A Survey of Deadlock Detection Studies: State of Art and Future Trends

Rafael Brandao Lobo

Center of Informatics, Federal
University of Pernambuco, Brazil
rbl@cin.ufpe.br

Abstract. The increasing popularity of multicore platforms made software design adopt more parallelism and concurrency. Such designs are difficult to maintain and they often require use of locks to achieve mutual exclusion and avoid data races. However, a very common issue that can appear from use of locks is known as deadlock. Many studies have proposed libraries and/or tools to detect deadlocks by examining source codes or execution traces, and some even provide recovery from such faults at runtime. This work will focus on recent studies in this area to identify state of art and future trends.

Keywords: deadlock concurrency multicore locks

1 Introduction

Real-world applications use concurrency to parallelize computation in multiple threads or processes, taking more advantage of multicore processors. However concurrenct code is difficult to write correctly, as it is well documented in [1]. 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 simply not feasible. When multiple threads access concurrently a certain memory position, data races may occur. One way to solve this problem is to first identify parts of the code that should not allow threads to run simultaneously – they are called critical sections – and then protect them by using locks.

Locks are used to avoid data races in concurrent code. When a thread acquires a certain lock, no other thread can acquire the same lock. Any thread that attempt to acquire that lock will be blocked until that resource is released. Unfortunately locks cannot be easily composed in the code and a very common bug caused by composing locks in unexpected ways is called deadlock. A deadlock manifests when threads are waiting each other in a cycle, each one holding locks that other thread is trying to acquire.

Suppose a thread X acquires a lock A and then tries to acquire a lock B. Given the nature of parallelism, another thread Y simultaneously acquires a lock B and then tries to acquire lock A. Since thread X is blocked waiting for lock B to be released and thread Y which owns lock B is blocked waiting for thread X to release lock A, both will never finish waiting, thus creating a deadlock. When a deadlock happens, a program fail to make progress without notice to users and developers, making harder to identify the problem.

Deadlocks can be avoided if locks are acquired in any order, as long as the other threads follows that particular order. For example, if both thread X and thread Y always acquired locks A and B in this order, then no deadlock could ever occur because there's no waiting relationship cycle between them. Some techniques to avoid deadlock require to impose a global order on locks and check whether this order is ever violated when the locks are acquired [4].

There are two types of deadlocks: resource deadlocks and communication deadlocks [2] [3]. Resource deadlocks are illustrated in the previous example, where threads waits for resources to become available. Communication deadlocks happens when threads are blocked waiting messages, so messages are the resources for which threads wait. In this survey and studied techniques [6] [9], we focus on resource deadlocks and we will call them simply deadlocks from now on.

Potential deadlocks can be detected mostly via static analysis, dynamic analysis, or their integration. Different categories of techniques have its own advantages and disadvantages and they complement one another.

In this survey, we will go through the most recent work related to deadlock detection tools and classify them into one of those categories. They were collected by examining publications from 2011 to 2014 in conferences such as CAV, FSE, ICSE, ICST, ISSTA, OOPSLA and PLDI, but also through citations among them.

Static analysis techniques are presented first, followed by dynamic analysis techniques. After that, hybrid analyis techniques are presented. Next section we bring a brief discussion on the techniques and analyse potential trends for the future.

2 Static analysis

Static analysis techniques read source code from a program and try to identify potential problems present in the code without executing it. In order to detect deadlocks, they generally attempt to detect cyclic relationships of resource acquisition between threads, where cycles represent possible deadlocks.

Many false positives are reported in most static analysis techniques. Its precision depends on identifying and ignoring cycles that could not happen in practice, but this is a challenge for static analysis techniques as they only manipulate source code.

2.1 Detecting deadlock in programs with data-centric synchronization.

Marino et al. [4] proposed a static analysis technique to detect potential deadlocks in AJ programs, where AJ is an extension of Java that implements atomic sets for fields of classes as an abstraction of locks to prevent data races and atomicity violations by construction. Additional ordering annotations were added to AJ to handle cases with recursive data structures.

The declarative nature of AJ's data-centric synchronization allows the algorithm presented in [4] to infer which locks each thread may acquire and then it computes a partial order for those atomic sets which is consistent with analysed lock acquisition order. Whenever such order is detected, the program is guaranteed to be deadlock-free, otherwise possible deadlock is reported.

They implemented the deadlock analysis as an extension of their existing AJ-to-Java compiler and data-centric synchronization annotations were given as special Java comments which would be parsed and given to the type checker and deadlock analysis. Then the AJ source is translated to Java and written into a separated project with the transformed code, which would be compiled into bytecode and executed by JVM. The deadlock analysis uses WALA program analysis framework[1] to construct the call graph and the propagation of atomic sets in that analysis is reduced to a distributive data flow problem.

The problem with their approach is that AJ is a research language and does not have real users, thus obtaining suitable subject programs to evaluation was difficult. They used program converted from Java in their previous work [5] for this study which might not represent concurrent programming styles that occur in practice. However for the majority of their subject programs (7 out of 10), deadlock-freedom could be demonstrated without any programmer intervention, where two programs required small refactoring in order to remove spurious deadlock reports, so they were optimistic that their proposed deadlock analysis could scale to large programs.

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. So if a given run does not identify potential deadlocks, it does not mean the program is free from bugs. 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, posing another challenge.

[1] See wala.sourceforge.net

3.1 MulticoreSDK: a practical and efficient deadlock detector for real-world applications.

Zuo et al. [6] present a dynamic deadlock detection tool called MulticoreSDK which is capable of handling even large real-world applications and its algorithm consists of two different phases.

In the first phase, the algorithm works offline by examining a single execution trace obtained after an instrumented program finishes running. Then it creates a lightweight lock graph based on program locations of lock events according to specific rules and also identify locks that may be deadlock-related in this graph by finding cycles on it.

In the second phase, it examines the execution trace again and constructs a filtered lock graph based on the lock id of lock events, analysing only deadlock-related locks found in the first phase. Finally, it find cycles in this reduced graph to report as potential deadlocks in the program.

The algorithm requires to pass over the program trace two times, but according to experiments done in [6], it does not pose a big performance overhead since most time is consumed in the deadlock analysis itself.

MulticoreSDK works for java programs and its instrumentation is done by deploying a well known bytecode instrumentation technique [7] to insert extra bytecode around instructions such as *monitorenter* and *monitorexit* to record the thread and the id of lock objects. Extra bytecode is also inserted around methods lock and unlock of Lock interface in java.util.concurrent package and around the enter/exit of synchronized methods.

The evaluation of MulticoreSDK used some Java multithreaded programs including both open source projects and commercial software. Instrumented version of each program is run once to obtain a single execution trace for each of them which will be analysed by this algorithm and a traditional approach [8] to compare accuracy and performance of them. For small applications, the performance gain in analysis time is small but memory consumption is reduced by about half. However, for large applications, the performance gain is considerable and some cases reaching about 33% performance gain and consuming about 10 times less memory. The better performance and reduced memory consumption did not compromise accuracy of the analysis and deadlocks reported by two approaches were exactly the same.

3.2 MagicFuzzer: Scalable deadlock detection for large-scale applications.

Cai et al. [9] present a dynamic analysis technique known as MagicFuzzer which consists of three phases: on first phase, it monitors critical events such as a thread creation or lock acquisition and release 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 four sets where one of them is called cyclic-set and must contain all the target lock dependencies, then it constructs a set of thread-specific lock-dependency relations based on the locks in that cyclic-set which is tranversed to find potential deadlock cycles; on the third phase, all deadlock cycles found in second phase is used as an input and actively executes the program to observe if any execution will trigger any of those potential deadlock cycles found, thus reporting the deadlocks if they occur in this phase, so it never reports a false positive due to its confirmation of each potential deadlock cycle.

MagicFuzzer uses an active random scheduler to check against a set of deadlock cycles. It improves the likehood that a match between a cycle and an execution will be found. If not all cycles are confirmed in a run, then the scheduler iteratively proceeds to confirm the remaining cycles in the next run until all cycles are confirmed or until it reaches a certain number of executions given by the user. It is left as future work a report on the probability of this scheduler to confirm deadlock for a given set of potential deadlock cycles.

MagicFuzzer was implemented using a dynamic instrumentation analysis tool known as Pin 2.9 [10] and it only work for C/C++ programs that use Pthreads libraries on a Linux System. It was tested on a suite of widely-used and large-scale C/C++ programs such as *MySQL* [11], *Firefox* [12] and *Chromium* [13]. It was collected measurements of time consumption and memory usage to compute the maximum amount of memory used for each run. In the table below those measurements are displayed:

Table 1. MagicFuzzer's benchmark results

Benchmark	Memory(MB)			Time(s)			# of cycles			# of real deadlocks
	iGoodlock	MSDK	Magiclock	iGoodlock	MSDK	Magiclock	iGoodlock	MSDK	Magiclock	
SQLite	1.05MB	1.05MB	1.05MB	0.002s	0.003s	0.002s	1	1	1	1
MySQL	>2.8GB	1.15MB	1.05MB	>2m5s	6m38s	1.73s	>1	1	1	1
Chromium	>2.8GB	>48.2MB	8.01MB	>1h47m	>1h	1m42s	ND	ND	3	UKN
Firefox	>2.8GB	122.41MB	4.14MB	>10m40s	7.43s	3.06s	ND	0	0	0
OpenOffice	245.20MB	>48.4MB	8.01MB	1h46m	>1h	0.67s	0	ND	0	0
Thunderbird	298.83MB	40.09MB	4.15MB	16m13s	4.75s	1.18s	0	0	0	0

For some programs, the test simple starts the program and then close it when the interface appears. Other programs were tested with test cases adopted from bug reports in their own repositories. The reason of different test strategies was due to the inexistence of benchmarks of large-scale C/C++ programs with test cases that could repeat occurrences of deadlocks. The results of tests show that MagicFuzzer could run efficiently with low memory consumption compared to similar techniques, like MulticoreSDK [6].

3.3 Avoiding deadlock avoidance.

Pyla et al. [14] present Sammati as a dynamic analysis tool which is capable of detecting deadlocks automatically and recovering for any threaded application that use the pthreads interface, transparently detecting and eliminating deadlocks in threaded codes.

Sammati is implemented as a library that overloads the standard pthreads interface and turns locks into a deadlock-free operation. Since it is just a library overload, it does not require any modifications to application source code.

This approach makes memory updates to be only visible outside of the critical section when all parent locks are released, by associating memory accesses with locks and specifying ownership of memory updates within a critical section to a set of locks. 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 all memory updates done. Then it is 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, thus providing recovery from deadlock.

Differently from transactional memory systems, it does not significantly impact performance because its runtime privatizes memory at page granularity, instead of instrumenting individual load/store operations. To implement containment for threaded codes, it was used a technique described in [15]: 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 (a single threaded process). In Sanmati, threads were converted to cords transparently. Since all coords share a global address space, the deadlock detection algorithm has access to the holding and waiting sets of each coord and the deadlock detection can be performed at the time a lock is acquired.

The 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. The algorithm then tries to acquire some requested lock in a non-blocking attempt. When the acquisition is successful, the acquired lock is inserted in the first hash table and into that cord's queue. However, if the acquisition fails, then the algorithm finds which cord is the owner of that lock and checks if that cord is waiting for another lock. If this is not the case, then there's no cycle, so deadlock algorithm inserts that lock in the second hash table and finally try to attempt the same lock in a blocking way. But if that is the case, then it moves on to check which cord is the owner of the next lock, thus transversing the waiting relationship graph until a cycle is detected. The complexity of this detection algorithm is linear and has upper bound of $\mathcal{O}(\mathrm{n})$.

The perfomance evaluation analysed threaded applications such as SPLASH [16], Phoenix [17] and a few synthetic benchmark suites which contain programs written intentionally to create deadlocks. Sanmati shows a performance comparable to pthreads for most applications and in a few cases it even performs better. Memory overhead remained below 2 MB independently of the number of the cords in all programs tested.

Sanmati does not support condition variables for communication between threads in its current implementation. This limitation is planned to be removed in future work.

4 Hybrid 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. 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.

4.1 Preventing database deadlocks in applications.

Grechanik et al. [18] created a novel approach known as REDACT that statically detects all hold-and-wait cycles among resources in SQL statements and prevents database deadlocks automatically: a supervisory program that prevents database deadlocks was designed to intercept transactions sent by applications to databases dynamically. Once a potential deadlock is detected, the conflicting transaction is delayed and the deadlock cycle is broken.

REDACT performs its static analysis at compile time to prevent deadlocks at runtime. This separation enables the algorithm to avoid expensive computations at runtime and instead it does a simple lookup in the hold-and-wait cycles already detected earlier, turning deadlock prevention fast and efficient.

The first step involves manual-effort on every application that uses a database to extract transactions used by them. Once the SQL statements were extracted, the static analysis phase begins and these transactions are parsed. These parsed results are used as input for a modeler that automatically transforms SQL statements into abstract operations and synchronization requests which is used by hold-and-wait detection algorithm. Once these cycles are detected, they are used as input to a supervisory control in the dynamic analysis.

The dynamic phase is responsible to prevent database deadlocks automatically. The first step is to add interceptors in the application with callbacks associated with particular events. Instead of sending SQL statements from application to the database, the interceptors 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 currently in the execution queue. If no hold-and-wait cycle is present, then it forwards these statements to the database for execution; 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 statament is finally sent to be executed by the database, thus effectively preventing the database deadlock.

Database deadlocks are typically detected within database engines at runtime, raising deadlock exceptions and rolling back transactions involved in the circular wait. Programmers can write special handling for such situations, but most of the time they just repeat aborted operations. Thus database deadlocks may degrade performance significantly and the effort of REDACT is to prevent them to ever happen.

However not all hold-and-wait cycles detected in the static analysis will actually lead to database deadlocks, so false positives are possible. Some optimizations in the database may prevent it from raising an exception even in cases where a deadlock can be easily identified. Thus it's only possible to know if there's a deadlock in the database when the statement is executed and this approach to prevent deadlocks consequently imposes an overhead in addition to some reduction in the level of parallelism in applications.

In a performance evaluation, an extreme case scenario with a total of 5,000 SQL statements took aproximately 6.5 hours for the static analysis to finish and detect all hold-and-wait cycles. In a more realistic case of a large scale application with more than 50 statements, all cycles were found in less than 2 seconds. However, all subject programs used in the experimental evaluation were relatively small mostly because it is difficult to find large open-source applications that largely use non-trivial database transactions.

REDACT's cycle analysis has exponential complexity and this poses a limitation for it to deal with ultra large-scale transactions. Also, when the static analysis produces too many false positives of hold-and-wait cycles, it may cause big performance overhead, but this evaluation was left as a proposal for future work.

5 Discussion

There's no silver bullet in terms of deadlock detection. All techniques presented in this study have downsides and no solution is capable of handling deadlocks in all programming languages with little or no effort from developers. In the table below is presented a small comparision of techniques.

Table 2. Techniques in comparision

Technique	Type	Language	Effort	Performance	False Positives
Marino's	Static	Java with AJ	High	Inconclusive	Yes
MulticoreSDK	Dynamic	Java	Low	Medium	No
MagicFuzzer	Dynamic	C/C++ with pthreads	Low	Low	No
Sanmati	Dynamic	pthreads	Low	Low	No
REDACT	Hybrid	SQL	Medium	High	Yes

In this comparision, MagicFuzzer and Sanmati are the techniques with lowest effort from programmer and better performance. Since they are dynamic tools, they do not produce false positives. However, MagicFuzzer does not guarantee that its executions will identify and report all existent deadlocks in a given program. Meanwhile, Sanmati guarantees deadlock-freedom by providing transparent recovery for deadlocks. All in all, Sanmati is good candidate to be the most promising alternative for programs that use pthread interface.

It seems that in the future, more dynamic deadlock detection techniques will appear because they are more scalable and does not produce spurious deadlock reports. Low effort is also important to make a specific technique successful and early adopted by developers.

Hopefully there will be solutions even more generic that could either detect deadlocks fast enough or avoid them without imposing high performance overheads.

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. Dolby, Julian, et al: A data-centric approach to synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 34.1 (2012): 4.
6. 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.
7. Tanter, ric, et al: Altering Java semantics via bytecode manipulation. Generative Programming and Component Engineering. Springer Berlin Heidelberg, 2002.
8. ConcurrentTesting - Advanced Testing for Multi-Threaded Java Applications, `https://www.research.ibm.com/haifa/projects/verification/contest/`
9. 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.
10. Luk, Chi-Keung, et al. Pin: building customized program analysis tools with dynamic instrumentation. ACM Sigplan Notices 40.6 (2005): 190-200.
11. MySQL, available at: `http://www.mysql.com`
12. Firefox, available at `http://www.mozilla.org/firefox`
13. Chromium, available at `http://code.google.com/chromium`
14. Pyla, Hari K., and Srinidhi Varadarajan: Avoiding deadlock avoidance. Proceedings of the 19th international conference on Parallel architectures and compilation techniques. ACM, 2010.
15. Berger, Emery D., et al: Grace: Safe multithreaded programming for C/C++. ACM Sigplan Notices. Vol. 44. No. 10. ACM, 2009.
16. SPLASH-2. SPLASH-2 benchmark suite, available at `http://www.capsl.udel.edu/splash`
17. Ranger, Colby, et al: Evaluating mapreduce for multi-core and multiprocessor systems. High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on. IEEE, 2007.
18. Grechanik, Mark, et al: Preventing database deadlocks in applications. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013.