PinDB: A Pin-based Debugger for Multi-threaded Programming

Xiaoming Shi
Department of Computer Sciences, UW Madison
xiaoming@cs.wisc.edu

Venkatesh Karthik Srinivasan
Department of Computer Sciences, UW Madison
venk@cs.wisc.edu

Madhu Ramanathan
Department of Computer Sciences, UW Madison
madhurm@cs.wisc.edu

Yiqing Yang
Department of Computer Sciences, UW Madison
breakds@cs.wisc.edu

1. Abstract

Debugging multi-threaded programs is a tough job, primarily due to the complex interactions between threads, in the form of shared variables and the inherent non-determinism involved in the execution order of threads. There is very limited support built into commercial debuggers like GDB to debug multi-threaded programs in a user-friendly manner. The aim of this project is to build a tool for debugging multi-threaded programs, with the help of Pin, a dynamic binary instrumentation and analysis tool. Our Pin-based tool, PinDB provides the basic features such as setting a breakpoint, stepping and listing local and shared variables and their values. Apart from these, the tool provides three new, user-friendly features built to enable multi-threaded program debugging namely displaying the access/modification history of a shared variable across threads, keeping track of locks held by various treads and enforcing a deterministic interleaving across the various threads. It could be seen that these additional features help in finding concurrency bugs in many real world applications which could not have been otherwise detected using the features provided by GDB.

2. Introduction

2.1 Motivation

Execution of multi-threaded programs involves a lot of non-determinism in terms of order of execution and the state of shared variables. This non-determinism is introduced by the specific interleaving of threads during execution. The non-determinism makes the manual debugging of multi-threaded programs all the more difficult. There is very limited support for debugging multi-threaded programs, built into commercial debuggers like GDB [1]. Currently, GDB provides very simplistic features for multi-threaded debugging [2], namely, setting thread-specific breakpoints and watchpoints, switching from one thread to another and listing thread specific information. Intuitively, the primary reason for the developer's using a debugger to debug a multi-threaded program would be to look at the interleaving that led to a breakpoint and the write history across threads, for the shared variable at the breakpoint. The primary questions popping in the mind of the developer would be, *what* was written to a shared variable, *who* wrote it and *when* was it written. Further, it would be very advantageous if the developer were able to impose a deterministic order of threads to execute, while single stepping or multi-stepping. Also, if there is a data race that leads to a deadlock then additional information about the locks held by each thread may help the programmer in debugging. These three features would be absolutely necessary to make any conclusions about the behavior of the multi-threaded program. They would also make it easy to catch concurrency bugs like atomicity violations, order violations, data races and deadlocks. Further, providing this information to the developer through the interactive interface of a debugger would significantly simplify the arduous process of concurrency bug detection.

2.2 Contributions

Our project contributions can be divided into three broad categories:

2.2.1 Interactive interface

Concurrency bug detection due to buggy interleaving is by itself a complicated task. The absence of a not so user friendly, interactive interface will add to the already existing complications which makes a highly user friendly, interactive interface a mandatory option. In our Pintool we have implemented the interface similar to the one provided by GDB with a similar set of commands.

2.2.2 Basic debugging features

PinDB provides the following set of basic debugging features like any other debugger:

- Setting a breakpoint
- Stepping
- Listing local and shared variables and their values.

2.2.3 Advanced multi-threaded debugging features

In addition to these, the tool provides three new features built in to enable multi-threaded program debugging

- Displaying the access/modification history of a shared variable across threads
- Keeping track of locks held by various treads and displaying them at the breakpoints
- Enforcing a deterministic interleaving across the various threads.

3. Overview of the paper

The remaining part of the paper is divided into 5 sections. Section 4 gives a brief background overview of how Pin works and also the about the types of concurrency bugs addressed. Section 5 talks about the preprocessing requried. Section 6 talks about the design and the implementation of the various features. Section 7 is the evaluation section where we discuss about 3 real world bugs that were detected using PinDB. Section 8 talks about the related work done about concurrency bug detection and about the existing tools. Section 7 is the Conclusion and Future Work and Section 8 is the Appendix and Code section.

4. Background

The debugger is to be implemented using Pin, a binary instrumentation and analysis tool [3]. The debugger itself is a plugin that makes use of Pin and is called a Pintool. The Pintool is given the multi-threaded executable that needs to be debugged, pretty much like a regular debugger, like GDB. The executable is a native executable generated from a C source file. This section gives some basic background information about how Pin works.

4.1 Pin

Pin is a dynamic binary instrumentation and analysis tool. Pin allows the user to write instrumentation code that decides when and where analysis code needs to be inserted in an application executable, and analysis code that actually does the binary analysis. The bundle of the user-written instrumentation and analysis code is called the Pintool. Pin attaches itself and the Pintool to the address space of the executable so that all three components, namely, Pin, the Pintool and the executable share a single process' address space. Pin acts as a Just in Time (JIT) compiler and dynamically creates and compiles instrumentation code by attaching the user-defined instrumentation code to various pieces of the application code (could be routines, traces or individual instructions). The application (along with the jitted instrumentation code) is then executed on Pin's Virtual Machine, with Pin creating a context for the application and performing emulation when necessary. The generated code is cached in a code cache for future use. Figure 1 represents the overall architecture of Pin. Pin can also be used for the instrumentation of multi-threaded executables. In this case, several threads of the Pintool are created, each working on its dedicated user thread from the multi-threaded executable. For this purpose, resources shared across analysis routines need to be protected by locks and Pin offers locking primitives that can be used in the code of the Pintool. Further, Pin provides Thread Local Storage (TLS) API to store thread-specific analysis information.

4.2 Concurrency bug types

PinDB makes possible the detection of 4 types of concurrency bugs namely:

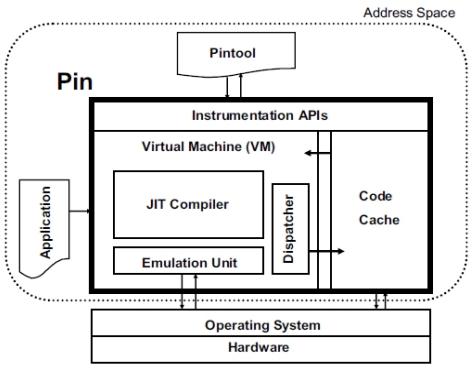

Figure 1. Structure of Pin

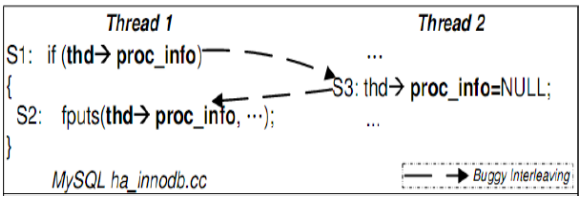

Figure 2. Atomicity Violation Bug from MySQL [12]

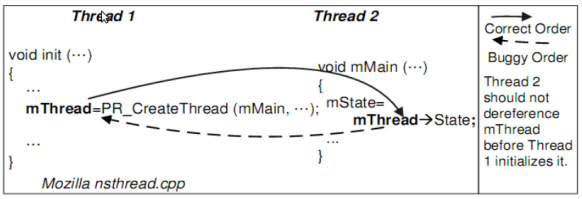

Figure 3. Order Violation Bug from Mozilla[12]

4.2.1 Atomicity violation

Atomicity violation occurs when a sequence of instructions that have to be executed as one atomic unit are not enclosed inside the same critical region. It will lead to a failure/crash if it is accessed by some other thread in the middle of the to-be-atomic unit. Atomicity violations are difficult to manifest as they can occur even in places where data races do not occur. Figure 2 shows an example of an atomicity violation where no data races occurs [11].

4.2.2 Order violation

Order violation occurs when instructions are not executed in the expected order across threads. Detection of order violation bugs is also difficult because the probability of that specific buggy interleaving occuring while testing may be very low. Figure 3 shows an example of a real world order violation bug.

4.2.3 Data race

Data races occur when two conflicting accesses of a shared variable are executed without proper synchronization. It is mainly produced when the programmer fails to protect the shared variable using a common lock.

4.2.4 Deadlocks

Deadlocks occur when two or more threads circularly wait for the release of the same shared resource. Deadlocks in most cases are again caused by some specific buggy interleavings and are difficult to manifest.

5. Preprocessing

Pin provides a rich API to get the various components of an instruction in the executable. These components can be extracted from the current instruction and can be passed for usage in the analysis code. These components include information like instruction address, effective address of a memory reference in the instruction and so on. Since the pintool to be designed is an interactive debugger, referencing instructions through instruction address and shared variables' locations through effective address, makes little, if not any sense, to the developer. For this purpose, the debugger must interact with the developer through instruction line numbers and variable names. To use this information in Pin, line numbers must be mapped to instruction addresses within threads and variable names must be mapped to effective memory addresses. This preprocessing needs to be done during every interaction of the developer with the debugger. We use the libdwarf library to get this mapping of effective memory address to variable name. To get a mapping between line numbers and instruction addresses, we use an API provided by Pin a pintool that dumps the address of every instruction in the executable is executed. With these two maps set up, the debugger can now interact with the user. This pre-processing step is analogous to GDB's populating the symbol table from the given executable.

6. Implementation

6.1 Thread Management

The first step towards debugging multi-threaded target program is to be aware of their existence. The debugger should have the knowledge of the things such as the number of running threads, the states of each thread's execution, and the thread-dependent information maintained by the debugger itself. To achieve this, it is very important to know when a thread is created, and when a thread finishes. The debugger should also be able to identify each thread.

Pin provides convenient interfaces for us to write callbacks for each thread's creation and removal. When a thread of target program is created, this call to thread creation code will be intercepted by Pin. Upon a successful creation, Pin will inject our callback code into the target program right after it, so that our code is executed every time when a new thread is created. The thread removal instrumentation has a similar work-flow. The whole process is described in Figure 4.

In the callback code, we can access the newly-created thread's ID (provided as an argument by Pin) which can be used to identify the thread now and thereafter. This ID is added to the list of current running threads. We then initialize the space for storing the thread-dependent information. This space will be modified and read by the debugger in the future. The space is released in the callback of thread removal.

6.2 Instrumentation Granularity

Pin provides instrumentation for several levels of granularity, typically program level, image level, routine level, basic block level and instruction level.

For each level of granularity, the instrumentation process is pretty the same. When the execution comes to the beginning of an unit of this level, Pin checks whether the instrumentation has been done for it. If not, Pin will call a user-defined **Instrumentation Function** to inject another user-defined **Analysis Function** to a proper position of the currently encountered unit. Where (BEFORE, AFTER, etc) and which analysis function to inject is specified by the instrumentation function. The instrumentation process is shown in Figure 5.

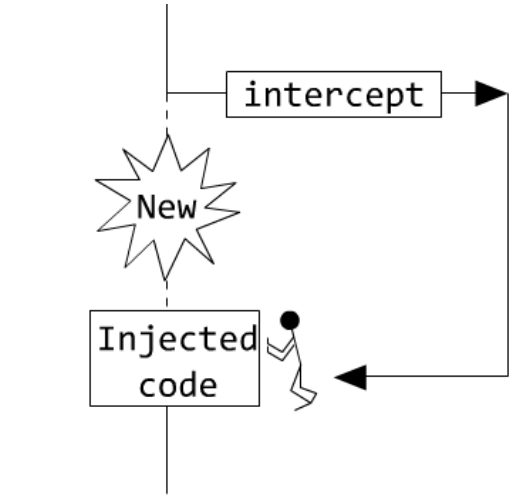

Figure 4. Thread Creation

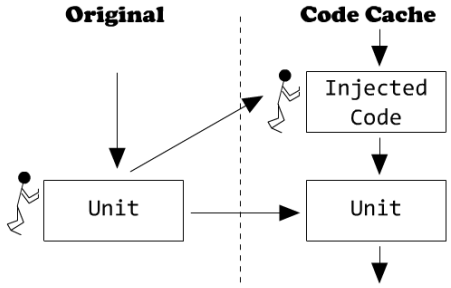

Figure 5. Instrumentation

Pin does not actually check whether or not a unit has been instrumented every time, since this is very expensive. In fact, Pin put instrumented units into a place called code cache. Normally, the execution is in the code cache. Only when the execution transfers to a unit who is not in the code cache, Pins call the instrumentation function do the code injection, and move them into code cache.

Unfortunately, Pin does not directly provide the granularity level of source code line, which is our major concern. Usually, each line of the source code is consisting of several instructions. However, instrumenting every instruction would lead to a huge overhead. Our basic idea is to instrument only the first instruction of each line in the source code. Injecting code BEFORE such instructions has no difference from injecting code BEFORE corresponding lines in the source code. Pin provides API to help us identify such instructions in writing instrumentation functions.

We are aware that this method only works for the situation that we want to inject codes before a line. Though the injection-after case can also be done in a similar way, we developed a work-around to avoid such overhead, since we do not rely on injection-after in most cases. This will be discussed in the next section.

6.3 Access History

The debugger is designed to be able to log the access history of user-specified shared objects, including shared variable and locks. This feature requires instrumentation to related instructions, including those of PThread *lock()/unlock()* and memory read/write.

For memory accesses, we call Pin API *INS_IsMemoryRead()/INS_IsMemory* in the instrumentation function to check whether memory accesses are involved in the current instruction. If they does, we will then inject corresponding analysis functions to this instruction. In the analysis function, the accessed memory addresses will be analyzed to test whether they are among the variables the user cares about. We use a global data structure to record the memory accesses that pass the test.

For PThread locks, we need to refer to the source code. In the instrumentation function, we first get the line in the source code that the current instruction belongs to. We then check whether there are PThread *lock()* / *unlock()* calls in this line. For those whose corresponding lien contains PThread locks, we inject the analysis function. In the analysis function, we then find from the source code the name of the lock variable on which the instruction operates. We also maintain a global data structure to record PThread class in which the user are interested.

The work-flow of the instrumentation function works like below:

Algorithm 1 Instrumentation Function

```
ins ← current instruction address
line ← Get_Line( ins )
if Is_First_Instruction ( ins, line ) then
    Inject ( ins, Line_Analysis_Function )
end if
if Is_Memory_Access ( ins ) then
    Inject ( ins, Memory_Analysis_Function )
end if
if Is_PThread_lock ( ins, line ) then
    Inject ( ins, Lock_Analysis_Function )
end if
```

6.4 Breakpoints and Force Interleaving

Traditional breakpoints are marks on lines in the source code. When one of these lines is reached, the target program will be stopped and the control will be transferred to an interactive command line interface (we are going to call it "command mode" in the rest of this paper). On the other hand, forcing a particular interleaving sequence doesn't need to transfer the control to command mode on reaching every interleaving point. Instead, the transfer of control only happens when the last interleaving point is reached.

However, interleaving points and breakpoints are fundamentally the same in a sense that when they are reached, the program/thread will pause for a while and do something. So we designed a framework to make them exactly the same.

6.4.1 Token Mechanism

The debugger maintains a list of tokens. At the beginning, the debugger holds all the tokens. Each thread can **wait** for or **request** for tokens.

If a thread **requests** for a token from the debugger, the debugger will issue one of the tokens it currently holds, and passes it to the requesting thread. The debugger then doesn't hold that token unless **the thread returns it back**, and the thread is said to hold that token before it returns the token back.

The debugger can also ask a thread to **wait** for a token. If a thread **waits** for a token, and the debugger does not currently hold that token, it stops and won't resume until the debugger get reclaim the hold of that particular token. However, if the debugger currently hold that token, the thread will continue as if nothing has happened. In a $wait$ situation, the debugger doesn't need to pass the token to the $waiting$ thread.

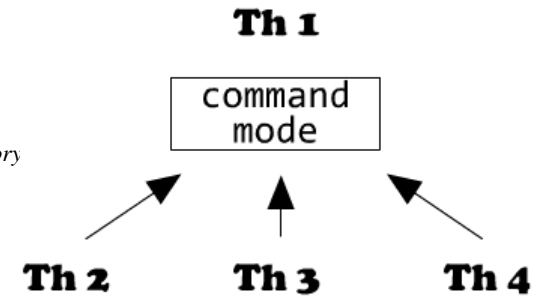

Figure 6. Dependencies of Normal Breakpoints

A thread can hold multiple tokens at the same time, and may also be waiting for multiple tokens at the same time.

6.4.2 Normal Breakpoints

When a normal breakpoint in a thread (say, thread A) is reached, we need to ask all the rest threads to stop and transfer thread A to command mode. This is can be done as below following the token mechanism.

First, thread A requests a token from the debugger. Then the rest of threads will be asked to wait for this particular token. When thread A get the token, it will then transfer the control to command mode. On exit of the command mode, thread A will return the token it asked for this breakpoint, and the other threads will resume. The algorithm is described as in 2.

Algorithm 2 Normal Breakpoint

```
Ω ← the set of threads
token ← A.Request()
for X ∈ Ω - {A} do
    X.Wait(token)
end for
A.CommandMode()
A.Release(token)
```

The token mechanism works as a simple lock here, which describes a dependency shown as in Figure 6.

6.5 Interleaving Breakpoints

When the user specifies a sequence of interleaving points with an order, all the threads start, but the interleaving points have to be executed in an order consistent with what the user specified.

This can be done in a simple way under the token mechanism. Suppose the interleaving breakpoints are in such an order:

$$b_1, b_2, b_3, \cdots, b_n$$

Before start running all the threads, every thread involves in this interleaving will request a token from the debugger **for every interleaving point of it**. Only when such interleaving point has been executed, the thread will release the corresponding token. However, on reaching breakpoint b_i, before actually execute it, the debugger will ask b_i to wait for the token hold by another thread due to b_{i-1}. There are only two exceptions. b_1 has no preceding interleaving points, so it does not wait for any one to be executed. However, it needs to release its token after being executed. b_n is just

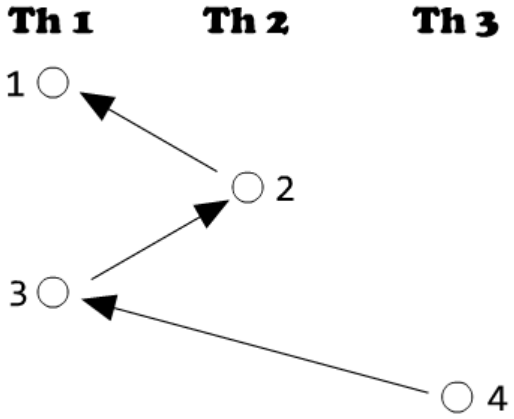

Figure 7. Dependencies of Interleaving Breakpoints

a normal breakpoint which will transfer to command mode after being executed. However, it needs to wait for b_{n-1}'s token.

The work-flow before starting the force interleaving works as below in 3. And the work-flow on reaching an interleaving breakpoints b_i is described in 4.

Algorithm 3 Before Start Force Interleaving

$n \leftarrow$ number of interleaving points
for i $\in 1 \cdots n-1$ **do**
  b_i.holdToken = $\leftarrow$ b_i.request()
  b_{i+1}.waitToken $\leftarrow$ b_i.holdToken
end for

Algorithm 4 Before Start Force Interleaving

if $i > 1$ **then**
  b_i.thread.Wait(b_i.waitToken)
end if
b_i.thread.NextLine()
if $i < n$ **then**
  b_i.thread.Release(b_i.holdToken)
end if

There is a trick involved. We do not want to inject code **after** each line, but only do injection **before** each line. Since we need to release the token after one interleaving point being executed, we must find a place to do so. The trick is to do this before the next line is executed. The analysis function for each line would first check whether the previously executed line is an interleaving point. If it is, then do the the token release.

The algorithm described above construct such a dependency Figure 7, and can also be modified to adapt to more complicated dependencies Figure 8.

6.6 Print Variables

Pin itself provides API to access the value of arbitrary memory addresses, which favors us a lot in printing the value of variables. However, it is very difficult to map variable names to memory addresses. So that we adopt libdwarf to do such work.

Libdwarf provides APIs to extract symbol information from source code. However, the information is organized in a very complicated way so that it might need huge coding effort to parse it. Further more, for those information we only care about the offsets to register BP and maybe SP sometimes.

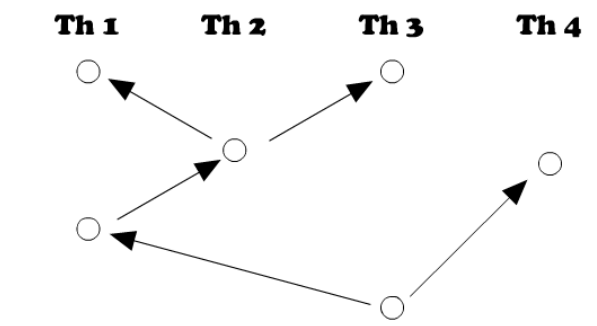

Figure 8. Dependencies of Complicated Interleaving

So we use an alternative way to exploit libdwarf. There is a tool in libdwarf package that can dump the symbol information into a more developer-friendly formatted file. The formatted file will be further processed to a simplified text file. The debugger will fork a python script when starting running. Every time when it need symbol information, it uses socket to communicate with the python script. On receiving such request, the python script parse the text file and return the information back to the debugger through that socket.

With these information, we can calculate the memory address of a variable (either global or local) by adding the offset and certain registers (such as EBP). The member variable of classes is under one or more layer of abstraction, so it is more complicated to deal with. We should first get the address of the class instance it belongs to, and then convert it to the member variable's address by adding the offset.

There is a trick concerning the member variables. Inside a member method it seems hard to get the address of the class instance. However, the pointer to this instance will be passed into the member method as the hidden parameter "this".

7. Evaluation

7.1 Apache Atomicity Violation Bug

The Apache Atomicity Violation bug is a real world concurrency bug that was found in Apache httpd - 2, Apache's web server version 2 [Bug Page]. The buggy code is present in the file mod_mem_cache.c. The buggy code is shown below:

```
static apr_status_t decrement_refcount(){
  ...
  apr_atomic_dec(&obj->refcount);
  if(!obj->refcount){
    if(obj->cleanup){
      cleanup_cache_object(obj);
  }
  ...
}
```

In the code, the function *atomic_dec* is not really atomic in that there are no locks surrounding the function call or withing the body of the function. This is the function that primarily causes the bug. When the given source code is concurrently executed by two threads, the following interleaving may be observed.

THREAD 1

```
apr_atomic_dec(&obj->refcount);
```

THREAD 2

```
apr_atomic_dec(&obj->refcount);
```

At this point the variable *refcount* becomes 0 and assume thread 2 continues with its execution, finds that *refcount* has become 0 and thus de-allocates the associated *obj*. This causes a dangling pointer pointing to unallocated storage in the case of thread 1. Now, thread 1 resumes and dereferences *obj*. If the deallocated storage is not part of the address space, this causes a Segment Violation and results in a program crash. The interleaving is shown in the code below:

THREAD 1

```
if(!obj->refcount){
    if(obj->cleanup){
        cleanup_cache_object(obj);
  }
```

THREAD 2

```
if(!obj->refcount)

~~~~~~~~~~~~~CRASH~~~~~~~~~~~~~
```

This bug would not have existed had, the decrement of *refcount* and the subsequent dereference and cleanup were protected by a lock. This was the logical atomicity violation that caused the bug.

Given just the point of crash, debugging this using GDB would require setting watchpoints on every shared variable and stopping current thread and switching to another thread at every watchpoint and trying out the same for various steps in each thread.

The bug was extracted to create a simple multi-threaded C program that reproduces the bug in code. The dangling pointer problem was converted to a NULL pointer dereference by assigning a NULL pointer to the de-allocated pointer. Knowing the point of crash, a breakpoint was set at the crash point. Now at the breakpoint, the access history for the shared variable *refcount* was listed. The history looked something like:

THREAD 1	THREAD 2
W: refcount = 2	
W: refcount−−	
R: refcount	
	W: refcount−−
	R: refcount
	W: obj = NULL

From, the access history, a suspicious interleaving was tested. The interleaving involved assigning the value NULL to *refcount* before it was read. The forced interleaving looked like:

THREAD 1	THREAD 2
W: refcount = 2	
W: refcount−−	
	W: refcount−−
	R: refcount
	W: obj = NULL
R: refcount	

Once this interleaving was forced, the program crashed. Once the decrement and the subsequent dereference and cleanup were protected by a single lock and when the above mentioned interleaving was given, the interleaving failed, pointing to the fact that shared variable's use was protected by locks and an atomicity violation could not have occurred.

7.2 Transmission Order Violation Bug

The Transmission Order Violation Bug is a real world concurrency bug that was found in Transmission, an open-source BitTorrent Client [bug link]. The buggy code is present in the files session.c and bandwidth.c. The buggy code is shown below:

```
/* session.c */
271:h->peerMgr = tr_peerMgrNew( h );

   ......

281:h->bandwidth = tr_bandwidthNew( h, NULL );


/* bandwidth.c */
...

251:assert( tr_isBandwidth( b ) );

...
```

In this code, *tr_peerMgrNew* creates a timer that will fire off every 500 ms and will assign a value to the *bandwidth* object. However, the memory for the *bandwidth* object is allocated only later by the function *tr_bandwidthNew*. Before assigning a bandwidth value, the assert in bandwidth.c checks if storage for the *bandwidth* object has been allocated. If it has not been allocated, the assertion fails and the program crashes. The manifestation of the bug really depends upon the order in which the allocation and assignment happen. If the allocation does not happen before the assignment, an order violation occurs, causing the bug to result in a crash. The buggy interleaving is shown below:

THREAD 1

```
h->peerMgr = tr_peerMgrNew( h );
```

THREAD 2

```
assert( tr_isBandwidth( b ) );
```

THREAD 3

```
h->bandwidth = tr_bandwidthNew( h, NULL );
```

Over here the *assert* gave the point of crash and a breakpoint was set at the *assert*. The access history for the *bandwidth* member of *h* looked like:

THREAD 1	THREAD 2
W: h = NULL	
allocate memory for h	
W: bandwidth = 10	
	R: bandwidth

We forced the interleaving such that bandwidth was checked for allocation before if was indeed allocated storage. The forced interleaving looked like:

THREAD 1	THREAD 2
W: h = NULL	
	R: bandwidth

```
allocate memory for h
W: bandwidth = 10
```

This resulted in an assertion violation. This showed that such an interleaving that violated the general order of allocation-assignment was indeed possible thus exposing the order violation bug.

7.3 Mozilla Atomicity Violation Bug

Mozilla Atomicity Violation Bug is a very interesting javascript loading bug extracted from the Mozilla code. The buggy code is present in the file nsXULDocument.cpp. One thread loads javascript, and sets gScript to be the loading protocol, so that after loading finishes someone would observe this and it can use this handler to continue compilation and execution of the script. However, the programmer decided to permit brutal shared loading. The other thread overwrites the gScript and begins loading the script too. When the first thread finishes loading, it uses a wrong gScript to do compile/execute and then sets the gScript to be NULL. Now when the first thread tries to use the gScript to compile a NULL pointer dereference occurs and the program crashes.

The code that causes atomicity violation is shown below:

```
void* work(void* param){
        lock(l);
        gScript=aspt;
        unlock(l);
        ...
        lock(l)
        gScript->compile();
        gScript=NULL;
        unlock(l);
}
```

In the code, the function *work_dec* is not really atomic in that there are no locks surrounding the function call or withing the body of the function. This is the function that primarily causes the bug. When the given source code is concurrently executed by two threads, the following interleaving may be observed.

```
THREAD 1
--------

        gScript=aspt;

THREAD 2
--------

        gScript=aspt;
```

At this point the variable *gScript* in both threads refer to aspt and assume thread 2 continues with its execution, and completes the execution of the instruction which sets gScript to NULL. Now, thread 1 resumes assuming that gScript is still aspt and and dereferences *gScript* without checking. A NULL pointer dereference occurs and the program crashes. The buggy interleaving is shown in the code below:

```
THREAD 1
--------

        lock(l);
        gScript=aspt;
        unlock(l);

THREAD 2
--------

        lock(l)
        gScript->compile();
        gScript=NULL;
        unlock(l);

THREAD 1
--------

        lock(l)
        gScript->compile();

~~~~~~~~~~~~CRASH~~~~~~~~~~~~
```

This bug would not have existed if only one loading was allowed at a time and if the others were added to the wait list which would not have allowed overwrite of the gScript variable.

Given just the point of crash, debugging this using GDB would require setting watchpoints on every shared variable and stopping current thread and switching to another thread at every watchpoint and trying out the same for various steps in each thread.

The original source code is implemented in a pseudo-multi-thread design, using task-queue. Since the original Mozilla GUI is too slow to start in Pin, the original code was extracted to a simple C++ code named the mozilla_ex.cpp and mozilla_ex.hpp. Knowing the point of crash, a breakpoint was set at the crash point. Now at the breakpoint, the access history for the shared variable *gScript* was listed. The history looked something like:

```
THREAD 1                THREAD 2
--------                --------

W: gScript=aspt
R: gScript

                        R: gScript
                        W: gScript=NULL
```

From, the access history, a suspicious interleaving was tested. The interleaving involved assigning the value NULL to *gScript* before it was read. The forced interleaving looked like:

```
THREAD 1                THREAD 2
--------                --------

W: gScript=aspt

                        R: gScript
                        W: gScript=NULL

R: gScript
```

Once this interleaving was forced, the program crashed. Then we tried protecting the initialization and the call of the compile function and assignment to NULL by a single lock and then giving the above interleaving. As expected no crash occurred indicating that it was indeed the atomicity violation that had caused the crash before.

8. Related Work

GDB [1] is the only commercially available debugger that has support for debugging multi-threaded applications. GDB has very limited features for multi-threaded debugging, namely, thread-switching, setting breakpoints and watchpoints. It does not provide functionality to view write and interleaving history. [10] describes an addition to GDB, to implement the *non-stop* semantics. There is a lot of literature describing the various features of the Pin tool [3]. A very significant work is [5], which describes the technical details of how support for multi-threaded programs was built into Pin. [9] describes Pinplay, a Pintool that is used for deterministic replay of parallel programs. Finally [7] and [6] build bug detection techniques into Pin, to detect Concurrency bugs. They describe many general details about concurrency bugs and they describe implementing basic features for bug detection using Pin.

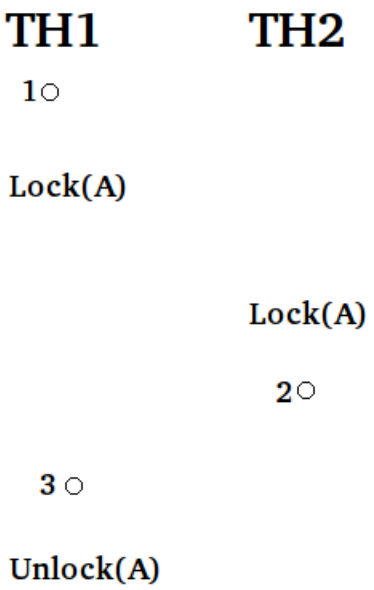

Figure 9. Possible Deadlock due to Force Interleaving

9. Conclusion

Working on this project is indeed a pleasant experience. Studying the classification and pattern of concurrency bugs gave us a more deep understanding of how do they occur and how complicated they might be. We are even more impressed by the complexity of dealing with concurrency when we start to consider the possible deadlock problem in force interleaving.

Here is an example of such deadlock. The interleaving points and their order are shown in Figure 9. If Thread1 get the lock before Thread2, then Thread2 will never be able to reach interleaving point 2. Since interleaving point 3 depends on interleaving point 2, Thread1 will not be able to execute interleaving point 3 at all. This leads to a deadlock. However, this interleaving is not impossible.

One solution to this problem is to add one more interleaving point. In order to do this, the user should first analyze the lock history of that dead-locked interleaving. Though this requires user's extra effort, it is indeed a work around. Automatically detection of potential interleaving deadlocks is possible, but it can be really expensive for this debugger (such as what adopted in Dimunix), and the user might need to run this target program more than once to collect necessary information.

There are also other small problems we tried and managed to solve in this project. Some of them are related to Pin. Pin is really a fancy instrumentation tool, and we have learned a lot of how to use it through working on this project. By searching for and finding the solution to our Pin-related problems, we become more and more familiar with this tool, as well as understanding the way Pin works.

References

[1] http://www.gnu.org/software/gdb/

[2] http://www.delorie.com/gnu/docs/gdb/gdb_25.html

[3] http://www.pintool.org/

[4] http://x10.codehaus.org/

[5] Chi-keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, Reddi Kim Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI 05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pages 190–200, ACM Press.

[6] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, Thomas W. Reps: ConSeq: detecting concurrency bugs through sequential errors. ASPLOS 2011: 251-264.

[7] Wei Zhang, Chong Sun, Shan Lu: ConMem: detecting severe concurrency bugs through an effect-oriented approach. ASPLOS 2010: 179-192.

[8] Kim M. Hazelwood, Greg Lueck, Robert Cohn: Scalable support for multithreaded applications on dynamic binary instrumentation systems. ISMM 2009: 20-29.

[9] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, James Cownie: PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. CGO 2010: 2-11.

[10] Non-Stop Multi-Threaded Debugging in GDB, Nathan Sidwell, Vladimir Prus, Pedro Alves, Sandra Loosemore, and Jim Blandy, 2008 GCC Developers' Summit.

[11] ColorSafe:Architectural Support for Debugging and Dynamically avoiding Multi-Variate Atomicity Violations, Brandon Lucia, Luis Ceze, Karin Strauss, ISCA' 10, June 19-23, 2010.

[12] Learning from Mistakes: A comprehensive study on real world concurrency bug characteristics, Shan Lu, Soyeun Park, Eunsoo Seo, Yuanyuan Zhou, ASPLOS '08.

A. PinDB Commands

p	print variable value
ph	print history of variable
n	next line
lines	show current source code line
his	start recording access history of one object
ll	list lock logs
sw	switch thread
b	set one-time breakpoint
bt	set infinite-time breakpoint
inter	start interleaving