they were more careful. We have observed during the experiment that many students wrote their answers but they were reluctant to ask for the next question because they still have plenty of time left and they wanted to make sure it was correct. We did not observe such behavior with the second group of students and we believe it is because they did not have the same pressure to deliver correct results as the first group had.

#### 5.2.4.2 Exercises

We understand that the two questions we've used to evaluate the students are far easier than what most software engineers have to deal with in the real world. However we could not use any real world issue because it would easily take the time limit of the experiment for each bug.

On the other hand, we've created two questions based on real world bugs that we have found while searching for deadlock bugs in open source repositories. Each question had a particular level of granularity, where one should be easier to find a bug because of the less amount of code to examine and another that should be more difficult because of the reasonable amount of different files to look at.

Some researchers actually believe that empiric evaluations should not be limited to real projects. Buse claims there are benefits of using non-real artifacts ([BUSE; SADOWSKI; WEIMER, 2011](#)) because it's easier for researchers to translate research questions into successful experiments as it allows a greater control over confounding factors. Otherwise, it would be necessary to turn all participants familiar with the codebase of a real and complex system before

even starting the experiment.

### 5.2.5 Threats to External Validity

Let's consider a few conditions that might limit the generalization power of our findings in this experiment.

#### 5.2.5.1 Students

Each student which participated in this experiment had a different background. What we did to minimize the differences was to select groups where students had at least basic experience in concurrent programming and they should be familiarized with the types of bugs such codes can have: the first group of students with undergraduate students attended the class Paradigms of Computacional Languages where deadlocks are covered in classes and exercises; the second group with graduate students attended the class Parallel Programming which covered concurrent programming in low level detail in classes and exercises, including deadlock detection.

Some studies have already addressed the problem of drawing conclusions made with students but some suggest that using students as subjects is as good as using industry professionals ([STARON, 2007](#)). Runes ran an experiment which shows that there's not much significant differences between undergraduate, graduate and industry professionals, with the exception that undergraduate students often take more time to complete the tasks ([RUNESON, 2003](#)).

## 5.3 Performance Analysis

We conducted a preliminary set of experiments to analyze the overhead of our approach. We compared the our implementation of ReentrantLock with deadlock exceptions, the original ReentrantLock and Eclipse's `OrderedLock` ([ORDEREDLOCK, 2014](#)) implementation in a synthetic benchmark.

`OrderedLock` is a deadlock-safe implementation of a lock which relies on Eclipse's code architecture. It is similar our approach in the sense that it attempts to detect deadlocks at runtime. However, it aims to be general, detecting  $N$ -thread deadlocks without much concern for performance. When a deadlock happens, it releases all locks by a given thread and suspends it, allowing other threads to proceed. Later, the suspended thread will acquire such locks again. Since it allows threads to temporarily give up all its owned locks, it does not guarantee exclusive access policy for *critical zones* as locks usually do.

In order to use it in our evaluation, as it deeply relies on Eclipse's code architecture to function, we had to perform some small code changes, removing only Eclipse-specific bits that did not affect the core functionality of `OrderedLock`. The source code for these lock implementations is available elsewhere ([GITHUB, 2015](#)).

**Table 5.9:** Benchmark time measurements (in seconds)

# Threads	ReentrantLock	ReentrantLock Modified	OrderedLock
10	0.084184	0.105729	0.159503
50	0.089094	0.136507	1.094718
100	0.090978	0.159541	3.395974
200	0.131739	0.194075	11.258714

We developed a synthetic benchmark that creates  $N$  threads that perform additions to ten integer counters where each increment in a counter is protected by explicit locks. Each thread would have to increment its corresponding counter 1000 times before finishing its execution and the counters were evenly distributed across the threads. Therefore, each counter will have exactly  $(N / 10)$  threads doing increments on it and higher values of  $N$  result in higher contention, that is, more threads will compete against each other for a particular counter. Since each thread in the benchmark never acquires more than one lock at the same time, deadlocks cannot occur. We emphasize that this setup is very conservative, since every operation that each thread performs requires locking. Thus, the obtained overhead will be a worst-case estimate and thus much higher than one would encounter in a real-world application (LOZI et al., 2012).

In this preliminary evaluation, we have conducted measurements for values of  $N$  equal to 10, 50, 100, and 200. The measurements were made on an Intel Core™ i7 3632QM Processor (6Mb Cache, 2.2GHz) running Ubuntu 12.04.4 LTS and each cell in Table 5.9 is the average of 50 executions (preceded by 20 executions that served as a warm-up).

The difference of results between our implementation and the original `ReentrantLock` gives a range of increased time from about 50% to 90%. Meanwhile, `OrderedLock` performed a lot worse, reaching a 8446.3% increase in time for the worst case.

To get a rough estimate of the impact that this overhead would have on actual application execution time, we analyzed the results obtained by LOZI et al. (2012). The authors profiled 19 real-world applications and small benchmarks in order to measure the time these systems spend on their critical sections. Worst-case results ranged between 0.3% and 92.7%. If we consider the average time spent on the critical sections of 12 of these systems, the impact of our approach on the overall execution time would be less than 6% in the worst case. The remaining cases are extreme, in the sense that these systems spend more time in their critical sections than out of them (LOZI et al., 2012).

# 6

## Concluding Remarks

In this work, we have initially discussed how challenging it is to handle deadlocks and resolve them in real world software, and present many previous approaches that have tried to identify or solve deadlocks with either static analysis, dynamic analysis or a mix of the two approaches.

Given that it's very costly to detect deadlocks in its general form, we've investigated which kind of deadlock is the most popular by looking at actual bug reports in relevant open source projects and classifying each deadlock manually. We've confirmed a previous study claim that classic deadlocks between two threads and two resources are the most frequent case of deadlock. This claim allowed us to focus on this specific case instead and minimally modify Java's *ReentrantLock* to provide a lightweight version of a lock that detects this kind of deadlock in runtime.

However, differently from previous approaches, we provided a runtime exception which should be raised on both threads involved in a TTTL deadlock when our Java's *ReentrantLock* implementation is used, as we believe that developers would find bugs faster if deadlocks were just exceptions.

Our usability evaluation with students shows that the presence of deadlock exception indeed reduce the amount of time spent to identify a bug in a code they were not familiar with. It also shows that less experienced students could identify the source of the deadlock correctly more easily.

### 6.1 Research Contributions

We can summarize our main contributions as follows:

1. Investigated deadlock bug reports in open source projects and confirmed a previous study claim that TTTL deadlocks are the most popular type of deadlock (92.07% of all resource deadlocks we've identified).

2. Implemented a lightweight version of `ReentrantLock` that detects TTTL deadlocks and throw runtime exception on both threads.
3. We measured implementation performance overhead with a very conservative benchmark and estimated our cost to be less than 6% for worse case on real world applications.
4. Presented a sketch of a proof to demonstrate why TTTL deadlock detection protocol is correct.
5. Empirical evaluation shows that deadlock exceptions help to reduce time spent identifying deadlock bugs.
6. Found evidence that allows us to hypothesize that deadlock exceptions are more helpful for less experienced developers.

## 6.2 Future Work

We finish this work with some proposals for future work based on some problems we found during this research or opportunities we identified.

1. Significant amount of deadlock bugs we saw during bug reports study used synchronized blocks rather than explicit locks such as `ReentrantLock`. In order to reach more Java developers, implementing deadlock detection on synchronized blocks should be investigated.
2. Investigate if it is possible to add more bug context data into `DeadlockException` while still keeping overhead low. It should be easier to understand more accurately details about deadlock in the exception message there is clear information about which threads were affected and which locks.
3. We collected data about number of comments and time spent to solve tasks for bug reports we found, but we did not use in this research. It should help to investigate how difficult deadlock bugs are in comparison to non-deadlock bugs. Are developers discussing more on them, or are they spending more time to close tasks?
4. Run empirical experiments with other groups of students grouped by experience in concurrent programming development to confirm whether deadlock exception benefits we saw are indeed more significant for less experienced developers.
5. Search for deadlock bugs in projects written with other popular languages and investigate if there are significant differences for TTTL deadlocks distribution.



# References

- ACCIOLY, P. **Comparing Different Testing Strategies for Software Product Lines**. 2012. Dissertação (Mestrado em Ciência da Computação) — UFPE.
- AGRESTI, A. A survey of exact inference for contingency tables. **Statistical Science**, [S.l.], p.131–153, 1992.
- AIMONETTI, M. **Go Bootcamp**. URL: <http://www.golangbootcamp.com/book/concurrency>.
- BERGER, E. D. et al. Grace: safe multithreaded programming for c/c++. In: ACM SIGPLAN NOTICES. **Anais...** [S.l.: s.n.], 2009. v.44, n.10, p.81–96.
- BISWAS, S. et al. Efficient, Software-Only Data Race Exceptions. , [S.l.], 2015.
- BOX, G. E.; HUNTER, J. S.; HUNTER, W. G. Statistics for experimenters: design, innovation, and discovery. **AMC**, [S.l.], v.10, p.12, 2005.
- BOYAPATI, C.; LEE, R.; RINARD, M. Ownership types for safe programming: preventing data races and deadlocks. In: ACM SIGPLAN NOTICES. **Anais...** [S.l.: s.n.], 2002. v.37, n.11, p.211–230.
- BUSE, R. P.; SADOWSKI, C.; WEIMER, W. Benefits and barriers of user evaluation in software engineering research. **ACM SIGPLAN Notices**, [S.l.], v.46, n.10, p.643–656, 2011.
- CAI, Y.; CHAN, W. MagicFuzzer: scalable deadlock detection for large-scale applications. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34. **Proceedings...** [S.l.: s.n.], 2012. p.606–616.
- CHROMIUM. URL: <http://code.google.com/chromium>.
- DA LUO, Z.; DAS, R.; QI, Y. Multicore SDK: a practical and efficient deadlock detector for real-world applications. In: SOFTWARE TESTING, VERIFICATION AND VALIDATION (ICST), 2011 IEEE FOURTH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2011. p.309–318.
- ENGLER, D.; ASHCRAFT, K. RacerX: effective, static detection of race conditions and deadlocks. In: ACM SIGOPS OPERATING SYSTEMS REVIEW. **Anais...** [S.l.: s.n.], 2003. v.37, n.5, p.237–252.
- FARCHI, E.; TZOREF, R. **ConcurrentTesting - Advanced Testing for Multi-Threaded Java Applications**. URL: <http://www.golangbootcamp.com/book/concurrency>.
- FIREFOX. URL: <http://www.mozilla.org/firefox>.
- FLANAGAN, C. et al. PLDI 2002: extended static checking for java. **ACM Sigplan Notices**, [S.l.], v.48, n.4S, p.22–33, 2013.
- GITHUB. URL: <https://github.com/rafaelbrandao/java-lock-deadlock-exception>.

- GRECHANIK, M. et al. Preventing database deadlocks in applications. In: JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.356–366.
- HANSEN, P. B. The programming language concurrent pascal. **Software Engineering, IEEE Transactions on**, [S.l.], n.2, p.199–207, 1975.
- HAVELUND, K.; PRESSBURGER, T. Model checking java programs using java pathfinder. **International Journal on Software Tools for Technology Transfer**, [S.l.], v.2, n.4, p.366–381, 2000.
- KNAPP, E. Deadlock detection in distributed databases. **ACM Computing Surveys (CSUR)**, [S.l.], v.19, n.4, p.303–328, 1987.
- LOZI, J.-P. et al. Remote Core Locking: migrating critical-section execution to improve the performance of multithreaded applications. In: USENIX ANNUAL TECHNICAL CONFERENCE. **Anais...** [S.l.: s.n.], 2012. p.65–76.
- LU, S. et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ACM SIGPLAN NOTICES. **Anais...** [S.l.: s.n.], 2008. v.43, n.3, p.329–339.
- LUK, C.-K. et al. Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN NOTICES. **Anais...** [S.l.: s.n.], 2005. v.40, n.6, p.190–200.
- MARINO, D. et al. Detecting deadlock in programs with data-centric synchronization. In: SOFTWARE ENGINEERING (ICSE), 2013 35TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2013. p.322–331.
- MARLOW, S. **Parallel and Concurrent Programming in Haskell**: techniques for multicore and multithreaded programming. [S.l.]: " O'Reilly Media, Inc.", 2013.
- MYSQL. URL: <http://www.mysql.com>.
- ORDEREDLOCK. URL: <http://www.cct.lsu.edu/~rguidry/ecl31docs/api/org/eclipse/core/internal/jobs/OrderedLock.html>.
- PRAUN, C. von. **Detecting synchronization defects in multi-threaded object-oriented programs**. 2004. Tese (Doutorado em Ciência da Computação) — ETH. Zürich.
- PYE, I. Locks, deadlocks and abstractions: experiences with multi-threaded programming at cloudflare, inc. In: DSM'11, TMC'11, AGERE! 2011, AOPES'11, NEAT'11, & VMIL'11. **Proceedings...** [S.l.: s.n.], 2011. p.129–132.
- PYLA, H. K. Safe Concurrent Programming and Execution. , [S.l.], 2013.
- PYLA, H. K.; VARADARAJAN, S. Avoiding deadlock avoidance. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 19. **Proceedings...** [S.l.: s.n.], 2010. p.75–86.
- PYLA, H. K.; VARADARAJAN, S. Transparent runtime deadlock elimination. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 21. **Proceedings...** [S.l.: s.n.], 2012. p.477–478.

QIN, F. et al. Rx: treating bugs as allergies—a safe method to survive software failures. In: ACM SIGOPS OPERATING SYSTEMS REVIEW. **Anais...** [S.l.: s.n.], 2005. v.39, n.5, p.235–248.

RUNESON, P. Using students as experiment subjects—an analysis on graduate and freshmen student data. In: INTERNATIONAL CONFERENCE ON EMPIRICAL ASSESSMENT IN SOFTWARE ENGINEERING.—KEELE UNIVERSITY, UK, 7. **Proceedings...** [S.l.: s.n.], 2003. p.95–102.

SANCHEZ, I. **Latin Squares and Its Applications on Software Engineering**. 2004. Dissertação (Mestrado em Ciência da Computação) — UFPE.

SHANBHAG, V. K. Deadlock-detection in java-library using static-analysis. In: SOFTWARE ENGINEERING CONFERENCE, 2008. APSEC'08. 15TH ASIA-PACIFIC. **Anais...** [S.l.: s.n.], 2008. p.361–368.

SINGHAL, M. Deadlock detection in distributed systems. **Computer**, [S.l.], v.22, n.11, p.37–48, 1989.

STARON, M. Using Students as Subjects in Experiments—A Quantitative Analysis of the Influence of Experimentation on Students' Learning Proces. In: SOFTWARE ENGINEERING EDUCATION & TRAINING, 2007. CSEET'07. 20TH CONFERENCE ON. **Anais...** [S.l.: s.n.], 2007. p.221–228.

TANTER, É. et al. Altering Java semantics via bytecode manipulation. In: GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING. **Anais...** [S.l.: s.n.], 2002. p.283–298.

WILLIAMS, A.; THIES, W.; ERNST, M. D. Static deadlock detection for Java libraries. In: **ECOOP 2005-Object-Oriented Programming**. [S.l.]: Springer, 2005. p.602–629.



# **Appendix**





## Bug Reports Study Attachments

**Table A.1:** Sampled bug reports (part 1)

BUG	CATEGORY	# of THREADS	# of RESOURCES	TYPE
ECLIPSE-14109	A	2	2	locks/synchronized
ECLIPSE-154060	A	2	2	locks/synchronized
ECLIPSE-168028	A	2	2	locks/synchronized
ECLIPSE-187916	A	2	2	locks/synchronized
ECLIPSE-189054	A	2	2	locks/synchronized
ECLIPSE-194149	A	2	2	locks/synchronized
ECLIPSE-196919	A	2	2	synchronized
ECLIPSE-201254	A	2	2	locks/synchronized
ECLIPSE-216996	A	2	2	locks/synchronized
ECLIPSE-225891	A	2	2	synchronized
ECLIPSE-236664	A	2	2	locks/synchronized
ECLIPSE-236665	A	2	2	locks/synchronized
ECLIPSE-237596	A	2	2	locks/synchronized + static initialization
ECLIPSE-238304	A	2	2	locks/synchronized
ECLIPSE-239309	A	2	2	locks/synchronized
ECLIPSE-239672	A	2	2	locks/synchronized
ECLIPSE-241713	A	2	2	locks/synchronized
ECLIPSE-246284	A	2	2	locks/synchronized
ECLIPSE-248908	A	2	2	synchronized
ECLIPSE-260777	A	2	2	locks/synchronized
ECLIPSE-261476	A	2	2	locks/synchronized
ECLIPSE-261664	A	2	2	locks/synchronized
ECLIPSE-261910	A	2	2	locks/synchronized
ECLIPSE-268572	A	2	2	locks/synchronized
ECLIPSE-271840	A	2	2	locks/synchronized
ECLIPSE-288945	A	2	2	locks/synchronized
ECLIPSE-290550	A	2	2	synchronized
ECLIPSE-29212	A	2	2	locks/synchronized
ECLIPSE-294913	A	2	2	locks/synchronized
ECLIPSE-300596	A	2	2	locks/synchronized
ECLIPSE-306219	A	2	2	locks/synchronized
ECLIPSE-309781	A	1	1	locks/synchronized
ECLIPSE-312504	A	2	2	locks/synchronized
ECLIPSE-314176	A	2	2	locks/synchronized
ECLIPSE-31891	A	2	2	synchronized

**Table A.2:** Sampled bug reports (part 2)

BUG	CATEGORY	# of THREADS	# of RESOURCES	TYPE
ECLIPSE-320182	A	2	2	synchronized
ECLIPSE-320760	A	2	2	locks/synchronized
ECLIPSE-327247	A	1	1	locks/synchronized
ECLIPSE-329302	A	2	2	locks/synchronized
ECLIPSE-341849	A	2	2	locks/synchronized
ECLIPSE-343012	A	2	2	locks/synchronized
ECLIPSE-347183	A	1	1	locks
ECLIPSE-358396	A	3	2	locks/synchronized
ECLIPSE-359580	A	2	2	locks/synchronized + static initialization
ECLIPSE-359698	A	2	2	locks/synchronized
ECLIPSE-359851	A	2	2	locks/synchronized
ECLIPSE-361533	A	2	2	locks/synchronized + static initialization
ECLIPSE-374272	A	3	1	locks/synchronized
ECLIPSE-375601	A	2	2	locks/synchronized
ECLIPSE-381580	A	2	2	locks/synchronized
ECLIPSE-384204	A	2	2	locks/synchronized
ECLIPSE-389630	A	2	2	locks/synchronized
ECLIPSE-390963	A	3	2	locks/synchronized
ECLIPSE-39104	A	2	2	locks/synchronized
ECLIPSE-393277	A	2	2	locks/synchronized
ECLIPSE-395211	A	2	2	locks/synchronized
ECLIPSE-397442	A	3	2	locks/synchronized
ECLIPSE-40114	A	2	2	locks/synchronized
ECLIPSE-405511	A	2	2	locks/synchronized + output stream
ECLIPSE-407513	A	2	2	locks/synchronized
ECLIPSE-413730	A	2	2	locks/synchronized
ECLIPSE-416088	A	2	2	locks/synchronized
ECLIPSE-428335	A	2	2	locks/synchronized
ECLIPSE-47647	A	2	2	locks/synchronized
ECLIPSE-49708	A	2	2	locks/synchronized
ECLIPSE-50949	A	2	2	synchronized
ECLIPSE-58892	A	2	2	synchronized
ECLIPSE-60856	A	2	2	locks/synchronized
ECLIPSE-61179	A	2	2	locks/synchronized
ECLIPSE-62398	A	2	2	locks/synchronized
ECLIPSE-67181	A	2	2	locks/synchronized
ECLIPSE-70557	A	2	2	locks/synchronized
ECLIPSE-75115	A	2	2	locks/synchronized
ECLIPSE-79394	A	2	2	locks/synchronized
ECLIPSE-80610	A	2	2	locks/synchronized
ECLIPSE-86614	A	2	2	locks/synchronized
JDK-4101618	A	2	2	locks/synchronized
JDK-4192666	A	2	2	locks/synchronized
JDK-4258193	A	2	2	locks/synchronized
JDK-4265501	A	2	2	locks/synchronized
JDK-4294865	A	2	2	locks/synchronized
JDK-4826573	A	2	2	locks/synchronized
JDK-4828019	A	2	2	locks/synchronized
JDK-4852569	A	2	2	locks/synchronized
JDK-4944382	A	2	2	locks/synchronized
JDK-4965227	A	2	2	locks/synchronized
JDK-5006427	A	2	2	locks/synchronized



**Table A.3:** Sampled bug reports (part 3)

BUG	CATEGORY	# of THREADS	# of RESOURCES	TYPE
JDK-6190373	A	2	2	locks/synchronized
JDK-6296324	A	2	2	locks/synchronized
JDK-6349844	A	2	2	locks/synchronized
JDK-6362921	A	2	2	locks/synchronized
JDK-6370241	A	4	4	locks/synchronized
JDK-6373352	A	2	2	locks/synchronized
JDK-6440846	A	2	2	locks/synchronized
JDK-6814140	A	2	2	locks/synchronized
JDK-6834576	A	2	2	locks/synchronized
JDK-8026356	A	2	2	locks/synchronized
LUCENE-1200	A	2	2	synchronized
LUCENE-1358	A	2	2	synchronized
LUCENE-2783	A	2	2	locks/synchronized
LUCENE-3705	A	2	2	lock + synchronized
ECLIPSE-188320	B			wait/notify
ECLIPSE-199565	B	2		synchronized + notify/signal
ECLIPSE-199568	B	2		synchronized + notify/signal
ECLIPSE-220452	B	2	1	synchronized + join
ECLIPSE-226376	B	2	1	synchronized + join
ECLIPSE-283490	B			wait + message
ECLIPSE-295796	B			wait/notify
ECLIPSE-296570	B			wait/notify
ECLIPSE-305611	B			locks/synchronized + notify
ECLIPSE-311863	B			wait/notify
ECLIPSE-323680	B	2	2	synchronized + wait
ECLIPSE-330937	B	2	1	synchronized + wait
ECLIPSE-341196	B	2		wait/notify
ECLIPSE-357278	B	2	1	synchronized + wait
ECLIPSE-362238	B	2	1	locks/synchronized + wait
ECLIPSE-362288	B			wait/notify
ECLIPSE-385921	B			locks/synchronized + join
ECLIPSE-391046	B	2	1	locks/synchronized
ECLIPSE-411029	B	2		
ECLIPSE-426785	B	2		
ECLIPSE-52859	B			wait/notify
ECLIPSE-53466	B			wait/notify
ECLIPSE-71382	B	2		wait
JDK-4529917	B			wait/notify
JDK-4636269	B			wait/notify
JDK-4760364	B	2		wait/notify
JDK-4962516	B			wait/notify
JDK-6253009	B	2	1	lock + wait
LUCENE-1544	B	2		
LUCENE-3528	B			wait/notify
LUCENE-4071	B			wait/notify
LUCENE-5002	B	2	1	synchronized + wait/notify

**Table A.4:** Sampled bug reports (part 4)

BUG	CATEGORY	# of THREADS	# of RESOURCES	TYPE
ECLIPSE-148367	C			
ECLIPSE-164848	C			
ECLIPSE-168034	C			
ECLIPSE-186783	C			
ECLIPSE-223154	C			
ECLIPSE-231274	C			
ECLIPSE-234452	C			
ECLIPSE-234916	C			
ECLIPSE-246795	C			
ECLIPSE-246983	C			
ECLIPSE-249085	C			
ECLIPSE-261667	C			
ECLIPSE-328082	C			
ECLIPSE-361541	C			
ECLIPSE-43202	C			
JDK-4352221	C			
JDK-6225372	C			
JDK-6381164	C			
LUCENE-1210	C			
LUCENE-2095	C			
LUCENE-2216	C			
LUCENE-2820	C			
LUCENE-3348	C			
ECLIPSE-101944	D			
ECLIPSE-122144	D			
ECLIPSE-150198	D			
ECLIPSE-196290	D			locks/synchronized
ECLIPSE-203306	D			
ECLIPSE-212617	D			
ECLIPSE-218145	D			
ECLIPSE-236259	D			
ECLIPSE-243450	D			
ECLIPSE-246151	D			
ECLIPSE-259143	D			
ECLIPSE-263689	D			
ECLIPSE-26540	D			
ECLIPSE-267801	D			
ECLIPSE-270762	D			
ECLIPSE-282188	D			
ECLIPSE-289122	D			
ECLIPSE-289553	D			
ECLIPSE-291877	D	2		
ECLIPSE-296056	D	2		
ECLIPSE-299184	D			
ECLIPSE-300451	D			locks/synchronized + wait?
ECLIPSE-30140	D			
ECLIPSE-303670	D			
ECLIPSE-306914	D			
ECLIPSE-308120	D			
ECLIPSE-310007	D			

**Table A.5:** Sampled bug reports (part 5)

BUG	CATEGORY	# of THREADS	# of RESOURCES	TYPE
ECLIPSE-314923	D			
ECLIPSE-319168	D			
ECLIPSE-326274	D			
ECLIPSE-327472	D			
ECLIPSE-328697	D			
ECLIPSE-330505	D	2		
ECLIPSE-330964	D			
ECLIPSE-335235	D			
ECLIPSE-338965	D			
ECLIPSE-348284	D			
ECLIPSE-359199	D			
ECLIPSE-361302	D	2	1	locks/synchronized + wait?
ECLIPSE-366784	D			
ECLIPSE-37614	D			
ECLIPSE-390600	D			
ECLIPSE-42096	D			
ECLIPSE-428937	D			
ECLIPSE-434645	D			
ECLIPSE-43571	D			
ECLIPSE-4365	D			
ECLIPSE-50639	D			
ECLIPSE-59816	D			
ECLIPSE-83415	D			
JDK-4176738	D			
JDK-4366691	D			
JDK-4646297	D			
JDK-4762958	D			
JDK-4881251	D			
JDK-5083318	D	2		
JDK-5106770	D			
JDK-6303187	D			
JDK-6377483	D			
JDK-6386537	D			
JDK-6899078	D			
JDK-6949705	D			
JDK-7100952	D			
JDK-7105040	D			
JDK-7171163	D			
JDK-7174887	D			
LUCENE-1171	D			
LUCENE-159	D			
LUCENE-4147	D			

**Table A.6:** Bug report study: search queries and results

Project	Query Parameters	Results URL
Lucene	lucene-core, bug, status:"closed", assignee:all, "deadlock"	<a href="http://goo.gl/DhVI3t">http://goo.gl/DhVI3t</a>
Eclipse	summary:"deadlock", resolution:"fixed", status:"resolved"	<a href="http://goo.gl/qQnrEm">http://goo.gl/qQnrEm</a>
OpenJDK	project:"JDK", summary:"deadlock", issue type:"bug", status:"resolved", resolution:"fixed"	<a href="http://goo.gl/xYFfsO">http://goo.gl/xYFfsO</a>
Sampled		<a href="http://goo.gl/zNsIGz">http://goo.gl/zNsIGz</a>



# B

## Evaluation Chapter Attachments

### B.1 Time Analysis

#### *R Instructions*

```

expl.dat = read.table(file="/home/me/r_input.dat", header = T)
attach(expl.dat)

replica = factor(replica.)
student = factor(student.)
program = factor(program.)
lock = factor(lock.)

# Calculate average time for each lock

aggregate(time, by=list(lock), FUN=mean)

# Plot the box plot graphic using the response variable
# associated with the techniques

plot(time~lock,col="gray",xlab="Lock",ylab="Time(seconds)")

# Through the following command we adjust the effect model
# that will serve as basis for posterior analysis.

anova.ql<-aov(time~replica+student:replica+program+lock)

library(MASS)
bc <- boxcox(anova.ql,lambda = seq(-3, 5, 1/10))

```

```

# Calculate lambda for boxcox transformation

lambda <- bc$x[which.max(bc$y)]

# If lambda is not approximately 1, run the next command
# with [a] replaced by lambda's value found previously:
# anova.q1<-aov(time**[a]~replica+student:replica+program+lock)
# Then confirm that the new curve maximum is close to 1:
# bc <- boxcox(anova.q1,lambda = seq(-3, 5, 1/10))

TukeyNADD.QL.REP<-function(objeto1)
{
y1<-NULL
y2<-NULL
y1<- fitted(objeto1)
y2<- y1^2
objeto2<- aov(y2 ~ objeto1[13]$model[,2] +
objeto1[13]$model[,3]:objeto1[13]$model[,2]
+ objeto1[13]$model[,4]+ objeto1[13]$model[,5])
ynew <- resid(objeto1)
xnew <- resid(objeto2)
objeto3 <- lm(ynew ~ xnew)
M <- anova(objeto3)
MSN <- M[1,3]
MSErr <- M[2,2]/(objeto1[8]$df.residual-1)

F0 <- MSN/MSErr
p.val <- 1 - pf(F0, 1,objeto1[8]$df.residual-1)
p.val
}

# Return p-value for Tukey additivity test
TukeyNADD.QL.REP(anova.q1)

# Run ANOVA to see what factor was most significant
anova(anova.q1)

```

### *First Experiment Input*

```

replica, student, program, lock, time

```

---

1, 1, p1, A, 4967  
1, 1, p2, B, 5413  
1, 2, p1, B, 5400  
1, 2, p2, A, 5320  
2, 3, p1, A, 2340  
2, 3, p2, B, 5290  
2, 4, p1, B, 5400  
2, 4, p2, A, 4210  
3, 5, p1, A, 5400  
3, 5, p2, B, 5360  
3, 6, p1, B, 3600  
3, 6, p2, A, 2406  
4, 7, p1, A, 3290  
4, 7, p2, B, 5370  
4, 8, p1, B, 5070  
4, 8, p2, A, 5260  
5, 9, p1, A, 3000  
5, 9, p2, B, 5315  
5, 10, p1, B, 5400  
5, 10, p2, A, 3641  
6, 11, p1, A, 3424  
6, 11, p2, B, 5356  
6, 12, p1, B, 5400  
6, 12, p2, A, 4788  
7, 13, p1, A, 5400  
7, 13, p2, B, 5400  
7, 14, p1, B, 5400  
7, 14, p2, A, 5160  
8, 15, p1, A, 5280  
8, 15, p2, B, 5160  
8, 16, p1, B, 4490  
8, 16, p2, A, 4017  
9, 17, p1, A, 2700  
9, 17, p2, B, 5306  
9, 18, p1, B, 5090  
9, 18, p2, A, 4450  
10, 19, p1, A, 4271  
10, 19, p2, B, 5569  
10, 20, p1, B, 3377

10, 20, p2, A, 4473  
11, 21, p1, A, 5160  
11, 21, p2, B, 5430  
11, 22, p1, B, 4174  
11, 22, p2, A, 4886  
12, 23, p1, A, 2027  
12, 23, p2, B, 4271  
12, 24, p1, B, 5400  
12, 24, p2, A, 3804  
13, 25, p1, A, 4996  
13, 25, p2, B, 5367  
13, 26, p1, B, 5310  
13, 26, p2, A, 5390  
14, 27, p1, A, 3860  
14, 27, p2, B, 5119  
14, 28, p1, B, 4705  
14, 28, p2, A, 4535  
15, 29, p1, A, 2593  
15, 29, p2, B, 5279  
15, 30, p1, B, 5250  
15, 30, p2, A, 4246

### *Second Experiment Input*

replica, student, program, lock, time  
1, 1, p1, A, 1757  
1, 1, p2, B, 2404  
1, 2, p1, B, 1777  
1, 2, p2, A, 1716  
2, 3, p1, A, 1342  
2, 3, p2, B, 2552  
2, 4, p1, B, 2597  
2, 4, p2, A, 1238  
3, 5, p1, A, 1572  
3, 5, p2, B, 2248  
3, 6, p1, B, 3168  
3, 6, p2, A, 2460  
4, 7, p1, A, 1822  
4, 7, p2, B, 2455  
4, 8, p1, B, 2486



4, 8, p2, A, 2434  
5, 9, p1, A, 3503  
5, 9, p2, B, 3600  
5, 10, p1, B, 2454  
5, 10, p2, A, 1753  
6, 11, p1, A, 1830  
6, 11, p2, B, 3300  
6, 12, p1, B, 2880  
6, 12, p2, A, 890  
7, 13, p1, A, 648  
7, 13, p2, B, 940  
7, 14, p1, B, 2247  
7, 14, p2, A, 1363

## B.2 Accuracy Analysis

### *R Instructions*

```
# Use TSV export feature on each set of results:  
# [1] https://goo.gl/EPxmEa (first experiment)  
# [2] https://goo.gl/rsWPwi (second experiment)  
# Run http://pastebin.com/rSBjTiYj on each  
  
fisher.test(rbind(c(29,2),c(16,15)), alternative="two.sided")  
fisher.test(rbind(c(13,1),c(10,4)), alternative="two.sided")
```

## B.3 Experiment First Assignment



### Segundo Exercício Escolar de PLC (IF686)



Data: 24 de julho de 2014. Tempo limite (por questão): 1h00m. Prof.: Fernando Castor Modelo: A

Nome: \_\_\_\_\_ Assinatura: \_\_\_\_\_

**Instruções:** Faça o download desta questão disponível em <http://webserver.cin.ufpe.br/~rbl/if686/q1aw.zip> e salve este arquivo localmente. No Eclipse, clique com o botão direito dentro do Package Explorer (ou clique em "File") e selecione "Import...". A seguir, escolha a opção "General" e depois selecione "Existing Projects into Workspace". Marque o campo "Select archive file", clique no botão "Browse" correspondente e escolha o arquivo que você salvou localmente. Agora você está pronto para iniciar a questão.

**Importante:** Ao finalizar sua questão, levante seu braço e aguarde a vinda de um monitor com a sua próxima questão. Sua folha de prova será recolhida e você não poderá mais fazer alterações a respostas de questões anteriores.

1. (5,0 pts) Execute o programa em seu computador. Identifique que tipo de problema este programa está reproduzindo e aponte quais chamadas do código estão envolvidas no problema. Explique como o bug acontece.

## B.4 Experiment Second Assignment



### Segundo Exercício Escolar de PLC (IF686)



Data: 24 de julho de 2014. Tempo limite (por questão): 1h00m. Prof.: Fernando Castor Modelo: A

Nome: \_\_\_\_\_ Assinatura: \_\_\_\_\_

**Instruções:** Faça o download desta questão disponível em <http://webserver.cin.ufpe.br/~rbl/if686/mqb2.zip> e salve este arquivo localmente. No Eclipse, clique com o botão direito dentro do Package Explorer (ou clique em "File") e selecione "Import...". A seguir, escolha a opção "General" e depois selecione "Existing Projects into Workspace". Marque o campo "Select archive file", clique no botão "Browse" correspondente e escolha o arquivo que você salvou localmente. Agora você está pronto para iniciar a questão.

**Importante:** Ao finalizar sua questão, levante seu braço e aguarde a vinda de um monitor com a sua próxima questão. Sua folha de prova será recolhida e você não poderá mais fazer alterações a respostas de questões anteriores.

**2. (5,0 pts)** Execute o programa em seu computador. Identifique que tipo de problema este programa está reproduzindo e aponte quais chamadas do código estão envolvidas no problema. Explique como o bug acontece.