

# Language constructs for transactional memory

Tim Harris

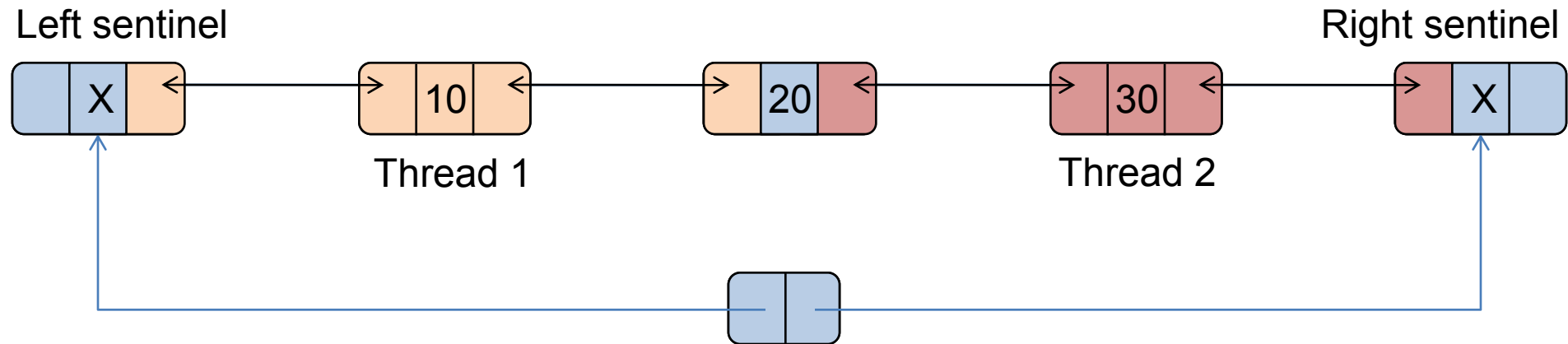
Disclaimer: these are my personal opinions

# 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());
    }

    ...
}
```

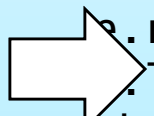
# Implementing this: atomic blocks

```

Class Q {
  QElem l
  QElem r

  void pu
  atomi
  QEl
  .r
  .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());
  }
  ...
}

```



# “Atomic blocks are transactions”

consider how the problem of mutual exclusion through message systems or databases, not to address distributed systems.

Even in this setting, concurrent programming is extremely difficult. The dominant programming technique is based on locks, an approach that is simple and direct, but that simply does not scale with program size and complexity. To ensure correctness, programmers must identify which operations conflict, to ensure fairness, they must avoid introducing deadlock, to ensure good performance, they must balance the granularity of which locking is performed against the costs of fine-grained locking. Perhaps the most fundamental objection, though, is that lock-based programs do not compose: correct fragments may fail when combined.

For example, consider a bank table with thousands of rows and update operations. First suppose that we need to debit one row A from table *tbl*, and insert into table *tbl2*, but the intermediate state in which neither table contains the debit must not be visible to other threads. Unless the implementor of the lock table anticipates this need, there is simply no way to satisfy this requirement. Even if, at the time, all the user-visible update methods such as `tbl.debit()` and `tbl2.credit()` — but as well as handling the bookkeeping abstraction, they create lock-induced deadlock, depending on the order in which the client takes the locks, or even conditions if the client depends for more completion to happen one, another, they must be careful the presence of A in *tbl*, but this blocking behavior must not lock the table (since A cannot be re-used). In short, operations that are individually correct (insert, debit) cannot be composed into larger correct operations.

The same phenomenon shows up trying to compose alternative locking operations. Suppose a procedure `get` waits for one of two input pipes to have data, using an internal call to the client's `waiter` procedure, and suppose another procedure `put` does the same thing, on two different pipes. In Unix there is no way to perform a `select` between `get` and `put`, a fundamental loss of composability. Instead, Unix programmers have invented programming techniques to gather up all of the descriptors that must be waited for, perform a single `select` on all of them, and then dispatch back to the correct handler. Again, two individually-correct abstractions, `get` and `put`, cannot be composed into a larger one, unless they must be re-ordered and arbitrarily merged, in direct conflict with the goals of abstraction.

Rather than being locks, a more promising and radical alternative is to have concurrency control on atomic memory transactions, also known as transactional memory. We will show that transactional memory offers a solution to the tension between concurrency and abstraction. For example, with memory transactions we can manipulate the bank table like:

```
atomic { withdraw(tbl, A); insert(tbl2, A); }
```

and to wait for either `get` or `put` we can say

```
waiter { get; waiter; put; }
```

These simple constructions require no knowledge of the implementation of `waiter`, `debit`, `get`, or `put`, and they continue to work correctly if these operations may block, as we shall see.

### 2.1 Transactional memory

The idea of transactions is not new: they have been a fundamental mechanism in database design for many years, and there has been much recent work on transactional memories [1, 10, 9, 6, 11].

The key idea is that a block of code, including nested calls, can be executed by an atomic block, with the guarantee that it runs atomically with respect to every other atomic block. Transactional memory can be implemented using optimistic read-write memory instead of using locks, so atomic blocks may without locking, maintaining a three-valued semantics for that records every memory read and write it makes. When the block completes, it first updates

its log, to check that it has seen a consistent view of memory, and then commits its changes to memory. If validation fails, because memory read by the writer was altered by another thread during the block's execution, then the block is re-executed from scratch.

Transactional memory abstracts, by construction, many of the low-level difficulties that plague lock-based programming [1]. There are no lock-induced deadlocks (because there are no locks); 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.

Finally, 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 if it were re-executed. It might also launch one de-scheduled after the thread modified both legs for a processor performs memory operations in an order different from the order block is a disaster for which the two threads has to maintain. If several operations occur, they may be interleaved in a way that is not intended.

```
Done get() {
  atomic { n, k;
  }
```

The thread waits until reading the block. In its second call `get()` finds an interesting `n` in a nested access to `n` from the other block is a disaster for which the two threads has to maintain. If several operations occur, they may be interleaved in a way that is not intended.

Thirdly, no process compiled by the `gcc` has 7.2 on C++ memory by generating memory the language C++.

### 2.2 Composable lock-free, functional legs

abstractions for some values also suffice as necessary to be able to read through cache lines in the key of what performed, one.

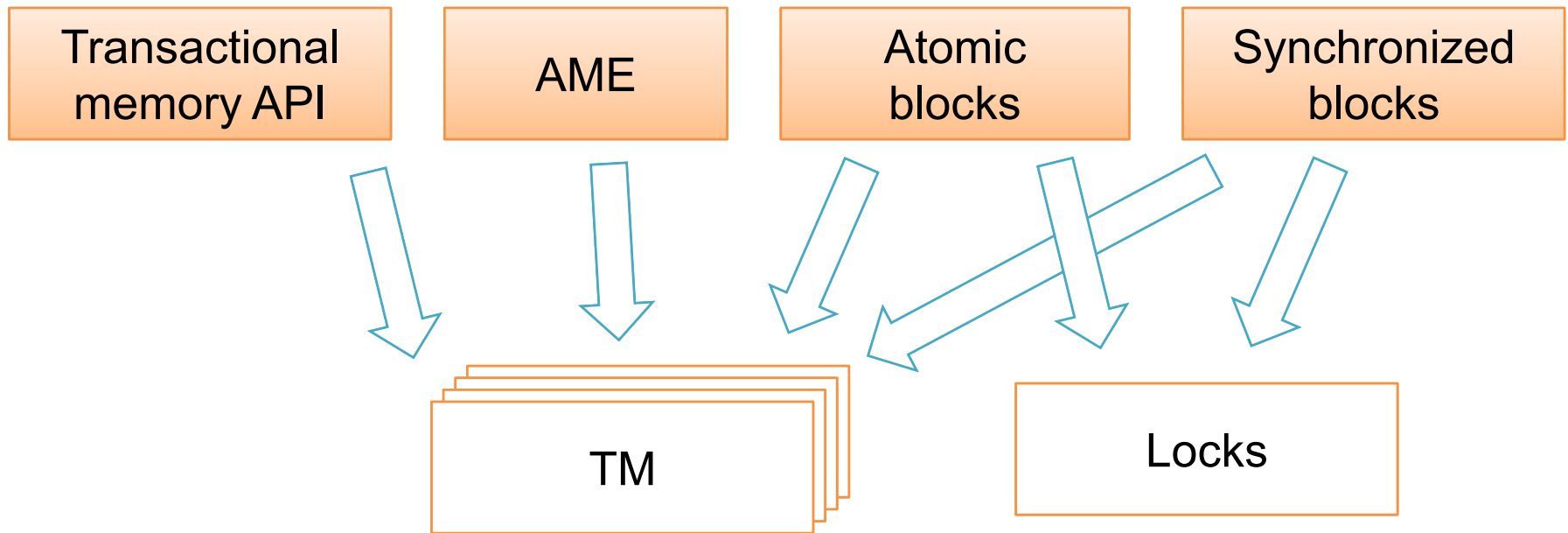
```
4. For example, the
particular 11: the
getChar 11: 11: the
```

That is, `getChar` takes a `Char` and delivers an `IO` action that, when performed, prints the character on the standard output, while `putChar` is an action that, when performed, reads a character from the console and delivers it as the result of the action. A complete

There are no lock-induced deadlocks (because there are no locks); 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 thread modified both before the thread was resumed. This problem



# Abstractions vs implementations

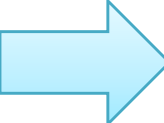


# Defining “atomic” without saying “TM”

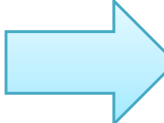
- “Strong semantics”
  - Simple interleaved execution of threads
  - If a thread starts an atomic block then only it can take steps
  - Blocking operations (e.g. “retry”, “orElse”, “blockUntil”) can be incorporated – see refs below
- This means:
  - Atomic blocks are atomic wrt normal memory accesses
  - 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

# Example: a privatization idiom

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



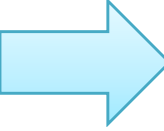
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



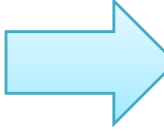
```
atomic {  
    x_shared = false;  
}  
x++;
```

# Example: a privatization idiom

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



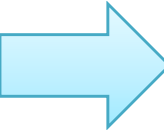
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



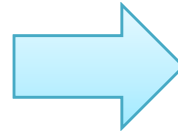
```
atomic {  
    x_shared = false;  
}  
x++;
```

# Example: a privatization idiom

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



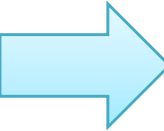
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



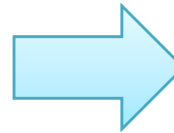
```
atomic {  
    x_shared = false;  
}  
x++;
```

# Example: a privatization idiom

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



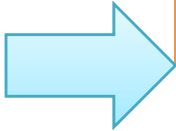
```
atomic {  
  if (x_shared) {  
    x = 100;  
  }  
}
```



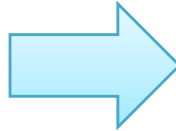
```
atomic {  
  x_shared = false;  
}  
x++;
```

## Example: a privatization idiom

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



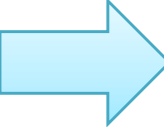
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



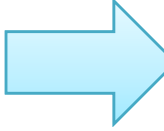
```
atomic {  
    x_shared = false;  
}  
x++;
```

# Example: a privatization idiom

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



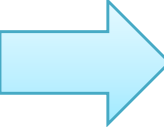
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



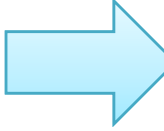
```
atomic {  
    x_shared = false;  
}  
x++;
```

## Example: a privatization idiom

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



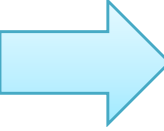
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



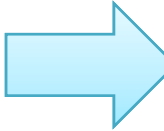
```
atomic {  
    x_shared = false;  
}  
x++;
```

## Example: a privatization idiom

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



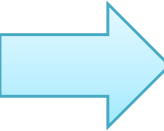
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



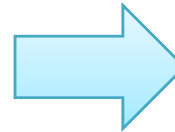
```
atomic {  
    x_shared = false;  
}  
x++;
```

## Example: a privatization idiom

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



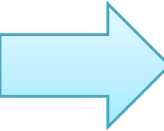
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



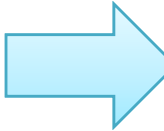
```
atomic {  
    x_shared = false;  
}  
x++;
```

## 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?

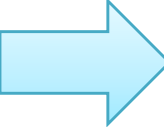
Untangling “atomic” from TM

**Hiding TM from programmers**

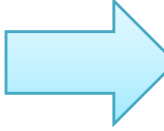
Current performance

# Example: a privatization idiom

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



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

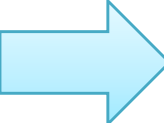


```
atomic {  
    x_shared = false;  
}  
x++;
```

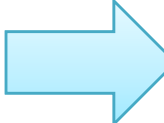
Not valid  
execution under  
strong semantics

## Example: a privatization idiom

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



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

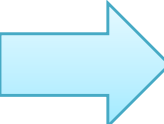


```
atomic {  
    x_shared = false;  
}  
x++;
```

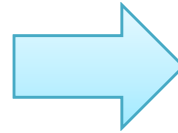
Not valid  
execution under  
strong semantics

## Example: a privatization idiom

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



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

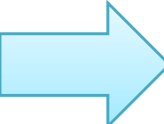


```
atomic {  
    x_shared = false;  
}  
x++;
```

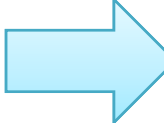
Not valid  
execution under  
strong semantics

## Example: a privatization idiom

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



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

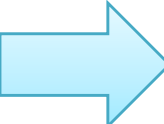


```
atomic {  
    x_shared = false;  
}  
x++;
```

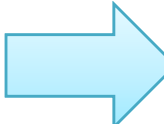
Not valid  
execution under  
strong semantics

## Example: a privatization idiom

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



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

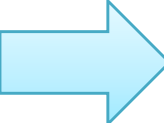


```
atomic {  
    x_shared = false;  
}  
x++;
```

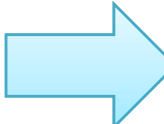
Not valid  
execution under  
strong semantics

## Example: a privatization idiom

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



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

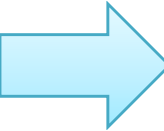


```
atomic {  
    x_shared = false;  
}  
x++;
```

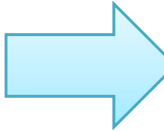
Not valid  
execution under  
strong semantics

## 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

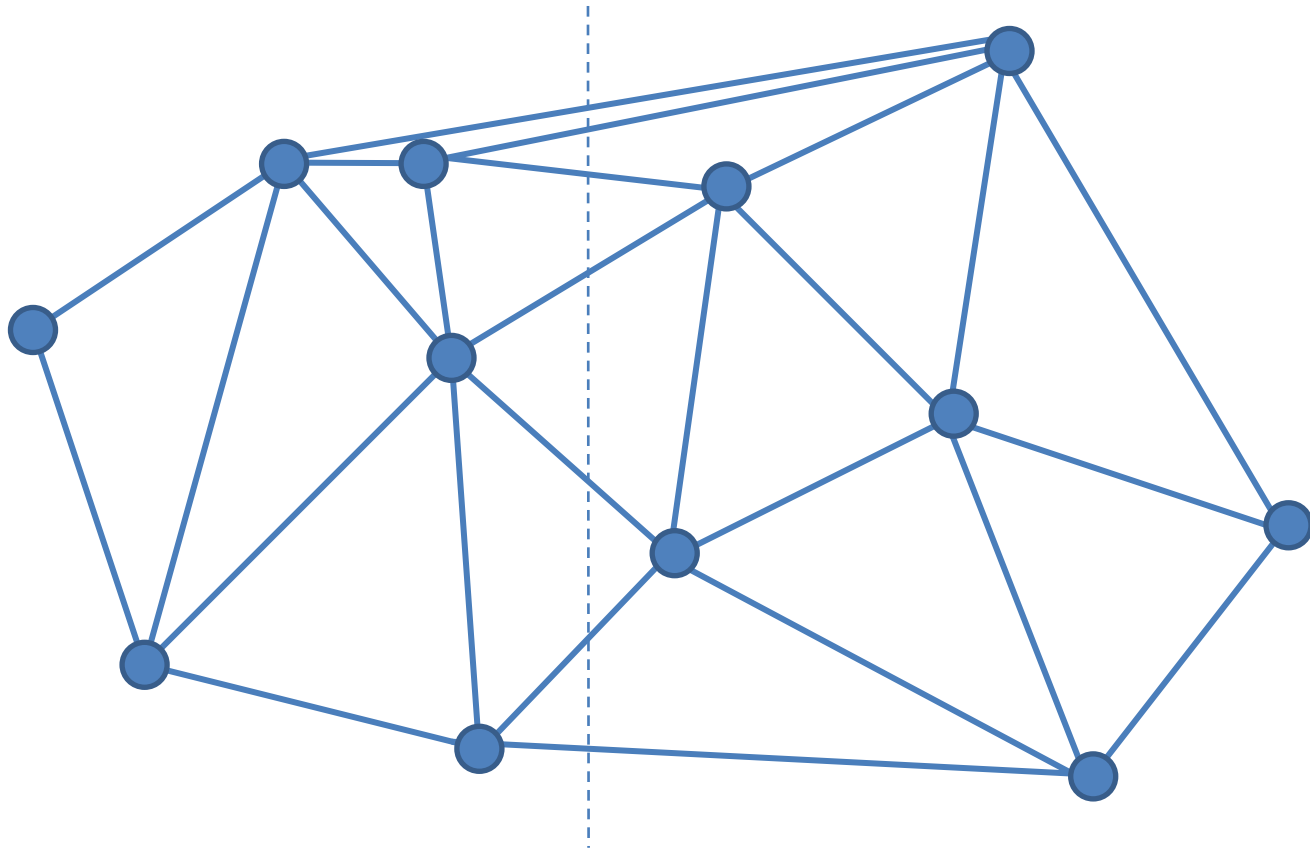
## Static separation

- 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;
            this.leftSentinel.right = e;
        }
    }
}
```

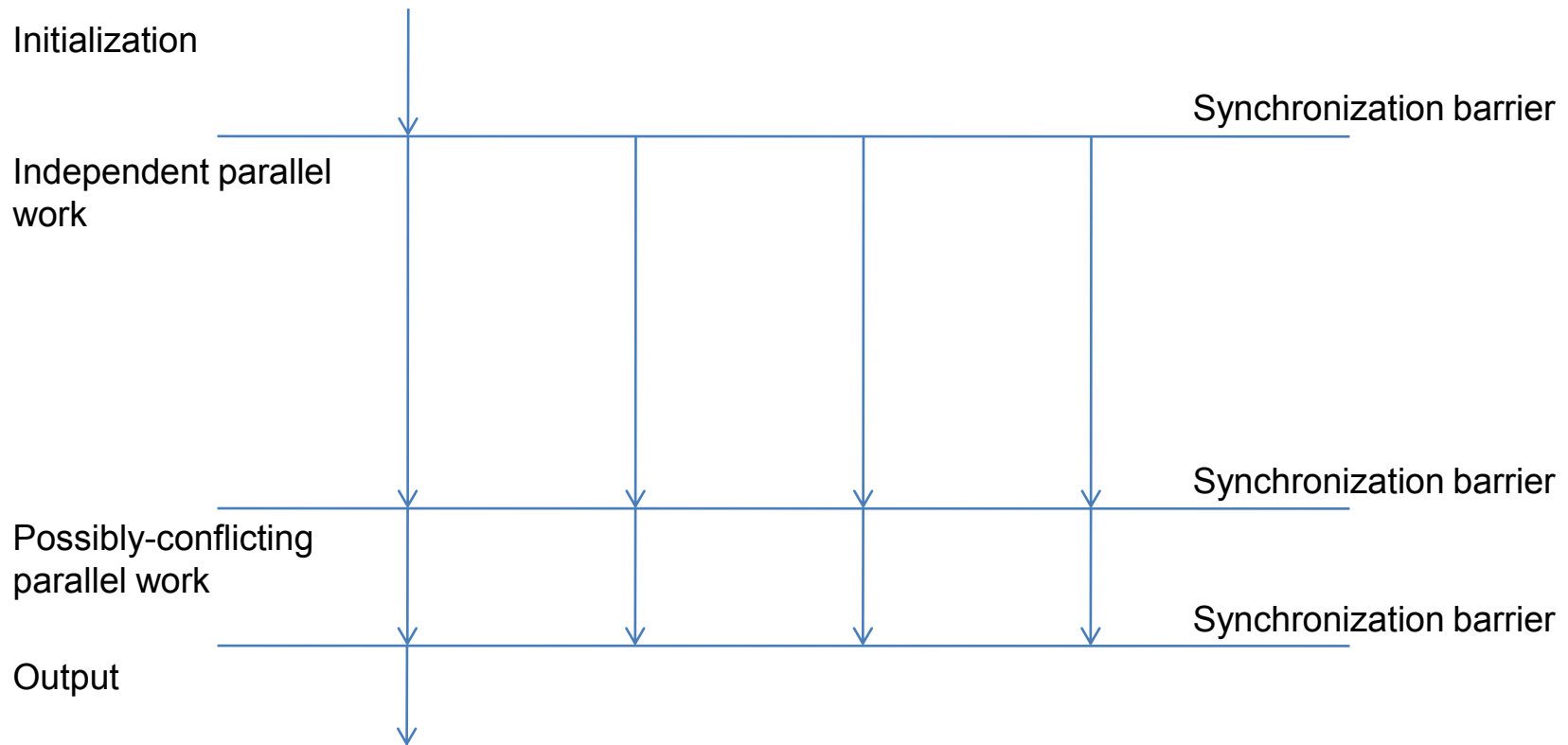
# Delaunay triangulation



“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

## 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

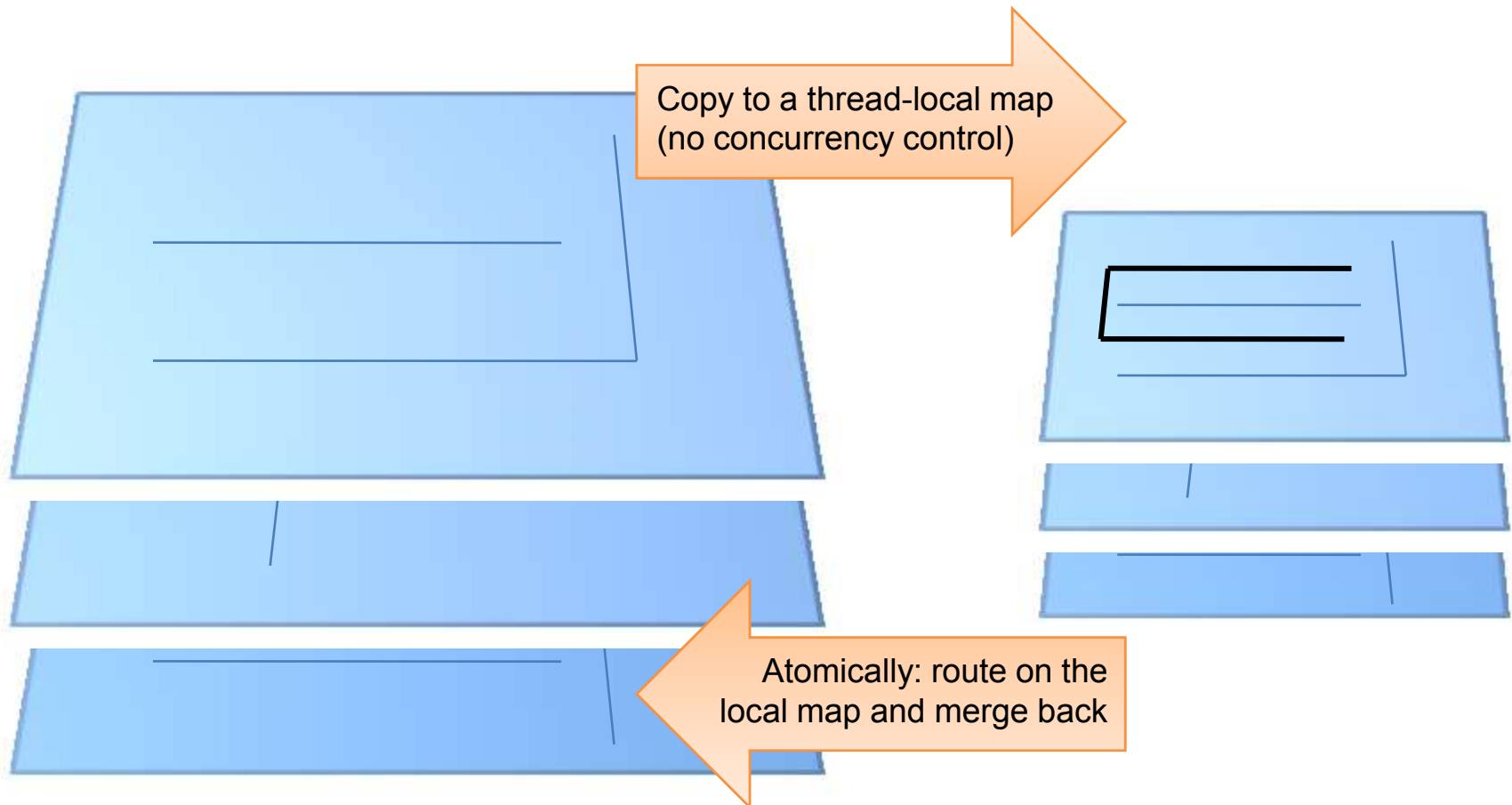
## Example: a privatization idiom

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

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

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

# Programming with violations



## 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 $\neq$ 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

# questions

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

```
class Q {  
    QElem leftSentinel;  
    QElem rightSentinel;  
    ...  
    void enqueue(QElem item) {  
        ...  
        QElem e;  
        e.right = this.leftSentinel.right;  
        e.left = this.leftSentinel;  
        this.leftSentinel.right.left = e;  
        this.leftSentinel.right = e;  
    }  
}
```

“What about memory access violations, e.g. error

“Again, ...

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

“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

Atomic blocks

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

Locks

TM

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

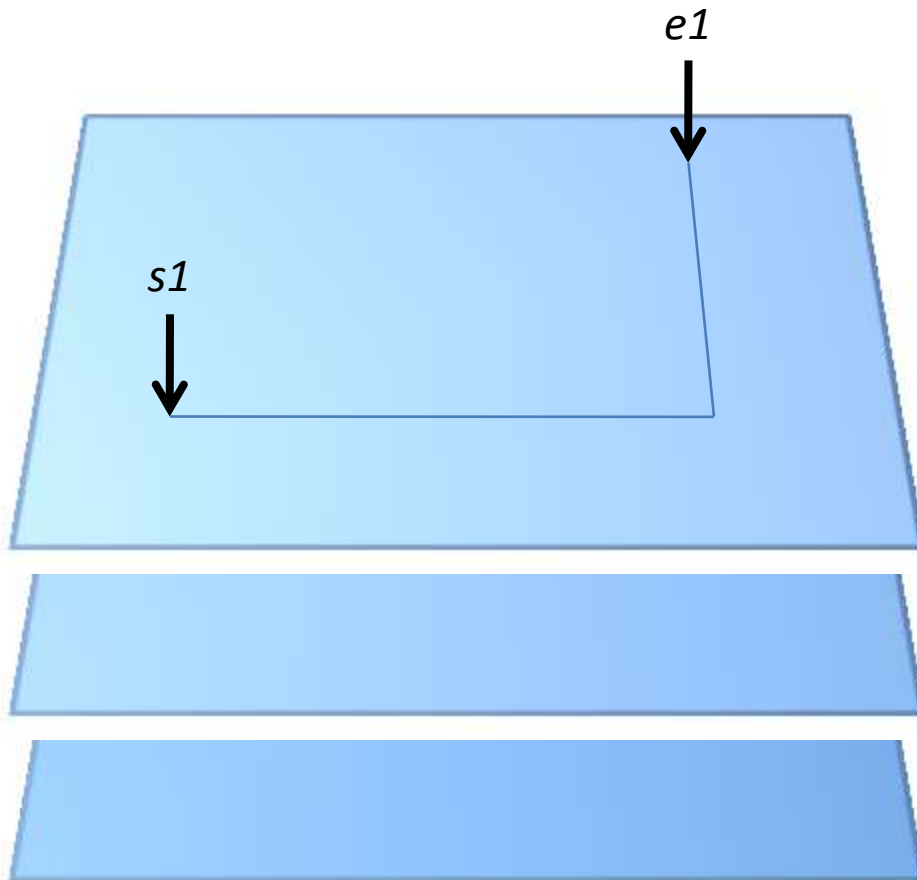
Hiding TM from programmers

**Current performance**

# Perf. figures depend on...

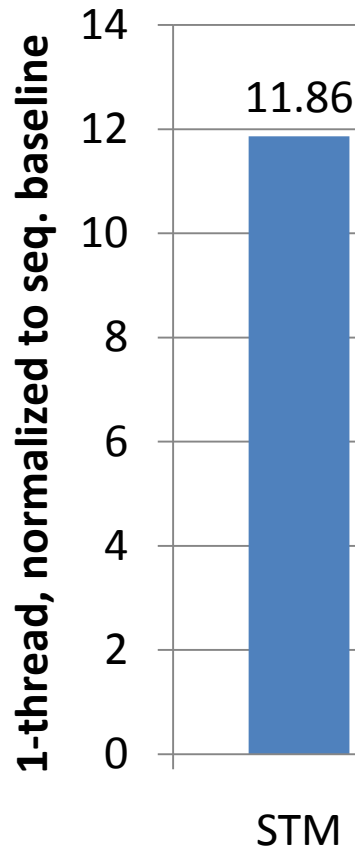
- **Workload** : What do the atomic blocks do? How long is spent inside them?
- **Baseline implementation**: Mature existing compiler, or prototype?
- **Intended semantics**: Support static separation? Violation freedom? Strong atomicity?
- **STM implementation**: In-place updates, deferred updates, 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



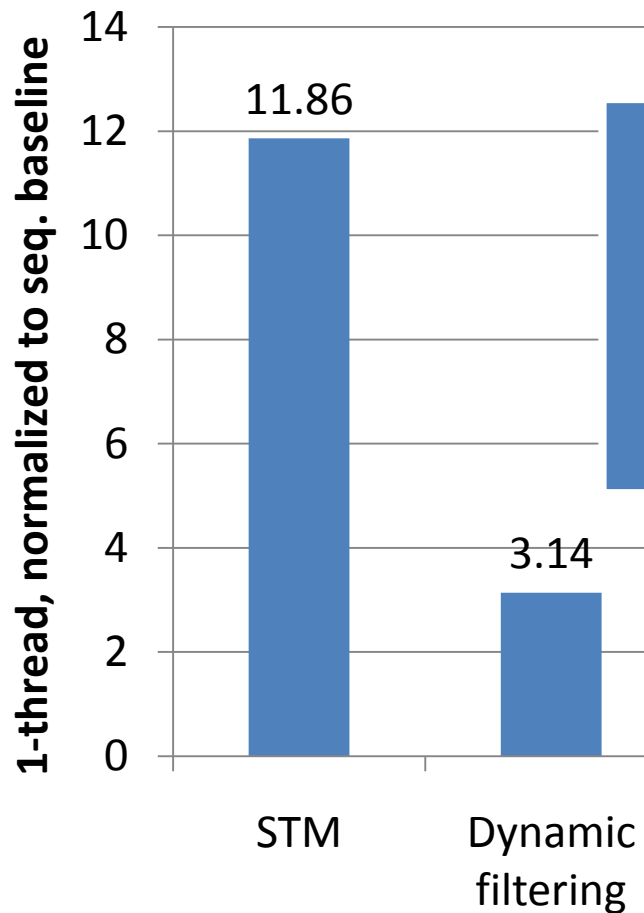
- STAMP v0.9.10
- 256x256x3 grid
- Routing 256 paths
- Almost all execution inside atomic blocks
- Atomic blocks can attempt 100K+ updates
- C# version derived from original C
- Compiled using Bartok, whole program mode, C# -> x86 (~80% perf of original C with VS2008)
- Overhead results with Core2 Duo running Windows Vista

# 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

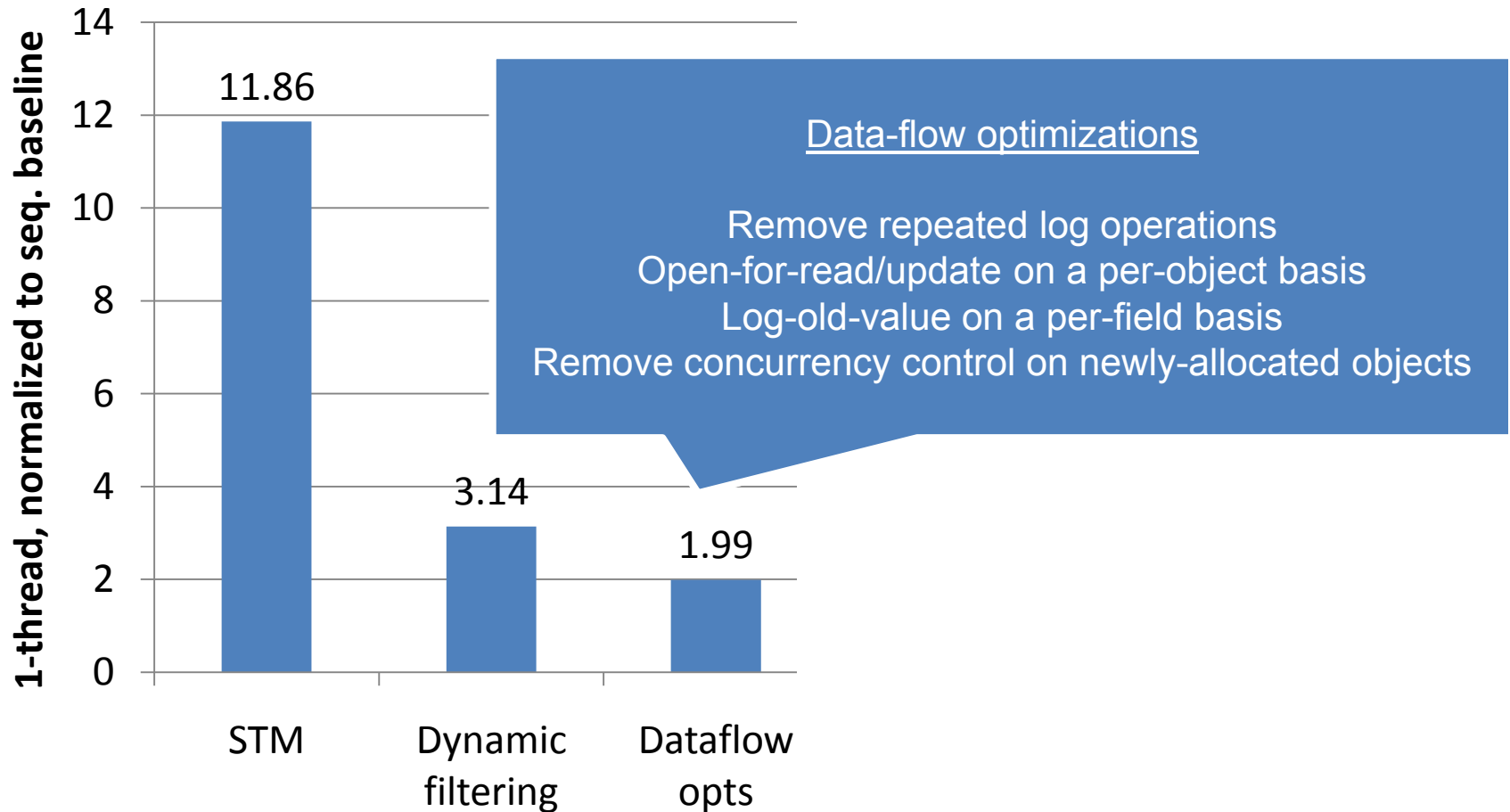
# Sequential overhead



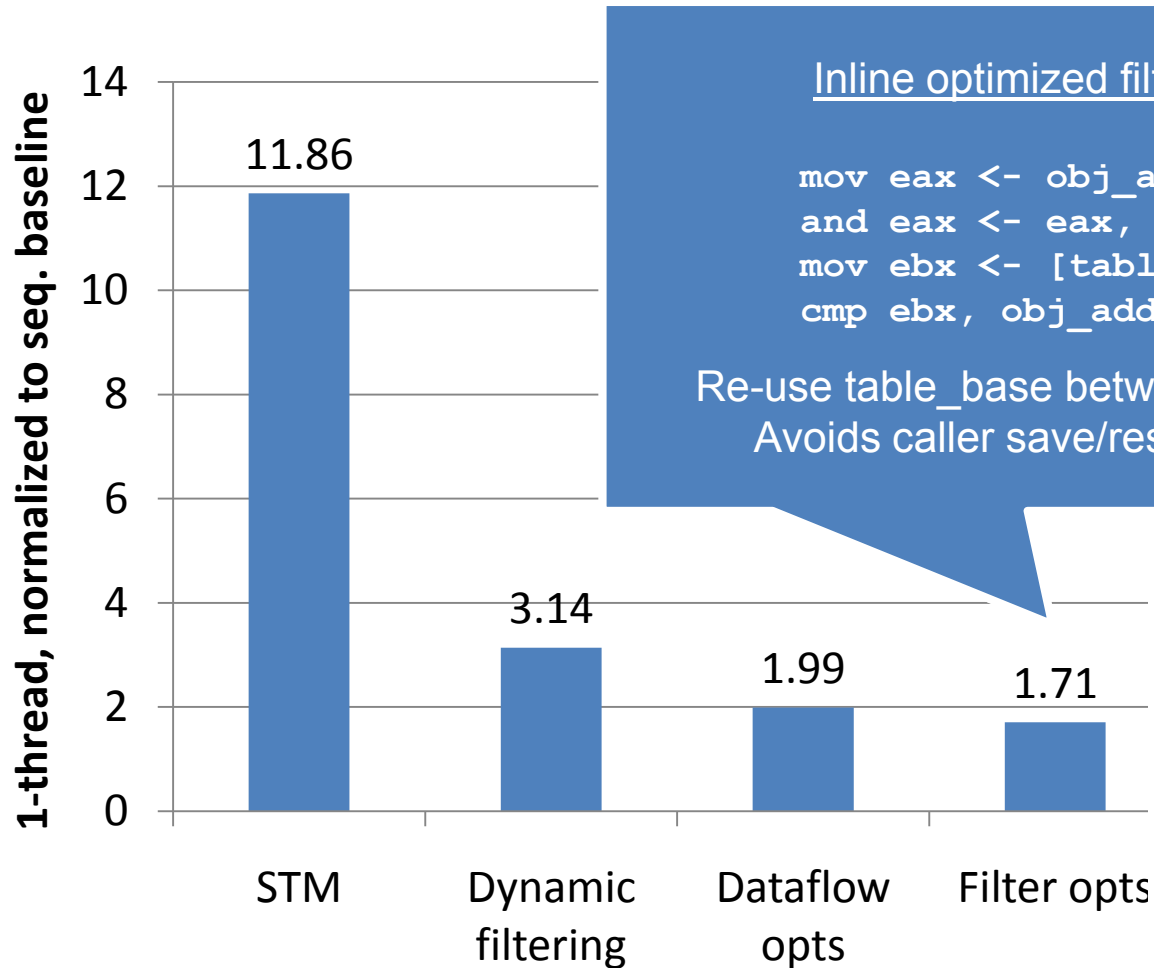
Dynamic filtering to remove redundant logging

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

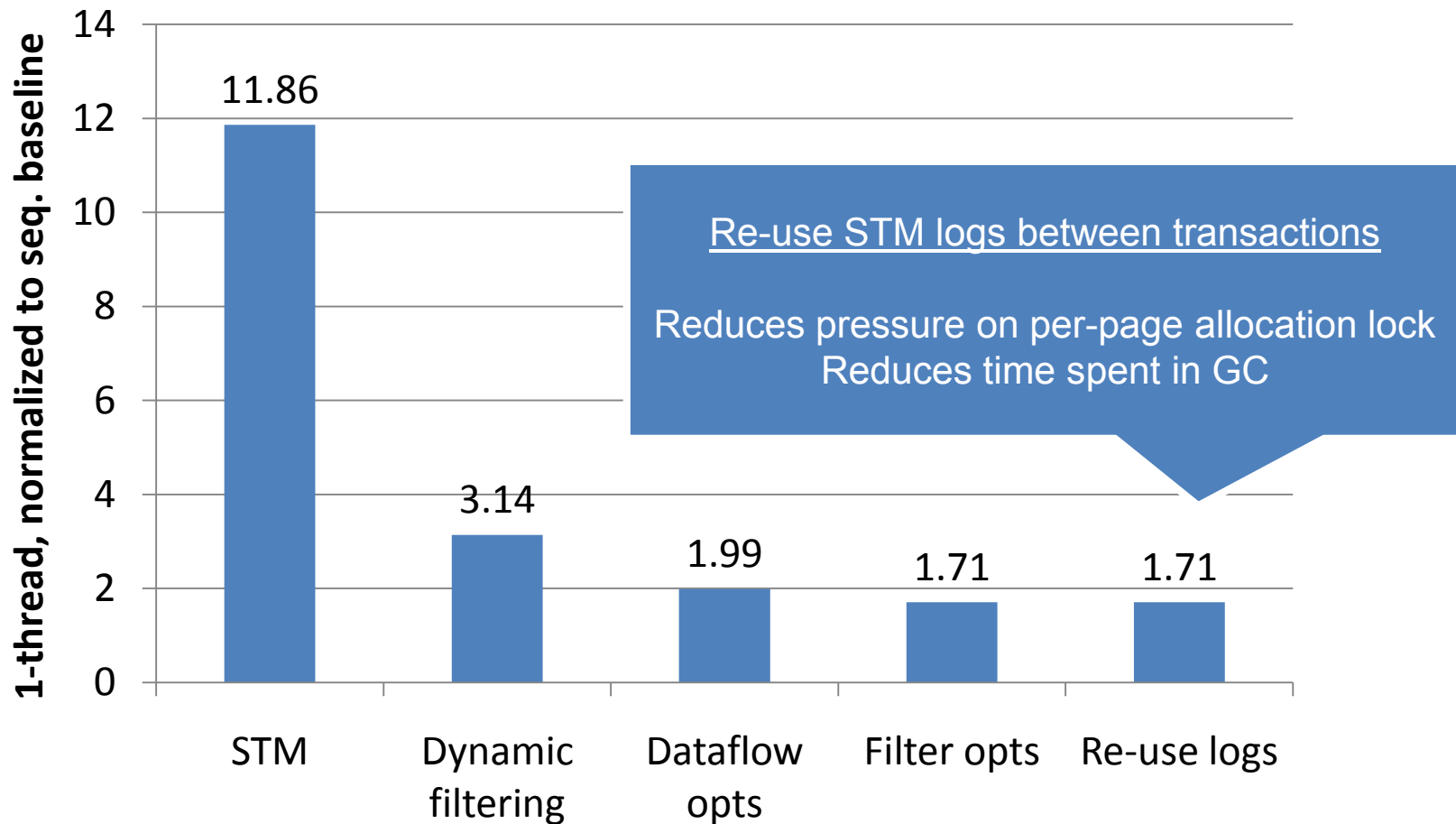
# Sequential overhead



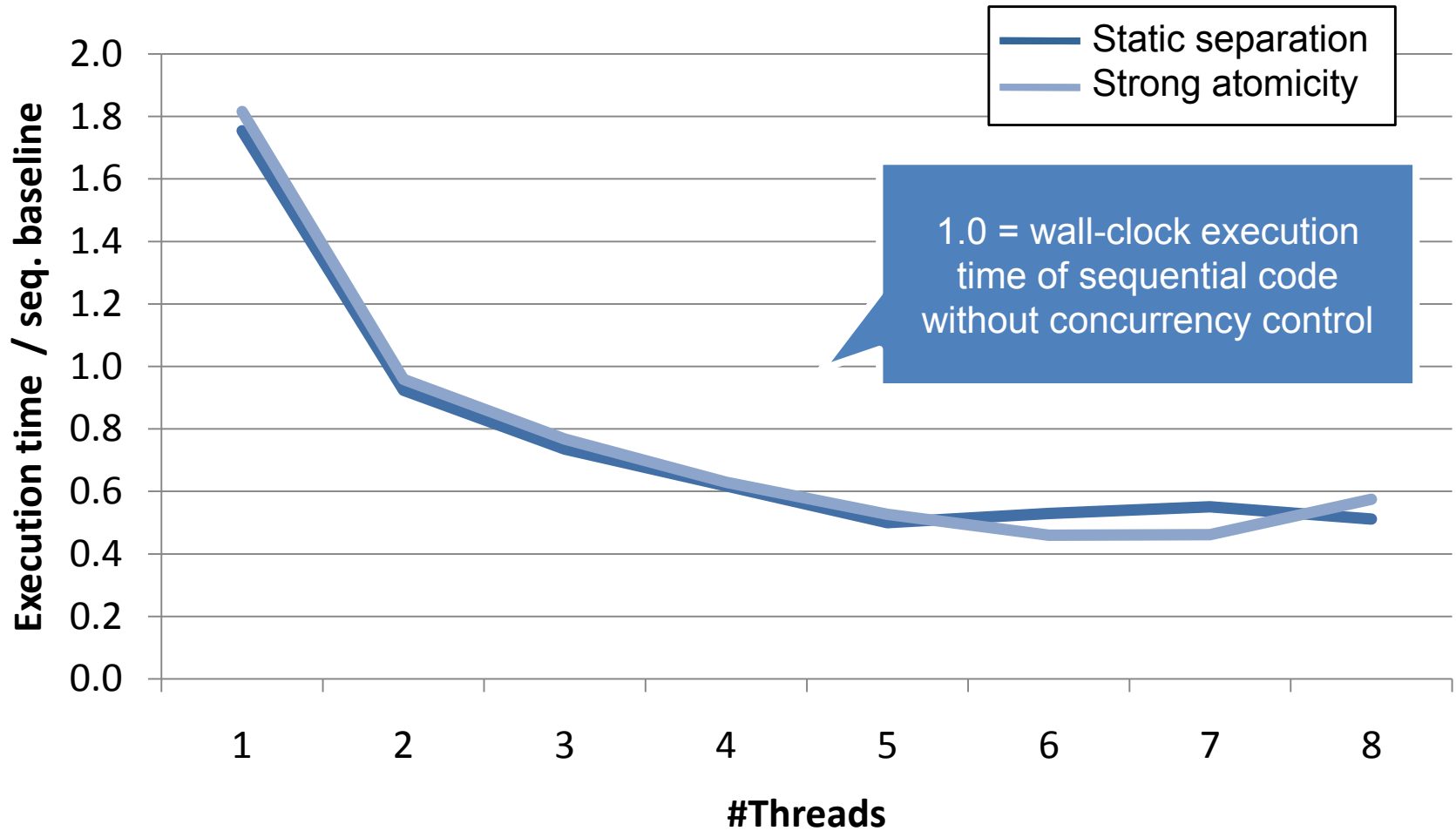
# Sequential overhead



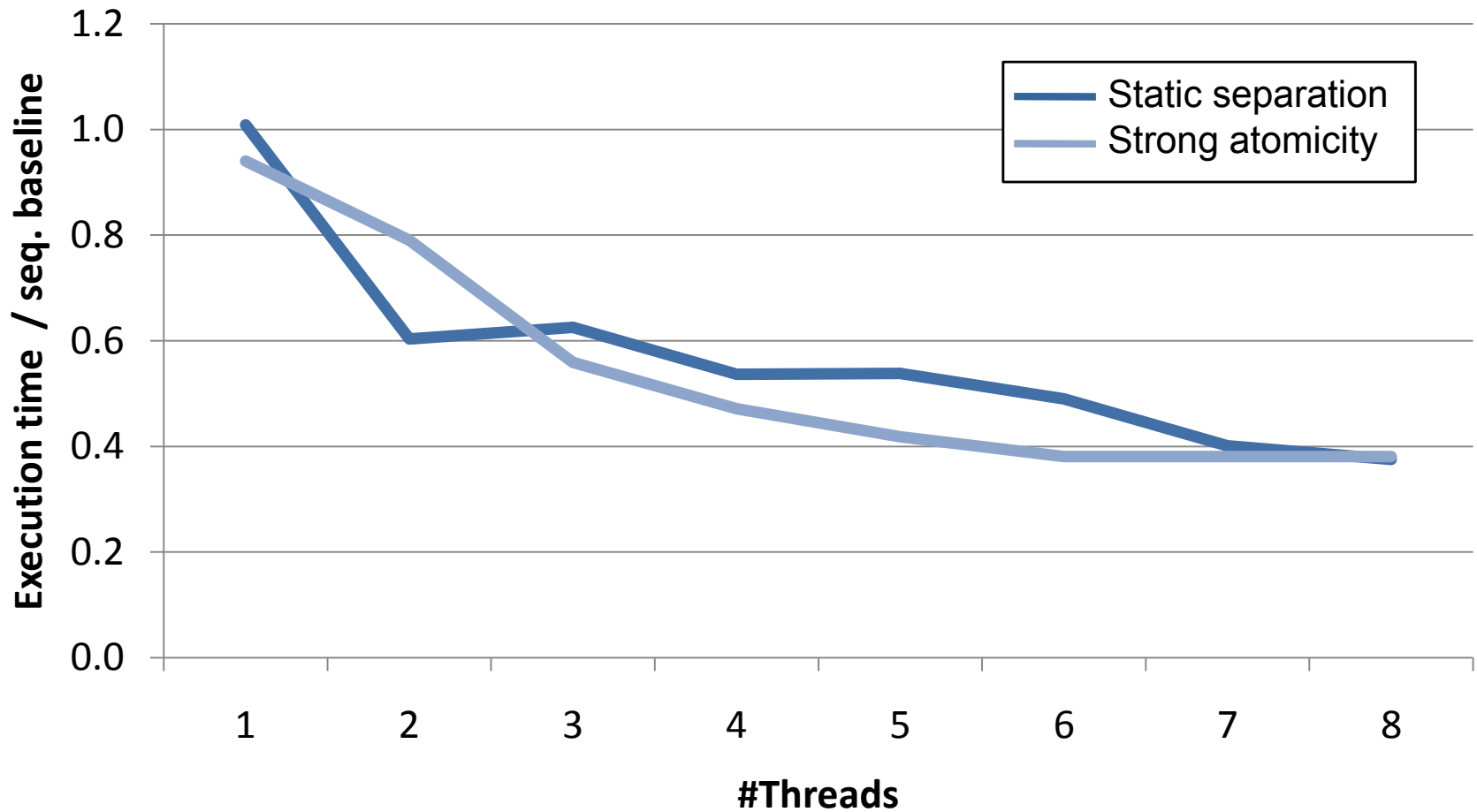
# Sequential overhead



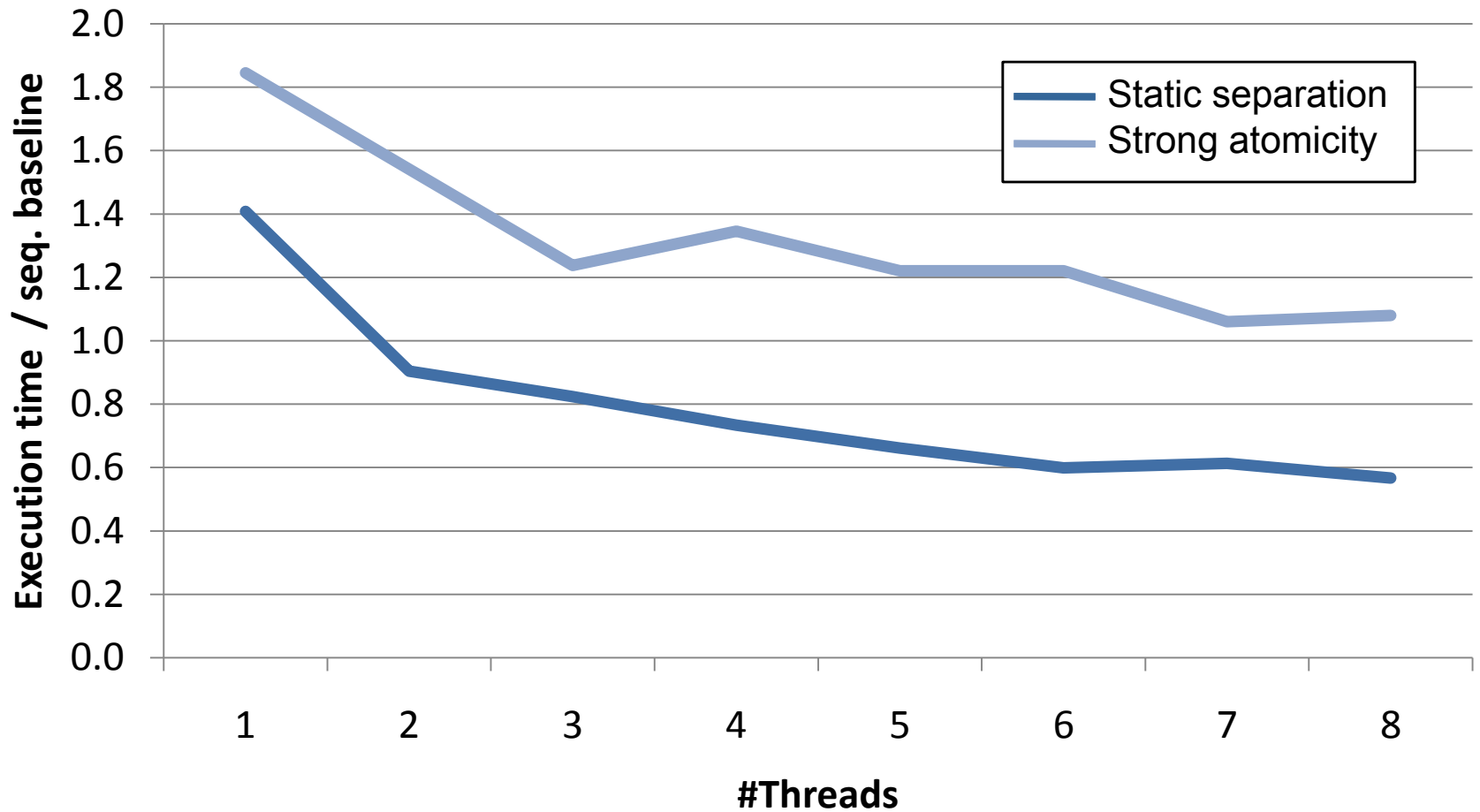
# Scaling – Labyrinth



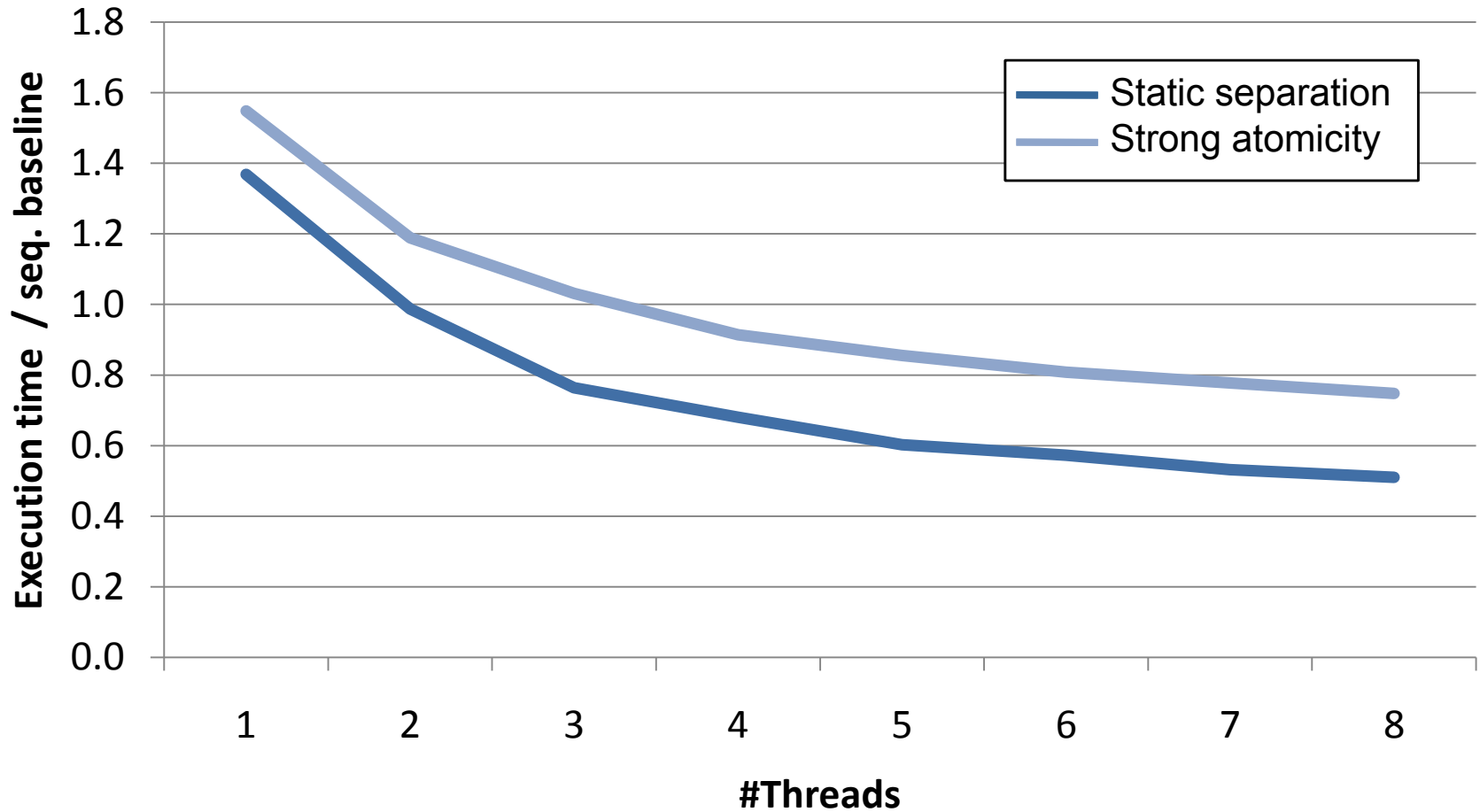
# Scaling – Delaunay



# Scaling – Genome



# Scaling – Vacation

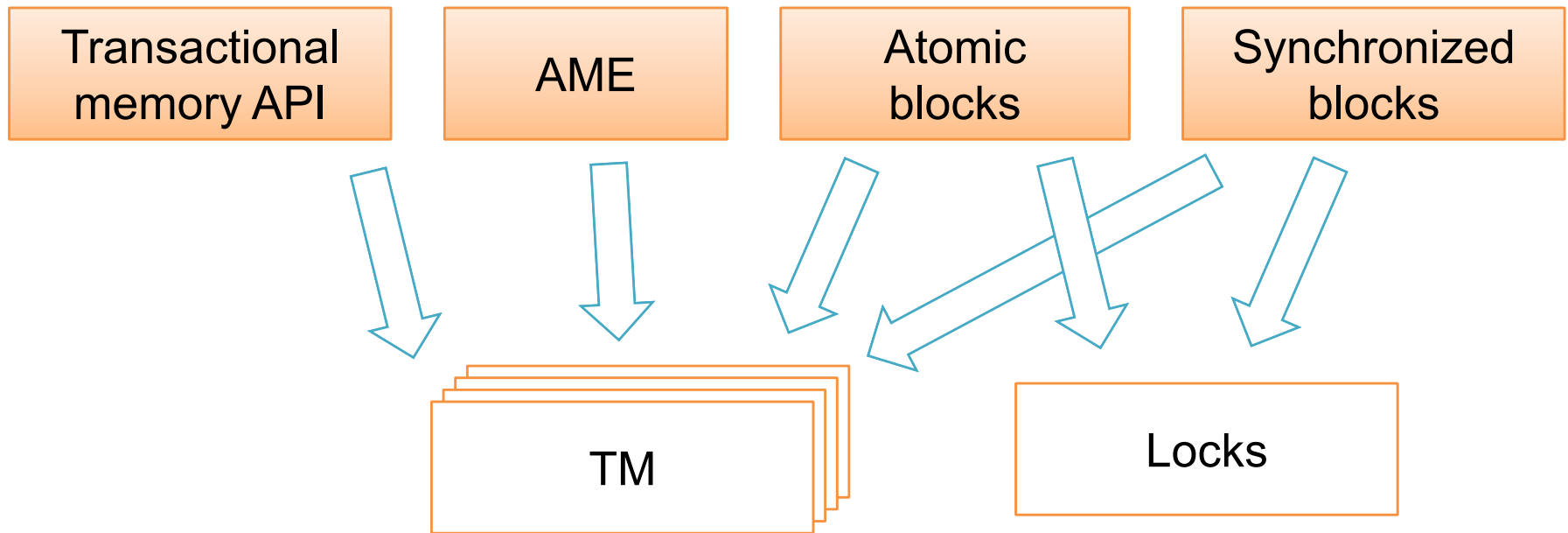


Untangling “atomic” from TM

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?

# Acknowledgements

- 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