Experience with a Concurrency Bugs Benchmark

Yaniv Eytani, Rachel Tzoref, and Shmuel Ur
University of Illinois at Urbana-Champaign, USA — *IBM Haifa Research Lab, Haifa, Israel*
yeytani2@cs.uiuc.edu — rachelt@il.ibm.com, ur@il.ibm.com

Abstract

We describe a benchmark of publicly-available multi-threaded programs with documented bugs in them. This project was initiated a few years ago with the goal of helping research groups in the fields of concurrent testing and debugging to develop tools and algorithms that improve the quality of concurrent programs. We present a survey of usage of the benchmark, concluding that the benchmark had an impact on the research in the field of testing and debugging of concurrent programs. We also present new possible directions to foster a discussion about new goals to be set for this initiative.

1. Introduction

The increasing popularity of concurrent programming – for the Internet as well as on the server side – has brought the issue of concurrent defect analysis to the forefront. Concurrent defects such as unintentional race conditions or deadlocks are difficult and expensive to uncover and analyze, and such faults often escape to the field. Now that all processors being made are multi-core, the testing of multi-threaded programs becomes even more important.

There is a large body of research that seeks to improve the quality of multi-threaded programs, both in academic circles and in industry. We believe that greater impact, and better tools, could result if use was made of the variety of relevant technologies [16]. Toward this end, a few years ago we began an initiative to help researchers create useful technologies, evaluate them, and share knowledge in the realm of concurrent testing and debugging. For this purpose, we previously created a benchmark containing multi-threaded programs with documented bugs in them. When combined with other artifacts, such as publicly available instrumentation engines [9], this type of a framework enables technology developers to concentrate on their components and use other ready-made components to create a testing solution.

We view this as a project to be shared by the entire concurrent testing and debugging community. We have conducted discussions about this project at the PADTAD workshop series [25], starting in 2003 [15], and with additional groups, such as the AspectJ developers. Quite a few groups and researchers have expressed interest in participating in this project. We published initial results for developing a benchmark that formally assesses the quality of different tools and technologies and compares them at PADTAD 2004 [17], followed by a dedicated special issue of "Concurrency and Computation: Practice & Experience" for parallel testing and debugging [24] [16]. Initially, we had created and documented about forty annotated programs, most of which were small examples annotated by undergrad students. Since then, we have had additional contact with other researchers who agreed to share their own repositories used mostly for static [18][19] and dynamic [1] [14] [20] [3] atomicity and race detection tools. We have incorporated these into the publicly available repository.

This paper provides the following contributions:

- Presenting our efforts in creating the benchmark so far. We believe it is important to publicize the benchmark content to researchers who may want to use it and participate in it. Furthermore, sharing our experience during the last four years with others may promote additional such efforts.
- Showing how this type of benchmark has helped to facilitate research with different academic groups. We survey works that used programs from the benchmark to demonstrate that it had an impact for the concurrent testing and debugging community.

- Promoting an open discussion in the testing community on how to further extend the benchmark.

2. Motivation and Problem Description

There are a number of distinguishing features between concurrent defect analysis and sequential testing. These differences make it especially challenging if the set of possible interleavings is huge and it is not practical to try all of them. First, only a few of the interleavings actually produce concurrent faults; thus, the probability of producing a concurrency fault can be very low. Second, under the simple conditions of unit testing, the scheduler is deterministic; therefore, executing the same tests repeatedly does not help. As a result, concurrent bugs are often not found early in the process, but in stress tests or by the customer. The problem of testing multi-threaded programs is even more costly because tests that reveal a concurrent fault in the field or in a stress test are usually long and run under different environmental conditions. As a result, such tests are not necessarily repeatable, and when a fault is detected, much effort must be invested to recreate the conditions under which it occurred. When the conditions of the bug are finally recreated, the debugging itself may mask the bug (the observer effect).

One might ask why creating a benchmark is a useful effort at all. The benchmark we are describing here is different than many others available, for various domains, in that it not only contains programs against which the tools are evaluated, but also a number of additional artifacts that are useful for developing the testing tools. For example, the bugs are annotated, so that if a race detection tool suspects a variable, assessment can be made to determine whether it is a false negative or a real result. Thus, the benchmark provides an infrastructure and a standardized way to compare different solutions. The benchmark contains a large number of publicly available small and well-understood programs that help develop and debug new testing tools, as well as several publicly available large programs with documented bugs to allow evaluation of scalability of tools. The large repository of programs can also help educate about concurrent bug patterns.

Creating a benchmark for run-time testing tools is a more challenging task than it might initially seem, for several reasons. First, it requires a good understanding of the programs under test. Augmenting the benchmark with programs requires either the manual work of writing a small program exhibiting a known bug pattern or adapting publicly available programs to the benchmark. In the latter case, such programs containing concurrent bugs must first be located. It is usually difficult to take such an arbitrary program, and then configure and run it without previous familiarity with the functionality of the program. In addition, it may be even harder to write a specific test driver that exercises code containing the bug. Specifications for these programs may not be available in many cases. Hence, manual work must be invested to reason about the program behavior and write oracles that detect at run time whether an error had occurred. Unless the error is an uncaught exception or a deadlock, knowledge of the excepted results or exact calculation done is needed. A related problem is that a violation of general criteria, such as a race condition or an atomicity violation, may not directly translate to a visible bug. For example, for an exception that leads to an actual crash to occur in the Jigsaw web server, many races ([20]) must occur during the program execution. Thus, an exact description of the bug's scenario can be especially important for writing a test oracle and for evaluating tools debugging multi-threaded code.

3. Current State, Dissemination and Feedback

Currently the benchmark contains about 60 programs written in Java. As mentioned earlier, most of the programs in the benchmark are small programs that exhibit known bug patterns [26]. Other small and medium examples are either taken from programs used for testing by NASA, open source, Java standard library components (Collection, stringbuffer, apache tomcat logger, and apache commons pool collections and libraries [3]) and other larger programs, adapted from open source resources. Though the benchmark is publically available, it requires an initial registration process. This enables some tracking of the benchmark usage. Registered users include people from academia as well as the industry. Even though the benchmark became available to researchers less than four year ago, it has already been used by many groups and in a number of published academic papers.

Below is a partial list of universities and industry research centers that use the benchmark:

- Cornell University, Ithaca, NY
- Purdue University, West Lafayette, India
- University of Texas at Arlington

- University of Illinois at Urbana-Champaign
- NEC Laboratories America
- Brigham Young University, Provo, Utah
- Technical University of Valencia, Spain
- Queen's University, Kingston, Canada
- University of Sao Paulo, Brazil
- IBM TJ Watson Research Center
- Hong Kong University of Science and Technology
- UniTESK team, ISP RAS
- Haifa University, Israel

Below is a list of published work that used the benchmark artifacts for development and evaluation, segmented to different research sub-domains:

- Testing concurrent components by combining code inspection and static analysis [2] and static [5] and dynamic [3][4],[28] tools for race detection and finding atomicity violations. Random test generation [13].
- Debugging of concurrent code and bug explanation [23][12].
- Defining new program mutation operators for testing concurrent code [10][11].
- Evaluating different path controlling factors [6], selection heuristics [7][8] and new testing methods [21] for model checking multi-threaded Java code.

By examining the different publications and other publications [27] [18], we can draw some observations about the usefulness of the benchmark programs. We notice that researchers from the model checking field were interested in relatively small to medium programs that can be easily understood and run in a model checker. Such programs make it easy to write violation assertions and can feasibly be model-checked. Hence, model checking researchers found the student programs valuable. Other researchers building static and runtime tools that scale better prefer to try larger open source programs and components. Smaller programs are still very useful at the development stage for testing the tool implementation, as carried out by Chen, Serbanuta, and Rosu [3]. Most of the research work is focused around detecting races and atomicity violation as opposed to deadlock and missing signals (e.g., wait and notify).

4. Conclusions and Future Direction

A wide range of technologies has already been developed to tackle the problem of testing multi-thread code. However, no silver bullet solution exists and current research focuses on a variety of partial solutions. One of the original goals for creating our concurrent benchmark was to allow rapid prototyping and development of technologies for concurrent testing and debugging while reducing the validation overheard. Considering the wide range of work that has taken place using the benchmark since its establishment, we believe that this goal has been achieved.

Additional infrastructure can help further reducing the development effort of tools for concurrent testing and debugging. Examples of such infrastructures are a generic infrastructure for instrumentation (such as [9]), a standard for automatic test drivers to allow easy setup (a possible solution might be similar to [22]), and a generic open API that allows different components to exchange knowledge about concurrent programs and executions without having a-priori knowledge about each other. We are currently considering which infrastructures to include in the extension of our benchmark.

Another direction we believe could be improved is the use of experience reports, based on experience with the programs and different tools using the concurrency benchmarks. There are a limited number of taxonomies available for concurrent bugs that mostly predate the benchmark creation [26]. Hence a revisited taxonomy should be devised taking into account programs in the benchmark that contain real concurrent bugs found in open source projects and various thread-safe libraries. Further, more experimental evaluation is needed. This should include a comparative work that would compare the benchmark programs across techniques and tools.

5. References

[1] K. Sen and G. Agha, “A Race-Detection and Flipping Algorithm for Automated Testing of Multi-Threaded Programs”, Haifa Verification conference, Haifa, Israel, 2006, Revised Selected Papers. Lecture Notes in Computer Science 4383, Springer, 2007.

[2] M. Wojcicki and P. Strooper. "Maximising the Information Gained from a Study of Static Analysis Technologies for Concurrent Software”, Journal of Empirical Software Engineering. 2007.

[3] F. Chen and T. F. Serbanuta and G. Rosu. “jPredictor: A Predictive Runtime Analysis Tool for Java”, ICSE. 2008.

[4] F. Chen, M. d'Amorim and G. Rosu, “Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP” , RV'05, ENTCS 144, issue 4, pp 3-20. 2005.

[5] N. Kidd, T Reps, J. Dolby, and M. Vaziri, “Static Detection of Atomic-Set Serializability Violations”, TR-1623, Computer Sciences Department, University of Wisconsin, Madison, WI, October 2007.

[6] M. B. Dwyer, S. Person, and S Elbaum, “Controlling Factors in Evaluating Path-Sensitive Error Detection Techniques”, FSE. 2006.

[7] N. Rungta and E. G. Mercer, “Generating Counter-examples through Randomized Guided Search”, SPIN Workshop on Model Checking of Software, Berlin, Germany, 2007.

[8] N. Rungta and E. G. Mercer, “Hardness for Explicit State Software Model Checking Benchmarks”, 5th IEEE International Conference on Software Engineering and Formal Methods, London, U.K, 2007.

[9] Y Nir-Buchbinder and S. Ur “ConTest Listeners: a Concurrency-Oriented Infrastructure for Java Test and Heal Tools”, International Workshop on Software Quality Assurance, 2007.

[10] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Comparative Assessment of Testing and Model Checking Using Program Mutation", Workshop on Mutation Analysis (Mutation 2007), UK, 2007.

[11] J. S. Bradbury, J. R. C. and J. Dingel. "Mutation Operators for Concurrent Java (J2SE 5.0)", In Proc. of the Workshop on Mutation Analysis (Mutation 2006*)*, pages 83-92, Raleigh, North Carolina, USA, 2006.

[12] Y. Eytani and T. Latvala. “Explaining Intermittent Concurrent Bugs by Minimizing Scheduling Noise”. Haifa Verification conference, Haifa, Israel, 2006, Revised Selected Papers. Lecture Notes in Computer Science 4383, Springer, 2007.

[13] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. "Noise Makers Need to Know Where to be Silent – Producing Schedules that Find Bugs". International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA), 2006.

[14] C. von Praun and T. R. Gross, “Static Detection of Atomicity Violations in Object-Oriented Programs”, Journal of Object Technology 3(6): 103-122 (2004)

[15] K. Havelund, S. D. Stoller, and S. Ur, “Benchmark and framework for encouraging research on multi-threaded testing tools”, Workshop on Parallel and Distributed Testing and Debugging, 2003.

[16] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur, “Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs”, Concurrency and Computation: Practice & Experience, 19(3): 267-279 (2007).

[17] Y. Eytani and S. Ur, “Compiling a Benchmark of Documented Multi-threaded Bugs”, Workshop on Parallel and Distributed Testing and Debugging, 2004.

[18] M. Naik, A. Aiken and J. Whaley, “Effective Static Race Detection for Java”, Conference on Programming Language Design and Implementation (PLDI), 2006.

[19] M. E. Keremoglu, S. Tasiran, T. Elmas, “Classification of Concurrency Bugs in Java Benchmarks by Developer Intent**”,** Workshop on Parallel and Distributed Testing and Debugging, 2006.

[20] L. Wang and S.D. Stoller, “Runtime Analysis of Atomicity for Multi-threaded Programs”, IEEE Transactions on Software Engineering, 32(2):93-110, 2006.

[21] V. Mutilin, “Concurrent Testing of Java Components Using Java PathFinder”, ISOLA. 2006.

[22] C. Artho, Z. Chen, S. Honiden, “AOP-Based Automated Unit Test Classification of Large Benchmarks”, AoAsia. 2007.

[23] Rachel Tzoref, Shmuel Ur, Elad Yom-Tov: "Instrumenting Where It Hurts: An Automatic Concurrent Debugging Technique". ISSTA 2007:

[24] Shmuel Ur, “Special Issue: Parallel and Distributed Systems: Testing and Debugging (PADTAD)”. Concurrency and Computation: Practice and Experience 19(3): 265-266 (2007)

[25] http://www.haifa.ibm.com/projects/verification/padtad/ index.html

[26] Eitan Farchi, Yarden Nir, Shmuel Ur, "Concurrent Bug Patterns and How to Test Them", Workshop on Parallel and Distributed Testing and Debugging, 2003.

[27] Robert O'Callahan, Jong-Deok Choi, "Hybrid Dynamic Data Race Detection", PPOPP 2003.

[28] C. Hammer, J. Dolby, M. Vaziri, F. Tip, "Dynamic Detection of Atomic-Set-Serializability Violations", Accepted for publication at the 30th International Conference on Software Engineering (ICSE'08), Germany, May 2008