Language constructs for transactional memory

Tim Harris

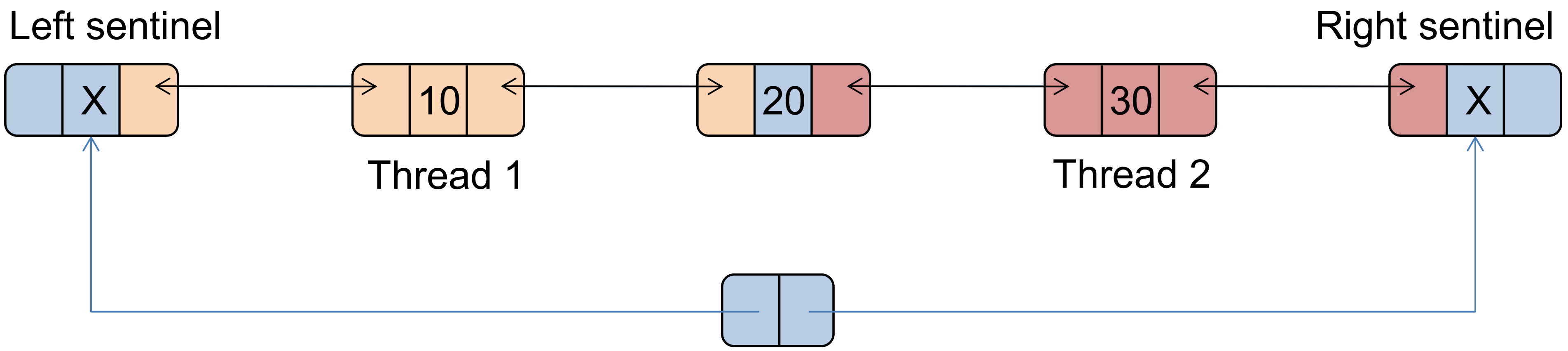
Disclaimer: these are my personal opinions

Untangling "atomic" from TM

Hiding TM from programmers

Current performance

Example: double-ended queue



- Support push/pop on both ends
- Allow concurrency where possible
- Avoid deadlock

Implementing this: TM

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

  void pushLeft(int item) {
    QElem e = new QElem(item);
    do {
      StartTx();
      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 (!CommitTx());
  }

  ...
}
```

Broadly based on word-based STM from “Concurrent programming without locks”
Keir Fraser & Tim Harris, ACM TOCS

Implementing this: atomic blocks

```
Class Q {
  QElem l
  QElem r

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

  ...
}
```

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

  void pushLeft(int item) {
    do {
      StartTx();
      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 (!CommitTx());
  }

  ...
}
```

Design questions

```
Class Q {
  QElem leftSentinel;
                  ;

                tem) {

      QElem e = new QElem(item);
      e.right = this.leftSentinel.right;
      e.left = this.leftSentinel;
      this.leftSentinel.right.left = e;
      this.leftSentinel.right = e;
    }
  }
```

"What about I/O?

"What happens to this object if the atomic block is rolled back?

"What about memory access violations, exceptions, security error logs, ...?

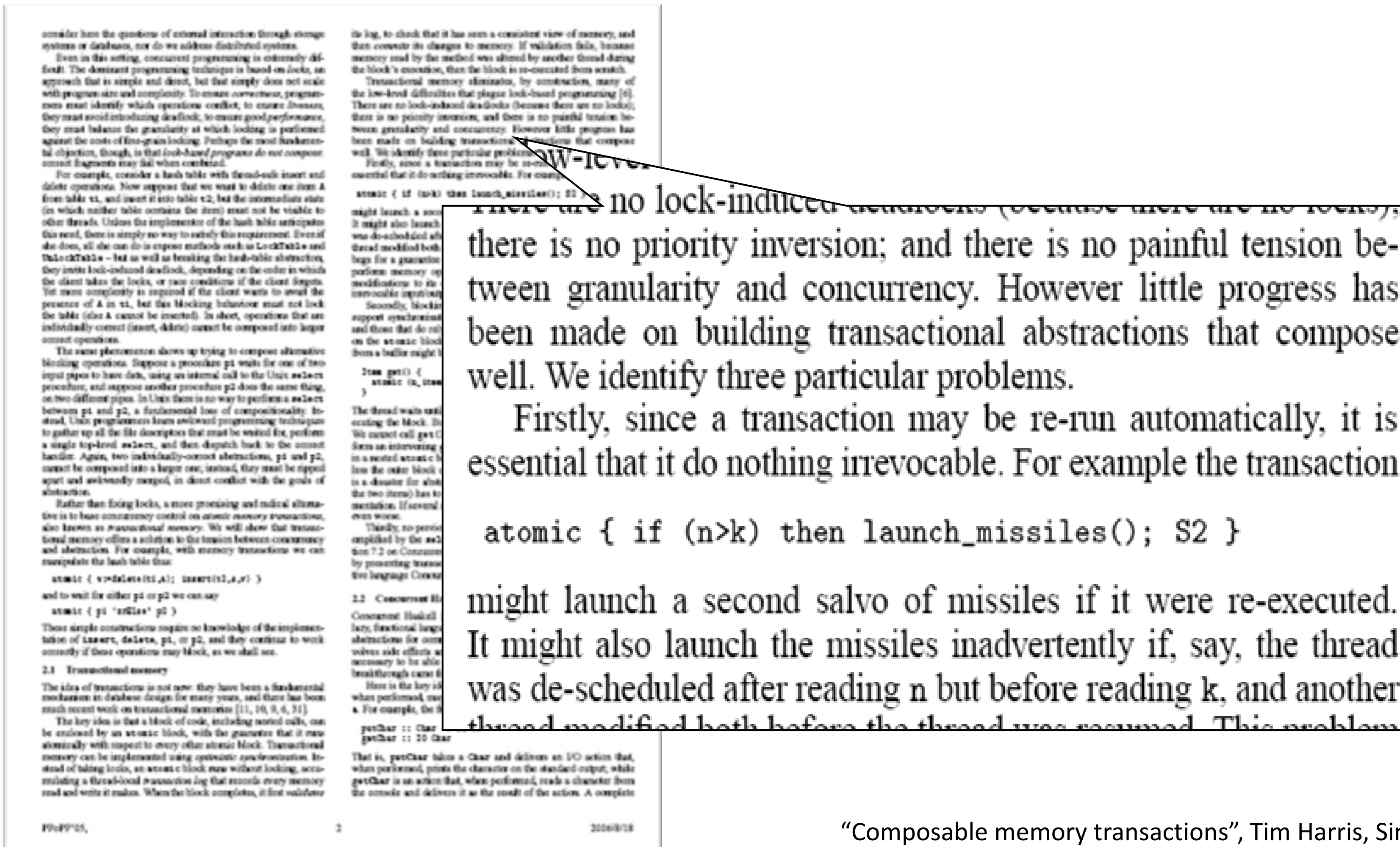
"What happens if I single-step through this, and there's a conflict with a concurrent transaction

"What happens if this fails with an exception; are the other updates rolled back?

"What if another thread tries to access one of these fields without being in an atomic block?

"What if another atomic block updates one of these fields? Will I see the value change mid-way through my atomic block?

“Atomic blocks are transactions”



there is no priority inversion; and there is no painful tension between granularity and concurrency. However little progress has been made on building transactional abstractions that compose well. We identify three particular problems.

Firstly, since a transaction may be re-run automatically, it is essential that it do nothing irrevocable. For example the transaction

```
atomic { if (n>k) then launch_missiles(); S2 }
```

might launch a second salvo of missiles if it were re-executed. It might also launch the missiles inadvertently if, say, the thread was de-scheduled after reading n but before reading k, and another

“Composable memory transactions”, Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy, PPoPP ‘05

Microsoft Research

“Atomic blocks are locks”

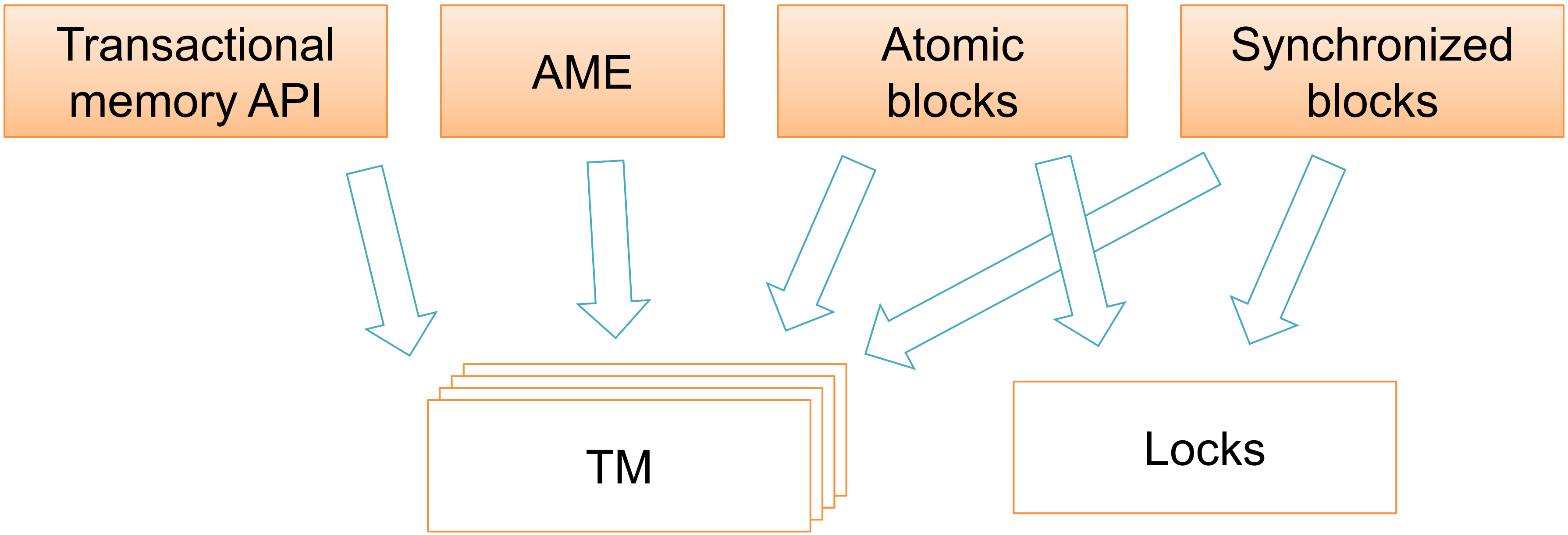
- Consequences explored methodically by Menon *et al* (Transact ’08, SPAA ’08)

Single Global Lock Semantics in a Weakly Atomic STM

icant performance overhead. As an alternative, many in the community have informally proposed that a *single global lock semantics* [16, 9], where transaction semantics are mapped to those of regions protected by a single global lock, provide an intuitive and efficiently implementable model for programmers.

In this paper, we explore the implementation and performance implications of single global lock semantics in a weakly atomic STM from the perspective of Java, and we discuss why even recent STM implementations fall short of these semantics. We describe a new weakly atomic Java STM implementation that provides single global lock semantics while permitting concurrent execution, but we show that this comes at a significant performance cost. We also

Abstractions vs implementations



Microsoft Research

Defining "atomic" without saying "TM"

- "Strong semantics"
  - Simple interleaved execution of threads
  - If a thread starts an atomic block then only it can take steps
  - Blocking operations (e.g. "retry", "orElse", "blockUntil") can be incorporated – see refs below
- This means:
  - Atomic blocks are atomic wrt normal memory accesses
  - Do not need to model conflict detection / resolution
  - Can choose whether or not to retain the effects of an atomic block that raises an exception

"Composable memory transactions", Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. PPoPP '05
"Semantics of Transactional Memory and Automatic Mutual Exclusion", Martín Abadi, Andrew Birrell, Tim Harris, Michael Isard. POPL '08

Example: a privatization idiom

```
x_shared = true;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Execution 1

Example: a privatization idiom

```
x_shared = true;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Execution 1

Microsoft Research

Example: a privatization idiom

```
x_shared = true;    x = 100;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Execution 1

Example: a privatization idiom

```
x_shared = false;    x = 100;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Execution 1

Microsoft Research

Example: a privatization idiom

```
x_shared = false;    x = 101;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Example: a privatization idiom

```
x_shared = true;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Execution 2

Example: a privatization idiom

```
x_shared = true;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Execution 2

Microsoft Research

Example: a privatization idiom

```
x_shared = false;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Execution 2

Example: a privatization idiom

```
x_shared = false;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Execution 2

Microsoft Research

Example: a privatization idiom

```
x_shared = false;   x = 1;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Strong semantics

- We’ve not talked about “inconsistent reads”, “roll backs”, “in-place vs lazy updates”, “weak atomicity”, “strong atomicity”, ...
- We’ve not ruled out anything (e.g. I/O)
- We’ve not considered program transformations

- Is this a pipe-dream? Can we implement it?

Microsoft Research

Untangling “atomic” from TM

Hiding TM from programmers

Current performance

Example: a privatization idiom

```
x_shared = true;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Microsoft Research

Not valid execution under strong semantics

Example: a privatization idiom

```
x_shared = true;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Not valid execution under strong semantics

Example: a privatization idiom

```
x_shared = true;    x = 100;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Not valid execution under strong semantics

Microsoft Research

Example: a privatization idiom

```
x_shared = false;    x = 100;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Microsoft Research

Not valid execution under strong semantics

Example: a privatization idiom

```
x_shared = false;    x = 101;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Not valid execution under strong semantics

Example: a privatization idiom

```
x_shared = false;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Not valid execution under strong semantics

Example: a privatization idiom

```
x_shared = false;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Hiding TM from programmers

Strong semantics

atomic, retry, ... What, ideally, should these constructs do?

Programming discipline(s)

What does it mean for a program to use the constructs correctly?

Low-level semantics & actual implementations

Transactions, lock inference, optimistic concurrency, program transformations, weak memory models, ...

Programming disciplines

- Based on a program’s execution under the strong semantics

All programs

Violation-free programs

Obeying dynamic separation

Obeying static separation

More programs satisfy the discipline

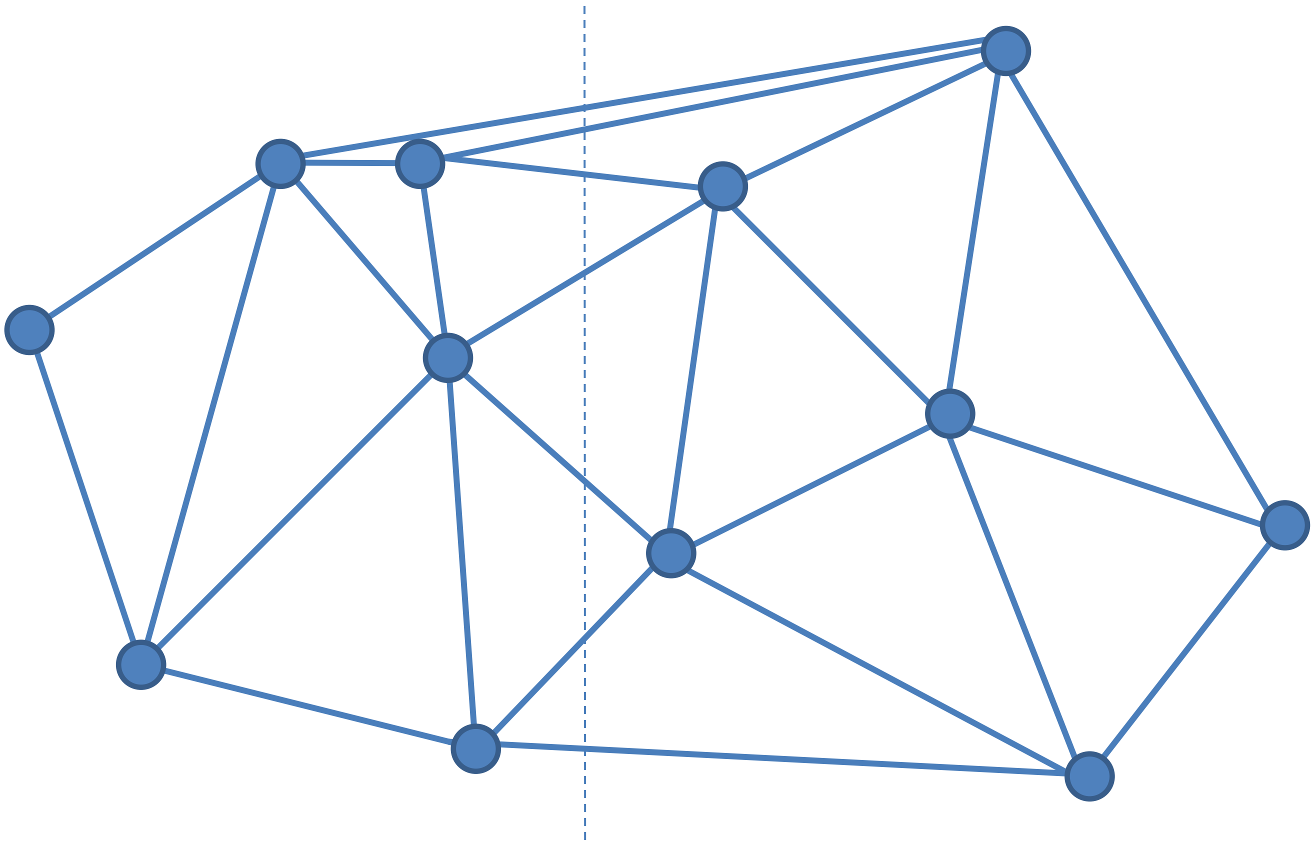
Fewer programs satisfy the discipline

Static separation

- Atomic blocks can only access local variables and designated “atomic variables”
- “atomic variables” cannot be accessed outside atomic blocks

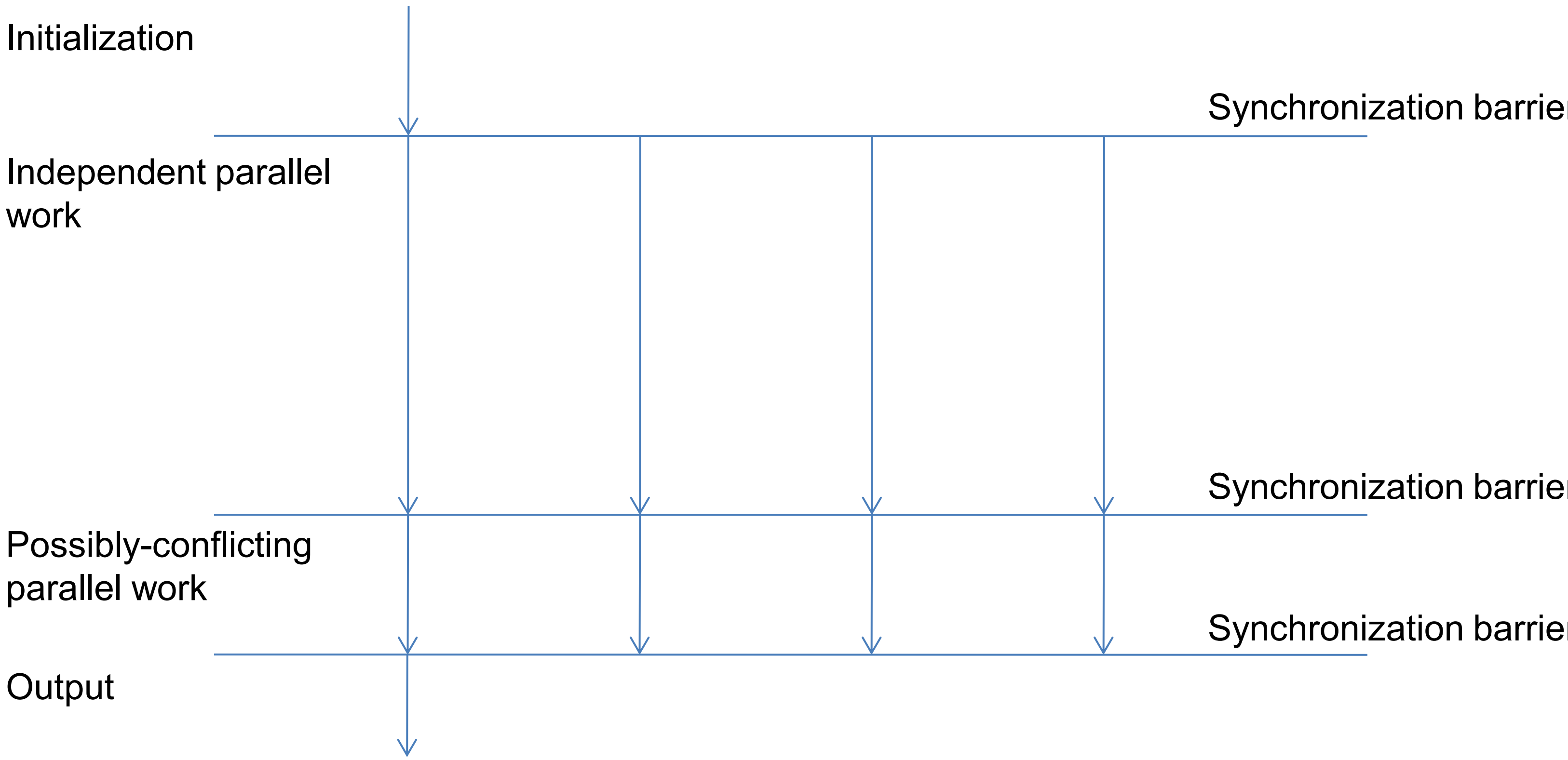
```
Class Q {
  atomic QElem leftSentinel;
  atomic QElem rightSentinel;

  void pushLeft(int item) {
    atomic {
      QElem e = new QElem(item);
      e.right = this.leftSentinel.right;
      e.left = this.leftSentinel;
```

Delaunay triangulation



“Delaunay Triangulation with Transactions and Barriers”
Michael L. Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe, IISWC 2007

Delaunay triangulation (2)



Dynamic separation

- Add explicit operations to indicate whether data is accessed inside atomic blocks, or accessed outside them
- Correctly synchronized: data is always in the correct mode when it is accessed
- Robust dynamic checking is possible:
  - Either the program runs with strong semantics
  - Or it fails with an error

“Implementation and use of transactional memory with dynamic separation”,
Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, Michael Isard, CC ’09 (to appear)

Violation freedom (VF)

- Allow data's access mode to change implicitly
- To be correctly synchronized:
  - Conflicting data accesses must not be attempted concurrently inside & outside atomic blocks
- Reminiscent of rules for programs to be free from data races

This example is violation free

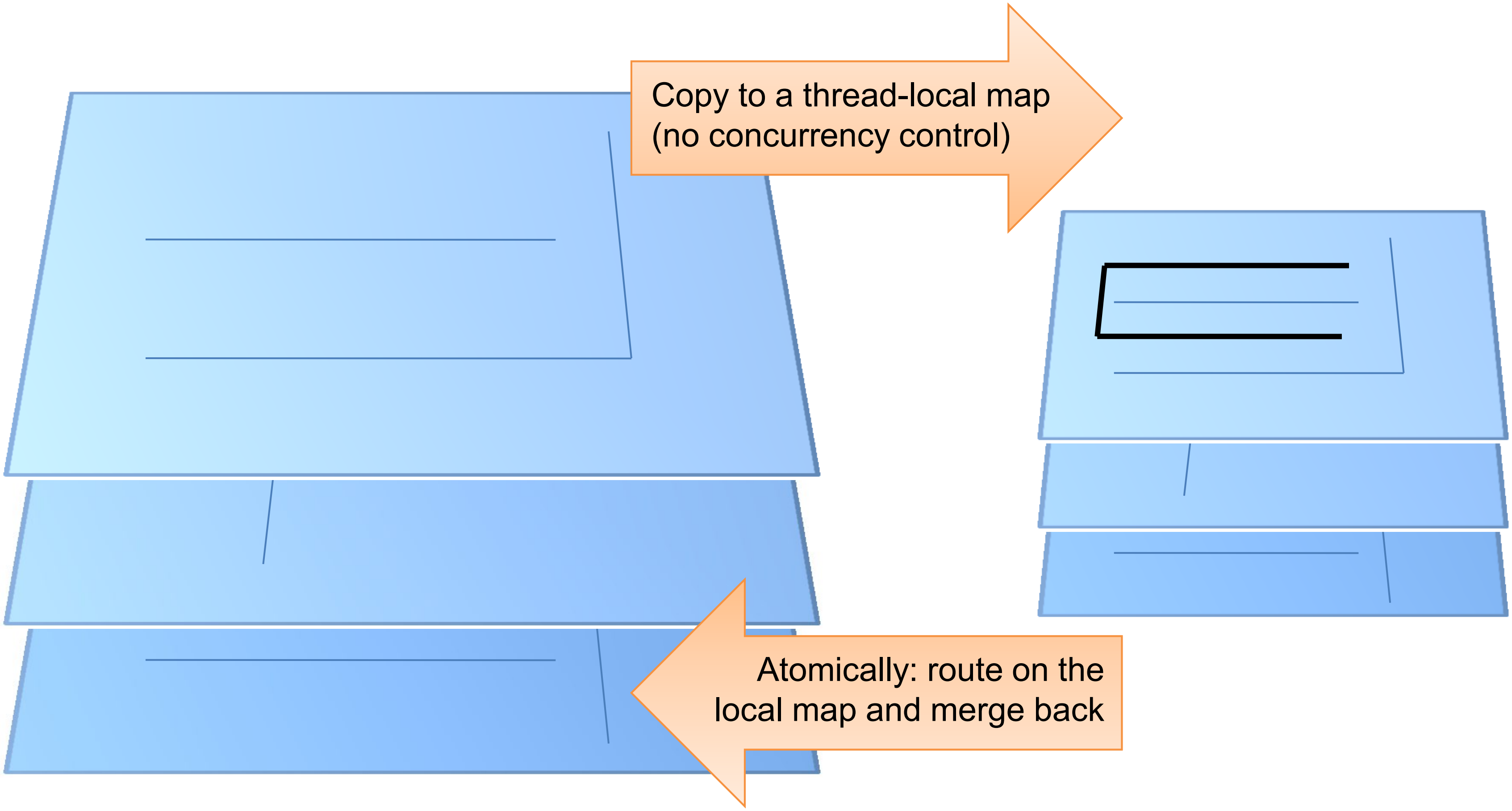
Example: a privatization idiom

```
x_shared = true;    x = 0;
```

```
atomic {
   if (x_shared) {
      x = 100;
   }
}
```

```
atomic {
   x_shared = false;
}
x++;
```

Programming with violations



C# version of Labyrinth, derived from "STAMP: Stanford Transactional Applications for Multi-Processing" Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun , IISWC '08

Strong atomicity

- Similar to typical HTM behavior
- Trade off implementation complexity for (hopefully) scalability & straight-line speed,
- Two recent approaches:
  - "Dynamic Optimization for Efficient Strong Atomicity", Schneider et al, OOPSLA '08
  - "Transactional memory with strong atomicity using off-the-shelf memory protection hardware", Abadi et al, PPoPP '09

Strong atomicity =/> strong semantics

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

```
tmp1 = ready;
if (tmp1 == true) {
    tmp2 = data;
}
```

- Can tmp1==true, tmp2==0?
- Under strong semantics: no
- Under plausible implementations with strong atomicity: yes

Example from “What do high-level memory models mean for transactions?”
Dan Grossman, Jeremy Manson, William Pugh, MSPC ’06

Microsoft Research

questions

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

                      t item) {

              QElem(item);
      e.right = this.leftSentinel.right;
      e.left = this.leftSentinel;
      this.leftSentinel.right.left = e;
      this.leftSentinel.right = e;
    }
  }
```

"Atomic blocks are for building shared memory data structures; use explicit synchronization for I/O

"In correctly synchronized programs, any use of speculation must be hidden by the implementation

"What about memory access violations, ex… erro…

"Again, ...

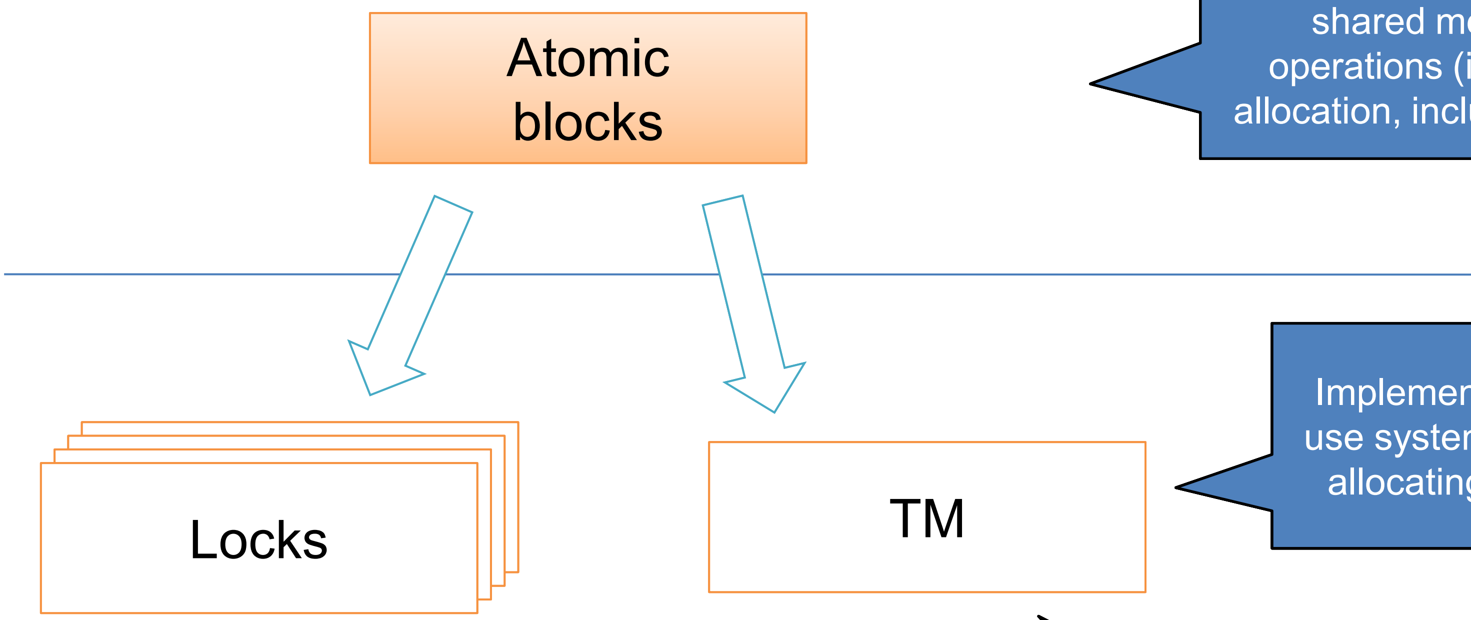
"In correctly synchronized programs, speculation won't be revealed by the debugger

"This depends on the language (Personally: no roll back, to avoid overhead on lock-inference impl's)

"If it's a conflicting access, then the program is not correctly synchronized

"Again, in correctly synchronized programs, speculation won't be revealed by the implementation

Open nesting, boosting, system calls



Programming abstraction is "atomic blocks". Just shared memory operations (including allocation, including GC).

Implementation may use system calls, e.g. allocating memory.

Open nesting, boosting are "TM-level" operations, possibly used in the implementation of allocation during atomic blocks. Mark uses as "unsafe" if explicit in applications.

Untangling “atomic” from TM

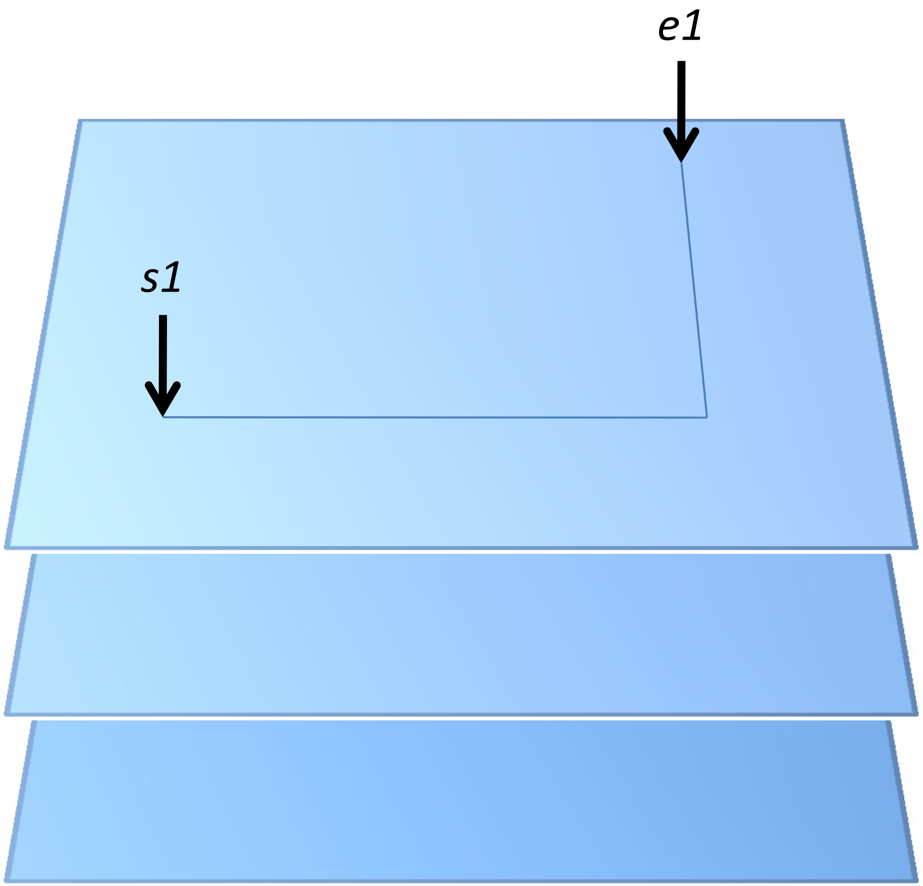
Hiding TM from programmers

Current performance

Perf. 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? Strong atomicity?
- 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

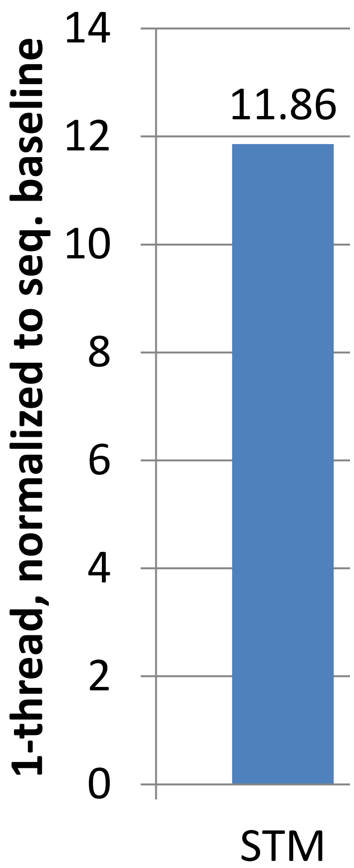
Labyrinth



- 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

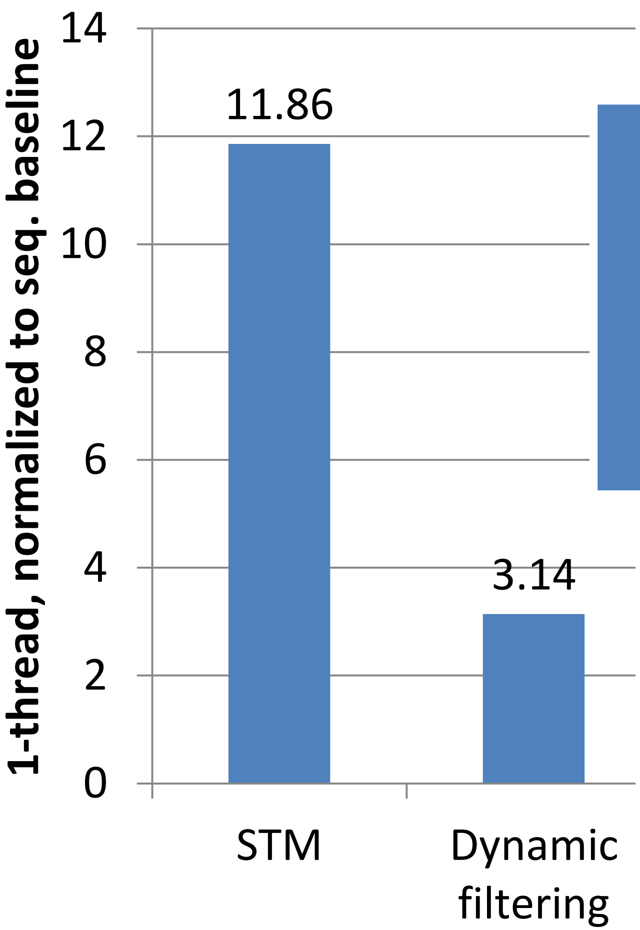
“STAMP: Stanford Transactional Applications for Multi-Processing”
Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun , IISWC 2008

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

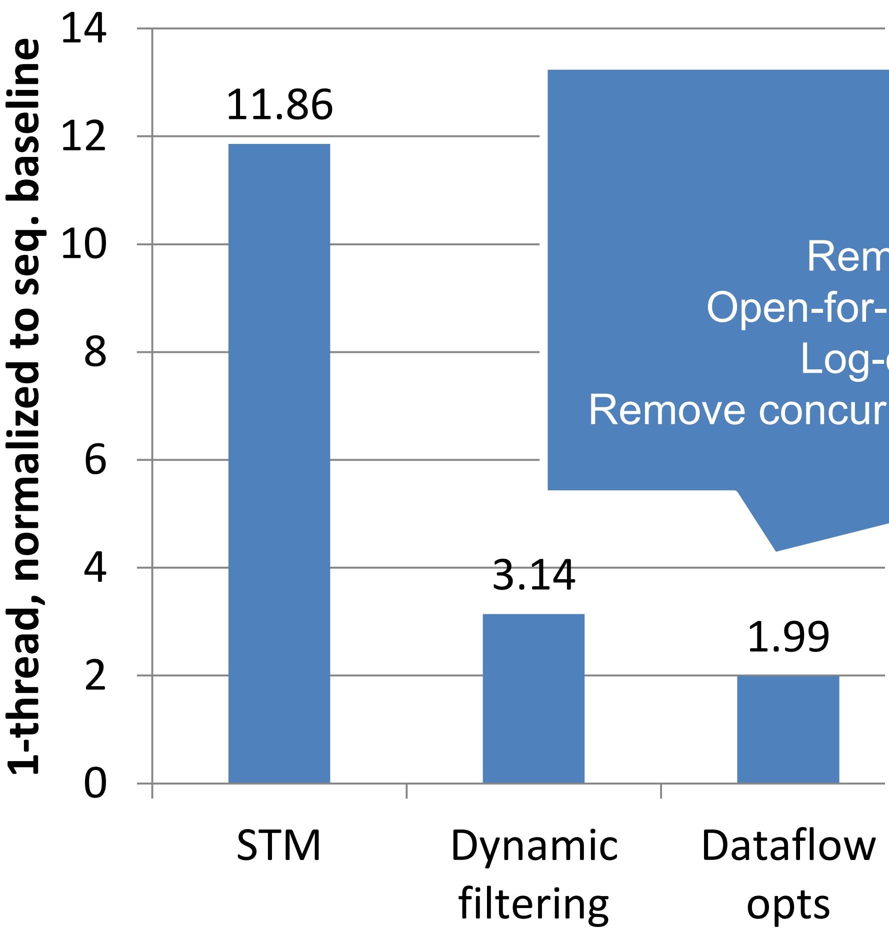
Sequential overhead



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

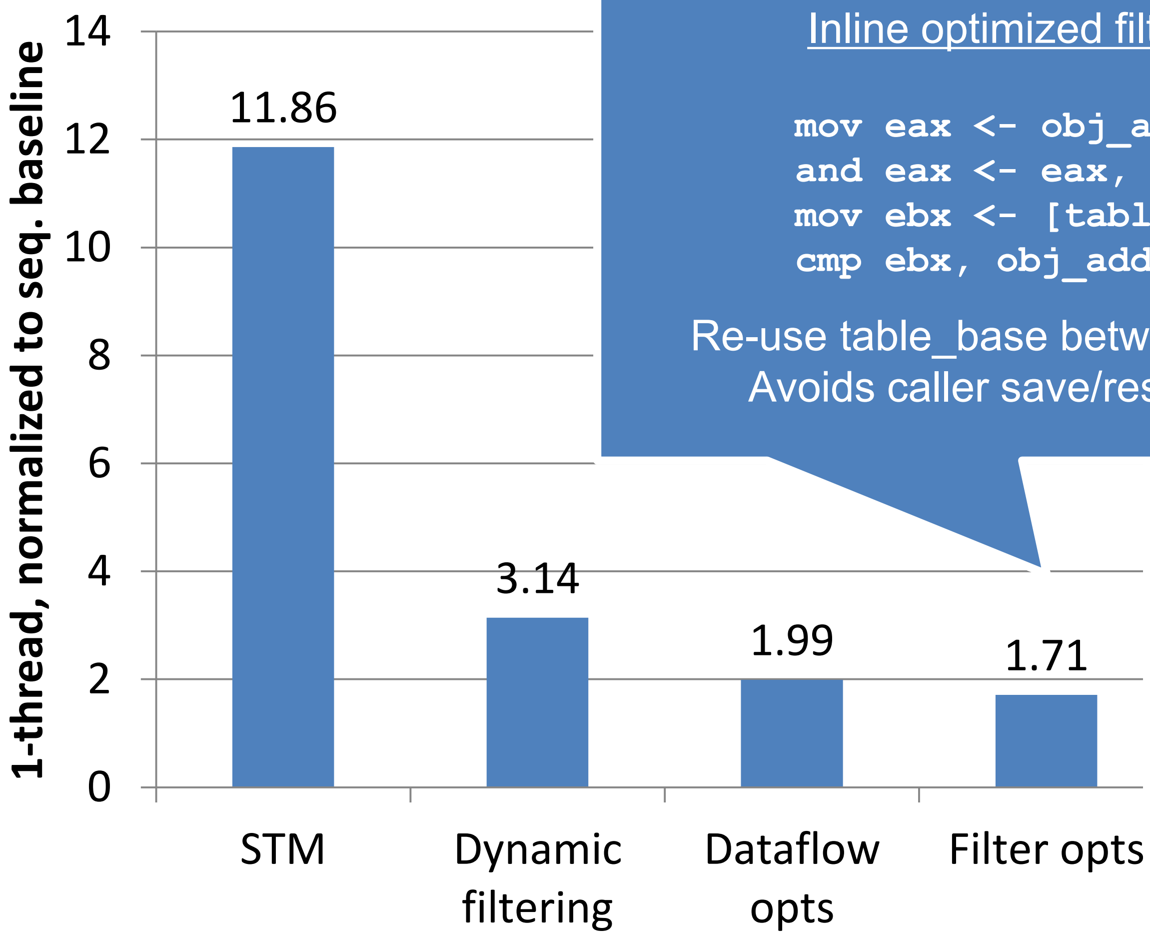
Sequential overhead



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



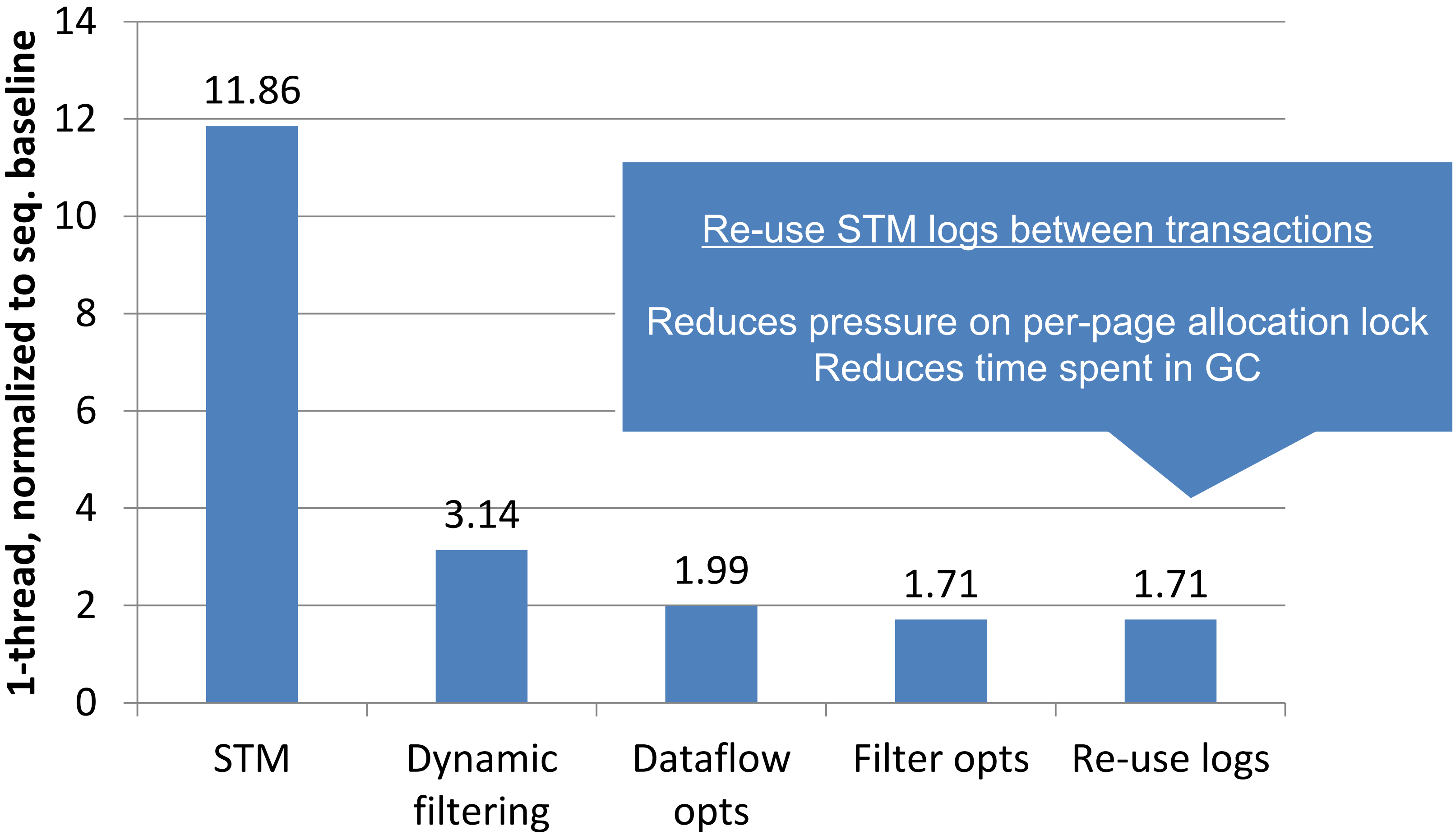
<u>Inline optimized filter operations</u>

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

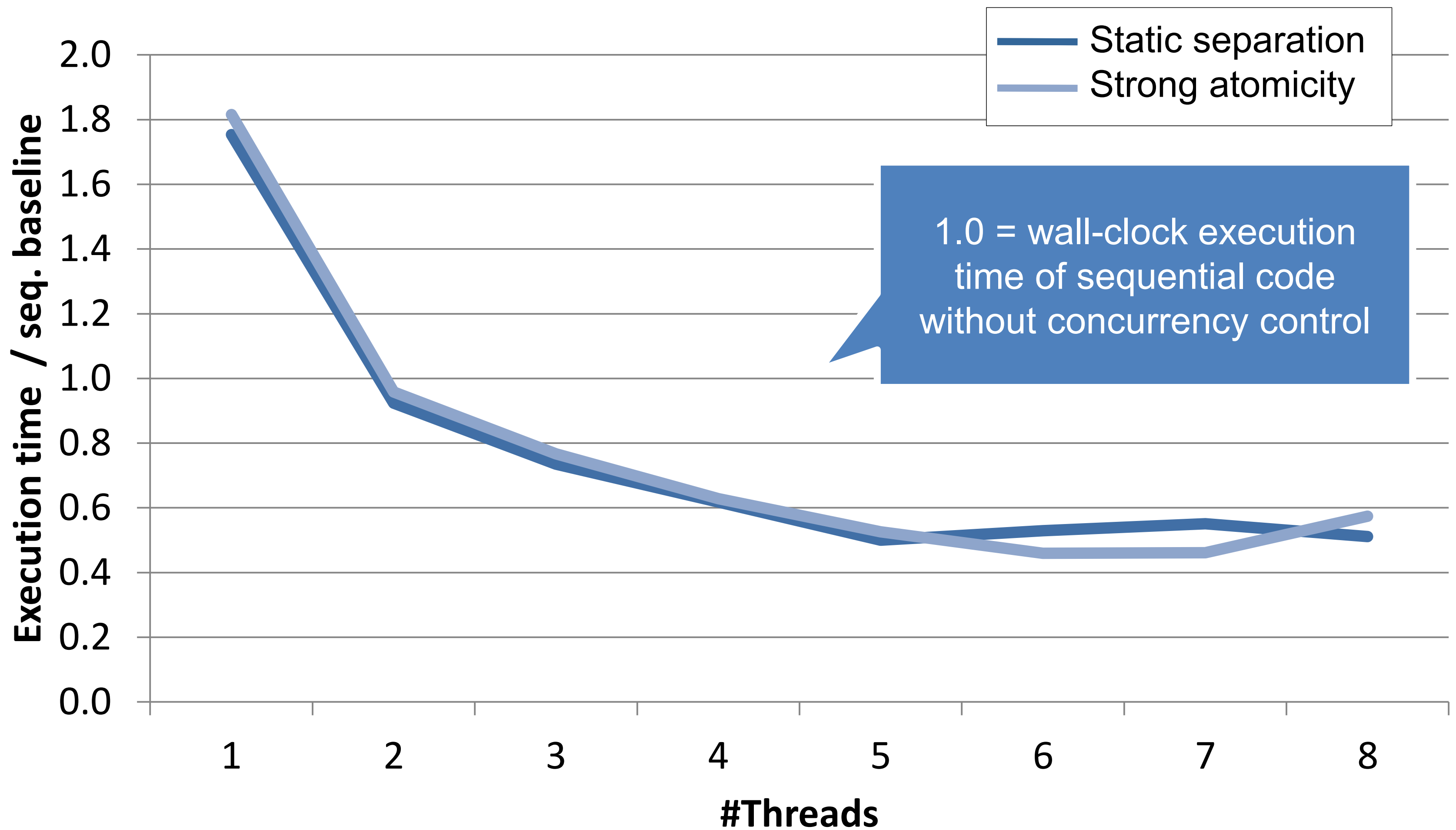
Re-use table_base between filter operations

Avoids caller save/restore on filter hits

Sequential overhead

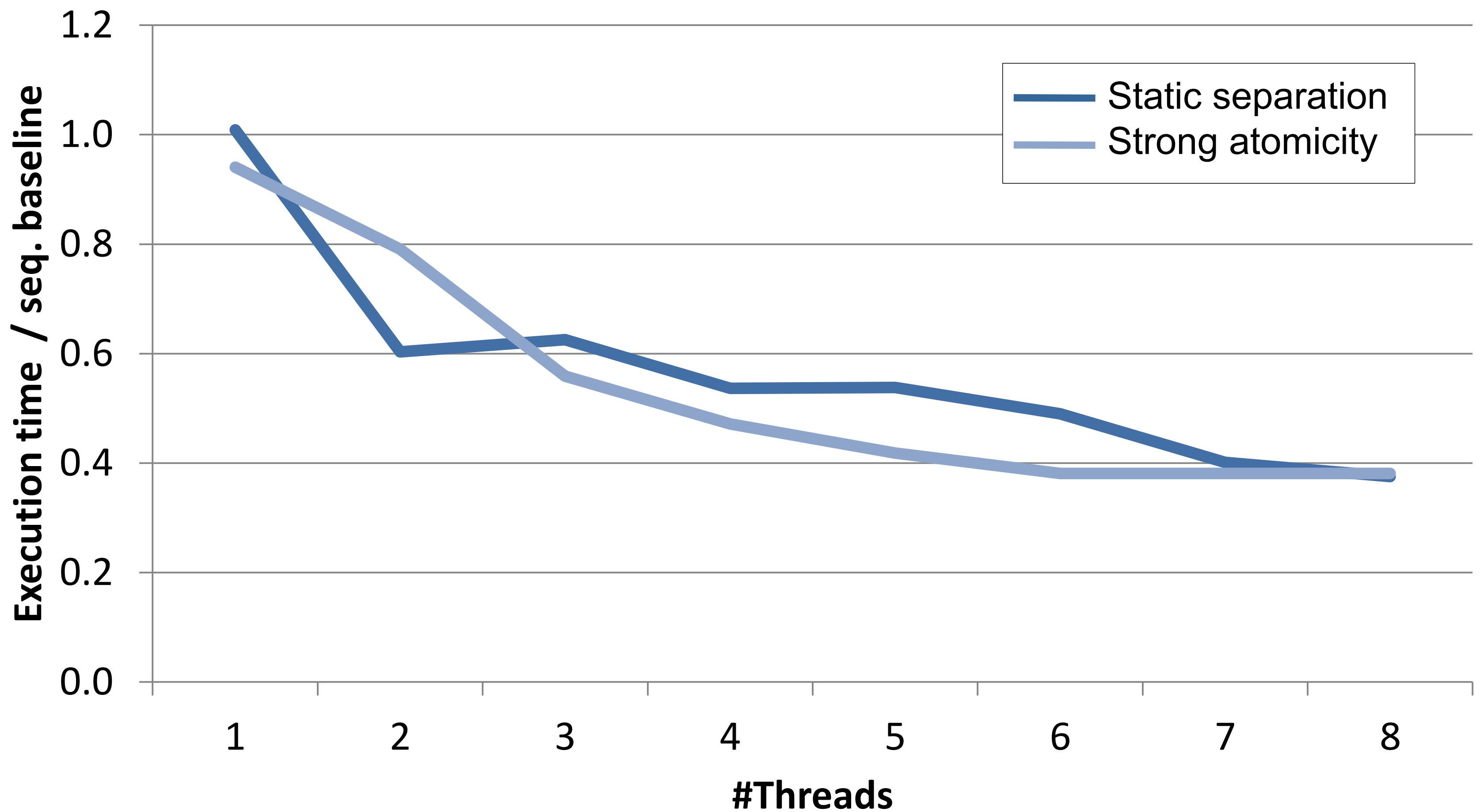


Scaling – Labyrinth

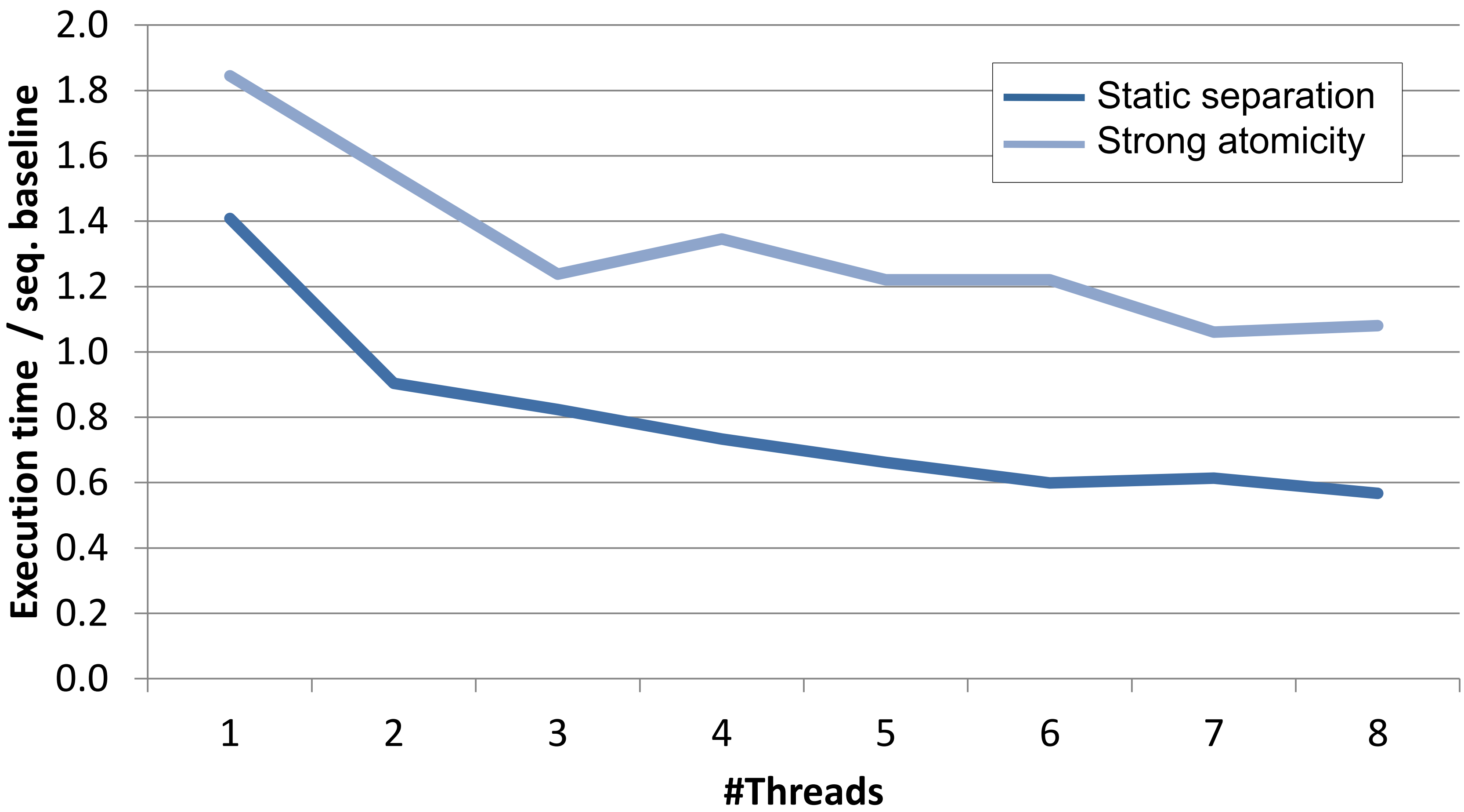


Microsoft Research

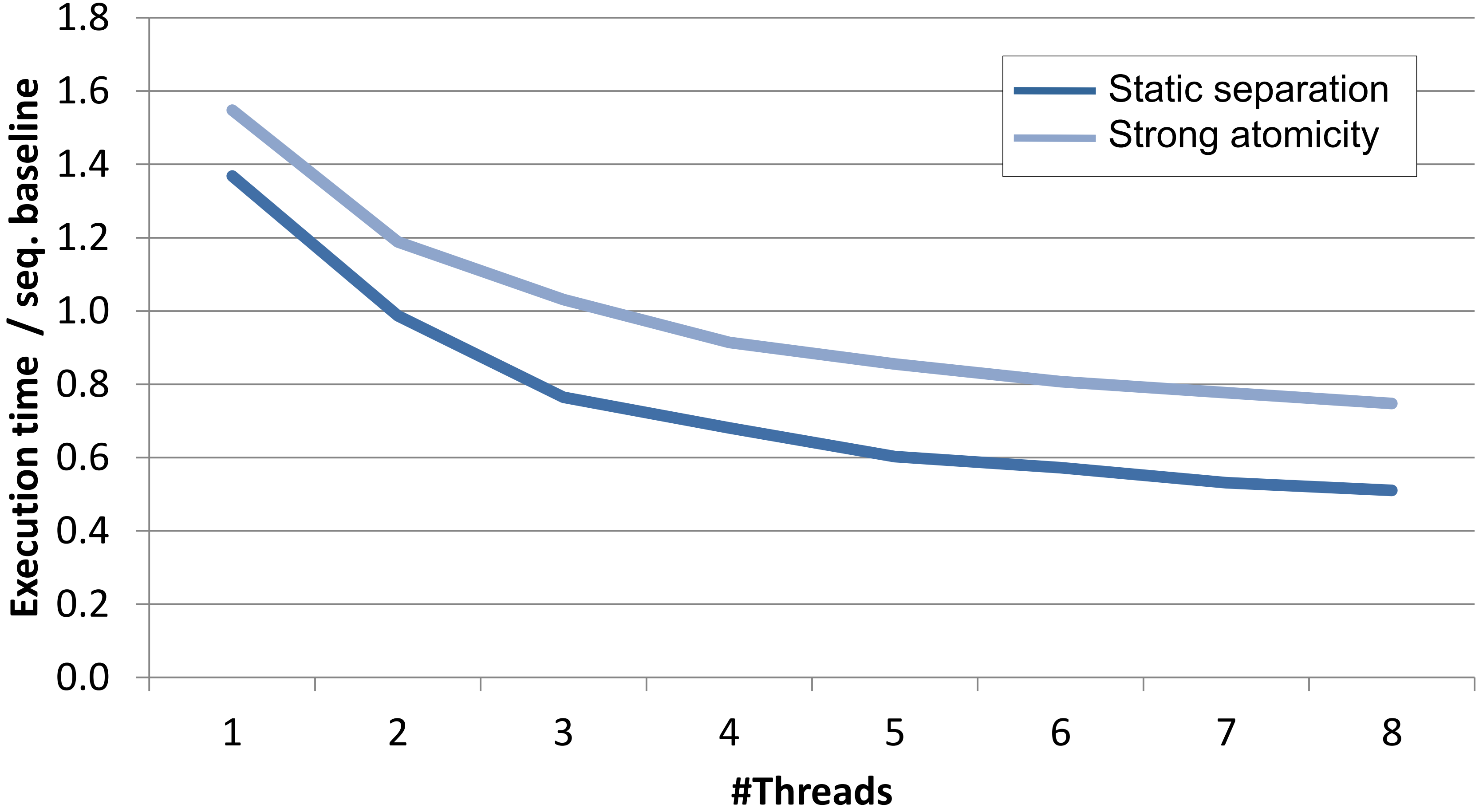
Scaling – Delaunay



Scaling – Genome



Scaling – Vacation

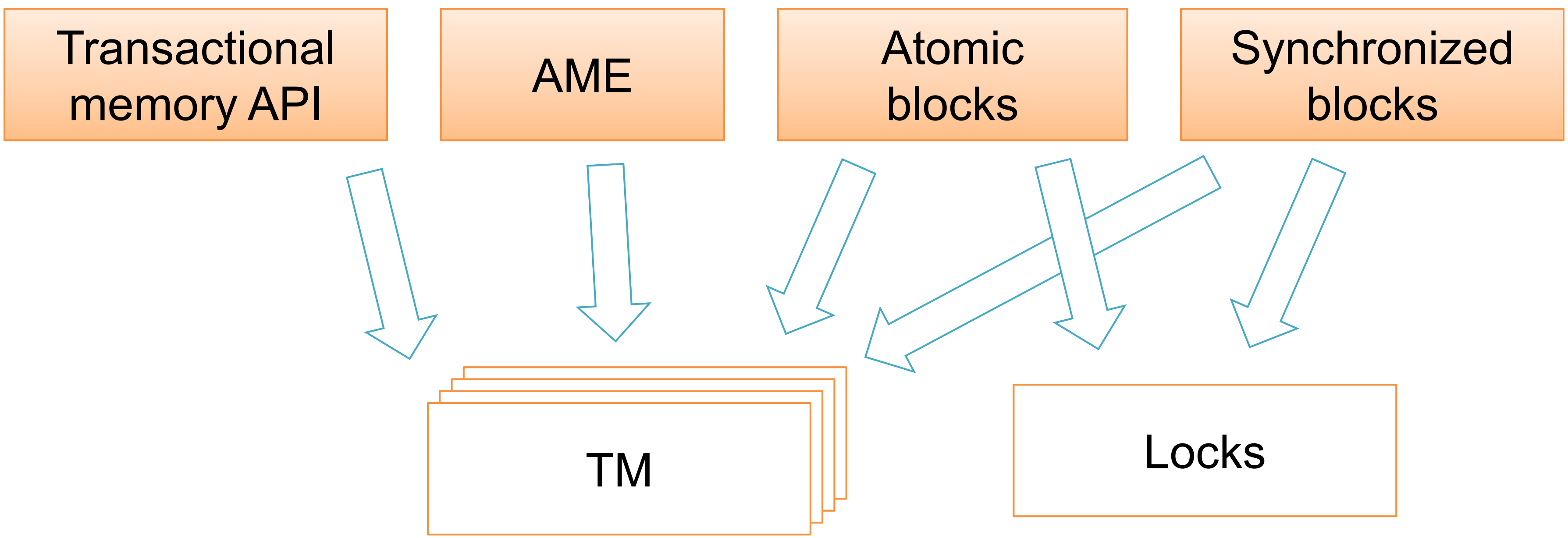


Untangling “atomic” from TM

Hiding TM from programmers

Current performance

Abstractions vs implementations



Future directions

- Which programming discipline should we settle on
  - ...in a language like C#?
  - ...in future languages?
- H/W acceleration based on mature optimized S/W implementations
- Progress guarantees, interactions with implementation techniques and performance
- What asymptotic bounds on STM performance can we give when supporting different programming disciplines?
- How do we define correctness of an STM interface, as opposed to the whole language implementation?

Acnkowledgements

- Most of the work described in these slides has been collaborative; I'd like to thank colleagues at MSR Cambridge, MSR Redmond, MSR Mountain View, the Microsoft Parallel Computing Platform Group, the University of Cambridge Computer Lab, and the MSR-BSC joint research centre.

- Material is drawn from the following publications:

  – Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, Michael Isard. "Implementation and use of transactional memory with dynamic separation". *Compiler Construction, March 2009*
  – Martín Abadi, Tim Harris, Mojtaba Mehrara. "Transactional memory with strong atomicity using off-the-shelf memory protection hardware". *PPoPP, February 2009*
  – Martín Abadi, Tim Harris, Katherine Moore. "A model of dynamic separation for transactional memory". *CONCUR, August 2008*
  – Martín Abadi, Andrew Birrell, Tim Harris, Michael Isard. "Semantics of Transactional Memory and Automatic Mutual Exclusion". *POPL, January 2008*
  – Keir Fraser, Tim Harris. "Concurrent programming without locks". *ACM TOCS, May 2007*
  – Tim Harris, Mark Plesko, Avraham Shinnar, David Tarditi. "Optimizing Memory Transactions" *PLDI*, June 2006
  – Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. "Composable memory transactions" *PPoPP*, June 2005
  – Tim Harris, Keir Fraser. "Language Support for Lightweight Transactions". *OOPSLA,* October 2003

- The material on performance is current at Jan 2009, and reflects a slightly later more optimized implementation than that described in the PPoPP 2009 paper