A Dynamic Deadlock Prediction, Confirmation and Fixing Framework for Multithreaded Programs

Yan Cai
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
yancai2@student.cityu.edu.hk

Abstract. Deadlocks widely exist in real-world multithreaded programs. Existing predictive strategies are not consistently scalable; existing confirmation strategies may miss to trigger deadlocks, and existing fixing strategies may incur false positives or high runtime overheads. This paper presents an overview of my approach to automatic deadlock prediction, confirmation, and fixing.

1 Introduction

Deadlock [4] widely exists in real-world multithreaded programs such as `Chromium`, `MySQL`, Apache `httpd`, and `OpenOffice`. Its occurrence in a program run prevents the run to proceed further. *Communication* deadlocks and *resource* deadlocks are two board categories [1, 5, 8]. An instance of the former kind occurs when a thread waits for a message that has been sent before the thread starts to wait or every sender is blocked from sending such a message. An instance of the latter kind occurs when each thread in a set *T* cyclically waits for a resource held by another thread in *T*.

As we are going to present, existing deadlock prediction, confirmation, and fixing strategies are still inadequate to handle the complex nature of real-world programs reliably. This paper presents our framework to be built that addresses the challenges in these areas in the context of object-oriented programs.

Table 1. Memory and Time Comparisons among *iGoodlock*, *MulticoreSDK*, and *Magiclock*

Benchmark	Memory (MB)			Time in second (s)		
	iGoodlock	MulticoreSDK	Magiclock	iGoodlock	MulticoreSDK	Magiclock
SQLite	1.05MB	1.05MB	1.05MB	0.002s	0.003s	0.002s
MySQL	>2800MB	1.15MB	1.05MB	>125s	398s	1.73s
Chromium	>2800MB	>48.2MB	8.01MB	>6420s	>3600s	1m42s
Firefox	>2800MB	122.41MB	4.14MB	>640s	7.43s	3.06s
OpenOffice	245.20MB	>48.4MB	8.01MB	6360s	>3600s	0.67s
Thunderbird	298.83MB	40.09MB	4.15MB	973s	4.75s	1.18s

2 Dynamic Deadlock Prediction and Confirmation

We [3] have shown that *MulticoreSDK* [10] and *iGoodLock* [8] could not consistently scale up to analyze the execution traces of large-scale programs to detect deadlock potentials. Table 1, taken from [3], shows that they may exhaust all available memory (2.8GB) for a process or run over an excessive period.

The general idea of *MulticoreSDK* [10] and *iGoodLock* [8] are, implicitly or explicitly, search over the lock-order graph (see the rightmost graph in Fig. 1) formed by the execution trace to locate every (minimal) circular chain of edges, and reports every such chain as a deadlock potential (and we also refer to it as a *cycle*).

This work is an unpublished copy and the content of it has been informally presented at the *Doctoral Symposium* of *the 26th European Conference on Object-Oriented Programming* (*ECOOP'12*, *DS*).

Our Idea: In an execution trace of a real-world program, only a small fraction of all lock dependencies between threads may involve in such a cycle; otherwise, the program may have numerous amounts of deadlock potentials. *Magiclock* exploits this insight. It iteratively infers and removes edges that each *cannot* involve in such a cycle from the graph. It eliminates false positives by enforcing the sets of locks holding by the threads of a cycle to be mutually disjoint, and avoids the generation of duplicated cycles by searching the graph starting with all unique combinations of any two threads. The algorithm can be found in [3] and the column in Table 1 entitled *Magiclock* shows the result of this technique. We will generalize our approach to handle conditional variables and communication deadlocks. We will evaluate the generalized *Magiclock* by systematically reproducing the deadlock bugs reported over a period in the bug repository of the set of benchmarks shown in Table 1.

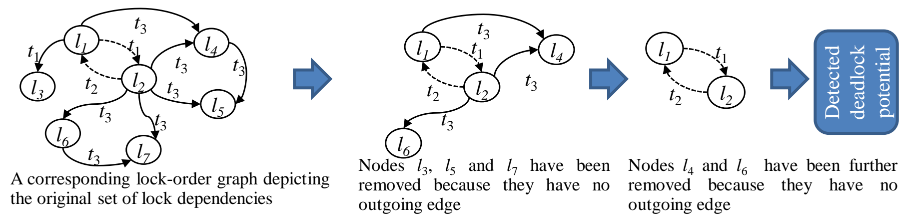

Fig. 1 Lock-order graph (t_i is a thread; l_i is a lock; and an arrow refers to lock acquisition order)

Isolating the real deadlocks from the predictive ones is the next target. Our experiment [3] shows that triggering deadlocks from these large-scale programs was difficult, and the probability was significantly lower than that reported in [8], which used a suite of small- and medium-scale benchmarks. We observe that *DeadlockFuzzer* [8], in our experiment [3], often suspended some threads that caused the run ceased to proceed further naturally (i.e., *thrashing*), and resumed one holding thread from suspension, making the deadlock unable to be confirmed. The experiment reported in [2] shows that *PCT* can be of very low probability in detecting deadlocks for large-scale programs, even though it suffered from no thrashing. *BTrigger* [12] required manual efforts to determine the variable matching conditions to suspend a thread and judge a good timeout period to resume a thread from suspension. *ConTest* [6] simply injected arbitrary timeouts to alter the thread schedule with the intent to trigger deadlocks.

Our Idea: We are developing strategies to address these challenges. Our plan is to firstly conduct an empirical investigation to find clues on why thrashing builds up, why injecting timeouts is an effective strategy, and why the deadlock code has been passed through in an execution trace without triggering any deadlock. Then, we will either enhance the predictive phase of deadlock detection to collect data about such clues if the clues require whole trace analysis or extract partial information from such an execution trace for on-the-fly condition determination in the confirmation run. We target to develop a technique that can result in a consistently high probability (e.g., 80%) in confirming real deadlocks. We plan to evaluate our technique against the above-mentioned existing techniques on the benchmarks used to evaluate *Magiclock* in terms of detection probability, rate of thrashing, slowdown factors, and memory footprint.

3 Dynamic Deadlock Fixing

Once a deadlock is revealed, it can be fixed. Recent deadlock fixing approaches [9, 11, 13] aim to serialize the execution of the program portion involved in deadlocks.

Nir-Buchbinder et al. [11] proposed inserting a gate lock right before each thread involved in a deadlock acquires its problematic lock, which nonetheless, cannot handle communication deadlocks and non-trivial resource deadlocks, and may introduce new deadlocks due to gate lock insertions. *Dimmunix* [9] prevented the second occurrence of a deadlock by recording the pattern of the first occurrence of the deadlock and matching the pattern in later execution traces. Such a pattern matching strategy is imprecise, failing to avoid deadlocks from re-occurrence. Its slowdown factor is good (e.g., 15% [9]). *Gadara* [13] detected all cycles offline. At runtime, any matched cycle along the run triggers a serialization of the corresponding deadlock potential code, and many such occurrences in the same run may prolong the run significantly.

To ease our presentation, we refer to a lock involved in a deadlock as a *wait-lock*, and a thread involved in the same deadlock as a *wait-thread*.

Our Idea: With respect to a deadlock, we plan to actively assign the corresponding wait-lock of a wait-thread to the wait-thread when the wait-thread acquires any wait-lock of the deadlock. We aim to develop a technique that introduces no deadlock.

We expect that this active lock assignment strategy breaks the circular waiting condition [4] for deadlock formation. Moreover, many programming languages support reentrant locks. This feature allows the same thread successfully acquires the same lock that it is holding. Hence, a pre-acquisition of a wait-lock by a wait-thread does not block the thread to acquire the wait-lock at the deadlocking position.

Thread t_1: `synchronized(A) { synchronized (B) {…}}` Thread t_2: `synchronized(B) { synchronized (A) {…}}`
(a) Example deadlock code
Thread t_1: **`synchronized(B) {`** `synchronized(A) { synchronized (B) {…}}`**`}`** Thread t_2: `synchronized(B) { synchronized (A) {}}`
(b) Pre-acquisition of the lock B by thread t_1
Thread t_1: `synchronized(A) { synchronized (B) {…}}` Thread t_2: **`synchronized(A) {`** `synchronized(B) { synchronized (A) {…}}`**`}`**
(c) Pre-acquisition of the lock A by the thread t_2

Fig. 2 Two pre-acquisition solutions illustrated in (b) and (c) for the deadlocking scenario illustrated in (a). The dynamically inserted codes are shown in the form of **inserted code**.

Fig. 2(a) illustrates a deadlock scenario. Our fixing strategy leads to two possible fixes for the scenario. They are the pre-acquisition of the locks `B` and `A` by the threads t_1 and t_2, respectively, which are depicted as Fig. 2(b) and Fig. 2(c). (The gate-lock fixing strategy of Nir-Buchbinder et al. [11] does not work if there is a pair of `wait()-notidfy()` statements between t_1 and t_2 within the inserted gate-lock block.)

Several technical challenges still exist: A pre-acquisition of a lock by a thread may alter the original lock acquisition order of the program, introducing new deadlocks. We will analyze the lock-order graphs to determine whether some potential fixes are undesirable and avoid generating them. Because the involved static or dynamic analysis on lock-order graphs could be imprecise, we will study new dynamic lock retreat strategy to release the actively pre-acquired wait-lock from a wait-thread to resolve "thrashing". Besides, it may not be generally feasible to pre-acquire a wait-lock at the positions as illustrated by Fig 2. Multiple deadlocks may interfere with one another.

We plan to evaluate to what extent our technique (1) introduces no new deadlocks and (2) handles general scenarios (e.g., communication deadlocks) that *Dimmunix* and *Gadara* could not handle well. We plan to use several large real-world applications that we have presented in Table 1 as benchmarks.

4 Conclusion

In this paper, we have reviewed existing work on deadlock prediction, confirmation, and fixing. We have sketched our framework that addresses some selected technical challenges. We believe that having a highly effective strategy in each phase is essential to address the challenges posted by deadlocks in real-world programs.

5 References

1. S. Bensalem and K. Havelund, Scalable Dynamic Deadlock Analysis of Multi-threaded Programs. In *the 2005 workshop on Parallel and Distributed Systems: Testing and Debugging* (*PADTAD'05*), 2005.
2. S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS'10*), 167–178, 2010.
3. Y. Cai and W.K. Chan. MagicFuzzer: Scalable Deadlock Detection for Large-Scale Applications. To appear in *Proceedings of the 34th International Conference on Software Engineering* (*ICSE'12*), research track, 11 pages, 2012. Also available at: http://www.cs.cityu.edu.hk/~wkchan/papers/icse12-cai+chan.pdf.
4. Deadlock, http://en.wikipedia.org/wiki/Deadlock.
5. J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (*ASE'09*), 480–491, 2009.
6. E. Farchi, Y. Nir-Buchbinder, and S. Ur. A Cross-Run Lock Discipline Checker for Java. In *the 2005 workshop on Parallel and Distributed Systems: Testing and Debugging* (*PADTAD'05*), 2005.
7. P. Joshi, M. Naik, K, Sen, and D. Gay. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE'10*), 327–336, 2010.
8. P. Joshi, C.S. Park, K. Sen, amd M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'09*), 110–120.
9. H. Jula, D. Tralamazza, C. Zamfir, and G.e Candea. Deadlock Immunity: Enabling Systems to Defend against Deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (*OSDI'08*), 295–308, 2008.
10. Z.D. Luo, R. Das, and Y. Qi. MulticoreSDK: A Practical and Efficient Deadlock Detector for Real-World Applications. In *Proceedings of the 2011 fourth IEEE Conference on Software Testing, Verification and Validation* (*ICST'11*), 309–318, 2011.
11. Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: From Exhibiting to Healing. In *Proceedings of the 8th Workshop on Runtime Verification* (*RV'08*), 104–118, 2008.
12. C.-S. Park and K. Sen. Concurrent Breakpoints. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPOPP'12*), poster article, 331–332, 2012.
13. Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (*OSDI'08*), 281–294, 2008.