Concurrency Mistakes That Matter (discussion topic)

William Pugh
Dept. of Computer Science
Univ. of Maryland
College Park, MD
pugh@cs.umd.edu

ABSTRACT

There has been much concern and attention on the possibilities of errors arising from the use of concurrency. Bugs such as data races, deadlocks and insufficient atomicity can cause serious failures, and can be difficult to test for or to reproduce. Writing correct concurrent isn't easy, many misconceptions abound, and many developers do not have adequate training in the topic. As multicore processors start becoming ubiquitous, many worried that pervasive problems with concurrency bugs would prevent their wide adoption or effective use. Static analysis for concurrency bugs is of questionable value and sees limited use.

Despite all of these concerns, some concurrency bugs do not yet seem to be the big problem that many feared they would be. There is some evidence that certain kinds of concurrency bugs (e.g., deadlock due to inconsistent lock ordering) manifest themselves very rarely in practice. Other concurrency mistakes may be very real and serious problems.

Rather than concern ourselves with the huge universe of potential concurrency bugs, we need a better understanding of the concurrency bugs that cause problems in practice, and practical techniques to help identify and prevent or eliminate them. Rather than being a research or position paper, this submission calls for a round-table discussion of the topic.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; D.2.4 [**Software/Program Verification**]: Reliability

General Terms

Experimentation, Reliability, Security

Keywords

FindBugs, static analysis, bugs, software defects, bug patterns, false positives, Java, software quality, concurrency

A drunk loses the keys to his house and is looking for them under a lamppost. A policeman comes over and asks what he's doing.

"I'm looking for my keys" he says. "I lost them over there".

The policeman looks puzzled. "Then why are you looking for them all the way over here?"

"Because the light is so much better here".

1. DISCUSSION

In most systems that support parallelism using threads, locks, and shared memory, writing correct code can be very daunting. Internal locking protocols are often subtle and tricky, and API's for classes documented as thread-safe can be tricky. For example, StringBuffer is documented as being thread safe, but if during an operation `sb1.append(sb2)`, another thread modifies `sb2`, an atomicity failure can cause an unexpected runtime exception or other data corruption [3]. Is this a defect that needs to be fixed, or simply an undocumented corner case of the API?

While FindBugs [4] reports various kinds of concurrency errors, concurrency is perhaps one of the least successful and accurate bug categories reported by FindBugs [5, 7]. Even when it finds clear violations of recommending programming practice (e.g., ignoring the return value of `ConcurrentMap.putIfAbsent`), the violations only manifest themselves under very particular situations and it is unclear how much problem they cause in practice. Some may manifest themselves by computing incorrect results, but never be noticed. Others, such as deadlock, can be easily detected. The topic of deadlock due to inconsistent lock ordering is a well studied topic, and there have been a number of papers on static and dynamic analysis to detect such problems. But even without the use of such techniques, in over a year of monitoring production Java runtimes at Google, only a single such deadlock was detected (across all of Google's servers).

Even if concurrency bugs don't cause huge problems now, that might change. Most applications on run on systems with a small number of cores. A problem that manifests itself rarely if at all on a 4 core machine may cause frequent problems on a 128 core machine. But I think the community has very little understanding of the concurrent mistakes that are currently causing problems, and without that, research on concurrency mistakes is in danger of looking where the light is best, rather than where we could improve things the most.

This isn't a research paper, or even really a position paper. Rather, I think the workshop and community would benefit from a discussion and cataloging of the concurrency mistakes that are actually causing problems, and techniques that might be practically effective in finding them. Without a good bestiary of concurrency bugs, effective research in the field will be limited. Some possible discussion questions:

- Do dataraces and atomicity failures tend to occur directly in user code, or in non-thread safe system classes inappropriate shared between threads by user code?
- Are dataraces or atomicity problems more of a problem in practice?
- What problems are caused because developers didn't understand the intended concurrency usage rules for an API?
- Should libraries check for threading violations and fail with runtime exceptions when they are detected? For example, should two concurrent unsynchronized updates of a HashMap cause a runtime exception ? All the time or just if a special enable-concurrency-assertions flag is set (as is done at Azul)? What should a JVM do when deadlock is detected? [1]
- What are effective techniques for testing concurrent code that scale to production services? [9]
- Which APIs and idioms are particularly difficult to use, or are often misused in practice?

There have been other attempts to understand and catalog concurrency bug patterns [8, 2, 10, 6, 11]. This discussion would continue and further those efforts.

At the workshop, I'd plan to introduce the topic in no more than 5 minutes, than open it up to discussion.

2. REFERENCES

[1] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. A case for system support for concurrency exceptions. In *HotPar'09: Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 6–6, Berkeley, CA, USA, 2009. USENIX Association.

[2] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society.

[3] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM.

[4] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM.

[5] D. Hovemeyer and W. Pugh. Finding concurrency bugs in java. In *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[6] M. E. Keremoglu, S. Tasiran, and T. Elmas. A classification of concurrency bugs in java benchmarks by developer intent. In *PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 23–26, New York, NY, USA, 2006. ACM.

[7] D. Kester, M. Mwebesa, and J. S. Bradbury. How good is static analysis at finding concurrency bugs? In *Proc. of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010)*, Sept. 2010.

[8] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not.*, 43(3):329–339, 2008.

[9] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[10] F. Otto and T. Moschny. Finding synchronization defects in java programs: extended static analyses and code patterns. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 41–46, New York, NY, USA, 2008. ACM.

[11] S. K. Sahoo, J. Criswell, and V. Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 485–494, New York, NY, USA, 2010. ACM.