An Evaluation of Data Race Detectors Using Bug Repositories

Jochen Schimmel, Korbinian Molitorisz, Walter F. Tichy
Karlsruhe Institute of Technology (KIT), Germany
{schimmel, molitorisz, tichy}@kit.edu

***Abstract*—Multithreaded software is subject to data races. A large number of data race detectors exists, but they are mainly evaluated in academic examples. In this paper we present a study in which we applied data race detectors to real applications. In particular, we want to show, if these tools can be used to locate data races effectively at an early stage in software development.**

We therefore tracked 25 data races in bug repositories back to their roots, created parallel unit tests and executed 4 different data race detectors on these tests. We show, that with a combination of all detectors 92% of the contained data races can be found, whereas the best data race detector only finds about 50%.

***Index Terms*—Data Races, Unit Testing, Multicore Software Engineering, Empirical Study**

I. Introduction

ALMOST all race detection approaches are evaluated in source code, that is freely available or in productive use. But can these detectors also be effectively used during software development before delivery and prevent shipping errors? We conducted an empirical study to answer this question. For our experimental setup we browsed bug repositories of open source applications for reports of data races. For each report we tracked back the revision history to the point, at which the defect has been checked into the code repository for the first time. We used this revision to evaluate which of the data race detectors would have been able to find the race at the distinct moment where it was unintentionally inserted.

A first key finding was, that it was almost impossible to simply apply a race detector on our evaluation programs: The application of most race detectors was impractical and the true data races were outbalanced by the huge number of false positives. Furthermore, most data race detectors available consume too much memory and computation time. Results tend to be impressive when applied to small programs, but with increasing sizes of real world applications, race detection approaches become increasingly impractical. Our solution to this problem was to create parallel unit tests[1] for all programs: A parallel unit test calls two (or more) methods under test in parallel within separate threads. In contrast to regular unit tests, they do not contain assertions. A data race detector decides on the test result. It executes parallel test cases rather than the program itself. By writing such parallel unit tests, we divided the programs into smaller fractions and focused error detection on the relevant portions, that were small enough to be handled by the race detectors. As a consequence, we could successfully apply all four race detectors to the bug repositories and locate 92% of the data races.

This paper is structured as follows: In section II, we introduce the sample applications we used as benchmark for the data race detectors and present the bugs we found in the respective repositories. Section III presents the four data race detectors we evaluate. We discuss the results of our study in section IV and detail on parallel unit tests in section V. The paper concludes with related studies in section VI and a conclusion of the key findings.

II. Sample Applications and Bug Repositories

Apache Tomcat is a web server written in Java and uses Bugzilla as bug tracking system. In Tomcat, each web request is handled by a separate thread and all of them access common data. No or incorrect synchronization of the common data leads to program stalls or incorrect data. In Bugzilla we tracked down 23 reports of data races in Tomcat due to synchronization errors.

Spring is an application development framework library for Java and uses Jira to track bugs. Spring contains framework classes, that can be executed both sequentially or in parallel. We tracked 24 data race reports in the bug database caused by altered program semantics when executing the parallel versions of the framework classes.

Eclipse is an integrated development environment for Java and other programming languages and tracks bugs using Bugzilla. Eclipse executes long-running computations in background threads to keep the user interface responsive. We tracked 18 synchronisation errors in Bugzilla concering long-running background threads.

Defect Classification: We categorize the defects according to their root into four different error patterns: (1) Atomicity violation, (2) wrong usage of Java library, (3) if-race and (4) bad optimization. A data race may account to more than one of the four error patterns. However, some of the defects we found are specific and do not apply to any of these categories.

Atomicity violations are data races caused by incorrect granularity of synchonization. Here, different memory location have data- or control flow dependencies. They form a logical unit and may only be changed atomically or in a transaction. Even if each location might be synchonized separately, the access to the whole unit is not.

The Java library contains thread-safe classes, i.e. they can be used in multithreaded applications without additional locks.

TABLE I
DATA RACES AND DETECTION RESULTS

Program Details			Bug Details					MTRAT	ConTest	Jinx	Jchord		Enriched PUT
Bug-ID	Program	Σ Methods	Impact	Atom.	Lib	if-Race	Opt.	Det.	Det. (%)	Det. (%)	Det.	FPs	Det. (%)
4418	Tomcat	1	Crash			x	x	x	70	60	x	1	90
31018	Tomcat	1	Crash		x	x			60	0			0
48790	Tomcat	1	Wrong Results			x		x	0	0	x		0
728	Tomcat	2	Crash				x	x	80	0	x	1	0
48177	Tomcat	1	Crash			x		x	100	90	x		100
46085	Tomcat	2	Deadlock					x	0	0	x	10	0
48172	Tomcat	2	Wrong Results					x	0	0	x	1	0
36173	Tomcat	2	Crash		x		x	x	0	0		2	0
SPR-5658	Spring	1	Crash			x			0	0			0
SPR-4932	Spring	1	Crash			x		x	20	20			10
SPR-4938	Spring	1	Deadlock	x	x	x	x		70	10		10+	90
INT-748	Spring	2	Crash		x				30	0		10+	30
SPR-3228	Spring	2	Crash		x				70	100			80
SPR-4672	Spring	1	Crash	x					0	0			0
SPR-2000	Spring	2	Crash					x	100	0			100
SPR-3432	Spring	1	Crash						0	0	x		0
INT-1072	Spring	2	Crash		x				80	90		10+	0
44809	Eclipse	2	Crash					x	20	0	x	10+	0
104294	Eclipse	1	Crash		x	x			100	100	x	10+	100
163685	Eclipse	2	Crash					x	0	0			0
296822	Eclipse	2	Wrong Results					x	0	0	x	1	0
31159	Eclipse	1	Wrong Results		x	x			0	50			100
272742	Eclipse	1	Crash					x	30	0	x		0
298648	Eclipse	1	Wrong Results		x	x			90	0	x	10+	0
36659	Eclipse	1	Crash			x			60	100	x		0
Σ (25 total)		1: 14 / 2: 11		2	9	11	4	13	15	9	13	12	9

The second error-pattern we define classifies code locations as defect in which classes are treated as thread-safe but that are not designed to be used like this.

Almost half of all errors we found are if-races: An if-race occurs, when a variable is checked for a certain value inside a conditional expression leading to a branch. In there, the variable is updated to a new value. For correct program semantics, both statements must be within the same lock because the thread could otherwise be interrupted in between and the variable is changed by another parallel operation.

The bad optimization pattern does not reflect a certain code design pattern, but rather describes different kinds of errors which come from the intention to improve program runtime but not its code behaviour. In fact, these code changes have unintended side effects. We could only identify these errors because of comments we found in the bug repositories.

We summarize our results in table I. One central finding is, that practically all data races can already be reproduced by a twofold parallel execution, although the programs may execute the parallel regions at a higher degree of parallelism. If the parallel regions execute in the wrong order at runtime, the defect manifests. 76% of all defects belong to this category. 20% require three or four operations, while only 4% require at least five operations in an unexpected order. With just one exception, we could successfully reproduce the defects using two threads. The column *Method sum* under *Program Details* shows, that roughly half of the data races are caused by a parallel execution of two different methods, whereas the other half is caused by parallel execution of the same method. This relates to the key finding by Shan Lu et al. [2], according to which most data races can be reproduced with only two methods. According to our finding, we can generalize the term of *two methods* to either depict the parallel execution of two separate methods as in task parallelism or to depict the parallel execution of the same method as in data parallelism.

III. DATA RACE DETECTORS

We present the four data race detectors used in our study. The selected detectors had to be freely available and were required to support Java code natively for comparability reasons. We included three dynamic and one static tool. Further studies should also include other prominent tools available such as Helgrind+ [3] or Racer-X [4].

MTRAT: Multi-Thread Run-time Analysis Tool for Java (MTRAT) is a dynamic detection tool for data races and deadlocks developed by IBM [5]. MTRAT uses a combination of the happens-before and lockset race detection algorithms. For this work, we used its Eclipse plugin for Windows in order to define which classes of an application to instrument. MTRAT can also instrument libraries at bytecode level, except for Java core libraries. For self-written code, MTRAT returns the line numbers where the error occured; for errors in libraries, only the class name is returned. MTRAT captures the execution path and makes the data race reproducable. Unlike other race detectors, it is unable to identify alternative control flows. For us to work with MTRAT, we manually wrote test cases, that induced the problematic control flow. Thus, the investigation of complex programs such as Eclipse is not feasable due to slowdown limitations.

ConTest is another dynamic tool developed at IBM alphaworks [5]. ConTest inserts sleep and yield instructions heuristically into Java bytecode to create different thread interleavings. When re-executing an application, ConTest varies the thread scheduling to provoke data races and deadlocks. ConTest is available as an Eclipse plugin. It offers numerous options to adjust the interleaving heuristics. In contrast to other race detection tools, ConTest does not identify data races, it only provokes different interleavings, so ConTest raises the chance for a data race to occur. The developer finally has to identify the race by invalid program behaviour using assertions.

Jinx is a commercial tool to find errors in multithreaded applications [6]. It supports programs in Java, C/C++ and .NET-languages. Jinx is a dynamic race detector and executes a program several times, altering thread schedules. When a race is detected, Jinx can replay the problematic schedule. Like ConTest, Jinx relies on programs throwing exceptions as soon as a data race occurs.

Jchord is an open source static data race detector [7]. Jchord is a command line tool which expects the Java classes under test as input; it will also detect errors within compiled Java libraries, such as the Java core libraries.

IV. Results

Table I summarizes all 25 bugs and the data race detection results of all 4 evaluated tools. The column *Bug-ID* references the respective bug tracking system. We evaluated each of the race detectors with the same unit tests as program input. For ConTest and Jinx we had to extend the parallel test cases to include exceptions and assertions. We found, that by executing 9 defects could even be found without any data race detector. We call this extension *enriched parallel unit tests*. The results are shown in column *enriched PUT*.

MTRAT is unaware of atomicity violations and cannot find defects due to wrong library usage, as it does not check the Java core libraries. To verify the library limitation, we used an open source implementation of the Java library. With this change MTRAT would have found all 9 library defects. MTRAT exhibited false positives on one occasion only. According to its heuristics, MTRAT could have found 15 data races, 13 were in fact found.

ConTest alternates thread interleavings, so its results are not reliable. ConTest can only find data races, when they actually occur. We therefore executed ConTest 10 times and measured if the race was reported at least once. Another weakness is, that ConTest reports races on the basis of hand-written exceptions or assertions that fail, so we extended our benchmark to use enriched PUTs. With this, ConTest could identify 9 additional defects. A third drawback of ConTest is the lack of information to resolve the defect: It only executes the test case. It does not provide any information about what caused the data race or at which code line.

As Jinx is also unreliable we used the same reproduction logic and used our enriched PUTs as input. Jinx offers several *intensity levels* to improve defect detection, that showed no noticeable difference in our experiments. Jinx is able to detect errors due to wrong library usage, atomicity violations and did not produce false positives, but found only 36% of the errors. With 9 defects, the enriched PUTs found as many errors as Jinx, but 2 of them were only found by Jinx. The slowdown of Jinx is within a few seconds, so it may be used as a supportive tool.

Jchord is a static race detector. It can be applied to the regular evaluation applications, but we used it on our test cases for comparability reasons. Also, this extension reduced the execution time drastically: With the regular test cases Jchord required 4 minutes on average and in 4 cases it crashed with out of memory exceptions. With enriched PUTs each test case executes within a few seconds; From the remaining 21 bugs, 13 were found. Jchord is unable to find atomicity violations and was the only tool to produce a significant number of false positives. For 6 test cases it reported more than 10 false positives. This severly lowers its benefit for real-life scenarios.

V. Parallel Unit Tests

Our evaluation shows the efficiency of data race detection supported by parallel unit tests. If parallel unit tests are available, they can be used as input for different race detectors. Combining all 4 detectors, we could identify 92% of the bugs. A combination of the two best race detectors MTRAT and ConTest still found 84%. This shows that parallel unit tests may be a veritable approach to ease data race detection. Some race detectors like CHESS [8] are specificially designed for parallel unit tests. However, writing sound parallel unit tests is hard. Therefore, the exploration of automatic generation of parallel unit tests is an active research topic [9, 10]. The parallel test cases we wrote for this study conform to this research: A parallel test case is a test method calling at least two program methods in separate threads; the test method exits as soon as the threads have returned. A parallel test method does not alter the thread schedule or influence the program execution in any way - this is left to the data race detector that executes the parallel test method. Parallel unit tests do not contain assertions or throw exceptions deliberately, the decision whether a race is found or not is completely left to the detector. As we showed, some race detectors break with this definition of a parallel unit test, as they require assertions in the test case. If a parallel test case contains assertions to detect the presence or negative effects of data races, we call it enriched.

Figure 1 shows a sample with a parallel test case and an enriched version. The test case executes *inc()* concurrently in two threads. After they return, the test exits. The results and side effects of the test are not evaluated, this is left to the execution environment, i.e. the race detector. In the second case, the enriched test case waits for both methods and will report an error if the value of *val* is not 2. This test is able to detect malicious race behaviour, but it depends on the concrete thread schedule and the race detector influences the probability to provoke unintended behaviour. Using enriched PUTs, complexity is transfered from detector design to test development; this may be a good approach for bugs that are hard to detect, like atomicity violations. Here, semantic information on the programmer's intention is required to identify an error. Even in our small sample, we show that MTRAT cannot find them, whereas detectors using enriched tests such

```
class CInc {

private int val;

public void inc() {
  val++;
}

public int getVal() {
  return val;
}

}
```

(a) The sample class CInc.

```
class IncrementTest  {
static CInc inc = new CInc();

public static void Main() {
  Thread t1 = new Thread(
    new Runnable() {
    public void run() {inc.inc();}
  });
  Thread t2 = new Thread(
    new Runnable() {
    public void run() {inc.inc();}
  });
  t1.start(); t2.start();
}
```

(b) A parallel test case for CInc.

```
class IncrementTestX {
static CInc inc = new CInc();

public static void Main() {
  Thread t1 = new Thread(
    new Runnable() {
    public void run() {inc.inc();}});
  Thread t2 = new Thread(
    new Runnable() {
    public void run() {inc.inc();}});
  t1.start(); t2.start();

  try { t1.join(); t2.join(); }
  catch (InterruptedException e) { }
  if (inc.getVal() != 2)
    System.err.println("Error!");
}
```

(c) Enriched parallel test case for CInc.

Fig. 1. Code excerpt of the Bank Account Sample with its instrumented versions.

as Jinx can. Nevertheless, developing enriched, sound parallel test cases is harder than usual parallel test cases and to our current knowledge, no automatic generation approaches exist.

VI. Related Work

Bug evaluation has been performed before: Shan Lu et al. [2] evalute 105 synchronisation bugs from large applications for bug patterns. In contrast to our work, no data race detectors have been evaluated. In [11] and [12], different Java race detectors are evaluated. However, the used defects are from artificial sample applications, not from real bug repositories. They indicate that static race detectors produce too much false positives and are hard to use. In [13], programs written in C/C++ are evaluated.

VII. Conclusion

In this work, we searched bug repositories of four large Java applications for historic data races reported by users. We then tested 4 well-known data race detectors with the program revisions which contained the bugs for the first time. Seen individually, each of the four data race detectors found about 50% of the bugs. Together, the detectors found 92% of these bugs. In order to efficiently use the detectors, it is necessary to write specific test cases for data race detection. Our results indicate that a good, test based detection infrastructure combining different race detection approaches may help to find most data races early. However, writing good parallel test cases is hard and time-consuming. We therefore see our results as a motivation to automatically generate parallel unit tests. Different works heading in this direction have been mentioned. For future work, we plan to extend this study to more evaluation programs and other race detectors. As a combination of different tools seems promising, it would be interesting to know if a certain combination of detection strategies leads to optimal results. Furthermore, we want to search for data races using generated test cases from the works presented above.

Acknowledgment

The authors would like to thank Yana Stoeva for her support with this project.

References

[1] G. Szeder, "Unit testing for multi-threaded java programs," in *Proceedings of the 7th Workshop on Parallel and Distributed Systems*, 2009.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," ser. ASPLOS XIII, 2008.

[3] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," ser. IPDPS '09, 2009.

[4] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 237–252, Oct. 2003.

[5] *alphaWorks: Advanced Testing for Multi-Threaded Applications*, September 2010. [Online]. Available: http://www.alphaworks.ibm.com/tech

[6] *Corensic: Jinx*, November 2012. [Online]. Available: http://wiki.corensic.com/wiki

[7] *Jchord*, September 2010. [Online]. Available: http://code.google.com/p/jchord

[8] S. Q. Madanlal Musuvathi and T. Ball, "Chess: A systematic testing tool for concurrent software," Microsoft Research, Tech. Rep., Nov 2007.

[9] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code," in *Proceedings of the 2012 International Conference on Software Engineering*, 2012.

[10] J. Schimmel, K. Molitorisz, A. Jannesari, and W. F. Tichy, "Automatic generation of parallel unit tests," in *ACM AST '13*, 2013.

[11] C. Artho, "Finding faults in multi-threaded programs," Masters thesis, Tech. Rep., 2001.

[12] A. K. Md Abdullah, Al Mamun, "Concurrent software testing: A systematic review and an evaluation of static analysis tools," 2009.

[13] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.