

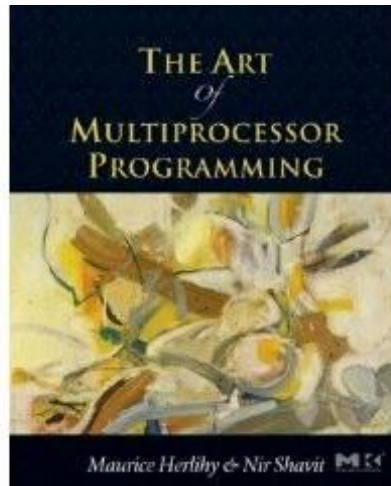
NON-BLOCKING DATA STRUCTURES AND TRANSACTIONAL MEMORY

Tim Harris, 25 Oct 2019

Lecture 3/3

- Problems with locks
- Atomic blocks and composition
- Hardware transactional memory
- Software transactional memory

Transactional Memory



Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

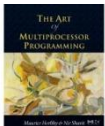
Our Vision for the Future

In this course, we covered

Best practices ...

New and clever ideas ...

And common-sense observations.



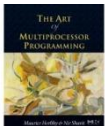
Our Vision for the Future

Nevertheless ...

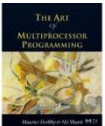
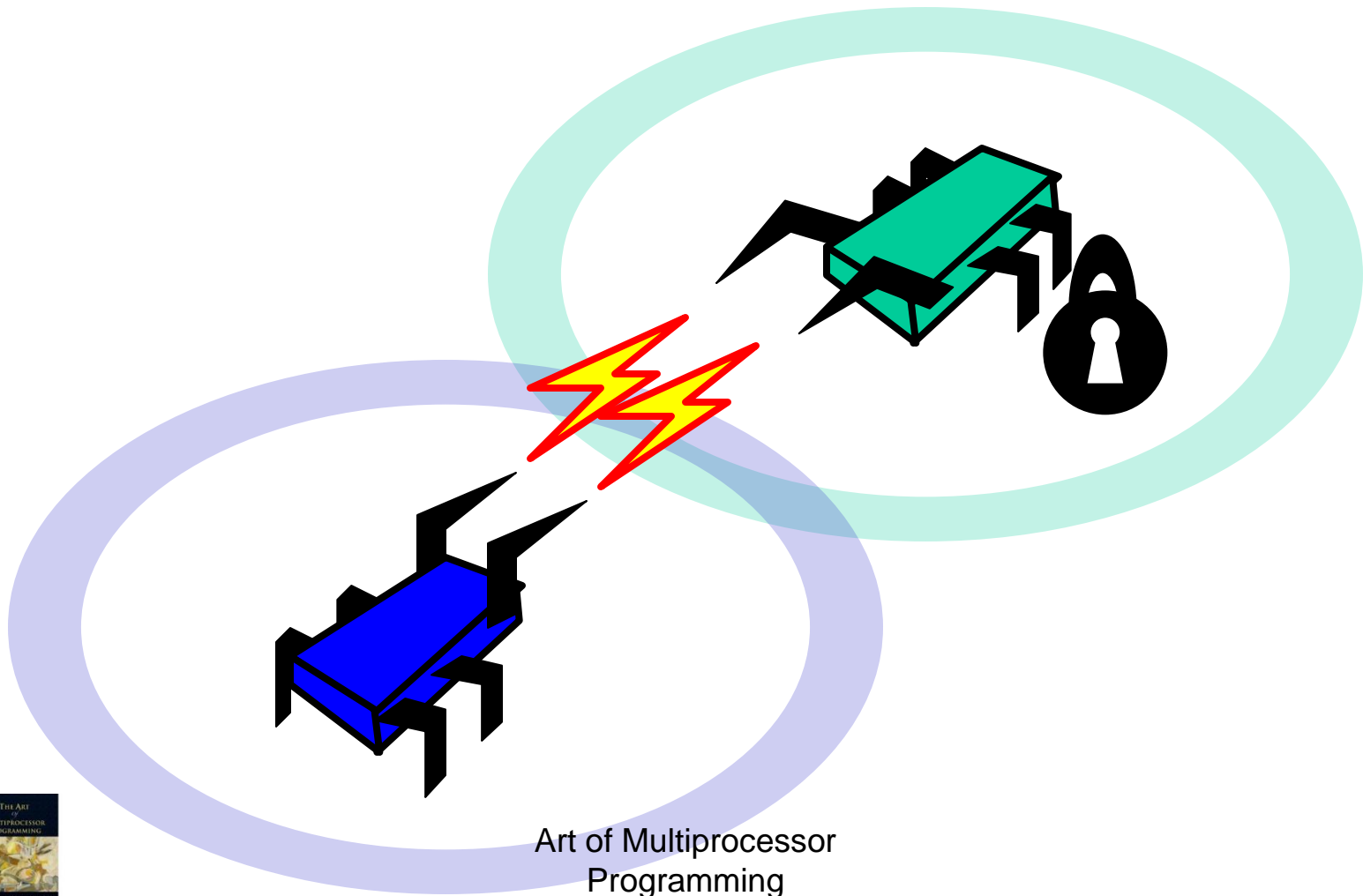
Concurrent programming is still too hard ...

Here we explore why this is

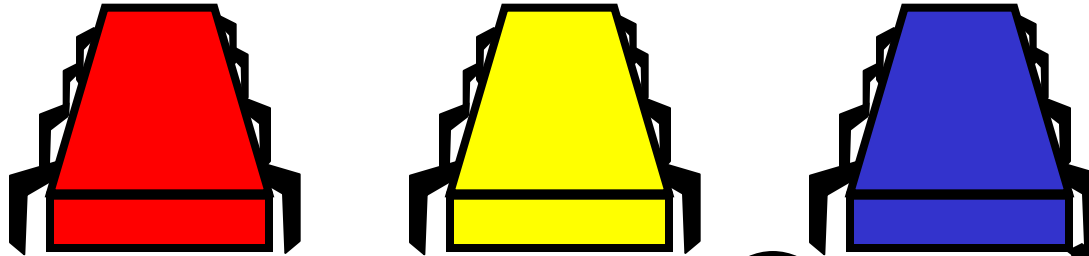
And what we can do about it.



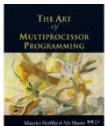
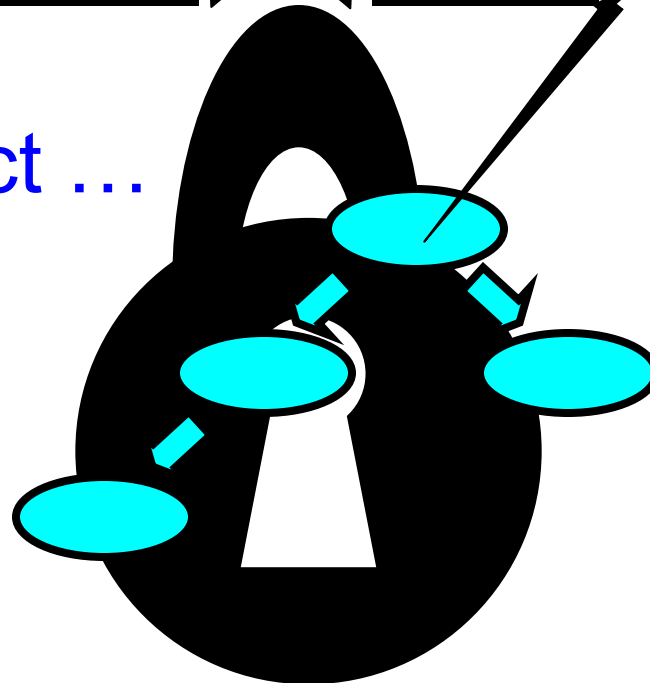
Locking



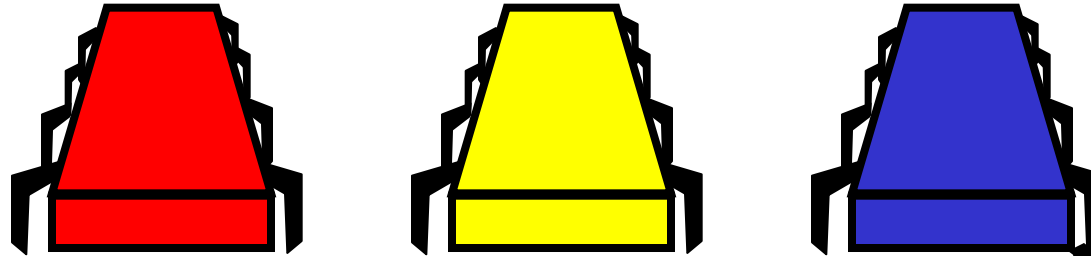
Coarse-Grained Locking



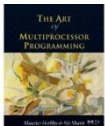
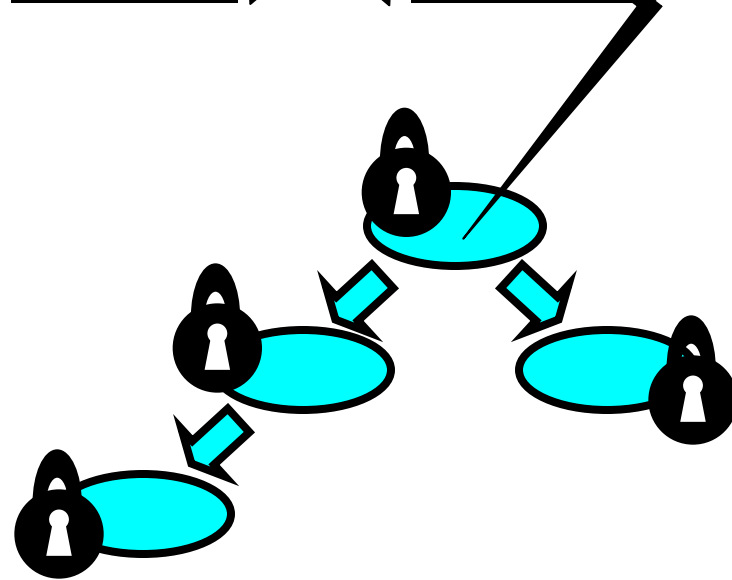
Easily made correct ...
But not scalable.



Fine-Grained Locking

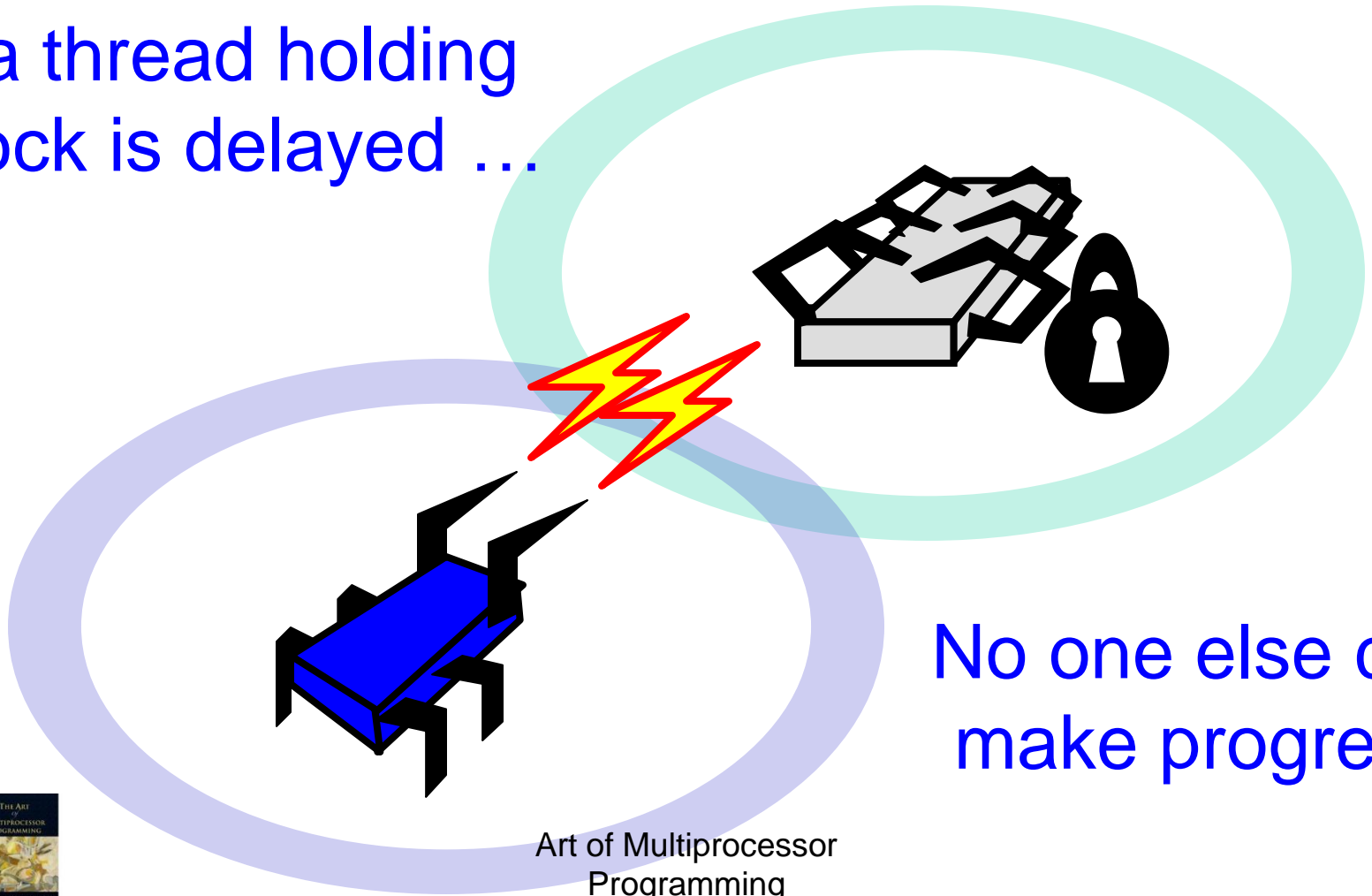


Can be tricky ...

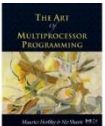


Locks are not Robust

If a thread holding a lock is delayed ...



No one else can make progress

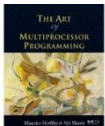


Locking Relies on Conventions

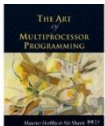
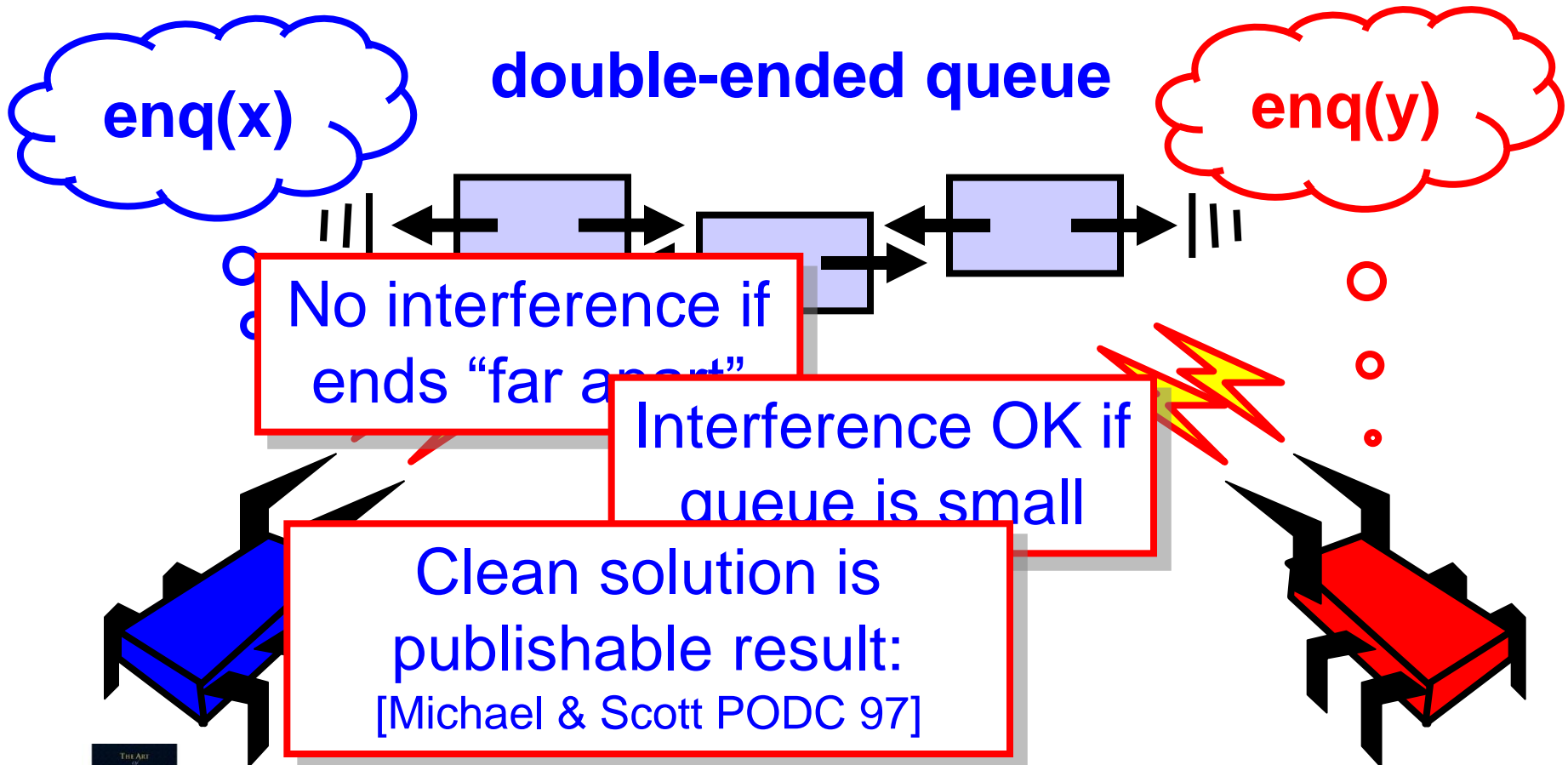
- Relation between
 - Locks and objects
 - Exists only in programmer's

Actual comment
from Linux Kernel
(hat tip: Bradley Kuszmaul)

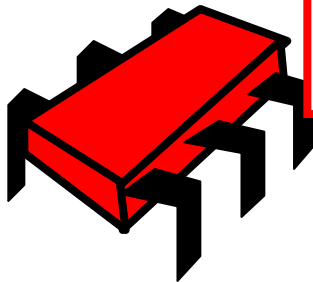
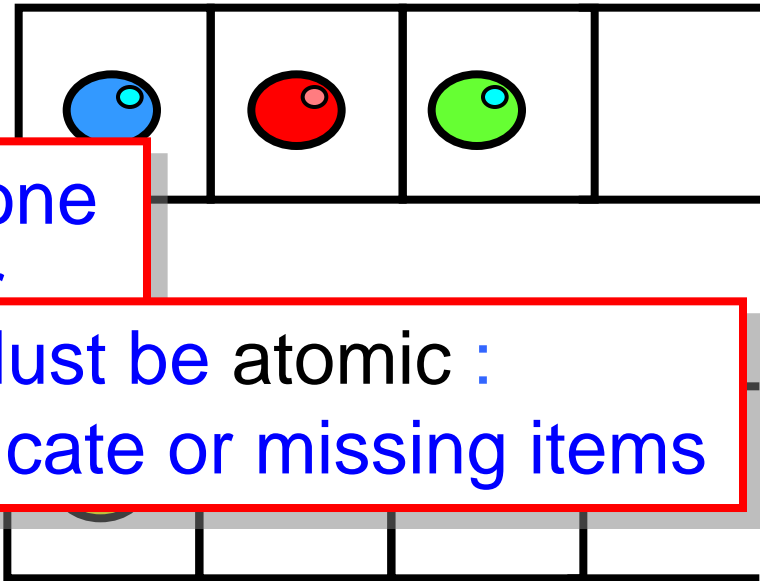
```
/*  
 * When a locked buffer is visible to the I/O layer  
 * BH_Laundry is set. This means before unlocking  
 * we must clear BH_Laundry, mb() on alpha and then  
 * clear BH_Lock, so no reader can see BH_Laundry set  
 * on an unlocked buffer and then risk to deadlock.  
 */
```



Simple Problems are hard



Locks Not Composable



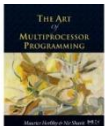
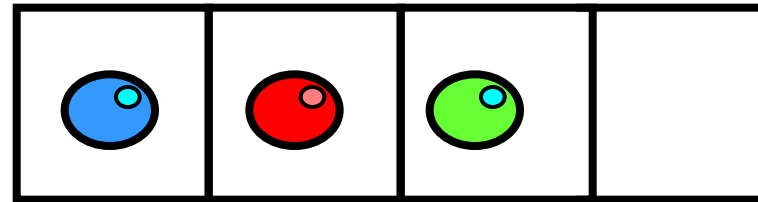
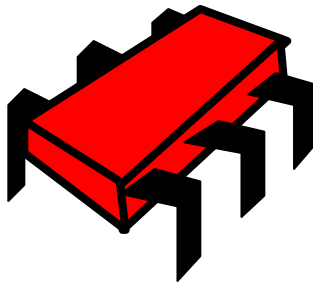
Locks Not Composable



Lock source

Unlock source
& target

Lock target



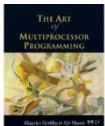
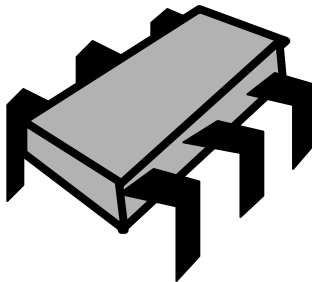
Locks Not Composable

Methods cannot provide
internal synchronization

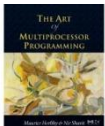
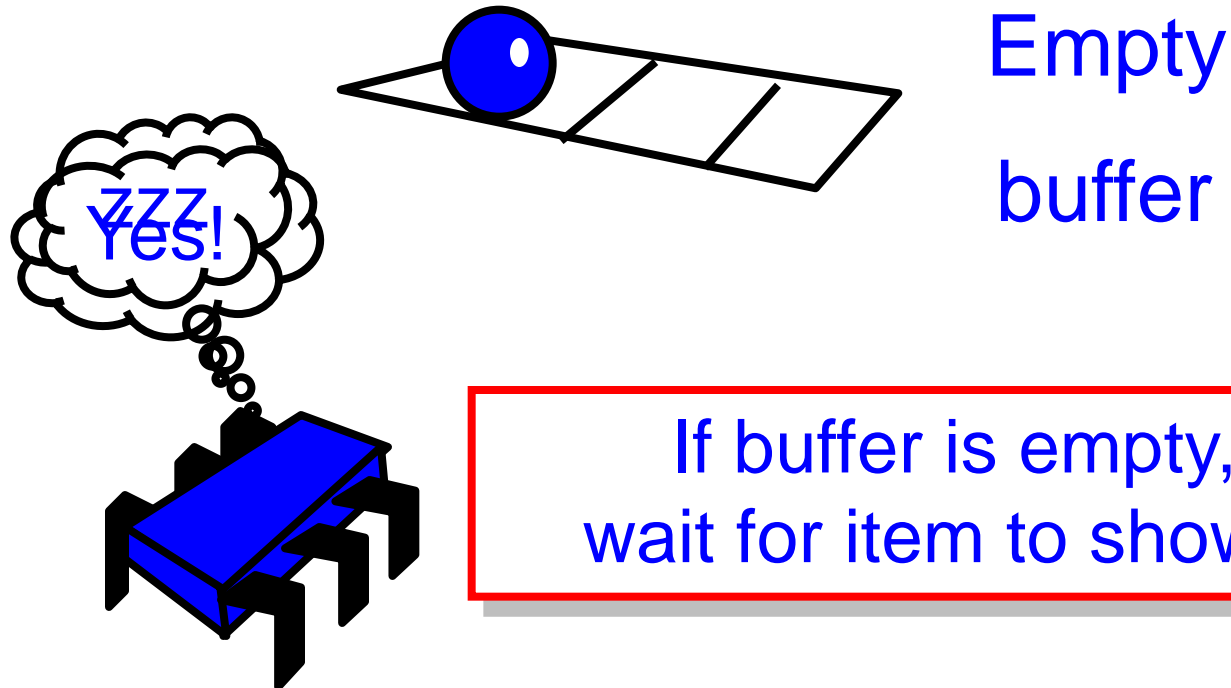
Objects must expose
locking protocols to clients

Clients must devise and
follow protocols

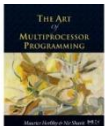
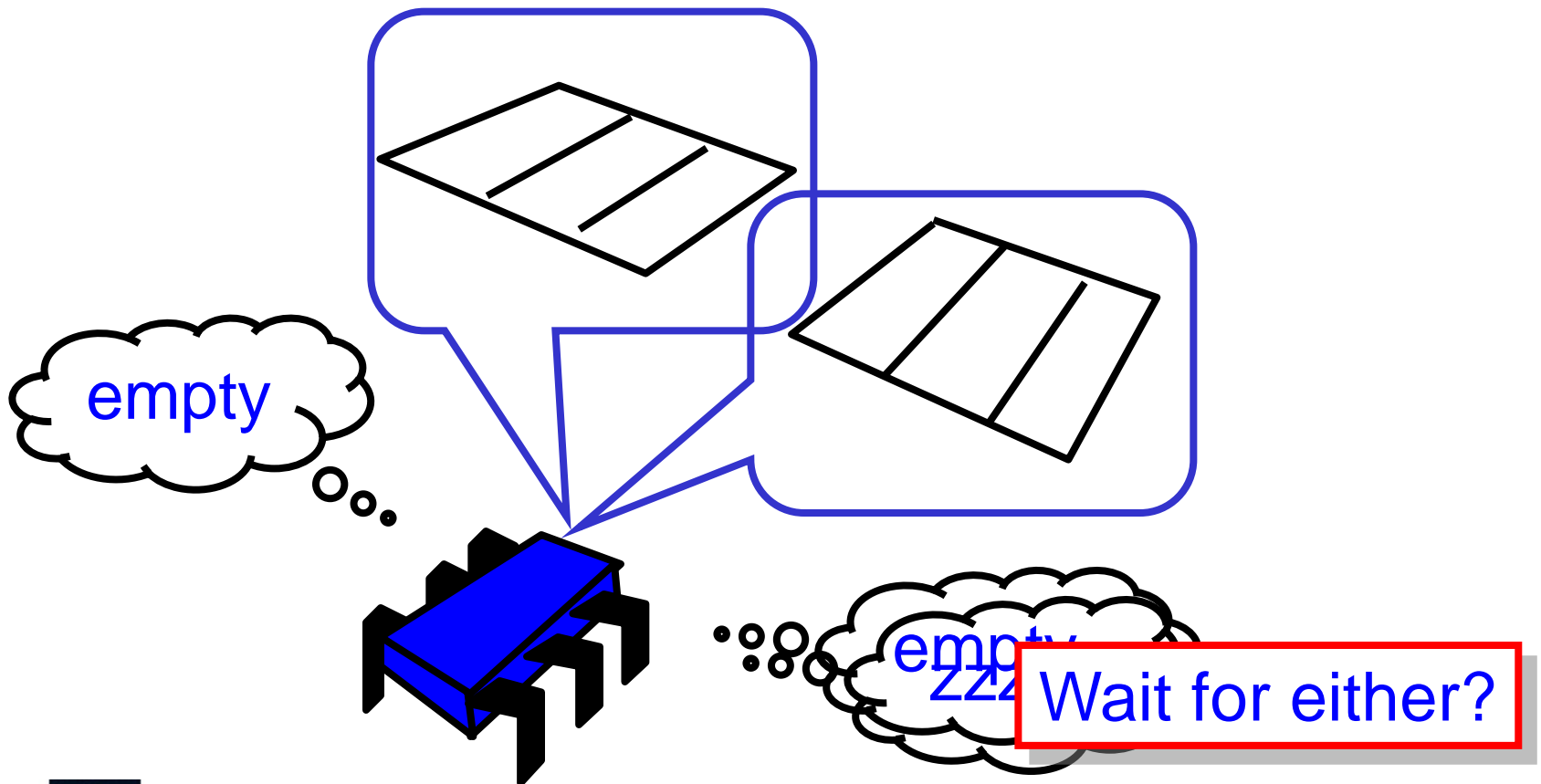
Abstraction broken!



Monitor Wait and Signal

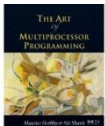


Wait and Signal do not Compose



The Transactional Manifesto

- Current practice inadequate
 - to meet the multicore challenge
- Research Agenda
 - Replace **locking** with a **transactional API**
 - **Design** languages or libraries
 - **Implement** efficient run-time systems



Transactions

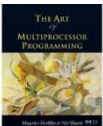
Block of code

Atomic: appears to happen instantaneously

Serializable: all appear to happen in one-at-a-time

Commit: takes effect (atomically)

Abort: has no effect (typically restarted)



Atomic Blocks

```
atomic {  
  x.remove(3);  
  y.add(3);  
}
```

```
atomic {  
  y = null;  
}
```



Atomic Blocks

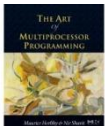
```
atomic {  
  x.remove(3);  
  y.add(3);  
}
```

y.add(3);

No data race

```
atomic {  
  y = null;  
}
```

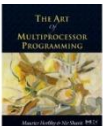
y = null;



A Double-Ended Queue

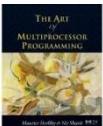
```
public void LeftEnq(item x) {  
    Qnode q = new Qnode(x);  
    q.left = left;  
    left.right = q;  
    left = q;  
}
```

Write sequential Code



A Double-Ended Queue

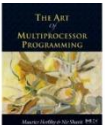
```
public void LeftEnq(item x)
atomic {
    Qnode q = new Qnode(x);
    q.left = left;
    left.right = q;
    left = q;
}
}
```



A Double-Ended Queue

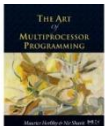
```
public void LeftEnq(item x) {  
    atomic {  
        Qnode q = new Qnode(x);  
        q.left = left;  
        left.right = q;  
        left = q;  
    }  
}
```

Enclose in atomic block

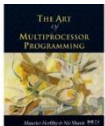
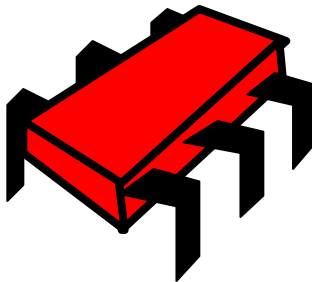
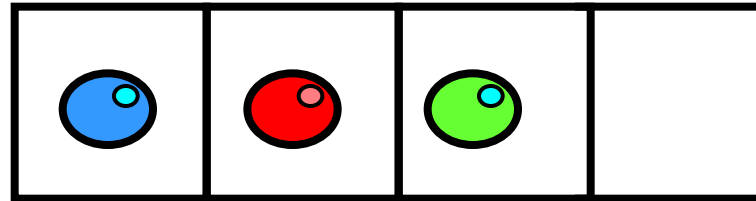


Warning

- Not always this simple
 - Conditional waits
 - Enhanced concurrency
 - Complex patterns
- But often it is...



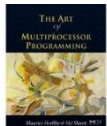
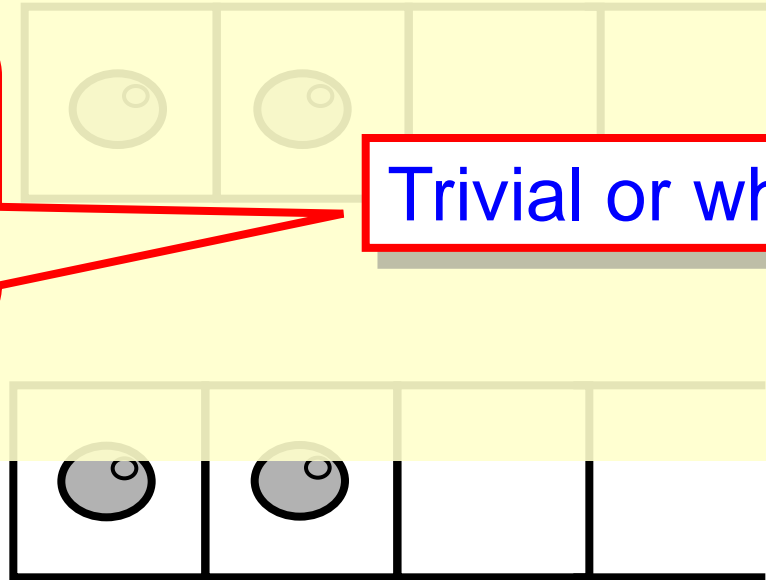
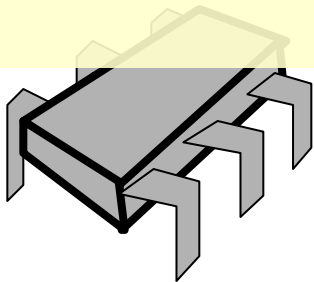
Composition?



Composition?

```
public void Transfer (Queue<T> q1, q2)
{
  atomic {
    T x = q1.deq();
    q2.enq(x);
  }
}
```

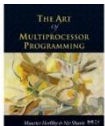
Trivial or what?



Conditional Waiting

```
public T LeftDeq() {  
    atomic {  
        if (left == null)  
            retry;  
        ...  
    }  
}
```

**Roll back transaction
and restart when
something changes**



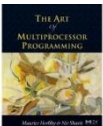
Composable Conditional Waiting

```
atomic {  
  x = q1.deq();  
} orElse {  
  x = q2.deq();  
}
```

Run 1st method. If it retries

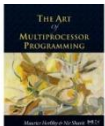
Run 2nd method. If it retries ...

Entire statement retries

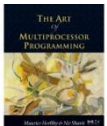
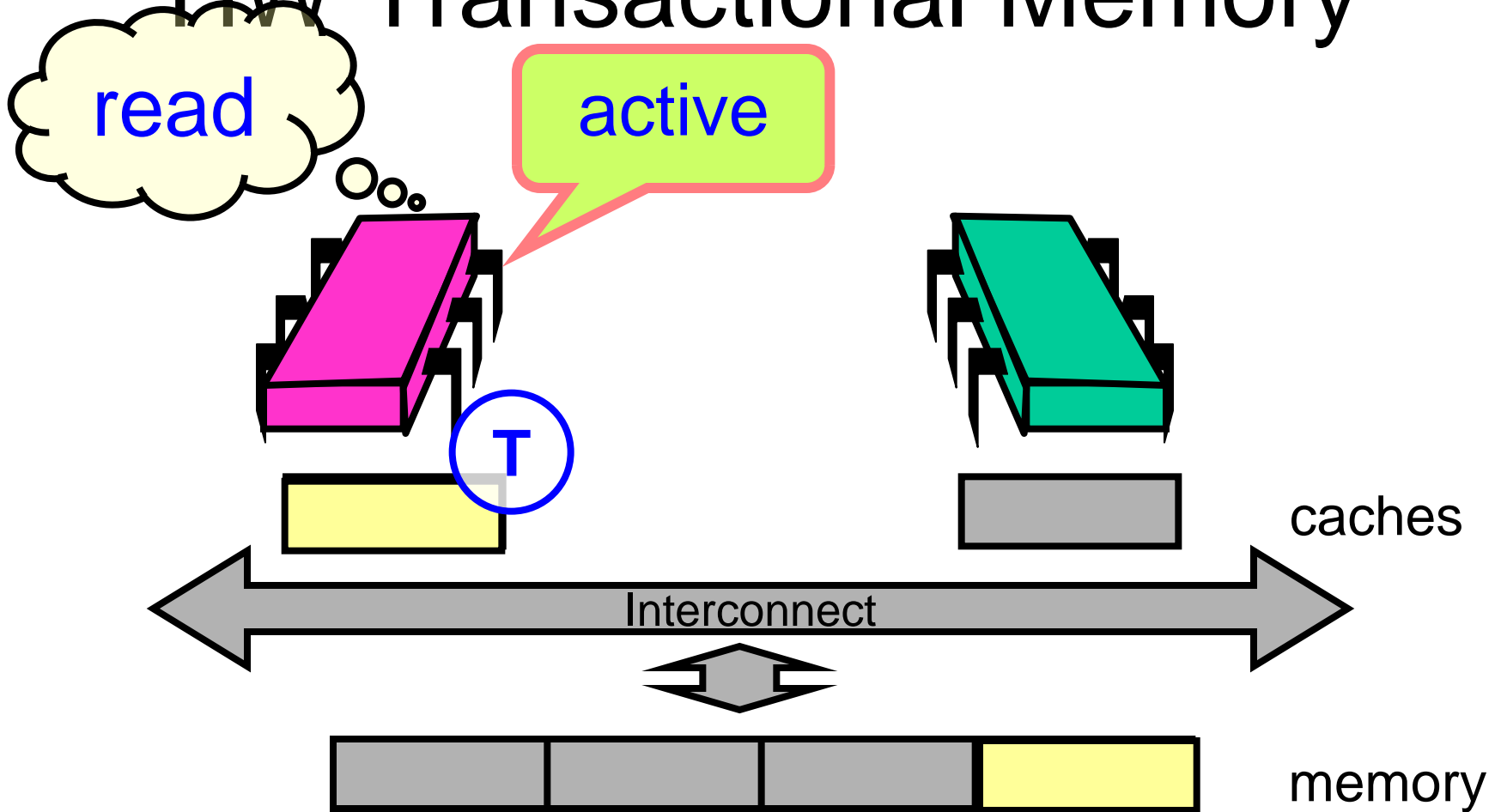


Hardware Transactional Memory

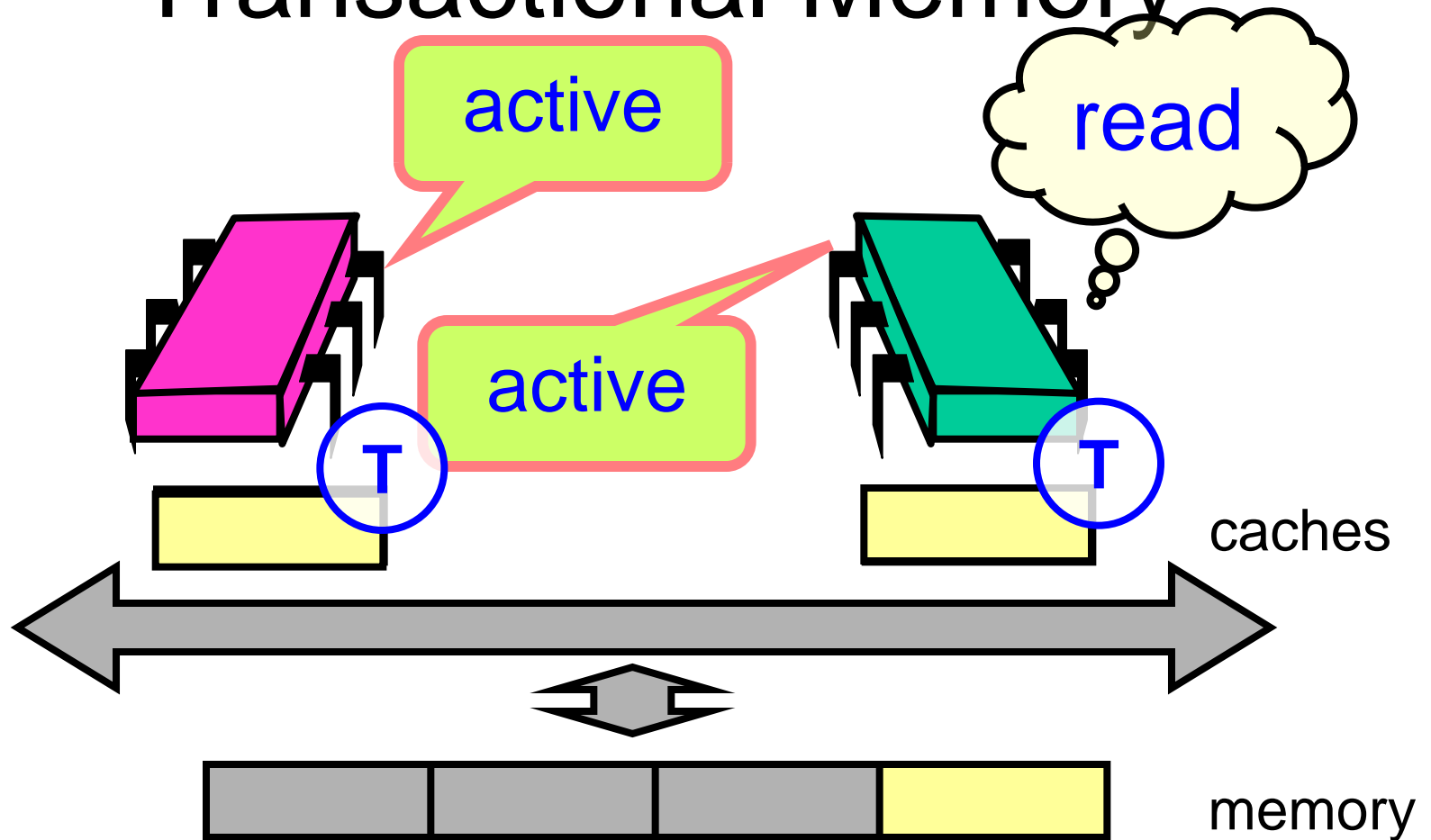
- Exploit Cache coherence
- Already almost does it
 - Invalidation
 - Consistency checking
- Speculative execution
 - Branch prediction = optimistic synch!



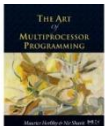
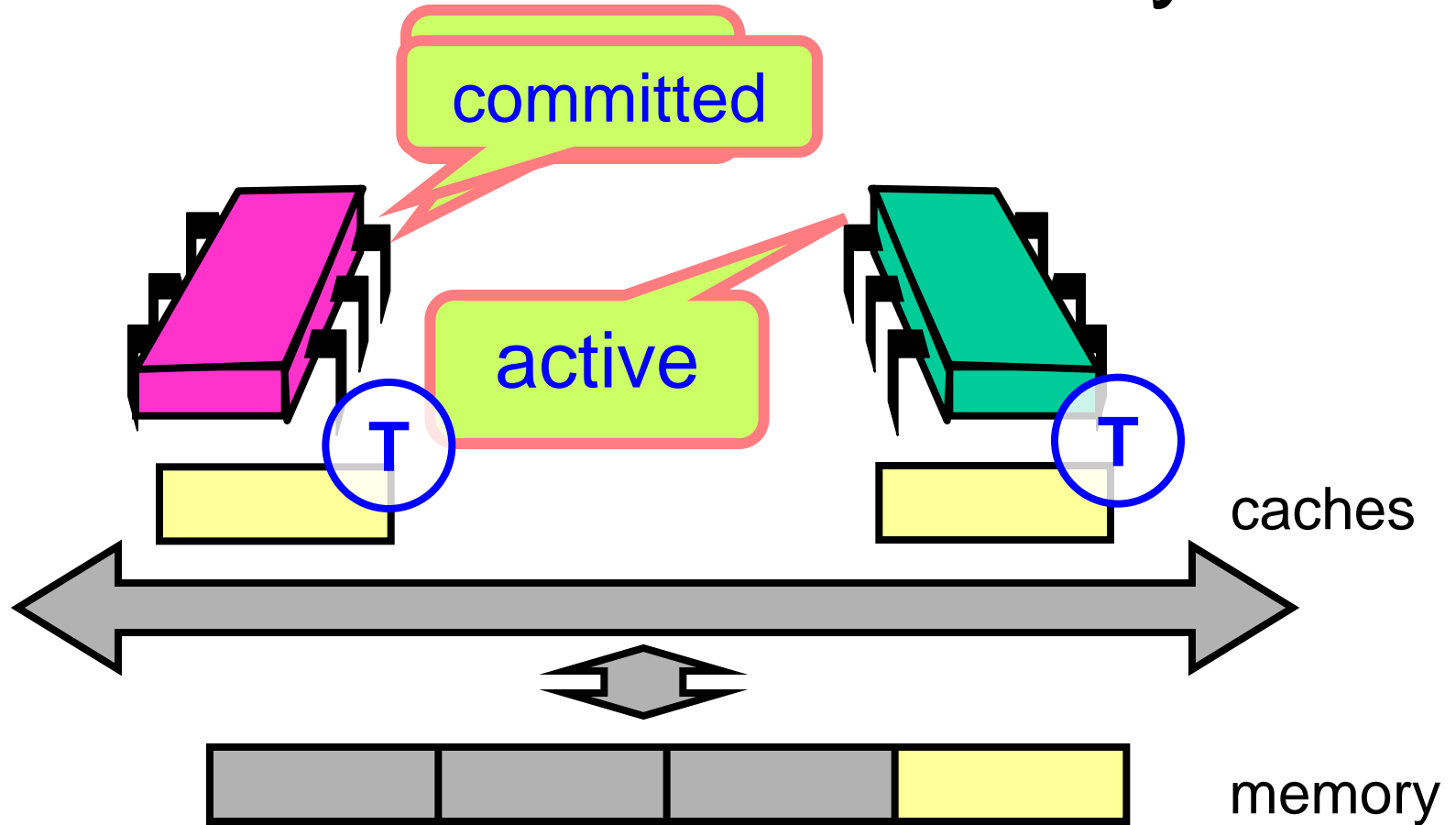
HW Transactional Memory



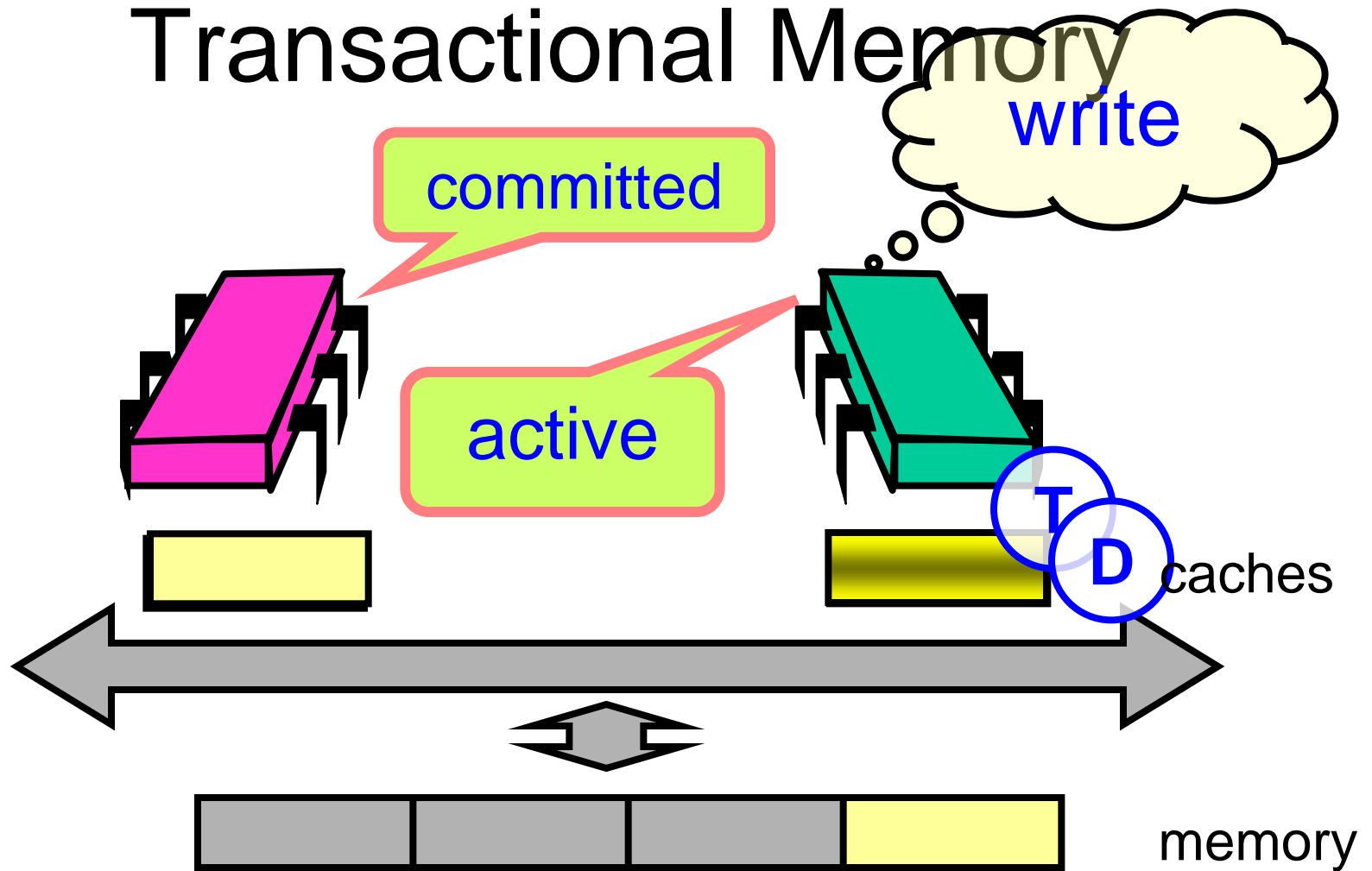
Transactional Memory

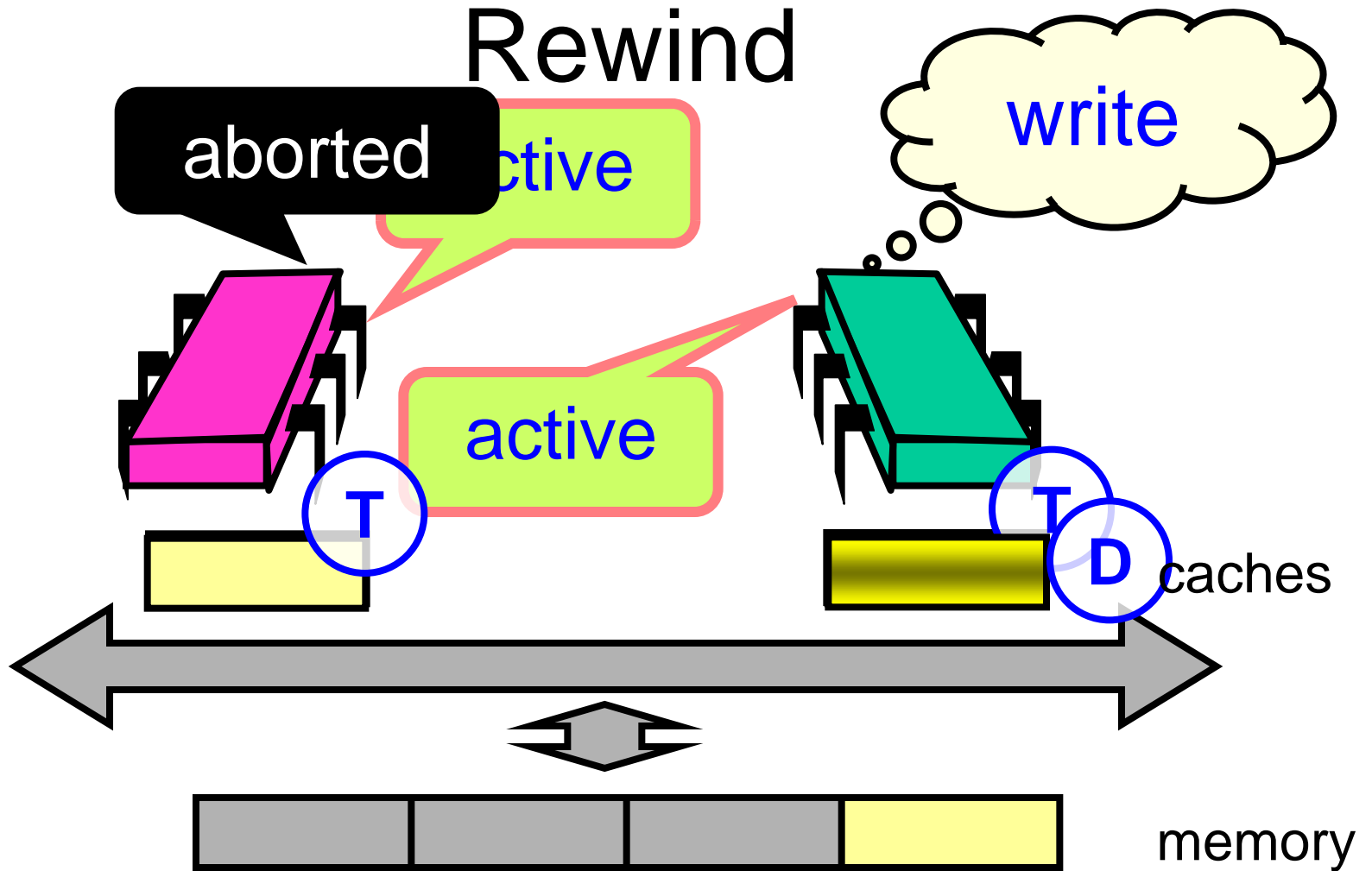


Transactional Memory



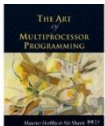
Transactional Memory





Transaction Commit

- At commit point
 - If no cache conflicts, we win.
- Mark transactional entries
 - Read-only: valid
 - Modified: dirty (eventually written back)
- That's all, folks!
 - Except for a few details ...



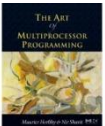
Not all  and



- Limits to
 - Transactional cache size
 - Scheduling quantum
- Transaction cannot commit if it is
 - Too big
 - Too slow
 - Actual limits platform-dependent

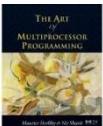
HTM Strengths & Weaknesses

- Ideal for lock-free data structures



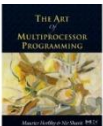
HTM Strengths & Weaknesses

- Ideal for lock-free data structures
- Practical proposals have limits on
 - Transaction size and length
 - Bounded HW resources
 - Guarantees vs best-effort



HTM Strengths & Weaknesses

- Ideal for lock-free data structures
- Practical proposals have limits on
 - Transaction size and length
 - Bounded HW resources
 - Guarantees vs best-effort
- On fail
 - Diagnostics essential
 - Try again in software?



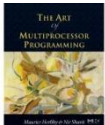
Composition

Locks don't compose, transactions do.

Composition necessary for Software Engineering.

But practical HTM doesn't *really* support composition!

Why we need STM



Composability in Transactions
publication version, 2006
Herlihy
Simon Peyton Jones
Cambridge
[...}@microsoft.com
Tim Harris
{tharris, simon@cam.ac.uk}
vent threads from
provide a convincing story
ner we resolve these shorter
contributions:
the ideas of transactional memory in the setting
11 (Section 3). This is much more than a
declarative language, and we
ive far stronger guarantees
more transactions
together to
which appear
retry (S

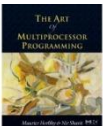
Note: this post-publication cited in the journal has been fixed, and an Appendix is included.

Abstract
Writing concurrent programs is notoriously difficult. Increasing practical importance. A particular source of complexity is even correctly-implemented concurrency abstractions cannot be composed together to form larger abstractions. In this paper we present a new concurrency model, based on transactional memory composition. All the usual benefits of transactional memory (e.g. freedom from deadlock), but in forms of blocking and choice

- We present a new programming model for STM that is more expressive than the previous models, and is more robust to the programmer's errors.

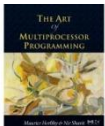
Transactional Consistency

- Memory Transactions are collections of reads and writes executed atomically
- They should maintain *consistency*
 - *External*: with respect to the interleavings of other transactions (*linearizability*).
 - *Internal*: the transaction itself should operate on a consistent state.



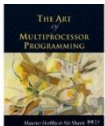
A Simple Lock-Based STM

- STMs come in different forms
 - Lock-based
 - Lock-free
- Here : a simple lock-based STM
- Lets start by Guaranteeing External Consistency

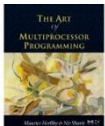
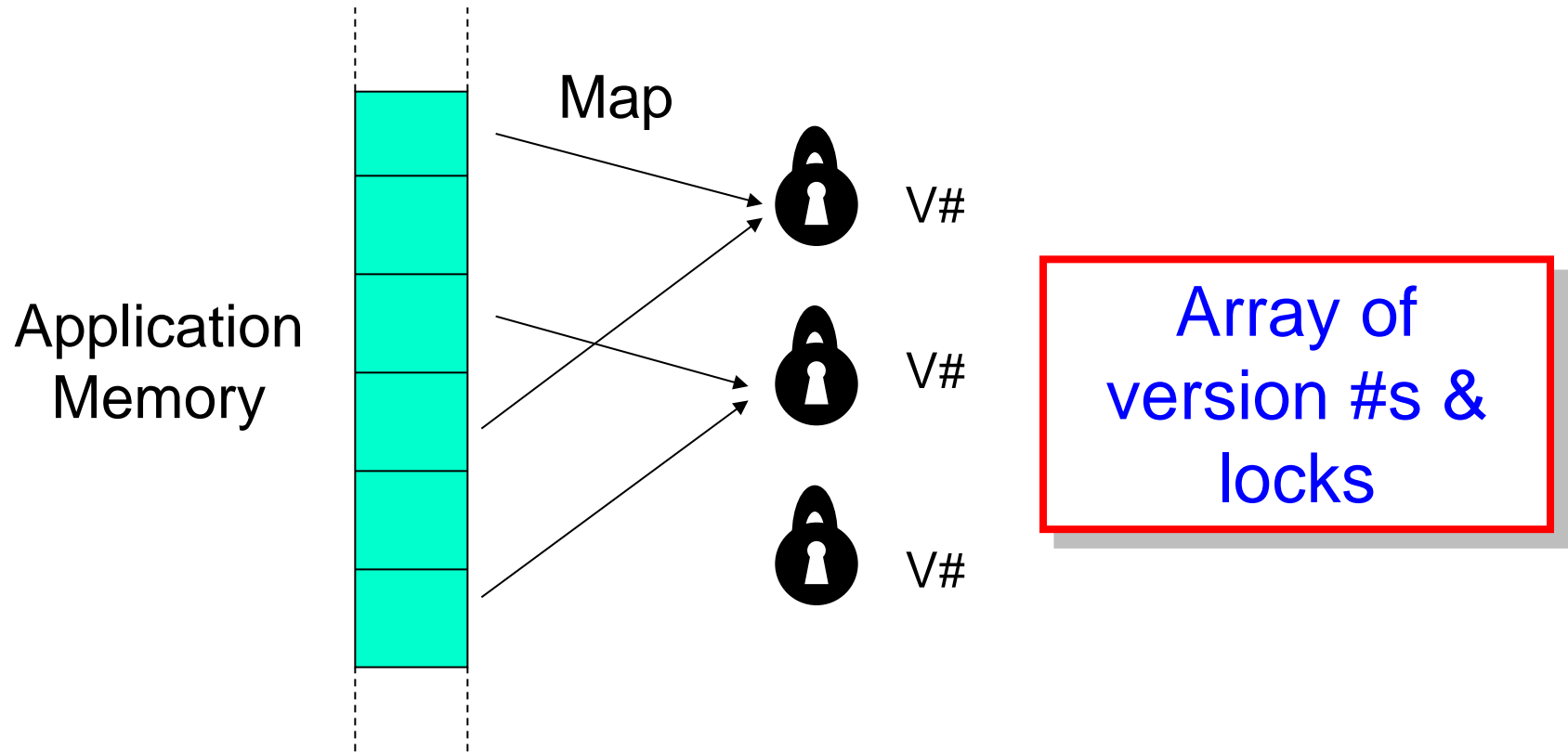


Synchronization

- Transaction keeps
 - **Read set**: locations & values read
 - **Write set**: locations & values to be written
- Deferred update
 - Changes installed at commit
- Lazy conflict detection
 - Conflicts detected at commit



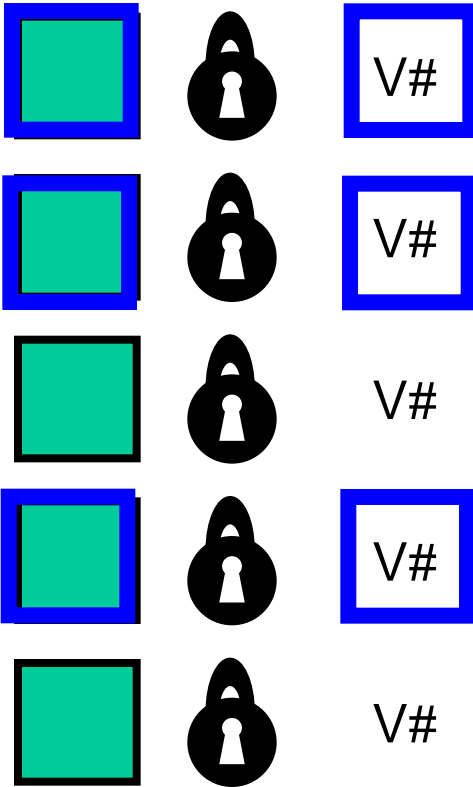
STM: Transactional Locking



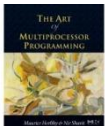
Reading an Object

Mem

Locks



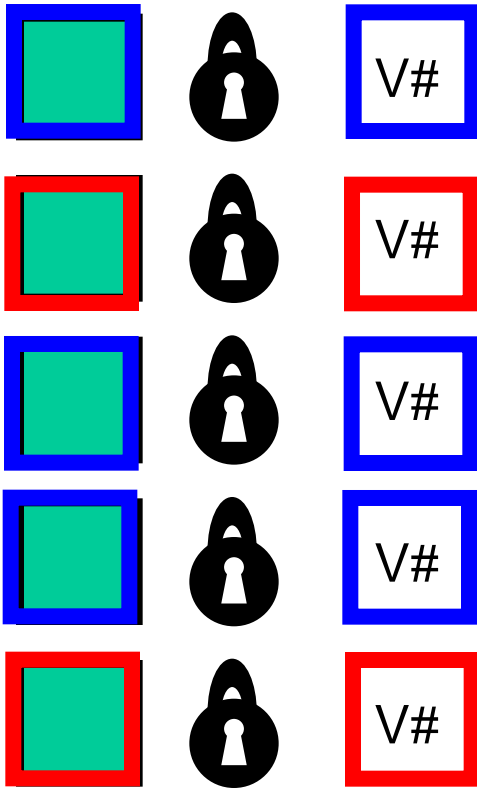
Add version numbers
& values to **read set**



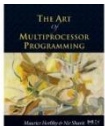
To Write an Object

Mem

Locks



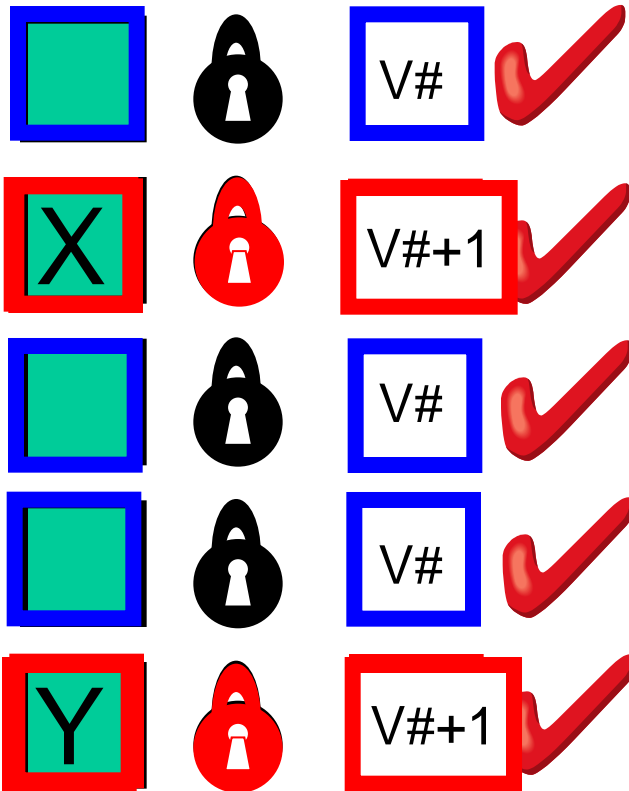
Add version numbers & new values to write set



To Commit

Mem

Locks



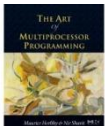
Acquire write locks

Check version numbers
unchanged

Install new values

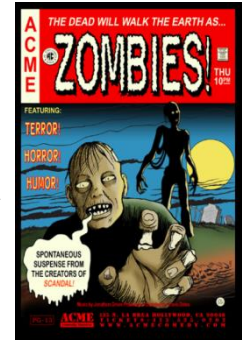
Increment version numbers

Unlock.

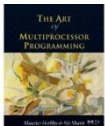




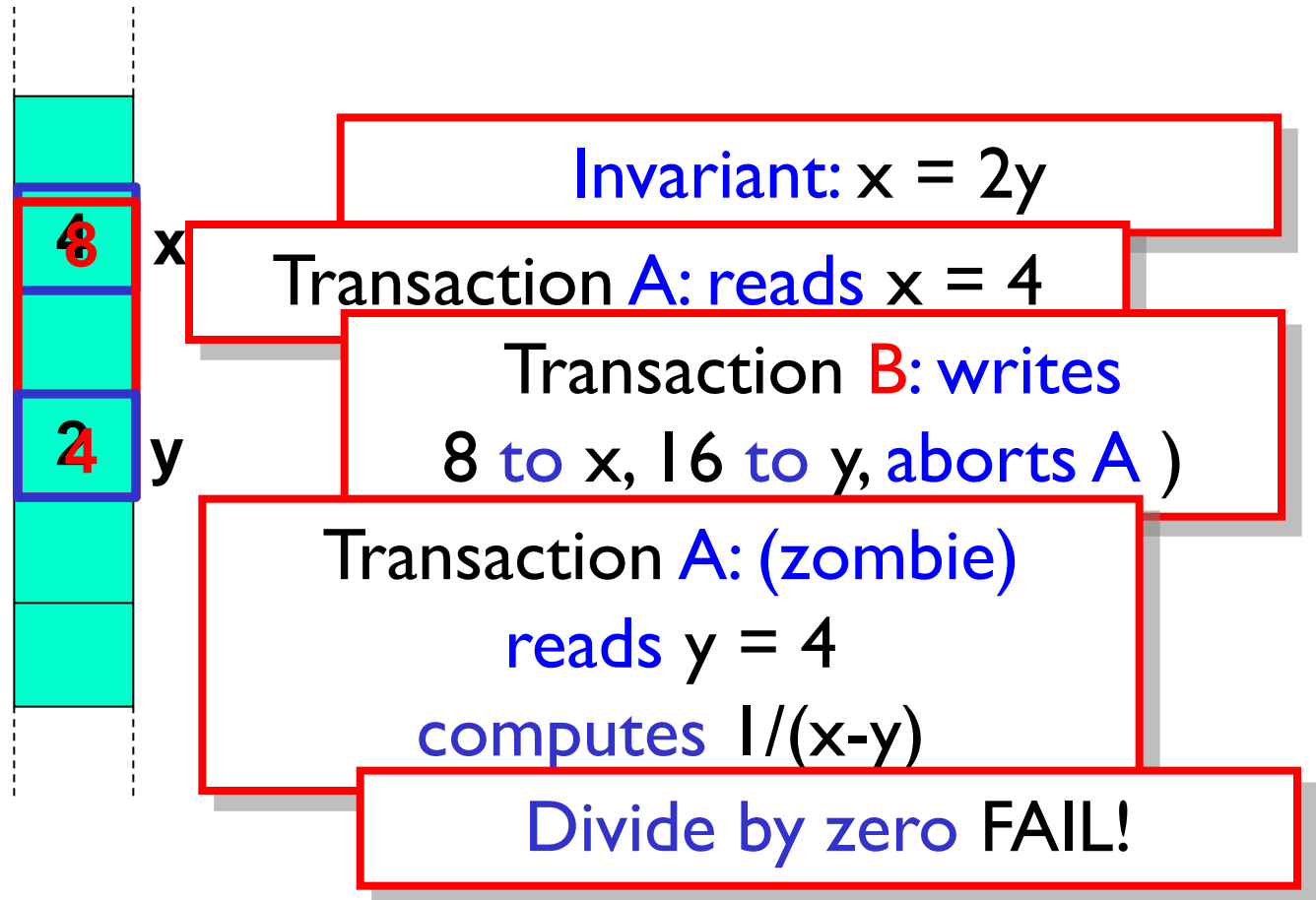
Problem: Internal Inconsistency



- A Zombie is an active transaction destined to abort.
- If Zombies see inconsistent states bad things can happen

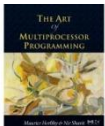


Internal Consistency



Solution: The Global Clock (The TL2 Algorithm)

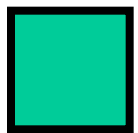
- Have one shared global clock
- Incremented by (small subset of) writing transactions
- Read by all transactions
- Used to validate that state worked on is always consistent



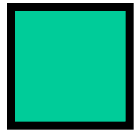
Read-Only Transactions

Mem

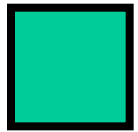
Locks



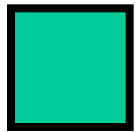
12



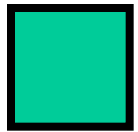
32



56

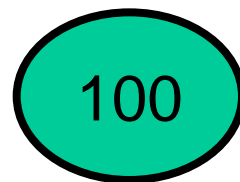


19

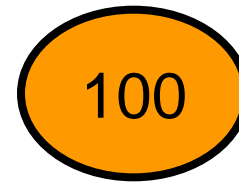


17

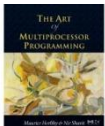
Copy version clock to local read version clock



Shared Version Clock



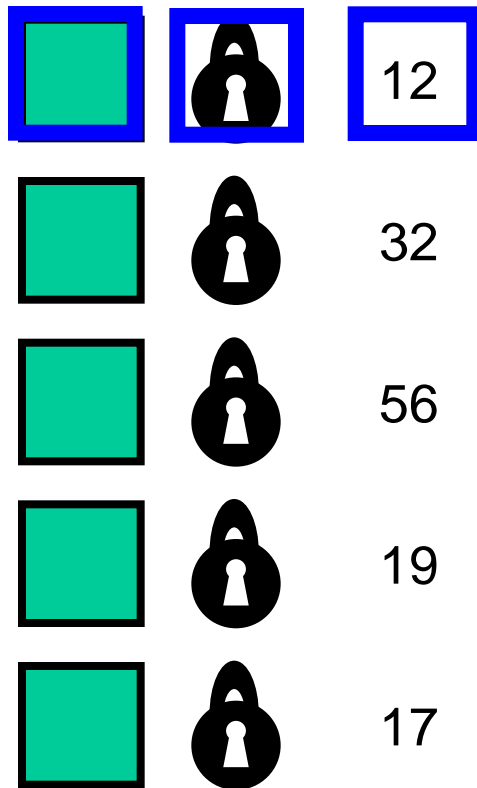
Private Read Version (RV)



Read-Only Transactions

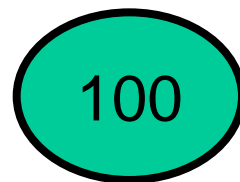
Mem

Locks

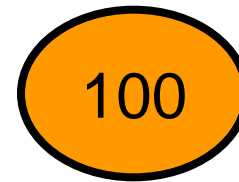


Copy version clock to local

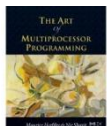
Read lock, version #, and memory



Shared Version Clock



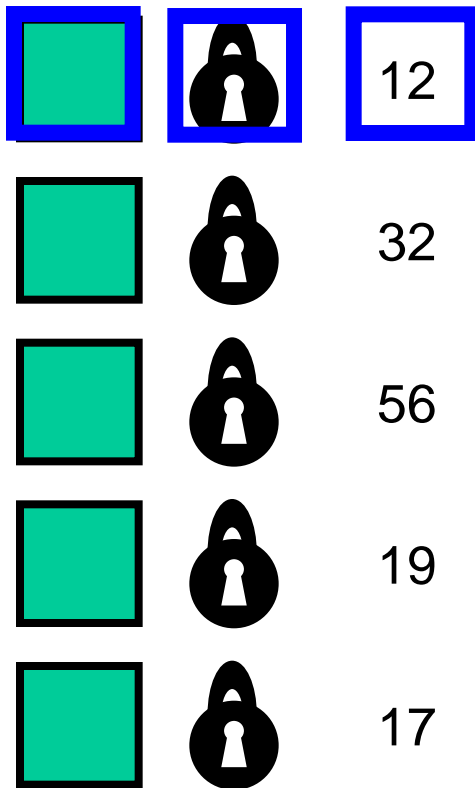
Private Read Version (RV)



Read-Only Transactions

Mem

Locks



Copy version clock to local

Read lock, version #, and

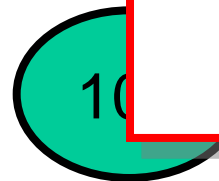
memory address to local version #s

On Commit:

check unlocked &

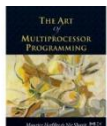
version #s less than

local read clock



Shared Version
Clock

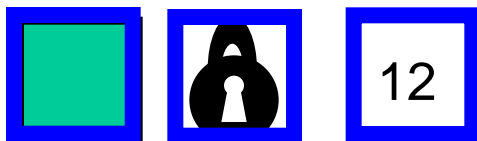
Private Read
Version (RV)



Read-Only Transactions

Mem

Locks

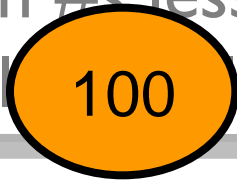
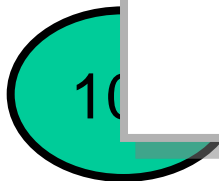


Copy version clock to local

Read lock, version #, and

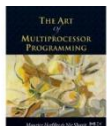
We have taken a snapshot without keeping an explicit read set!

version #s less than local lock



Shared Version Clock

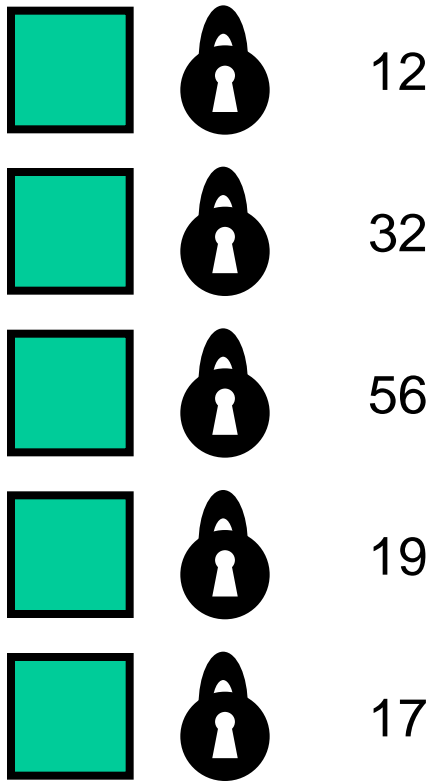
Private Read Version (RV)



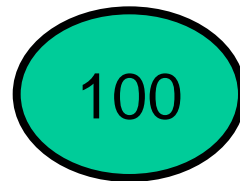
Ordinary (read/write) Transactions

Mem

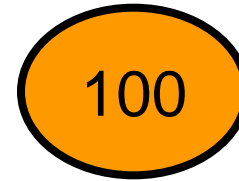
Locks



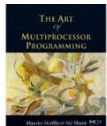
Copy version clock to local read version clock



Shared Version Clock



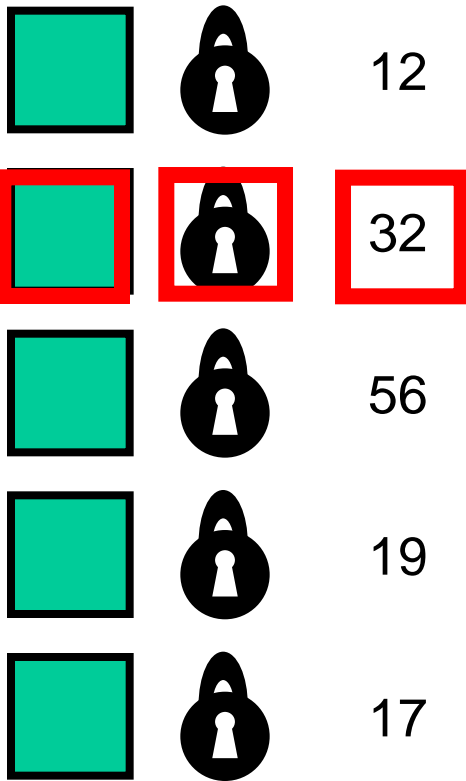
Private Read Version (RV)



Ordinary (read/write) Transactions

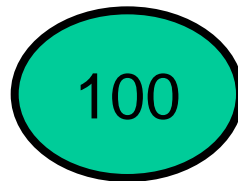
Mem

Locks

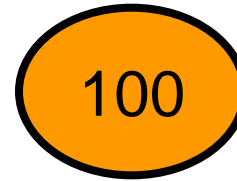


Copy version clock to local read version clock

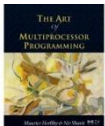
On read/write, check:
Unlocked & version # < RV
Add to R/W set



Shared Version Clock



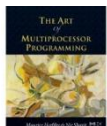
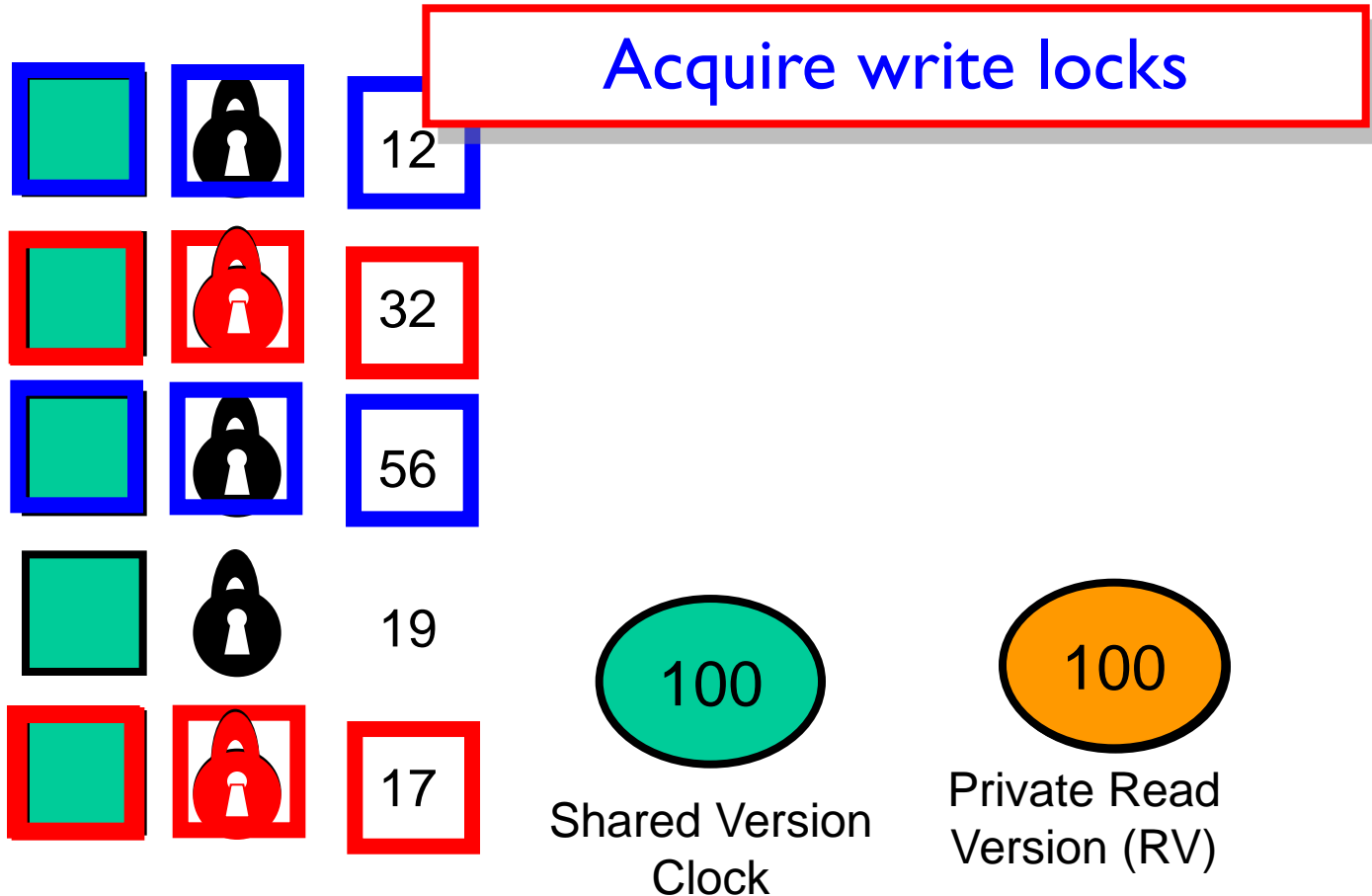
Private Read Version (RV)



On Commit

Mem

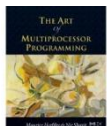
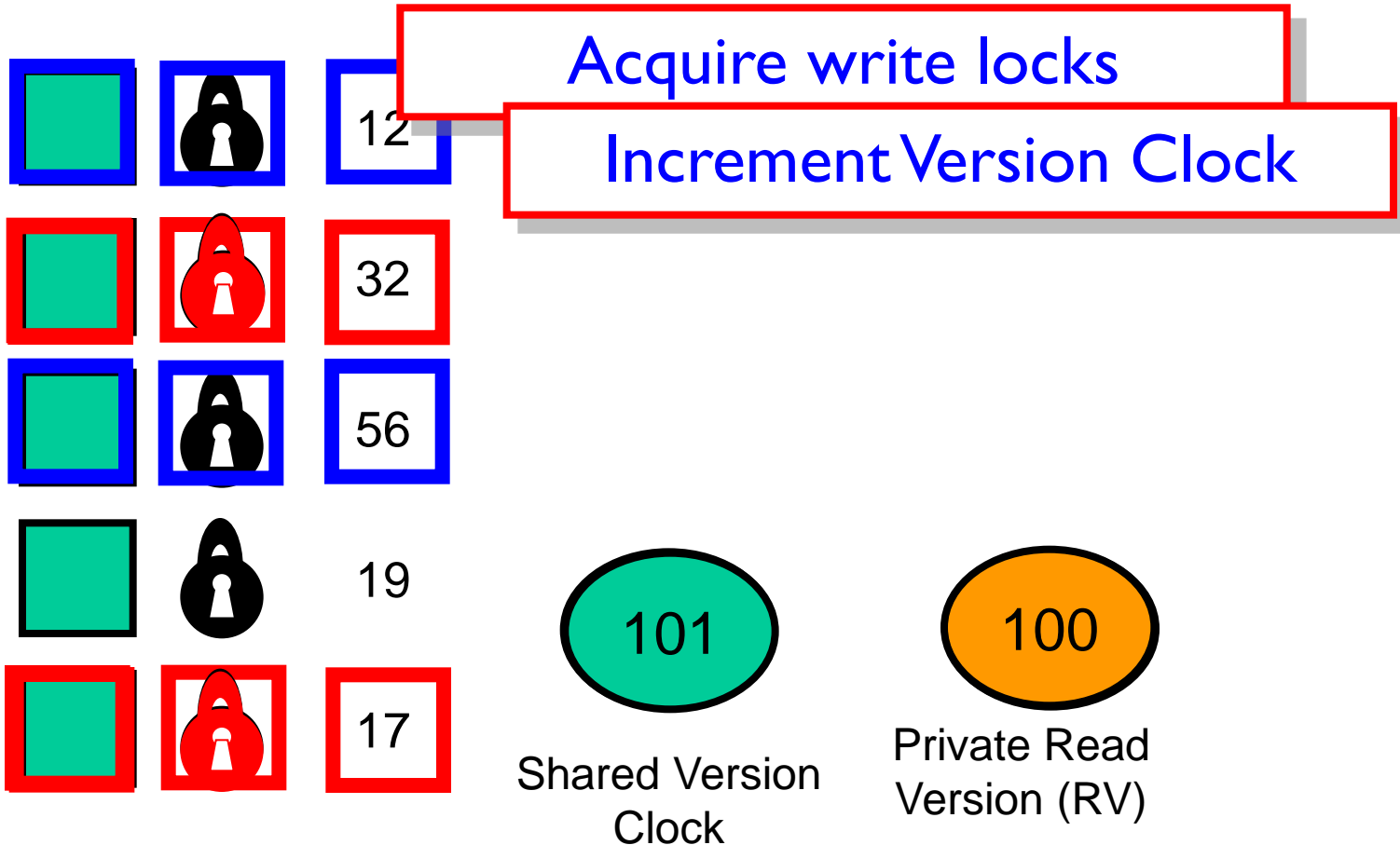
Locks



On Commit

Mem

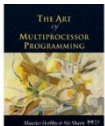
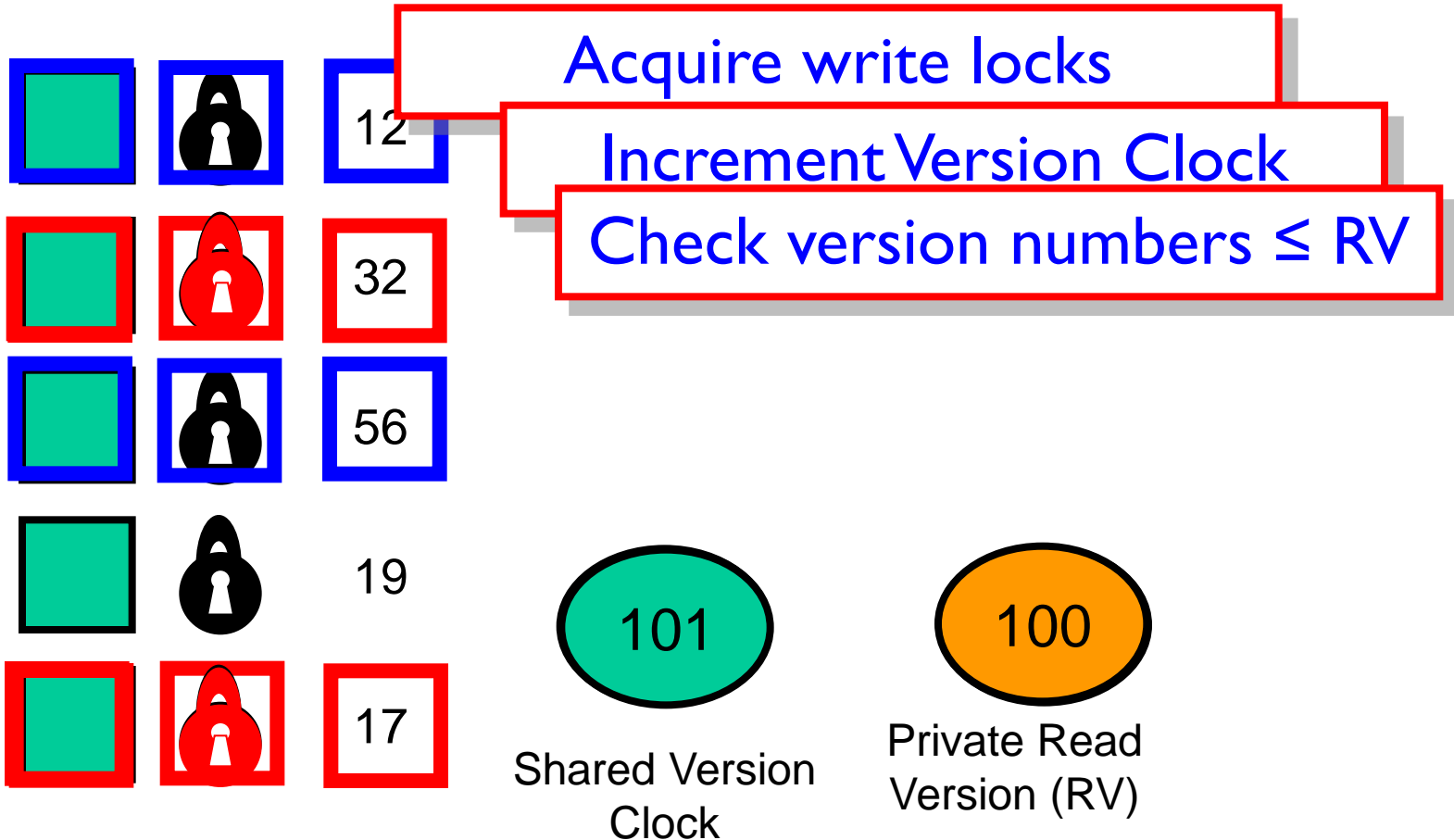
Locks



On Commit

Mem

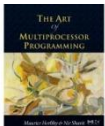
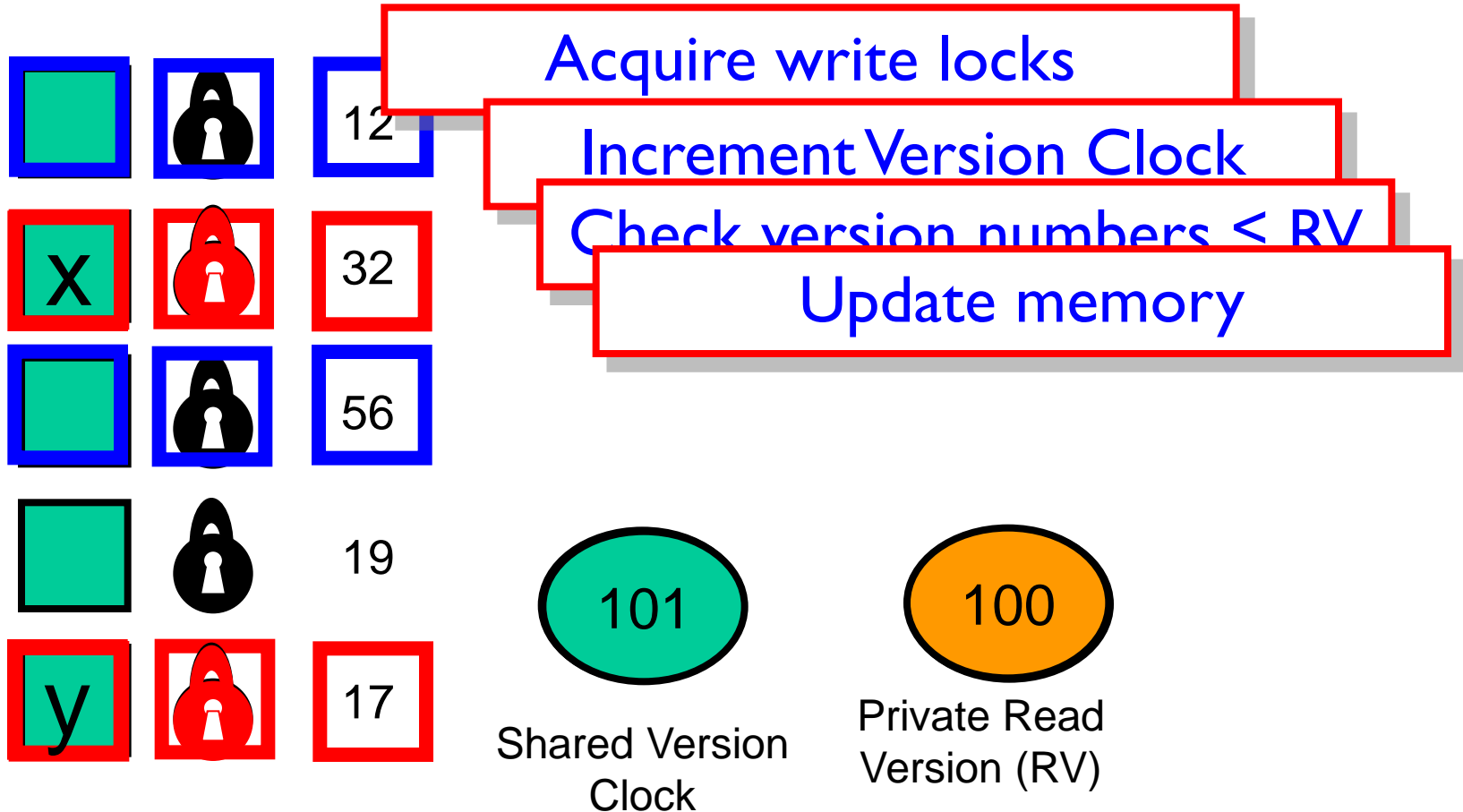
Locks



On Commit

Mem

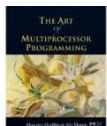
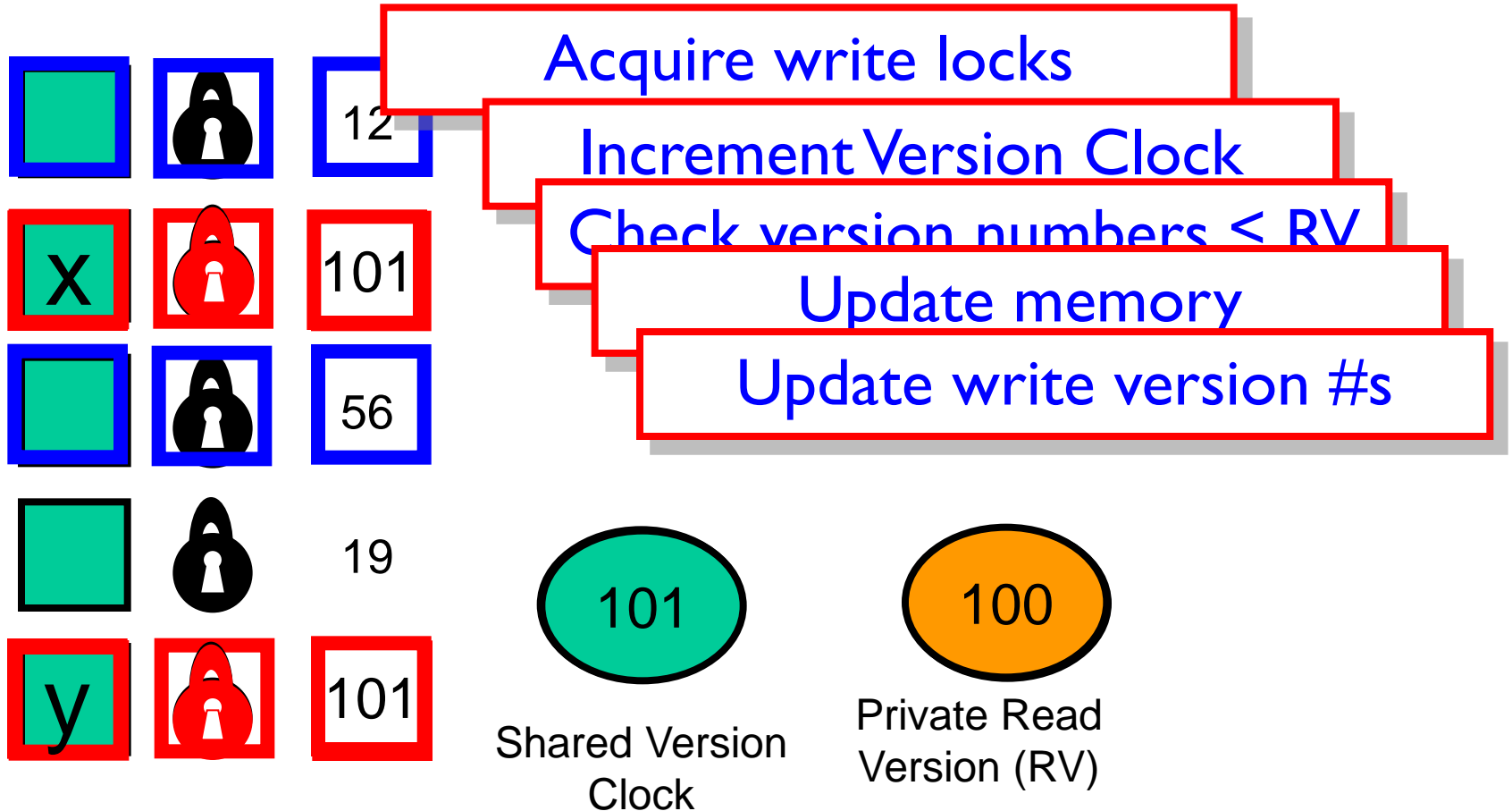
Locks



On Commit

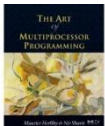
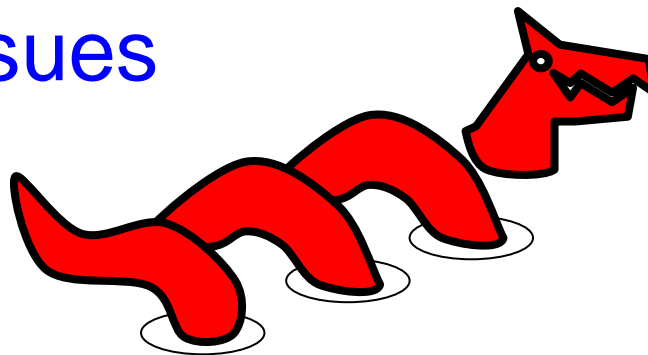
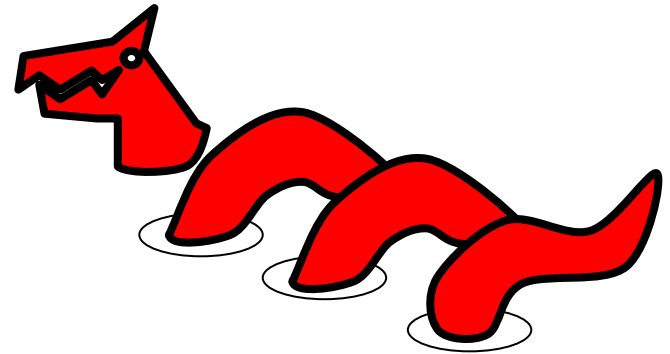
Mem

Locks



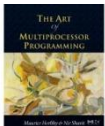
TM Design Issues

- Implementation choices
- Language design issues
- Semantic issues



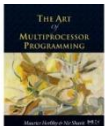
Deferred/Direct Update

- *Deferred*
 - modify private copies & install on commit
 - Commit requires work
 - Consistency easier
- *Direct*
 - Modify in place, roll back on abort
 - Makes commit efficient
 - Consistency harder



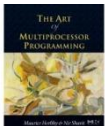
Conflict Detection

- Eager
 - Detect before conflict arises
 - “Contention manager” module resolves
- Lazy
 - Detect on commit/abort
- Mixed
 - Eager write/write, lazy read/write ...



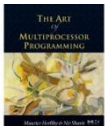
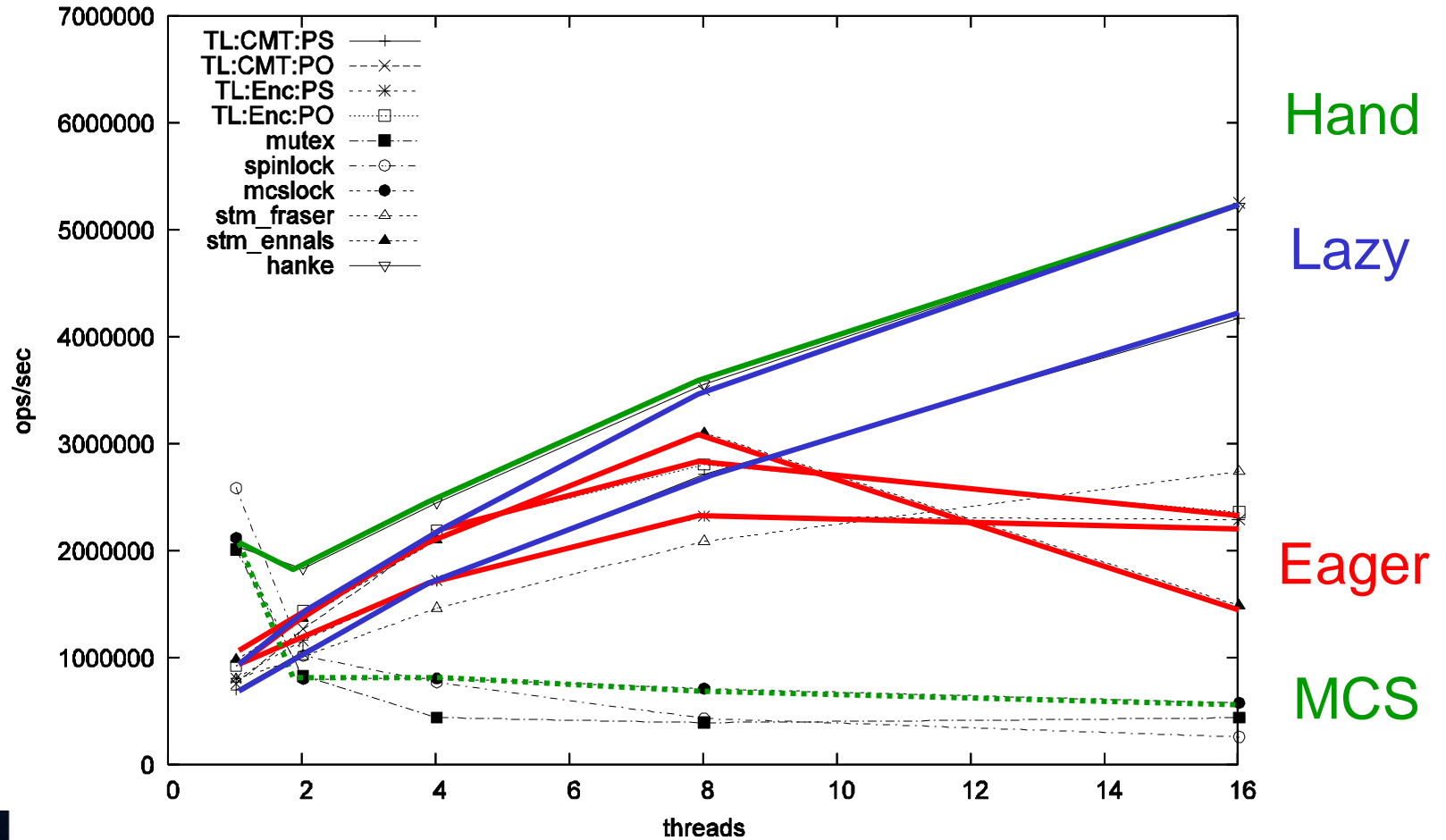
Conflict Detection

- Eager detection may abort transactions that could have committed.
- Lazy detection discards more computation.

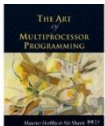
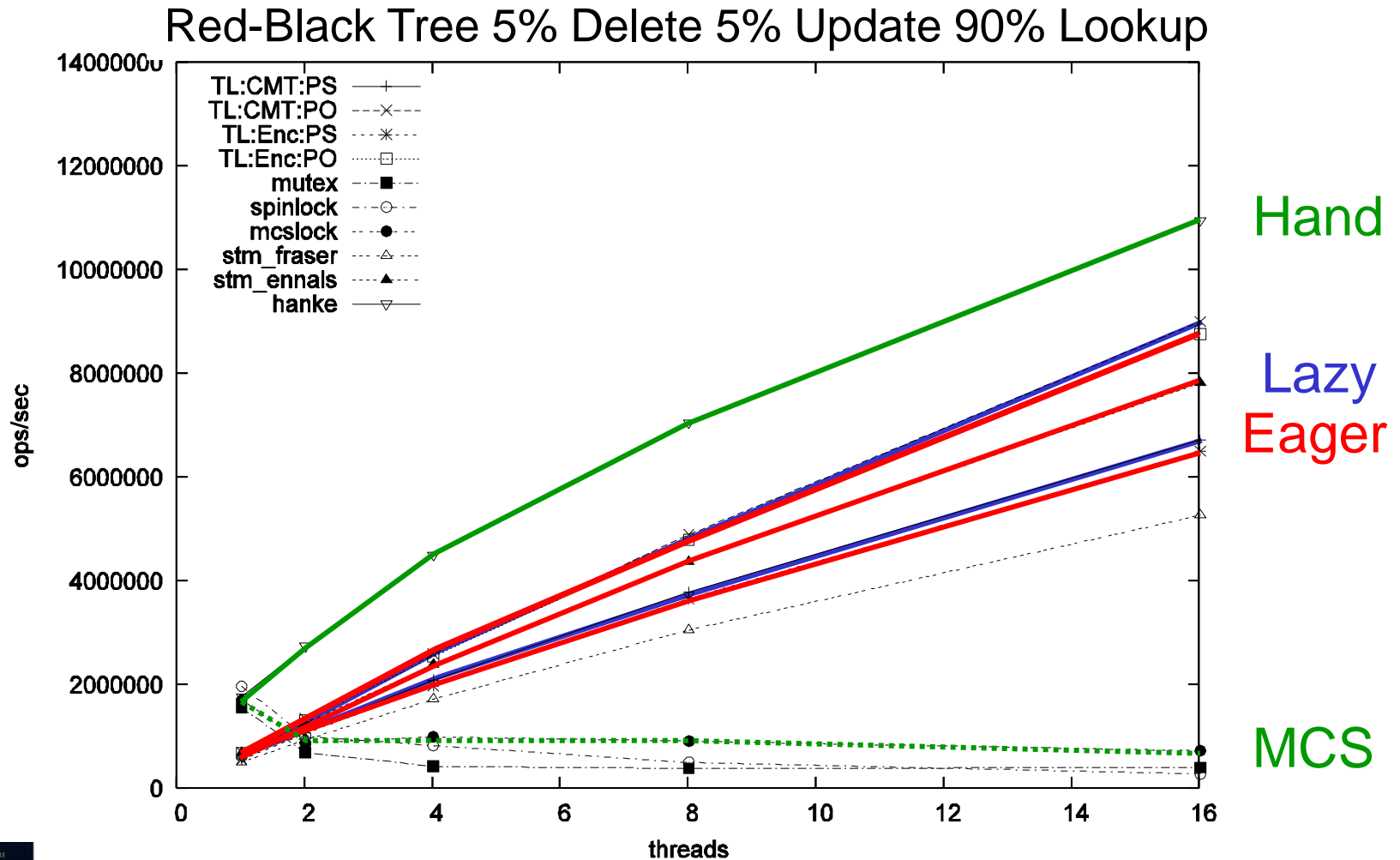


Eager vs lazy conflict detection (high load)

Red-Black Tree 20% Delete 20% Update 60% Lookup

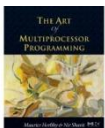


Eager vs lazy conflict detection (high load)



Contention Management & Scheduling

- How to resolve conflicts?
- Who moves forward and who rolls back?
- Lots of empirical work but formal work in infancy



Contention Manager Strategies

- Exponential backoff
- Priority to
 - Oldest?
 - Most work?
 - Non-waiting?
- None Dominates
- But needed anyway



Judgment of Solomon

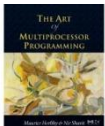
I/O & System Calls?

- Some I/O revocable
 - Provide transaction-safe libraries
 - Undoable file system/DB calls
- Some not
 - Opening cash drawer
 - Firing missile



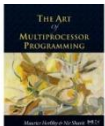
Granularity

- Object
 - managed languages, Java, C#, ...
 - Easy to control interactions between transactional & non-trans threads
- Word
 - C, C++, ...
 - Hard to control interactions between transactional & non-trans threads



I/O & System Calls

- One solution: make transaction irrevocable
 - If transaction tries I/O, switch to irrevocable mode.
- There can be only one ...
 - Requires serial execution
- No explicit aborts
 - In irrevocable transactions



Exceptions

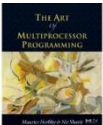


```
int i = 0;
try {
    atomic {
        i++;
        node = new Node();
    }
} catch (Exception e) {
    print(i);
}
```

Exceptions

Throws OutOfMemoryException!

```
int i = 0;
try {
    atomic {
        i++;
        node = new Node();
    }
} catch (Exception e) {
    print(i);
}
```

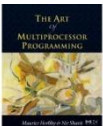


Exceptions

Throws OutOfMemoryException!

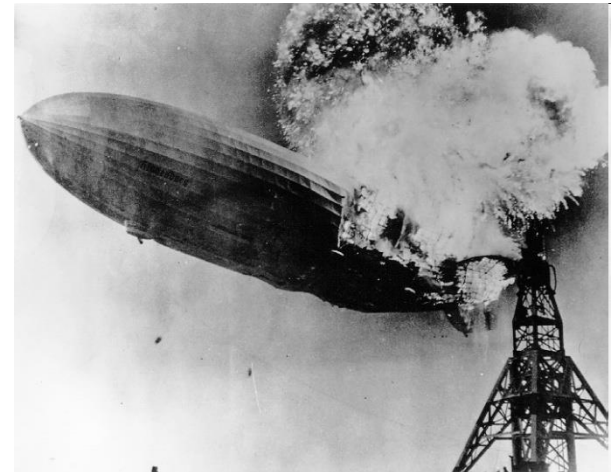
```
int i = 0;
try {
    atomic {
        i++;
        node = new Node();
    }
} catch (Exception e) {
    print(i);
}
```

What is
printed?



Unhandled Exceptions

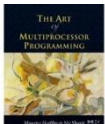
- Aborts transaction
 - Preserves invariants
 - Safer
- Commits transaction
 - Like locking semantics
 - What if exception object refers to values modified in transaction?



Nested Transactions

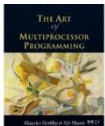
```
atomic void foo() {  
    bar();  
}
```

```
atomic void bar() {  
    ...  
}
```



Nested Transactions

- Needed for modularity
 - Who knew that `cosine()` contained a transaction?
- Flat nesting
 - If child aborts, so does parent
- First-class nesting
 - If child aborts, partial rollback of child only



Hatin' on TM

STM is too inefficient

software transactional memory

- Interaction with code that as a result of either commu
- Livelock, or the system gu

In addition to the intrin:
also implementation-specifi
high transactional overh
nondeterminism introd
tions complicates debuggin
executed and aborted on co
repeatabe behavior. Both o
argument for transactions, t
implementations.
Given all these issues, wi
yet matured to the point wi

→ why is it only a research toy?

degradation when overflow occurs, and proposals for
managing overflows (for example, signatures¹⁷) incur false
positives that add complexity to the programming model.
Therefore, from an industrial perspective, HTM designs
provide more benefits for the cost on a more
workloads (with varying transactional char-

Hatin' on TM

The problem with STM: your languages still suck

Published by [Brian](#) at 12:13 PM under [To Be Categori](#)

So, I'm reading [Brandon Werner's post](#) caught on yet. There are three Act, allowing many to Java, C#,

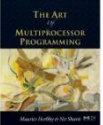
- Interaction with code that as a result of either commu a requirement barring spec
- Livelock, or the system gu make progress even in the f
- In addition to the intrin: implementation-specifi
- Transactional overh for excluding F
- Mechanism introd debuggin
- on co
- at tc

Requires radical change in programming style

→ why is it o

degradation when overflow - managing overflows (for exam, positives that add complexity to L. therefore, from an industrial persp provide more benefits for the c. workloads (with varying tran.

Software Transactional Memory has succeed in implementing STM for "mainstream" languages l that make it a CS Researcher Full Employment



Hatin' on TM

Wrong Programming Model

Erlang-style shared nothing only true path to salvation

→ why is it ok

degradation when overflow -
managing overflows (for exam,
positives that add complexity to L
therefore, from an industrial persp
provide more benefits for the c
workloads (with varying tran



Hatin' on TM

Monday Nov 03, 2008

Concurrency's Shysters

For as long as I've been in computing, the subject of concurrency has always induced a certain amount of trepidation. It was coming up, the name of the apocalypse was symmetric multiprocessing -- and I was a software engineer who -- despite the fact that I had worked for different computer companies -- confidently asserted that it would not scale beyond 8 CPUs. Needless to say, I was wrong.

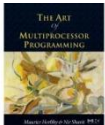
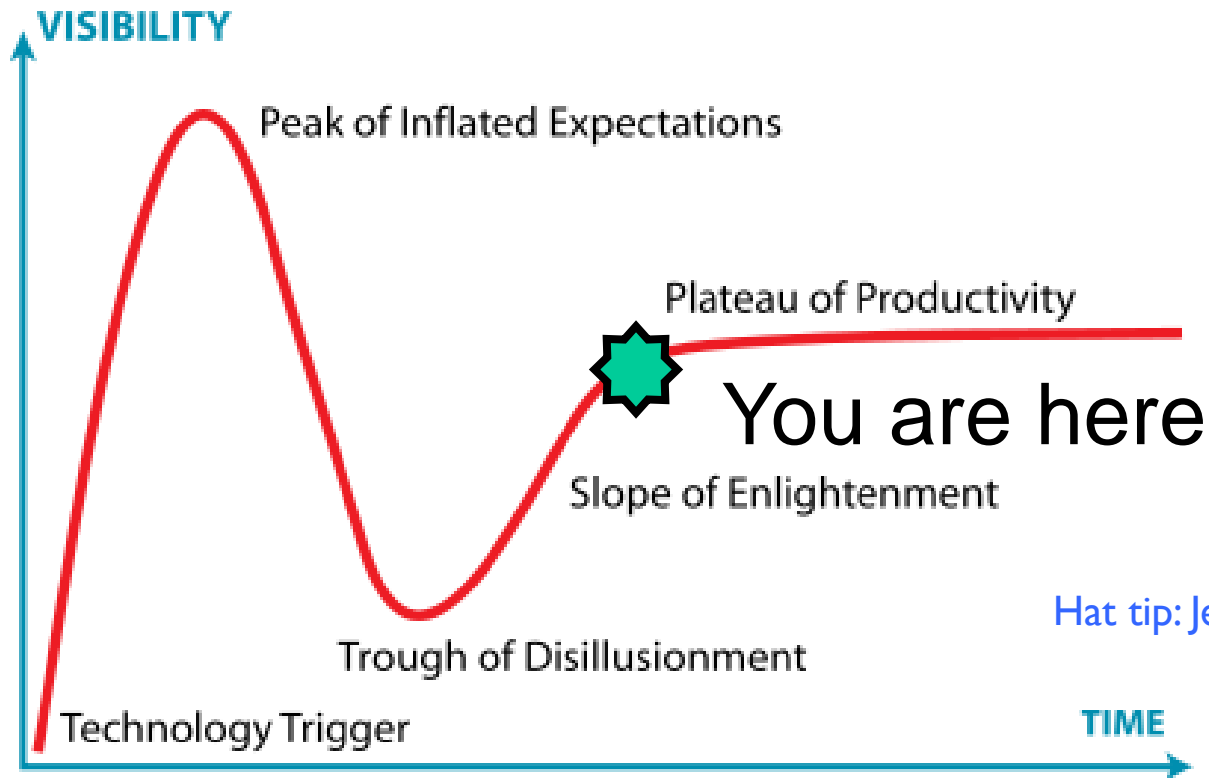
There is nothing wrong with what we do today.

→ why is it

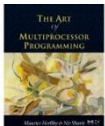
degradation when overflow
managing overflows (for exam
positives that add complexity to
therefore, from an industrial persp
provide more benefits for the c
workloads (with varying tran



Gartner Hype Cycle



Thanks ! תודה



Overview

- Building shared memory data structures
 - Lists, queues, hashtables, ...
- Why?
 - Used directly by applications (e.g., in C/C++, Java, C#, ...)
 - Used in the language runtime system (e.g., management of work, implementations of message passing, ...)
 - Used in traditional operating systems (e.g., synchronization between top/bottom-half code)
- Why not?
 - Don't think of "threads + shared data structures" as a default/good/complete/desirable programming model
 - It's better to have shared memory and not need it...

Different techniques for different problems

