Multicore Programming

Transactional memory

29 Nov 2010

Peter Sewell Jaroslav Ševčík Tim Harris

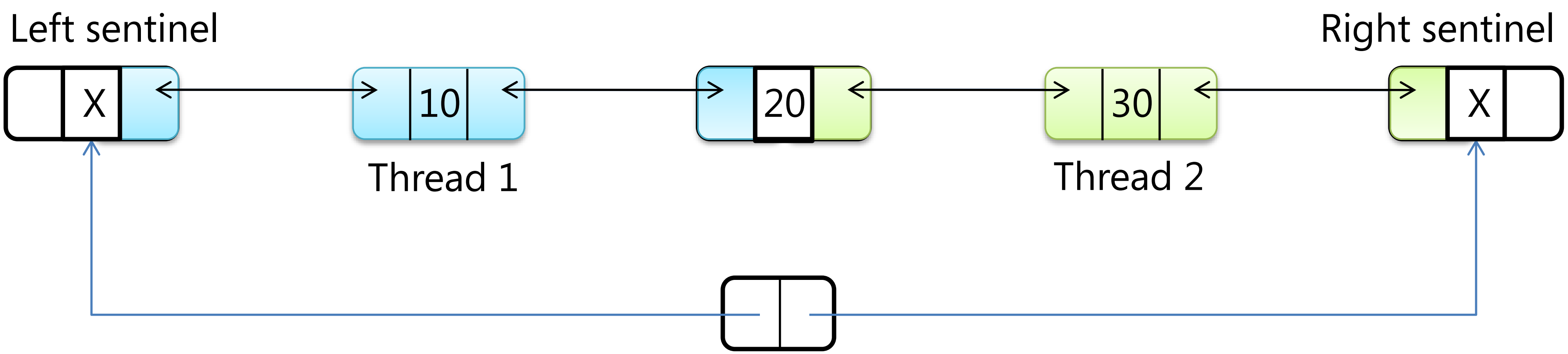
Transactional memory

Bartok-STM prototype

Strong isolation

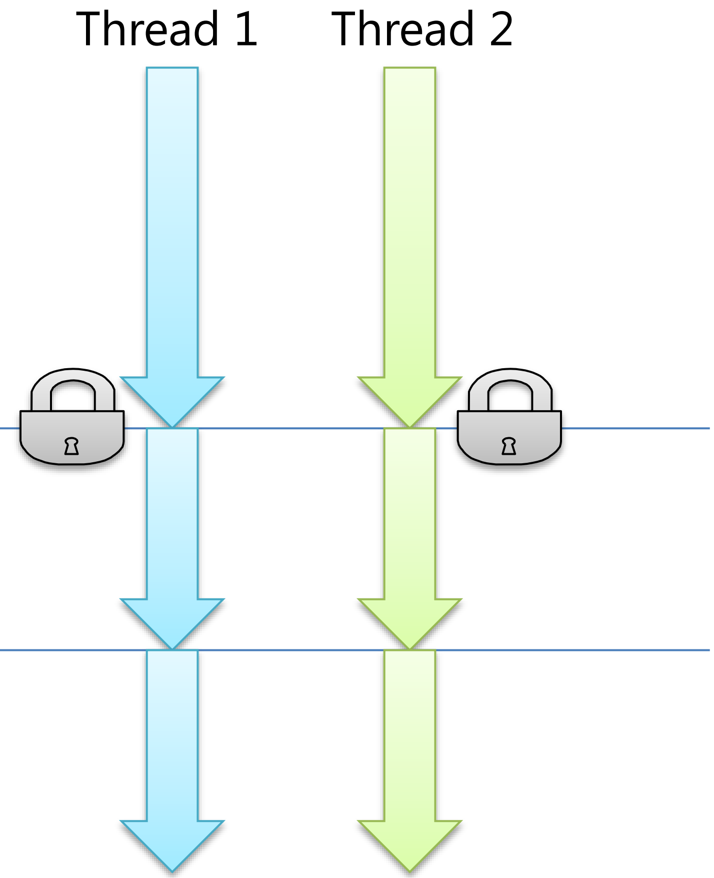
Current performance

Example: double-ended queue



Example: coarse-grained locking

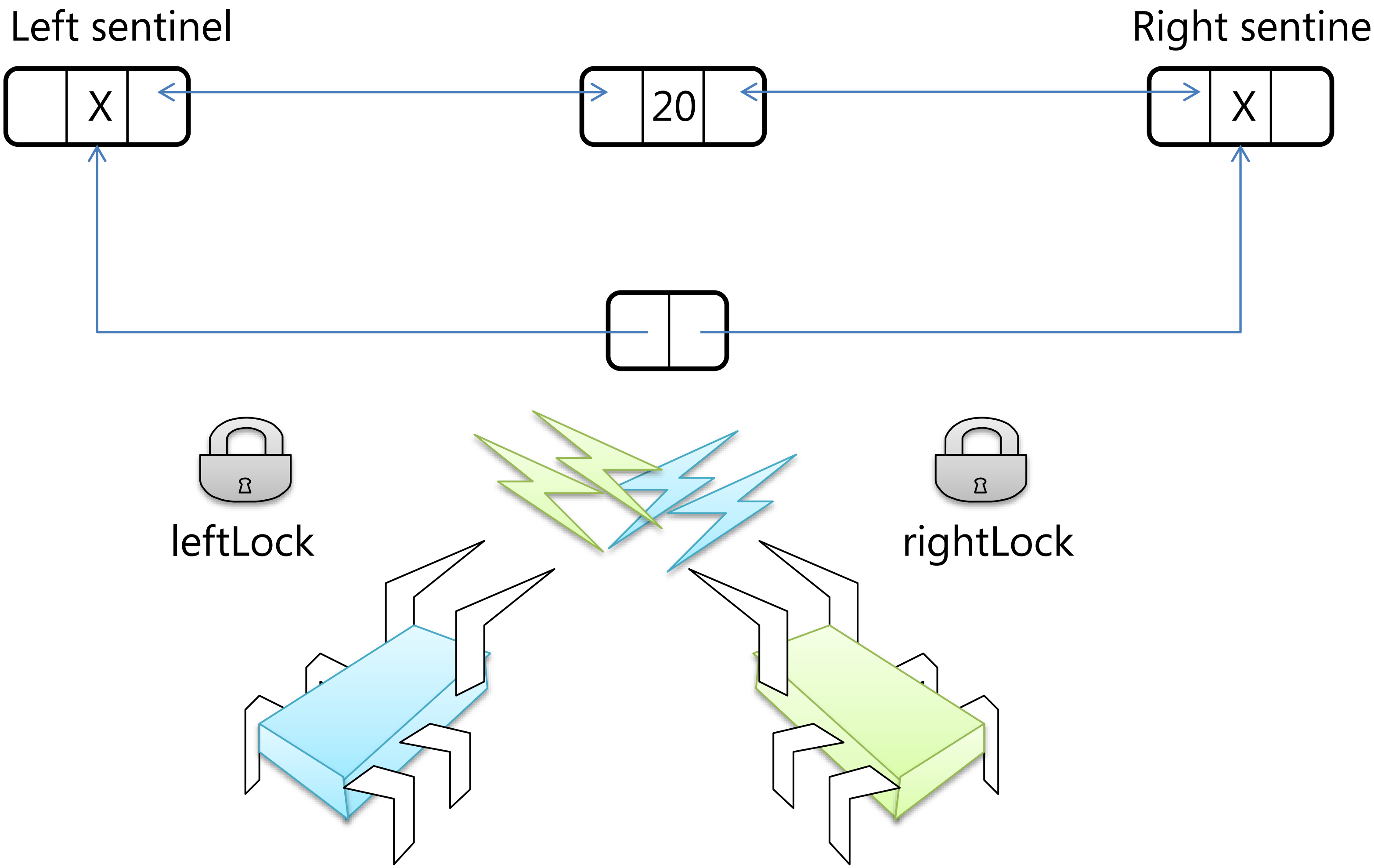
```
Class Q {
  Lock qLock = new Lock();
  QElem leftSentinel;
  QElem rightSentinel;

  void pushLeft(int item) {
    QElem e = new QElem(item);
    qLock.Acquire();
    e.right = this.leftSentinel.right;
    e.left = this.leftSentinel;
    this.leftSentinel.right.left = e;
    this.leftSentinel.right = e;
    qLock.Release();
  }

  ...
}
```



Example: fine-grain locking

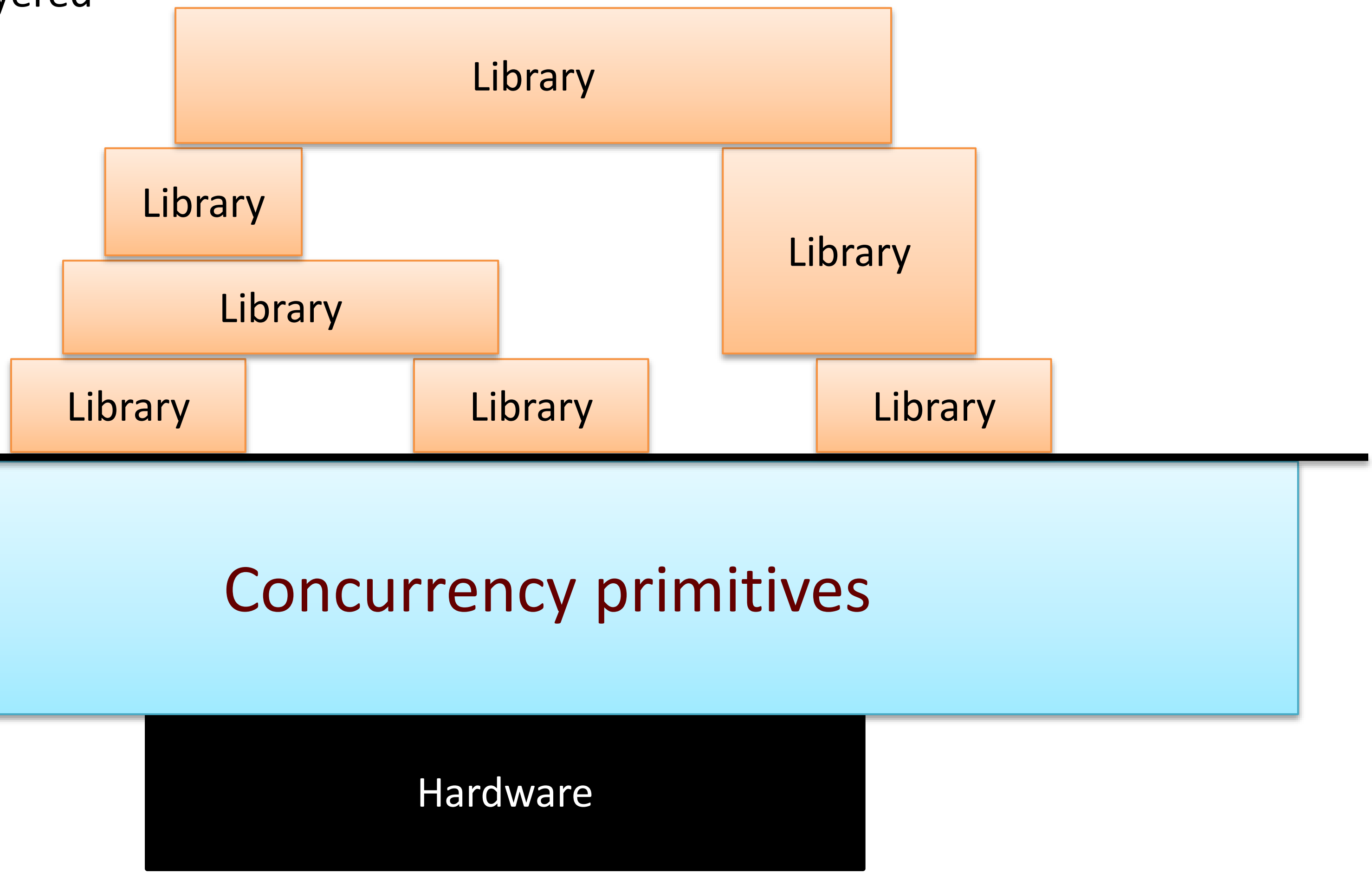
```
Class Q {
  Lock leftLock = new Lock();
  Lock rightRlock = new Lock();
  QElem leftSentinel;
  QElem rightSentinel;

  void pushLeft(int item) {
    QElem e = new QElem(item);
    leftLock.Acquire();
    e.right = this.leftSentinel.right;
    e.left = this.leftSentinel;
    this.leftSentinel.right.left = e;
    this.leftSentinel.right = e;
    leftLock.Release();
  }

  ...
}
```

Example: fine-grain locking



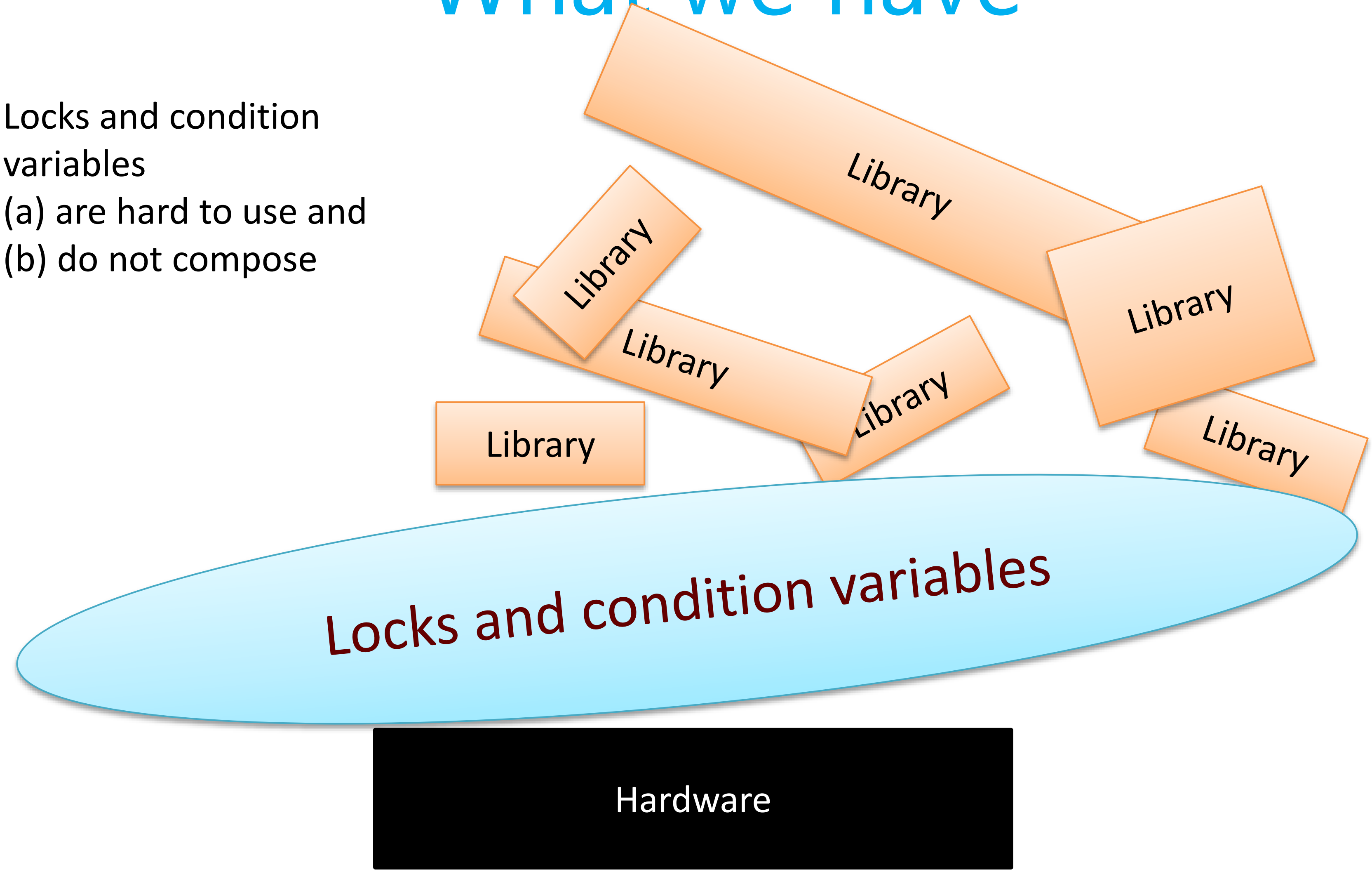
What we want

Libraries build layered concurrency abstractions

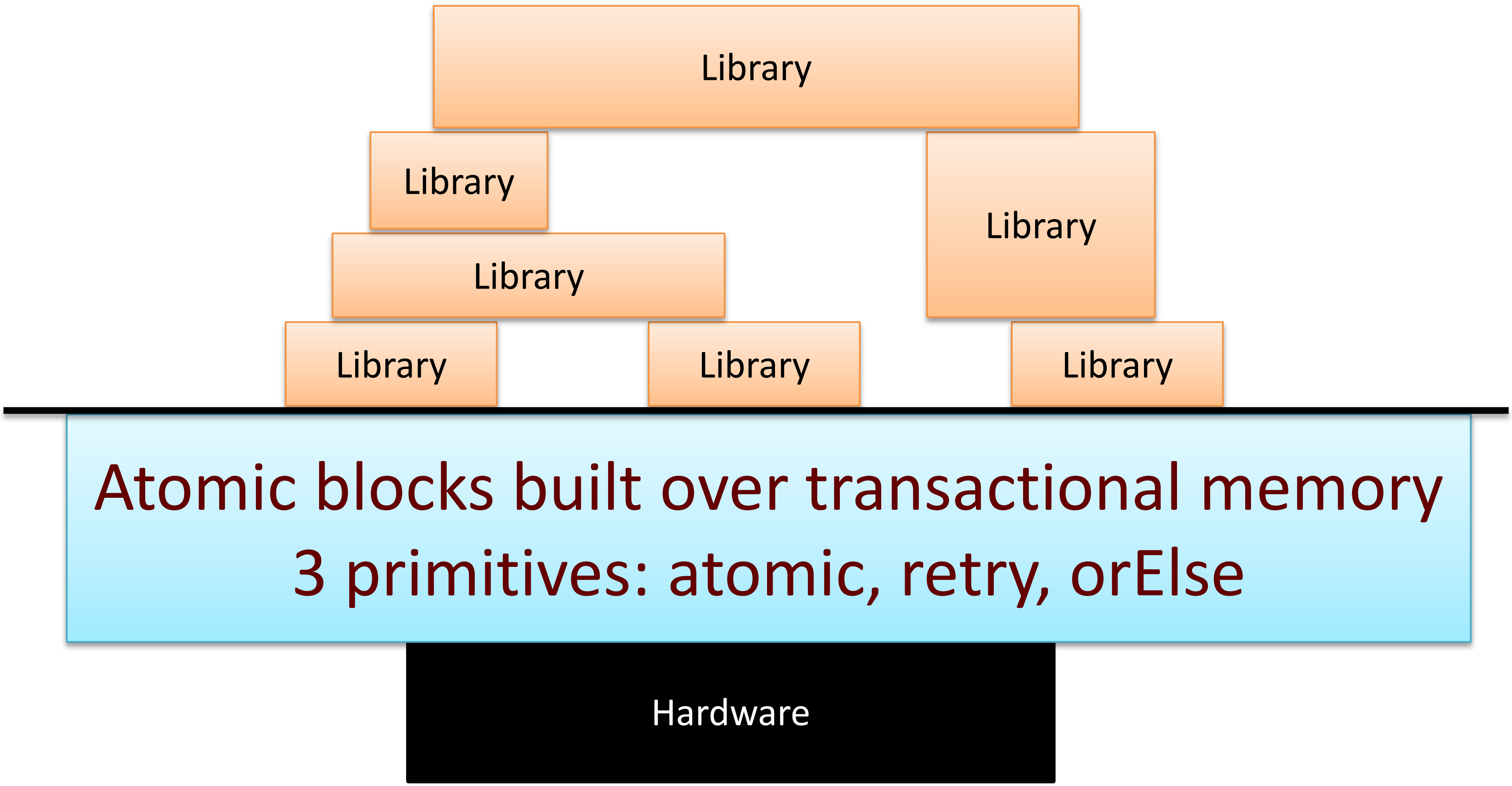


What we have

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



Atomic blocks



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 );
}
```

Composition is THE way we build big programs that work

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

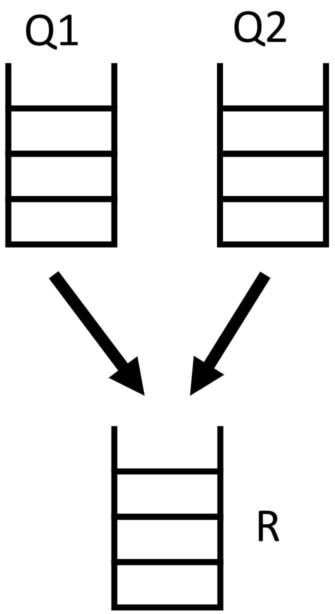
Blocking: how does PopLeft wait for data?

```
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

```
void GetEither() {
        atomic {
                do { i = Q1.Get(); }
                orelse { i = Q2.Get(); }
                R.Put( i );
}       }
```



- **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

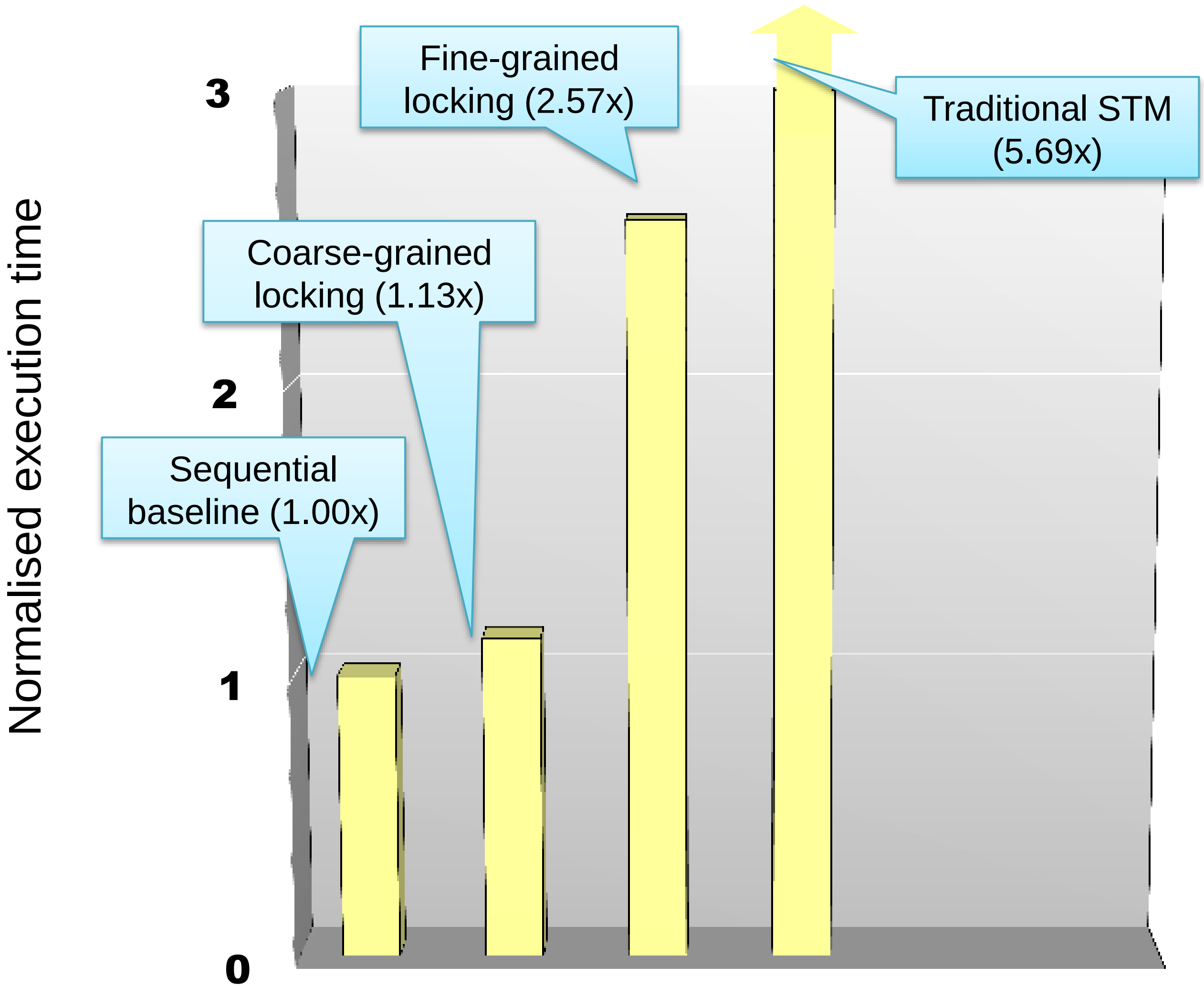
- What are the **invariants** (e.g. the tree is balanced)?
- Each atomic block maintains the invariants
- **Purely sequential reasoning** within a block, which is dramatically easier
- Much easier setting for static analysis tools

Microsoft Research

Summary so far

- Atomic blocks (atomic, retry, orElse) are a real step forward
- It's like using a high-level language instead of assembly code: whole classes of low-level errors are eliminated.
- Not a silver bullet:
  - you can still write buggy programs;
  - concurrent programs are still harder to write than sequential ones;
  - just aimed at shared memory.
- But the improvement is very substantial

State of the art ~ 2003

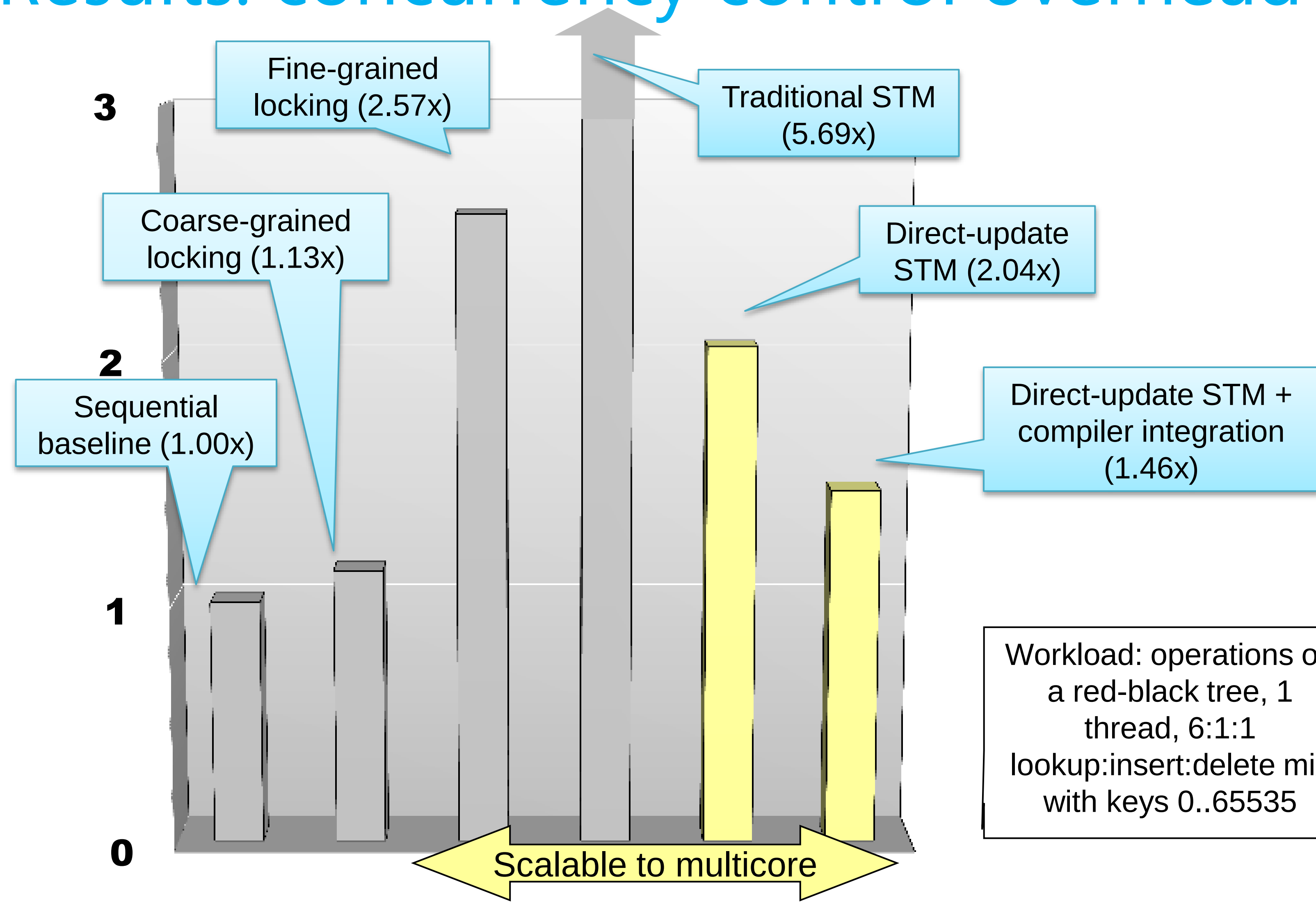


Workload: operations on a red-black tree, 1 thread, 6:1:1 lookup:insert:delete mix with keys 0..65535

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

Results: concurrency control overhead



Workload: operations on a red-black tree, 1 thread, 6:1:1 lookup:insert:delete mix with keys 0..65535

Microsoft Research

Transactional memory

Bartok-STM prototype

Strong isolation

Current performance

Direct update STM

- Transactional write:
  - Lock objects before they are written to (abort if another thread has that lock)
  - Log the overwritten data – we need it to restore the heap case of *retry,* transaction *abort*, or a conflict with a concurrent thread
- Transactional read:
  - Log a *version number* we associate with the object
- Commit:
  - Check the version numbers of objects we've read
  - Increment the version numbers of object we've written

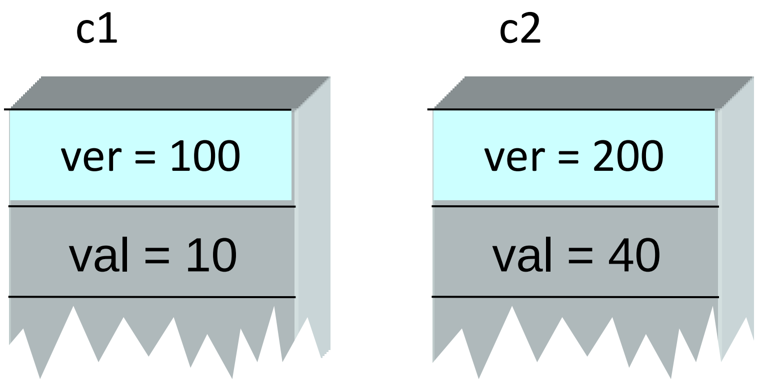
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

T2's log:

Example: contention between transactions

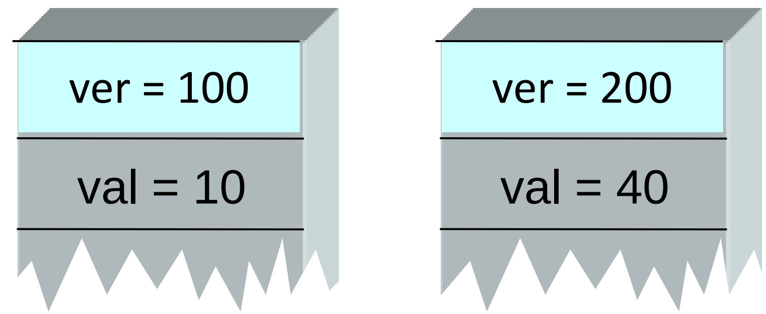
Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```

c1 c2



T1's log:

c1.ver=100

T2's log:

T1 reads from c1: logs that it saw version 100

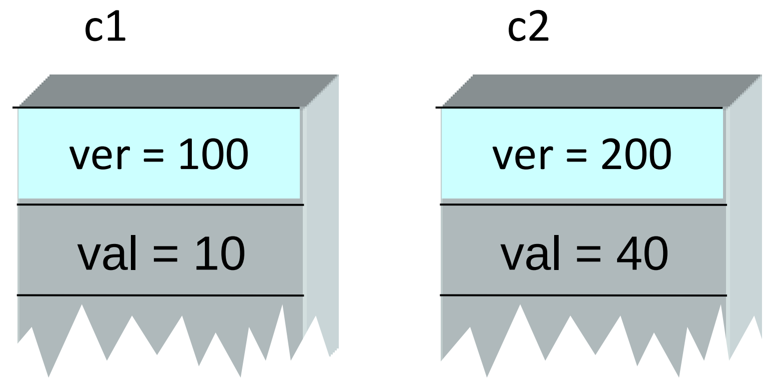
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

c1.ver=100

T2's log:

c1.ver=100

T2 also reads from c1: logs that it saw version 100

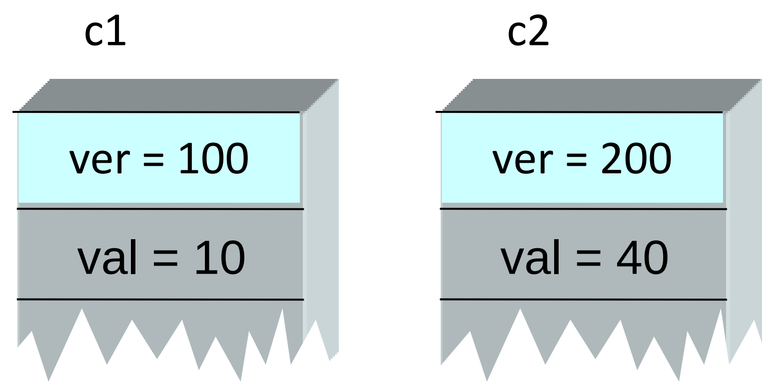
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

c1.ver=100
c2.ver=200

T2's log:

c1.ver=100

Suppose T1 now reads from c2, sees it at version 200

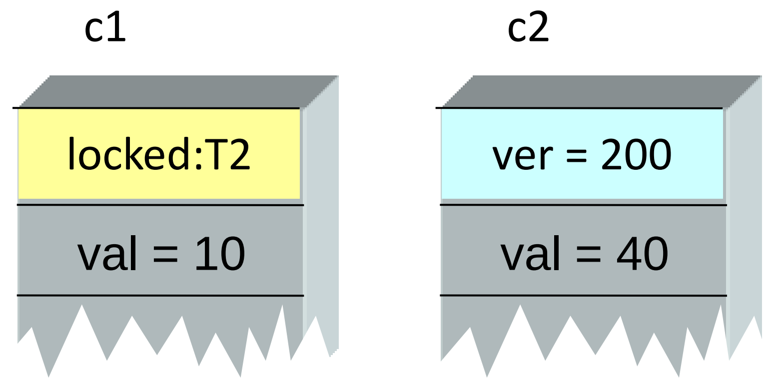
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

c1.ver=100
c2.ver=200

T2's log:

c1.ver=100
lock: c1, 100

Before updating c1, thread T2 must lock it: record old version number

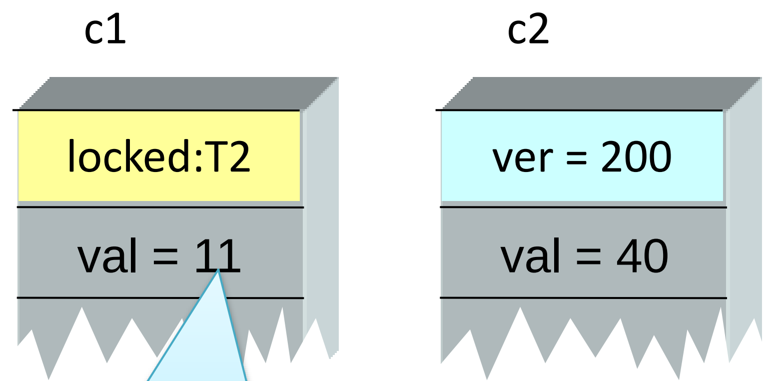
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



(2) After logging the old value, T2 makes its update in place to c1

T1's log:

c1.ver=100
c2.ver=200

T2's log:

c1.ver=100
lock: c1, 100
c1.val=10

(1) Before updating c1.val, thread T2 must log the data it's going to overwrite

Example: contention between transactions

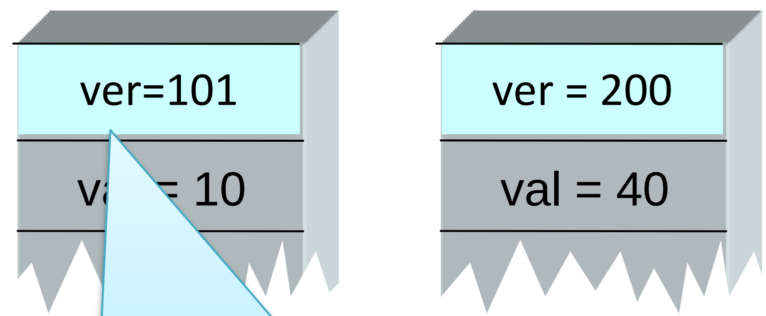
Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```

c1 c2



(2) T2's transaction commits successfully. Unlock the object, installing the new version number

T1's log:

c1.ver=100
c2.ver=200

T2's log:

c1.ver=100
lock: c1, 100
c1.val=10

(1) Check the version we locked matches the version we previously read

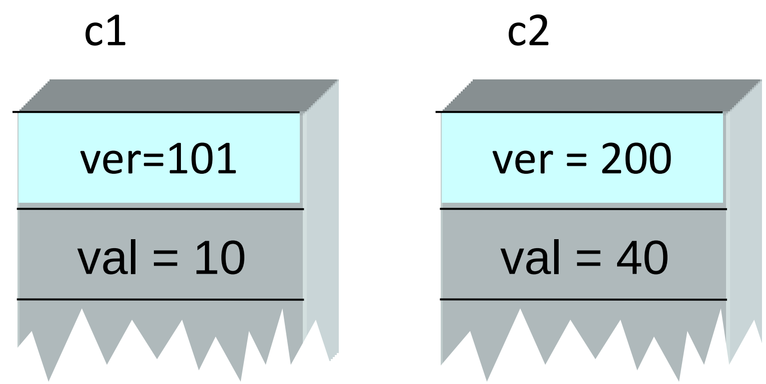
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

c1.ver=100
c2.ver=100

T2's log:

(1) T1 attempts to commit. Check the versions it read are still up-to-date.

(2) Object c1 was updated from version 100 to 101, so T1's transaction is aborted and re-run.

Microsoft Research

Transactional memory

Bartok-STM prototype

Strong isolation

Current performance

Zombie transactions

Initially: x==y==z==0

```
atomic {
        x = 1;
        y = 1;
}
```

```
atomic {
   if (x != y) z = 1;
}
```

```
temp = z;
```

- temp==0 is the only correct result here if these blocks really are atomic

Zombie transactions

Direct update, lazy conflict detection

```
atomic {
        x = 1;
        y = 1;
}
```

```
atomic {
   if (x != y) z = 1;
}
```

```
temp = z;
```

- x == 0
- y == 0
- z == 0

Zombie transactions

Direct update, lazy conflict detection

```
atomic {
        x = 1;
        y = 1;
}
```

```
atomic {
   if (x != y) z = 1;
}
```

0

```
temp = z;
```

- x == 0
- y == 0
- z == 0

Zombie transactions

Direct update, lazy conflict detection

```
atomic {
      x = 1;
      y = 1;
}
```

```
atomic {
   if (x != y) z = 1;
}
```

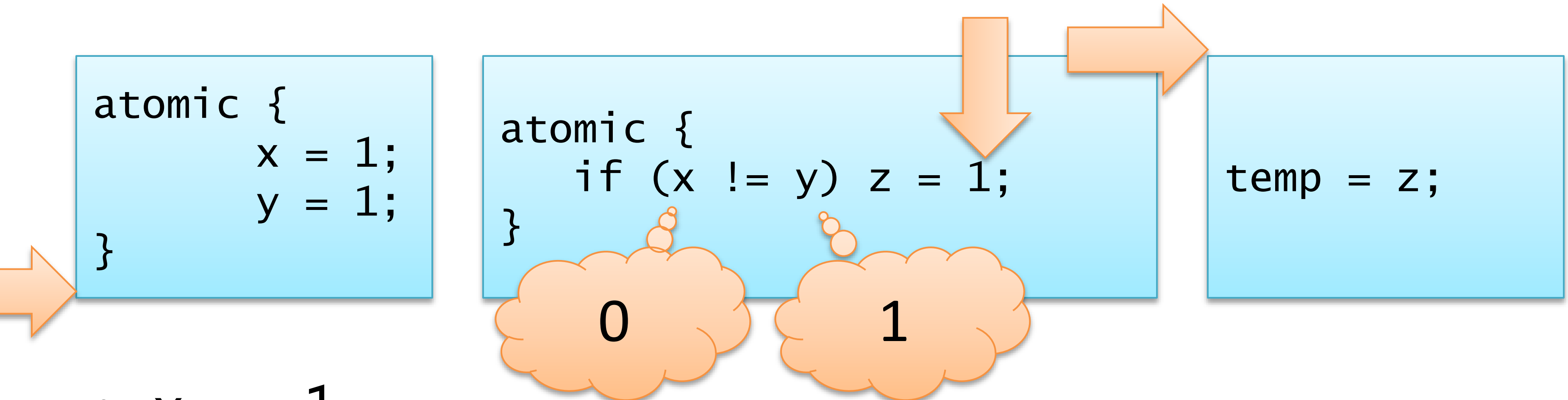
0

```
temp = z;
```

- x == 1
- y == 1
- z == 0

Zombie transactions

Direct update, lazy conflict detection



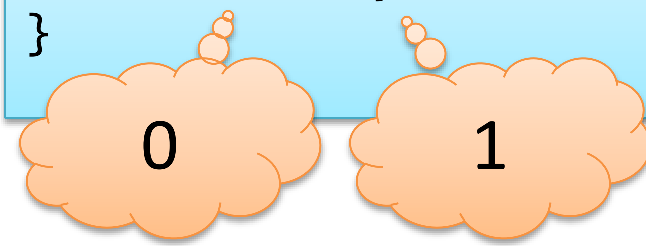
- x == 1
- y == 1
- z == 1

Zombie transactions

Direct update, lazy conflict detection

```
atomic {
        x = 1;
        y = 1;
}
```

```
atomic {
   if (x != y) z = 1;
}
```



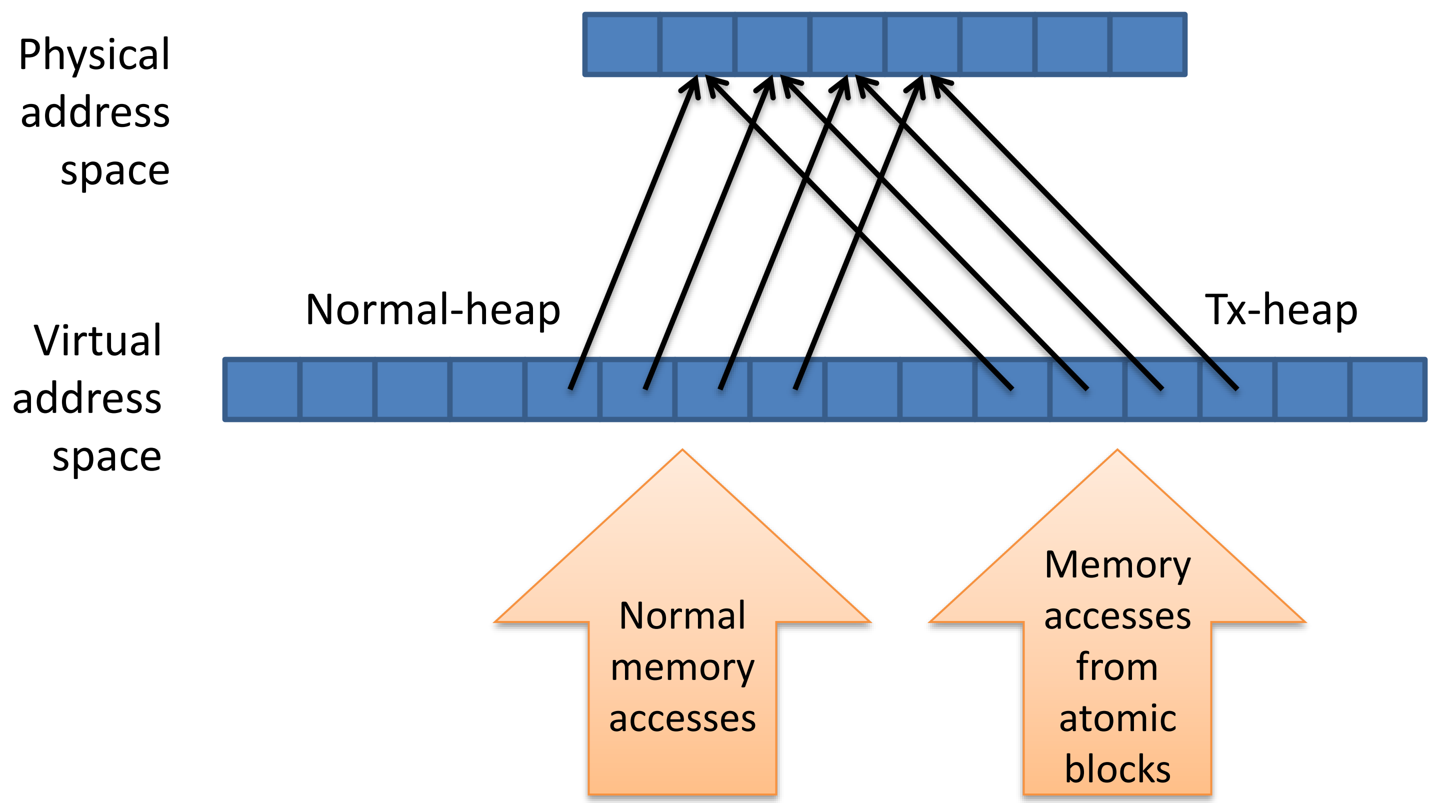

```
temp = z;
```

- x == 1
- y == 1
- z == 1

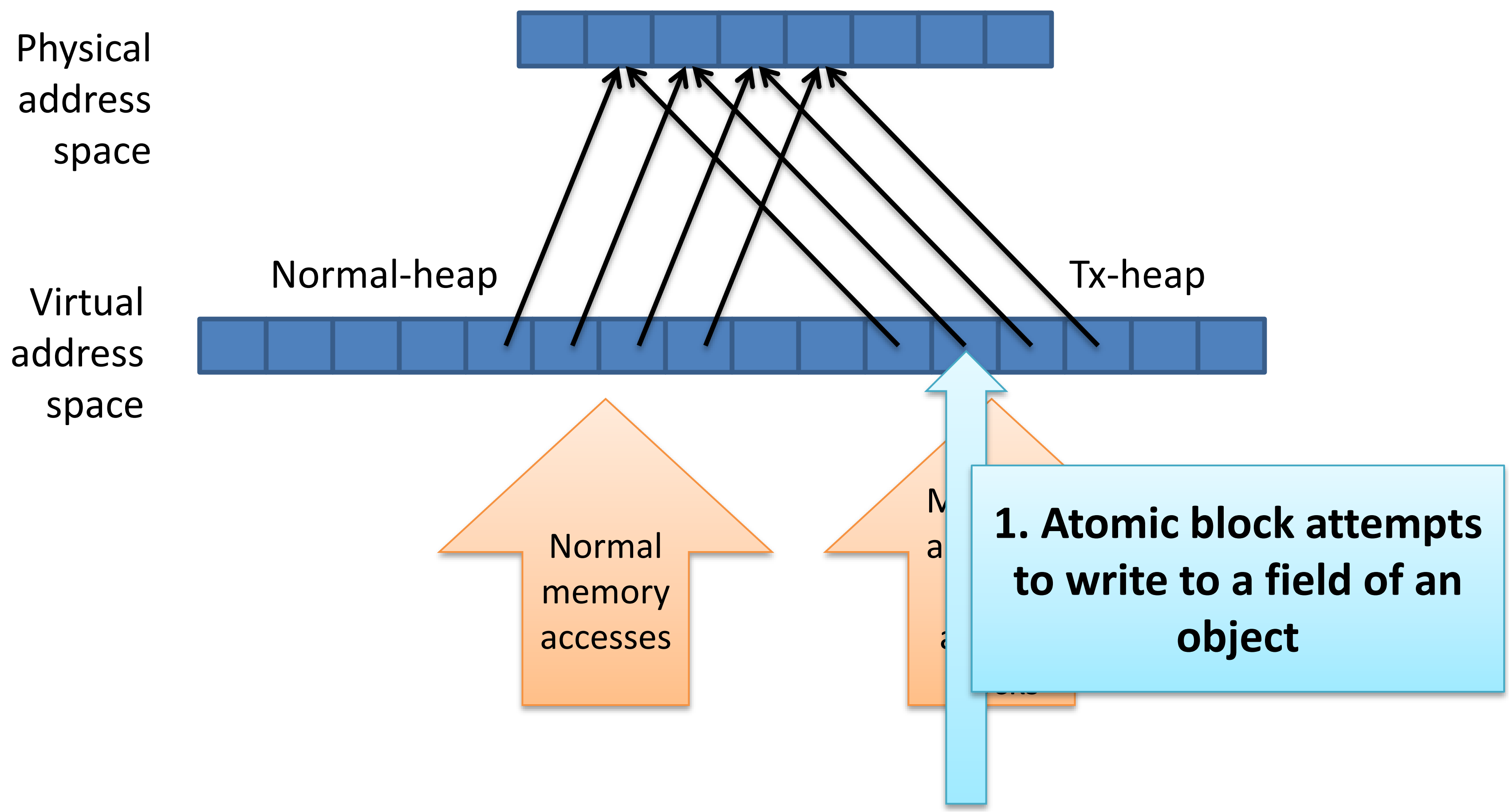
Strong isolation

- Add a mechanism to detect conflicts between tx and normal accesses
  - e.g. ‘z’ in this example
- We would like:
  - Implementation flexibility – e.g. different STMs
  - No overhead on non-transactional accesses
  - Predictable performance
  - Little overhead over weak atomicity

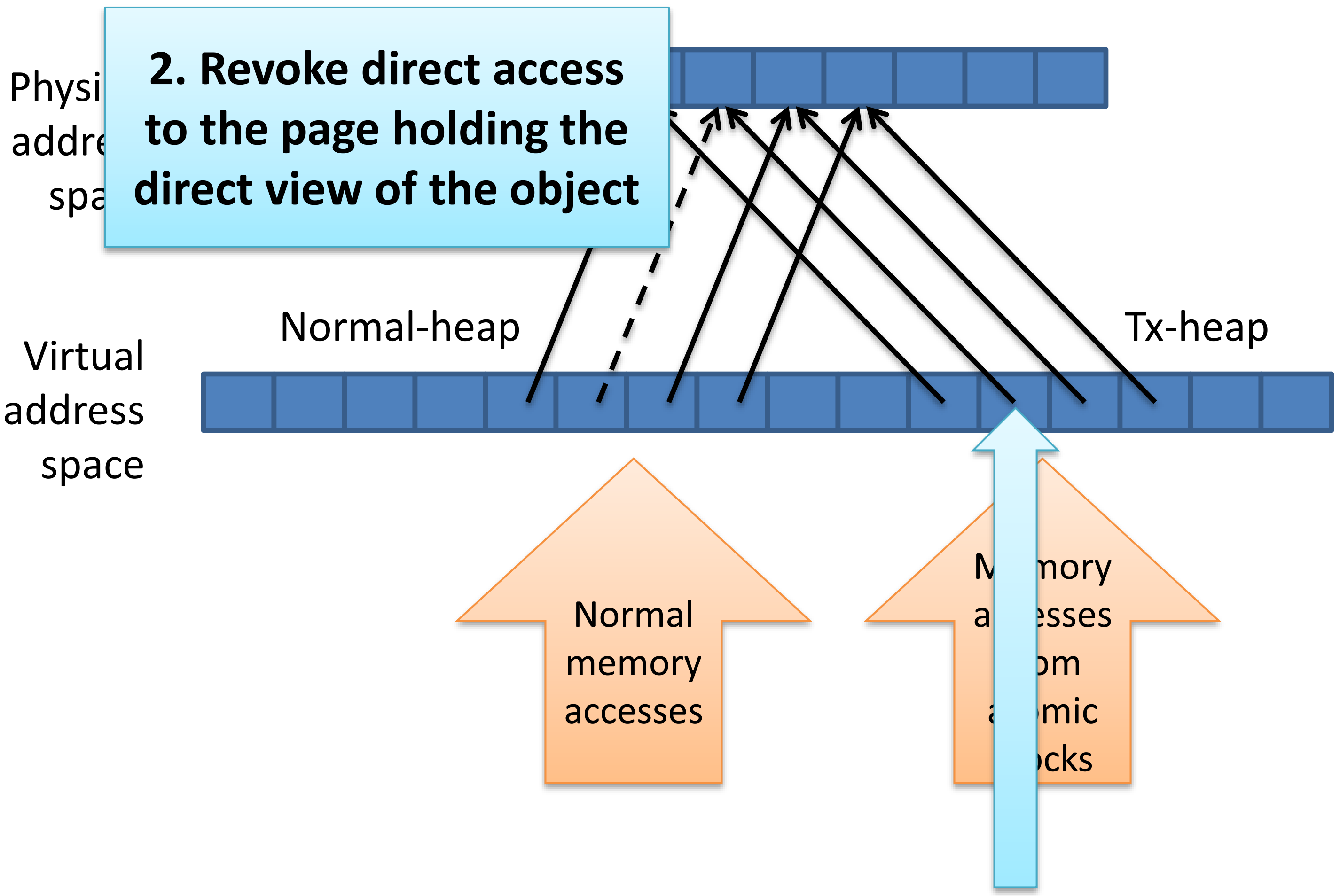
Strong isolation: implementation



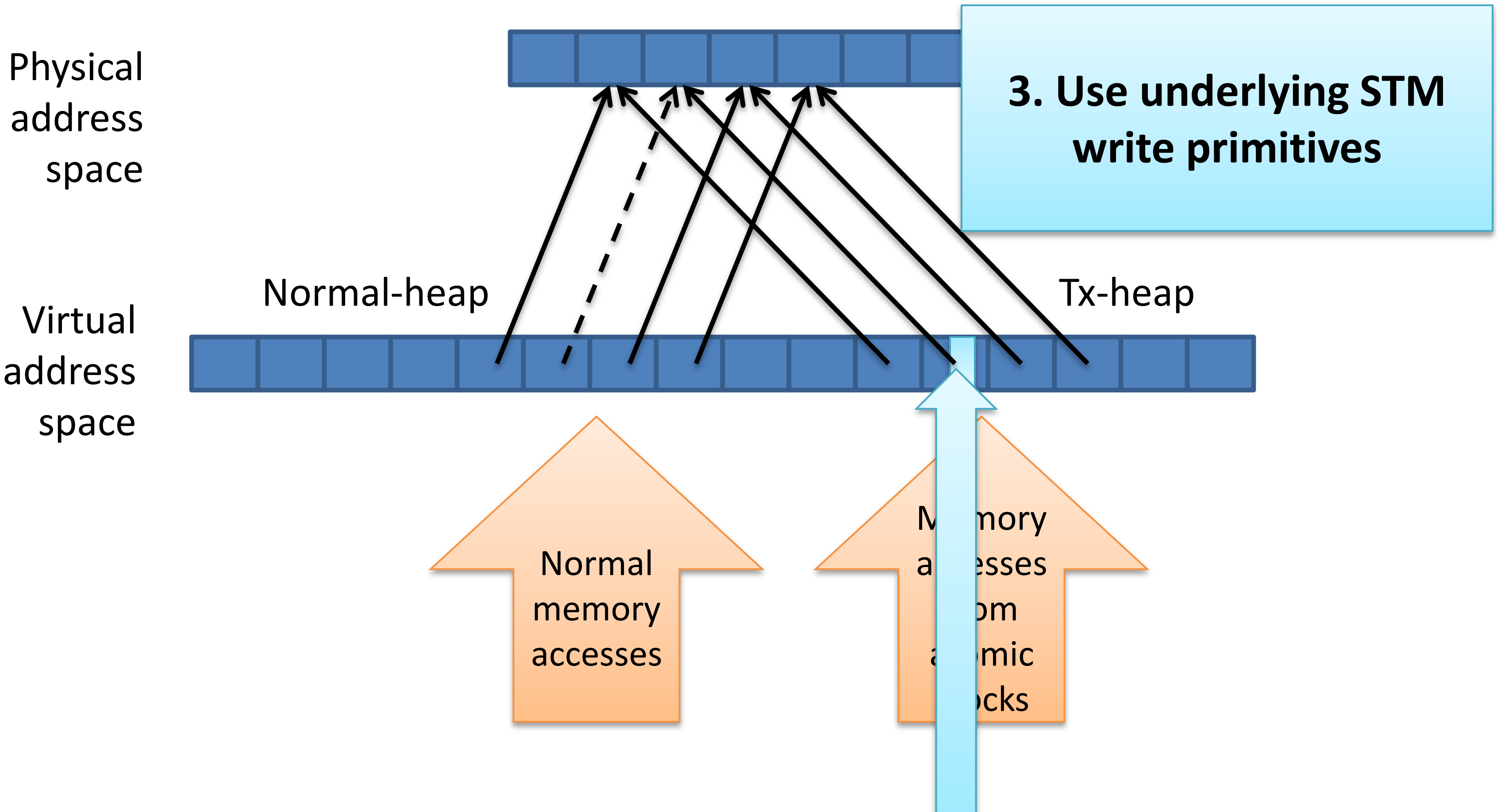
Writes from atomic blocks



Writes from atomic blocks

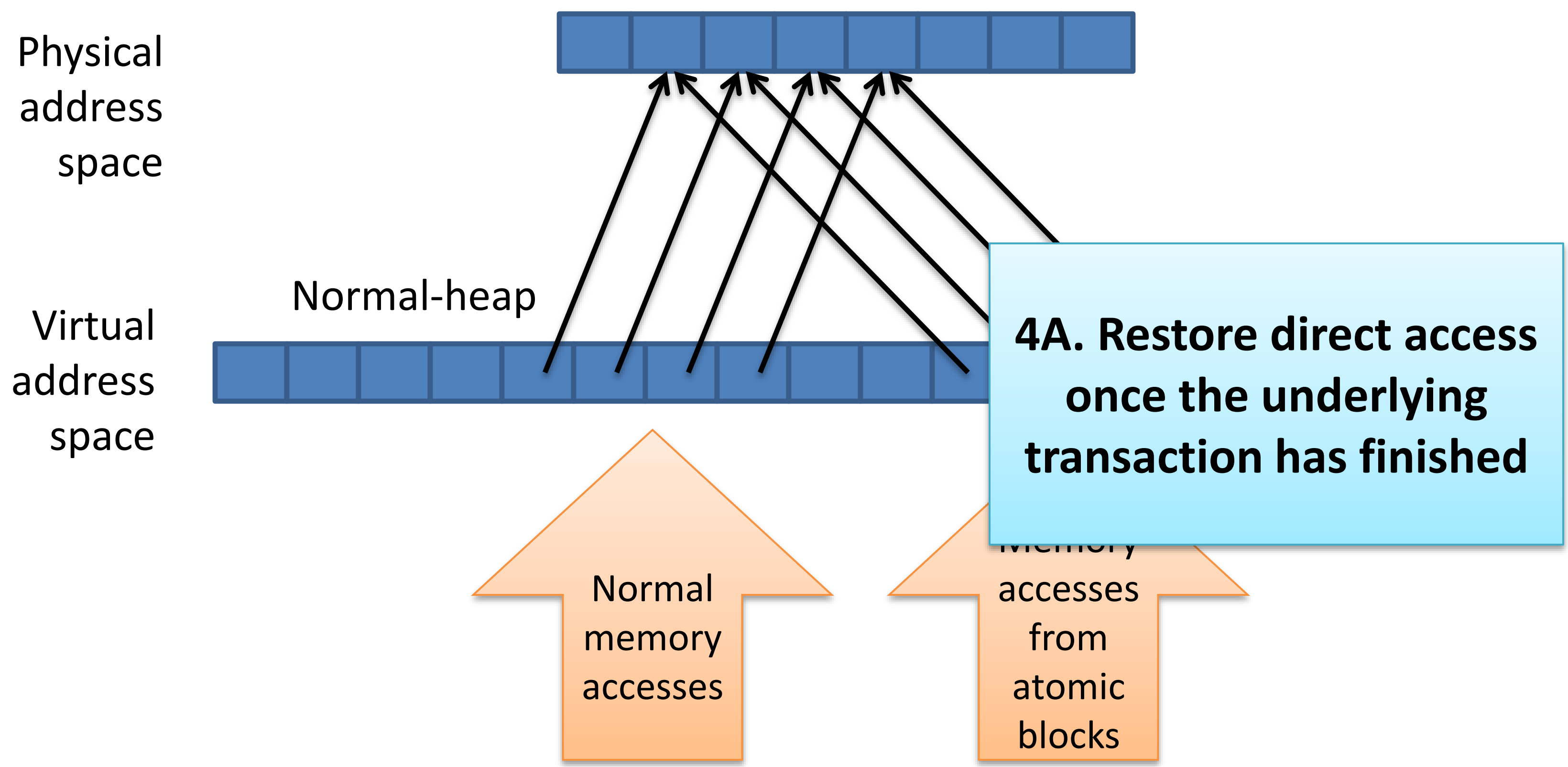


Writes from atomic blocks



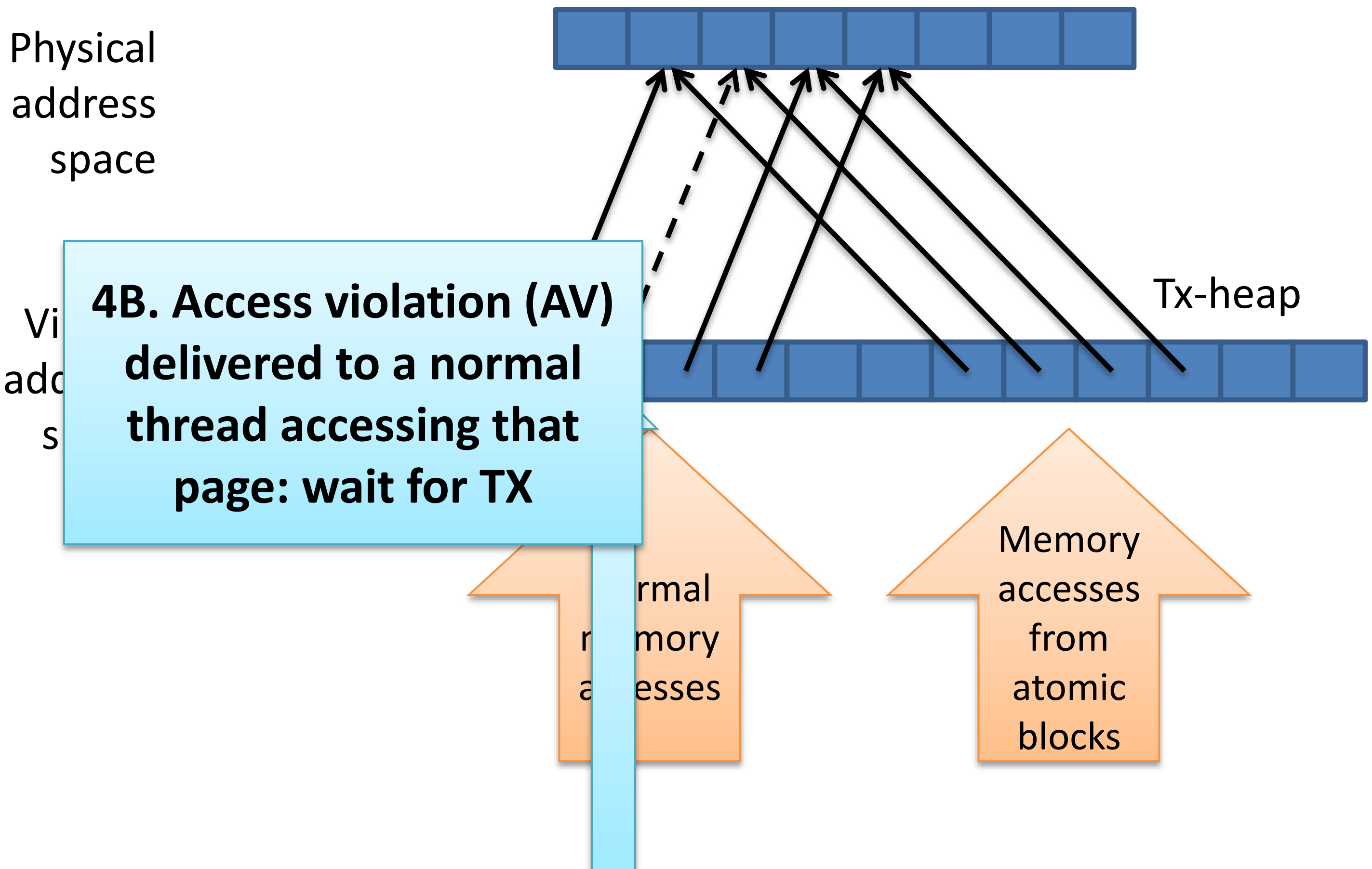
Case 1

Writes from atomic blocks



Case 2

Conflicting normal access



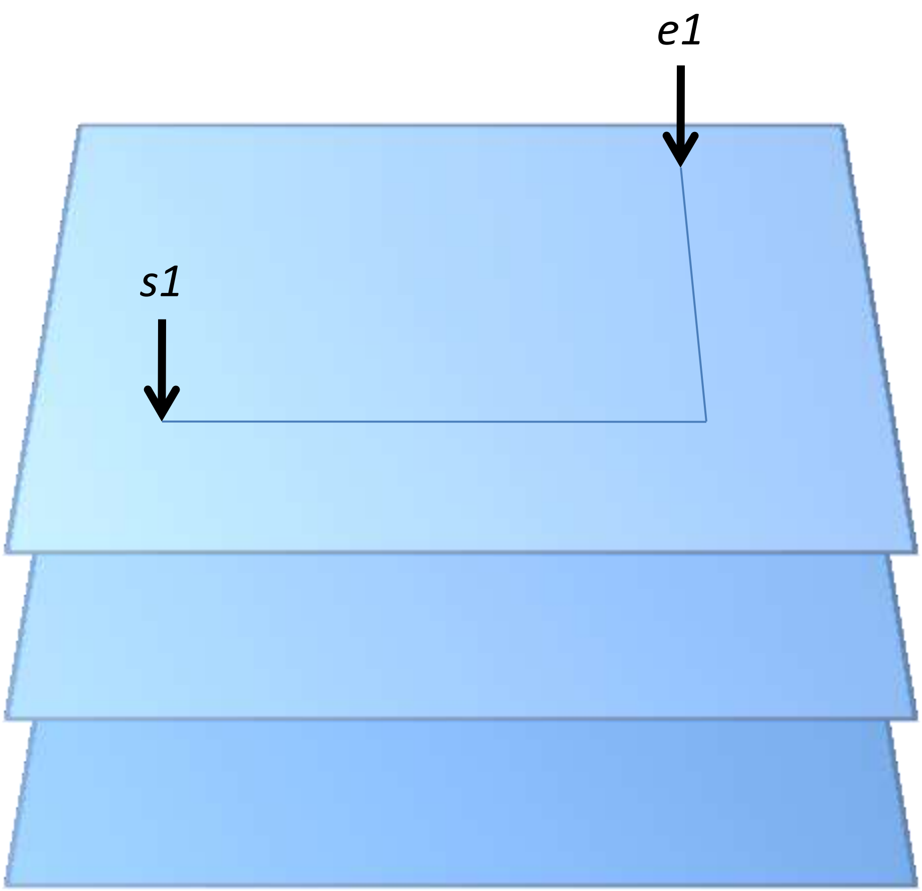
Transactional memory

Bartok-STM prototype

Strong isolation

Current performance

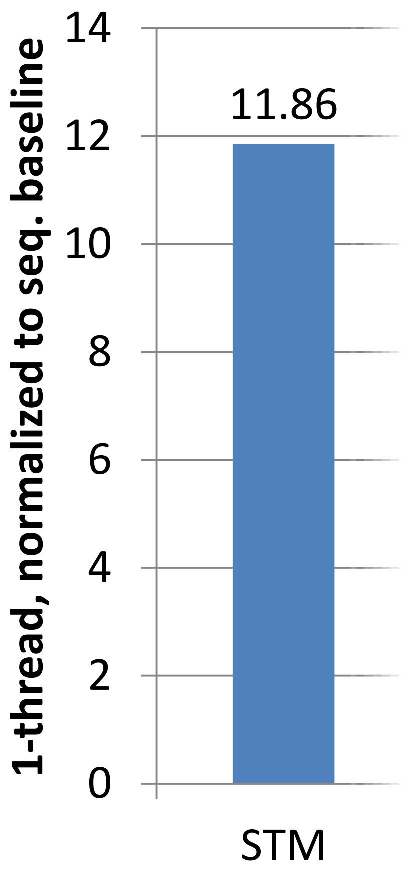
Labyrinth



- 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

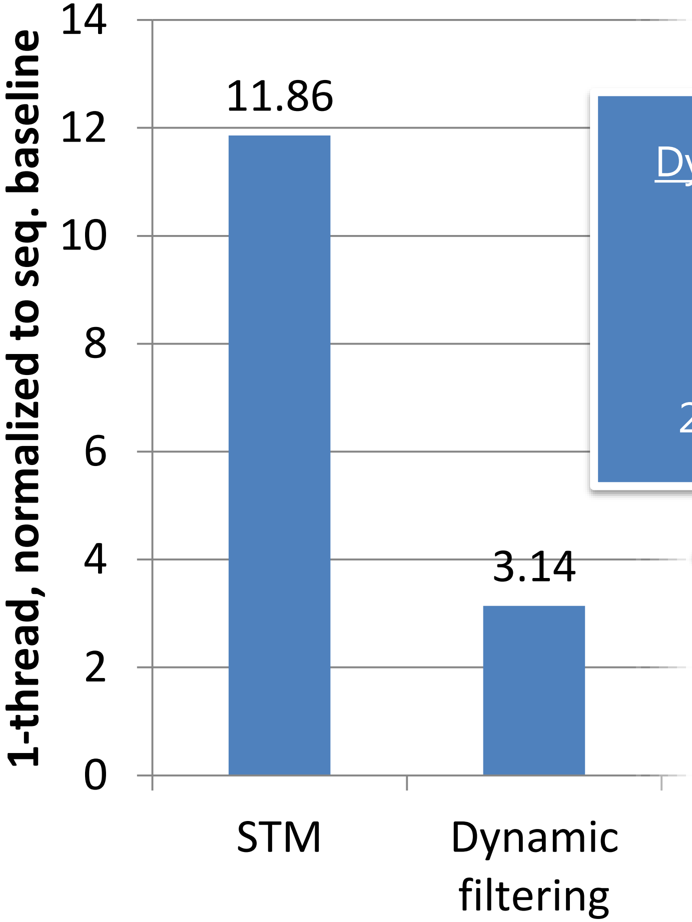
“STAMP: Stanford Transactional Applications for Multi-Processing”
Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun , IISWC 2008

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

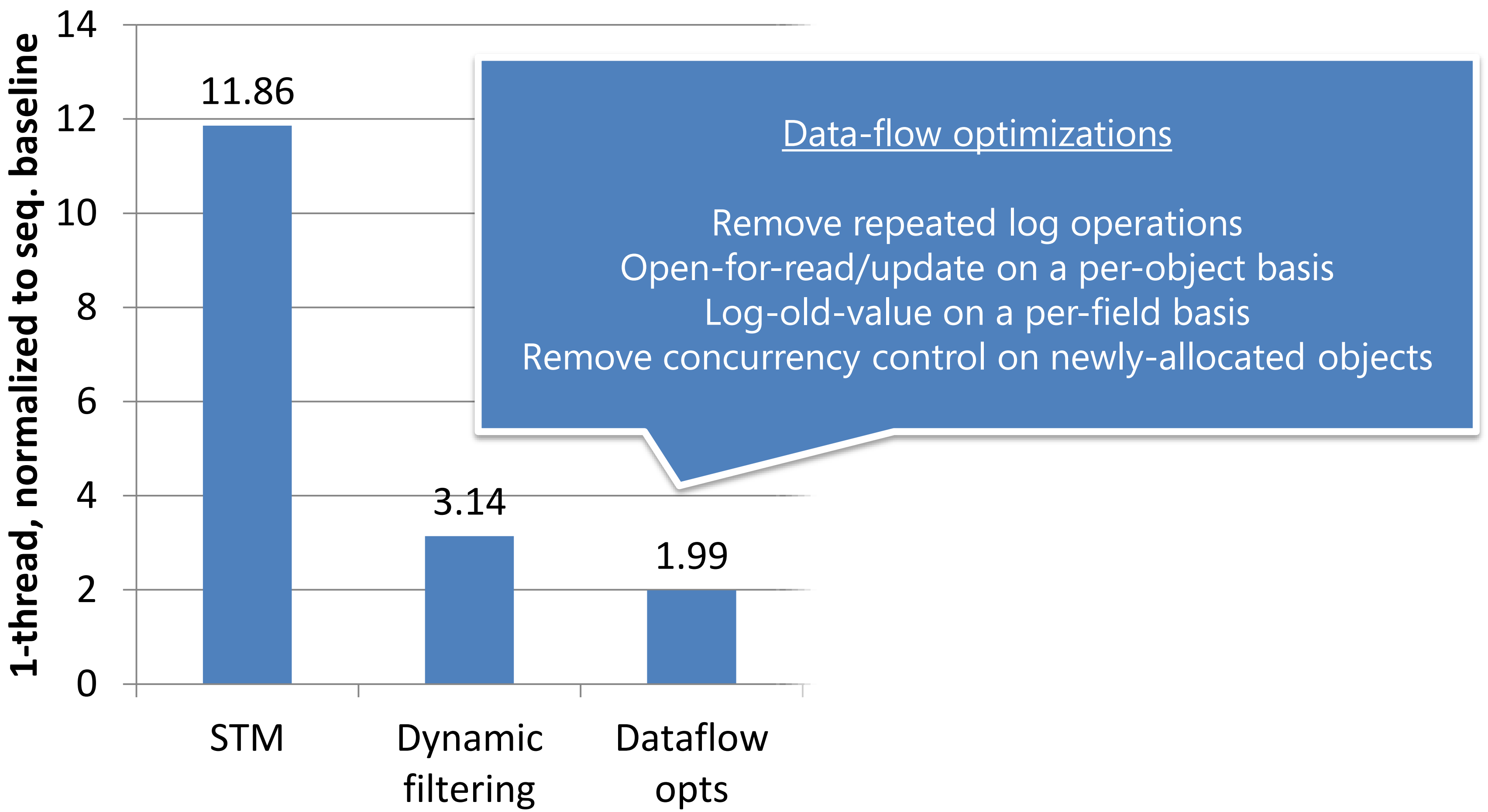
Sequential overhead



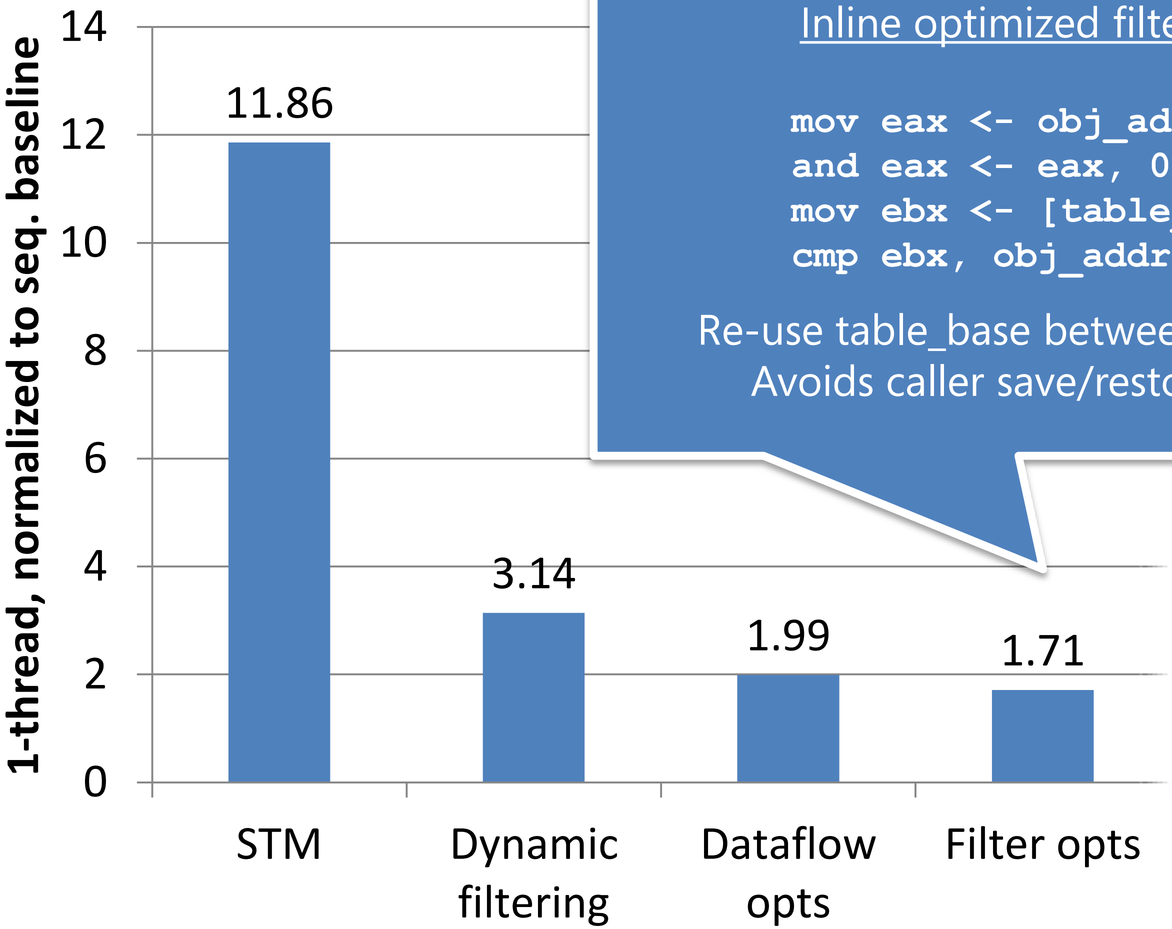
<u>Dynamic filtering to remove redundant logging</u>

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



Sequential overhead

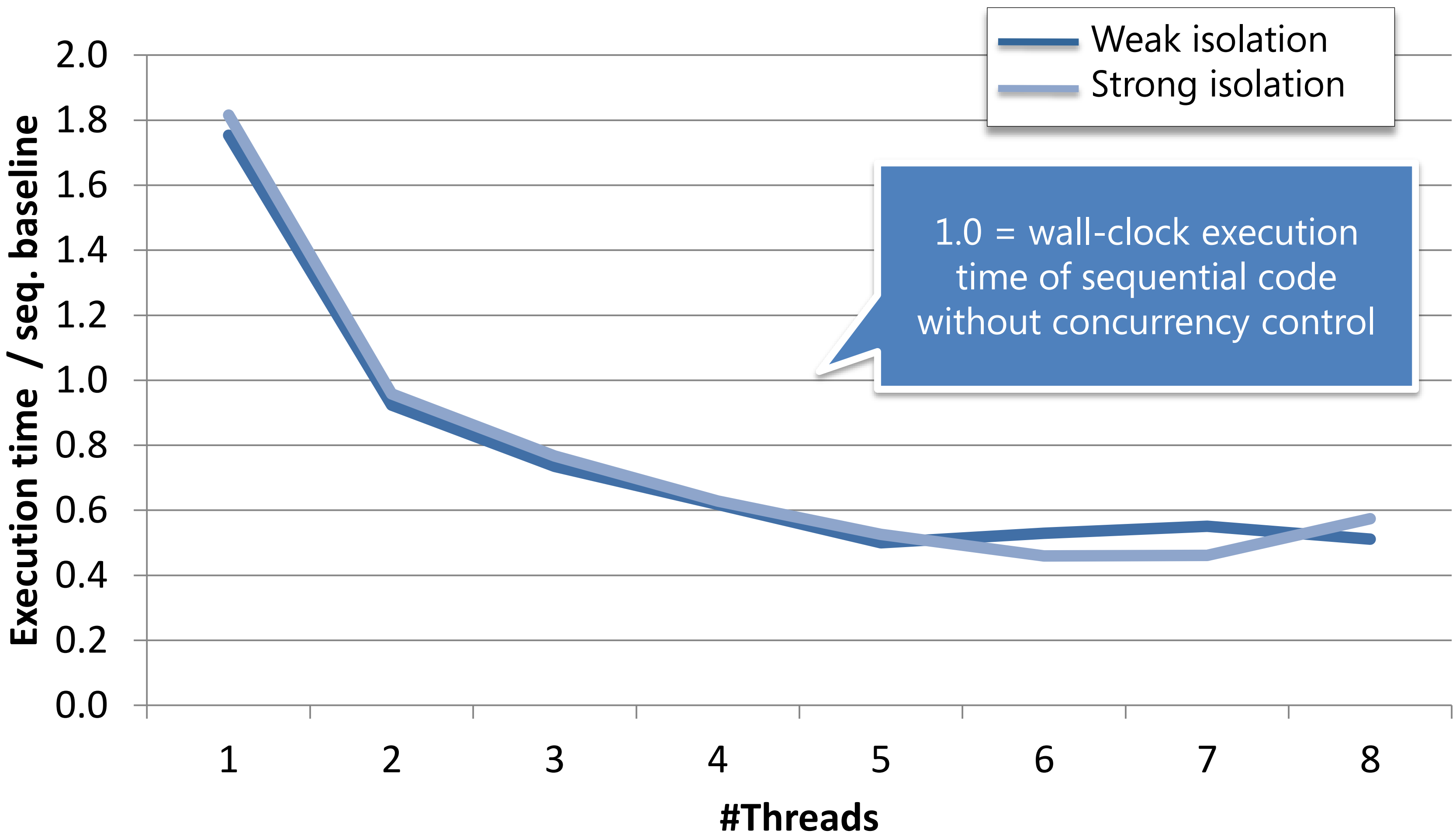


Inline optimized filter operations

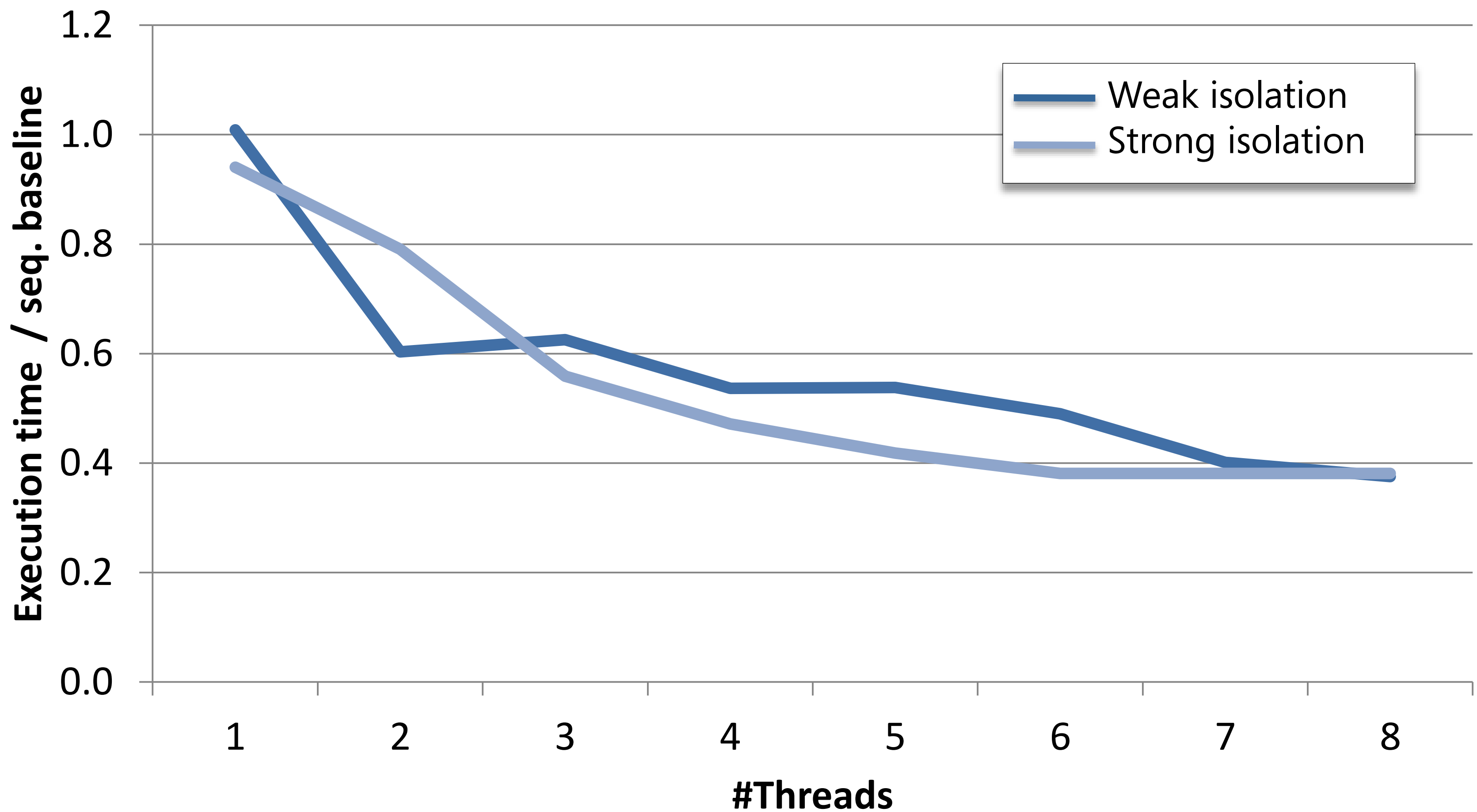
```
mov eax <- obj_addr
and eax <- eax, 0xffc
mov ebx <- [table_base + eax]
cmp ebx, obj_addr
```

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

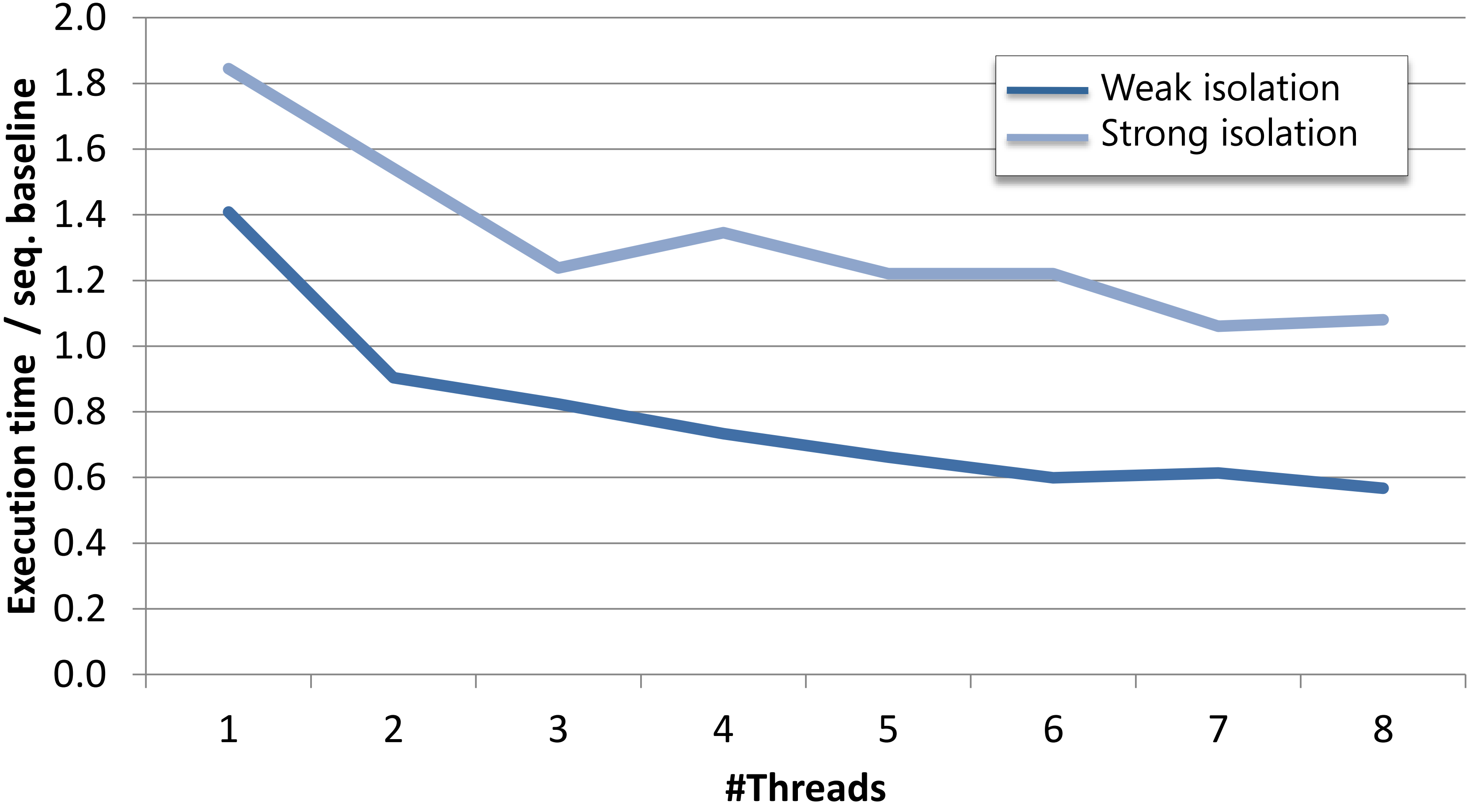
Scaling – Labyrinth



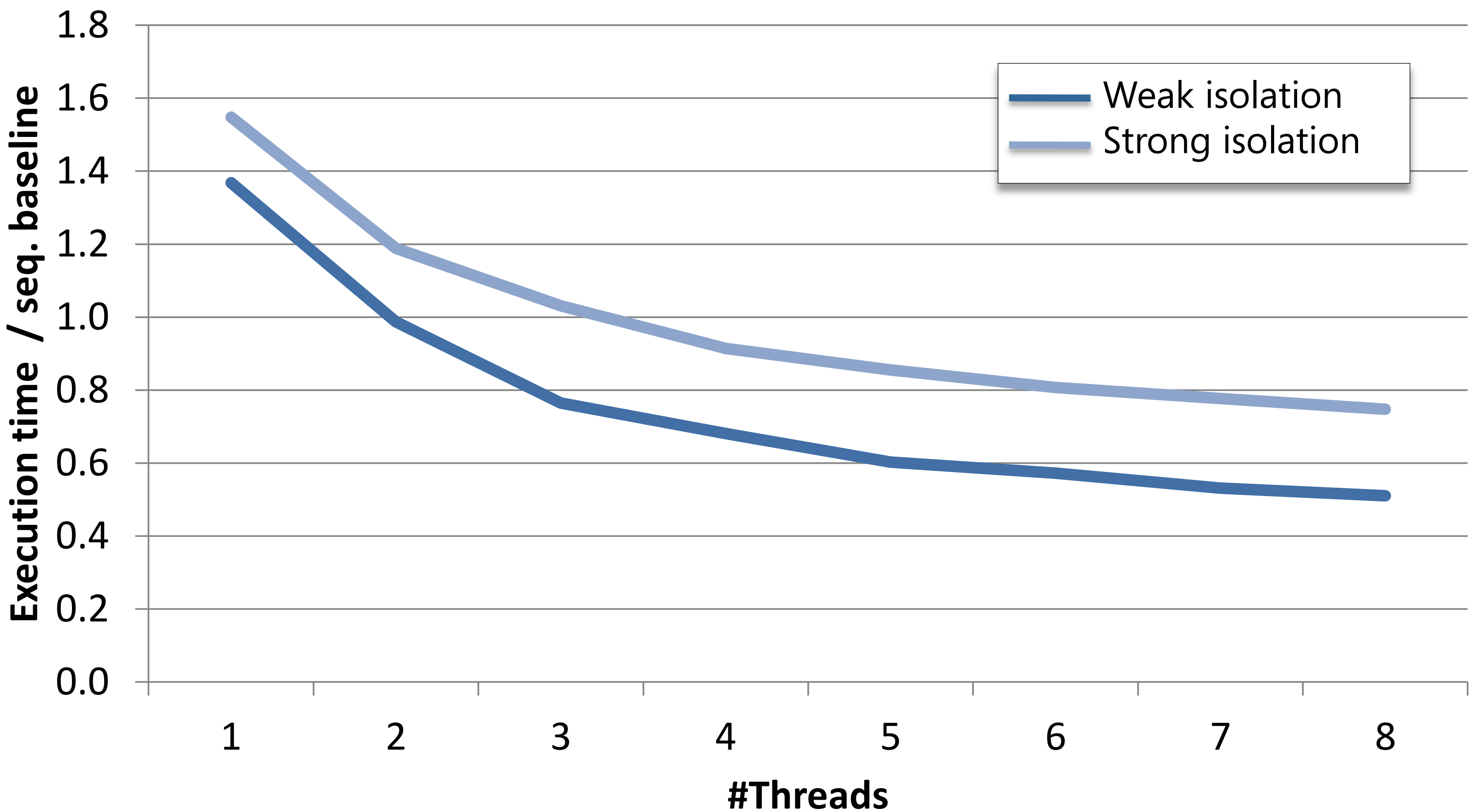
Scaling – Delaunay



Scaling – Genome



Scaling – Vacation



Microsoft Research

Conclusion

- What are atomic blocks good for?
  - Shared memory data structures
- Implementations involve work throughout the software stack
  - Language design
  - Compiler
  - Language runtime system
  - OS-runtime-system interfaces
- Two different experiences
  - STM-Haksell
  - STM.Net