An Empirical Study of Data Race Detector Tools

Jalal S Alowibdi[1,2],Leon Stenneth[1]
1. Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607
E-mail: {jalowibd,lstennet@cs.uic.edu}
2. Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, KSA

Abstract: The industry of software applications has been increased significantly because of the high demand of using the software applications. This revolution leads on developing many concurrent software systems. Noticeably, some of these concurrent software systems have falsely report data race condition to one or more of their shared variables. Debugging such concurrent software systems to find the race condition is a challenge, especially for large and complex software systems. Since the race condition concerned mostly ignored in the concurrent software systems, adopting it could help to ensure the efficiency of these software systems. There are few detector tools that have been known in the industry focusing on data race detectors. This paper aims to study those tools. We are going to conduct empirical study of data race using well known tools in order to measure the correctness, performances and effectiveness of those tools in practical by using some benchmarks. Those benchmarks will be tested on each tool and compare it with others to see the similarity and differentiate.

Key Words: Data race, Performance, Correctness, Concurrent program

1 INTRODUCTION

When two or more processes are sharing the same variables and each one of them is trying to access the variable before the others at the same time without proper synchronization, this behaviour leads to false value of variable for one of the processes which is called race condition [1]. There are three types of data race detector tools. Static data race detector tools that analyze the source code of the program, study the behaviour of the program and report all the possible race condition. This type of detector tools target the source code level and cannot be analyzed at the run-time compile. Oppositely, dynamic data race detector tools that lead the program to be executed and then analyze and study the history of its memory accesses behaviour in order to predict the race condition. This can be done at the run-time compile [2]. The third type is the hybrid data race detector tools which lead to combination of both static and dynamic data race detector tools behaviour. Because of the increase of the race conditions among the concurrent software systems which leads to faulty behaviour, there are quite few race detector tools available that can effectively analyze the behaviour of the software systems and report back with the possible race condition and some suggestion if needed. Understanding these detector tools might help the developers to pay more attention and try to get the race condition out of the software system as soon as possible. Using one of those detector tools to analyze and debug the product before the delivery, could prevent the race condition that might lead to major problem.

In this research, we aim to explore and provide a concrete study of different data race detector tools that have been used in the industry. We study the behaviour and the performance of each tool by considering the same benchmarks. In our empirical study, we will test five different static data race detector tools where two of the processes can be hybrid. These tools are: RaceFuzzer [3], RacerAJ [4], Jchord [5], Race condition Checker RCC [6] [7], and Java RaceFinder JRF [8].

This paper organized as follows: section 2 provides a brief explanation of the data race detector tools. Section 3 introduces the benchmarks that we are using. The analysis and results discussion are in section 4. Section 5 provides an extension future work. The last section concludes the study and discussion.

2 DATA RACE DETECTOR TOOLS

Traditionally, most of the users and the developers are not aware of the race condition fault. As previously mentioned, there are five tools that we are going to test, study, and analyze their behavior and performance. These detector tools are RaceFuzzer, RacerAJ, Jchord, Race Condition Checker (RCC), and Java RaceFinder (JRF). Each detector tool has its own specific technique and checker in reporting the race condition. Few works have been conducted in the field of race condition in both side of static and dynamic detection. Nevertheless there is no work has been conduct in the field of comparing these data race detector tools. In this section, we aim to explore and provide initial study of five different data race detector tools that have been known in the industry. These tools are next discussed as follow:

2.1 RaceFuzzer

RaceFuzzer is an algorithm for determining real data races in concurrent programs that uses a randomized scheduler to produce real conditions of data races. In RaceFuzzer, an Enabled thread is a thread that is not waiting to get a lock held by another thread; an alive thread is a thread that

978-1-4673-5534-6/13/$31.00

has not yet terminated its execution [9]. CalFuzzer is the RaceFuzzer tool that we are using for the purposes of this study. Calfuzzer is framework of active testing for producing analysis of concurrent bugs such as data races as well as scheduler for active testing. The technique of calfuzzer for testing concurrent program is called active testing which has two phases; First phase uses off-the-shelf analysis tools to extract potential data races. Second phase uses the outcomes report of off-the-shelf to feed back the system for more accuracy in discovering data races [3].

2.2 RacerAJ

RacerAJ introduced an aspect-oriented approach to detect program events by evolving an extension to the oriented programming language called AspectJ. AspectJ allows the developers to easily predicate bug for concurrent program using pointcuts lock(), unlock() and maybeShared(). The first two allow us to acquisition and release of a lock using synchronized blocks and methods in java program and maybeShared() helps in gain runtime efficiency by monitoring the access of those fields among threads. The Racer algorithm that has been proposed is using the AspectJ language extension which is a variant of the Eraser algorithm for data race detection. The Racer algorithm is adoption of the Eraser algorithm in regard to the java memory model where the Racer detects data races in java program and the Eraser detect data race in C program [4].

2.3 JCHORD

JChord is a static and dynamic program analysis framework for Java. It has four key characteristics which are stand-alone program, extensible, compositional and efficient. The algorithm consists of four stages that involve memory access which are reachable-pairs computation, aliasing-pairs computation, escaping-pairs computation and unlocked-pairs computation [5].

2.4 Race Condition Checker (RCC)

RCC is a static analysis system for detecting race conditions in Java programs. The RCC analysis supports the lock-based synchronization discipline plus other indicators [6]. RCC detects data race condition early in the development cycle before producing the entire code of the program. It requires the developers to write some annotations that can be inserted as function of the lacking technique used by the program. RCC has number of features among them are: classes parameterized by lock, safety accessibility and escaping mechanism [7].

2.5 Java RaceFinder (JRF)

Java PathFinder, is ”a model checker for java byte code that simulates its own virtual machine with specified properties after it reads java class files” [10]. From the approaches of informal reasoning exercises by writing a concurrent program to the approaches of formal reasoning methods and tools such as model checkers, the sequential consistency is their start assumption. It is not a valid assumption for current concurrent program. The proposed tool called Java RaceFinder which is ”an extended tool of Java PathFinder that use the JPF listener interface to intercept memory model relevant operations such as locking, unlocking, start new thread, reading or writing a variable, etc.), and that can represent the heuristic function h”. Furthermore, JRF considers a synchronization approach to analyze the extensive set of the concurrent data structure [8].

3 BENCHMARKS

The benchmarks are the data tests that used in the data race detector tools in order to know the performance and the efficiency. In our benchmark, we start with around 20 lines of code and end up with around 700 lines of code. we chose our benchmark by looking through each detector tools' benchmarks and consider the benchmark that is not used in any other race detector tools as well as the benchmark that is used in all tools. Our consideration of choosing the benchmarks is to help us in predicting the performance of the race condition detector tools as well as to assist us in detecting and reporting different potential data race in each tool. Our technique on how to choose the right benchmarks is to go from the easier to the moderate benchmarks and from the small to somehow larger benchmarks. We have decided on choosing five different benchmarks to be used in the experiment. We categorize these benchmarks to three different categories: simple benchmarks, moderate benchmarks and hard benchmarks. These benchmarks can be summarized as follow:

1. Simple benchmarks with small line of code. In this category, we have three different benchmarks that can be summarized as follow:
   - Benchmark has one thread and does not have data race. This benchmark has no data race to be discover and the result must be obviously negative.
   - Benchmark has two threads and has one data race that is disable or got covered using synchronization. This benchmark has a real data race nevertheless it will not affect the program since we used synchronization. The result must be negative as well.
   - Benchmark has two threads and has two data races. One of the data race is disable or got covered using synchronization and the other one is real data race. Since there are two real data races but one of them will not affect the program, the result must be positive with one data race to be reported.
2. Moderate benchmarks with medium line of code. In this category, benchmark has ten threads and has two data races. This benchmark has real data races and the result must be positive with two data races to be reported.
3. Hard benchmarks with large line of code. In this category, benchmark has few threads and has nine data races. This benchmark has real nine data races that will affect the program and the result must be positive with nine data races to be reported.

4 ANALYSIS AND RESULTS

Analysing the race condition detector tools is not trivial since there is no standard measurements to be considered. However, there must be initiative questions to be asked in order to be able clearly to answer them. Among of the questions are; How powerful is the performance of each tool?, How much work need to be done in order to use each tool?, What type of race condition does each tool detect?, How many race condition each tool is detect?, Is there any suggestion that each tool does when it detects race condition? And how many line of code the benchmarks have? We have to pay attention to the benchmarks that we are testing. The lines of code of those benchmarks are the exact number of lines in the file to be tested and any other related files to be considered in the benchmarks directly. All other files, that are not related to the benchmarks and they are part of the program, are not considered to the benchmark such as thread classes, string classes, buffer classes and etc. Also, we have numbered all five benchmarks that we have as the following:

- benchmark has one thread and does not have data race. This benchmark is represented by B1.
- Benchmark has two threads and has one data race that is disable or got covered using synchronization. This benchmark is represented by B2.
- Benchmark has two threads and has two data races. This benchmark is represented by B3.
- Benchmark has ten threads and has two data races. This benchmark is represented by B4.
- Benchmark has few threads and has nine data races.This benchmark is represented by B5.

We have tested five different benchmarks on five different data race detector tools. Each one of the tools has been represented on the following tables and each table has a comparison of five different benchmarks versus one data race detector tool. We considered eight different criteria in our experiment. These criteria are:

- Compilation Time (CT) in second.
- Number of Line of Code (LOC).
- Number of Classes (NOC).
- Number of Reported Data Race (RDR).
- Numbe of Actual Data Race (ADR).
- Type of Data Race (TDR).
- Output Result (OR).
- Tool Usage (TU).

There are two types of data race which are:

- Not Applicable (NA).
- Not Classified (NC).

Table 1: RaceFuzzer is represented by comparing five benchmarks against eight criteria.

Criteria\benchmarks	B1	B2	B3	B4	B5
CT	~ 0.305 Sec	~ 0.306 Sec	~ 0.35 Sec	~ 0.4 Sec	~ 0.749
LOC	~ 21	~ 22	~ 25	~ 100	~ 700
NOC	1 Class	1 Class	1 Class	3 Classes	3 Classes
RDR	0	0	2	2	9
ADR	0	0	1	2	9
TDR	NA	NA	NC	NC	NC
OR	number of lines it detects and shared variables				
TU	Moderate				

Also, in the tool usage, there are three different types we faced when we used the tool for detecting data race. These three types are: Simple, Moderate and hard. We have tested the experiment on Mac BookPro Machine with 2.4GHz of CPU and 4GB of memory. The following tables are comparison of the five different benchmarks for each tool.
For the RaceFuzzer, we need to make some configurations to the test race condition in order to have the tool to work perfectly.

Table 2: RacerAJ is represented by comparing five benchmarks against eight criteria.

Criteria\benchmarks	B1	B2	B3	B4	B5
CT	~ 160 Sec	~ 210 Sec	~ 240 Sec	~ 110 Sec	FAILED
LOC	~ 21	~ 22	~ 25	~ 100	~ 700
NOC	1 Class	1 Class	1 Class	3 Classes	3 Classes
RDR	1	0	1	6	FAILED
ADR	0	0	1	2	9
TDR	NC	NA	NC	NC	NC
OR	number of lines it detects and shared variables				
TU	Moderate				

For the RaceAJ, there are two features provided with the tool. We only used the optimized feature because the other feature is not supported by MAC BookPro Machine. Technically, using the tool with the optimized feature takes some time to compile. Once it get optimized, running the race detector takes millisecond to analyse he program with finding results. However, we have to consider the overall running time with no exception. For benchmark B5, RaceAJ fails to run on this benchmark.
For the JChord, the output result is not easy to read. You can not read how many data races have been found. I have to count the number of data races from the output result by counting the number of lines of the occurrence data races that are provided by the tool. For benchmark B3, JChord is reported 3 data races occurrence where two of them are false reported data races and one real data race.
For the RCC, it is hard to use the tool because you are required to inject some statements that are defined by the tool in order to have the tool to work perfectly. Also, if you have implemented some classes to runnable type, then the tool

Table 3: JChord is represented by comparing five benchmarks against eight criteria.

Criteria\benchmarks	B1	B2	B3	B4	B5
CT	~ 110 Sec	~ 113 Sec	~ 102 Sec	~ 118 Sec	~ 100 Sec
LOC	~ 21	~ 22	~ 25	~ 100	~ 700
NOC	1 Class	1 Class	1 Class	3 Classes	3 Classes
RDR	3	2	3 or 1 without false	2	9
ADR	0	0	1	2	9
TDR	NC	NC	NC	NC	NC
OR	number of lines it detects + details for methods that report race + memory access				
TU	Moderate				

Table 4: RCC is represented by comparing five benchmarks against eight criteria.

Criteria\benchmarks	B1	B2	B3	B4	B5
CT	~ < 1 Sec	~ < 1 Sec	~ < 1 Sec	~ < 1 Sec	~ < 1 Sec
LOC	~ 21	~ 22	~ 25	~ 100	~ 700
NOC	1 Class	1 Class	1 Class	3 Classes	3 Classes
RDR	2 + errors	0 + errors	3	6 + errors	15
ADR	0	0	1	2	9
TDR	NC	NC	NC	NC	NC
OR	number of lines it detects				
TU	Hard				

will not work unless you remove them. For benchmarks B1, B2 and B3, RCC is reported some errors beside the data races which is odd since the files are free of error.

Table 5: JRF is represented by comparing five benchmarks against eight criteria.

Criteria\benchmarks	B1	B2	B3	B4	B5
CT	~ 0.001 sec	~ 0.001 sec	~ 0.001 sec	~ 0.005 sec	~ 0.002 sec
LOC	~ 21	~ 22	~ 25	~ 100	~ 700
NOC	1 Class	1 Class	1 Class	3 Classes	3 Classes
RDR	0	0	1	1 (unknown)	1 (unknown)
ADR	0	0	1	2	9
TDR	NA	NA	NC	NC	NC
OR	Number of lines it detects + suggested solution				
TU	Moderate				

For the JRF, it reports the first occurrence of data race and stops after that. it seems to me that JRF reports only the first one of the data race regardless of the other data races that have been known. For benchmarks B4 and B5, unknown means that we can not tell neither data race nor error.
Based on the previous tables, we can come to the conclusion that there are significant differences in the performance of detecting data races. Java RaceFinder (JRF) is the best performance and RacerAJ is the worst performance. The following graph represents the performance of the five tools in seconds:
Based on the previous tables, we can come to the conclusion that there are significant differences in the efficiency of detecting data races. RaceFuzzer is the best efficiency

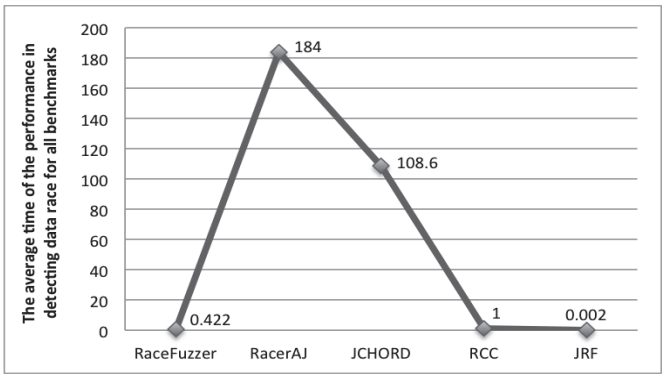

Figure 1: Shows the average time of the performance in detecting data race for all five benchmarks in seconds

and RCC is the worst efficiency in the case of ignoring the JRF. Figure 1 represents the efficiency of the five tools in term of the number of the data race occurrences compare to the actual data race occurrences:
Figure 2 shows that the actual data race occurrences are 0, 0, 1, 2 and 9 for benchmarks B1, B2, B3, B4 and B5 respectively. Each tool has reported different results which can be summarized as follow:

- RaceFuzzer has reported 0, 0, 2, 2 and 9 data races for benchmarks B1, B2, B3, B4 and B5 repsectively.
- RacerAJ has reported 1, 0, 1, 6 and 12 data races for benchmarks B1, B2, B3, B4 and B5 repsectively.
- JCHORD has reported 3, 2, 3, 2 and 9 data races for B1, B2, B3, B4 and B5 repsectively.
- RCC has reported 2, 0, 3, 6 and 15 data races for benchmarks B1, B2, B3, B4 and B5 repsectively.
- JRF reported 0, 0, 1, 1 and 1 data races for benchmarks B1, B2, B3, B4 and B5 repsectively.

based on the outcomes, we came to the conclusion that helps us on deciding which tool to use in order to test our

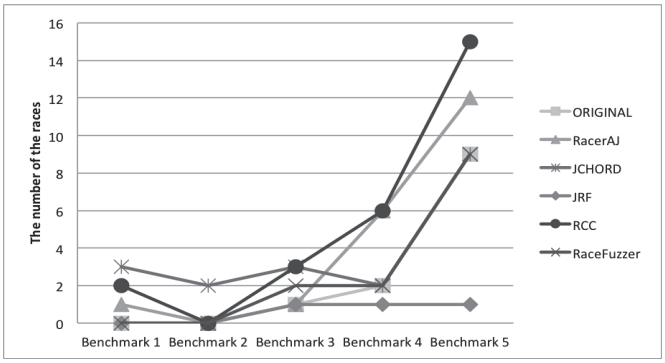

Figure 2: Shows the number of data races found by each tool for all the five benchmarks compare to the actual data races which are 0, 0, 1, 2 and 9.

program. We found that RaceFuzzer would be the best choice. Through the experiment, we notice that the five tools are not really quite easy to use since they are different in configuration, usage level and model. We can summarize the experiment outcomes experience as follow:

- RaceFuzzer is a moderated tool for data race detection. You need to work on your code as well as the tool configurations in order to perfectly detect data races.
- RacerAJ is also a moderated tool for data race detection. You need to work on your code as well as more on the tool configurations in order to perfectly detect data races.
- JCHORD is a moderated tool for data race detection as well as deadlock detection. You need to work on your code as well as more on the tool configurations in order to perfectly detect data races.
- Race Condition Checker (RCC) is a hard tool for data race detection. You need to inject some statements in the code in order to have the tool to work perfectly where in my opinion is like finding the data race manually.
- Java RaceFinder JRF is a moderated tool for data race detection. You need to work on your code as well as the tool configurations in order to perfectly detect data races. Also, it reported the first occurrence of data race and ignore the rest if any.

All in all, we recommend the RaceFuzzer data race detector tool to be used in order to free your code from data race.

5 FUTURE WORKS

We plan to extend our work to go deeper in the race condition by providing more benchmarks to be tested. Also we plan to extend our work to include the deadlock detector tools since it is mostly ignored.

6 CONCLUSION

As for the author's knowledge, there is no work done neither on an empirical study of data race detector tools nor deadlock detector tools. In this study we explore five different tools that detect the data race which are RaceFuzzer [3], RacerAJ [4], Jchord [5], Race condition Checker RCC [6], and Java RaceFinder JRF [8]. The study have been shown that each one of them has significant feature that can be uniquely identified its possibility of detecting race condition. A good data race detector tool leads to successfully clean the software system from race condition faulty. Indeed, the results shown that each tool has its own specific way in detecting the race condition and each tool reported differently from the others which leads to have comparative tools. Based on the experiment we conduct, we showed that the most important things to be considered are what is the average time to detect the race condition? And how many race condition the tool finds compared to the actual race in the code that we test? The results showed that the best time is registered for the Java RaceFinder (JRF) race condition detector tool that is not accurately detecting the race condition when it comes to have many races which we can't depend on. Also, the worst time is registered to RacerAJ race condition detector tool that is also not totally dependable since it comes second best tool in detecting the race condition. Furthermore, when it comes to the number of the races that have been reported, the best result is registered to the RaceFuzzer race condition detector tool that leads to almost most dependable tool to be considered when you plan to free you program from the race condition. Also, the worst result is registered to the Race Condition Checker (RCC) race condition detector tool that never detects the exact amount of race compare to the actual races.

REFERENCES

[1] Netzer, R. H. and Miller, B. P. 1992. What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst. 1, 1 (Mar. 1992), 74-88.

[2] Yu, Y., Rodeheffer, T., and Chen, W. 2005. RaceTrack: efficient detection of data race conditions via adaptive tracking. In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (Brighton, United Kingdom, October 23 - 26, 2005). SOSP '05. ACM, New York, NY, 221-234.

[3] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "An Extensible Active Testing Framework for Concurrent Programs," in Proc. 21st International Conference on Computer Aided Verification (CAV'09), 2009

[4] Bodden, E. and Havelund, K. 2008. Racer: effective race detection using aspectj. In Proceedings of the 2008 international Symposium on Software Testing and Analysis (Seattle, WA, USA, July 20 - 24, 2008). ISSTA '08. ACM, New York, NY, 155-166.

[5] Mayur Naik, Alex Aiken, and John Whaley. E?ective static race detection for java. In Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI), pages 308?319. ACM Press, 2006.

[6] Cormac Flanagan and Stephen N. Freund. 2001. Detecting race conditions in large programs. In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01). ACM, New York, NY, USA, 90-96.

[7] Cormac Flanagan and Stephen N. Freund. 2000. Type-based race detection for Java. SIGPLAN Not. 35, 5 (May 2000), 219-232.

[8] Kim, K., Yavuz-Kahveci, T., and Sanders, B. A. 2009. Precise Data Race Detection in a Relaxed Memory Model Using Heuristic-Based Model Checking. In Proceedings of the 2009 IEEE/ACM international Conference on Automated Software Engineering (November 16 - 20, 2009). Automated Software Engineering. IEEE Computer Society, Washington, DC, 495-499.

[9] K. Sen, "Race Directed Random Testing of Concurrent Programs," in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08), 2008, pp. 11-21.

[10] G. Lindstrom, P. Mehlitz and W. Visser, "Model Checking Real Time Java Using JavaPathfinder", In Proceedings of the Third International Symposium on Automated Technology for the Verification and Analysis (ATVA), October 2005.