Scalability Analysis of Signatures in Transactional Memory Systems

Ricardo Quislant, Eladio Gutiérrez, Óscar Plata
Dept. Computer Architecture
University of Malaga, Spain
Outline

• Introduction & Background
• The Eigenbench Benchmark
• Methodology
• Experimental Evaluation
• Conclusions
Introduction

- **Transactional Memory (TM)** is proposed by Herlihy and Moss [ISCA’93] as an alternative to locks, aiming at facilitating parallel programming

- **Transaction**: block of computations that appears to execute *atomically* and *isolated*

  - Memory operations of different transactions interleave each other (**optimistic model**), unless a conflict is detected (data dependence)
  - Abstraction and composition are provided

```c
Acquire_lock(L);
    Critical section body;
Release_lock(L);
```

```c
Atomic {
    Critical section body;
}
```
Introduction

- TM systems can be implemented in software (STM) or hardware (HTM)
- HTM systems implement TM mechanisms at the core level:
  - **Data management**: new versions of data written inside transactions must be isolated (buffers, logs)
  - **Conflict detection**: access to shared data must be tracked to ensure atomicity.

Alternatives:

» Cache metadata: R and W bits per cache block
» **Signatures**: per-thread structures to track the read set (RS) and write set (WS) of transactions
Introduction

- Signatures: implemented as **Bloom filters**
- Signature **advantages** over cache metadata:
  - Cache is left untouched
  - HTM is easier to virtualize: migrations, context switches
  - Not limited to cache size (apparently unbounded)

• However, signatures may yield **false positives** (FP) that can harm execution:
  - FP involve false conflicts that **limit concurrency**
Introduction

Signatures and Bloom Filters

- **Signatures** are implemented as Bloom filters and were first proposed by Ceze et al. [ISCA’06]
- **Bloom filters** were devised by Burton H. Bloom [Comm’70] as a time- and space-efficient way of representing a set of elements (memory addresses):

![Hash function indexes](image)

- **Insertion**
- **Test**
Introduction

Evaluating conflict probability

Let $sz$, $sz'$ the size of two xacts. in a space of $D$ elements, the conflict probability is (Yu et al. [SBAC-PAC’10]):

$$p_c(sz, sz') = 1 - \frac{(D - sz)}{sz'} \frac{D}{sz'}$$

Considering $N$ concurrent transactions, the probability of one of them conflicting with any other:

$$p_c = 1 - (1 - p_c(sz, sz'))^{N-1}$$
Scalability Analysis of Signatures in Transactional Memory Systems

Introduction

Evaluating conflict probability

Let $sz$, $sz'$ the size of two xacts. in a space of $D$ elements, the conflict probability is (Yu et al. [SBAC-PAC’10]):

$$p_c(sz, sz') = 1 - \frac{(D - sz)}{sz'}$$

- Probability of false positives of a Bloom filter of size $M$ bits, $k$ hashes and $n$ insertions (Bose et al. [IPL’08])

$$p_{FP} \approx \left(1 - e^{-knM}\right)^k$$

Trade-off: $k = 4$
Introduction

The effect of false positives is increasing the effective size of transactions, due to the aliases caused by hash functions

- Mitigating these effects is important:
  - Locality sensitive signatures (LS-Sig)

\[(r, \delta_P)\text{-LS-Sig: addresses are mapped with different granularities}\]

<table>
<thead>
<tr>
<th>Addr</th>
<th>Hash Function Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>h_0</td>
</tr>
<tr>
<td>0xffff0</td>
<td>240</td>
</tr>
<tr>
<td>0xffff1</td>
<td>586</td>
</tr>
<tr>
<td>0xffff2</td>
<td>90</td>
</tr>
<tr>
<td>0xffff3</td>
<td>736</td>
</tr>
<tr>
<td>0xffff4</td>
<td>181</td>
</tr>
<tr>
<td>0xffff5</td>
<td>527</td>
</tr>
<tr>
<td>0xffff6</td>
<td>31</td>
</tr>
<tr>
<td>0xffff7</td>
<td>677</td>
</tr>
</tbody>
</table>

Introduction

The effect of false positives is increasing the effective size of transactions, due to the aliases caused by hash functions

- Mitigating these effects is important:
  - Multiset (unified) signatures
  - Asymmetric signatures


**EigenBench**

• Benchmarks based on real applications can be configurable

• But, it is difficult to **decouple transacational characteristics** (xact size, contention, …)

• **EigenBench** emulates orthogonal TM characteristics:
  
  ▪ **Contention**: Probability of conflict
  ▪ **Transaction Length**: Number of reads and writes
  ▪ **Temporal locality of references**
  ▪ **Concurrency**: Number of concurrent threads

• Some modifications added for our analysis
  
  ▪ **Spatial locality**

---

global long array_hot[N_HOT], global long array_mild[N_MILD];
void test_core(tid, loops, lct, R_HOT, W_HOT, R_MILD, W_MILD, R_OUT, W_OUT) {
    long val=0;5
    long total = W_HOT + W_MILD + R_HOT + R_MILD;
    for (i=0; i<loops; i++) {
        BEGIN_TM();
        for (j=0; j<total ; j++) {
            switch(rand_action(r_hot, w_hot, r_mild, w_mild)) {
            case READ_HOT:
                index = rand_index(tid, lct, array_hot);
                val += TM_READ(array_hot[index]);
                break;
            case WRITE_HOT:
                index = rand_index(tid, lct, array_hot);
                TM_WRITE(array_hot[index], val);
                break;
            case READ_MILD:
                index = rand_index(tid, lct, array_mild);
                val += TM_READ(array_mild[index]);
                break;
            case WRITE_MILD:
                index = rand_index(tid, lct, array_mild);
                TM_WRITE(array_mild[index], val);
                break;
            }
        }
        END_TM();
        val += local_ops(R_OUT, W_OUT, val, tid);
    }
}
global long array_hot[N_HOT], global long array_mild[N_MILD];
void test_core(tid, loops, lct, R_HOT, W_HOT, R_MILD, W_MILD, R_OUT, W_OUT) {
    long val=0;5
    long total = W_HOT + W_MILD + R_HOT + R_MILD;
    for (i=0; i<loops; i++) {
        BEGIN_TM();
        for (j=0; j<total ; j++) {
            switch(rand_action(r_hot, w_hot, r_mild, w_mild)) {
                case READ_HOT:
                    index = rand_index(tid, lct, array_hot);
                    val += TM_READ(array_hot[index]);
                    break;

                case WRITE_HOT:
                    index = rand_index(tid, lct, array_hot);
                    TM_WRITE(array_hot[index], val);
                    break;

                case READ_MILD:
                    index = rand_index(tid, lct, array_mild);
                    val += TM_READ(array_mild[index]);
                    break;

                case WRITE_MILD:
                    index = rand_index(tid, lct, array_mild);
                    TM_WRITE(array_mild[index], val);
                    break;
            }
        }
        END_TM();
        val += local_ops(R_OUT, W_OUT, val, tid);
    }
}
global long array_hot[N_HOT], global long array_mild[N_MILD];
void test_core(tid, loops, lct, R_HOT, W_HOT, R_MILD, W_MILD, R_OUT, W_OUT) {
    long val=0;5
    long total = W_HOT + W_MILD + R_HOT + R_MILD;
    for (i=0; i<loops; i++) {
        BEGIN_TM();
        for (j=0; j<total; j++) {
            switch(rand_action(r_hot, w_hot, r_mild, w_mild)) {
                case READ_HOT:
                    index = rand_index(tid, lct, array_hot);
                    val += TM_READ(array_hot[index]);
                    break;
                case WRITE_HOT:
                    index = rand_index(tid, lct, array_hot);
                    TM_WRITE(array_hot[index], val);
                    break;
                case READ_MILD:
                    index = rand_index(tid, lct, array_mild);
                    val += TM_READ(array_mild[index]);
                    break;
                case WRITE_MILD:
                    index = rand_index(tid, lct, array_mild);
                    TM_WRITE(array_mild[index], val);
                    break;
            }
        }
        END_TM();
        val += local_ops(R_OUT, W_OUT, val, tid);
    }
}
Each thread traverses a given number of transactions.

Each transaction executes the same total number of memory operations distributed randomly according to \{W,R\}_{HOT,MILD} parameters.

Temporal locality is taken into account.
EigenBench

- Modification to EigenBench:
  - Routines to generate random numbers recoded in order to be adapted to the implicit HTM system we used:
    - Mersenne twister pseudorandom generator per thread, similar to this one found in the library of the STAMP benchmark suite
    - To keep the TM system from tracking those implicit accesses we use escape actions
  - Spatial locality:
    - Random walk through a one-direction integer array [Thiebaut et al., 1992]
    - Jumps (gaps) between steps governed by the following prob. distribution:
      \[ \Pr[X > u] = \left(\frac{u_0}{u}\right)^\theta, \Pr[X = u] = \Pr[X = -u], \text{ with } u_0, \theta \text{ constants} \]
    - Chosen \( u_0 = \theta = 1 \), equivalent to Zipf distribution (1, 1/2, 1/3, …)
long history_buffer[N_HB];

long rand_index(tid, lct, lcs, array) {

... // Original code

if(// generate a locality random walk with probability lcs) {
    int sign = random([-1, 1]); // The jump can be positive or negative
    int rand = random([0, 1023]); // A random number between 0 and 1023
    if(rand in [0, 303) the jump is 1.
    if in [303, 454) the jump is 2, ...
    for(jump=1; jump<=16; jump++)
        if(rand < zipf[jump-1]) break;

    addr = top(history_buffer); // Get the last accessed location
    x = (addr+sign*jump); // Perform the jump
    push(hist, x); // Insert the new accessed location in the history buffer

    return x;
}
Methodology

• Simulation Framework

  ▪ **Simics**: full-system simulator modeling Sun Fire server with UltraSPARC CPUs
  ▪ OS Solaris 10 installed
  ▪ Wisconsin GEMS toolset: **Ruby** module models **LogTM-SE HTM** from Yen et al. [HPCA’07]
  ▪ Ruby was modified to include our proposals

Simics: full-system simulator modeling Sun Fire server with UltraSPARC CPUs
OS Solaris 10 installed
Wisconsin GEMS toolset: Ruby module models LogTM-SE HTM from Yen et al. [HPCA’07]
Ruby was modified to include our proposals
Methodology

• Target System Overview

<table>
<thead>
<tr>
<th>Processor</th>
<th>16 in order single-issue cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>Split, 32KB Instructions + 32KB Data</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Shared banked, unified, 8MB, 8 ways, 64B blocks</td>
</tr>
<tr>
<td>L2 Directory</td>
<td>Full bit-vector of sharers</td>
</tr>
<tr>
<td>Network</td>
<td>Packet-switched tiled interconnect</td>
</tr>
</tbody>
</table>
Methodology

- Modifications to Ruby:
  - Enhanced signature schemes:
    - LS-Sig \( (3, \delta_p) \)
    - LS-Sig \( (5, \delta_p) \)
    - MS \( s = 3 \) L2-Sig (more granularity in shared functions)
    - ASYM \( a = 6 \) and \( a = 5 \)
  - Performance compared with perfect signatures (no false conflicts)
  - Used 15 out of 16 available cores (1 left for OS)
Locality sensitive signatures

- Analyzed configurations: LS-Sig \((3, \delta_p)\) and LS-Sig \((5, \delta_p)\)

- \((r, \delta_p)\)-LS-Sig: addresses within intervals of hamming distance \(r\) are mapped with different granularities for different hash functions

\[
\delta_p(d(x, y)) = \begin{cases} 
1, & \text{if } d(x, y) = 1 \\
2, & \text{if } 2 \leq d(x, y) \leq 2^{\lfloor r/2 \rfloor} - 1 \\
3, & \text{if } 2^{\lfloor r/2 \rfloor} \leq d(x, y) \leq 2^r - 1 
\end{cases}
\]

<table>
<thead>
<tr>
<th>Addr</th>
<th>(h_0)</th>
<th>(h_1)</th>
<th>(h_2)</th>
<th>(h_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffffff</td>
<td>240</td>
<td>158</td>
<td>889</td>
<td>554</td>
</tr>
<tr>
<td>0xffffffff1</td>
<td>586</td>
<td>158</td>
<td>889</td>
<td>554</td>
</tr>
<tr>
<td>0xffffffff2</td>
<td>90</td>
<td>347</td>
<td>889</td>
<td>554</td>
</tr>
<tr>
<td>0xffffffff3</td>
<td>736</td>
<td>347</td>
<td>889</td>
<td>554</td>
</tr>
<tr>
<td>0xffffffff4</td>
<td>181</td>
<td>906</td>
<td>484</td>
<td>554</td>
</tr>
<tr>
<td>0xffffffff5</td>
<td>527</td>
<td>906</td>
<td>484</td>
<td>554</td>
</tr>
<tr>
<td>0xffffffff6</td>
<td>31</td>
<td>591</td>
<td>484</td>
<td>554</td>
</tr>
<tr>
<td>0xffffffff7</td>
<td>677</td>
<td>591</td>
<td>484</td>
<td>554</td>
</tr>
</tbody>
</table>

Example for \((3, \delta_p)\) LS-Sig

Granularity \(2^0\)

Granularity \(2^r\)
Multiset / Asymmetric

- Analyzed configurations:
  - MS $s = 3$ L2-Sig (more granularity in shared functions),
  - ASYM $a = 6$ and $a=5$. 
Results

Contestation Results

15 threads, lct=0
4Kbits per data-set

\[ p_c \approx 1 - \left( 1 - \min \left( 1, \frac{(n\text{Threads} - 1)W'_\text{HOT}}{N_{\text{HOT}}} \right) \right)^{W'\text{HOT} + R'\text{HOT}} \]

\( (W'_{\text{HOT}}, R'_{\text{HOT}} \) exclude repetitions)
Results

Contestion Results

15 threads, lct=0 4Kbits per dataset

Large Transactions

Locality: 25%

Locality: 50%

Locality: 75%
Results

- **Transaction Size Results**
  - Locality: 25%
  - No contention

- Asymmetric transactions:
  - RS to WS ratio = 3
  - Real RS to WS ratio < 2
Results

- **Concurrency Results**
  - Speedup w.r.t unprotected serial
  - Contention: 3%
  - \((5,\delta_p)\)-LS-Sig speedup over parallel: [1, 1.02, 1.08, 1.23, 1.41]
Results

- Iso-speedup analysis
  - speedup w.r.t unprotected serial
  - 15 threads, symmetric workload
  - No real contention, only the one due to false positive
Conclusions

**Scalability analysis:**

- HTM Scalability limited by HW resources
- Signatures (BF) allow virtually unbounded transactions but also limited (degradation beyond 1/5 BF size; even worse than serial but can be mitigated with locality exploitation)
- Orthogonal TM characteristic benchmark: EigenBench: Contention, transaction length and concurrency were analyzed
- Our signatures ameliorate the effect of false positives and scale better than conventional signatures