A Scalable, Non-blocking Approach to Transactional Memory

Hassan Chafi
Austen McDonald
Chi Cao Minh
Christos Kozyrakis

Jared Casper
Brian D. Carlstrom
Woongki Baek
Kunle Olukotun

Computer System Laboratory
Stanford University
http://tcc.stanford.edu
Transactional Memory

- Problem: Parallel Programming is hard and expensive.
  - Correctness vs. performance

- Solution: Transactional Memory
  - Programmer-defined isolated, atomic regions
  - Easy to program, comparable performance to fine-grained locking
  - Done in software (STM), hardware (HTM), or both (Hybrid)

- Conflict Detection
  - Optimistic: Detect conflicts at transaction boundaries
  - Pessimistic: Detect conflicts during execution

- Version management
  - Lazy: Speculative writes kept in cache until end of transaction
  - Eager: Speculatively write “in place”, roll back on abort
So what’s the problem? (Haven’t we figured this out already?)

- Cores are the new GHz
  - Trend is 2x cores / 2 years: 2 in ‘05, 4 in ‘07, > 16 not far away
  - Sun: N2 has 8 cores with 8 threads = 64 threads
- It takes a lot to adopt a new programming model
  - Must last tens of years without much tweaking
  - Transactional Memory must (eventually) scale to 100s of processors
- TM studies so far use a small number of cores!
  - Assume broadcast snooping protocol
- If it does not scale, it does not matter
Lazy optimistic vs. Eager pessimistic

High contention

– Eager pessimistic
  • Serializes due to blocking
  • Slower aborts (result of undo log)

– Lazy optimistic
  • Optimistic parallelism
  • Fast aborts

Low contention

– Eager pessimistic
  • Fast commits

– Lazy optimistic
  • Slower commits… good enough??
What are we going to do about it?

- Serial commit ⇒ Parallel commit
  - At 256 proc, if 5% of the work is serial, maximum speedup is 18.6x
  - Two-phase commit using directories
- Write-through ⇒ write-back
  - Bandwidth requirements must scale nicely
  - Again, using directories
- Rest of talk:
  - Augmenting TCC with directories
  - Does it work?
Protocol Overview

- During the transaction
  - Track read and write sets in the cache
  - Track sharers of a line in the directory

- Two-phase commit
  - Validation: Mark all lines in write-set in directories
    - Locks line from being written by another transaction
  - Commit: Invalidate all sharers of marked lines
    - Dirty lines become “owned” in directory

- Require global ordering of transactions
  - Use a Global Transaction ID (TID) Vendor
Directory Structure

Directory tracks sharers of each line at home node
- Marked bit is used in the protocol

Now serving TID: transaction currently being serviced by directory
- Used to ensure a global ordering of transactions
- Skip vector used to help manage NSTID (see paper)
Cache Structure

• Each cache line tracks if it was speculatively read (SR) or modified (SM)
  ▪ Meaning that line was read or written in the current transaction
• Sharing and Writing vectors remember directories read from or written to
  ▪ Simple bit vector
Commit procedure

• Validation
  - Request TID
  - Inform all directories not in writing vector we will not be writing to them (Skip)
  - Request NSTID of all directories in writing vector
    • Wait until all NSTIDs ≥ our TID
  - Mark all lines that we have modified
    • Can happen in parallel to getting NSTIDs
  - Request NSTID of all directories in sharing vector
    • Wait until all NSTIDs ≥ our TID

• Commit
  - Inform all directories in writing vector of commit
  - Directory invalidates all other copies of written line, and marks line owned
    • Invalidation may violate other transaction
Parallel Commit Example

A Scalable, Non-blocking Approach to Transactional Memory
Parallel Commit Example

**NSTID: 1  Directory 0**

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>M</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NSTID: 1  Directory 1**

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>M</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TID Request**

- **Tid:** P1
- **Tid:** P2
- **Tid:** P1
- **Tid:** P2

**Commit**

- **Commit:** LD Y
- **Commit:** ST Y
- **Commit:** LD X
- **Commit:** ST X
Parallel Commit Example

A Scalable, Non-blocking Approach to Transactional Memory
Parallel Commit Example

A Scalable, Non-blocking Approach to Transactional Memory

HPCA 2007
Parallel Commit Example

```
<table>
<thead>
<tr>
<th>TID</th>
<th>Vendor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tid:1</td>
<td>P1</td>
</tr>
<tr>
<td>Tid:2</td>
<td>P2</td>
</tr>
</tbody>
</table>

Commit
```

```
<table>
<thead>
<tr>
<th>NSTID: 1  Directory 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>Y</td>
</tr>
</tbody>
</table>

Commit
```

```
<table>
<thead>
<tr>
<th>NSTID: 2  Directory 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

Commit
```

```
LD Y ST Y
Commit
```

```
LD X ST X
Commit
```

```
Conflict Resolution Example

```
TID Vendor

Load X

Load Y

LD Y
LD X
ST X
Commit

Tid:? P1

Data Y

NSTID: 1  Directory 0
P1  P2  M  O
X

Tid:? P2

Data X

LD X
ST X
...
Commit

NSTID: 1 Directory 1
P1  P2  M  O
Y

Commit

LDJX
STJX
Commit
```
Conflict Resolution Example

A Scalable, Non-blocking Approach to Transactional Memory
Conflict Resolution Example

A Scalable, Non-blocking Approach to Transactional Memory
Conflict Resolution Example

NSTID: 1  Directory 0
P1  P2  M  O
X  ...  

NSTID: 2  Directory 1
P1  P2  M  O
Y  ...  

Tid: 1  P1
LD Y  MD X  ST X  Commit
Mark: X
NSTID: 3

Tid: 2  P2
LD X  ST X  ...  Commit

NSTID Probe

TID Vendor
Conflict Resolution Example

Tid: 1
- P1
- LD Y
- LD X
- ST X
- Commit

Tid: 2
- P2
- LD X
- ST X
- Commit

NSTID: 2
Directory 0
- P1
- P2
- M
- O
- X
- Violation!

NSTID: 3
Directory 1
- P1
- P2
- M
- O
- Y

Vendor
Conflicts Resolution Example (Write-back)

Request: X

Tid: 1
P1

Tid: 2
P2

Load X

TID
Vendor

NSTID: 2
Directory 0

P1 P2 M O
X

TID: 3
Directory 1

NSTID: 3

P1 P2 M O
Y

WB: X

Data X

LD X
ST X
Commit

Tid: 1
P1

Tid: 2
P2
## Evaluation environment

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>1 - 64 single-issue PowerPC cores</td>
</tr>
<tr>
<td>L1</td>
<td>32 KB, 32 byte cache line, 4-way, 1 cycle latency</td>
</tr>
<tr>
<td>L2</td>
<td>512 KB, 32 byte cache line, 8-way, 16 cycle latency</td>
</tr>
<tr>
<td>Interconnection</td>
<td>2D grid topology, 14 cycle link latency</td>
</tr>
<tr>
<td>Main Memory</td>
<td>100 cycle latency</td>
</tr>
<tr>
<td>Directory</td>
<td>1 per node, 10 cycle latency</td>
</tr>
</tbody>
</table>
It Scales!

- Commit time (red) is small and decreasing, or non-existent
Results for small transactions

- Small transactions with a lot of communication magnifies commit latency
- Commit overhead does not grow with processor count, even in the worst case
Latency Tolerance

- 32 Processor system
Remote traffic bandwidth

- Comparable to published SPLASH-2
- Total bandwidth needed (at 2 GHz) between 2.5 MBps and 160 MBps
Take home

• Transactional Memory systems must scale for TM to be useful

• Lazy optimistic TM systems have inherent benefits
  ▪ Non-blocking
  ▪ Fast abort

• Lazy optimistic TM system scale
  ▪ Fast parallel commit
  ▪ Bandwidth efficiency through write-back commit
Questions?

Whew!

Jared Casper
jaredc@stanford.edu

Computer Systems Lab
Stanford University
http://tcc.stanford.edu