Detecting Define-Use Bugs On Multithreaded Programs

Rebecca Shapiro (Undergraduate, Wellesley College), Shan Lu (Graduate Mentor, UIUC),
Yuanyuan Zhou (Faculty Mentor, UIUC)

Abstract

As dual core processors that take advantage of multithreaded programs become more popular it becomes important to be able to produce programs free from concurrency and other types of bugs. Concurrency bug detection is extremely difficult, thus it is important to know what types of concurrency bugs are common in real world applications so we can address these problems efficiently. We have found that automicity violations and ordering bugs are common and will be addressing how to detect ordering bugs.

1. Introduction

1.1. Motivation

Today, multithreaded programs are extremely popular. With the increase in popularity of multi core technology, it is becoming extremely important to distribute bug free multithreaded programs. Due to the complexity of multithreaded programs, it is difficult to guarantee their correctness. Therefore, it is important to be able to provide programmers with tools to help them test and debug their applications.

Synchronization bugs can cause a good deal of damage. For example, the great blackout in the northeast of the United States in 2003 that caused tens of millions of people to lose power was initiated by a concurrency bug. [5]

1.2. Previous Work

Concurrency bug detection is an active area of research with much on-going research occurring in this area. Both static and dynamic methods have been used in the past to detect data races. Previous concurrency bug detectors use the *Lockset* algorithm, like Eraser, the *happens-before* algorithm [3], or a combination of the two [1]. The *Lockset* algorithm detects possible concurrency bugs by keeping track of the locks and memory accesses. The *happensbefore* method is an algorithm that uses request queues to manage shared resources.

The work that this research is based on, and is extending, is AVIO. [4] AVIO detects automicity violation bugs by statistically inferring which variable accesses should be atomic. This approach is extremely novel because it infers what regions of code are assumed to be atomic by the programmers and it uses this information to detect atomicity violations that cause software bugs.

One problem that all race detectors encounter is the trade-off between its bug exposing capability and testing cost in terms of time and resources. It is important that future work understand what types of races happen in real world applications so that researchers can make more informed decisions as to how to approach problem of efficiently detecting most or all bugs before software release

2. Bug Survey

Our work aims to survey the types of races that have occurred in real applications such as Mozilla, MySQL, KDE and Gnome. Because all of these programs are developed by the open source community, we were able to study bug reports for each of these programs. By studying different types of software, we can determine the best way to proceed and work to detect the most common types of data race bugs.

Because our sample size was small for each piece of software we studied, we cannot make any conclu-

sions about what type of data race bugs are prevalent in which types of software. However, we were able to use this information to get a overall sense of the types of bugs that real world multitheraded software contain.

Through our study, we found two types of bugs cropped up many times that haven't been explored much with previous race detection work including AVIO. AVIO can infer which sections of code should not have unseriable access interleavings for single variables, but not for multi variables. Through our survey of bugs, we found that our accuracy will improve if we can do the same for atomic sections containing multiple variables. The second thing we found was that many of the bugs occurred due swapped orderings of define-use pairs memory accesses being swapped. This will be explained in further detail in the next section of this paper.

3. Define-Use Bugs

AVIO [4] is able to detect a decent percentage of bugs that we have studied. It is is a good start, however, we can do better. Nearly one third of the order related data race bugs could not currently be detected with AVIO. It is possible to extend AVIO to detect define-use bugs. We must carelly integrate the define-use detection into AVIO so that we can minimize the number of false positives.

3.1. Definition

We are interested in the following things when looking at define-use pairs in a threaded applications: the variable in question, the thread and instruction that wrote to the variable and the thread and instruction that read the variable. For our purpose, the define should happen before the time the use happened. Also, for any variable that is written to (defined) by some thread, we are looking for the next read (use) made by a different thread.

More formally, we can say that some program, P contains a set of memory uses, U, which is all of the memory reads and writes made by P. Each usage, u_k^i, is defined by two things: k, the thread that is making the memory access and i, the instruction in that thread that is making the access. A define-use pair is $< v, u_k^i, u_m^j >$, where v is the variale that is being written to and read. Note that if $i = j$, then the variable is being defined and used by the same thread. This is a trivial define-use pair and we will be ignoring such define-use pairs.

4. Example Define-Use Bugs

To better understand the type of define-use bugs that can occur it is helpful to look at specific examples. These examples give us motivation and things to think about in concurrency bug detection.

4.1. Example 1: Window focus race (Mozilla)

In this bug, the focus isn't given to the correct window. What we found is that if one thread resets the PaintSuppressionTimer (which is a shared variable) before ther other, the focus gets incorrectly set. The folowing are examples of how the execution of code is ordered during good and bad (buggy) runs.

The following is a good run:

window thread	timer therad
mPaintSuppressionTimer->CreateInstance()	
	if(mPaintSuppressionTimer)
	mPaintSuppressionTimer=null
if(mPaintSuppressionTimer)	
mPaintSuppressionTimer=null	

with a define-user pair that looks like:
<mPaintSuppressionTimer,
(timer,mPaintSuppressionTimer=null),
(window,mPaintSuppressionTimer=null)>

And this is what happens during a bad run:

window thread	timer therad
mPaintSuppressionTimer->CreateInstance()	
if(mPaintSuppressionTimer)	
mPaintSuppressionTimer=null	
	if(mPaintSuppressionTimer)
	mPaintSuppressionTimer=null

define-use pair:
<mPaintSuppressionTimer, (window,mPaintSuppressionTimer=null),
(timer,mPaintSuppressionTimer=null)>

We must be careful when evaluating bugs. What we imagined the buggy execution to look like is that there is some variable or data structure marking which container gets focus. We would then expect to see

two different threads writing to it so each can claim focus, and if the ordering is messed up then focus would be incorrectly set. We found what may be causing this bug, but we cannot tell if this truly is the problem or just a symptom of the problem. If one ordering is off, it is possiable to have other important oderings swapped. If we were to keep track of all the define-use relationships we may find more then one swapped ordering in a buggy run. These extra differently ordered define-use relationships may be detected as false positives so we may need to do extra work to prevent such things.

4.2. Example 2: Creating New Threads (Mozilla)

This example adds to our motivation of extending AVIO to detect define-use bugs. The following is a define-use bug happened in an older version of Mozilla during the creation of a child thread. In this example, the state variable is not shared but the mThread structure is shared between threads.

This is what happens when the bug dosen't surface:

```
   Parent Thread            Child Thread
mThread = CreateThread()
mThread->state = ...
                         state = mThread->state
```

The define-use pair looks like the following:
<mThread->state, (Parent, mThread->state=...), (Child, state = mThread->state)>

This is what happens during a run when the bug surfaces:

```
    Parent Thread            Child Thread
mThread = CreateThread()
                        state = mThread->state  (bug!)
 mThread->state = ...
```

With a define-use pair such as:
<mThread->state, (Parent, mThread = CreateThread()), (Child, state=mThread->state)>

This is a buggy run because the mthread->state variable gets set incorrectly. We must note however that when CreateThread() is invoked, mThread->state may or may not be initialized. If it isn't set, then there is a bug bug because it is an uninitialized variable. The use of uninitialized variables are bugs that can be detected using existing bug detection methods. If it is initialized then we have a different define-use pair because the variable is defined by a different instruction then its use, so there will be a define-use pair in the correct run that is missing from the buggy run. When we look at the differences between define-use pairs in subsequent runs, we want to look for missing define-use pairs that cause different defining statement for the same use for some shared variable.

5. Implementation

We are implementing our ordering bug detection tool bugs using Pin [2]. Pin is tool that allows us to instrument programs during runtime. It enables us to trace memory accesses (reads and writes) made by each thread. We can use this information to keep track of all the set of define-use pairs that occur in the first run onf the program. Lets call this set D_1. When the program is run subsequent times, we can find the differences between the sets define-use pairs and discard the pairs that differ if we had a correct run. So for each run we collect a set of define-use pairs D_i and find the set D that contains the define-use pairs that aren't changing between correct runs so D=$D \cap D_i$. We can keep on running the program with the pin tool wrapped around it to train our detector so it knows what define-use pairs are consistent between correct runs. If we have an incorrect run of the program we can see which define-use pairs that are consistent in good runs thar are missing from the buggy run. The missing pairs tell us that the programmer may be assuming that those orderings are important and the program may run incorrectly if they are violated.

The detector we are building will eventually integrated with AVIO to make AVIO more sensitive and powerful. AVIO is also implemented using Pin, so our biggest concern while integrating the two is making sure that they run efficiently and in harmony with each other.

6. Conclusion

Through our bug servey we were able to pin down common types of bugs to work to detect. AVIO does a good job at detecting many of these bugs, but there is

still room for improvement. If we combine the power of AVIO and something that infers what orderings are needed for a correct program, we can have an even more powerful tool.

Although there is much more work to be done in define-use/ordering bugs, our research has allowed us to find motivation and make a push to continue the work.

7. Acknowledgements

I would to thank everyone who have made this enlightening summer of resarch possible for me. First and foremost, I would like to thank my faculy mentor, Yuanyuan Zhou. She gave the the opportunity to spend the summer at the University of Illinois and taught me valuable lessons about life as a researcher and as a graduate student. Shan Lu was my graduate student mentor, who gave me inspiration to take on the project and supported me through the rough times. The other graduate students in the OPERA research group were also exremely wonderful and helpful including, but not limited to: Qingbo Zhu who helped me settle in, Joe Tucek who let me work in the server room with him, Pin Zhou, Spiros Xanthos, Soyeon Park, Zhenmin Li and Feng Qin, Finally I would like to thank the Distributed Mentor Program for the funding that made my research possible.

References

[1] A. Dinning and E. Schonberg. Detecting Access ANomalies in Programs with Critical Sections. *ACM,* 85-96, 1991

[2] IBM Pin. http://rogue.colorado.edu/pin

[3] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACP,* 21(7):558-565, July 1978.

[4] Lu, Shan "AVIO: Detecting Atomicity Violations via Access Interleaving Invariants," *University of Illinois Urbana-Champaign*, Unpublished.

[5] Poulsen, Kevin. *SecurityFocus,* Software Bug Contributed to Blackout. Feburary 11, 2004.

[6] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs *ACM TOCS,* 1997