Hybrid Dynamic Data Race Detection

Robert O’Callahan
Jong-Deok Choi
IBM T.J. Watson Research Center

ACM SIGPLAN 2003 Symposium on
Principles and Practice of Parallel Programming
(PPoPP’03)
San Diego, CA, June 11–13
Key contributions

Presents a method for dynamic detection of data races

• Synergistically combines \textit{lockset-based detection} and \textit{happens before-based detection}
  – Fewer false positives than just lockset-based detection
  – Fewer false negatives than just happens before-based detection
  – Little extra computational expense over lockset-based detection

• Provides more diagnostic information about potential race
  – Including stack traces for both threads involved in the conflicting events

• Improves scalability of race detection
  – Eliminates need for whole-program static analysis to justify optimizations
  – Instead:
    ◦ Perform dynamic optimizations: \textit{thread-locality check} and \textit{over-sized lockset check} to identify \textit{redundant events}
    ◦ Run program twice with different instrumentation in each run
Program execution model

Scenario: Their race detector observes a stream of events generated by instrumentation inserted into the program and serialized via locks inside the detector.

This scenario justifies:

A *program* is an abstract machine that outputs a sequence of events to the detector.

The *components* of an event are abstracted by:

- A set of *memory locations*, \( \mathcal{M} \)
- A set of *locks*, \( \mathcal{L} \)
- A set of *threads*, \( \mathcal{T} \)
- A set of *message IDs*, \( \mathcal{G} \)
- A set of *access types*, \( \mathcal{A} = \{ \text{RD, WR} \} \)
Program execution model

Notational conventions:

\[ m, m_i \in M \quad l, l_i \in L \quad t, t_i \in T \quad g, g_i \in G \quad a, a_i \in A \]

\[ e, e_i \text{ denote events} \]

The events generated by a program are abstracted by:

- **MEM**(\(m, a, t\)): thread \(t\) performed an access of type \(a\) to location \(m\).
- **ACQ**(\(l, t\)): thread \(t\) acquired lock \(l\) (and did not already hold \(l\)).
- **REL**(\(l, t\)): thread \(t\) released lock \(l\) (and no longer holds \(l\)).
- **SND**(\(g, t\)): thread \(t\) sent the message \(g\).
- **RCV**(\(g, t\)): thread \(t\) received the message \(g\).

Latter two types are observed exclusively by the happens-before detector; lockset detector observes only memory accesses and lock events.
Example: events generated (for lockset algorithm)

// MAIN THREAD
class Main {
    int gflag;
    CThr cthr;
    void execute() {
        gflag = 1;
        // MEM(main.gflag,WT,MAIN)
        cthr = new CThr(this);
        // MEM(main.cthr,WT,MAIN)
        cthr.start();
        // MEM(main.cthr,RD,MAIN)
        ...
        synchronized (this) {
            // ACQ(main,MAIN)
            L: chthr.interrupt();
            // MEM(main.cthr,RD,MAIN)
        }
        // REL(main,MAIN)
    }
}

// CTHR THREAD
class CThr extends Thread {
    Main main;
    CThr(Main main) {
        this.main = main;
        // MEM(c thr.main,WT,CTHR)
    }
    void run() {
        if (main.gflag == 1) ...
        // MEM(main.gflag,RD,CTHR)
        ...
        main.cthr = null;
        // MEM(main.cthr,WT,CTHR)
    }
}
Lockset-based detection: preliminaries

Given an event sequence $<e_i>_{i=1,2,...}$ the locks before step $i$ held by a thread $t$, $L_i(t)$, is defined as:

$$L_i(t) = \{ l \mid (\exists j \cdot j < i \land e_i = \text{ACQ}(l, t) \land \left( \forall k \cdot j < k < i \Rightarrow e_k \neq \text{REL}(l, t) \right)) \}$$

The lockset hypothesis: whenever two threads access the same location and one of the accesses is a WT, the accesses are performed holding a common lock.

$$\text{PossLSR}(i, j) =$$

$$e_i = \text{MEM}(m, a_i, t_i) \land e_j = \text{MEM}(m, a_j, t_j) \land t_i \neq t_j \land$$

$$(a_i = \text{WT} \lor a_j = \text{WT}) \land L_i(t_i) \cap L_j(t_j) = \emptyset$$
Lockset-based detection: the algorithm

Simple version: for each \( m \in \mathcal{M} \), maintains

- a set \( S_m \) of tuples \((a, t, L)\) representing an access of type \( a \) to location \( m \) by thread \( t \) while holding lockset \( L \).
- a flag \( checking_m \) indicating whether accesses to \( m \) are being checked

Given an event sequence \(<e_i>\) \( i = 1, 2, ... \)

For each \( m \in \mathcal{M} \) do: initialize \( S_m = \emptyset \) \( \land \) \( checking_m = true \)

For each \( e_i \in <e_i> \) \( i = 1, 2, ... \) do:

if ( \( e_i = \text{MEM}(m, a, t) \land checking_m \) ) {
    if ( \((a, t, L_i(t)) \in S_m \) ) {   \( \land \) \( e_i \) is redundant, ignore it
    }
else if ( \( \exists (a', t', L') \in S_m \) .  \( \land \) \( e_i \) is a race
    \((a = \text{WT} \lor a' = \text{WT}) \land t \neq t' \land L_i(t) \cap L' = \emptyset \) ){
    report a race
    \( checking_m = false \)
}
else {  \( \land \) \( e_i \) remember the new tuple
    \( S_m = S_m \cup \{ (a, t, L_i(t)) \} \)
}
}
## Lockset-based detection: example

<table>
<thead>
<tr>
<th></th>
<th>// M THREAD (MAIN)</th>
<th>( L_i(m) )</th>
<th>// C THREAD (CTHR)</th>
<th>m.gflag</th>
<th>m.cthr</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_1 )</td>
<td>gflag = 1;</td>
<td>( \emptyset )</td>
<td>m.gflag</td>
<td>(WT, 0, M)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>// MEM(m.gflag,WT,M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e_2 )</td>
<td>cthr = new CThr(this);</td>
<td></td>
<td></td>
<td></td>
<td>(WT, 0, M)</td>
</tr>
<tr>
<td></td>
<td>// MEM(m.cthr,WT,M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e_3 )</td>
<td>cthr.start();</td>
<td></td>
<td></td>
<td></td>
<td>(WT, 0, M)</td>
</tr>
<tr>
<td></td>
<td>// MEM(m.cthr,RD,M)</td>
<td></td>
<td></td>
<td></td>
<td>(RD, 0, M)</td>
</tr>
<tr>
<td>( e_4 )</td>
<td>this.m = m;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>// MEM(c.m,WT,C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e_5 )</td>
<td>if (m.gflag == 1)</td>
<td></td>
<td></td>
<td>PossLSR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>// MEM(m.gflag,RD,C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e_6 )</td>
<td>synchronized (this){</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>// ACQ(m,M)</td>
<td>{ m }</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e_7 )</td>
<td>m.cthr = null;</td>
<td></td>
<td></td>
<td>PossLSR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>// MEM(m.cthr,WT,C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lockset-based detection: false positives

Violations of the lockset hypothesis are not necessarily synchronization errors

- In the example, the reported PossLSR on $m.gflag$ is not a true race since $gflag = 1$ necessarily happens before $cthr.start()$ in Main, while the latter necessarily happens before evaluation of $m.gflag == 1$ in CThr.

- More generally, programs may use message passing for synchronization instead of locks.
Lockset-based detection: false positives

// THREAD A
BigObj big = new BigObj();
big.init(...);
...
pool.recycle(big);

// THREAD B
BigObj big = pool.removeOne();
big.init(...);
...
pool.recycle(big);

Since threads access big without any locks, a potential race will be reported.
Happens before-based detection: preliminaries

Key idea: \( e_i \) happens before \( e_j \) if communication between threads could allow information to be transferred from \( e_i \) to \( e_j \).

Message send and receive events are introduced to capture this relation.

Formally, given an event sequence \(<e_i>_{i=1,2,...}\), the happens-before relation, \( \rightarrow \), is the smallest relation satisfying

- if Thread(\( e_i \)) = Thread(\( e_j \)) and \( i < j \), then \( i \rightarrow j \)
- if \( e_i = \text{SND}(g,t_1) \) \( \land \) \( e_j = \text{RCV}(g,t_2) \), then \( i \rightarrow j \)
- if \( i \rightarrow j \) \( \land \) \( j \rightarrow k \), then \( i \rightarrow k \)

It is defined over indices because the same event may occur many times in an execution and each occurrence has its own happens-before relationships.
Happens before-based detection: messages

Key idea: Want a message to be sent from $t_1$ to $t_2$ whenever $t_1$ takes an action that later affects $t_2$.

Capture explicit synchronization in Java program, e.g,
- When $t_1$ calls $t_2$.start(), generate a $\text{SND}(g, t_1)$ and, when $t_2$ first executes in $\text{run()}$, generate a $\text{RCV}(g, t_2)$, using a fresh $g$
- When $t_1$ calls $\text{obj.notify()}$, generate a $\text{SND}(g, t_1)$ and also $\text{RCV}(g, t_2)$’s, for all $t_2$ waiting on $\text{obj}$, using a fresh $g$.

Capture interactions arising from shared memory accesses
- After $\text{MEM}(m, \text{WT}, t)$ generate a $\text{SND}(g, t)$ using a fresh $g$.
- After $\text{MEM}(m, \text{RD}, t)$ generate a $\text{SND}(g, t)$ using the $g$ sent by the most recent writer to $m$. (I’m not generating this if most recent writer is $t$. I think that’s OK.)

Capture interactions arising from locking
- After $\text{REL}(l, t)$ generate a $\text{SND}(g, t)$ using a fresh $g$.
- After (all but the first) $\text{ACQ}(l, t)$ generate a $\text{SND}(g, t)$ using the $g$ sent by the most recent $\text{REL}(l, t)$.
### Happens before-based detection: messages

```java
// M THREAD (MAIN)
gflag = 1;
// MEM(m.gflag,WT,M)
// SND(gflag1,M)
cthr = new CThr(this);
// MEM(m.cthr,WT,M)
// SND(cthr1,M)
cthr.start();
// MEM(m.cthr,RD,M)
// SND(start,M)
synchronized (this){
    // ACQ(m,M)
    L: chthr.interrupt();
}
```

```java
// C THREAD (CTHR)

this.m = main; not checking
// RCV(start,C)
if (m.gflag == 1)
    // MEM(m.gflag,RD,C)
    // RCV(gflag1,C)
main.cthr = null;
// MEM(m.cthr,WT,C)
// SND(cthr2,C)
```
Happens before-based detection

A potential race occurs if two events by different threads access the same location, at least one access is a write, and the accesses are not causally ordered by the happens-before relation.

Formally:

$$\text{PossHBR}(i, j) =$$

$$e_i = \text{MEM}(m, a_i, t_i) \land e_j = \text{MEM}(m, a_j, t_j) \land t_i \neq t_j \land$$

$$(a_i = \text{WT} \lor a_j = \text{WT}) \land L_i(t_i) \cap L_j(t_j) = \emptyset \land i \not\rightarrow j \land j \not\rightarrow i$$

The happens-before relation is computed on-line using vector-clocks
Happens before-based detection: vector clocks

Each $t$ maintains a vector clock indexed by thread IDs:
- the entry for $t'$ indicates the last event in $t$ that could have influenced $t$.

Each $g$ carries a vector clock $V(g)$
- the vector clock of the sending thread at the time $g$ is sent.

Given an event sequence $<e_i>_{i=1,2,...}$ the vector clock $V_i(t)$ of $t$ at the completion of $e_i$ is defined inductively.
- Initial vector clock: $V_0(t)(t) = 1$ and $V_0(t)(t') = 0$, for $t \neq t'$
- Subsequently:
  - If $e_{i-1} = \text{SND}(g,t)$, then $V_i(t)(t) = V_{i-1}(t)(t) + 1$
  - If $e_i = \text{RCV}(g,t)$ and $t \neq t'$, then $V_i(t)(t') = \max\{V_{i-1}(t)(t'), V(g)(t')\}$
  - Otherwise, $V_i(t)(t') = V_{i-1}(t)(t')$

Additionally, $V(g) = V_i(t)$, where $e_i = \text{SND}(g,t)$
Happens before-based detection: vector clocks

\[ i \rightarrow j \text{ iff } i < j \text{ and } V_j(t_j)(t_i) \geq V_i(t_i)(t_i) \text{ where } t_i = \text{Thread}(e_i) \text{ and } t_j = \text{Thread}(e_j) \]

<table>
<thead>
<tr>
<th></th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_0 )</td>
<td>(1,0,0)</td>
<td>(0,1,0)</td>
<td>(0,0,1)</td>
</tr>
<tr>
<td>( V_1 )</td>
<td>(1,0,0)</td>
<td>(0,1,0)</td>
<td>(0,0,1)</td>
</tr>
<tr>
<td>( V_2 )</td>
<td>(2,0,0)</td>
<td>(0,1,0)</td>
<td>(0,0,1)</td>
</tr>
<tr>
<td>( V_3 )</td>
<td>(2,0,0)</td>
<td>(0,2,0)</td>
<td>(0,1,1)</td>
</tr>
<tr>
<td>( V_4 )</td>
<td>(2,0,0)</td>
<td>(0,2,0)</td>
<td>(0,1,1)</td>
</tr>
<tr>
<td>( V_5 )</td>
<td>(2,0,0)</td>
<td>(1,2,0)</td>
<td>(0,1,2)</td>
</tr>
<tr>
<td>( V_6 )</td>
<td>(2,1,1)</td>
<td>(1,2,0)</td>
<td>(0,1,2)</td>
</tr>
</tbody>
</table>
Relation between HBD and LSD

If a full HB detector reports a race, then there are thread schedules in which the two events occur consecutively in time in either order.

The potential races reported by a full HB detector are a subset of those reported by a LS detector.

A full HB detector is subject to more false negatives than a LS detector—i.e., it may fail to detect real races that would be detected by a LS detector.

• Core problem: not all happens-before relationships correspond to true synchronization.
Example false negative introduced by HB

// THREAD A

global = 7; // * //
// MEM(global,WT,A)
...
synchronized (l) {
    // ACQ(l,A)
    ctr ++;
    ...
}
// REL(l,A)
// SND(l1,A)

// THREAD B

synchronized (l) {
    // ACQ(l,B)
    // RCV(l1,B)
    ctr++
}
// REL(l,B)
// SND(l2,B)
int x = global; // ** //
// MEM(global,RD,B)

Here, the happens-before relation masks the true race between the accesses at * and **
Hybrid race detection

Add limited HB checking to a LS detector

- Generate messages for `start()`, `join()`, `wait()`, `notify()`
- No messages generated for memory access
- No messages generated for locking

Relation computed using limited HB information is a subset of the full HB relation; use it instead of $\rightarrow$ to compute $\text{PossHybridRace}(i,j)$.

Observations

- Significantly fewer messages means significantly fewer vector clock updates, which are expensive; HB checks are cheap
- Store thread’s local time (not full vector clock) for all stored tuples:
  
  - $(a,t,L,v)$, for $e_i = \text{MEM}(m,a,t)$, $L = L_i(t)$, $v = V_i(t)(t)$
- Maintain full vector clocks for only active threads and pending messages
Hybrid race detection: the algorithm

Simple version: for each \( m \in M \), maintains

- a set \( S_m \) of tuples \((a, t, L, v)\) representing an access of type \( a \) to location \( m \) by thread \( t \) at local time \( v \) while holding lockset \( L \).
- a flag \( \text{checking}_m \) indicating whether accesses to \( m \) are being checked

Given an event sequence \(<e_i>_{i=1,2,...}\)

For each \( m \in M \) do: initialize \( S_m = \emptyset \land \text{checking}_m = \text{true} \)

For each \( e_i \in <e_i>_{i=1,2,...} \) do:

if \( (e_i = \text{MEM}(m, a, t) \land \text{checking}_m) \) { 
  if \( ((a, t, L_i(t), V_i(t)(t)) \in S_m) \) \{ \hspace{1cm} \text{// \( e_i \) is redundant, ignore it} 
  \}
else if \( \exists (a', t', L', v') \in S_m \). \hspace{1cm} \text{// \( e_i \) is a race} 
  (a = \text{WT} \lor a' = \text{WT}) \land t \neq t'(t) \land L_i(t) \cap L' = \emptyset \land V_i(t)(t') < v') \{ 
  \text{report a race} 
  \text{checking}_m = \text{false} 
\}
else { \hspace{1cm} \text{// remember the new tuple} 
  S_m = S_m \cup \{(a, t, L_i(t), V_i(t)(t))\} 
\}
}
### Hybrid detection: example

<table>
<thead>
<tr>
<th>// M THREAD (MAIN)</th>
<th>// C THREAD (CTHR)</th>
<th>m.gflag</th>
<th>m.cthr</th>
<th>V(M)</th>
<th>V(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gflag = 1;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>// MEM(m.gflag,WT,M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cthr = new CThr...;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>// MEM(m.cthr,WT,M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cthr.start();</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>// MEM(m.c thr,RD,M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>// SND(start,M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>this.m = m;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>// RCV(start,M)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>// MEM(c.m,WT,C)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (m.gflag == 1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>// MEM(m.gflag,RD,C)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronized (this){</td>
<td></td>
<td>m.cthr = null;</td>
<td></td>
<td>PossHR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>// MEM(m.c thr,WT,C)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hybrid race detection: optimizations

- Heuristics to identify redundant events.
  - Lockset-subset condition: If the thread, access type, and time-stamp of a previous event $e_i$ match those of a new event $e_j$ and if $e_i$’s lockset is a subset of $e_j$’s lockset, then is $e_j$ redundant.
  - Oversized lockset condition:
    ◊ Based on two observations (see paper for details)
      1. The number of threads held by any thread at any time tends to be small.
      2. As the algo collects more non-racing, non-redundant accesses to a location, for a new access to be non-redundant, the size of the lockset of future racing events must be large.
    ◊ Helps reduce overhead of checking accesses of “read only” data, which threads tend to access while holding various unrelated locks.
Hybrid race detection: optimizations

• Two-phase mode selection
  – Run detector first in “simple” mode to identify data that potentially incur races.
    ◇ In this mode, detector performs LS-based detection equivalent to Eraser
    ◇ A thread-locality check is performed by extending each field with an owning thread, which is set to the first thread that accesses the field; access by a non-owning thread triggers logging of accesses.
  – Rerun the detector in “detailed” mode, instrumenting accesses to only “race-prone” fields as detected in simple mode.
Hybrid race detection: experimental results

“... our tool achieves very precise results with reasonable slowdown”

See the paper for details. :-(