NON-BLOCKING DATA STRUCTURES

AND TRANSACTIONAL MEMORY

Tim Harris, 31 October 2012

Lecture 7

- Transactional memory
- HTM
- STM
- Programming models for TM
- Perspectives on TM research

Updating memory, one location at a time

- Theoretical result (informally): CAS is a strong enough primitive to build a wait-free *anything*
- "Consensus hierarchy":
  - For a given primitive, what is the max number of processes (n) between which it can solve wait-free consensus?
  - Read/write memory locations: n=1
  - FIFO queues: n=2
  - Swap, TAS, FADD, …: n=2
  - CAS: n unbounded
- (=> fundamental limitation to the primitives on IBM 360, NYU Ultracomputer, early SPARC, …)

Updating memory, one location at a time

- Practical observation: building with CAS is difficult, and incurs overheads not present in lock-based algorithms
- Two ways to look at this:

> Each individual read, write, CAS, must maintain the data structure's invariants. Typically, a CAS at the linearization point must update the logical state of the data structure by touching *just one word*.

Updating memory, one location at a time

- Practical observation: building with CAS is difficult, and incurs overheads not present in lock-based algorithms
- Two ways to look at this:

Suppose a thread is pre-empted just before making a write or CAS. The OS can re-schedule the thread at any time. The application better make sure it is safe to do so!
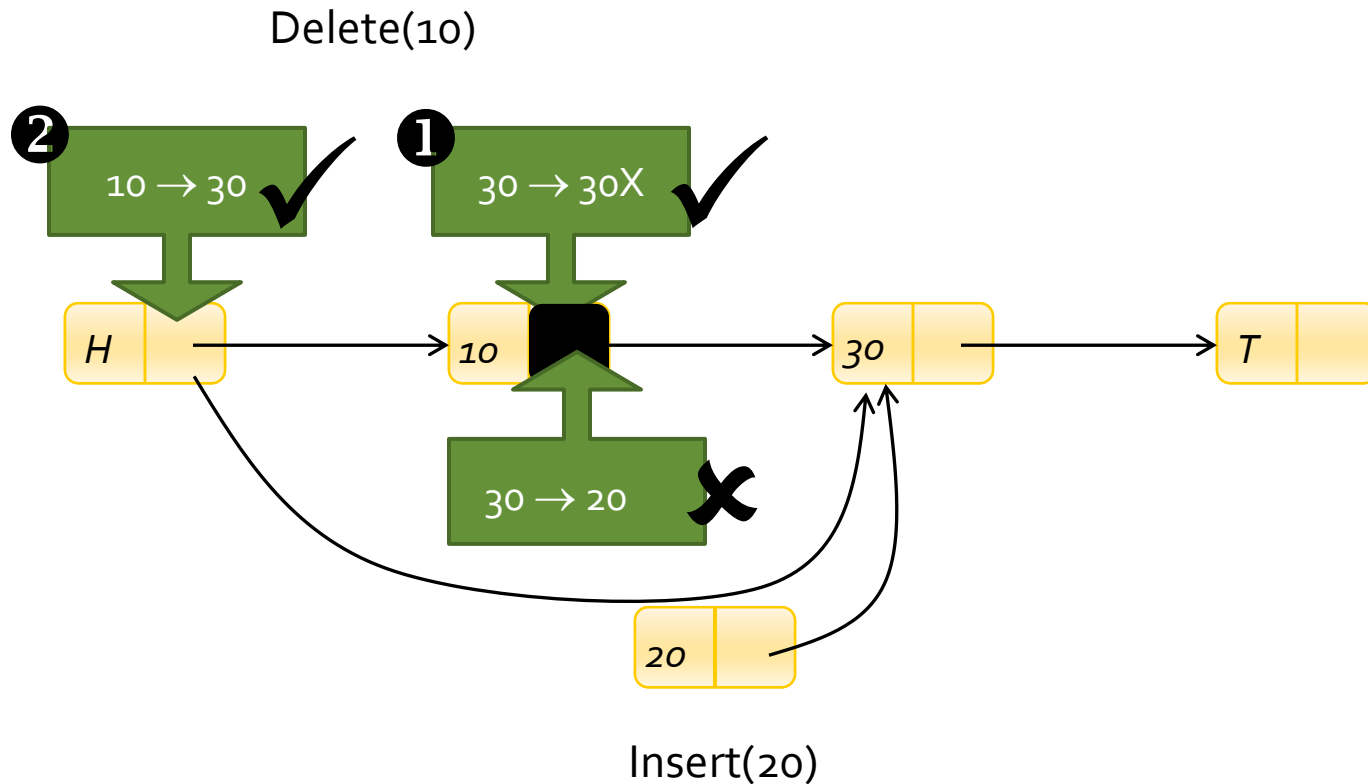
- => provide support for accessing multiple locations as a single atomic step
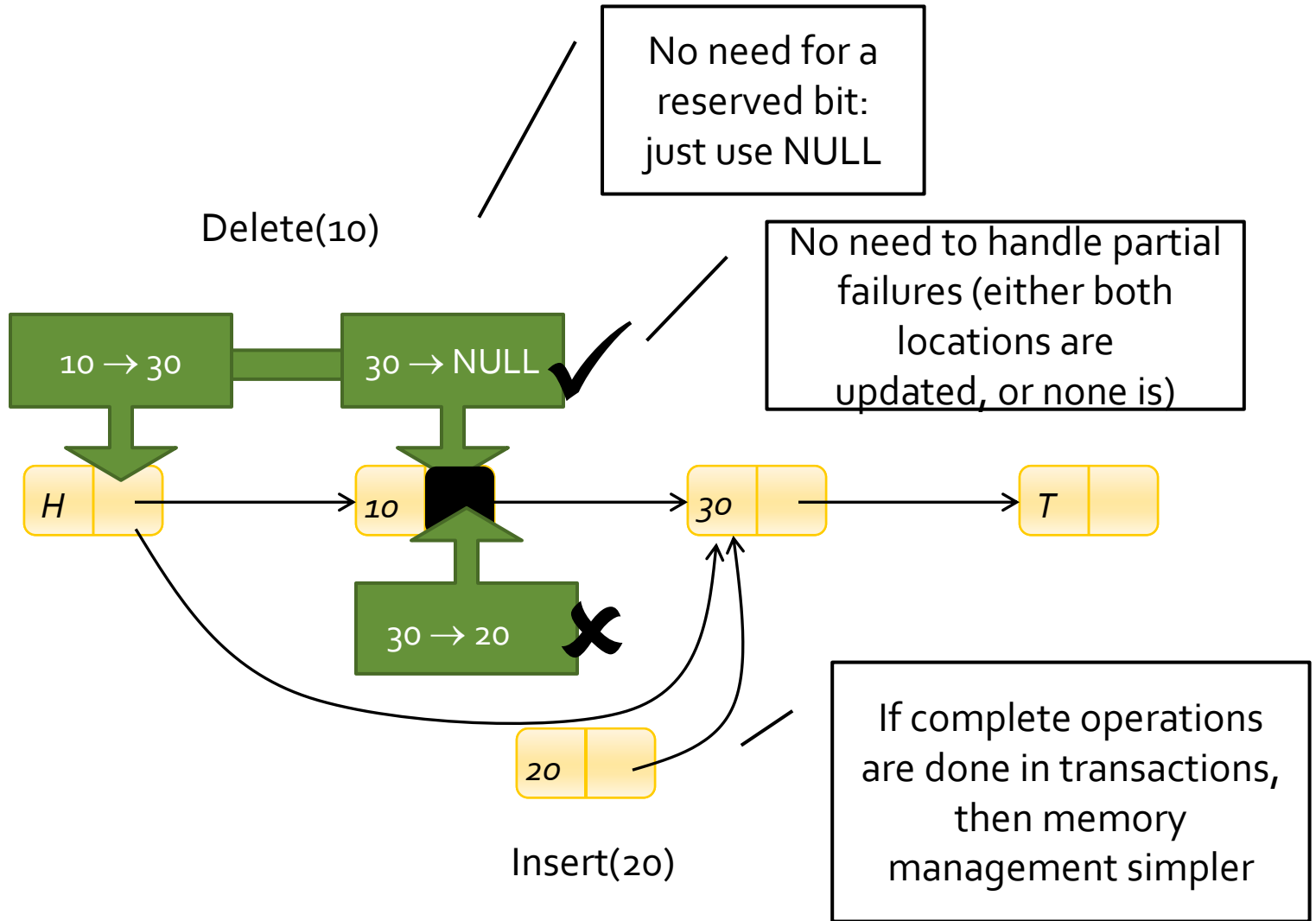
Transactional memory

- Perform a series of reads / writes / computation
  - The TM implementation is responsible for making it appear as a single atomic action
  - More than just "multi-word CAS": can compute based on the values read, and decide where/what to write
- Like a database transaction:
  - A – Atomicity (either all the accesses appear to occur, or none)
  - C – Consistency (invariants restored between transactions, not at each step within one)
  - I – Isolation (between transactions on concurrent threads)
  - (D – Durability, people argue about what the terminology means here…)

Logical vs physical deletion

- Use a 'spare' bit to indicate logically deleted nodes:

Delete(10)

❷ 10 → 30 ✔

❶ 30 → 30X ✔

H   10   30   T

30 → 20 ✘

20

Insert(20)

Deletion with TM

No need for a
reserved bit:
just use NULL

Delete(10)

No need to handle partial
failures (either both
locations are
updated, or none is)

10 → 30   30 → NULL ✔

| H | | 10 | | 30 | | T | |

30 → 20 ✘

20

Insert(20)

If complete operations
are done in transactions,
then memory
management simpler

Overview :implementation techniques

- Software TM
  - Build from CAS (or whatever the hardware provides)
  - Use techniques such as locking, version numbers, internally
  - Implement the complexity once, rather than per data structure
- Hardware TM
  - Typically, build on the cache coherence protocol
  - Cache already tracks which lines are in which modes
  - Track if a line is "stolen" during a transaction's execution
  - Sand-box the transaction's behavior until it commits

TM in hardware

HTM interfaces

- Focus here is on the ISA (programming API) rather than micro-architecture (implementation)

- Bounded:
  - Bound on the number of locations a tx may access
  - "up to 2", "up to 4", …

- Best effort:
  - Any transaction may fail (indeed, some transactions may *always* fail, even in isolation, for reasons hard to explain via the ISA)

- Fairness:
  - Is there any guarantee about how different threads make progress with their transactions?
  - (How does this compare with read, write, or CAS?)

Rock interface:

- chkpt <fail_addr>

- commit

- abort

- Best-effort
  - Reads must fit in L1 (NB: limited associativity, as well as size)
  - Writes must fit in the store buffer
  - Some instructions are prohibited
  - Speculation in the h/w can lead to surprises...

Intel TSX ("Haswell") interface

- Restricted transactional memory mode:

- XBEGIN <fail_addr>
- XEND
- XABORT

- Explicit instructions to start/commit speculation
- No guarantee for bounded size transactions
- Nesting (flattening into outermost tx)
- Broad support for most of user-mode ISA

Intel TSX ("Haswell") interface

- Hardware lock-elision mode:

- XACQUIRE *
- XRELEASE *

- Prefixes added to instructions that acquire/release locks
- Check the lock is available
- If so, speculatively run the critical section, monitoring the lock's address for writes
- Get to XRELEASE without conflict? Commit the updates, stop speculating

Basic mutex

```
// Atomic CAS to acquire lock
mov eax <- 0 // Old value
Mov ecx <- 1 // New value
lock cmpxchg &flag, ecx

// Fall-back in case lock not available
jnz slow_path

// Do critical section
…

// Release lock
mov &flag <- 0
```

Basic mutex with HLE

```
// Atomic CAS to acquire lock
mov eax <- 0 // Old value
Mov ecx <- 1 // New value
xacquire lock cmpxchg &flag, ecx

// Fall-back in case lock not available
jnz slow_path

// Do critical section
…

// Release lock
xrelease mov &flag <- 0
```

CAS fails: go to slow path

CAS succeeds: add lock to read set, start monitoring reads/writes

If we read from the lock: see 1. If others read: see 0. If others speculatively acquire: OK. If others actually write: abort.

Storing 0 back without intervening writes to flag: finish speculation

TM in software

Implementation techniques

- Direct-update STM
  - Allow transactions to make updates in place in the heap
  - Avoids reads needing to search the log to see earlier writes that the transaction has made
  - Makes successful commit operations faster at the cost of extra work on contention or when a transaction aborts
- Compiler integration
  - Decompose the transactional memory operations into primitives
  - Expose the primitives to compiler optimization (e.g. to hoist concurrency control operations out of a loop)
- Runtime system integration
  - Integration with the garbage collector or runtime system components to scale to atomic blocks containing 100M memory accesses
  - Memory management system used to detect conflicts between transactional and non-transactional accesses

Bartok-STM

- Use per-object meta-data ("TMWs")
- Each TMW is either:
  - Locked, holding a pointer to the transaction that has the object open for update
  - Available, holding a version number indicating how many times the object has been locked
- Writers eagerly lock TMWs to gain access to the object, using eager version management
  - Maintain an undo log in case of roll-back
- Readers log the version numbers they see and perform lazy conflict detection at commit time

Example: uncontended swap

a:
```
V150
1000
```

b:
```
V250
2000
```

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```
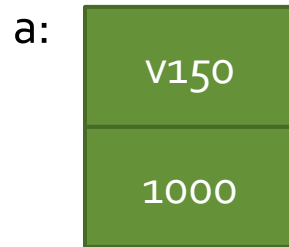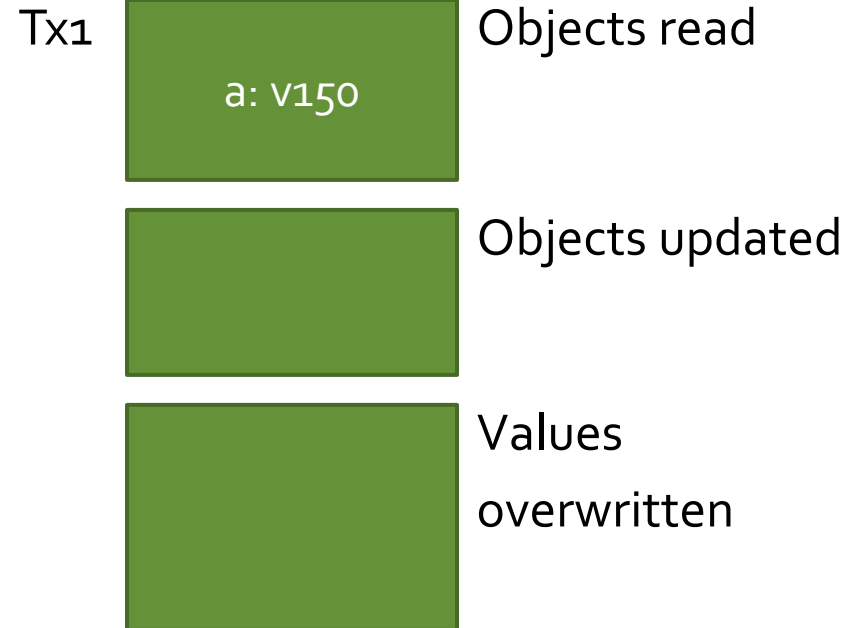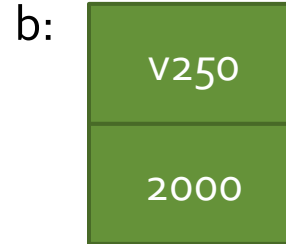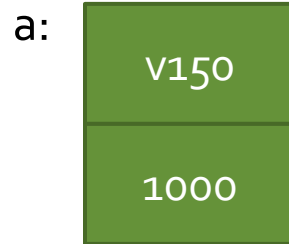
Example: uncontended swap

a:

V150
1000

b:

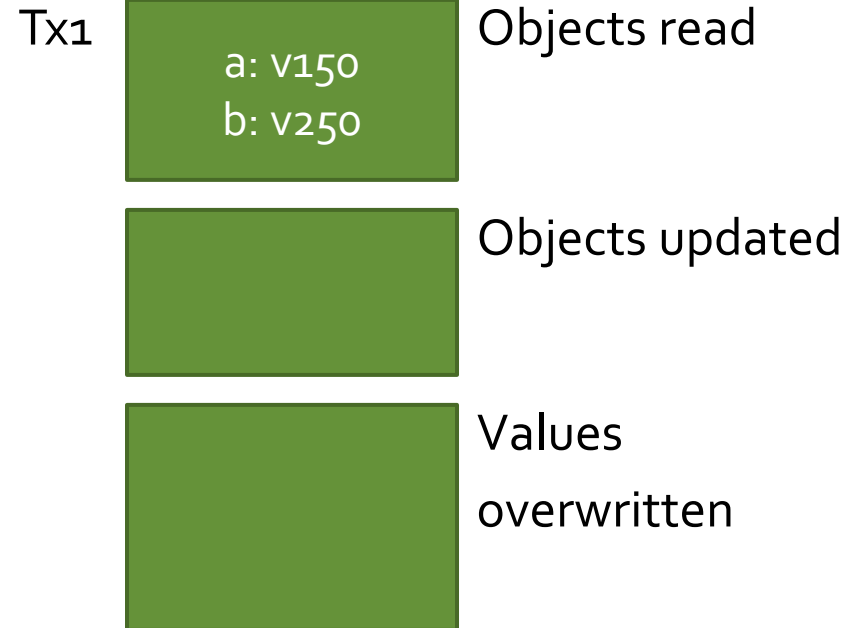| V250 |
|------|
| 2000 |

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```

Tx1

Objects read

Objects updated

Values
overwritten

Example: uncontended swap

a:

v150
1000

b:

v250
2000

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```
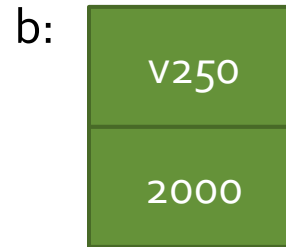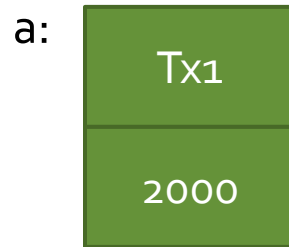
Tx1

a: v150

Objects read

Objects updated

Values overwritten

Example: uncontended swap

a:
v150

1000

b:
v250
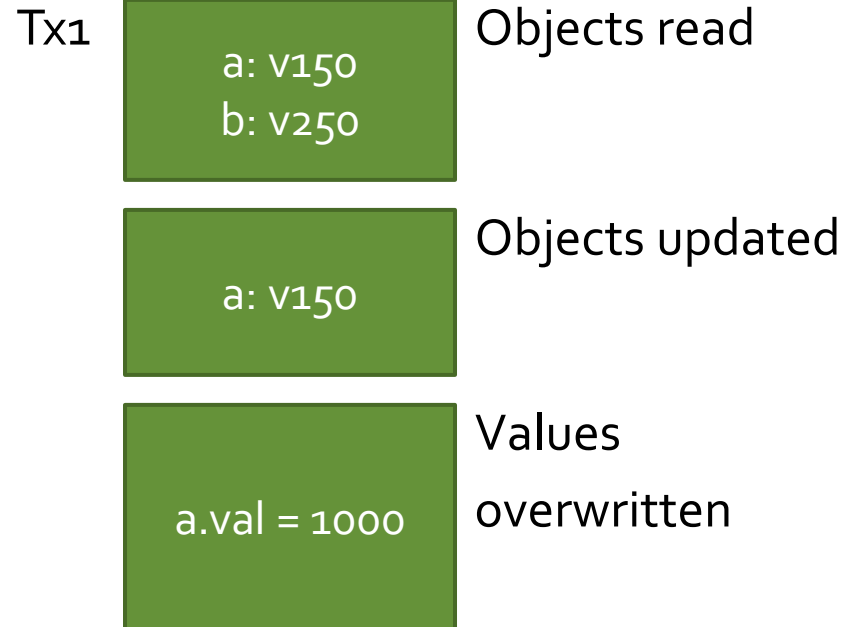
2000

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```

Tx1

a: v150
b: v250

Objects read

Objects updated

Values
overwritten

Example: uncontended swap

a:
Tx1
2000

b:
v250
2000

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```
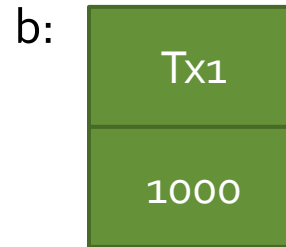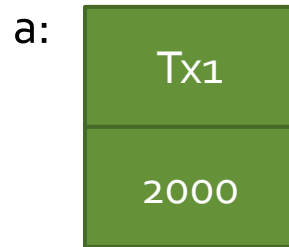
Tx1

a: v150 b: v250

Objects read

a: v150

Objects updated

a.val = 1000

Values overwritten

Example: uncontended swap

a:

Tx1
2000

b:

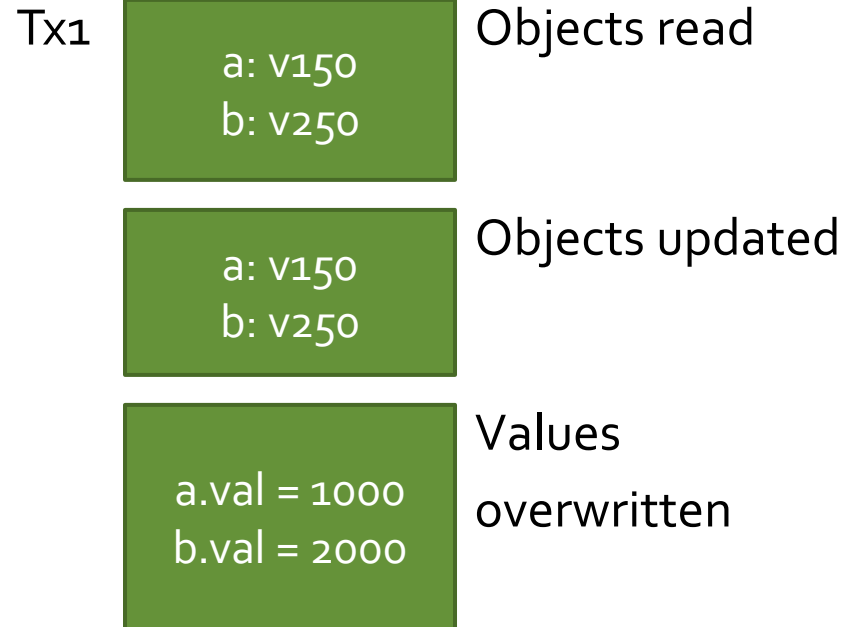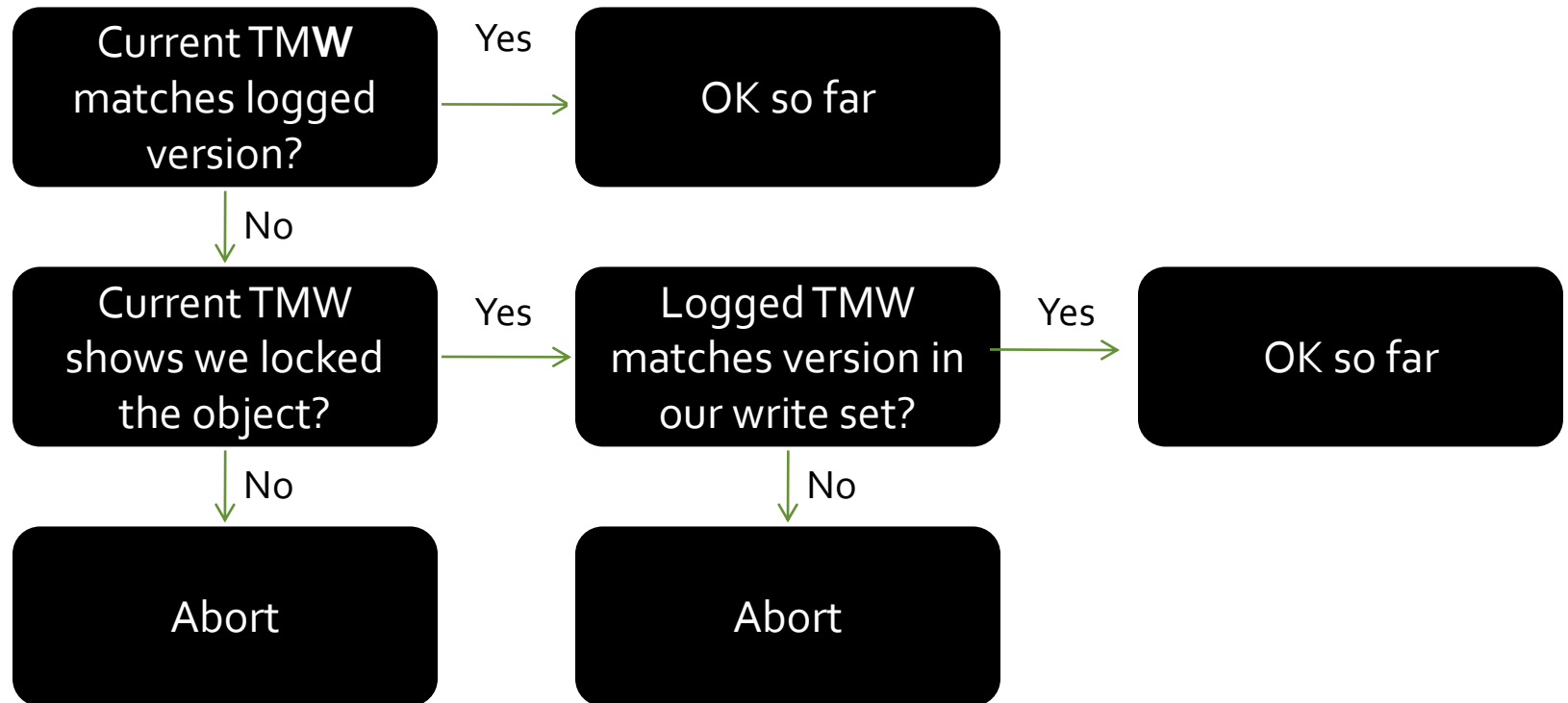| Tx1 |
| --- |
| 1000 |

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```
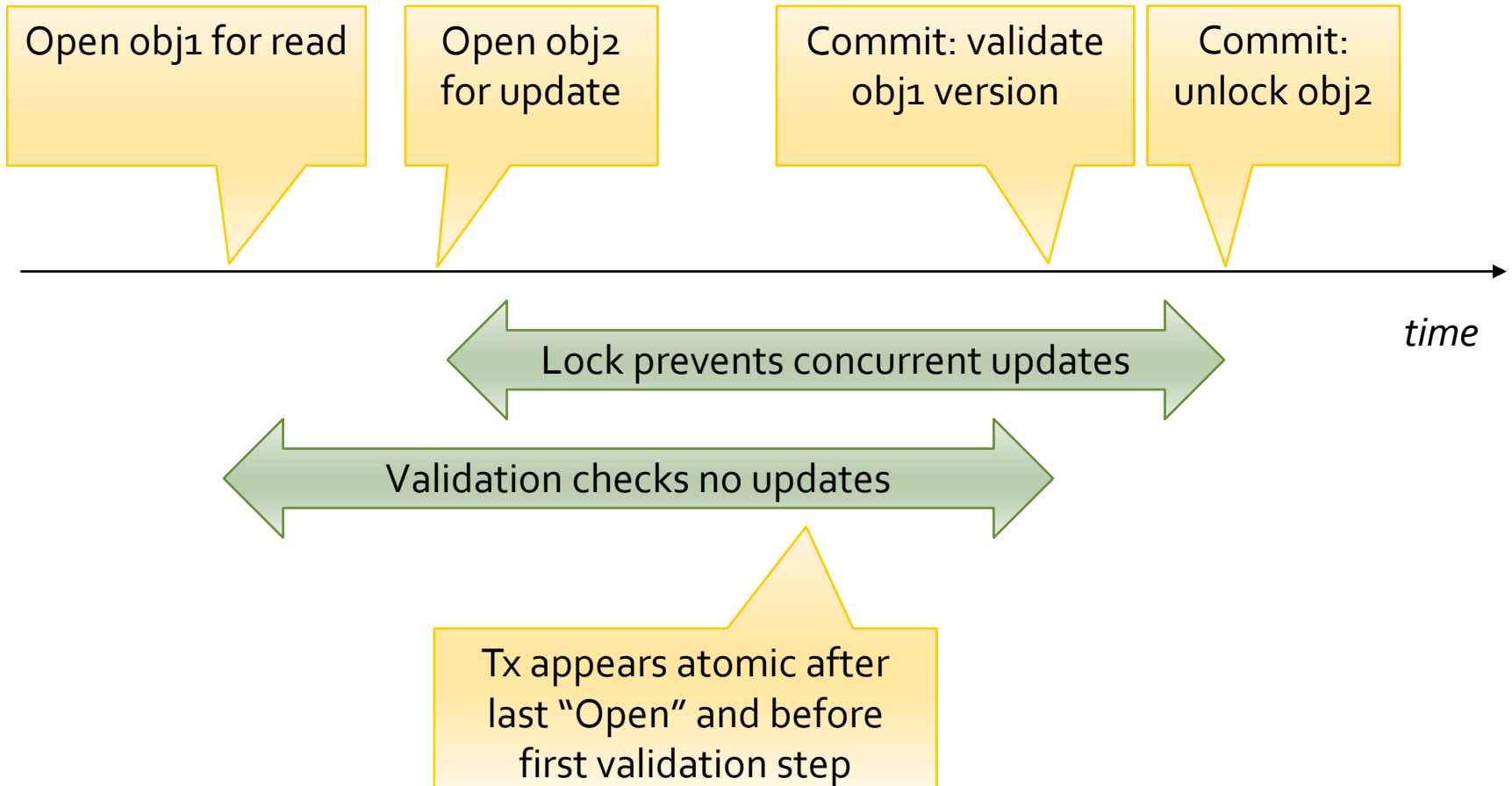
Tx1

a: v150 b: v250

Objects read

a: v150 b: v250

Objects updated

a.val = 1000 b.val = 2000

Values overwritten

Commit in Bartok-STM

Iterate over the read set:

Current TM**W** matches logged version?
— Yes → OK so far
— No ↓

Current TMW shows we locked the object?
— Yes → Logged TMW matches version in our write set?
— No ↓ Abort

Logged TMW matches version in our write set?
— Yes → OK so far
— No ↓ Abort

Correctness sketch

Open obj1 for read

Open obj2 for update

Commit: validate obj1 version

Commit: unlock obj2

time

Lock prevents concurrent updates

Validation checks no updates

Tx appears atomic after last "Open" and before first validation step

Example: uncontended swap

a: Tx ... 2000

b: Tx1 ... 1000

"We locked the object…"

Tx1 — Objects read
a: ...
b: ...

"…and no-one else got there first!" — ...dated
a: ...
b: v250

Values overwritten
a.val = 1000
b.val = 2000

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```
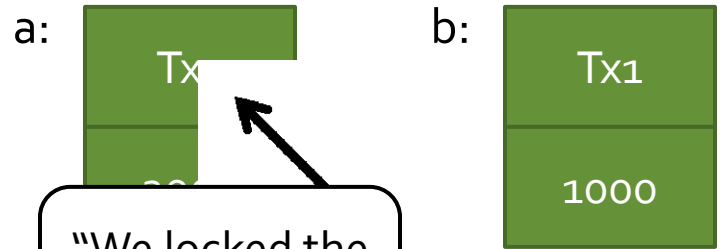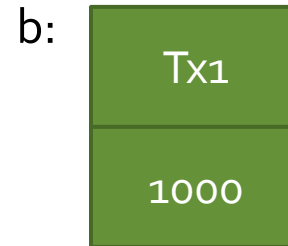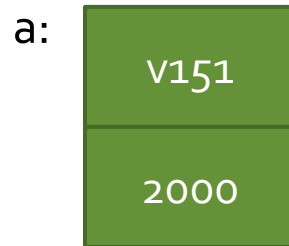
Example: uncontended swap

a:
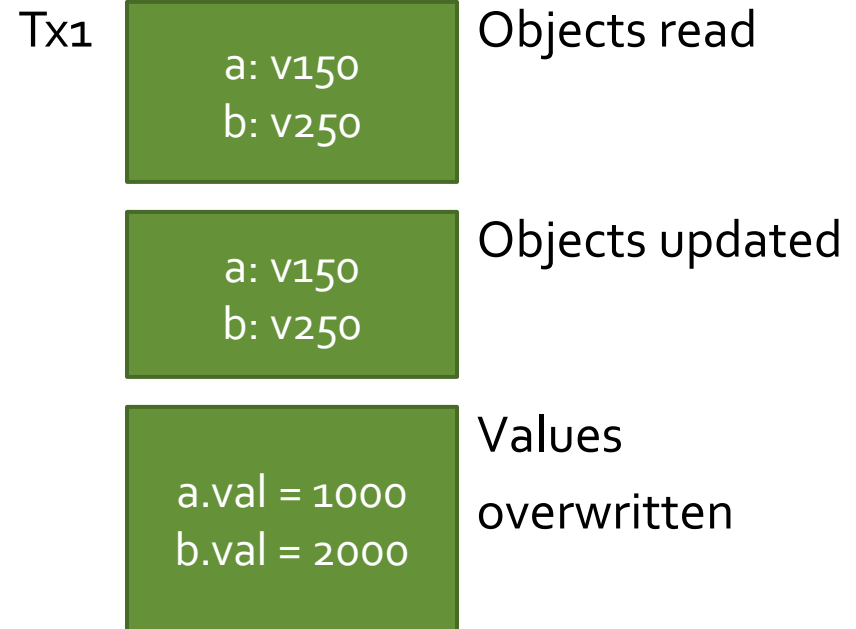| v151 |
|---|
| 2000 |

b:
Tx1
1000

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```

Tx1

a: v150 b: v250	Objects read

a: v150 b: v250	Objects updated

a.val = 1000 b.val = 2000	Values overwritten

Example: uncontended swap

a:

v151
2000

b:

v251
1000

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```
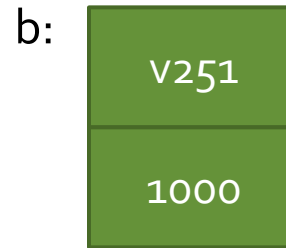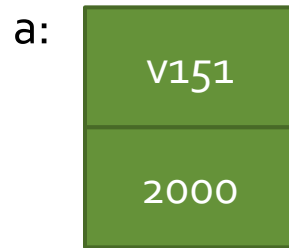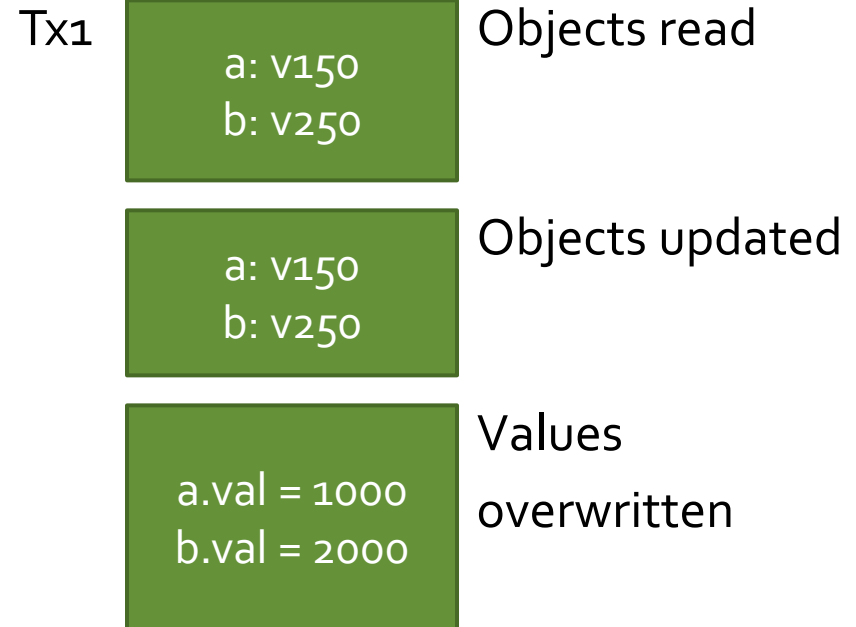
Tx1

| a: v150
b: v250 | Objects read |

| a: v150
b: v250 | Objects updated |

| a.val = 1000
b.val = 2000 | Values overwritten |

Example: uncontended swap

a:

V151
2000

b:

V251
1000

```
void Swap(int *a, int *b) {
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```

Tx-tx interaction in Bartok-STM

- Read-read: no problem, both readers see the same version number and verify it at commit time

- Read-write: reader sees that the writer has the object locked. Reader always defers to writer

- Write-write: competition for lock serializes writers (drop locks, then spin to avoid deadlock)

Taxonomy: consistency during tx

- Gold standard:
  - During execution a transaction runs against a consistent view of memory
  - Won't be "tricked" into looping, etc.
  - "Opacity"
- What are the advantages / disadvantages when compared with an implementation giving weaker guarantees?

Taxonomy: lazy/eager versioning

- We need some way to manage the tentative updates that a transaction is making
    - Where are they stored?
    - How does the implementation find them (so a transaction's read sees an earlier write)?
- Lazy versioning: only make "real" updates when a transaction commits
- Eager versioning: make updates as a transaction runs, roll them back on abort
- What are the advantages, disadvantages?

Taxonomy: lazy/eager conflict detection

- We need to detect when two transactions conflict with one another

- Lazy conflict detection: detect conflicts at commit time

- Eager conflict detection: detect conflicts as transactions run

- Again, what are the advantages, disadvantages?

Taxonomy: word/object based

- What granularity are conflicts detected at?

- Object-based:
  - Access to programmer-defined structures (e.g. objects)

- Word-based:
  - Access to words (or sets of words, e.g. cache lines)
  - Possibly after mapping under a hash function

- What are the advantages and disadvantages of these approaches?

Bartok-STM

- Designed to work well on low-contention workloads
  - Eager version management to reduce commit costs
  - Eager locking to support eager version management
- Primitives do not guarantee that transactions see a consistent view of the heap while running
  - Can be sandboxed in managed code...
  - ...harder in native code
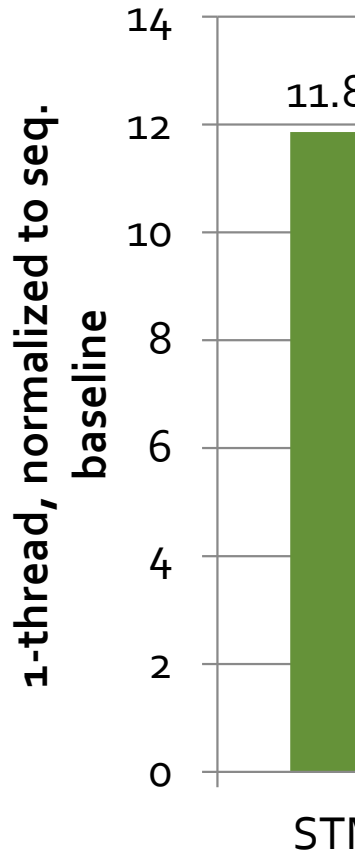
Performance figures depend on…

- **Workload** : What do the atomic blocks do? How long is spent inside them?

- **Baseline implementation:** Mature existing compiler, or prototype?

- **Intended semantics:** Support static separation? Violation freedom (TDRF)?

- **STM implementation:** In-place updates, deferred updates, eager/lazy conflict detection, visible/invisible readers?

- **STM-specific optimizations:** e.g. to remove or downgrade redundant TM operations

- **Integration:** e.g. dynamically between the GC and the STM, or inlining of STM functions during compilation

- **Implementation effort:** low-level perf tweaks, tuning, etc.

- **Hardware:** e.g. performance of CAS and memory system

Labyrinth
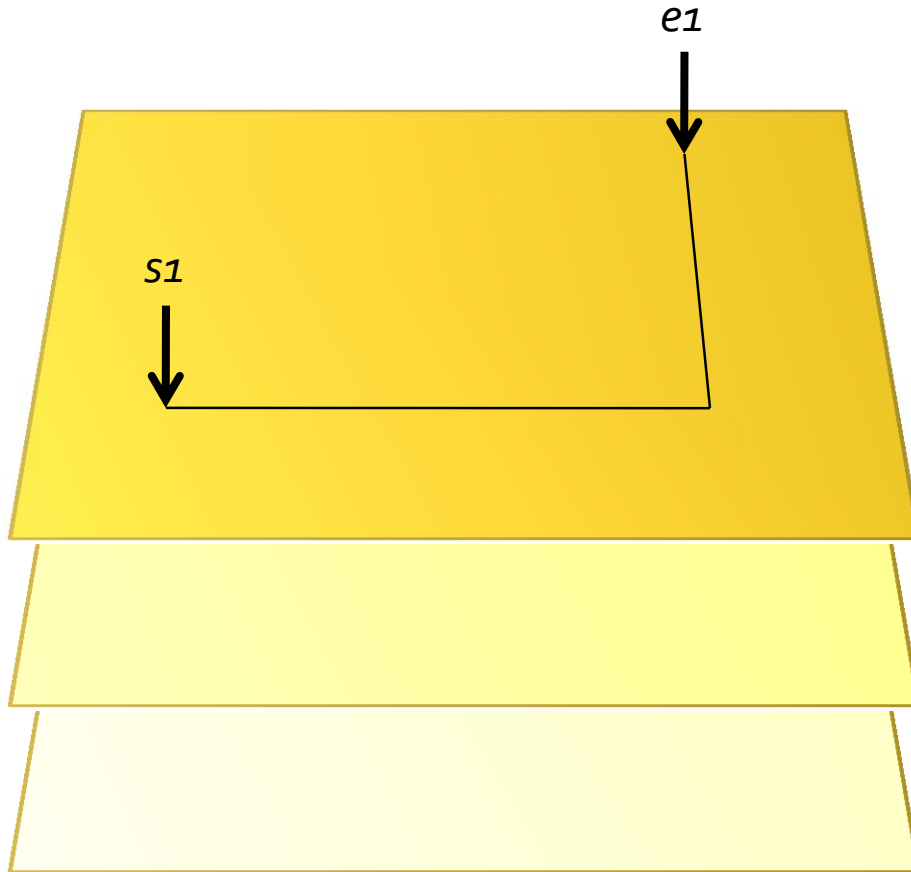
- STAMP v0.9.10

- 256x256x3 grid

- Routing 256 paths

- Almost all execution inside atomic blocks

- Atomic blocks can attempt 100K+ updates

- C# version derived from original C

- Compiled using Bartok, whole program mode, C# -> x86 (~80% perf of original C with VS2008)

- Overhead results with Core2 Duo running Windows Vista

"STAMP: Stanford Transactional Applications for Multi-Processing"
Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun , IISWC 2008

Sequential overhead

STM implementation supporting static separation
In-place updates
Lazy conflict detection
Per-object STM metadata
Addition of read/write barriers before accesses
Read: log per-object metadata word
Update: CAS on per-object metadata word
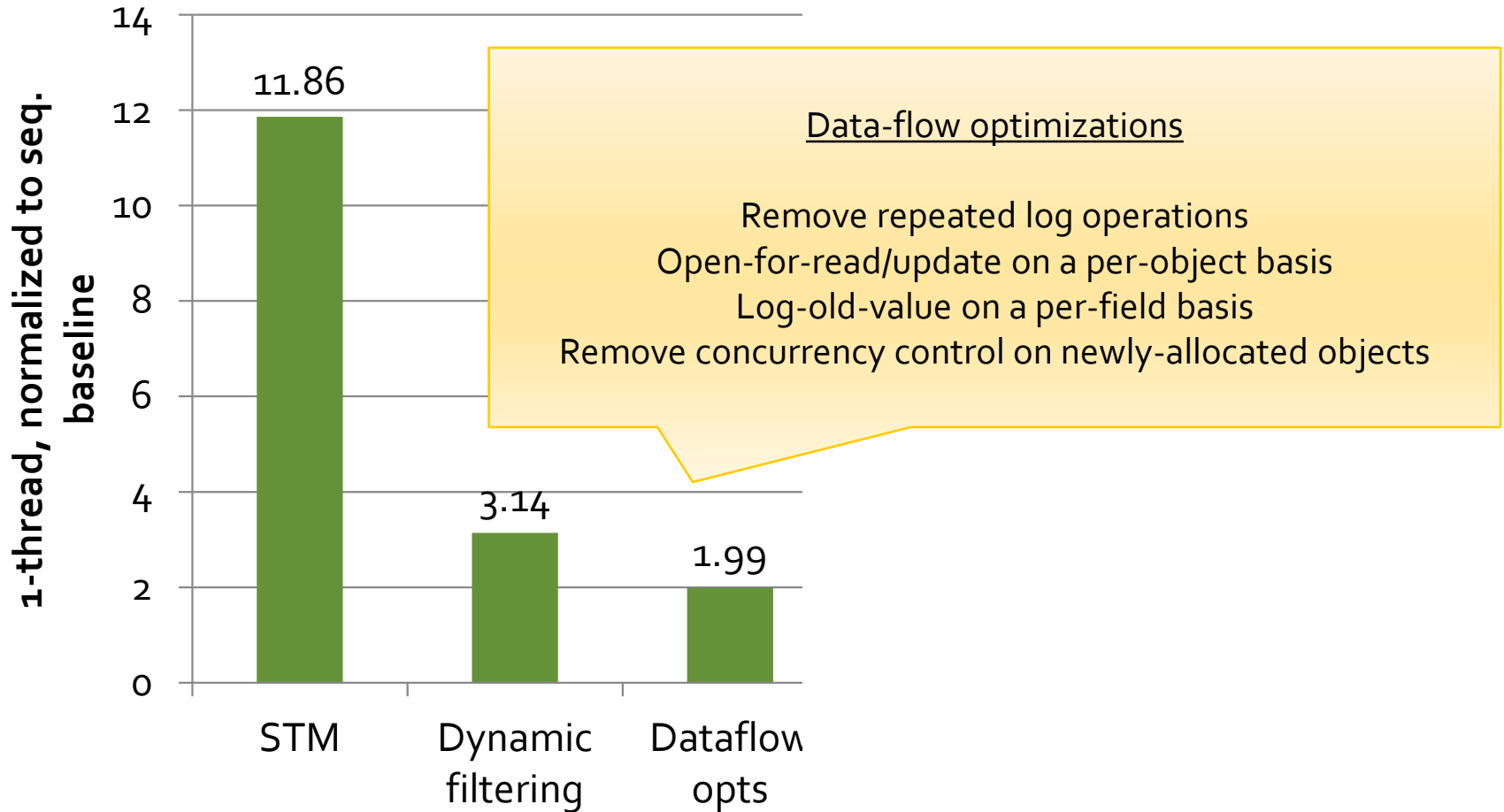Update: log value being overwritten

Sequential overhead

1-thread, normalized to seq. baseline
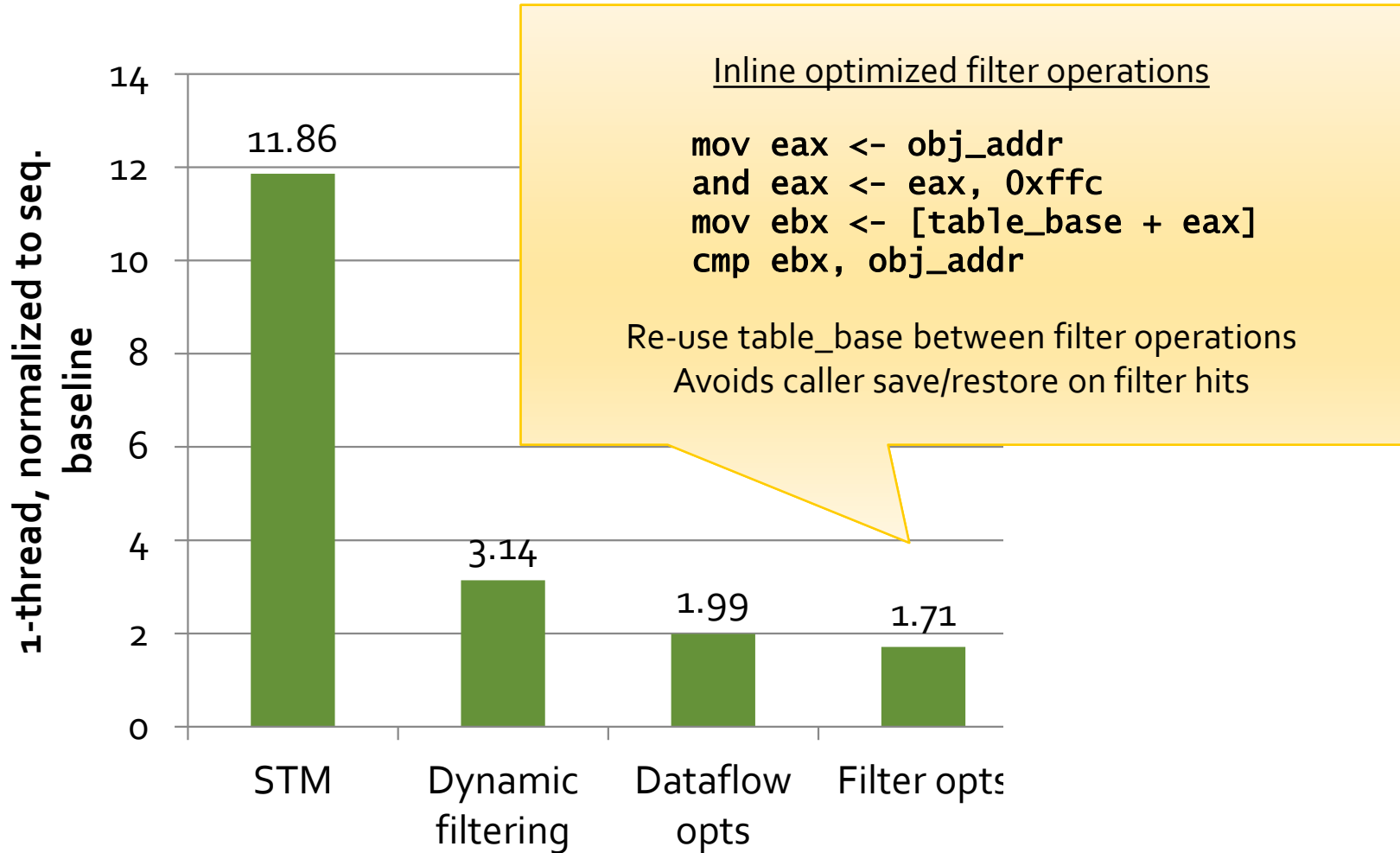
11.86

3.14

STM
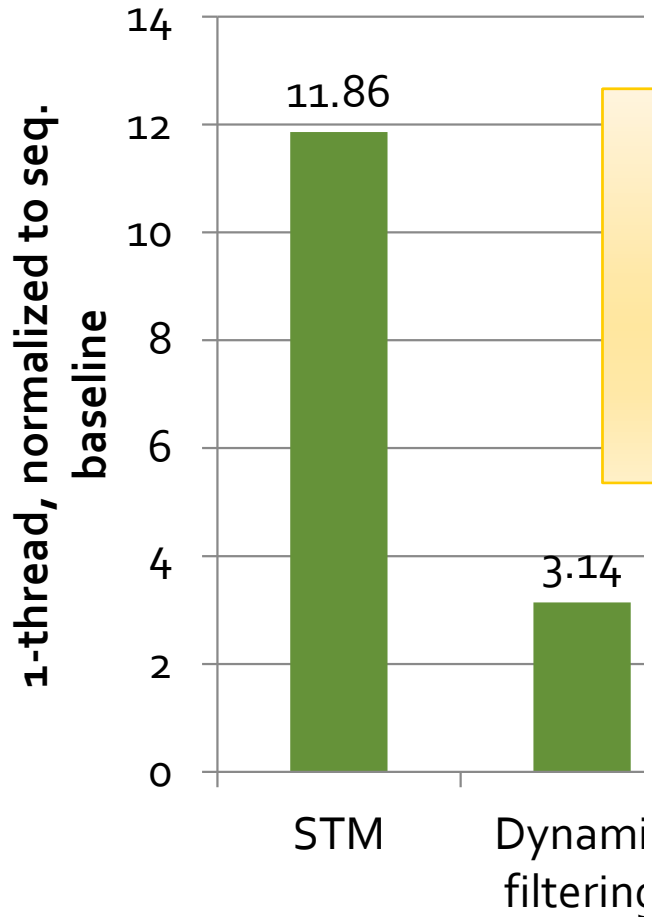
Dynamic filtering

Dynamic filtering to remove redundant logging

Log size grows with #locations accessed
Consequential reduction in validation time
1st level: per-thread hashtable (1024 entries)
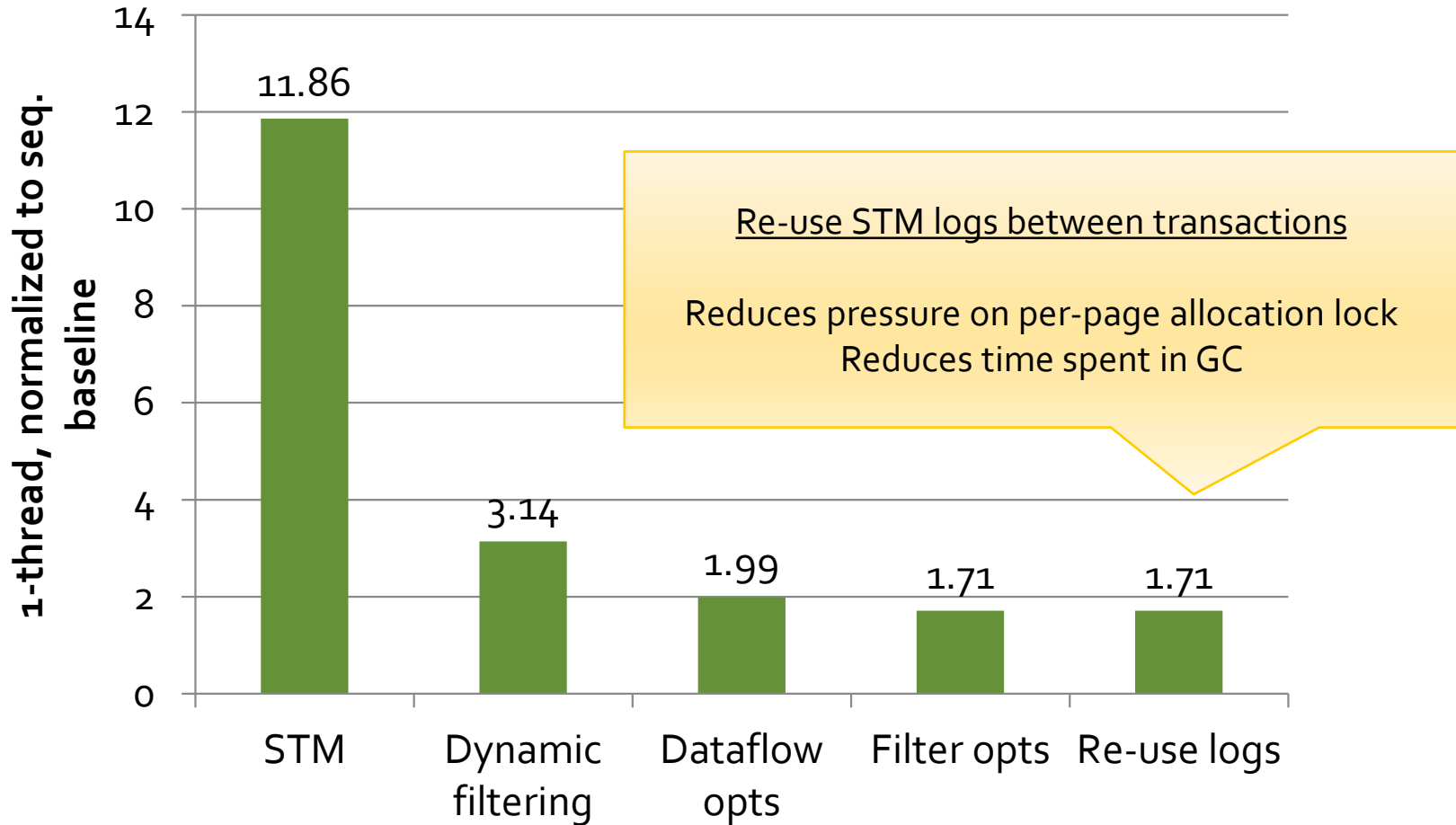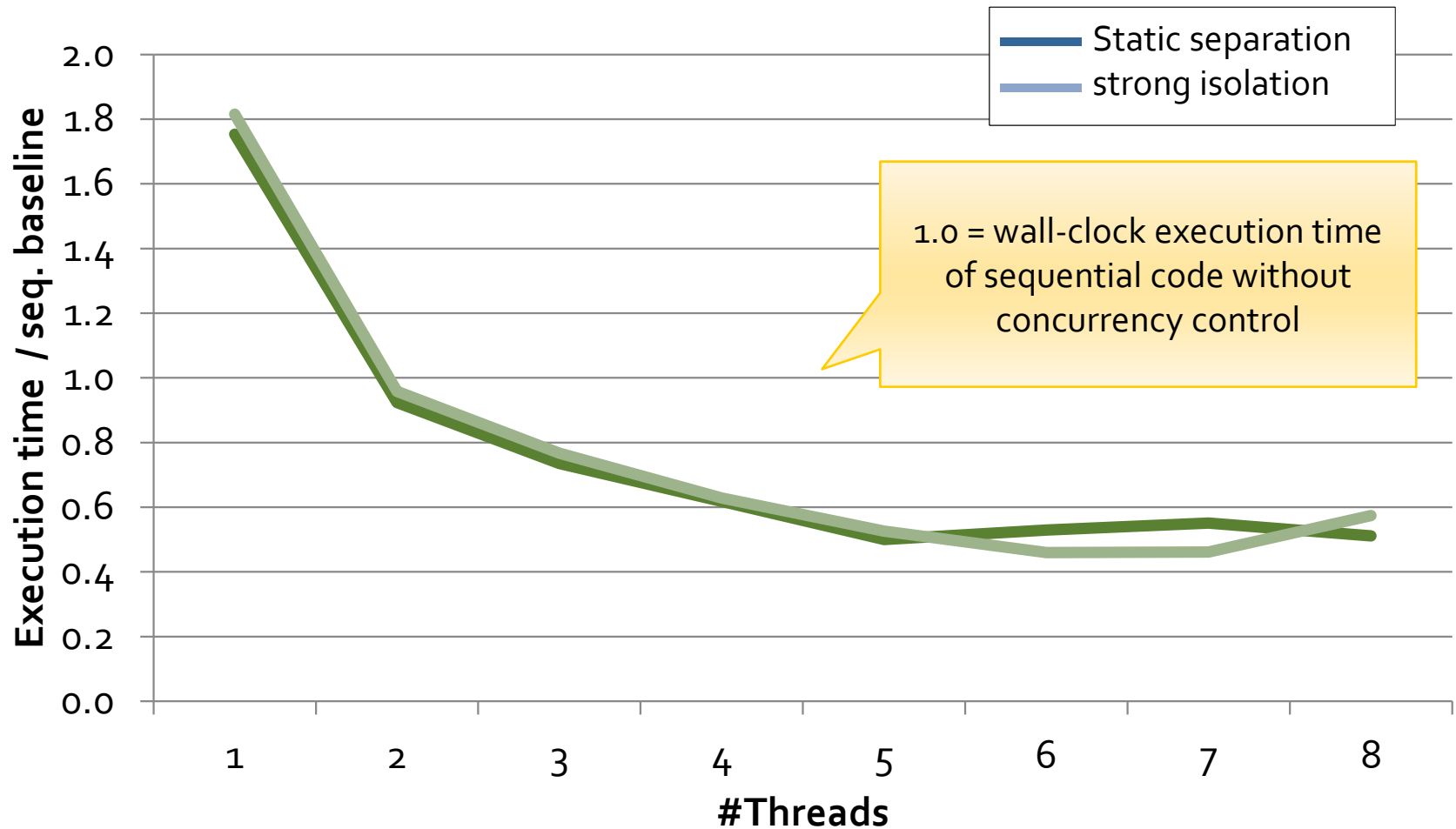2nd level: per-object bitmap of updated fields

Sequential overhead

Data-flow optimizations

Remove repeated log operations
Open-for-read/update on a per-object basis
Log-old-value on a per-field basis
Remove concurrency control on newly-allocated objects

Sequential overhead

Inline optimized filter operations

```
mov eax <- obj_addr
and eax <- eax, 0xffc
mov ebx <- [table_base + eax]
cmp ebx, obj_addr
```

Re-use table_base between filter operations
Avoids caller save/restore on filter hits

Sequential overhead

Re-use STM logs between transactions

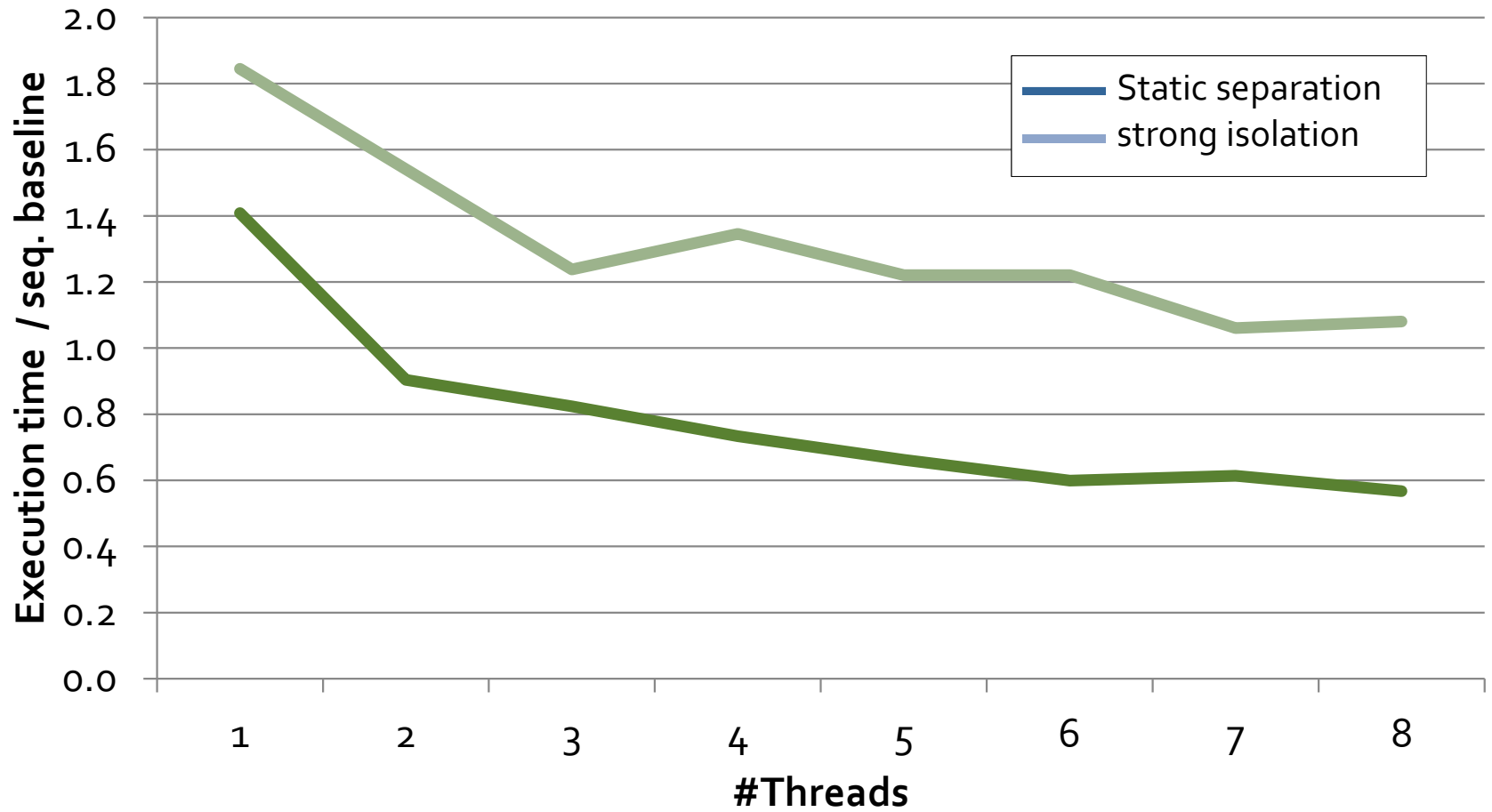Reduces pressure on per-page allocation lock
Reduces time spent in GC

Scaling – Labyrinth

Scaling – Genome

TM programming models

What we want

Libraries build layered concurrency abstractions

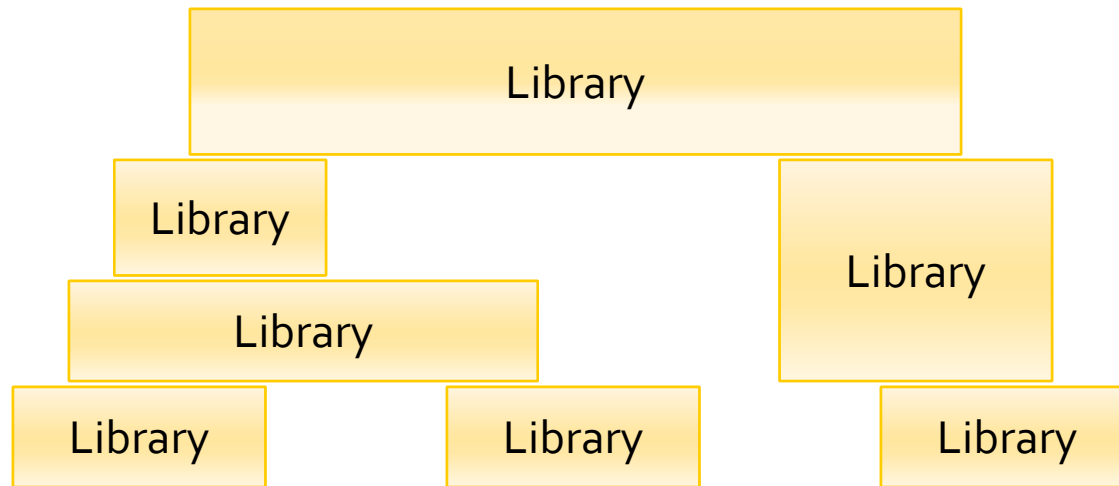Library

Library

Library

Library

Library

Library

Concurrency primitives
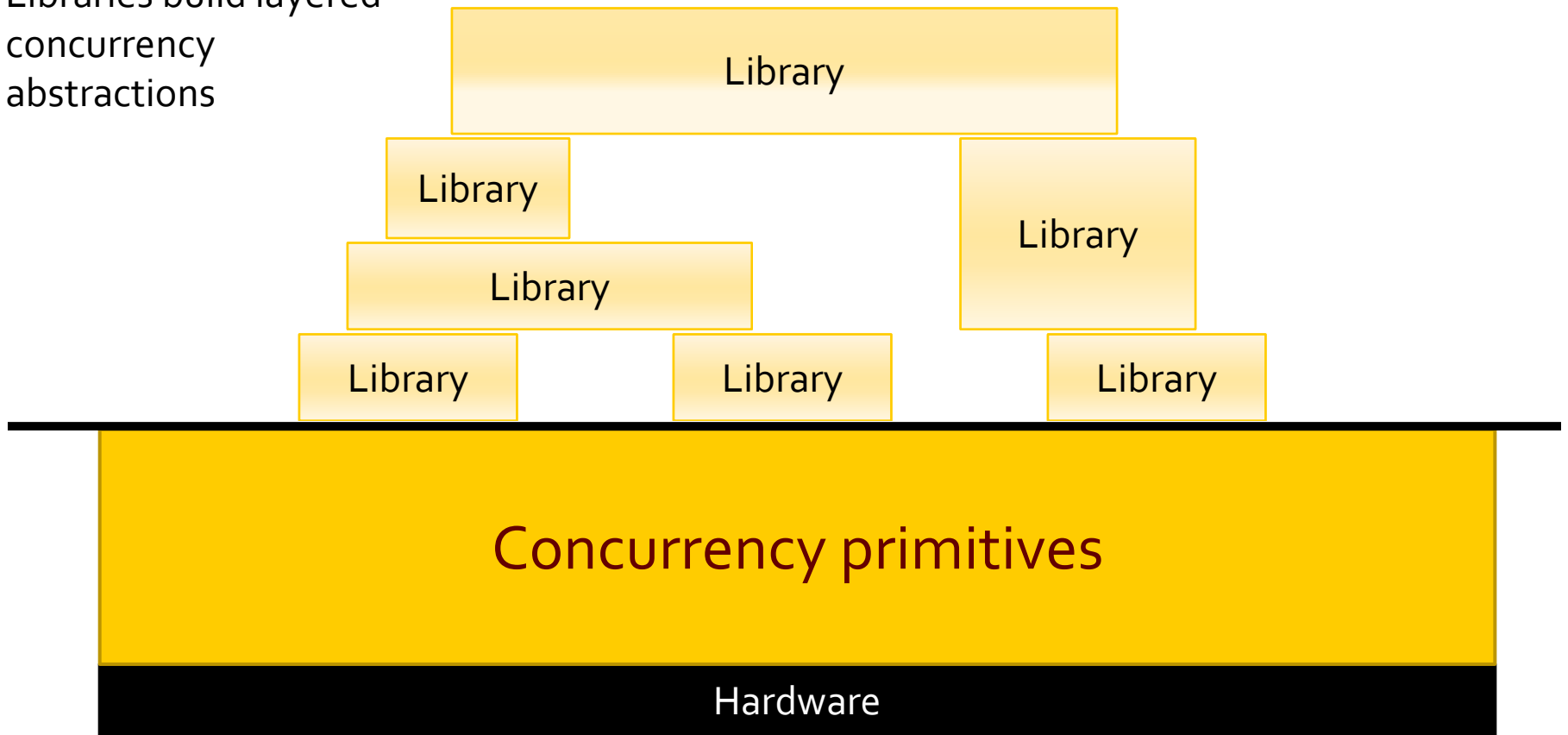
Hardware

What we have

Locks and condition variables
(a) are hard to use and
(b) do not compose

Atomic blocks

Library

Library

Library

Library

Library

Library

Library
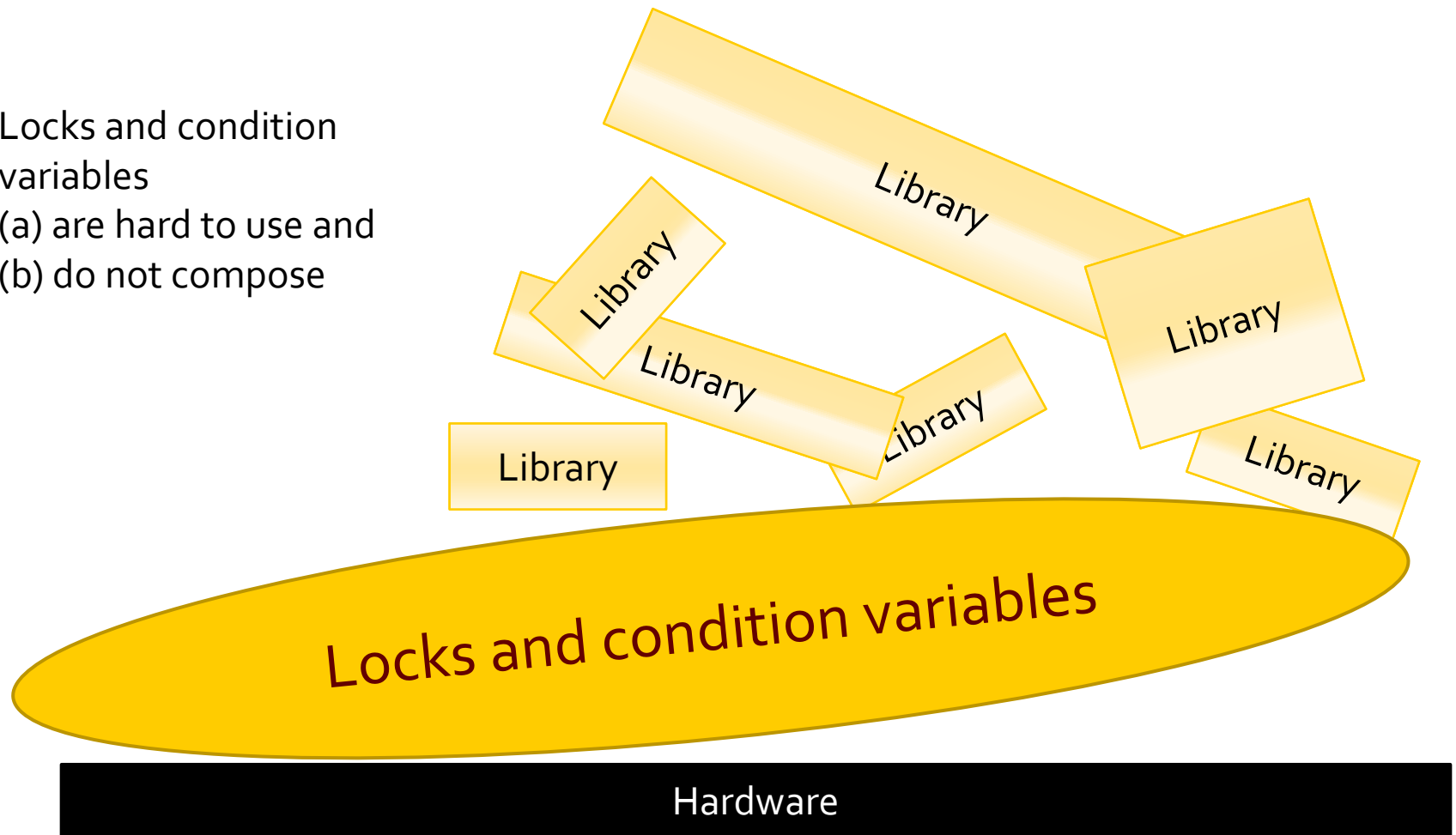
Atomic blocks built over transactional memory. In Haskell: 3 primitives: atomic, retry, orElse

Hardware

Atomic memory transactions

```
Item PopLeft() {
        atomic { … sequential code … }
}
```

Like database transactions

- To a first approximation, just write the sequential code, and wrap **atomic** around it
- All-or-nothing semantics: **Atomic** commit
- Atomic block executes in **Isolation**
- Cannot deadlock (there are no locks!)
- Atomicity makes error recovery easy (e.g. exception thrown inside the **PopLeft** code)

AcID

Atomic blocks compose (locks do not)

```
void GetTwo() {
        atomic {
                i1 = PopLeft();
                i2 = PopLeft();
        }
        DoSomething( i1, i2 );
}
```

- Guarantees to get two consecutive items
- PopLeft() is unchanged
- Cannot be achieved with locks (except by breaking the PopLeft abstraction)

Composition is THE way we build big programs that work

Blocking: how does PopLeft wait for data?

```
Item PopLeft() {
        atomic {
                if (leftSentinel.right==rightSentinel)  {
                        retry;
                } else { …remove item from queue… }
}        }
```

- **retry** means "abandon execution of the atomic block and re-run it (when there is a chance it'll complete)"
- No lost wake-ups
- No consequential change to GetTwo(), *even though GetTwo must wait for there to be **two** items in the queue*

Choice: waiting for either of two queues

```
void GetEither() {
        atomic {

                do { i = Q1.Get(); }
                orelse { i = Q2.Get(); }

                R.Put( i );
}        }
```

Q1   Q2

R
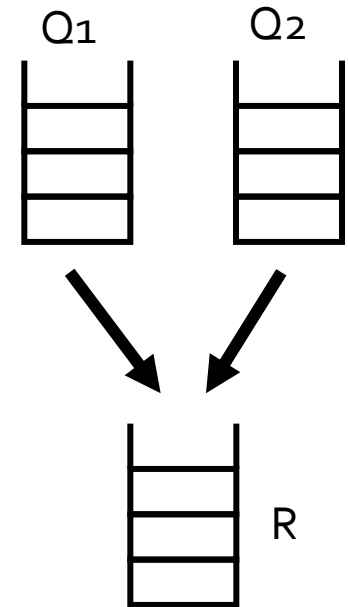
- **do** {...this...} **orelse** {...that...} tries to run "this"
- If "this" retries, it runs "that" instead
- If both retry, the do-block retries. GetEither() will thereby wait for there to be an item in *either* queue

Programming with atomic blocks

With locks, you think about:

- Which lock protects which data? What data can be mutated when by other threads? Which condition variables must be notified when?

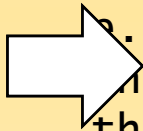- None of this is explicit in the source code

With atomic blocks you think about

- What are the **invariants** (e.g. the tree is balanced)?

- Each atomic block maintains the invariants

- **Purely sequential reasoning** within a block, which is dramatically easier

- Much easier setting for static analysis tools

Atomic blocks

```
Class Q {
  QElem l
  QElem r

  void pu
    atomi
      QEl
      e.r
      .l
      thi
      thi
    }
  }

  ...
}
```

```
Class Q {
  QElem leftSentinel;
  QElem rightSentinel;

  void pushLeft(int item) {
    do {
      TxStart();
      QElem e = new QElem(item);
      TxWrite(&e.right, TxRead(&this.leftSentinel.right));
      TxWrite(&e.left, this.leftSentinel);
      TxWrite(&TxRead(&this.leftSentinel.right).left, e);
      TxWrite(&this.leftSentinel.right, e);
    } while (!TxCommit());
  }

  ...
}
```

Compilation

Source to bytecode compiler; typically "csc" in C#, "javac" for Java

Bytecode-to-native compiler; JIT or traditional compilation

Source code

Bytecode

Native code

Why divide things this way?

- Little information loss from source code to bytecode

- Source-to-bytecode works a file at a time, bytecode-to-native can see the whole program (or, at least, see all of the parts needed so far in execution)

- Lower level transformations possible at bytecode-to-native

- Integration between the STM and other parts of the runtime system

```
void Swap(Pair p) {
  try {
    va = p.a;
    vb = p.b;
    p.a = vb;
    p.b = va;
  } catch (AtomicException) {
  }
}
```

Boilerplate around transactions

```
void Swap(Pair p) {
   do {
    done = true;
    try {
      try {
        tx = StartTx();
        va = p.a;
        vb = p.b;
        p.a = vb;
        p.b = va;
      } finally {
        CommitTx();
      }
    } catch (TxInvalid) {
      done = false;
    }
  } while (!done);
}
```

Keep running the atomic block in a fresh tx each time

Commit (on normal or exn exit)

Commit fails by raising a TxInvalid exception; re-execute

(I'm using source code examples for clarity; in reality this would be in the compiler's internal intermediate code)

Naïve expansion of data accesses
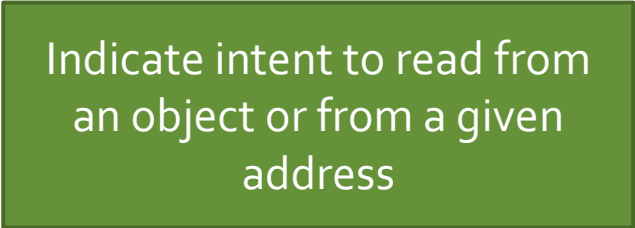
```
void Swap(Pair p) {
   do {
    done = true;
    try {
      try {
        tx = StartTx();
        TxWrite(tx, &va, TxRead(tx, &p.a));
        TxWrite(tx, &vb, TxRead(tx, &p.b));
        TxWrite(tx, &p.a, TxRead(tx, &vb));
        TxWrite(tx, &p.b, TxRead(tx, &va));
      } finally {
        CommitTx();
      }
    } catch (TxInvalid) {
      done = false;
    }
  } while (!done);
}
```

What are the problems here with STM?

- Using the STM for thread-private local variables

- Repeatedly mapping from addresses to concurrency control info

- Duplicating concurrency control work if it's implemented at a per-object granularity
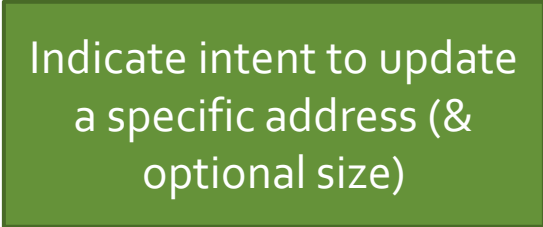
Decomposed STM primitive API

- OpenForRead(tx, obj)
- OpenForRead(tx, addr)
- OpenForUpdate(tx, obj)
- OpenForUpdate(tx, addr)

- LogForUndo(tx, addr)

Indicate intent to read from an object or from a given address

Indicate intent to update a specific address (& optional size)

Using the decomposed API

```
x = p.a;
```

→

```
OpenForRead(tx, p);
x = p.a;
```

```
p.b = y;
```

→

```
OpenForUpdate(tx, p);
LogForUndo(tx, &p.b);
p.b = y;
```

Implementation using decomposed API

```
...
OpenForUpdate(tx, p);
~~OpenForRead(tx, p);~~
va = p.a;
~~OpenForRead(tx, p);~~
vb = p.b;
~~OpenForUpdate(tx, p);~~
LogForUndo(tx, &p.a);
p.a = vb;
~~OpenForUpdate(tx, p);~~
LogForUndo(tx, &p.b);
p.b = va;
...
```

Always need update
access: get it first

Second OpenForRead
made unnecessary by first

Second OpenForUpdate
made unnecessary by first

Improved expansion of data accesses

```
void Swap(Pair p) {
    do {
     done = true;
     try {
       try {
         tx = StartTx();
         OpenForUpdate(tx, p);
         va = p.a;
         vb = p.b;
         LogForUndo(tx, &p.a);
         p.a = vb;
         LogForUndo(tx, &p.b);
         p.b = va;
       } finally {
         CommitTx();
       }
     } catch (TxInvalid) {
       done = false;
     }
   } while (!done);
}
```

Keeping optimizations safe

```
void Clear_tx(Pair p) {
   for (int i = 0; i < 10; i ++) {
      p.a = 10;
      p.b = i;
   }
}
```

Keeping optimizations safe

Expanded with decomposed API operations

```
void Clear_tx(Pair p) {
   for (int i = 0; i < 10; i ++) {
      OpenForUpdate(tx, p);
      LogForUndo(tx, &p.a);
      p.a = 10;
      LogForUndo(tx, &p.b);
      p.b = i;
   }
}
```

Keeping optimizations safe

Hoisting loop-invariant code

```
void Clear_tx(Pair p) {
  p.a = 10;
  for (int i = 0; i < 10; i ++) {
    OpenForUpdate(tx, p);
    LogForUndo(tx, &p.a);
    LogForUndo(tx, &p.b);
    p.b = i;
  }
}
```

Keeping optimizations safe

Introduce dependencies

```
void Clear_tx(Pair p) {
for (int i = 0; i < 10; i ++) {
    tmp1 = OpenForUpdate(tx, p);
    tmp2 = LogForUndo(tx, &p.a) <tmp1>;
    p.a = 10 <tmp2>;
    tmp3 = LogForUndo(tx, &p.b) <tmp1>;
    p.b = i <tmp3>;
  }
}
```

Keeping optimizations safe

Transformations must respect dependencies

```
void Clear_tx(Pair p) {
  tmp1 = OpenForUpdate(tx, p);
  tmp2 = LogForUndo(tx, &p.a) <tmp1>;
  tmp3 = LogForUndo(tx, &p.a) <tmp1>;
  p.a = 10 <tmp2>;
  for (int i = 0; i < 10; i ++) {
    p.b = i <tmp3>;
  }
}
```

Condition synchronization

```
atomic {
   buffer.data = 42;
   buffer.full = true;
}
```

```
atomic {
   if (!buffer.full) {
      retry;
   }
   result = buffer.data;
   buffer.full = false;
}
```

This atomic block is only ready to run when buffer.full is true

- Semantically: in STM-Haskell we required the scheduler to only run atomic blocks when they succeed without calling "retry"

Primitive for synchronization

- void WaitTX(tx)

  - Semantically equivalent to AbortTx

  - Implementation may assume caller will immediately re-execute a (deterministic) tx

  - Implementation may introduce a delay to avoid unnecessary spinning

- Intuition:

  - No point re-executing the consumer until the producer has run

Compiling "retry" to "WaitTx"

```
atomic {
  if (!buffer.fu
    retry;
  }
  result = buffe
  buffer.full = 
}
```

```
void Consume(Buffer b) {
    do {
     done = true;
     try {
       try {
         tx = StartTx();
         OpenForRead(tx, b);
         if (!b.full) {
           WaitTx();
         }
         OpenForUpdate(tx, b);
         result = b.data;
         LogForUndo(tx, &b.full);
         b.full = false;
       } finally {
         CommitTx();
       }
     } catch (TxInvalid) {
       done = false;
     }
   } while (!done);
}
```
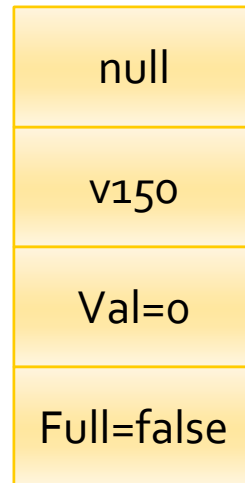
Implementing WaitTx

buffer:

v150
Val=0
Full=false
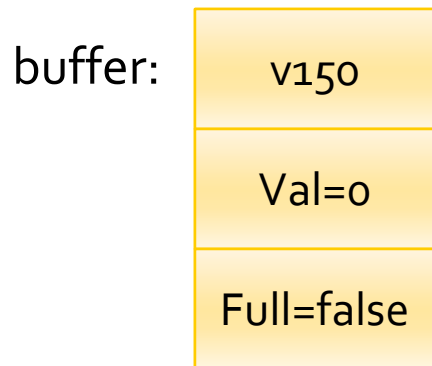
Implementing WaitTx

1. Extend object header with list of waiters

buffer:

null
v150
Val=0
Full=false

Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

buffer:

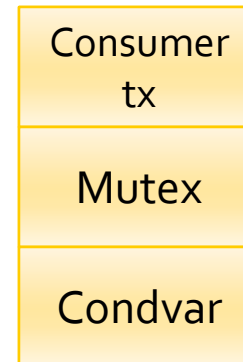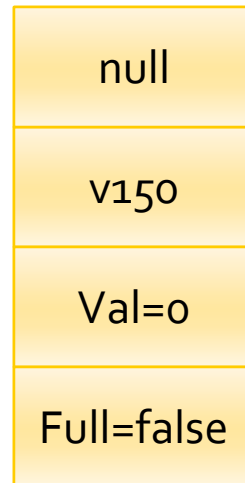| |
|---|
| null |
| v150 |
| Val=0 |
| Full=false |

Consumer tx
Mutex
Condvar

Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set

v150
Val=0
Full=false

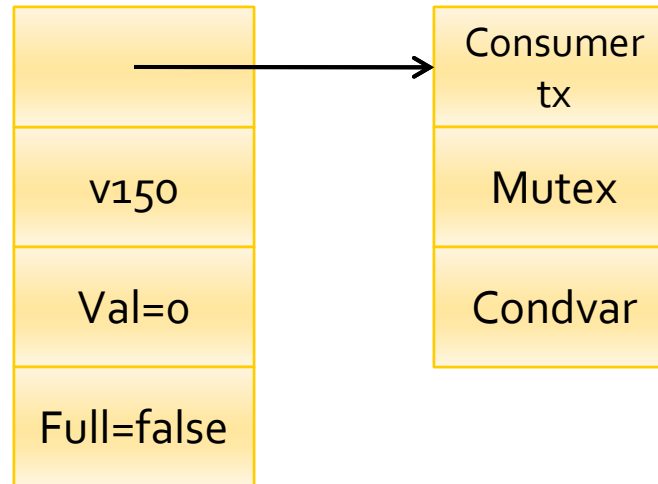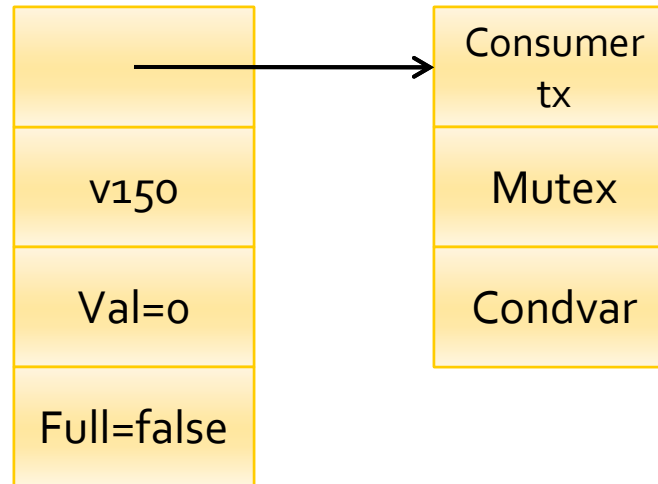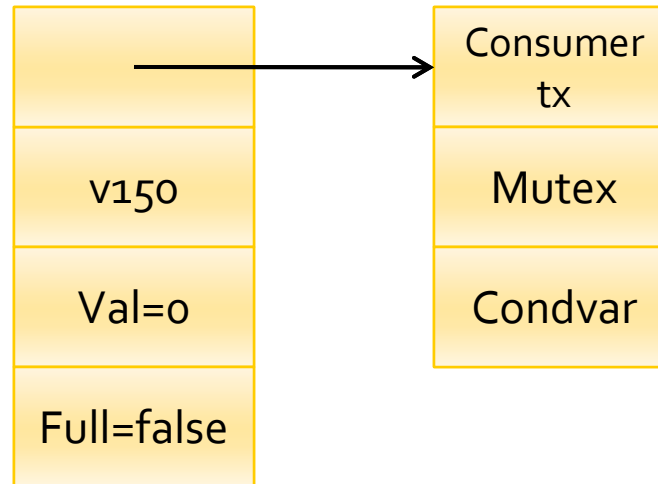| Consumer tx |
|-------------|
| Mutex |
| Condvar |

Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set

4. WaitTx validates, locks the mutex, updates its status, blocks

v150
Val=0
Full=false

Consumer tx
Mutex
Condvar

Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set

4. WaitTx validates, locks the mutex, updates its status, blocks

5. CommitTx wakes waiters on objects in its write set

v150
Val=0
Full=false

Consumer tx
Mutex
Condvar

Implementing WaitTx

1. Extend object header with list of waiters

Use "thin locks" style tricks to avoid fixed header word allocation

NB: many-to-many relationship, so probably use separate doubly linked list

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set

4. WaitTx validates, locks the mutex, updates its status, blocks

5. CommitTx wakes waiters on objects in its write set

| v150 |
| Val=0 |
| Full=false |

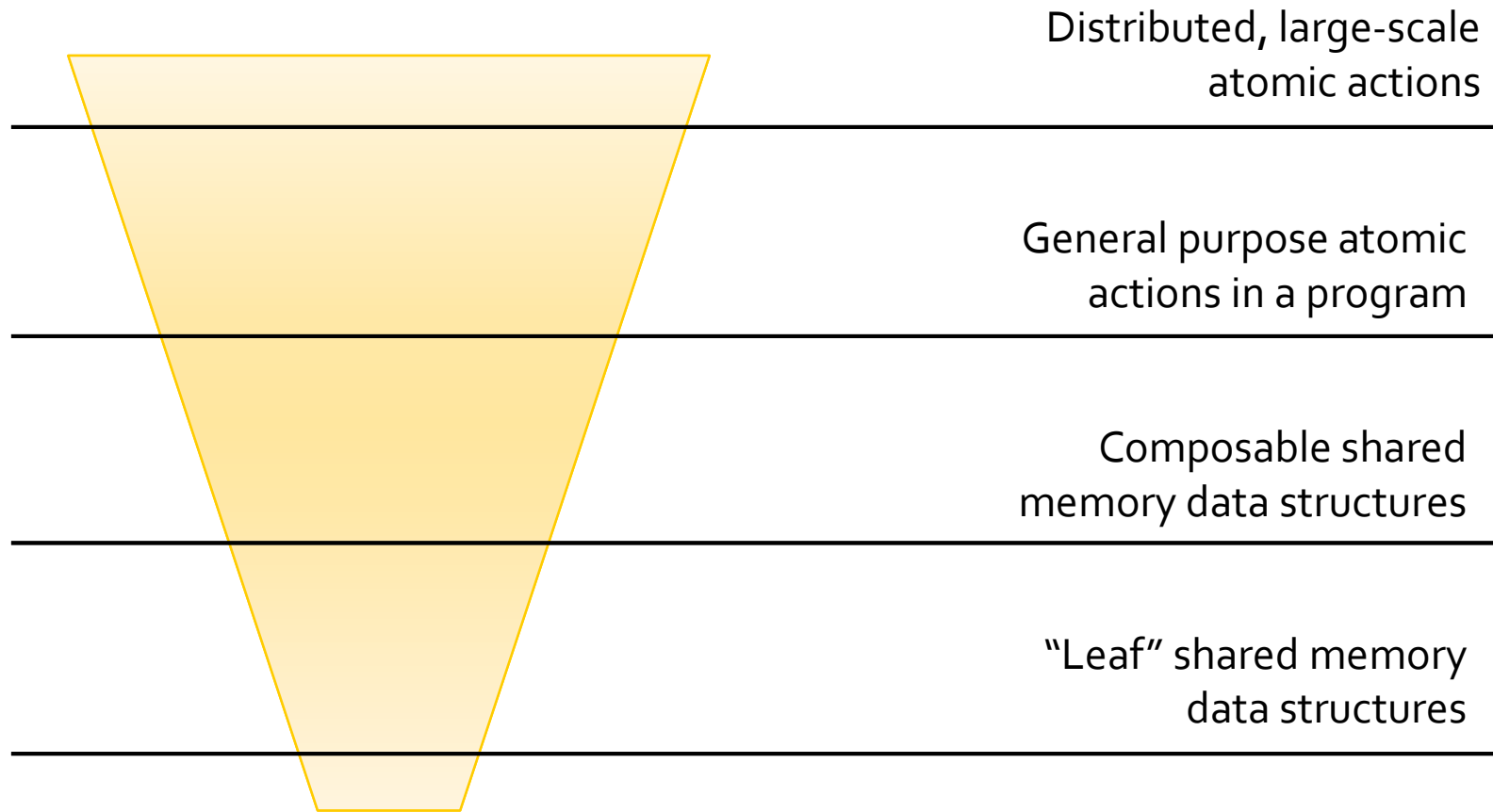| Consumer tx |
| Mutex |
| Condvar |
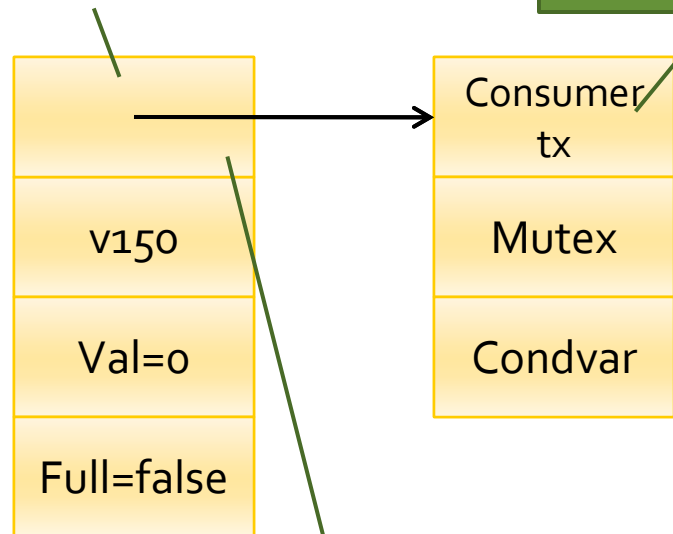
Use latch in the header for concurrency control on the list

Are we done?

- Local variables

- By-ref parameters

- Method calls

- Sandboxing zombie transactions

- IO and other native operations

TM perspectives

Granularity

Distributed, large-scale atomic actions

General purpose atomic actions in a program

Composable shared memory data structures

"Leaf" shared memory data structures

Programming abstraction

Lock elision

The program's semantics is defined using locks. TM is used as an implementation mechanism.

Atomic
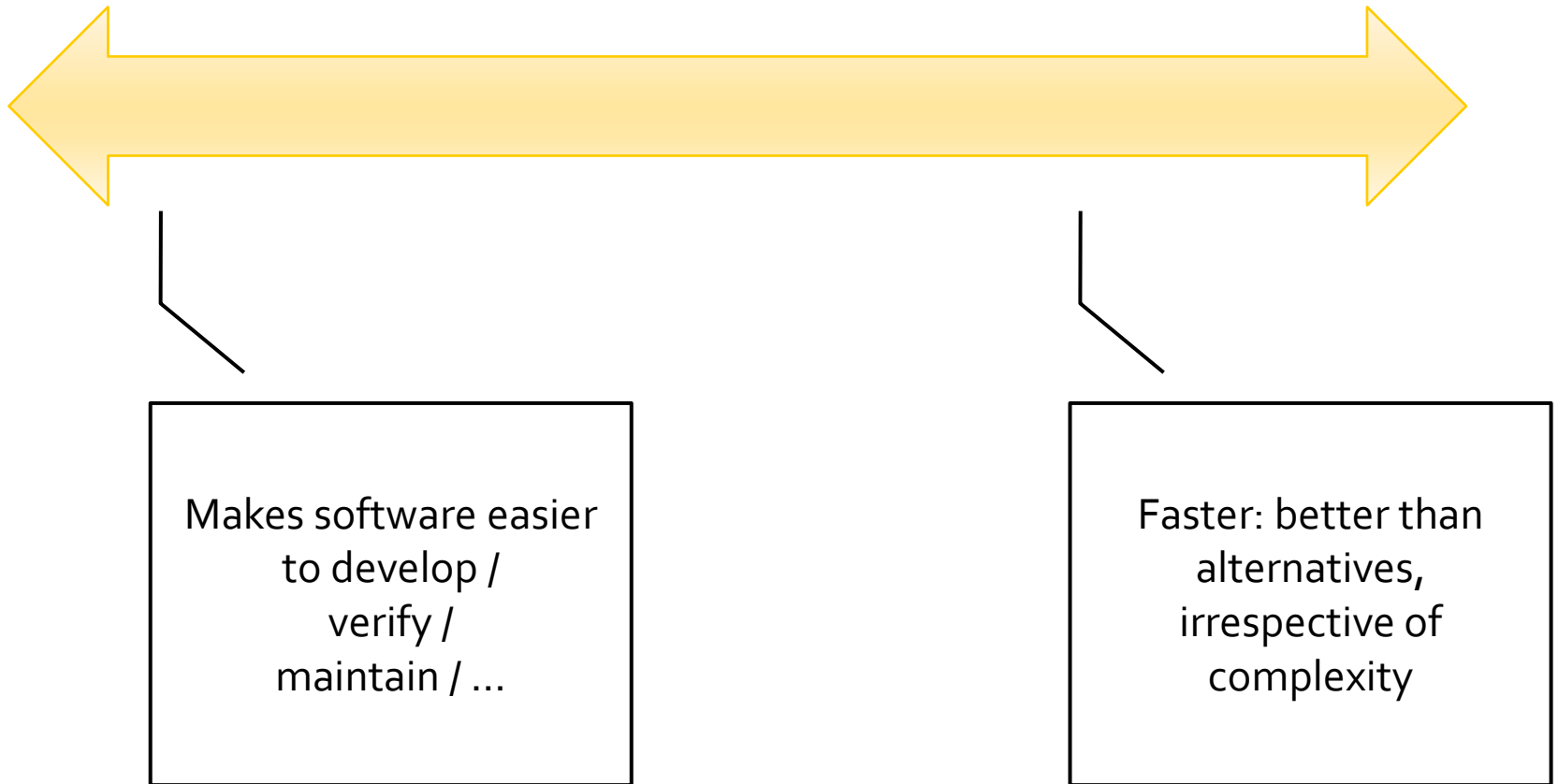
Semantics defined by atomic execution (e.g. "atomic { X }"). Speculation, if used, is abstracted by the implementation.

Speculation

Semantics defined by speculative execution, commit, etc. (either implicitly, or explicitly)

Purpose

Makes software easier to develop / verify / maintain / …

Faster: better than alternatives, irrespective of complexity

Design points that I like

HW DCAS / 3-CAS / …
Granularity: leaf data structures
Abstraction: atomic multi-word CAS
Purpose: faster

HTM with limited guarantees (~ASF)
Granularity: leaf data structures
Abstraction: short transactions
Purpose: faster

Static separation (e.g., STM-Haskell)
Granularity: composable data structures
Abstraction: atomic actions
Purpose: easier, decent perf

Design points I am sceptical about

Speculative lock elision on general-purpose s/w

"atomic" blocks over normal data in a high-level language (C#/Java)

(prove me wrong, I would like either of these to work!)

What do we care about?

Ease to write

Correctness

When can it be used?

How fast is it?

How well does it scale?