FastTrack: Efficient and Precise Dynamic Race Detection

Cormac Flanagan
Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064
Stephen N. Freund

Computer Science Department
Williams College
Williamstown, MA 01267
Background

Multithreaded Programs: Race Conditions

Race Condition:
Two concurrent accesses to the same data, and at least one access is a write
One of the most common concurrency bugs
Solutions

Dynamic race detector:

- Fast but imprecise race detectors
  - Lockset based approaches
    - Imprecise (false positive)
    - Eraser

- Slow but precise race detectors
  - Vector-clock based approaches
    - Precise
    - Fast Track
Multithreaded Program Traces

Figure 1: Multithreaded Program Traces

\[ \alpha \in \text{Trace} \quad = \quad \text{Operation}^* \]

\[ a, b \in \text{Operation} = \quad \text{rd}(t, x) \mid \text{wr}(t, x) \]
\[ \quad \mid \text{acq}(t, m) \mid \text{rel}(t, m) \]
\[ \quad \mid \text{fork}(t, u) \mid \text{join}(t, u) \]

\[ s, u, t \in \text{Tid} \quad x, y \in \text{Var} \quad m \in \text{Lock} \]

- \text{rd}(t, x) and \text{wr}(t, x), which read and write a value from \( x \);
- \text{acq}(t, m) and \text{rel}(t, m), which acquire and release a lock \( m \);
- \text{fork}(t, u), which forks a new thread \( u \); and
- \text{join}(t, u), which blocks until thread \( u \) terminates.
Happened-before relation

happens-before relation $a <_\alpha b$ holds

- **Program order**: The two operations are performed by the same thread.
- **Locking**: The two operations acquire or release the same lock.
- **Fork-join**: One operation is $fork(t, u)$ or $join(t, u)$ and the other operation is by thread $u$.

If two operations in a trace are not related by the happened-before relation, then they are considered concurrent. Two memory access conflict if they both access (read or write) the same variable, and at least one of the operations is a write.
Vector Clock (VC): records a clock for each thread in the system

\[ V_1 \subseteq V_2 \iff \forall t. \ V_1(t) \leq V_2(t) \]
\[ V_1 \sqcup V_2 = \lambda t. \ \max(V_1(t), V_2(t)) \]
\[ \bot_V = \lambda t. \ 0 \]
\[ \text{inc}_t(V) = \lambda u. \ \text{if} \ u = t \ \text{then} \ V(u) + 1 \ \text{else} \ V(u) \]
$DJIT^{\uparrow+}$ Algorithm

\[
\begin{array}{c|c|c|c}
C_0 & C_1 & L_m & W_x \\
\hline
\langle 4,0,... \rangle & \langle 0,8,... \rangle & \langle 0,0,... \rangle & \langle 0,0,... \rangle \\
\hline
\downarrow wr(0,x) & & & \\
\langle 4,0,... \rangle & \langle 0,8,... \rangle & \langle 0,0,... \rangle & \langle 4,0,... \rangle \\
\hline
\downarrow rel(0,m) & & & \\
\langle 5,0,... \rangle & \langle 0,8,... \rangle & \langle 4,0,... \rangle & \langle 4,0,... \rangle \\
\hline
\downarrow acq(1,m) & & & \\
\langle 5,0,... \rangle & \langle 4,8,... \rangle & \langle 4,0,... \rangle & \langle 4,0,... \rangle \\
\hline
\downarrow wr(1,x) & & & \\
\langle 5,0,... \rangle & \langle 4,8,... \rangle & \langle 4,0,... \rangle & \langle 4,8,... \rangle \\
\end{array}
\]

$W_x = \langle 4,0,... \rangle \subseteq \langle 4,8,... \rangle = C_1$

Since this check passes, the two writes are not concurrent, and no race condition is reported.
The FASTTRACK Algorithm

Reads and Writes account for over 96% of monitored operations.

The Key insight behind Fast-Track is that the full generality of vector clocks is not necessary in over 99% of these read and write operations: a more light weight representation of the happens before information can be used instead.
The FASTTRACK Algorithm

*epoch*

We refer to a pair of a clock $c$ and a thread $t$ as an epoch, denoted $c@t$.

An epoch $c@t$ happens before a vector clock $V$ ($c@t \preceq V$) if and only if the clock of the epoch is less than or equal to the corresponding clock in the vector.

$$c@t \preceq V \quad \text{iff} \quad c \leq V(t)$$
The FASTTRACK Algorithm

At the first write to $x$, FASTTrack performs an $O(1)$-time epoch write $W_x := 4@0$. FASTTrack subsequently ensures that the second write is not concurrent with the preceding write via the $O(1)$-time comparison:

$$W_x = 4@0 \leq \langle 4,8,... \rangle = C_1$$
Figure 2: **FASTTrack Race Detection Algorithm** and its Comparison to DJIT$^+$.  

<table>
<thead>
<tr>
<th><strong>FASTTrack State:</strong></th>
<th><strong>DJIT$^+$ State:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$ : $Tid \rightarrow VC$</td>
<td>$C$ : $Tid \rightarrow VC$</td>
</tr>
<tr>
<td>$L$ : $Lock \rightarrow VC$</td>
<td>$L$ : $Lock \rightarrow VC$</td>
</tr>
<tr>
<td>$W$ : $Var \rightarrow Epoch$</td>
<td>$W$ : $Var \rightarrow VC$</td>
</tr>
<tr>
<td>$R$ : $Var \rightarrow (Epoch \cup VC)$</td>
<td>$R$ : $Var \rightarrow VC$</td>
</tr>
</tbody>
</table>

Figure 2 presents the key details of how **FASTTrack** (left column) and **DJIT$^+$** (right column) handle read and write operations of the target program. Expensive $O(n)$-time operations are highlighted in grey. That table also shows the instruction frequencies observed in our program traces, as well as how frequently each rule was applied. For example, 82.3% of all memory and synchronization operations performed by our benchmarks were reads, and rule [FT READ SAME EPOCH] was used to check 63.4% of those reads.
Comparison

Reads: □ 82.3% of all Operations

[FT read same epoch]
\[
\begin{align*}
R_x &= E(t) \\
(C, L, R, W) &\Rightarrow^{rd(t,x)} (C, L, R, W)
\end{align*}
\]
63.4% of reads

[FT read shared]
\[
\begin{align*}
R_x &\in VC \\
W_x &\preceq C_t \\
R' &= R[x := R_x[t := C_t(t)]] \\
(C, L, R, W) &\Rightarrow^{rd(t,x)} (C, L, R', W)
\end{align*}
\]
20.8% of reads

[FT read exclusive]
\[
\begin{align*}
R_x &\in \text{Epoch} \\
R_x &\preceq C_t \\
W_x &\preceq C_t \\
R' &= R[x := E(t)] \\
(C, L, R, W) &\Rightarrow^{rd(t,x)} (C, L, R', W)
\end{align*}
\]
15.7% of reads

[DJIT + read same epoch]
\[
\begin{align*}
R_x(t) &= C_t(t) \\
(C, L, R, W) &\Rightarrow^{rd(t,x)} (C, L, R, W)
\end{align*}
\]
78.0% of reads

[DJIT + read]
\[
\begin{align*}
W_x &\subseteq C_t \\
R' &= R[x := R_x[t := C_t(t)]] \\
(C, L, R, W) &\Rightarrow^{rd(t,x)} (C, L, R', W)
\end{align*}
\]
22.0% of reads
Comparison

Writes: \(14.5\%\) of all Operations

[FT WRITE SAME EPOCH]
\[
W_x = E(t) \\
(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W)
\]
71.0% of writes

[FT WRITE EXCLUSIVE]
\[
R_x \in\ Epoch \\
R_x \subseteq C_t \\
W_x \subseteq C_t \\
W' = W[x := E(t)] \\
(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W')
\]
28.9% of writes

[FT WRITE SHARED]
\[
R_x \in\ VC \\
R_x \subseteq C_t \\
W_x \subseteq C_t \\
W' = W[x := E(t)] \\
R' = R[x := \bot_e] \\
(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R', W')
\]
0.1% of writes

[Djit\(^+\) WRITE SAME EPOCH]
\[
W_x(t) = C_t(t) \\
(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W)
\]
71.0% of writes

[Djit\(^+\) WRITE]
\[
W_x \subseteq C_t \\
R_x \subseteq C_t \\
W' = W[x := W_{x'}[t := C_t(t)]] \\
(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W')
\]
29.0% of writes
Other: 3.3% of all Operations

[FT ACQUIRE]
\[
C' = C[t := (C_t \sqcup L_m)]
\]
\[
(C, L, R, W) \Rightarrow_{acq(t,m)} (C', L, R, W)
\]

[FT RELEASE]
\[
L' = L[m := C_t]
C' = C[t := inc_t(C_t)]
\]
\[
(C, L, R, W) \Rightarrow_{rel(t,m)} (C', L', R, W)
\]

[FT FORK]
\[
C'' = C[u := C_u \sqcup C_t, t := inc_t(C_t)]
\]
\[
(C, L, R, W) \Rightarrow_{fork(t,u)} (C'', L, R, W)
\]

[FT JOIN]
\[
C'' = C[t := C_t \sqcup C_u, u := inc_u(C_u)]
\]
\[
(C', L, R, W) \Rightarrow_{join(t,u)} (C'', L, R, W)
\]
Example Trace

- Rx is $\perp_e$ indicating that x has not yet been read.
- After the first read operation $rd(1; x)$, Rx becomes the epoch 1@1 recording both the clock and the thread identifier of that read
- The second read $rd(0; x)$ at clock 8
- After the two threads join, the write operation $wr(0; x)$ happens after all reads.
- The last read in the trace then sets Rx to a non-minimal epoch.
Implementation

• Component of ROADRUNNER

```java
class ThreadState {
    int tid;
    int C[];
    int epoch;  // invariant: epoch == C[tid]
}

class VarState {
    int W, R;
    int Rvc[];  // used iff R == READ_SHARED
}

class LockState {
    int L[];
}

void read(VarState x, ThreadState t)
if (x.R == t.epoch) return;  // Same Epoch 63.4%
// write-read race?
if (x.W > t.C[TID(x.W)]) error;
// update read state
if (x.R == READ_SHARED) {
    x.Rvc[t.tid] = t.epoch;  // Shared 20.8%
} else {
    if (x.R <= t.C[TID(x.R)]) {
        x.R = t.epoch;  // Exclusive 15.7%
    } else {
        if (x.Rvc == null) {
            x.Rvc = newClockVector();  // (SLOW PATH)
            x.Rvc[TID(x.R)] = x.R;
            x.Rvc[t.tid] = t.epoch;
            x.R = READ_SHARED;
        } else {
            x.Rvc[t.tid] = t.epoch;
            x.R = READ_SHARED;
        }
    }
}

void write(VarState x, ThreadState t)
if (x.W == t.epoch) return;  // Same Epoch 71.0%
// write-write race?
if (x.W > t.C[TID(x.W)]) error;
// read-write race?
if (x.R != READ_SHARED) {
    if (x.R > t.C[TID(x.R)]) error;
} else {
    // Exclusive 0.1%
    if (x.Rvc[u] > t.C[u] for any u) error;  // (SLOW PATH)
}
x.W = t.epoch;  // update write state
```
<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>luifact</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>mirt</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>2.6</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>4</td>
<td>0.0</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>hec*</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>
</tbody>
</table>

**Average**

|              | 4.1 | 8.6 | 21.7 | 31.6 | 24.2 | 89.8 | 20.2 | 8.5 | 27 | 5 | 3 | 8 | 8 | 8 |

**Table 1: Benchmark Results.** Programs marked with "*" are not compute-bound and are excluded when computing average slowdowns.
## Result

<table>
<thead>
<tr>
<th>Program</th>
<th>Vector Clocks Allocated</th>
<th>Vector Clock Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIJT⁺</td>
<td>FAST TRACK</td>
</tr>
<tr>
<td>colt</td>
<td>849,765</td>
<td>76,209</td>
</tr>
<tr>
<td>crypt</td>
<td>17,332,725</td>
<td>119</td>
</tr>
<tr>
<td>lufact</td>
<td>8,024,779</td>
<td>2,715,630</td>
</tr>
<tr>
<td>moldyn</td>
<td>849,397</td>
<td>26,787</td>
</tr>
<tr>
<td>montecarlo</td>
<td>457,647,007</td>
<td>25</td>
</tr>
<tr>
<td>mtrt</td>
<td>2,763,373</td>
<td>40</td>
</tr>
<tr>
<td>raja</td>
<td>1,498,557</td>
<td>3</td>
</tr>
<tr>
<td>raytracer</td>
<td>160,035,820</td>
<td>14</td>
</tr>
<tr>
<td>sparse</td>
<td>31,957,471</td>
<td>456,779</td>
</tr>
<tr>
<td>series</td>
<td>3,997,307</td>
<td>13</td>
</tr>
<tr>
<td>sor</td>
<td>2,002,115</td>
<td>5,975</td>
</tr>
<tr>
<td>tsp</td>
<td>311,273</td>
<td>397</td>
</tr>
<tr>
<td>elevator</td>
<td>1,678</td>
<td>207</td>
</tr>
<tr>
<td>philo</td>
<td>56</td>
<td>12</td>
</tr>
<tr>
<td>hdec</td>
<td>886</td>
<td>82</td>
</tr>
<tr>
<td>jbb</td>
<td>109,544,709</td>
<td>1,859,828</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>796,816,918</td>
<td>5,142,120</td>
</tr>
</tbody>
</table>

*Table 2: Vector Clock Allocation and Usage.*
<table>
<thead>
<tr>
<th>Program</th>
<th>Memory (MB)</th>
<th>Memory Overhead</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fine</td>
<td>Coarse</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DJIT+ FAST TRACK</td>
<td>DJIT+ FAST TRACK</td>
</tr>
<tr>
<td>colt</td>
<td>36</td>
<td>4.3 2.4</td>
<td>2.0 1.8</td>
</tr>
<tr>
<td>crypta</td>
<td>41</td>
<td>44.3 10.5</td>
<td>1.2 1.2</td>
</tr>
<tr>
<td>lufacte</td>
<td>80</td>
<td>9.8 4.1</td>
<td>1.1 1.1</td>
</tr>
<tr>
<td>moldynb</td>
<td>37</td>
<td>3.3 1.7</td>
<td>1.3 1.2</td>
</tr>
<tr>
<td>montecarlob</td>
<td>595</td>
<td>6.1 2.1</td>
<td>1.1 1.1</td>
</tr>
<tr>
<td>mtrt</td>
<td>51</td>
<td>3.9 2.2</td>
<td>2.6 1.9</td>
</tr>
<tr>
<td>raja</td>
<td>35</td>
<td>1.3 1.3</td>
<td>1.2 1.3</td>
</tr>
<tr>
<td>raytracerb</td>
<td>36</td>
<td>6.2 1.9</td>
<td>1.4 1.2</td>
</tr>
<tr>
<td>sparsec</td>
<td>131</td>
<td>23.3 6.1</td>
<td>1.0 1.0</td>
</tr>
<tr>
<td>serisc</td>
<td>51</td>
<td>8.5 3.1</td>
<td>1.1 1.1</td>
</tr>
<tr>
<td>soc</td>
<td>40</td>
<td>5.3 2.1</td>
<td>1.1 1.1</td>
</tr>
<tr>
<td>tsp</td>
<td>33</td>
<td>1.7 1.3</td>
<td>1.2 1.2</td>
</tr>
<tr>
<td>elevator+</td>
<td>32</td>
<td>1.2 1.2</td>
<td>1.2 1.2</td>
</tr>
<tr>
<td>philo+</td>
<td>32</td>
<td>1.2 1.2</td>
<td>1.2 1.2</td>
</tr>
<tr>
<td>hede+</td>
<td>33</td>
<td>1.4 1.4</td>
<td>1.3 1.3</td>
</tr>
<tr>
<td>jbb+</td>
<td>236</td>
<td>4.1 2.4</td>
<td>2.3 1.9</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>7.9 2.8</td>
<td>1.4 1.3</td>
</tr>
</tbody>
</table>
FASTTRACK is a new precise race detection algorithm that achieves better performance than existing algorithms by tracking less information and dynamically adapting its representation of the happens-before relation based on memory access patterns. The FASTTRACK algorithm and adaptive epoch representation is also straightforward to implement, and may be useful in other dynamic analyses for multithreaded software.
Conclusion

- It provides a significant improvement in precision over earlier, imprecise race detectors such as Eraser [33], while providing comparable performance.

- Despite its efficiency, it is still a comparatively simple algorithm that is straightforward to implement. It uses an adaptive lightweight representation for the happens-before relation that reduces both time and space overheads.

- It contains optimized constant-time fast paths that handle upwards of 96% of the operations in benchmark programs.

- It provides a 2.3x performance improvement over the prior DJIT+ algorithm, and typically incurs less than half the memory overhead of DJIT+.

- FASTTRACK also improves the performance of more heavyweight dynamic analysis tools by identifying millions of irrelevant, race-free memory accesses that can be ignored. It provides a 5x speedup for the VELODROME dynamic atomicity checker [17] and an 8x speedup for the SINGLETRACK determinism checker [32].