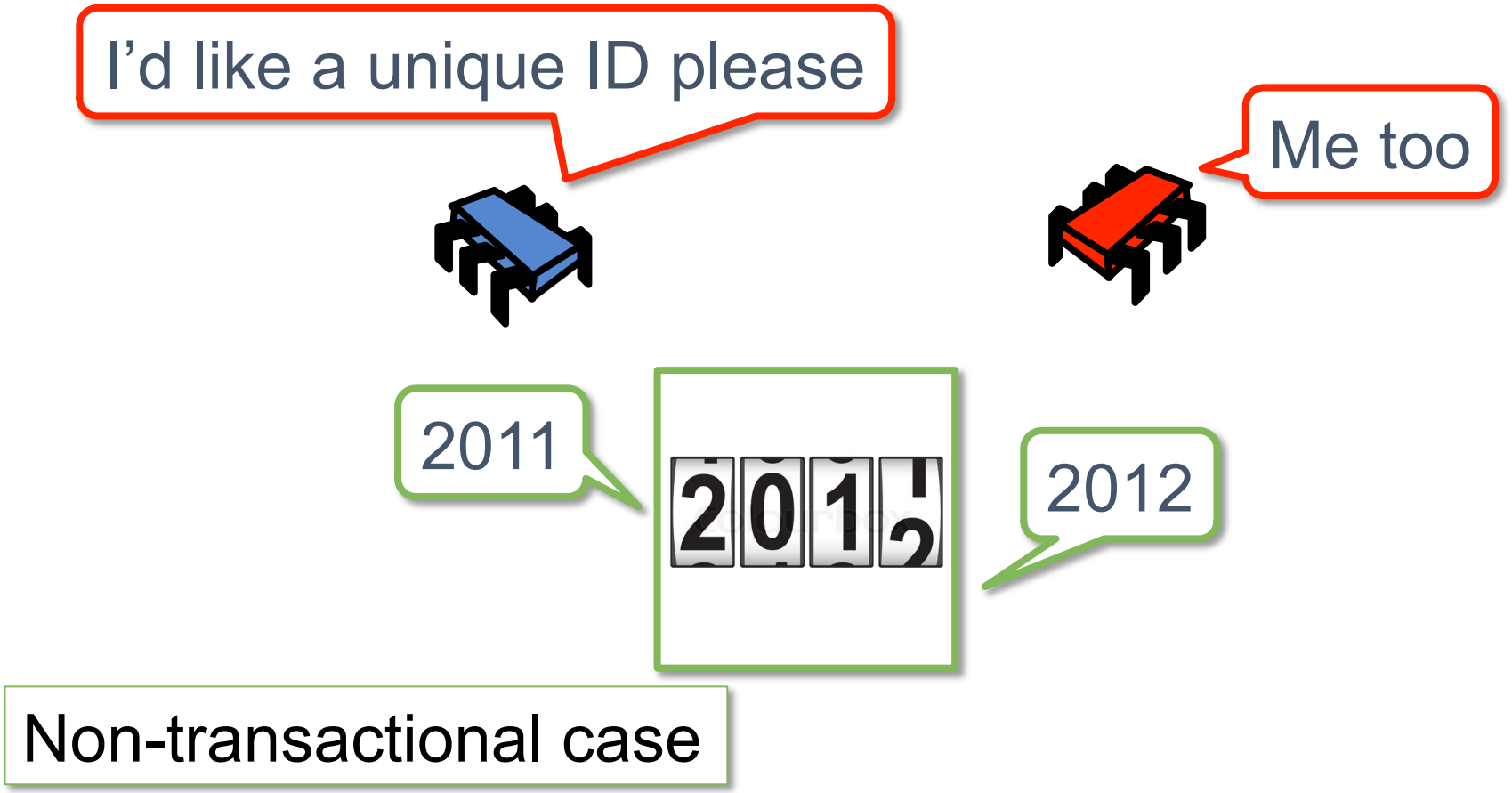
Implementation techniques for libraries of transactional concurrent data types

Liuba Shrira

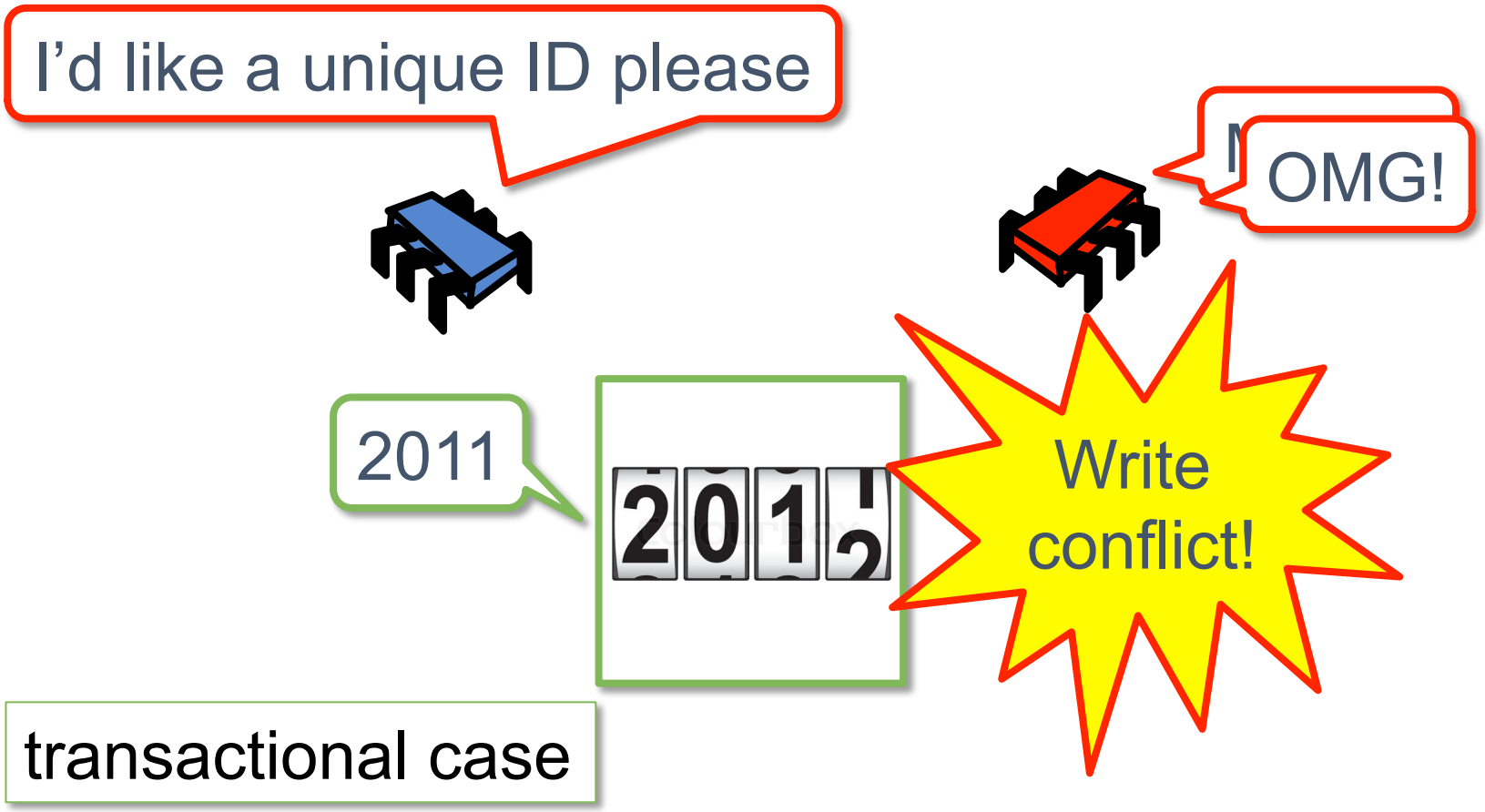
Brandeis University

Or: Type-Specific Concurrency Control and STM

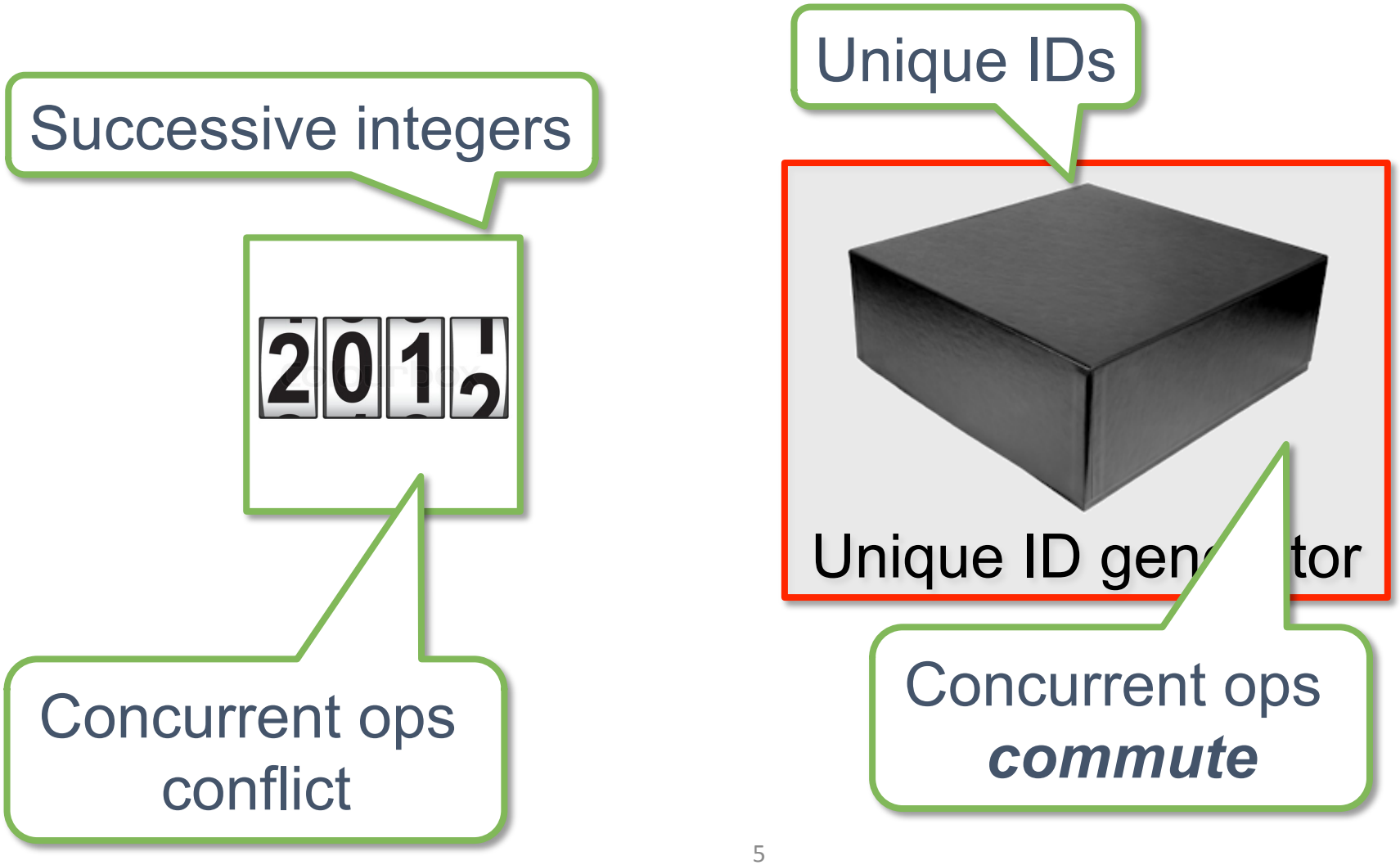
Where Modern STMs Fail



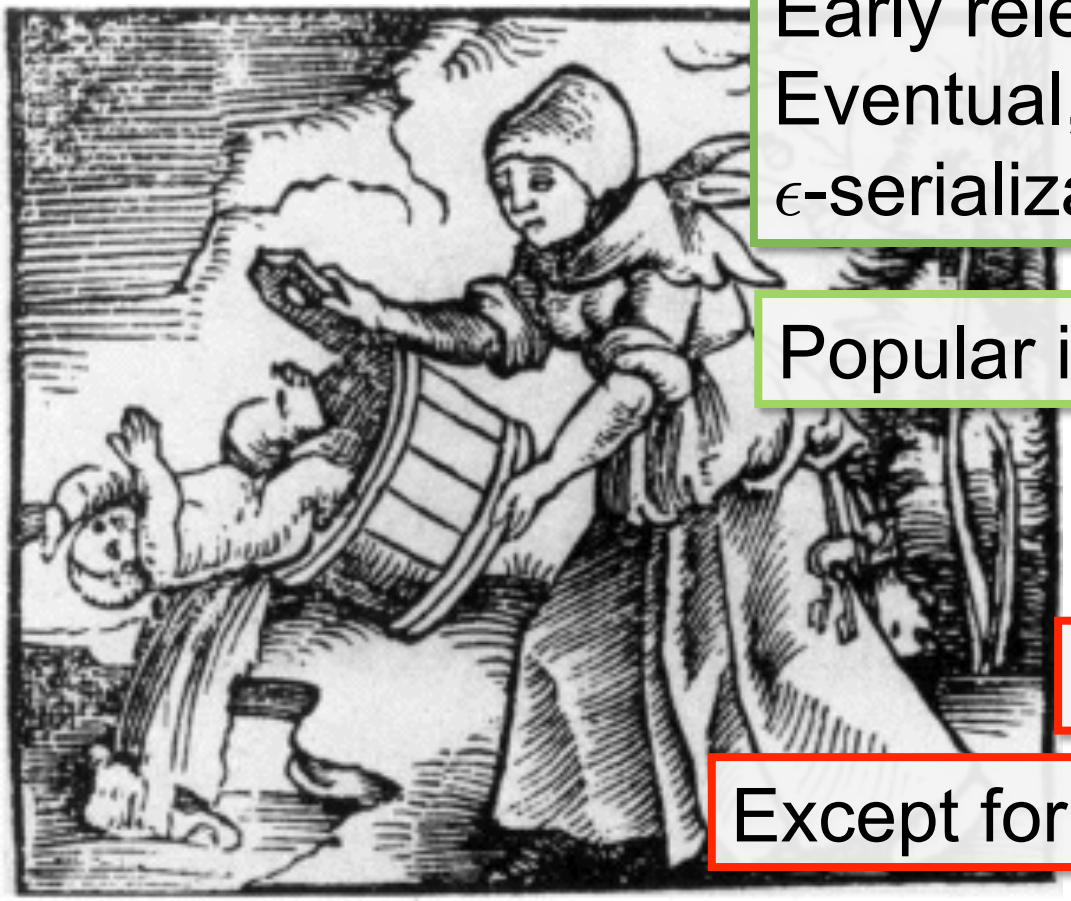
Where Modern STMs Fail



It’s not the STMs problem really



Relaxed Atomicity



Early release, open-nested, Eventual, elastic, ϵ-serializability, etc.

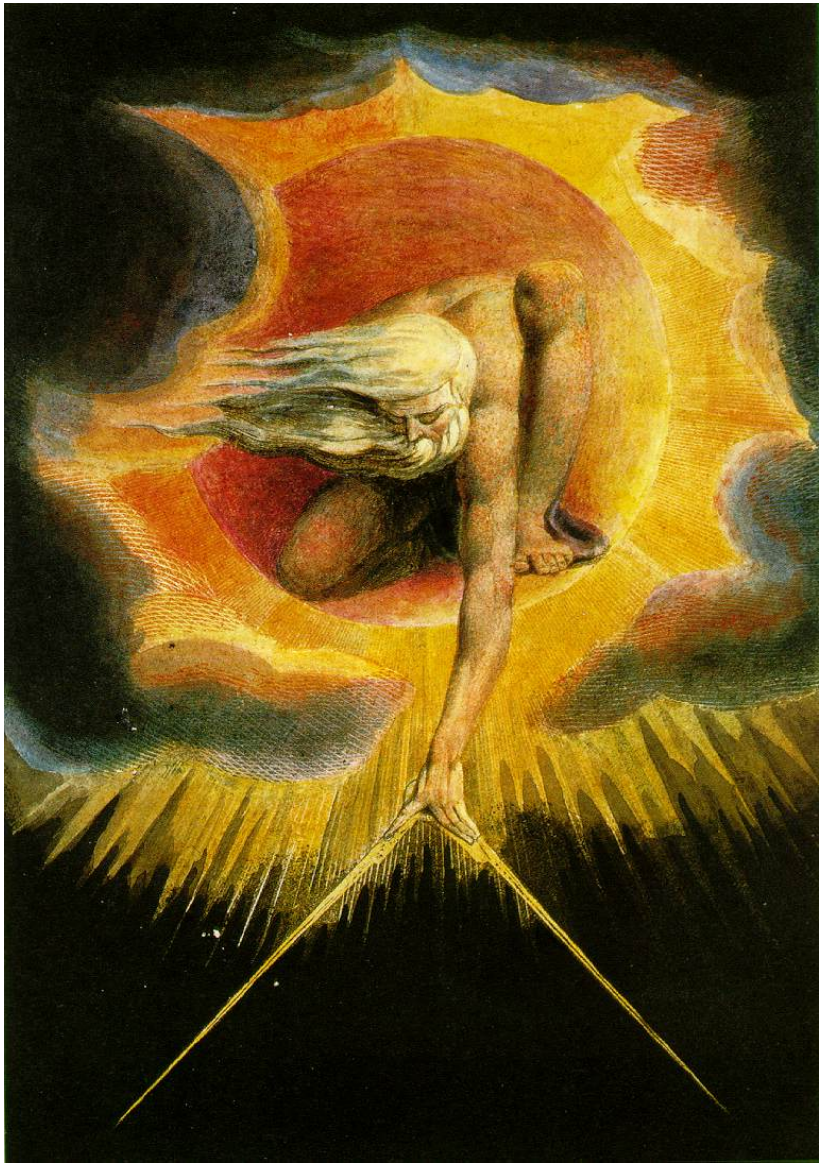
Popular in 80s & 90s …

DB and distributed

Mostly forgotten …

Except for snapshot isolation.

Type-Specific Concurrency Control



Also from 80s …

Exploit

- Commutativity …
- Non-determinism

For example

- Escrow …
- Exo-leasing …

TM raises different questions

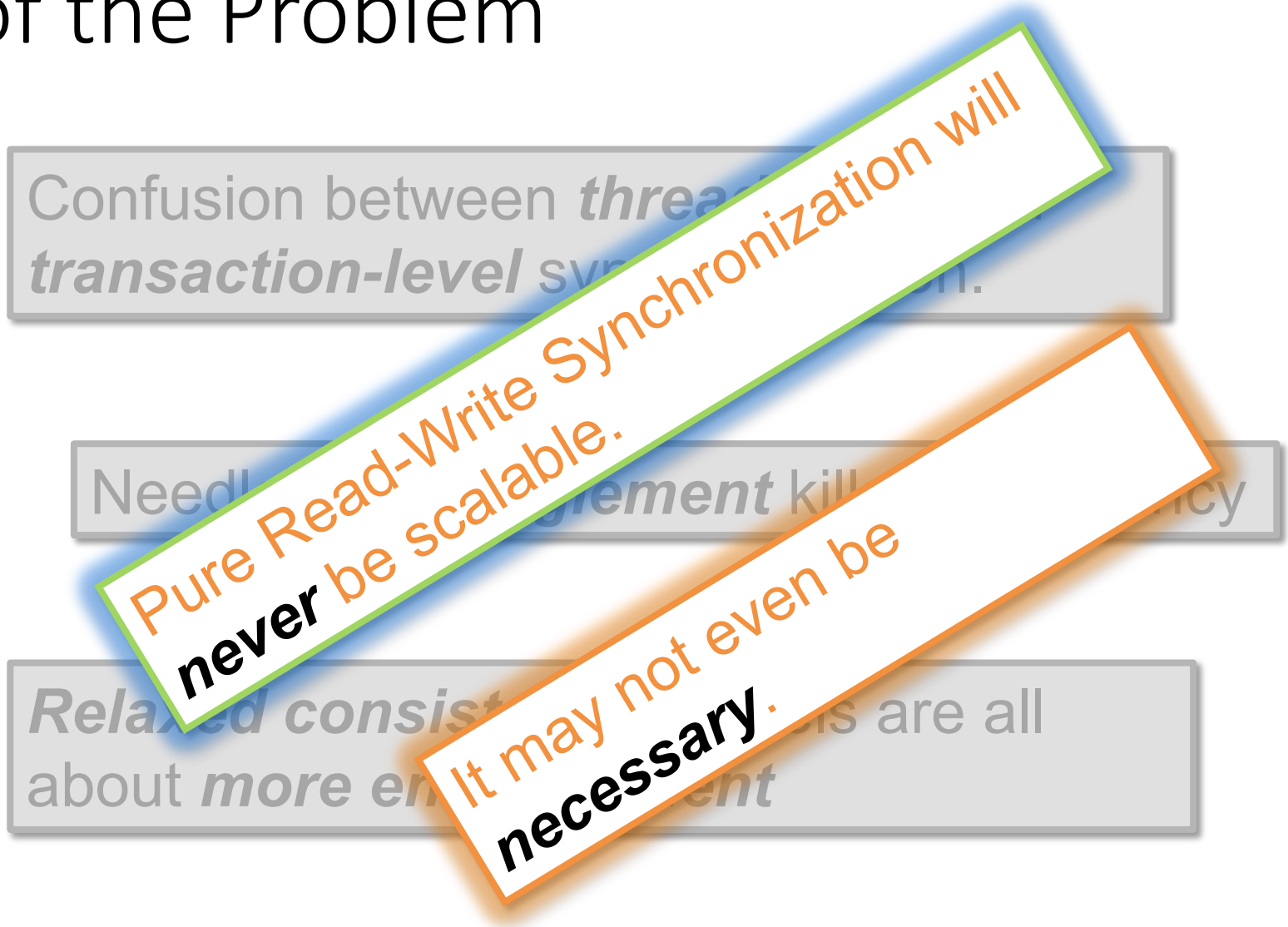
Heart of the Problem

Confusion between ***thread-level*** and ***transaction-level*** synchronization.

Needless ***entanglement*** kills concurrency

Relaxed consistency models are all about ***more entanglement***

Heart of the Problem



50 Shades of Synchronization

2

Short-lived, fine-grained

Atomic instruction (CAS)

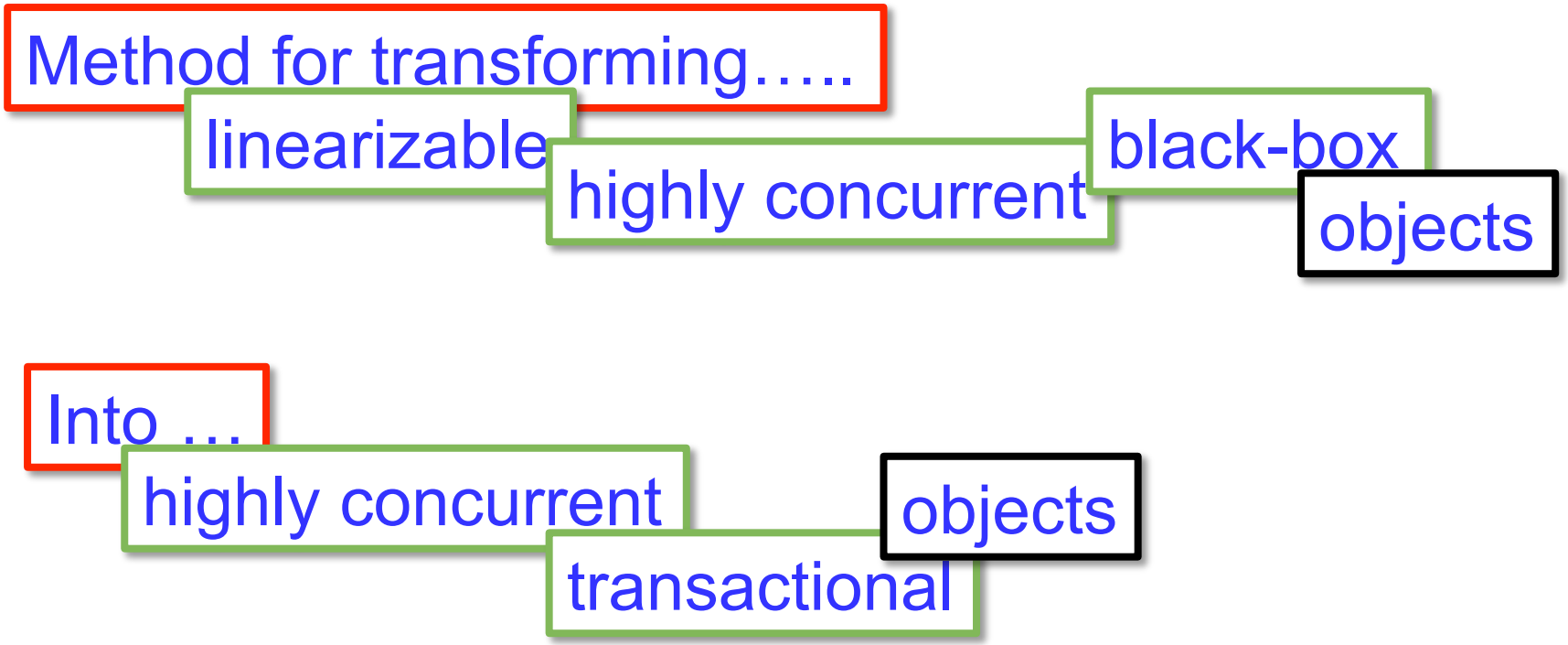
Hardware Transaction

Critical Sections

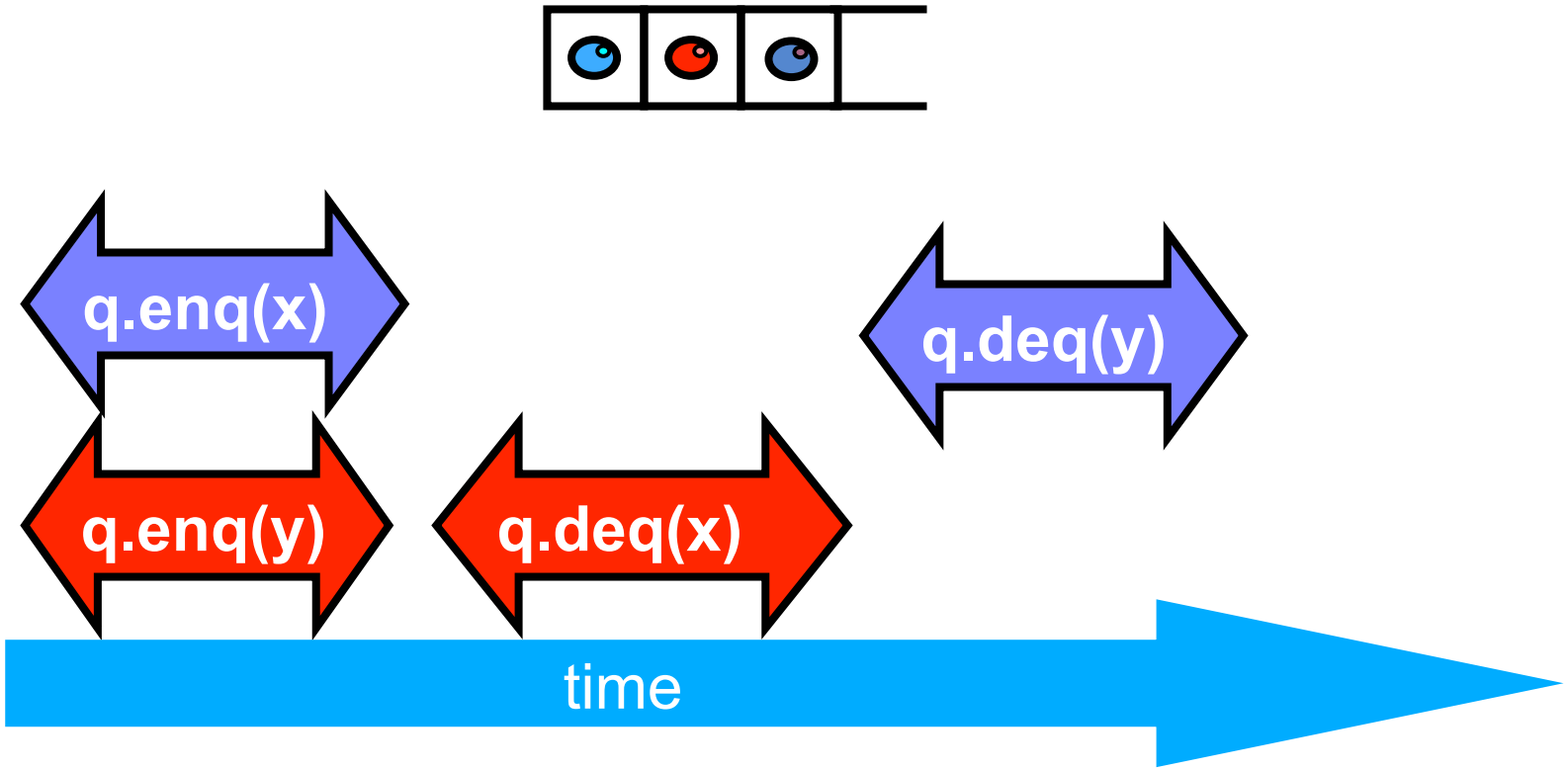
Long-lived, coarse-grained

Software transaction

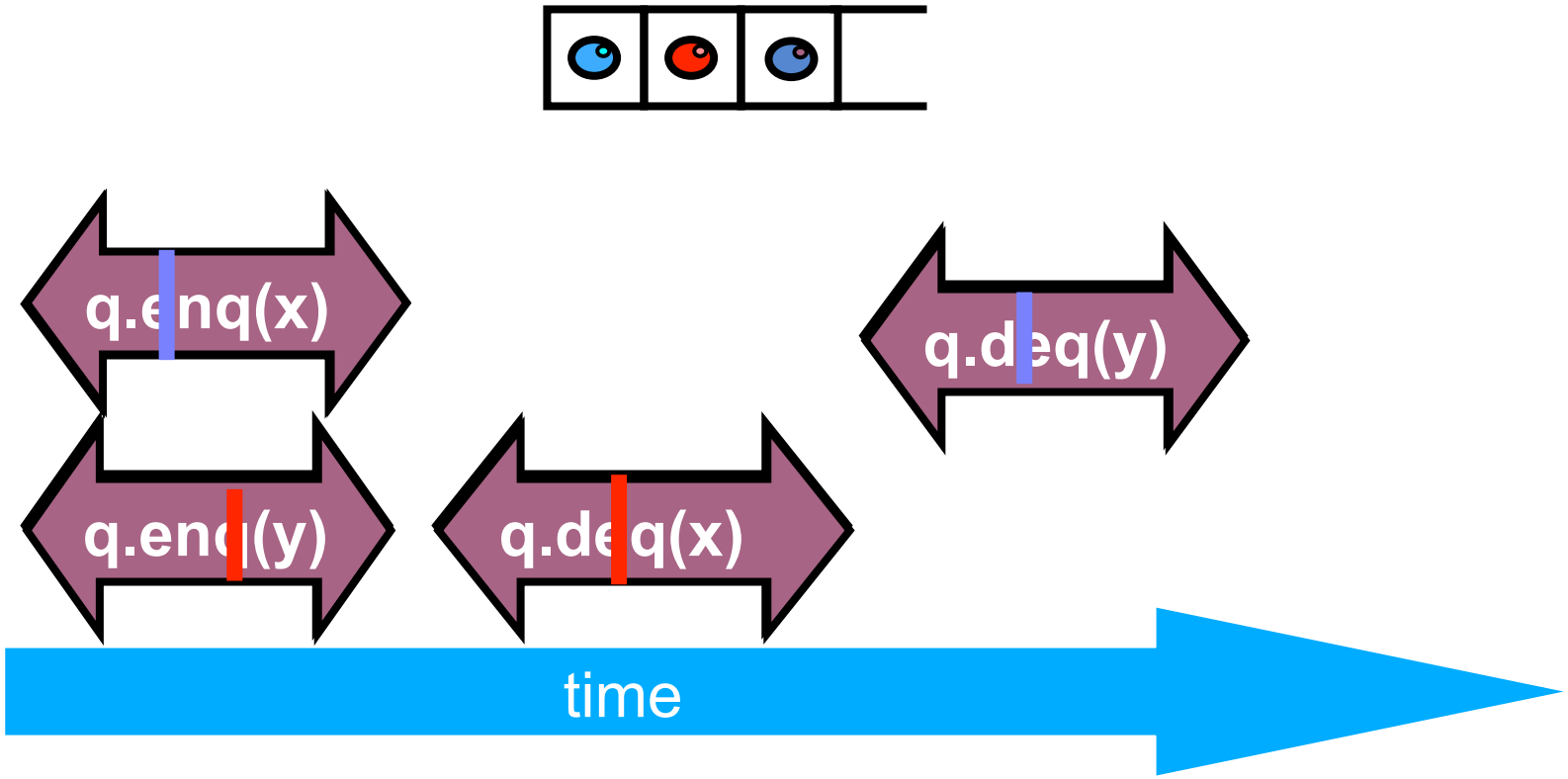
Transactional Boosting



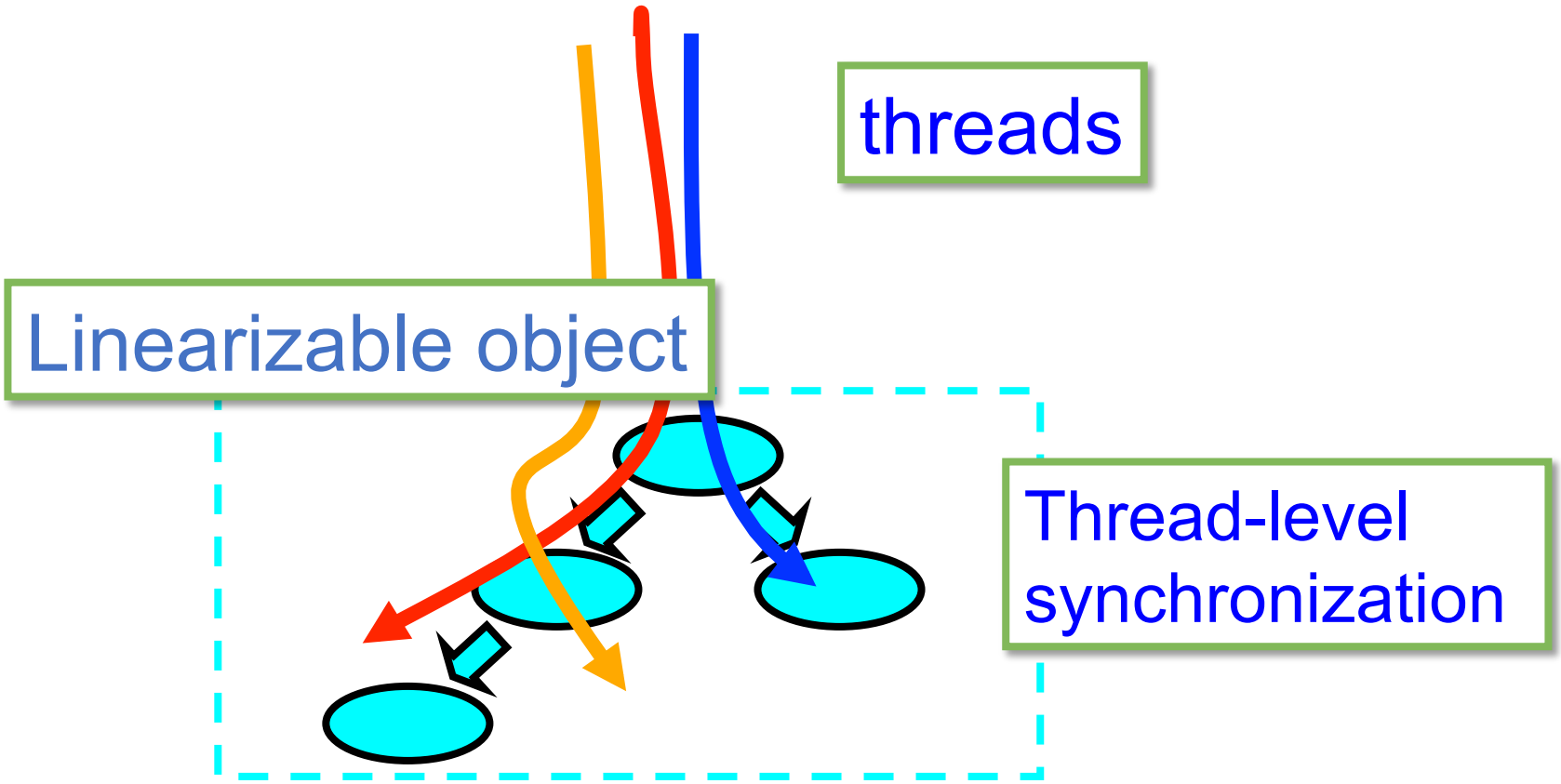
Concurrent Objects



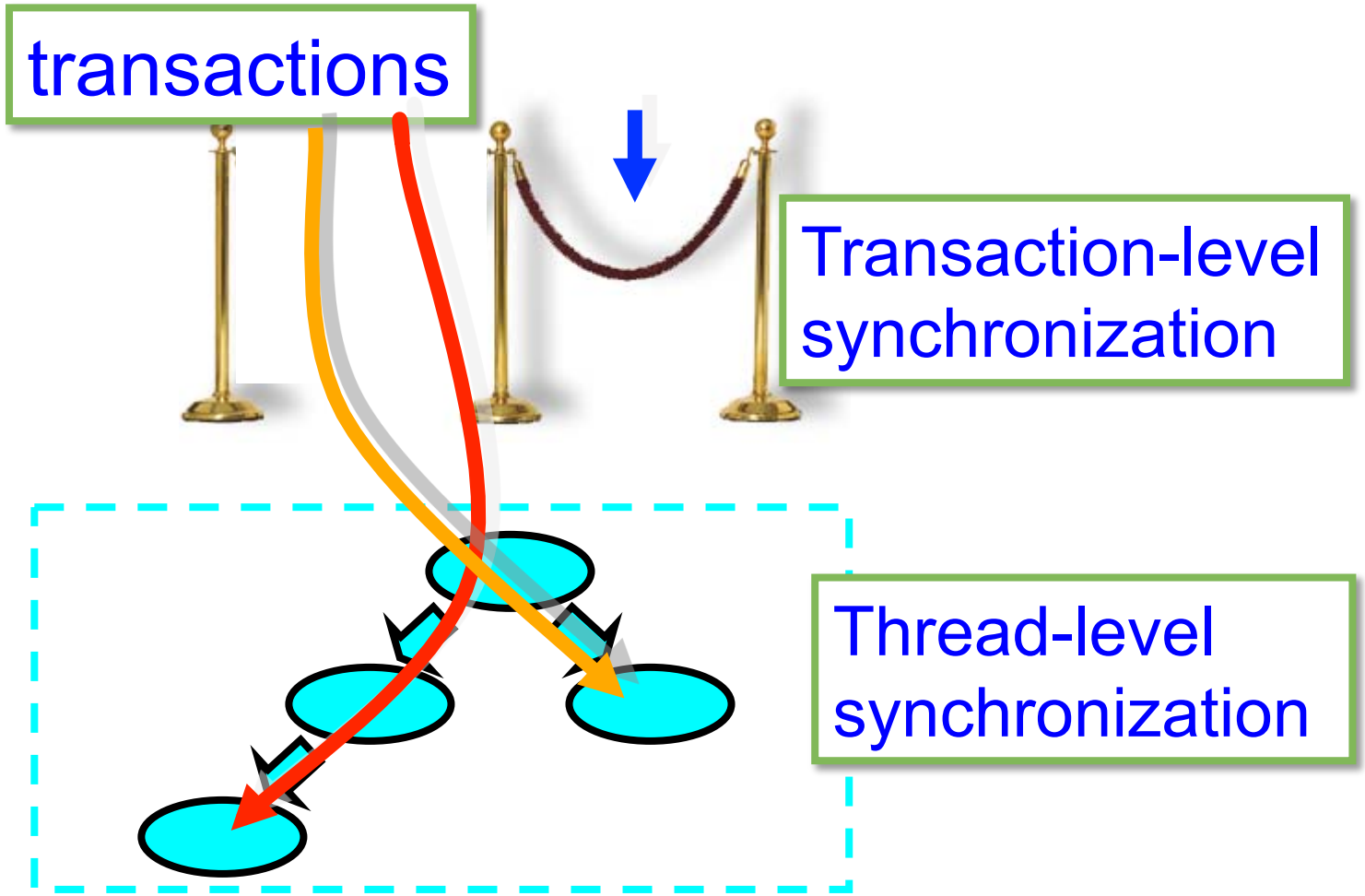
Linearizability



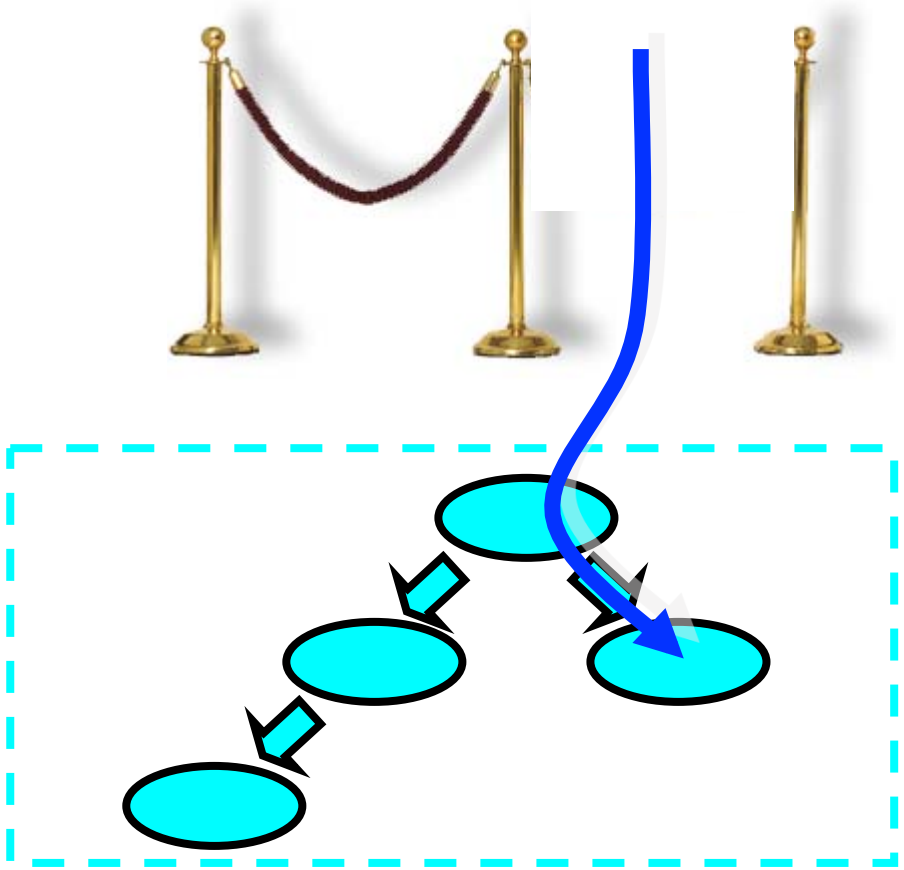
Linearizable Objects



Transactional Boosting



Disentangled Run-Time

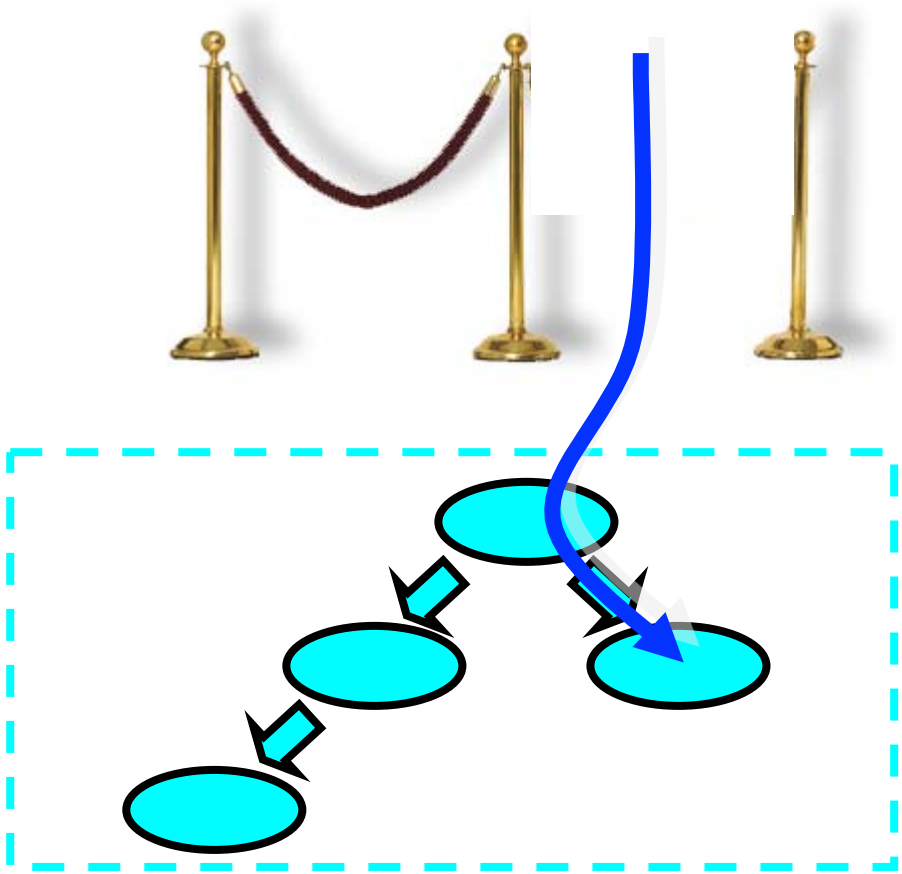


Library:
abstract locks,
Inverse logs

Your favorite
fine-grained
algorithms

HW transa

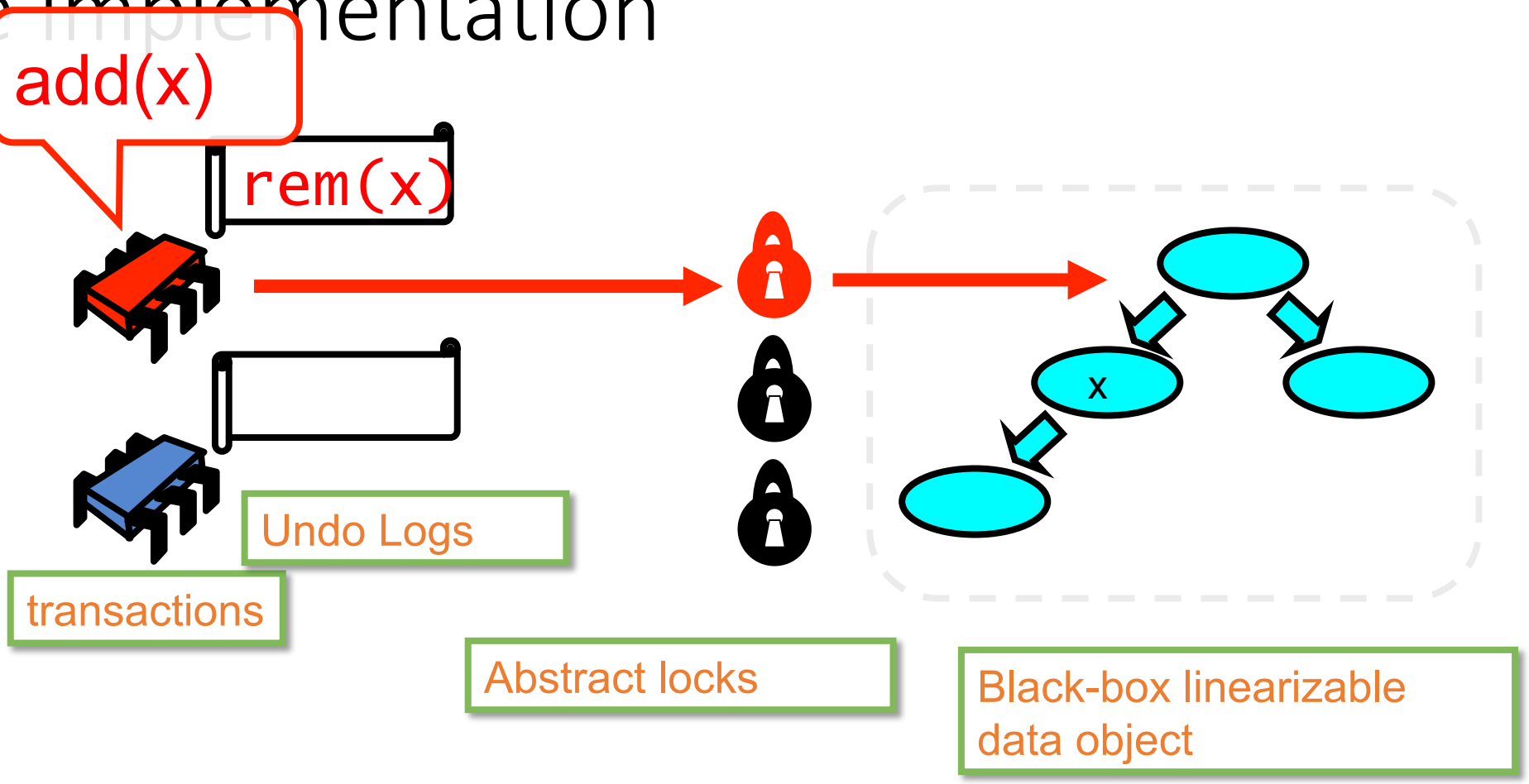
Disentangled Reasoning



Commutativity & inverses

Linearizability
e.g., rely-guarantee …

One Implementation



Lets look at some code

- Example 1: Transactional Set
    - implemented by boosting ConcurrentSkipList object, using LockKey for synchronization

Set Specification

Method	Inverse
add$(x)/$*false*	noop()
add$(x)/$*true*	remove$(x)/$*true*
remove$(x)/$*false*	noop()
remove$(x)/$*true*	add$(x)/$*true*
contains$(x)/_$	noop()

Commutativity

insert $(x)/\_ \Leftrightarrow$ insert $(y)/\_, x \neq y$

remove$(x)/_ \Leftrightarrow$ remove$(y)/_, x \neq y$

insert $(x)/\_ \Leftrightarrow$ remove$(y)/_, x \neq y$

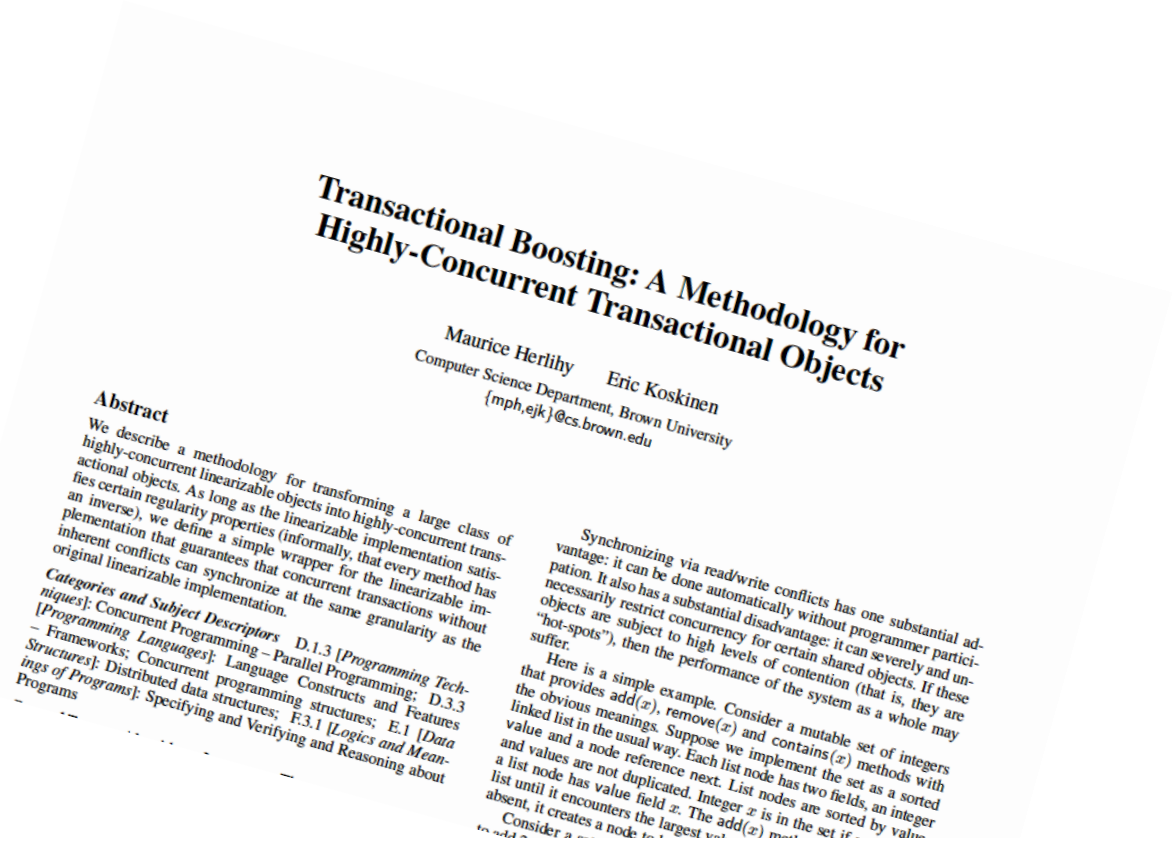
add$(x)/$*false* $\Leftrightarrow$ remove$(x)/$*false* $\Leftrightarrow$ contains$(x)/_$

```
1  public class SkipListKey {
2    ConcurrentSkipListSet<Integer> list;
3    LockKey lock;
4     ...
5    public boolean add(final int v) {
6      lock.lock(v);
7      boolean result = list.add(v);
8      if (result) {
9        Thread.onAbort(new Runnable() {
10         public void run() { list.remove(v);}}
11         );
12     }
13     return result;
14   }
15    ...
16 }
```

```
17 public class LockKey {
18   ConcurrentHashMap<Integer,Lock> map;
19   public LockKey() {
20     map = new ConcurrentHashMap<Integer,Lock>();
21   }
22   public void lock(int key) {
23     Lock lock = map.get(key);
24     if (lock == null) {
25       Lock newLock = new ReentrantLock();
26       Lock oldLock = map.putIfAbsent(key, newLock);
27       lock = (oldLock == null) ? newLock : oldLock;
28     }
29     if (LockSet.add(lock)) {
30       if (!lock.tryLock(LOCK_TIMEOUT,
31        TimeUnit.MILLISECONDS);) {
32         lockSet.remove(lock);
33         Thread.getTransaction().abort();
34         throw new AbortedException();
35       }
36     }
37   }
38    ...
39 }
```

More examples:

- Transactional Priority Queue, Pipelining, UniqueID …
  - implemented by boosting concurrent objects from Java concurrency packages



Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects

Maurice Herlihy Eric Koskinen
Computer Science Department, Brown University
{mph,ejk}@cs.brown.edu

Abstract

We describe a methodology for transforming a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. As long as the linearizable implementation satisfies certain regularity properties (informally, that every method has an inverse), we define a simple wrapper for the linearizable implementation that guarantees that concurrent transactions without inherent conflicts can synchronize at the same granularity as the original linearizable implementation.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features – Frameworks; Concurrent programming structures; E.1 [*Data Structures*]: Distributed data structures; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Synchronizing via read/write conflicts has one substantial advantage: it can be done automatically without programmer participation. It also has a substantial disadvantage: it can severely and unnecessarily restrict concurrency for certain shared objects. If these objects are subject to high levels of contention (that is, they are "hot-spots"), then the performance of the system as a whole may suffer.

Here is a simple example. Consider a mutable set of integers that provides add(x), remove(x) and contains(x) methods with the obvious meanings. Suppose we implement the set as a sorted linked list in the usual way. Each list node has two fields, an integer value and a node reference next. List nodes are sorted by value and values are not duplicated. Integer x is in the set if a list node has value field x. The add(x)

Performance of boosting

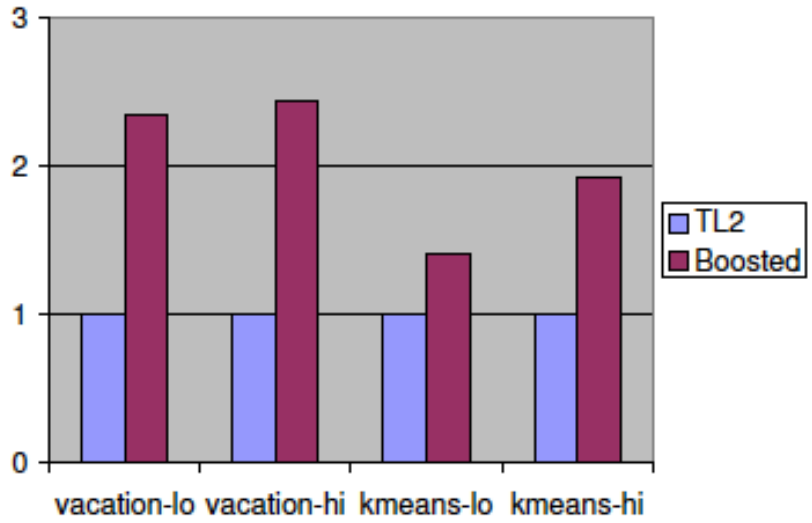


Figure 8. Throughput for boosted STAMP benchmarks normalized against the throughput of conventional TL2, which maintains shadow copies.

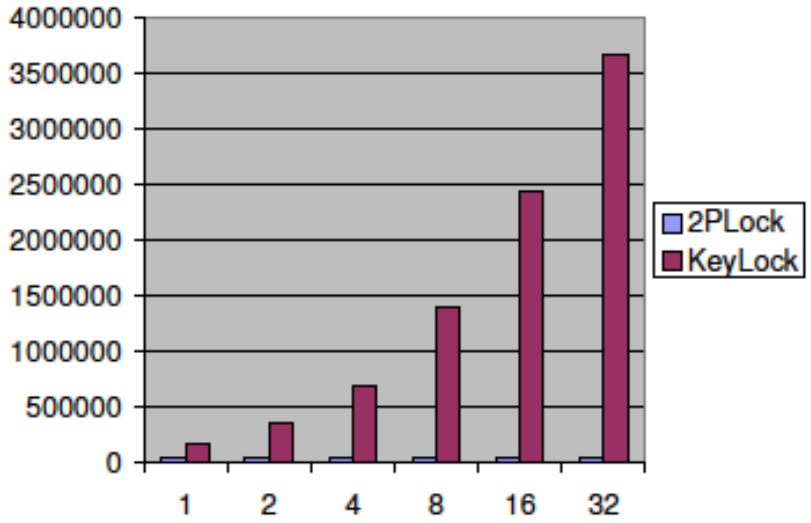


Figure 9. Throughput for transactionally-boosted skip lists using simple (left) and key-based (right) two-phase locks.

What's the Catch?

Concurrent calls must ***commute***

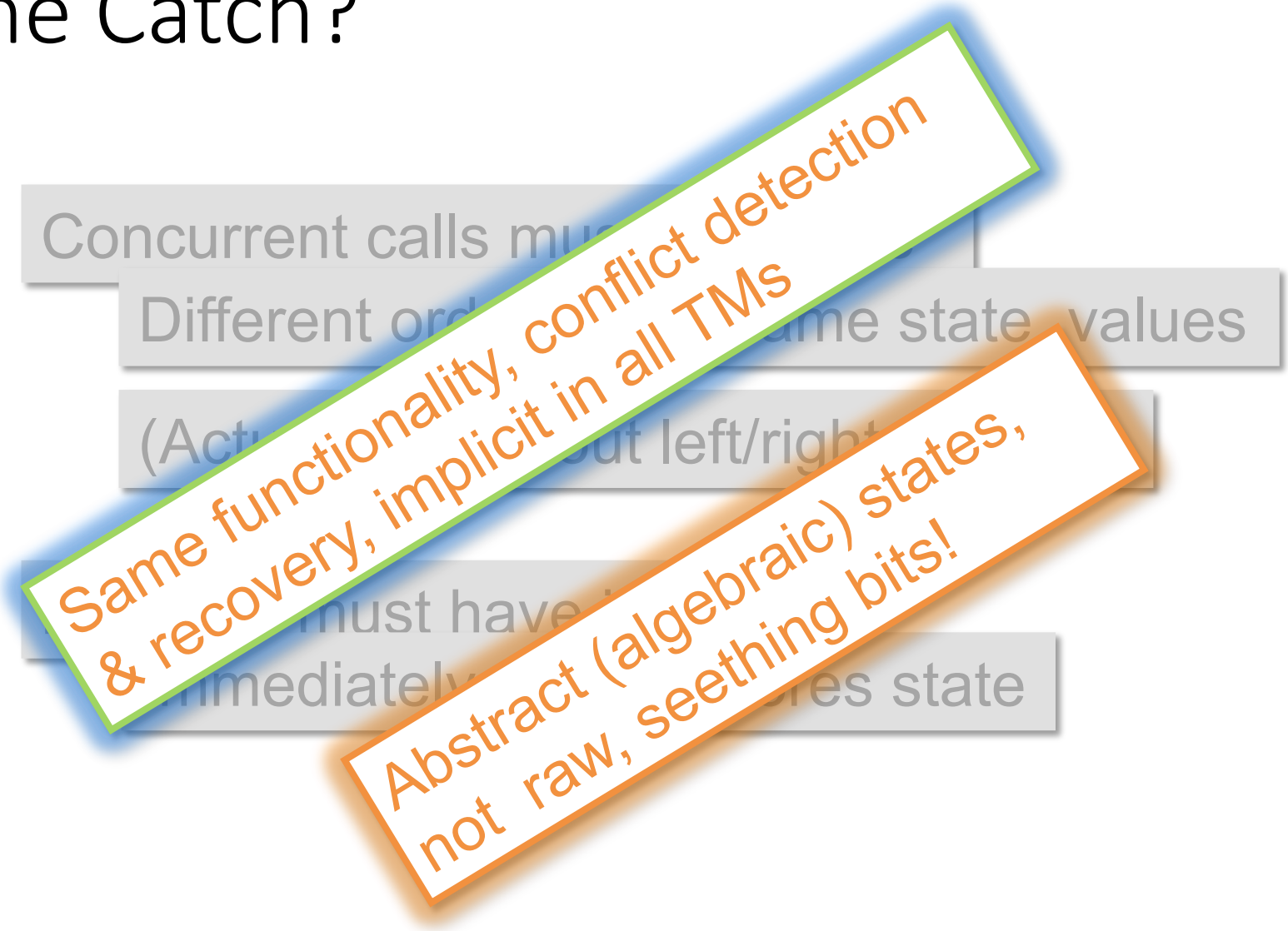
Different orders yield same state, values

(Actually, all about ***left/right movers***)

Methods must have ***inverses***

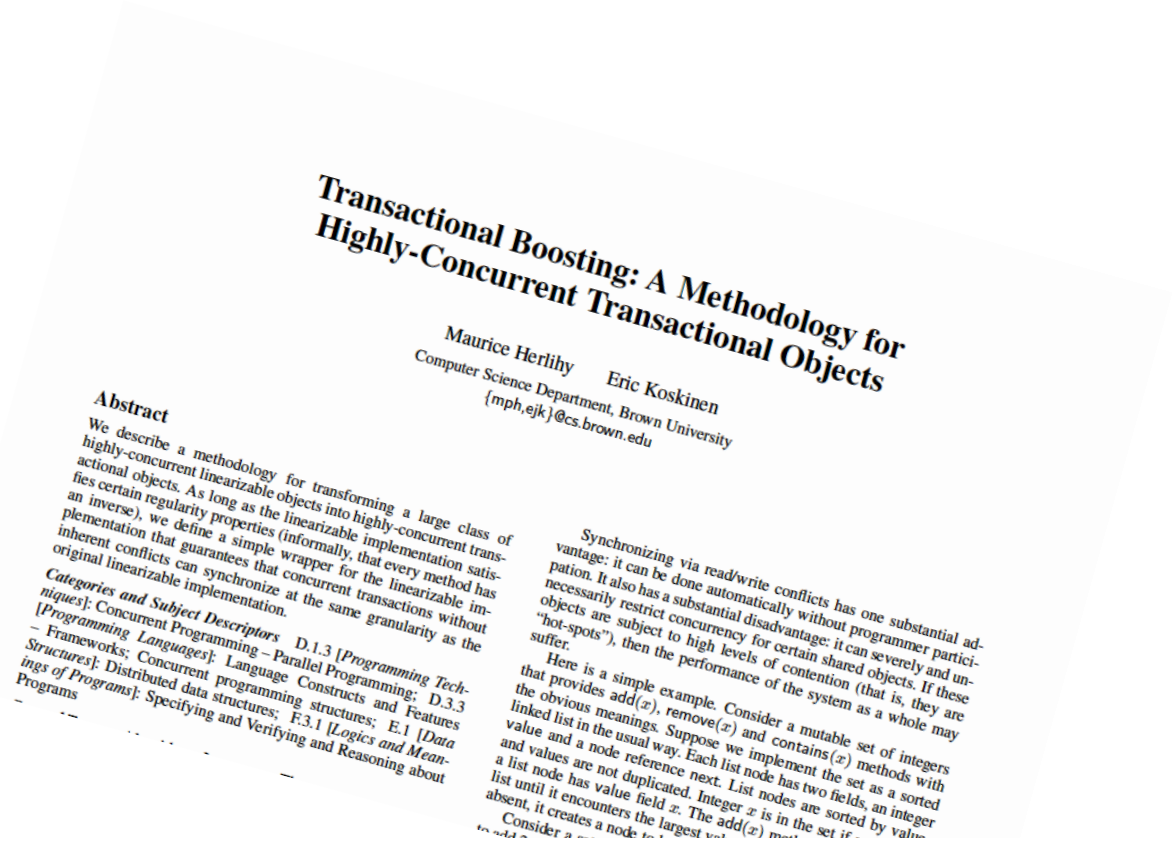
Immediately after, restores state

What's the Catch?



Boosting

- Reuse code, improve performance
- But inverses



Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects

Maurice Herlihy Eric Koskinen
Computer Science Department, Brown University
{mph,ejk}@cs.brown.edu

Abstract

We describe a methodology for transforming a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. As long as the linearizable implementation satisfies certain regularity properties (informally, that every method has an inverse), we define a simple wrapper for the linearizable implementation that guarantees that concurrent transactions without inherent conflicts can synchronize at the same granularity as the original linearizable implementation.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features – Frameworks; Concurrent programming structures; E.1 [*Data Structures*]: Distributed data structures; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Synchronizing via read/write conflicts has one substantial advantage: it can be done automatically without programmer participation. It also has a substantial disadvantage: it can severely and unnecessarily restrict concurrency for certain shared objects. If these objects are subject to high levels of contention (that is, they are "hot-spots"), then the performance of the system as a whole may suffer.

Here is a simple example. Consider a mutable set of integers that provides add(x), remove(x) and contains(x) methods with the obvious meanings. Suppose we implement the set as a sorted linked list in the usual way. Each list node has two fields, an integer value and a node reference next. List nodes are sorted by value and values are not duplicated. Integer x is in the set if a list node has value field x. The add(x)

And is there ever enough performance?

How to improve performance?

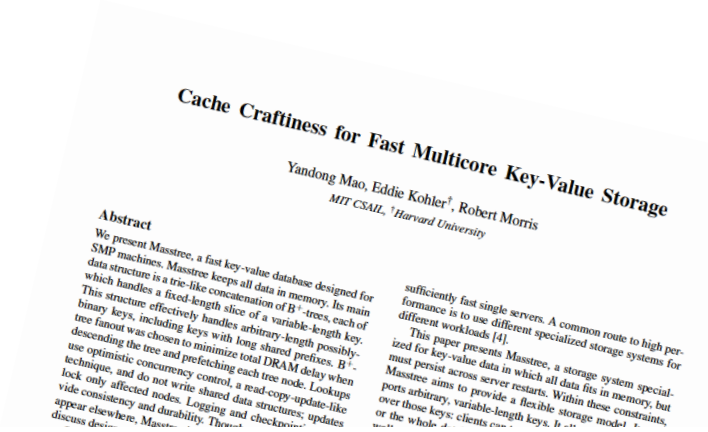
Recall, we want

- Good performance when synchronization is required
- Scalability

- E.g., for in-memory key-value store

Up next: how to improve the performance of a transactional data structure?

- MassTree: a high performance data structure
- Silo: high performance transactions over MassTree using a different approach
- STO: a general framework and methodology for building libraries of customized high performance transactional objects



Cache Craftiness for Fast Multicore Key-Value Storage

Yandong Mao, Eddie Kohler†, Robert Morris

MIT CSAIL, †Harvard University

Abstract

We present Masstree, a fast key-value database designed for SMP machines. Masstree keeps all data in memory. Its main data structure is a trie-like concatenation of B^+-trees, each of which handles a fixed-length slice of a variable-length key. This structure effectively handles arbitrary-length possibly-binary keys, including keys with long shared prefixes. B^+-tree fanout was chosen to minimize total DRAM delay when descending the tree and prefetching each tree node. Lookups use optimistic concurrency control, a read-copy-update-like technique, and do not write shared data structures; updates lock only affected nodes. Logging and checkpointing provide consistency and durability. Thorough

sufficiently fast single servers. A common route to high performance is to use different specialized storage systems for different workloads [4].

This paper presents Masstree, a storage system specialized for key-value data in which all data fits in memory, but must persist across server restarts. Within these constraints, Masstree aims to provide a flexible storage model—ports arbitrary, variable-length keys, lookups over those keys; clients can

Speedy Transactions in Multicore In-Memory Databases

Stephen Tu, Wenting Zheng, Eddie Kohler†, Barbara Liskov, and Samuel Madden

MIT CSAIL and †Harvard University

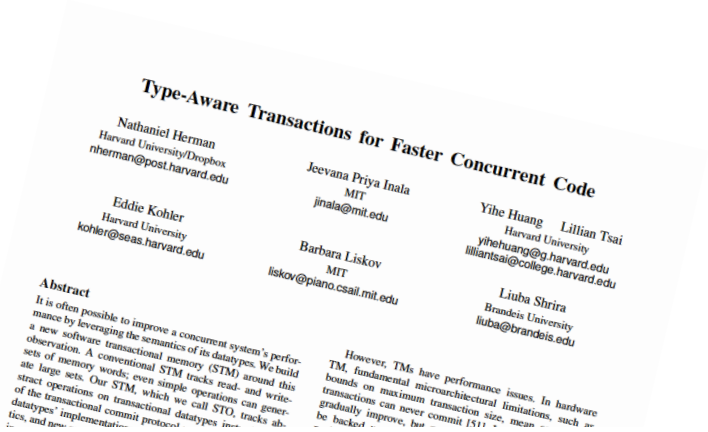
Abstract

Silo is a new in-memory database that achieves excellent performance and scalability on modern multicore machines. Silo was designed from the ground up to use system memory and caches efficiently. For instance, it avoids all centralized contention points, including that of centralized transaction ID assignment. Silo's key contribution is a commit protocol based on optimistic concurrency control that provides serializability while avoiding all shared-memory writes for records that were only read. Though this might seem to complicate the enforcement of a serial order, correct logging and recovery is provided by linking periodically-updated *epochs* with the commit protocol. Silo provides the same guarantees as any serializable database without unnecessary scalability bottlenecks or much additional latency. Silo achieves almost 700,000 transactions per second on a standard TPC-C

nization scale with the data, allowing larger databases to support more concurrency.

Silo uses a Masstree-inspired tree structure for its underlying indexes. Masstree [23] is a fast concurrent B-tree-like structure optimized for multicore performance. But Masstree only supports non-serializable, single-key transactions, whereas any real database must support transactions that affect multiple keys and occur in some serial order. Our core result, the Silo commit protocol, is a minimal-contention serializable commit protocol that provides these properties.

Silo uses a variant of optimistic concurrency control (OCC) [18]. A



Type-Aware Transactions for Faster Concurrent Code

Nathaniel Herman
Harvard University/Dropbox
nherman@post.harvard.edu

Jeevana Priya Inala
MIT
jinala@mit.edu

Yihe Huang
Harvard University
yihehuang@g.harvard.edu

Lillian Tsai
Harvard University
lilliantsai@college.harvard.edu

Eddie Kohler
Harvard University
kohler@seas.harvard.edu

Barbara Liskov
MIT
liskov@piano.csail.mit.edu

Liuba Shrira
Brandeis University
liuba@brandeis.edu

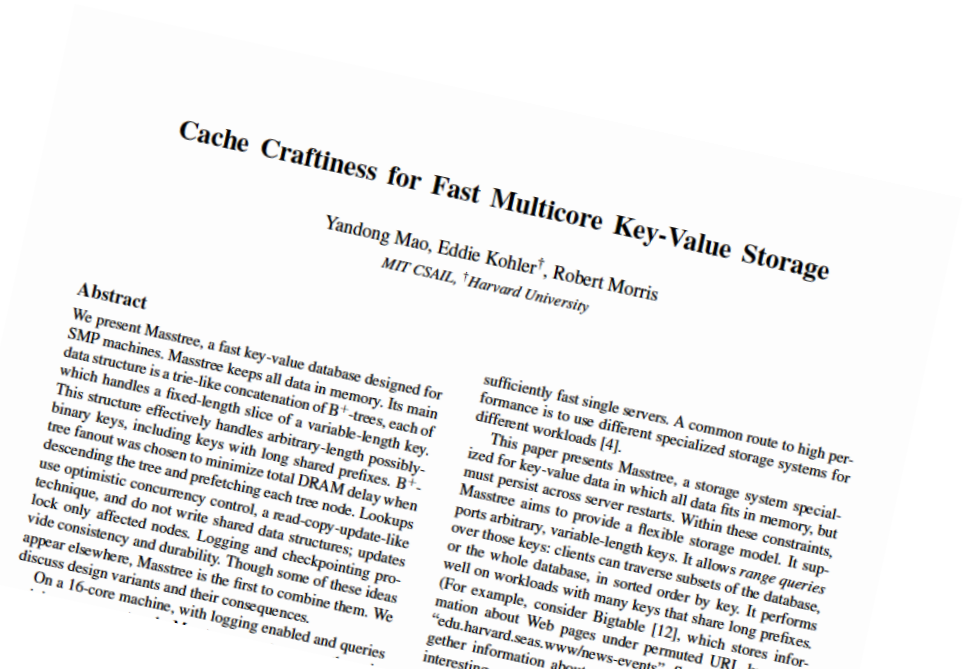
Abstract

It is often possible to improve a concurrent system's performance by leveraging the semantics of its datatypes. We build a new software transactional memory (STM) around this observation. A conventional STM tracks read- and write-sets of memory words; even simple operations can generate large sets. Our STM, which we call STO, tracks abstract operations on transactional datatypes instead. Integration of the transactional commit protocol with datatypes' implementations, and new

However, TMs have performance issues. In hardware TM, fundamental microarchitectural limitations, such as bounds on maximum transaction size, mean transactions can never commit [51]. In software TM, gradually improve, but be backed

MassTree

- High-performance key/value store
  - In shared primary memory
  - Cores run put, get, and delete requests



Cache Craftiness for Fast Multicore Key-Value Storage

Yandong Mao, Eddie Kohler†, Robert Morris
MIT CSAIL, †Harvard University

Abstract

We present Masstree, a fast key-value database designed for SMP machines. Masstree keeps all data in memory. Its main data structure is a trie-like concatenation of B^+-trees, each of which handles a fixed-length slice of a variable-length key. This structure effectively handles arbitrary-length possibly-binary keys, including keys with long shared prefixes. B^+-tree fanout was chosen to minimize total DRAM delay when descending the tree and prefetching each tree node. Lookups use optimistic concurrency control, a read-copy-update-like technique, and do not write shared data structures; updates lock only affected nodes. Logging and checkpointing provide consistency and durability. Though some of these ideas appear elsewhere, Masstree is the first to combine them. We discuss design variants and their consequences.

On a 16-core machine, with logging enabled and queries sufficiently fast single servers. A common route to high performance is to use different specialized storage systems for different workloads [4].

This paper presents Masstree, a storage system specialized for key-value data in which all data fits in memory, but must persist across server restarts. Within these constraints, Masstree aims to provide a flexible storage model. It supports arbitrary, variable-length keys. It allows *range queries* over those keys: clients can traverse subsets of the database, or the whole database, in sorted order by key. It performs well on workloads with many keys that share long prefixes. (For example, consider Bigtable [12], which stores information about Web pages under permuted URL

Review: Memory Model

- Each core has a cache
- Hitting in the cache matters a lot for reads!
- What about a write?
  - TSO (Total Store Order)

X86-TSO

- Thread t1 modifies x and later y
- Thread t2 sees modification to y
- t2 reads x

- Implies t2 sees modification of x

MassTree structure

- Nodes and records
- Nodes
  - Cover a range of keys
  - Interior and leaf nodes
- Records
  - Store the values

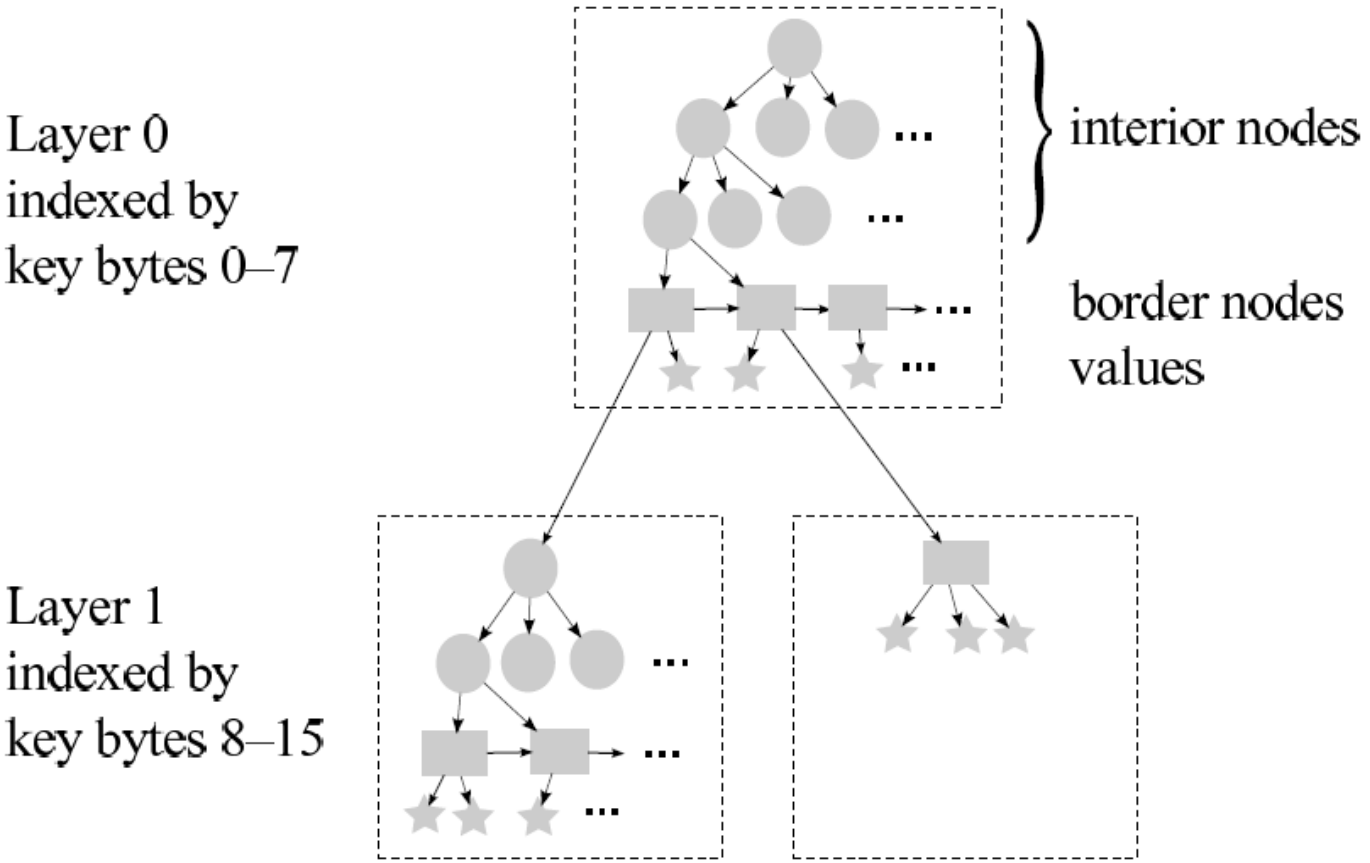


Figure 1. Masstree structure: layers of B^+-trees form a trie.

Concurrency Control

- Reader/writer locks?

Thread-level Concurrency Control

- Base instructions
  - Compare and swap
    - On one memory word
  - Fence

Concurrency Control for multi-word

- First word of nodes and records
  - version number (v#) and lock bit

Concurrency control

- Write
  - Set lock bit (spin if necessary)
    - uses compare and swap
  - Update node or record
  - Increment v# and release lock

Concurrency control

- Write (locking)
- Read (no locking)
  - Spin if locked
  - Read contents
  - If v# has changed or lock is set, try again

Concurrency control

- Writes are pessimistic
- Reads are optimistic

- A mix!
- No writes for reads

Inserting new keys

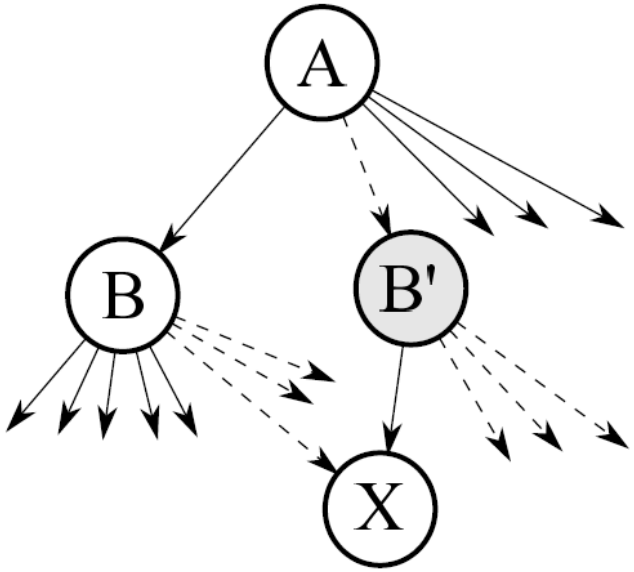
- Into leaf node if possible
- Else split

Inserting new keys

- Into leaf node if possible
- Else split

- Split locks nodes up the path
  - No deadlocks

Interesting Issue with splitting



From MassTree to Silo

- High-performance database
- With transactions

Speedy Transactions in Multicore In-Memory Databases

Stephen Tu, Wenting Zheng, Eddie Kohler†, Barbara Liskov, and Samuel Madden
MIT CSAIL and †Harvard University

Abstract

Silo is a new in-memory database that achieves excellent performance and scalability on modern multicore machines. Silo was designed from the ground up to use system memory and caches efficiently. For instance, it avoids all centralized contention points, including that of centralized transaction ID assignment. Silo's key contribution is a commit protocol based on optimistic concurrency control that provides serializability while avoiding *all* shared-memory writes for records that were only read. Though this might seem to complicate the enforcement of a serial order, correct logging and recovery is provided by linking periodically-updated *epochs* with the commit protocol. Silo provides the same guarantees as any serializable database without unnecessary scalability bottlenecks or much additional latency. Silo achieves almost 700,000 transactions per second on a standard TPC-C workload mix on a 32-core machine, as well as near-linear scalability. Considered per core, this is several times higher than previously reported

nization scale with the data, allowing larger databases to support more concurrency.

Silo uses a Masstree-inspired tree structure for its underlying indexes. Masstree [23] is a fast concurrent B-tree-like structure optimized for multicore performance. But Masstree only supports non-serializable, single-key transactions, whereas any real database must support transactions that affect multiple keys and occur in some serial order. Our core result, the Silo commit protocol, is a minimal-contention serializable commit protocol that provides these properties.

Silo uses a variant of optimistic concurrency control (OCC) [18]. An OCC transaction tracks the records it reads and writes in thread-local storage. At commit time, after validating that no concurrent transaction's
overlapped with its read set, the
written records at once. If
aborts. This

Silo

- Database is in primary memory
- Runs one-shot requests

Silo

- Database is in primary memory
- Runs one-shot requests

- A tree for each table or index
- Worker threads run the requests
  - One thread per core
- Workers share memory

Transactions

```
begin {
 % do stuff: run queries
 % using insert, lookup, update, delete,
 % and range
}
```

Running Transactions

- MassTree operations release locks before returning
  - Hold locks longer?

Running Transactions

- OCC (Optimistic Concurrency Control)
  - Thread maintains read-set and write-set
    - Read-set contains version numbers
    - Write-set contains new state
- At end, attempts commit

Commit Protocol

- Phase 1: lock all objects in write-set
  - Bounded spinning

Commit Protocol

- Phase 1: lock all objects in write-set
- Phase 2: verify v#’s of read-set
  - Abort if locked or changed

Commit Protocol

- Phase 1: lock all objects in write-set
- Phase 2: verify v#'s of read-set
- Select Tid (>v# of r- and w-sets)
  - Without a write to shared state!

Commit Protocol

- Phase 1: lock all objects in write-set
- Phase 2: verify v#'s of read-set
- Select Tid (>v# of r- and w-sets)
- Phase 3: update objects in write-set
  - Using Tid as v#

Commit Protocol

- Phase 1: lock all objects in write-set
- Phase 2: verify v#'s of read-set
- Select Tid (>v# of r- and w-sets)
- Phase 3: update objects in write-set
- Release locks

Additional Issues

- Range queries
- Absent keys
- Garbage collection

Performance

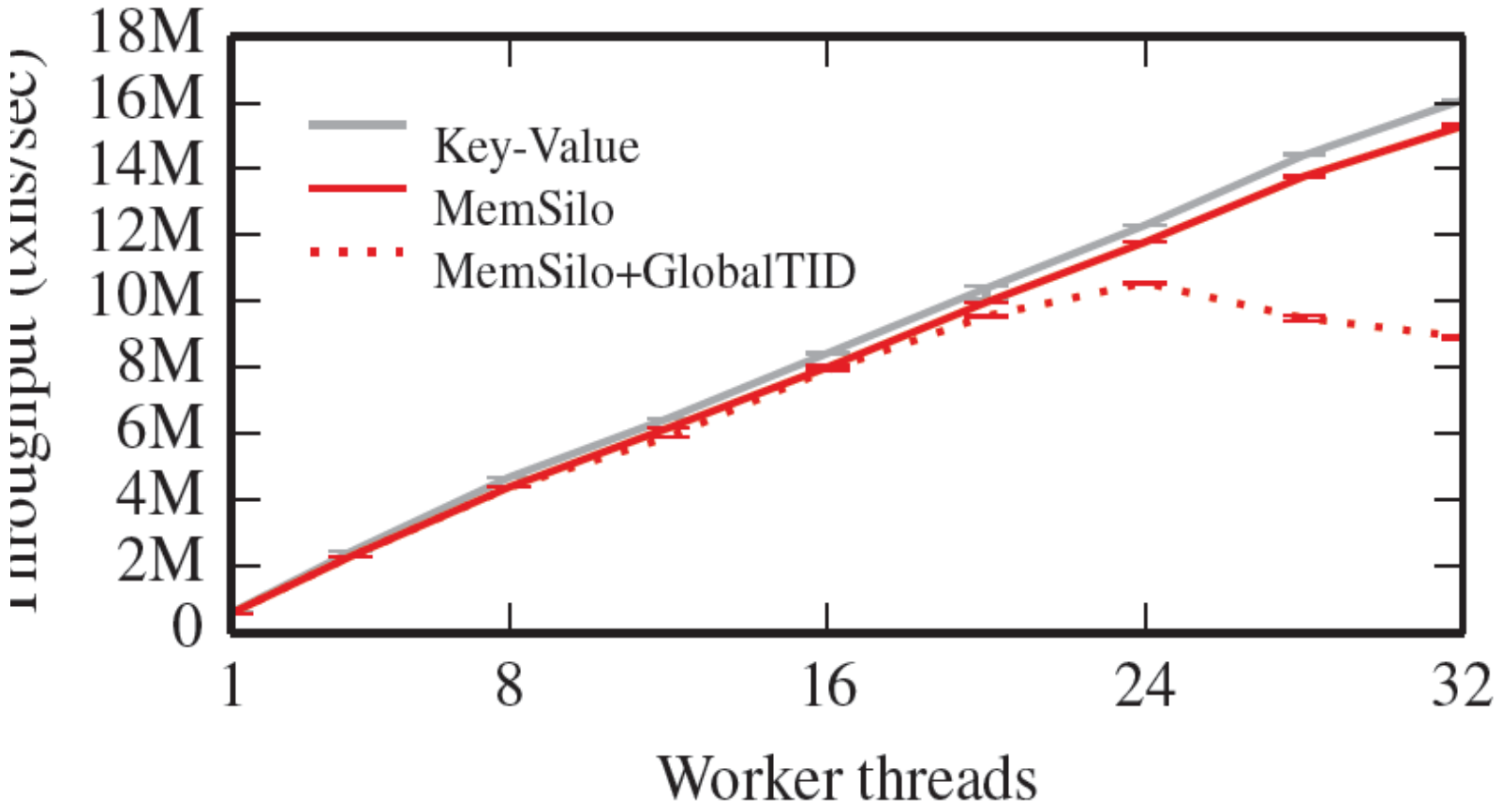


Figure 4: Overhead of *MemSilo* versus *Key-Value* when running a variant of the YCSB benchmark.

Performance

- vs. Hstore
  - M. Stonebraker et al, The end of an architectural era: (it's time for a complete rewrite), VLDB '07

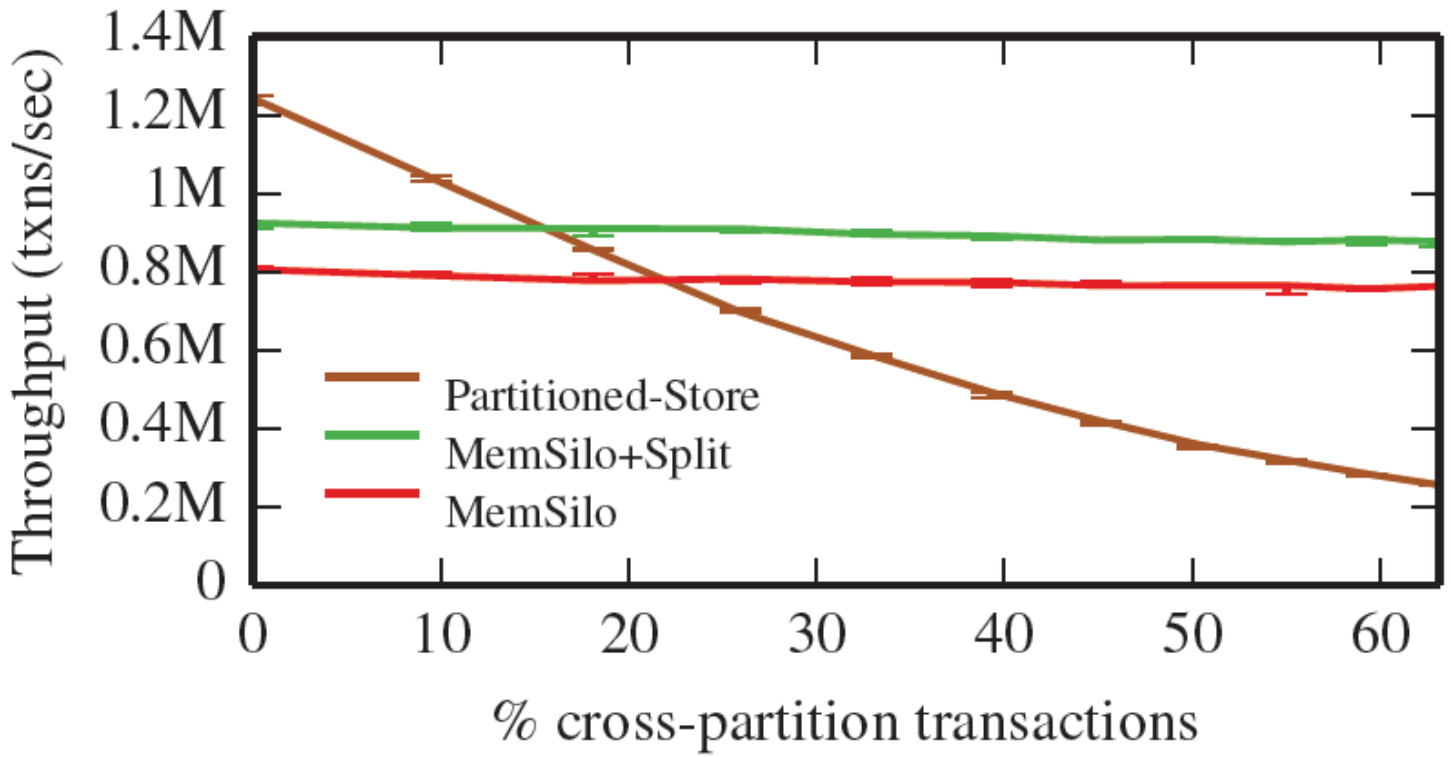
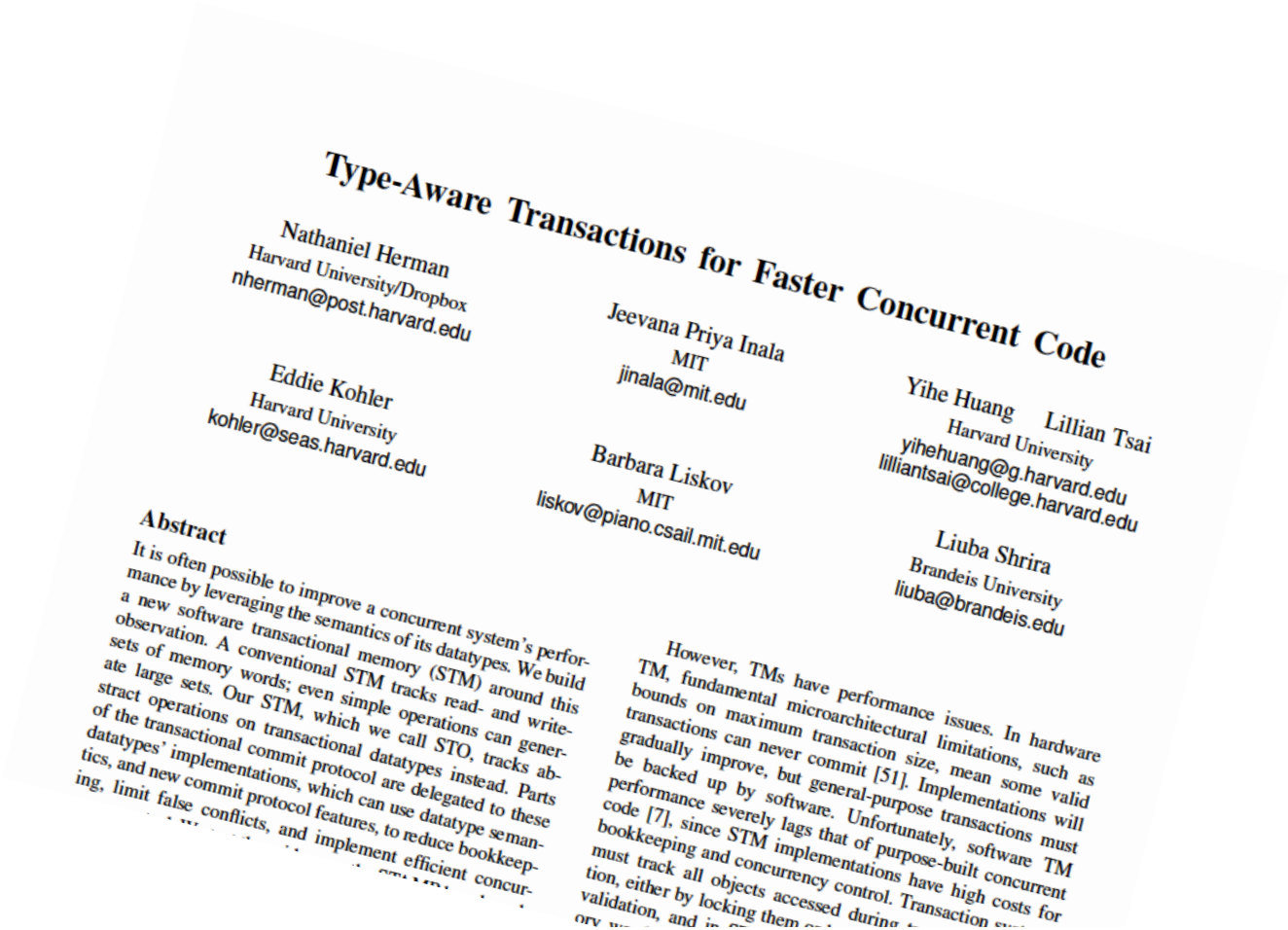


Figure 8: Performance of *Partitioned-Store* versus *MemSilo* as the percentage of cross-partition transactions is varied.

Silo to STO

- STO (Software Transactional Objects)



Type-Aware Transactions for Faster Concurrent Code

Nathaniel Herman
Harvard University/Dropbox
nherman@post.harvard.edu

Jeevana Priya Inala
MIT
jinala@mit.edu

Yihe Huang
Harvard University
yihehuang@g.harvard.edu

Lillian Tsai
Harvard University
lilliantsai@college.harvard.edu

Eddie Kohler
Harvard University
kohler@seas.harvard.edu

Barbara Liskov
MIT
liskov@piano.csail.mit.edu

Liuba Shrira
Brandeis University
liuba@brandeis.edu

Abstract

It is often possible to improve a concurrent system's performance by leveraging the semantics of its datatypes. We build a new software transactional memory (STM) around this observation. A conventional STM tracks read- and write-sets of memory words; even simple operations can generate large sets. Our STM, which we call STO, tracks abstract operations on transactional datatypes instead. Parts of the transactional commit protocol are delegated to these datatypes' implementations, which can use datatype semantics, and new commit protocol features, to reduce bookkeeping, limit false conflicts, and implement efficient concur-

However, TMs have performance issues. In hardware TM, fundamental microarchitectural limitations, such as bounds on maximum transaction size, mean some valid transactions can never commit [51]. Implementations must gradually improve, but general-purpose transactions must be backed up by software. Unfortunately, software TM performance severely lags that of purpose-built concurrent code [7], since STM implementations have high costs for bookkeeping and concurrency control. Transaction sys- must track all objects accessed during a tion, either by locking them or validation, and in

STO

- Silo trees are an highly concurrent data structures
  - Specification determines potential concurrency
  - Implementation is hidden
    - Including concurrency control

A vision for concurrent code

- Apps run transactions

A vision for concurrent application code, like boosting

- Apps run transactions
- Using transaction-aware datatypes
  - E.g., sets, maps, arrays, boxes, queues

Transactions

```
begin {
 % do stuff: run queries
 % using insert, lookup, update, delete,
 % and range
}
```

Back to our vision for concurrent code

- Apps run transactions
- Using fast transaction-aware datatypes
  - Designed by experts
  - Require sophistication to implement
  - But so are concurrent datatypes in Java

STO

- Think Silo broken into two parts:
  - STO platform
  - Transaction-aware datatypes

STO Platform

- Runs transactions
  - Transaction { … }
- Provides transaction state
  - Read- and write-sets
- Runs commit protocol using callbacks

Transaction-aware datatypes

- Provide ops for user code
  - E.g., lookup, update, insert, delete, range
- Record reads and writes via platform
- Provide callbacks
  - lock, unlock, check, install, cleanup

Transaction-aware datatypes

- Provide ops for user code
- Record reads and writes via platform
- Provide callbacks
  - lock, unlock, check, install, cleanup
  - cleanup for abort, after-commit
  - E.g., deleting a key

Transaction-aware types

- Maps
- Hash tables
- Counters
  - void incr( ) vs. int incr( )
  - Uses check and install

Designing fast STO's data types:

- Specification
- Some common tricks
  - Inserted elements: direct updates
  - Absent elements: extra version numbers
  - Read-my-writes: adjustments
- Correctness

Specification

Inserted elements and repeated lookup

- Hybrid strategy
  - T1: insert “poisoned” element
  - T2: abort on observing a “poisoned” element
  - T1: no need to validate insertion at commit

Absent elements

- T1: get(K) : K is absent
  - How to validate at commit?

- Extra version numbers
  - For hash table: on bucket of absent key
  - BTree : on parent node of absent key

Read-my-writes

- T1: scan a range A..Z; insert a key C
  - how to validate range ?

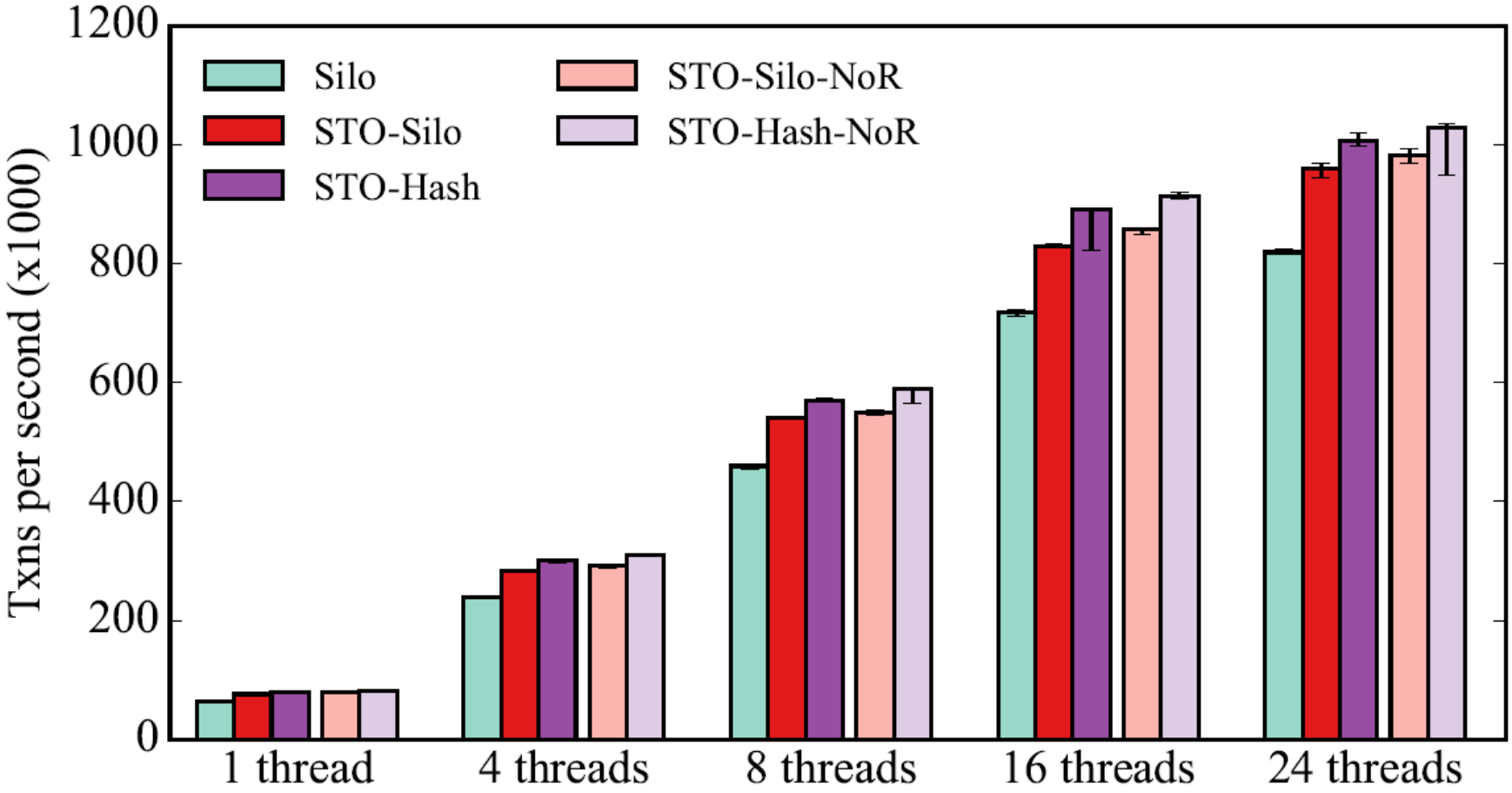
Correctness

- Version numbers on all shared state
- Exclusive locks
- Check must fail if segment locked or version number changed
- Modifications invisible to other transactions before install

Performance

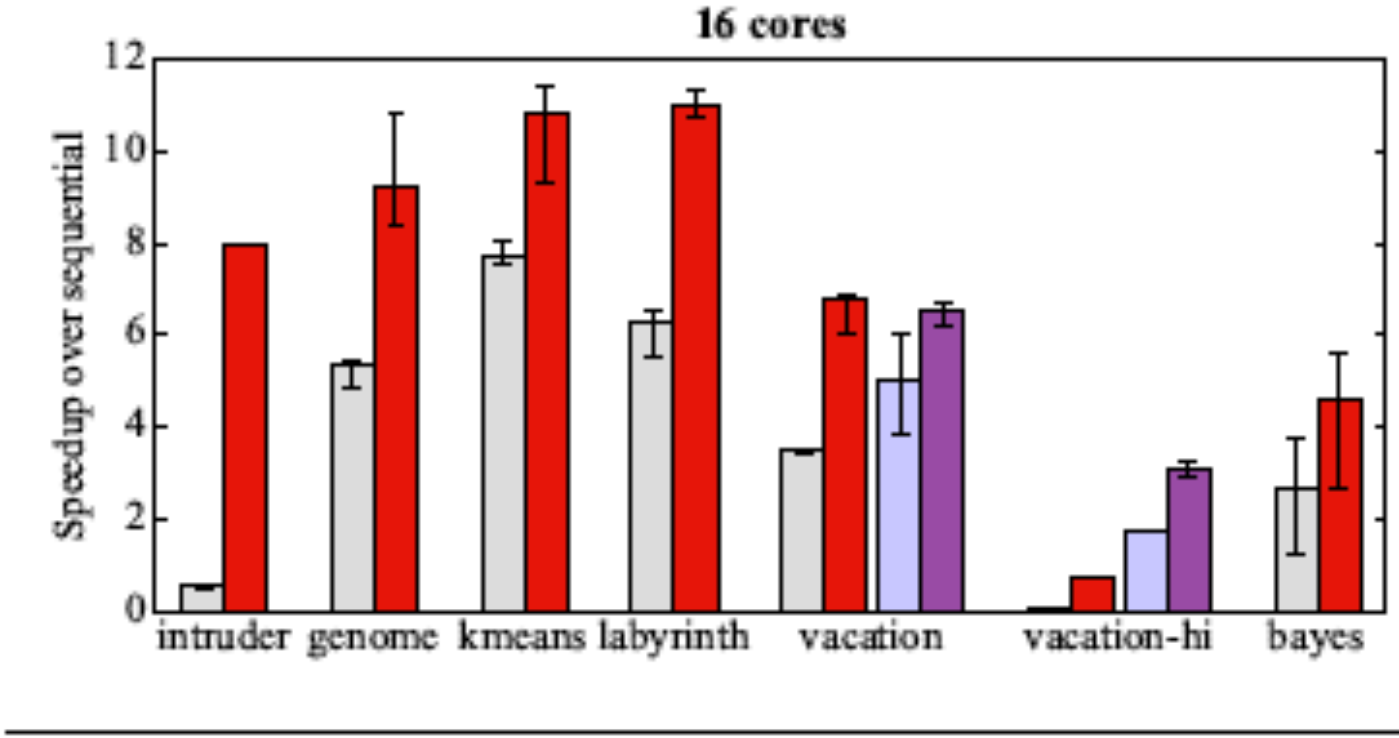
Implementation

- Silo: 7000 lines of code
- STO-Silo: 3000 lines of code
  - Uses hash tables and trees


Txns per second (x1000)
1200
1000
800
600
400
200
0
Silo
STO-Silo
STO-Hash
STO-Silo-NoR
STO-Hash-NoR
1 thread
4 threads
8 threads
16 threads
24 threads

Performance

- vs. TL2 (grey)
- And boosting (lilac)



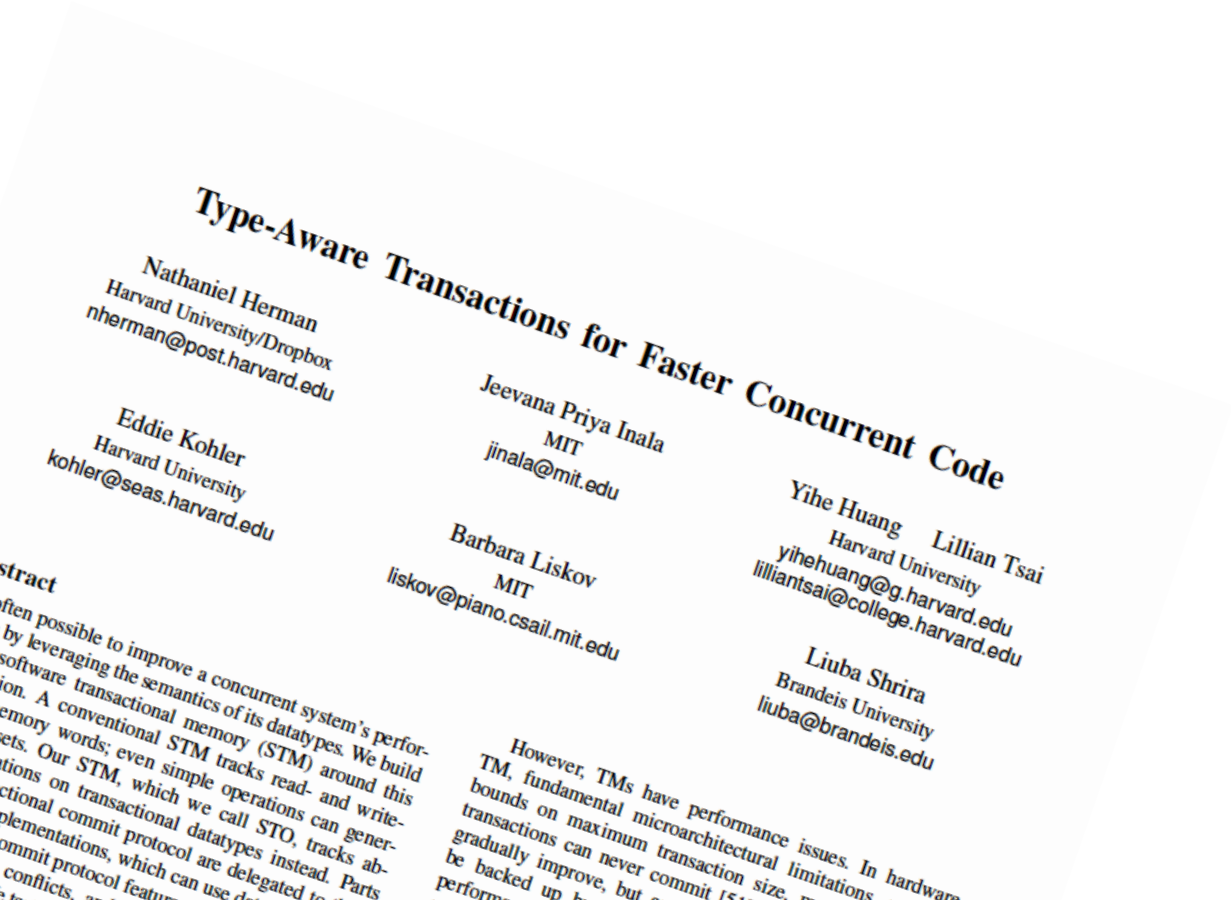
Optimism vs Pessimism?

Effects of pessimism and boosting on a hash table micro benchmark.

Numbers are speedup at 16 threads relative to single-threaded STO

STO	12.91x
Boosting	7.02x
Boosting in STO	6.67x
Pessimistic STO	9.82x

More examples of powerful optimizations



Type-Aware Transactions for Faster Concurrent Code

Nathaniel Herman
Harvard University/Dropbox
nherman@post.harvard.edu

Eddie Kohler
Harvard University
kohler@seas.harvard.edu

Jeevana Priya Inala
MIT
jinala@mit.edu

Barbara Liskov
MIT
liskov@piano.csail.mit.edu

Yihe Huang
Harvard University
yihehuang@g.harvard.edu

Lillian Tsai
lilliantsai@college.harvard.edu

Liuba Shrira
Brandeis University
liuba@brandeis.edu

stract

ften possible to improve a concurrent system's perfor-
by leveraging the semantics of its datatypes. We build
software transactional memory (STM) around this
ion. A conventional STM tracks read- and write-
emory words; even simple operations can gener-
ets. Our STM, which we call STO, tracks ab-
tions on transactional datatypes instead. Parts
ctional commit protocol are delegated to
plementations, which can use
ommit protocol featu
conflicts,

However, TMs have performance issues. In hardware
TM, fundamental microarchitectural limitations
bounds on maximum transaction size,
transactions can never commit
gradually improve, but
be backed up
performa

STO: last word for exploiting ADT in TM?

- Needs more work
  - More datatypes
  - Methodology
  - Programming language integration
  - Distribution

ummary:
mplementing a Library of Transactional Data types:

- Distinction between short Thread level vs coarse grain Transaction-level coordination is key
- Can re-use data structure code or co-design and customize:
  - Boosting: a black box approach, first ADT/STM (code re-use, restrictions)
  - STO: high-performance pessimistic/optimistic approach (co-design and customize)
- (Thanks to M.Herlihy and B. Liskov for help with slides!)

Questions