On the Performance of Window-Based Contention Managers for Transactional Memory

Gokarna Sharma and Costas Busch
Louisiana State University
Agenda

• Introduction and Motivation

• Previous Studies and Limitations

• Execution Window Model
  ➢ Theoretical Results
  ➢ Experimental Results

• Conclusions and Future Directions
Retrospective

• 1993

• Today
  ➢ Several STM/HTM implementation efforts by Intel, Sun, IBM; growing attention

• Why TM?
  ➢ Many drawbacks of traditional approaches using Locks, Monitors: error-prone, difficult, composability, ...

Lock: only one thread can execute

```
lock data
modify/use data
unlock data
```

TM: many threads can execute

```
atomic {
  modify/use data
}
```
Transactional Memory

- **Transactions** perform a sequence of read and write operations on shared resources and appear to execute atomically.

- **TM** may allow transactions to run concurrently but the results must be equivalent to some sequential execution.

**Example:**

- Initially, `x == 1, y == 2`

```
atomic {
    x = 2;
    y = x+1;
}
```

```
atomic {
    r1 = x;
    r2 = y;
}
```

T1 then T2: `r1 == 2, r2 == 3`

T2 then T1: `r1 == 1, r2 == 2`

```
x = 2;
```

```
y = 3;
```

```
r2 = 3;
```

Incorrect: `r1 == 1, r2 == 3`

- **ACI(D)** properties to ensure correctness.
Software TM Systems

Conflicts:
- A contention manager decides
- Aborts or delay a transaction

Centralized or Distributed:
- Each thread may have its own CM

Example:
Initially, $x = 1$, $y = 1$

T1

```
atomic {
    ...
    x = 2;
}
```

T2

```
atomic {
    ...
    x = 3;
}
```

Conflicts
Abort undo changes (set $x=1$) and restart

T1

```
atomic {
    ...
    x = 2;
    ...
    x = 3;
}
```

T2

```
atomic {
    ...
    x = 2;
    ...
    x = 3;
}
```

Conflicts
Abort (set $y=1$) and restart
OR wait and retry
Transaction Scheduling

The most common model:
- \( m \) concurrent transactions on \( m \) cores that share \( s \) objects
- Sequence of operations and a operation takes one time unit
- Duration is fixed

Throughput Guarantees:
- **Makespan**: the time needed to commit all \( m \) transactions
- Competitive Ratio: \( \frac{\text{Makespan of my CM}}{\text{Makespan of optimal CM}} \)

Problem Complexity:
- **NP-Hard** (related to vertex coloring)

Challenge:
- How to schedule transactions so that makespan is minimized?
Literature

• Lots of proposals
  ➢ Polka, Priority, Karma, SizeMatters, ...

• Drawbacks
  ➢ Some need globally shared data (i.e., global clock)
  ➢ Workload dependent
  ➢ Many have no theoretical provable properties
    ✓ i.e., Polka - but overall good empirical performance

• Mostly empirical evaluation using different benchmarks
  ➢ Choice of a contention manager significantly affects the performance
  ➢ Do not perform well in the worst-case (i.e., contention, system size, and number of threads increase)
Literature on Theoretical Bounds

Guerraoui et al. [PODC’05]:
First contention manager *GREEDY* with \(O(s^2)\) competitive bound

Attiya et al. [PODC’06]:
Bound of *GREEDY* improved to \(O(s)\)

Schneider and Wattenhofer [ISAAC’09]:
*RandomizedRounds* with \(O(C \cdot \log m)\) (\(C\) is the maximum degree of a transaction in the conflict graph)

Attiya et al. [OPODIS’09]:
*Bimodal* scheduler with \(O(s)\) bound for read-dominated workloads

Sharma and Busch [OPODIS’10]:
Two algorithms with \(O(\sqrt{s})\) and \(O(\sqrt{s} \cdot \log n)\) bounds for balanced workloads
Objectives

Scalable transactional memory scheduling:

- Design contention managers that exhibit both good theoretical and empirical performance guarantees
- Design contention managers that scale well with the system size and complexity
Execution Window Model

- Collection of $n$ sets of $m$ concurrent transactions that share $s$ objects

Assuming maximum degree in conflict graph $C$ and execution time duration $\tau$

Serialization upper bound: $\tau \cdot \min(Cn, mn)$

One-shot bound: $O(sn)$ [Attiya et al., PODC'06]

Using Randomized Rounds: $O(\tau \cdot Cn \log m)$
Theoretical Results

- **Offline Algorithm:** *(maximal independent sets)*
  - For *scheduling with conflicts* environments, i.e., traffic intersection control, dining philosophers problem
  - **Makespan:** $O(\tau \cdot (C + n \log (mn)))$, ($C$ is the conflict measure)
  - **Competitive ratio:** $O(s + \log (mn))$ whp

- **Online Algorithm:** *(random priorities)*
  - For online scheduling environments
  - **Makespan:** $O(\tau \cdot (C \log (mn) + n \log^2 (mn)))$
  - **Competitive ratio:** $O(s \log (mn) + \log^2 (mn))$ whp

- **Adaptive Algorithm**
  - Conflict graph and maximum degree $C$ both not known
  - Adaptively guesses $C$ starting from 1
Intuition (1)

- Introduce random delays at the beginning of the execution window

- Random delays help conflicting transactions shift avoiding many conflicts
Intuition (2)

- Frame based execution to handle conflicts

**Makespan:** $\max \{q_i\} + \text{No of frames} \times \text{frame size}$
Experimental Results (1)

- **Platform used**
  - Intel i7 (4-core processor) with 8GB RAM and hyperthreading on

- **Implemented window algorithms in DSTM2, an eager conflict management STM implementation**

- **Benchmarks used**
  - List, RBTree, SkipList, and Vacation from STAMP suite.

- **Experiments were run for 10 seconds and the data plotted are average of 6 experiments**

- **Contention managers used for comparison**
  - **Polka** – Published best CM but no theoretical provable properties
  - **Greedy** – First CM with both theoretical and empirical properties
  - **Priority** – Simple priority-based CM
Experimental Results (2)

Performance throughput:

- No of txns committed per second
- Measures the useful work done by a CM each time step

List Benchmark

SkipList Benchmark

Graphs showing the performance of different benchmarks with varying number of threads.
Performance throughput:

- **RBTree Benchmark**

- **Vacation Benchmark**

**Conclusion #1:** Window CMs always improve throughput over Greedy and Priority.

**Conclusion #2:** Throughput is comparable to Polka (outperforms in Vacation).
Experimental Results (4)

Aborts per commit ratio:

- No of txns aborted per txn commit
- Measures efficiency of a CM in utilizing computing resources

### List Benchmark

<table>
<thead>
<tr>
<th>No of aborts/commit</th>
<th>No of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>14</td>
<td>35</td>
</tr>
</tbody>
</table>

### SkipList Benchmark

<table>
<thead>
<tr>
<th>No of aborts/commit</th>
<th>No of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>14</td>
<td>35</td>
</tr>
</tbody>
</table>

- Polka
- Greedy
- Priority
- Online
- Adaptive
Experimental Results (5)

**Aborts per commit ratio:**

**Conclusion #3:** Window CMs always reduce no of aborts over *Greedy* and *Priority*

**Conclusion #4:** No of aborts are comparable to *Polka* (outperform in Vacation)
Experimental Results (6)

Execution time overhead:
- Total time needed to commit all transactions
- Measures scalability of a CM in different contention scenarios

**List Benchmark**

- Low
- Medium
- High

**SkipList Benchmark**

- Low
- Medium
- High
Experimental Results (7)

Execution time overhead:

**RBTree Benchmark**

**Vacation Benchmark**

Conclusion #5: Window CMs generally reduce execution time over Greedy and Priority (except SkipList)

Conclusion #6: Window CMs good at high contention due to randomization overhead
Future Directions

• Encouraging theoretical and practical results

• Plan to explore (experimental)
  ➢ Wasted Work
  ➢ Repeat Conflicts
  ➢ Average Response Time
  ➢ Average committed transactions durations

• Plan to do experiments using more complex benchmarks
  ➢ E.g., STAMP, STMBench7, and other STM implementations

• Plan to explore (theoretical)
  ➢ Other contention managers with both theoretical and empirical guarantees
Conclusions

• TM contention management is an important online scheduling problem

• Contention managers should scale with the size and complexity of the system

• Theoretical as well as practical performance guarantees are essential for design decisions

• Need to explore mechanisms that scale well in other multi-core architectures:
  ➢ ccNUMA and hierarchical multilevel cache architectures
  ➢ Large scale distributed systems
Thank you for your attention!!!