Data Race Detection II

Yizi Gu

Department of Computer Science

October 20, 2015
What is a data race?
What is a data race?

- When parallel threads access the same memory location concurrently, and at least one of the accesses is a write[1].
What is a data race?

- When parallel threads access the same memory location concurrently, and at least one of the accesses is a write[1].

Why is it critical?
Data Races: Definition and Solution

What is a data race?
- When parallel threads access the same memory location concurrently, and at least one of the accesses is a write[1].

Why is it critical?
- Make program outcomes weird and inconsistent.
What is a data race?
- When parallel threads access the same memory location concurrently, and at least one of the accesses is a write[1].

Why is it critical?
- Make program outcomes weird and inconsistent.
- Hard to debug without proper tools.
Data Races: Definition and Solution

What is a data race?
- When parallel threads access the same memory location concurrently, and at least one of the accesses is a write[1].

Why is it critical?
- Make program outcomes weird and inconsistent.
- Hard to debug without proper tools.

How to deal with data races? Data race detectors.
What is a data race?
  - When parallel threads access the same memory location concurrently, and at least one of the accesses is a write[1].

Why is it critical?
  - Make program outcomes weird and inconsistent.
  - Hard to debug without proper tools.

How to deal with data races? Data race detectors.
  - Nondeterminator[2], Eraser [3].
What is a data race?
- When parallel threads access the same memory location concurrently, and at least one of the accesses is a write.

Why is it critical?
- Make program outcomes weird and inconsistent.
- Hard to debug without proper tools.

How to deal with data races? Data race detectors.
- Nondeterminator[2], Eraser [3].
- Nondeterminator-2 [4], ThreadSanitizer [1], FastTrack [5].
Depending on the workflow: Static || Dynamic.
Depending on the workflow: Static || Dynamic.
Static detectors: Detect data races at compile time.
Depending on the workflow: Static || Dynamic.
Static detectors: Detect data races at compile time.
  + Might prove the program to be ’data race free’[4].
Depending on the workflow: Static || Dynamic.
Static detectors: Detect data races at compile time.
  + Might prove the program to be ’data race free’[4].
  – Potential for large numbers of false alarms[5].
Data Race Detector: Classification

Depending on the workflow: Static || Dynamic.

Static detectors: Detect data races at compile time.
  + Might prove the program to be ’data race free’[4].
  – Potential for large numbers of false alarms[5].

Dynamic detectors: Detect data races with a specific input.
Depending on the workflow: Static || Dynamic.
Static detectors: Detect data races at compile time.
  + Might prove the program to be ’data race free’[4].
  – Potential for large numbers of false alarms[5].
Dynamic detectors: Detect data races with a specific input.
  Postmortem
Depending on the workflow: Static || Dynamic.

Static detectors: Detect data races at compile time.
- + Might prove the program to be ’data race free’[4].
- – Potential for large numbers of false alarms[5].

Dynamic detectors: Detect data races with a specific input.
- Postmortem
- On-the-fly
Depending on the workflow: Static || Dynamic.
Static detectors: Detect data races at compile time.
  + Might prove the program to be 'data race free'[4].
  – Potential for large numbers of false alarms[5].
Dynamic detectors: Detect data races with a specific input.
  Postmortem
  On-the-fly
  Debuggers rather than verifiers[2].
Data Race Detector: Classification

Depending on the workflow: Static || Dynamic.
Static detectors: Detect data races at compile time.
  + Might prove the program to be ’data race free’[4].
  – Potential for large numbers of false alarms[5].
Dynamic detectors: Detect data races with a specific input.
  Postmortem
  On-the-fly
  Debuggers rather than verifiers[2].
Depending on the precision: Precise || Imprecise[5].
Two ways of avoiding data races:
Two ways of avoiding data races:

- Locking discipline[3]:

Happens-before & Locking discipline
Two ways of avoiding data races:

- Locking discipline[3]:
  - Protect shared values in critical sections.
  - e.g. Eraser[3].
Two ways of avoiding data races:

- **Locking discipline[3]:**
  - Protect shared values in critical sections.
  - e.g. Eraser[3].
  - False alarm problem.

- **Happens-before relations[6]:**
Two ways of avoiding data races:

- **Locking discipline[3]:**
  - Protect shared values in critical sections.
  - e.g. Eraser[3].
  - False alarm problem.

- **Happens-before relations[6]:**
  - Avoid concurrent accesses.
  - e.g. Nondeterminator[2].
Two ways of avoiding data races:

- **Locking discipline[3]:**
  - Protect shared values in critical sections.
  - e.g. Eraser[3].
  - False alarm problem.

- **Happens-before relations[6]:**
  - Avoid concurrent accesses.
  - e.g. Nondeterminator[2].
  - Ignores locks.
Two ways of avoiding data races:

- **Locking discipline[3]:**
  - Protect shared values in critical sections.
  - e.g. Eraser[3].
  - False alarm problem.

- **Happens-before relations[6]:**
  - Avoid concurrent accesses.
  - e.g. Nondeterminator[2].
  - Ignores locks.

Combine two ways together: Hybrid algorithm.
Outline

1 Detecting Data Races in Cilk Programs that Use Locks
   - All-Sets Algorithm
   - Abelian Programs
   - Brelly Algorithm
   - Experiment Results
   - Main Contributions

2 FastTrack: Efficient and Precise Dynamic Race Detection
   - Vector Clock
   - Experiment Results

3 ThreadSanitizer: Data race detection in practice
   - Overview
   - Hybrid Race Detection Algorithm
   - Dynamic Annotation
   - Experiment Results

4 Summary
Detecting Data Races in Cilk Programs that Use Locks

Detecting Data Races in Cilk Programs that Use Locks

A program using locks that have data races

```c
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);
}
```

if foo2 and foo3 execute concurrently, they may both see $x = 0$. 

Yizi Gu

Data Race Detection II
All-Set Algorithm: Key Points

- Records locks held in each access with *lock set* $H$
All-Set Algorithm: Key Points

- Records locks held in each access with *lock set* $H$
- For each memory location $l$, records $<e,H>$ in lockers[$l$]
Records locks held in each access with lock set $H$
For each memory location $l$, records $<e,H>$ in lockers[$l$]
- Prune the lockers when possible.
Detecting Data Races in Cilk Programs that Use Locks

All-Sets Algorithm

\[ \text{ACCESS}(l) \text{ in thread } e \text{ with lock set } H \]

\begin{algorithm}
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 \( e' \prec e \) 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 \( \text{lockers}[l] \leftarrow \text{lockers}[l] \cup \{\langle e, H \rangle\} \)
\end{algorithm}

- Use \textit{fake lock} for read access.
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 \(e' \prec e\) and \(H' \supseteq H\)
7. then \(\text{lockers}[l] \leftarrow \text{lockers}[l] - \{\langle e', H' \rangle\}\)
8. if \(e' \parallel \{\langle e, H \rangle\}\)
9. then redundant \(\leftarrow\) TRUE
10. if redundant = FALSE
11. then \(\text{lockers}[l] \leftarrow \text{lockers}[l] \cup \{\langle e, H \rangle\}\)

- Use *fake lock* for read access.
- Line 2 detects the data race.
Detecting Data Races in Cilk Programs that Use Locks

All-Sets Algorithm

\begin{align*}
\text{ACCESS}(l) \text{ in thread } e \text{ with lock set } H \\
1 & \quad \textbf{for} \ each \ \langle e', H' \rangle \in \text{lockers}[l] \\
2 & \quad \quad \textbf{do if} \ e' \parallel e \text{ and } H' \cap H = \{\} \\
3 & \quad \quad \quad \textbf{then} \ declare \ a \ data \ race \\
4 & \quad \quad \textit{redundant} \leftarrow \text{FALSE} \\
5 & \quad \textbf{for} \ each \ \langle e', H' \rangle \in \text{lockers}[l] \\
6 & \quad \quad \textbf{do if} \ e' \prec e \text{ and } H' \supseteq H \\
7 & \quad \quad \quad \textbf{then} \ \text{lockers}[l] \leftarrow \text{lockers}[l] - \{\langle e', H' \rangle \} \\
8 & \quad \quad \textbf{if} \ e' \parallel e \text{ and } H' \subseteq H \\
9 & \quad \quad \quad \textbf{then} \ \textit{redundant} \leftarrow \text{TRUE} \\
10 & \quad \textbf{if} \ \textit{redundant} = \text{FALSE} \\
11 & \quad \quad \textbf{then} \ \text{lockers}[l] \leftarrow \text{lockers}[l] \cup \{\langle e, H \rangle \}
\end{align*}

- Use \textit{fake lock} for read access.
- Line 2 detects the data race.
- Line 5-9 prune redundant lockers.
L: the maximum number of distinct lock sets in $lockers[l]$.  
k: the maximum number of locks held simultaneously.  
V: number of shared memory locations.
All-Sets Algorithm: Property

L: the maximum number of distinct lock sets in lockers\[l\].
k: the maximum number of locks held simultaneously.
V: number of shared memory locations.

- Time complexity: \(O(TL(k + \alpha(V, V)))\)

No false positives; No false negatives.
Detects apparent data races, not feasible ones.
All-Sets Algorithm: Property

L: the maximum number of distinct lock sets in $lockers[l]$.  
k: the maximum number of locks held simultaneously.  
V: number of shared memory locations.  

- Time complexity: $O(TL(k + \alpha(V, V)))$  
- Space complexity: $O(kLV)$.  

No false positives; No false negatives.  
Detects apparent data races, not feasible ones.
All-Sets Algorithm: Property

L: the maximum number of distinct lock sets in lockers[l].
k: the maximum number of locks held simultaneously.
V: number of shared memory locations.

- Time complexity: $O(TL(k + \alpha(V, V)))$
- Space complexity: $O(kLV)$.

if $k \ll n$, $L \leq \sum_{i=0}^{k} \binom{n}{i} = O(n^k/k!)$
Detecting Data Races in Cilk Programs that Use Locks

All-Sets Algorithm

All-Sets Algorithm: Property

L: the maximum number of distinct lock sets in \( \text{lockers}[l] \).
k: the maximum number of locks held simultaneously.
V: number of shared memory locations.

- **Time complexity:** \( O(TL(k + \alpha(V, V))) \)
- **Space complexity:** \( O(kLV) \).

If \( k \ll n, \ L \leq \sum_{i=0}^{k} \binom{n}{i} = O(n^k/k!) \)
- **Time complexity:** \( O(n^k T\alpha(V, V)) \).
All-Sets Algorithm: Property

L: the maximum number of distinct lock sets in lockers[\(l\)].
k: the maximum number of locks held simultaneously.
V: number of shared memory locations.

- Time complexity: \(O(TL(k + \alpha(V, V)))\)
- Space complexity: \(O(kLV)\).

if \(k \ll n\), \(L \leq \sum_{i=0}^{k} \binom{n}{i} = O(n^k/k!\)
- Time complexity: \(O(n^kT\alpha(V, V))\).
- Space complexity: \(O(n^kV)\).
L: the maximum number of distinct lock sets in lockers\([l]\).
k: the maximum number of locks held simultaneously.
V: number of shared memory locations.

- Time complexity: \(O(TL(k + \alpha(V, V)))\)
- Space complexity: \(O(kLV)\).

If \(k \ll n\), \(L \leq \sum_{i=0}^{k} \binom{n}{i} = O(n^k/k!\)

- Time complexity: \(O(n^kT\alpha(V, V))\).
- Space complexity: \(O(n^kV)\).

No false positives; No false negatives.
All-Sets Algorithm: Property

L: the maximum number of distinct lock sets in lockers\([l]\).

- **Time complexity:** \(O(TL(k + \alpha(V, V)))\)
- **Space complexity:** \(O(kLV)\).

if \(k \ll n, L \leq \sum_{i=0}^{k} \binom{n}{i} = O(n^k / k!)\)

- **Time complexity:** \(O(n^k T\alpha(V, V))\).
- **Space complexity:** \(O(n^k V)\).

No false positives; No false negatives. Detects apparent data races, not feasible ones.
Detecting Data Races in Cilk Programs that Use Locks

Apparent & Feasible Data Races

```c
Thread1
obj->UpdateMe();
mu.Lock();
flag = true;
mu.Unlock();

Thread2
mu.Lock();
bool f = flag;
mu.Unlock();
if (f) obj->UpdateMe();
```
Abelian Programs

- **Abelian Program**: If any pair of parallel critical sections that are protected by the same lock commute and the locks are properly nested[4].

Yizi Gu
Abelian Programs

- **Abelian Program**: If any pair of parallel critical sections that are protected by the same lock commute and the locks are properly nested[4].
- **Property of abelian programs**: Abelian programs that produce dead-lock-free computation with no data races are determinate[4].
In the worst case, All-Sets is slower than Nondeterminator.
In the worst case, All-Sets is slower than Nondeterminator.
Appropriate locking discipline helps to avoid data races.
In the worst case, All-Sets is slower than Nondeterminator. Appropriate locking discipline helps to avoid data races. Checking the violation of a specific locking discipline may be faster.
The Umbrella Locking Discipline

- **An umbrella of accesses to a location** $l$: A subtree rooted at a P-node containing accesses to $l$ in both its left and right subtrees[4].
An umbrella of accesses to a location \( l \): A subtree rooted at a P-node containing accesses to \( l \) in both its left and right subtrees[4].

Protected: If the umbrella of accesses to \( l \) have a nonempty lock set.
The Umbrella Locking Discipline

- **An umbrella of accesses to a location \( l \):** A subtree rooted at a P-node containing accesses to \( l \) in both its left and right subtrees[4].
- **Protected:** If the umbrella of accesses to \( l \) have a nonempty lock set.
- **Umbrella Locking Principle:** The program contains no unprotected umbrellas.
Two shadow spaces of shared memory:
Brelly Algorithm: Key Points

Two shadow spaces of shared memory:

- $accessor[l]$: The thread that performed the last 'serial access' to $l$. 
Brelly Algorithm: Key Points

Two shadow spaces of shared memory:

- \textit{accessor}[l]: The thread that performed the last 'serial access' to \( l \).
- \textit{locks}[l]: The lock set of \( \text{accessor}[l] \).
Two shadow spaces of shared memory:

- **accessor\([l]\)**: The thread that performed the last 'serial access' to \(l\).
- **locks\([l]\)**: The lock set of **accessor\([l]\)**.

Brelly keeps only a single lock set for each \(l\).
Brelly Algorithm: Key Points

Two shadow spaces of shared memory:

- $accessor[l]$: The thread that performed the last 'serial access' to $l$.

Brelly keeps only a single lock set for each $l$.
Each lock $h \in locks[l]$ is tagged with $nonlocker[h]$ and $alive[h]$.
Brelly Algorithm: Key Points

Two shadow spaces of shared memory:

- $accessor[l]$: The thread that performed the last ’serial access’ to $l$.

Brelly keeps only a single lock set for each $l$. Each lock $h \in locks[l]$ is tagged with $nonlocker[h]$ and $alive[h]$.

- $alive[h]$: Records whether $h$ still belongs to locks that protect the umbrella.
Two shadow spaces of shared memory:

- **accessor**[$l$]: The thread that performed the last 'serial access' to $l$.
- **locks**[$l$]: The lock set of **accessor**[$l$].

Brelly keeps only a single lock set for each $l$.

Each lock $h \in \text{locks}[l]$ is tagged with **nonlocker**[$h$] and **alive**[$h$].

- **alive**[$h$]: Records whether $h$ still belongs to locks that protect the umbrella.
- **nonlocker**[$h$]: The thread that accesses $l$ without holding $h$. 
Detecting Data Races in Cilk Programs that Use Locks

Brelly Algorithm

ACCESS(l) in thread e with lock set H

1. if accessor[l] \leftarrow e
2. then \triangleright serial access

\text{locks}[l] \leftarrow H, leaving nonlocker[h] with its old
   nonlocker if it was already in \text{locks}[l] but
   setting nonlocker[h] \leftarrow accessor[l] otherwise

3. for each lock h \in \text{locks}[l]
4. do alive[h] \leftarrow \text{TRUE}
5. accessor[l] \leftarrow e
6. else \triangleright parallel access

7. for each lock h \in \text{locks}[l] \setminus H
8. do if alive[h] = \text{TRUE}
9. then alive[h] \leftarrow \text{FALSE}
10. nonlocker[h] \leftarrow e
11. for each lock h \in \text{locks}[l] \cap H
12. do if alive[h] = \text{TRUE} and nonlocker[h] \parallel e
13. then 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 accessor[l]
16. for each lock h \in H \cap \text{locks}[l]
17. do report access to l without h
   by nonlocker[h]
Brelly Algorithm: An Example

<table>
<thead>
<tr>
<th>thread</th>
<th>accessor[\ell]</th>
<th>locks[\ell]</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 Algorithm: Property

$k$: the maximum number of locks held simultaneously.
$V$: number of shared memory locations.
Brelly Algorithm: Property

- k: the maximum number of locks held simultaneously.
- V: number of shared memory locations.
- Time complexity: $O(k T \alpha(V, V))$
k: the maximum number of locks held simultaneously.
V: number of shared memory locations.

- Time complexity: $O(kT\alpha(V, V))$
- Space complexity: $O(kV)$. 

No false positives; No false negatives.
**Brelly Algorithm: Property**

- $k$: the maximum number of locks held simultaneously.
- $V$: number of shared memory locations.
  - Time complexity: $O(kT\alpha(V, V))$
  - Space complexity: $O(kV)$.

No false positives; No false negatives.
Brelly Algorithm: Property

\(k\): the maximum number of locks held simultaneously.

\(V\): number of shared memory locations.

- Time complexity: \(O(kT\alpha(V, V))\)
- Space complexity: \(O(kV)\).

No false positives; No false negatives. Detects apparent data races, not feasible ones.
## Experiment Results

### Parameters

<table>
<thead>
<tr>
<th>Program</th>
<th>Parameters</th>
<th>Time (sec.)</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>orig.</td>
<td>ALL.</td>
</tr>
<tr>
<td>maxflow</td>
<td>sp. 1K</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>sp. 4K</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>d. 256</td>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>d. 512</td>
<td>2</td>
<td>512</td>
</tr>
<tr>
<td>n-body</td>
<td>1K</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2K</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bucket</td>
<td>100K</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rad</td>
<td>iter. 1</td>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>iter. 2</td>
<td>2</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>iter. 5</td>
<td>2</td>
<td>168</td>
</tr>
<tr>
<td></td>
<td>iter. 13</td>
<td>2</td>
<td>528</td>
</tr>
</tbody>
</table>
Main Contributions

All-Sets Algorithm
All-Sets Algorithm

- On-the-fly algorithm.
Main Contributions

All-Sets Algorithm
- On-the-fly algorithm.
- No false positives; No false negatives.
Main Contributions

All-Sets Algorithm
- On-the-fly algorithm.
- No false positives; No false negatives.

Brelly Algorithm
Main Contributions

All-Sets Algorithm
- On-the-fly algorithm.
- No false positives; No false negatives.

Brelly Algorithm
- Asymptotically faster than All-Sets.
Main Contributions

All-Sets Algorithm

- On-the-fly algorithm.
- No false positives; No false negatives.

Brelly Algorithm

- Asymptotically faster than All-Sets.
- Reports violation of the ’umbrella’ locking principle.
Main Contributions

All-Sets Algorithm
- On-the-fly algorithm.
- No false positives; No false negatives.

Brelly Algorithm
- Asymptotically faster than All-Sets.
- Reports violation of the 'umbrella' locking principle.

Definition of 'abelian' programs
Main Contributions

All-Sets Algorithm
- On-the-fly algorithm.
- No false positives; No false negatives.

Brelly Algorithm
- Asymptotically faster than All-Sets.
- Reports violation of the ’umbrella’ locking principle.

Definition of ’abelian’ programs
- Apparent data races are not necessarily feasible.
Main Contributions

All-Sets Algorithm
- On-the-fly algorithm.
- No false positives; No false negatives.

Brelly Algorithm
- Asymptotically faster than All-Sets.
- Reports violation of the ‘umbrella’ locking principle.

Definition of ‘abelian’ programs
- Apparent data races are not necessarily feasible.
- The algorithms fix the scheduling of threads.
Main Contributions

All-Sets Algorithm
- On-the-fly algorithm.
- No false positives; No false negatives.

Brelly Algorithm
- Asymptotically faster than All-Sets.
- Reports violation of the ’umbrella’ locking principle.

Definition of ’abelian’ programs
- Apparent data races are not necessarily feasible.
- The algorithms fix the scheduling of threads.
- ’Abelian’ programs produce the same result regardless of scheduling.
FastTrack: Efficient and Precise Dynamic Race Detection

Cormac Flanagan and Stephan N. Freund. In *PLDI’09*
## Experiment Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (loc)</th>
<th>Thread Count</th>
<th>Base Time (sec)</th>
<th>Instrumented Time (slowdown)</th>
<th>Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EMPTY</td>
<td>ERASER</td>
</tr>
<tr>
<td>colt</td>
<td>111,421</td>
<td>11</td>
<td>16.1</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>crypt</td>
<td>1,241</td>
<td>7</td>
<td>0.2</td>
<td>7.6</td>
<td>14.7</td>
</tr>
<tr>
<td>lutact</td>
<td>1,627</td>
<td>4</td>
<td>4.5</td>
<td>2.6</td>
<td>8.1</td>
</tr>
<tr>
<td>moldyn</td>
<td>1,402</td>
<td>4</td>
<td>8.5</td>
<td>5.6</td>
<td>9.1</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3,669</td>
<td>4</td>
<td>5.0</td>
<td>4.2</td>
<td>8.5</td>
</tr>
<tr>
<td>mtrt</td>
<td>11,317</td>
<td>5</td>
<td>0.5</td>
<td>5.7</td>
<td>6.5</td>
</tr>
<tr>
<td>raja</td>
<td>12,028</td>
<td>2</td>
<td>0.7</td>
<td>2.8</td>
<td>3.0</td>
</tr>
<tr>
<td>raytracer</td>
<td>1,970</td>
<td>4</td>
<td>6.8</td>
<td>4.6</td>
<td>6.7</td>
</tr>
<tr>
<td>sparse</td>
<td>868</td>
<td>4</td>
<td>8.5</td>
<td>5.4</td>
<td>11.3</td>
</tr>
<tr>
<td>series</td>
<td>967</td>
<td>4</td>
<td>175.1</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>sor</td>
<td>1,005</td>
<td>4</td>
<td>0.2</td>
<td>4.4</td>
<td>9.1</td>
</tr>
<tr>
<td>tsp</td>
<td>706</td>
<td>5</td>
<td>0.4</td>
<td>4.4</td>
<td>24.9</td>
</tr>
<tr>
<td>elevator</td>
<td>1,447</td>
<td>5</td>
<td>5.0</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>philo</td>
<td>86</td>
<td>6</td>
<td>7.4</td>
<td>1.1</td>
<td>1.0</td>
</tr>
<tr>
<td>hede</td>
<td>24,937</td>
<td>6</td>
<td>5.9</td>
<td>1.1</td>
<td>0.9</td>
</tr>
<tr>
<td>jbb</td>
<td>30,491</td>
<td>5</td>
<td>72.9</td>
<td>1.3</td>
<td>1.5</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td>4.1</td>
<td>8.6</td>
</tr>
</tbody>
</table>
ThreadSanitizer: Data race detection in practice

Konstantin Serebryany and Timur Iskhodzhanov. In WBIA’09
Main Contributions

A dynamic race detector used in industry.
Main Contributions

A dynamic race detector used in industry.

- Hybrid algorithm based on happens before and locksets.
Main Contributions

A dynamic race detector used in industry.

- Hybrid algorithm based on happens before and locksets.
- Informative race reports.
Main Contributions

A dynamic race detector used in industry.

- Hybrid algorithm based on happens before and locksets.
- Informative race reports.

Dynamic Annotations.
A dynamic race detector used in industry.

- Hybrid algorithm based on happens before and locksets.
- Informative race reports.

Dynamic Annotations.

- Teach the detector non-standard synchronization methods developed by users.
Main Contributions

A dynamic race detector used in industry.
- Hybrid algorithm based on happens before and locksets.
- Informative race reports.

Dynamic Annotations.
- Teach the detector non-standard synchronization methods developed by users.
- Hide benign races and reduce false positives.
Main Contributions

A dynamic race detector used in industry.
- Hybrid algorithm based on happens before and locksets.
- Informative race reports.

Dynamic Annotations.
- Teach the detector non-standard synchronization methods developed by users.
- Hide benign races and reduce false positives.

Examples of real races and false positives occurred in practice.
Main Contributions

A dynamic race detector used in industry.
- Hybrid algorithm based on happens before and locksets.
- Informative race reports.

Dynamic Annotations.
- Teach the detector non-standard synchronization methods developed by users.
- Hide benign races and reduce false positives.

Examples of real races and false positives occurred in practice.
Advises on writing multithreaded programs.
Hybrid Race Detection Algorithm: Definitions

- **Tid**: A unique ID of a thread.
- **ID**: A unique ID of a memory location.
- **EventType**: Read, Write, WrLock, RdLock, WrUnlock, RdUnlock, Signal, Wait.
- **Event**: A triple \{EventType, Tid, ID\}.
- **Segment**: A sequence of events containing only memory access events.
- **Happens-before arc**: Event pair \(< X, Y >\)
  - \(X = \{\text{Signal}, \text{Tx}, \text{Ax}\}\), \(Y = \{\text{Wait}, \text{Ty}, \text{Ay}\}\)
  - \(\text{Tx} \neq \text{Ty}, \text{Ax} = \text{Ay}\).
**Happens-before Relation: Definition**

**Happens-before**: a partial order on the set of events[1].

Given two events $X = \{\text{EventType}_X, T_x, A_x\}$, $Y = \{\text{EventType}_Y, T_y, A_y\}$

- $T_x = T_y$
- $\{X, Y\}$ is a happens-before arc
- $\exists E_1, E_2 : X \preceq E_1 \prec E_2 \preceq Y$
Happens-before: a partial order on the set of events[1]. Given two events \( X = \{ \text{EventType}X, \, T_x, \, A_x \} \), \( Y = \{ \text{EventType}Y, \, T_y, \, A_y \} \)
- \( T_x = T_y \)
- \( \{X, Y\} \) is a happens-before arc
- \( \exists E_1, E_2 : X \preceq E_1 \prec E_2 \preceq Y \)

Segment Sets: \( \{S_1, S_2, \ldots, S_N\} \) such that \( \forall i, j : S_i \not\prec S_j \)
Happens-before Relation: Example

Figure: $S_1 \prec S_4; S_1 \prec S_5; S_1 \prec S_7; S_4 \not\prec S_2$
ThreadSanitizer: Data race detection in practice

Hybrid Race Detection Algorithm

Hybrid Race Detection Algorithm

HANDLE-READ-OR-WRITE-EVENT(IsWrite, Tid, ID)
1 ▷ Handle event Read \( T_{id}(I) \) or Write \( T_{id}(I) \)
2 \((SS_{Wr}, SS_{Rd}) \leftarrow \text{GET-PER-ID-STATE}(I)\)
3 \(Seg \leftarrow \text{GET-CURRENT-SEGMENT}(T_{id})\)
4 if IsWrite
5 then ▷ Write event: update \( SS_{Wr} \) and \( SS_{Rd} \)
6 \( SS_{Rd} \leftarrow \{s : s \in SS_{Rd} \land s \notin Seg\} \)
7 \( SS_{Wr} \leftarrow \{s : s \in SS_{Wr} \land s \notin Seg\} \cup \{Seg\} \)
8 else ▷ Read event: update \( SS_{Rd} \)
9 \( SS_{Rd} \leftarrow \{s : s \in SS_{Rd} \land s \notin Seg\} \cup \{Seg\} \)
10 \text{SET-PER-ID-STATE}(I, SS_{Wr}, SS_{Rd})
11 if \text{IS-RACE}(SS_{Wr}, SS_{Rd})
12 then ▷ Report a data race on \( I \)
13 \text{REPORT-RACE}(IsWrite, Tid, Seg, I)
14
15 IS-RACE(SS_{Wr}, SS_{Rd})
16 1 ▷ Check if we have a race.
17 2 \( N_{W} \leftarrow \text{SEGMENT-SET-SIZE}(SS_{Wr})\)
18 for \( i \leftarrow 1 \) to \( N_{W} \)
19 do \( W_{1} \leftarrow SS_{Wr}[i] \)
20 \text{LS}_{1} \leftarrow \text{GET-WRITER-LOCK-SET}(W_{1})
21 ▷ Check all write-write pairs.
22 for \( j \leftarrow i + 1 \) to \( N_{W} \)
23 do \( W_{2} \leftarrow SS_{Wr}[j] \)
24 \text{LS}_{2} \leftarrow \text{GET-WRITER-LOCK-SET}(W_{2})
25 \text{ASSERT}(W_{1} \not\subseteq W_{2} \land W_{2} \not\subseteq W_{1})
26 if \( \text{LS}_{1} \cap \text{LS}_{2} = \emptyset \)
27 then return \text{true}
28 ▷ Check all write-read pairs.
29 for \( R \in SS_{Rd} \)
30 do \( \text{LS}_{R} \leftarrow \text{GET-READER-LOCK-SET}(R) \)
31 if \( W_{1} \not\subseteq R \) and \( \text{LS}_{1} \cap \text{LS}_{R} = \emptyset \)
32 then return \text{true}
33 return \text{false}
Segment Size

- **Default**: Created when entering a new super-block.
**Segment Size**

- **Default**: Created when entering a new super-block.
  - Nearly precise.
Segment Size

- **Default**: Created when entering a new super-block.
  - Nearly precise.
- **Fast**: Created only after synchronization events.
Segment Size

- **Default**: Created when entering a new super-block.
  - Nearly precise.

- **Fast**: Created only after synchronization events.
  - Contexts are mixed, only used for regression testing.
Segment Size

- **Default**: Created when entering a new super-block.
  - Nearly precise.
- **Fast**: Created only after synchronization events.
  - Contexts are mixed, only used for regression testing.
- **Precise**: Created on each memory access.
**Segment Size**

- **Default**: Created when entering a new super-block.
  - Nearly precise.

- **Fast**: Created only after synchronization events.
  - Contexts are mixed, only used for regression testing.

- **Precise**: Created on each memory access.
  - Precise stack traces for all previous accesses.

ThreadSanitizer: Data race detection in practice

Hybrid Race Detection Algorithm

Yizi Gu

Data Race Detection II
Variations of the state machine

- Pure happens-before state machine

```c
Thread1
obj->UpdateMe();
mu.Lock();
flag = true;
mu.Unlock();

Thread2
mu.Lock();
bool f = flag;
mu.Unlock();
if (f) obj->UpdateMe();
```
Variations of the state machine

**Thread1**
- obj->UpdateMe();
- mu.Lock();
- flag = true;
- mu.Unlock();

**Thread2**
- mu.Lock();
- bool f = flag;
- mu.Unlock();
- if (f) obj->UpdateMe();

- Pure happens-before state machine
  - To suppress false alarms in some subtle cases.
Variations of the state machine

Thread1
- obj->UpdateMe();
- mu.Lock();
- flag = true;
- mu.Unlock();

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

- **Pure happens-before state machine**
  - To suppress false alarms in some subtle cases.
  - Pure happens-before state machine may generate false negatives.
Variations of the state machine

Thread1

```c
obj->UpdateMe();
mu.Lock();
flag = true;
mu.Unlock();
```

Thread2

```c
mu.Lock();
bool f = flag;
mu.Unlock();
if (f) obj->UpdateMe();
```

- **Pure happens-before state machine**
  - To suppress false alarms in some subtle cases.
  - Pure happens-before state machine may generate false negatives.

- **Fast-mode state machine**
Variations of the state machine

Thread1
- obj->UpdateMe();
- mu.Lock();
- flag = true;
- mu.Unlock();

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

- Pure happens-before state machine
  - To suppress false alarms in some subtle cases.
  - Pure happens-before state machine may generate false negatives.

- Fast-mode state machine
  - Observation: Majority of memory locations are never shared.
Variations of the state machine

Thread1

- obj->UpdateMe();
- mu.Lock();
- flag = true;
- mu.Unlock();

Thread2

- mu.Lock();
- bool f = flag;
- mu.Unlock();
- if (f) obj->UpdateMe();

- Pure happens-before state machine
  - To suppress false alarms in some subtle cases.
  - Pure happens-before state machine may generate false negatives.

- Fast-mode state machine
  - Observation: Majority of memory locations are never shared.
  - Ignore all accesses to a cache line until detector sees an access from another thread.
Variations of the state machine

**Thread1**
- `obj->UpdateMe();`
- `mu.Lock();`
- `flag = true;`
- `mu.Unlock();`

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

- **Pure happens-before state machine**
  - To suppress false alarms in some subtle cases.
  - Pure happens-before state machine may generate false negatives.

- **Fast-mode state machine**
  - Observation: Majority of memory locations are never shared.
  - Ignore all accesses to a cache line until detector sees an access from another thread.
  - Reduce false positives; Hide real races.
Detector should understand the synchronization mechanisms.
Detector should understand the synchronization mechanisms.
Have to teach the detector different means of synchronizations.
Dynamic Annotation

- Detector should understand the synchronization mechanisms.
- Have to teach the detector different means of synchronizations.
- Dynamic Annotation: A C macro definition.
Detector should understand the synchronization mechanisms.

Have to teach the detector different means of synchronizations.

Dynamic Annotation: A C macro definition.

- `ANNOTATE_HAPPENS_BEFORE(ptr)`: Creates `Signal(ptr)`
Detector should understand the synchronization mechanisms.

Have to teach the detector different means of synchronizations.

Dynamic Annotation: A C macro definition.

- \texttt{ANNOTATE\_HAPPENS\_BEFORE(ptr)}: Creates Signal\(\text{(ptr)}\)
- \texttt{ANNOTATE\_HAPPENS\_AFTER(ptr)}: Creates Wait\(\text{(ptr)}\)
Performance:
Performance:
- Average slow down: 20-50 times
Performance:

- Average slow down: 20-50 times
- Extreme case:
Performance:

- Average slow down: 20-50 times
- Extreme case:
  - 10000 times: stress test.
  - 2 times: mostly sleeps or waiting IO
Performance:
  * Average slow down: 20-50 times
  * Extreme case:
    - 10000 times: stress test.
    - 2 times: mostly sleeps or waiting IO

Memory consumption decomposition:
Performance:
  - Average slow down: 20-50 times
  - Extreme case:
    - 10000 times: stress test.
    - 2 times: mostly sleeps or waiting IO

Memory consumption decomposition:
  - Constant size buffer for storing segments: $\approx 800M$. 
Experiment Results

Performance:
- Average slow down: 20-50 times
- Extreme case:
  - 10000 times: stress test.
  - 2 times: mostly sleeps or waiting IO

Memory consumption decomposition:
- Constant size buffer for storing segments: \( \approx 800M \).
- Vector time clocks attached to each segment: Quite small.
Performance:
- Average slow down: 20-50 times
- Extreme case:
  - 10000 times: stress test.
  - 2 times: mostly sleeps or waiting IO

Memory consumption decomposition:
- Constant size buffer for storing segments: $\approx 800M$.
- Vector time clocks attached to each segment: Quite small.
- Per-ID state: Could apply compression techniques.
Experiment Results

Performance:
- Average slow down: 20-50 times
- Extreme case:
  - 10000 times: stress test.
  - 2 times: mostly sleeps or waiting IO

Memory consumption decomposition:
- Constant size buffer for storing segments: \( \approx 800M \).
- Vector time clocks attached to each segment: Quite small.
- Per-ID state: Could apply compression techniques.
- Segment Sets and Locksets
Experiment Results

Performance:

- Average slow down: 20-50 times
- Extreme case:
  - 10000 times: stress test.
  - 2 times: mostly sleeps or waiting IO

Memory consumption decomposition:

- Constant size buffer for storing segments: $\approx 800\text{M}$.
- Vector time clocks attached to each segment: Quite small.
- Per-ID state: Could apply compression techniques.
- Segment Sets and Locksets

Memory consumption overhead: 3x-4x.
## Experiment Results

<table>
<thead>
<tr>
<th></th>
<th>app</th>
<th>base</th>
<th>ipc</th>
<th>net</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>native</td>
<td>3s</td>
<td>172M</td>
<td>77s</td>
<td>1811M</td>
<td>5s</td>
</tr>
<tr>
<td>Memcheck-no-hist</td>
<td>6.7x</td>
<td>2.0x</td>
<td>1.7x</td>
<td>1.1x</td>
<td>5.2x</td>
</tr>
<tr>
<td>Memcheck</td>
<td>10.5x</td>
<td>2.6x</td>
<td>2.2x</td>
<td>1.1x</td>
<td>8.2x</td>
</tr>
<tr>
<td>Helgrind-no-hist</td>
<td>13.9x</td>
<td>2.7x</td>
<td>1.8x</td>
<td>1.8x</td>
<td>5.4x</td>
</tr>
<tr>
<td>Helgrind</td>
<td>14.9x</td>
<td>3.8x</td>
<td>1.7x</td>
<td>1.9x</td>
<td>6.7x</td>
</tr>
<tr>
<td>TS-fast-no-hist</td>
<td>6.2x</td>
<td>4.2x</td>
<td>2.2x</td>
<td>1.2x</td>
<td>11.1x</td>
</tr>
<tr>
<td>TS-fast</td>
<td>7.9x</td>
<td>7.6x</td>
<td>2.4x</td>
<td>1.5x</td>
<td>12.0x</td>
</tr>
<tr>
<td>TS-full-no-hist</td>
<td>8.4x</td>
<td>4.2x</td>
<td>2.4x</td>
<td>1.2x</td>
<td>11.3x</td>
</tr>
<tr>
<td>TS-full</td>
<td>13.8x</td>
<td>7.4x</td>
<td>2.8x</td>
<td>1.5x</td>
<td>11.9x</td>
</tr>
<tr>
<td>TS-phb-no-hist</td>
<td>8.3x</td>
<td>4.2x</td>
<td>2.8x</td>
<td>1.2x</td>
<td>11.2x</td>
</tr>
<tr>
<td>TS-phb</td>
<td>14.2x</td>
<td>7.4x</td>
<td>2.6x</td>
<td>1.5x</td>
<td>11.8x</td>
</tr>
</tbody>
</table>
General Advices

- Always use mutex unless performance is critical.
General Advices

- Always use mutex unless performance is critical.
- Reuse existing standard synchronization primitives instead of re-inventing the wheel.
General Advices

- Always use mutex unless performance is critical.
- Reuse existing standard synchronization primitives instead of re-inventing the wheel.
- Wrap condition loop into a function and use dynamic annotation.
General Advices

- Always use mutex unless performance is critical.
- Reuse existing standard synchronization primitives instead of re-inventing the wheel.
- Wrap condition loop into a function and use dynamic annotation.
- Never use sleep() as synchronization between threads.
General Advices

- Always use mutex unless performance is critical.
- Reuse existing standard synchronization primitives instead of re-inventing the wheel.
- Wrap condition loop into a function and use dynamic annotation.
- Never use sleep() as synchronization between threads.
- Do not over-synchronize.
Nondeterminator-2
Nondeterminator-2
  On-the-fly dynamic race detector.
Nondeterminator-2
- On-the-fly dynamic race detector.
- All-Sets and Brelly algorithm.
Nondeterminator-2

- On-the-fly dynamic race detector.
- All-Sets and Brelly algorithm.
- No false positives; No false negatives.
Summary

- **Nondeterminator-2**
  - On-the-fly dynamic race detector.
  - All-Sets and Brelly algorithm.
  - No false positives; No false negatives.

- **ThreadSanitizer**
Summary

- **Nondeterminator-2**
  - On-the-fly dynamic race detector.
  - All-Sets and Brelly algorithm.
  - No false positives; No false negatives.

- **ThreadSanitizer**
  - On-the-fly dynamic race detector.
Summary

- **Nondeterminator-2**
  - On-the-fly dynamic race detector.
  - All-Sets and Brelly algorithm.
  - No false positives; No false negatives.

- **ThreadSanitizer**
  - On-the-fly dynamic race detector.
  - Hybrid algorithm.
Summary

- **Nondeterminator-2**
  - On-the-fly dynamic race detector.
  - All-Sets and Brelly algorithm.
  - No false positives; No false negatives.

- **ThreadSanitizer**
  - On-the-fly dynamic race detector.
  - Hybrid algorithm.
  - Make trade-off between precision and performance.
Summary

- **Nondeterminator-2**
  - On-the-fly dynamic race detector.
  - All-sets and Brelly algorithm.
  - No false positives; No false negatives.

- **ThreadSanitizer**
  - On-the-fly dynamic race detector.
  - Hybrid algorithm.
  - Make trade-off between precision and performance.

- **FastTrack**
Nondeterminator-2
- On-the-fly dynamic race detector.
- All-Sets and Brelly algorithm.
- No false positives; No false negatives.

ThreadSanitizer
- On-the-fly dynamic race detector.
- Hybrid algorithm.
- Make trade-off between precision and performance.

FastTrack
- On-the-fly dynamic race detector.
Summary

- **Nondeterminator-2**
  - On-the-fly dynamic race detector.
  - All-Sets and Brelly algorithm.
  - No false positives; No false negatives.

- **ThreadSanitizer**
  - On-the-fly dynamic race detector.
  - Hybrid algorithm.
  - Make trade-off between precision and performance.

- **FastTrack**
  - On-the-fly dynamic race detector.
  - Aims at improving performance.
Nondeterminator-2
- On-the-fly dynamic race detector.
- All-Sets and Brelly algorithm.
- No false positives; No false negatives.

ThreadSanitizer
- On-the-fly dynamic race detector.
- Hybrid algorithm.
- Make trade-off between precision and performance.

FastTrack
- On-the-fly dynamic race detector.
- Aims at improving performance.
- Represents happens-before in a more concise way.


