Deadlock Detector and Solver (DDS)

Eman Aldakheel
University of Illinois at Chicago
Chicago, Illinois, USA
Princess Nourah bint Abdulrahman University
Riyadh, Saudi Arabia
ealdak2@uic.edu

ABSTRACT

Deadlock is among the most complex problems affecting the reliability of programs containing multiple, asynchronous threads. When undetected, deadlocks can lead to permanent thread blockage. Current detection methods are typically based on timeout and rollback of computations, resulting in significant delays. This paper presents *Deadlock Detector and Solver (DDS)*, which can quickly detect and resolve circular deadlocks in Java programs. DDS uses a supervisory controller, which monitors program execution and automatically detects deadlocks resulting from hold-and-wait cycles on monitor locks. When a deadlock is detected, DDS uses a preemptive strategy to break the deadlock. Based on our experiments, DDS can in fact resolve deadlocks without significant run-time overhead.

CCS CONCEPTS

• **Software and its engineering** → **Deadlocks**; **Multithreading**; **Monitors**; *Software testing and debugging*;

1 INTRODUCTION

The onset of multicore hardware is fueling a trend toward concurrent software systems. With increasing frequency, software developers are using such language constructs as *Java threads* in order to take advantage of multicore hardware capabilities [14].

When multiple threads share the same data structures, object synchronization is necessary to avoid data races. Data races occur if multiple threads access simultaneously the same structure while at least one such access modifies the shared structure. A well-known disadvantage of object locking is that it can cause deadlocks, for instance, when a hold-and-wait cycle occurs involving locked objects. Deadlocks are notoriously difficult to detect through testing because they tend to manifest themselves randomly.

Current Java tools for detecting deadlocks are either unable to resolve deadlocks [5, 7–9, 16], or suffer from performance degradation [7, 12]. Our *Deadlock Detector and Solver (DDS)* is a run-time monitoring toolkit that detects and resolves deadlocks involving Java monitor locks without the need for code annotations and with modest performance overhead. This is in contrast with the existing systems [11–13]. DDS does not force a deterministic code execution in a way similar to UnDead [17]. Other authors have used lock graphs and resource graphs to monitor the state of running program and to detect deadlocks [3–6, 9, 10, 15]. We also use lock graphs; however, we use lock graphs not only for detecting deadlocks, but also for resolving the deadlocks. Better yet, DDS only imposes an average overhead of about 5% for real-world Java applications as measured in our empirical studies.

This paper is organized into two sections; the first section presents the deadlock detection mechanism underlying DDS; the second section evaluates empirically the performance of DDS. It is worth noting that the evaluation is done on the basis of the various benchmarks for both deadlocked and non-deadlocked Java programs.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3190331

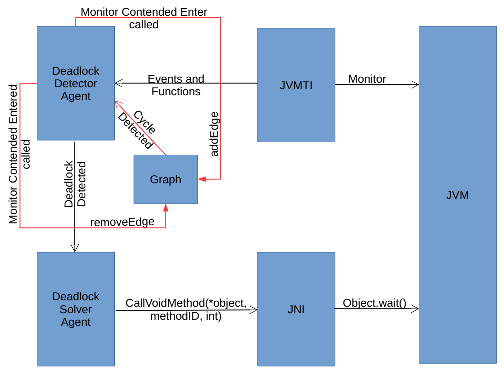

Figure 1: Architecture of Deadlock Detector and Solver

2 OVERVIEW OF DDS TOOLKIT

Figure 1 is a schematic flow diagram showing how DDS works. We use an existing library called the Java Virtual Machine Toolset Interface (JVMTI) to monitor the execution of a Java program by the Java Virtual Machine (JVM). JVMTI notifies a DDS component called the Deadlock Detector Agent when two kinds of events occur. The first event, *monitor_contended_enter*, occurs when a Java thread tries to acquire a monitor lock, for instance, in order to access the object associated with that lock. The second event, *monitor_contended_entered*, occurs when a thread actually obtains the lock. DDS registers suitable callbacks for these two events.

When the events occur, our DDS callbacks are actually invoked by the JVMTI. The two callbacks allow us to create and maintain a *resource mapping* graph. Vertexes V in the graph represent locks held by threads. Edges E represent threads waiting for a lock held

Table 1: Empirical results of DDS. For each benchmark, we list the line count of Java source code, the number of *synchronized* statements contained therein, and the size of input sets we used.

Benchmarks Name	Line Count	Num. of Syn. Stat.	First Size Set	Second Size Set
elevator	587	8	8 floors	50 floors
sor	783	14	1,000,000 elements	2,250,000 elements
moldyn	1338	14	2048 particles	6656 particles
raytracer	1906	15	150x150 pixels	500x500 pixels
montecarlo	3605	14	10,000 samples	60,000 samples
hedc	25040	462	566-byte input fle	-

by another thread. See Figure 1. Edges are typically added to graph in the *monitor_contended_enter* callback. Edges are removed from the graph in the *monitor_contended_entered* callback. After adding an edge, we check whether a cycle was formed. In this case a deadlock may have occurred. The formation of a cycle does not always mean that a deadlock occurred. There is a delay between an event of interest happening in the JVM and the JVMTI calling DDS callbacks; the agent may have outdated information about the running Java program. Our experiments indicate that this delay is in the order of 10 to 20 milliseconds; however, this delay does not affect the validity of our analysis because deadlock is persistent. Once a deadlock occurs, it will be hold until the agent resolves it. For this reason, we recheck the conditions of a circular hold-and-wait pattern after an additional delay of 20 ms. After verifying that a deadlock has in fact occurred, we use our Deadlock Solver Agent to remove the deadlock. Our algorithm for cycle detection uses a variant of depth first search to search the graph in O(V+E) time.

The implementation of DDS uses the Java Native Interface (JNI) to “call out” from Java code to external functions and to “call in” from external functions into Java code. Using the calling-in capability of the JNI our agent issues a function call to a thread holding a lock involved in a circular hold-and-wait. We specifically call the *wait* method forcing the receiving thread to release one lock in the hold-and-wait cycle. This thread will be awaken and given back the lock after the lock is again available. As of now, we chose the “victim” thread randomly; however, in the future we plan to use static analysis to identify threads whose locks can be safely removed. In particular, we seek to identify program locations were the acquisition of a second lock by a thread occurs after structures involved in a first lock acquisition were accessed and modified. In that case, the thread is not a good candidate for victim.

3 EMPIRICAL RESULTS

The goal of the DDS is to detect any deadlock that may have occurred in the system during runtime and to resolve it effectively. The agent’s effectiveness is measured on the basis of (1) its functional accuracy and (2) the overhead that was imposed on the monitored system. All the experiments reported below were carried out on a Linux Ubuntu 16.04 LTS machine with 2.20 GHz Intel Core i7 processor and 6 GB RAM. So far, the agent has shown to accurately detect and resolve all deadlocks that occurred in our benchmarks.

The performance of the toolkit was evaluated on two different sets of experiments. The first set was based on two versions of dining philosophers program. The second set contained benchmarks obtained from two different sources. First, we used the benchmarks Moldyn, Raytracer, and Montecarlo from the Java Grande suite [2]. Three additional benchmarks were obtained from ETH Zurich [1], namely, Sor, Hedc, and Elevator. We ran each benchmark thirty times; the average run times and elapsed times with and without DDS are reported in Table 1. To double check the accuracy of our measurements, we independently measured the run time of the DDS agent as well. A non-deadlocking alternative of the philosophers program was used as the control program to determine the raw CPU time and also the elapsed time for each program. In reference to the results obtained, we observed an average increase of 2.7% in the CPU time when a non-deadlocking program was run with the agent supervision. The increase in elapsed time was measured at 10.9%. Moreover, we noted a linear relation between DDS’s overhead and the number of philosophers. The numbers indicate that our approach is actually quite scalable.

In addition to our control experiment, we also chose six different benchmarks to calculate the DDS overhead on real-world applications. DDS was evaluated using two input sizes as illustrated in Table 1. Different thread numbers were used in the evaluation, except for the Hedc benchmark which used a single input size. As illustrated in Table 2, the overhead in the elapsed time ranged from 0.2% up to 8%. Further, the CPU time overhead for both input sizes and the number of threads was observed to be between 0.4% and 7% except for Hedc benchmark which was as high as 16%.

4 CONCLUSIONS

Our results indicate that the run-time monitoring approach underlying DDS has considerable potential for addressing the problem of deadlocks in Java programs. DDS’s average performance overhead was quite modest, 5.6% overall, which indicated that DDS does not have adverse effects on program efficiency. The absence of a deadlocks does not result in significant slowdown in the execution of the monitored Java program. Furthermore, this approach can be extended to other programming languages that use object locking for synchronization. In the future, we plan to combine our approach with static analysis in order to ensure data consistency in the thread whose lock is forcibly removed.

Table 2: Empirical results for real-world applications. Two input sizes are reported for each benchmark. Columns report thread count, average CPU and elapsed time in seconds, average overhead percentage of agent for CPU and elapsed time. Input set sizes are identical to the case of Table 1.

Bench. Name	# Ths.	First Input Size Set				Second Input Size Set			
		Avg. CPU	Avg. Elps.	CPU OH%	Elps. OH%	Avg. CPU	Avg. Elps.	CPU OH%	Elps. OH%
elevator	4	0.24	21.72	0.42	0.55	0.36	54.13	0.57	0.21
	8	0.33	21.72	0.91	0.52	0.45	51.43	0.62	0.21
	16	0.39	21.72	1.79	0.50	0.56	51.43	1.34	0.22
sor	4	1.4	0.33	3.21	4.85	3.98	1.03	0.50	1.15
	8	2.90	0.40	5.08	4.88	8.35	1.17	1.93	2.05
	16	22.85	2.90	7.81	7.82	24.29	3.16	6.74	6.7
moldyn	4	6.94	0.88	5.01	3.13	52.12	6.54	1.32	2.38
	8	11.66	1.45	2.49	3.97	60.90	7.65	0.97	1.94
	16	28.47	3.58	5.35	5.29	100.75	12.63	4.38	4.92
raytacer	4	3.05	0.41	3.28	2.44	26.53	3.38	1.51	3.07
	8	3.63	0.48	3.80	4.18	27.69	3.53	2.37	4.04
	16	5.74	0.75	3.71	3.8	29.30	3.73	1.98	4.09
montecarlo	4	4.24	0.64	1.60	1.32	22.87	3.25	1.36	1.55
	8	4.34	0.65	2.07	0.95	23.08	3.16	1.10	1.77
	16	4.35	0.64	3.43	2.34	23.487	3.20	1.49	3.06
hedc	4	0.57	0.88	15.58	3.04	-	-	-	-
	8	0.65	1.03	16.18	1.29	-	-	-	-
	16	0.91	1.55	6.28	9.0	-	-	-	-

REFERENCES

[1] [n. d.]. Areas of Research in Computer Science. ([n. d.]). https://www.inf.ethz.ch/research.html

[2] [n. d.]. Java Grande benchmark suite | EPCC at The University of Edinburgh. ([n. d.]). https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite

[3] Rahul Agarwal, Liqiang Wang, and Scott D Stoller. 2005. Detecting potential deadlocks with static analysis and run-time monitoring. In *Haifa Verification Conference*. Springer, 191–207.

[4] Saddek Bensalem and Klaus Havelund. 2005. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conference*. Springer, 208–223.

[5] Yan Cai and WK Chan. 2012. MagicFuzzer: Scalable Deadlock Detection for Large-scale Applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 606–616. http://dl.acm.org/citation.cfm?id=2337223.2337294

[6] Yan Cai and WK Chan. 2014. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering* 40, 3 (2014), 266–281.

[7] Yan Cai, Changjiang Jia, Shangru Wu, Ke Zhai, and Wing Kwong Chan. 2015. ASN: A Dynamic Barrier-Based Approach to Confirmation of Deadlocks from Warnings for Large-Scale Multithreaded Programs. *IEEE Transactions on Parallel and Distributed Systems* 26, 1 (Jan. 2015), 13–23. https://doi.org/10.1109/TPDS.2014.2307864

[8] Yan Cai and Qiong Lu. 2016. Dynamic Testing for Deadlocks via Constraints. *IEEE Transactions on Software Engineering* 42, 9 (2016), 825–842. https://doi.org/10.1109/TSE.2016.2537335

[9] Mahdi Eslamimehr and Jens Palsberg. 2014. Sherlock: Scalable Deadlock Detection for Concurrent Programs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 353–365. https://doi.org/10.1145/2635868.2635918

[10] Klaus Havelund. 2000. Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. Springer-Verlag, London, UK, UK, 245–264. http://dl.acm.org/citation.cfm?id=645880.672085

[11] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. 2010. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 327–336. https://doi.org/10.1145/1882291.1882339

[12] Daniel Marino, Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, and Jan Vitek. 2013. Detecting Deadlock in Programs with Data-centric Synchronization. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 322–331. http://dl.acm.org/citation.cfm?id=2486788.2486831

[13] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*. 11–20. https://doi.org/10.1145/1985793.1985796

[14] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad Hoc Synchronization Considered Harmful. In *OSDI*, Vol. 10. 163–176.

[15] Zhen Yu, Xiaohong Su, Tiantian Wang, and Peijun Ma. 2016. Mocklinter: Linting mutual exclusive deadlocks with lock allocation graphs. *International Journal of Hybrid Information Technology* 9, 3 (2016), 355–374.

[16] Hao Yue and Keyi Xing. 2014. Robust supervisory control for avoiding deadlocks in automated manufacturing systems with one specified unreliable resource. *Transactions of the Institute of Measurement and Control* 36, 4 (2014), 435–444.

[17] Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai, and Tongping Liu. 2017. UNDEAD: Detecting and Preventing Deadlocks in Production Software. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 729–740. http://dl.acm.org/citation.cfm?id=3155562.3155654