ThreadSanitizer – data race detection in practice

Konstantin Serebryany
OOO Google
7 Balchug st.
Moscow, 115035, Russia
kcc@google.com

Timur Iskhodzhanov
MIPT
9 Institutskii per.
Dolgoprudny, 141700, Russia
timur.iskhodzhanov@phystech.edu

ABSTRACT

Data races are a particularly unpleasant kind of threading bugs. They are hard to find and reproduce – you may not observe a bug during the entire testing cycle and will only see it in production as rare unexplainable failures. This paper presents ThreadSanitizer – a dynamic detector of data races. We describe the hybrid algorithm (based on happens-before and locksets) used in the detector. We introduce what we call dynamic annotations – a sort of race detection API that allows a user to inform the detector about any tricky synchronization in the user program. Various practical aspects of using ThreadSanitizer for testing multithreaded C++ code at Google are also discussed.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging — *Testing tools.*

General Terms

Algorithms, Testing, Reliability.

Keywords

Concurrency Bugs, Dynamic Data Race Detection, Valgrind.

1. INTRODUCTION

A data race is a situation when two threads concurrently access a shared memory location and at least one of the accesses is a write.

Such bugs are often difficult to find because they happen only under very specific circumstances which are hard to reproduce. In other words, a successful pass of all tests doesn't guarantee the absence of data races. Since races can result in data corruption or segmentation fault, it is important to have tools for finding existing data races and for catching new ones as soon as they appear in the source code.

The problem of precise race detection is known to be NP-hard (see [20]). However, it is possible to create tools for finding data races with acceptable precision (such tools will miss some races and/or report false warnings).

Virtually every C++ application developed at Google is multithreaded. Most of the code is covered with tests, ranging from tiny unit tests to huge integration and regression tests. However, our codebase had never been studied using a data race detector. Our main task was to implement and deploy a continuous process for finding data races.

2. RELATED WORK

There are a number of approaches to data race detection. The three basic types of detection techniques are: static, on-the-fly and postmortem. On-the-fly and postmortem techniques are often referred to as dynamic.

Static data race detectors analyze the source code of a program (e.g. [11]). It seems unlikely that static detectors will work effectively in our environment: Google's code is large and complex enough that it would be expensive to add the annotations required by a typical static detector.

Dynamic data race detectors analyze the trace of a particular program execution. On-the-fly race detectors process the program's events in parallel with the execution [14, 22]. The postmortem technique consists in writing such events into a temporary file and then analyzing this file after the actual program execution [18].

Most dynamic data race detection tools are based on one of the following algorithms: happens-before, lockset or both (the hybrid type). A detailed description of these algorithms is given in [21]. Each of these algorithms can be used in the on-the-fly and postmortem analysis.

3. HISTORY OF THE PROJECT

Late in 2007 we tried several publicly available race detectors, but all of them failed to work properly "out of the box". The best of these tools was Helgrind 3.3 [8] which had a hybrid algorithm. But even Helgrind had too many false positives and missed many real races. Early in 2008 we modified the Helgrind's hybrid algorithm and also introduced an optional pure happens-before mode. The happens-before mode had fewer false positives but missed even more data races than the initial hybrid algorithm. Also, we introduced dynamic annotations (section 5) which helped eliminate false positive reports even in the hybrid mode.

Still, Helgrind did not work for us as effectively as we would like it to — it was still too slow, missed too many races in the pure happens-before mode and was too noisy in the hybrid mode[1].

So, later in 2008 we implemented our own race detector. We called this tool **"ThreadSanitizer"**. ThreadSanitizer uses a new simple hybrid algorithm which can easily be used in a pure happens-before mode. It supports the dynamic annotations we have suggested for Helgrind. Also, we have tried to make the race reports as informative as possible to make the tool easier to use.

4. ALGORITHM

ThreadSanitizer is implemented as a Valgrind [19] tool[2]. It observes the program execution as a sequence of events. The most important events are *memory access* events and *synchronization* events. Memory access events are READ and WRITE. Synchronization events are either *locking* events or *happens-before* events. Locking events are WRLOCK, RDLOCK, WRUNLOCK and RDUNLOCK. Happens-before events are SIGNAL and WAIT[3].

These events, generated by the running program, are observed by ThreadSanitizer with the help of the underlying binary translation framework (Valgrind). The detector keeps the *state* based on the history of the observed events and updates it using a certain *state machine*. To formally describe the state and the state machine we will need some definitions.

4.1 Definitions

Tid (thread ID): a unique number identifying a thread of the running program.

ID: a unique ID of a memory location[4].

EventType: one of READ, WRITE, WRLOCK, RDLOCK, WRUNLOCK, RDUNLOCK, SIGNAL, WAIT.

Event: a triple $\{\text{EVENTTYPE}, Tid, ID\}$. We will write $\text{EVENTTYPE}_{Tid}(ID)$ or $\text{EVENTTYPE}(ID)$ if the Tid is obvious from the context.

Lock: an ID that appeared in a locking event.

A lock L is **write-held** by a thread T at a given point of time if the number of events $\text{WRLOCK}_T(L)$ observed so far is greater than the number of events $\text{WRUNLOCK}_T(L)$.

A lock L is **read-held** by a thread T if it is write-held by T or if the number of events $\text{RDLOCK}_T(L)$ is greater than the number of events $\text{RDUNLOCK}_T(L)$.

Lock Set (LS): a set of locks.

Writer Lock Set (LS_{Wr}): the set of all write-held locks of a given thread.

Reader Lock Set (LS_{Rd}): the set of all read-held locks of a given thread.

Event Lock Set: LS_{Wr} for a WRITE event and LS_{Rd} for a READ event.

Event context: the information that allows the user to understand where the given event has appeared. Usually, the event context is a stack trace.

Segment: a sequence of events of one thread that contains only memory access events (i.e. no synchronization events). The context of a segment is the context of the first event in the segment. Each segment has its writer and reader LockSets (LS_{Wr} and LS_{Rd}). Each memory access event belongs to exactly one segment.

Happens-before arc: a pair of events $X = \text{SIGNAL}_{T_X}(A_X)$ and $Y = \text{WAIT}_{T_Y}(A_Y)$ such that $A_X = A_Y$, $T_X \neq T_Y$ and X is observed first.

Happens-before: a partial order on the set of events. Given two events $X = \text{TYPEX}_{T_X}(A_X)$ and $Y = \text{TYPEY}_{T_Y}(A_Y)$, the event X *happens-before* or *precedes* the event Y (in short, $X \prec Y$; $\preceq$ and $\not\preceq$ are defined naturally) if X has been observed before Y and at least one of the following statements is true:

- $T_X = T_Y$.
- $\{X, Y\}$ is a happens-before arc.
- $\exists E_1, E_2 : X \preceq E_1 \prec E_2 \preceq Y$ (i.e. $\prec$ is transitive).

The happens-before relation can be naturally defined for segments since segments don't contain synchronization events. Figure 1 shows three different threads divided into segments.

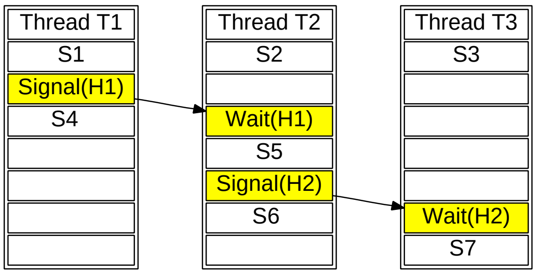

Figure 1: Example of happens-before relation. $S_1 \prec S_4$ (same thread); $S_1 \prec S_5$ (happens-before arc $\text{SIGNAL}_{T_1}(H_1)$ – $\text{WAIT}_{T_2}(H_1)$); $S_1 \prec S_7$ (happens-before is transitive); $S_4 \not\prec S_2$ (no relation).

Segment Set: a set of N segments $\{S_1, S_2, ..., S_N\}$ such that $\forall i, j : S_i \not\preceq S_j$.

Concurrent: two memory access events X and Y are concurrent if $X \not\preceq Y$, $Y \not\preceq X$ and the intersection of the lock sets of these events is empty.

Data Race: a data race is a situation when two threads concurrently access a shared memory location (i.e. there are two *concurrent* memory access events) and at least one of the accesses is a WRITE.

4.2 Hybrid state machine

The state of ThreadSanitizer consists of global and per-ID states. The global state is the information about the synchronization events that have been observed so far (lock sets, happens-before arcs). Per-ID state (also called *shadow memory* or *metadata*) is the information about each memory location of the running program.

ThreadSanitizer's per-ID state consists of two segment sets: the writer segment set SS_{Wr} and the reader segment set SS_{Rd}. SS_{Wr} of a given ID is a set of segments where the writes to this ID appeared. SS_{Rd} is a set of all segments where the reads from the given ID appeared, such

[1]The current version of Helgrind (3.5) is different; it is faster but has only a pure happens-before mode.
[2]At some point it was a PIN [15] tool, but the Valgrind-based variant has proved to be twice as fast.
[3]In the original Lamport's paper [14] these are called *Send* and *Receive*.
[4]In the current implementation ID represents one byte of memory, so on a 64-bit system it is a 64-bit pointer.

that $\forall S_r \in SS_{Rd}, S_w \in SS_{Wr} : S_r \not\preceq S_w$ (i.e. all segments in SS_{Rd} happen-after or are unrelated to segments in SS_{Wr}).

Each memory access is processed with the following procedure. It adds and removes segments from SS_{Wr} and SS_{Rd} so that SS_{Wr} and SS_{Rd} still match their definitions. At the end, this procedure checks if the current state represents a race.

```
HANDLE-READ-OR-WRITE-EVENT($IsWrite, Tid, ID$)
 1  $\triangleright$ Handle event READ $_{Tid}(ID)$ or WRITE $_{Tid}(ID)$
 2  $(SS_{Wr}, SS_{Rd}) \leftarrow$ GET-PER-ID-STATE($ID$)
 3  $Seg \leftarrow$ GET-CURRENT-SEGMENT($Tid$)
 4  if $IsWrite$
 5     then $\triangleright$ WRITE event: update $SS_{Wr}$ and $SS_{Rd}$
 6          $SS_{Rd} \leftarrow \{s : s \in SS_{Rd} \wedge s \not\preceq Seg\}$
 7          $SS_{Wr} \leftarrow \{s : s \in SS_{Wr} \wedge s \not\preceq Seg\} \cup \{Seg\}$
 8     else $\triangleright$ READ event: update $SS_{Rd}$
 9          $SS_{Rd} \leftarrow \{s : s \in SS_{Rd} \wedge s \not\preceq Seg\} \cup \{Seg\}$
10  SET-PER-ID-STATE(ID, $SS_{Wr}, SS_{Rd}$)
11  if IS-RACE($SS_{Wr}, SS_{Rd}$)
12     then $\triangleright$ Report a data race on $ID$
13          REPORT-RACE($IsWrite, Tid, Seg, ID$)
```

Checking for race follows the definition of race (4.1). Note that the intersection of lock sets happens in this procedure, and not earlier (see also 4.5).

```
IS-RACE($SS_{Wr}, SS_{Rd}$)
 1  $\triangleright$ Check if we have a race.
 2  $N_W \leftarrow$ SEGMENT-SET-SIZE($SS_{Wr}$)
 3  for $i \leftarrow 1$ to $N_W$
 4     do $W_1 \leftarrow SS_{Wr}[i]$
 5        $LS_1 \leftarrow$ GET-WRITER-LOCK-SET($W_1$)
 6        $\triangleright$ Check all write-write pairs.
 7        for $j \leftarrow i + 1$ to $N_W$
 8           do $W_2 \leftarrow SS_{Wr}[j]$
 9              $LS_2 \leftarrow$ GET-WRITER-LOCK-SET($W_2$)
10              ASSERT($W_1 \not\preceq W_2$ and $W_2 \not\preceq W_1$)
11              if $LS_1 \cap LS_2 = \emptyset$
12                 then return $true$
13        $\triangleright$ Check all write-read pairs.
14        for $R \in SS_{Rd}$
15           do $LS_R \leftarrow$ GET-READER-LOCK-SET($R$)
16              if $W_1 \not\preceq R$ and $LS_1 \cap LS_R = \emptyset$
17                 then return $true$
18  return $false$
```

Our ultimate goal is the race-reporting routine. It prints the contexts of all memory accesses involved in a race and all locks that were held during each of the accesses. See appendix B for an example of output. Once a data race is reported on ID, we ignore the consequent accesses to ID.

```
REPORT-RACE($IsWrite, Tid, Seg, ID$)
 1  $(SS_{Wr}, SS_{Rd}) \leftarrow$ GET-PER-ID-STATE($ID$)
 2  PRINT( "Possible data race: " )
 3  PRINT( IsWrite ? "Write" : "Read" )
 4  PRINT( " at address " , $ID$)
 5  PRINT-CURRENT-CONTEXT($Tid$)
 6  PRINT-CURRENT-LOCK-SETS($Tid$)
 7  for $S \in SS_{Wr} \setminus Seg$
 8     do PRINT( "Concurrent writes: " )
 9        PRINT-SEGMENT-CONTEXT($S$)
10        PRINT-SEGMENT-LOCK-SETS($S$)
11  if not $IsWrite$
12     then return
13  for $S \in SS_{Rd} \setminus Seg$
14     do if $S \not\preceq Seg$
15           then PRINT( "Concurrent reads: " )
16                PRINT-SEGMENT-CONTEXT($S$)
17                PRINT-SEGMENT-LOCK-SETS($S$)
```

4.3 Segments and context

As defined in (4.1), the segment is a sequence of memory access events and the context of the segment is the context of its first event. Recording the segment contexts is critical because without them race reports will be less informative. ThreadSanitizer has three different modes[5] with regard to creation of segments:

1 (default): Segments are created each time the program enters a new super-block (single-entry multiple-exit region) of code. So, the contexts of all events in a segment belong to a small range of code, always within the same function. In practice, this means that the stack trace of the previous access is *nearly* precise: the line number of the topmost stack frame may be wrong, but all other line numbers and all function names in the stack traces are exact.

0 (fast): Segments are created only after synchronization events. This means that events inside a segment may have very different contexts and the context of the segment may be much different from the contexts of other events. When reporting a race in this mode, the contexts of the previous accesses are not printed. This mode is useful only for regression testing. For performance data see 7.2.

2 (precise, slow): Segments are created on each memory access (i.e. each segment contains just one event). This mode gives precise stack traces for all previous accesses, but is very slow. In practice, this level of precision is almost never required.

4.4 Variations of the state machine

The state machine described above is quite simple but flexible. With small modifications it can be used as a pure happens-before detector or else it can be enhanced with a special state similar to the *initialization state* described in [22]. ThreadSanitizer can use either of these modifications (adjustable by a command line flag).

4.4.1 Pure happens-before state machine

As any hybrid state machine, the state machine described above has false positives (see 6.4). It is possible to avoid most (but not all) false positives by using the pure happens-before mode.

Extended happens-before arc: a pair of events (X, Y) such that X is observed before Y and one of the following is true:

- $X = \text{WRUNLOCK}_{T_1}(L)$, $Y = \text{WRLOCK}_{T_2}(L)$.
- $X = \text{WRUNLOCK}_{T_1}(L)$, $Y = \text{RDLOCK}_{T_2}(L)$.
- $X = \text{RDUNLOCK}_{T_1}(L)$, $Y = \text{WRLOCK}_{T_2}(L)$.

[5]Controlled by the `--keep-history=[012]` command flag; Memcheck and Helgrind also have similar modes controlled by the flags `--track-origins=yes|no` and `--history-level=none|approx|full` respectively.

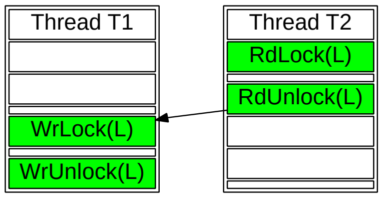

Figure 2: Extended happens-before arc.

- (X, Y) is a happens-before arc.

If we use the extended happens-before arc in the definition of happens-before relation, we will get the pure happens-before state machine similar to the one described in [9][6].

The following example explains the difference between pure happens-before and hybrid modes.

```
Thread1                 | Thread2
obj->UpdateMe();        | mu.Lock();
mu.Lock();              | bool f = flag;
flag = true;            | mu.Unlock();
mu.Unlock();            | if (f) obj->UpdateMe();
```

The first thread accesses an object without any lock and then sets the `flag` under a lock. The second thread checks the `flag` under a lock and then, if the `flag` is true, accesses the object again. The correctness of this code depends on the initial value of the `flag`. If it is false, the two accesses to the object are synchronized correctly; otherwise we have a race. ThreadSanitizer cannot distinguish between these two cases. In the hybrid mode, the tool will always report a data race on such code. In the pure happens-before mode, ThreadSanitizer will behave differently: if the race is real, the race may or may not be reported (depends on timing, this is why the pure happens-before mode is less predictable); if there is no race, the tool will be silent.

4.4.2 Fast-mode state machine

In most real programs, the majority of memory locations are never shared between threads. It is natural to optimize the race detector for this case. Such an optimization is implemented in ThreadSanitizer and is called **fast mode**[7].

Memory IDs in ThreadSanitizer are grouped into *cache lines*. Each cache line contains 64 IDs and the *Tid* of the thread which made the first access to this cache line. In fast mode, we ignore all accesses to a cache line until we see an access from another thread. This indeed makes the detection faster — according to our measurements it may increase the performance by up to 2x, see 7.2.

This optimization affects accuracy. Eraser [22] has the *initialization state* that reduces the number of false positives produced by the lock-set algorithm. Similarly, the ThreadSanitizer's fast mode reduces the number of false positives in the hybrid state machine. Both these techniques may as well hide real races.

The fast mode may be applied to the pure happens-before state machine, but we don't do this because the resulting detector will miss too many real races.

[6]Controlled by the `--pure-happens-before` command line flag.

[7]Controlled by the `--fast-mode` command line flag.

4.5 Comparison with other state machines

ThreadSanitizer and Eraser [22, 13] use locksets differently. In Eraser, the per-ID state stores the intersection of locksets. In ThreadSanitizer, the per-ID state contains original locksets (locksets are stored in segments, which are stored in segment sets and, hence, in the per-ID state) and lockset intersection is computed each time when we check for a race. This way we are able to report all locks involved in a race. Surprisingly enough, this extra computation adds only a negligible overhead.

This difference also allows our hybrid state machine to avoid a false report on the following code. The accesses in three different threads do not have any common lock, yet they are correctly synchronized[8].

```
Thread1          | Thread2          | Thread3
mu1.Lock();      | mu2.Lock();      | mu1.Lock();
mu2.Lock();      | mu3.Lock();      | mu3.Lock();
obj->Change();   | obj->Change();   | obj->Change();
mu2.Unlock();    | mu3.Unlock();    | mu3.Unlock();
mu1.Unlock();    | mu2.Unlock();    | mu1.Unlock();
```

ThreadSanitizer's pure happens-before mode finds the same races as the classical Lamport's detector [14] (we did not try to prove it formally though). On our set of unit tests [7], it behaves the same way as other pure happens-before detectors (see appendix A)[9]. The noticeable advantage of ThreadSanitizer in the pure happens-before mode is that it also reports all locks involved in a race — the classical pure happens-before detector knows nothing about locks and can't include them in the report.

5. DYNAMIC ANNOTATIONS

Any dynamic race detector must understand the synchronization mechanisms used by the tested program, otherwise the detector will not work. For programs that use only POSIX mutexes, it is quite possible to hard-code the knowledge about the POSIX API into the detector (most popular detectors do this). However, if the tested program uses other means of synchronization, we have to explain them to the detector. For this purpose we have created a set of *dynamic annotations* — a kind of race detection API.

Each dynamic annotation is a C macro definition. The macro definitions are expanded into some code which is later intercepted and interpreted by the tool[10]. You can find our implementation of the dynamic annotations at [7].

The most important annotations are:

- `ANNOTATE_HAPPENS_BEFORE(ptr)`,
- `ANNOTATE_HAPPENS_AFTER(ptr)`
  These annotations create, respectively, SIGNAL(*ptr*) and

[8]These cases are rare. During our experiments with Helgrind 3.3, which reported false positives on such code, we saw this situation only twice.

[9]It would be interesting to compare the accuracy of the detectors on real programs, but in our case it appeared to be too difficult. Other detectors either did not work with our OS and compiler or did not support our custom synchronization utilities and specific synchronization idioms (e.g. synchronization via I/O). Thus we have limited the comparison to the unit tests.

[10]Currently, the dynamic annotations are expanded into functions calls, but this is subject to change.

WAIT(ptr) events for the current thread; they are used to annotate cases where a hybrid algorithm may produce false reports, as well as to annotate lock-free synchronization. Examples are provided in section 6.4.

Other annotations include:

- `ANNOTATE_PURE_HAPPENS_BEFORE_MUTEX(lock)`
  Tells the detector to treat `lock` as in pure-happens-before mode (even if all other locks are handled as in hybrid mode). Using this annotation with the hybrid mode we can selectively apply pure hapens-before mode to some locks. In the pure happens-before mode this annotations is a no-op.
- `ANNOTATE_CONDVAR_LOCK_WAIT(cv,mu)`
  Creates a WAIT(cv) event that matches the `cv`.Signal() event (`cv` is a conditional variable, see 6.4.1).
- `ANNOTATE_BENIGN_RACE(ptr)`
  Tells that races on the address "`ptr`" are benign.
- `ANNOTATE_IGNORE_WRITES_BEGIN`,
- `ANNOTATE_IGNORE_WRITES_END`
  Tells the tool to ignore all writes between these two annotations. Similar annotations for reads also exist.
- `ANNOTATE_RWLOCK_CREATE(lock)`,
- `ANNOTATE_RWLOCK_DESTROY(lock)`,
- `ANNOTATE_RWLOCK_ACQUIRED(lock, isRW)`,
- `ANNOTATE_RWLOCK_RELEASED(lock, isRW)`
  Is used to annotate a custom implementation of a lock primitive.
- `ANNOTATE_PUBLISH_MEMORY_RANGE(ptr,size)`
  Reports that the bytes in the range $[ptr, ptr + size)$ are about to be published safely. The race detector will create a happens-before arc from this call to subsequent accesses to this memory. Usually required only for hybrid detectors.
- `ANNOTATE_UNPUBLISH_MEMORY_RANGE(ptr,size)`
  Opposite to `ANNOTATE_PUBLISH_MEMORY_RANGE`. Reports that the bytes in the range $[ptr, ptr + size)$ are not shared between threads any more and can be safely used by the current thread w/o synchronization. The race detector will create a happens-before arc from all previous accesses to this memory to this call. Usually required only for hybrid detectors.
- `ANNOTATE_NEW_MEMORY(ptr, size)`
  Tells that a new memory has been allocated by a custom allocator.
- `ANNOTATE_THREAD_NAME(name)`
  Tells the name of the current thread to the detector.
- `ANNOTATE_EXPECT_RACE(ptr)`
  Is used to write unit tests for the race detector.

With the dynamic annotations we can eliminate all false reports of the hybrid detector and hide benign races. As a result, ThreadSanitizer will find more races (as compared to a pure happens-before detector) but will not report false races.

6. RACE DETECTION IN PRACTICE

6.1 Performance

Performance is critical for the successful use of race detectors, especially in large organizations like Google. First, if a detector is too slow, it will be inconvenient to use it for manual testing or debugging. Second, slower detector will require more machine resources for regular testing, and the machine resources cost money. Third, most of the C++ applications at Google are time-sensitive and will simply fail due to protocol timeouts if slowed down too much.

When we first tried Helgrind 3.3 on a large set of unit tests, almost 90% of them failed due to slowdown. With our improved variant of Helgrind and, later, with ThreadSanitizer we were able to achieve more than 95% pass rate. In order to make the remaining tests pass, we had to change various timeout values in the tests[11].

On an average Google unit test or application the slowdown is 20-50 times, but in extreme cases the slowdown could be as high as 10000 times (as for example an artificial stress test for a race detector) or as low as 2 times (the test mostly sleeps or waits for I/O). See also section 7.2.

ThreadSanitizer spends almost all the time intercepting and analyzing memory accesses. If a given memory location has been accessed by just one thread, the analysis is fast (especially in the fast mode, see section 4.4.2). If a memory location has been accessed by many threads and there have been a lot of synchronization events, the analysis is slow.

So, there are two major ways to speed up the tool: make the analysis of one memory access faster and analyze fewer memory accesses. In order to make the analysis of one access faster, we used various well known techniques and algorithms such as vector time-stamps ([9]) and caching. We also limited the size of a segment set with a small constant (currently, 4) to avoid huge slowdowns in corner cases. But whatever we do to speed up the analysis, the overhead will always remain significant: remember that we replace a memory access instruction with a call to a quite sophisticated function that usually runs for few hundreds of CPU cycles.

A much more attractive approach is to reduce the number of analyzed memory accesses. For example, ThreadSanitizer does not instrument the internals of the threading library (there is no sense in analysing races on internal representation of a mutex). The tool also supports a mechanism to ignore parts of the program marked as safe by the user[12]. In some cases this allows to speed up the run by 2-3 times by ignoring a single hot spot.

Adaptive instrumentation [16] seems promising. We also plan to use static analysis performed by a compiler to skip instrumentation when we can prove thread-safety statically.

Another way to reduce the number of analyzed memory accesses is to run the tool on an optimized binary. Unfortunately, the current implementation does not work well with fully optimized code (e.g. `gcc -O2`)[13], but the following gcc flags give 50%-100% speedup compared to a non-optimized

[11]This had a nice side effect. Many of the tests that were failing regularly under ThreadSanitizer were known to be flaky (they were sometimes failing when running natively) and ThreadSanitizer helped to find the reason of that flakiness just by making tests slower.

[12]Controlled by the `--ignore` command line flag.

[13]This is a limitation of both gcc, Valgrind and ThreadSanitizer.

compilation while maintaining the same level of usability: `gcc -O1 -g -fno-inline -fno-omit-frame-pointer -fno-builtin`[14].

6.2 Memory consumption

The memory consumption of ThreadSanitizer consists mostly of the following overhead:

- A constant size buffer that stores segments, including stack traces. By default, there are 2^{23} segments and each occupies ≈100 bytes (≈50 bytes in 32-bit mode). So, the buffer is ≈800M. Decreasing this size may lead to loosing some data races. If we are not tracking the contexts of previous accesses (see 4.3), the segments occupy much less memory (≈250M).

- Vector time clocks attached to each segment. This memory is limited by the number of threads times the number of segments, but in most cases it is quite small.

- Per-ID state. In the fast mode, the memory required for per-ID state linearly depends on the amount of memory shared between more than one thread. In the full hybrid and in the pure happens-before modes, the footprint is a linear function of all memory in the program. However, these are the worst case assumptions and in practice a simple compression technique reduces the memory usage significantly.

- Segment sets and locksets may potentially occupy arbitrary large amount of memory, but in reality they constitute only a small fraction of the overhead.

All these objects are automatically recycled when applicable.

On an average Google unit test the memory overhead is within 3x-4x (compared to a native run). Obviously, a test will fail under ThreadSanitizer if there is not enough RAM in the machine. Almost all unit tests we have tried require less than 4G when running under ThreadSanitizer. Real applications may require 8G and more. See also 7.2 for the actual numbers.

6.2.1 Flushing state

Even though the memory overhead of ThreadSanitizer is sane on average, there are cases when the tool would consume all the memory it could get. In order to stay robust, ThreadSanitizer flushes all its internal state when the memory overhead is above a certain limit (supplied by the user or derived from `ulimit`) or when the tool has used all available segments and none of them can be recycled. Obviously, if a flush happens between two memory accesses which race with each other, such a race will be missed, but the probability of such situation is low.

6.3 Common real races

In this section we will show the examples of the most frequent races found in our C++ code. The detailed analysis of some of these races is given at [7].

6.3.1 Simple race

The simplest possible data race is the most frequent one: two threads are accessing a variable of a built-in type without any synchronization. Quite frequently, such races are benign (the code counts some statistic that is allowed to be imprecise). But sometimes such races are extremely harmful (e.g. see 7.1).

```
   Thread1      | Thread2
int v; ...      |
v++;            |  v++;
```

6.3.2 Race on a complex type

Another popular race happens when two threads access a non-thread-safe complex object (e.g. an STL container) without synchronization. These are almost always dangerous.

```
          Thread1          |      Thread2
std::map<int,int> m; ...   |
m[123] = 1;                | m[345] = 0;
```

6.3.3 Notification

A data race occurs when a boolean or an integer variable is used to send notifications between threads. This may work correctly with some combination of compiler and hardware, but for portability we do not recommend programmers to assume implicit semantics of the target architecture.

```
        Thread1          |     Thread2
bool done = false; ...   |
                         |
while (!done)            | done = true;
  sleep(1);              |
```

6.3.4 Publishing objects without synchronization

One thread initializes an object pointer (which was initially null) with a new value, another thread spins until the object pointer becomes non-null. Without proper synchronization, the compiler may do surprising transformations (code motion) with such code which will lead to (occasional) failures. In addition to that, on some architectures this race may cause failures due to cache-related effects.

```
      Thread1        |       Thread2
MyObj* obj = NULL;   | while(obj == NULL)
...                  |   yield();
obj = new MyObj();   | obj->DoSomething();
```

6.3.5 Initializing objects without synchronization

```
static MyObj *obj = NULL;
void InitObj() {
  if (!obj)
    obj = new MyObj();
}
```

```
   Thread1   |  Thread2
InitObj();   | InitObj();
```

This may lead e.g. to memory leaks (the object may be constructed twice).

6.3.6 Write during a ReaderLock

Updates happening under a reader lock.

```
      Thread1        |       Thread2
mu.ReaderLock();     | mu.ReaderLock();
var++;               | var++;
mu.ReaderUnlock();   | mu.ReaderUnlock();
```

[14]This applies to gcc 4.4 on x86_64.

6.3.7 *Adjacent bit fields*

The code below looks correct at the first glance. But if `x` is “`struct { int a:4, b:4; }`”, we have a bug.

```
Thread1 | Thread2
x.a++;  | x.b++;
```

6.3.8 *Double-checked locking*

The so called doubled-checked locking is well known to be an anti-pattern ([17]), but we still find it occasionally (mostly in the old code).

```
bool inited = false;
void Init() {
  // May be called by multiple threads.
  if (!inited) {
    mu.Lock();
    if (!inited) {
      // .. initialize something
    }
    inited = true;
    mu.Unlock();
  }
}
```

6.3.9 *Race during destruction*

Sometimes objects are created on the stack, passed to another thread and then destroyed without waiting for the second thread to finish its work.

```
void Thread1() {
  SomeType object;
  ExecuteCallbackInThread2(
    SomeCallback, &object);
  ...
  // "object" is destroyed when
  // leaving its scope.
}
```

6.3.10 *Race on vptr*

Class `A` has a function `Done()`, virtual function `F()` and a virtual destructor. The destructor waits for the event generated by `Done()`. There is also a class `B`, which inherits `A` and overrides `A::F()`.

```
class A {                     |
 public:                      |
  A() {                       |
   sem_init(&sem_, 0, 0);     |
  }                           | class B : public A {
  virtual void F() {          |  public:
   printf ("A::F\n");         |   virtual void F() {
  }                           |    printf("B::F\n");
  void Done() {               |   }
   sem_post(&sem_);           |   virtual ~B() { }
  }                           | };
  virtual ~A() {              |
   sem_wait(&sem_);           | static A *obj =
   sem_destroy(&sem_);        |   new B;
  }                           |
 private:                     |
  sem_t sem_;                 |
};                            |
```

An object `obj` of static type `A` and dynamic type `B` is created. One thread executes `obj->F()` and then signals to the second thread. The second thread calls `delete obj` (i.e. `B::~B`) which then calls `A::~A`, which, in turn, waits for the signal from the first thread. The destructor `A::~A` overwrites the vptr (pointer to virtual function table) to `A::vptr`. So, if the first thread executes `obj->F()` after the second thread started executing `A::~A`, then `A::F` will be called instead of `B::F`.

```
Thread1       | Thread2
obj->F();     |
obj->Done();  | delete obj;
```

6.4 Common false positives

Here we show the three most common types of false positives, i.e. the situations where the code is correctly synchronized, but ThreadSanitizer will report a race. The annotations given in the code examples explain the synchronization to the tool; with these annotations no reports will appear.

6.4.1 *Condition variable*

```
Thread1            | Thread2
                   | mu.Lock();
obj->UpdateMe();   | while(!c)
mu.Lock();         |   cv.Wait(&mu);
c = true;          | ANNOTATE_CONDVAR_LOCK_WAIT(
cv.Signal();       |   &cv, &mu);
mu.Unlock();       | mu.Unlock();
                   | obj->UpdateMe();
```

This is a typical usage of a condition variable [12]: the two accesses to `obj` are serialized. Unfortunately, it may be misunderstood by the hybrid detector. For example, Thread1 may set the condition to ”true” and leave the critical section before Thread2 enters the critical section for the first time and blocks on the condition variable. The condition of the `while(!c)` loop will never be true and `cv.Wait()` method won’t be called. As a result, the happens-before dependency will be missed.

6.4.2 *Message queue*

Some message queues may also be unfriendly to the hybrid detector.

```
class Queue {
 public:
  void Put(int* ptr) {
    mu_.Lock();
    queue_.push_back(ptr);
    ANNOTATE_HAPPENS_BEFORE(ptr);
    mu_.Unlock();
  }
  int* Get() {
    int *res = NULL;
    mu_.Lock();
    if (!queue_.empty()) {
      res = queue_.front();
      ANNOTATE_HAPPENS_AFTER(res);
      queue_.pop_front();
    }
    mu_.Unlock();
    return res;
  }
 private:
  std::queue queue_;
  Mutex mu_;
};
```

The queue implementation above does not use any happens-before synchronization mechanism but it does actually create a happens-before dependency between Put() and Get().

```
          Thread1           |          Thread2
                            |
                            |
     *ptr = ...;            |
     queue.Put(ptr);        | ptr = queue.Get();
                            | if (ptr)
                            |   *ptr = ...;
```

A message queue may be implemented via atomic operations (i.e. without any Mutex). In this case even a pure happens-before detector may report false positives.

6.4.3 *Reference counting*

Another frequent cause of false positives is reference counting. As with message queues, mutex-based reference counting will result in false positives in the hybrid mode, while a reference counting implemented via atomics will confuse even the pure happens-before mode. And again, the annotations allow the tool to understand the synchronization.

```
class SomeReferenceCountedClass {
 public:
  void Unref() {
    ANNOTATE_HAPPENS_BEFORE(&ref_);
    if (AtomicIncrement(&ref_, -1) == 0) {
      ANNOTATE_HAPPENS_AFTER(&ref_);
      delete this;
    }
  }  ...
 private:
  int ref_;
}
```

6.5 General advice

Applying a data race detector to an arbitrary C++ program may be arbitrarily hard. However, if the developers follow several simple rules, race detectors can be used at full power. Here we summarize the recommendations we give to C++ developers at Google.

First of all, variables shared between threads are best protected by a mutex. Always use mutex unless you know for sure that it causes a significant performance loss.

When possible, try to reuse the existing standard synchronization primitives (e.g. message queues, reference counting utilities, etc) instead of re-inventing the wheel. If you really need your own synchronization mechanism, annotate it with dynamic annotations (section 5).

Avoid using condition variables directly as they are not friendly to hybrid detectors. Instead, wrap the condition loop `while(!c) cv.Wait(&mu)` into a separate function and annotate it (6.4.1). In Google's internal C++ library such function is a part of the Mutex API.

Try not to use atomic operations directly. Instead, wrap the atomic operations into functions or classes that implement certain synchronization patterns.

Remember that dynamic data race detection (as well as most other kinds of dynamic analysis) is slow. Do not hardcode any timeout values into your program. Instead, allow the timeout values to be changed via command line flags, environment variables or configuration files.

Never use `sleep()` as synchronization between threads, even in unit tests.

Don't over-synchronize. Excessive synchronization may be just as incorrect as no synchronization at all, but it may hide real races from data race detectors.

6.5.1 *Choosing the mode*

Which of the three modes of ThreadSanitizer should one choose?

If you are testing an existing software project, we suggest you to start with the pure happens-before mode (4.4.1). Unless you have lock-free synchronization (which you will have to annotate), every reported race will be real.

Once you fixed all reports from the pure happens-before mode (or if you are starting a new project), switch to the fast mode (4.4.2). You may see few false reports (6.4), which can be easily eliminated. If your aim is to find the maximal number of bugs and agree to spend some more time for annotations, use the full hybrid mode (4.2).

For regression testing prefer the hybrid mode (either full or fast) because it is more predictable. It is often the case that a race is detected only on one of 10-100 runs by the pure happens-before mode, while the hybrid mode finds it in each run.

7. RACE DETECTION FOR CHROMIUM

One of the applications we test with ThreadSanitizer is Chromium [1], an open-source browser project.

The code of Chromium browser is covered by a large number of tests including unit tests, integration tests and interactive tests running the real application. All these tests are continuously run on a large number of test machines with different operating systems. Some of these machines run tests under Memcheck (the Valgrind tool which finds memory-related errors, see [8]) and ThreadSanitizer. When a new error (either a test failure or a race report from ThreadSanitizer) is found after a commit to the repository, the committer of the change is notified. These reports are available for other developers and maintainers as well.

We have found and fixed a few dozen data races in Chromium itself, and in some third party components used by this project. You may find all these bugs by searching for `label:ThreadSanitizer` at `www.crbug.com`.

7.1 Top crasher

One of the first data races we found in Chromium happened to be the cause of a serious bug, which had been observed for several months but had not been understood nor fixed[15]. The data race happened on a class called `RefCounted`. The reference counter was incremented and decremented from multiple threads without synchronization. When the race actually occurred (which happened very rarely), the value of the counter became incorrect. This resulted in either a memory leak or in two calls of `delete` on the same memory. In the latter case, the internals of the memory allocator were corrupted and one of the subsequent calls to `malloc` failed with a segmentation fault.

The cause of these failures was not understood for a long time because the failure never happened during debugging, and the failure stack traces were in a different place. ThreadSanitizer found this data race in a single run.

The fix for this data race was simple. Instead of the `RefCounted` class we needed to use `RefCountedThreadSafe`, the class which implements reference counting using atomic instructions.

[15]See the bug entries http://crbug.com/18488 and http://crbug.com/15577 describing the race and the crashes, respectively.

Table 1: Time and space overhead compared to Helgrind and Memcheck on Chromium tests. The performance of ThreadSanitizer is close to Memcheck. On large tests (e.g. *unit*), ThreadSanitizer can be twice as fast as Helgrind. The memory consumption is also comparable to Memcheck and Helgrind.

	app		base		ipc		net		unit	
native	3s	172M	77s	1811M	5s	325M	50s	808M	43s	914M
Memcheck-no-hist	6.7x	2.0x	1.7x	1.1x	5.2x	1.1x	3.0x	1.6x	14.8x	1.7x
Memcheck	10.5x	2.6x	2.2x	1.1x	8.2x	1.2x	5.1x	2.3x	29.7x	1.9x
Helgrind-no-hist	13.9x	2.7x	1.8x	1.8x	5.4x	1.5x	4.5x	2.2x	48.7x	3.4x
Helgrind	14.9x	3.8x	1.7x	1.9x	6.7x	1.7x	11.9x	2.5x	62.3x	3.8x
TS-fast-no-hist	6.2x	4.2x	2.2x	1.2x	11.1x	1.8x	3.9x	1.7x	19.2x	2.2x
TS-fast	7.9x	7.6x	2.4x	1.5x	12.0x	3.6x	4.7x	2.4x	21.6x	2.8x
TS-full-no-hist	8.4x	4.2x	2.4x	1.2x	11.3x	1.8x	4.7x	1.6x	22.3x	2.3x
TS-full	13.8x	7.4x	2.8x	1.5x	11.9x	3.6x	6.3x	2.3x	28.6x	2.5x
TS-phb-no-hist	8.3x	4.2x	2.8x	1.2x	11.2x	1.8x	4.7x	1.8x	23.0x	6.2x
TS-phb	14.2x	7.4x	2.6x	1.5x	11.8x	3.6x	6.2x	2.3x	28.6x	2.5x

7.2 Performance evaluation on Chromium

We used Chromium unit tests for performance evaluation of ThreadSanitizer. We compared our tool with Helgrind and Memcheck 3.5.0 [8]. Even though Memcheck is not a race detector, it performs similar instrumentation; this tool is well known for its high quality and practical usefulness.

Table 1 gives the summary of the results. ThreadSanitizer was run in three modes: `--pure-happens-before=yes` (phb), `--fast-mode=yes` (fast) and `--fast-mode=no` (full). Similarly to ThreadSanitizer, Helgrind and Memcheck have modes where the history of previous accesses is not tracked (4.3). In the table, such modes are marked with *no-hist*. The tests were built using `gcc -O1 -g -fno-inline -fno-omit-frame-pointer -fno-builtin` flags for the x86_64 platform and run on Intel Core 2 Duo Q6600 with 8Gb or RAM.

As may be seen from Table 1, the performance of ThreadSanitizer is close to Memcheck. The average slowdown compared to the native run is less than 30x. On large tests like *unit*, ThreadSanitizer can be twice as fast as Helgrind.

The memory consumption is also comparable to Memcheck and Helgrind. ThreadSanitizer allocates a large constant size buffer of segments (see 6.2), hence on small tests it consumes more memory than other tools.

ThreadSanitizer flushes its state (see 6.2.1) 90 times on *unit*, 34 times on *net* and 4 times on *base* test sets when running in the full or pure happens-before modes with history tracking enabled. In the fast mode and with disabled history tracking ThreadSanitizer never flushes its state on these tests.

8. CONCLUSIONS

In this paper we have presented ThreadSanitizer, a dynamic detector of data races. ThreadSanitizer uses a new algorithm; it has several modes of operation, ranging from the most conservative mode (which has few false positives but also misses real races) to a very aggressive one (which has more false positives but detects the largest number of real races). To the best of our knowledge ThreadSanitizer has the most detailed output and it is the only dynamic race detector with hybrid *and* pure happens-before modes.

We have introduced the dynamic annotations, a sort of API for a race detector. Using the dynamic annotations together with the most aggressive mode of ThreadSanitizer enables us to find the largest number of real races while keeping zero noise level (no false positives or benign races are reported).

ThreadSanitizer is heavily used at Google for testing various C++ applications, including Chromium. In this paper we discussed a number of practical issues which we have faced while deploying ThreadSanitizer.

We believe that our ThreadSanitizer has noticeable advantages over other dynamic race detectors in terms of practical use. The current implementation of ThreadSanitizer is built on top of the Valgrind binary translation framework and it can be used to test C/C++ programs on Linux and Mac. The source code of ThreadSanitizer is published under the GPL license and can be downloaded at [7].

9. ACKNOWLEDGMENTS

We would like to thank Mike Burrows, the co-author of Eraser [22], for his great support of our project at Google and for many algorithmic suggestions, and Julian Seward, the author of Valgrind and Helgrind [8, 19], for his amazing tools and fruitful discussions.

10. REFERENCES

[1] Chromium project. http://dev.chromium.org.

[2] Intel Parallel Studio. http://software.intel.com/en-us/intel-parallel-studio-home.

[3] Intel Thread Checker. http://software.intel.com/en-us/intel-thread-checker.

[4] Multi-Thread Run-time Analysis Tool for Java. http://www.alphaworks.ibm.com/tech/mtrat.

[5] Pin - a dynamic binary instrumentation tool. http://www.pintool.org.

[6] Sun Studio. http://developers.sun.com/sunstudio.

[7] ThreadSanitizer project: documentation, source code, dynamic annotations, unit tests. http://code.google.com/p/data-race-test.

[8] Valgrind project. Home of Memcheck, Helgrind and DRD. http://www.valgrind.org.

[9] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A theory of data race detection. In *PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 69–78, New York, NY, USA, 2006. ACM.

[10] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. Unraveling Data Race Detection in the Intel® Thread

Checker. In *First Workshop on Software Tools for Multi-core Systems (STMCS), in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO), March*, volume 26, 2006.

[11] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM.

[12] F. Garcia and J. Fernandez. Posix thread libraries. *Linux J.*, 70es (Feb. 2000):36, 2000.

[13] J. J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 331–342, London, UK, 2000. Springer-Verlag.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[16] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 134–143, New York, NY, USA, 2009. ACM.

[17] S. Meyers and A. Alexandrescu. C++ and the Perils of Double-Checked Locking: Part I. *DOCTOR DOBBS JOURNAL*, 29:46–49, 2004.

[18] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–31, New York, NY, USA, 2007. ACM.

[19] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[20] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[21] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, New York, NY, USA, 2003. ACM.

[22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

APPENDIX

A. OTHER RACE DETECTORS

Here we briefly describe some of the race detectors available for download.

Helgrind is a tool based on Valgrind [19, 8]. Helgrind 3.5 is a pure happens-before detector; it supports a subset of dynamic annotations described in section 5. Part of ThreadSanitizer's instrumentation code is derived from Helgrind. **DRD** is one more Valgrind-based race detector with similar properties.

Intel Thread Checker [3, 10, 9] is a pure happens-before race detector. It supports an analog of dynamic annotations (a subset). It works on Linux and Windows. Thread Checker's latest reincarnation is called **Intel Parallel Inspector** [2] and is based on PIN [15, 5]. As of November 2009, the Parallel Inspector is available only for Windows.

Sun Thread Analyzer, a part of Sun Studio [6]. A hybrid race detector. It supports an analog of dynamic annotations (a small subset). It works only together with the Sun Studio compiler (so we did not try it for our real tasks).

IBM MTRAT [4] is a race detector for Java. It uses some variant of hybrid state machine and does not support any annotations. As of the version from March 2009, the noise level seems to be rather high.

B. EXAMPLE OF OUTPUT

Here we give a simple test case where a wrong mutex is used in one place. For more examples refer to [7].

```
Mutex mu1;  // This Mutex guards var.
Mutex mu2;  // This Mutex is not related to var.
int   var;
// Runs in thread named 'test-thread-1'
void Thread1() {
  mu1.Lock();  // Correct Mutex.
  var = 1;
  mu1.Unlock();
}
// Runs in thread named 'test-thread-2'
void Thread2() {
  mu2.Lock();  // Wrong Mutex.
  var = 2;
  mu2.Unlock();
}
```

The output of ThreadSanitizer will contain stack traces for both memory accesses, names of both threads, information about locks held during each access and the description of the memory location.

```
WARNING: Possible data race during write of size 4
   T2 (test-thread-2) (locks held: {L134}):
    #0  Thread2() racecheck_unittest.cc:7034
    #1  MyThread::ThreadBody(MyThread*) ...
  Concurrent write(s) happened at these points:
   T1 (test-thread-1) (locks held: {L133}):
    #0  Thread1() racecheck_unittest.cc:7029
    #1  MyThread::ThreadBody(MyThread*) ...
  Address 0x63F260 is 0 bytes inside data symbol "var"
  Locks involved in this report:  {L133, L134}
   L133
    #0  Mutex::Lock() ...
    #1  Thread1() racecheck_unittest.cc:7028 ...
   L134
    #0  Mutex::Lock() ...
    #1  Thread2() racecheck_unittest.cc:7033 ...
```