Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support

Serdar Tasiran
Koc University, Istanbul, Turkey
Microsoft Research, Redmond

Hassan Salehe Matar, Ismail Kuru,
Koc University, Istanbul, Turkey

Roman Dementiev
Intel, Munich, Germany
This Work

- Races are bad
  - Cause non-deterministic execution, sequential consistency violations
  - Symptomatic of higher-level programming errors
  - Difficult to catch and debug
- Precise, dynamic detection of races is good
  - Useful, timely debugging information
  - No false alarms
- But dynamic race detection is too expensive: 10X for Java, 100X for C/C++
- Dynamic race detection slowdown
  - **Instrumentation:** Detecting memory and synchronization accesses
  - **Analysis:** The race detection algorithm
    - **Computation:** Update, compare vector clocks, locksets
    - **Synchronization:** Synchronize vector clocks, locksets
    - **This work:** Reduce this using Intel Hardware Transactional Memory support
What this talk is NOT about

• This work is about
  – using Intel hardware transactional memory support to make dynamic race detection in lock-based applications faster

• This work is **not** about
  – replacing lock-based synchronization in applications with hardware transactional memory instead
  – race detection for applications that use transactional memory (and maybe locks)
  – Using hardware transactional memory purely for conflict detection/avoidance
    • Are there conflicting accesses to the same address by two different threads “at the same time”? although our experimental results will give some indication of how successful these approaches might be
This Work in Context

• Goldilocks: PLDI ‘07, CACM ’10
  – DataRaceException is a good idea for Java
  – Needs to be supported by continuous, precise run-time happens-before race detection
  – Later work, by others: Hardware support for concurrency exceptions

• Why **precise**: Tools with too many false alarms do not get used
• Why **dynamic**: A concrete error trace is very useful for debugging
• Why **online** (vs post-mortem):
  – To support accessing race information within the program
• FastTrack: Faster than Goldilocks, state of the art
  – But still too expensive: 10X for Java, 100X for C/C++

• **Goal**: Make precise race detection more practical using only mainstream hardware and software.
This Work in Context

• Our previous efforts:
  – Parallelize race detection using the GPU
    • Faster than dynamic race detection on the CPU only
    • Checking lags behind application
    • Event buffer between CPU and GPU the bottleneck
  – Parallelize race detection using software TM running on sibling threads
    • Not faster
    • Synchronization cost between application and sibling threads offsets benefit of parallelization

• This work:
  – Had access to Intel TSX prototype before it was commercially available
  – Experimented with using hardware TM support to make analysis synchronization faster

• Result: Up to 40% faster compared to lock-based version of FastTrack on C programs.
Happens-before race detection

Var X = 1;

Thread 1

Thread 2

Lock(L)
read X
write X
Unlock(L)

Lock(L)
write X
Unlock(L)

Lock(L)
read X
write X
Unlock(L)

Synchronizes-with
Happens-before
Happens-before race detection

Thread 1

Var X = 1;

Lock(L)
read X
write X
Unlock(L)

Thread 2

Lock(L)
write X
Unlock(L)

Program-order

race
write X
Unlock(L)

Program-order

Synchronizes-with

Happens-before
Anatomy of Dynamic Race Detection

Memory access or synchronization operation

Dynamic instrumentation
- detects access,
- calls race-detection function:
  FastTrack_Process_Access(addr, thrd);

FastTrack_Process_Access(addr, thrd);
- Read analysis state for addr
- Determine if there is a race
- Update analysis state
The FastTrack Algorithm

**FastTrack State:**

- $C$: Tid $\rightarrow$ VC
- $L$: Lock $\rightarrow$ VC
- $W$: Var $\rightarrow$ Epoch
- $R$: Var $\rightarrow$ (Epoch $\cup$ VC)

**Writes:** 14.5% of all Operations

**[FT WRITE SAME EPOCH]**

$$W_x = E(t)$$

$$(C, L, R, W) \xrightarrow{wr(t,x)} (C, L, R, W)$$

71.0% of writes

**[FT WRITE EXCLUSIVE]**

$$R_x \in Epoch$$

$$R_x \leq C_t$$

$$W_x \leq C_t$$

$$W' = W[x := E(t)]$$

$$(C, L, R, W) \xrightarrow{wr(t,x)} (C, L, R, W')$$

28.9% of writes

**[FT WRITE SHARED]**

$$R_x \in VC$$

$$R_x \leq C_t$$

$$W_x \leq C_t$$

$$W' = W[x := E(t)]$$

$$R' = R[x := \perp_e]$$

$$(C, L, R, W) \xrightarrow{wr(t,x)} (C, L, R', W')$$

0.1% of writes

Figure taken from “FastTrack: Efficient and Precise Dynamic Race Detection”
Flanagan and Freund, PLDI ‘07
The FastTrack Algorithm

Figure 5: FastTrack Instrumentation State and Code

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[];
}

Figure taken from “FastTrack: Efficient and Precise Dynamic Race Detection”
Flanagan and Freund, PLDI ‘07
```java
@Override
public void access(final AccessEvent fae) {
    final ShadowVar orig = fae.getOriginalShadow();
    final ShadowThread td = fae.getThread();

    if (orig instanceof FastTrackGuardState) {
        FastTrackGuardState x = (FastTrackGuardState)orig;

        final int tdEpoch = ts_get_epoch(td);
        final CV tdCV = ts_get_cv(td);

        Object target = fae.getTarget();
        if (target == null) {
            CV initTime = classInitTime.getInfo().getInfo().getOwner();
            tdCV.max(initTime);
        }

        synchronized(x) {
            if (fae.isWrite()) {
                // WRITE
                final int lastWriteEpoch = x.lastWrite;
                if (lastWriteEpoch == tdEpoch) {
                    return;
                }

                final int lastWriter = Epoch.tid(lastWriteEpoch);
                if (lastWriteEpoch > tdCV.get(lastWriter)) {
                    error(fae, 1, "write-by-thread-", lastWriter, "write-by-thread-", td.getTid());
                }

                final int lastReadEpoch = x.lastRead;
                if (lastReadEpoch != Epoch.EMPTY) {
                    final int lastReader = Epoch.tid(lastReadEpoch);
                    if (lastReadEpoch > tdCV.get(lastReader)) {
                        error(fae, 2, "read-by-thread-", lastReader, "read-by-thread-", td.getTid());
                    }
                    else {
                        if (x.anyGt(tdCV)) {
                            for (int prevReader = x.nextGt(tdCV, 0); prevReader > -1; prevReader = x.nextGt(tdCV, 0)) {
                                if (prevReader != td.getTid()) {
                                    error(fae, 3, "read-by-thread-", prevReader, "write-by-thread-", td.getTid());
                                }
                            }
                        }
                    }
                }
            }
            x.lastWrite = tdEpoch;
            x.lastRead = tdEpoch;
        }
    }
}
```
Table 1. Concurrency data on benchmarks studied

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>69.08%</td>
<td>64.20%</td>
<td>57.06%</td>
<td>45.93%</td>
<td>34.80%</td>
</tr>
<tr>
<td>FFT</td>
<td>52.38%</td>
<td>47.06%</td>
<td>49.29%</td>
<td>36.53%</td>
<td>33.33%</td>
</tr>
<tr>
<td>Lu_cb</td>
<td>66.144%</td>
<td>63.80%</td>
<td>61.026%</td>
<td>50.91%</td>
<td>48.98%</td>
</tr>
<tr>
<td>Lu_ncb</td>
<td>68.21%</td>
<td>63.25%</td>
<td>61.68%</td>
<td>54.26%</td>
<td>51.67%</td>
</tr>
<tr>
<td>Radix</td>
<td>35.01%</td>
<td>32.74%</td>
<td>32.96%</td>
<td>29.78%</td>
<td>24.27%</td>
</tr>
</tbody>
</table>
IDEA

• Intel TSX
  – Hardware support for atomically-executed code regions
  – Optimistic concurrency
  – Available on mainstream processors

• Use Intel TSX to ensure atomicity of FastTrack code blocks
  – Instead of lock-protected regions
Intel TSX instructions

• Hardware instructions to tell processor to start and transaction

  ```
  TSX_BEGIN;
  Sequence of instructions
  TSX_END;
  ```

• Processor hardware ensures transactional memory semantics
Before instrumentation

lock (L1)
temp = acc
temp = temp + 100
acc = temp
Unlock (L1)

After instrumentation

lock (L1)
FastTrack_Process_Lock(L1)

temp = acc
TSX_BEGIN;
FastTrack_Process_Read(acc)
TSX_END;

temp = temp +100
acc = temp
TSX_BEGIN;
FastTrack_Process_Write(acc)
TSX_END;

FastTrack_Process_Unlock(L1)
Unlock (L1)
Also Sound Instrumentation

**Before instrumentation**

- `lock (L1)`
- `temp = acc`
- `temp = temp + 100`
- `acc = temp`
- `Unlock (L1)`

**After instrumentation**

- `lock (L1)`
- `FastTrack_Process_Lock(L1)`
- `TSX_BEGIN;`
  - `temp = acc`
  - `FastTrack_Process_Read(acc)`
  - `temp = temp + 100`
  - `acc = temp`
  - `FastTrack_Process_Write(acc)`
- `TSX_END;`
- `FastTrack_Process_Unlock(L1)`
- `Unlock (L1)`
Lock-based vs TSX-based FastTrack (4 threads, 4 cores)
Lock-based vs TSX-based FastTrack (8 threads, 4 cores)
TSX Speedup vs # of Application Threads
For fun: Comparison with single-global-lock-based FastTrack
Speedup over fine-grain lock-based FastTrack vs TSX block size
Conclusions, Future Work

• TSX-based FastTrack up to 40% faster than lock-based FastTrack for C benchmarks

• Future work
  – Integrate with PIN dynamic instrumentation
  – Randomize TSX block boundaries
  – Race avoidance in legacy x86 binaries