Reinforcement Learning Aided Concurrency Bug Detector

July 27, 2020

Abstract

Nowadays multi-thread programs have become pervasive because of the development of multicore computing. However, writing correct concurrent programs is nontrivial due to the nondeterministic behavior of concurrent program. In other words, if a multi-thread program is executed several times, the behavior of the program may differ based on different thread interleaving patterns. Existing approaches managed to reveal concurrency bugs to some extent, but often they lead to false negatives. Our approach proposed utilizing reinforcement learning to assist the process of detecting concurrency bugs. It is expected that our research would be the first to implement reinforcement learning to detecting concurrency bugs and would reduce the number of false negatives.

1 Introduction

Concurrency bugs are bugs that only manifest in concurrent programs. Finding concurrency bugs in multi-thread programs are nontrivial because of the non-deterministic nature of them. The manifest of concurrency bugs depends on specific thread interleaves.

2 Existing Proposal

PCT and Maple are two of the state of the arts in the area of detecting concurrency bugs. PCT provides a randomized scheduler for finding concurrency bugs. Specifically, it randomly assigns different priority values to each thread and randomly inserts priority change points into the program. If the target program is with n threads and k steps, PCT guarantees the probability of finding a concurrency bug of depth d with probability at least

$$1/nk^{d-1}$$

. But as the depth of bug increases, it becomes difficult for PCT to detect the bug because the probability will decrease exponentially.

Meanwhile, Maple can trigger bugs by allowing the scheduler actively exposing a set of predefined concurrency bug idioms. However, besides this set of predefined idioms there are concurrency bugs of other idioms that Maple cannot find, which lead to the consequence of false negatives.

2.1 Re-implement Maple

In order to better understand the methodology utilized by Maple, we tried to re-implement the idea of it in the following section. Specifically, for each idiom mentioned in Maple, we made some samples and used Intel Pin to implement it. In brief, two threads and at most two shared variables are involved in all six idioms.

2.1.1 Overview of Maple

Maple provides a coverage-driven testing methodology for concurrent programs. Based on value-independence hypothesis and small scope hypothesis, it provides 6 idioms commonly seen in concurrency bugs. An idiom is a pattern that indicates memory dependencies between threads. Two instructions are dependency if they access the same memory address. An iRoot is an instance of an idiom. Like in the original paper, the implementation is mainly divided into two parts: profiler and actives scheduler. During profiling phase, instrumentation collects useful information, such as accessed memory address and lockset of some instruction, to predict possible buggy thread interleaves. If according to the information collected during profiling phase, a thread interleave is possible to trigger a concurrency bug, then profiler will produce an execution order that is able to expose this specific iRoot and send the execution order to actives scheduler. We implement an active scheduler that can strictly schedule threads as implied by execution order.

2.1.2 Definition of Suspicious

Suppose we have a target concurrent program; we want to find concurrency bugs in that program. Imagine there is a set of interleaves that are possible for this program. After profiling phase of maple, by collecting essential information and leverage the information to do some judgement and extract the suspicious interleaves. Specifically, suspicious means if those interleaves are executed, it is likely to trigger a concurrency bug. We feed those suspicious to next phase, testing phase. By actually executing the program according to interleaves, we will be able to find which are actually bug-triggering and which are bug-free. As for deciding whether an interleave is suspicious or not, it is easy to say if at least one of the instructions is a write instruction among all instructions accessing the same memory address between different threads, it is suspicious. But it does not mean if all instructions are read it is impossible to trigger concurrency bugs. As indicated in the paper of Maple, authors provided an example of instance of idiom 4 in real world programs. Four instructions are involved and all of four are read instructions. Therefore, in our implementation, if instructions are read instructions, as long as they are accessing the same memory address, we identify them as suspicious and create corresponding execution orders that can reveal them.

2.1.3 Effect of Instruction Position

Position of instruction affects how to produce execution orders. As for idiom 1, there are 9 different scenarios based on instruction position: at the beginning, in the middle, or at the end of their residing thread.

2.1.4 idiom1

Idiom 1 describes a scenario that certain execution order is assumed by programmer but no mutual exclusion is utilized in the code. Therefore, if execution order is different from what the programmer expected, a concurrency bug will manifest. In other words, it is an order violation bug.

```
Thread 1:
void init()
{
 int age = std.age;
}
```

```
Thread 2:
int main()
{
 printf("%d", age);
}
```

In this example, correct behavior of this program depends on the initialization of int type variable age in thread 1 executed before the usage of variable age in thread 2. Therefore, if usage of variable age is executed first, an order violation bug happens.

2.1.5 idiom2

This idiom could be considered an atomicity violation. If the correct behavior of a program depends on two or more instructions executed together, or no interruption between these instructions, then these instructions are considered to be atomicity.

```
Thread 1:
void func()
{
 if(std1.age)
 {
  printf("%d", std1.age);
 }
}
```

```
Thread 2:
```

```
int main()
{
 std1.age = NULL;
}
```

The if statement and usage of std1.age in the if statement should be considered atomicity. In other words, there should not be any thread interleaving happens when execution if in between these two executions. If std1.age is assigned to be NULL in thread 2 right after the if statement in thread 1, an atomicity violation bug manifests.

2.1.6 idiom3

2.1.7 idiom4

2.1.8 idiom5

Idiom 5 in Maple describes a typical deadlock concurrency bug. Deadlock happens when a thread (Thread 1) is holding a lock (L1) and waiting for another one (L2); meanwhile the other thread (Thread 2) that holds lock L2 is waiting for the release of lock L1.

```
Thread 1:
void foo()
{
 pthread_mutex_lock(&mutex1);
 pthread_mutex_lock(&mutex2);
 ...
}

void bar()
{
 pthread_mutex_lock(&mutex2);
 pthread_mutex_lock(&mutex1);
 ...
}
```

In this example, the following thing could happen: Thread 1 is holding lock mutex1 and requiring lock mutex2, meanwhile Thread 2 is holding lock mutex2 and requiring lock mutex1. This will lead to an idiom 5 concurrency bug.

2.1.9 idiom6

2.2 Practical Binary Analysis Exercise

In this part, we utilize Intel Pin API to create our own pintool, in order to instrument an executable of a multi-thread program, so that we could tell the following information about each instruction: i) Which thread the instruction belongs to. ii) The type of memory access (read or write). iii) The specific memory address that is been accessed.

Here is the implementation environment. OS: macOS Cataline 10.15.6. Intel Pin: version 3.15.

Different from instrumenting a single-thread program, instrumenting a multi-thread program requires the tools to be thread safe. Therefore, for functions that are thread-unsafe, we have to use lock and unlock to make them critical section to avoid concurrency bugs, for example, order violation and deadlock. Unfortunately, the standard library of Unix-like OS for multi-thread program such as pthread cannot be used in a Pintool. Therefore, we had to use library provided by Pin, such as `PIN_GetLock()` and `PIN_ReleaseLock()` to instrument multi-thread programs written in C programming language.

Specifically, as for analysis routines, we implemented 4 functions, shown as below:

```
VOID ThreadStart(THREADID threadid, CONTEXT
    *ctxt, INT32 flags, VOID *v)
{
 PIN_GetLock(&pinLock, threadid+1);
 fprintf(out, "thread begin %d\n", threadid)
     ;
 fflush(out);
 PIN_ReleaseLock(&pinLock);
}

VOID ThreadFini(THREADID threadid, const
    CONTEXT *ctxt, INT32 code, VOID *v)
{
 PIN_GetLock(&pinLock, threadid+1);
 fprintf(out, "thread end %d, code %d\n",
     threadid, code);
 fflush(out);
 PIN_ReleaseLock(&pinLock);
}

VOID MemoryWrite(VOID * addr, THREADID
    threadid)
{
 PIN_GetLock(&pinLock, threadid+1);
 fprintf(out, "thread id %d, memory write,
     memory address %p\n",
  threadid, addr);
 fflush(out);
 PIN_ReleaseLock(&pinLock);
}
```

```
VOID MemoryRead(VOID * addr, THREADID
    threadid)
{
 PIN_GetLock(&pinLock, threadid+1);
 fprintf(out, "thread id %d, memory read,
     memory address %p\n",
  threadid, addr);
 fflush(out);
 PIN_ReleaseLock(&pinLock);
}
```

The first routine is executed every time a new thread is created. The second routine is executed every time a thread is destroyed. The third routine is executed whenever an instruction is a memory-write instruction, meanwhile, the last routine is executed whenever an instruction is a memory-read one.

As for instrumentation routines, the most important one is:

```
VOID Instruction(INS ins, VOID *v)
{
 if (INS_IsMemoryWrite(ins))
 {
  INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR
       (MemoryWrite),
   IARG_MEMORYWRITE_EA, IARG_THREAD_ID,
        IARG_END);
 }
 if (INS_IsMemoryRead(ins))
 {
  INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR
       (MemoryRead),
   IARG_MEMORYREAD_EA, IARG_THREAD_ID,
        IARG_END);
 }
}
```

In Pintools, instrumentation routins will be executed only once. In this case, by executing this instrumentation routine, if an instruction is a memory write one, analysis routine MemoryWrite will be called. If an instruction is a memory read one, analysis routine MemoryRead will be called.

By executing this customized Pintool on a sample concurrency c programming code, we obtain a file that tells when a thread is created and destroyed, the type of instruction, which thread the instruction belongs to, and the specific memory address the instruction is accessing.