Dynamic Race Detection with LLVM compiler

Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, Dmitriy Vyukov

{kcc,glider,timurrrr,dvyukov}@google.com

ThreadSanitizer

WBIA'09: **ThreadSanitizer – data race detection in practice**

Initially implemented using Valgrind.

Found a number of errors in Google code, including the Chromium browser.

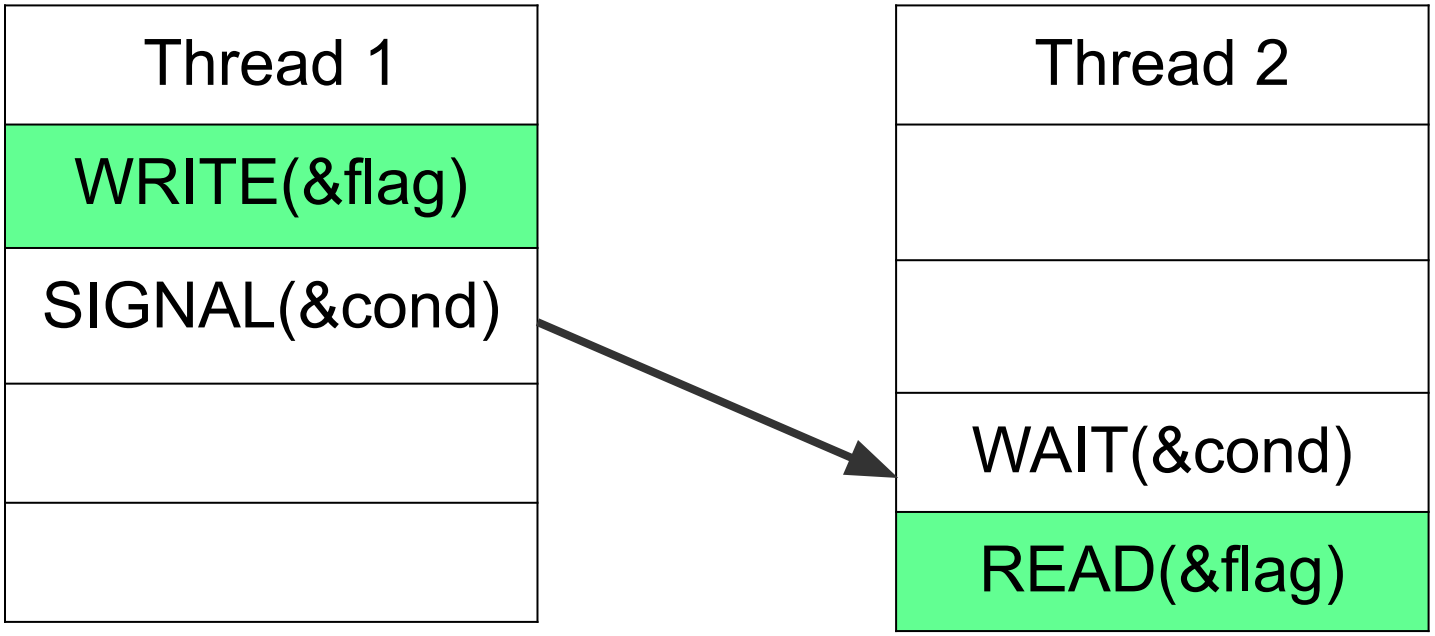
ThreadSanitizer algorithm

A state machine for processing program events:

- memory accesses (READ, WRITE)
- locking events (LOCK, UNLOCK)
- happens-before events (SIGNAL, WAIT)

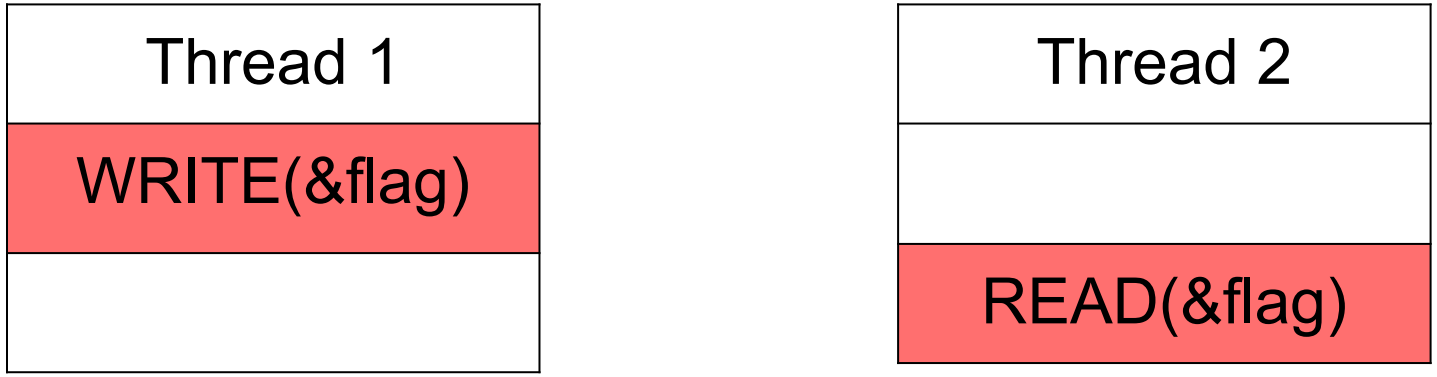
The program execution is split into segments – sequences of memory accesses delimited by synchronization (locking and happens-before) events.

HB relation: ordered accesses



The happens-before relation is transitive and extends to segments as well as the program events.

HB relation: concurrent accesses



Two events are concurrent when they are not ordered by the happens-before relation.

Locksets: correct locking

Thread 1
LOCK(&l)
WRITE(&flag)
UNLOCK(&l)

Thread 2
LOCK(&l)
READ(&flag)
UNLOCK(&l)

Locksets: incorrect locking

Thread 1
LOCK(&L)
WRITE(&flag)
UNLOCK(&L)

Thread 2
LOCK(&L)
UNLOCK(&L)
READ(&flag)

ThreadSanitizer algorithm (continued)

For each memory location we keep two sets of concurrent segments that read and wrote to that memory.

For each segment we track the happens-before relation and the set of locks (to refine the race reports).

Additional program events

- Memory allocation (MALLOC, FREE)
- Call stack information (RTN_CALL, RTN_EXIT, SBLOCK_ENTER)
- Threading events (THR_START, THR_END, THR_JOIN)

Example of a ThreadSanitizer report

```
INFO: T1 has been created by T0. T2 has been created by T0.
 WARNING: Possible data race during write of size 4 at 0x6457E0:
   T2 (test-thread-2) ():
     #0  RaceReportDemoTest::Thread2 demo_tests.cc:53
     #1  MyThread::ThreadBody thread_wrappers_pthread.h:341
  Concurrent write(s) happened at (OR AFTER) these points:
   T1 (test-thread-1) (L{L3}):
     #0  RaceReportDemoTest::Thread1 demo_tests.cc:48
     #1  MyThread::ThreadBody thread_wrappers_pthread.h:341
  Address 0x6457E0 is 0 bytes inside data symbol "demo_var"
  Locks involved in this report (reporting last lock sites): {L3}
   L3 (0x645780)
     #0  RaceReportDemoTest::Thread1 demo_tests.cc:47
```

Problems with TSan-Valgrind

- Great slowdowns because of the translation overhead of Valgrind.
- Great slowdowns because Valgrind is single-threaded.
- Overall: 5x–50x slowdown depending on the test.

Advantages of compile-time instrumentation

- Instrumentation is done on the LLVM IR level – the code is independent of the target platform and the programming language.
- Instrumentation code is optimized together with the program code.
- Multithreading is supported out of the box.
- You can actually test ThreadSanitizer using ThreadSanitizer!

How does this work?

1. Compile each module using a modified LLVM compiler that instruments the code.
2. Link the program with the ThreadSanitizer runtime library.

Instrumentation

- For each module:
  - Insert the declarations of runtime functions;
  - Instrument the functions.

Instrumentation (2)

- For each function:
  - Build the superblocks and their passports;
  - Insert the code that updates the shadow call stack on entry / each call instruction / each return instruction;
  - Instrument the superblocks.

Instrumentation (3)

- For each superblock:
  - For each memory access:
    - Insert the code that records the address into the thread-local event buffer, or TLEB;
  - Instrument each exit with a call to `FlushTLEB(&passport)`.

Runtime library

- Sets up the race detector;
- Wraps some functions called by the client:
  - synchronization routines from libpthread;
  - memory allocation routines;
  - libc functions that imply happens-before relation;
- Provides entry points for the instrumented code.

Sampling: Literace (Marino et al., PLDI'09)

An idea of adapting the sampling rate for a code region (a function) depending on the number of its executions.

Cold-region hypothesis: *data races are likely to occur when a thread is executing a “cold” (infrequently accessed) region in the program. Data races that occur in hot regions of well-tested programs either have already been found and fixed, or are likely to be benign.*

Sampling in ThreadSanitizer

Sampling is done per basic block or superblock (Literace samples functions).

Changed the way the sampling rate is calculated.

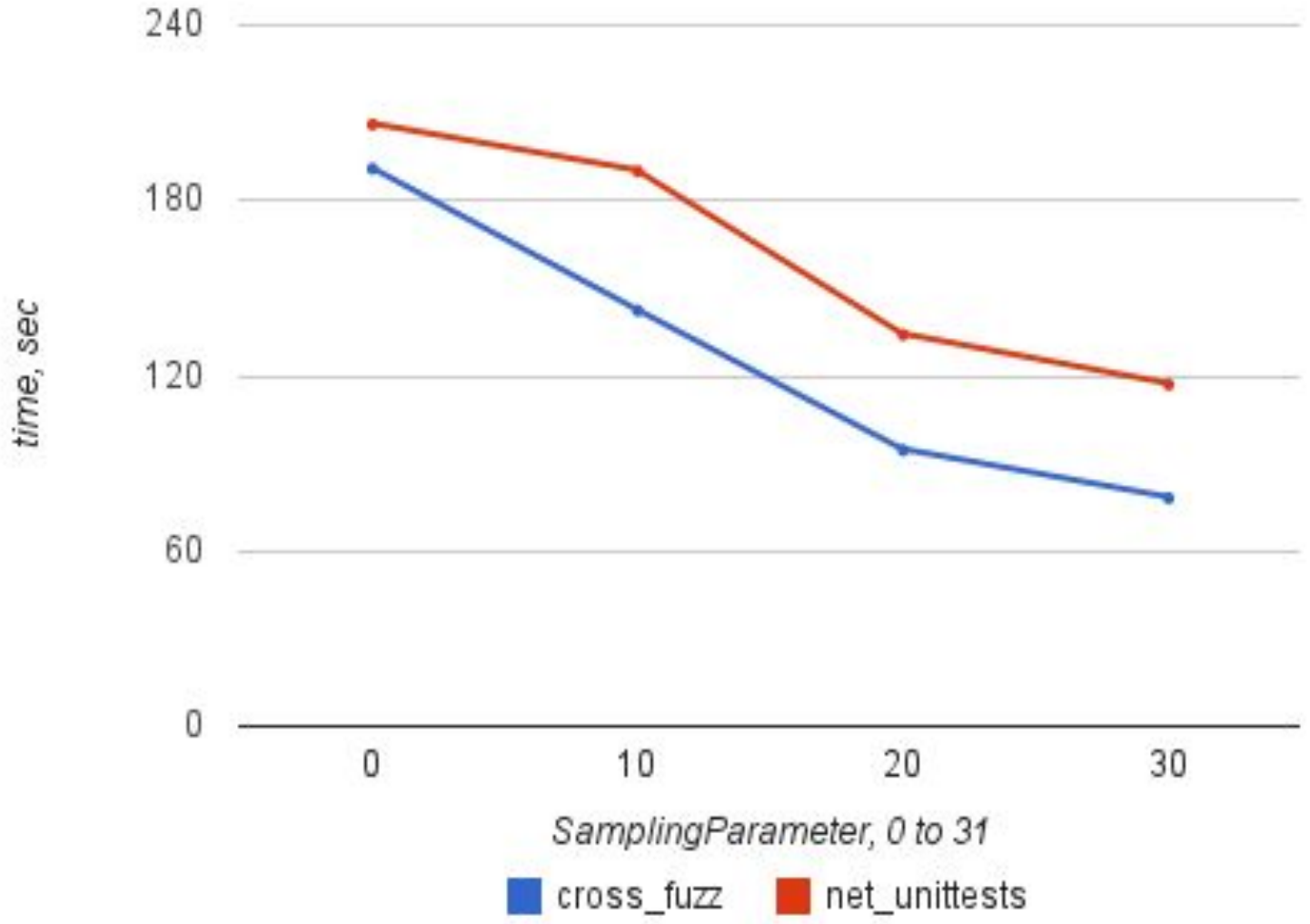
Memory access addresses are always recorded, but not always flushed (Literace branches between instrumented and non-instrumented code).

Sampling (continued)

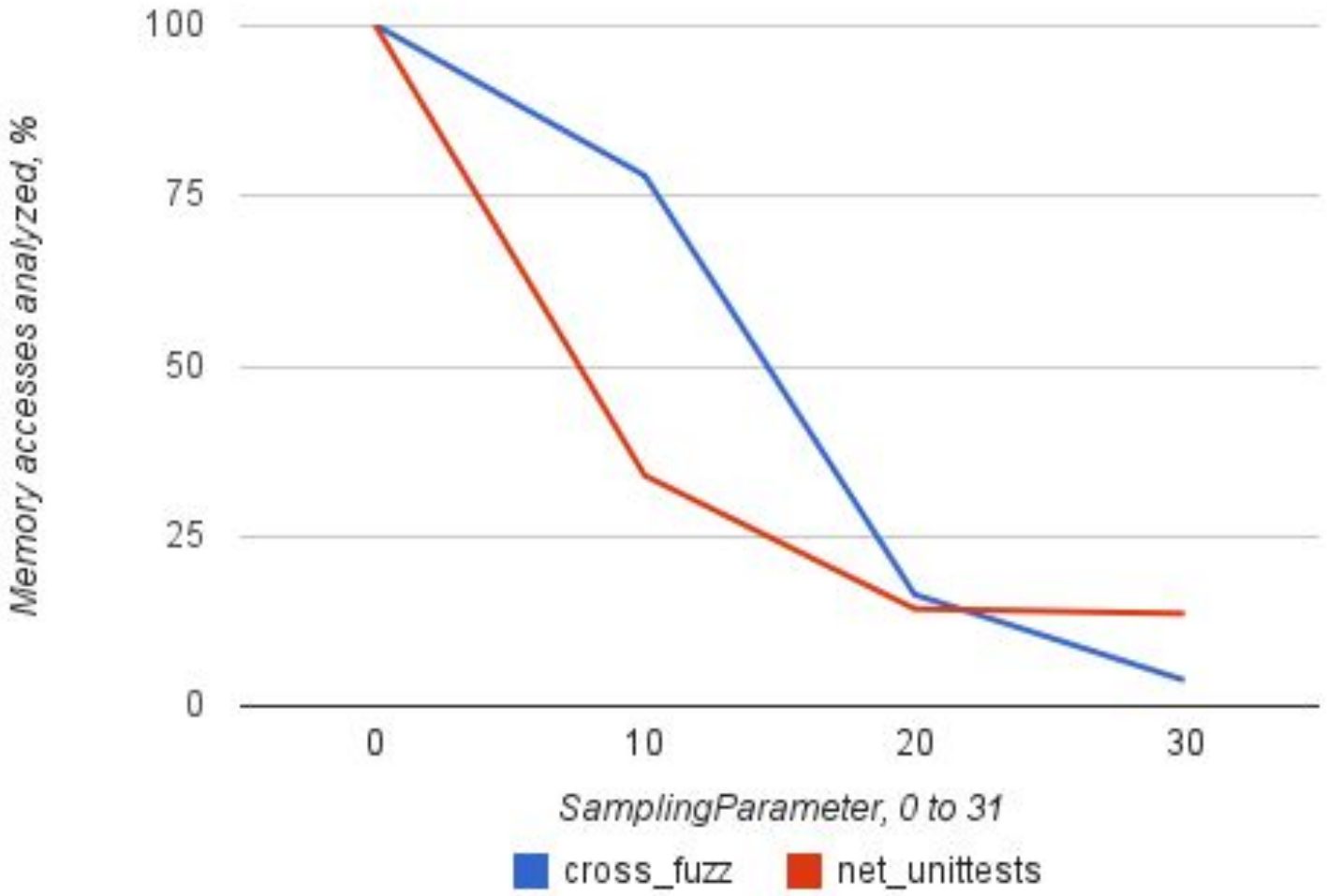
Each time a block is executed, the execution counter for that block is updated.

```
Do-Sampling(Skip, Total)
1  Skip ← Skip − 1
2  if Skip = 0
3    then ▷ Flush the TLEB and update the counters
4         FlushTleb
5         Skip ← (Total >> (32 − SamplingParameter)) + 1
6         Total ← Total + Skip
```

Sampling for Chromium tests



Sampling for Chromium tests (2)



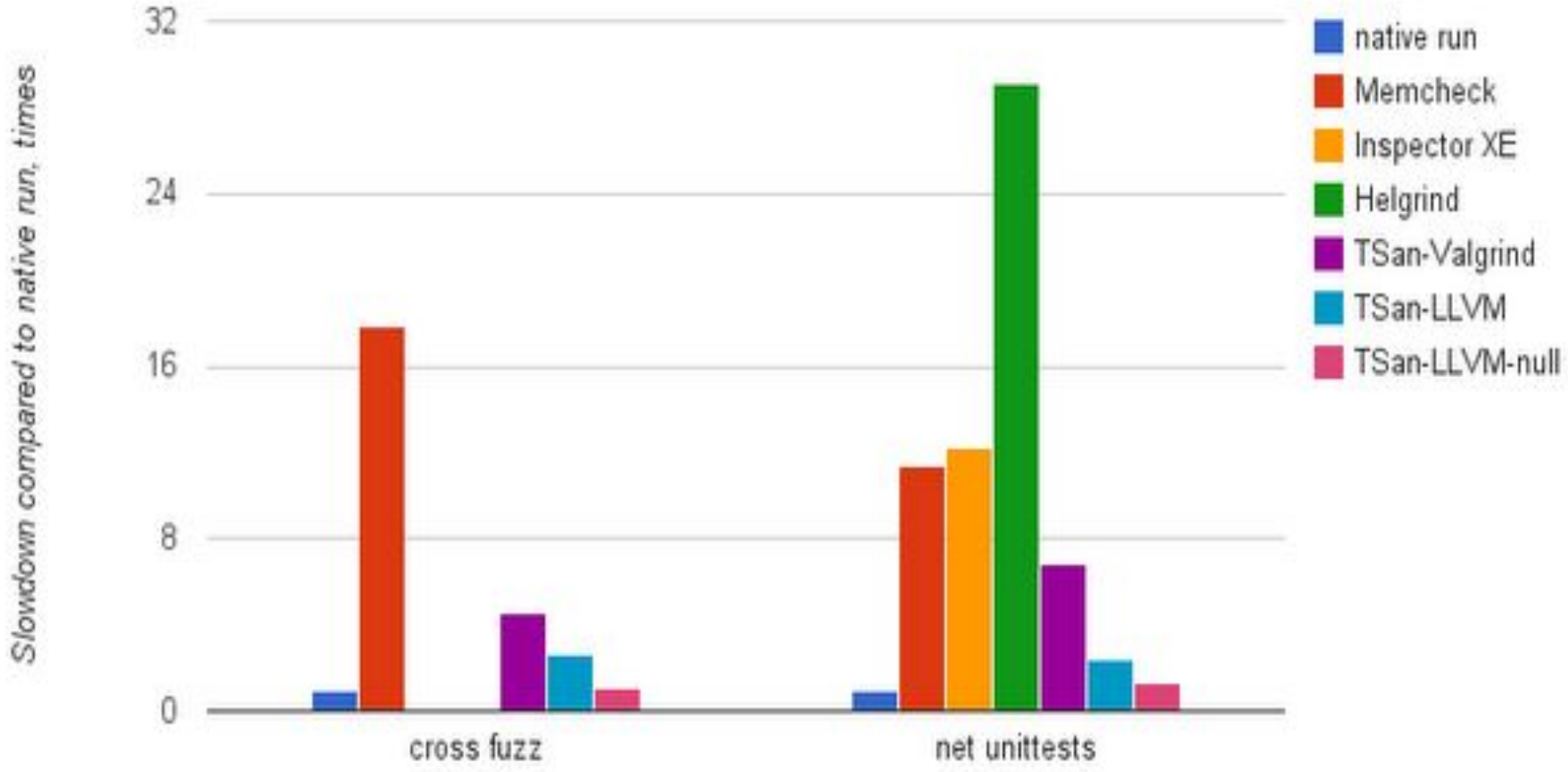
Results on Chromium tests

1.7x–2.9x speedup over the Valgrind-based ThreadSanitizer implementation;

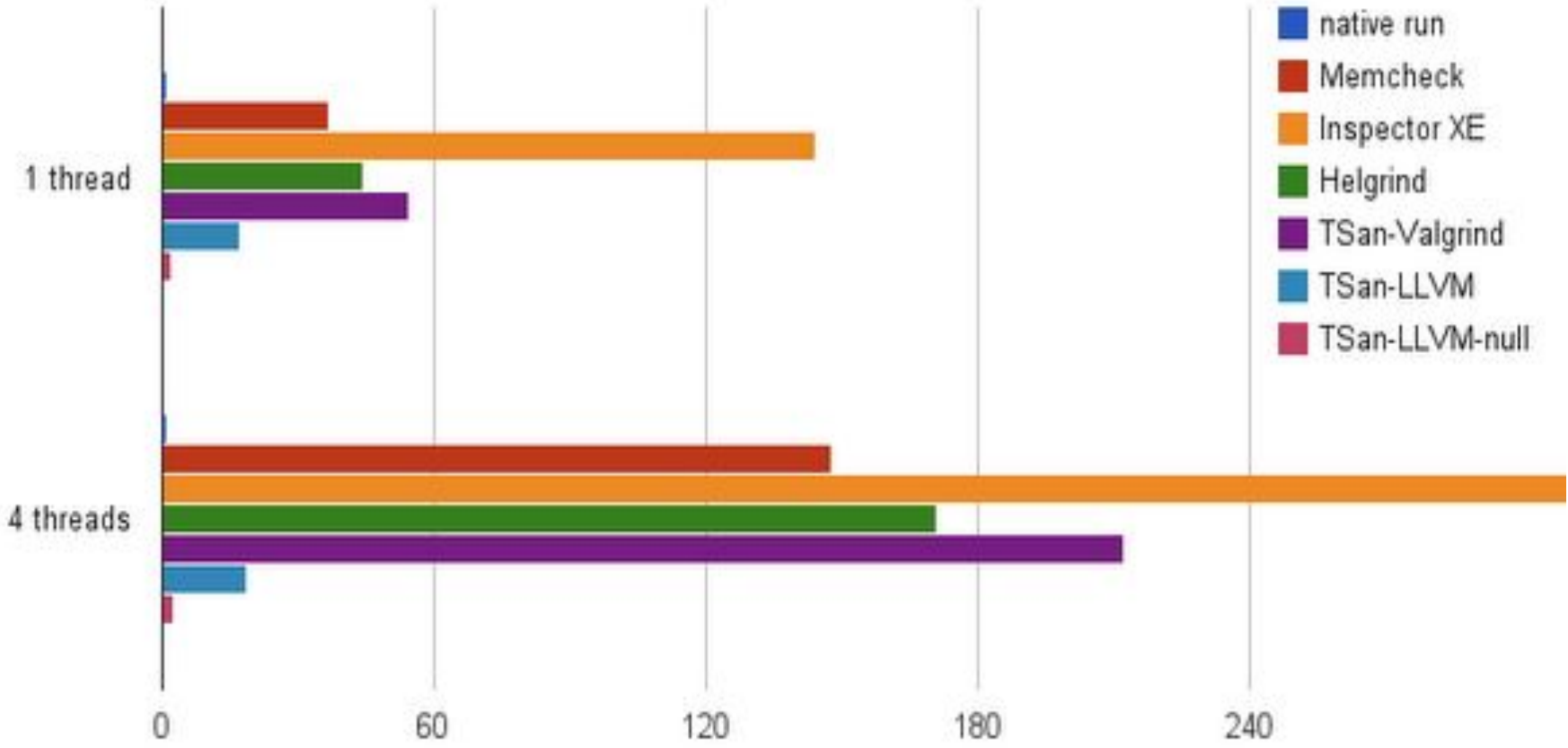
2.5x slowdown compared to non-instrumented code;

Only 1.5x slowdown if sampling is used – enough to browse the web and run interactive UI tests.

Performance on the Chromium tests



A synthetic benchmark



Limitations

A custom build is required, but:

- other tools need it as well;
- incremental builds are fast enough.

Cannot find races in libraries or JIT-compiled code.

Possible solution: translate this code dynamically using a binary translation system (e.g. PIN or DynamoRIO, not Valgrind).

Further improvements

Reduce the number of instrumented accesses:

- employ the escape analysis to skip accesses that are proven to be thread-local;

Reduce the time spent on each access:

- less contention on the shared shadow memory;
- record the memory access events for offline analysis.

Q&A

Thank you!

http://code.google.com/p/data-race-test