Vector-Clock based Race Detector for C++ Programs

Shreyas Hirday,
Advisor: Professor Santosh Nagarakatte
Rutgers University, Architecture and Programming Languages Lab (APL)

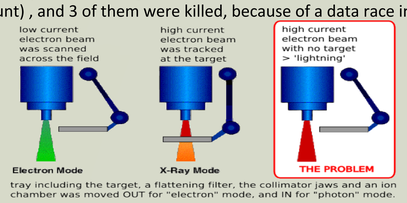
RUTGERS
Aresty Research Center for Undergraduates

Abstract

Modern computers use multi-core processors and a programmer must write multithreaded code to get optimal performance. However, writing correct multi-threaded code is difficult because unintended interactions between two threads can cause errors. The occurrence of data races, in which two threads simultaneously read and write the same shared memory location, is one such error. By detecting data races, programmers will have an easier time writing correct code. In this project, we built a precise, vector clock-based race detector. We tested this tool on a few, simple C/C++ programs that were either race-free or racy, and we were able to detect bugs in the programs that were racy.

Background

- Vector clocks are mechanisms used to determine if there is a partial ordering between events in a distributed system
- A happens-before relationship occurs if two events happen before each other, and the result shows that
- Data races occur when there is no happens-before relationship between two events that access the same memory location, and at least one is a write operation
- From 1985-1987, at least 6 patients were given massive overdoes of radiation (100 times the original amount) , and 3 of them were killed, because of a data race in the Therac-25 software.



tray including the target, a flattening filter, the collimator jaws and an ion chamber was moved OUT for "electron" mode, and IN for "photon" mode.

Thread 1	Thread 2
//reads that x = 0	//reads that x = 0
x+=4;	**x+=5;**
//x is now 4	**//x is now 5**

- Locks, a method of synchronization, ensure that there's no data race by ordering the threads.

Thread 1	Thread 2
lock(m);	lock(m);
//reads that x = 0	//reads that x = 4
x+=4;	x+=5;
//x is now 4	//x is now 9
unlock(m);	unlock(m):

- Thus, it is very important to capture data races if multithreaded code is not written properly

Objectives

Build a precise data race detector in C/C++ programs using a vector clock-based algorithm. Vector clocks enable us to identify a happens-before relationship between events occurring on two different threads.

Methods

- In order to develop this race detector, we used Pin, a dynamic binary instrumentation framework provided by Intel and the race detector was written in C++
- Our data race detector changes the straight line code sequence of the executable it takes as input to gather data
- We also used a Vector-Clock based algorithm to keep track of the ordering between threads and their memory accesses to determine if there was a happens-before relationship so we could detect if there was a data race.
- Here is an example:

Main Thread	Thread 1 (*T1*)	Thread 2 (*T2*)	Read X , *R*	Write X, *W*
1) [0,0,0] ↓(create T1)	[0,0,0]	[0,0,0]	n/a	n/a
2) [1,0,0] ↓ (create T2)	[0,1,0]	[0,0,0]	n/a	n/a
3) [2,0,0]	[0,1,0] ↓(reads to X)	[1,0,1]	n/a	n/a
4) [2,0,0]	[0,1,0] ↓(writes to x)	[1,0,1]	[0,1,0]	n/a
5) [2,0,0]	[0,1,0]	[1,0,1] ↓(does a read, but there is a RACE because every element of *T2* is not greater than every element of *W*)	[0,1,0]	[0,1,0]

- If *I* is the thread ID, then as long as every element of thread *i* 's vector clock is greater than or equal W, there is no data race
- A snippet of our PinTool that instruments the executable:

```
    RTN wRtn = RTN_FindByName(img,"_deposit");
    if(RTN_Valid(wRtn)){
        RTN_Open(wRtn);
        for(INS ins1 = RTN_InsHead(wRtn);INS_Valid(ins1); ins1 = INS_Next(ins1)\
){
            UINT32 mems = INS_MemoryOperandCount(ins1);
            for (UINT32 mem = 0; mem < mems; mem++){
                if(INS_MemoryOperandIsRead(ins1,mem)){
                    INS_InsertPredicatedCall(ins1,IPOINT_BEFORE,(AFUNPTR)docoun\
tR,IARG_MEMORYOP_EA,mem,IARG_THREAD_ID,IARG_END);
                }
                if(INS_MemoryOperandIsWritten(ins1,mem)){
                    INS_InsertPredicatedCall(ins1,IPOINT_BEFORE,(AFUNPTR)docoun\
tW,IARG_MEMORYOP_EA,mem,IARG_THREAD_ID,IARG_END);
                }
            }
        }
        RTN_Close(wRtn);
    }

    RTN dRtn = RTN_FindByName(img,"_withdraw");
    if(RTN_Valid(dRtn)){
        RTN_Open(dRtn);
        for(INS ins1 = RTN_InsHead(dRtn);INS_Valid(ins1); ins1 = INS_Next(ins1)\
){
            UINT32 mems = INS_MemoryOperandCount(ins1);
            for (UINT32 mem = 0; mem < mems; mem++){
                if(INS_MemoryOperandIsRead(ins1,mem)){
                    INS_InsertPredicatedCall(ins1,IPOINT_BEFORE,(AFUNPTR)docoun\
tR,IARG_MEMORYOP_EA,mem,IARG_THREAD_ID,IARG_END);
                }
                if(INS_MemoryOperandIsWritten(ins1,mem)){
                    INS_InsertPredicatedCall(ins1,IPOINT_BEFORE,(AFUNPTR)docoun\
tW,IARG_MEMORYOP_EA,mem,IARG_THREAD_ID,IARG_END);
                }
            }
        }
        RTN_Close(dRtn);
    }
```

- In order to test if our tool worked, we created a mock bank account program where on thread deposits 100 dollars and another thread withdraws a 100 dollars from a balance of 0
- We modified the code to create different scenarios, those that definitely had races, those that didn't, and those that could have had races based on the execution

Results/Discussion

```
void *deposit(void *arg){
  ShreyasLock(&lock);
  balance+=100;
  printf("after deposit: %f\n",balance);
  ShreyasUnlock(&lock);
  balance+=5;
  return NULL;

}
```

```
void *withdraw(void *arg){
  ShreyasLock(&lock);
  balance-=100;
  printf("after withdraw: %f\n",balance);
  ShreyasUnlock(&lock);
  return NULL;

}
```

On the **left:** a deposit function that properly implements locks, but modifies balance after ShreyasUnlock() is called **right:** a withdraw function that properly implements locks

- The scenario above is tricky, because even though both functions properly implement locks, "balance" is being modified outside of the locks in *deposit* as well.
- In this case, there are two possible executions and only one of them is a data race.
- If withdraw is called first, then there is a happens-before inducing edge and there is no race. However, if deposit is called first, there is no edge and therefore there is a race.

```
nbp-154-90:ManualExamples shreyashirday$ ../../../pin -t obj-intel64/pinatrace.d
ylib -- ~/Desktop/Other/bank
acquired
after withdraw: -100.000000
acquired
after deposit: 0.000000
done
nbp-154-90:ManualExamples shreyashirday$ cat pinatrace.out
Race: NO
done
```

```
nbp-154-90:ManualExamples shreyashirday$ ../../../pin -t obj-intel64/pinatrace.d
ylib -- ~/Desktop/Other/bank
acquired
after deposit: 100.000000
acquired
after withdraw: 5.000000
done
nbp-154-90:ManualExamples shreyashirday$  cat pinatrace.out
Race: YES, at: 0x18590c058
done
```

On the **left:** One of the the two possible executions, in which withdraw is called first and there is a happens-before edge so the race detector properly recognizes that there is no race **right:** The other possible execution in which deposit is called fitst and there IS a race so the race detector properly recognizes that there is a race and gives the memory address of the variable where it occurs

- We achieved the desired results with other modifications of the program and that indicates that our race detector was successfully built.

Future Directions

Future work includes creating a race detector for more complex systems such as the Android framework, or other event-driven frameworks, so that prevalent programs such as mobile apps can function as best as possible.

Acknowledgments

I would like to thank Professor Santosh Nagarakatte for introducing me to this topic and helping me learn a lot. I'd also like to thank Ibrahim Umar for his advice and assistance on the project. I'd also like to thank the Aresty center for recruiting me to the Summer Science program and letting me explore research