ScheduleInst: Using DynInst to Fuzz Test Multithreaded Program Schedules

Robert Wespetal Zachary Welch

December 1, 2014

Abstract

1 Introduction

Designing, coding, and debugging multithreaded programs present a number of unique challenges to developers. Unlike singly threaded programs in which (barring some exception or interrupt) the order of operations is well defined, multithreaded applications have an additional degree of freedom in that the next instruction to run could be any of a set of instructions. The ability to run multiple operations in roughly any order is part of the appeal of parallelism, as orthogonal sets of work can be run simultaneously on different processors. However, It is rarely the case that a program will launch threads that never interact, and a significant part of the cognitive effort that goes into parallel programming is ensuring that these interactions are handled in a safe way. Safe could mean restricting access to a data structure so only one thread may change it at a time, or it could mean that some sort of necessary order is enforced, such as allocating memory in one thread before it is used in another. When these cases are not handled correctly, the application can deadlock, crash, or corrupt valuable data. And because no single thread is guaranteed to be the next to run, the program could be launched with identical inputs and produce correct results. This non determinism makes multithreaded applications difficult to debug.

The source of this non determinism is the OS scheduler, which must take into account a host of factors to decide which thread gets control of the CPU next and for how long. In theory any interleaving of thread operations is possible, but we would argue that in practice the vast majority of runs have highly similar patterns of execution. Assume for instance that there is a deadlock bug in a code of software that is triggered in some subset of of thread interleavings. If these interleavings are common, it is likely that the developers would have found the bug through normal testing. Latent multi threading bugs then are likely in low-probability interleavings of threads, which is why they can be very difficult to find.

There are numerous ways to tackle this problem of bugs hiding in low-probability interleavings. To us the two most obvious possibilities are that you could run the same tests over and over again with the probabilistic argument that you will generate the bug in the limit or you could force every possible interleaving of execution.

The problem with both solutions is they do not scale; trying to debug something like Apache this way is functionally impossible. Other, more sophisticated techniques attempt to use a range of techniques, from machine learning to static and dynamic analysis, to intelligently guess which runs might lead to bugs. We found this route unappealing due to a high bar of domain knowledge necessary to use any single technique. Instead we were drawn to the relatively simple but effective technique of fuzz random testing. Fuzzing a program involves passing programs randomly generated input until the program crashes. We embraced this method of testing in multithreaded programs; we believe that simply by randomly altering the priorities of threads during execution, we can stumble upon potential bugs more quickly than some brute force technique. Combining this random reprioritization with logging information about the state of each thread during the error-inducing execution allows for identification of bugs. In fact, using our tools logging mechanisms we were able to quickly identify bugs in large code bases we were unfamiliar with. Our tool, scheduleInst, can attach to program binaries and dynamically instrument the program to employ random thread priorities during execution, as well as log details of these executions. We have used scheduleInst to find potential bugs in pbzip2, ffmpeg, Apache, multithreaded wget, etc IN SECTION, WE...

2 Related Work

Fuzz Random Testing originated as a method for testing the reliability of Unix utilities by passing random character strings as input. Fuzzing asserts correct programs should be robust to this random input, and programs that crash or hang indefinitely under this stream of random data must then have bugs. Random input strings can then be generated until bugs are found or the developer is confident that their tool is robust to random inputs. Fuzzing, like many bug finding tools, can only reveal bugs it finds; the lack of bugs from Fuzz runs does not guarantee any sort of correctness. This relatively simple approach to debugging has been found to be surprisingly effective; the authors in the original study found bugs in between 25 and 33 percent of utilities tested across multiple versions of Unix.

Dyninst is a tool and API which allows users to instrument and patch binary code, which may or may not be running at the time it is instrumented. Dyninst has been used to do a wide variety of tasks, from to . Dyninst provides a major convenience in that large programs can be instrumented to perform difficult and potentially tedious tasks, for example reporting the number of calls to every function in an execution, quickly and without the need to alter the binary in question. In Dyninst parlance, a mutator program is one which calls the Dyninst API to instrument some other program, the mutatee. In about 100 lines of C++ code, a mutator which tracks the number of function calls can be written; any binary can then be used as a mutatee. We use Dyninst primarily as a way of controlling and manipulating thread priorities in a uniform manner across test applications.

Using dynamic instrumentation to control or alter the execution of threads is not novel to ScheduleInst.

Program testing and bug finding tools have long been a research area, and research in multithreaded testing and debugging techniques has been particularly active in recent years.

3 Methods

ScheduleInst is a tool designed to alter thread priorities in some user specified way, with the goal of moving away from the normal execution path. It is our belief that testing these less probable interleavings of code that bugs will be revealed in a relatively short amount of time. Threads are altered by using Dyninst to insert code snippets which alter the priority of a thread at a user-specified place. Dyninst is also used to provide sufficient logging to discover approximately where and how a bug manifested itself. ScheduleInst is primarily designed to find deadlocks and program crashes. Other types of concurrency bugs like race conditions can be tested in some cases, for instance in pbzip2 we can check if any data was corrupted in the process of compressing the file, though our logging in that case would likely be unhelpful.

ScheduleInst has a range of options which can affect how often thread priorities are altered. Users can specify the sequence of priorities to pass to the mutatee to be randomly generated, all priorities set to equal, or users can provide their own sequence to allow arbitrary scheduling schemes to be emulated using ScheduleInst. Users can also specify at what points in execution priorities can be changed; by default priorities are set when a thread is created and after all thread synchronization function calls. The granularity of logging can also be altered by the user; by default we log entrances and exits of all user-defined functions (for implementation reasons we do not instrument signal handlers) as well Pthread functions. We allow this granularity to be changed because we have found for some applications like ffmpeg the thousands of user-defined functions generates prohibitively large execution logs for no additional information.

Information about each function entry and exit is written to a single log file. The name of the method in question, the thread executing it, and whether an entry or exit is occurring is logged. For long running applications, this log can grow arbitrarily large, and we currently make no effort to remove old information. The standard output and standard error of the application are also logged in a separate file for reference, and a core dump is saved if one is generated during the run. Using the set of saved information and a set of scripts for some additional processing, we can provide useful information about the recent events in the source code leading up to a crash or deadlock.

While ScheduleInst can be reasonably expected to increase the diversity of execution interleavings, it cannot alone robustly test a multithreaded program. In a program with many potential branching paths of execution, ScheduleInst can only work in the one that has been chosen to run. ScheduleInst is best suited for use in conjunction with some suite of stress tests that provide good code coverage. This point may seem obvious, but we feel it needs stressing. Several of the applications tested are monolithic in design, with a single application having many possible, mostly disjoint modes. Having little to no experience with the use and operation of these applications made it difficult to provide adequate loadings of them for testing.

4 Experimental Setup

4.1 Environments used

4.2 Programs tested

4.3 How we decided deadlock/inifite loops

5 Results

5.1 Bugs found for each program

5.2 Types of bugs

5.3 Known or new bugs

6 Discussion

6.1 Why we might have found these bugs

6.2 Why we might be missing some bugs that are there

6.3 Heuristics can work great or crap

7 Conclusions

7.1 Reiterate major points

7.2 talk about future avenues of investigation

8 Thanks

Bill

Bart?

References

[1] J. E. Smith. Characterizing computer performance with a single number. *Commun. ACM*, 31(10):1202–1206, October 1988.

[2] Michael J Donahoo and Kenneth L Calvert. *TCP/IP sockets in C: practical guide for programmers.* Morgan Kaufmann, 2009.

[3] Socketpair() in c/unix. `http://stackoverflow.com/a/11461302`. Accessed: 2014-09-10.