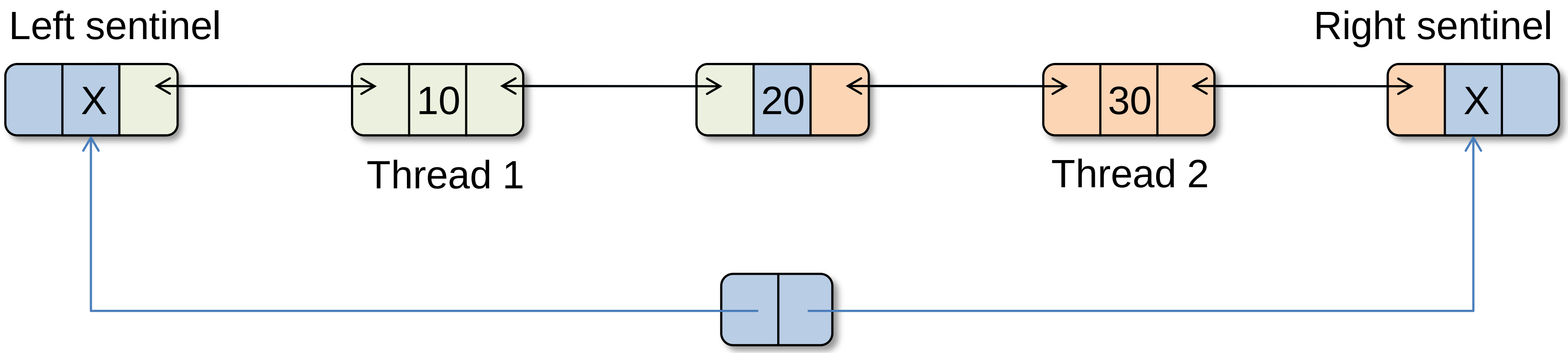
Making sense of transactional memory

Tim Harris (MSR Cambridge)

Based on joint work with colleagues at MSR Cambridge, MSR Mountain View,
MSR Redmond, the Parallel Computing Platform group, Barcelona Supercomputing Centre,
and the University of Cambridge Computer Lab

Example: double-ended queue



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

Implementing this: atomic blocks

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

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

  ...
}
```

Design questions

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

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

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

“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

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

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;
```

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

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

Old val
x=0

Example: a privatization idiom

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

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

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

The main argument

Program

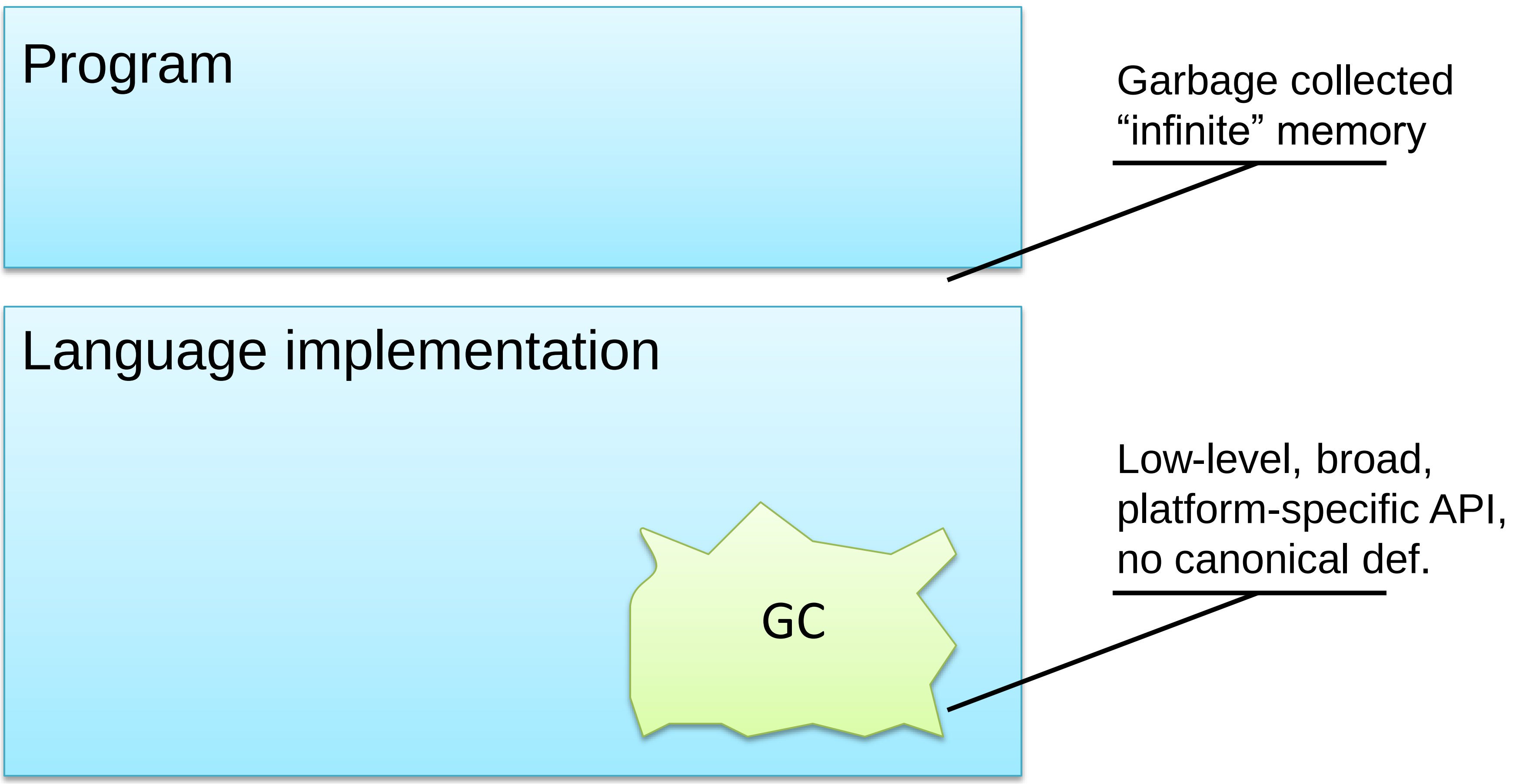
Threads,
atomic blocks

Language implementation

mitTx
Write

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.

An analogy

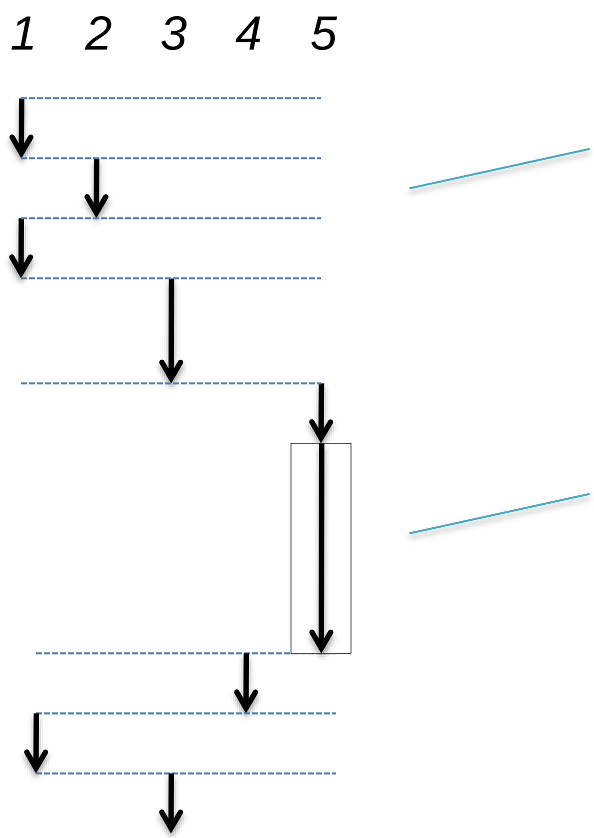

Program
Language implementation
GC
Garbage collected "infinite" memory
Low-level, broad, platform-specific API, no canonical def.

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

Microsoft Research

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

Microsoft Research

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++;
```

Microsoft Research

Execution 2

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

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;
}
```

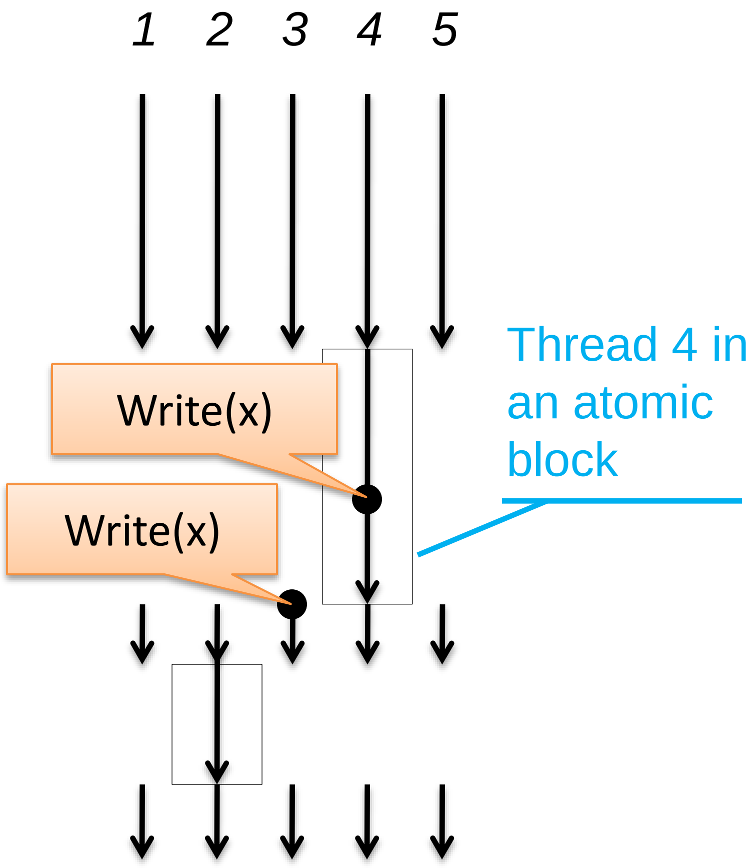
```
temp = x;
Console.WriteLine(temp);
```

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

<u>Strong semantics</u>

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

<u>Programming discipline(s)</u>

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

<u>Low-level semantics & actual implementations</u>

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
}
```

Microsoft Research

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

Microsoft Research

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

<u>Desired semantics</u>
atomic blocks, retry, ...

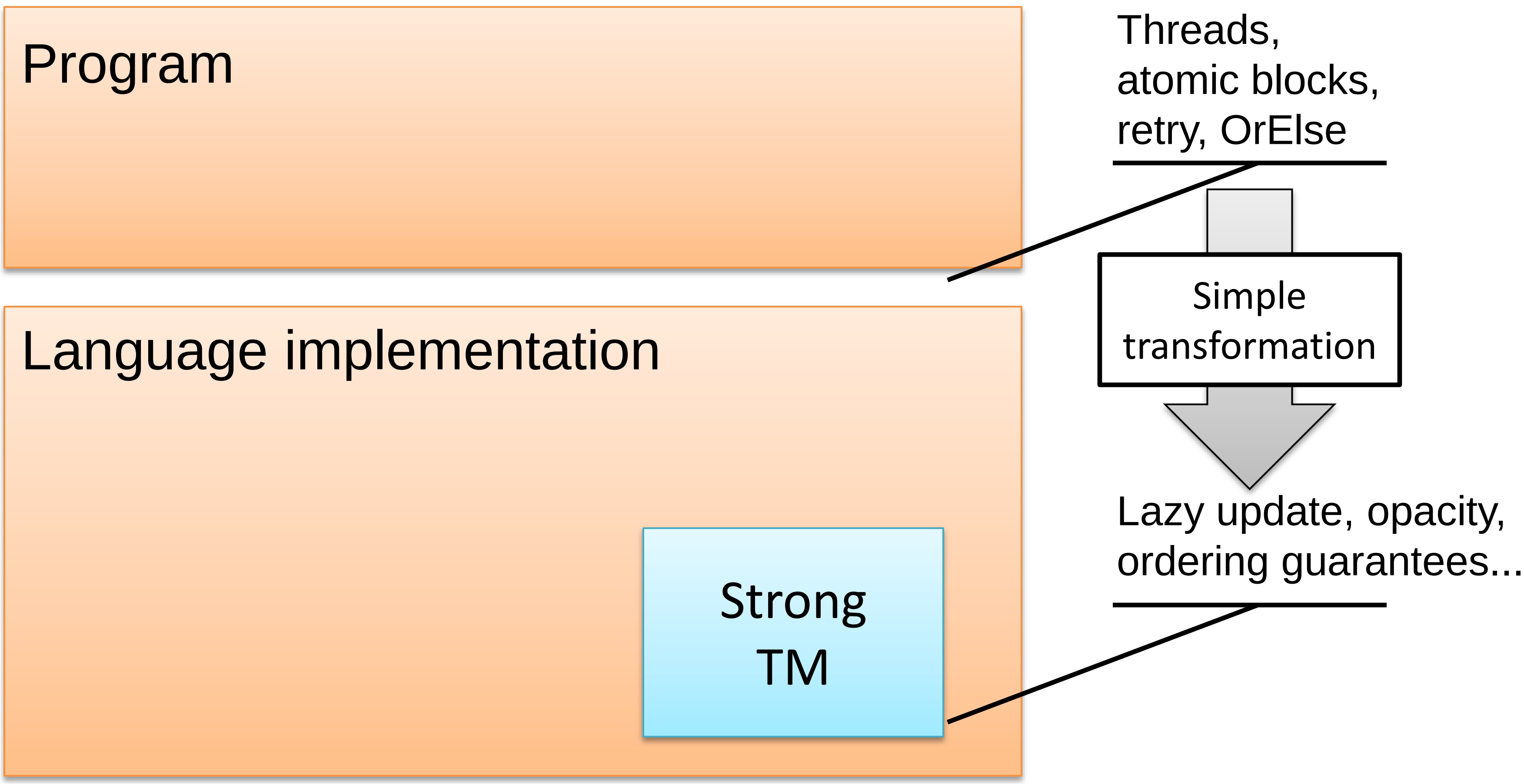
<u>STM primitives</u>
StartTx, CommitTx, ReadTx, WriteTx, ...

<u>Hardware primitives</u>
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



Microsoft Research

Implementation 2: very weak TM

Program

Threads, atomic blocks

Language implementation

Program analyses

Isolation of tx via MMU

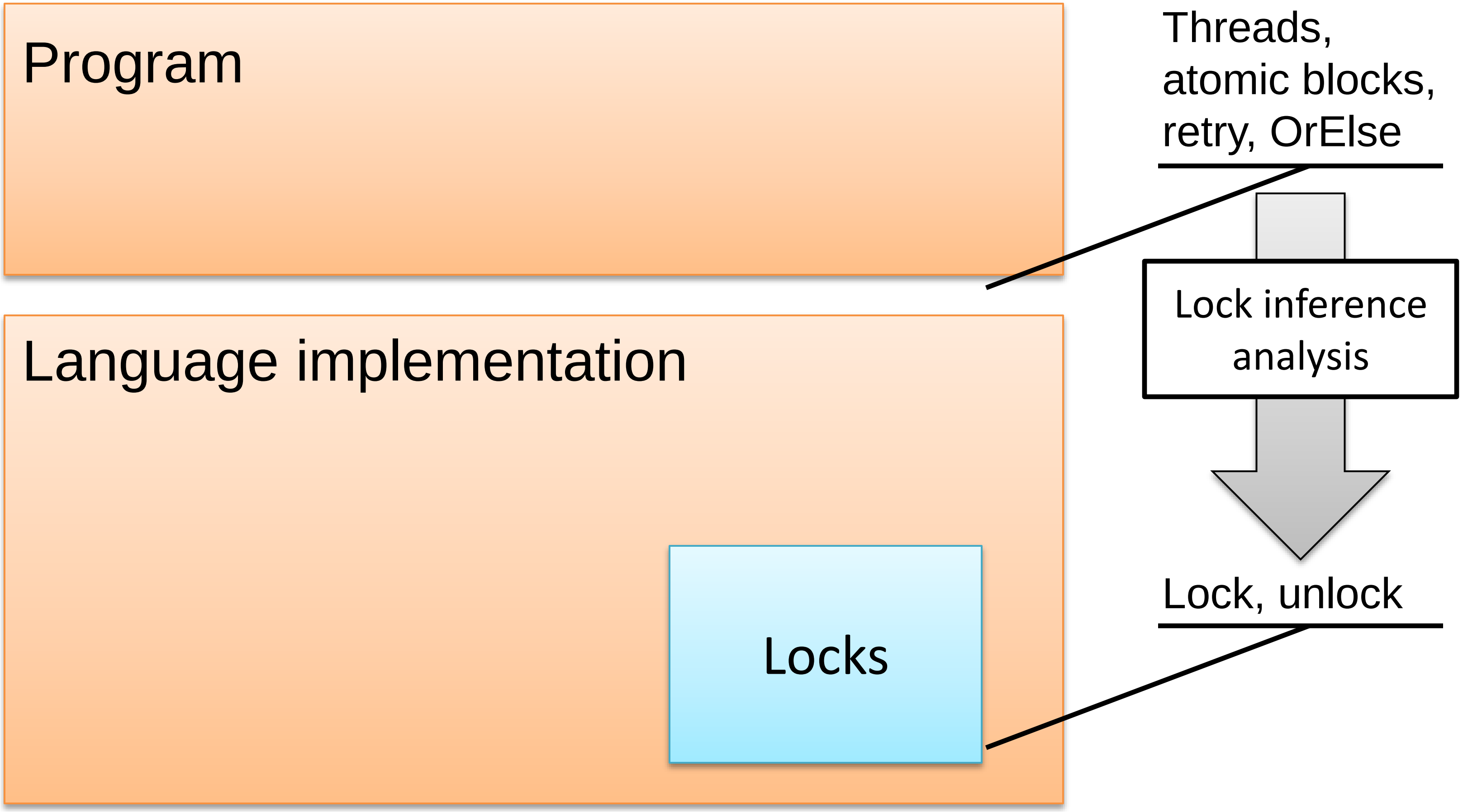
GC support

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 over
- 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 exec
- 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.

Microsoft Research

Defining “atomic”, not “TM”

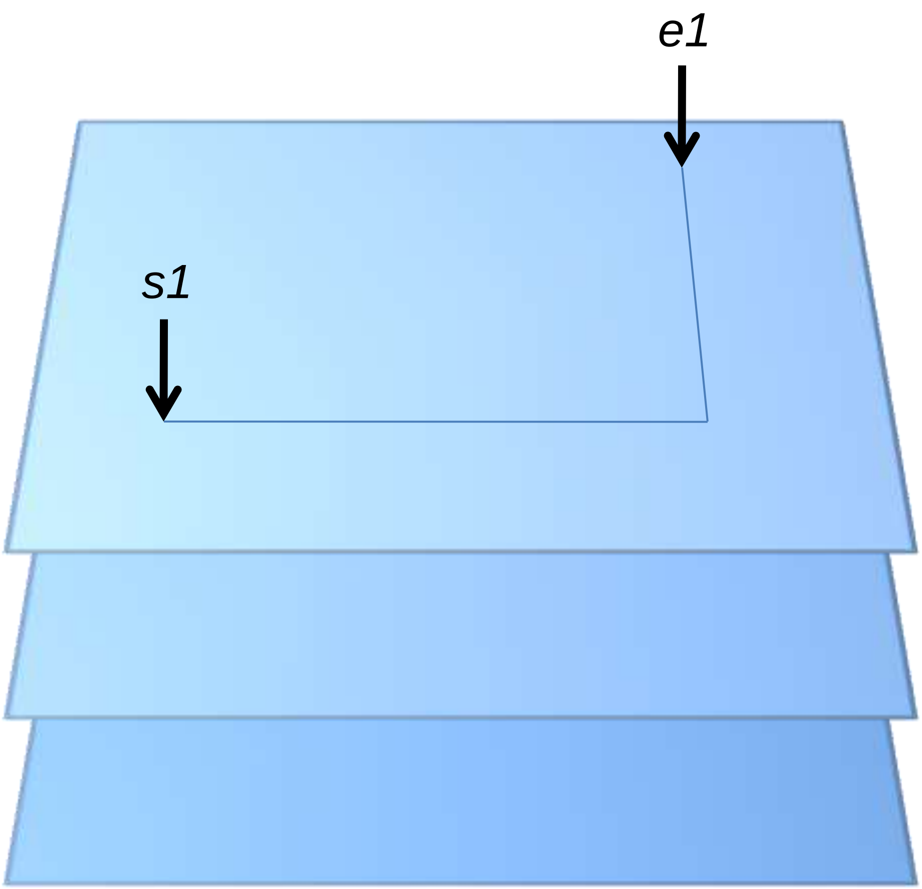
Implementing atomic over TM

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

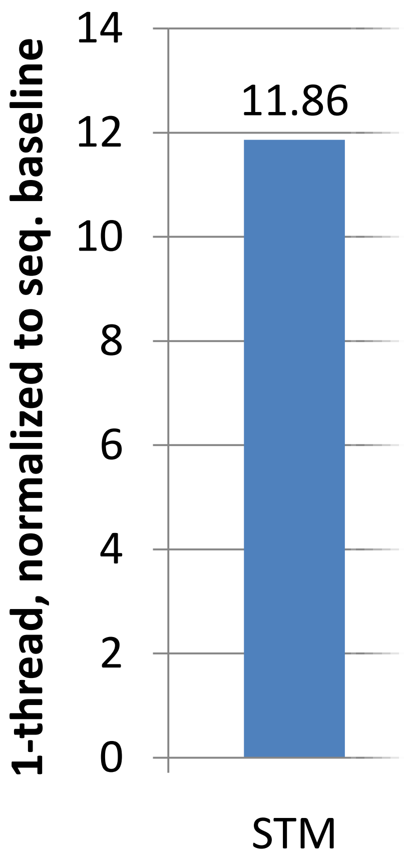
Labyrinth



- 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

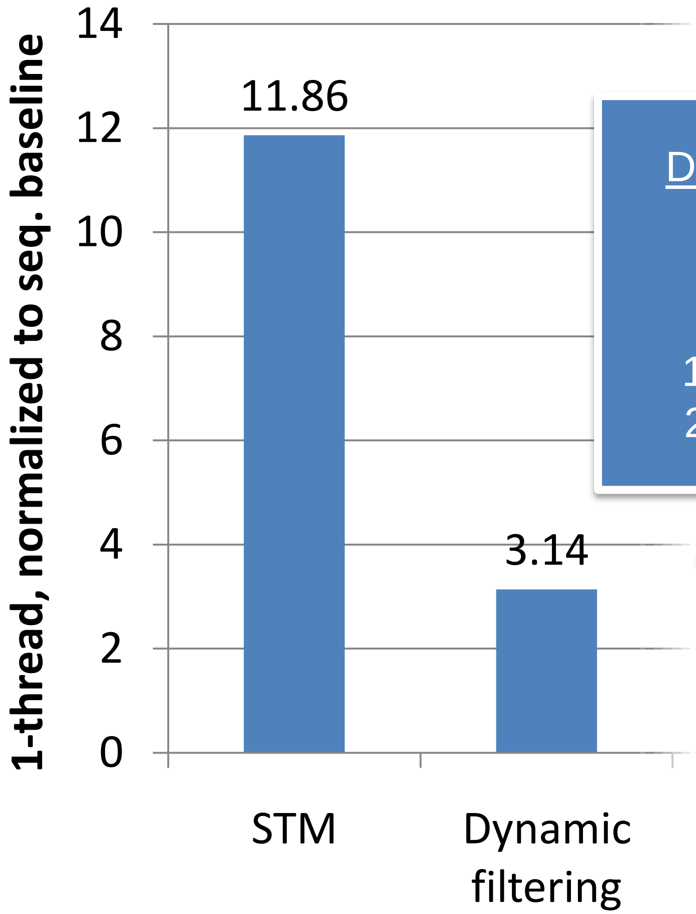
"STAMP: Stanford Transactional Applications for Multi-Processing"
Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun , IISWC 2008

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

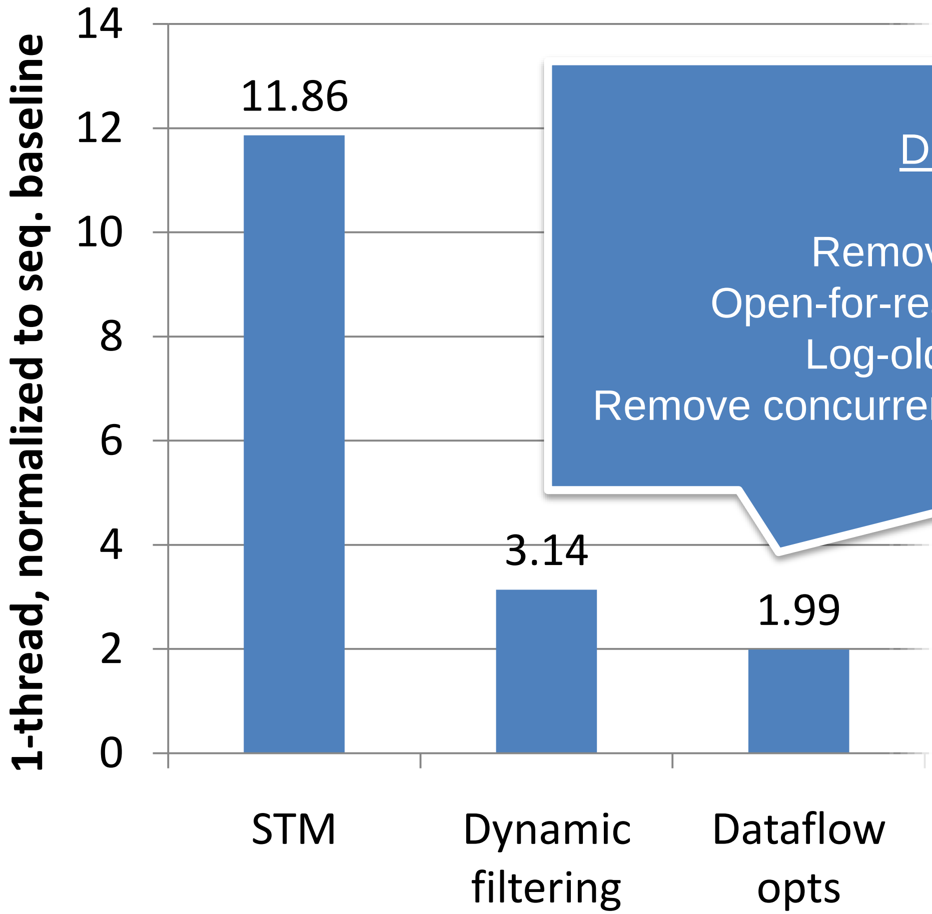


Dynamic filtering to remove redundant logging

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

Microsoft Research

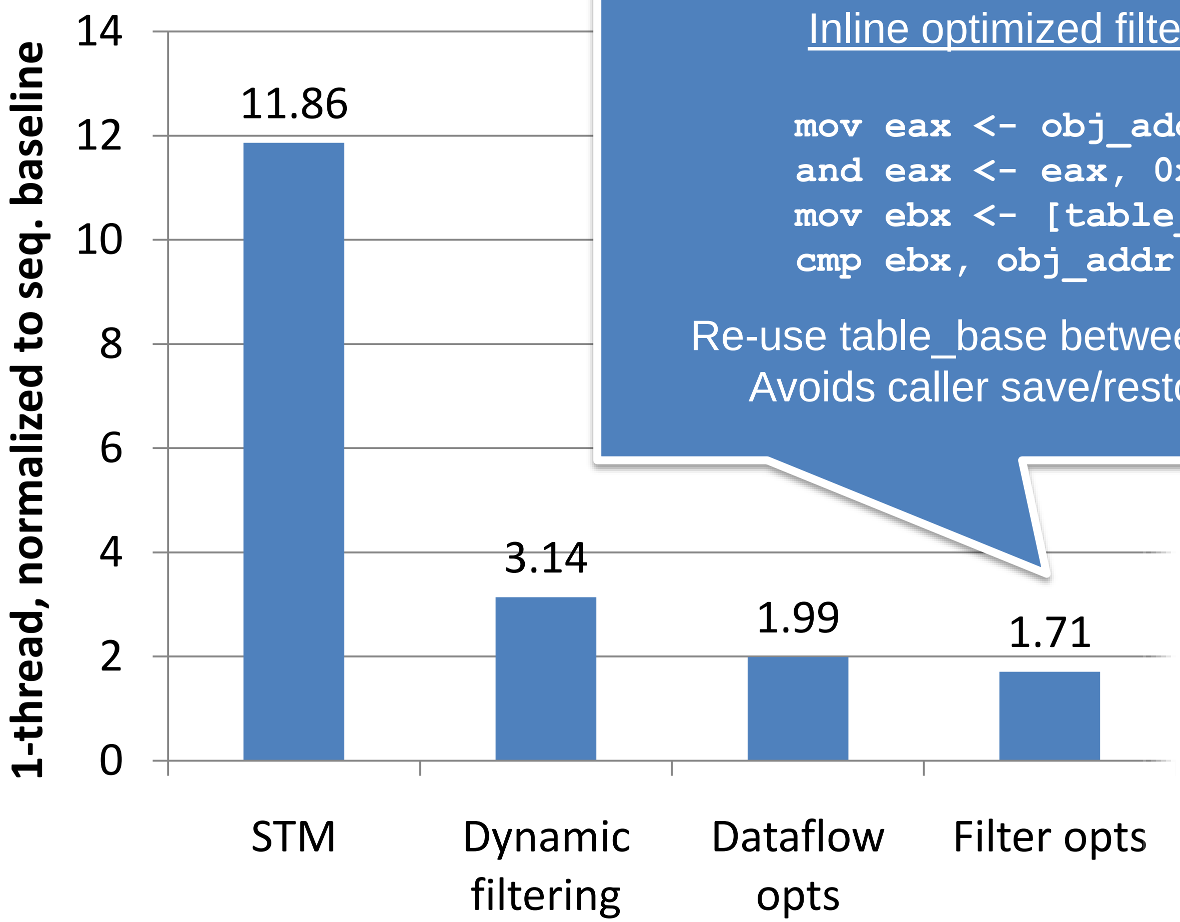
Sequential overhead



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

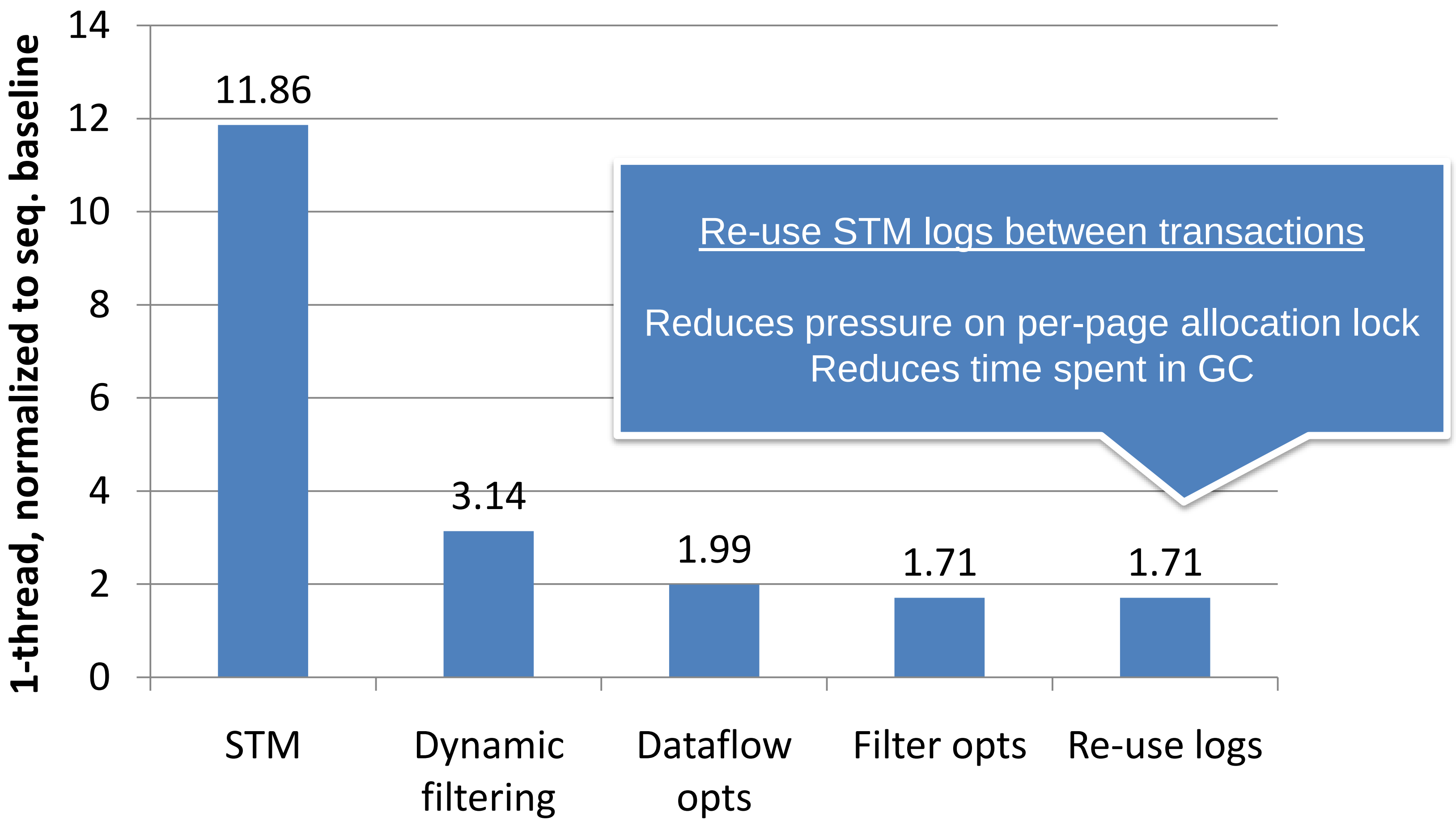


<u>Inline optimized filter operations</u>

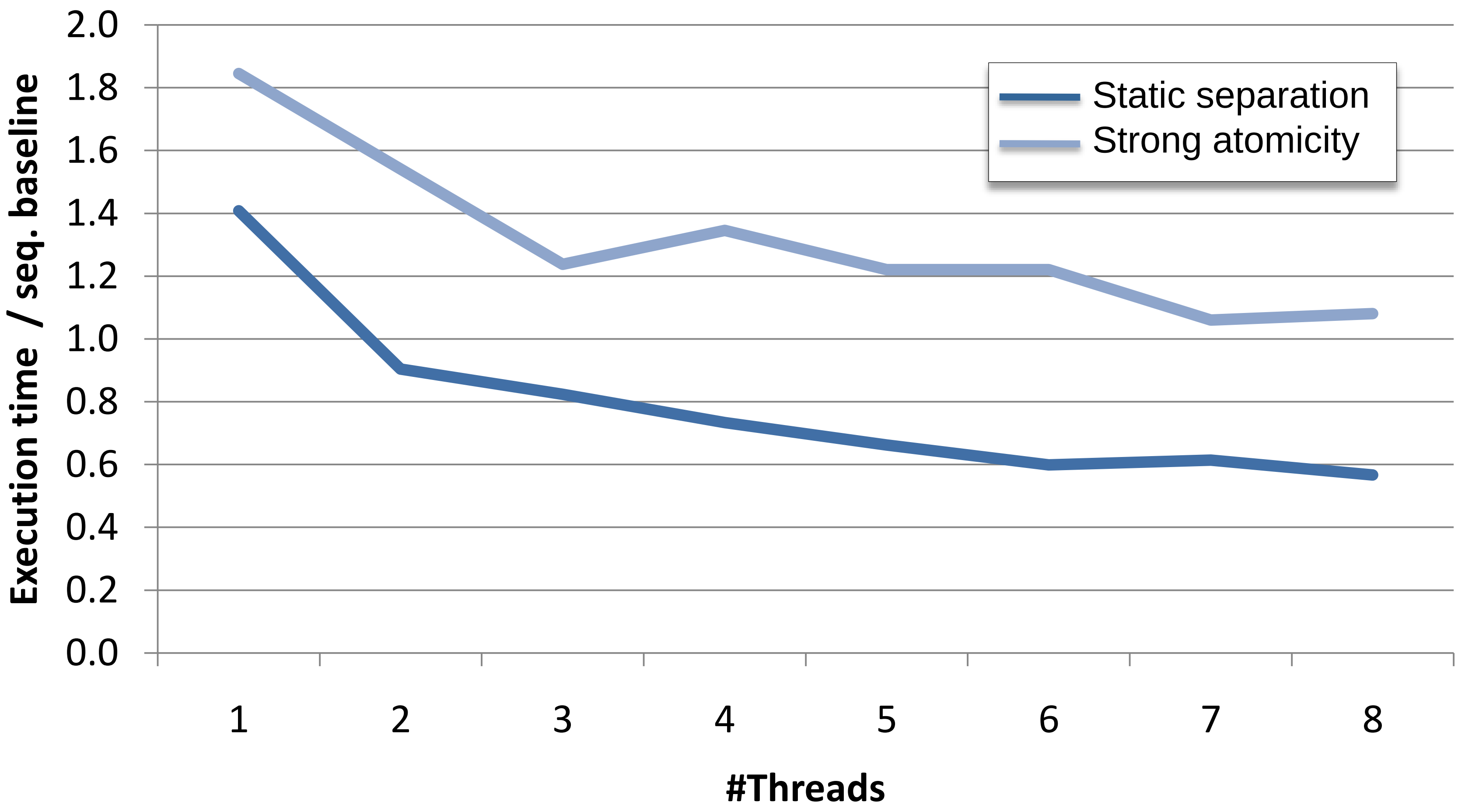
```
mov eax <- obj_addr
and eax <- eax, 0xffc
mov ebx <- [table_base + eax]
cmp ebx, obj_addr
```

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

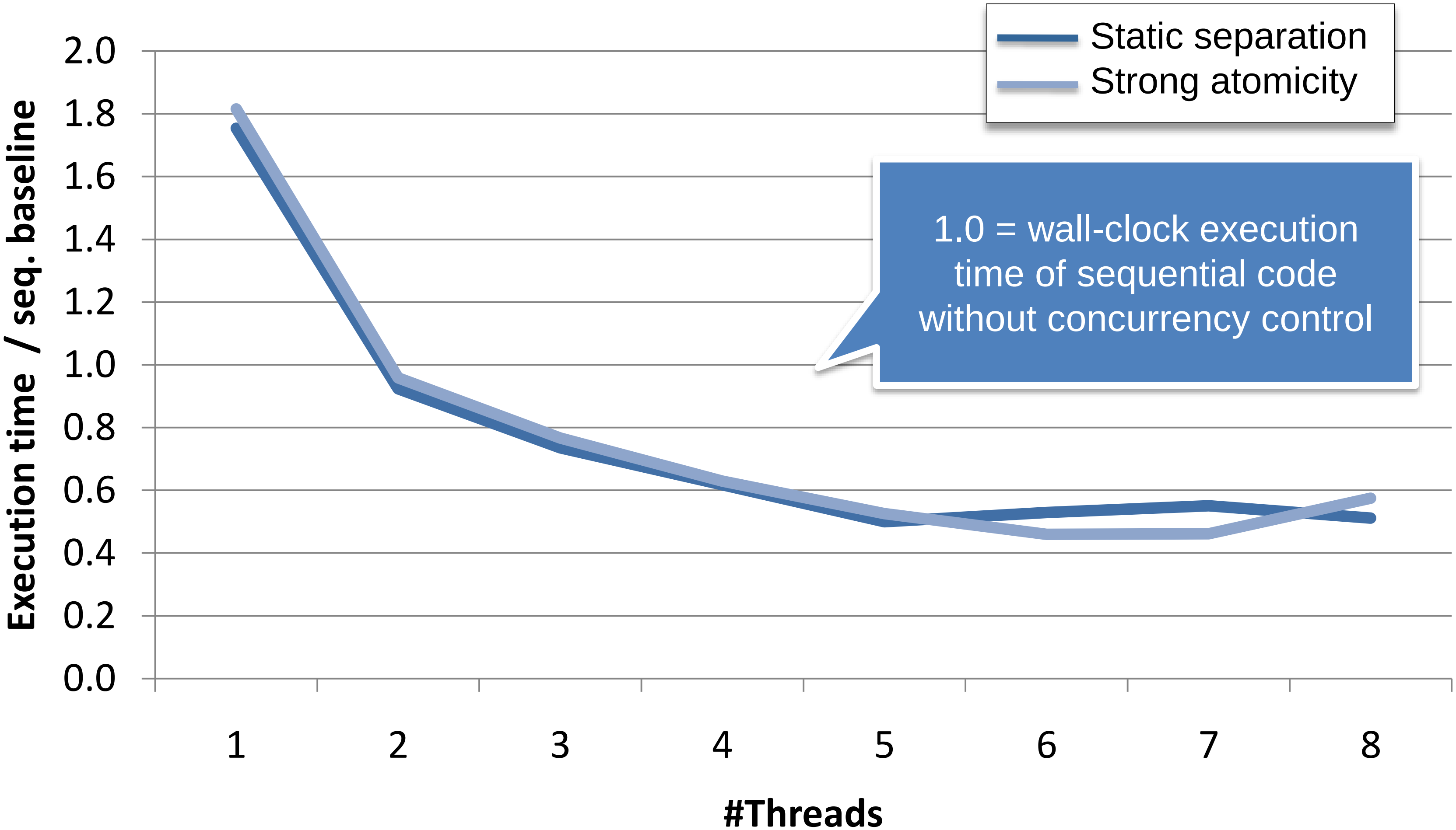
Sequential overhead



Scaling – 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)