

Language constructs for transactional memory

Tim Harris

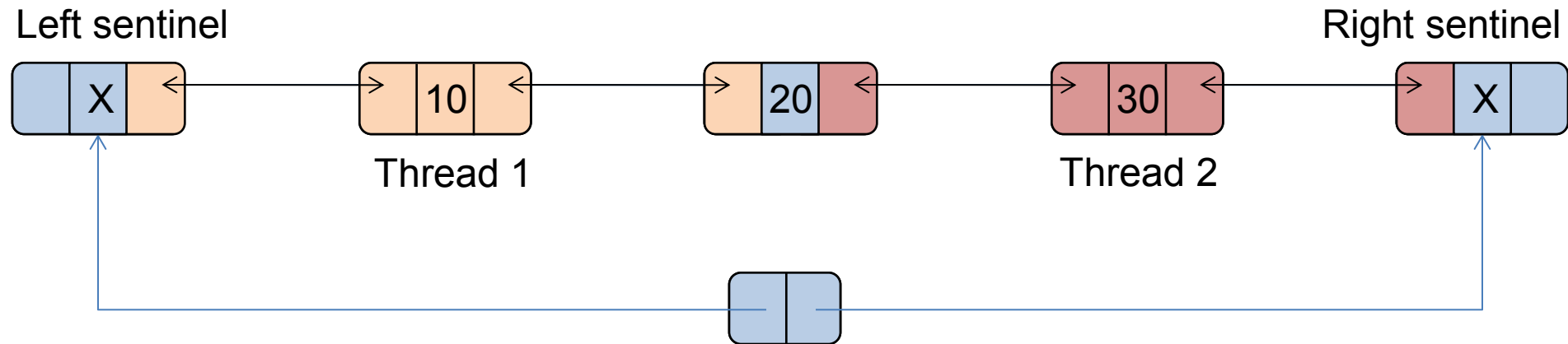
Disclaimer: these are my personal opinions

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());  
    }  
  
    ...  
}
```

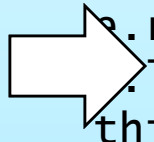
Implementing this: atomic blocks

```

Class Q {
    QElem l;
    QElem r;

    void pushLeft(int item) {
        atomic {
            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());
    }
    ...
}

```



```

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;
```

“What about I/O?”

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

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

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

“What happens if I single-step through this, and there’s a conflict with a concurrent transaction?”

“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?”

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

“Atomic blocks are transactions”

consider how the problems of external interaction through storage systems or databases, nor do we 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 *linearizability*, they must avoid introducing deadlocks, to ensure good *performance*, they must balance the granularity at 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 hash table with thread-safe insert and delete operations. Now suppose that we want to delete one item A from table v1, and insert it into table v2, but the intermediate state (in which neither table contains the item) must not be visible to other threads. Unless the implementor of the hash table anticipates this need, there is simply no way to satisfy this requirement. Even if she does, all she can do is expose methods such as `LockTable` and `UnlockTable` – but as well as breaking the hash-table abstraction, they invite lock-induced deadlock, depending on the order in which the client takes the locks, or race conditions if the client forgets. Yet more complexity: suppose if the client wants to avoid the presence of A in v1, but its blocking behaviour must not lock the table (else A cannot be inserted). In short, operations that are individually correct (insert, delete) cannot be composed into larger correct operations.

The same phenomenon shows up trying to compose alternative blocking operations. Suppose a procedure p1 waits for one of two input pipes to have data, using an internal call to the Unix `select` procedure, and suppose another procedure p2 does the same thing, on two different pipes. In Unix there is no way to perform a `select` between p1 and p2, a fundamental loss of compositionality. Instead, Unix programmers learn awkward programming techniques to gather up all the file descriptors that must be waited for, perform a single top-level `select`, and then dispatch back to the correct handler. Again, two individually-correct abstractions, p1 and p2, cannot be composed into a larger one, instead, they must be ripped apart and awkwardly merged, in direct conflict with the goals of abstraction.

Rather than fixing locks, a more promising and radical alternative is to base 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 hash table thus:

```
atomic { v.delete(v1.A); insert(v2,A,v) }
```

and to wait for either p1 or p2 we can say

```
atomic { p1 'waitfor' p2 }
```

These simple constructions require no knowledge of the implementation of `insert`, `delete`, `p1`, or `p2`, and they continue to work correctly if those 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 memory [11, 16, 9, 6, 31].

The key idea is that a block of code, including nested calls, can be enclosed 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 synchronization. Instead of taking locks, an *atomic block runs without locking*, accumulating a thread-local transaction log that records every memory read and write it makes. When the block completes, it first validates

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

Transactional memory eliminates, by construction, many of the low-level difficulties that plague lock-based programming [6]. 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

It might also launch

was de-scheduled after

thread modified both

begs for a guarantee

perform memory op

modifications to re-

irrevocable operation.

Secondly, blocks

support synchronisation

and those that do not

on the atomic block

from a buffer might

from `get()` <

`atomic { n, t :=`

`}`

The thread waits until

creating the block. In

We expect call `get()`

form an intervening

in a nested atomic b

from the outer block i

is a disaster for altho

the two threads has to

modification. If several

even worse.

Thirdly, no person

compiled by the ad

tion 7.2 on Consume

by presenting future

the language C++

2.2 Concurrent li

Concurrent block-l

lary, functional lang

abstractions for con

volves side effects w

necessary to be able

breakthrough came f

Here is the key id

when performed, m

a. For example, the

`putChar :: Char`

`getChar :: IO Char`

That is, `putChar` takes a `Char` and delivers an `IO action` that,

when performed, prints the character on the standard output, while

`getChar` 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

“Atomic blocks are locks”

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

Single Global Lock Semantics in a Weakly Atomic STM

Vijay Menon¹ Steven Bakensieffer² Tatiana Shpeisman¹ Ali-Reza Adl-Tabatabai¹
Richard L. Hudson¹ Brahm Saha¹ Adam Welc²

¹Intel Labs
Santa Clara, CA 95054
{vijay.s.menon,tatiana.shpeisman,ali-aza.a.adl-tabatabai,rich.hudson,brahm.saha,adam.welc}@intel.com
{shloika}@cs.washington.edu

²Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195
{shloika}@cs.washington.edu

Abstract

As memory transactions have been proposed as a language-level replacement for locks, there is growing need for well-defined semantics. In contrast to database transactions, transaction memory (TM) semantics are complicated by the fact that programs may access the same memory locations both inside and outside transactions. Simply atomic semantics, where non-transactional accesses are treated as implicit single-operation transactions, remain difficult to provide without specialized hardware support and/or significant performance overhead. As an alternative, many in the community have informally proposed that a *single global lock semantics*, where transaction semantics are mapped to those of a program executed by a single global lock, provide an intuitive and efficiently implementable model for programmers.

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

```
Initially data = 42, ready = false, val = 0
Thread 1      Thread 2
1:             atomic {
2:             tap = data;
3:             data = 1;
4:             atomic {
5:             ready = true;
6:             }
7:             }
8:             if (ready)
9:             val = tap;
10:            }
11:            Cnt.val += 42?
```

Figure 1. Publication example with a seemingly benign race

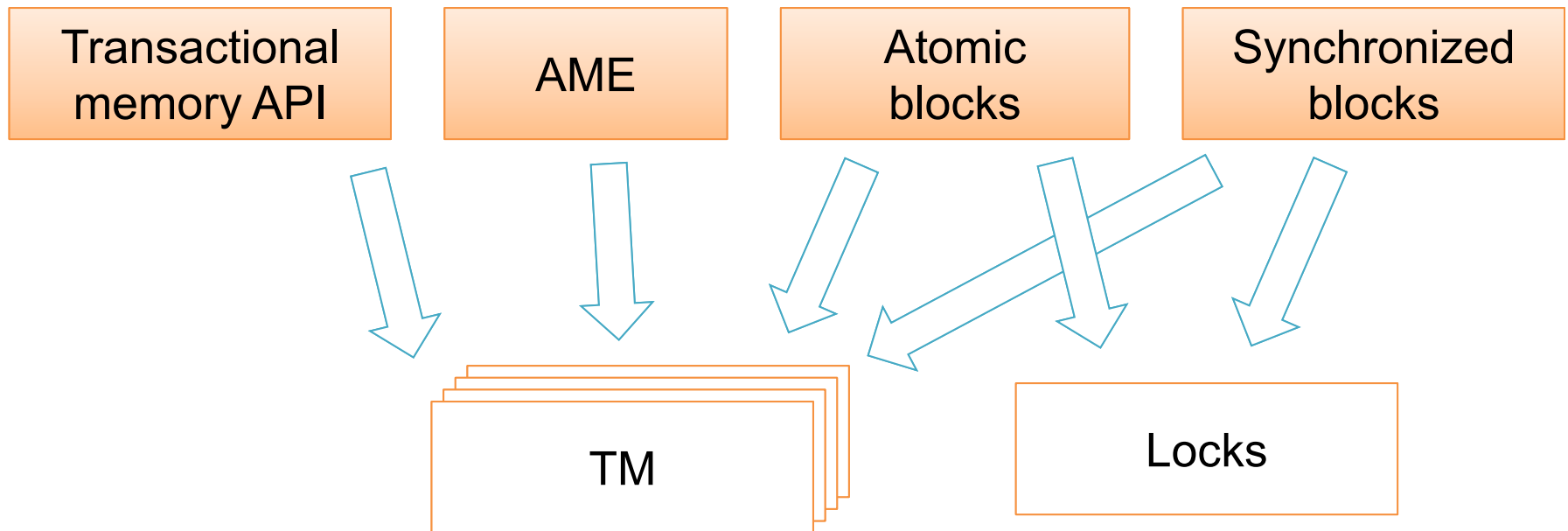
situations when the same data is accessed both transactionally and non-transactionally. Because of this, nearly all published STM systems provide weaker guarantees for transactions than locks. Consider the publication example in Figure 1. Thread 1 reads data early into the private location *tap*. However, that value is only used if *ready* is set. Otherwise, *tap* is dead and never scanned again. If the transactions are replaced with locks, this program will run correctly: *val* would either be 0 or 1 depending upon which critical section executed first. Nevertheless, most STMs would produce the value 42 in Thread 2 given the interleaving in Figure 1. As Thread 1 writes data outside a transaction, a weakly atomic STM would not detect any conflict and, thus, would not invalidate Thread 2's read operation.

It should be noted that, even under locks, this program has a data race: Thread 1 and Thread 2 may access data simultaneously. However, from the programmer's perspective, the value is never used in this case, and the race should be benign. Java's memory model [17] (assuming lock semantics) specifically disallows any execution that produces 42 in Thread 2. In fact, this example also has implications for correctly synchronized programs. Standard compiler ordering can inadvertently introduce a data race. Consider a correctly synchronized variant of this program in Figure 2 where Thread 2 only accesses data inside the conditional. Compiler optimizations such as speculative redundancy elimination or instruction scheduling may still introduce a data race by hoisting the access (e.g., to line 2 in Figure 1) if profitable. Such optimizations assume that introduced races are benign.

An appealing solution is to provide strong atomicity [5, 22]¹, where non-transactional memory accesses are analogous to single instruction transactions and prevented from violating the isolation of transactions. In this model, transactions are strictly more restrictive

¹ The term *strong isolation* is also used in the literature.

Abstractions vs implementations

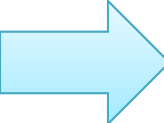


Defining “atomic” without saying “TM”

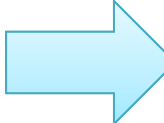
- “Strong semantics”
 - Simple interleaved execution of threads
 - If a thread starts an atomic block then only it can take steps
 - Blocking operations (e.g. “retry”, “orElse”, “blockUntil”) can be incorporated – see refs below
- This means:
 - Atomic blocks are atomic wrt normal memory accesses
 -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



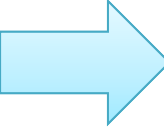
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



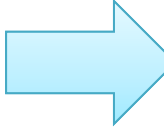
```
atomic {  
    x_shared = false;  
}  
x++;
```

Example: a privatization idiom

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



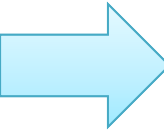
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



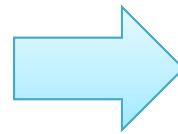
```
atomic {  
    x_shared = false;  
}  
x++;
```

Example: a privatization idiom

`x_shared = true;    x = 100;`



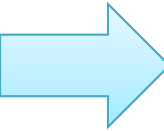
```
atomic {  
  if (x_shared) {  
    x = 100;  
  }  
}
```



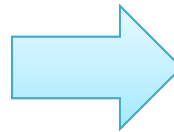
```
atomic {  
  x_shared = false;  
}  
x++;
```

Example: a privatization idiom

`x_shared = false;    x = 100;`



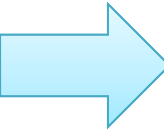
```
atomic {  
  if (x_shared) {  
    x = 100;  
  }  
}
```



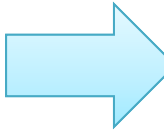
```
atomic {  
  x_shared = false;  
}  
x++;
```

Example: a privatization idiom

`x_shared = false;    x = 101;`



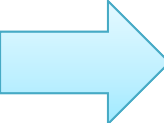
```
atomic {  
  if (x_shared) {  
    x = 100;  
  }  
}
```



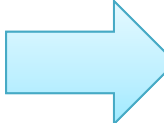
```
atomic {  
  x_shared = false;  
}  
x++;
```

Example: a privatization idiom

`x_shared = true;    x = 0;`



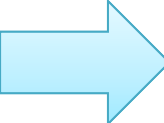
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



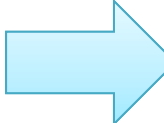
```
atomic {  
    x_shared = false;  
}  
x++;
```

Example: a privatization idiom

`x_shared = true;    x = 0;`



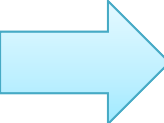
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



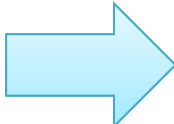
```
atomic {  
    x_shared = false;  
}  
x++;
```

Example: a privatization idiom

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



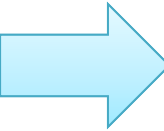
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



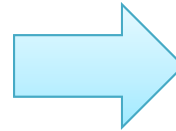
```
atomic {  
    x_shared = false;  
}  
x++;
```

Example: a privatization idiom

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



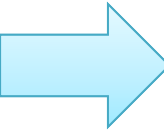
```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



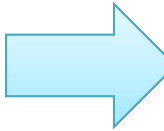
```
atomic {  
    x_shared = false;  
}  
x++;
```

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?

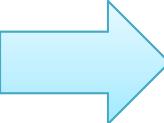
Untangling “atomic” from TM

Hiding TM from programmers

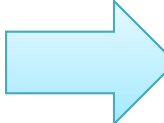
Current performance

Example: a privatization idiom

`x_shared = true;    x = 0;`



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

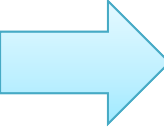


```
atomic {  
    x_shared = false;  
}  
x++;
```

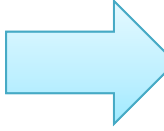
Not valid
execution under
strong semantics

Example: a privatization idiom

`x_shared = true;    x = 0;`



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

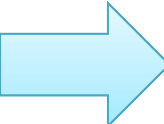


```
atomic {  
    x_shared = false;  
}  
x++;
```

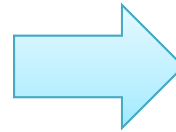
Not valid
execution under
strong semantics

Example: a privatization idiom

`x_shared = true;    x = 100;`



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

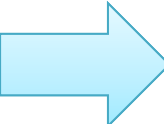


```
atomic {  
    x_shared = false;  
}  
x++;
```

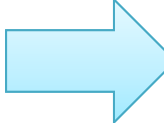
Not valid
execution under
strong semantics

Example: a privatization idiom

`x_shared = false;    x = 100;`



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

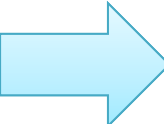


```
atomic {  
    x_shared = false;  
}  
x++;
```

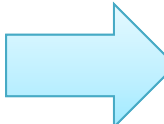
Not valid
execution under
strong semantics

Example: a privatization idiom

`x_shared = false;    x = 101;`



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

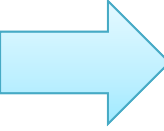


```
atomic {  
    x_shared = false;  
}  
x++;
```

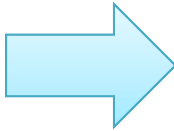
Not valid
execution under
strong semantics

Example: a privatization idiom

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



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

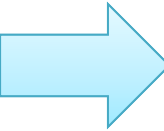


```
atomic {  
    x_shared = false;  
}  
x++;
```

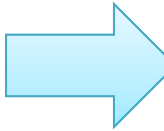
Not valid
execution under
strong semantics

Example: a privatization idiom

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

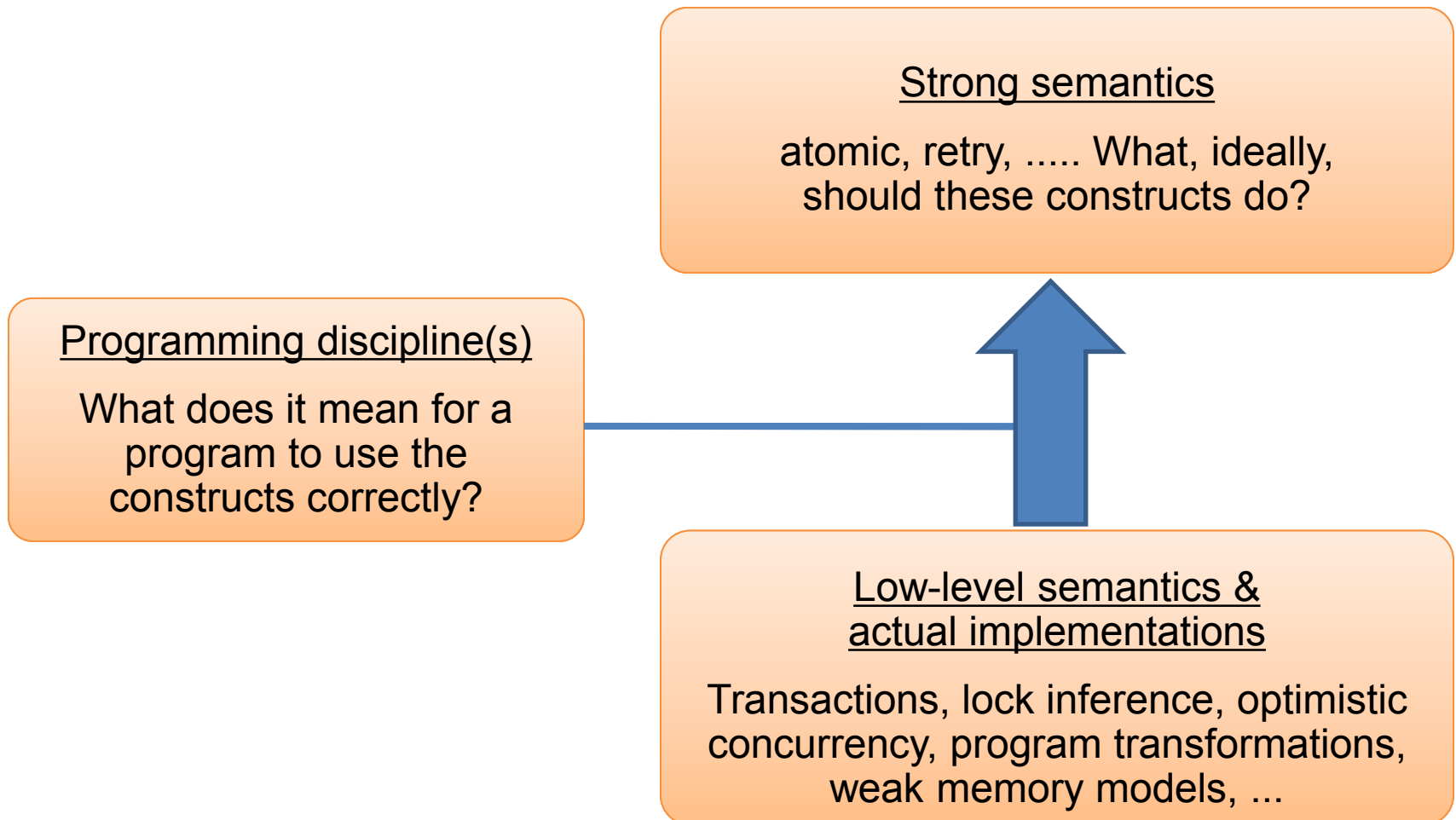


```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```



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

Hiding TM from programmers



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

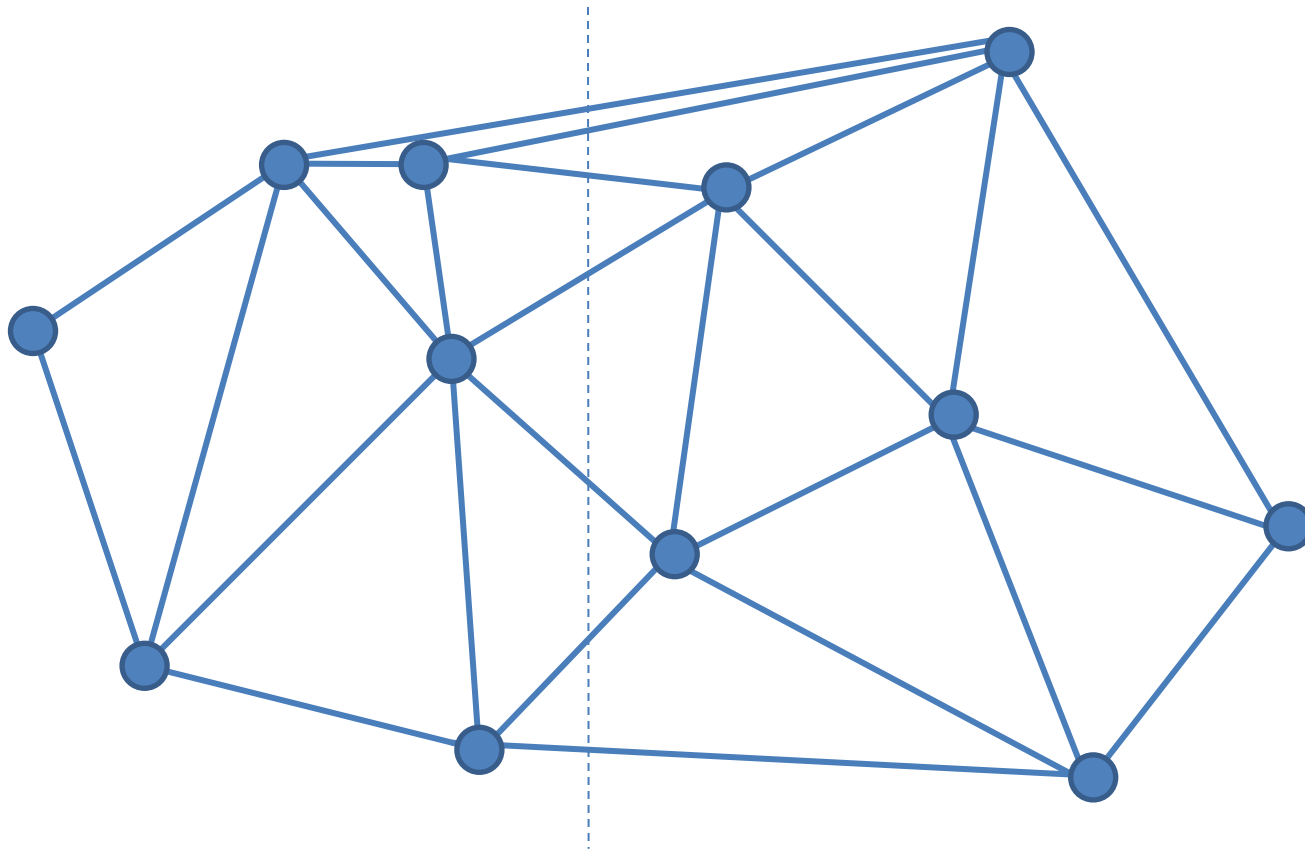
Fewer programs
satisfy the discipline

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;  
        }  
    }  
}
```

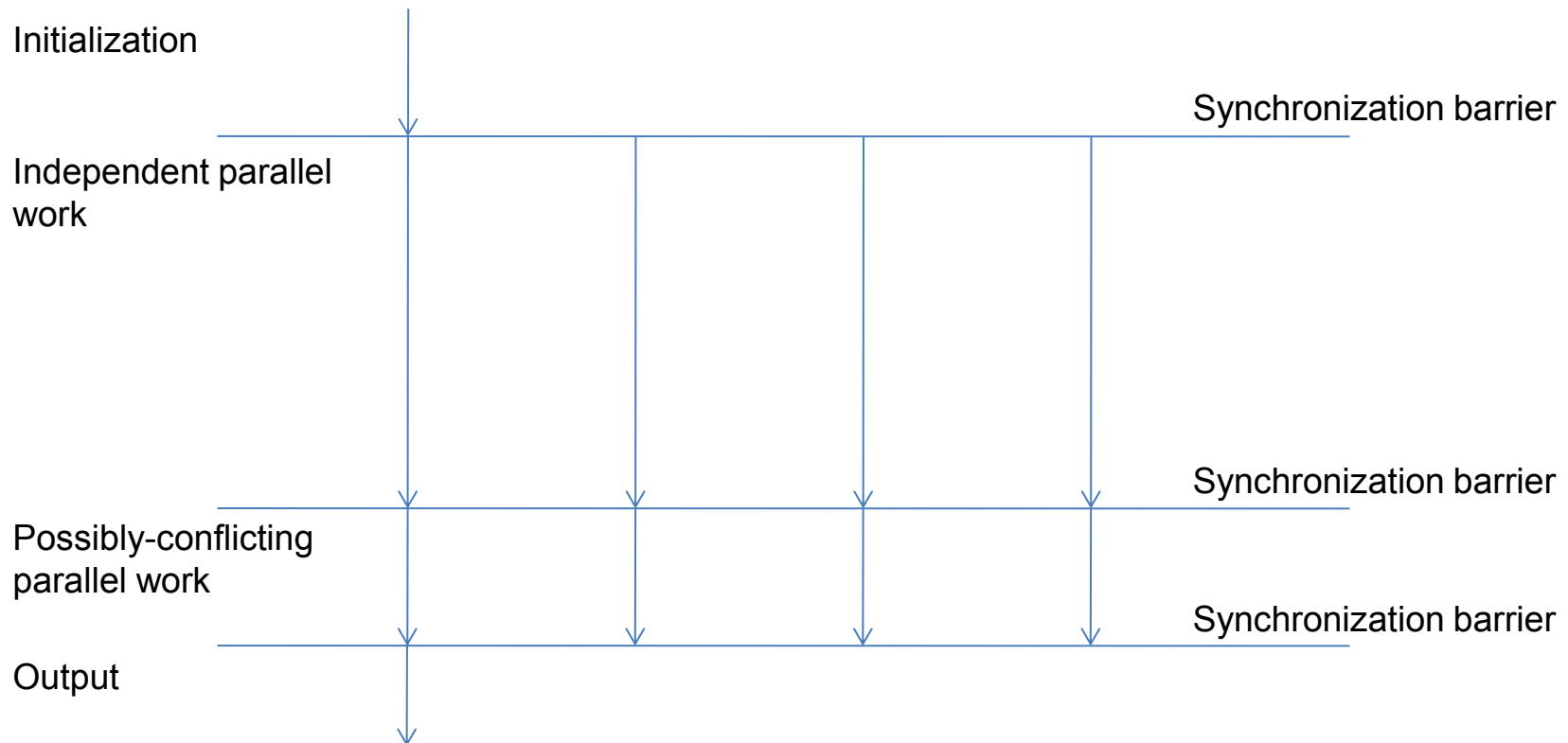
Delaunay triangulation



“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

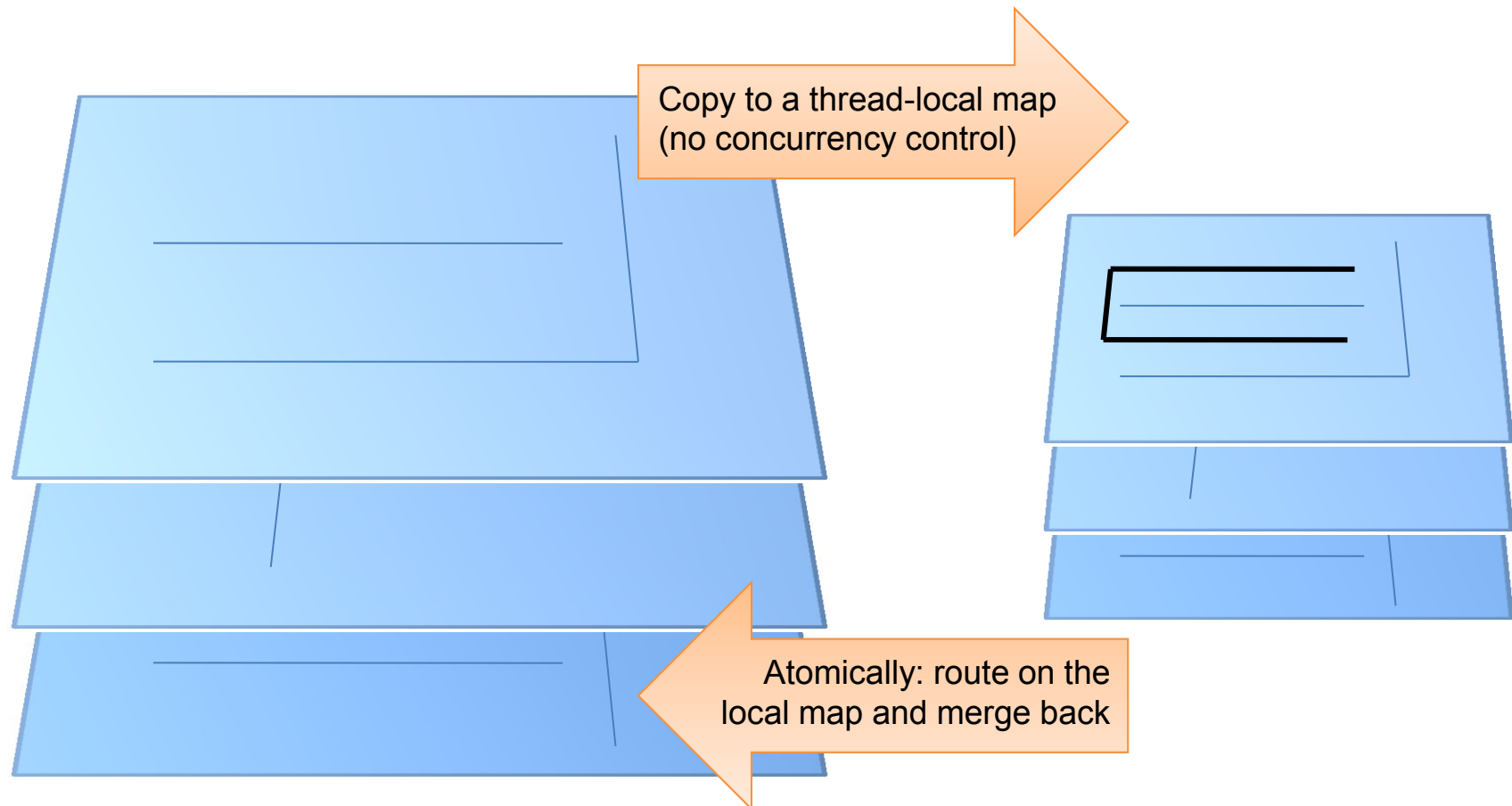
Example: a privatization idiom

```
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 insert(QElem item) {
```

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

“Again, ...

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

“If it’s a conflicting access, then the program is not correctly synchronized

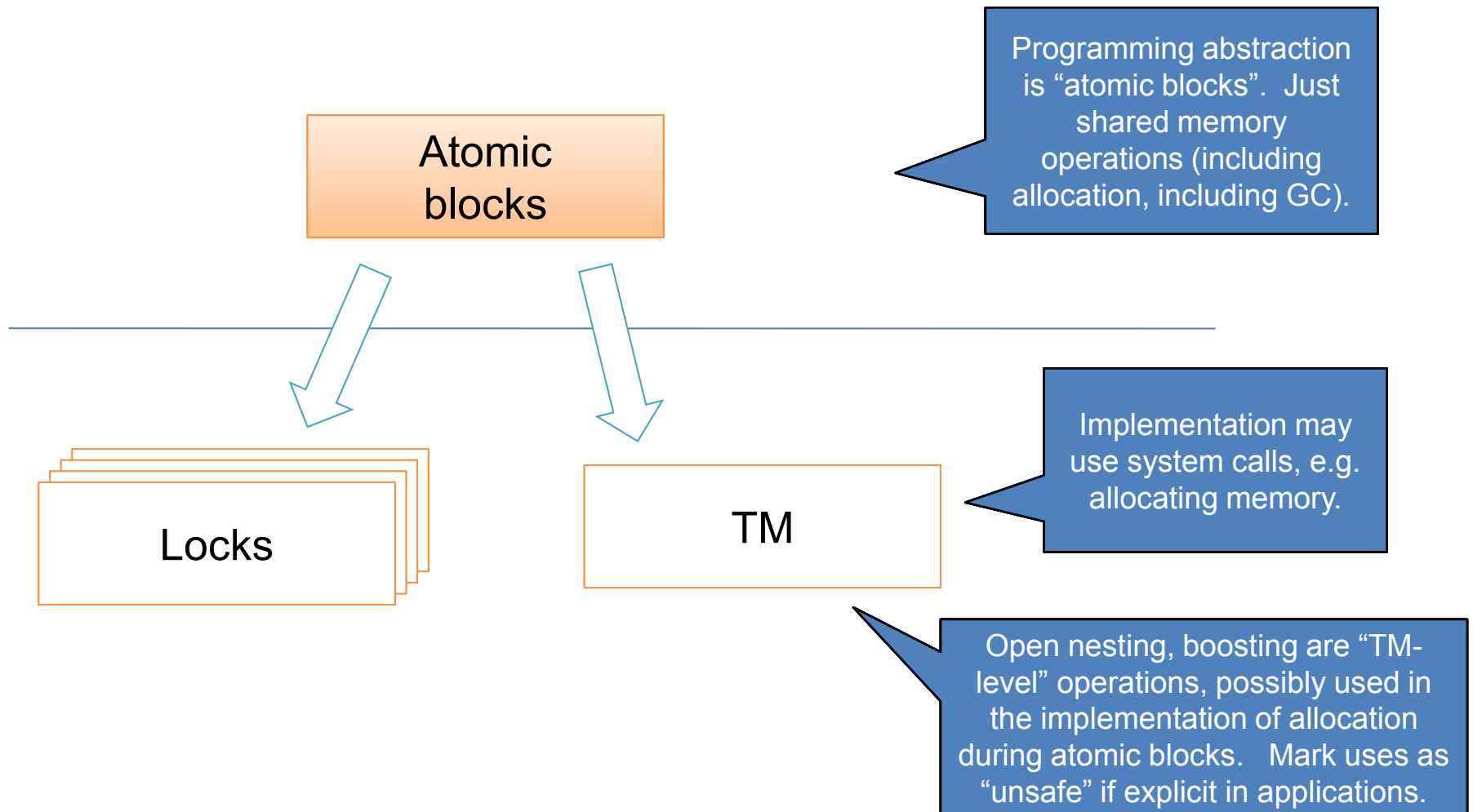
“Again, in correctly synchronized programs, speculation won’t be revealed by the implementation

“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)

Open nesting, boosting, system calls



Untangling “atomic” from TM

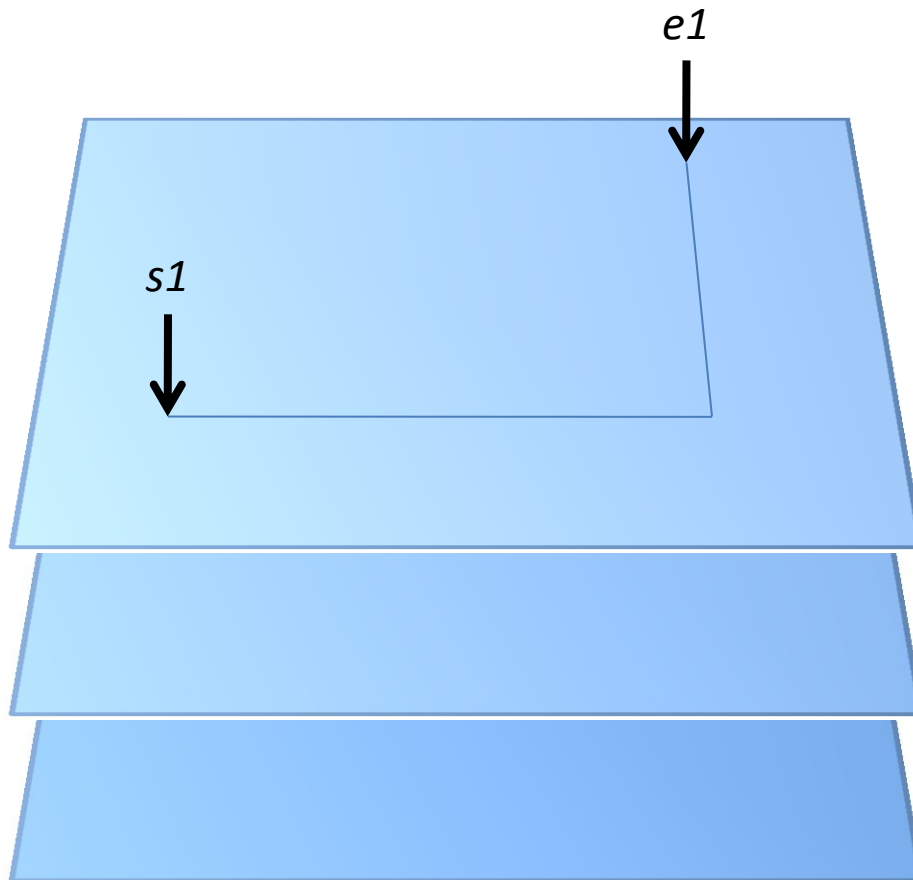
Hiding TM from programmers

Current performance

Perf. figures depend on...

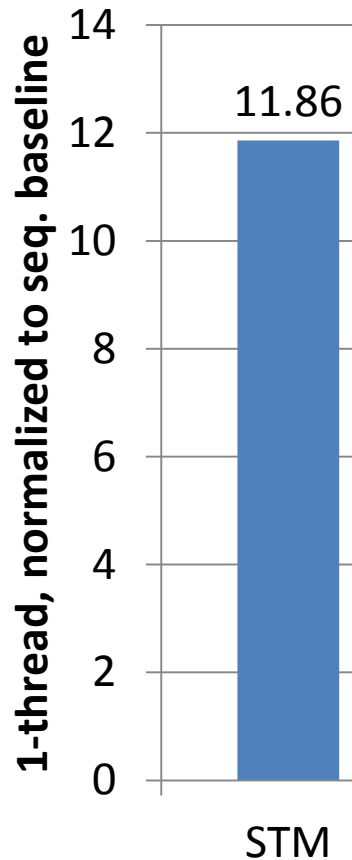
- **Workload** : What do the atomic blocks do? How long is spent inside them?
- **Baseline implementation**: Mature existing compiler, or prototype?
- **Intended semantics**: Support static separation? Violation freedom? Strong atomicity?
- **STM implementation**: In-place updates, deferred updates,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

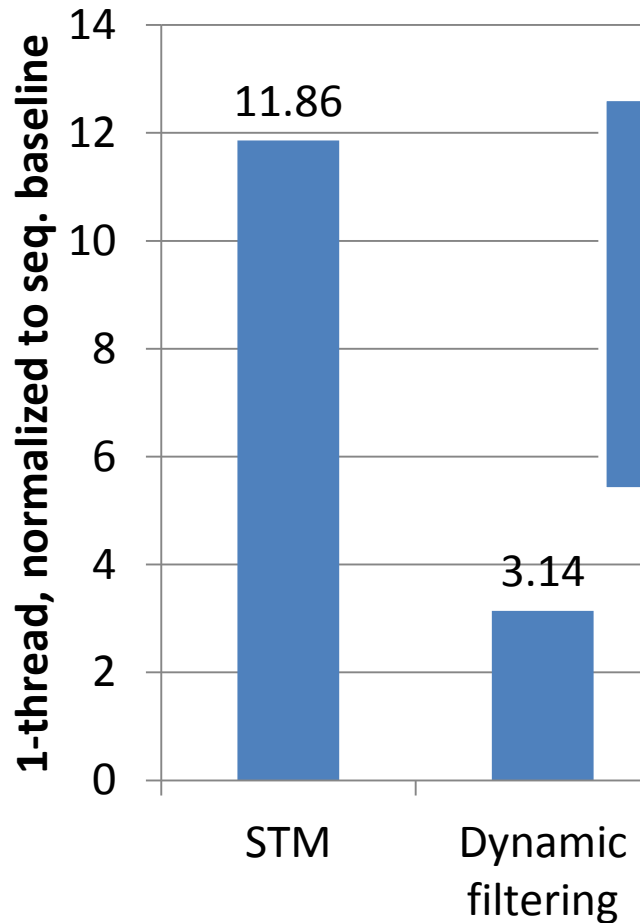
Sequential overhead



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

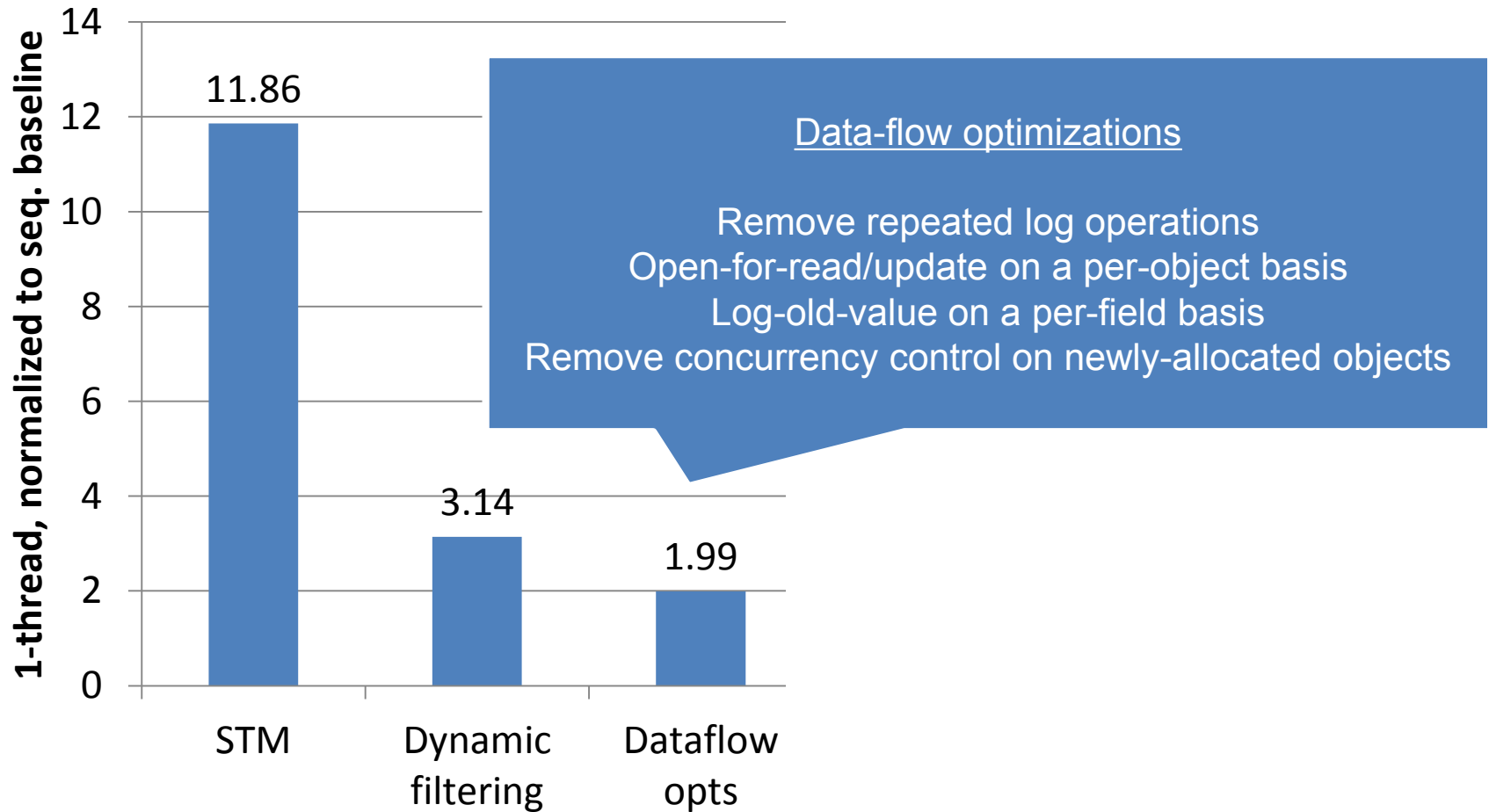
Sequential overhead



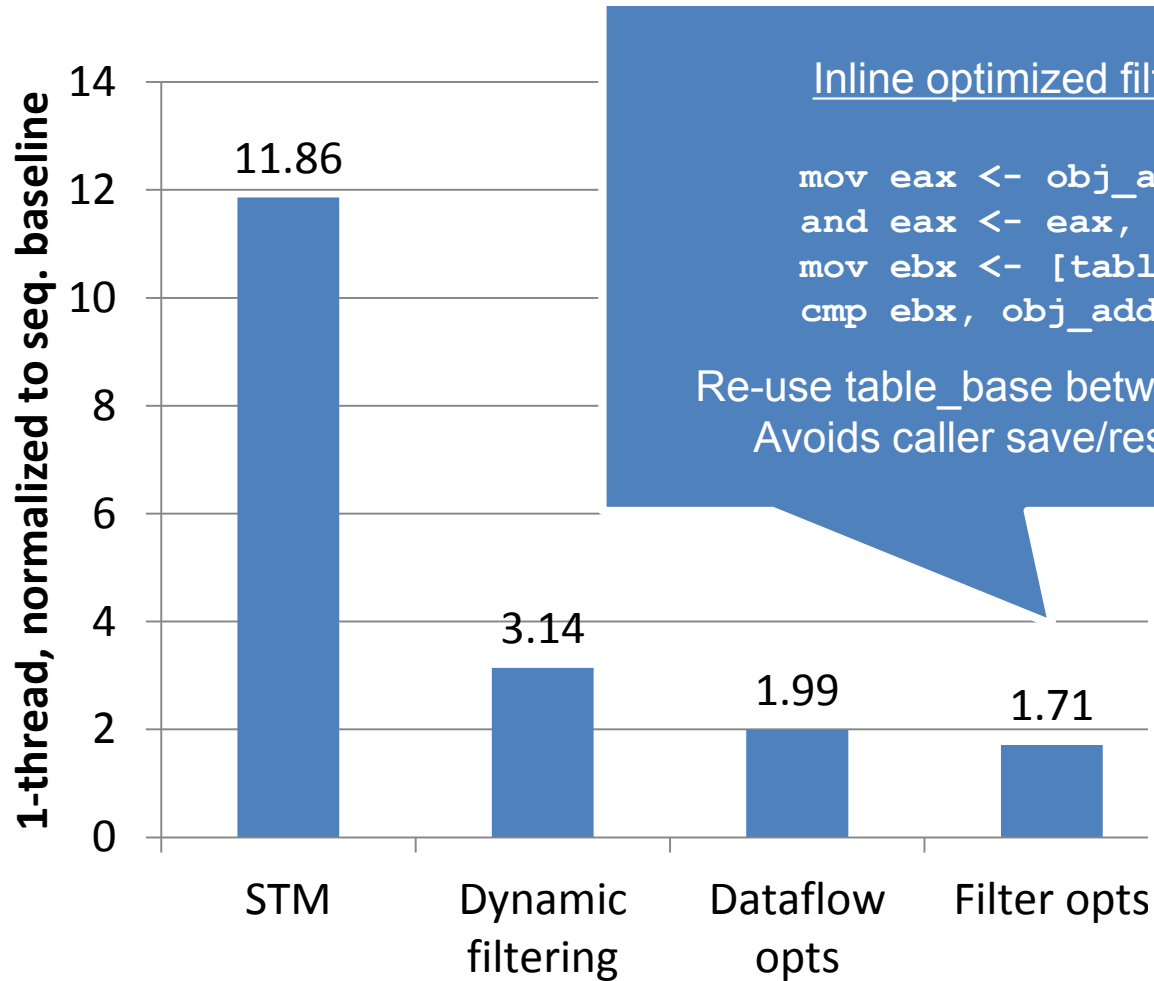
Dynamic filtering to remove redundant logging

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

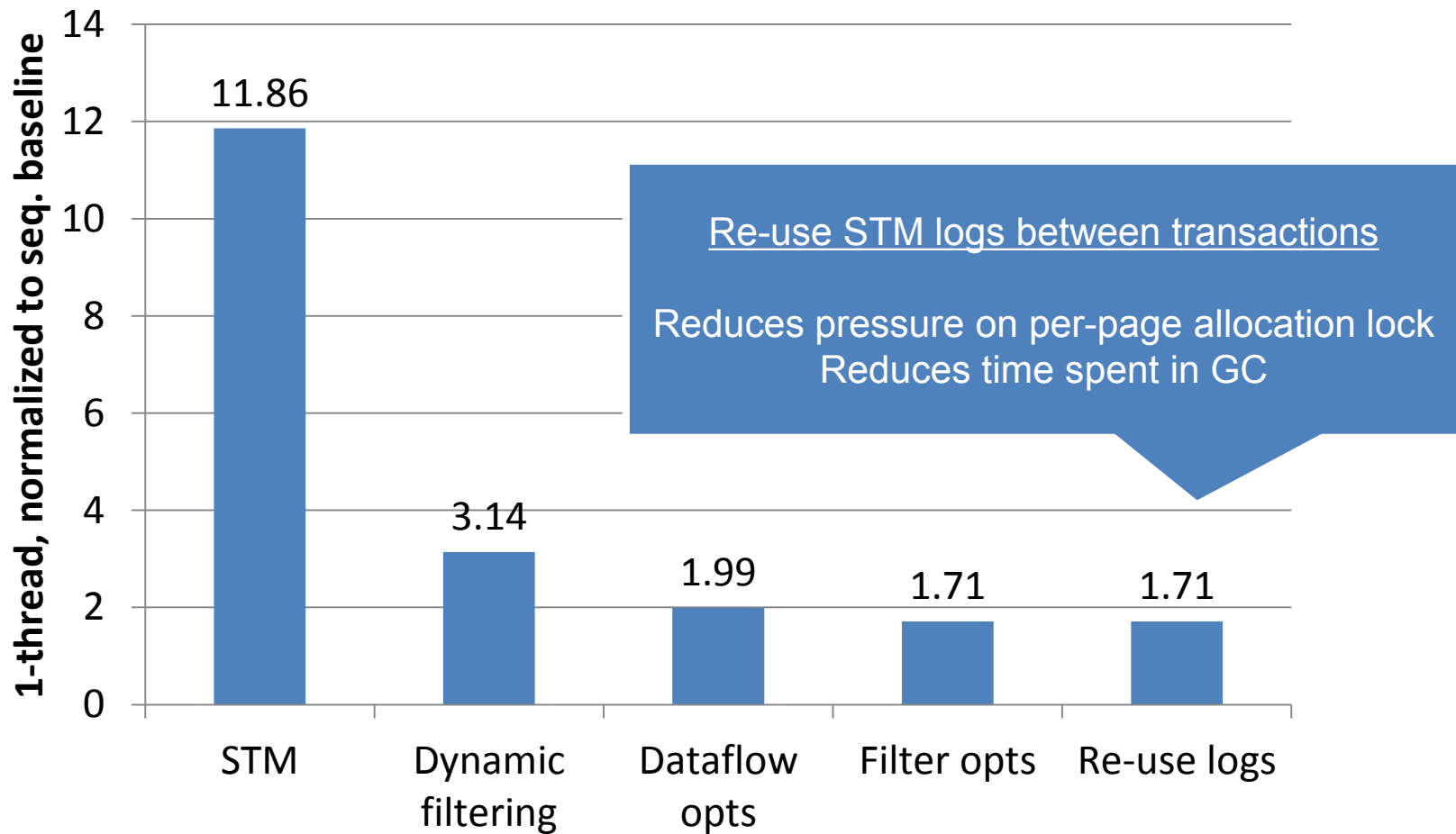
Sequential overhead



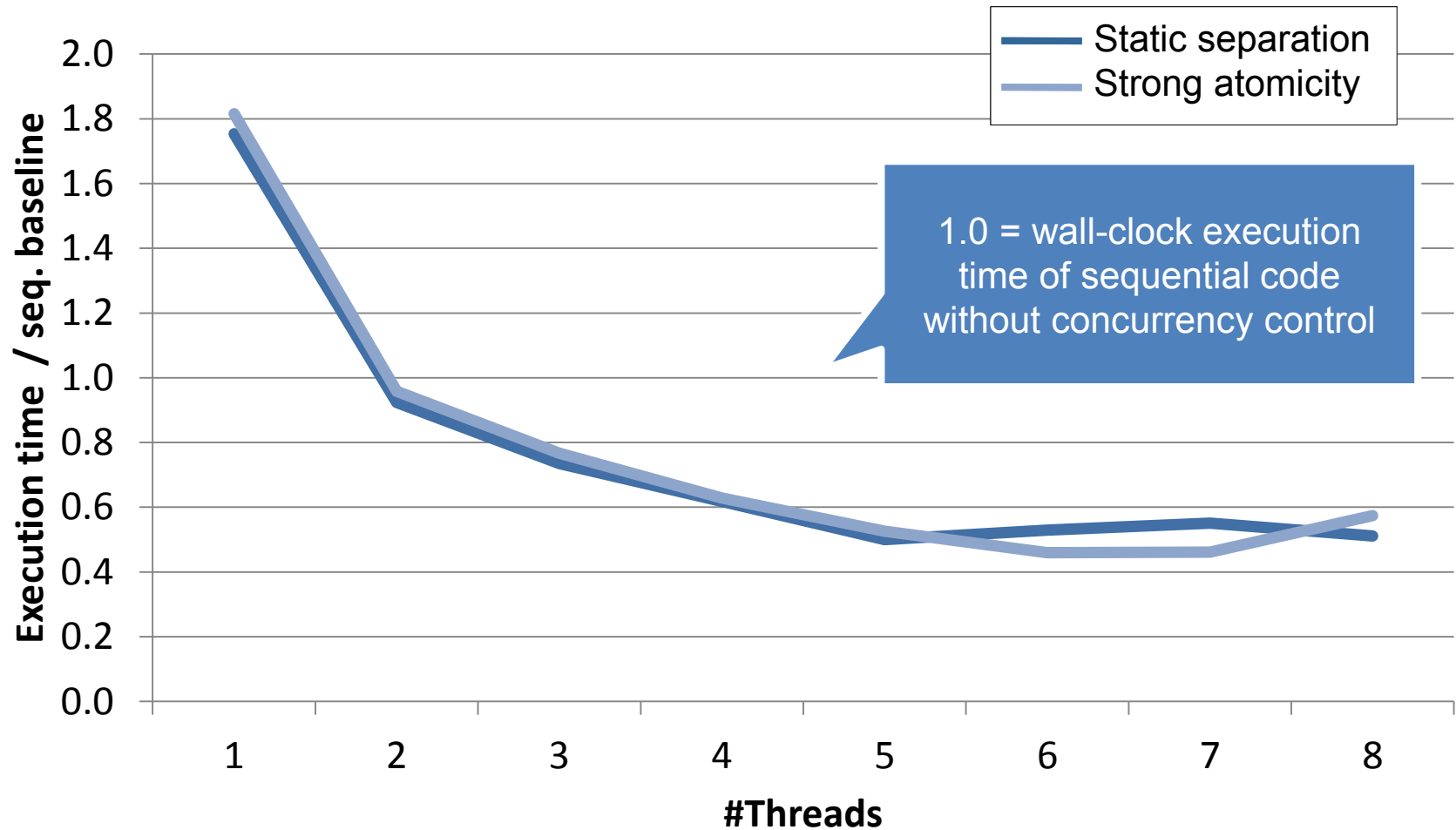
Sequential overhead



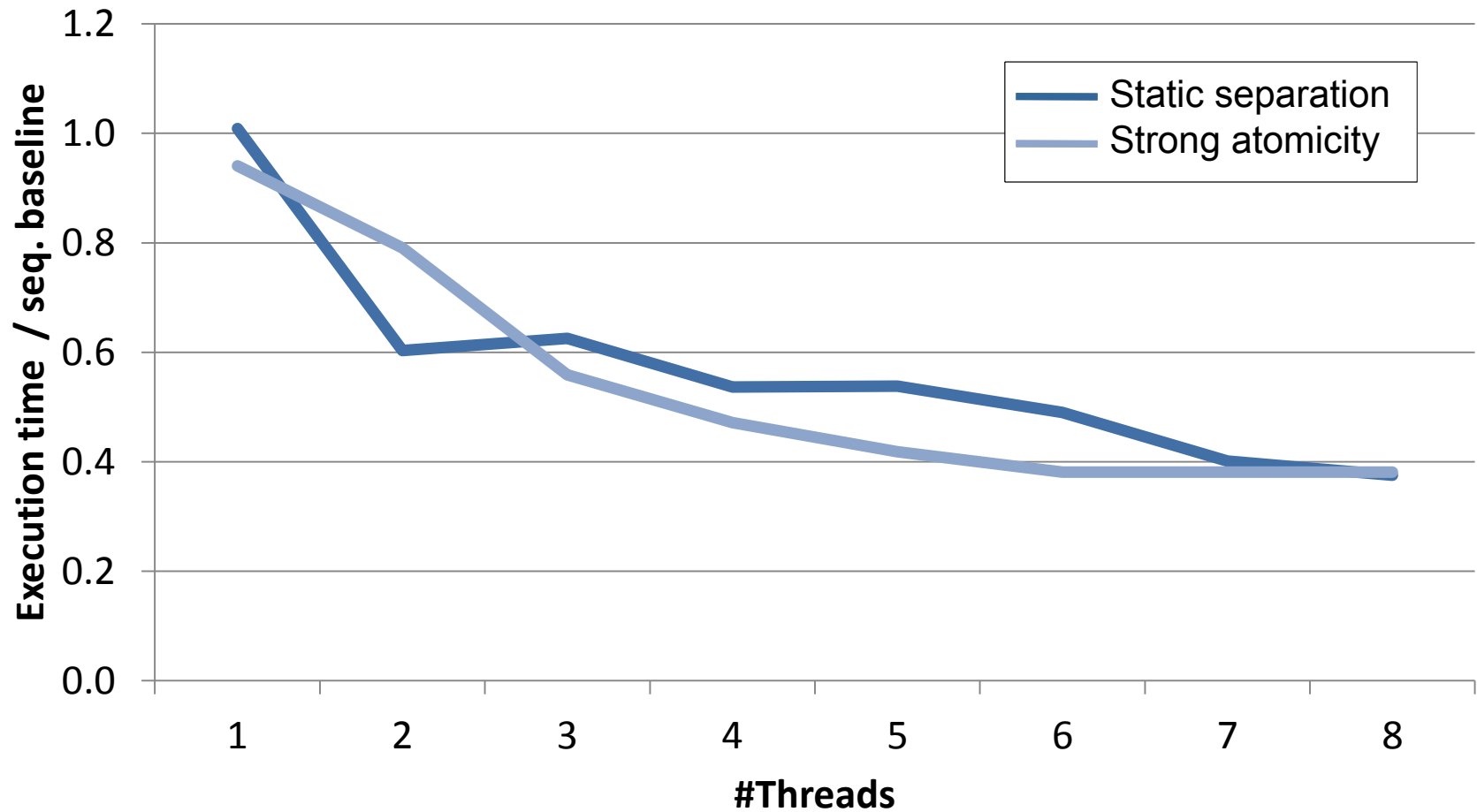
Sequential overhead



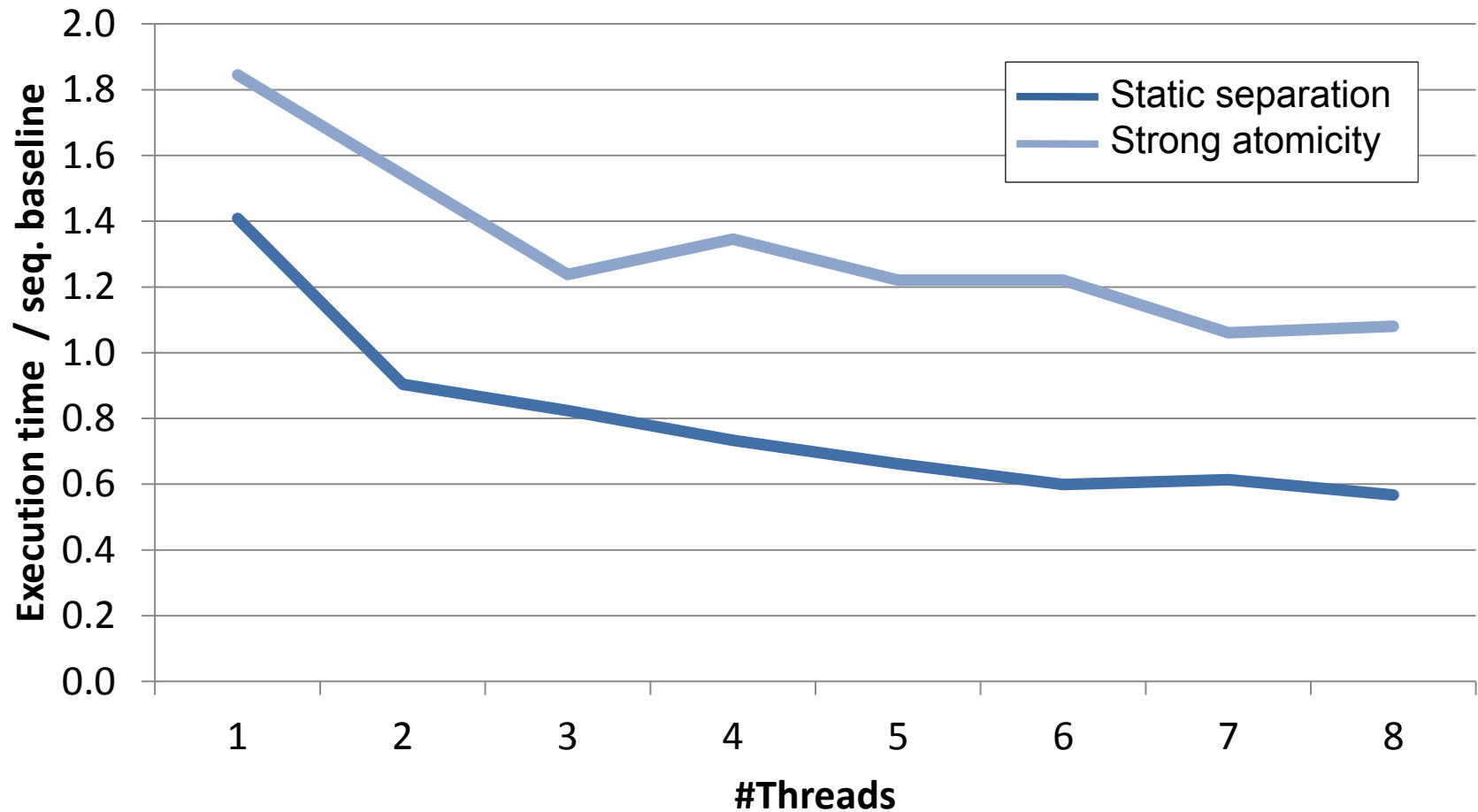
Scaling – Labyrinth



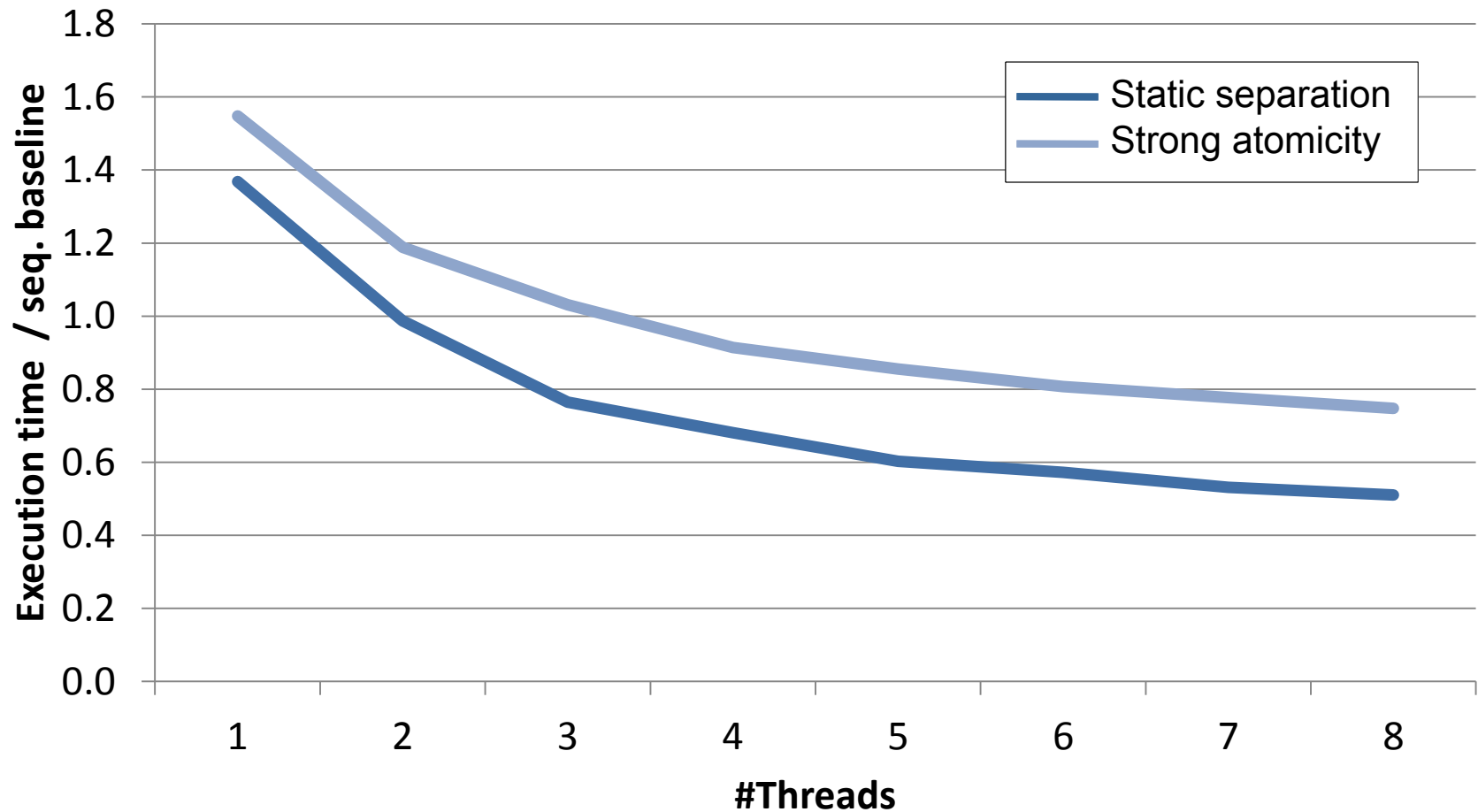
Scaling – Delaunay



Scaling – Genome



Scaling – Vacation

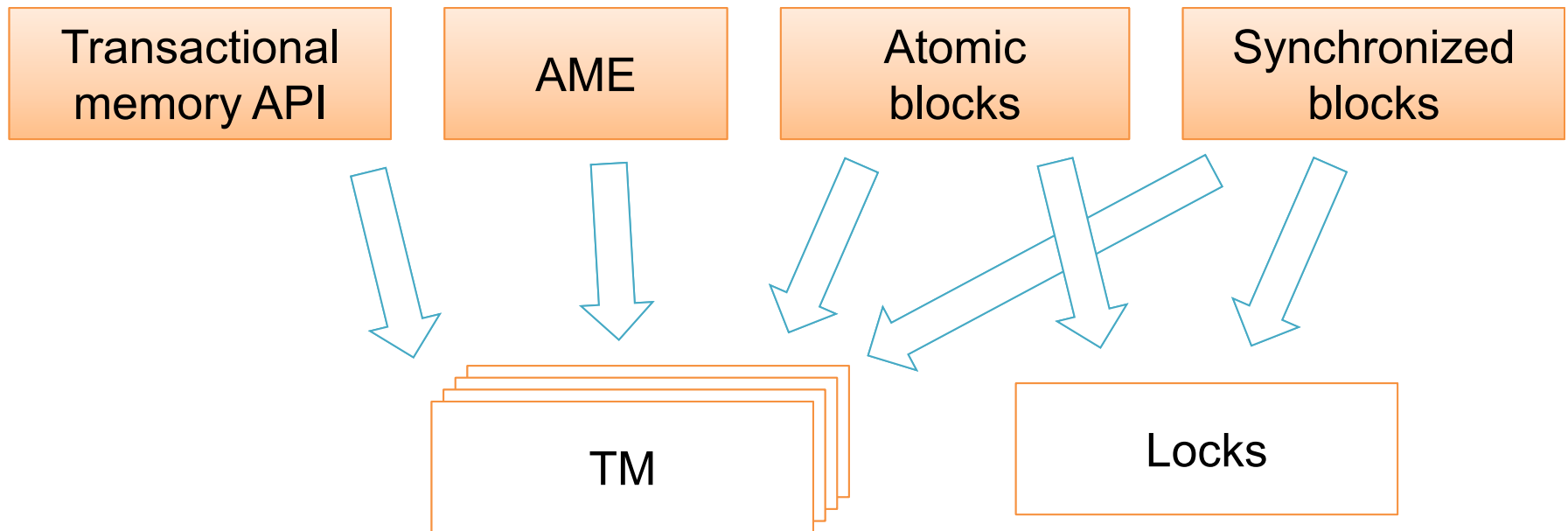


Untangling “atomic” from TM

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