Software Testing Final Project Report
Electrical and Computer Engineer
University of Texas at Austin

ConAir: A Concurrency Bug Recovery Tool

Submitted to
School of Engineering
by

Shiyu Dong
Xuebin Yan
Supervisor
Sarfraz Khurshid

May 3,2014

Abstract

Multimedia-based ontology construction and reasoning have recently been recognized as two important issues in video search, particularly for bridging semantic gap. The lack of coincidence between low-level features and user expectation makes concept-based ontology reasoning an attractive mid-level framework for interpreting high-level semantics. In this report, we propose a novel model, namely ontology-enriched semantic space (OSS), to provide a computable platform for modeling and reasoning concepts in a linear space. OSS enlightens the possibility of answering conceptual questions such as a high coverage of semantic space with minimal set of concepts, and the set of concepts to be developed for video search. More importantly, the query-to-concept mapping can be more reasonably conducted by guaranteeing the uniform and consistent comparison of concept scores for video search. We explore OSS for several tasks including concept-based video search, word sense disambiguation and detecor fusion. Our empirical findings show that OSS is a feasible solution to timely issues such as the measurement of concept combination and query-concept dependent fusion.

Contents

Chapter 1

Introduction

1.1 Background

Multi-thread programming becomes more and more popular with the advent of multi-core. With multi-core, multi-thread programming has a large advantage over its CPU utility and efficiency. From the low level operating systen (OS)to high level application, multi-thread gains its popularity. A lot of elements should be considered in multi-thread programming, such as threads synchronization, the protection of critical section and the prevention of deadlock. Whereas, the debug of multi-thread programming is very hard because some bugs may happen in certain execution of threads. Unfortunately, the order of threads exectution is decided by scheduler of OS, which is limitedly revised by users. So it's very hard for a user to claim he or she writes a bug-free multi-thread program. A very common way to debug it is to execute the program for many times, such as tens of, hundreds of or even thousands of, and then compare the output. But the problem is still there. If one program is executed for ten times with the same output, there's still some possibility than it will crush on 11th try. Most multi-thread programming program is hidden in it. Even though, the user knows there are some bugs in his or her

programs, it's still very difficult to solve it due to its uncertain execution order. One bug may be caused by one of thread or by multiple ones. In that case, with different execution order, user can not decide where the bug is or how to solve it.

In general, multi-thread programming is very commonly used nowadays and the debug of it is very troublesome and time-consuming. The design of ConAir is based on users' requirements. It is an auto concurrency bug recovery tool which can be used to recovery a buggy multi-thread program.

1.2 Standards

There are four standards which are used to judge a tool's effectiveness, including:

Compatibility: It is used to judge whether there is OS/Hardware modification for the tool. Some tools need low level modification, for example, the change of scheduler of OS. Wheras, OS is designed to be stable and limitedly changeable. If a tool needs OS modification, it may cause some future problems and demage the stableness of the whole system. Sometimes, it may even cause the crush of the system.

Correctness: It is used to determine whether the usage of a tool will generate results infeasible for original software. If so, the usage of a tool causes more problems.

Generality: With high generality, it indicates the tool can solve a wide variety of problems. The effectiveness of a tool is judged based on it because for multi-thread programming, problems are varied. If a tool can solve most common types of concurrency bugs, it is more feasible and practical.

Performance: It is used to tell whether the tool adds small amount of overhead to the original program and has an ability of fast failure recovery.

The design of ConAir takes all four standards into consideration. It does not

require the change of OS or hardware. It guarantees not adding more bugs in origin software. It can solve most types of concurrency bugs and it limits the overhead to the least amount of time.

Chapter 2

ConAir Overview

2.1 Two observations

The design of ConAir is based on two ovservations:

- Roll back a single thread is sufficient to recover from most concurrency-bug failures.
- Reexecute an idempotent region is sufficient to recover from many concurrency-bug failures.

2.2 Observation I

For most concurrency bugs, reexecuting one failing thread is sufficient to fix bugs.In the following part, most common concurrency bugs will be discussed separately:

```
//Violating RAW atomicity
//Roll back thread 1 to recover
/*Thread 1*/
ptr = aptr;
tmp = *ptr;
/*Thread 2*/
ptr = NULL;
```

Figure 2.1: RAW atomic violation

```
//Violating WAW atomicity
//Roll back thread 2 to recover
/*Thread 1*/
log = CLOSE;
log = OPEN;
/*Thread 2*/
if(log != OPEN)
    //output failure
```

Figure 2.2: Violating WAW atomicity

```
//Violating RAR atomicity
//Roll back thread 1 to recover
/*Thread 1*/
if(ptr)
    fputs(ptr);
/*Thread 2*/
ptr = NULL;
```

Figure 2.3: RAR atomic violation

```
//Violating WAR atomicity
//Roll back thread 1 to recover
/*Thread 1*/
cnt += number1;
printf("cnt in total:%d\n",cnt);
/*Thread 2*/
cnt += number2;
```

Figure 2.4: Violating WAR atomicity

2.2.1 Recovering atomicity-violation bugs

Atomicity violations contribute to about 70 % of real-word non-deadlock bugs[].For read and write operations, there are four kinds of atomic violation operations, including Read after Write (RAW), Read after Read (RAR), Write after Read (RAW) and Write after Write (WAW).

As in Figure 2.1, in thread 1, a global pointer is set to point to aptr and then, dereference it and give the value to a local variable tmp. In thread 2, the global pointer is initialized to NULL. Consider the situation that the first sentence of thread 1 is executed and then thread 2 is executed. When ptr is dereferenced and give its value to tmp, segamentation fault happens. In this case, roll back thread 1 until thread 2 finishes and do the given value part. The bug can be fixed.

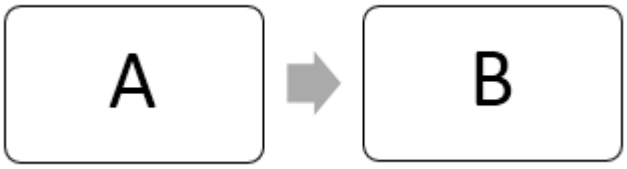

Figure 2.5: Order violation bug

As in Figure 2.2, if thread 2 happens before the end of thread 1, error happens. In this case, rolling back thread 2 can fix the bug.
As in Figure 2.3 and Figure 2.4, still rolling back one thread can fix the possible concurrency bugs. In general, for atomic violation bugs, rolling back one thread is sufficient to fix them.

2.2.2 Recovering order-violation bugs

Another kind of concurrency bug is called order-violation. That is one thread is required to be finished before another one. For example, in Figure 2.5, A is required to finish before B. In this case, roll back B until A finishes can fix the bug.

2.2.3 Rocovering deadlock bugs

A very common concurrency bug in multi-thread programming is deadlock problem. As in Figure 2.6, thread A holds lock 1, thread B holds lock 2 and thread C holds lock 3. A still wants lock 2, while it's held by B. So A is blocked. Similar situation happens in B and C. In this case, roll back any thread can fix deadlock. If roll back B, B releases lock 2, so that C can finish and release lock 3 and 2. Then B can finish. Finally, A can get all resource it wants and finish itself.

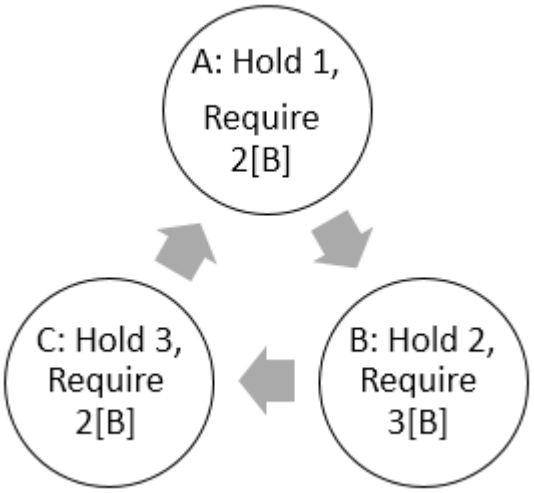

Figure 2.6: Deadlock

2.3 Observation II

For observation II, it indicates that roll back certain area of one thread is sufficient to fix bugs and that area is defined as idempotent region.

Idempotent region:a code region that can be reexecuted for any number of times without changing the program semantics.

It should end before the failing point because it is meaningless to roll back the error part. Besides it, the idempotent region should not contain any writes to shared variables. If it is, roll back it will continously change a global variable, which is unwanted by users. Furthermore, it will not contain any I/O operations. According to investigation, only 15 % concurrency bugs contain I/0 operation. In order to guarantee the performance of ConAir, I/O operation is eliminated from idempotent region. Also, it should not contain any writes to local variables that could cause incorrect execution. This rule is also added to guarantee the correnctness of the ConAir. Some local variables, such as static

ones is initialized once and changed extends for the entire run of the whole program. If it is included in the idempotent region, potentially incorrect output may happen.

All rules are set to guarantee the correctness and performance of the usage of ConAir. Above all, the working principle of the ConAir is to roll back a single thread (failing thread) with its idempotent region.

Chapter 3

Design and Implementation of ConAir

According to the working principle of ConAir, there are three main challenges of the design of ConAir.

- How to decide the failing point of the program
- How to find the idempotent region of the thread
- How to realize roll back step

3.1 Identify Failure Sites

ConAir discusses some types of common errors, including the assertion failures, wrong output, segmentation fault and deadlock. Most concurrency bugs happen in the former types. For the assertion failures, the tool will identify the invocation of __assert_fail(), if it happens, it means assertion error happens. For the wrong output error, it is very hard for the tool to detect it because it does not know what the correct output should be. If user can add assertion in the program to tell the tool what the correct output should be.

```
//assert(e);

if(e) {}
else {
    __assert_fail(...);
}
```

Figure 3.1: Assertion Failure

```
//tmp = *G_ptr;
I_ptr = G_ptr;
if( I_ptr > LowerBound) {}
else{
    Failure:...
}
```

Figure 3.3: Segmentation Fault

```
//printf("...", e, ...);
if(ASSERT(e)) {}
else{
    Failure:...
}
printf("...", e, ...);
```

```
tmp = *G_ptr;

//pthread_mutex_lock(..);
int ret = pthread_mutex_timelock(..);
if(ret != ETIMEOUT) {}
else{
    Failure:...
}
```

Figure 3.4: Deadlock

Then the tool can identify the failure site as mentioned. For the segmentation faults, ConAir identifies every dereference of a heap/ global pointer variable as a potential segmentation fault failure site. For the deadlock, ConAir changes every every pthread_mutex_lock into pthread_mutex_timelock. If it timesout, ConAir detects a deadlock problem in the thread.

3.2 Identify Idempotent Region

It is very hard to defect failure sites in source code because it is written by users directly without a uniform type. So ConAir chooses to analyze the code in bitcode level. It uses LLVM to transform source code into bitcode. According the rules mentioned in Chapter 2.3, in bitcode level, there still exists rules:

Idempotency-destroying with LLVM bitcode:

(1)Writes to global or heap variables; (2)Writes to local variables that are not allocated in virtual registers (static single assignment); (3)Function-call instructions; Eliminate codes which are against idempotency, idempotent region in bitcode level can be found.

3.3 Realize Roll Back Step

The roll back is realized by two functions in C++, named setjmp() and longjmp(). longjmp() is inserted in the line before failing site and setjmp() is inserted in the beginning of idempotent region. When program goes to longjmp(), it will jump to setjmp() instead of carrying on. Then the roll back is realized.Figure 3.6 is the code which ConAir execute and the Figure 3.5 is the origin one.

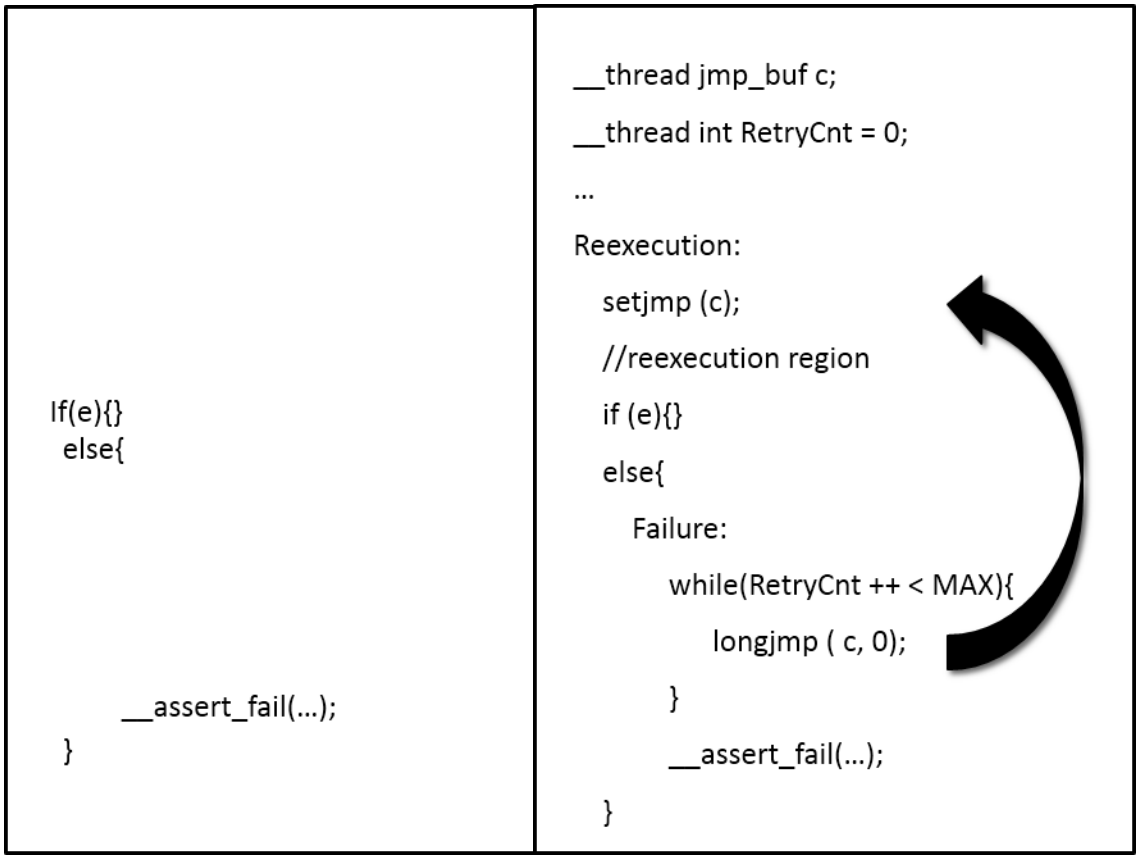

Figure 3.5: The Origin Code

Figure 3.6: The Code with Roll Back