International Journal of Computer Science and Software Engineering (IJCSSE), Volume 8, Issue 11, November 2019
ISSN (Online): 2409-4285 www.IJCSSE.org Page: 276-279

Dynamic Data Race Detection in Java Applications

Mehmet S Unluturk[1], Ilhan Sofuoglu[2], Rifat Atar[3], Emre Nakilcioglu[4] and Abdulhakim Unlu[5]

[1, 2] Department of Software Engineering, Yasar University, Izmir, Turkey
[3, 4, 5] Ericsson ITU Ayazaga Kampusu Koru Yolu, ARI2 Teknokent Binasi B Blok 4-1, 34390, Maslak, Istanbul, Turkey

[1, 2] mehmet.unluturk@yasar.edu.tr, [3, 4, 5]rifat.atar@ericsson.com

ABSTRACT

A data race is occurred when two threads concurrently access a shared variable and at least one of those threads is the write thread. Data race related bugs are hard to catch because they can happen only under very specific conditions. Even the multi-threaded application passes all the tests successfully, it does not mean that the data races are all removed from the source code. As a result, it is important to have tools to catch the existing data races and the new ones as soon as they are added into the source code. The problem of catching data race is known to be NP-hard. There are number of approaches for catching the data races. Basic ones are the static, on-the-fly and postmortem. Two multithreaded software applications developed at Ericsson Turkey called Notification Engine (NE) and Multi-Mediation (MM) have been already tested by using unit tests and regression tests. However, these two applications had never been investigated using a data race detector. Our main objective here is to implement and deploy a continuous methodology for finding data races in these kind of multithreaded applications. Furthermore, improvement in the performance of the proposed analysis tools and a significant reduction of false positives have been achieved.

Keywords: *Dynamic Data Race Detection, Static Data Race Detection, Multi-threaded Applications, On-the-fly Data Race Detector, Post-mortem Data Race Detector*.

1. INTRODUCTION

A thread can be described as a sequential flow of control within a program that starts running at point A and continues to run until point B. Before the introduction of the multithreaded programs, computers were able to do just one job at a time. All threads were running in a sequential order at a given time and more than one thread was not able to run concurrently. After the multithreading concept starts, multithreading has become a very useful programming technique that allows multiple threads to be used concurrently. This programming technique provides high responsivity, efficient resource sharing, scalability, etc. For instance while a webpage is loading, other threads can help to edit photos or playing music with the help of multithreading techniques. Although a multithreading program provides better performance than a single threaded program, these techniques have also few disadvantages. They are programming complexity, testing and debugging, deadlocks and data races.

Data race occurrence is increased by extensive usage of multithreaded programming languages like Java. Data races are the most critical issues in multithreading because it causes data inconsistency and directly affects the flow of the program. Data races occur when at least two threads try to access the same memory location where at least one of those accesses is the write access. The order of threads accesses are not deterministic and computation results can be different in different runs of the same program. Therefore, data races are hard to detect but there is a common solution called synchronization mechanisms to prevent data races. For decades, different data race detection algorithms have been proposed and they can be divided into two groups: static data race detection algorithms and dynamic data race detection algorithms. Both have advantages and disadvantages. In static data race detection algorithms, the source code is analyzed without running the Java program and shared variables are detected. There are static data race detection studies in the literature [1][2][3][4][5][6][7] to detect data races. The static data race detection algorithms are offline. Therefore static analysis does not consume too much time unlike dynamic data race detection algorithms. Especially, if there is a huge amount of code, static analysis is more advantageous than that of dynamic analysis. On the other hand, static algorithms cannot gather information about states of memory and program paths during execution, therefore static algorithms produce more false positive errors than those of dynamic algorithms. Dynamic data race detection algorithms analyze the Java program during its execution. Dynamic algorithms need much more computational power and operating time than those of static algorithms. Dynamic data race detection algorithms find data races on the executed paths. In dynamic analysis, there are two essential algorithms called as happens–before and lockset.

Lamport's happens-before algorithm [8] is one of the dynamic data race algorithms utilized in this research. There are numerous studies which use happens before algorithms [9][10][11][12][13][14]. Happens-before

algorithm tries to find data races by examining the order of events in a distributed system. According to Lamport, there is a causality relationship amongst concurrent events. The Lamport clock utilized in the algorithm has the essential information about the ordering of these events. This clock presents if an event has happened before another event. Furthermore, the Happens–before algorithm only finds data races on the executed path. On the other hand, the Lockset algorithm is more accurate approach than that of Happens-before. Eraser's Lockset algorithm [15] keeps track of a set of candidate locks for each variable. Lockset algorithm detects violations of locking disciplines. The Lockset algorithm is the second algorithm used in this research together with the Happens-before algorithm to generate less false-positives in the proposed analysis tool to catch the data races in two of the Ericsson's Java applications.

A hybrid approach is used that utilizes both Lockset and Happens-before algorithms [16][17][18][19][20]. In these studies, the number of false positive error reports have been reduced. In general, the Hybrid algorithms try to find more true-positive data races. Moreover, these studies tried to develop efficient and fast algorithms but the performance problem has been arisen due to the fact that these two algorithms operate at the same time. The post-mortem algorithms are another type of data race detection algorithms. They try to catch the data races by logging memory accesses and analyzing them at the end of the program execution.

This paper is organized as follows. Section 2 describes the preliminary concept and motivation of this research. Section 3 presents the experimental results. Finally, Section 4 summarizes and concludes this paper.

2. MATERIALS AND METHODS

This paper presents a next-generation dynamic analysis tool that scales with the complexity of future mobility systems, covering multi-core targets and industrial standards for model-driven development proves advanced safety properties of concurrent systems. Ericsson Turkey is the leading provider of Telecommunications equipment, software and services. Because of its experience in Telecommunications software, Ericsson aims to contribute by bringing its Java applications from this domain, applying the implemented algorithms on these software applications, working on fine tuning of these algorithms, as well as on the development of new algorithms. These applications are explained in the following sections.

2.1 Application #1 Notification Engine

Notification engine (Figure 1) is a solution based on event engine framework, an asynchronous, event driven state machine, persisted in the database. It is used in a Telco environment to notify subscribers or other relevant systems of a telecom operator on events that they are interested in.

Typical example is to relay SMS message notifications to subscribers, when their packages are renewed every month. In a telecom environment, on average 5 million events (9.5 million peak) are dispatched to event engine daily. These events must be processed and resulting notifications must be dispatched within the order of seconds. Event engine framework works in a multithreaded manner to provide this throughput. Multiple threads are executing to process a very high number of events and there exist possibilities of data race conditions. A hybrid dynamic data race detector has been developed to handle synchronization primitives and analysis of data race conditions.

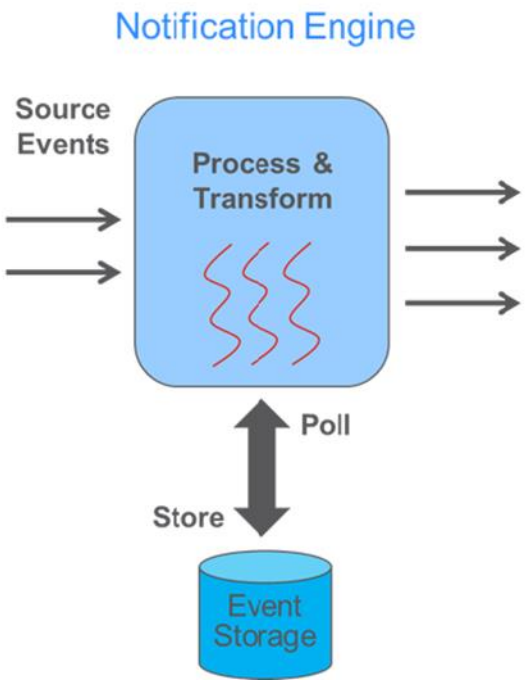

Fig. 1. Application #1 Notification Engine (NE)

2.2 Application #2 Multi-Mediation

Multi-Mediation (MM) is a business critical Ericsson software application used for multiple purposes, like real time charging, billing or data warehousing (Figure 2). MM can collect data from multiple sources in multiple formats, process data as per the operator's business needs and distribute the processed data to a desired system. MM should be able to run concurrently under high load, provide near real-time response, and support 99,999 % availability. Especially, the real-time charging workflows

have zero tolerance against any defects due to data race conditions, due to direct impact on the revenue.

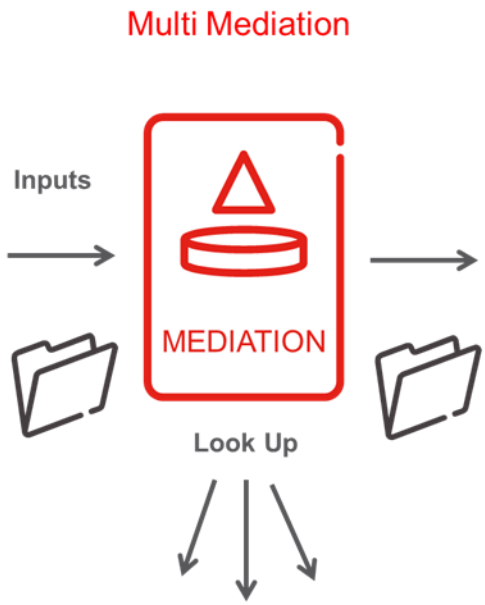

Fig. 2. Application #2 Multi-Mediation (MM)

2.3 DRDCheck Tool

The DRDCheck tool [21] which is an open source tool was utilized to detect the data races in the above two Java applications. The DRDCheck tool implements Happens-before algorithm [8]. A data race occurs if multiple threads access a shared variable concurrently and at least one of them is a write access. The purpose of Happens-before relationship is to establish partial ordering of events across different threads, such as, an unlock operation happens before a lock operation. Two threads accessing to a shared variable concurrently can create a data race if they cannot be ordered using Happens-before. Furthermore, there might be false-positive reports due to the detection depends on the scheduler. Performance impact is high due to high instrumentation.

2.4 Hybrid Algorithm

In this paper, a hybrid algorithm based on the above open source DRDCheck tool was developed to test the same two Java applications. Hybrid algorithm includes the LockSet algorithm to reduce the number of the above false alarms. The LockSet presents a different approach and defines the data race as an access to a shared variable that is not controlled by lock(s) [15]. Below is the proposed hybrid algorithm:

```
Bool IsRace (Write, Read)
        If (Write happens-after || unrelated to Read) &&
(Write-Lockset ∩ Read-LockSet = 0)
                Return true;
        Return false;
```

where || represents the logical OR, && represents the logical AND and ∩ represents the intersection operator of two sets Write-Lockset and Read-Lockset. Next section presents the results of both analysis tools.

3. RESULTS

The two tools were utilized to test the Application #1. Results are presented in Table 1. The results for the Applications #2 for the same tools are also depicted in Table 2.

Table 1: Comparison of DRDCheck and Hybrid Tool for Application #1

Dynamic Data Race Tools	# of Distinct Races	False Positives	Response Time per Request
DRDCheck Tool	47	43	131 msec
Hybrid Tool	30	2	120 msec

With the DRDCheck tool, out of 47 data races detected, 43 of them were false positives. And the response time per request for Application #1 was measured as 131 msec. When the proposed Hybrid tool was utilized, the number of false positives was decreased (from 91% to 6%) and the response time was measured as 120 msec which is 10% less than that of the DRDCheck tool.

Table 2: Comparison of DRDCheck and Hybrid Tool for Application #2

Dynamic Data Race Tools	# of Distinct Races	False Positives	Response Time per Request
DRDCheck Tool	39	24	698 msec
Hybrid Tool	23	0	70 msec

With the DRDCheck tool for Application #2, out of 39 data races, 24 of them were false positives. And the response time per request for Application #2 was measured as 698 msec. When the proposed Hybrid tool was utilized, the number of false positives was decreased dramatically (from 61% to 0%) and the response time

was measured as 70 msec which is 10x less than that of the DRDCheck tool.

4. CONCLUSIONS

In this research, a Hybrid algorithm was proposed to detect the data races in the Ericsson's two of the mission-critical Java applications. Applications were tested by the DRDCheck tool first and later, they were evaluated with the Hybrid algorithm. According to the results, running the Hybrid algorithm provides good results with low number of false-positives in a fractional amount of time.
It can be inferred that the most accurate method to catch the data races in a multi-threaded Java source code consists of running both tools, DRDCheck and the Hybrid algorithm, in succession. The future work will be testing the following methodology to detect the data races in any Java multi-threaded applications:

- Start with the pure happens before algorithm (DRDCheck tool)
- Once all the race reporting is fixed in the source code, run with DRDCheck tool again,
- A few false reports may be generated which can be easily eliminated,
- Then, use the Hybrid algorithm

ACKNOWLEDGMENTS

This study was supported by Scientific and Technical Research Council of Turkey (TUBITAK), Project No: 9150181, Project Name: ITEA3-ASSUME: Affordable Safe & Secure Mobility Evolution.
Authors acknowledge the support of the company Ericsson Turkey for the use cases they provided.

REFERENCES

[1] Internet: V. Vojdani, "Static Data Race Analysis for C", http://kodu.ut.ee/~varmo/seminar/sem09S/final/vojdani.pdf, 07.10.2019
[2] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith", ACM Trans. Program. Lang. Syst., 2011, vol. 33, no. 1, pp. 1–55
[3] J. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code", Fse, 2007, pp. 205–214
[4] C. Radoi and D. Dig, "Effective Techniques for Static Race Detection in Java Parallel Loops", ACM Transactions on Software Engineering and Methodology, 2015, vol. 24, no. 4
[5] C. Radoi and D. Dig, "Practical Static Race Detection for Java Parallel Loops", Issta'14, 2013, pp. 178–190
[6] V. Kahlon, , N. Sinha, E. Kruus and Y. Zhang, "Static Data Race Detection for Concurrent Programs with Asynchronous Calls", Proc. 7th Jt. Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng., 2009, pp. 13–22
[7] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks", Race, 2003, vol. 37, no. 5, pp. 237–252
[8] L., Lamport, "Time, clocks, and the ordering of events in a distributed system", Commun. ACM, 1978, vol. 21, no. 7, pp. 558–565
[9] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks", Inf. Process. Lett., 1992, vol. 43, no. 1, pp. 47–52
[10] K. Zhai, B. Xu, W. K. Chan and T. H. Tse, "{CARISMA:} a context-sensitive approach to race-condition sample-instance selection for multithreaded applications", Proc. 2012 Int. Symp. Softw. Test. Anal., 2012, pp. 221–231
[11] M. D. Bond, K. E. Coons and K. S. McKinley, "Pacer: Proportional detection of data races", Pldi, 2010, no. 2, pp. 255–268
[12] L Effinger-Dean, B. Lucia, L. Ceze, D. Grossman and H. J. Boehm, "IFRit: interference-free regions for dynamic data-race detection", Oopsla'12, 2012, p. 467
[13] C. Flanagan and S. N. Freund, "FastTrack : Efficient and Precise Dynamic Race Detection", Pldi'09, 2009, vol. 44, no. 6, pp. 121–133
[14] S. Biswas, M. Zhang, M. D. Bond and B. Lucia, "Valor: efficient, software-only region conflict exceptions", OOPSLA 2015 Proceedings 2015 ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl., 2015, vol. 50, no. 10, pp. 241–259
[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs", ACM Trans. Comput. Syst., 1997, vol. 15, no. 4, pp. 391–411
[16] Yu, M. and Bae, D.H.: 'SimpleLock : Fast and Accurate Hybrid Data Race Detection', Comput. J., 2016, vol. 59, no. 6, pp. 793–809
[17] X. Xie and J. Xue, "Acculock: Accurate and Efficient Detection of Data Races", CGO '11Proceedings 9th Annu. IEEE/ACM Int. Symp. Code Gener. Optim., 2011, pp. 201–212
[18] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice", Proc. Work. Bin. Instrum. Appl., 2009, pp. 62–71
[19] M. Yu, J. S. Lee and D. H. Bae, "AdaptiveLock: Efficient Hybrid Data Race Detection Based on Real-World Locking Patterns", Int. J. Parallel Program., pp 1–33, https://doi.org/10.1007/s10766-018-0579-5, 2018.
[20] R. O'Callahan and J. D. Choi, "Hybrid Dynamic Data Race Detection", PPoPP, 2003, vol. 38, no. 10, pp. 167–178
[21] D. Tsitelov and V. Trifanov, "Dynamic data race detection in Java-programs using synchronization contracts", Proc. - 2013 Tools Methods Progr. Anal. TMPA 2013, 2015, pp. 3–8.