Detecting Data Races in Parallel Programs (Part 2)

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@cs.rice.edu
Detecting Data Races in Cilk Programs that use Locks

Guang-Ien Cheng, Mingdong Feng,
Charles Leiserson, Keith Randall,
Andrew Stark
Mutual Exclusion in Cilk: Locks

cilk_lock(L)

critical section

cilk_unlock(L)

Assumptions about Locking

- Lock/unlock pair is contained in a single thread
- Holding a lock across a parallel control construct is forbidden

Terminology

- “Lock set” of an access: set of locks held when access is performed
- Lock set of several accesses: intersection of individual sets
A Cilk Program with a Data Race

```cilk
int x;
Cilk_lockvar A, B;

cilk void foo1() {
    Cilk_lock(&A);
    Cilk_lock(&B);
    x += 5;
    Cilk_unlock(&B);
    Cilk_unlock(&A);
}

cilk void foo2() {
    Cilk_lock(&A);
    x -= 3;
    Cilk_unlock(&A);
}

cilk void foo3() {
    Cilk_lock(&B);
    x++;
    Cilk_unlock(&B);
}

cilk int main() {
    Cilk_lock_init(&A);
    Cilk_lock_init(&B);
    x = 0;
    spawn foo1();
    spawn foo2();
    spawn foo3();
sync;
    printf("%d", x);
}
```

- Conflicting accesses: at least one is a WRITE
- No ordering by happens before and no common lock
int x;
Cilk_lockvar A, B;

cilk void foo1() {
    Cilk_lock(&A);
    Cilk_lock(&B);
    x += 5;
    Cilk_unlock(&B);
    Cilk_unlock(&A);
}

cilk void foo2() {
    Cilk_lock(&A);
    x -= 3;
    Cilk_unlock(&A);
}

cilk void foo3() {
    Cilk_lock(&B);
    x++;
    Cilk_unlock(&B);
}

cilk int main() {
    Cilk_lock_init(&A);
    Cilk_lock_init(&B);
    x = 0;
    spawn foo1();
    spawn foo2();
    spawn foo3();
    sync;
    printf("%d", x);
}
Apparent vs. Feasible Races

initial condition: $x = 0$

T1
$z = 1$
lock(L)
$x = 2$
unlock(L)

T2
lock(L)
y = x
unlock(L)
if (y == 2) ... = z
Detecting Races in Cilk

• Data race if the lock set for two parallel accesses to the same location is empty and at least one is a WRITE

• Problem: “At least one is a WRITE” is cumbersome

• Simplification
  — introduce a fake R-LOCK
    – as if implicitly acquired and held for the duration of a read
    – for race detector: R-LOCK behaves as regular lock
  — if the lock set of two parallel accesses to the same location is empty, then a data race exists
Two Algorithms for Race Detection

- **ALL-SETS** - general serial race detection algorithm
- **BRELLEY** - faster serial race detection algorithm limited to “umbrella locking discipline”
ALL-SETS uses SP-Bags Representation

Use SP-Bags to determine concurrency relationship
**ALL-SETS Protocol**

**ACCESS(l) in thread e with lock set H**

1. for each \( \langle e', H' \rangle \in \text{lockers}[l] \)
2. do if \( e' \parallel e \) and \( H' \cap H = \{\} \)
3. then declare a data race
4. redundant \( \leftarrow \) FALSE

5. for each \( \langle e', H' \rangle \in \text{lockers}[l] \)
6. do if \( e' < e \) and \( H' \supseteq H \)
7. then \( \text{lockers}[l] \leftarrow \text{lockers}[l] - \{\langle e', H' \rangle\} \)

8. if \( e' \parallel e \) and \( H' \subseteq H \)
9. then redundant \( \leftarrow \) TRUE
10. if redundant = FALSE
11. then \( \text{lockers}[l] \leftarrow \text{lockers}[l] \cup \{\langle e, H \rangle\} \)

**lockers(L): set of tuples <thread, lock set>**

set of locks held by previous access to L by thread

**check for race:**
- parallel accesses
- non-overlapping lock sets
- prune redundant lock sets
- precedes & larger set
- add new lock set if not redundant

```cilk
Cilk_lock(&A); Cilk_lock(&B);
READ(1) \{A,B,R-LOCK\}
Cilk_unlock(&B); Cilk_unlock(&A);
Cilk_lock(&B); Cilk_lock(&C);
WRITE(1) \{B,C\}
Cilk_unlock(&C); Cilk_unlock(&B);
```
ALL-SETS Detects Races

Detects a race in a Cilk execution based on a given input if and only if a data race exists in the execution.

- if: any race reported between accesses by ALL-SETS meets the condition for a race: no common lock

- only if: if a race between accesses A and C exists in the computation, a race will be reported
  - if lock set for A was not added to lockers, there must be another parallel access with a smaller lock set. a race will be reported.
  - what if there was an intervening non-racing access B that caused a lock set for A to be removed from the lock set?
    - there can be no such access B
      B must have a larger lock set if it doesn’t race
      a lock set will be removed only if its lock set is larger than B’s thus, the A won’t have its lock set removed
ALL-SETS Properties

- Cilk program executes in time $T$
- Uses $V$ variables
- Uses a total of $n$ locks; no more than $k$ simultaneously
- Let $L = \text{max number of distinct lock sets used for any location}$

- **Time:** $O(TL(k + \alpha(V,V)))$
  - loose upper bound for $L$: $L \leq \text{sum of } n \choose i, i = 0, k = O(n^k/k!)$
  - at most $2L$ series/parallel tests (lines 2, 6) at cost of $O(\alpha(V,V))$
  - lock set comparisons take at most $O(k)$ time

- **Space:** $O(kLV)$
  - each lock set takes at most $k$ space
ALL-SETS vs. BRELLY

• ALL-SETS detects data races directly
  — but at asymptotically high cost: factor of $n^k$ slower than SP-bags protocol

• Umbrella locking discipline
  — requires each that each location be protected by the same lock within every parallel subcomputation
  — threads in series may use different locks (or none)

• BRELLY only detects violations of the “umbrella” locking discipline, which precludes races
  — more restrictive locking discipline than ALL-SETS requires
What’s Not in the Umbrella Discipline?

- Umbrella discipline requires that all sections in a parallel subcomputation use the same lock for a variable
- One thread uses A&B
- Two serial computations in parallel with first use:
  - only A
  - only B

```
lock(A)
x = unlock(A)
lock(B)
x = unlock(B)
lock(A)
lock(B)
x = unlock(B)
unlock(A)
```
Umbrellas in SP-Parse Tree
Understanding our Example with its SP-Parse

lock(A)
x =
unlock(A)

lock(B)
x =
unlock(B)
unlock(A)

lock(A)
lock(B)

P
S
P
P
P

\{A\}
{A,B}

\{B\}
A Cilk computation with a data race violates the umbrella discipline

• Any two threads involved in a race must have a P-node as their LCA in the SP-Parse

• The LCA P-node is the root of an unprotected umbrella
  — both threads access the same location
  — their lock sets are disjoint
BRELLY Protocol

Simplication: unlike ALL-SETS, keep only single lock set per location

ACCESS\(l\) in thread \(e\) with lock set \(H\)

1. if \(\text{accessor}[l] < e\)
   
   2. then \(\triangleright\) serial access
      
      \[
      \text{locks}[l] \leftarrow H, \text{leaving } \text{nonlocker}[h] \text{ with its old}
      \text{nonlocker if it was already in locks}[l] \text{ but}
      \text{setting } \text{nonlocker}[h] \leftarrow \text{accessor}[l] \text{ otherwise}
      \]
   
   3. for each lock \(h \in \text{locks}[l]\)
   
   4. do \(\text{alive}[h] \leftarrow \text{TRUE}\)
   
   5. \(\text{accessor}[l] \leftarrow e\)

   6. else \(\triangleright\) parallel access
   
   7. for each lock \(h \in \text{locks}[l] - H\)
   
   8. do if \(\text{alive}[h] = \text{TRUE}\)
   
   9. then \(\text{alive}[h] \leftarrow \text{FALSE}\)
   
   10. \(\text{nonlocker}[h] \leftarrow e\)

   11. for each lock \(h \in \text{locks}[l] \cap H\)

   12. do if \(\text{alive}[h] = \text{TRUE}\) and \(\text{nonlocker}[h] \parallel e\)

   13. then \(\text{alive}[h] \leftarrow \text{FALSE}\)

   14. if no locks in \(\text{locks}[l]\) are alive (or \(\text{locks}[l] = \{\}\))
   
   15. then report violation on \(l\) involving

   \(e\) and \(\text{accessor}[l]\)

   16. for each lock \(h \in H \cap \text{locks}[l]\)

   17. do report access to \(l\) without \(h\)

   by \(\text{nonlocker}[h]\)

Tag lock \(h\) in the lock set for \(L\) with
- \(\text{nonlocker}[h]\) - a thread accessing \(L\) without holding \(h\)
- \(\text{alive}[h]\) - whether \(h\) should be considered as belonging to the umbrella
  - kill \(h\) rather than removing from lock set to improve precision of race reports
**BRELLY at Work**

**Notation**

- $A(x): x$ is non-locker of $A$
- $A: A$ is not alive

- $e_7$ finds itself in parallel with non-locker $e_4$ for $B$
- Kills lock $B$ leaving no live locks
- Causes a data race to be detected

<table>
<thead>
<tr>
<th>thread</th>
<th>accessor[$l$]</th>
<th>locks[$l$]</th>
<th>access type</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>$e_0$</td>
<td>${}$</td>
<td>serial</td>
</tr>
<tr>
<td>$e_1$</td>
<td>$e_1$</td>
<td>${A(e_0), B(e_0)}$</td>
<td>parallel</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$e_1$</td>
<td>${A(e_0), B(e_2)}$</td>
<td>parallel</td>
</tr>
<tr>
<td>$e_3$</td>
<td>$e_1$</td>
<td>${A(e_0), B(e_2)}$</td>
<td>parallel</td>
</tr>
<tr>
<td>$e_4$</td>
<td>$e_4$</td>
<td>${}$</td>
<td>serial</td>
</tr>
<tr>
<td>$e_5$</td>
<td>$e_5$</td>
<td>${A(e_4), B(e_4)}$</td>
<td>serial</td>
</tr>
<tr>
<td>$e_6$</td>
<td>$e_5$</td>
<td>${A(e_6), B(e_4)}$</td>
<td>parallel</td>
</tr>
<tr>
<td>$e_7$</td>
<td>$e_5$</td>
<td>${A(e_6), B(e_4)}$</td>
<td>parallel</td>
</tr>
</tbody>
</table>
BRELLY Properties

• Cilk program executes in time $T$
• Uses $V$ variables
• Uses a total of $n$ locks; no more than $k$ simultaneously

• Time: $O(kT \alpha (V,V))$
  — tests if nonlocker[h] || e dominate running time
  — at most $k$ series/parallel tests at cost of $O(\alpha (V,V))$ each

• Space: $O(kV)$
  — at most $k$ locks per variable
Cilkscreen

• Detects and reports **data races** when program terminates
  — finds all data races even those by third-party or system libraries

  ```c
  // code with a data race
  int sum = 0;
  cilk_for (int i = 0; i < n; i++) {
    sum += a[i];
  }
  ```

• Does not report determinacy races
  — e.g. two concurrent strands use a lock to access a queue
    – enqueue & dequeue operations could occur in different order potentially leads to different result
Race Detection Strategies in Cilkscreen

• Lock covers
  — two conflicting accesses to a variable don’t race if some lock L is held while each of the accesses is performed by a strand

• Happens-before
  — two conflicting accesses do not race if one must happen before the other
    – access A is by a strand X, which precedes the spawn of strand Y which performs access B
    – access A is performed by strand X, which precedes a sync that is an ancestor of strand Y
Cilkscreen Race Example

```c
#include <stdio.h>
#include <cilk++/cilk_mutex.h>

int sum = 0;
cilk::mutex m;

#ifdef SYNCH
#define LOCK m.lock()
#define UNLOCK m.unlock()
#else
#define LOCK
#define UNLOCK
#endif

int cilk_main()
{
    do_accum(0, 1000);
    printf("sum = %d\n", sum);

    int ssum = 0;
    for (int i = 0; i <= 1000; i++) ssum += i;
    printf("serial sum = %d\n", ssum);
}

void do_accum(int l, int u)
{
    if (u == l) { LOCK; sum += l; UNLOCK; }
    else {
        int mid = (u+l)/2;
        cilk_spawn do_accum(l, mid);
        do_accum(mid+1, u);
    }
}
```
Cilkscreen Limitations

- Only detects races between Cilk++ strands
  - depends upon their strict fork/join paradigm
- Only detects races that occur given the input provided
  - does not prove the absence of races for other inputs
  - choose your testing inputs carefully!
- Cilkscreen runs serially, 15-30x slower
- Cilkscreen increases the memory footprint of an application
  - could cause an error if too large
- If you build your program with debug information, cilkscreen will associate races with source line numbers
Race on location 0x6033c0 between

/users/johnmc/tests/race.cilk:17: _Z8do_accumii+0x31 (eip=0x40167d)

and

/users/johnmc/tests/race.cilk:17: _Z8do_accumii+0x31 (eip=0x40167d)

/users/johnmc/tests/race.cilk:21: _Z8do_accumii+0x6a (eip=0x4016b6) called from here

/users/johnmc/tests/race.cilk:20: __cilk_spawn_do_accum_000+0x79 (eip=0x40161d) called from here

/users/johnmc/tests/race.cilk:20: _Z8do_accumii+0x5c (eip=0x4016a8) called from here

/users/johnmc/tests/race.cilk:20: __cilk_spawn_do_accum_000+0x79 (eip=0x40161d) called from here

/users/johnmc/tests/race.cilk:20: _Z8do_accumii+0x5c (eip=0x4016a8) called from here

/users/johnmc/tests/race.cilk:20: __cilk_spawn_do_accum_000+0x79 (eip=0x40161d) called from here

/users/johnmc/tests/race.cilk:20: _Z8do_accumii+0x5c (eip=0x4016a8) called from here

/users/johnmc/tests/race.cilk:20: __cilk_spawn_do_accum_000+0x79 (eip=0x40161d) called from here

/users/johnmc/tests/race.cilk:20: _Z8do_accumii+0x5c (eip=0x4016a8) called from here

/users/johnmc/tests/race.cilk:20: __cilk_spawn_do_accum_000+0x79 (eip=0x40161d) called from here

...
SigRace: Signature-based Race Detection

Abdullah Muzahid, Dario Suarez, Shanxiang Qi, Josep Torrellas
• People like shared-memory models for parallel programming

• Data races are a significant problem
  — most people don’t write programs in languages like Ct or NESL

• Software-only data race detection is slow
  — perhaps as much as 50x

• Every 18 months: 2x transistors on a chip
Hardware Support for Race Detection

• Monitor accesses in hardware and detect races

• Typical approach
  — tag data in caches with timestamps as accesses occur
  — piggyback tags & race detection on cache coherence protocol
    – invalidation, external read of a dirty line

• Specific approaches
  — happened-before (ReEnact, CORD, Min & Choi)
  — locksets (HARD)

• SigRace approach
  — don’t require changes to L1 cache!
  — don’t change the coherence protocol
FastTrack: Efficient and Precise Dynamic Race Detection (+ identifying destructive races)

Cormac Flanagan
UC Santa Cruz

Stephen Freund
Williams College
Dynamic Race Detection

- Compute partial order of operations
- Ensure conflicting access are not concurrent
- Sound & Complete

Happens Before
[Lamport 78]

Eraser
[SBN+ 97]
Dynamic Race Detection

- Track locks held on all accesses to var.
  - empty lock set implies possible race
- Unsound & Incomplete

Eraser [SBN+ 97]

Happens Before [Lamport 78]
Dynamic Race Detection

FastTrack

- Design Criteria:
  - sound (find at least 1st race on each var)
  - complete (no false alarms)
  - efficient

- Insight: Accesses to a var are almost always totally ordered in the Happens-Before relation

Happens Before
[Lamport 78]

Vector Clocks [M 88]
RaceTrack [PS 03]
Hybrid
Initialization [vPG 01]
...
Happens-Before

- Event Ordering:
  - program order
  - synchronization order

- Types of Races:
  - Write-Write
  - Write-Read
    - (write before read)
  - Read-Write
    - (read before write)
<table>
<thead>
<tr>
<th>$VC_A$</th>
<th>$VC_B$</th>
<th>$L_m$</th>
<th>$W_x$</th>
<th>$R_x$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write-Write Check: \( W_x \subseteq V_{C_A} \)?

\[
\begin{array}{c}
3 & 0 \\
\hline
4 & 1
\end{array}
\]

\[
\begin{array}{c}
3 & 0 \\
\hline
4 & 1
\end{array}
\]

Yes

Read-Write Check: \( R_x \subseteq V_{C_A} \)?

\[
\begin{array}{c}
0 & 1 \\
\hline
4 & 1
\end{array}
\]

\[
\begin{array}{c}
0 & 1 \\
\hline
4 & 1
\end{array}
\]

Yes

\( O(n) \) time
\[ VC_A \]

\[
\begin{array}{cc}
4 & 1 \\
\end{array}
\]

\[ x = 0 \]

\[ rel(m) \]

\[
\begin{array}{cc}
4 & 1 \\
\end{array}
\]

\[
\begin{array}{cc}
5 & 1 \\
\end{array}
\]

\[ VC_B \]

\[
\begin{array}{cc}
2 & 8 \\
\end{array}
\]

\[
\begin{array}{cc}
2 & 8 \\
\end{array}
\]

\[
\begin{array}{cc}
2 & 8 \\
\end{array}
\]

\[ L_m \]

\[
\begin{array}{cc}
2 & 1 \\
\end{array}
\]

\[
\begin{array}{cc}
2 & 1 \\
\end{array}
\]

\[
\begin{array}{cc}
4 & 1 \\
\end{array}
\]

\[ W_x \]

\[
\begin{array}{cc}
3 & 0 \\
\end{array}
\]

\[
\begin{array}{cc}
4 & 0 \\
\end{array}
\]

\[
\begin{array}{cc}
4 & 0 \\
\end{array}
\]

\[ R_x \]

\[
\begin{array}{cc}
0 & 1 \\
\end{array}
\]

\[
\begin{array}{cc}
0 & 1 \\
\end{array}
\]

\[
\begin{array}{cc}
0 & 1 \\
\end{array}
\]

\[
\begin{array}{cc}
0 & 1 \\
\end{array}
\]
### VC_A

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### VC_B

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

**rel(m)**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### L_m

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### W_x

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### R_x

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**acq(m)**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

### L_m

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### W_x

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### R_x

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### x = 1

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

**acq(m)**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### L_m

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### W_x

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

### R_x

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Write-Read Check: $W_x \subseteq VC_A$?

\[ \begin{array}{c}
4 & 8 \\
5 & 1
\end{array} \subseteq \begin{array}{c}
5 & 1
\end{array} \]?

No

$O(n)$ time
Write-Write and Write-Read Races

Thread A  Thread B  Thread C  Thread D

x = 0  x = 1  x = 1  x = 3

read x

O(n)
No Races Yet: Writes Totally Ordered!

Thread A

x = 0 → x = 1

Thread B

x = 1 → x = 3

Thread C

read x

Thread D

x = 3

O(n)
No Races Yet: Writes Totally Ordered!

Thread A  Thread B  Thread C  Thread D

x = 0  x = 1  x = 1  x = 3
read x  

O(1)
Write-Write Check: $W_x \subseteq V_C^A$?

$1@B \leq 4 1$?

Yes

$(1 \leq 1?)$

$O(1)$ time

Last Write "Epoch"
\[ VC_A \quad VC_B \quad L_m \quad W_x \]

\[
\begin{array}{cc}
4 & 1 \\
5 & 1 \\
5 & 1 \\
\end{array}
\quad \begin{array}{cc}
2 & 8 \\
2 & 8 \\
4 & 8 \\
\end{array}
\quad \begin{array}{cc}
2 & 1 \\
2 & 1 \\
4 & 1 \\
\end{array}
\quad \begin{array}{cc}
3@A \\
4@A \\
4@A \\
\end{array}
\]

\[ x = 0 \]

\[ rel(m) \]

\[ x = 1 \]

\[ acq(m) \]

\[ 8@B \]
Write-Read Check: $W_x \subseteq VC_A$?

$8@B \leq 51$? No

$(8 \leq 1?)$ O(1) time
Read-Write Races -- Ordered Reads

Thread A  Thread B  Thread C  Thread D

Most common case: thread-local, lock-protected, ...
Read-Write Races -- Unordered Reads

Thread A  Thread B  Thread C

\[ x = 0 \]

\[ \text{fork} \]

\[ \text{read } x \]

\[ ? \]

\[ ? \]

\[ ? \]

\[ x = 2 \]
Read-Write Check: $R_x \subseteq VC_A$ ?

8 1 \subseteq 8 0 ? No
Thread A  Thread B  Thread C  Thread D

read x  read x  ?  ?

x = 2

O(n)
Thread A

read x

Thread B

read x

Thread C

x = 2

Thread D
Thread A  Thread B  Thread C  Thread D

read x  read x

x = 2  x = 3

O(n)
Thread A

Thread B

Thread C

Thread D

read \( x \)  

read \( x \)  

\( x = 2 \)

\( x = 3 \) \( \text{O}(1) \)

Forget VC for \( R_x \) and switch back to "last read epoch"
RoadRunner Architecture

Standard JVM

Instrumented Bytecode

Event Stream
- A: acq(m)
- A: read(x)
- B: write(y)
- A: rel(m)

Back-End Checker

Error: race on x...
**Validation**

- Six race condition checkers
  - all use RoadRunner
  - share common components (eg, VectorClock)
  - profiled and optimized
- Further optimization opportunities
  - unsound extensions, dynamic escape analysis, static analysis, implement inside JVM, hardware support, ...
- 15 Benchmarks
  - 250 KLOC
  - locks, wait/notify, fork/join, barriers, ...
Warnings

22 false positives
3 false negatives

- Eraser [SBN+ 97]
- MultiRace [PS 03]
- GoldiLocks [EQT 07]
- Basic VC [M 88]
- DJIT+ [PS 03]
- FastTrack
Slowdown (x Base Time)

<table>
<thead>
<tr>
<th></th>
<th>Empty</th>
<th>Eraser</th>
<th>MultiRace</th>
<th>Goldilocks</th>
<th>Basic VC</th>
<th>DJIT+</th>
<th>FastTrack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.1</td>
<td>8.6</td>
<td>21.7</td>
<td>31.6</td>
<td>89.8</td>
<td>20.2</td>
<td>8.5</td>
</tr>
</tbody>
</table>
O(n) Vector Clock Operations
O(n) Vector Clock Operations

96.4% of all ops are Reads/Writes

R/W ops requiring O(n) time:

<table>
<thead>
<tr>
<th>Basic VC</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>DJIT+</td>
<td>26.0%</td>
</tr>
<tr>
<td>FastTrack</td>
<td>&lt;0.1%</td>
</tr>
</tbody>
</table>
Memory Usage

- FastTrack allocated ~200x fewer VCs

<table>
<thead>
<tr>
<th>Checker</th>
<th>Memory Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic VC, DJIT+</td>
<td>7.9x</td>
</tr>
<tr>
<td>FastTrack</td>
<td>2.8x</td>
</tr>
</tbody>
</table>

(Note: VCs for dead objects can be garbage collected)

- Improvements
  - accordion clocks [CB 01]
  - analysis granularity [PS 03, YRC 05] (see paper)
Eclipse 3.4

• Scale
  - > 6,000 classes
  - 24 threads
  - custom sync. idioms

• Precision (tested 5 common tasks)
  - Eraser: ~1000 warnings
  - FastTrack: ~30 warnings

• Performance on compute-bound tasks
  - > 2x speed of other precise checkers
  - same as Eraser
Beyond Detecting Race Conditions

- FastTrack finds real race conditions
  - races correlated with defects
  - cause unintuitive behavior on relaxed memory

- Which race conditions are real bugs?
  - that cause erroneous behaviors (crashes, etc)
  - and are not “benign race conditions”
class Point {
    double x, y;
    static Point p;

    Point() { x = 1.0; y = 1.0; }

    static Point get() {
        Point t = p;
        if (t != null) return t;
        synchronized (Point.class) {
            if (p == null) p = new Point();
            return p;
        }
    }

    static double slope() {
        return get().x / get().y;
    }

    public static void main(String[] args) {
        fork { System.out.println( slope() ); }
        fork { System.out.println( slope() ); }
    }
}
Thread 0

p = null
px = 0
py = 0
fork 1,2

Thread 1

read p // null
acquire
read p // null
p = new Point
px = 1
py = 1
release
read px // get 1
read py // get 1

Thread 2

read p // non-null
read px // ?
Thread 0

\[
p = \text{null} \\
x = 0 \\
y = 0 \\
\text{fork 1,2}
\]

Thread 1

\[
\text{read } p \quad // \quad \text{null} \\
\text{acquire} \\
\text{read } p \quad // \quad \text{null} \\
p = \text{new Point} \\
x = 1 \\
y = 1 \\
\text{release} \\
\text{read } x \quad // \quad \text{get 1} \\
\text{read } y \quad // \quad \text{get 1}
\]

Thread 2

\[
\text{read } p \quad // \quad \text{non-null} \\
\text{read } x \quad // \quad ?
\]
Thread 0

\[ p = \text{null} \]
\[ px = 0 \]
\[ py = 0 \]
fork 1,2

Thread 1

\[ \text{read } p \] // null
acquire
\[ \text{read } p \] // null
\[ p = \text{new Point} \]
\[ px = 1 \]
\[ py = 1 \]
release
\[ \text{read } px \] // get 1
\[ \text{read } py \] // get 1

Thread 2

read p // non-null
read px // ?

• Race: can return either write (mm non-determinism)
• Typical JVM: mostly sequentially consistent
• Adversarial memory
  - use heuristics to return older stale values
ThreadSanitizer, MemorySanitizer

Scalable run-time detection of uninitialized memory reads and data races with LLVM instrumentation

Timur Iskhodzhanov, Alexander Potapenko, Alexey Samsonov, Kostya Serebryany, Evgeniy Stepanov, Dmitry Vyukov

LLVM developers' meeting, Nov 8 2012
ThreadSanitizer
data races
ThreadSanitizer v1

- Race detector based on Valgrind
- Used since early 2009
- Slow (20x–300x slowdown)
  - Still, found thousands races
  - Faster & more usable than others
    - Helgrind (Valgrind)
    - Intel Parallel Inspector (PIN)
- WBIA'09
ThreadSanitizer v2 overview

- Simple compile-time instrumentation
  - ~400 LOC

- Redesigned run-time library
  - Fully parallel
  - No expensive atomics/locks on fast path
  - Scales to huge apps
  - Predictable memory footprint
  - Informative reports
TSan report example: data race

```c
void Thread1() { Global = 42; }
int main() {
    pthread_create(&t, 0, Thread1, 0);
    Global = 43;
    ...
% clang -fsanitize=thread -g a.c -fPIE -pie && ./a.out
WARNING: ThreadSanitizer: data race (pid=20373)
    Write of size 4 at 0x7f... by thread 1:
        #0 Thread1 a.c:1
    Previous write of size 4 at 0x7f... by main thread:
        #0 main a.c:4
    Thread 1 (tid=20374, running) created at:
        #0 pthread_create ???:0
        #1 main a.c:3
```
void foo(int *p) {
    *p = 42;
}

void foo(int *p) {
    __tsan_func_entry(__builtin_return_address(0));
    __tsan_write4(p);
    *p = 42;
    __tsan_func_exit()
}
Direct shadow mapping (64-bit Linux)

Shadow = 4 * (Addr & kMask);
Shadow cell

An 8-byte shadow cell represents one memory access:

- ~16 bits: TID (thread ID)
- ~42 bits: Epoch (scalar clock)
- 5 bits: position/size in 8-byte word
- 1 bit: IsWrite

Full information (no more dereferences)
### 4 shadow cells per 8 app. bytes

<table>
<thead>
<tr>
<th>TID</th>
<th>Epo</th>
<th>Pos</th>
<th>IsW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TID</th>
<th>Epo</th>
<th>Pos</th>
<th>IsW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TID</th>
<th>Epo</th>
<th>Pos</th>
<th>IsW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TID</th>
<th>Epo</th>
<th>Pos</th>
<th>IsW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: first access

T1
E1
0 : 2
W

Write in thread T1
Example: second access

Read in thread T2
Example: third access

Read in thread T3

T1
E1
0:2
W

T2
E2
4:8
R

T3
E3
0:4
R
Example: race?

Race if $E_1$ does not "happen-before" $E_3$
Fast happens-before

- Constant-time operation
  - Get TID and Epoch from the shadow cell
  - 1 load from thread-local storage
  - 1 comparison

- Similar to FastTrack (PLDI'09)
Shadow word eviction

- When all shadow cells are filled, one random cell is replaced
Informative reports

- Stack traces for two memory accesses:
  - current (easy)
  - previous (hard)

- TSan1:
  - Stores fixed number of frames (default: 10)
  - Information is never lost
  - Reference-counting and garbage collection
Stack trace for previous access

- Per-thread cyclic buffer of events
  - 64 bits per event (type + PC)
  - Events: memory access, function entry/exit
  - Information will be lost after some time
  - Buffer size is configurable

- Replay the event buffer on report
  - Unlimited number of frames
Function interceptors

- 100+ interceptors
  - malloc, free, ...
  - pthread_mutex_lock, ...
  - strlen, memcmp, ...
  - read, write, ...
Atomics

- LLVM atomic instructions are replaced with \_\_tsan\_* callbacks

\%0 = load atomic i8* \%a acquire, align 1

\%0 = call i8
@\_\_tsan\_atomic8\_load(i8* \%a, i32 504)
### TSan slowdown vs clang -O1

<table>
<thead>
<tr>
<th>Application</th>
<th>TSan1</th>
<th>TSan2</th>
<th>TSan1/TSan2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RPC benchmark</strong></td>
<td>40x</td>
<td>7x</td>
<td>5.5x</td>
</tr>
<tr>
<td><strong>Web server test</strong></td>
<td>25x</td>
<td>2.5x</td>
<td>10x</td>
</tr>
<tr>
<td><strong>String util test (1 thread)</strong></td>
<td>50x</td>
<td>6x</td>
<td>8.5x</td>
</tr>
</tbody>
</table>
Trophies

- 200+ races in Google server-side apps (C++)
- 80+ races in Go programs
  - 25+ bugs in Go stdlib
- Several races in OpenSSL
  - 1 fixed, ~5 'benign'
- More to come
  - We've just started testing Chrome : )
Key advantages

- **Speed**
  - > 10x faster than other tools

- **Native support for atomics**
  - Hard or impossible to implement with binary translation (Helgrind, Intel Inspector)
Limitations

- Only 64-bit Linux

- Hard to port to 32-bit platforms
  - Small address space
  - Relies on atomic 64-bit load/store

- Heavily relies on TLS
  - Slow TLS on some platforms

- Does not instrument:
  - pre-built libraries
  - inline assembly
ThreadSanitizer, MemorySanitizer

Scalable run-time detection of uninitialized memory reads and data races with LLVM instrumentation

Timur Iskhodzhanov, Alexander Potapenko, Alexey Samsonov, Kostya Serebryany, Evgeniy Stepanov, Dmitry Vyukov

LLVM developers' meeting, Nov 8 2012
Agenda

- **AddressSanitizer** (aka ASan)
  - recap from 2011
  - detects use-after-free and buffer overflows (C++)

- **ThreadSanitizer** (aka TSan)
  - detects data races (C++ & Go)

- **MemorySanitizer** (aka MSan)
  - detects uninitialized memory reads (C++)

- Similar tools, find different kinds of bugs
AddressSanitizer (recap from 2011)

- **Finds**
  - buffer overflows (stack, heap, globals)
  - use-after-free
  - some more

- **LLVM compiler module (~1KLOC)**
  - instruments all loads/stores
  - inserts red zones around Alloca and GlobalVariables

- **Run-time library (~10KLOC)**
  - malloc replacement (redzones, quarantine)
  - Bookkeeping for error messages
ASan report example: use-after-free

```c
int main(int argc, char **argv) {
    int *array = new int[100];
    delete [] array;
    return array[argc];  } // BOOM

% clang++ -O1 -fsanitize=address a.cc && ./a.out
==30226== ERROR: AddressSanitizer heap-use-after-free
READ of size 4 at 0x7faa07fce084 thread T0
    #0 0x40433c in main a.cc:4
0x7faa07fce084 is located 4 bytes inside of 400-byte region freed by thread T0 here:
    #0 0x4058fd in operator delete[](void*) _asan_rtl_
    #1 0x404303 in main a.cc:3
previously allocated by thread T0 here:
    #0 0x405579 in operator new[](unsigned long) _asan_rtl_
    #1 0x4042f3 in main a.cc:2
```
ASan shadow memory

Virtual address space

<table>
<thead>
<tr>
<th>0xffffffff</th>
<th>0x1fffffff</th>
<th>0x03fffffff</th>
<th>0x00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20000000</td>
<td>0x40000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

Instrumentation

```
char *shadow = addr >> 3;
if (*shadow)
    ReportError(a);
*a = ...;
```
ASan *marketing* slide

- 2x slowdown (Valgrind: 20x and more)

- 1.5x-4x memory overhead

- 500+ bugs found in Chrome in 1.5 years
  - Used for tests and fuzzing, 2000+ machines 24/7
  - 100+ bugs by external researchers

- 1000+ bugs everywhere else
  - Firefox, FreeType, FFmpeg, WebRTC, libjpeg-turbo, Perl, Vim, LLVM, GCC, MySQL
Plea to hardware vendors

Trivial hardware support may reduce the overhead from 2x to 20%
ThreadSanitizer
data races
ThreadSanitizer v1

- Race detector based on Valgrind
- Used since early 2009
- Slow (20x–300x slowdown)
  - Still, found thousands races
  - Faster & more usable than others
    - Helgrind (Valgrind)
    - Intel Parallel Inspector (PIN)
- WBIA'09
ThreadSanitizer v2 overview

- Simple compile-time instrumentation
  - ~400 LOC

- Redesigned run-time library
  - Fully parallel
  - No expensive atomics/locks on fast path
  - Scales to huge apps
  - Predictable memory footprint
  - Informative reports
void Thread1() { Global = 42; }
int main() {
    pthread_create(&t, 0, Thread1, 0);
    Global = 43;
    ...
% clang -fsanitize=thread -g a.c -fPIE -pie & & ./a.out
WARNING: ThreadSanitizer: data race (pid=20373)
Write of size 4 at 0x7f... by thread 1:
    #0 Thread1 a.c:1
Previous write of size 4 at 0x7f... by main thread:
    #0 main a.c:4
Thread 1 (tid=20374, running) created at:
    #0 pthread_create ???:0
    #1 main a.c:3
Compiler instrumentation

void foo(int *p) {
    *p = 42;
}

void foo(int *p) {
    __tsan_func_entry(__builtin_return_address(0));
    __tsan_write4(p);
    *p = 42;
    __tsan_func_exit()
}
Direct shadow mapping (64-bit Linux)

Shadow = 4 \times (\text{Addr} \& \text{kMask});
Shadow cell

An 8-byte shadow cell represents one memory access:

- ~16 bits: TID (thread ID)
- ~42 bits: Epoch (scalar clock)
- 5 bits: position/size in 8-byte word
- 1 bit: IsWrite

Full information (no more dereferences)
4 shadow cells per 8 app. bytes

<table>
<thead>
<tr>
<th></th>
<th>TID</th>
<th></th>
<th>TID</th>
<th></th>
<th>TID</th>
<th></th>
<th>TID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Epo</td>
<td></td>
<td>Epo</td>
<td></td>
<td>Epo</td>
<td></td>
<td>Epo</td>
</tr>
<tr>
<td></td>
<td>Pos</td>
<td></td>
<td>Pos</td>
<td></td>
<td>Pos</td>
<td></td>
<td>Pos</td>
</tr>
<tr>
<td></td>
<td>IsW</td>
<td></td>
<td>IsW</td>
<td></td>
<td>IsW</td>
<td></td>
<td>IsW</td>
</tr>
</tbody>
</table>
Example: first access

Write in thread T1

T1

E1

0 : 2

W
Example: second access

Read in thread T2
Example: third access

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>E2</td>
<td>E3</td>
</tr>
<tr>
<td>0:2</td>
<td>4:8</td>
<td>0:4</td>
</tr>
<tr>
<td>W</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

Read in thread T3
Example: race?

Race if \textbf{E1} does not "happen-before" \textbf{E3}
Fast happens-before

- Constant-time operation
  - Get TID and Epoch from the shadow cell
  - 1 load from thread-local storage
  - 1 comparison

- Similar to FastTrack (PLDI'09)
Shadow word eviction

- When all shadow cells are filled, one random cell is replaced
Informative reports

- Stack traces for two memory accesses:
  - current (easy)
  - previous (hard)

- TSan1:
  - Stores fixed number of frames (default: 10)
  - Information is never lost
  - Reference-counting and garbage collection
Stack trace for previous access

- Per-thread cyclic buffer of events
  - 64 bits per event (type + PC)
  - Events: memory access, function entry/exit
  - Information will be lost after some time
  - Buffer size is configurable

- Replay the event buffer on report
  - Unlimited number of frames
Function interceptors

- 100+ interceptors
  - malloc, free, ...
  - pthread_mutex_lock, ...
  - strlen, memcmp, ...
  - read, write, ...
Atomics

- LLVM atomic instructions are replaced with __tsan_* callbacks

%0 = load atomic i8* %a acquire, align 1

%0 = call i8
@__tsan_atomic8_load(i8* %a, i32 504)
## TSan slowdown vs clang -O1

<table>
<thead>
<tr>
<th>Application</th>
<th>TSan1</th>
<th>TSan2</th>
<th>TSan1/TSan2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RPC benchmark</strong></td>
<td>40x</td>
<td>7x</td>
<td>5.5x</td>
</tr>
<tr>
<td><strong>Web server test</strong></td>
<td>25x</td>
<td>2.5x</td>
<td>10x</td>
</tr>
<tr>
<td><strong>String util test (1 thread)</strong></td>
<td>50x</td>
<td>6x</td>
<td>8.5x</td>
</tr>
</tbody>
</table>
Trophies

- 200+ races in Google server-side apps (C++)
- 80+ races in Go programs
  - 25+ bugs in Go stdlib
- Several races in OpenSSL
  - 1 fixed, ~5 'benign'
- More to come
  - We've just started testing Chrome :)


Key advantages

- **Speed**
  - > 10x faster than other tools

- **Native support for atomics**
  - Hard or impossible to implement with binary translation (Helgrind, Intel Inspector)
Limitations

- Only 64-bit Linux
- Hard to port to 32-bit platforms
  - Small address space
  - Relies on atomic 64-bit load/store
- Heavily relies on TLS
  - Slow TLS on some platforms
- Does not instrument:
  - pre-built libraries
  - inline assembly
MemorySanitizer
uninitialized memory reads (UMR)
int main(int argc, char **argv) {
    int x[10];
    x[0] = 1;
    if (x[argc]) return 1;
    ...

% clang -fsanitize=memory -fPIE -pie a.c -g
% ./a.out
WARNING: MemorySanitizer: UMR (uninitialized-memory-read)
    #0 0x7ff6b05d9ca7 in main stack_umr.c:4
ORIGIN: stack allocation: x@main
Shadow memory

- Bit to bit shadow mapping
  - 1 means 'poisoned' (uninitialized)

- Uninitialized memory:
  - Returned by malloc
  - Local stack objects (poisoned at function entry)

- Shadow is propagated through arithmetic operations and memory writes

- Shadow is unpoisoned when constants are stored
Direct 1:1 shadow mapping

Shadow = Addr - 0x400000000000;
Shadow propagation

- Reporting UMR on first read causes false positives
  - E.g. copying `struct {char x; int y;}`

- Report UMR only on some uses (branch, syscall, etc)
  - That's what Valgrind does

- Propagate shadow values through expressions
  - \( A = B + C: \quad A' = B' \mid C' \)
  - \( A = B \land C: \quad A' = (B' \land C') \mid (\neg B \land C') \mid (B' \land \neg C) \)
  - Approximation to minimize false positives/negatives
  - Similar to Valgrind

- Function parameter/return: shadow is stored in TLS
  - Valgrind shadows registers/stack instead
Tracking origins

- Where was the poisoned memory allocated?
  \[ a = \text{malloc}() \ldots \]
  \[ b = \text{malloc}() \ldots \]
  \[ c = *a + *b \ldots \]
  \[ \text{if} \ (c) \ldots \quad \text{// UMR. Is 'a' guilty or 'b'?} \]

- Valgrind \texttt{--track-origins}: propagate the origin of the poisoned memory alongside the shadow

- MemorySanitizer: secondary shadow
  - Origin-ID is 4 bytes, 1:1 mapping
  - 2x additional slowdown
Secondary shadow (origin)

Origin = Addr - 0x200000000000;
MSan overhead

- Without origins:
  - CPU: 3x
  - RAM: 2x

- With origins:
  - CPU: 6x
  - RAM: 3x + malloc stack traces
Tricky part :(

- Missing any write instruction causes false reports
- Must monitor ALL stores in the program
  - libc, libstdc++, syscalls, etc

Solutions:

- Instrumented libc++, wrappers for libc
  - Works for many "console" apps, e.g. LLVM
- Instrument libraries at run-time
  - DynamoRIO-based prototype (SLOW)
- Instrument libraries statically (is it possible?)
- Compile everything, wrap syscalls
  - Will help AddressSanitizer/ThreadSanitizer too
MSan trophies

- Proprietary console app, 1.3 MLOC in C++
  - Not tested with Valgrind previously
  - 20+ unique bugs in < 2 hours
  - Valgrind finds the same bugs in 24+ hours
  - MSan gives better reports for stack memory

- 1 Bug in LLVM
  - LLVM bootstraps, ready to set regular runs

- A few bugs in Chrome (just started)
  - Have to use DynamoRIO module (MSanDR)
  - 7x faster than Valgrind
Summary (all 3 tools)

- **AddressSanitizer (memory corruption)**
  - A "must use" for everyone (C++)
  - Supported on Linux, OSX, CrOS, Android,
  - WIP: iOS, Windows, *BSD (?)

- **ThreadSanitizer (races)**
  - A "must use" if you have threads (C++, Go)
  - Only x86_64 Linux

- **MemorySanitizer (uses of uninitialized data)**
  - WIP, usable for "console" apps (C++)
  - Only x86_64 Linux
Q&A

http://code.google.com/p/address-sanitizer/

http://code.google.com/p/thread-sanitizer/

http://code.google.com/p/memory-sanitizer/
## ASan/MSan vs Valgrind (Memcheck)

<table>
<thead>
<tr>
<th></th>
<th>Valgrind</th>
<th>ASan</th>
<th>MSan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap out-of-bounds</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Stack out-of-bounds</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Global out-of-bounds</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Use-after-free</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Use-after-return</td>
<td>NO</td>
<td>Sometimes</td>
<td>NO</td>
</tr>
<tr>
<td>Uninitialized reads</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>CPU Overhead</td>
<td>10x-300x</td>
<td>1.5x-3x</td>
<td>3x</td>
</tr>
</tbody>
</table>
Why not a single tool?

- Slowdowns will add up
  - Bad for interactive or network apps

- Memory overheads will multiply
  - ASan redzone vs TSan/MSan large shadow

- Not trivial to implement