Dynamic Race Detection with LLVM Compiler

Compile-time instrumentation for ThreadSanitizer

Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov

OOO Google, 7 Balchug st., Moscow, 115035, Russia
{kcc,glider,timurrrr,dvyukov}@google.com

Abstract. Data races are among the most difficult to detect and costly bugs. Race detection has been studied widely, but none of the existing tools satisfies the requirements of high speed, detailed reports and wide availability at the same time. We describe our attempt to create a tool that works fast, has detailed and understandable reports and is available on a variety of platforms. The race detector is based on our previous work, ThreadSanitizer, and the instrumentation is done using the LLVM compiler. We show that applying compiler instrumentation and sampling reduces the slowdown to less than 1.5x, fast enough for interactive use.

1 Introduction

Recently the growth of CPU frequencies has transformed into the growth of the number of cores per CPU. As a result, multithreaded code became more popular on desktops, and concurrency bugs, especially *data races*, became more frequent. The classical approach to dynamic race detection assumes that program code is instrumented and program events are passed to an analysis algorithm [6, 10]. Some of the publicly available race detectors for native code [13, 5, 11] use *run-time instrumentation*. There are also tools that use *compiler instrumentation* [8, 4, 1, 9], but none is publicly available on most popular operating systems.

In [11] we described ThreadSanitizer (TSan-Valgrind), a dynamic race detector for native code based on *run-time instrumentation*. The tool has found hundreds of harmful races in a number of C++ programs at Google, including some in the Chromium browser [2]. Significant slowdown remains the largest problem of ThreadSanitizer: for many tests we observed 5x–30x slowdown due to the complex race detection algorithm; on heavy web applications the slowdowns were even greater (50x and more) because of the underlying translation system (Valgrind, [13])[1]. Another problem with Valgrind is that it serializes all threads; with multicore machines this becomes a serious limitation. Finally, Valgrind is not available on some platforms we are interested in (entirely unavailable on Windows, hard to deploy on ChromiumOS).

In this paper we present TSan-LLVM, a dynamic race detector that uses *compile-time instrumentation* based on a widely available LLVM compiler [7].

[1] Mainly because Valgrind had to execute much single-threaded JavaScript code.

The new tool shares the race detection logic with ThreadSanitizer, but has greater speed and better portability. Our work resembles LiteRace [8] (both use compiler instrumentation and sampling, the performance figures are comparable), but the significant advantages of our tool are the more precise race detection algorithm [11], the granularity of sampling and public availability. We have also made an instrumentation plugin for GCC, but do not describe it here due to the limited space.

2 Compiler instrumentation

The compiler instrumentation is implemented as a pass for the LLVM compiler. The resulting object files are linked against our runtime library.

2.1 Runtime library

As opposed to a number of popular race detection algorithms [10, 13, 8], ThreadSanitizer [11] tracks both locksets and the happens-before relation. This allows it to switch between the pure happens-before mode, which reports no false positives, but may miss bugs, and the hybrid mode, which finds more potential races, but may give false reports. In both modes the tool reports the stacks of all the accesses constituting the race, along with the locks taken and the origin of memory involved. This is vital in order to give all the necessary information to the tool users.

The algorithm is basically a state machine – it receives program events, updates the internal state and, when appropriate, reports a potential race. The major events handled by the state machine are: Read, Write (memory accesses); Signal, Wait (happens-before events); Lock, Unlock (locking events).

RTL, or the runtime library, provides entry points for the instrumented code, keeps all the information about the running program (e.g. the location and size of thread stacks and thread-local storage) and generates the events by wrapping the functions that are of interest for the race detector:

– libpthread synchronization primitives and thread manipulation routines;
– memory allocation routines (e.g. `malloc()`/`free()`, `mmap()`, `new`/`delete`);
– other functions that imply happens-before relations in the real world programs (`atexit()`, `read()`/`write()`);
– dynamic annotations [11], which tell the detector about specific means of synchronization.

2.2 Instrumentation

The instrumentation is done at the LLVM IR level, which gives a number of advantages over the more high-level approaches [9].

For each translation unit the following steps are done:

Call stack instrumentation. In order to report nearly precise contexts for all memory accesses that constitute a race, ThreadSanitizer has to maintain a correct call stack for every thread at all times. We keep a per-thread stack with a pointer to its top; at every function entry and exit the stack is changed, and at every basic block start the top is updated[2].

To keep the call stack consistent, the tool also needs to intercept `setjmp()` and instrument the LLVM `invoke` instruction to roll back the stack pointer when necessary. This is not done yet, because these features are rarely used at Google.

Memory access instrumentation. Each memory access event is a tuple of 5 attributes: *thread id*, *ADDR*, *PC*, *isWrite*, *size*. The last three are statically known. Memory accesses that happen in one basic block[3] are grouped together; for each block the compiler module creates a *passport* – an array of tuples representing each memory access. Every memory access is instrumented with the code that records the effective address of the access into a thread-local buffer. The buffer contents are processed by ThreadSanitizer at the end of each block.

2.3 Sampling

In order to decrease the runtime overhead even more, we've experimented with sampling the memory accesses. We exploit the *cold-region hypothesis* [8]: data races are more likely to occur in cold regions of well-tested programs, because the races in hot regions either have been already found and fixed or are benign.

The technique we use for sampling is quite similar to that suggested in LiteRace [8]: ThreadSanitizer adapts the thread-local sampling rate per code region (basic block or superblock, as opposed to LiteRace, which tracks functions) such that the sampling rate decreases logarithmically with the total number of executions. Unlike in LiteRace, the instrumented code is always executed and the memory access addresses are put into the buffer, which is then either processed or just ignored depending on the value of the execution counter region. To save memory, we do not keep this counter for each thread – instead we take the thread id modulo 8 and keep only 8 pairs of counters for each region storing them such that false sharing is minimized.

2.4 Limitations and further improvements

The compiler-based instrumentation has some disadvantages over the run-time instrumentation: the races in the code which was not re-compiled with the race detection enabled (system libraries, JIT-ed code) will be missed, the tool usage is less convenient since it requires a custom build[4]. As we show in the next section, the benefit of much higher speed outweighs these limitations for our use cases.

Much could be done to decrease the overhead even further by reducing the number of instrumented memory accesses without loosing races. A promising direction is to use compiler's static analysis to skip accesses that never escape the current thread. Another optimization is to instrument only one of the accesses to the same memory location on the same path.

[2] Optimizations may apply, e.g. leaf functions do not need to maintain the call stack.
[3] We also extend this approach to handle larger acyclic regions of code (superblocks).
[4] Valgrind-based tools also usually require a custom build to avoid false positives.

3 Results

To estimate the performance of our tool, we ran it on two Chromium tests and a synthetic microbenchmark. We've already used TSan-Valgrind to test Chromium (see [11]) and were able to compare the results and assess the benefits of the compile-time instrumentation approach for a real-word application. Cross_fuzz [3] is a cross-document DOM binding fuzzer that is known to stress the browser and reveal complex bugs, including races. Net_unittests [2] is a set of nearly 2000 test cases that test various networking features and create many threads. The third test we ran just calls a simple non-inlined function[5] many times:

```
void IncrementMe(int *x) {  (*x)++;  }
```

One variant of the test is single-threaded, the other variant spawns 4 threads that access separate memory regions.

The measurements were done on an HP Z600 machine with 2 quad-core Intel Xeon E5620 CPUs (with hyper-threading enabled) and 12G RAM.

Table 1 contains execution times for uninstrumented binaries run natively and under TSan-Valgrind compared to the instrumented binaries tested in two modes: with full memory access analysis (TSan-LLVM, sampling disabled) and with race detection disabled (TSan-LLVM-null, an empty stub is called at the end of each block, therefore a highly sampled run is slightly faster). We've also measured run times under Intel Inspector XE [5], Memcheck[6] and Helgrind version 3.6.1 [13].

Table 1. TSan-LLVM compared to other tools. Time in seconds.

tool	cross_fuzz	net_unittests	synthetic, 1 thread	synthetic, 4 threads
native run	71.6	87	0.9	0.9
Memcheck	1275	991	33	133
Inspector XE	failed	1064	130	480
Helgrind	failed	2529	40	154
TSan-Valgrind	325.2	592	49	191
TSan-LLVM	190.9	206	15.5	17
TSan-LLVM-null	78.6	119	2	2.1

The comparison shows that TSan-LLVM outperforms TSan-Valgrind by 1.7x–2.5x on the big tests. The tool did not instrument libc and other system libraries, but we estimate their performance impact to be within 2%–3%.

Table 2 shows how the performance depends on the sampling parameter (a number k which, if greater than 0, means that sampling starts after 2^{32-k} executions of a block). Using the sampling value of 20 is 1.5x–2x faster than without

[5] Part of `racecheck_unittest` [12], a test suite for data race detectors. TSan-Valgrind passes all of them, TSan-LLVM passes those that do not require instrumenting libc.

[6] Memcheck, the Valgrind memory error detector, does different kind of instrumentation and can not ignore JavaScript, but its figures may still serve as a data point.

sampling on the chosen benchmarks. In this mode the slowdown compared to the native run is less than 1.5x, and the tool is still capable of finding a number of known races. Better analysis of sampling vs. accuracy is still to be done.

Table 2. TSan-LLVM performance with various sampling values.

test name	sampling parameter (0–31)	0	10	20	30
cross_fuzz	time, sec	190.9	142.3	94.5	78.1
	accesses analyzed, %	100.0	77.8	16.2	3.6
net_unittests	time, sec	206	190	134	117
	accesses analyzed, %	100.0	33.7	14.1	13.4

4 Conclusions

We present a dynamic race detector based on low-level compiler instrumentation. This detector has a large speed advantage (1.7x–2.5x on the real-world applications) over our previous Valgrind-based tool, and a slowdown factor of 2.5x (less than 1.5x, if used with sampling), which is fast enough to run interactive UI tests on the instrumented Chromium browser and find otherwise hidden bugs in it. The achieved speedup could be improved even further if additional compile-time static analysis is employed.

References

1. Sun Studio, http://developers.sun.com/sunstudio
2. Chromium browser, http://dev.chromium.org
3. Cross Fuzz, http://lcamtuf.coredump.cx/cross_fuzz
4. Duggal, A.: Stopping Data Races Using Redflag. Master's thesis, Stony Brook University (May 2010), technical Report FSL-10-02
5. Intel Inspector XE, http://software.intel.com/en/articles/intel-parallel-studio-xe
6. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM (1978)
7. The LLVM Compiler Infrastructure, http://llvm.org
8. Marino, D., Musuvathi, M., Narayanasamy, S.: Literace: effective sampling for lightweight data-race detection. In: PLDI (2009)
9. Pozniansky, E., Schuster, A.: Efficient on-the-fly data race detection in multithreaded c++ programs. In: PPoPP '03. pp. 179–190. ACM (2003)
10. Savage, S., Burrows, M., et al.: Eraser: a dynamic data race detector for multithreaded programs. ACM TOCS 15(4), 391–411 (1997)
11. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: data race detection in practice. WBIA (2009)
12. ThreadSanitizer project: documentation, source code, dynamic annotations, unit tests, http://code.google.com/p/data-race-test
13. Valgrind, Helgrind., http://www.valgrind.org