ScheduleInst: Using Dyninst to Fuzz Test Multithreaded Program Schedules

Robert Wespetal Zachary Welch

December 3, 2014

Abstract

ScheduleInst is a tool designed to Fuzz Test multithreaded programs. SecheduleInst uses dynamic instrumentation to force applications to use real time threads. By frequently altering real time thread priorities, ScheduleInst attempts to reveal concurrency bugs by generating unusual application executions. By randomly changing real time thread priorities at thread synchronization calls and with the help of dynamically instrumented logging techniques, ScheduleInst has been successful in finding real bugs in large, actively used programs. We test ScheduleInst with five popular open source applications and three simple proof of concept cases.

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 2, we discuss Related Work, in Section 3 we detail ScheduleInst and its approach to finding bugs. In Section 4 we describe our experimental testing of a number of open source projects

2 Related Work

2.1 Original Fuzzing

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.

2.2 Dyninst

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.

2.3 Multithreaded Debuggers

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. For the defualt randomly-generated thread priorities, scheduleInst allows users to specify a random seed. This feature allows users to easily recreate the environment which lead to a failed run. Note that this is not the same thing as saying the failing runs are

guaranteed to be reproducible; scheduleInst can only recreate the priorities of the threads, it cannot force the OS to yield the CPU at exactly the same point in the execution as it did in a failing run. Additionally, we have found that in practice it is difficult to consistently reproduce runs with identical priority sequences, and the rate seems to be dependent on the specifics of a given bug. In some cases the bug only revealed itself on 2% of runs with a known buggy priority sequence. In other cases the bug appears regularly. Due to the fact that consistently generating the bug is not guaranteed, the rationale for providing thorough logging information is that we want to make it as easy as possible to find the bug with a single run.

3.1 Default scheduler vs Real Time Threads

ScheduleInst uses real time threads to exert some level of control over how threads run. POSIX threads provide two real time options: SCHED_FIFO and SCHED_RR for First-In-First-Out and Round Robin scheduling respectively. By default ScheduleInst uses SCHED_RR, though in practice we found little difference between the two. Real time schedulers run by scheduling higher priority threads first until they give up the CPU by going to sleep or ending. In round robin the concept of time slices only exists for threads with equal priorities. The value of using real time threads for ScheduleInst is that thread priorities can be explicitly set at any point during program execution. Unlike the default Linux scheduler (SCHED_OTHER) which uses a multilevel feedback queue and internally alters the priority of threads based on how they use their time slices, realtime scheduling allows us to declare a certain thread more important to the operating system. In real time threads, a lower priority only gets the CPU when all higher priority threads have been put to sleep. This level of control is not perfect, but we feel using real time threads is a happy medium between doing nothing and writing our own kernel scheduler to suite our needs. ScheduleInst forces all mutatee applications into the user-specified real time thread scheduling algorithm by using Dyninst to insert the necessary code. By switching thread priorities often enough we hope to throw the application in question into atypical yet valid states in the hope of finding bugs. Note that in many cases this use of real time threads and the instrumented logging will lead to decreased performance.

3.2 Instrumenting methods

At each instrumentation point, scheduleInst checks the current threads scheduler, sets it to the real time scheduler if necessary, and alters the threads real-time priority to some other (usually random) valid priority value. If the pthread_create function is one of the instrumentation points, the new thread will be created with the round robin scheduler and appropriate priority. Initially scheduleInst only set thread priority as part of the create call, but after testing we found setting thread priorities at the time of thread creation and after each synchronization mechanism call (eg mutex lock) stresses the application more than simply setting priorities initially and allowing the program to run. Intuitively, our goal in altering thread priorities after synchronization mechanisms is to encourage threads to give up the cpu immediately after grabbing a lock, which would allow other threads to run and potentially enter a deadlock state if one exists.

3.3 logging

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.

3.4 Targeted Bugs

As previously stated, scheduleInst is primarily focussed on revealing bugs which lead to a premature end to execution. This early stop can be due to a program crash (often a segmentation fault) or a deadlock/infinite loop where the program remains alive but makes no progress and hangs indefinitely.

We have designed ScheduleInsts default behavior to specifically target these two types of errors. In the simple deadlock example provided below, the system only deadlocks if both threads grab their first lock before either grabs their second one. In the course of execution using the default Linux scheduler, the likelihood of this is relatively small. Using ScheduleInst, however, the thread priority is changed immediately after a lock is grabbed. This increases the possibility that the thread will have to give up its CPU before it has a chance to grab the second lock. This gives the slower thread a better chance to catch up, and in general increases the likelihood of execution running into a deadlock.

Deadlock Thread 1

```
1 ...
2 mutex_lock(lock1);
3 mutex_lock(lock2);
4 //do some work
5 mutex_unlock(lock2);
6 mutex_unlock(lock1);
```

Deadlock Thread 2

```
1 ...
2 mutex_lock(lock2);
3 mutex_lock(lock1);
4 //do some work
5 mutex_unlock(lock1);
6 mutex_unlock(lock2);
```

Likewise in the code below, there is the possibility for dereferencing a null pointer in threadSecond if it is run before the pointer is allocated in threadFirst. Normally, if the thread responsible for threadFirst is created before the thread responsible for threadSecond, threadFirst will likely run before threadSecond and the code will run without crashing. However, when using real time thread priorities, it is possible that a highly prioritized threadSecond could preempt a lowly prioritized threadFirst and cause the bug to reveal itself in a crash.

Bad Memory Access Thread 1

```
1 int * myPtr = NULL;
2 void threadFirst(){
3
4 myPtr = malloc(sizeof(int));
```

Bad Memory Access Thread 2

```
1 void threadSecond(){
2
3 *myPtr = 4;
4 printf("%d\n"*myPtr);
```

In both cases ScheduleInst increases the chance that the bug will occur in execution. We may also encourage other types of bugs like race conditions and infinite loops, but the default choices made by ScheduleInst specifically target bugs which incorrectly use synchronization mechanisms. Our logging mechanisms are also targetted at making these bugs easy to locate once they occur. ScheduleInst may make it more likely that a program corrupts its data, but currently ScheduleInst would have no way to locate this error in the source code unless it generated a crash.

3.5 Code coverage issue

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

To test ScheduleInst, we took a number of sample applications, instrumented them using ScheduleInst, and repeatedly ran each application under some load a large number of times. The goal here is not to show that ScheduleInst finds bugs in most of its runs, or that it finds many bugs in each application, the primary goal at this stage of development is to study whether ScheduleInst finds bugs faster than simply running the application many times and hoping to stumble on a bug. Since ScheduleInst dynamically instruments logging information and scheduling controls, both of which could alter the execution of the application, we test results using only logging and both logging/schedule controls. This should demonstrate whether the use and manipulation of real time threads have an effect on the likelihood of bugs being revealed. These tests were run in batch; a log of all thread interactions, stdout and stderr, and a core file if it was generated were saved for each iteration.

4.1 Environments used

Five department Virtual Machines were used to run all tests. The VMs all used Ubuntu 12.04; similar results were found with some initial tests using ?????. Dyninst 8.2.1 was used as our Dyninst installation.

Application	Version	Type
pbzip2	Version	File Compression
aget	Version	File Downloading
ffmpeg	Version	Video Compression
mysql	Version	Database
apache	Version	Web Server
Toy ThreadSafe	N/A	Safely changes, prints global value
Toy Deadlock	N/A	ThreadSafe with deadlock
Toy Bad Mem Access	N/A	ThreadSafe with null pointer access

Table 1: Applications Tested

All test applications were compiled from source with debugging information to allow Dyninst to correctly log application specific methods. A list of applications tested and their versions can be found in Table 1. These applications were chosen because they were popular Open Source applications known to make heavy use of PThreads. Additionally we include our original test applications as a proof of concept to show that ScheduleInst finds bugs that exist but does not find bugs in a simple application with no concurrency bugs.

4.2 Application Loadings

For each real application tested, we had to choose some way of loading the application. The toy applications have no external inputs and thus do not have this problem. For each application the choice of loading is an important factor in the type of bugs we might find. We describe our application loadings below; in all cases we feel our loadings are representative of a fairly normal use case of the application.

- pbzip2: To test the parallel compressor, we generated a random 100 MB file using Linux's dd command. pbzip2 would then attempt to zip this file using 100 threads specified on the command line.
- aget: Download large file (Boost source) from department web server.
- ffmpeg: BOBBY
- mysql: Attach to server and run mysql provided test suite
- apache: BOBBY

It is likely that different bugs would be found with different loadings, and due to our lack of knowledge about the implementation details of these applications, it is possible some of these loadings are inappropriate to provide a good test of a system. Regardless, none of our loadings are invalid inputs to their respective programs and all work successfully with no dynamic instrumentation, so we feel that while we may not be testing these tools as well as possible, we are finding bugs in them. In addition to our logging and full ScheduleInst runs, we tested all applications extensively with no dynamic instrumentation and never hit a crash.

Application	Just Logging		ScheduleInst	
	# Runs	# Bugs Found	# Runs	# Bugs Found
Toy ThreadSafe	100	X	100	X
Toy Deadlock	100	X	100	X
Toy Bad Mem Access	100	X	100	X
pbzip2	100	X	100	X
aget	100	X	100	X
ffmpeg	100	X	100	X
mysql	100	X	100	X
apache	100	X	100	X

Table 2: Number of Potential Bugs Found

Application	# Total Bugs	# Seg Faults	# Hangs
pbzip2	X	X	X
aget	X	X	X
ffmpeg	X	X	X
mysql	X	X	X
apache	X	X	X
Toy ThreadSafe	X	X	X
Toy Deadlock	X	X	X
Toy Bad Mem Access	X	X	X

Table 3: Types of Bugs found

4.3 How we decided deadlock/inifite loops

One important design decision is how to decide when an application has deadlocked or infinite looped. For program crashes, we simply check for the existance of a core file after every application has reached the end of it's execution. Deadlocks, however, will run indefinitely without intervention. We decided to add a timeout to our framework that would kill the running process after 4X the run time of a normal run had elapsed. So for instance a pbzip2 run on our 100 MB file took 30 seconds in all the uninstrumented runs we performed, hence we set a timeout of 120 seconds for our instrumented pbzip2. Once the testing was complete and certain runs were flagged as potential deadlocks, it was a simple matter of manually inspecting the thread log and suspected location of the code to decide if a deadlock had indeed occured. In practice all of the runs flagged by ScheduleInst turned out to be deadlocks.

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?