

Lock-free programming and transactional memory

Tim Harris

4/4

Overview

Multi-core systems, and concurrent programming without locks

Transactional memory for managing shared-memory data structures

Integrating transactional memory into a modern, object-oriented programming language

Making sense of transactional memory: combining transactions with libraries, locking, and IO

Design questions

```

class Q {
    QElem leftSentinel;
    QElem rightSentinel;

    void add(int 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 about I/O?”

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

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


“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?”

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;
```

x_shared
== true

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

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

Example: a privatization idiom

`x_shared = true; x = 0;`

`x_shared == true`

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

Old val
`x=0`

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

Example: a privatization idiom

`x_shared = false; x = 0;`

`x_shared == true`

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

Old val
`x=0`

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

Example: a privatization idiom

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

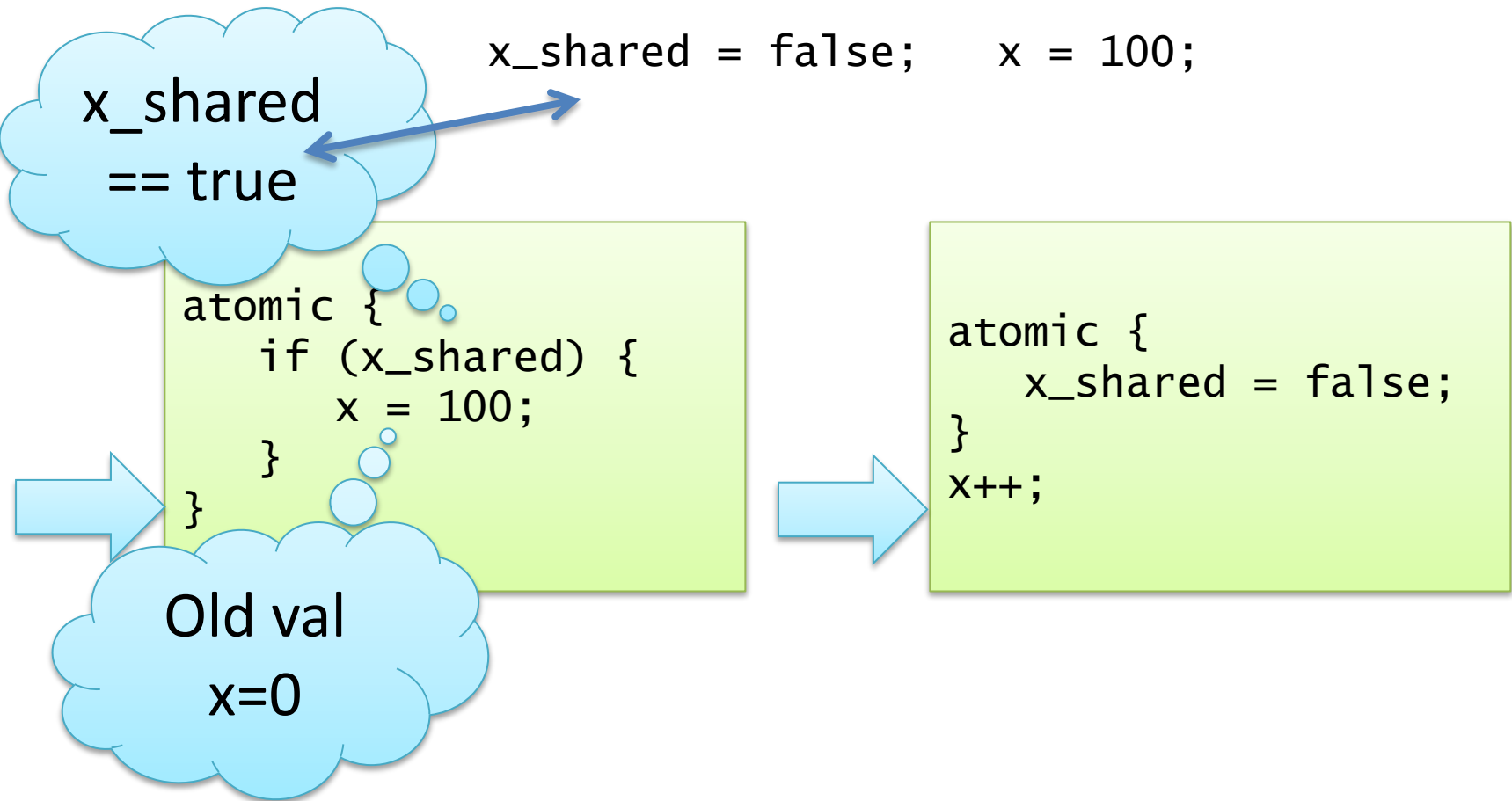
x_shared
== true

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

Old val
x=0


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

Example: a privatization idiom

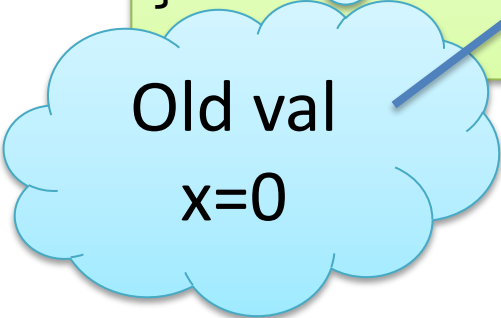


Example: a privatization idiom

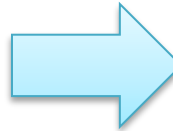
`x_shared = false; x = 0;`



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



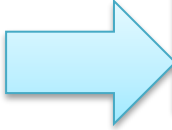
Old val
x=0




```
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++;
```

The main argument

Program

Threads,
atomic blocks

Language implementation

1. We need a methodical way to define these constructs.
2. We should focus on defining this programmer-visible interface, rather than the internal “TM” interface.

nitTx
ite

An analogy

Program

Garbage collected
“infinite” memory

Language implementation



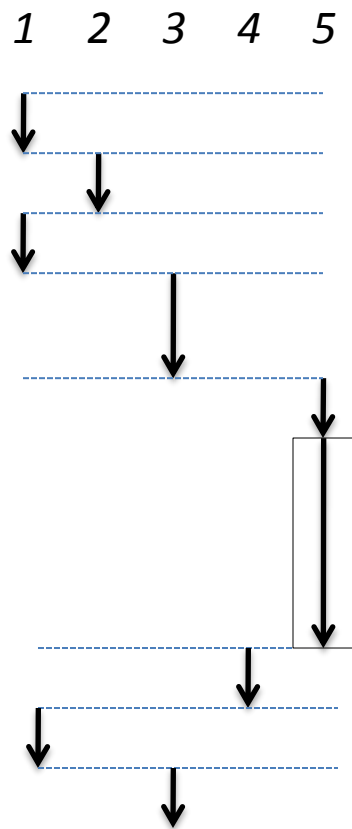
Low-level, broad,
platform-specific API,
no canonical def.

Defining “atomic”, not “TM”

Implementing atomic over TM

Current performance

Strong semantics: a simple interleaved model




Sequential interleaving of operations by threads.
 No program transformations (optimization, weak memory, etc.)


Thread 5 enters an atomic block: prohibits the interleaving of operations from other threads

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;  
    }  
}
```



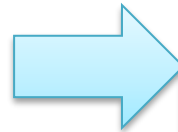
```
atomic {  
    x_shared = false;  
}  
x++;
```

Example: a privatization idiom

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




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



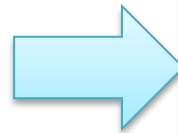
```
atomic {  
    x_shared = false;  
}  
x++;
```

Example: a privatization idiom

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




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



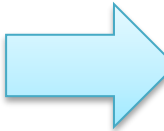
```
atomic {  
    x_shared = false;  
}  
x++;
```

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;  
    }  
}
```



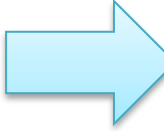
```
atomic {  
    x_shared = false;  
}  
x++;
```

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;  
    }  
}
```



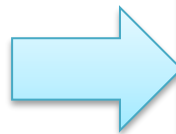
```
atomic {  
    x_shared = false;  
}  
x++;
```

Example: a privatization idiom

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




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



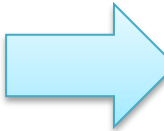
```
atomic {  
    x_shared = false;  
}  
x++;
```

Example: a privatization idiom

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



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



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

Pragmatically, do we care about...

```
x = 0;
```

```
atomic {  
    x = 100;  
    x = 200;  
}
```

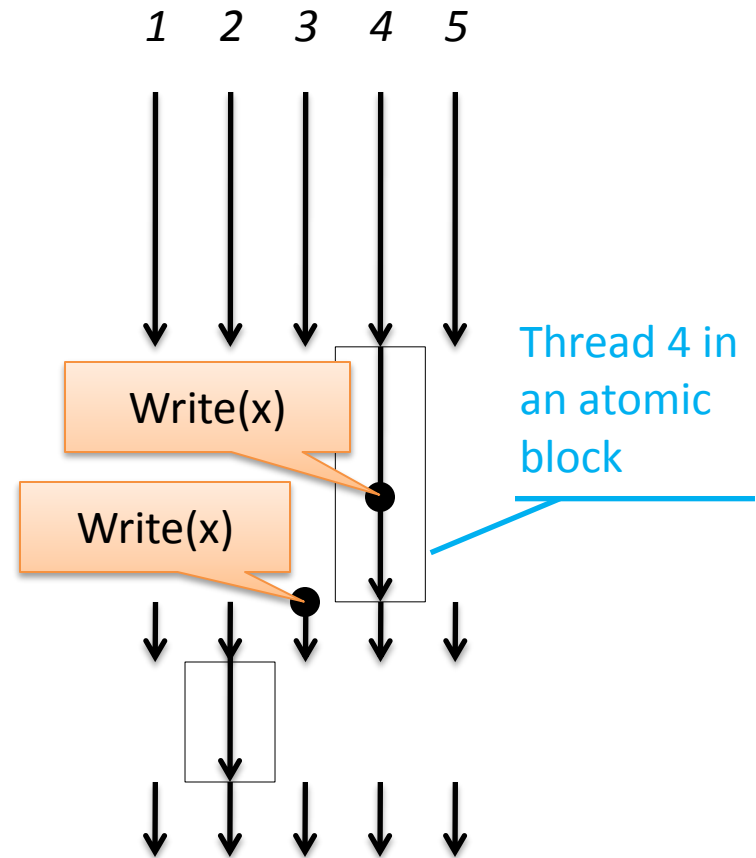
```
temp = x;  
Console.WriteLine(temp);
```

How: strong semantics for race-free programs

Strong semantics: simple interleaved model of multi-threaded execution

Data race: concurrent accesses to the same location, at least one a write

Race-free: no data races (under strong semantics)



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, ...

Example: a privatization idiom

Correctly synchronized: no concurrent access to “x” under strong semantics

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

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

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

Example: a “racy” publication idiom

Not correctly synchronized: race on “x_shared” under strong semantics

```
x_shared = false;    x = null;
```

```
atomic {  
    x = new Foo(...);  
    x_shared = true;  
}
```

```
if (x_shared) {  
    // Use x  
}
```

What about...

- ...I/O?
- ...volatile fields?
- ...locks inside/outside atomic blocks?
- ...condition variables?

Methodical approach: what happens under the simple, interleaved model?

1. Ideally, what does it do?
2. Which uses are race-free?

What about I/O?

```
atomic {  
    Console.WriteLine("what is your name?");  
    x = Console.ReadLine();  
    Console.WriteLine("Hello " + x);  
}
```

The entire write-read-write sequence should run (as if) without interleaving with other threads

What about C#/Java volatile fields?

```
volatile int x, y = 0;
```

```
atomic {  
    x = 5;  
    y = 10;  
    x = 20;  
}
```

```
r1 = x;
```

```
r2 = y;
```

```
r3 = x;
```

r1=20, r2=10, r3=20

r1=0, r2=10, r3=20

r1=0, r2=0, r3=20

r1=0, r2=0, r3=0

What about locks?

Correctly synchronized: both threads would need “obj1” to access “x”

```
atomic {  
    lock(obj1);  
    x = 42;  
    unlock(obj1);  
}
```

```
lock(obj1);  
x = 42;  
unlock(obj1);
```

What about locks?

Not correctly synchronized: no consistent synchronization

```
atomic {  
    x = 42;  
}
```

```
lock(obj1);  
x = 42;  
unlock(obj1);
```

What about condition variables?

Correctly synchronized: ...and works OK in this example

```
atomic {  
    lock(buffer);  
    while (!full) buffer.wait();  
    full = true;  
    ...  
    unlock(buffer);  
}
```

What about condition variables?

Correctly synchronized: ...but program doesn't work in this example

Programmer says must
run atomically

```
atomic {  
  lock(barrier);  
  waiters ++;  
  while (waiters < N) {  
    barrier.wait();  
  }  
  unlock(barrier);  
}
```

Should run before
waiting

Should run after
waiting

Defining “atomic”, not “TM”

Implementing atomic over TM

Current performance

Division of responsibility

Desired semantics
atomic blocks, retry, ...



STM primitives
StartTx, CommitTx, ReadTx, WriteTx, ...

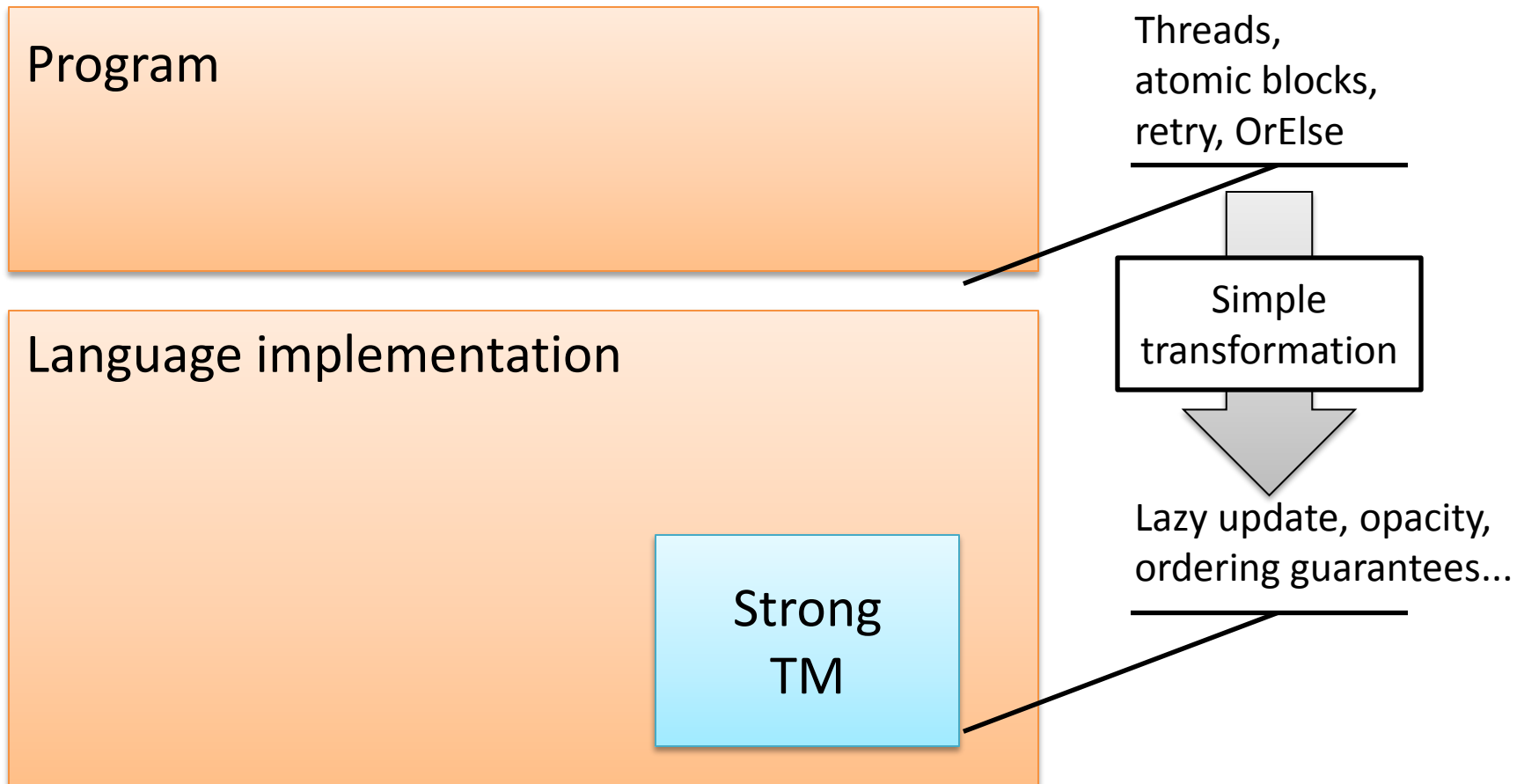


Hardware primitives
Conventional h/w: read, write, CAS

Build strong guarantees
by segregating tx /
non-tx in the runtime
system

Lets us keep a very
relaxed view of what
the STM must do...
zombie tx, etc

Implementation 1: “classical” atomic blocks on TM



Implementation 2: very weak TM

Program

Threads,
atomic blocks

Language implementation

Isolation of
tx via MMU

Program
analyses

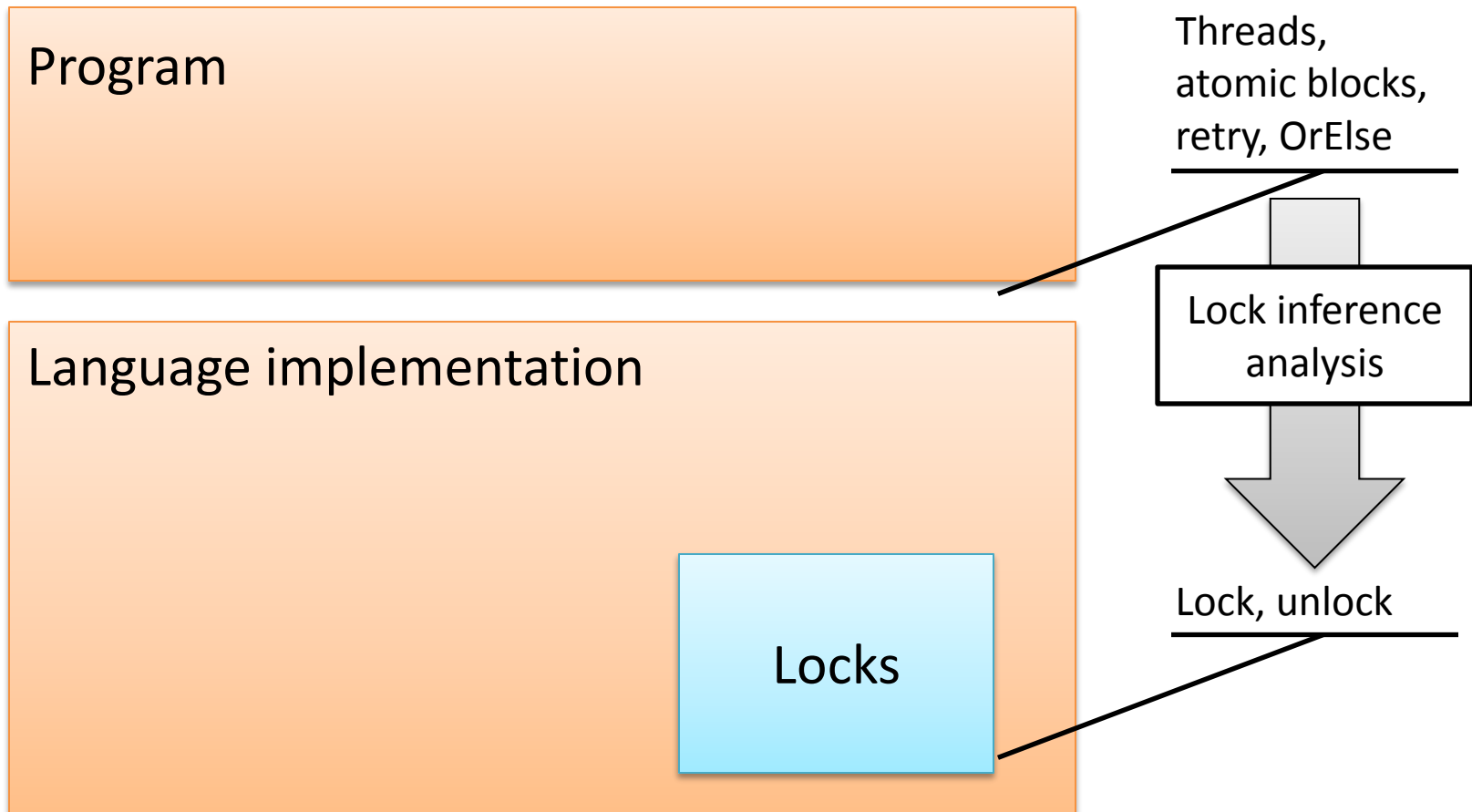
GC
support

Sandboxing
for zombies

Very weak
STM

StartTx, CommitTx,
ValidateTx,
ReadTx(addr)->val,
WriteTx(addr, val)

Implementation 3: lock inference



Integrating non-TM features

- **Prohibit**
- Directly execute on
- Use irrevocable execution
- Integrate it with TM

Normal mutable state in STM-Haskell

“Dangerous” feature combinations, e.g,
condition variables inside atomic blocks

Integrating non-TM features

- Prohibit
- **Directly execute over TM**
- Use irrevocable ex
- Integrate it with TM

e.g., an “ordinary” library abstraction
used in an atomic block

Is this possible?

Will it scale well?

Will this be correctly synchronized?

Integrating non-TM features

- Prohibit
- Directly execute over TM
- **Use irrevocable execution**
- Integrate it with TM

Prevent roll-back, ensure the transaction wins all conflicts.

Fall-back case for I/O operations.
Use for rare cases, e.g., class initializers

Integrating non-TM features

- Prohibit
- Directly execute over TM
- Use irrevocable execution
- **Integrate it with TM**

Provide conflict detection, recovery, etc,
e.g. via 2-phase commit

Low-level integration of GC, memory
management, etc.

Defining “atomic”, not “TM”

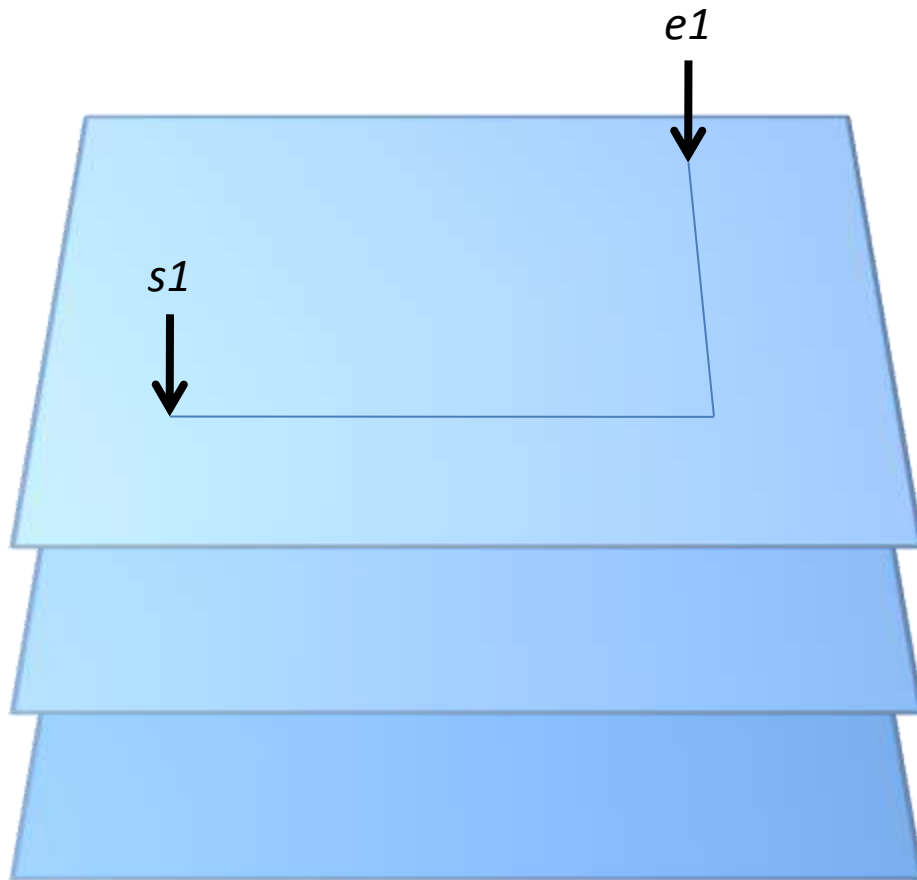
Implementing atomic over TM

Current performance

Performance figures depend on...

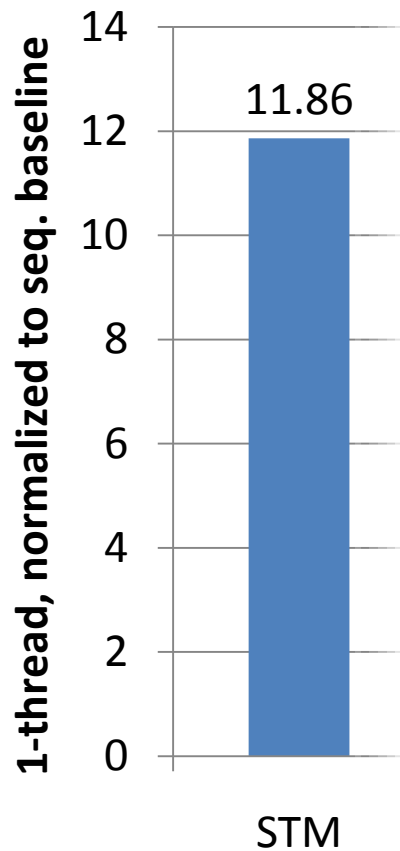
- **Workload** : What do the atomic blocks do? How long is spent inside them?
- **Baseline implementation**: Mature existing compiler, or prototype?
- **Intended semantics**: Support static separation? Violation freedom (TDRF)?
- **STM implementation**: In-place updates, deferred updates, eager/lazy conflict detection, visible/invisible readers?
- **STM-specific optimizations**: e.g. to remove or downgrade redundant TM operations
- **Integration**: e.g. dynamically between the GC and the STM, or inlining of STM functions during compilation
- **Implementation effort**: low-level perf tweaks, tuning, etc.
- **Hardware**: e.g. performance of CAS and memory system

Labyrinth



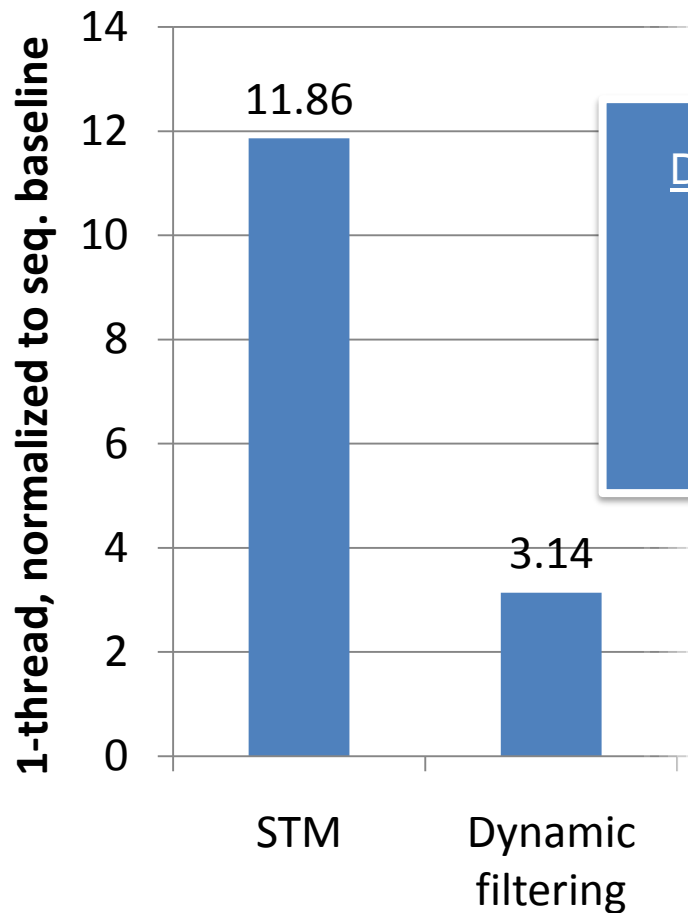
- STAMP v0.9.10
- 256x256x3 grid
- Routing 256 paths
- Almost all execution inside atomic blocks
- Atomic blocks can attempt 100K+ updates
- C# version derived from original C
- Compiled using Bartok, whole program mode, C# -> x86 (~80% perf of original C with VS2008)
- Overhead results with Core2 Duo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

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

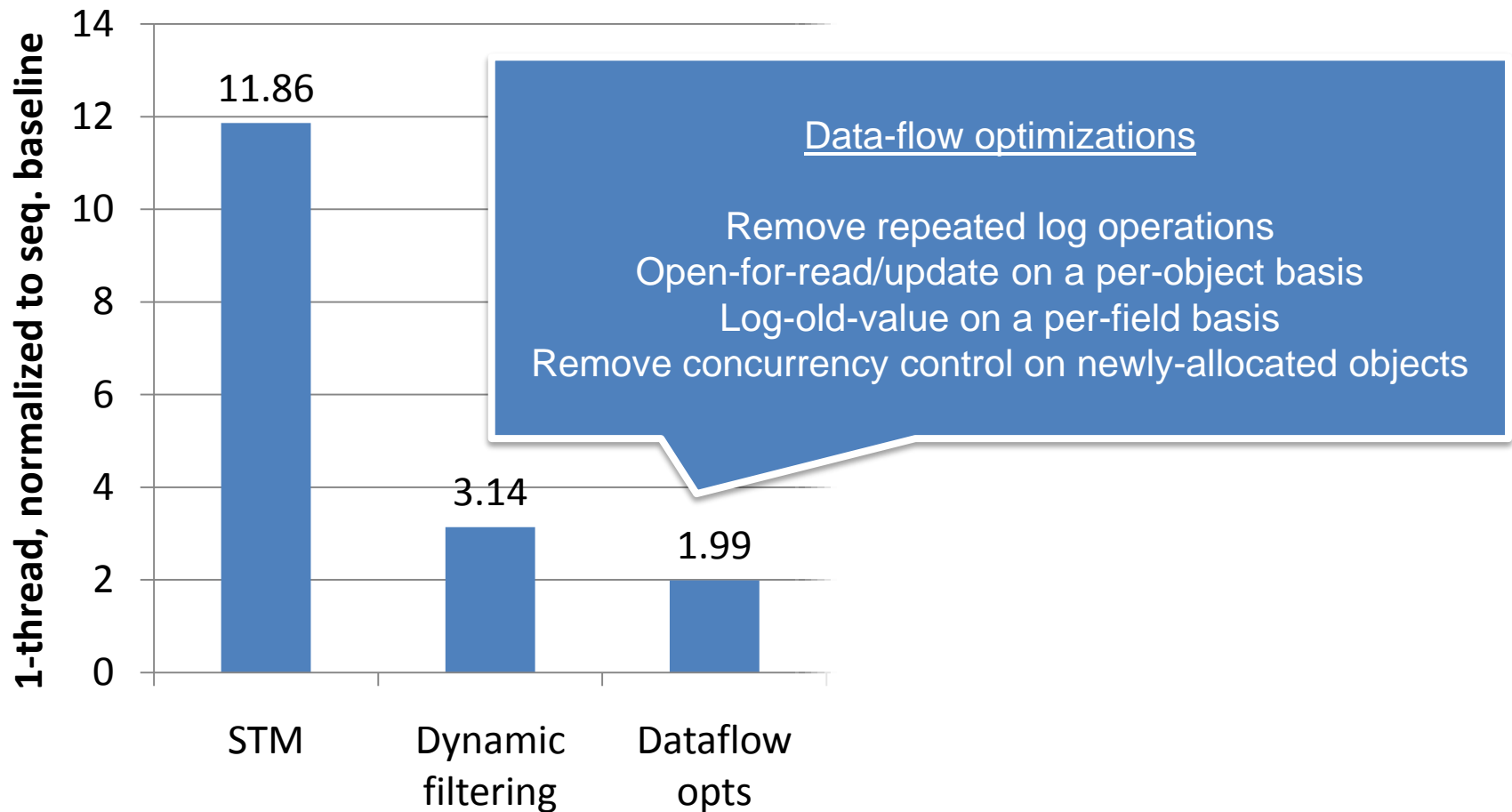
Sequential overhead



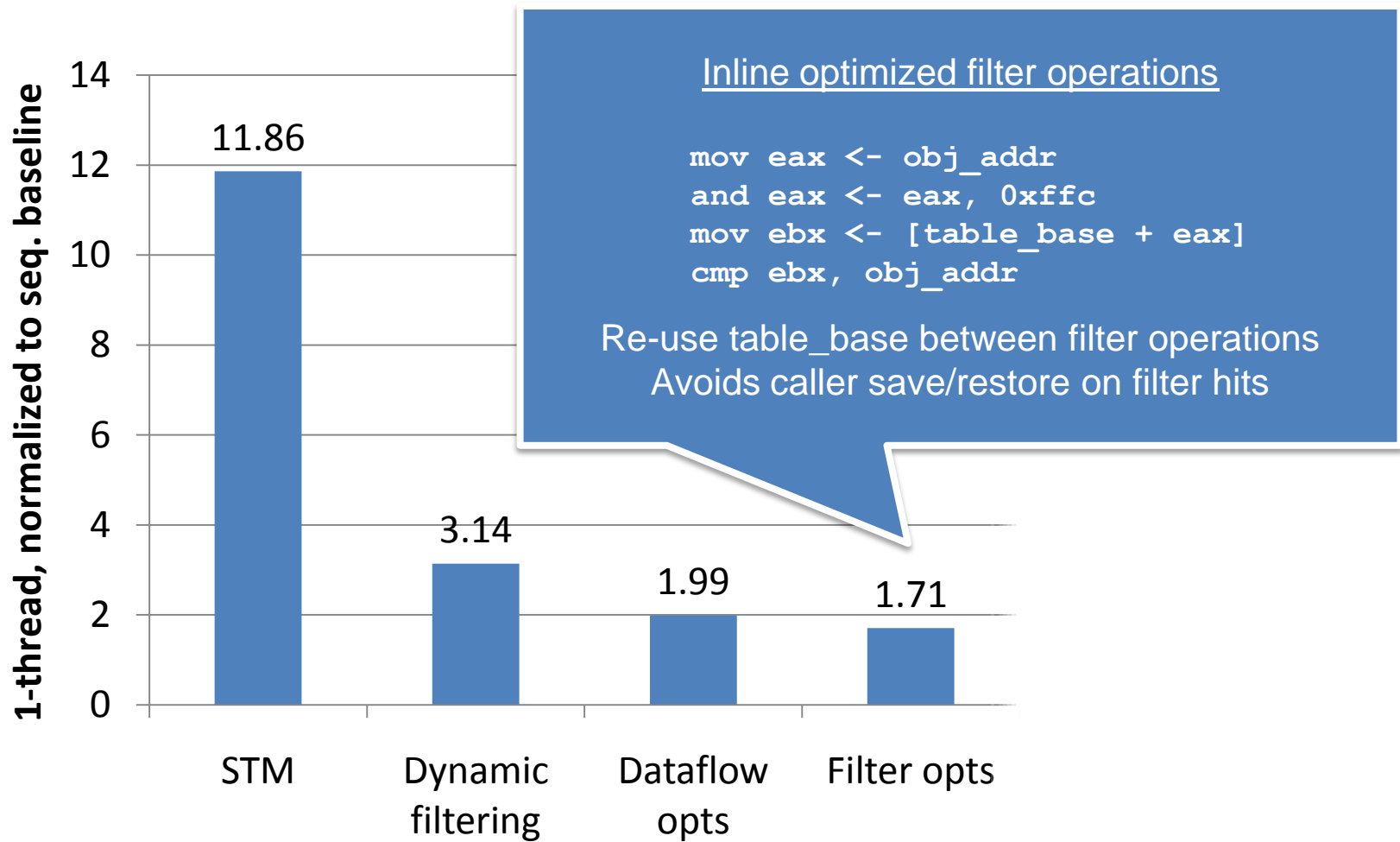
Dynamic filtering to remove redundant logging

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

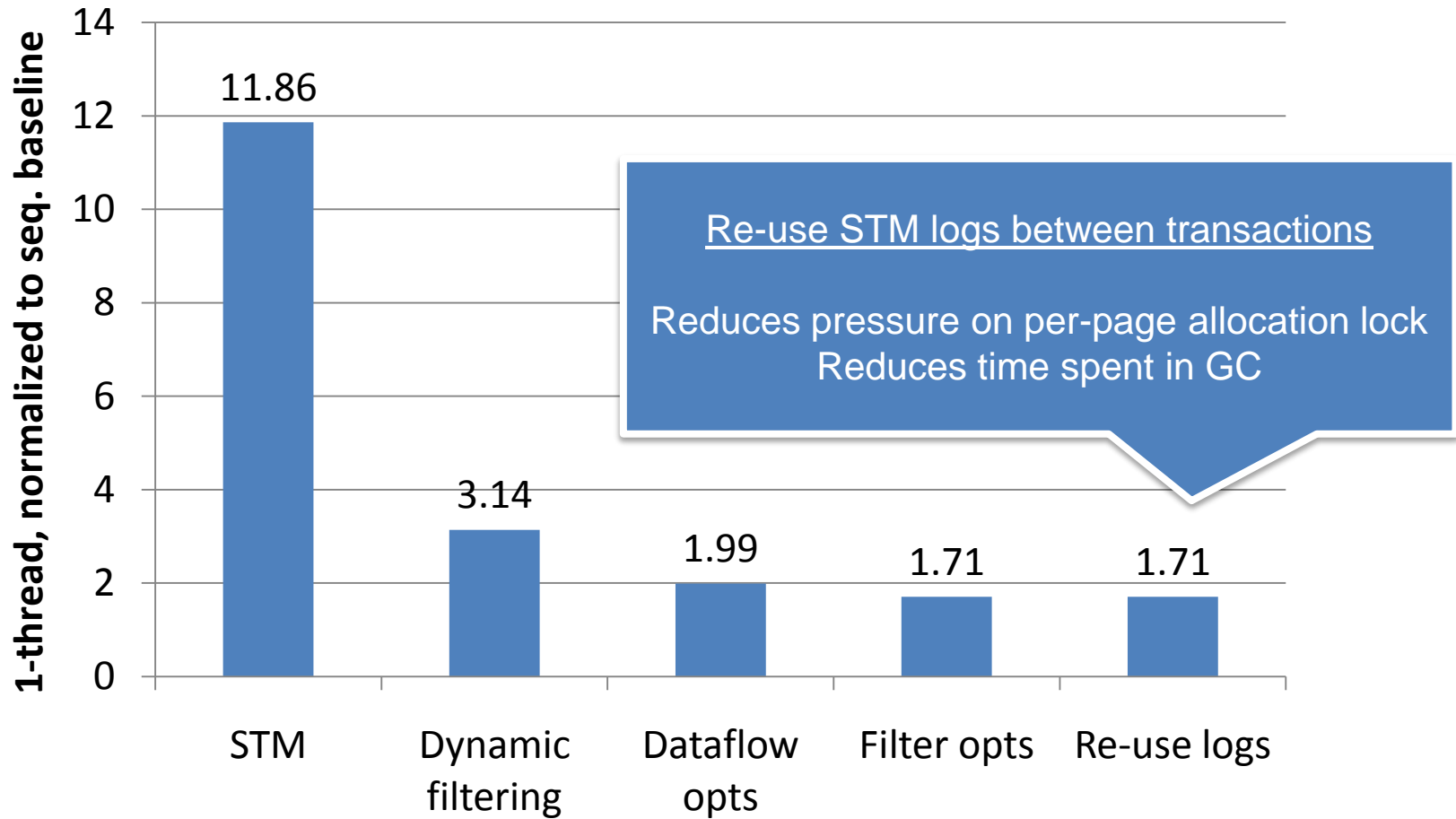
Sequential overhead



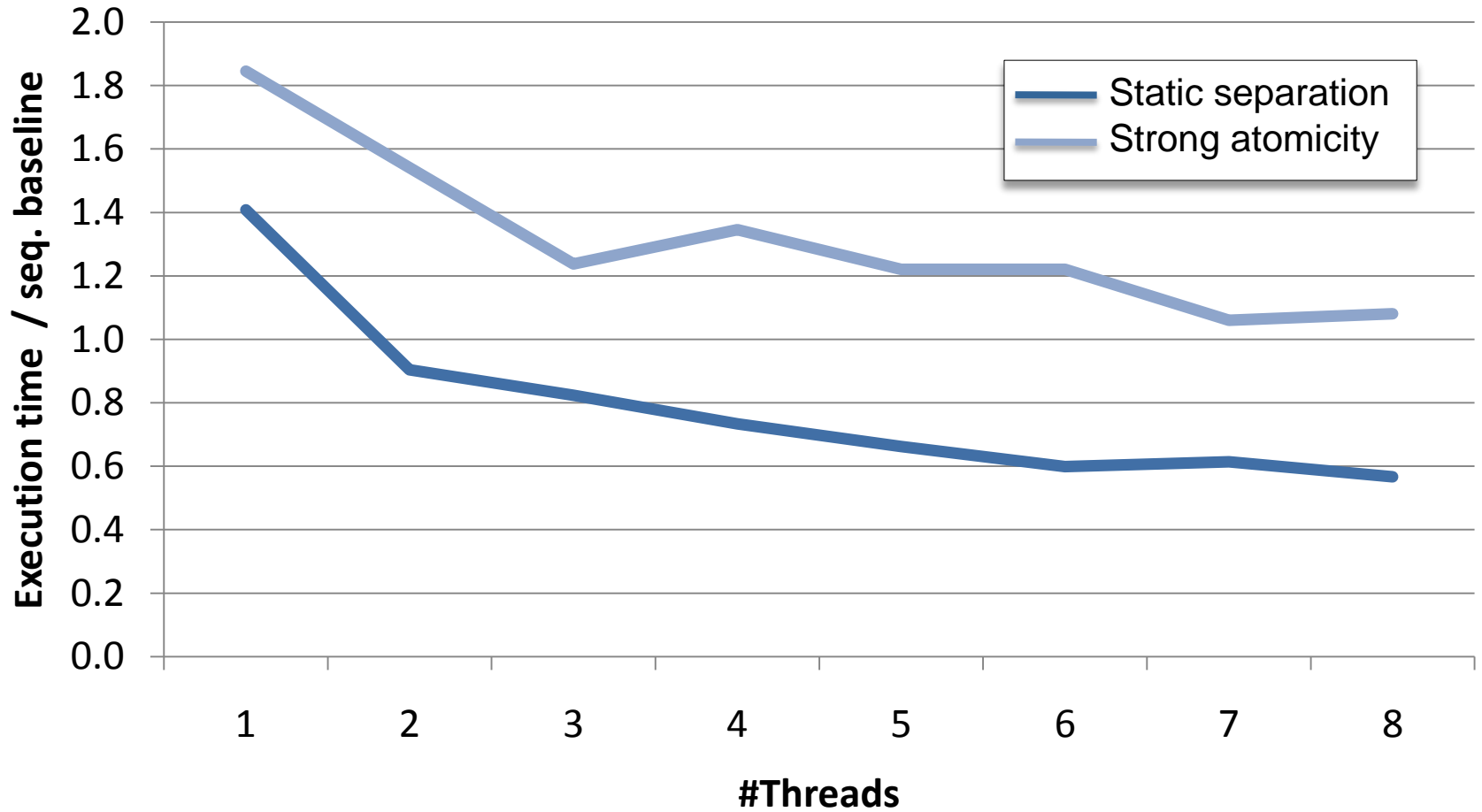
Sequential overhead



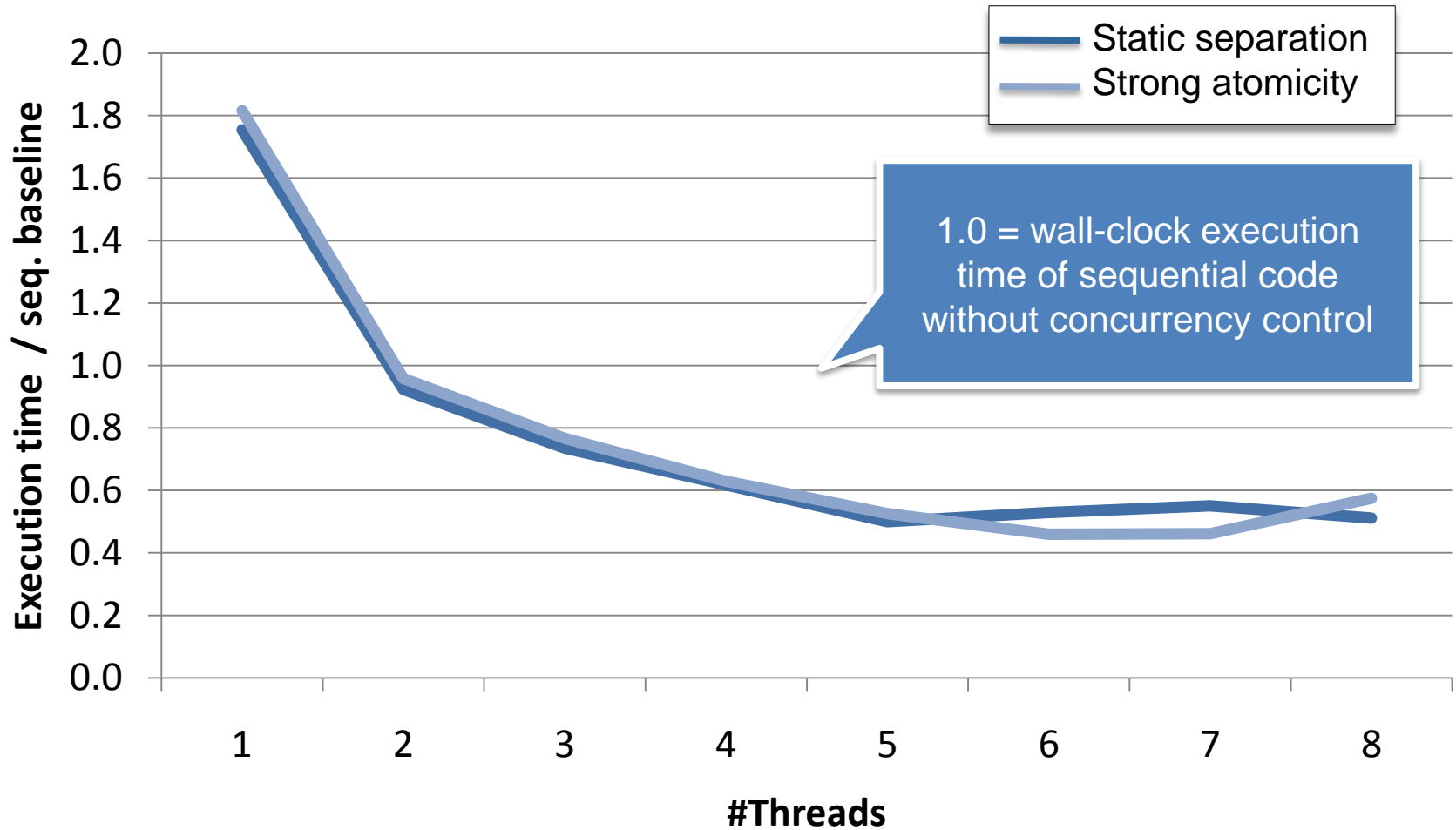
Sequential overhead



Scaling – Genome



Scaling – Labyrinth



Making sense of TM

- Focus on the interface between the language and the programmer
 - Talk about atomicity, not TM
 - Permit a range of tx and non-tx implementations
- Define idealized “strong semantics” for the language (c.f. sequential consistency)
- Define what it means for a program to be “correctly synchronized” under these semantics
- Treat complicated cases methodically (I/O, locking, etc)

Overview

Multi-core systems, and concurrent programming without locks

Transactional memory for managing shared-memory data structures

Integrating transactional memory into a modern, object-oriented programming language

Making sense of transactional memory: combining transactions with libraries, locking, and IO