Designing an Effective Hybrid Transactional Memory System

Chí Cao Minh
28 May 2008
The Need for Multiprocessors

- Uniprocessor systems hitting limits
  - Design complexity overwhelming
  - Power consumption increasing dramatically
  - Instruction-level parallelism exhausted

- Solution is multiprocessor systems
  - Simpler processor design (but many of them)
  - Reduce power requirements
  - Expose opportunity for thread-level parallelism
Commonly achieved via lock-based parallel programs

Unfortunately, parallel programming with locks is hard
- Option 1: Coarse-grain locks
  - Simplicity at less concurrency
- Option 2: Fine-grain locks
  - Better performance (maybe) at more complexity
Parallel Programming With Locks

Thread 1: insert 2

Thread 2: insert 10
Parallel Programming With Locks

Thread 1: insert 2
Thread 2: insert 10
Parallel Programming With Locks

Thread 1: insert 2

Thread 2: insert 5
What is a transaction?
- Group of instructions in computer program:
  ```java
  atomic {
    if (x != NULL) x.foo();
    y = true;
  }
  ```
- Required properties: Atomicity, Isolation, Serializability

Key idea: Use transactions to build parallel programs
- Large atomic blocks simplify parallel programming
- Simplicity of coarse-grain locks with speed of fine-grain locks
**Optimistic Concurrency Control**

- Life cycle of a transaction:
  - Start
  - Speculative execution (optimistic)
  - Build read-set and write-set
    - Write-set manages write versioning
  - Commit
    - Fine-grain R-W & W-W conflict detection
  - Abort & rollback
Thread 1: insert 2
Read-set:  6, 3, 1
Write-set:  1

Thread 2: insert 5
Read-set:  6, 3, 4
Write-set:  4
Thread 1: insert 2
Read-set: 6, 3, 1
Write-set: 1

Thread 2: insert 0
Read-set: 6, 3, 1
Write-set: 1
TM Implementations

- TM can be implemented in hardware or software

- Hardware-based (HTM)
  - [Herlihy 93], [Rajwar 02], [Hammond 04], [Moore 06]
  - Strengths: high performance & predictable semantics
  - Weaknesses: costly & inflexible

- Software-based (STM)
  - [Shavit 95], [Herlihy 03], [Harris 03], [Saha 06], [Dice 06]
  - Strengths: low-cost & flexible
  - Weaknesses: low performance & unpredictable semantics
TM Community Wishlist

- Standard method to compare TM systems
  - Each TM system evaluated with different apps
  - How to pick the better of two HTMs?

- TM system that combines strengths of HTM and STM
  - High-performance
  - Flexibility
  - Low-cost
  - Predictable semantics
My Contributions

- STAMP: Benchmark suite for TM
  - 8 applications specifically for evaluating TM
  - Comprehensive breadth and depth analysis
  - Portable to many kinds of TMs
  - Public release: http://stamp.stanford.edu
  - IEEE Intl. Symposium on Workload Characterization (IISWC) 2008

- Signature-Accelerated TM (SigTM): Hybrid TM
  - Hardware acceleration of software transactions
  - Fast, flexible, cost-effective, & predictable semantics
Outline

- Background & Motivation

- STAMP: Benchmark suite for TM

- SigTM: Effective hybrid TM
  - Fast, flexible, low-cost
  - Predictable semantics

- Conclusions
**Computer Benchmarks**

- **What is a benchmark?**
  - Program used to evaluate computer performance
  - Help accelerate innovation in computer design

- **Benchmarks for multiprocessors**
  - Not good for evaluating TM
    - Regular algorithms without synchronization problems

- **Benchmarks for TM systems**
  - Microbenchmarks from RSTMv3 (2006)
  - STMBench7 (2007)
  - Haskell applications by Perfumo et. al (2007)
TM Benchmark Requirements

- **Breadth**: variety of algorithms & app domains
- **Depth**: wide range of transactional behaviors
- **Portability**: runs on many classes of TM systems

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Breadth</th>
<th>Depth</th>
<th>Portability</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSTMv3</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>Microbenchmarks</td>
</tr>
<tr>
<td>STMbench7</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>Single program</td>
</tr>
<tr>
<td>Perfumo et al.</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>Microbenchmarks; Written in Haskell</td>
</tr>
</tbody>
</table>
STAMP Meets 3 Requirements

- **Breadth**
  - 8 applications covering different domains & algorithms
  - Applications not trivially parallelizable

- **Depth**
  - Wide range of transactional behaviors
    - Transaction length
    - Read and write set size
    - Contention amount
  - Most spend significant execution time in transactions

- **Portability**
  - Written in C with macro-based transaction annotations
  - Works with HTM, STM, and hybrid TM
## STAMP Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bayes</td>
<td>Machine learning</td>
<td>Learns structure of a Bayesian network</td>
</tr>
<tr>
<td>genome</td>
<td>Bioinformatics</td>
<td>Performs gene sequencing</td>
</tr>
<tr>
<td>intruder</td>
<td>Security</td>
<td>Detects network intrusions</td>
</tr>
<tr>
<td><strong>kmeans</strong></td>
<td>Data mining</td>
<td>Implements K-means clustering</td>
</tr>
<tr>
<td>labyrinth</td>
<td>Engineering</td>
<td>Routes paths in maze</td>
</tr>
<tr>
<td>ssca2</td>
<td>Scientific</td>
<td>Creates efficient graph representation</td>
</tr>
<tr>
<td><strong>vacation</strong></td>
<td>Online transaction processing</td>
<td>Emulates travel reservation system</td>
</tr>
<tr>
<td>yada</td>
<td>Scientific</td>
<td>Refines a Delaunay mesh</td>
</tr>
</tbody>
</table>
Kmeans Description

- Groups data into K clusters

Possible applications:
- **Biology**: plant and animal classification
- **WWW**: analyze web traffic for patterns
Kmeans Algorithm

1. Guess centers
2. Analyze data
3. Compute adjustments to centers
4. Update centers
5. Privatization
6. Transaction
7. Converged?

- yes
- no
Emulates travel reservation system
  - Similar to 3-tier design in SPECjbb2000
Vacation Algorithm

Get task?

Task kind?

Manager does reservation

Manager does cancelation

Manager does update

Done

yes

no

reserve

cancel

update
## STAMP Characterization

<table>
<thead>
<tr>
<th>Application</th>
<th>Instructions</th>
<th>Reads</th>
<th>Writes</th>
<th>Retries</th>
<th>Time in Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>bayes</td>
<td>60584</td>
<td>24</td>
<td>9</td>
<td>0.59</td>
<td>83%</td>
</tr>
<tr>
<td>genome</td>
<td>1717</td>
<td>32</td>
<td>2</td>
<td>0.14</td>
<td>97%</td>
</tr>
<tr>
<td>intruder</td>
<td>330</td>
<td>71</td>
<td>16</td>
<td>3.54</td>
<td>33%</td>
</tr>
<tr>
<td>kmeans</td>
<td>153</td>
<td>25</td>
<td>25</td>
<td>0.81</td>
<td>3%</td>
</tr>
<tr>
<td>labyrinth</td>
<td>219571</td>
<td>35</td>
<td>36</td>
<td>0.94</td>
<td>100%</td>
</tr>
<tr>
<td>ssc2</td>
<td>50</td>
<td>1</td>
<td>2</td>
<td>0.00</td>
<td>17%</td>
</tr>
<tr>
<td>vacation</td>
<td>3161</td>
<td>401</td>
<td>8</td>
<td>0.02</td>
<td>92%</td>
</tr>
<tr>
<td>yada</td>
<td>9795</td>
<td>256</td>
<td>108</td>
<td>2.51</td>
<td>100%</td>
</tr>
</tbody>
</table>
First comprehensive benchmark suite for TM
- Meets breadth, depth, and portability requirements
- Useful tool for analyzing TM systems (including SigTM)

Public release: http://stamp.stanford.edu
- Early adopters:
  - Industry: Microsoft, Intel, Sun, & more
  - Academia: U. Wisconsin, U. Illinois, & more
- TL2-x86 STM
Outline

- Background & Motivation

- STAMP: Benchmark suite for TM

- SigTM: Effective hybrid TM
  - Fast, flexible, low-cost
  - Predictable semantics

- Conclusions
Hardware vs. Software TM

- **HTM:** HW does write versioning & conflict detection
  - **Advantages:**
    - High performance
    - Predictable semantics
  - **Disadvantages:**
    - Expensive (e.g., requires cache modifications)
    - Inflexible (e.g., fixed capacity for write versioning)

- **STM:** SW does write versioning & conflict detection
  - **Advantages:**
    - Low-cost
    - Easy to change and evolve
  - **Disadvantages:**
    - High overhead
    - Unpredictable semantics
Hybrid hardware and software TM design
- Fast, flexible, cost-effective
- Predictable semantics

Design approach:
- Start with software transactions flexible & cost-effective
- Add hardware ("signatures") to accelerate fast
  - Also provides predictable semantics

<table>
<thead>
<tr>
<th></th>
<th>HTM</th>
<th>STM</th>
<th>SigTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write versioning</td>
<td>HW</td>
<td>SW</td>
<td>SW</td>
</tr>
<tr>
<td>Conflict detection</td>
<td>HW</td>
<td>SW</td>
<td>HW</td>
</tr>
</tbody>
</table>
Software Transactions

- Program: atomically remove head of linked-list

```
High-level

ListNode n;
atomic {
    n = head;
    if (n != null) {
        head = head.next;
    }
}

Compiler

Low-level

ListNode n;
STMstart();
    n = STMread(&head);
    if (n != null) {
        ListNode t;
        t = STMread(&head.next);
        STMwrite(&head, t);
    }
STMcommit();
```
STMstart

- Called at transaction start init transaction meta data

```c
STMstart()
{
    checkpoint(); // used to rollback
    other_initialization();
}
```

- Constant total cost per transaction
- Expensive only for short transactions
**STMread**

- Called to read shared data [W] add to read-set

```c
STMread(addr) {
    if (addr in WriteSet) // get latest value
        return WriteSet.getValue(addr);

    ReadSet.insert(addr);
    return *addr;
}
```

- Building read-set is expensive
- Total cost per transaction varies
  - Locality of read accesses, size of read-set, transaction length
STMwrite

- Called to write shared data \[\text{add to write-set}\]

```java
STMwrite(addr, val) {
    WriteSet.insert(addr, val);
}
```

- Total cost per transaction varies
  - Locality of write accesses, size of write-set, transaction length
  - Less cost than STMread (\(\# \text{ reads} \geq \# \text{ writes}\))
STMcommit

- Called at transaction end atomically commit changes

STMcommit() {
    foreach (addr in WriteSet)  // write set scan 1
        lock(addr);
    foreach (addr in ReadSet)   // read set scan
        validate(addr);          // someone wrote?
    foreach (addr in WriteSet)  // write set scan 2
        *addr = WriteSet.getValue(addr);
    foreach (addr in WriteSet)  // write set scan 3
        unlock(addr);
}

- Expensive: scan read-set (1x); scan write-set (3x)
How Slow Are SW Transactions?

- Measured single-thread STM performance

  1.8x – 5.6x slowdown over sequential

- Hybrid TM should focus on STMread and STMcommit
Each HW thread has 2 HW signatures (read & write)
  - Each signature implemented by a Bloom filter
    - Fixed-size bit array with set of hash functions
    - No other HW modifications (e.g., no extra cache bits)

Operations on signature (Bloom filter): insert & lookup

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>hash(N) = N mod 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

insert(2) -> 0 1 2 3
insert(6) -> aliasing
lookup(2) -> hit
lookup(3) -> miss
lookup(10) -> false hit
How SigTM uses its signatures:
- Tx read/write insert address into read/write signature
- Coherence messages look up address in signature
  - Enabled/disabled by software

If lookup hits in signature, either:
- Trigger SW abort handler (conflict detection)
- NACK remote request (atomicity & isolation enforcement)

Signatures may generate false conflicts
- Performance but not correctness issue
- Reduce with longer signatures & better hash functions

With this HW, how does the SW change?
SigTMread

SigTMread(addr) {
    if (addr in WriteSet) // get latest value
        return WriteSet.getValue(addr);
    read_sig_insert(addr); // 1 instruction
    return *addr;
}

- No need to build SW read-set
  - Replaced by read signature
- Read signature provides continuous validation
  - Snoops coherence messages & any hits cause abort
SigTMcommit

SigTMcommit() {
    enable_write_sig_lookup();
    foreach (addr in WriteSet) // write set scan 1
        fetch_exclusive(addr);

    enable_write_sig_nack();
    foreach (addr in WriteSet) // write set scan 2
        *addr = WriteSet.getValue(addr);
    disable_write_sig_lookup();
}

- Read signature eliminates scan of read-set to validate
- Write signature eliminates locks
  - Snoops coherence messages & NACKs any hits
- Two write-set scans instead of three
Experimental Setup

- Execution-driven simulation
  - 1–16 core x86 chip-multiprocessor with MESI coherence
  - Supports HTM, STM, and SigTM

- Used STAMP benchmark suite for evaluation

- Three experiments:
  - Does SigTM reduce the overhead of SW transactions?
  - How fast is SigTM?
  - How much hardware does SigTM cost?
How Much Smaller is the Overhead?

- Measured single-thread performance on STM and SigTM

- SigTM effectively accelerates read & commit
How Fast is SigTM?

- Measured speedup on 1–16 cores

![Graph showing speedup on 1-16 cores for kmeans and vacation applications with different processor cores.]

- In general, SigTM faster than STM but slower than HTM
Measured performance drop as signatures get shorter

Recommend 1024 bits for read sig, 128 bits for write sig
Outline

- Background & Motivation

- STAMP: Benchmark suite for TM

- SigTM: Effective hybrid TM
  - Fast, flexible, low-cost
  - Predictable semantics

- Conclusions
Two acceptable outcomes:

- T1 commits first; T1 uses only non-incremented n.val
- T2 commits first; T1 uses only incremented n.val

Works correctly with lock-based synchronization

- Race-free program
Unpredictable Results with STM?

- All STMs may give unexpected results
  - T1 may use both old & new value after privatization

- Cause: Non-transactional accesses are not instrumented
  - Non-Tx writes do not cause Tx to abort
  - Tx commit not atomic with respect to non-Tx accesses

```java
Thread 1
ListNode n;
atomic {
    n = head;
    if (n != null)
        head = head.next;
}
// use n.val many times

Thread 2
atomic {
    ListNode n = head;
    while (n != null) {
        n.val++;
        n = n.next;
    }
}
**Strong Isolation**

- **Definition**: Transactions isolated from non-Tx accesses

- **HTM** inherent strong isolation
  - Non-Tx cause coherence messages
  - Conflict detection mechanism enforces strong isolation

- **STM** supplemented strong isolation
  - Additional annotations needed for non-Tx accesses
  - Some can be optimized but still a source of overhead

- **SigTM** inherent strong isolation
  - Without additional instrumentation or overhead
How SigTM Provides Strong Isolation

- STMs have unpredictable results because:
  - Non-Tx writes do not cause transactions to abort
  - Tx commit not atomic with respect to non-Tx accesses

- Non-Tx writes cause SigTM to abort a transaction
  - Coherence messages looked up in read signature
  - Hits in read signature trigger transaction abort

- SigTM commit is atomic with respect to non-Tx accesses
  - Write signature used to provide atomic writeback
  - Coherence messages looked up in write signature
  - Hits in write signature NACK non-Tx accesses
Conclusions

- TM is promising for simplifying parallel programming

  My contributions to the TM community:

- **STAMP**
  - Comprehensive benchmark suite for TM
  - Public release: http://stamp.stanford.edu
  - Early adopters: MSFT, Intel, U. Wisconsin, U. Illinois, & more

- **Signature-Accelerated TM (SigTM)**
  - Hardware acceleration of software transactions
  - Fast, flexible, cost-effective, & predictable semantics
  - Attractive design for industry