Deterministic Replay of Concurrency Bugs
— Background and Progress Report —

Hugo Gibson
hg311@doc.ic.ac.uk

Supervisor: Dr. Alastair Donaldson
Course: CO530/533, Imperial College London

8th June, 2012

1 Abstract

As a result of the large numbers of interleaving threads, detecting errors in concurrent sofware is a challenging task.

Debugging concurrent software is notoriously difficult using traditional tools such as gdb. Debuggers can slow down execution, hiding many race conditions and other concurrency bugs from the developer.

KLEE-Race has been effective in detecting concurrency bugs but such bugs are difficult to reproduce. Some bugs depend on very specific conditions arising in a given execution and the chances that these conditions will occur may be slim.

A replay tool which takes a pre-defined schedule produce by KLEE-Race will allow efficient and fast reproduction of concurrency bugs and allow developers to observe the conditions under which bugs can occur.

2 Background

The aim of this project is to provide deterministic replay in concurrent programs. Traditionally, concurrent programs are very hard to debug and often once a bug has been found it can be very difficult to replay the exact situation that caused such bug to arise. The aim of this project is to build a tool that will be given a schedule and will follow the schedule to replay a program according to the schedule to reproduce the bug.

The tool will take a schedule produced by KLEE-Race and reproduce the exact conditions under which a bug found by KLEE-Race occurred. This will allow developers to observe the non-deterministic behaviour of their programs and eliminate bugs that might have gone unnoticed using traditional debugging techniques.

There are currently very few tools which provide determinstc replay for POSIX threads. There is a tool called Tinertia [1] which specifically targets POSIX threads and provides a dynamic execution trace for help in debugging concurrent programs but it does not provide a replay feature like the one this project aims to provide.

For other platforms I have looked at two other existing tools. DejaVu [2] which targets the Java Virtual Machine and provides deterministic replay for multithreaded Java programs. DejaVu works by caputring a logical thread schedule, identifying critical events in this thread schedule and then providing deterministic replay to capture the bug found in the logical thread schedule. The second tool I have looked at is LEAP, which also targets the Java Virtual Machine. LEAP aims to capture the thread access history of each shared variable and use theoretical models to guarantee their correctness.

Another more recent development in concurrent debugging is the Output-Deterministic Replay (ODR) tool [3]. It solves the output-failure replay problem, which is to ensure that all of the failures visible in the output of the original program are visible in replays of the same program. ODR achieves this by using output determinism which dictates that the replay run outputs the same values as the original run. ODR does not use traditional replay features such as thread

[1] Choi J-D. and Srivinasen H. 1998

[2] http://www.eecs.berkeley.edu/~jalbert/papers/fse10.pdf

[3] http://www.sigops.org/sosp/sosp09/papers/altekar-sosp09.pdf

scheduling. In fact output determinism does not guarantee that the same read and writes of variables will occur in a given program run – that is the input might change but the output will not. For this reason ODR has less in common with the replay tool this project aims to create, but it is still an interesting way to think about debugging concurrent software.

Of the tools mentioned above, I think that DejaVu has the most in common with my tool as my tool will follow a thread schedule, much like DejaVu, and attempt to recreate the situation in which a specific bug occurred. LEAP uses a global semaphore in its replayer to sleep and wake threads on demand. I have used a similar technique, using a pthread_condition_varable to achieve the same waking and sleeping effect.

3 Progress

So far I have familiarised myself the relevant documentation for pthreads and intercepting functions in the Linux kernel, as well the background papers on LEAP and DeJaVu. I have experimented making concurrent programs with deterministic outputs using the pthreads library. I have also experimented with creating non-deterministic programs to simulate race conditions and deadlocks. The aim of such experiments was to gain greater familiarity with using the pthreads library and become more familiar with basic concurrent programming techniques such as using mutexes and condition variables to synchronize thread operations and also to raise my awareness about some of the concurrency problems that can occur.

I have also experimented with function intercepting using the LD_PRELOAD environment variable in the Linux kernel. Intercepting calls to pthread_create(...) allows a schedule to be checked to determine if the thread is eligible to be executed.

One of the main difficulties I encountered while intercepting pthread_create(...) was that if the ID of the thread being created did not match the ID of the thread in the schedule then the thread would never be created, resulting in the tool hanging if this thread was the first to be created.

For example consider the following scenario:

> Imagine the schedule given to the tool is as follows: [3,2,1] (with the numbers representing the thread IDs).
>
> Then on the first call to pthread_create(...), the assigned ID of the thread is 1.
>
> Thus when the schedule is checked, the IDs will not match as 3 is the first thread in the schedule and the thread being started has ID 1.
>
> So the program will hang indefinitely.

I have also come across other complications in function intercepting in the Linux kernel, since many of the available tutorials and web posts on the Internet are outdated and therefore the function intercepting that they describe do not work at all. While I have overcome this problem and managed to intercept functions, I have decided focus on creating a wrapper class for the POSIX threads library in Linux rather than using function intercepting.

While implementing the wrapper class, I have found it challenging to force threads to follow a particular schedule. At times, I was calling the wrong functions at the wrong times, resulting

in threads running before they were meant to. Reasoning about the correct order of execution solved these issues.

So far I have started to implement a wrapper library for pthreads in the Linux kernel. Currently, a user can pass a schedule to the program and call my thread creation function to get the threads to follow the schedule they have provided. For the time being I have only focused on pthread creation.

A call to intercept_pthread_create(...) creates a new thread to check the schedule provided and to determine whether or not the thread is eligible to be run. The call also assigns the thread an ID, based on the order in which it is created. The calling program (i.e int main(..)) is Thread 0. The first independent thread to be create is Thread 1 and so on with each new call to intercept_pthread_create(...), incrementing the Thread ID. Only a thread, which appears in the schedule at the time of the call will be run. Any other threads must wait until notified. This scheduling is achieved using a condition variable and a loop to check the thread's position in the schedule. A thread will be blocked until its ID matches the ID in the schedule. Thread IDs can occur more than once in a given schedule, since a thread may perform synchronisation actions after its creation, such as locking and unlocking a mutex or signalling and waiting on a condition variable.

I have made threads wait and notify other waiting threads and run according to their schedule by using two functions: step_and_notfiy(...) and wait(...). step_and_notfiy(...) increments a global variable to move the schedule along and notfies the threads that are waiting on the condition variable. wait(...) causes threads to wait until their ID matches the ID shown in the schedule. This is done by using a condition variable and having threads obtain the variable before they can continue executing. Using a condition variable in this way allows accurate scheduling of threads. A thread will not terminate until it no longer appears in the schedule or a call to pthread_join(...) is made.

Below is a code listing of the step_and_notfiy(...) and wait(...) functions:

```
/*wait*/
void wait(int thread_id)
{
  pthread_mutex_lock(&step_mutex);
  while(thread_id != schedule[step] && step < schedule_size()-1)
    {
      fprintf(stderr, "Thread %d, sleeping...\n", thread_id);
      pthread_cond_wait(&condition, &step_mutex);
    }
  pthread_mutex_unlock(&step_mutex);

}


/*step and notify */
void step_and_notify()
{

  pthread_mutex_lock(&step_mutex);

  step++;


  pthread_cond_broadcast(&condition);
  pthread_mutex_unlock(&step_mutex);
}
```

4 Aims

For the time being, I will continue working on the wrapper function intercept_pthread_create(...) which deals with thread creation by creating some schedules for the tool to follow as well as some incorrect schedules to demonstrate concurrency bugs and show the points at which such bugs occurred.

After I have completed the wrapper for thread creation I will focus on joining threads before moving on to mutex locking and unlocking and condition variables, and from there I will turn to other functions in the pthread library. Each call to pthread_mutex_lock(...) will again require the schedule to be checked to determine if the calling thread is allowed to access the mutex variable. I will implement this using a pthread_mutex_trylock(...) statement as this will make the calling thread wait until it obtains the mutex, thus ensuring that the schedule is followed correctly.

The aim is to provide a wrapper class for the rest of the pthread library in a similar way as I have already done with pthread_create(...). This will be done using similar techniques. That is checking a schedule on calls to pthread functions and forcing threads either to wait or continue depending on their position in the schedule.

The final aim for the project is to create a tool that takes a known trace to a bug from the KLEE-Race tool, develop a schedule for this trace and feed the schedule into the tool, which will then follow the schedule accordingly and reproduce the conidtions which caused the bug to occur. The tool itself will only offer a replay feature rather than the record and replay features found in other similar tools, since the record feature is provided by KLEE-Race.

While I am using a wrapper library for pthreads, the idea is that a user of the tool should not have to explicitly call my library functions to gain deterministic replay for their program. To overcome this, I will replace calls to the pthread library with calls to my wrapper library either by creating a simple script or by using "#include" guards. Using a script will allow greater usability for the tool, since a user can simply run the script rather than having to include libraries in their source code.

5 Literature Survey

There is very little literature on pthread wrapping and deterministic replay for concurrent programs. Most of the applicable literature relates to debugging concurrent software rather than using record and replat features to simulate the conditions under which bugs occur. These publications have been particularly useful in understanding the problems inherent in concurrent software and have allowed me to gain a greater understanding of the reasoning involved in correctly evaluating non-deterministic concurrent execution.

As a result of the dearth of literature I have focused on reading documentation about pthreads, as well as blog posts and web pages written by people who have attempted similar things.

Publications:

Altekar G. and Stoica I. *ODR: Output-Deterministic Replay for Multicore Debugging*, Berkeley.

Butenhof D. *Programming with POSIX Threads*, 1991, Addison Wesley.

Carver, R.H Tai, K.-C. "Replay and testing for concurrent programs," *Software, IEEE* , vol.8, no.2, pp.66-74, March 1991
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=73751&isnumber=2484

Choi J-D. and Srivinasen H. *Deterministic Replay of Java Multithreaded Applications*, 1998, New York.

Huang J. Liu P. and Zhang C. *LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs.*
URL: http://www.cse.ust.hk/ smhuang/academic/leap.pdf

Jalbert N. and Sen K. *A Trace Simplification Technique for Effective Debugging of Concurrent Programs.*
URL: http://www.eecs.berkeley.edu/ jalbert/papers/fse10.pdf

Nuno Machado E.A. *Fault Replication in Concurrent Programs.*
URL: http://www.gsd.inesc-id.pt/ ler//reports/nunomachadoea.pdf

Russinovich M. and Cogswell B. *Replay for Concurrent Non-Deterministic Shared-Memory Applications*, Oregon.

Pthread Documentation and Tutorial Websites

http://www.linuxforu.com/2011/08/lets-hook-a-library-function/

https://computing.llnl.gov/tutorials/pthreads/

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

http://linux.die.net/man/7/pthreads

http://www.manpagez.com/man/3/pthread/