Feedback-Directed Barrier Optimization in a Strongly Isolated STM

Nathan Bronson
Christos Kozyrakis
Kunle Olukotun

POPL ’09, 22 Jan 2009

Concurrency with Threads: How Is Shared Mutable State Managed?

- Locks – widely used, but…
  - Not composable
  - Correctness is a whole-program property
- Transactional memory (TM)
  - `atomic` blocks appear to be serialized
  - Runtime provides atomicity and isolation
  - Enables local correctness reasoning
    - ***Unless atomicity or isolation is not complete***

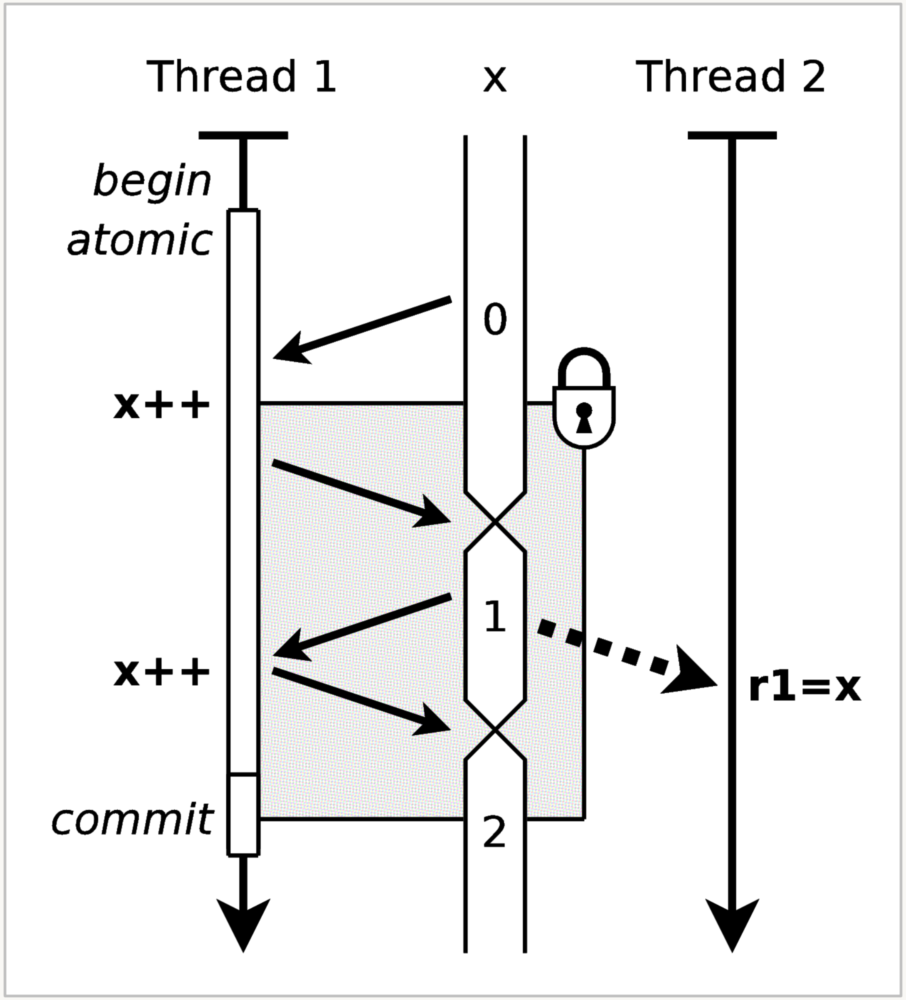
Implementing Software TM

- Txn reads and writes replaced by barriers
  - Code that implements atomic and isolated access
  - One way: eager versioning with optimistic conflict detection
    - Read barrier records version number for later validation
    - Write barrier grabs lock and stores old value in an undo log
    - Rollback on deadlock or validation failure
- "Isolation barriers" for non-txn access?

  No → weak isolation
    - Non-txn reads and writes bypass txn illusion

  Yes → strong isolation
    - Txns are always atomic and isolated

Isolation Failure in a Weakly Isolated STM

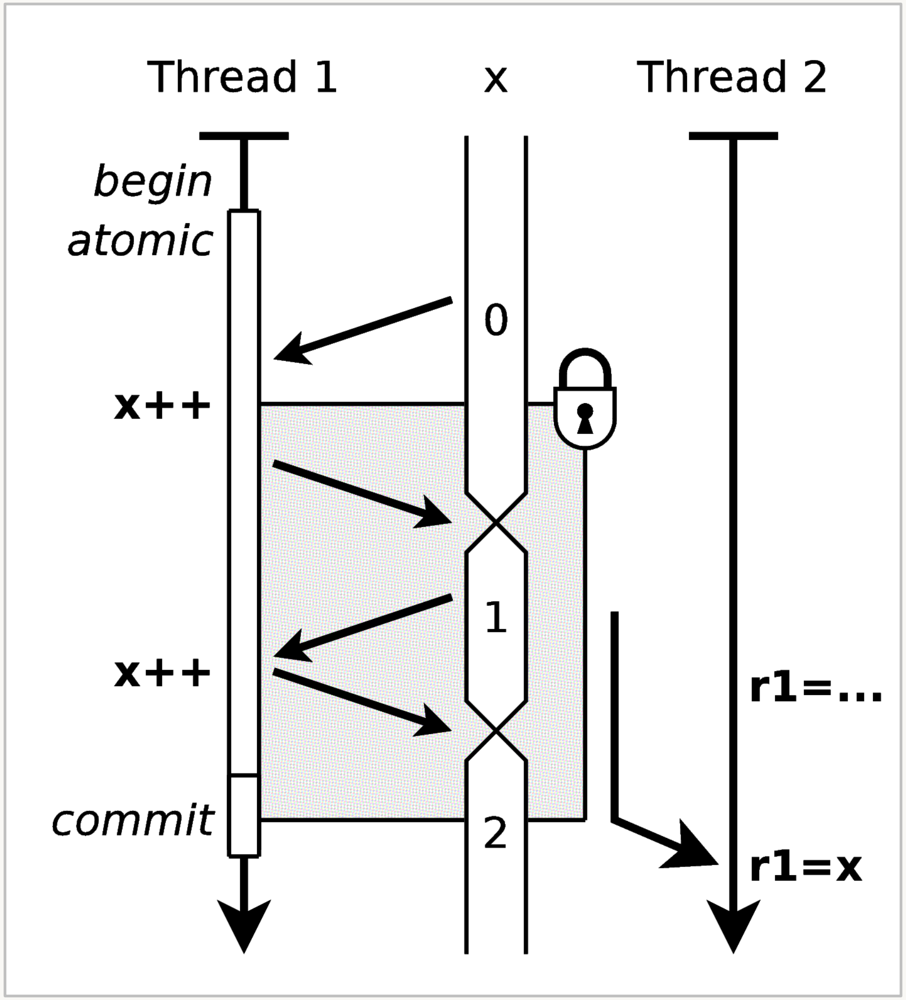



```
// Initially x==0
```

```
// Thread 1
atomic {
   txnBegin()
   txnOpenForRead(x)
   txnOpenForWrite(x)
   x++;
   x++;
   txnCommit()
}
```

```
// Thread 2
r1 = x;
assert (r1%2 == 0);
```

Strongly Isolated Non-Txn Access with an Isolation Barrier




```
// Initially x==0
```

```
// Thread 1
atomic {
   txnBegin()
   txnOpenForRead(x)
   txnOpenForWrite(x)
   x++;
   x++;
   txnCommit()
}
```

```
// Thread 2
r1 = nonTxnRead(x);
assert (r1%2 == 0);
```

Tradeoffs Between Weak and Strong Isolation

- Weak isolation → ***fast but unsafe***
  - − Undefined results if any heterogeneous access occurs
    - Values from-thin-air
    - Catch-fire semantics
  - − Following the rules is much harder than expected
    - Invalid txns may run for a while before rolling back
    - Inconsistent txns may execute accesses from impossible branches
    - Library and legacy code cannot safely be called from a txn
  - + Minimal performance impact on non-txn code
- Strong isolation → ***safe but slow***
  - + Easy formal and informal reasoning
  - − Prohibitively slow

Our goal: strong isolation with good performance

Result: average overhead reduced from 505% to 16%

Safe Access Patterns that Don’t Need Isolation Barriers

- One safe pattern is Unmodified-After-Heterogeneous-Access (UAHA)
  - Ignore reads and writes to provably thread-local data
  - All txns that write x commit or roll back before first non-txn access
  - Last non-txn write to x finishes before first txn access
- Many simpler properties imply UAHA
  - Not-Accessed-In-Txn (NAIT)
  - Read-Only (RO)
  - Unmodified-After-Txn-Commit (UATC)

Our Approach: Dynamically Verify that Accesses Follow a Safe Pattern

- Hypothesize that a safe access pattern holds for field ***`f`***
- Replace ***`f`***'s txn and isolation barriers with "checking barriers"
  - Checking barriers dynamically verify the access pattern
  - Checking barriers block if access pattern isn't followed
  - ***By blocking all threads that would violate our hypothesis, we make it a self-fulfilling prophecy***
- Rescue blocked threads by using hot swap to replace all of the barriers for ***`f`***
  - Install checking barriers for a new hypothesis if possible
  - Revert to full (slow) txn and isolation barriers if necessary

Checking Barrier Synchronization Costs

- General UAHA pattern produces mutual exclusion and happens-before relationships for accesses to the same instance
  - For all accesses a, b to a field of an escaped instance r

    $\neg(a = \textsc{NonTxnWrite} \wedge b = \textsc{TxnOpenForWrite})$

    $a = \textsc{NonTxnWrite} \wedge b = \textsc{TxnOpenForRead} \quad \Rightarrow \quad a \rightarrow_{\mathrm{hb}} b$

    $a = \textsc{TxnWriteCompleted} \wedge b = \textsc{NonTxnRead} \quad \Rightarrow \quad a \rightarrow_{\mathrm{hb}} b$
  - Dynamic check requires synchronization on r's metadata
- Simpler patterns need less or no synchronization
  - For example NAIT just prohibits half of each conflicting pair

    $a = \textsc{NonTxnRead} \vee a = \textsc{NonTxnWrite}$
- Context-sensitivity is much less expensive than state
  - Very cheap to record whether an object was created in a txn
  - Select among two simpler access patterns, such as NAIT and UATC
- See the paper for 23 hypotheses that allow speedup for our STM

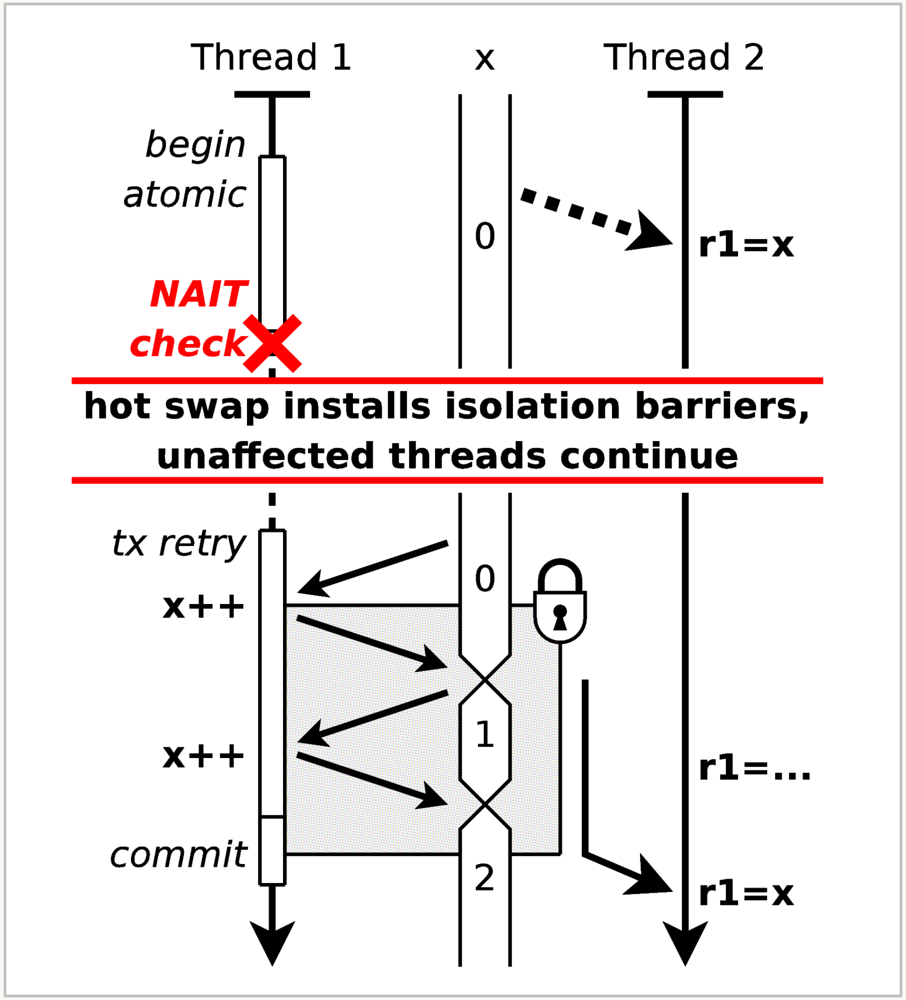
Checking Barriers for the Not-Accessed-In-Txn Pattern

```
// allowed by NAIT
nonTxnRd$f(ref)     { return ref.f; }
nonTxnWr$f(ref, v) { ref.f = v; }

// not allowed by NAIT
txnOpenRd$f(ref) { observed$f |= OBS_TXN_READ;
                   rollbackAndChangeHypoth(); }
txnOpenWr$f(ref) { observed$f |= OBS_TXN_WRITE;
                   rollbackAndChangeHypoth(); }
```

- Hypothesis correct → checking barrier is free
- Hypothesis incorrect → still strongly isolated
  - Retry txn after all barriers for ***f*** have been hot swapped

Strong Isolation Even With an Incorrect Hypothesis



- Before txn access to *x*
  - NAIT is hypothesized
  - Non-txn accesses are fast while hypothesis still holds
- First access from txn
  - Rollback
  - Hot swap installs full txn and isolation barriers
- After
  - Non-txn accesses use isolation barrier

How Do We Form Hypotheses?

- Patterns trade generality for the cost of checking
- Start aggressive
  - Assume Not-Accessed-In-Txn
  - Hot swap to fix incorrect hypotheses
- Start conservative
  - Count isolation barrier invocations
  - Hot swap to tighten hypothesis for hot barriers
  - Faster than aggressive in our implementation
- Start with hypotheses from the last execution
  - Works well, safe even if changes have been made to app
- Minimize the impact of hot swap on other threads
  - Two-phase swap blocks only threads that call a changing barrier

Experimental Validation

- Run in AJ, a bytecode-rewriting STM in/for Java
  - Elapsed time on 2×4-core Xeon with HotSpot™ Server JVM
  - Barriers are static methods, hot swap replaces bytecode
- Success: lowered non-txn overheads of strong isolation
  - 10 apps from Dacapo, SpecJBB2005
  - ***Strong isolation overhead reduced from 505% to 16%***
- Success: accelerated mixed txn benchmark
  - Based on SpecJBB2005
  - Weakly isolated execution accelerated by 31%
  - Strongly isolated execution accelerated by 34%
- See paper for more details and hypothesis prevalence

Thank You

- Questions?

A Privatization Problem in a Weakly Isolated Java STM

// Initially coll = { {x=0,y=0} }

```
// Thread 1
atomic {
  for (item: coll) {
    item.x++;
    item.y++;
  }
            → rollback
}
```

```
// Thread 2
atomic {
  r1 = coll.removeFirst();
}
r2 = r1.x
r3 = r1.y
assert (r2 == r3);
```

- Thread 2 may observe **.x** and **.y** while rollback is in progress

Example from Menon et al, Transact '08

A Publication Problem in a Weakly Isolated Java STM

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

```
// Thread 1


data = 1;
atomic { ready = true; }
```

```
// Thread 2
atomic {
  r1 = data;


  if (ready) val = r1;
}
assert (val != 42);
```

- Despite race, with locks Java memory model disallows val == 42
- Weakly isolation exposes benign race
- Object-granularity STM can introduce early reads

Our Family of Optimization Hypotheses

- All of our OHs imply Unmodified After Heterogeneous Access (**UAHA**)
  - Quite general, but too expensive to check
- Ignore accesses from objects statically proven thread-local
- Stateless optimization hypotheses
  - **ANY** = no acceleration
  - **RO** = Read Only (after escape)
  - **NAIT** = Not Accessed In Txn
  - **NAOT** = Not Accessed Outside Txn
- Stateful optimization hypotheses, set per-field bit on event
  - **UATC** = Unmodified After creating Txn Commit
  - **UATX** = Unmodified After TXn access
  - **UANT** = Unmodified After Non-Txn access
- Compound hypotheses predicated on whether object was created in a txn
  - Examples <nt=**UATX**,tx=**ANY**>
- For our system, 23 OHs have checking barriers faster than TM's barriers
  - <**RO**,**UATC**> and <**NAIT**,**NAIT**> have optimal isolation barriers

Software Transactional Memory (A Typical Eager Versioning Implementation)

- Write barrier replaces all stores inside **`atomic`** block
  - Lock *x*
  - Log old value,
  - Update in-place
- Read barrier replaces all loads inside **`atomic`** block
  - Verify not locked by another txn
  - Record version from *x*'s metadata
  - Read value
- On commit
  - Validate all reads by rechecking versions
  - Increment versions for written values
  - Release all locks
- Rollback on deadlock or validation failure
  - Apply undo log
  - Releases all locks

AJ: A Bytecode-Rewriting STM in Java

- Atomic execution for Java without language extensions
  - **`static void TM.atomic(Runnable task)`**
- Eager versioning, object granularity, optimistic read set validation using version numbers
- Java + HotSpot's **`sun.misc.Unsafe`**
- Classes are rewritten during class loading
  - Core Java libraries pre-instrumented (to avoid circularity)
  - Methods split into txn and non-txn
  - Java **`long`** added to objects for metadata
  - State bits for arrays hidden in the 25 unused header bits on 64-bit HotSpot, array locks and versions hashed
- Hot swap uses Java's Instrumentation API
  - Barriers are static methods in auto-generated auxiliary classes

Swapping with Minimal Blocking

- Requirement
  – Old and new barrier versions may not execute at the same time
- Goal
  – Don’t block code that does not use a changing barrier
- Solution: swap twice
  1. Non-txn code periodically copies a global timestamp to a per-thread field
  2. Hot swap installs a blocking “quiescing barrier”
  3. Increment the global timestamp
  4. Wait until all threads have blocked or copied the new timestamp value
  5. Swap in new barriers
  6. Unblock quiesced threads