API Race Detector

Jing Zhang[*], Savraj Deep[†]

December 17, 2012

1 Introduction

Data races are a particular kind of threading bugs in shared memory systems, which are difficult and expensive to detect and reproduce. Although static and dynamical tools for detecting data races have been developed, the memory cost and runtime overhead still significantly limit their application in large and complex code bases.

1.1 why tsan is slow

Before we describe the design of API race detector, it is worthwhile to investigate the reasons that cause the overhead in ThreadSanitizer (tsan). There are mainly two steps for tsan to detect data races: (1) first, tsan instruments all load/store instructions (also pointer arithmetic and atomic operations, but we do not discuss here); (2) then it detects the race conditions dynamically via happen-before method. The instrumentation of load/store is done by inserting a call to `__tsan_read` or `__tsan_write` just before the load/store instruction. It should be noted that those instrumented calls are improbable to be optimized (at least at -O1 level), which means if such a call is insert into the hotspot, the code will exhibit severe runtime overhead. Besides, although optimization is applied to reduce some redundant reads before the write, tsan basically treats one load/store at a time and insert one instrumentation each time. In runtime, this leads so many function invocations to tsan rtl.

2 Design

2.1 API race

The tool proposed aims to detect a subset of data races, which we called *API Race*. API race is defined in the context of object-oriented language and refers to the data race occurs at the *data field* of an object. In Object-oriented language, objects, which are the instances of classes, consist of the data fields (memory) and associated procedures (operations accessing the memory). Unlike the procedure language, in which the memory layout and allocation are usually random and disordered, OO has forced (1) the data fields

[*]jz2300@columbia.edu
[†]sd2754@columbia.edu

of an object to be contiguous, and (2) most memory assessment (load/store) operations are in the object's methods. These two distinct features grant the possibility to detect API races with less overhead.

The basic idea is that, instead of instrumenting every load/store exactly before the instruction, we can collect all loads and stores on the object's data fields to a summary and invoke the tsan rtl library only at the end of method. The next two subsections describe the structure of the summary and how to generate the summary.

2.2 Method summary

The *method summary* is a collection of objects' members which are loaded and/or stored in a given method. The *objects* in this definition include different instances of this class as well as the instances of other accessible classes (for example, the public member of another class). Thus, we define *object summary* as a collection of members which are loaded and/or stored in the given object, then a *method summary* is a combination of *object summaries*.

Two data structure designs of *object summary* are considered. The first one is via a *bit array*. Given an instance of a class containing N class members, a bit array of the size of $2N$ bits is used. Each 2 bits is assigned to one member as the load and store marks. The alternative design is via two *typed pointer arrays*. Supposing $M1$ and $M2$ members in this object are loaded and stored in the method, respectively, one typed pointer array of size $M1$ is used to record all loaded members' addresses, and similarly another array of size $M2$ is used to record all stored members' addresses.

It is obvious that each design has pros and cons. The first one can potentially reduce the number of instrumentations (i.e. function calls to rtl), at the expense of fixed size of summary and complexity of implementation; while the second one is simpler in implementation and equally efficient in small method, however, has much more instrumentations in lengthy method. Although the second one is chosen in this initial work, the author believe that if the rtl is also modified to adopt this compact representation (instead of present shadow memory approach), the first one is potentially to reduce both memory and speed overhead in API race detection.

2.3 Summary generation

As mentioned in 2.2, we use the second data structure to represent the method summary, therefore, the summary generation process is to statically collect all loaded members and all stored members respectively. This process is simple, by going though the function code instruction by instruction, and collect loaded members and stored members into two separate sets. However, there is definition issue when instrumenting the rtl calls, for example the code in Listing 1,

```
1  void Foo::Thread1() {
2      if (flag) member++;
3      else xyz++;
4      return;
5  }
6
7  void Foo::Thread2() {
8      xyz = 0;
9      return;
10 }
```

Listing 1: example code with flag dependent branches

The definition problem is that when branch exists in the body of method, the member load/store inside the branch (inner scope) may have no definition at the point just before return, because at runtime it is possible that this branch has never been run. For example, in Listing 1, if `flag==0`, the `member` has no definition at `return` point.

Many solutions are available, (1) move GEPs to the head of function, or (2) duplicate GEPs before ret. However, those solutions are cumbersome and sensitive to the actual code. For example, in (1) it is difficult to decide the moving destination, which must after the GEP of base of the member GEP and before some functional instructions; in (2), duplication could result in a lot of code and complexity when considering the base and offsets are dependent on other variables, especially temporary variables in inner scope.

A simple and clean solution is to insert a store instruction just after member GEP to store the GEP's result address (i.e. the address of the data field) to an allocated memory, and before return point those addresses can be loaded and used to invoke rtl. As the branch condition is still dependent on runtime, if the store instruction is not executed, the allocated memory will contain uninitialized address value. Therefore, after allocation of those temporary memory, they must be set *NULL* value; in additional, the rtl must be able to handle NULL pointer (i.e. do nothing and return). Listing 2 demonstrates the pesudo c code of instrumented `Foo:Thread1` in Listing 1.

```
void Foo::Thread1() {
    int *flag_ptr   = NULL,                    // alloca and null
        *member_ptr = NULL,
        *xyz_ptr    = NULL;
    %flag = &flag;                             // GEP of flag
    store flag_ptr, %flag;                     // store instruction of flag
    if (flag) {
        %member = &member;
        store member_ptr, %member;
        member++;
    }
    else {
        %xyz = &xyz;
        store xyz_ptr, %xyz;
        xyz++;
    }
    %1 = load flag_ptr;                        // load flag
    %2 = load member_ptr;
    %3 = load xyz_ptr;
    __tsan_read4(%1);                          // instrument flag
    __tsan_write4(%2);
    __tsan_write4(%3);
}
```

Listing 2: pesudocode after instrumentations

We remind the readers that, (1) the solution above adds a little runtime overhead, even if the load/store is embedded in the hotspot region (for example, for loop). Because if the GEP is loop independent, compile should and will move GEP outside of the loop, therefore the inserted store instruction associated to the GEP will not appear inside of the loop. (2) this approach is partially collecting runtime load/store information, thus it can correct handle some of if-branch case, such as code in Listing1, and it generates fewer false positives comparing pure static approach. However, false positive still exists, for example,

```
void Foo::Thread1() {
```

```
2     if (flag) mem2 = mem1;
3     else mem1++;
4 }
5
6 void Foo::Thread2() {
7     mem3 = mem1;
8 }
```

Listing 3: suggested code for if-case

in method `Thread1`, two StoreInsts to store the address of `mem1` associated to the load and store operations are inserted after GEP of `mem1`, and will always race with load operation in `Thread2` regardless of `flag`.

Last, this approach could be extended to improve the tsan itself, however, we will not discuss it here as it is not relevant to the context of API race detection.

2.4 Synchronization function call

It is arguable what is the largest size of code to be summarized for data race detection. However, it is obviously not optimum at function size due to foreseeable false positives. This subsection we focus on one of the many problems, existence of synchronization call in function body.

No matter what level of summary (function, basic block), the summary should not go across the synchronization barrier, if we use the happen-before method to detect data race. If synchronization call exists within the given function, and instrumentations only inserted before return, both false positives and false negatives may be generated.

Supposing we know whether it is a sync call for every function call inside the given function (this is the issue discussed later), we could simply propose two solutions: (1) if there exists a sync call, all member loads/stores are instrumented at exact position for the whole basic block or whole function. This is an expensive and *unsafe* solution. As the caller's function level summary is still across the sync barrier inside of the callee, there are chance to trigger false negative or positive in caller. If we continue to apply exact instrumentation to caller, eventually a large portion of threading program is instrumented exactly, leading to similar runtime overhead comparing to tsan. (2) A better solution is to instrument everything already in summary just before sync call, and clear all records after sync call. This approach guarantees summary will not go across the sync barrier. The cost should be optimum as synchronization itself is expensive.

Given a white list of sync functions (for example, pthread_barrier_), it is simple to decide whether a call is a sync call by comparing the function name (and function signature if necessary). However, this is not enough, if considering embedded function calls. Therefore, traveling the static call graph is required to decide the sync call. We use simpel DFS along with cache to do lazy evaluation. There are heuristics to make it working cheap: (1) call graph can contain loop (such as recursive algorithm), DFS results in infinite loop; thus a max depth is set. (2) if one directly call to white list member, we consider all parent callers are sync call. This ignores the real runtime results, and is conservative to treat possible sync call (can only be confirmed at runtime) as sync call. This is a place we do not explore much, intuitively better heuristics (such as restriction on depth) should help to balance speed and false results.

2.5 Instrumentation process

The pesudocode below illustrates how to perform instrumentation for a given function.

```
loop each basic block:
    loop each instruction:
        if LoadInst or StoreInst:
            push in LocalLoadsAndStores
        else if GetElementPtrInst:
            push in LocalGEPs
        else if ReturnInst:
            push in RetVec
        else if CallInst:
            if it is SyncCall:
                select class members from LocalGEPs
                select member loads and stores from LocalLoadsAndStores
                instrument each load and store before the CallInst
                inserts stores to null all temps
    end loop
    select class members from LocalGEPs
    select member loads and stores from LocalLoadsAndStores
end loop

instrument all returns
instrument function entry and exit
```

3 Implementation

The implementation of API Race detector is based on the ThreadSanitizer V2 [1,2] available in llvm3.2. We directly use the tsan runtime library with only a modification to make it safely treat NULL pointer as mentioned in 2.3. The modification is written as patches provided in APIRace src code.

Instrumentation code is via llvm function pass. Unlike tsan, we do not integrate it into clang, therefore, the invocation of instrumentation must via `opt` in llvm, see AppendixA.4 for detailed commands. We want to emphasize the importance of compilation flags when using clang with APIRace (this is also apply to tsan). First, the optimization flag can only be `-O0` or `-O1`, and `-O2` must be used with `-fno-inline-functions`. As function inline make loads and stores escape from instrumentation, and cause false negative. Second, the tsan rtl is compiled with `-fPIE` to facilitate manipulation of address space and mmap, thus, user program must also be compiled with `-fPIE` and linked with `-pie` flag.

4 Results

4.1 simple tests

We rewrote most of simple tests available in tsan rtl to semantic equivalent C++ code as shown in Table 1. The false negative of race-on-heap case is understandable. Although we instrument the member with pointer type, the allocated memory is not instrumented, and escapes from the runtime checking.

Tests	Result
simple read-read no race	PASS
simple read-write race	PASS
simple write-write race	PASS
sleep synchronization race	PASS
thread leak cases	PASS
barrier races	PASS
if-conditional races	PASS
race on heap	FAIL, False Negative

Table 1: list of simple tests

Next, we checked the correctness for partial C++ grammar to ensure the code correctly collect all members for objects of different classes. Further work is required to check the correctness on templates.

4.2 simple benchmarks

We choose two simple case to show the performance benefit of APIRace comparing to tsan, fibonacci and for-loop. Note, these two are not multithreaded program, we just use them as two extreme case to demonstrate the runtime overhead. The fibonacci code is to compute the fibonacci number via *recursion*. The major runtime cost is push/pop on stack due to function recursive calls. The for-loop code is simple for loop with complex branches inside to avoid compiler optimization. This for-loop can be used to simulating usual hotspot. The two program are compiled without any instrument(a), with tsan, with APIRace, using identical compiling flag, and the timing is displayed in Table2.

Test	(a) w/o	(b) tsan	(c) api
fibonacci	1.446s	9.491s	8.435s
	1.526s	9.057s	8.413s
	1.510s	9.082s	8.888s
for-loop	2.745s	61.341s	2.731s
	2.604s	60.218s	2.714s
	2.731s	60.844s	2.726s

Table 2: simple benchmark of APIRace vs tsan

We want to remind the reader, to insure the results are valid and fair, make sure (1) the compiling flag, especially optimization flag must be identical; (2) the tsan and APIRace must use revision of tsan_rtl library. As the tsan is still under rapid development, the performance of tsan_rtl is constantly changing.

In fibonacci case, APIRace is slightly faster then tsan within 5%. In for-loop case, APIRace show 22x speedups comparing to tsan, and overhead comparing no instrumented code is negligible.

4.3 larger tests

4.3.1 pbzip2

This is not a good testing case, because the code is just c++ parallel wrapper of zlib. When in runtime zlib calls are the hotspot, thus threading overhead is not obvious. Nevertheless, we describe the compilation and provide the result of our study.

You can download pbzip2 at [3]. The version we tested is 1.1.6. Please note the code itself has a (use uninitialized) bug which causes segmentation fault at `-O1` and `-O2` optimization level with clang. The patch is provided in our source code. The compilation flag is `-O1 -fPIE -fno-builtin` for all cases. The test is to compress a 100MB MP4 file, via

```
pbzip2.apirace.x -v -p2 -c < test.mp4 > /dev/null
```

the timing results are listed in table**??**. Basically, there is no speed difference between tsan and APIRace as expected.

Test	(a) w/o	(b) tsan	(c) api
pbzips	16.81s	19.98s	19.85s
	15.80s	20.05s	19.38s
	15.67s	19.71s	19.92s

Table 3: benchmark of speed for pbzip2

4.3.2 GameKit

The next test is a game engine called *GameKit*, you can find it at [4]. The revision tested is 1219. We have written bash wrapper of clang++ with APIRace pass to substitute the g++ in cmake. The compilation with APIRace is done without error.

We tested the simple game demo at `$gamekit/build/Samples/Runtime/AppOgreKit`. As the test requires user interaction, we manually do the same operation in each tests: (1) first start the game, wait the loading finish, (2) when we see the cartoon, and observe the box at right side fall onto the ground, press 'w' to move the monkey forward to the tree, and the press 's' bring it back, last press 'esc' to exit.

All output bugs are saved and available in our src. We use tsan results as the reference, and manually match the data races based on traceback on call stack and size of read/write memory, further we look at the code to confirm the data races variable. All results are available in `GameKitDemo.bug` file, and summarized in Table4. Overall, tsan detected 22 data races, APIRace found 14 data races; 13 out of 17 data races from APIRace can be matched to tsan's result; there is 1 false positive, and 7 false negative. 4 out of 7 false negatives involve of c semaphore, which very likely escape from instrumentation (needs further confirmation).

Except the data race bugs, we got 38 warnings on `heap-use-after-free`. The major issue here is the instrumented `__tsan_read/write` is placed after `delete`. To fix it, we need to also track pointers in `delete`, which is left for future work.

We have not done the speed tests on fps, as we are not familiar on this game engine and no time to finish it.

	real data races	FP	FN	total detected
tsan	22			22
APIRace	13	1	7	14

Table 4: Data races detected in GameKit by tsan and APIRace.

4.3.3 firefox

We cannot successfully compile firefox even with tsan due to linking errors. As time limitation is applied, we give up this test.

References

[1] data-race-test,race detection tools and more. "http://code.google.com/p/data-race-test/wiki/ThreadSanitizer".

[2] Threadsanitizer v2. "http://code.google.com/p/thread-sanitizer/wiki/ThreadSanitizerDevelopment".

[3] pbzip2, version 1.1.6. "http://compression.ca/pbzip2/".

[4] gamekit. "http://code.google.com/p/gamekit/".

A Compilation of APIRace

A.1 Prerequisites

- llvm 3.2, earlier version is NOT working.
- gcc 4.6.3 or above, earlier version cause compilation failure of llvm 3.2.
- x86_64, i386 is not compatible to tsan_rtl
- svn

A.2 compilation of llvm and clang

llvm 3.2 is just released in Dec 11, 2012. Our development is based on their trunk revision 167525. It is possible that something is broken in release version 3.2 (as we do not test); however, revision 167525 is guaranteed to work. Wisely choose the one you want. Given the revision, you can compile llvm, clang and rtl by following,

```
R=167525
svn co -r $R http://llvm.org/svn/llvm-project/llvm/trunk llvm
cd llvm
(cd tools && svn co -r $R http://llvm.org/svn/llvm-project/cfe/trunk clang)
(cd projects && svn co -r $R http://llvm.org/svn/llvm-project/compiler-rt/trunk
   compiler-rt)
```

```
mkdir build
(cd build && ../configure --enable-optimized && make -j 8)
```

A.3 compilation of API Race pass

```
cd APIRace
# compilation of pass
(cd opt && make)
# compilation of rtl
(cd tsan-rtl && rtl.sh)
# tests
(cd tests && make)
# benchmark
(cd benchmark && make)
# large test see README in src
```

A.4 how to run apirace

It is suggested to refer the Makefile in test or benchmark first. *Compilation flags DOES matter!!!* Basically you can instrument your code via the following four steps,

```
# given foo.cc

# 1. Get llvm binary code
clang++ -fPIE -fno-builtin -Wall -O1 -emit-llvm -c foo.cc -o foo.cc.bc
# 2. do instrumentation
opt -O1 -load=$APIRACE_DIR/APIRaceInstr.so -apirace < foo.cc.bc > foo.instr.bc
# 3. compile to object file
clang++ -fPIE -fno-builtin -O1 -c foo.instr.bc -o foo.o
# 4. linking to tsan rtl
clang++ -pie -O1 -fno-builtin -o foo.api.x foo.o -lpthread -ldl $APIRACE_DIR/lib/
    libtsan_mod.a
```

alternatively, you can use the sh wrapper in $APIRACE_DIR/bin, for example clang-apirace.sh, like

```
jing@linux-crsq:~/src/APIRace/tests>../bin/clang-apirace.sh -O1  -c simple-member-
    read-read-norace.cc
/home/jing/src/APIRace/tests/../bin/clang-apirace.sh
EXEC: /usr/local/bin/clang++ -fPIE -Wall -fno-builtin -O1  -O1 -emit-llvm -c
    simple-member-read-read-norace.cc -o  simple-member-read-read-norace.cc.bc
EXEC: /usr/local/bin/opt -load=/home/jing/src/APIRace/lib/APIRaceInstr.so -apirace
     simple-member-read-read-norace.cc.bc -o  simple-member-read-read-norace.cc.
    instr.bc > simple-member-read-read-norace.api.stat 2 > & 1
EXEC: /usr/local/bin/clang++ -fPIE -Wall -fno-builtin -O1  -O1 -c  simple-member-
    read-read-norace.cc.instr.bc -o simple-member-read-read-norace.o
```

this script should be able to walkaround make and cmake configure. However, it is too simple to be robust, use it at caution.