Agenda

Multithreaded Programming

Transactional Memory (TM)
- TM Introduction
- TM Implementation Overview
- Hardware TM Techniques
- **Software TM Techniques**

Q&A
Software Transactional Memory

Bratin Saha
Programming Systems Lab
Intel Corporation
Outline

→ Software Transactional Memory
  • Translating a language construct
  • Runtime support
  • Compiler support
→ Hybrid Transactional Memory
→ Open issues & conclusions
Compiling Atomic

... A = 5; atomic { B = A + 5; } C = B; ...

... A = 5; stmStart(); temp = stmRead(A); stmWrite(B, temp + 5); stmCommit(); C = B;

Transactional memory accessed via STM read & write functions
• Compiler inserts appropriate calls
• Code generation, control flow, optimizations in later slides

STM tracks accesses & detects data conflicts
Runtime Data Structures

Per-thread

• Transaction Descriptor
  - Read set, write set, & log
  - For validation, commit, & rollback

• Transaction Memento
  - Checkpoint of transaction descriptor
  - For nesting & partial rollback

Per-data

• Transaction Record (TxR)
  - Pointer-sized field guarding shared data
  - Track transactional state of data
    • Shared: Read-only access by multiple readers
    • Exclusive: write-only access by single owner
Mapping Data to Transaction Records

Every data item has an associated transaction record

Object granularity (Java/C#)

```java
class Foo {
    int x;
    int y;
}
```

TxR embedded in object

Cache line or word granularity (C/C++)

```c
struct Foo {
    int x;
    int y;
}
```

Address-based hash into global TxR table
Implementing Atomicity: Example

We will show one way to implement atomicity in a STM

Uses two phase locking for writes

Uses optimistic concurrency for reads

Illustrates how transaction records are used
Example

T1 atomic {
    t = foo.x;
    bar.x = t;
    t = foo.y;
    bar.y = t;
}

Reads <foo, 3> <foo, 3>
Writes <bar, 5>
Undo <bar.x, 0> <bar.y, 0>

T1 copies foo into bar
T2 should read [0, 0] or should read [9,7]

T2 atomic {
    t1 = bar.x;
    t2 = bar.y;
}

Reads <bar, 5> <bar, 7>
## Ensuring Atomicity: Options

<table>
<thead>
<tr>
<th>Memory Ops</th>
<th>Mode</th>
<th>Reads</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Pessimistic Concurrency
- **Reads**: Read lock on TxR (reader-writer lock or reader list)
- **Writes**: 

### Optimistic Concurrency
- **Reads**: Use versioning on TxR
## Ensuring Atomicity: Options

<table>
<thead>
<tr>
<th>Memory Ops →</th>
<th>Reads</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode ↓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pessimistic Concurrency</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- Caching effects</td>
<td>- Lock operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimistic Concurrency</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ Caching effects</td>
<td>+ Avoids lock operations</td>
</tr>
</tbody>
</table>

See Saha et al. PPoPP ’06 paper for quantitative results.
## Ensuring Atomicity: Options

<table>
<thead>
<tr>
<th>Memory Ops</th>
<th>Mode</th>
<th>Reads</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pessimistic Concurrency</td>
<td></td>
<td></td>
<td>Write lock on TxR</td>
</tr>
<tr>
<td>Optimistic Concurrency</td>
<td></td>
<td></td>
<td>Buffer writes &amp; acquire locks at commit</td>
</tr>
</tbody>
</table>
## Ensuring Atomicity: Options

<table>
<thead>
<tr>
<th>Memory Ops</th>
<th>Reads</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pessimistic Concurrency</td>
<td>+ In place updates + Fast commits + Fast reads</td>
<td>- Slow commits - Reads have to search for latest value</td>
</tr>
<tr>
<td>Optimistic Concurrency</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Saha et al. PPoPP '06 paper for quantitative results.
Java Virtual Machine Support

On-demand cloning of methods called inside transactions

JIT compiler automatically inserts read/write barriers
  • Maps barriers to first class opcodes in intermediate representation
  • Good compiler representation → greater optimization opportunities
  • Determine conflict detection granularity on per-type basis

Garbage collection support
  • Enumeration of references in STM data structures
  • Filtering to remove redundant log entries
  • Mappings are valid across moving GC
Representing Read/Write Barriers

Coarse-grain barriers hide redundant locking/logging

```
atomic {
    a.x = t1
    a.y = t2
    if(a.z == 0) {
        a.x = 0
        a.z = t3
    }
}
```

...  

```
stmWr(&a.x, t1)
stmWr(&a.y, t2)
if(stmRd(&a.z) != 0) {
    stmWr(&a.x, 0);
    stmWr(&a.z, t3)
}
```
An STM IR for Optimization

Redundancies exposed:

```plaintext
atomic {
    a.x = t1
    a.y = t2
    if(a.z == 0) {
        a.x = 0
        a.z = t3
    }
}
```

```plaintext
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnOpenForWrite(a)
txnLogObjectInt(&a.y, a)
a.y = t2
txnOpenForRead(a)
if(a.z != 0) {
    txnOpenForWrite(a)
    txnLogObjectInt(&a.x, a)
a.x = 0
    txnOpenForWrite(a)
    txnLogObjectInt(&a.z, a)
a.z = t3
}
```
Optimized Code

Fewer & cheaper STM operations

```c
atomic {
    a.x = t1
    a.y = t2
    if(a.z == 0) {
        a.x = 0
        a.z = t3
    }
}
```

```c
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
```

```c
txnLogObjectInt(&a.y, a)
a.y = t2
```

```c
if(a.z != 0) {
    a.x = 0
    txnLogObjectInt(&a.z, a)
    a.y = t3
}
```
Compiler Optimizations for Transactions

Standard optimizations

• CSE, Dead-code-elimination, ...
• Careful IR representation exposes opportunities and enables optimizations with almost no modifications
• Subtle in presence of nesting

STM-specific optimizations

• Immutable field / class detection & barrier removal (vtable/String)
• Transaction-local object detection & barrier removal
• Partial inlining of STM fast paths to eliminate call overhead
Effect of Compiler Optimizations

1P overheads over thread-unsafe baseline

Prior STMs typically incur ~2x on 1P
With compiler optimizations:
- < 40% over no concurrency control
- < 30% over synchronization
Hybrid TM: Combining HTM with STM

General approach:
- Try transaction using HTM first
- Fall back on STM if HTM aborts
- Atomic blocks multiversioned for HTM & STM execution

Accelerates simple transaction
- Small
- Flat transactions

STM-STM conflicts detected by the STM machinery
HTM-HTM conflicts detected by the HTM machinery
HTM-STM conflicts requires additional code in the HTM code path
Hybrid TM: Basic Mechanism

HTMReadBarrier(addr)
check transaction record for addr is not locked by a SW transaction
if (transaction record free)
    read the address
else
    abort

HTMWriteBarrier(addr)
check transaction record for addr is not locked by a SW transaction
if (transaction record is free)
    perform the write
    increment version number to indicate HTM modification
else
    abort

HTM check ensures no concurrent SW TM modification
Research challenges

Performance
- Right mix of HW & SW components
- Good diagnostics & contention management

Semantics
- I/O & communication
- Nested parallelism

Debugging & performance analysis tools

System integration
Conclusions

Multi-core architectures: an inflection point in mainstream SW development

Navigating inflection requires new parallel programming abstractions

Transactions are a better synchronization abstraction than locks
  - Software engineering and performance benefits

Lots of research on implementation and semantics issues
  - Great progress, but there are still open problems