Transactional Memory

Lecture 16

CS 501
May 21, 2013
Motivation

• Scalability no longer just about large-scale distributed systems

• Modern processors have multiple cores
  • Natural result of power / performance challenges
  • No longer just servers
    • ... but now clients: desktops, laptops, and, increasingly, mobile devices

• Good performance requires scaling over multiple cores
Threads and Locks

• Traditional programming model for concurrency
  
  • multiple threads of execution (e.g., one per core)

  • shared memory

  • locks to coordinate

• Difficult to program

  • Fine-grain locking required for good scalability

  • Avoiding deadlock, maintaining lock discipline, composing modules, ...
Transactional Memory

- Replace locked regions with *transactions*
  - sets of atomic operations that *appear* to execute sequentially

- Two key ideas
  - Optimistic concurrency in the implementation
    - Assume potentially conflicting operations can run concurrently
  - Declarative safety in the language
    - Declare *what* properties rather than *how* to get them
Programming with Locks

synchronized (data) {
    x = "apple";
}

synchronized (data) {
    y = "";
    z = "cat";
}
Programming with Locks

synchronized (data) {
    x = "apple";
}

synchronized (data) {
    z = "cat";
}
Programming with Locks

synchronized (data) {
    x = "apple";
}

synchronized (data) {
    z = "cat";
}

x: "apple"
y: ""
z: "cat"
Programming with Transactions

```
atomic {
  x = "apple";
}

atomic {
  y: ""
  z: ""
  atomic {
    z = "cat";
  }
}
```
atomic {
  x = "apple";
}

atomic {
  z = "cat";
}

x: "apple"
y: 

atomic {
  y = " ";
}

z: "cat"
Resolving Conflicts

\[
\text{atomic} \{ \\
\hspace{1em} y = \text{"banana"}; \\
\}
\]

\[
x: \text{"apple"} \\
y: \text{""} \\
z: \text{"cat"}
\]

\[
\text{atomic} \{ \\
\hspace{1em} y = \text{"bird"}; \\
\}
\]
Resolving Conflicts

```
atomic {
  y = "banana";
}

atomic {
  y = "bird";
}
```

```
x: "apple"
y: "banana"
z: "cat"
```
Resolving Conflicts

```
atomic {
  y = "banana";
}
```

```
atomic {
  y = "bird";
  z: "cat"
}
```

```
atomic {
  y = "bird";
}
```
Overview

• Motivation

• Implementing Optimistic Concurrency
  • Hardware Transactional Memory (HTM)
  • Software Transactional Memory (STM)

• Integrating Transactions into Programming Languages

• Summary
Optimistic Concurrency in Transactions

• Key idea: transactions should be performed in parallel

  • Old idea from databases ... but now, apply to memory

  • Allow independent transactions to overlap

  • Detect and recover from conflicting accesses

  • Maintain original and new versions of memory for each transaction

  • If no conflict, commit and publish new version all at once

  • If conflict, abort and restore original version
Hardware Transactional Memory

• Seminal work: Herlihy / Moss 1993

• Idea: leverage existing multi-processor hardware
  • Provide small scale transactions
  • Versioning: cache contains new values, main memory retains original
  • Conflict detection: extend existing cache coherency mechanisms
  • Commit: cached values copied into memory
  • Abort: cached values dropped
State of Commercial Hardware TM

- Azul - Large scale Java servers (> 500 cores)
  - TM used under the hood: *speculative lock elision*
    - Execute locked sections in Java as transactions
    - On failure, fall back to locks
  - Intel (Haswell), etc.,
  - Hardware TM exposed to programmers
  - Transactions limited in size and allowable operations
Software Transactional Memory

• Implement all necessary bookkeeping in software
  • Leverage Virtual machine, Compiler, GC
• Sacrifice raw performance, but gain flexibility
  • Support existing machines
  • Not bounded by hardware resources & limitations
• More powerful language constructs / semantics
Conflict Resolution

- Eager: Assume potential conflict, prevent ahead of time
  - Read: shared lock - no concurrent writers
  - Write: exclusive lock - no concurrent readers or writers

- Lazy: Assume no conflict, recover if necessary
  - Read: record & validate on commit
  - Write: buffer & validate on commit

- HTMs are typically lazy, STMs vary significantly
Versioning

- Eager: Assume transaction will commit
  - write in place to memory
  - record old values in log
  - restore on abort

- Lazy: Assume transaction is likely to abort
  - buffer writes
  - publish on commit
<table>
<thead>
<tr>
<th>Eager Detection</th>
<th>Eager Versioning</th>
<th>Lazy Detection</th>
<th>Lazy Versioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomCaml</td>
<td></td>
<td>McrtSTM*</td>
<td>STMHaskell</td>
</tr>
</tbody>
</table>
Example: McRT-STM

- C++ / Java STM from Intel (PLDI ’06, whatif.intel.com)

  - Writes:
    - Two-phase exclusive locking (eager conflict resolution)
    - Write in-place & maintain undo log (eager versioning)

  - Reads:
    - Record version number & validate on commit (lazy conflict detection)
Bookkeeping: Per-Data

- Transaction record
  - Protects corresponding set of data (analogous to lock)
  - Not user visible - managed by implementation
    - STM automatically checks/updates record on every load or store
  - Pointer-sized value in one of two states
    - Exclusive / write: pointer to owning thread (even/aligned)
    - Shared / read: version number of data (odd)
Bookkeeping: Per-Thread

• Read set: \{ (<record>, <version number>), ... \}
  • Unlocked: append record and its current version to read set
  • Locked by another thread: either wait or abort

• Write set: \{ (<record>, <version number>), ... \}
  • Undo set: \{ (<address>, <value>), ... \}
    • Unlocked: lock, append to write set, record old value, and write new value
    • Locked by this thread: record old value and write new value
Optimistic Concurrency with STM

Transaction Record →

protects

\{ 

<table>
<thead>
<tr>
<th>Transaction Record</th>
<th>Object A</th>
<th>Object B</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>vtable</td>
<td>type</td>
</tr>
<tr>
<td>rec</td>
<td>7</td>
<td>rec</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
<td>y</td>
</tr>
</tbody>
</table>

← Version Number (odd)
Optimistic Concurrency with STM

Atomically copy A into B

Object A

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

Object B

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>
Optimistic Concurrency with STM

Atomically copy A into B

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Object A</th>
<th>Object B</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic {</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tmp = A.x;</td>
<td>type</td>
<td>vtable</td>
<td>type</td>
</tr>
<tr>
<td>B.x = tmp;</td>
<td>rec</td>
<td>7</td>
<td>rec</td>
</tr>
<tr>
<td>tmp = A.y;</td>
<td>x</td>
<td>10</td>
<td>x</td>
</tr>
<tr>
<td>B.y = tmp;</td>
<td>y</td>
<td>20</td>
<td>y</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td>atomic {</td>
</tr>
</tbody>
</table>
|            | p    | B.x;   | p = B.x;
|            | q    | B.y;   | q = B.y;
|            | }    |        | }
Optimistic Concurrency with STM

Atomically copy A into B

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Object A</th>
<th>Object B</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic {</td>
<td>type vtable</td>
<td>type vtable</td>
<td>atomic {</td>
</tr>
<tr>
<td>tmp = A.x;</td>
<td>rec 7 x 10</td>
<td>rec 1 x 0</td>
<td>p = B.x;</td>
</tr>
<tr>
<td>B.x = tmp;</td>
<td>x 10</td>
<td>x 0</td>
<td>q = B.y;</td>
</tr>
<tr>
<td>tmp = A.y;</td>
<td>y 20</td>
<td>y 0</td>
<td>}</td>
</tr>
<tr>
<td>B.y = tmp;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Only valid results: \((p, q) = (0, 0)\) or \((p, q) = (10, 20)\)
Optimistic Concurrency with STM

Thread 1

atomic {
    tmp = A.x;
    B.x = tmp;
    tmp = A.y;
    B.y = tmp;
}

Read Set:  {}
Write Set:  {}
Undo Set:  {}

Object A

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

Object B

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>

Thread 2

atomic {
    p = B.x;
    q = B.y;
}

Read Set:  {}
Write Set:  {}
Undo Set:  {}
# Optimistic Concurrency with STM

## Thread 1

```plaintext
atomic {
    tmp = A.x;
    B.x = tmp;
    tmp = A.y;
    B.y = tmp;
}
```

<table>
<thead>
<tr>
<th></th>
<th>Object A</th>
<th>Object B</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>vtable</td>
<td>type</td>
</tr>
<tr>
<td>rec</td>
<td>7</td>
<td>rec</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
<td>y</td>
</tr>
</tbody>
</table>

- **Read Set:** \{ \}
- **Write Set:** \{ \}
- **Undo Set:** \{ \}

## Thread 2

```plaintext
atomic {
    p = B.x;
    q = B.y;
}
```

- **Read Set:** \{ (B, 1) \}
- **Write Set:** \{ \}
- **Undo Set:** \{ \}

- `p == 0`
Optimistic Concurrency with STM

Thread 1

atomic {
  tmp = A.x;
  B.x = tmp;
  tmp = A.y;
  B.y = tmp;
}

Thread 2

atomic {
  p = B.x;
  q = B.y;
}

Read Set: \{ (A, 7) \}
Write Set: \{ \}
Undo Set: \{ \}

tmp == 10

Read Set: \{ (B, 1) \}
Write Set: \{ \}
Undo Set: \{ \}
p == 0

Object A

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

Object B

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>
### Optimistic Concurrency with STM

**Thread 1**

```plaintext
atomic {
    tmp = A.x;
    ➤
    B.x = tmp;
    tmp = A.y;
    B.y = tmp;
}
```

**Object A**

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

**Object B**

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>Txn1</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>

**Thread 2**

```plaintext
atomic {
    ➤
    p = B.x;
    q = B.y;
    }
```

**Read Set:**

- `{ (A, 7) }`
- `{ (B, 1) }`
- `{ (B, 1) }
- `{ }`

**Write Set:**

- `{ (B, 1) }
- `{ }`

**Undo Set:**

- `{ (&B.x, 0) }`
- `{ }`

$p == 0$
Optimistic Concurrency with STM

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic {</td>
<td>Atomic {</td>
</tr>
<tr>
<td>tmp = A.x;</td>
<td>p = B.x;</td>
</tr>
<tr>
<td>B.x = tmp;</td>
<td>q = B.y;</td>
</tr>
<tr>
<td>tmp = A.y;</td>
<td>}</td>
</tr>
<tr>
<td>B.y = tmp;</td>
<td>}</td>
</tr>
</tbody>
</table>

Read Set: \{ (A, 7), (A, 7) \}  Read Set: \{ (B, 1) \}
Write Set: \{ (B, 1) \}  Write Set: \{ \} 
Undo Set: \{ (&B.x, 0) \}  Undo Set: \{ \} 

<table>
<thead>
<tr>
<th>Object A</th>
<th>Object B</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>vtable</td>
</tr>
<tr>
<td>rec</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
<tr>
<td>type</td>
<td>vtable</td>
</tr>
<tr>
<td>rec</td>
<td>Txn1</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>

tmp == \textbf{20}
p == 0

Monday, May 27, 13
Optimistic Concurrency with STM

Thread 1

\[
\begin{align*}
\text{atomic ~} & \quad \text{tmp} = A.x; \\
& \quad B.x = \text{tmp}; \\
& \quad \text{tmp} = A.y; \quad \text{➤ B.y = tmp;}
\end{align*}
\]

Read Set: \ \{ (A, 7), (A, 7) \}

Write Set: \ \{ (B, 1) \}

Undo Set: \ \{ (&B.x, 0), (B.y, 0) \}

Thread 2

\[
\begin{align*}
\text{atomic ~} & \quad p = B.x; \quad q = B.y; \\
\end{align*}
\]

Read Set: \ \{ (B, 1) \}

Write Set: \ \{ \}

Undo Set: \ \{ \}

\[
\begin{array}{|c|c|}
\hline
\text{type} & \text{vtable} \\
\hline
\text{rec} & 7 \\
\hline
x & 10 \\
\hline
y & 20 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{type} & \text{vtable} \\
\hline
\text{rec} & \text{Txn1} \\
\hline
x & 10 \\
\hline
y & 20 \\
\hline
\end{array}
\]
Optimistic Concurrency with STM

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Object A</th>
<th>Object B</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic {</td>
<td></td>
<td></td>
<td>atomic {</td>
</tr>
<tr>
<td>tmp = A.x;</td>
<td></td>
<td></td>
<td>p = B.x;</td>
</tr>
<tr>
<td>B.x = tmp;</td>
<td></td>
<td></td>
<td>q = B.y;</td>
</tr>
<tr>
<td>tmp = A.y;</td>
<td></td>
<td>vtable</td>
<td>}</td>
</tr>
<tr>
<td>B.y = tmp;</td>
<td></td>
<td>rec</td>
<td>}</td>
</tr>
</tbody>
</table>

✔

Read Set: \( \{ (A, 7), (A, 7) \} \)
Write Set: \( \{ (B, 1) \} \)
Undo Set: \( \{ (&B.x, 0), (&B.y, 0) \} \)

Read Set: \( \{ (B, 1) \} \)
Write Set: \( \{ \} \)
Undo Set: \( \{ \} \)

p == 0

Monday, May 27, 13
Optimistic Concurrency with STM

### Thread 1

```
atomic {
    tmp = A.x;
    B.x = tmp;
    tmp = A.y;
    B.y = tmp;
}
```

Read Set: \( \{(A, 7), (A, 7)\} \)
Write Set: \( \{(B, 1)\} \)
Undo Set: \( \{(&B.x, 0), (&B.y, 0)\} \)

### Thread 2

```
atomic {
    p = B.x;
    q = B.y;
}
```

Read Set: \( \{(B, 1)\} \)
Write Set: \( \{\}\) 
Undo Set: \( \{\}\) 
\( p == 0 \)
Optimistic Concurrency with STM

Thread 1

\[
\text{atomic} \{ \\
\text{tmp} = A.x; \\
B.x = \text{tmp}; \\
\text{tmp} = A.y; \\
B.y = \text{tmp}; \\
\}
\]

Read Set: \{ \}
Write Set: \{ \}
Undo Set: \{ \}

Object A

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

Object B

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>3</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

Thread 2

\[
\text{atomic} \{ \\
\text{p} = B.x; \\
q = B.y; \\
\}
\]

Read Set: \{ (B, 1) \}
Write Set: \{ \}
Undo Set: \{ \}

p == 0
Optimistic Concurrency with STM

| Thread 1 |
|------------------|------------------|
| atomic {
    | tmp = A.x;
    | B.x = tmp;
    | tmp = A.y;
    | B.y = tmp;
|}

<table>
<thead>
<tr>
<th>Object A</th>
<th>Object B</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>vtable</td>
</tr>
<tr>
<td>rec</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 2</th>
</tr>
</thead>
</table>
| atomic {
    | p = B.x;
    | q = B.y;
|}

Read Set: {} \hspace{1cm} Read Set: \{(B, 1), (B, 3)\}
Write Set: {} \hspace{1cm} Write Set: {}
Undo Set: {} \hspace{1cm} Undo Set: {}

p == 0 \hspace{1cm} q == 20
Optimistic Concurrency with STM

Thread 1
atomic {
  tmp = A.x;
  B.x = tmp;
  tmp = A.y;
  B.y = tmp;
}

Thread 2
atomic {
  p = B.x;
  q = B.y;
}

Object A
<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

Object B
<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>3</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

Read Set: {} Read Set: {(B, 1), (B, 3)}
Write Set: {} Write Set: {}
Undo Set: {} Undo Set: {}

p == 0  q == 20
Optimistic Concurrency with STM

Thread 1
─ atomic {
  tmp = A.x;
  B.x = tmp;
  tmp = A.y;
  B.y = tmp;
}

Read Set:  
Write Set:  
Undo Set:  

Thread 2
─ atomic {
  p = B.x;
  q = B.y;
}

Read Set:  
Write Set:  
Undo Set:  

Object A

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>7</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

Object B

<table>
<thead>
<tr>
<th>type</th>
<th>vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec</td>
<td>3</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>
Optimistic Concurrency with STM

Thread 1
atomic {
    tmp = A.x;
    B.x = tmp;
    tmp = A.y;
    B.y = tmp;
}

Thread 2
atomic {
    p = B.x;
    q = B.y;
}

Read Set: {} Read Set: { (B, 3) }
Write Set: {} Write Set: {}
Undo Set: {} Undo Set: {}
p == 10
Optimistic Concurrency with STM

Thread 1
atomic {
    tmp = A.x;
    B.x = tmp;
    tmp = A.y;
    B.y = tmp;
}

Read Set:  
Write Set:  
Undo Set:  

Thread 2
atomic {
    p = B.x;
    q = B.y;
}

Read Set:  
Write Set:  
Undo Set:  
p == 10        q == 20
Optimistic Concurrency with STM

Thread 1
atomic {
    tmp = A.x;
    B.x = tmp;
    tmp = A.y;
    B.y = tmp;
}

Thread 2
atomic {
    p = B.x;
    q = B.y;
    ✔
}

Read Set:    { }    Read Set:    { (B, 3), (B, 3) }
Write Set:    { }    Write Set:    { }
Undo Set:    { }    Undo Set:    { }

p == 10    q == 20
Overview

• Motivation

• Implementing Optimistic Concurrency

• Integrating Transactions into Programming Languages

• Summary
Transactions in Programming Languages

- Major value in transactions is not just scalability, but programmability

- Transactions offers much simpler programming model compared to locks
  
  - Declarative vs operational

  - Composable: atomic \{ operation1, operation2 \}

  - Stronger guarantees closer to what programmers really want
  
    - Deadlock avoidance

  - Atomicity, Consistency, Isolation
Memory Transactions

- **ACID** properties from database transactions

  - **Atomicity**: Provide appearance of instantaneous execution. ✓
  
  - **Consistency**: Enforce validity of memory state. ✓
  
  - **Isolation**: Prevent observation of intermediate state. ✓
  
  - **Durability**: Unlike disk, RAM is not persistent. ✗
class StringBuffer {

    public synchronized void append(StringBuffer buffer) {
        int m = this.size();
        int n = buffer.size();
        this.resize(m+n);
        for(int i = 0; i < n; ++i)
            this.array[m+i] = buffer.array[i];
    }
}

• Distillation of old bug in Java class library (Flanagan PLDI ’03)

• This code is not thread-safe ....

• synchronized != thread-safe
class StringBuffer {
    ...
    public synchronized void append(StringBuffer buffer) {
        int m = this.size();
        int n = buffer.size();
        this.resize(m + n);
        for(int i = 0; i < n; ++i)
            this.array[m+i] = buffer.array[i];
    }
}

• Subtle bug: argument buffer is not locked

• Another thread may change buffer (including its size) during append

• append may copy inconsistent data or generate exception
Locks and atomicity

```java
class StringBuffer {
...
    public atomic void append(StringBuffer buffer) {
        int m = this.size();
        int n = buffer.size();
        this.resize(m+n);
        for(int i = 0; i < n; ++i)
            this.array[m+i] = buffer.array[i];
    }
}
```

- Thread-safe: another transaction cannot interfere with append
Condition Synchronization

• Retry (from STM Haskell)

  • Transaction cannot make progress
    - must block

  • Wait for a change in memory state
    (encapsulated by read set)

  • On state change, restart
    transaction

• Elegant alternative to wait / notify

Object blocking_deque(...) {
    // Block until a condition holds
    atomic {
        if(isEmpty())
            retry;
        return ...;
    }
}
Alternatives

• OrElse (from STM Haskell)

  • Multiple alternatives
  • Exactly one executed atomically
  • Left to right bias
  • Retry shifts to next alternative

```java
Object nonblocking_deque(...) ... {
    // Return an error instead of waiting
    atomic {
        blocking_deque(...);
    } orelse {
        throw new QueueEmptyException();
    }
}
```
Composability

- Atomic, retry, orelse are powerful building blocks

```java
void transfer(Account from1, Account from2, Account to, int amount) {
    atomic {
        atomic {
            from1.withdraw(amount);
        } orelse {
            from2.withdraw(amount);
        }
        to.deposit(amount);
    }
}
```

- Code above will block until either account has sufficient funds (nested transaction)

- Withdraw combined atomically with deposit
Leveraging the Compiler / Runtime

- StarJIT / ORP: High-performance Java runtime
  - Identify transactional regions in Java STM code
  - Map STM API to first class opcodes in IR
  - Insert Read and Write barriers in transactional code
  - On GC, enumerate STM data structures
- Good compiler representation -> good optimization opportunities
Representing Read/Write Barriers

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

Redundancies hidden

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:

```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
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

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

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

Fewer & cheaper operations
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
Problems ...

- Transactions are isolated only from other transactions
  - Most STMs behave this way
  - Non-transactional accesses bypass STM access protocol

- Requires segregation of transactional and non-transactional data
  - Hard to enforce and error-prone
  - Violations lead to subtle bugs
Unintuitive Behavior

Thread 1

atomic  {
   if (a != null)
      r = a.x
}

Monday, May 27, 13
Unintuitive Behavior

Thread 1

```java
atomic {
    if (a != null)
    r = a.x
}
```

Thread 2

```java
a = null
```

Monday, May 27, 13
Unintuitive Behavior

Thread 1

```java
atomic {
    if (a != null)
        r = a.x
}
```

Thread 2

```java
a = null
```

Null Pointer Exception
Unintuitive Behavior

Thread 1

```java
atomic {
    if (a != null)
        r = a.x
}
```

Thread 2

```java
a = null
```

Null Pointer Exception

Locks behave the same way
Unintuitive Behavior

Thread 1

```
atomic {
  if (a != null)
    r = a.x
}
```

Thread 2

```
a = null
```

Null Pointer Exception

Locks behave the same way

*Things get worse ...*
Privatization Example: Locks
[Larus & Rajwar 2006, Hudson 2006]

Thread 1

```java
synchronized (list) {
    e = list.removeFirst();
}

r1 = e.x;
r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
synchronized (list) {
    if (!list.isEmpty()) {
        e = list.getFirst();
        e.x = 1;
    }
}
```

![Diagram showing list structures and operations]
Privatization Example: Locks

[Larus & Rajwar 2006, Hudson 2006]

Thread 1

```java
synchronized (list) {
    e = list.removeFirst();
}

r1 = e.x;
r2 = e.x;
```

Thread 2

```java
synchronized (list) {
    if (!list.isEmpty()) {
        e = list.getFirst();
        e.x = 1;
    }
}
```

Can r1 != r2?

No

This program is correctly synchronized
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

list

 Monday, May 27, 13
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

Diagram:

```
list
   0       0
     ↓       ↓
     0       0
```

Monday, May 27, 13
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}

r1 = e.x;
r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

Can r1 != r2?
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}

r1 = e.x;

r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

list

p

Monday, May 27, 13
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}
```

r1 = e.x;

r2 = e.x;

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

list

p

0

0

Monday, May 27, 13
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

list

p

0

0
Privatization Example: Weak Atomicity

Thread 1

atomic {
    e = list.removeFirst();
}

r1 = e.x;
r2 = e.x;

Can r1 != r2?

Thread 2

atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}

Can r1 != r2?
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

Monday, May 27, 13
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}

r1 = e.x;
r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

Monday, May 27, 13
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}
```

`r1 = e.x;`

`r2 = e.x;`

Can `r1 != r2`?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

`list`

```
0   1
```

`p` -> `1` -> `0`
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}

r1 = e.x;
r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

Abort

Monday, May 27, 13
Privatization Example: Weak Atomicity

Thread 1

atomic {
    e = list.removeFirst();
}

r1 = e.x;
r2 = e.x;

Can r1 != r2?

Thread 2

atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}

Abort
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
```

Can r1 != r2?

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

Abort
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}
```

r1 = e.x;

r2 = e.x;

Can r1 != r2?

Yes!

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

Abort

Can r1 != r2? Yes!
Privatization Example: Weak Atomicity

Thread 1

```java
atomic {
    e = list.removeFirst();
}

r1 = e.x;

r2 = e.x;
```

Can r1 != r2? Yes!

Lazy-versioning? Yes!

Thread 2

```java
atomic {
    if (!list.isEmpty()) {
        p = list.getFirst();
        p.x = 1;
    }
}
```

Abort

Lazy-versioning? Yes!

Can r1 != r2? Yes!
Weak Atomicity Pitfalls

• Weak atomicity exhibits subtle unintuitive behavior
  – Can fail where locks work

  See paper for taxonomy of issues
  – Speculative dirty read
  – Ordering violations in lazy-versioning STMs
  – Granular lost update
  – Granular inconsistent read
  and many others

  Behavior depends on STM implementation
Strong Atomicity

• Avoids **all** undesirable behaviors of weak atomicity

• **Isolates transactions from all** memory accesses
  – Intuitive behavior with clean semantics

• **Requires barriers for non**–**transactional accesses**
  – Can affect performance of non–transactional code
• First strongly atomic multi-processor, scalable STM

• Non-transactional memory accesses
  – Heavily optimized ‘mini-transactions’
  – Efficient inlined code sequences

Uses run-time and compiler optimizations to reduce the overheads
Dynamic Escape Analysis

Tracks thread-local objects at run-time
– Objects reachable from static fields may be visible to multiple threads

Avoids unnecessary operations in barriers
– Shortens fast path to a single compare-branch
JIT Optimizations

• Simple barrier elimination
  – Immutable data
  –Thread-local objects

• Local barrier aggregation
  – Combines multiple accesses to the same object into a single transaction
Whole-Program Optimizations

• Not-Accessed-In-Transaction Analysis (NAIT)
  – Objects not accessed in transactions do not require barriers
  – Object not written in transactions do not require read barriers

• Thread-Local Analysis (TL)
  – Classical optimization
  – Eliminates barriers for thread-local objects
Huge variations

Dynamic escape analysis is usually highly effective
Strongly atomic STM scales as well as weakly atomic STM
Criticisms

• Strongly atomic STM implementations have 2 major pitfalls:
  – They slow down non-transactional code
    • Penalizes programs that use little or no transactions
    • Not pay-as-you-go
  – They require whole-program-optimizations for best performance
    • Not always viable in production settings

• Alternative approach
  – Make weak STM sane
  – Augment weakly atomic STMs to handle privatization
  – As good as locks?
Example: Publication Idiom

Initially data = 42, ready = false, val = 0

Thread 1

```
data = 1
atomic { // transaction T1
  ready = true;
}
```

Thread 2

```
atomic { // transaction T2
  if (ready)
    val = data;
}
Can val == 42?
```

Global variable data is accessed in & out of transaction

Program is still race-free under locks
- If T1 before T2 then val == 1
- If T2 before T1 then val is not written (i.e., val == 0)

This behavior is guaranteed by locks and STMs (Java, C++, ...).
Benign Modification?

Initially data = 42, ready = false, val = 0

Thread 1

    data = 1
    atomic { // transaction T1
        ready = true;
    }

Thread 2

    atomic { // transaction T2
        tmp = data;
        if (ready)
            val = tmp;
    }

Can val == 42?

Access hoisted: compiler (speculative code motion) or STM (early copying)

- Apparent benign race introduced

With locks, we still expect same behavior as before:

- If T1 before T2 then val == 1; if T2 before T1 then val == 0 & tmp is dead
- val != 42 guaranteed by Java ... but, can break on most STMs
Publication Example in STM

Initially data = 42, ready = false, val = 0

Thread 1

1:    
2:    
3:    data = 1
4:      atomic { // transaction T1
5:        ready = true;
6:      }
7:    
8:    
9:    

Thread 2

10:   atomic { // transaction T2
11:       tmp = data;
12:     
13:       if (ready)
14:          val = tmp; // val == 42
15:     }

Most STMs can produce val == 42 because transactions can overlap
What is a correct execution?

- Sequential consistency (Lamport ‘79)
  - All threads agree on some total ordering
  - Observable effects must be consistent that total ordering
  - On single thread: allows standard compiler & hardware opts
  - Multiple threads: much more limiting
- Synchronization models (Adve/Hill ‘90)
  - Define correctly synchronized programs
  - Guarantee sequential consistency only for that subset
- How does TM fit in?
Java’s Memory Model (JMM)

• Data-race freedom == correctly synchronized
  - Strong guarantee:
    - Sequentially consistent behavior for race-free programs

• For racy programs
  - Safety and security paramount
  - No values “out-of-thin-air”
  - “Happens-before” ordering must be obeyed
  - JMM allows for benign races
Implications of JMM on STM

• Preventing out-of-thin-air values
  - Granular safety: no lost updates / inc. reads due to adjacent writes
  - Observable consistency: no artificial races due to inconsistent execution
  - Speculation safety: no visible speculative effects

• Preserving happens-before ordering
  - Privatization safety: order fromtxn access to non-txn access
  - Publication safety: order from non-txn access to txn access

• See SPAA ‘08 paper for details …
“Synchronization” Model for TM

• Specifies what values can be seen by the reads
  - Allows programmers to reason about their code
    - E.g, defines happens-before relations imposed by transactions
    - Determines what compiler transformations are legal
  - Sets the rules for TM implementation

Conflicting requirements

– Simplicity
  • Easy to use by non-expert programmer

– Flexibility
  • Allow for efficient TM implementation
Single Global Lock Atomicity (SGLA)

• Transactions execute as if they are protected by a single global lock

    atomic { synchronized (global_lock) {
    S;  ←→  S;
    } }

• Matches intuition of weakly atomic STM
  - Transactions are serialized wrt each other
  - Sequential consistency for race–free programs
  - In Java, well-defined behavior for races

• Has surprising consequences for TM implementations
Publication via Empty Transaction

Initially data = 42, ready = false, val = 0

Thread 1

\[
\begin{align*}
\text{data} &= 1 \quad // \text{S1} \\
\text{atomic} \quad \{} \quad // \text{T1} \\
\text{ready} &= \text{true};
\end{align*}
\]

Thread 2

\[
\begin{align*}
\text{atomic} \quad \{} \quad // \text{T2} \\
\text{tmp} &= \text{data}; \quad // \text{S2} \\
\text{if} \; (\text{ready}) \\
\text{val} &= \text{tmp}; \\
\}
\]

Can \(\text{val} == 42?\)

If T1 before T2 then \(\text{val} == 0\) or \(\text{val} == 1\)

If T1 after T2 then \(\text{ready} == \text{false}\) and \(\text{val}\) is not read

Under SGLA, empty transactions impose ordering constraints
Empty Transaction in STM

Initially data = 42, ready = false, val = 0

Thread 1
1: data = 1  // S1
2: atomic {}  // T1
3: ready = true;

Thread 2
atomic {}  // T2
tmp = data;  // S2

if (ready)
  val = tmp;
}

Most STMs can produce val == 42 because transactions can overlap

Difficult to provide concurrency in STM under SGLA restriction
Disjoint Lock Atomicity (DLA)

• Weaken SGLA: Only dynamically conflicting transactions execute as if they are protected by the same lock
  - T1 and T2 conflict if both access location x and one writes

  Same as single global lock atomicity for race-free programs

  Eliminates unnecessary (?) ordering constraints in racy code
  - Does not require handling publication via empty transaction

  Less intuitive than single global lock atomicity
  - Cannot statically construct an equivalent lock-based program
  - As if locks are “magically” acquired at transaction start
Publication via Anti-Dependence

Initially data = 42, ready = false, val = 0

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{data = 1} // S1</td>
<td>\texttt{atomic {}</td>
</tr>
<tr>
<td>atomic {}</td>
<td>\texttt{tmp = data; // S2}</td>
</tr>
<tr>
<td>\texttt{test = ready;} }</td>
<td>\texttt{ready = true;}</td>
</tr>
<tr>
<td></td>
<td>\texttt{val = tmp;}</td>
</tr>
</tbody>
</table>

Can test == false and val == 42 ?

Under SGLA & DLA: No

- If T1 before T2 then test == false and val == 1
  - Anti-dependence from T1 to T2 through ready
- If T1 after T2 then test == true

Complication: “invisible” read in T1 not detectable by T2 in STM
Asymmetric Lock Atomicity

Transactions execute as if every memory access is protected by a lock
- Read locks "magically" acquired at transaction start (like DLA)
- Write locks acquired lazily any time before first access (unlike DLA)
- All locks released at the end of transaction
- Provides sequential consistency for race-free programs
- Supports benign race in publication by flow dependence
Revisiting Pub. by Anti-dependence

Thread 1

data = 1;
atomic {
    // T1
    // read lock L_{ready}
    ...
    test = ready;
} // release L_{ready}

Thread 2

atomic {
    // T2
    // read lock L_{data}
    val = data
    // write lock L_{ready}
    ready = true;
    tmp = val
} // release L_{ready}, L_{data}

ALA does not support publication by anti-dependence

- No ordering between write of data (Thread 1) and read of data (Thread 2)
- Consequence of encounter-time write locking
- Easier to implement: T2 does not need be aware of read of ready in T1
  - Only needs to detect conflict with earlier writes
  - Invisible readers not problematic
Encounter-time Lock Atomicity

Thread 1

```c
data = 1;
atomic { // T1
    ...
    // write lock L_{ready}
    ready = true;
} // release L_{ready}
```

Thread 2

```c
atomic { // T2
    // read lock L_{data}
    val = data
    // read lock L_{ready}
    if(ready)
        tmp = val
} // release L_{ready}, L_{data}
```

Relax ALA to acquire all locks lazily

- No “magic” - effectively models a pessimistic, encounter-time locking STM
- Still provides sequential consistency for race-free programs
- Less tolerant of benign races
  - Does not support racy publication via flow dependence
  - Programmer must avoid benign races
  - Restrictions on compiler optimization / STM implementation
- Still requires privatization safety (e.g., quiescence with invisible readers)
# Summary of Models

<table>
<thead>
<tr>
<th>Safe Publication by</th>
<th>Linearization / Ordering Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty Txn</td>
<td>Anti-Dep</td>
</tr>
<tr>
<td>SGLA</td>
<td>Y</td>
</tr>
<tr>
<td>DLA</td>
<td>N</td>
</tr>
<tr>
<td>ALA</td>
<td>N</td>
</tr>
<tr>
<td>ELA</td>
<td>N</td>
</tr>
</tbody>
</table>

Implementation: Write buffering (no out-of-thin air) + Linearization (ordering)
- See paper for details
Traveling Sales Person Solver

Most time outside transactions (high strong atomicity overhead)
Program has a benign data race on current shortest path length
Single global lock atomicity same cost as unsafe write buffering
Strong atomicity is better for 16 threads
java.util.TreeMap

Synthetic workload: 80% gets, 20% updates
- Virtually all time is spent in transactions

SGLA and DLA scale up to 4 threads, degrade quickly beyond
ALA and ELA level out beyond 4 threads
- Privatization safety (quiescence) is still expensive
Dynamic NAIT analysis

• General idea:
  – Start optimistically: speculatively avoid generating SA barriers during compilation in the JIT
  – Add SA barriers on demand during runtime
  – 3 categories for memory locations in D–NAIT:
    • TxNone: Not accessed in a transaction
    • TxRead: Read only inside a transaction
    • TxAll: Read and written inside a transaction
Where do we need SA barriers in non-transactional code?

<table>
<thead>
<tr>
<th></th>
<th>Thread-local data</th>
<th>Shared data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never accessed inside transaction</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Read only inside transactional</td>
<td>None</td>
<td>Write barriers only</td>
</tr>
<tr>
<td>Read/write inside transactional</td>
<td>None</td>
<td>Read + Write barriers</td>
</tr>
</tbody>
</table>
D–NAIT state diagram

TxAll (read +written)

TxRead (read only)

TxNone (no transactional access)
D-NAIT state diagram

TxNone (no transactional access)

TxRead (read only)

TxAll (read +written)

Access: transactional load
Patch: write barriers

TxNone (no transactional access)
D–NAIT state diagram

TxAll (read + written)

Access: transactional store
Patch: read+write barriers

TxRead (read only)

Access: transactional load
Patch: write barriers

TxNone (no transactional access)
D-NAIT state diagram

- **TxNone (no transactional access)**
  - Access: transactional store
  - Patch: read barriers

- **TxRead (read only)**
  - Access: transactional load
  - Patch: write barriers

- **TxAll (read +written)**
  - Access: transactional store
  - Patch: read+write barriers

Monday, May 27, 13
D-NAIT state diagram

Access: transactional load + transactional store
Patch: none

Access: transactional load
Patch: none

Access: transactional store
Patch: read+write barriers

Access: transactional store
Patch: read barriers

Access: transactional load
Patch: write barriers

Access: transactional store
Patch: read barriers

Access: transactional load + transactional store
Patch: none
Simple D-NAIT

– Track state info per type or field, e.g.
  • String.value, Vector.array, char[], Object[]

– Optimistically start with all objects in state TxNone.
  • Generate phantom barriers: code prepared for later patching

– State transitions may require converting phantom barriers to real SA barriers by runtime patching

– On method compilation scan IR
  • Record transactional loads/stores + update D-NAIT states
D–NAIT example

- class A {
  - int[] x;
  - int[] y;
  - int[] z;
  - void m1();
  - void m2(int[] tmp);
  }

Monday, May 27, 13
D-NAIT example: Compiling

void m1() {
    S0: x = new int[N];
    S1: y = new int[N];
    S2: z = new int[N];
    S3: ... = y[i];
    S4: z[i] = ...;
    S5: ... = x[i];
    S6: y[i] = ...;
}

void m2(int[] tmp) {
    atomic {
        S7: ... = tmp[i];
        S8: x[i] = ...;
    }
}

<table>
<thead>
<tr>
<th>After m1</th>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Monday, May 27, 13
D-NAIT example: Compiling

- void m1() {
  - S0: x = new int[N];
  - S1: y = new int[N];
  - S2: z = new int[N];
  - S3: ... = y[i];
  - S4: z[i] = ...;
  - S5: ... = x[i];
  - S6: y[i] = ...;
}

- void m2(int[] tmp) {
  atomic {
    - S7: ... = tmp[i];
    - S8: x[i] = ...;
  }
}

<table>
<thead>
<tr>
<th>After m1</th>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td></td>
<td>S0</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
D-NAIT example: Compiling

```java
void m1() {
    S0:     x = new int[N];
    S1:     y = new int[N];
    S2:     z = new int[N];
    S3:     ... = y[i];
    S4:     z[i] = ...;
    S5:     ... = x[i];
    S6:     y[i] = ...;
}

void m2(int[] tmp) {
    atomic {
        S7:       ... = tmp[i];
        S8:       x[i] = ...;
    }
}
```

<table>
<thead>
<tr>
<th>After m1</th>
<th>After m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAIT-state</td>
<td>Reads-to-be-patched</td>
</tr>
<tr>
<td>A.x</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
</tr>
</tbody>
</table>
### D–NAIT example: Compiling

- void m1() {
  - S0: x = new int[N];
  - S1: y = new int[N];
  - S2: z = new int[N];
  - S3: ... = y[i];
  - S4: z[i] = ...;
  - S5: ... = x[i];
  - S6: y[i] = ...;
  - }

- void m2(int[] tmp) {
  - atomic {
    - S7: ... = tmp[i];
    - S8: x[i] = ...;
  - }

<table>
<thead>
<tr>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td></td>
<td>S0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td></td>
<td>S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td></td>
<td>S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### D-NAIT example: Compiling

```java
void m1() {
    S0: x = new int[N];
    S1: y = new int[N];
    S2: z = new int[N];
    S3: ... = y[i];
    S4: z[i] = ...
    S5: ... = x[i];
    S6: y[i] = ...
}
```

```java
void m2(int[] tmp) {
    atomic {
    S7: ... = tmp[i];
    S8: x[i] = ...;
    }
}
```

<table>
<thead>
<tr>
<th>After m1</th>
<th>After m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAIT-state</td>
<td>Reads-to-be-patched</td>
</tr>
<tr>
<td>A.x</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
</tr>
</tbody>
</table>

Monday, May 27, 13
D-NAIT example: Compiling

- void m1() {
  S0: x = new int[N];
  S1: y = new int[N];
  S2: z = new int[N];
  S3: ... = y[i];
  S4: z[i] = ...;
  S5: ... = x[i];
  S6: y[i] = ...;
}

- void m2(int[] tmp) {
  atomic {
    S7: ... = tmp[i];
    S8: x[i] = ...;
  }
}

<table>
<thead>
<tr>
<th>After m1</th>
<th></th>
<th>After m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAIT-state</td>
<td>Reads-to-be-patched</td>
<td>Writes-to-be-patched</td>
</tr>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td></td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td>S3</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td>S3</td>
</tr>
</tbody>
</table>
### D-NAIT example: Compiling

```java
• void m1() {
  • S0: x = new int[N];
  • S1: y = new int[N];
  • S2: z = new int[N];
  • S3: ... = y[i];
  • S4: z[i] = ...;
  • S5: ... = x[i];
  • S6: y[i] = ...;
  }

• void m2(int[] tmp) {
  atomic {
    • S7: ... = tmp[i];
    • S8: x[i] = ...;
  }
}
```

<table>
<thead>
<tr>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td>S0</td>
<td>A.x</td>
<td>TxNone</td>
<td>S0</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td>S3</td>
<td>A.y</td>
<td>TxNone</td>
<td>S3</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td>S4</td>
<td>A.z</td>
<td>TxNone</td>
<td>S4</td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td>S3</td>
<td>int[]</td>
<td>TxNone</td>
<td>S3</td>
</tr>
</tbody>
</table>
D-NAIT example: Compiling

void m1() {
    void m2(int[] tmp) {
        atomic {
            S7: ... = tmp[i];
            S8: x[i] = ...;
        }
    }
}

<table>
<thead>
<tr>
<th>After m1</th>
<th>After m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAIT-state</td>
<td>Reads-to-be-patched</td>
</tr>
<tr>
<td>A.x</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
</tr>
</tbody>
</table>
### D-NAIT example: Compiling

```java
void m1() {
    S0: x = new int[N];
    S1: y = new int[N];
    S2: z = new int[N];
    S3: ... = y[i];
    S4: z[i] = ...;
    S5: ... = x[i];
    S6: y[i] = ...;
}

void m2(int[] tmp) {
    atomic {
        S7: ... = tmp[i];
        S8: x[i] = ...;
    }
}
```

<table>
<thead>
<tr>
<th>After m1</th>
<th>After m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAIT-state</td>
<td>Reads-to-be-patched</td>
</tr>
<tr>
<td>A.x</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
</tr>
</tbody>
</table>
## D-NAIT example: Compiling

```java
void m1() {
    S0: x = new int[N];
    S1: y = new int[N];
    S2: z = new int[N];
    S3: ... = y[i];
    S4: z[i] = ...;
    S5: ... = x[i];
    S6: y[i] = ...;
}
```

```java
void m2(int[] tmp) {
    atomic {
        S7: ... = tmp[i];
        S8: x[i] = ...;
    }
}
```

<table>
<thead>
<tr>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td>S5</td>
<td>S0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td>S3</td>
<td>S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td>S4</td>
<td>S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td>S3, S5</td>
<td>S4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
D-NAIT example: Compiling

```java
void m1() {
    S0:     x = new int[N];
    S1:     y = new int[N];
    S2:     z = new int[N];
    S3:     ... = y[i];
    S4:     z[i] = ...;
    S5:     ... = x[i];
    S6:     y[i] = ...;
}

void m2(int[] tmp) {
    atomic {
        S7:       ... = tmp[i];
        S8:       x[i] = ...;
    }
}
```

<table>
<thead>
<tr>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td>S5</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td>S3, S6</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td>S4</td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td>S3, S5</td>
</tr>
</tbody>
</table>

After m1

<table>
<thead>
<tr>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>S5</td>
<td>S0</td>
</tr>
<tr>
<td>A.y</td>
<td>S3, S6</td>
<td>S1</td>
</tr>
<tr>
<td>A.z</td>
<td>S4</td>
<td>S2</td>
</tr>
<tr>
<td>int[]</td>
<td>S3, S5</td>
<td>S4, S6</td>
</tr>
</tbody>
</table>

After m2
### D-NAIT example: Compiling

```java
void m1() {
  void m2(int[] tmp) {
    atomic {
      S7:   ... = tmp[i];
      S8:   x[i] = ...;
    }
  }

  S0:   x = new int[N];
  S1:   y = new int[N];
  S2:   z = new int[N];
  S3:   ... = y[i];
  S4:   z[i] = ...;
  S5:   ... = x[i];
  S6:   y[i] = ...;
}
```

<table>
<thead>
<tr>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td>S5</td>
<td>S0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td>S3, S6</td>
<td>S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td>S4</td>
<td>S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td>S3, S5</td>
<td>S4, S6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
D-NAIT example: Compiling

void m1() {
  S0: x = new int[N];
  S1: y = new int[N];
  S2: z = new int[N];
  S3: ... = y[i];
  S4: z[i] = ...
  S5: ... = x[i];
  S6: y[i] = ...
}

void m2(int[] tmp) {
  atomic {
    S7: ... = tmp[i];
    S8: x[i] = ...
  }
}

<table>
<thead>
<tr>
<th></th>
<th>After m1</th>
<th>After m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAIT-state</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reads-to-be-patched</td>
<td>Writes-to-be-patched</td>
</tr>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td>S5</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td>S3, S6</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td>S4</td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td>S3, S5</td>
</tr>
</tbody>
</table>
**D-NAIT example: Compiling**

```java
void m1() {
    S0:     x = new int[N];
    S1:     y = new int[N];
    S2:     z = new int[N];
    S3:     ... = y[i];
    S4:     z[i] = ...
               patch
    S5:     ... = x[i];
    S6:     y[i] = ...
               patch
}
```

```java
void m2(int[] tmp) {
    atomic {
        S7:       ... = tmp[i];
        S8:       x[i] = ...
               }
    }
```

<table>
<thead>
<tr>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td>S5</td>
<td>S0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td>S3, S6</td>
<td>S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td>S4</td>
<td>S2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| int[]       | TxNone              | S3, S5               | S4, S6      | TxRead              | S3, S5               | {} }
D-NAIT example: Compiling

void m1() {
  S0:     x = new int[N];  \(\text{\textcolor{green}{patch}}\)
  S1:     y = new int[N];
  S2:     z = new int[N];
  S3:     ... = y[i];
  S4:     z[i] = ...  \(\text{\textcolor{green}{patch}}\)
  S5:     ... = x[i];
  S6:     y[i] = ...;
}

void m2(int[] tmp) {
  atomic {
    S7:       ... = tmp[i];
    S8:       x[i] = ...;
  }
}

| A.x | TxNone | S5 | S0 | TxRead | S5 | {} |
| A.y | TxNone | S3, S6 | S1 |        |    |    |
| A.z | TxNone | S4 | S2 |        |    |    |
| int[] | TxNone | S3, S5 | S4, S6 | TxRead | S3, S5 | {} |
D–NAIT example: Compiling

- void m1() {
  S0: x = new int[N];  // patch
  S1: y = new int[N];
  S2: z = new int[N];
  S3: ... = y[i];
  S4: z[i] = ...  // patch
  S5: ... = x[i];
  S6: y[i] = ...;
}

- void m2(int[] tmp) {
  atomic {
    S7: ... = tmp[i];
    S8: x[i] = ...;
  }
}

<table>
<thead>
<tr>
<th>After m1</th>
<th>After m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAIT-state</td>
<td>DNAIT-state</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Reads-to-be-patched</td>
<td>Reads-to-be-patched</td>
</tr>
<tr>
<td>Writes-to-be-patched</td>
<td>Writes-to-be-patched</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>A.x</td>
<td>TxNone</td>
</tr>
<tr>
<td></td>
<td>S5</td>
</tr>
<tr>
<td></td>
<td>S0</td>
</tr>
<tr>
<td></td>
<td>TxRead</td>
</tr>
<tr>
<td></td>
<td>S5</td>
</tr>
<tr>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
</tr>
<tr>
<td></td>
<td>S3, S6</td>
</tr>
<tr>
<td></td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>TxRead</td>
</tr>
<tr>
<td></td>
<td>S5</td>
</tr>
<tr>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
</tr>
<tr>
<td></td>
<td>S4</td>
</tr>
<tr>
<td></td>
<td>S2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
</tr>
<tr>
<td></td>
<td>S3, S5</td>
</tr>
<tr>
<td></td>
<td>S4, S6</td>
</tr>
<tr>
<td></td>
<td>TxAll</td>
</tr>
<tr>
<td></td>
<td>S3, S5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
D-NAIT example: Compiling

```java
void m1() {
    x = new int[N];  \textcolor{red}{\textit{patch}}
    y = new int[N];
    z = new int[N];
    ... = y[i];  \textcolor{red}{\textit{patch}}
    z[i] = ...;  \textcolor{red}{\textit{patch}}
    ... = x[i];  \textcolor{red}{\textit{patch}}
    y[i] = ...;  \textcolor{red}{\textit{patch}}
}

void m2(int[] tmp) {
    atomic {
        ... = tmp[i];
        x[i] = ...;
    }
}
```

<table>
<thead>
<tr>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
<th>DNAIT-state</th>
<th>Reads-to-be-patched</th>
<th>Writes-to-be-patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.x</td>
<td>TxNone</td>
<td>S5</td>
<td>S0</td>
<td>TxRead</td>
<td>S5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
<td>S3, S6</td>
<td>S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
<td>S4</td>
<td>S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
<td>S3, S5</td>
<td>S4, S6</td>
<td>TxAll</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{}</td>
</tr>
</tbody>
</table>

Monday, May 27, 13
D–NAIT example: Compiling

```java
    void m1() {
        S0:     x = new int[N];  // patch
        S1:     y = new int[N];
        S2:     z = new int[N];
        S3:     ... = y[i];    // patch
        S4:     z[i] = ...;    // patch
        S5:     ... = x[i];    // patch
        S6:     y[i] = ...;
    }

    void m2(int[] tmp) {
        atomic {
            S7:       ... = tmp[i];
            S8:       x[i] = ...;
        }
    }
```

<table>
<thead>
<tr>
<th>After m1</th>
<th>After m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAIT-state</td>
<td>Reads-to-be-patched</td>
</tr>
<tr>
<td>A.x</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.y</td>
<td>TxNone</td>
</tr>
<tr>
<td>A.z</td>
<td>TxNone</td>
</tr>
<tr>
<td>int[]</td>
<td>TxNone</td>
</tr>
</tbody>
</table>
Context-sensitive D-NAIT

• Problem: Simple type-based D-NAIT often too coarse
  • Common types (e.g., int[]) accessed inside + outside transactions

• Instead of single field/type, record D-NAIT state for tuple (Field, Type)
  • Adding 1 level of heap context

• Refine D-NAIT analysis by adding context info: e.g.
  • String.value → char[]
  • Vector.array → Object[]
Tracking of aliasing

- Dynamically track which field variables may alias
  - “Incremental optimistic alias analysis”
  - Two aliasing states for fields: unique and aliased

**Generate STM barriers specific for unique contexts**
- Change in aliasing state may trigger patching
- Details of algorithm in paper!
Phantom barriers: convertible barriers

- Phantom barriers = reads/writes where STM barriers may be needed in the future
- Phantom barriers may be converted at runtime into real STM barriers by patching
  - Barrier code pre-generated at compile-time
- Try to keep phantom barrier overhead low
  - Leverage existing compiler optimizations like profile guided code positioning, register
Phantom barriers on IA-32

- // Original instruction
- add [base+offset], 1
- <next statement1>
- next_block:
-   <next statement2>
-   ...
-   ret
- barrier_block:
-   <save live registers>
-   <stm lock base>
-   add [base+offset], 1
-   <stm unlock base>
-   <restore live registers>
-   <next statement1>
-   jmp next_block
Phantom barriers on IA-32

// Original instruction
• add [base+offset], 1
• <next statement1>

next_block:
• <next statement2>
• ...
• ret

• barrier_block:
• <save live registers>
• <stm lock base>
• add [base+offset], 1
• <stm unlock base>
• <restore live registers>
• <next statement1>
• jmp next_block

// Patch (5 bytes long)
jmp barrier_block

// Mini-transaction
Phantom barriers on IA-32

// Patch (5 bytes long)
jmp barrier_block

// Original instruction
jmp next_block

• add [base+offset], 1
• <next statement>
• next_block:
• <next statement2>
• ...
• ret

• barrier_block:
• <save live registers>
• <stm lock base>
• add [base+offset], 1
• <stm unlock base>
• <restore live registers>
• <next statement1>
• jmp next_block

Mini-transaction
save/restore live regs
Phantom barriers on IA-32

// Patch (5 bytes long)
jmp barrier_block

// Original instruction
jmp barrier_block

• add [base+offset], 1
• <next statement1>

next_block:
• <next statement2>
• ...
• ret

• barrier_block:
• <save live registers>
• <stm lock base>
• add [base+offset], 1
• <stm unlock base>
• <restore live registers>
• <next statement1>
• jmp next_block

Mini-transaction
save/restore live regs
lock/unlock object
Phantom barriers on IA-32

// Patch (5 bytes long)
jmp barrier_block

// Original instruction
add [base+offset], 1

next_block:

<br>next statement1<br>

...<br>

ret

barrier_block:

<save live registers><br>
<stm lock base><br>
add [base+offset], 1<br>
<stm unlock base><br>
<restore live registers><br>
<br>next statement1<br>

jmp next_block

Mini-transaction
save/restore live regs
lock/unlock object
perform original load/store
Phantom barriers on IA-32

// Patch (5 bytes long)
jmp barrier_block

// Original instruction
jmp barrier_block

• add [base+offset], 1
• <next statement1>
• next_block:
• <next statement2>
• ...
• ret
• barrier_block:
• <save live registers>
• <stm lock base>
• add [base+offset], 1
• <stm unlock base>
• <restore live registers>
• <next statement1>
• jmp next_block

Mini-transaction
save/restore live regs
lock/unlock object
perform original load/store
duplicate following instr.
Phantom barriers on IA-32

// Patch (5 bytes long)
jmp barrier_block

// Original instruction
add [base+offset], 1
<next statement1>

next_block:
<next statement2>
...
ret

barrier_block:
<save live registers>
<stm lock base>
add [base+offset], 1
<stm unlock base>
<restore live registers>
<next statement1>
jmp next_block

Mini-transaction
save/restore live regs
lock/unlock object
perform original load/store
duplicate following instr.
branch to next block
Runtime patching

1. Compiler determines patch locations
   • If nothing to patch: no further work, jump to step 6
2. Invoke stop-the-world mechanism
   • Signals threads to pause at next safe point
3. Wait until all threads suspended
4. Patch each candidate instruction
   • Installing a jump to the STM barrier code
5. Signal threads to execute a serializing instruction
   • Flush I-cache (CPUID on IA-32)
Non-transactional workloads: JVM98

- compress
- jess
- db
- javac
- mpegaudio
- mtrt
- jack

Overhead over weak atomicity

- Strong (Base)
- Strong (D-NAIT)
Non-transactional workloads: JVM98

Overhead over weak atomicity

- compress
- jess
- db
- javac
- mpegaudio
- mtrt
- jack

Legend:
- Strong (Base)
- Strong (D-NAIT)

Monday, May 27, 13
Non-transactional workloads: JVM98

Overhead over weak atomicity

- compress
- jess
- db
- javac
- mpegaudio
- mtrt
- jack

Strong (Base)
Strong (D-NAIT)
Non-transactional workloads: JVM98

Overhead over weak atomicity

- compress
- jess
- db
- javac
- mpegaudio
- mtrt
- jack

Strong (Base)
Strong (D-NAIT)
Non-transactional workloads: JVM98

Overhead over weak atomicity

- compress
- jess
- db
- javac
- mpegaudio
- mtrt
- jack

Strong (Base)
Strong (D-NAIT)
Non-transactional workloads: JVM98

Overhead over weak atomicity

- compress
- jess
- db
- javac
- mpegaudio
- mtrt
- jack

- Strong (Base)
- Strong (D-NAIT)
Non-transactional workloads: JVM98

Overhead over weak atomicity

- compress
- jess
- db
- javac
- mpegaudio
- mtrt
- jack

Strong (Base)
Strong (D-NAIT)
Non-transactional workloads: JVM98

Overhead over weak atomicity

- compress
- jess
- db
- javac
- mpegaudio
- mtrt
- jack

- Strong (Base)
- Strong (D-NAIT)
Non-transactional workloads: JVM98

Existing optimizations do not always work well (mpegaudio)
Non-transactional workloads: JVM98

Existing optimizations do not always work well (mpegaudio)

D-NAIT: Strong atomicity overhead down to <=8%
Transactional workloads: TSP

![Bar chart showing relative execution time for different numbers of threads and varying thread strengths.

- **Weak**
- **Strong (Base)**
- **Strong (Simple D-NAIT)**
- **Strong (Context D-NAIT)**

The chart depicts the relative execution time for transactional workloads under different thread configurations.
Transactional workloads: TSP

SA overhead almost 3x
Transactional workloads: TSP

### Diagram

**Relative Execution Time**

- **# of threads**: 1, 2, 4, 8
- **Relative Execution Time**: 0, 0.75, 1.5, 2.25, 3.0
- **Weak**
- **Strong (Base)**
- **Strong (Simple D-NAIT)**
- **Strong (Context D-NAIT)**

*SA overhead almost 3x*
**Transactional workloads: TSP**

**SA overhead almost 3x**
Transactional workloads: TSP

SA overhead almost 3x
Transactional workloads: TSP

SA overhead almost 3x
Transactional workloads: TSP

**SA overhead almost 3x**

**D-NAIT effective 1-8 threads (37% avg overhead)**
Transactional workloads: TSP

SA overhead almost 3x

D-NAIT effective 1-8 threads (37% avg overhead)
Transactional workloads: TSP

SA overhead almost 3x

D-NAIT effective 1-8 threads (37% avg overhead)
Transactional workloads: TSP

SA overhead almost 3x

D-NAIT effective 1-8 threads (37% avg overhead)
Transactional workloads: TSP

SA overhead almost 3x

D-NAIT effective 1-8 threads (37% avg overhead)
**Transactional workloads: TSP**

SA overhead almost 3x

D-NAIT effective 1-8 threads (37% avg overhead)

Context helps: SA overhead down to <12%
Summary

- Transactions are an attractive alternative to locks
  - Simpler programming model with stronger guarantees
  - Potentially better scalability with optimistic concurrency
- Lots of research and development still needed
  - Good, understandable performance for general transactions
  - Formalizing semantics
- Learn more: http://en.wikipedia.org/wiki/Software_transactional_memory