DYNAMIC DETECTION AND HEALING OF LOW LEVEL DATA RACES

Zdeněk Letko
Master Degree Programme (2), FIT BUT
E-mail: xletko00@stud.fit.vutbr.cz

Supervised by: Tomáš Vojnar
E-mail: vojnar@fit.vutbr.cz

ABSTRACT

Data races are a common problem in concurrent programming. This article describes a tool which is able to detect low level data races in Java programs and heal them – all at run-time. This tool is build on top of IBM ConTest, a concurrency testing software. The tool uses a modification of the Eraser algorithm to detect data races and implements two techniques of data race healing.

1 INTRODUCTION

Data races are difficult to find using traditional testing approaches because a multi-threaded program may execute differently from one run to another. This article considers only low level data races. The traditional definition of a low level data race [3] is as follows: A data race occurs when two concurrent threads access a shared variable and when at least one of the accesses is a write and the threads use no explicit mechanism to prevent the accesses from being simultaneous. The Eraser algorithm [3] has been proposed to find races in C programs. Our race detection and healing tool is designed for Java and is implemented on top of IBM ConTest [1]. Eraser algorithm was enriched with Java join synchronization [2] and two healing approaches were implemented.

2 DYNAMIC LOCALIZATION OF DATA RACES

Dynamic error detection technologies are applied at the run-time of the program. Most of them make use of instrumentation. An instrumentor is a tool that receives as its input the original program and instruments it, at different locations, with additional statements. During the execution of the program, the instructions embedded by the instrumentor are executed.

2.1 DYNAMIC RACE DETECTION ON TOP OF IBM CONTEST

Our race detector is built on top of IBM ConTest [1]. ConTest is a concurrency testing tool for Java applications. It provides us with an instrumentator, a noise provider and listeners architecture. The noise producer injects sleep and yield calls into specific places of the tested

code. Noise injection is a technique that forces different legal interleaving for each execution of the test application. In a sense, it simulates the behaviour of various possible schedulers.
The listeners architecture allows the race detector to connect to ConTest and execute its code at selected places of the instrumented application during the run. This way, the tool is able to gather useful information and influence the execution. The combination of noise injection and dynamic data race techniques rapidly increases data race detection efficiency.

2.2 ERASER AND ITS PROPOSED MODIFICATION

The Eraser algorithm [3] is based on the LockSet algorithm proposed to find low level data races in lock-based multithreaded programs. These algorithms are based on the idea that every shared variable must be protected with a lock. Our race detector modifies the Eraser algorithm to work with Java implicit locks [2] and enriches it with threads join synchronization [2]. This modification helps to significantly decrease the number of false alarms in a common environment.
The tool is also able to suggest which lock should be used by the thread causing a race because if some threads use lock with some shared variable, then the same lock should be used by all threads. This information is provided as a suggestion for the programmer to correct the code and can be also used as an input for an automatic healing process.
Proposed algorithm works with states maintained for each shared variable described in the Figure 1.

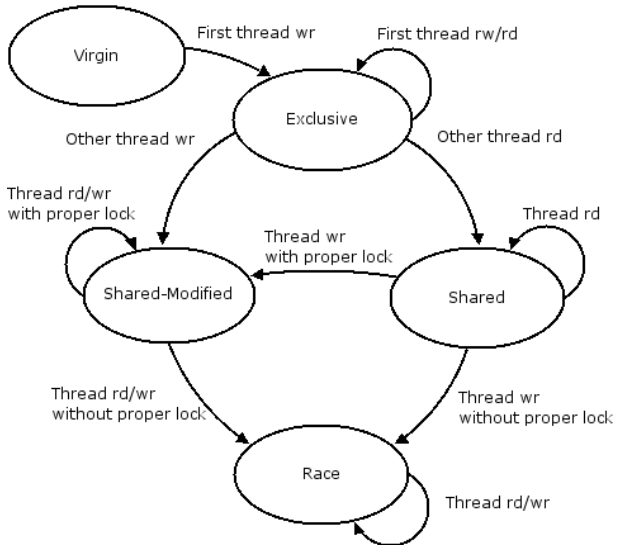

Figure 1: State diagram of the proposed race detector.

The variable is in the **Virgin** state till the end of its initialization. It is then in the **Exclusive** state as long as only one thread is accessing the variable. It goes to the **Shared** state when other threads start to read the variable or to the **Shared-Modified** state if at least one thread writes the variable.
Write changes the value of the variable and therefore introduce non-determinism of the variable's value at the time. This non-determinism can be avoided by using a lock. The algorithm checks if a proper lock is used by all threads accessing the variable. If an improper or none lock is used, race conditions are fulfilled and a race is possible. Whenever the **Race** state is encountered, healing process is applied.

3 DATA RACE HEALING

When a race is detected, healing process can be performed. The key issue in low level data race healing is forcing threads not to concurrently access a shared variable. For example, simple incrementation `x++;` must be performed atomically. This is problem because in Java bytecode this simple operation is represented by three instructions – `load x`, `increment x`, and `store x`. If another thread changes the value of `x` in between these instructions, a data race occurs and the variable might get to contain a wrong value. Two healing techniques has been implemented.

The first healing technique indirectly affects scheduler. Before a possible atomicity violation the active thread calls `yield()` which causes a thread switch. Next time the thread is scheduled, it gets whole time window from the scheduler and can with higher probability do the operation without an interruption. This can also be combined with setting of priorities of threads. This approach can only lower the probability of the race to occur.

Additional external lock is the second technique which prove that healing will be successful. Every time the variable is accessed, the threads must lock this lock. The nonatomic sections are completely covered by this lock so lock is not released within the problematic part that should be atomic. Problem of this technique is danger of deadlock.

4 CONCLUSIONS

This paper presents an integration of the Eraser algorithm with the IBM ConTest environment producing a new tool for race detection in Java programs. The use of ConTest improves chances of Eraser detecting a potential problem. The Eraser algorithm was optimized for use with Java and ConTest reducing the number of false alarms.

ACKNOWLEDGEMENTS

I thank to other members of SHADOWS project team at FIT BUT and also to Rachel Tzoref and Yarden Nir from IBM Research Center Israel. This work is partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD – project SHADOWS contract IST-035157. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein. This work is partially supported by the Czech Ministry of Education, Youth, and Sport under the project *Security-Oriented Research in Information Technology*, contract CEZ MSM 0021630528.

REFERENCES

[1] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *j-CCPE*, 15(3–5):485–499, 2003.

[2] Brian Goetz and Tim Peierls. *Java concurrency in practice*. Addison-Wesley, 2006.

[3] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.