Consistency
Transactions
Transactional Memory

Chris Rossbach
Outline for Today

• Questions?

• Administrivia
  • Have you started the next lab yet? 😊

• Agenda
  • Consistency
  • Transactions
  • Transactional Memory

• Acks: Yoav Cohen for some STM slides
Faux Quiz questions

• How are promises and futures related? Since there is disagreement on the nomenclature, don’t worry about which is which—just describe what the different objects are and how they function.
• How does HTM resemble or differ from Load-linked Stored-Conditional?
• What are some pros and cons of HTM vs STM?
• What is Open Nesting? Closed Nesting? Flat Nesting?
• How does 2PL differ from 2PC?
• Define ACID properties: which, if any, of these properties does TM relax?
Memory Consistency
Memory Consistency

• Formal specification of memory semantics
  • Statement of how shared memory will behave with multiple CPUs
  • Ordering of reads and writes
Memory Consistency

• Formal specification of memory semantics
  • Statement of how shared memory will behave with multiple CPUs
  • Ordering of reads and writes

• Memory Consistency != Cache Coherence
  • Coherence: propagate updates to cached copies
    • Invalidate vs. Update
  • Coherence vs. Consistency?
    • **Coherence:** ordering of ops. at a single location
    • **Consistency:** ordering of ops. at multiple locations
Sequential Consistency

• Result of any execution is same as if all operations execute on a uniprocessor.
• Operations on each processor are totally ordered in the sequence and respect program order for each processor.
Sequential Consistency

Any execution of operations on each processor is totally ordered by the sequence and respects the program order for each processor.
Sequential Consistency

• Result of any execution is same as if all operations execute on a uniprocessor
Sequential Consistency

• Result of *any* execution is same as if all operations execute on a uniprocessor
• Operations on each processor are *totally ordered* in the sequence and respect program order for each processor
Sequential Consistency

• Result of *any* execution is same as if all operations execute on a uniprocessor

• Operations on each processor are *totally ordered* in the sequence and respect program order for each processor

Trying to mimic Uniprocessor semantics:
• Memory operations occur:
  • One at a time
  • In program order
• Read returns value of last write
Sequential Consistency

• Result of \textit{any} execution is same as if all operations execute on a uniprocessor

• Operations on each processor are \textit{totally ordered} in the sequence and respect program order for each processor

How is this different from coherence?

Why do modern CPUs not implement SC?

Requirements: \textit{program order, write atomicity}
Sequential Consistency

• All operations are executed in some sequential order
• each process issues operations in program order
  • Any valid interleaving is allowed
  • All agree on the same interleaving
  • Each process preserves its program order

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: W(x)b</td>
</tr>
<tr>
<td>P3: R(x)b  R(x)a</td>
</tr>
<tr>
<td>P4: R(x)b  R(x)a</td>
</tr>
<tr>
<td>(a)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: W(x)b</td>
</tr>
<tr>
<td>P3: R(x)b  R(x)a</td>
</tr>
<tr>
<td>P4: R(x)a  R(x)b</td>
</tr>
<tr>
<td>(b)</td>
</tr>
</tbody>
</table>
Sequential Consistency

- All operations are executed in some sequential order.
- Each process issues operations in program order:
  - Any valid interleaving is allowed.
  - All agree on the same interleaving.
  - Each process preserves its program order.

\[
\begin{array}{c}
\text{P1: } W(x) a \\
\text{P2: } W(x) b \\
\text{P3: } R(x) b \quad R(x) a \\
\text{P4: } R(x) b \quad R(x) a \\
\end{array}
\]

\[
\begin{array}{c}
\text{P1: } W(x) a \\
\text{P2: } W(x) b \\
\text{P3: } R(x) b \quad R(x) a \\
\text{P4: } R(x) a \quad R(x) b \\
\end{array}
\]

(a) Are either of these SC?
Sequential Consistency: Canonical Example

Initially, Flag1 = Flag2 = 0

P1
Flag1 = 1
if (Flag2 == 0)
    enter CS

P2
Flag2 = 1
if (Flag1 == 0)
    enter CS
Sequential Consistency: Canonical Example

Initially, Flag1 = Flag2 = 0

**P1**
Flag1 = 1
if (Flag2 == 0)
   enter CS

**P2**
Flag2 = 1
if (Flag1 == 0)
   enter CS

Can both P1 and P2 wind up in the critical section at the same time?
Do we need Sequential Consistency?

Initially, Flag1 = Flag2 = 0

P1
Flag1 = 1

P2
Flag2 = 1
if(Flag1 == 0)
  data++

if(Flag2 == 0)
data++
Do we need Sequential Consistency?

Initially, Flag1 = Flag2 = 0

P1
Flag1 = 1

P2
Flag2 = 1
if(Flag1 == 0)
data++

if(Flag2 == 0)
data++

Key issue:
• P1 and P2 may not see each other’s writes in the same order
• Implication: both in critical section, which is incorrect
• Why would this happen?
Do we need Sequential Consistency?

Initially, Flag1 = Flag2 = 0

\[ P1 \]
Flag1 = 1
Flag2 = 1
if(Flag1 == 0) data++
if(Flag2 == 0) data++

Key issue:
- P1 and P2 may not see each other’s writes in the same order
- Implication: both in critical section, which is incorrect
- Why would this happen?

P2

Write Buffers
- P.0 write → queue op in write buffer, proceed
- P.0 read → look in write buffer,
- P.(x != 0) read → old value: write buffer hasn’t drained
Requirements for Sequential Consistency

• Program Order
  Processor's memory operations must complete in program order

• Write Atomicity
  Writes to the same location seen by all other CPUs
  Subsequent reads must not return value of a write until propagated to all
  Write acknowledgements are necessary
  Cache coherence provides these properties for a cache
Requirements for Sequential Consistency

• **Program Order**
  • Processor’s memory operations must complete in program order
Requirements for Sequential Consistency

• **Program Order**
  • Processor’s memory operations must complete in program order

• **Write Atomicity**
  • Writes to the same location seen by all other CPUs
  • Subsequent reads must not return value of a write until propagated to all
Requirements for Sequential Consistency

• **Program Order**
  • Processor’s memory operations must complete in program order

• **Write Atomicity**
  • Writes to the same location seen by all other CPUs
  • Subsequent reads must not return value of a write until propagated to all

• Write acknowledgements are necessary
  • Cache coherence provides these properties for a cache-only system
Requirements for Sequential Consistency

• **Program Order**
  • Processor’s memory operations must complete in program order

• **Write Atomicity**
  • Writes to the same location seen by all other CPUs
  • Subsequent reads must not return value of a write until propagated to all

• Write acknowledgements are necessary
  • Cache coherence provides these properties for a cache-only system

Disadvantages:
• Difficult to implement!
  • Coherence to (e.g.) write buffers is hard

• Sacrifices many potential optimizations
  • Hardware (cache) and software (compiler)
  • Major performance hit
Why Relax Consistency?

• Motivation, originally
  • Allow in-order processors to overlap store latency with other work
  • “Other work” depends on loads, so loads bypass stores using a store queue
• PC (processor consistency), SPARC TSO, IBM/370
  • Just relax read-to-write program order requirement
• Subsequently
  • Hide latency of one store with latency of other stores
  • Stores to be performed OOO with respect to each other
  • Breaks SC even further
• This led to definition of SPARC PSO/RMO, WO, PowerPC WC, Itanium
• What’s the problem with relaxed consistency?
  • Shared memory programs can break if not written for specific cons. model
Relaxed Consistency Models

• Program Order relaxations (different locations)
  • $W \rightarrow R; \ W \rightarrow W; \ R \rightarrow R/W$

• Write Atomicity relaxations
  • Read returns another processor's Write early

• Requirement: synchronization primitives for safety
  • Fence, barrier instructions, etc.
Relaxed Consistency Models

- **Program Order** relaxations  *(different locations)*
  - $W \rightarrow R$;  $W \rightarrow W$;  $R \rightarrow R/W$
Relaxed Consistency Models

• **Program Order** relaxations  *(different locations)*
  - $W \rightarrow R$;  $W \rightarrow W$;  $R \rightarrow R/W$

• **Write Atomicity** relaxations
  - Read returns another processor’s Write early
Relaxed Consistency Models

- **Program Order** relaxations  *(different locations)*
  - $W \rightarrow R$;  $W \rightarrow W$;  $R \rightarrow R/W$

- **Write Atomicity** relaxations
  - Read returns another processor’s Write early

- **Requirement**: synchronization primitives for safety
  - Fence, barrier instructions etc
Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - \( W \rightarrow R; \quad W \rightarrow W; \quad R \rightarrow R/W \)

- **Write Atomicity** relaxations
  - Read returns another processor’s Write early

- **Requirement**: synchronization primitives for safety
  - Fence, barrier instructions etc

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>( W \rightarrow R ) Order</th>
<th>( W \rightarrow W ) Order</th>
<th>( R \rightarrow RW ) Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety net</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC [16]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>serialization instructions</td>
</tr>
<tr>
<td>IBM 370 [14]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>TSO [20]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PC [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PSO [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WO [5]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>synchronization</td>
</tr>
<tr>
<td>RCsc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>RCpc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>Alpha [19]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MB, WMB</td>
</tr>
<tr>
<td>RMO [21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>various MEMBAR’s</td>
</tr>
<tr>
<td>PowerPC [17, 4]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SYNC</td>
</tr>
</tbody>
</table>
Relaxed Consistency

- **Program Order** relaxations (different locations)
  - \( W \rightarrow R; \quad W \rightarrow W; \quad R \rightarrow R/W \)

- **Write Atomicity** relaxations
  - Read returns another processor’s Write early

- **Requirement:** synchronization primitives for safety
  - Fence, barrier instructions etc

### Figure 7: Relaxations allowed by memory models.

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>( W \rightarrow R )</th>
<th>( W \rightarrow W )</th>
<th>( R \rightarrow R/W )</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety net</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC [16]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 370 [14]</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSO [20]</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC [13, 12]</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSO [20]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WO [5]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCsc [13, 12]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCpc [13, 12]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha [19]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMO [21]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC [17, 4]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example Commercial Systems Providing the Relaxation:
- AlphaServer 8200/8400, Cray T3D, Sequent Balance, SparcCenter1000/2000
- AlphaServer 8200/8400, Cray T3D
- AlphaServer 8200/8400, Cray T3D

### Figure 8: Simple categorization of relaxed models. A √ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The “Read Own Write Early” relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The “Read Others’ Write Early” relaxation is possible and detectable with complex implementations of RCsc.

### Figure 9: Some commercial systems that relax sequential consistency.

*Note: The code snippet is not relevant to the discussion and is included for context.*

```c
static inline void arch_write_lock(arch_rwlock_t *rw) {
    asm volatile("LOCK Prefix WRITE LOCK SUB(\%1) "(\%0)\n\t"
            "jz 1f\n"
            "call _write_lock_failed\n\t"
            "1:\n"
            "::LOCK PTR REG (\&rw->write), "i" (RW LOCK BIAS) : "memory"); }
```
Relaxed Consistency Models

- **Program Order** relaxations  
  (different locations)
  - $W \rightarrow R$;  
  - $W \rightarrow W$;  
  - $R \rightarrow R/W$

- **Write Atomicity** relaxations
  - Read returns another processor’s Write early

- **Requirement**: synchronization primitives for safety
  - Fence, barrier instructions etc

---

### Table

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>W → R Order</th>
<th>W → W Order</th>
<th>R → RW Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety net</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC [16]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM 370 [14]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>serialization instructions</td>
</tr>
<tr>
<td>TSO [20]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>RMW</td>
</tr>
<tr>
<td>PC [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PSO [20]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WO [5]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>synchronization</td>
</tr>
<tr>
<td>RCsc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>RCpc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>Alpha [19]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MB, WMB</td>
</tr>
<tr>
<td>RMO [21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>various MEmBAR’s</td>
</tr>
<tr>
<td>PowerPC [17, 4]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SYNC</td>
</tr>
</tbody>
</table>
Relaxed Consistency Models

- **Program Order** relaxations  (different locations)
  - W → R;  W → W;  R → R/W

static inline unsigned long
__arch_spin_trylock(arch_spinlock_t *lock)
{
    unsigned long tmp, token;
    token = LOCK_TOKEN;
    __asm__ __volatile__(
        "1: " PPC_LWARX(%0,0,%2,1) "\n"
        cmpwi 0,%0,0\n"
        bne- 2f\n"
        stwcx. %1,0,%2\n"
        bne- 1b\n"
        PPC_ACQUIRE_BARRIER
        "2:" : "=&r" (tmp)
        : "r" (token), "r" (&lock->slock)
        : "cr0", "memory";
    return tmp;
}
Relaxed Consistency Models

- **Program Order** relaxations  *(different locations)*
  - $W \rightarrow R$;  $W \rightarrow W$;  $R \rightarrow R/W$

- **Write Atomicity** relaxations
  - Read returns another processor’s Write early

- **Requirement**: synchronization primitives for safety
  - Fence, barrier instructions etc

---

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>$W \rightarrow R$ Order</th>
<th>$W \rightarrow W$ Order</th>
<th>$R \rightarrow RW$ Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety net</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC [16]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>serialization instructions</td>
</tr>
<tr>
<td>IBM 370 [14]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>TSO [20]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PC [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PSO [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WO [5]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>synchronization</td>
</tr>
<tr>
<td>RCsc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>RCpc [13, 12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>Alpha [19]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MB, WMB</td>
</tr>
<tr>
<td>RMO [21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>various MEMBAR’s</td>
</tr>
<tr>
<td>PowerPC [17, 4]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SYNC</td>
</tr>
</tbody>
</table>
Some Key Consistency Models

**TSO**
- x86
- Stores are totally ordered, reads not
- Differs from PC by allowing early reads of processor’s own writes

**RC: Release Consistency**
- Key insight: only synchronization references need to be ordered
- Hence, relax memory for all other references
  - Enable high-performance OOO implementation
- Programmer **labels** synchronization references
  - Hardware must carefully order these labeled references
- Labeling schemes:
  - Explicit synchronization ops (acquire/release)
  - Memory fence or memory barrier ops:
    - All preceding ops must finish before following ones begin
- Fence ops drain pipeline
Transactions and Transactional Memory
Transactions and Transactional Memory

• 3 Programming Model Dimensions:
Transactions and Transactional Memory

• 3 Programming Model Dimensions:
  • How to specify computation
Transactions and Transactional Memory

• 3 Programming Model Dimensions:
  • How to specify computation
  • How to specify communication
Transactions and Transactional Memory

• 3 Programming Model Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer
Transactions and Transactional Memory

• 3 Programming Model Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer
Transactions and Transactional Memory

• 3 Programming Model Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Threads, Futures, Events etc.
  • *Mostly about how to express control*
Transactions and Transactional Memory

• 3 Programming Model Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Threads, Futures, Events etc.
  • *Mostly about how to express control*

• Transactions
  • *Mostly about how to deal with shared state*
Transactions

*Core issue: multiple updates*

**Canonical examples:**

```
move(file, old-dir, new-dir) {
  delete(file, old-dir)
  add(file, new-dir)
}
```

```
create(file, dir) {
  alloc-disk(file, header, data)
  write(header)
  add(file, dir)
}
```
Transactions

Core issue: multiple updates

Canonical examples:

```c
move(file, old-dir, new-dir) {
    delete(file, old-dir)
    add(file, new-dir)
}
```

```c
create(file, dir) {
    alloc-disk(file, header, data)
    write(header)
    add(file, dir)
}
```

- Modified data in memory/caches
- Even if in-memory data is durable, multiple disk updates
Transactions

Core issue: multiple updates

Canonical examples:

move(file, old-dir, new-dir) {
    delete(file, old-dir)
    add(file, new-dir)
}

create(file, dir) {
    alloc-disk(file, header, data)
    write(header)
    add(file, dir)
}

Problems: crash in the middle / visibility of intermediate state

• Modified data in memory/caches
• Even if in-memory data is durable, multiple disk updates
Problem: Unreliability

• Want reliable update of two resources (e.g. in two disks, machines…)
• Move file from A to B
• Create file (update free list, inode, data block)
• Bank transfer (move $100 from my account to VISA account)
• Move directory from server A to B

• Machines can crash, messages can be lost
Problem: Unreliability

• Want reliable update of two resources (e.g. in two disks, machines...)
  • Move file from A to B
  • Create file (update free list, inode, data block)
  • Bank transfer (move $100 from my account to VISA account)
  • Move directory from server A to B
Problem: Unreliability

• Want reliable update of two resources (e.g. in two disks, machines...)
  • Move file from A to B
  • Create file (update free list, inode, data block)
  • Bank transfer (move $100 from my account to VISA account)
  • Move directory from server A to B

• Machines can crash, messages can be lost
Problem: Unreliability

• Want reliable update of two resources (e.g. in two disks, machines...)
  • Move file from A to B
  • Create file (update free list, inode, data block)
  • Bank transfer (move $100 from my account to VISA account)
  • Move directory from server A to B
• Machines can crash, messages can be lost

Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?
Problem: Unreliability

• Want reliable update of two resources (e.g. in two disks, machines...)
  • Move file from A to B
  • Create file (update free list, inode, data block)
  • Bank transfer (move $100 from my account to VISA account)
  • Move directory from server A to B

• Machines can crash, messages can be lost

Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?

No.
Not even if all messages get through!
General’s paradox
General’s paradox

• Two generals on separate mountains
General’s paradox

• Two generals on separate mountains
• Can only communicate via messengers
General’s paradox

• Two generals on separate mountains
• Can only communicate via messengers
• Messengers can get lost or captured
General’s paradox

• Two generals on separate mountains
• Can only communicate via messengers
• Messengers can get lost or captured
• Need to coordinate attack
  • attack at same time good, different times bad!
General’s paradox

• Two generals on separate mountains
• Can only communicate via messengers
• Messengers can get lost or captured
• Need to coordinate attack
  • attack at same time good, different times bad!
General’s paradox

• Two generals on separate mountains
• Can only communicate via messengers
• Messengers can get lost or captured
• Need to coordinate attack
  • attack at same time good, different times bad!
General’s paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

General A → General B: let’s attack at dawn
General’s paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

General A ➔ General B: let’s attack at dawn
General B ➔ General A: OK, dawn.
General’s paradox

• Two generals on separate mountains
• Can only communicate via messengers
• Messengers can get lost or captured
• Need to coordinate attack
  • attack at same time good, different times bad!

General A $\rightarrow$ General B: let’s attack at dawn
General B $\rightarrow$ General A: OK, dawn.
General A $\rightarrow$ General B: Check. Dawn it is.
General’s paradox

• Two generals on separate mountains
• Can only communicate via messengers
• Messengers can get lost or captured
• Need to coordinate attack
  • attack at same time good, different times bad!

General A → General B: let’s attack at dawn
General B → General A: OK, dawn.
General A → General B: Check. Dawn it is.
General B → General A: Alright already—dawn.
General’s paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

General A ➔ General B: let’s attack at dawn
General B ➔ General A: OK, dawn.
General A ➔ General B: Check. Dawn it is.
General B ➔ General A: Alright already—dawn.

- Even if all messages delivered, can’t assume—maybe some message didn’t get through.
- No solution: one of the few CS impossibility results.
Transactions can help

(but can’t solve it)
Transactions can help
(*but can’t solve it*)

• Solves weaker problem:
  • 2 things will either happen or not
  • not necessarily at the same time
Transactions can help
*(but can’t solve it)*

- Solves weaker problem:
  - 2 things will either happen or not
  - not necessarily at the same time
- Core idea: one entity has the power to say yes or no for all
  - Local txn: one final update (TxEND) irrevocably triggers several
  - Distributed transactions
    - 2 phase commit
    - One machine has final say for all machines
    - Other machines bound to comply
Transactions can help (but can’t solve it)

• Solves weaker problem:
  • 2 things will either happen or not
  • not necessarily at the same time

• Core idea: one entity has the power to say yes or no for all
  • Local txn: one final update (TxEND) irrevocably triggers several
  • Distributed transactions
    • 2 phase commit
    • One machine has final say for all machines
    • Other machines bound to comply

What is the role of synchronization here?
begin transaction;
   x = read("x-values", ....);
   y = read("y-values", ....);
   z = x+y;
   write("z-values", z, ....);
commit transaction;
Transactional Programming Model

begin transaction;
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
commit transaction;

What has changed from previous programming models?
ACID Semantics

• Atomic – all updates happen or none do
• Consistent – system invariants maintained across updates
• Isolated – no visibility into partial updates
• Durable – once done, stays done

• Are subsets ever appropriate?
• When would ACI be useful?
• ACD?
• Isolation only?
ACID Semantics

What are they?

- A
- C
- I
- D
ACID Semantics

- Atomic – all updates happen or none do
- Consistent – system invariants maintained across updates
- Isolated – no visibility into partial updates
- Durable – once done, stays done

Are subsets ever appropriate?

When would ACI be useful?

ACD?

Isolation only?
ACID Semantics

• Atomic – all updates happen or none do
• Consistent – system invariants maintained across updates
• Isolated – no visibility into partial updates
• Durable – once done, stays done

Are subsets ever appropriate?

When would ACI be useful?

ACD?

Isolation only?

begin transaction;
  x = read("x-values", ....);
  y = read("y-values", ....);
  z = x+y;
  write("z-values", z, ....);
commit transaction;
ACID Semantics

- Atomic – all updates happen or none do
- Consistent – system invariants maintained across updates
- Isolated – no visibility into partial updates
- Durable – once done, stays done

Are subsets ever appropriate?

When would ACI be useful?

ACD?

Isolation only?

```
begin transaction;
  x = read("x-values", ....);
  y = read("y-values", ....);
  z = x+y;
  write("z-values", z, ....);
commit transaction;
```
ACID Semantics

• Atomic – all updates happen or none do
• Consistent – system invariants maintained across updates
• Isolated – no visibility into partial updates
• Durable – once done, stays done

Are subsets ever appropriate?
When would ACI be useful?
ACD?
Isolation only?

begin transaction;
  x = read(“x-values”, ....);
  y = read(“y-values”, ....);
  z = x+y;
  write(“z-values”, z, ....);
commit transaction;
ACID Semantics

• Atomic – all updates happen or none do
• Consistent – system invariants maintained across updates
• Isolated – no visibility into partial updates

begin transaction;
  x = read(“x-values”, ....);
  y = read(“y-values”, ....);
  z = x+y;
  write(“z-values”, z, ....);
commit transaction;
ACID Semantics

- Atomic – all updates happen or none do
- Consistent – system invariants maintained across updates
- Isolated – no visibility into partial updates
- Durable – once done, stays done

begin transaction;

\[
\begin{align*}
\text{x} &= \text{read}(“x-values”, 
\ldots); \\
\text{y} &= \text{read}(“y-values”, 
\ldots); \\
\text{z} &= \text{x} + \text{y}; \\
\text{write}(“z-values”, \text{z}, 
\ldots); \\
\text{commit transaction;}
\end{align*}
\]
ACID Semantics

• Atomic – all updates happen or none do
• Consistent – system invariants maintained across updates
• Isolated – no visibility into partial updates
• Durable – once done, stays done
• Are subsets ever appropriate?
  • When would ACI be useful?
  • ACD?
  • Isolation only?

begin transaction;
  x = read(“x-values”, ....);
  y = read(“y-values”, ....);
  z = x+y;
  write(“z-values”, z, ....);
commit transaction;
Transactions: Implementation

• Key idea: turn multiple updates into a single one

• Many implementation Techniques
  • Two-phase locking
  • Timestamp ordering
  • Optimistic Concurrency Control
  • Journaling
  • 2,3-phase commit
  • Speculation-rollback
  • Single global lock
  • Compensating transactions
Transactions: Implementation

• Key idea: turn multiple updates into a single one
Transactions: Implementation

- Key idea: turn multiple updates into a single one
- Many implementation Techniques
  - Two-phase locking
  - Timestamp ordering
  - Optimistic Concurrency Control
  - Journaling
  - 2,3-phase commit
  - Speculation-rollback
  - Single global lock
  - Compensating transactions
Transactions: Implementation

• Key idea: turn multiple updates into a single one
• Many implementation Techniques
  • Two-phase locking
  • Timestamp ordering
  • Optimistic Concurrency Control
  • Journaling
  • 2,3-phase commit
  • Speculation-rollback
  • Single global lock
  • Compensating transactions

Key problems:
• output commit
• synchronization
Transactions: Implementation

• Key idea: turn multiple updates into a single one
• Many implementation Techniques
  • Two-phase locking
  • Timestamp ordering
  • Optimistic Concurrency Control
  • Journaling
  • 2,3-phase commit
  • Speculation-rollback
  • Single global lock
  • Compensating transactions

Key problems:
• output commit
• synchronization
Implementing Transactions

BEGIN_TXN();
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
COMMIT_TXN();
Implementing Transactions

BEGIN_TXN();
   x = read("x-values", ....);
   y = read("y-values", ....);
   z = x+y;
   write("z-values", z, ....);
COMMIT_TXN();
Implementing Transactions

BEGIN_TXN();

x = read("x-values", ....);

y = read("y-values", ....);

z = x+y;

write("z-values", z, ....);

COMMIT_TXN();
Implementing Transactions

BEGIN_TXN();

    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x + y;
    write("z-values", z, ....);

COMMIT_TXN();

BEGIN_TXN() {
    LOCK(single-global-lock);
}

COMMIT_TXN() {
    UNLOCK(single-global-lock);
}

Pros/Cons?
Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```
Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```
Two-phase locking

• Phase 1: only acquire locks in order
• Phase 2: unlock at commit
• avoids deadlock

BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();

BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
  LOCK(x);
}

COMMIT_TXN() {
  forall x in rwset
  UNLOCK(x);
}
Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();

BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
    LOCK(x);
}

COMMIT_TXN() {
  forall x in rwset
    UNLOCK(x);
}

Pros/Cons?
Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```

```
BEGIN_TXN() {
    rwset = Union(rset, wset);
    rwset = sort(rwset);
    forall x in rwset
        LOCK(x);
}

COMMIT_TXN() {
    forall x in rwset
        UNLOCK(x);
}
```

Pros/Cons?
What happens on failures?
Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();

BEGIN_TXN() {
    rwset = Union(rset, wset);
    rwset = sort(rwset);
    forall x in rwset
        LOCK(x);
}

COMMIT_TXN() {
    forall x in rwset
        UNLOCK(x);
}

Pros/Cons?
A: grab locks
A: modify x, y,
A: unlock y, x
B: grab locks
B: update x, y
B: unlock y, x
B: COMMIT
A: CRASH

What happens on failures?
Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- Avoids deadlock

BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();

B commits changes that depend on A’s updates

BEGIN_TXN()
{  
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
  LOCK(x);
}

COMMIT_TXN()
{  
  forall x in rwset
  UNLOCK(x);
}

Pros/Cons?
A: grab locks
A: modify x, y,
A: unlock y, x
B: grab locks
B: update x, y
B: unlock y, x
B: COMMIT
A: CRASH

What happens on failures?
B commits changes that depend on A’s updates

Pros/Cons?
A: grab locks
A: modify x, y,
A: unlock y, x
B: grab locks
B: update x, y
B: unlock y, x
B: COMMIT
A: CRASH
Two-phase commit

• N participants agree or don’t (atomicity)
• Phase 1: everyone “prepares”
• Phase 2: Master decides and tells everyone to actually commit
• What if the master crashes in the middle?
2PC: Phase 1

1. Coordinator sends REQUEST to all participants
2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Example—move: C → S1: delete foo from /, C → S2: add foo to /

Failure case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 decides permission problem
S2 writes/sends VOTE_ABORT

Success case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 writes add foo to /
S2 writes/sends VOTE_COMMIT
2PC: Phase 2

• Case 1: receive VOTE_ABORT or timeout
  • Write GLOBAL_ABORT to log
  • send GLOBAL_ABORT to participants

• Case 2: receive VOTE_COMMIT from all
  • Write GLOBAL_COMMIT to log
  • send GLOBAL_COMMIT to participants

• Participants receive decision, write GLOBAL_* to log
2PC corner cases

Phase 1
1. Coordinator sends REQUEST to all participants
2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Phase 2
- Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Participants recv decision, write GLOBAL_* to log

• What if participant crashes at X?
• Coordinator crashes at Y?
• Participant crashes at Z?
• Coordinator crashes at W?
2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!
- Can participants ask each other what happened?
- 2PC: always has risk of indefinite blocking
- Solution: (yes) 3 phase commit!
- Reliable replacement of crashed "leader"
- 2PC often good enough in practice
2PC limitation(s)

• Coordinator crashes at W, never wakes up
2PC limitation(s)

- Coordinator crashes at $W$, never wakes up
- All nodes block forever!

Solution: (yes) 3 phase commit!

Reliable replacement of crashed "leader"

2PC often good enough in practice
2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!
- Can participants ask each other what happened?
2PC limitation(s)

• Coordinator crashes at W, never wakes up
• All nodes block forever!
• Can participants ask each other what happened?
• 2PC: always has risk of indefinite blocking

Solution: (yes) 3 phase commit!

Reliable replacement of crashed “leader”

2PC often good enough in practice
2PC limitation(s)

• Coordinator crashes at W, never wakes up
• All nodes block forever!
• Can participants ask each other what happened?
• 2PC: always has risk of indefinite blocking
• Solution: (yes) 3 phase commit!
  • Reliable replacement of crashed “leader”
  • 2PC often good enough in practice
Nested Transactions

• Composition of transactions
  • E.g. interact with multiple organizations, each supporting txns
  • Travel agency: canonical example
• Nesting: view transaction as collection of:
  • actions on unprotected objects
  • protected actions that may be undone or redone
  • real actions that may be deferred but not undone
  • nested transactions that may be undone

• Open Nesting details:
  • Nested transaction returns name and parameters of compensating transaction
  • Parent includes compensating transaction in log of parent transaction
  • Invoke compensating transactions from log if parent transaction aborted

• Consistent, atomic, durable, but not isolated
Nested Transactions

• Composition of transactions
  • E.g. interact with multiple organizations, each supporting txns
  • Travel agency: canonical example
Nested Transactions

• Composition of transactions
  • E.g. interact with multiple organizations, each supporting txns
  • Travel agency: canonical example

• Nesting: view transaction as collection of:
  • actions on unprotected objects
  • protected actions that may be undone or redone
  • real actions that may be deferred but not undone
  • nested transactions that may be undone

• Open Nesting details:
  • Nested transaction returns name and parameters of compensating transaction
  • Parent includes compensating transaction in log of parent transaction
  • Invoke compensating transactions from log if parent transaction aborted

• Consistent, atomic, durable, but not isolated
Nested Transactions

• Composition of transactions
  • E.g. interact with multiple organizations, each supporting txns
  • Travel agency: canonical example

• Nesting: view transaction as collection of:
  • actions on unprotected objects
  • protected actions that may be undone or redone
  • real actions that may be deferred but not undone
  • nested transactions that may be undone

3 basic flavors:
* Flat: subsume inner transactions
* Closed: subsume w partial rollback
* Open: pause transactional context
Nested Transactions

• Composition of transactions
  • E.g. interact with multiple organizations, each supporting txns
  • Travel agency: canonical example

• Nesting: view transaction as collection of:
  • actions on unprotected objects
  • protected actions that may be undone or redone
  • real actions that may be deferred but not undone
  • nested transactions that may be undone

• Open Nesting details:
  • Nested transaction returns name and parameters of compensating transaction
  • Parent includes compensating transaction in log of parent transaction
  • Invoke compensating transactions from log if parent transaction aborted
  • Consistent, atomic, durable, but not isolated

3 basic flavors:
* Flat: subsume inner transactions
* Closed: subsume w partial rollback
* Open: pause transactional context
Nesting Semantics Exercise

1 BeginTX()
2 \( X = \text{read}(x) \)
3 \( Y = \text{read}(y) \)
4 \( \text{write}(x, X+1+Y) \)
5 BeginTX()
6 \( Z = \text{read}(z)+X+Y \)
7 \( \text{write}(z) \leftarrow \text{abort} \)
8 EndTX()
9 EndTX()

What if TX aborts btw 7,8
- Under flat nesting?
- Under closed nesting?
- Under open nesting?
Transactional Memory: ACI

Transactional Memory:
- Make multiple memory accesses atomic
- All or nothing – Atomicity
- No interference – Isolation
- Correctness – Consistency
- No durability, for obvious reasons

Keywords:
- Commit, Abort,
- Speculative access, Checkpoint
Transactional Memory: ACI

Transactional Memory:
• Make multiple memory accesses atomic
• All or nothing – Atomicity
• No interference – Isolation
• Correctness – Consistency
• No durability, for obvious reasons

Keywords:
Commit, Abort,
Speculative access, Checkpoint

```java
remove(list, x) {
    lock(list);
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    unlock(list);
}
```
Transactional Memory: ACI

Transactional Memory:
• Make multiple memory accesses atomic
• All or nothing – Atomicity
• No interference – Isolation
• Correctness – Consistency
• No durability, for obvious reasons

Keywords:
   Commit, Abort,
   Speculative access, Checkpoint

```
remove(list, x) {
    lock(list);
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    unlock(list);
}
```

```
remove(list, x) {
    TXBEGIN();
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    TXEND();
}
```
The Real Goal

```
remove(list, x) {
    lock(list);
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    unlock(list);
}
```

```
remove(list, x) {
    TXBEGIN();
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    TXEND();
}
```
The **Real** Goal

```c
remove(list, x) {
    lock(list);
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    unlock(list);
}
```

```c
remove(list, x) {
    TXBEGIN();
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    TXEND();
}
```
The **Real** Goal

```c
remove(list, x) {
    atomic {
        pos = find(list, x);
        if(pos)
            erase(list, pos);
    }
}
```
The **Real** Goal

```c
remove(list, x) {
    atomic {
        pos = find(list, x);
        if(pos)
            erase(list, pos);
    }
}
```

- Transactions: super-awesome
- Transactional Memory: also super-awesome, **but**: Transactions != TM
- TM is an *implementation technique*
- Often presented as programmer abstraction
- Remember Optimistic Concurrency Control
A Simple TM

```c
pthread_mutex_t g_global_lock;

begin_tx() {
    pthread_mutex_lock(g_global_lock);
}

end_tx() {
    pthread_mutex_unlock(g_global_lock);
}

abort() {
    // can't happen
}
```
A Simple TM

```c
pthread_mutex_t g_global_lock;

begin_tx() {
    pthread_mutex_lock(g_global_lock);
}

end_tx() {
    pthread_mutex_unlock(g_global_lock);
}

abort() {
    // can't happen
}

remove(list, x) {
    begin_tx();
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    end_tx();
}
```
A Simple TM

remove(list, x) {
    begin_tx();
    pos = find(list, x);
    if(pos)
        erase(list, pos);
    end_tx();
}

pthread_mutex_t g_global_lock;

begin_tx() {
    pthread_mutex_lock(g_global_lock);
}

end_tx() {
    pthread_mutex_unlock(g_global_lock);
}

abort() {
    // can't happen
}

Actually, this works fine... But how can we improve it?
Concurrency Control Revisited
Concurrency Control Revisited

Consider a hash-table
Concurrency Control Revisited

Consider a hash-table
Concurrency Control Revisited

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ht.add( );</td>
<td>ht.add( );</td>
</tr>
<tr>
<td>if(ht.contains( ));</td>
<td>if(ht.contains( ));</td>
</tr>
<tr>
<td>ht.del( );</td>
<td>ht.del( );</td>
</tr>
</tbody>
</table>

```
ht.add();
if(ht.contains())
    ht.del();
```
Concurrency Control Revisited

```
thread T1
ht.add();
if(ht.contains())
    ht.del();

thread T2
ht.add();
if(ht.contains())
    ht.del();
```
Concurrency Control Revisited

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ht.lock()</td>
<td>ht.lock();</td>
</tr>
<tr>
<td>ht.add(□)</td>
<td>ht.add(□);</td>
</tr>
<tr>
<td>if(ht.contains(□))</td>
<td>if(ht.contains(□))</td>
</tr>
<tr>
<td>ht.del(□);</td>
<td>ht.del(□);</td>
</tr>
<tr>
<td>ht.unlock();</td>
<td>ht.unlock();</td>
</tr>
</tbody>
</table>

![Concurrency Diagram](image)
Pessimistic concurrency control

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ht.lock();</code></td>
<td><code>ht.lock();</code></td>
</tr>
<tr>
<td><code>ht.add();</code></td>
<td><code>ht.add();</code></td>
</tr>
<tr>
<td><code>if(ht.contains())</code></td>
<td><code>if(ht.contains())</code></td>
</tr>
<tr>
<td><code>ht.del();</code></td>
<td><code>ht.del();</code></td>
</tr>
<tr>
<td><code>ht.unlock();</code></td>
<td><code>ht.unlock();</code></td>
</tr>
</tbody>
</table>
Pessimistic concurrency control

```
thread T1
ht.lock();
ht.add();
if(ht.contains())
    ht.del();
ht.unlock();

thread T2
ht.lock();
ht.add();
if(ht.contains())
    ht.del();
ht.unlock();
```
Optimistic concurrency control

```
thread T1
ht.lock();
ht.add( );
if(ht.contains( ))
    ht.del( );
ht.unlock();
thread T2
ht.lock();
ht.add( );
if(ht.contains( ))
    ht.del( );
ht.unlock();
```
Optimistic concurrency control

thread T1

ht.add( );
if(ht.contains( ))
    ht.del( );

thread T2

ht.add( );
if(ht.contains( ))
    ht.del( );
Optimistic concurrency control

What do we do when same data is accessed?

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ht.add( )</td>
<td>ht.add( )</td>
</tr>
<tr>
<td>if(ht.contains( ))</td>
<td>if(ht.contains( ))</td>
</tr>
<tr>
<td>ht.del( )</td>
<td>ht.del( )</td>
</tr>
</tbody>
</table>

ht.lock();
ht.add();
if(ht.contains())
    ht.del();
ht.unlock();
Key Ideas:
- Critical sections execute concurrently
- Conflicts are detected dynamically
- If conflict serializability is violated, rollback

Key Abstractions:
- Primitives
  - $xbegin, xend, xabort$
- Conflict
  \[ \emptyset \neq \{W_a\} \cap \{R_b \cup W_b\} \]
- Contention Manager
  - Need flexible policy
TM basics: example

```
xbegin
1: read A
2: read B
3: if(cpu % 2)
   4: write C
   5: else
   6: read C
   7: ...
   8: xend
```

```
xbegin;
1: read A
2: read B
3: if(cpu % 2)
   4: write C
   5: else
   6: read C
   7: ...
   8: xend
```
TM basics: example

CPU 0
PC: 1
Working Set
R{}
W{}

0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend

CPU 1
PC: 0
Working Set
R{}
W{}

0: xbegin;
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend
TM basics: example

```plaintext
0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend
```

CPU 0

PC: 1
Working Set
R{}
W{}

CPU 1

PC: 1
Working Set
R{}
W{}``
TM basics: example

```
0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ...
8: xend
```

```
0: xbegin;
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ...
8: xend
```
TM basics: example

0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ...
8: xend

0: xbegin;
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ...
8: xend
TM basics: example

cpu 0
PC: 3

Working Set
R{ A, B }
W{}

0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ...
8: xend

cpu 1
PC: 2

Working Set
R{ A }
W{}

0: xbegin;
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ...
8: xend

PC: 0
Working Set
R{}
W{}

PC: 1

PC: 2

PC: 3
Working Set
R{ A, B }
W{ }
TM basics: example

0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend

0: xbegin;
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend
TM basics: example

```
0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend
```
TM basics: example

```
0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend
```

Working Set
- R{ A, B }
- W{}

```
0: xbegin;
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend
```

Working Set
- R{ A, B }
- W{}
TM basics: example

0: `xbegin
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ...
8: xend

0: `xbegin;
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ...
8: xend

CPU 0
PC: 7
Working Set
R{ A,B,C }
W{}
TM basics: example

CONFLICT:
C is in the read set of cpu0, and in the write set of cpu1
TM basics: example

Assume contention manager decides cpu1 wins:

- cpu0 rolls back
- cpu1 commits
TM Implementation

0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ... 
8: xend

0: xbegin;
1: read A
2: read B
3: if(cpu % 2)
4: write C
5: else
6: read C
7: ... 
8: xend
TM Implementation

Data Versioning

• Eager Versioning
• Lazy Versioning
TM Implementation

Data Versioning
• Eager Versioning
• Lazy Versioning

Conflict Detection and Resolution
• Eager Detection (Pessimistic)
• Lazy Detection (Optimistic)
TM Implementation

Data Versioning
• Eager Versioning
• Lazy Versioning

Conflict Detection and Resolution
• Eager Detection (Pessimistic)
• Lazy Detection (Optimistic)

Conflict Detection Granularity
• ObjectGranularity
• Word Granularity
• Cache line Granularity
TM Design Alternatives

• Hardware (HTM)
  • Caches track RW set, HW speculation/checkpoint

• Software (STM)
  • Instrument RW
  • Inherit TX Object
Hardware Transactional Memory

• Idea: Track read/write sets in HW
  • commit/rollback in hardware as well
• Cache coherent hardware already manages much of this
• Basic idea: cache == speculative storage
  • HTM ~= smarter cache
• Can support many different TM paradigms
  • Eager, lazy
  • optimistic, pessimistic
Hardware TM

- “Small” modification to cache
Hardware TM

• “Small” modification to cache
Hardware TM

• “Small” modification to cache

Regular Accesses

Transactional Accesses

L1 $
Hardware TM

• “Small” modification to cache

Key ideas
• Checkpoint architectural state
• Caches: ‘versioning’ for memory
• Change coherence protocol
  • Conflict detection in hardware
• ‘Commit’ transactions if no conflict
• ‘Abort’ on conflict (or special cond)
• ‘Retry’ aborted transaction
Coherence for Conflict Detection and Versioning
Coherence for Conflict Detection and Versioning

- Lines in TMI state are speculative
- Lines in TS, TE have been read
- Invalidations/Upgrades for T* → transactional conflicts
- Commit: T* -> *
- Abort: T* → I, rollback registers
Coherence for Conflict Detection and Versioning

- Lines in TMI state are speculative
- Lines in TS, TE have been read
- Invalidations/Upgrades for T* \(\rightarrow\) transactional conflicts
- Commit: T* \(\rightarrow\) *
- Abort: T* \(\rightarrow\) I, rollback registers

Pros/Cons?
Case Study: SUN Rock

• Major challenge: diagnosing cause of Transaction aborts
  • Necessary for intelligent scheduling of transactions
  • Also for debugging code
  • debugging the processor architecture / µarchitecture
• Many unexpected causes of aborts
• Rock v1 diagnostics unable to distinguish distinct failure modes

<table>
<thead>
<tr>
<th>Mask</th>
<th>Name</th>
<th>Description and example cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x001</td>
<td>EXOG</td>
<td>Exogenous - Intervening code has run. cps register contents are invalid.</td>
</tr>
<tr>
<td>0x002</td>
<td>DOM</td>
<td>Coherence - Conflicting memory operation.</td>
</tr>
<tr>
<td>0x004</td>
<td>TCC</td>
<td>Trap Instruction - A trap instruction evaluates to “taken”</td>
</tr>
<tr>
<td>0x008</td>
<td>INSTR</td>
<td>Unsupported Instruction - Instruction not supported inside transactions.</td>
</tr>
<tr>
<td>0x010</td>
<td>PREC</td>
<td>Precise Exception - Execution generated a precise exception.</td>
</tr>
<tr>
<td>0x020</td>
<td>ASYNC</td>
<td>Async - Received an asynchronous interrupt.</td>
</tr>
<tr>
<td>0x040</td>
<td>SIZE</td>
<td>Size - Transaction write set exceeded the size of the store queue.</td>
</tr>
<tr>
<td>0x080</td>
<td>LD</td>
<td>Load - Cache line in read set evicted by transaction.</td>
</tr>
<tr>
<td>0x100</td>
<td>ST</td>
<td>Store - Data TLB miss on a store.</td>
</tr>
<tr>
<td>0x200</td>
<td>UTL</td>
<td>Control transfer - Mispredicted branch.</td>
</tr>
<tr>
<td>0x400</td>
<td>FP</td>
<td>Floating point - Divide instruction.</td>
</tr>
<tr>
<td>0x800</td>
<td>UCTR</td>
<td>Unresolved control transfer - Branch executed without resolving load on which it depends.</td>
</tr>
</tbody>
</table>

Table 1. cps register: bit definitions and example failure reasons that set them.
Case Study: SUN Rock

• Major challenge: diagnosing cause of Transaction aborts
  • Necessary for intelligent scheduling of transactions
  • Also for debugging code
  • debugging the processor architecture / µarchitecture
• Many unexpected causes of aborts
• Rock v1 diagnostics unable to distinguish distinct failure modes

<table>
<thead>
<tr>
<th>Mask</th>
<th>Name</th>
<th>Description and example cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x001</td>
<td>EX00</td>
<td>Exogenous - Intervening code has ran. cps register contents are invalid.</td>
</tr>
<tr>
<td>0x002</td>
<td>COH</td>
<td>Coherence - Conflicting memory operation.</td>
</tr>
<tr>
<td>0x004</td>
<td>TCC</td>
<td>Trap Instruction - A trap instruction evaluates to “taken”.</td>
</tr>
<tr>
<td>0x008</td>
<td>INST</td>
<td>Unsupported Instruction - Instruction not supported inside transactions.</td>
</tr>
<tr>
<td>0x010</td>
<td>PREC</td>
<td>Precise Exception - Execution generated a precise exception.</td>
</tr>
<tr>
<td>0x020</td>
<td>ASYN</td>
<td>Async - Received an asynchronous interrupt.</td>
</tr>
<tr>
<td>0x040</td>
<td>SIZE</td>
<td>Size - Transaction write set exceeded the size of the store queue.</td>
</tr>
<tr>
<td>0x080</td>
<td>LD</td>
<td>Load - Cache line in read set evicted by transaction.</td>
</tr>
<tr>
<td>0x100</td>
<td>ST</td>
<td>Store - Data TLB miss on a store.</td>
</tr>
<tr>
<td>0x200</td>
<td>CTL</td>
<td>Control transfer - Mispredicted branch.</td>
</tr>
<tr>
<td>0x400</td>
<td>FP</td>
<td>Floating point - Divide instruction.</td>
</tr>
<tr>
<td>0x800</td>
<td>UCST</td>
<td>Unresolved control transfer - Branch executed without resolving load on which it depends.</td>
</tr>
</tbody>
</table>

Table 1. cps register: bit definitions and example failure reasons that set them.
HTM: Strong Isolation vs Weak Isolation

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 atomic {</td>
<td></td>
</tr>
<tr>
<td>2     r1 = x;</td>
<td>x = 1;</td>
</tr>
<tr>
<td>3     r2 = x;</td>
<td></td>
</tr>
<tr>
<td>4      }</td>
<td></td>
</tr>
</tbody>
</table>
HTM: Strong Isolation vs Weak Isolation

Thread 1 | Thread 2
---|---
1 atomic {
2 r1 = x;  x = 1;
3 r2 = x;
4 }

Can r1 != r2?
HTM: Strong Isolation vs Weak Isolation

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 atomic {</td>
<td></td>
</tr>
<tr>
<td>2 ( r1 = x; ) ( x = 1; )</td>
<td></td>
</tr>
<tr>
<td>3 ( r2 = x; )</td>
<td></td>
</tr>
<tr>
<td>4 }</td>
<td></td>
</tr>
</tbody>
</table>

Can \( r1 \neq r2 \)?
Non-repeatable reads
HTM: Strong Isolation vs Weak Isolation

Initially, $x == 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 atomic {</td>
<td>1 atomic {</td>
<td>1 atomic {</td>
<td>1 atomic {</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>r1 = x;</td>
<td>r = x;</td>
<td>r = x;</td>
<td>r = x;</td>
</tr>
<tr>
<td></td>
<td>x = 1;</td>
<td>x = 1;</td>
<td>x = 1;</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>r2 = x;</td>
<td>x = r+1;</td>
<td>x = r+1;</td>
<td>x = r+1;</td>
</tr>
<tr>
<td>4 }</td>
<td>4 }</td>
<td>4 }</td>
<td>4 }</td>
</tr>
</tbody>
</table>

Can $r1 != r2$?

Non-repeatable reads
HTM: Strong Isolation vs Weak Isolation

1 atomic {
2 r1 = x;
3 r2 = x;
4 }

Thread 1
Thread 2

Can r1 != r2?
Non-repeatable reads

Initially, x == 0

1 atomic {
2 r = x;
3 x = r+1;
4 }

Thread 1
Thread 2

Can x==1?
HTM: Strong Isolation vs Weak Isolation

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 atomic {</td>
<td>x = 1;</td>
<td>1 atomic {</td>
<td>x = 10;</td>
</tr>
<tr>
<td>2 r1 = x;</td>
<td>x = 1;</td>
<td>2 r = x;</td>
<td>x = 10;</td>
</tr>
<tr>
<td>3 r2 = x;</td>
<td>3 x = r+1;</td>
<td>4 }</td>
<td>4 }</td>
</tr>
</tbody>
</table>

- Can r1 != r2?
  - Non-repeatable reads

- Can x==1?
  - Lost Updates
## HTM: Strong Isolation vs Weak Isolation

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Initially, x == 0</th>
<th>Initially, x is even</th>
</tr>
</thead>
</table>
| 1 atomic {  
2 r1 = x;  
3 r2 = x;  
4 } | x = 1; | 1 atomic {  
2 r = x;  
3 x = r+1;  
4 } | 1 atomic {  
2 r = x;  
3 x = r+1;  
4 } |
| Can r1 != r2? | | Can x==1? | |
| Non-repeatable reads | | Lost Updates | |
HTM: Strong Isolation vs Weak Isolation

Can r1 != r2?  
Non-repeatable reads

Initially, x == 0

1 atomic {
2  r1 = x;  
3  x = 1;  
4  r2 = x;  
}

Can x==1?  
Lost Updates

Initially, x is even

1 atomic {
2  r = x;  
3  x = r+1;  
4  }

Can r be odd?

Initially, x == 0

1 atomic {
2  r = x;  
3  x = 10;  
4  }

Initially, x is even

1 atomic {
2  x++;  
3  x++;  
4  r = x;  
}

Can r be odd?
### HTM: Strong Isolation vs Weak Isolation

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Initially, $x == 0$</th>
<th>Initially, $x$ is even</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 atomic {</td>
<td>1 atomic {</td>
<td>1 atomic {</td>
<td>1 atomic {</td>
</tr>
<tr>
<td>2 $r1 = x;$</td>
<td>2 $r = x;$</td>
<td>2 $x++;$</td>
<td>$r = x;$</td>
</tr>
<tr>
<td>3 $r2 = x;$</td>
<td>3 $x = r+1;$</td>
<td>3 $x++;$</td>
<td>$r = x;$</td>
</tr>
<tr>
<td>4 }</td>
<td>4 }</td>
<td>4 }</td>
<td>4 }</td>
</tr>
</tbody>
</table>

- **Can $r1$ != $r2$?**
  - **Non-repeatable reads**
- **Can $x$==1?**
  - **Lost Updates**
- **Can $r$ be odd?**
  - **Dirty reads**
TM Tricks

• Lock Elision
  • In many data structures, accesses are contention free in the common case
  • But need locks for the uncommon case where contention does occur
  • For example, double ended queue
  • Can replace lock with atomic section, default to lock when needed
  • Allows extra parallelism in the average case
Lock Elision

```javascript
hashTable.lock()
var = hashTable.lookup(X);
if (!var) hashTable.insert(X);
hashTable.unlock();

hashTable.lock()
var = hashTable.lookup(Y);
if (!var) hashTable.insert(Y);
hashTable.unlock();
```
Lock Elision

```
hashTable.lock()
var = hashTable.lookup(X);
if (!var) hashTable.insert(X);
hashTable.unlock();

hashTable.lock()
var = hashTable.lookup(Y);
if (!var) hashTable.insert(Y);
hashTable.unlock();
```

Hardware notices lock Instruction sequence!
Lock Elision

```javascript
hashTable.lock()
var = hashTable.lookup(X);
if (!var) hashTable.insert(X);
hashTable.unlock();

hashTable.lock()
var = hashTable.lookup(Y);
if (!var) hashTable.insert(Y);
hashTable.unlock();

Parallel Execution

atomic {
    if (!hashTable.isUnlocked()) abort;
    var = hashTable.lookup(X);
    if (!var) hashTable.insert(X);
}orElse ...

atomic {
    if (!hashTable.isUnlocked()) abort;
    var = hashTable.lookup(X);
    if (!var) hashTable.insert(X);
}orElse ...
```

Hardware notices lock Instruction sequence!
Privatization

```java
atomic {
    var = getWorkUnit();
    do_long_computation(var);
}
```
Privatization

atomic {
    var = getWorkUnit();
    do_long_computation(var);
}

VS

atomic {
    var = getWorkUnit();
}
do_long_computation(var);
Privatization

atomic {
    var = getWorkUnit();
    do_long_computation(var);
}

VS

atomic {
    var = getWorkUnit();
}
do_long_computation(var);

may only work correctly in TMs that support strong isolation. (why?)
Work Deferral

    atomic {
        do_lots_of_work();
        update_global_statistics();
    }
Work Deferral

atomic {
    do_lots_of_work();
    update_global_statistics();
}

Work Deferral

atomic {
    do_lots_of_work();
    update_global_statistics();
} atomic {
    do_lots_of_work();
    atomic open {
        atomic open {
            update_global_statistics();
        }
    }
}
Work Deferral

```c
atomic {
    do_lots_of_work();
    update_global_statistics();
}
```
Work Deferral

```
atomic {
    do_lots_of_work();
    update_global_statistics();
}
atomic {
    do_lots_of_work();
    atomic open {
        update_global_statistics();
    }
}
```
Work Deferral

atomic {
    do_lots_of_work();
    update_global_statistics();
}
atomic {
    do_lots_of_work();
    atomic open {
        atomic open {
            update_global_statistics();
        }
    }
    update_local_statistics(); //effectively serializes transactions
}
atomic{
    update_global_statististics_using_local_statistics()
}
STM: System Model

Memory cell support 4 operations:

- **Write** $i(L,v)$ - thread $i$ writes $v$ to $L$
- **Read** $i(L,v)$ - thread $i$ reads $v$ from $L$
- **LL** $i(L,v)$ - thread $i$ reads $v$ from $L$, marks $L$ read by $i$
- **SC** $i(L,v)$ - thread $i$ writes $v$ to $L$
  - returns success if $L$ is marked as read by $i$
  - Otherwise it returns failure.
STM: System Model

System == <threads, memory>

- **Write** \( i(L, v) \) - thread \( i \) writes \( v \) to \( L \)
- **Read** \( i(L, v) \) - thread \( i \) reads \( v \) from \( L \)
- **LL** \( i(L, v) \) - thread \( i \) reads \( v \) from \( L \), marks \( L \) read by \( i \)
- **SC** \( i(L, v) \) - thread \( i \) writes \( v \) to \( L \), returns success if \( L \) is marked as read by \( i \). Otherwise, it returns failure.
STM: System Model

System == <threads, memory>
Memory cell support 4 operations:

- **Write** \(i, (L, v)\) - thread \(i\) writes \(v\) to \(L\)
- **Read** \(i, (L, v)\) - thread \(i\) reads \(v\) from \(L\)
- **LL** \(i, (L, v)\) - thread \(i\) reads \(v\) from \(L\), marks \(L\) read by \(i\)
- **SC** \(i, (L, v)\) - thread \(i\) writes \(v\) to \(L\) returns success if \(L\) is marked as read by \(i\). Otherwise, it returns failure.
STM: System Model

System == <threads, memory>

Memory cell support 4 operations:

- **Write\(^i\)(L,v)** - thread \(i\) writes \(v\) to \(L\)

- **Read\(^i\)(L,v)** - thread \(i\) reads \(v\) from \(L\)

- **LL\(^i\)(L,v)** - thread \(i\) reads \(v\) from \(L\), marks \(L\) read by \(i\)

- **SC\(^i\)(L,v)** - thread \(i\) writes \(v\) to \(L\)
  - returns **success** if \(L\) is marked as read by \(i\).
  - Otherwise it returns **failure**.
STM: System Model

System == \langle\text{threads}, \text{memory}\rangle

Memory cell support 4 operations:

- \text{Write}^i(L,v) - thread \(i\) writes \(v\) to \(L\)
- \text{Read}^i(L,v) - thread \(i\) reads \(v\) from \(L\)
STM: System Model

System == <threads, memory>

Memory cell support 4 operations:

- **Write**\(^i\)(L,v) - thread \(i\) writes \(v\) to \(L\)
- **Read**\(^i\)(L,v) - thread \(i\) reads \(v\) from \(L\)
- **LL**\(^i\)(L,v) - thread \(i\) reads \(v\) from \(L\), marks \(L\) read by \(i\)
- **SC**\(^i\)(L,v) - thread \(i\) writes \(v\) to \(L\) returns **success** if \(L\) is marked as read by \(i\).
  
The other case it returns **failure**.
STM: System Model

System == <threads, memory>

Memory cell support 4 operations:
- $\text{Write}^i(L,v)$ - thread $i$ writes $v$ to $L$
- $\text{Read}^i(L,v)$ - thread $i$ reads $v$ from $L$
- $\text{LL}^i(L,v)$ - thread $i$ reads $v$ from $L$, marks $L$ read by $i$
- $\text{SC}^i(L,v)$ - thread $i$ writes $v$ to $L$
  - returns $\text{success}$ if $L$ is marked as read by $i$.
  - Otherwise it returns $\text{failure}$. 
STM Design Overview
STM Design Overview

This is the shared memory, (STM Object)
This is the shared memory, (STM Object)

Pointers to threads (Rec Objects)
Threads: Rec Objects

class Rec {
    boolean stable = false;
    boolean, int status= (false,0);  //can have two values...
    boolean allWritten = false;
    int version = 0;
    int size = 0;
    int locs[] = {null};
    int oldValues[] = {null};
}

Each thread → instance of Rec class (short for record).

Rec instance defines current transaction on thread
public class STM {
    int memory[];
    Rec ownerships[];

    public boolean, int[] startTranscation(Rec rec, int[] dataSet){...};

    private void initialize(Rec rec, int[] dataSet)
    private void transaction(Rec rec, int version, boolean isInitiator) {...};
    private void acquireOwnerships(Rec rec, int version) {...};
    private void releaseOwnershipd(Rec rec, int version) {...};
    private void agreeOldValues(Rec rec, int version) {...};
    private void updateMemory(Rec rec, int version, int[] newvalues) {...};
Flow of a transaction
Flow of a transaction
Flow of a transaction
Flow of a transaction

1. `startTransaction`
2. `STM`
3. `Thread i`
Flow of a transaction

startTransaction

STM

Threads

Thread i
Flow of a transaction
Flow of a transaction

1. startTransaction
2. initialize

STM

Threads

Thread i
Flow of a transaction

- `startTransaction`
- `initialize`
- `transaction`

STM

Threads

Thread i
Flow of a transaction

startTransaction

initialize

transaction

STM

Threads

Thread i
Flow of a transaction

1. startTransaction
2. initialize
3. transaction
   - acquire Ownerships: (Null, 0)
   - (Failure, failed loc)

STM

Threads

Thread i
Flow of a transaction

1. `startTransaction`
2. `initialize`
3. `transaction`
4. `acquire Ownership`
   - `(Null, 0)`
   - `(Failure, failed loc)`

STM

Threads
Flow of a transaction

1. `startTransaction`
2. `initialize`
3. `transaction`
4. `agreeOldValues` (return value: (Null, 0))
5. `acquire` (return values: (Failure, failed loc))
Flow of a transaction

1. Start Transaction
2. Initialize
3. Transaction
4. Acquire
   - Ownerships
   - *(Null, 0)*
   - *(Failure, failed loc)*

- STM
- Threads
Flow of a transaction

- `startTransaction`
- `initialize`
- `transaction`
- `agreeOldValues`
- `calcNewValues`
- `acquire Ownerships`: (Failure, failed loc)
- STM
- Threads:
  - Thread i
Flow of a transaction

1. **startTransaction**
2. **initialize**
3. **transaction**
4. **agreeOldValues**
5. **calcNewValues**

**STM** (Failure, failed loc)

**Threads**

- Thread i

- (Null, 0)

- Acquire Ownership
Flow of a transaction

1. **startTransaction**
2. **initialize**
3. **transaction**
4. **agreeOldValues**
5. **calcNewValues**
6. **updateMemory**

(Null, 0) (Failure, failed loc)

STM — Threads
Flow of a transaction

- `startTransaction`
- `initialize`
- `transaction`
- `agreeOldValues`
- `calcNewValues`
- `updateMemory`

STM

Threads

- `Thread i`

Diagrams:
- `STM` to `Threads`
- `updateMemory` to `initialize`
- `initialize` to `transaction`
- `transaction` to `agreeOldValues`
- `agreeOldValues` to `calcNewValues`
- `calcNewValues` to `updateMemory`
- `updateMemory` to `STM`
Flow of a transaction

startTransaction

Thread i

initialize

STM

(transaction)

acquire

Ownerships

(calcNewValues)

updateMemory

agreeOldValues

release

Ownerships

(Failure, failed loc)

(Null, 0)
Flow of a transaction

- startTransaction
  - initialize
    - transaction
      - acquireOwnerships
        - agreementOldValues
          - calcNewValues
            - updateMemory
              - releaseOwnerships
Flow of a transaction

- startTransaction
  - initialize
    - transaction
      - acquire Ownerships
        - (Null, 0)
      - calcNewValues
    - updateMemory
- release Ownerships

STM

Success

Threads

Thread i
Flow of a transaction

- release Ownerships
- updateMemory
- calcNewValues
- agreeOldValues

- startTransaction
- initialize
- transaction

STM

Success

Thread i

(Null, 0) (Failure, failed loc)
Flow of a transaction

- `release Ownerships`
- `updateMemory`
- `calcNewValues`
- `agreeOldValues`
- `startTransaction`
- `initialize`
- `transaction`
- `acquire Ownerships`
- `release Ownerships`

STM

Success

Threads

Thread i
Flow of a transaction

- `release Ownerships`
- `updateMemory`
- `calcNewValues`
- `agreeOldValues`
- `startTransaction`
- `initialize`
- `transaction`
- `acquire Ownerships`
- `release Ownerships`
Flow of a transaction

1. **Thread i**
   - `startTransaction`
   - `initialize`
   - `transaction`
   - `isInitiator?`
     - `T`
     - `F`
     - `(Null, 0)`
     - `(Failure, failed loc)`
   - `release Ownerships`

2. **STM**
   - `updateMemory`
   - `calcNewValues`
   - `agreeOldValues`
   - `release Ownerships`
Flow of a transaction

Thread i

1. startTransaction
2. initialize
3. transaction
4. acquire
5. release
6. updateMemory
7. calcNewValues
8. agreeOldValues
9. release Ownerships

STM

10. Success

isInitiator?

11. T
12. Initiate helping transaction to failed loc (isInitiator:=F)
13. F

(Null, 0)

(Transaction, 0)

(Failure, failed loc)

release Ownerships

Threads
Flow of a transaction

- startTransaction
- initialize
- transaction
- acquireOwnerships
- releaseOwnerships
- updateMemory
- calcNewValues
- agreeOldValues
- isInitiator?
  - isInitiator:=F
  - Null, 0
  - (Failure, failed loc)

STM

Threads

Success

Failure
Implementation

```java
public boolean startTransaction(Rec rec, int[] dataSet) {
    initialize(rec, dataSet);
    rec.stable = true;
    transaction(rec, rec.version, true);
    rec.stable = false;
    rec.version++;
    if (rec.status) return (true, rec.oldValues);
    else return false;
}
```
Implementation

```java
public boolean startTransaction(Rec rec, int[] dataSet) {
    initialize(rec, dataSet);
    rec.stable = true;
    transaction(rec, rec.version, true);
    rec.stable = false;
    rec.version++;
    if (rec.status) return (true, rec.oldValues);
    else return false;
}
```

rec – The thread that executes this transaction.

dataSet – The location in memory it needs to own.
Implementation

```java
public boolean startTransaction(Rec rec, int[] dataSet) {
    initialize(rec, dataSet);
    rec.stable = true;
    transaction(rec, rec.version, true);
    rec.stable = false;
    rec.version++;
    if (rec.status) return (true, rec.oldValues);
    else return false;
}
```

rec – The thread that executes this transaction.

dataSet – The location in memory it needs to own.

This notifies other threads that I can be helped.
private void transaction(Rec rec, int version, boolean isInitiator) {
    acquireOwnerships(rec, version); // try to own locations

    (status, failedLoc) = LL(rec.status);
    if (status == null) { // success in acquireOwnerships
        if (version != rec.version) return;
        SC(rec.status, (true, 0));
    }

    (status, failedLoc) = LL(rec.status);
    if (status == true) { // execute the transaction
        agreeOldValues(rec, version);
        int[] newVals = calcNewVals(rec.oldValues);
        updateMemory(rec, version);
        releaseOwnerships(rec, version);
    } else { // failed in acquireOwnerships
        releaseOwnerships(rec, version);
        if (isInitiator) {
            Rec failedTrans = ownerships[failedLoc];
            if (failedTrans == null) return;
            else { // execute the transaction that owns the location you want
                int failedVer = failedTrans.version;
                if (failedTrans.stable) transaction(failedTrans, failedVer, false);
            }
        }
    }
}
Implementation

```java
private void transaction(Rec rec, int version, boolean isInitiator) {
    acquireOwnerships(rec, version); // try to own locations

    (status, failedLoc) = LL(rec.status);
    if (status == null) { // success in acquireOwnerships
        if (version != rec.version) return;
        SC(rec.status, (true,0));
    }

    (status, failedLoc) = LL(rec.status);
    if (status == true) { // execute the transaction
        agreeOldValues(rec, version);
        int[] newVals = calcNewVals(rec.oldvalues);
        updateMemory(rec, version);
        releaseOwnerships(rec, version);
    } else { // failed in acquireOwnerships
        releaseOwnerships(rec, version);
        if (isInitiator) {
            Rec failedTrans = ownerships[failedLoc];
            if (failedTrans == null) return;
            else { // execute the transaction that owns the location you want
                int failedVer = failedTrans.version;
                if (failedTrans.stable) transaction(failedTrans, failedVer, false);
            }
        }
    }
}
```

rec – The thread that executes this transaction.
version – Serial number of the transaction.
isInitiator – Am I the initiating thread or the helper?
private void transaction(Rec rec, int version, boolean isInitiator) {
    acquireOwnerships(rec, version); // try to own locations
    (status, failedLoc) = LL(rec.status);
    if (status == null) { // success in acquireOwnerships
        if (version != rec.version) return;
        SC(rec.status, (true,0));
    }
    (status, failedLoc) = LL(rec.status);
    if (status == true) { // execute the transaction
        agreeOldValues(rec, version);
        int[] newVals = calcNewVals(rec.oldValues);
        updateMemory(rec, version);
        releaseOwnerships(rec, version);
    } else { // failed in acquireOwnerships
        releaseOwnerships(rec, version);
        if (isInitiator) {
            Rec failedTrans = ownerships[failedLoc];
            if (failedTrans == null) return;
            else { // execute the transaction that owns the location you want
                int failedVer = failedTrans.version;
                if (failedTrans.stable) transaction(failedTrans, failedVer, false);
            }
        }
    }
}
Implementation

```csharp
private void acquireOwnerships(Rec rec, int version) {
    for (int j=1; j<=rec.size; j++) {
        while (true) do {
            int loc = locs[j];
            if (LL(rec.status) != null) return; // transaction completed by some other thread
            Rec owner = LL(ownerships[loc]);
            if (rec.version != version) return;
            if (owner == rec) break; // location is already mine
            if (owner == null) {
                // acquire location
                if (SC(rec.status, (null, 0))) {
                    if (SC(ownerships[loc], rec)) {
                        break;
                    }
                }
            } else {
                // location is taken by someone else
                if (SC(rec.status, (false, j))) return;
            }
        }
    }
}
```

If I’m not the last one to read this field, it means that another thread is trying to execute this transaction. Try to loop until I succeed or until the other thread completes the transaction.
private void agreeOldValues(Rec rec, int version) {
    for (int j=1; j<=rec.size; j++) {
        int loc = locs[j];
        if ( LL(rec.oldvalues[loc]) != null ) {
            if (rec.version != version) return;
            SC(rec.oldvalues[loc], memory[loc]);
        }
    }
}

private void updateMemory(Rec rec, int version, int[] newvalues) {
    for (int j=1; j<=rec.size; j++) {
        int loc = locs[j];
        int oldValue = LL(memory[loc]);
        if (rec.allWritten) return; // work is done
        if (rec.version != version) return;
        if (oldValue != newvalues[j]) SC(memory[loc], newValues[j]);
    }
    if (! LL(rec.allWritten) ) {
        if (rec.version != version) SC(rec.allWritten, true);
    }
}
## HTM vs. STM

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast (due to hardware operations)</td>
<td>Slow (due to software validation/commit)</td>
</tr>
<tr>
<td>Light code instrumentation</td>
<td>Heavy code instrumentation</td>
</tr>
<tr>
<td>HW buffers keep amount of metadata low</td>
<td>Lots of metadata</td>
</tr>
<tr>
<td>No need of a middleware</td>
<td>Runtime library needed</td>
</tr>
<tr>
<td>Only short transactions allowed (why?)</td>
<td>Large transactions possible</td>
</tr>
</tbody>
</table>
# HTM vs. STM

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast (due to hardware operations)</td>
<td>Slow (due to software validation/commit)</td>
</tr>
<tr>
<td>Light code instrumentation</td>
<td>Heavy code instrumentation</td>
</tr>
<tr>
<td>HW buffers keep amount of metadata low</td>
<td>Lots of metadata</td>
</tr>
<tr>
<td>No need of a middleware</td>
<td>Runtime library needed</td>
</tr>
<tr>
<td>Only short transactions allowed (why?)</td>
<td>Large transactions possible</td>
</tr>
</tbody>
</table>

How would you get the best of both?
Hybrid-TM

• Best-effort HTM (use STM for long trx)
• Possible conflicts between HW,SW and HW-SW Trx
  • What kind of conflicts do SW-Trx care about?
  • What kind of conflicts do HW-Trx care about?

• Some initial proposals:
  • HyTM: uses an ownership record per memory location (overhead?)
  • PhTM: HTM-only or (heavy) STM-only, low instrumentation
Questions?