Data Race Detection Techniques In Java: A Comparative Study

Devesh Lowe, Dr. Mithilesh Kumar Dubey, Bhavna Galhotra

Abstract: Most modern programming languages support multi-threading, which is commonly required in executing parallel programs in mobile multi-media applications, web-servers and operating systems. Concurrency bugs are common in multi-threaded applications. Bugs like race conditions, data races and deadlocks are some problems which arise out of unsynchronized code but have to capacity to sneak in to large and complex structures and may result in giving unexpected results. Data Race is one such synchronization bug, arising out of unsynchronized code, which can stay hidden even after many rounds of testing, but can emerge as a system destroyer when it surfaces. Researchers, in last four decades have presented various tools to detect data races which are broadly classified into static, dynamic and hybrid categories. In this paper authors have focussed on data races arising in object oriented programming language- java and have presented a study of various research articles subjected towards early detection of data races. Authors hereby present a study of techniques used by different researchers and have tried to summarise the java oriented approaches to solve the problem. Authors also present a comparison on different java based tools used for data race detection.

Index Terms: Parallel Processing, Concurrency Bug, Race Condition, Dynamic Data Race Detection, Java

1. INTRODUCTION

Multi-threaded programs are the main stay of modern programming. Most modern programming languages support multithreaded applications, which definitely improves the responsiveness of programs. Use of multi-threaded applications give rise to concurrency bugs like race conditions, data races and deadlocks. Data race is one of the most common concurrency bug that is affecting multithreaded programming since its inception. A data race arises when two or more threads try to access a shared resource and one of them is in write mode [1] i.e. trying to change or update the shared resource. Data race is notoriously difficult to detect as ordering of thread execution is decided by operating system where program or programmer don't get any access. A data race if not handled properly has the potential to even crash a system. All parallel programming languages provide mechanisms which help to prevent data races. It is important that a data race caused by an imperfect code must be detected early to prevent them from causing much harm to the system.

In many situations, data races and race conditions are often mistaken for each other. There is a peculiar causal and non-causal relationship between Race Conditions and Data Races. Many race conditions are caused by data races and similarly many data races are caused by race conditions. There are also situations where data races are independent of race conditions and vice-versa. Race Condition occurs when timing or ordering of events gets affects by other undesirable or uncontrollable events resulting in hampered programs correctness. For example situations created by context switch, OS signals, hardware interrupts and memory operations on multi-processors can produce race conditions. Data race is more of a programming error of synchronization. As explained by Sebastian Burckhardt at Microsoft Research, Data Race happens when there are two memory accesses in a program where both: [2]

- Target the same location
- Are performed concurrently by two threads
- Are not reads
- Are not synchronization operations

Various data race detection techniques have been developed in last four decades which have been instrumental in early detection of this anomaly. Researchers have categorised these detection techniques as on-the-fly, ahead of time and post mortem techniques [3]. Broadly classified as Static and Dynamic techniques. Static data race detection techniques or ahead of time techniques use compile time heuristics. On-the-fly approaches are dynamic in nature. Considering the frequency of occurrences of data races in a multi-threaded application, most studies have focused upon dynamic detection of data races keeping a check over the number of false positives as minimum. We observed that race detectors or tools developed by various researchers were primarily enhancements over previously written tools with an attempt to minimize false positives. Since data races are more frequent in multi-threaded

- *Devesh Lowe is currently pursuing Phd and is an assistant professor, Jagan Institute of Management Studies, Rohini, New Delhi, India, 110085,devesh.lowe@jimsindia.org*
- *Mithilesh Kumar is an Associate Professor, Lovely Professional University Punjab, India, mithilesh.21436@lpu.co.in*
- *Bhavna Galhotra is pursuing Phd and is an assistant professor, Jagan Institute of Management Studies, Rohini, New Delhi, India, 110085, bhawna.galhotra@jimsindia.org*

programming architecture, many researchers have focussed on developing detection algorithms based on programming languages like java [4] [5] [6] which has been a main stay of object oriented programming based model development. In this paper authors have presented an outline of problems caused by this synchronization bug in java and have presented a study on various data race detection algorithms oriented towards race detection in java programming language. Section II of this paper focuses on how data races are caused in unsynchronised code in java [7]. It also discusses the critical section of program where data races may be hidden and highlights the existing tools used available in java to debug a data race. Section III provides an insight on data race detection techniques based on java as developed by various researchers. In section IV authors provide a summarized view of various techniques and a comparative analysis [1].

2. DATA RACES IN JAVA

Java is primarily a multi-threaded programming language which calls for parallel execution of threads. These threads execute same piece of code but independent of each other using some part of shared accessed memory. Threads are executed using a single or multiple hardware setup using the feature of time-slicing for parallel execution. For inter-thread communication and to avoid any run-time confusion regarding resources Java Virtual Machine uses a concept of synchronization in form of a monitor which is attached to every object in java which only a thread can lock or unlock. At any given time only one thread is allowed to lock a monitor and any other thread cannot unlock that particular monitor until the lock is held by previous monitor. A lock on an object is acquired using a synchronized statement which hold all other operations until a lock action is completed. An unlock action is only performed once the body of synchronized stamen is fully completed either sequentially or abruptly. Since shared memory is accessed in parallel mode, it gives a possibility that multiple threads may try to access same location in memory giving chance to a Data Race to happen if code is not properly synchronized for shared resources access.
As defined in Java Memory Model documentation (JMM) [8], a data race is an anomalous situation which arises when two or more threads try to access a common shared resource when one of them is in write mode. This generally happens when incorrectly synchronized programs start throwing counterintuitive results. A common way any compiler handles ordering of execution of threads is using happens before relation [9]. This partial or complete ordering is maintained to guarantee the sequential consistency. Sequential consistency ensures that there is total order over all individual actions making them atomic and completely visible to all threads. This also ensures that all executions of program will be sequentially consistent provided there are no data races. It has also been observed that a data race is a property of execution not of a program. A program which is properly synchronized and uses volatile shared variables is expected to produce no data race but can deviate from expected output during execution. Let's consider an example in java (figure 1) which demonstrates presence of race condition. In the code presented in Figure 1, we can observe data races in every part. Program uses 'x' variable which is not declared volatile. By definition we have possibility of data race in this code as there is no synchronized order in read and write operations of 'x'. Possible outcomes of this program include printing TRUE or False or an indefinite loop. Lamport [9] suggested happens before relationship between two events based on the sequence in which they are reported. In java the wait method of the Object class has lock and unlock actions associated with ordering of events to establish the HB relationship and order of execution. But the presence of HB relationship doesn't guarantee that the events actually happened in the same order. HB relation just defines when data race takes place. An inter-thread action that can be performed by any thread and can be directly or indirectly viewed or affected by another thread can be categorized in java as under:

- Read action: non-volatile read
- Write action: non-volatile write
- Synchronized action: volatile read and write, lock, unlock, start and stop of thread
- External action: action observable outside of its execution
- Thread divergence action: performed by a thread in an infinite loop

```
class Test{
  static boolean x = false;
  static void foo() {x = true;}
  public static void main(String []arg) {
    ForkJoinPool.commonPool().execute(Test::foo);
    while (!x);
    System.out.print(x);
     }
}
```

Figure 1 *Data Race Example*

3. DATA RACE DETECTION IN JAVA

- Problem of data race is generally termed as menace because of various reasons like they are not traceable until the program is moved into execution phase. Hardware and multicore systems involved in development and quality assurance are very different from the actual systems, which increases the volume of data, the unpredictability increases. Also, data race is not reproducible as it depends on the thread timings. But this menace can cause major damage to global data structures because program can continue to work normal with growing damage in the background [10]. Researchers have classified data race detection in three categories as [11]
- Static race detection algorithms (ahead of time)
- Dynamic race detection algorithms (on the fly methods)
- Hybrid race detection algorithms (post mortem techniques)
-
- All the three methods have their own benefits and drawbacks. In the scope of current research we have focussed on all previous work done and implemented using java as object oriented programming language implementation. Remarkable work has been done in the past where java based algorithms were used to detect data races. In the following sub-section we present the objective and outcomes of these algorithms as measured by researchers. [12]
-

- Static Race Detection
- A static technique for race detection is not dependent for program to execute. Since it makes its way into all the possible branches of code, it also explores the possibility of presence of a data race in that part of code which would otherwise rarely comes into execution. Many techniques are based on strong type checking [13] [14] [15] [16], where authors provided a strong type-checking mechanism to avoid data race and deadlock. They proposed a system where locking system for multi-threaded application is provided in form of type principle. They also proposed ownership classes for locking to be arranged in form of tree based data structure for partial ordering for classes. Use of ownership classes helps in controlling data races as lock that protects the object also protects its encapsulated objects. The implemented system was based on JVM with strongly typed annotations but also had a limitation that it was only available for java [3].
- Flanagan and Qadeer [4] suggested a stronger non-interference property atomicity which was to be included as annotation in statement block and functions with keyword atomic. This property guaranteed the execution of all atomic functions independently of each other in an interleaved multi-threaded program. This greatly reduced the interaction between threads. Its implementation is available in java.util.vector library. Its limitation was that it cannot be used independently i.e. its dependence on other special programming language and it also needed more tools to implement. David Clarks et al. [17] proposed a static analysis approach based on java which used shared variables in programs and summarizes them, and identifies to detect usage in data races. It prepares a comparison of these shared variables and categorises them as per whether data race may exist or a data race must exist.
- The benefit of using static techniques is that they analyse the most part of code and are able to detect every possible race. But researchers [18] also note that most static techniques have a tendency to raise a lot of false positives. So current programming techniques are of little help to programmers.

4. Dynamic Race Detection

- Dynamic data race detection algorithms are based on on-the-fly techniques i.e. program analysis. Since it traverses through the most visited paths and checks only those shared resources which are used in the execution of the programs, it gives the most accurate results. Probably that's the reason that most data race detection algorithms today are using dynamic data race detection. Moreover static analysis gives a problem of most false positives i.e. maximum false alarms, which dynamic race detectors overcome. But dynamic race detectors do come with extra overheads of computations and line of code. Moreover they can only detect data races that come in the current execution path and simply ignore the races which were not part of the current execution [19]. This has been a major area of concern as the number of feasible paths can grow exponentially under some execution but that is not considered under Dynamic DRD [18]. This opens up the space for many undetected data races. It is also observed [3] that under some instances we find that bulk part of the code never comes into execution leaving scope of undetected data races like in operating systems and device drivers.
- One of the most popular work in field of dynamic data race detection is Eraser [19] which implemented lockset based algorithm using binary rewriting technique to monitor shared resources. It is an extension of Lamport's Happens Before [9] algorithms. In original work Lamport defined a partially ordered set of locks for shared resources using synchronization events. Eraser's limitations was identified as extra effort of storing too much information about each thread and too much dependence on synchronization operation.
- Another race detection technique Ravegeddon was presented by Mahdi et. al. [18], where they proposed to race directed scheduling which was applicable beyond the use of java. It was based on created types in java along with concolic functions and helper functions for identifying data races. In TRaDe [20]Mark Christiens suggested an algorithms for identifying topological race detection.
- In 2009, Daniel Marino et. al. [21]presented an algorithms LiteRace, which was a lightweight race detector implemented using Microsoft Phoenix compiler, which samples and analyses only certain portions of execution code and claimed to detect 70% of more data races. Another approach of dynamic detection of data races came with Kaushik et.al.'s paper [22] with an algorithm which executed multiple replicas of the program with complementary thread schedules. Its aim was to detect a data race early and program to diverge. Their system Frost detected data race by comparing the state of replicas executing complementary schedules. Though it provided extra overhead for replication model but authors were able to keep it under control when spare cores were involved for increased utilization.
- With a novel approach of precision based dynamic data race detection authors [23] introduced structured parallelism. Their algorithm uses less space in memory and worked in parallel and utilized fewer resources. Konstantin [6] presented ThreadSanitizer: a dynamic data race detection tool based on hybrid technique as a combination of happens before and Lockset. They introduced dynamic annotations as user API which informs a user about any synchronization bug if occurs. Eric Bodden [5] presented an algorithm Racer which was primarily based on Eraser [19] and proposed an implementation in java using three new pointcuts in aspect oriented language AspectJ. In 2002 Choi et al. [24] presented another dynamic data race detection mechanism which they claimed to be more precise and had lowered the problem of false positives. Also they tried to keep runtime overhead in the range of 13% to 42%.
- Benjamin Wood [25] suggested the use of fast instrument bias which reduces or eliminates the overhead cost of implementing dynamic data race detection algorithms.

Hybrid Race Detection

- Hybrid detection techniques are based on a combination of static and dynamic race detection techniques. It uses the extensibility of static analysis with accuracy of dynamic tools. These techniques analyse log or trace data post execution of the program. They don't suffer from the extra computational overhead as dynamic techniques but have the same limitation i.e. they can detect data races along the execution path only.
- Konstantin [6] presented ThreadSanitizer: a dynamic data race detection tool based on hybrid technique as a combination of happens before and Lockset. They introduced dynamic annotations as user API which informs a user about any synchronization bug if occurs.
- Ronsse and Bosschere [26] presented a hybrid technique for cyclic debugging for non-deterministic parallel programs. They used two phases of program i.e tracing a program execution called record and using thus obtained information for next supported re-execution called Play. Once a data race occurs, it stops the execution and notifies the user. It uses the vector clock for detecting the conflicting memory access by concurrent segments and applies the happens before [9] relations to detect.
- Another similar tool for record and play specifically developed for java was presented by Choi and Srinivas [27]. They introduced a logical thread scheduling which maintains a sequence of intervals of critical events where each interval corresponds to the thread accessing shared resources. After execution it updates the global clock. It was implemented and tested on several java programs and updates of Sun Microsystems' Java Virtual Machine.

5. COMPARATIVE ANALYSIS

- In the current study we have relied upon the analysis and study performed by some noted researchers in their research papers. Mehdi Eslamimehr et al. [18] prepared a comparative analysis on various tools. They compared static detection tool of Mayur Naik et al. Chord [28], dynamic tools like FastTrak, goldilocks, RaceFuzzer and Pacer. Results indicated that static tool explored total 127136 races whereas all four dynamic tools together reported a total of 304 races. This indicated that dynamic tools though more popular but leave scope for more data races to be discovered. They also claimed that best Hybrid technique was as presented by Robert O'Callahan et al [29] which detected most accurate races. Though it also false positives and false negatives but still it provides a manageable size output to coders.
- Misun Yuet al. [1] prepared a comparative analysis on five pure dynamic data race detection techniques namely FastTrack, Acculock, Multi-Lock HB, SimpleLock+ and causally precedes. They performed analysis using same platform and same input. They observed that MultiLock-HB and CP Detection detected all data races accurately and precisely, however they were slow and took significant processing time. SimpleLock+ recorded almost similar detection capability and also had a similar speed of FastTrack.
- Aoun Raza [3] presented a detailed study of various race detection tools. He studied most of the race detection tools and categorized them under heads like on-the-fly, post-mortem and ahead of time techniques. [30]
- On the basis of above mentioned studies we present a comparative analysis of various data race detection techniques in table 1 below.
-

Table 1: *Comparative analysis of various data race detection techniques*

Algorithms	Based on	Common Algorithms	Advantages	Disadvantages
Static	Mostly based on strong type checking.	Rccjava, Relay	Detect almost every possible data race	Slow, Raise False Positives
Dynamic	SISE (single input single Execution) property, Mostly based on Lamport's happens Before, or Lockset based algorithms	FastTrack, AccuLock, RaceFuzzer, Eraser, Racer, SimpleLock+, GoldiLocks, MultiLockHB	Fast, Raise less false positives, Report only real races.	Extra computational overhead, detects only those races which come across the execution path and ignore many undetected
Hybrid	Combines the Techniques of both Static evaluation and Dynamic path evaluation	DejVu, RecPlay, RaceMob	Fast like dynamic algorithms	Can detect data race along execution path only. Many data races go undetected.

6. CONCLUSION

It is noteworthy that researchers have accepted data races and race conditions as problem areas for multi-threaded programs. Efforts are on to prepare suitable detection mechanisms for data races. Application logics have been developed which detect data races in static way or dynamic way but this problem persists. In this paper, authors have focused on problem of data races in java based programs. Authors acknowledge that data races are primarily synchronization bugs which when not detected or handled in-time can generate a system crash or cause inconsistency in vital transactions. In our future work we propose to focus on developing an optimized model for dynamic data race detection which should utilize fewer resources and generate less false positives.

REFERENCES

[1] S. M. P. I. C. D. H. B. Misun Yu, "Expermental performance comparison of dynamic data race detction techniaques," ETRI, vol. 39, no. 1, February 2017.

[2] M. M. S. B. K. O. John Erickson, "Effective Data-Race Detection for the Kernel," Operating System Design and Implementation , 2010.

[3] A. Raza, "A Review of Race detection Mechanisms," in Springer-Verlag LNCS 3967, pp 534-543, Berlin, 2006.

[4] S. Q. C Flanagan, "types of atomicity," in ACM SIGPLAN, 2003.

[5] K. H. Eric Bodden, "Racer: effective race detector using AspectJ," in ISSTA 08, Seatle, USA, 2008.

[6] T. I. konstantin serebryany, "thread Sanitizer i data race detection technique," in ACM 978-1-60558-793-6/12/9 WBIA 09, New York, 2012.

[7] A. Jimborean, P. Ekemark, J. Waern and S. Kaxiras, "Automatic Detection of Large Extended Data-Race-Free Regions with Conflict Isolation," IEEE Transactions on Parallel and Distributed Systems, vol. Volume 29 , no. Issue 3 , March-2018.

[8] "Chapter 17, threads and locks," [Online]. Available: https://docs.oracle.com/javase/specs/jls/se10/html/jls-17.html#jls-17.4.

[9] L. Lamport, "Time Clocks and ordering of events in distributed systems," CACM, vol. 21, no. 7, pp. 558-565, 1978.

[10] D. T. Vitaly Trifanov, "Data race detection in concurrent java programs," in SEE-SECR, Russia, 2012.

[11] A. Janessari, "Detection of High Level Synchronization anomalies in parallel programs," Springer- International Journal of parallel programming, vol. 43, pp. 656-678, 2015.

[12] U. M. V. Dileep Kini, "Data race detection on compressed traces," ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, no. ISBN: 978-1-4503-5573-5, 2018.

[13] M. R. C Boyapati, "A parameterized type system for race free java program," ACM conference on OOP languages, 2001.

[14] S. N. F. C Flanagan, "Type Based Race Detection for Java," in Programmin gLanguage des Implementation, Vancouver, Canada, 2000.

[15] M. A. C Flanagan, "Object Types against Races," in Conf Concurrency theory, Eindhovan, Netherland, 1999.

[16] T. R. G. C Von Praun, "Static conflict analysis for multithreaded object oriented programs," in Conf. Programming languages des implementations, San diego, CA, USA, 2003.

[17] T. M. A. M. David Clark, "Using "must" and "may" summaries to detect data races in Java Bytecode that does not rely on synchronized consgtruct," in ASWEC 15, 2015.

[18] J. P. Mahdi Eslamimehr, "Race Directed Scheduling ro Concurrent Programs," in ACM PPoPP 14, 978-1-4503-2656-8/14/02, Florida, USA, 2014.

[19] M. B. g. n. P. S. T. A. Stefan Savage, "Eraser: A Dynamic data race detector for multithreaded program," in ACM Transactions on Computer systems, Vol 15, No. 4, pages 391-411, 1997.

[20] K. D. B. Mark Christienss, "TRade: dagta racef detection for java," in Springer-Verlag ICCS 2001, LNCS 2074, pp 761-770, Berlin Heidelberg 2001, 2001.

[21] M. M. S. N. Daniel Marino, "LiteRAce: effective Sampling for Lightweight data race detection," in ACM 978-1-6058-392-1/09/06, Dublin Ireland, 2009.

[22] P. M. C. J. F. S. N. Kaushik Veeraraghavan, "Detecting and surviving data races using complementary schedules," in SOSP 11, Cascais, Portugal, 2011.

[23] J. Z. V. s. M. V. E. Y. Raghavan Raman, "Scalable and precise dynamic data race detection in structured parallelism," in ACM PLDI 12- 978-1-4503-1205-9/12/06, Beijing, China, 2012.

[24] K. L. A. L. V. S. M. S. R. O. Jong Deok Choi, "Efficient and Precise Data Race detection for Object Oriented Programming Languages," in PLDI 02 ACM 1-58113-463-0/02/0006, Berlin, 2002.

[25] M. C. M. B. G. Benjamin P Wood, "Instrumentation Bias for Dynamic Data Race Detection∗," in ACM OOPSLA, 2017.

[26] K. D. B. M ronse, "RecPlay: A fully integrated practical record player system," in ACM Transactions on computer systems, 1999.

[27] H. S. J D Choi, "Deterministic replay of java multi-threaded applications," in SIGMETRICS Symposium on parallel and Distributed tools, 1998.

[28] A. A. J. W. Mayur Naik, "Effective static race detection for Java," in ACM Sigplan: Conference on Programming language design and implementation, 2006.

[29] J. D. C. Robert O'Callahan, "Hybrid Dynamic Data Race Detection," in PPoPP '03, ACM 1-58113-588-2/03/0006, San Diego, California, 2003.

[30] N. G. W. O. S. SAM BLACKSHEAR, "RacerD: Compositional Static Race Detection," ACM Program. Lang, 2018.

[31] S. N. F. Cormac Flanagan, "FastTrack: Efficient and Precise Dynamic Race Detection," in ACM 978-1-60558-392-1/09/06, Dublin , Ireland, 2009.

[32] C. Z. G. C. BarisKasikci, "RaceMob: Crowdsourced Data Race Detection," in ACM SOSP'1, Fermington, Pennsylvania, USA, 2013.

[33] Z. h. Q. H. P. W. K Leung, "Data Race: Tame the Beast," Journal of Supercomput, vol. 51, pp. 258-278, 2010.

[34] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," in Communications of ACM, 1974. Vol 17, No. 10.

[35] D. D. P. M. c. J. F. S. N. Benjamin Wester, "Parallelizing Data Race Detection," in ACM ASPLOS'13 978-1-4503-1870-9/13/03, Houston, Texas, USA, 2013.