Deadlocks as Runtime Exceptions

Rafael Brandao Lobo and Fernando Jose Castor de Lima Filho

Center of Informatics, Federal
University of Pernambuco, Brazil
{rbl,castor}@cin.ufpe.br
`http://www.cin.ufpe.br/`

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 low. Empirical usability evaluation in two experiments showed that students finished tasks faster using the proposed approach and, in one of the experiments were also more accurate.

Keywords: deadlock, concurrency, exception handling, empirical studies

1 Introduction

Real-world applications use concurrency to do computation in parallel with multiple threads/processes taking more advantage of multicore processors. Unfortunately, concurrent code is difficult to write correctly, as it is well documented [1]. Deadlocks are a very common type of error in concurrent systems [1]. Deadlocks manifest when threads are waiting each other in a cycle, where each thread is waiting for another thread to release its desired lock. This produces a never-ending wait. Although there are two well-documented types of deadlocks, resource deadlocks and communication deadlocks [2][3], in this work our focus is on resource deadlocks, e.g., deadlocks that stem from threads attempting to obtain exclusive access to resources, and whenever the term *deadlock* is used we implicitly mean resource deadlock.

In practice, developers employ a number of approaches to deal with deadlocks: (i) static program analyses [4][8][9][10]; (ii) dynamic program analyses [11][12][13][16]; (iii) application-specific deadlock detection infrastructures [20]; (iv) techniques to guarantee the absence of deadlocks by construction [4]; (v) model checking [18]. 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 [1] and confirmed by us in Section 2); and (ii) it is possible to efficiently introduce deadlock detection for these two-thread, two-lock deadlocks (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[5] and Go[6] but they focus on different cases.

We present data from an empirical study showing that our assumption about the prevalence of TTTL deadlocks holds in practice. This confirms the findings of a previous study that focused on concurrency bugs in general [1]. 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.

2 Bug Reports Study

Attempting to generalize deadlock detection at runtime does not seem feasible from a performance viewpoint, since existing dynamic analyses take considerable time [12]. A previous study [1] that analyzed a number of concurrency bugs found that 30 out of 31 deadlock bug reports described deadlocks that involved at most two resources. We suspected that TTTL deadlocks were indeed much more common in real world systems than more complex deadlocks, so we investigated this further. This section presents the results of this investigation.

2.1 Data Collection

We selected three open source projects to investigate: Lucene, Eclipse and OpenJDK. Lucene[1] is a text search engine library. Eclipse[2] is one of the most popular IDEs for java developers. OpenJDK[3] is an open-source implementation of the Java Platform. These three projects share some key similarities: they're mostly written in Java; they have immense bug report repositories with easy tools to search into them; and lastly, their bug reports were usually well discussed and contained enough context that allowed us to classify them with some confidence, which was very important in this study.

In total, we collected 541 bug reports containing the word *deadlock* on their titles or on their descriptions. In Lucene, we found 27 closed issues of type "bug" in module "lucene-core".[4] In Eclipse, we found 406 resolved issues with resolution "fixed".[5] In OpenJDK, we found 108 issues of type "bug" on module "JDK" with resolution "fixed" and status "resolved".[6] We then proceeded to calculate the sample size that would allow us to have 95% of confidence level and 5% sampling error, which resulted in 225 bugs. Thus we created a random sample of that size to analyze further [7].

2.2 Data Labeling

We defined a set of fields to classify for each bug analyzed in the sample. First, we define a category. Then complete other fields based on how much we could understand of each bug report, like the number of threads involved, number of resources involved, type of locking mechanism used, and so on. Some extra information we collected was not used for this research.

We have four different values for the category field. Category A indicates that we are confident this is a resource deadlock. Thus, we should be able to describe the number of threads and locks that were involved. In contrast, category B represents the opposite: it is certainly not a resource deadlock. The reported bug is a communication deadlock, due to evidence of lost notify/signal in the bug context or anything else that supports it was not a resource deadlock. Category C refers to all the false-positive results: the term *deadlock* was used as a synonym of "hanging" or an "infinite loop", or to just mention another deadlock bug as a

[1] Lucene: http://lucene.apache.org/
[2] Eclipse: https://eclipse.org/
[3] OpenJDK: http://openjdk.java.net/
[4] Lucene bug reports list: http://goo.gl/DhVI3t
[5] Eclipse bug reports: http://goo.gl/qQnrEm
[6] OpenJDK bug reports: http://goo.gl/xYFfsO
[7] Bug reports sample: http://goo.gl/zNsIGz

reference, not as a cause of the current bug. Lastly, category D is set for all bugs that we could not understand clearly, due to a lack of evidence or discussion.

2.3 Results Analysis

Initially we consider only bugs we clearly identified, that is, bugs that were not labeled as category D. In Table 1, we can see in that from all resource deadlocks, 92.07% of them are indeed TTTL deadlocks. Another interesting finding is that 75.93% of all deadlocks are indeed resource deadlocks.

Table 1: Labeled Categories

Category	Number of Bugs
A	101
A and TTTL	93
B	32
C	23
D	69

If we now consider bugs we could not clearly classify, we can make some estimations of how many of them would be resource deadlocks and TTTL deadlocks. The first estimate is the worse case scenario, that is, all bugs in category D should be in category A but none of them would be TTTL deadlocks. In this case, only 54.7% of resource deadlocks would be TTTL deadlocks. If we look at the best case scenario, that is, all bugs in D would be TTTL deadlocks, then it would be 95.29% instead. However none of these two scenarios seems realistic. We believe that a more realistic scenario would be to assume that bugs in category D are distributed roughly in the same way as those in categories A, B, and C. If that is the case, out of all resource deadlocks, we estimate that 91.7% of them would also be TTTL deadlocks (Table 2. Thus TTTL deadlocks are certainly the most popular type of resource deadlocks, amouting 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.

Table 2: Estimated Labeled Categories Based On Distribution Found

Category	Number of Bugs
A	146
A+TTTL	134
B	46
C	33

2.4 Threats to Validity

Only one of the authors labeled all bug reports due to constraints on time and lack of resources. In counterpart, having only one reviewer makes it easier to guarantee that all bug reports were reviewed following the exact same procedure, but we would have preferred to have at least one more reviewer to label each bug independently and use it as a way to double check the labels accuracy. Furthermore, one factor that might limit generalizability of these findings is that we have looked at only three different open-source projects written mostly in Java, but different programming languages may also have a different distribution of deadlock bugs.

3 Deadlock Detection

We have modified the default implementation of Java's *ReentrantLock* to allow efficient runtime detection of TTTL deadlocks. We take advantage of *ReentrantLock* current algorithm and some of its guarantees to avoid the need to introduce extra synchronization mechanisms or costly atomic operations in this detection. It works as follows:

1. Each lock has a pointer for a thread which is the current owner or null when there's no thread owning that lock.
2. Each lock has an integer to represent its current state: 0 means the lock is free and no thread owning it (the *unlocked* state), 1 means there's a thread owning the lock (the *locked* state). For simplicity, we are only interested on these two states and its change holds the most complexity, but in the implementation of *ReentrantLock* each time the 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 of locks they are currently owning.
4. Each lock has a waiting queue of threads that are waiting to acquire it. Whenever a thread try to obtain a lock when it's already acquired, the thread will add itself on 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's deadlock. When the current thread is unable to acquire its desired lock, it must be because another thread is owning it already. 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 thread-local list of owned locks if the conflicting thread is waiting on it.
   (a) If positive, then we have a circular dependency (current thread is stuck waiting its desired lock and the conflicting thread is stuck waiting for a lock the current thread owns) thus a deadlock exception will be raised.
   (b) Otherwise, the thread parks.

This protocol relies on a few guarantees that were already provided by *ReentrantLock* default implementation such as:

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's guaranteed 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 is 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.

Now we show why this algorithm works:

Lemma 1. *Protocol can always detect deadlock TTTL deadlocks as soon as they happen.*

Proof. Suppose not and a TTTL deadlock occurred without deadlock exception being raised. Let's assume that threads A and B have both acquired locks a and b respectively, as follows:

$$write_A(state_a = locked) \rightarrow write_A(owner_a = A) \tag{1}$$

$$write_B(state_b = locked) \rightarrow write_B(owner_b = B) \tag{2}$$

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:

$$read_A(state_b == locked) \rightarrow write_A(waiting\_queue_b.insert(A)) \tag{3}$$

$$read_B(state_a == locked) \rightarrow write_B(waiting\_queue_a.insert(B)) \quad (4)$$

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:

$$read_A(owner_b == B) \rightarrow read_A(waiting\_queue_a.contains(B) == false) \quad (5)$$

$$read_B(owner_a == A) \rightarrow read_B(waiting\_queue_b.contains(A) == false) \quad (6)$$

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. □

Lemma 2. *Protocol never throw a deadlock exception for a non-existent TTTL deadlock.*

Proof. Suppose the opposite: a deadlock exception was raised and there's no real 2-deadlock. At least one of the following equations must be true in order to raise a deadlock exception:

$$read_A(owner_b == B) \rightarrow read_A(waiting\_queue_a.contains(B) == true) \quad (7)$$

$$read_B(owner_a == A) \rightarrow read_B(waiting\_queue_b.contains(A) == true) \quad (8)$$

Suppose without loss of generality the first equation is correct. It means 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 it means thread B is going to depend on the release of lock a to proceed. However, as we have seem 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. □

We have further extended the protocol to also guarantee the exception being raised on both threads involved. This does not affect how deadlock is detected but what should be done after a deadlock is detected, adding the following steps:

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 is desiring. Then the current thread (the owner of the desired lock) will add this conflicting thread in tainted threads list for that lock. After that, 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 into this list, then it must be because it was part of a deadlock before, so it removes its reference from the list and also raise a deadlock exception.
4. Every operation on the list of tainted threads of any locks (either reading or inserting values) should be followed up by some cleanup on all references of threads that are no longer running.

That is sufficient to force both threads to throw exceptions when only one of them would raise an exception in the initial protocol. Initial protocol would only throw exception on both if both threads simultaneously reach the point where they check for deadlock existence. However, for this particular case, this change introduces a different problem: dangling references: each thread would have added their conflicting thread on its owned locks's tainted threads list, but none of them would be able to acquire their respective desired locks (as in *item 3*), thus leaving their references behind for others to cleanup (as in *item 4*). We minimize this issue by asking other threads to clean these unneeded references as soon as they use any of the locks involved in the deadlock.

We modified OpenJDK *ReentrantLock* to implement this algorithm and its code can be found on our code repository[19]. We had to omit here details about that implementation for brevity, but our repository contains commits log history describing what changes we did and why we did them.

4 Evaluation

In this section 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 4.1); and (ii) a preliminary analysis of the performance overhead of our approach (Section 4.2). The exact input, instructions, and any additional document we have used in this section should be available at [19].

4.1 Usability Evaluation

We ran empirical evaluation to measure the efficiency of deadlock exceptions with regards 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 of bug description improved for implementation with deadlock exception when compared to the default one? To answer the first question, we watched for the time (in seconds) to finish each task. For the second question, each answer was evaluated based on three criteria with three possible values each: 0 for absence, 0.5 for partially present and 1 for fully present. Whenever $(A - B) + C \geq 1.5$ was true, we defined it as a correct answer, that is, whenever the bug was described as deadlock and at least one of the methods involved in the deadlock were identified correctly.

Table 3: Criteria evaluated for each answer

Type	Description
A	Correctly classified problem as deadlock.
B	Classified problem as different from deadlock.
C	Correctly identified method calls involved in the deadlock.

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 in 4 files, 3 threads, 3 explicit locks, and 82 lines of code in average. The second program, known as *Eclipse* 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*, where *LockA* was our implementation with deadlock detection on at least one thread involved in a deadlock, while *LockB* was just the default *ReentrantLock* implementation. Each student was assigned to either group A or B randomly. In group A, student would start with *LockA* in the first program but use *LockB* on the second program; meanwhile, in group B students would have the locks in opposite order.

All students started the experiment with the first question containing *Program 1*. When they finish to provide an answer, they should request for the second question. In this case, we collect the assignment, set a timestamp on it, and deliver second question with *Program 2*. Timestamp was written based on a chronometer visible to everyone in the laboratory. We set time limit for each question as 60 min, but the first group requested more time and we expanded to 90 min each.

We ran this experiment with two groups in two different days. First group consisted of undergraduate students attending Programming Language Paradigms course; they had classes about concurrent programming, including exercises in Java using *ReentrantLock* where deadlocks and other concurrent bugs should be avoided; however, these students were not experienced. Second group had graduate students enrolled in master's degree or PhD program attending Parallel Programming course; they had classes about advanced concepts of parallel programming and had practical exercises, including implementing their own lock; consequently they were expected to be very experienced in detecting concurrency bugs.

Time Analysis. We defined the following hypothesis to answer **RQ1**:

$$H_0 : \mu_{TimeLockA} \geq \mu_{TimeLockB} \tag{9}$$

$$H_1 : \mu_{TimeLockA} < \mu_{TimeLockB} \tag{10}$$

We used Latin Square Design to control two factors that might affect the metrics: subjects and program complexity. We designed one program that we considered easy to identify the bug even without exceptions and another program that was more difficult, composed by many files and classes and reflecting a more realistic case. We provided implementations of each program using either *LockA* or *LockB*: the two possible treatments we wanted to compare. 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. Consequently, we have replication, local control and randomization which are the three principles of experiment design [25]. Time analysis was conducted with R Statistical Software using the inputs extracted from each day. We used the linear model described in Figure 1 that considers the effect of different factors on the response variable similarly to Accioly's work[22], adding the effect between each replica and treatment [21].

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

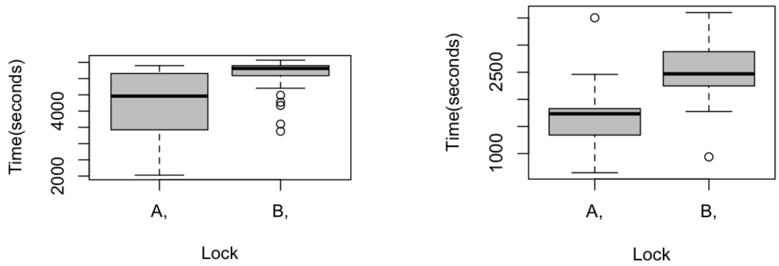
Y_{lijk} - response of l_{th} replica, i_{th} student, j_{th} program, k_{th} lock
τ_l - effect of l_{th} replica
$\tau\alpha_{li}$ - effect of interaction between l_{th} replica and i_{th} student
β_j - effect of j_{th} program
γ_k - effect of k_{th} lock
$\tau\gamma_{lk}$ - effect of interaction between l_{th} replica and k_{th} lock
ϵ_{lijk} - random error

Fig. 1: Regression model.

Initially, we plotted box-plot graphics shown in Figure 2 for both experiments. Then we run Box-Cox transformation to reduce anomalies such as non-additivity and non-normality. The value of λ at the maximum point in the curve drawn by box-cox function in R was not approximately 1 ($\lambda = 5$), thus we should apply the transformation: on our regression model, Y_{lijk} should be powered to λ. We did the same on the second experiment as $\lambda = 1.3636$.

After applying Box-Cox transformation, we ran Tukey Test of Additivity that checks whether effect model is additive, so we could evaluate whether interaction between factors displayed on the rows and columns of each latin square wouldn't affect significantly the response when the model is additive [25]. Considering the following hypothesis, in the first experiment, we obtained a p-value of 0.514 which is not lower than 0.05 thus we couldn't reject H_0, thus our model was additive; similarly for the second experiment, the model was also additive as p-value found was 0.914.

H_0 : The model is additive
H_1 : H_0 is $false$



(a) First experiment (b) Second experiment

Fig. 2: Box-plot on both experiment days

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. When a variable has *p-value* < 0.05, it means that factor was significant to the response. Table 4 and Table 5 shows the most important factor as the type of *Lock* for both experiments, allowing us to reject our null hypothesis defined for **RQ1**.

Table 4: First experiment ANOVA results.

	Df	Sum Sq	Mean Sq	F value	*p-value*
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		

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

$$H_0 : \mu_{CorrectAnswersLockA} \leq \mu_{CorrectAnswersLockB} \tag{11}$$

$$H_1 : \mu_{CorrectAnswersLockA} > \mu_{CorrectAnswersLockB} \tag{12}$$

Applying Fisher's exact test on data from Table 6 and Table 7, 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's an evidence of improvement on accuracy. However graduate students results presented a two-tailed P value equals 0.3259, which does not represent a statistically significant evidence.

Table 5: Second experiment 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		

	Correct	Incorrect
LockA	29	2
LockB	16	15

Table 6: First group accuracy

	Correct	Incorrect
LockA	13	1
LockB	10	4

Table 7: Second group accuracy

Discussion Although we cannot draw 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 Eclipse, thus being able to finish the tasks really quickly for both problems knowing exactly which points in the code were involved in the deadlock. This observation allows us to hypothetize whether deadlock exceptions are more helpful for less experienced programmers in general, but we leave this as a proposal for future work.

Threats to Validity We must consider a few remarks regarding the validity of our results. First remark is about the way we collected the timestamps: when a student finished any question, we manually wrote their name with the timestamp on the whiteboard so we could track time limit individually later, but we could potentially reduce overhead and increase timestamps precision if we used an automated alternative. Secondly, the first group did this experiment in replacement of their actual exam might have impacted the time we measured. We noticed some students spent more time on each question by purpose. We believe that they were reluctant to ask for the next question because they still had plenty of time left and they wanted to make sure it was correct. We did not notice 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. Third remark is related to programs' complexity: the ones we used to evaluate the students are considerably easier to understand than most programs in real world, but unfortunately we could not use any real world scenario as students would not be able to finish each assignment in time; with that in mind, we created two questions based on real world bugs we found on our bug report studies. Forth remark is about different background over different subjects: we tried to minimize their background differences by selecting groups where students had at least basic experience in concurrent programming and they should be familiarized with the types of bugs such codes could have; the first group was composed by undergraduate students who attended the class *Paradigms of Computaional Languages* where deadlocks were covered in classes and exercises; meanwhile, the second group had graduate students who attended the class *Parallel Programming* which covered concurrent programming in low level detail including deadlock detection. Last remark is about whether we are able to draw conclusions based on students data: some studies suggest that using students as subjects is as good as using industry professionals [24]; 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 [23].

4.2 Performance Overhead

We conducted a preliminary set of experiments to analyze the overhead of our approach. We compared our deadlock-safe implementation with the original `ReentrantLock` implementation available in the JDK and with Eclipse's deadlock-safe `OrderedLock` [20]. `OrderedLock` 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. OrderedLock deeply relies on Eclipse's code architecture. So, in order to use it in our evaluation, 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[19].

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. In this preliminary evaluation, we have conducted measurements for values of N equal to 10, 50, 100, and 200. 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 [26].

The measurements were made on an Intel CoreTM i7 3632QM Processor (6Mb Cache, 2.2GHz) running Ubuntu 12.04.4 LTS and each cell in Table 8 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.

Table 8: 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

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. [26]. 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 [26].

5 Related Work

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 [4][8][9][10] but in general they suffer of signficant amount of false positives being reported as there may exist deadlock cycles that are just impossible to happen during execution.

Dynamic analysis [11][12] finds potential deadlock cycles from execution traces of programs 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. Other dynamic techniques [13] [16] offer dynamic detection and recovery of deadlocks using *rollback* operations which are usually very costly and not always possible [15].

Lastly, we have a hybrid analysis, they are often using program analysis and compile time instrumentation to guide runtime and achieve better performance [14] or to run expensive computations during compile time to use later used at runtime [17]. Even though they employ both techniques simultaneously, they generate slower compile times, also requiring recompiling code to be able to find new potential deadlocks later. They also share limitations runtime techniques does, the main difference is that they tend to perform better than dynamic analysis alone during runtime.

6 Conclusion

In this work, we investigated which kind of deadlock is the most popular by looking at actual bug reports in relevant open source projects and confirmed a previous study claim that TTTL deadlocks are the most frequent case of deadlock, that is, 92.07% of all resource deadlocks we identified. This claim allowed us to focus on it instead and proposed an algorithm that efficiently detects them without any extra lock operations. We modified Java's *ReentrantLock* and provided a lightweight version of it that detects TTTL deadlock in runtime. We measured its performance overhead with a very conservative benchmark and we estimate our cost to be less than 6% for worse case on real world applications. Finally, we did an empirical evaluation to measure its usability and we found that deadlock exceptions speeds up finding deadlock bugs in code, and we also found some non-conclusive evidence showing that it may also improve accuracy of deadlock bug reports, but we leave for future work to verify whether this last observation is actually true.

References

1. Lu, Shan, et al: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. ACM Sigplan Notices. Vol. 43. No. 3. ACM, 2008.
2. Singhal, Mukesh. Deadlock detection in distributed systems. Computer 22.11 (1989): 37-48.
3. Knapp, Edgar: Deadlock detection in distributed databases. ACM Computing Surveys (CSUR) 19.4 (1987): 303-328.
4. Marino, Daniel, et al: Detecting deadlock in programs with data-centric synchronization. Software Engineering (ICSE), 2013 35th International Conference on. IEEE, 2013.
5. S. Marlow. Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming. O'Reilly, Aug 2013.
6. M. Aimonetti. Go Bootcamp: Chapter 8 - Concurrency. Chapter available at: `http://www.golangbootcamp.com/book/concurrency`
7. Dolby, Julian, et al: A data-centric approach to synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 34.1 (2012): 4.
8. Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. SIGOPS Operating Systems Review, 37(5):237–252, 2003.
9. Vivek K. Shanbhag. Deadlock-detection in java-library using static-analysis. Asia-Pacific Software Engineering Conference, 0:361–368, 2008.
10. Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In ECOOP 2005 - Object-Oriented Programming, pages 602–629, 2005.
11. Da Luo, Zhi, Raja Das, and Yao Qi: Multicore sdk: a practical and efficient deadlock detector for real-world applications. Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on. IEEE, 2011.
12. Cai, Yan, and W. K. Chan: MagicFuzzer: scalable deadlock detection for large-scale applications. Proceedings of the 2012 International Conference on Software Engineering. IEEE Press, 2012.

13. Pyla, Hari K., and Srinidhi Varadarajan: Avoiding deadlock avoidance. Proceedings of the 19th international conference on Parallel architectures and compilation techniques. ACM, 2010.
14. Hari K. Pyla and Srinidhi Varadarajan. Transparent Runtime Deadlock Elimination. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques, PACT '12, pages 477–478, New York, NY, USA, 2012. ACM.
15. Pyla, Hari Krishna. "Safe Concurrent Programming and Execution." (2013).
16. F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergiesa safe method to survive software failures. ACM Trans. Comput. Syst., 25(3), Aug. 2007.
17. Grechanik, Mark, et al: Preventing database deadlocks in applications. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013.
18. Havelund, Klaus, and Thomas Pressburger. Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer 2.4 (2000): 366-381.
19. Java's ReentrantLock with DeadlockException. Source code: `https://github.com/rafaelbrandao/java-lock-deadlock-exception`
20. Eclipe's OrderedLock class description: `http://www.cct.lsu.edu/~rguidry/ecl31docs/api/org/eclipse/core/internal/jobs/OrderedLock.html`
21. Iván Sanchez. Latin Squares and Its Applications on Software Engineering. Master's thesis, Federal University of Pernambuco, Recife, Brazil, 2011.
22. Paola Accioly. Comparing Different Testing Strategies for Software Product Lines. Master's thesis, Federal University of Pernambuco, Recife, Brazil, 2012.
23. Per Runeson. Using students as experiement subjects - an analysis on graduate and freshmen student data. In Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering. Keele University, UK, pages 95-102, 2003.
24. Miroslaw Staron. Using students as subjects in experiments - A quantitative analysis of the influence of experimentation on students' learning process. In CSEE&T, apges 221-228. IEEE Computer Society, 2007.
25. G. E. P. Box, J. S. Hunter, and W. G. Hunter, Statistics for experimenters: design, innovation, and discovery. Wiley-Interscience, 2005.
26. Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12). Berkeley, CA, USA, 2012.