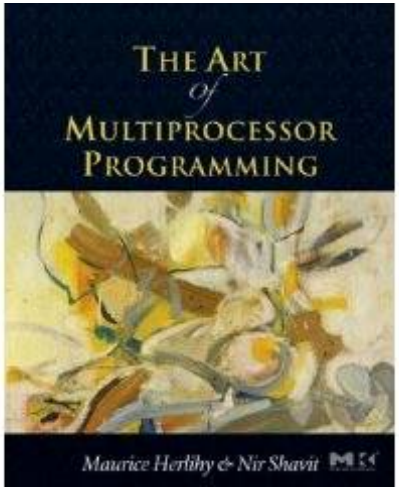
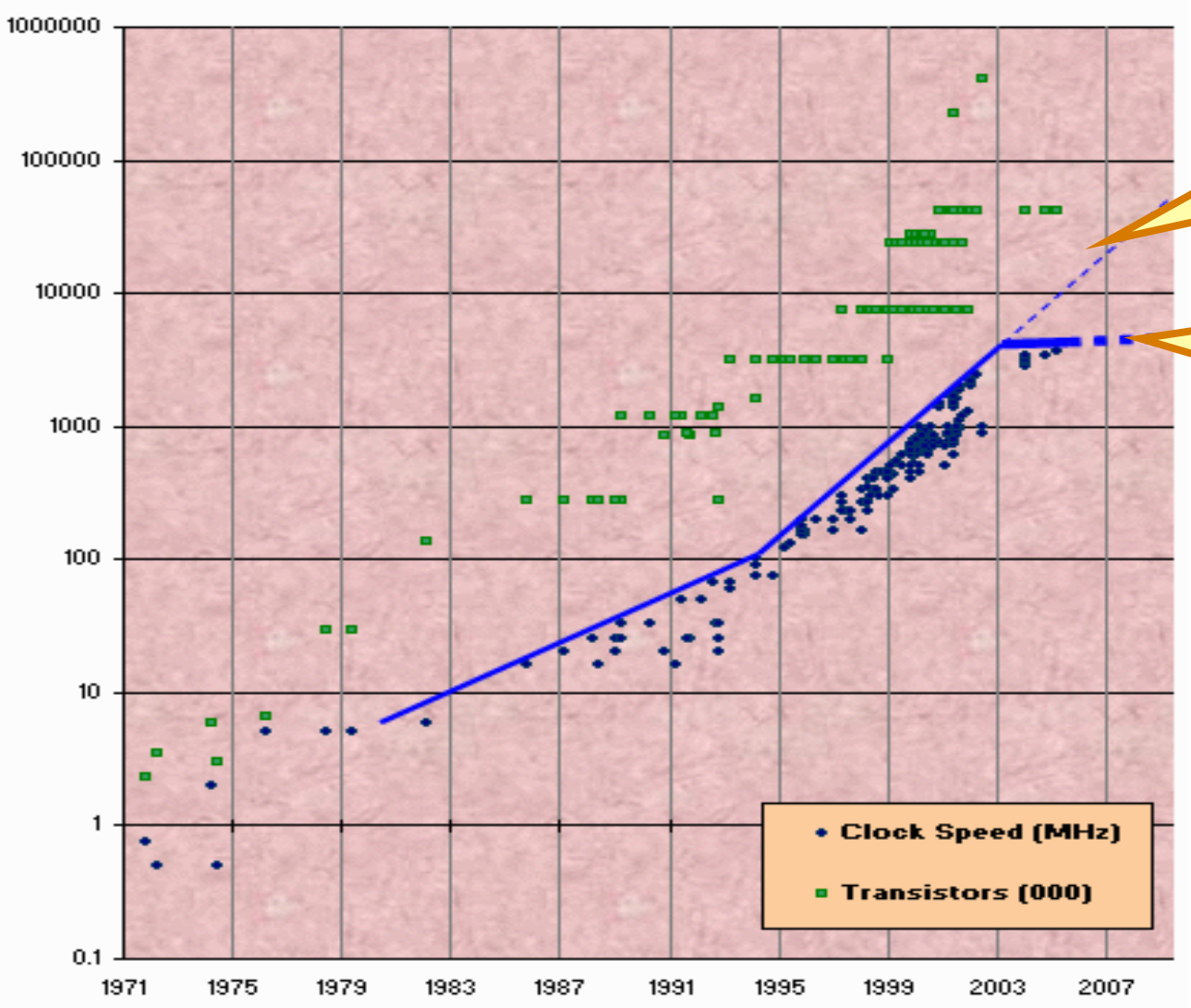
Transactional Memory



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

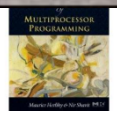
Moore's Law



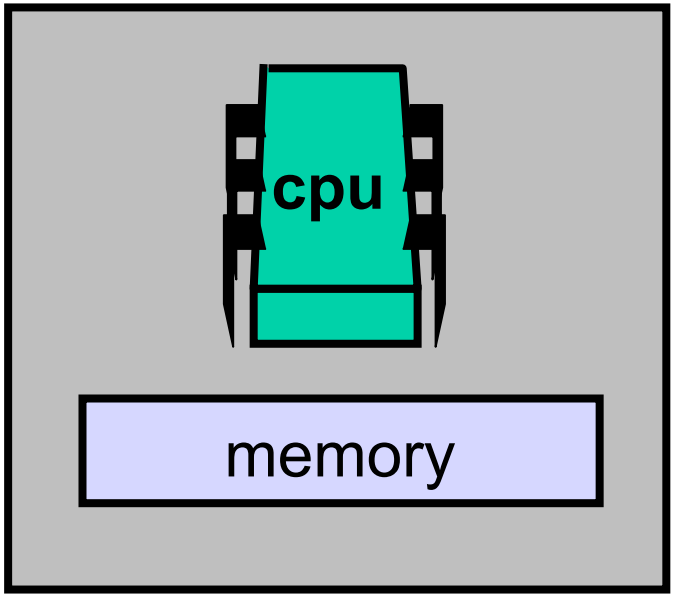
Transistor count still rising

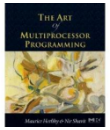
Clock speed flattening sharply

Moore's Law (in practice)

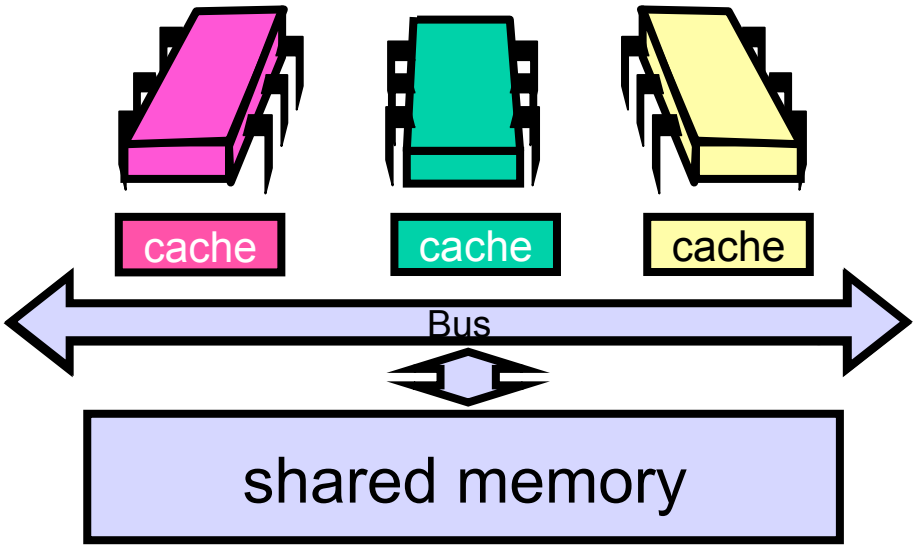


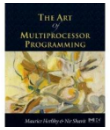
Nearly Extinct: the Uniprocesor





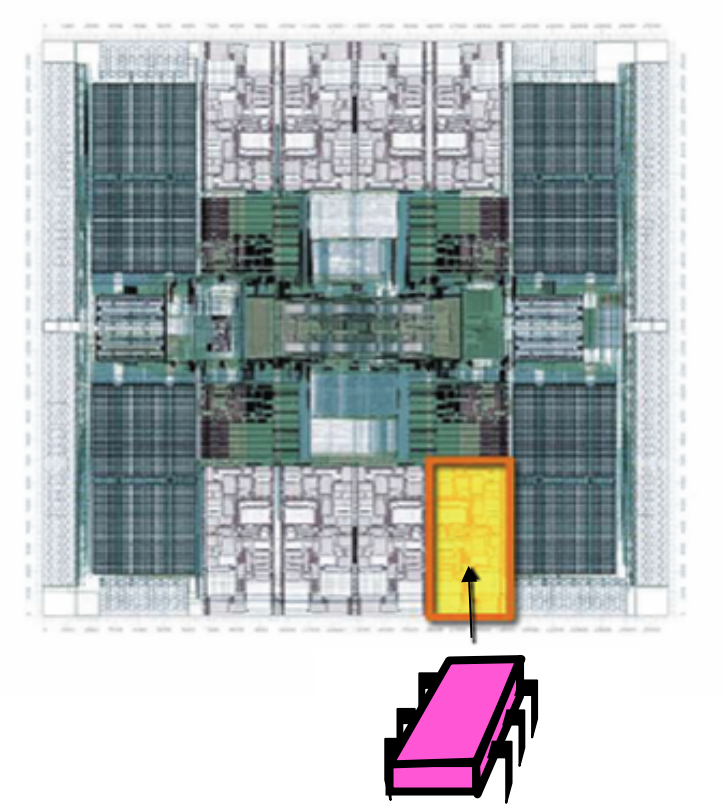
Endangered: The Shared Memory Multiprocessor (SMP)



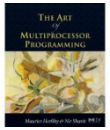


The New Boss: The Multicore Processor (CMP)

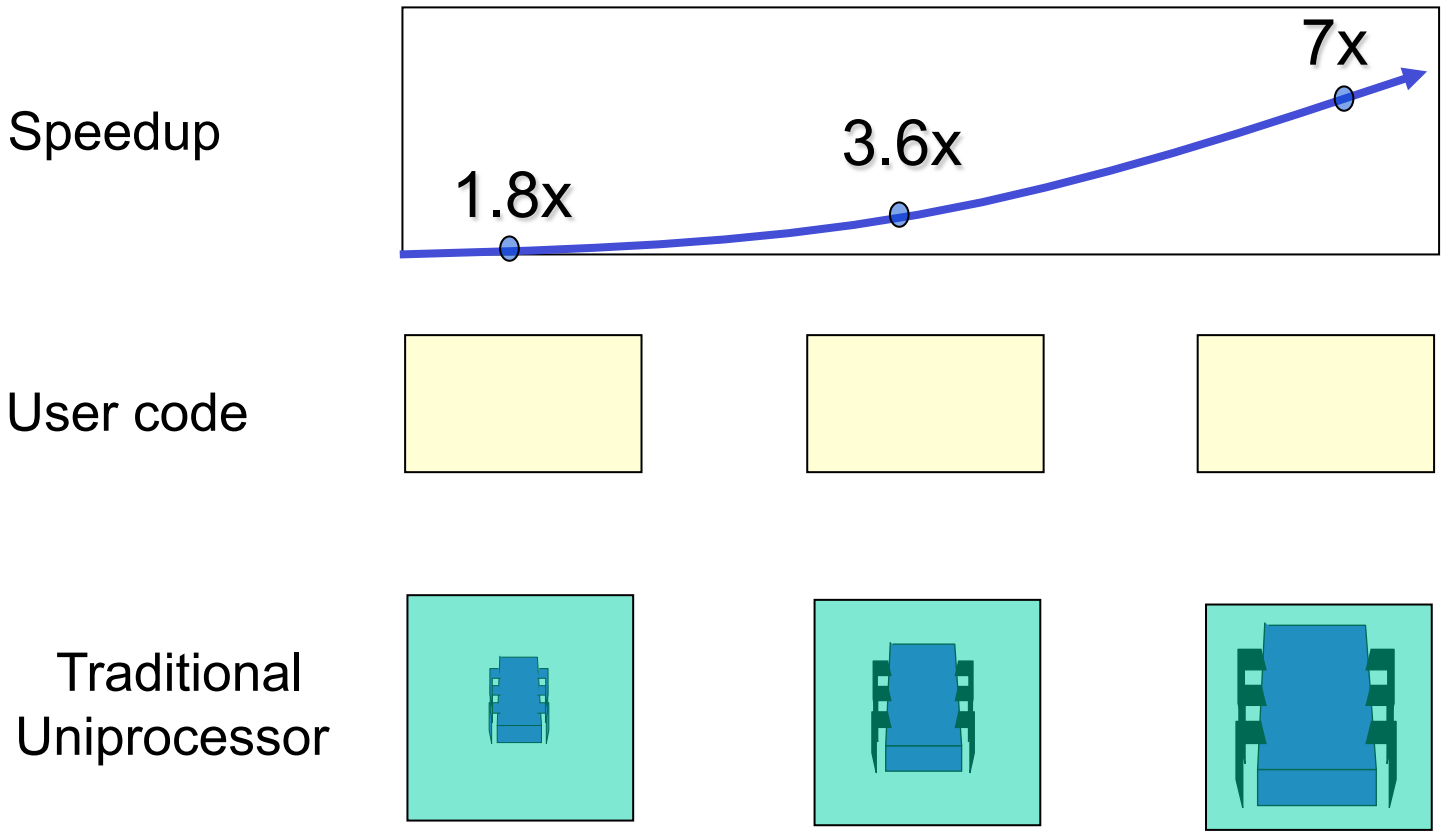
All on the same chip

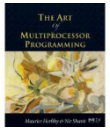


Sun T2000 Niagara

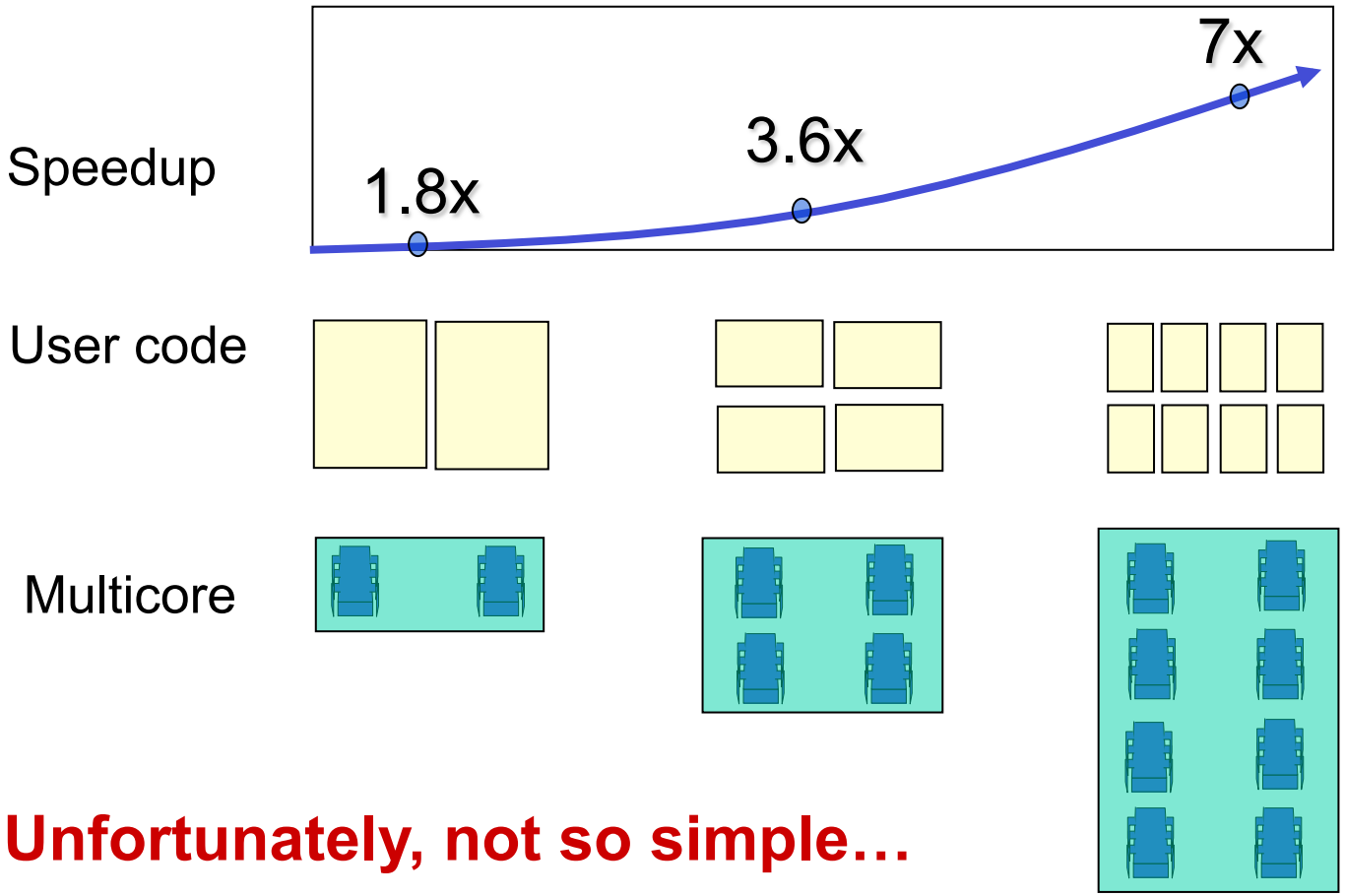


Traditional Scaling Process

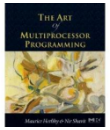




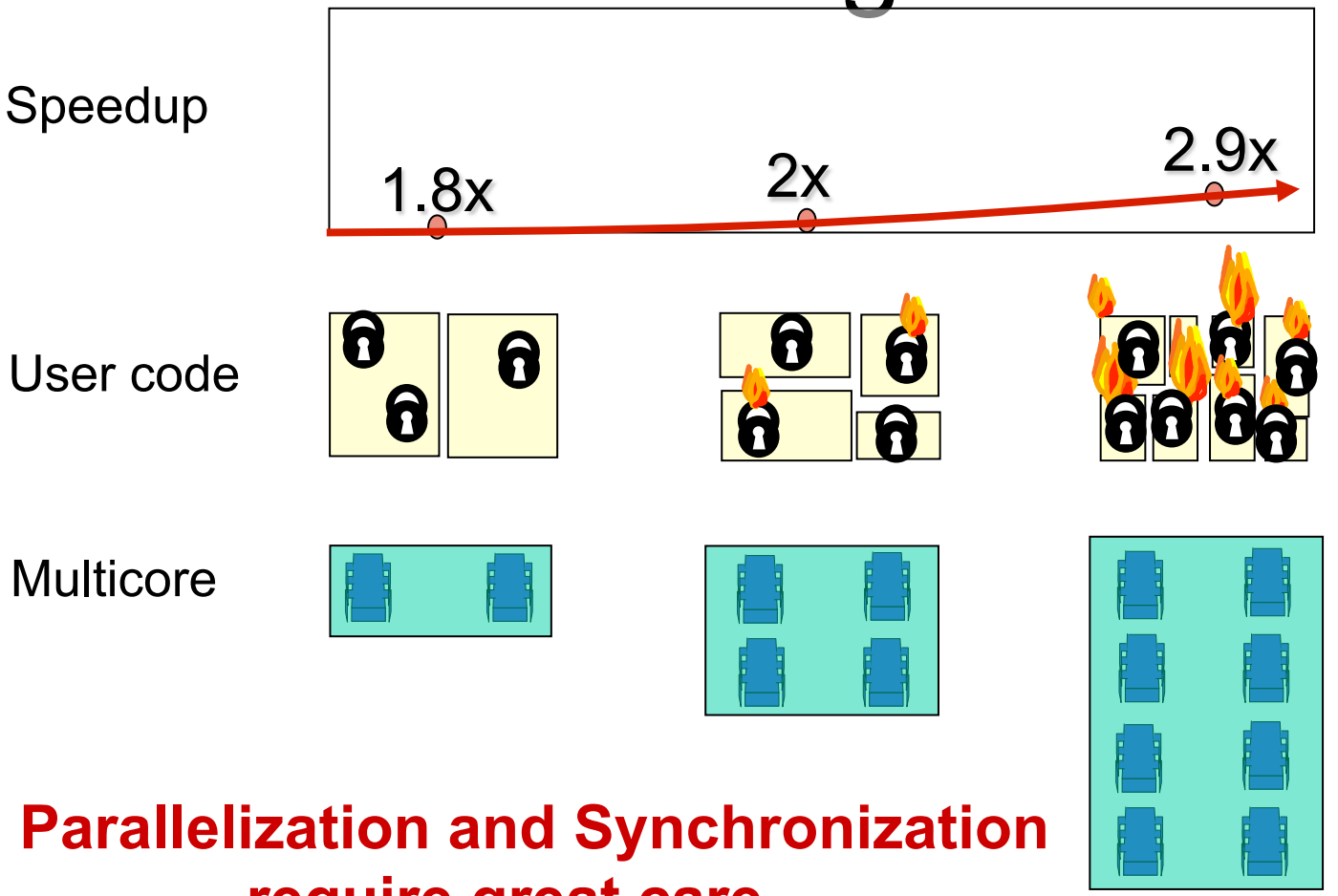
Ideal Scaling Process



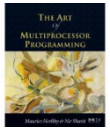
Unfortunately, not so simple…



Actual Scaling Process

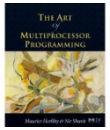


Parallelization and Synchronization require great care…



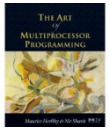
Amdahl's Law

$$\text{Speedup} = \frac{\text{1-thread execution time}}{\textit{n}\text{-thread execution time}}$$



Amdahl's Law

$$\text{Speedup} = \frac{1}{1+p+p/n}$$

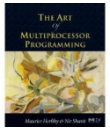


Amdahl's Law

Speedup=

$$1/1+p+p/n$$

parallel fraction



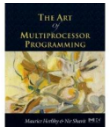
Amdahl's Law

parallel fraction

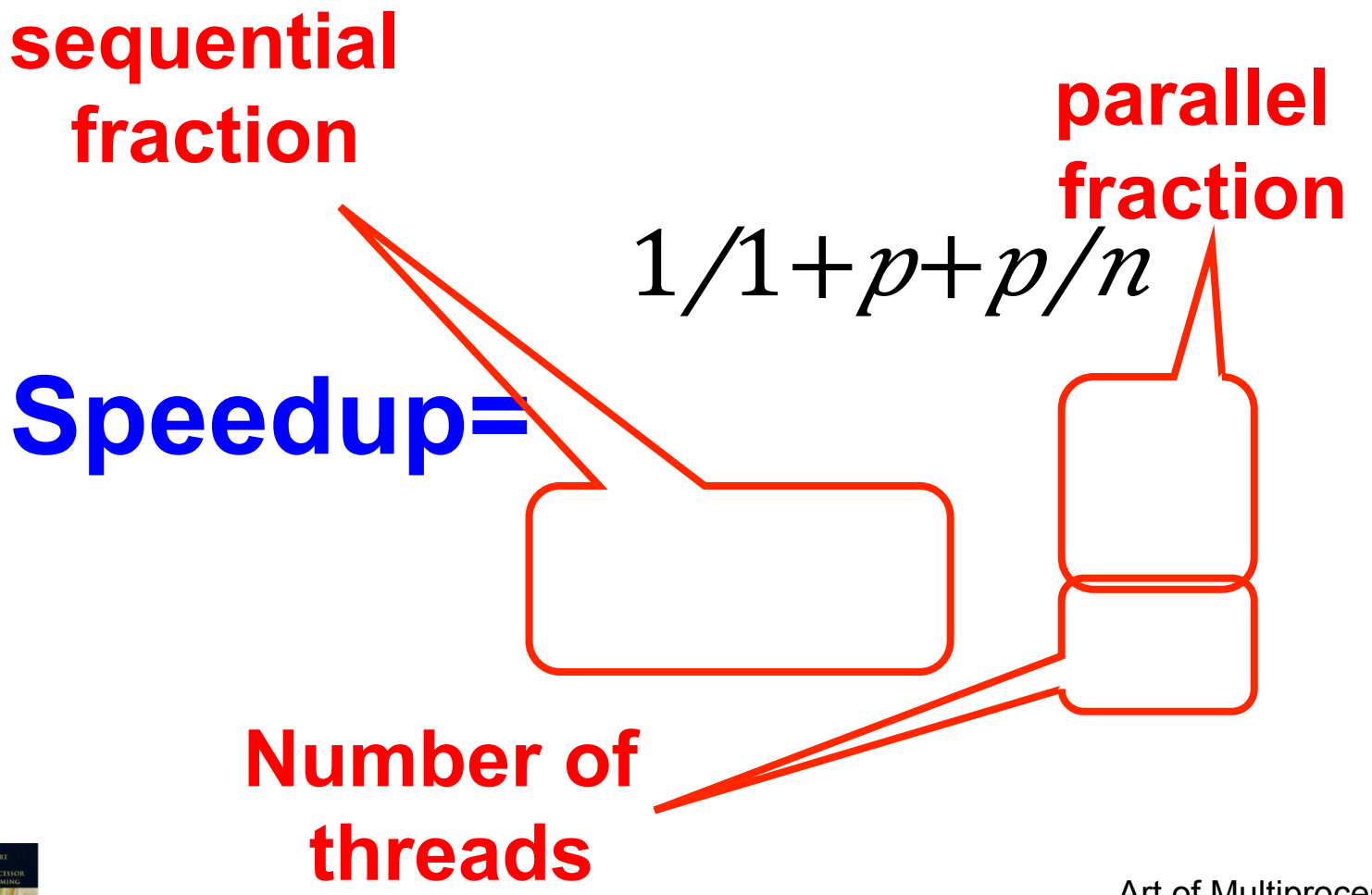
$$1/1+p+p/n$$

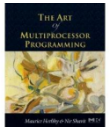
Speedup=

Number of threads



Amdahl's Law


sequential fraction
parallel fraction
$1/1+p+p/n$
Speedup=
Number of threads




Amdal’s Law
Bad synchronization ruins everything

Example

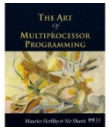
You buy a 10-core machine …

Your application is:

60% concurrent

40% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine …

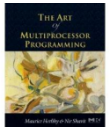
Your application is:

60% concurrent

40% sequential

How close to a 10-fold speedup?

$$1/1-0.6-0.6/10 = 2$$



Example

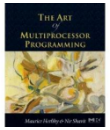
You buy a 10-core machine …

Your application is:

80% concurrent

20% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine …

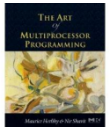
Your application is:

80% concurrent

20% sequential

$$1/1-0.8-0.8/10 = 3$$

How close to a 10-fold speedup?



Example

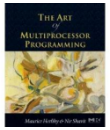
You buy a 10-core machine …

Your application is:

90% concurrent

10% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine …

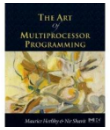
Your application is:

80% concurrent

20% sequential

$$1/1-0.9-0.9/10 = 5$$

How close to a 10-fold speedup?



Example

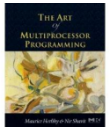
You buy a 10-core machine …

Your application is:

99% concurrent

01% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine …

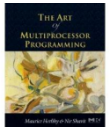
Your application is:

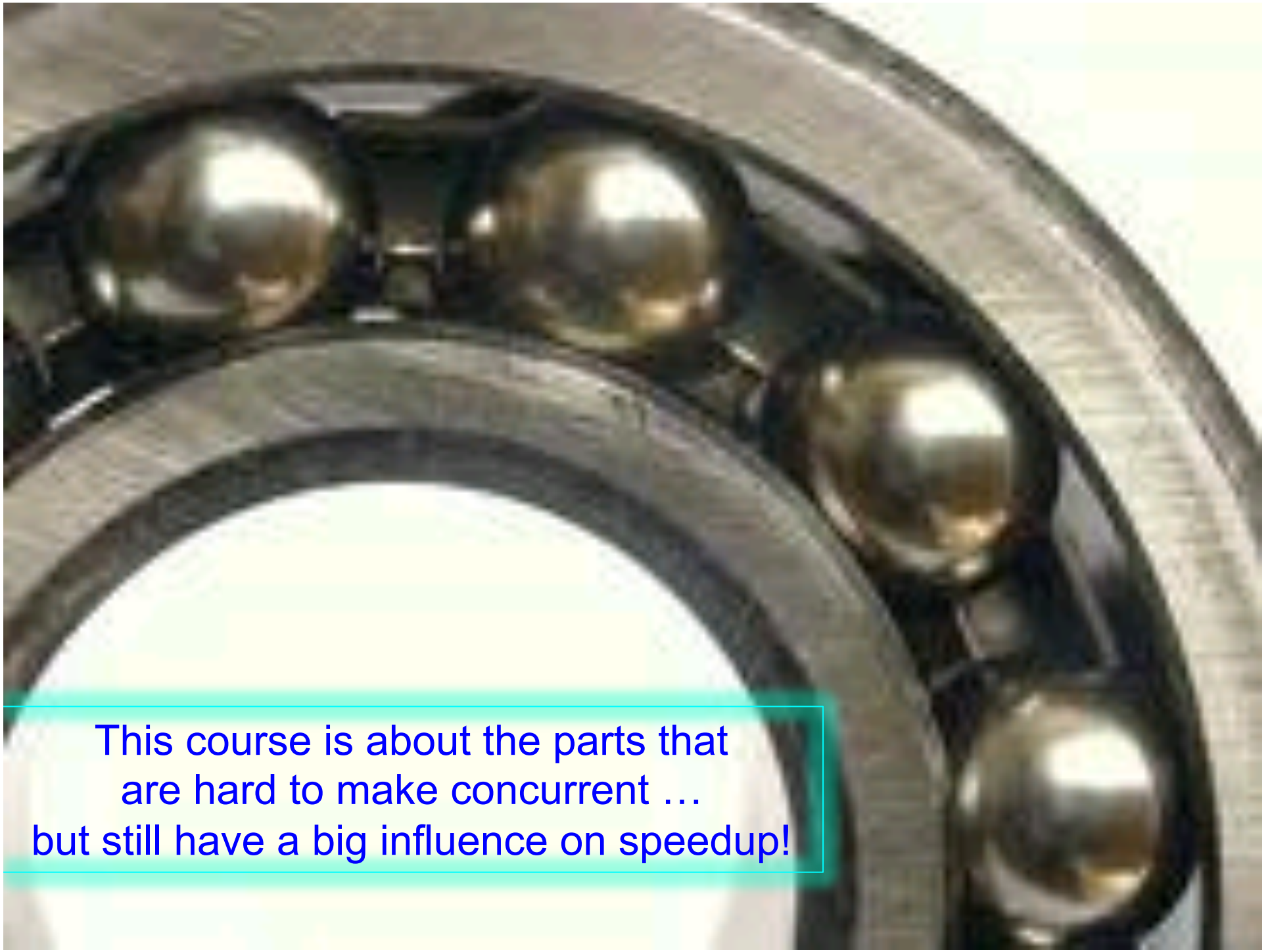
80% concurrent

20% sequential

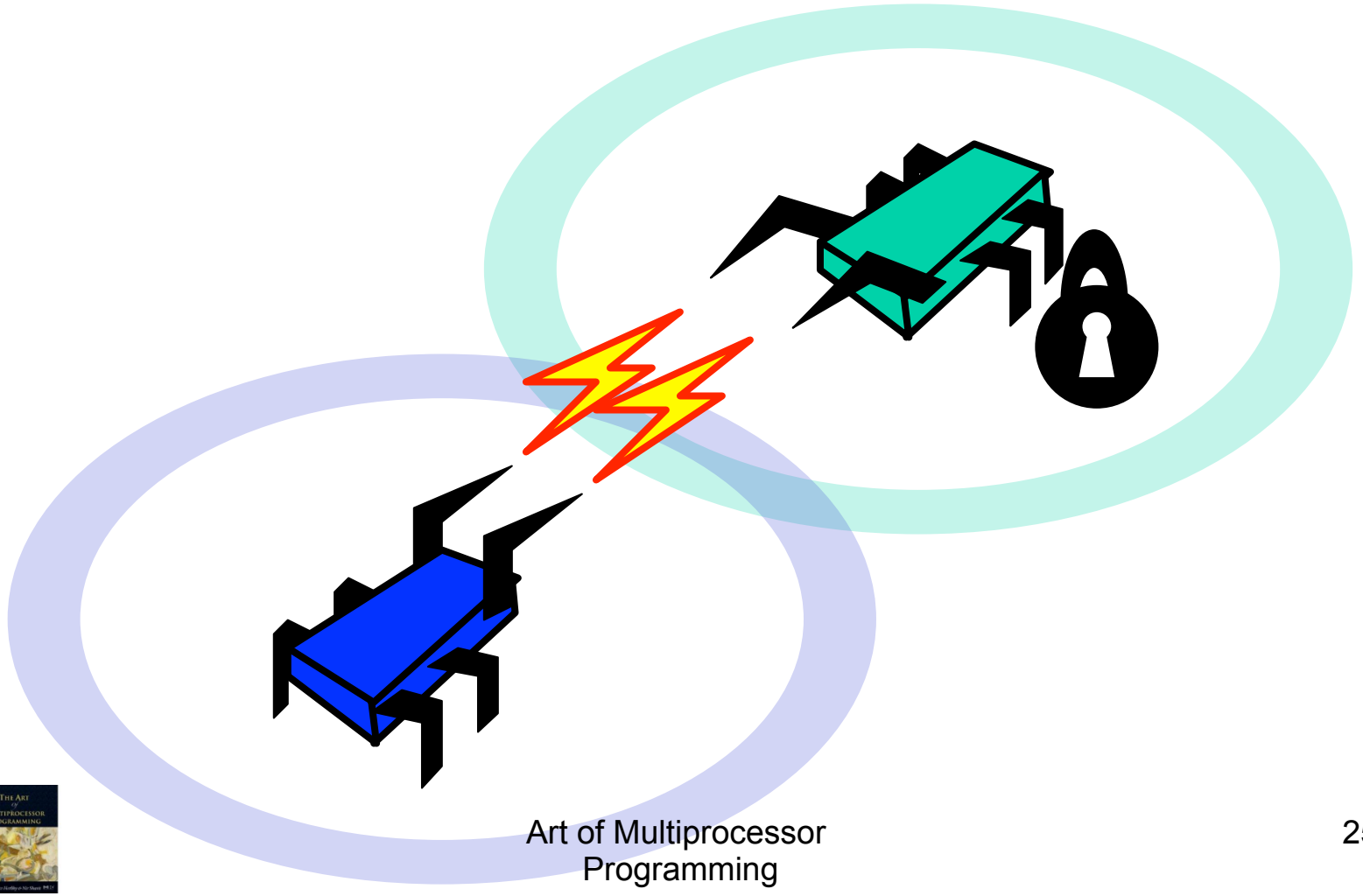
$$1/1-0.99-0.99/10 =$$

How close to a 10-fold speedup?

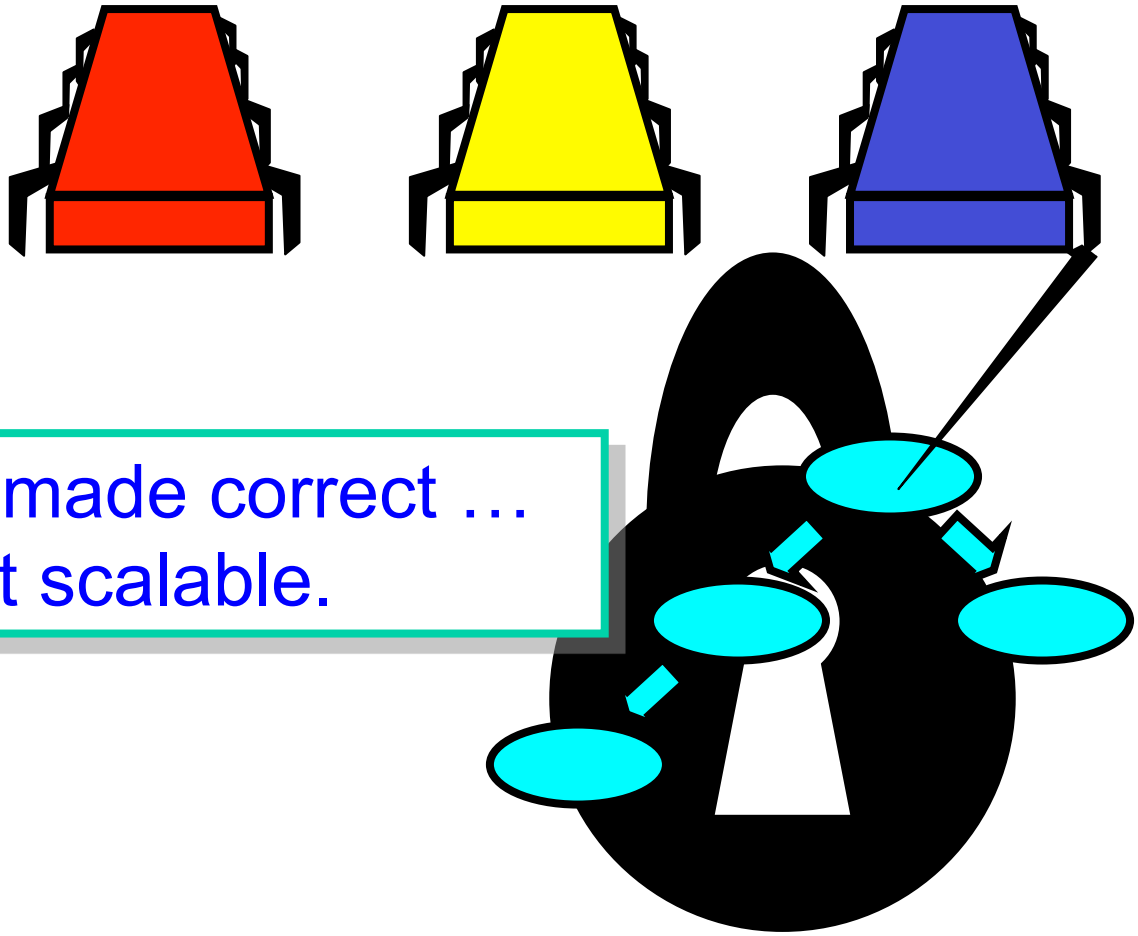



This course is about the parts that
are hard to make concurrent …
but still have a big influence on speedup!

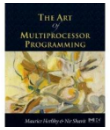
Locking



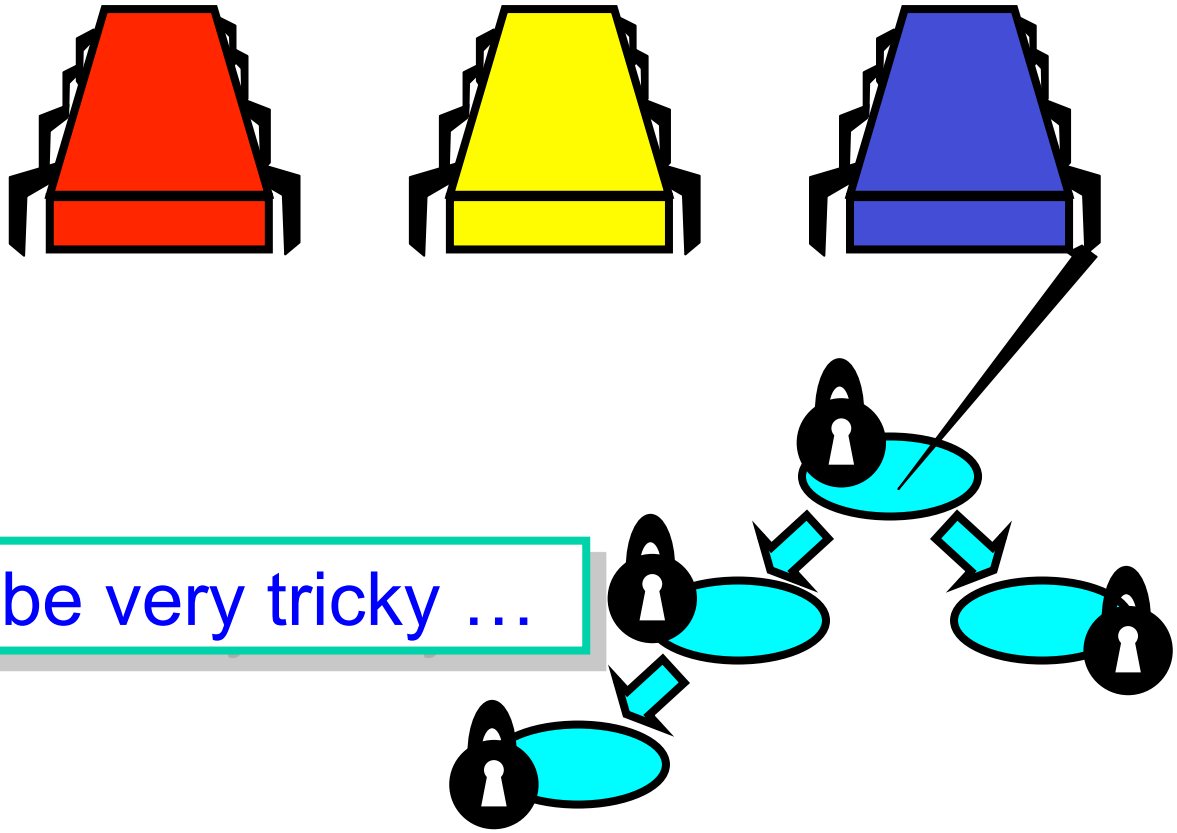
Coarse-Grained Locking



Easily made correct …
But not scalable.



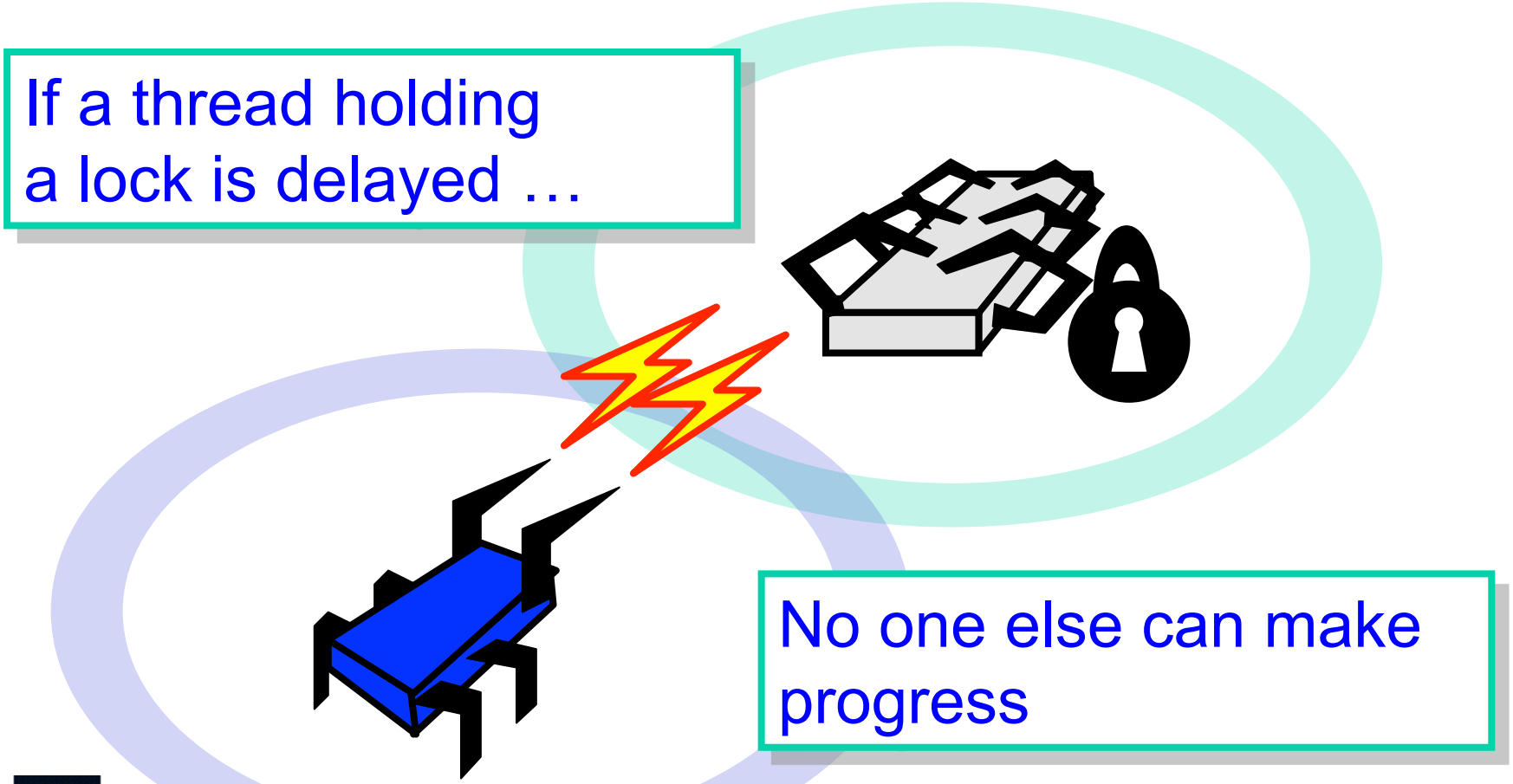
Fine-Grained Locking



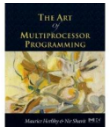
Can be very tricky …

Locks are not Robust

If a thread holding
a lock is delayed …



No one else can make
progress



Locking Relies on Conventions

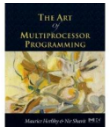
Relation between …

Lock data and object data

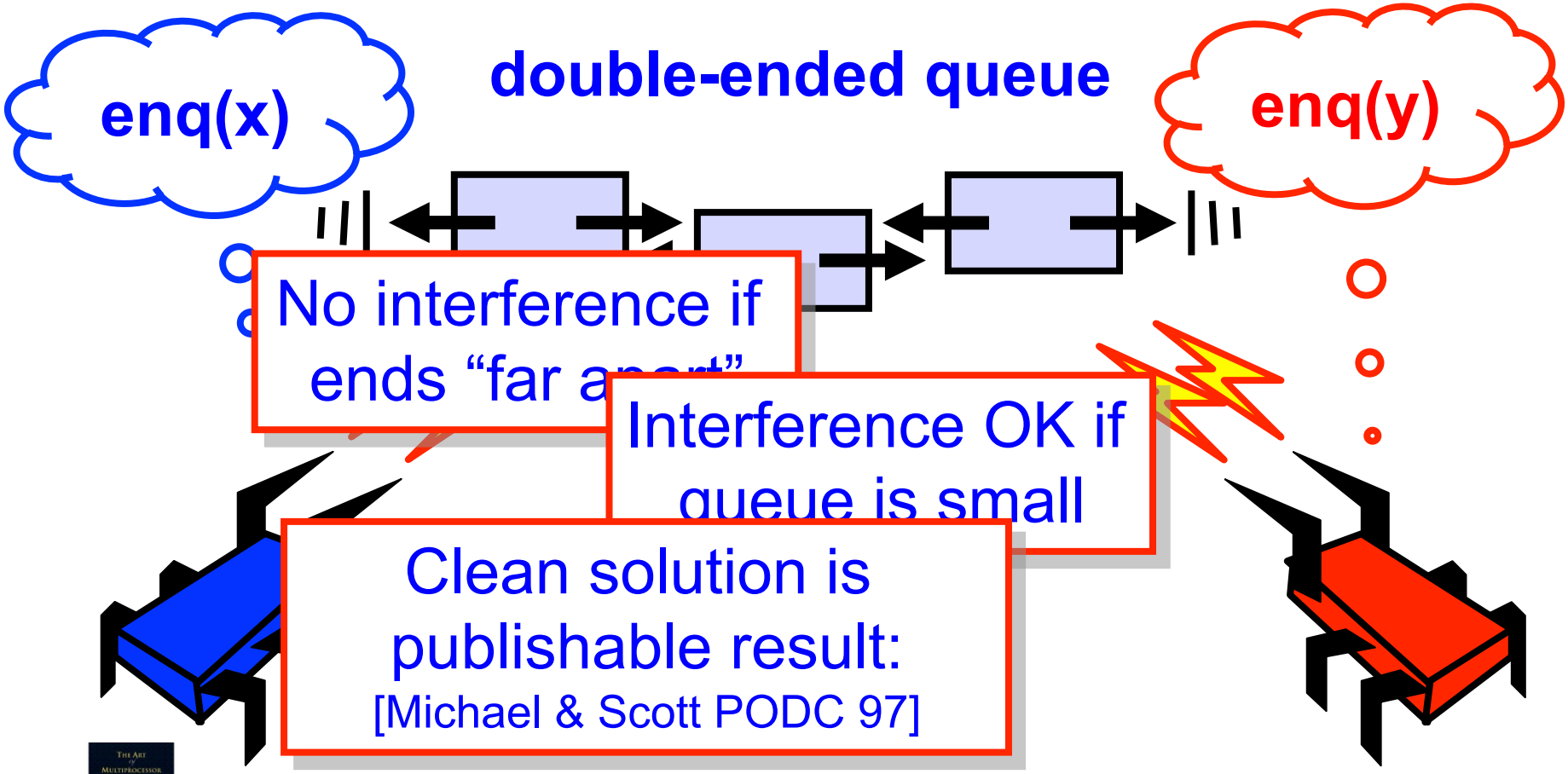
Exists only in programmer's

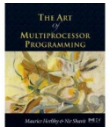
Actual comment from Linux Kernel
(hat tip: Bradley Kuszmaul)

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

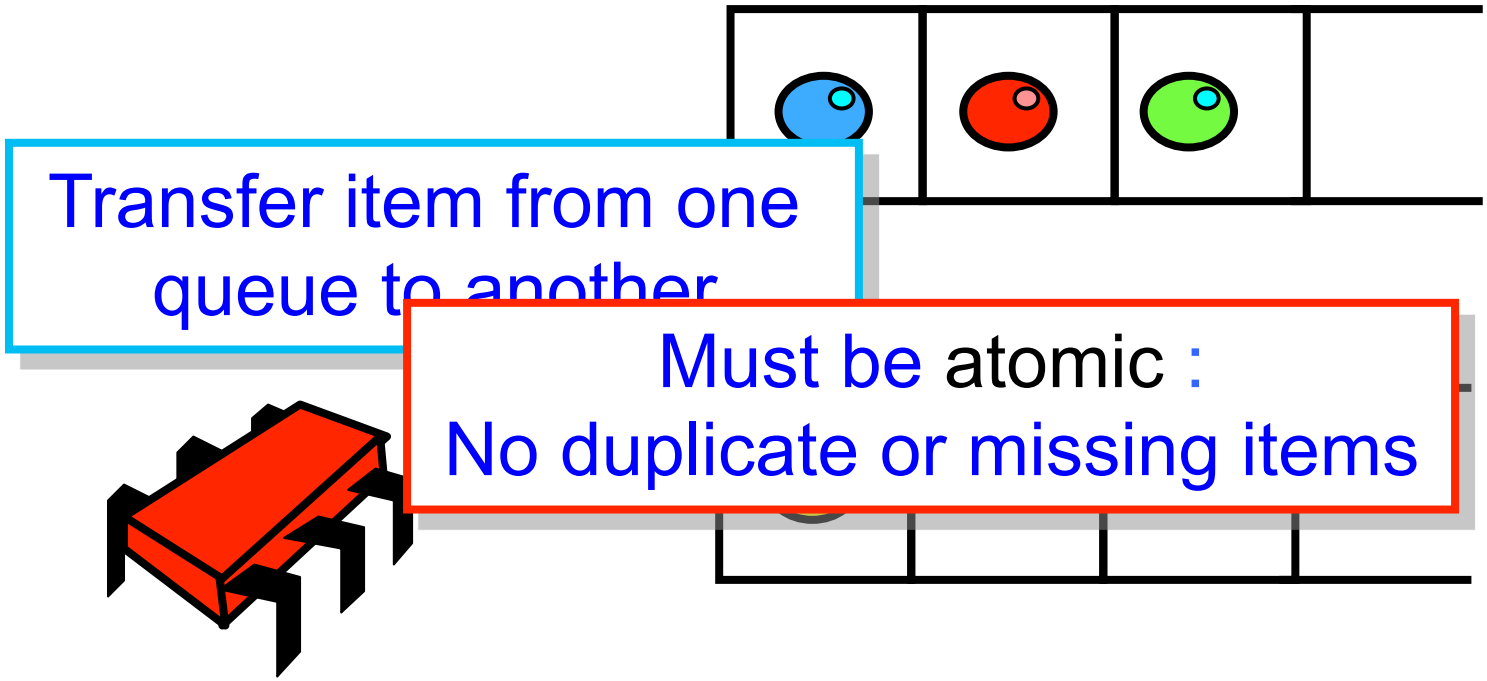


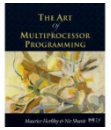
Simple Problems are hard



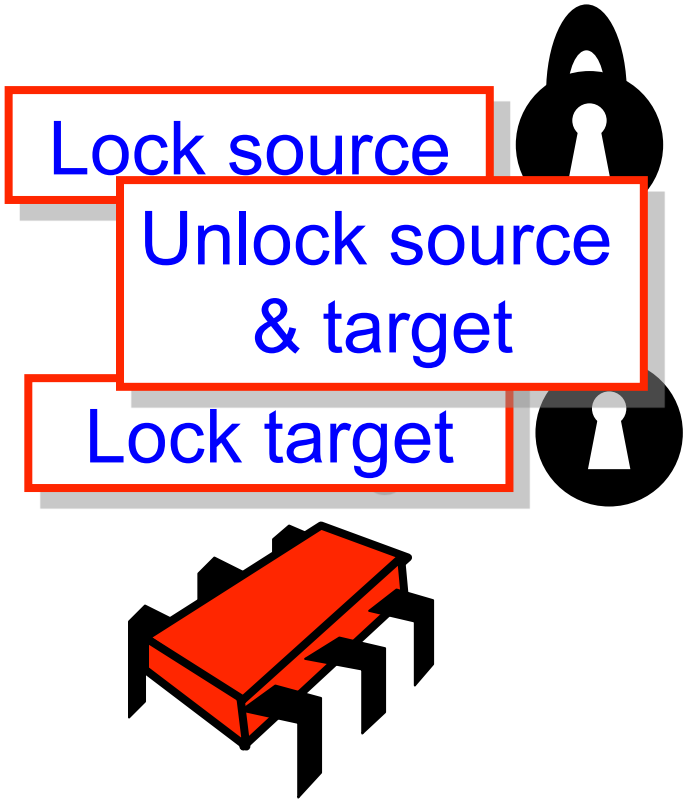


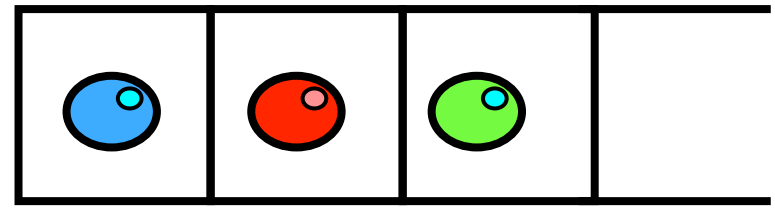
Locks Not Composable

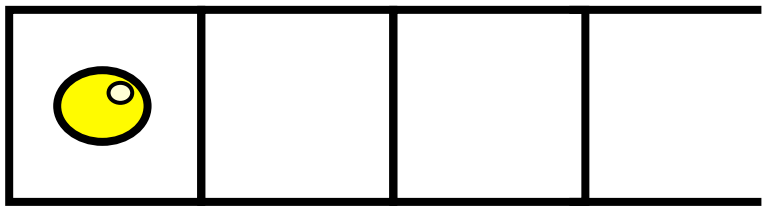


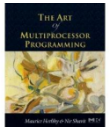


Locks Not Composable

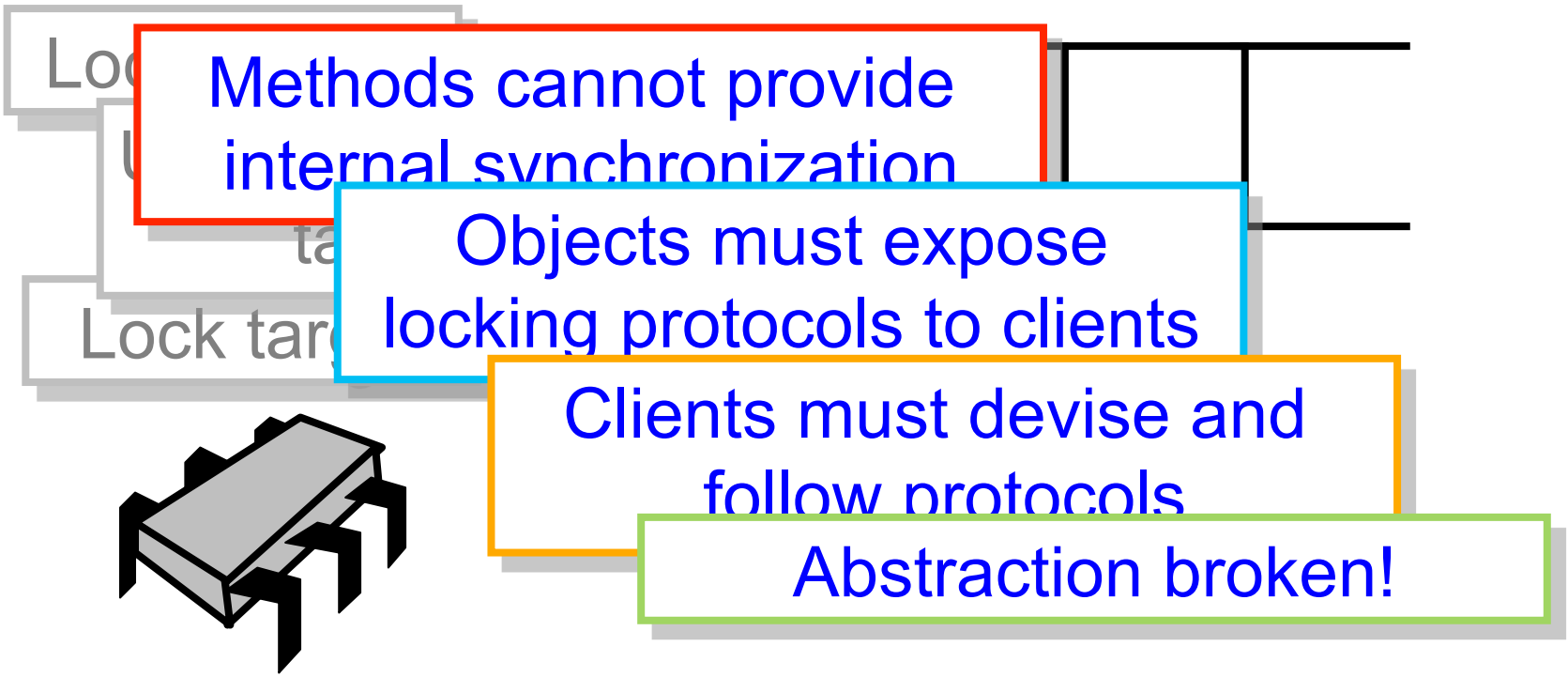


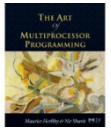




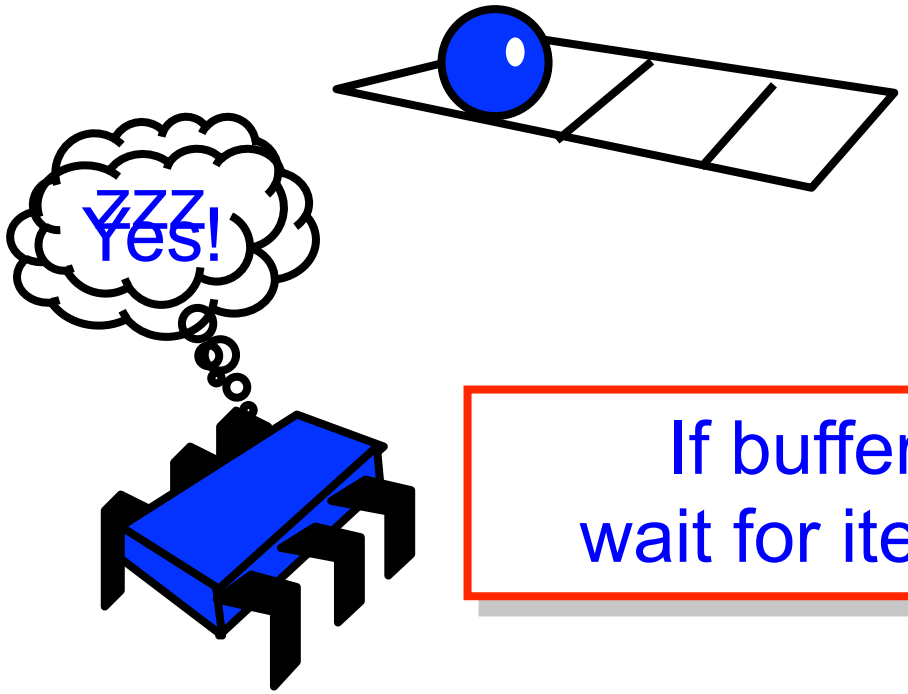


Locks Not Composable





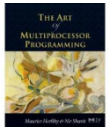
Monitor Wait and Signal



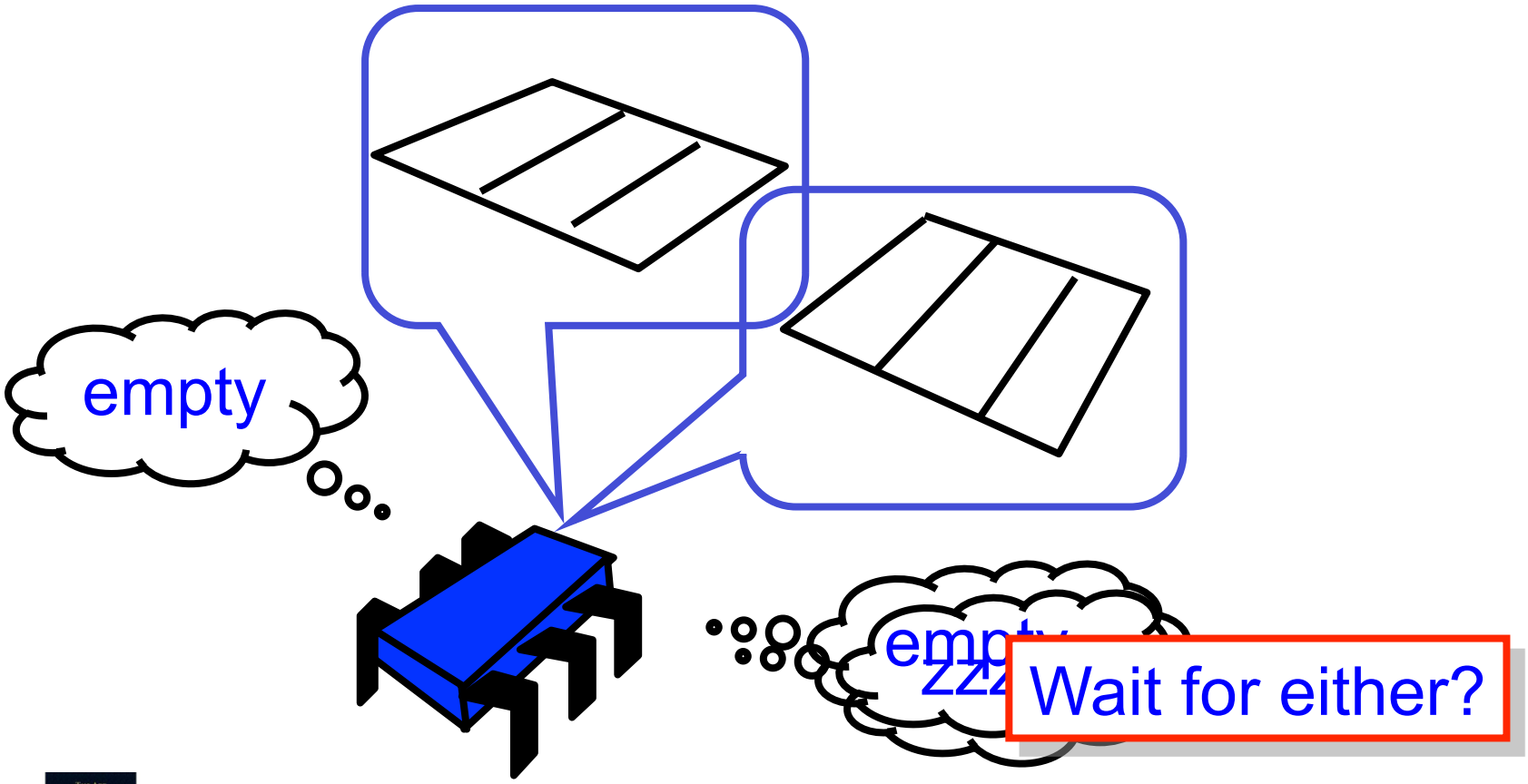
Empty

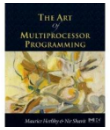
buffer

If buffer is empty,
wait for item to show up



Wait and Signal do not Compose





The Transactional Manifesto

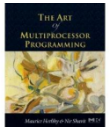
Much modern programming practice inadequate for multicore world

Agenda

Replace **locking** with a **transactional** API

Design **languages** and **libraries**

Implement efficient **run-times**



Road Map

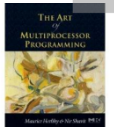
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

Research Questions



Road Map

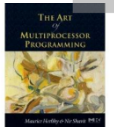
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

Research Questions



Transactions

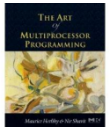
Block of code ….

Atomic: appears to happen instantaneously

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

Commit: takes effect (atomically)

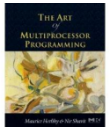
Abort: has no effect (typically restarted)



Atomic Blocks

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

atomic {
  y = null;
}
```

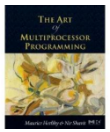


Atomic Blocks

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

atomic {
  y = null;
}
```

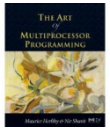
No data race



A Double-Ended Queue

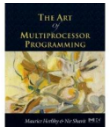
```
public void LeftEnq(item x) {
  Qnode q = new Qnode(x);
  q.left = left;
  left.right = q;
  left = q;
}
```

Write sequential Code



A Double-Ended Queue

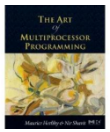
```
public void LeftEnq(item x)
 atomic {
  Qnode q = new Qnode(x);
  q.left = left;
  left.right = q;
  left = q;
 }
}
```



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

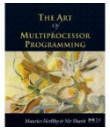
Not always this simple!

Conditional waits?

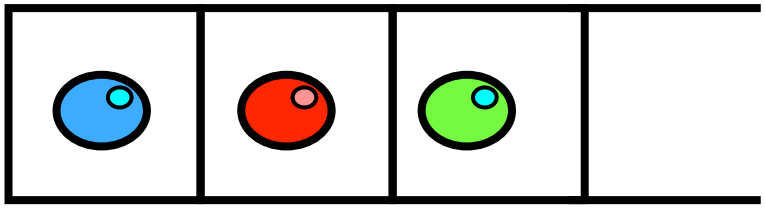
False conflicts?

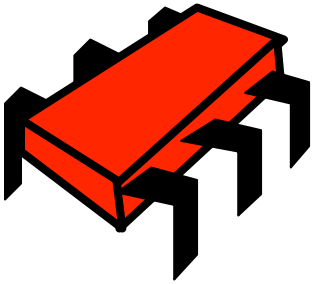
Resource limits?

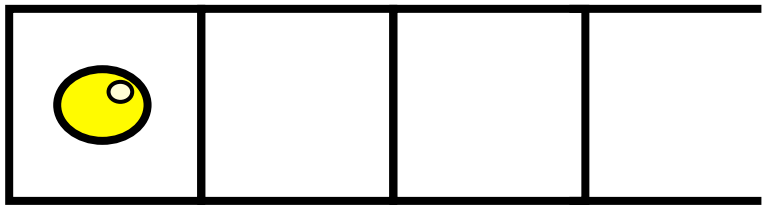
Better problems to have …

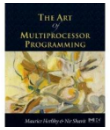


Composition?





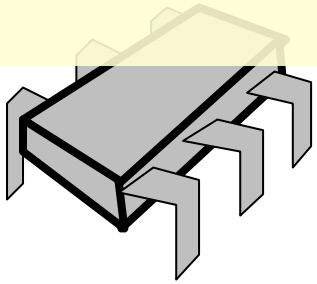


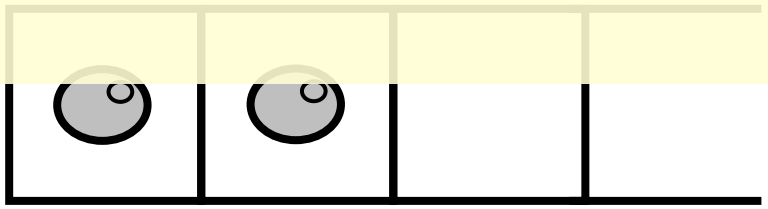


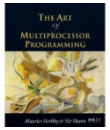
Composition?

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

Trivial or what?





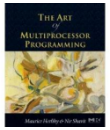


Conditional Waiting

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

 }
}
```

Roll back transaction and restart when something changes



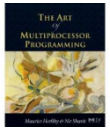
Composable Conditional Waiting

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

Run 1st method. If it retries …

Run 2nd method. If it retries …

Entire statement retries



Road Map

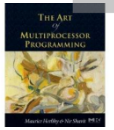
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

Research Questions

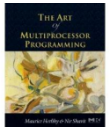


Hardware Transactional Memory

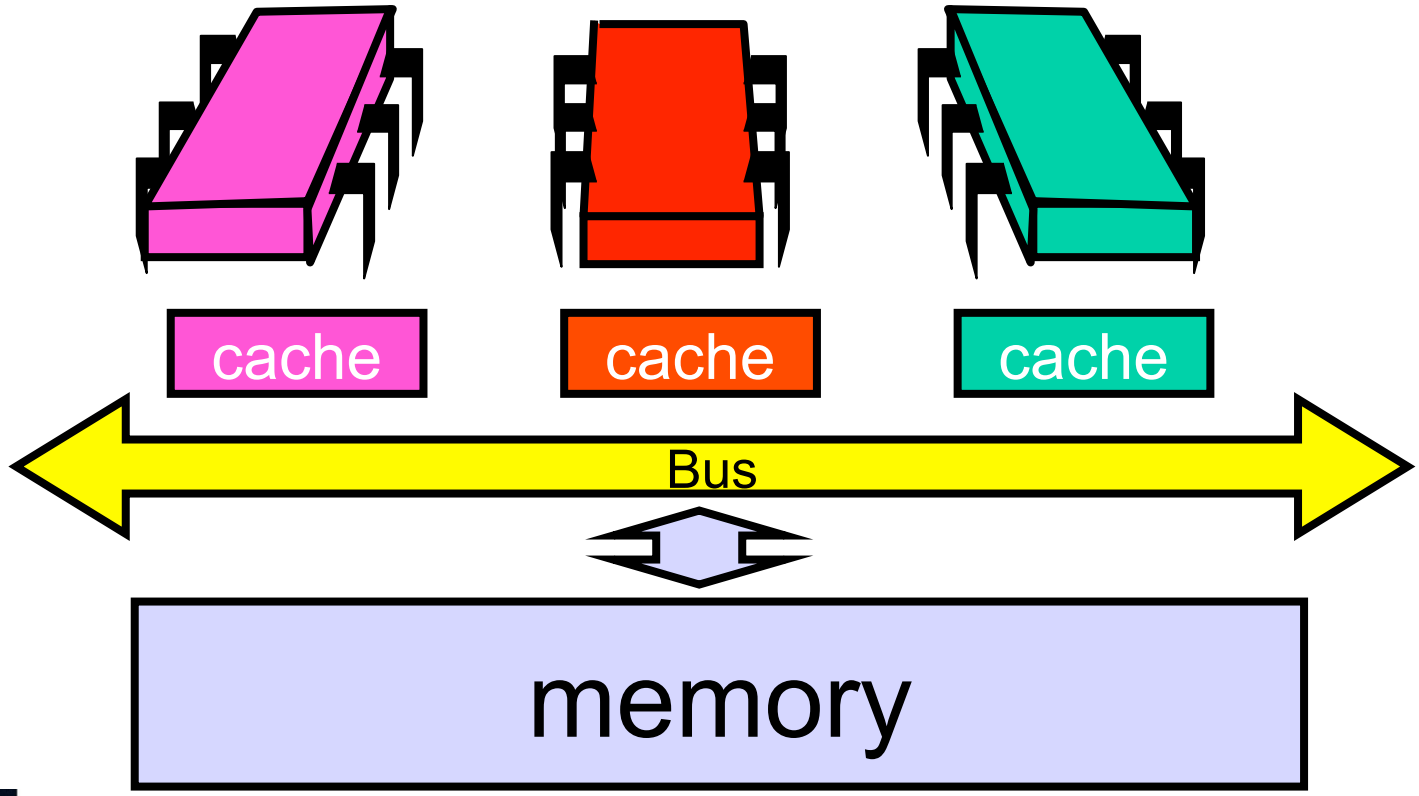
Exploit standard “cache coherence”

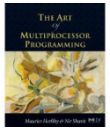
Detect synchronization conflicts …

Invalidate cached copies of data.

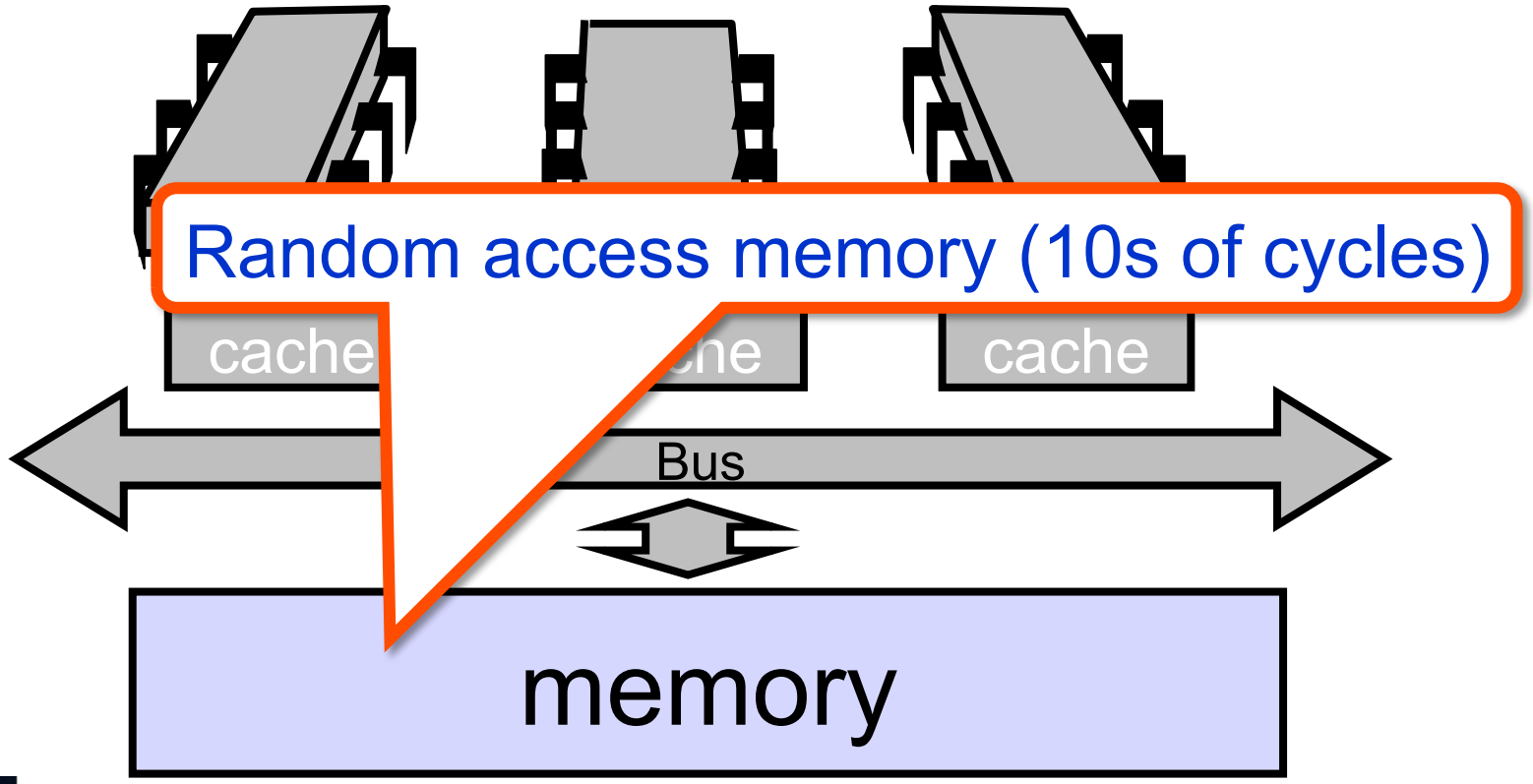


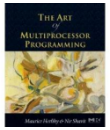
Standard Cache Coherence





Standard Cache Coherence

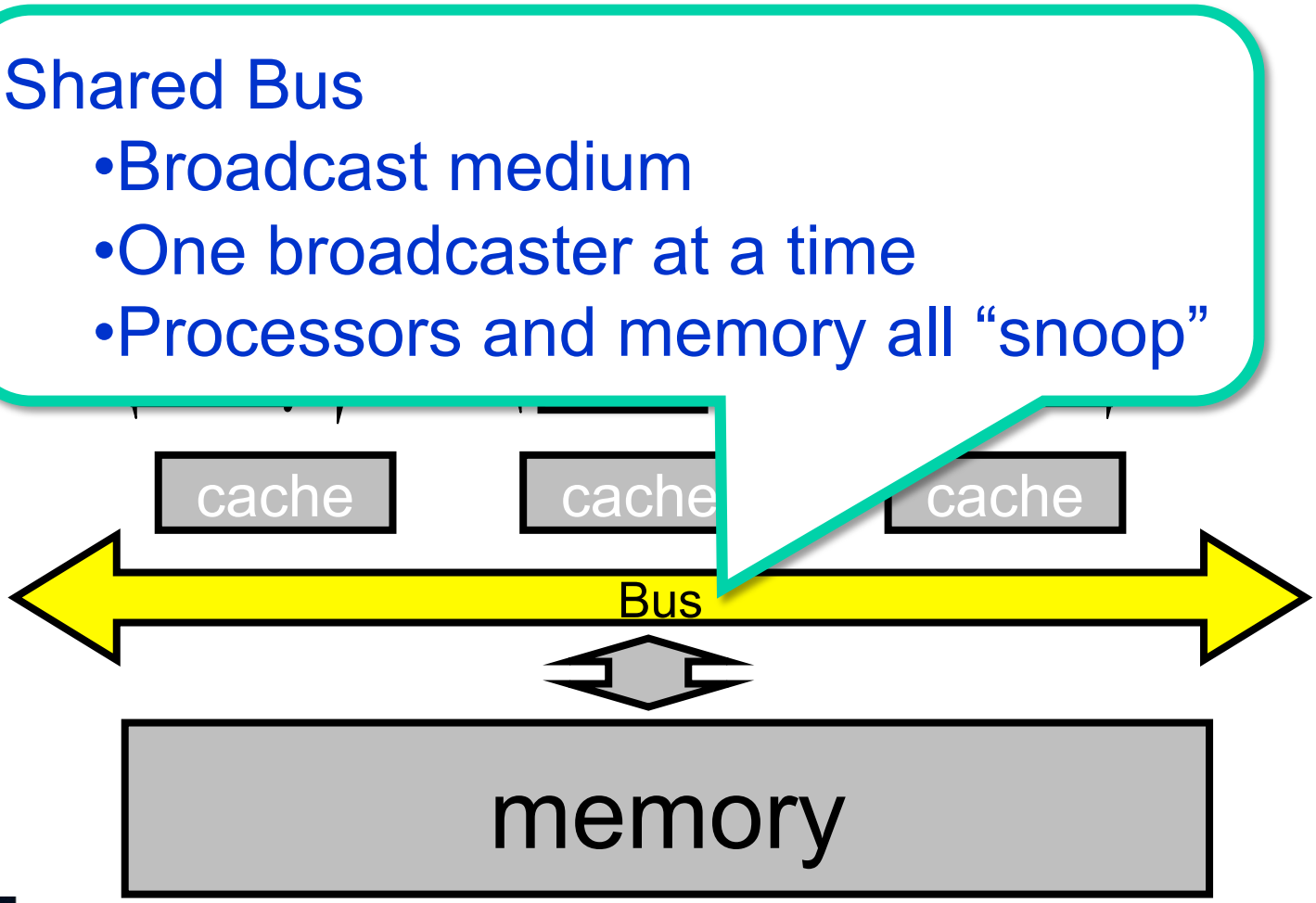


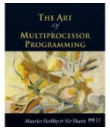


Standard Cache Coherence

Shared Bus

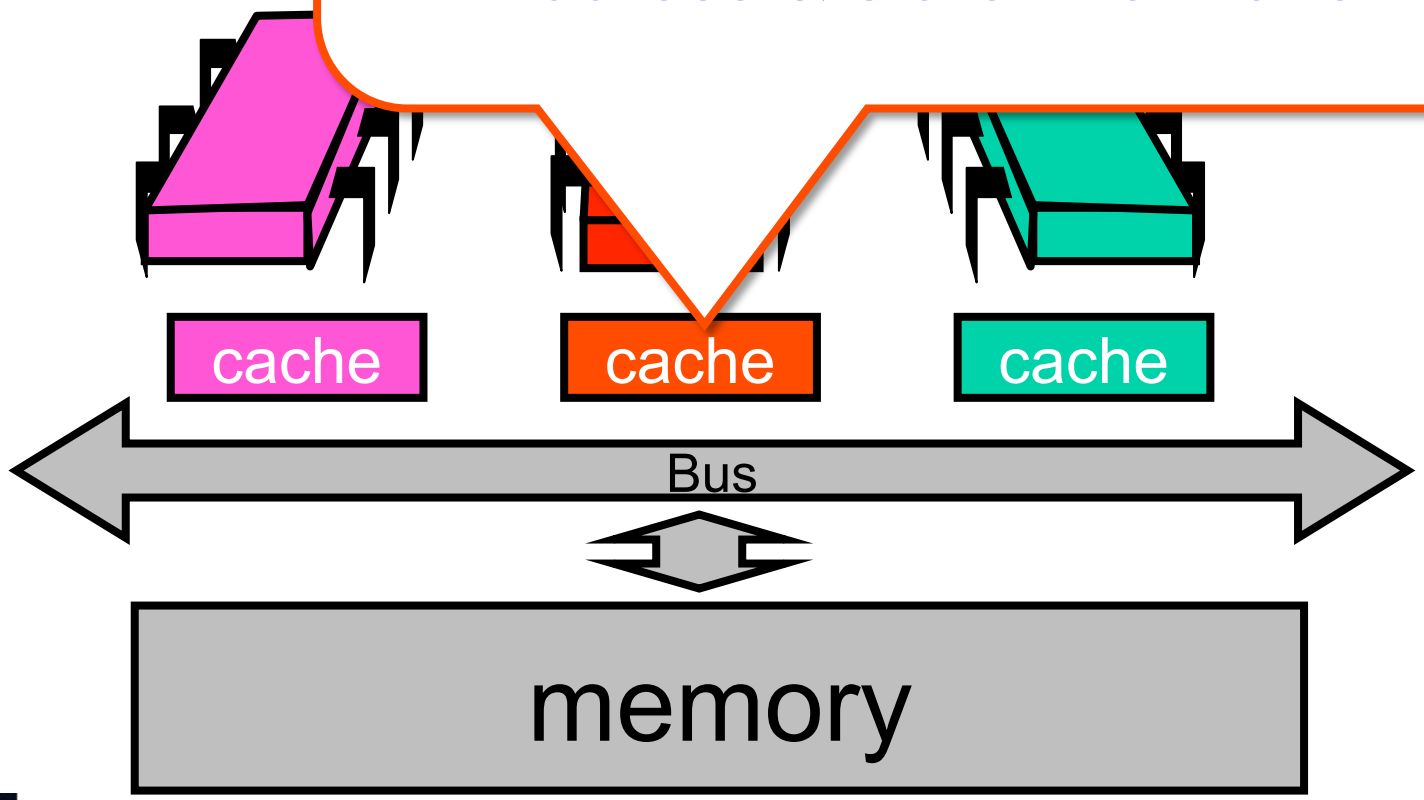
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”

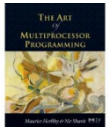




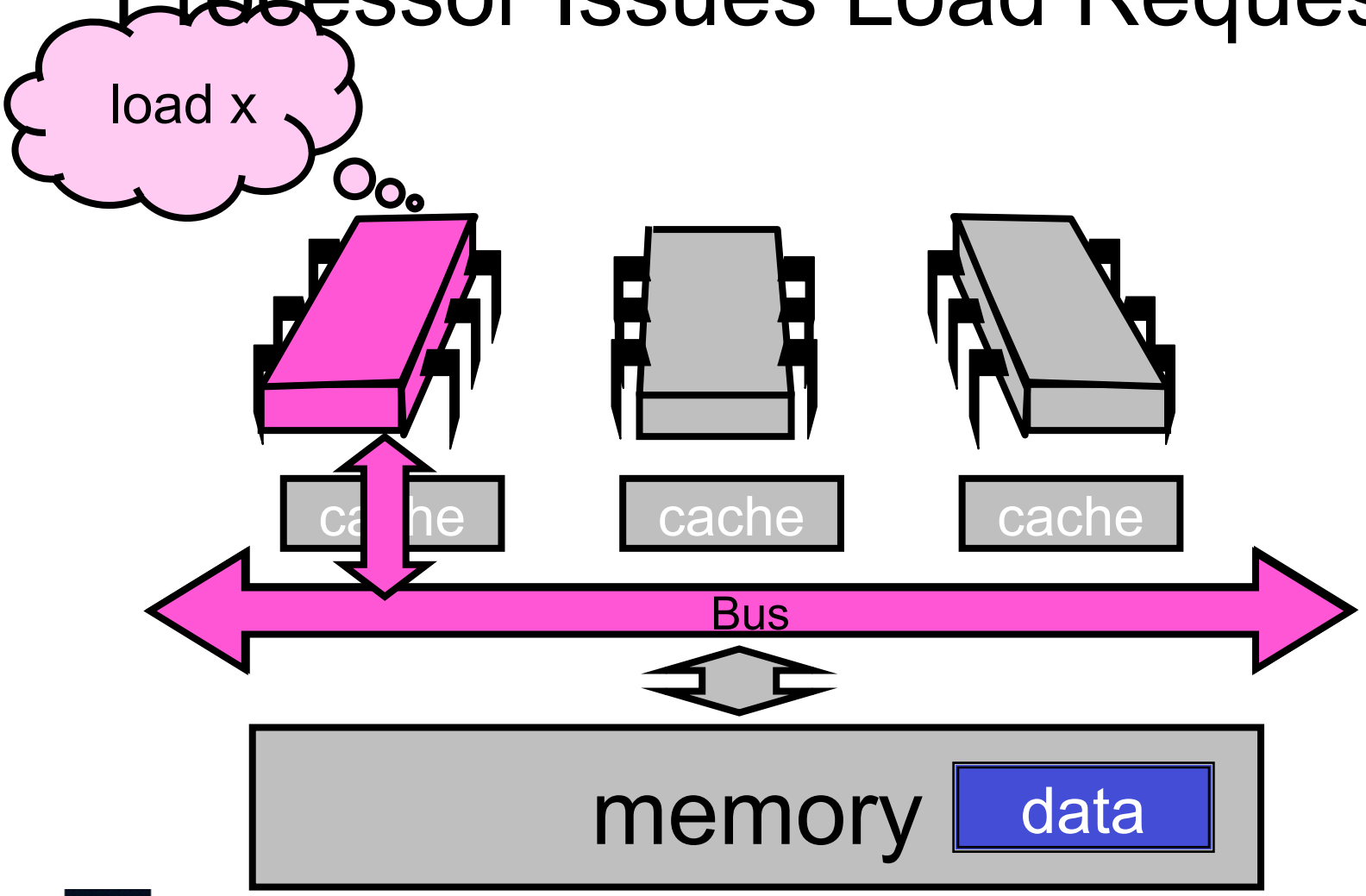
Stand

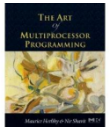
Per-Processor Caches
- Small
- Fast: 1 or 2 cycles
- Address & state information



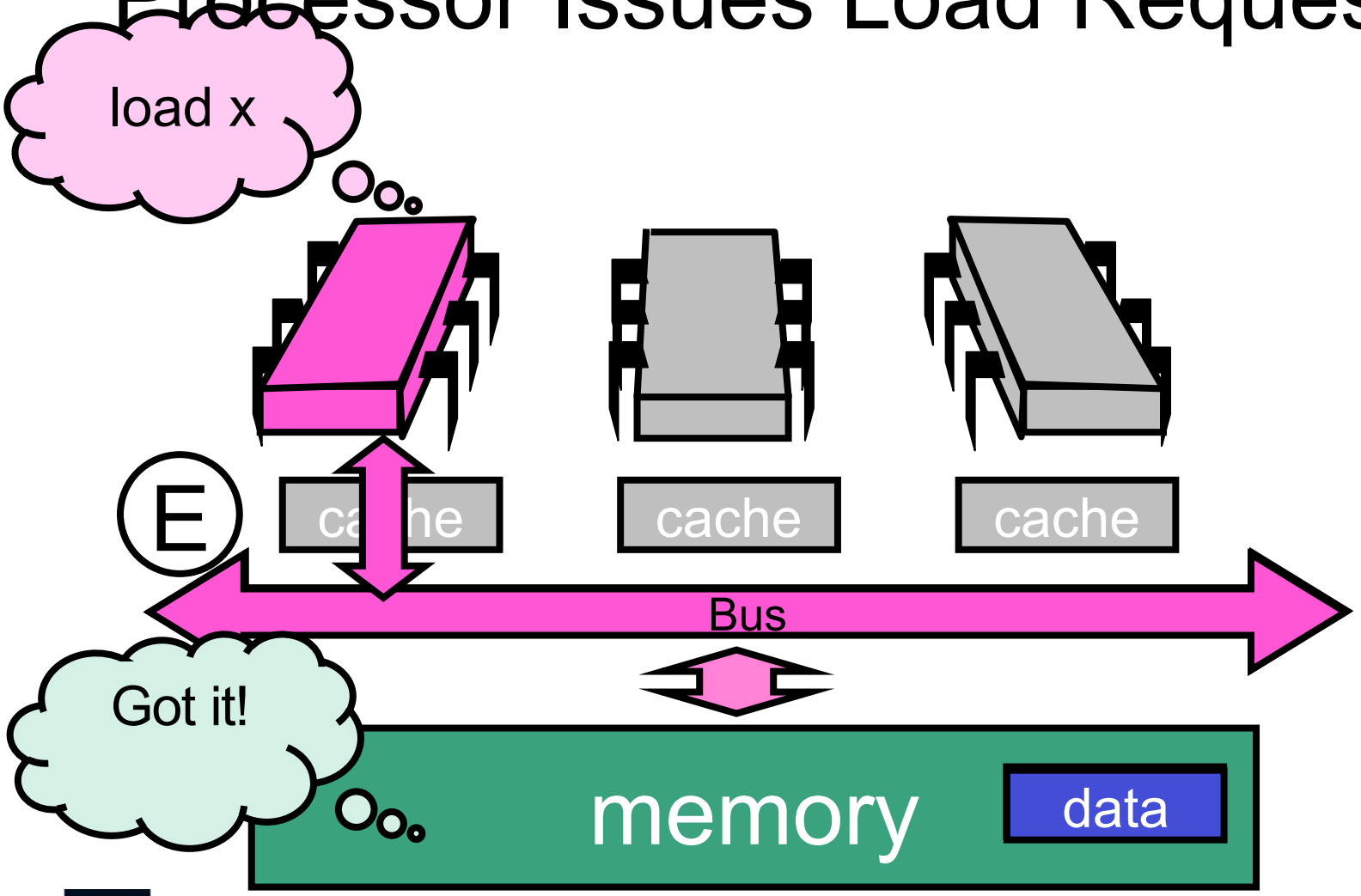


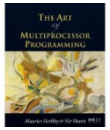
Processor Issues Load Request



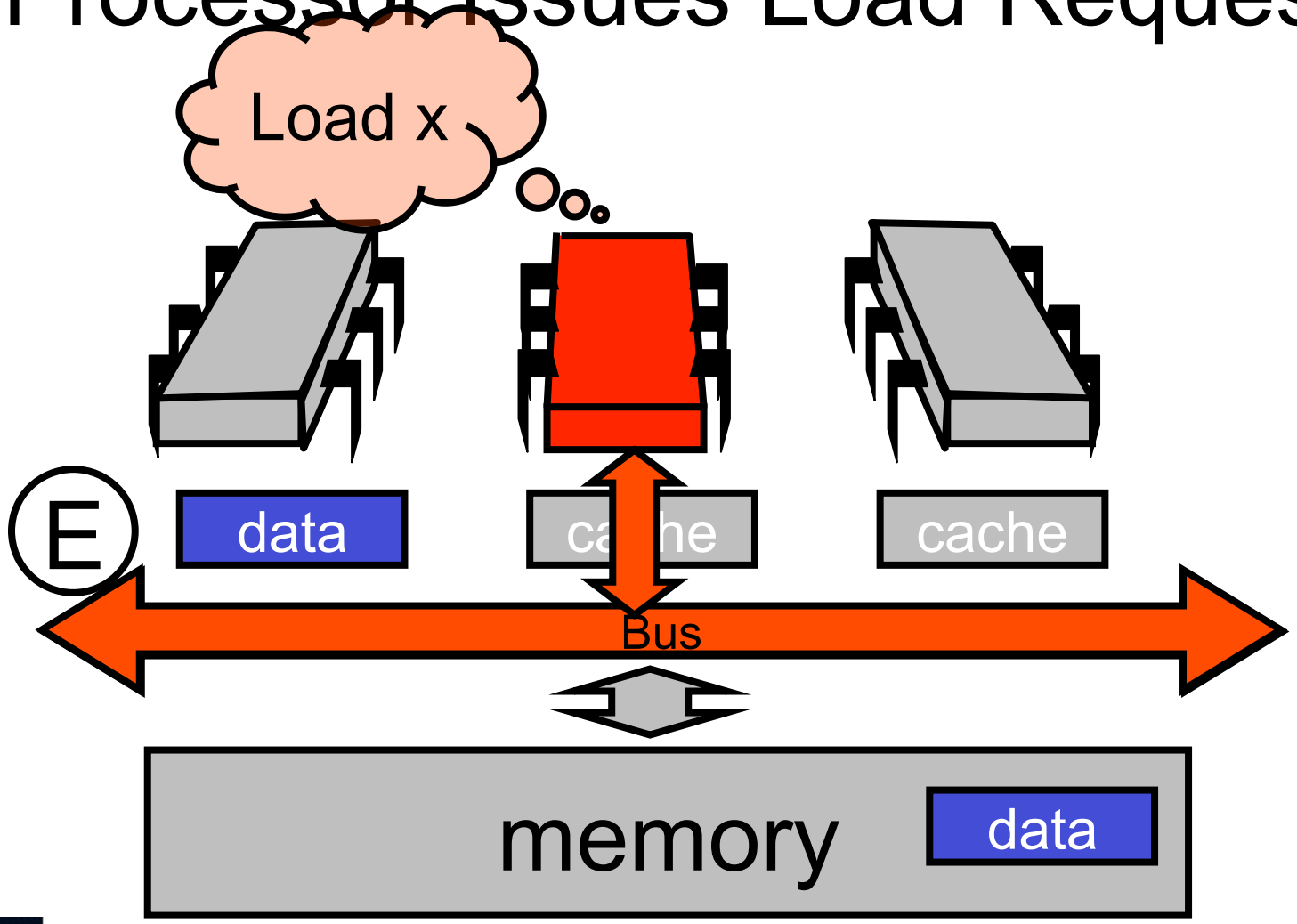


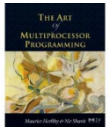
Processor Issues Load Request



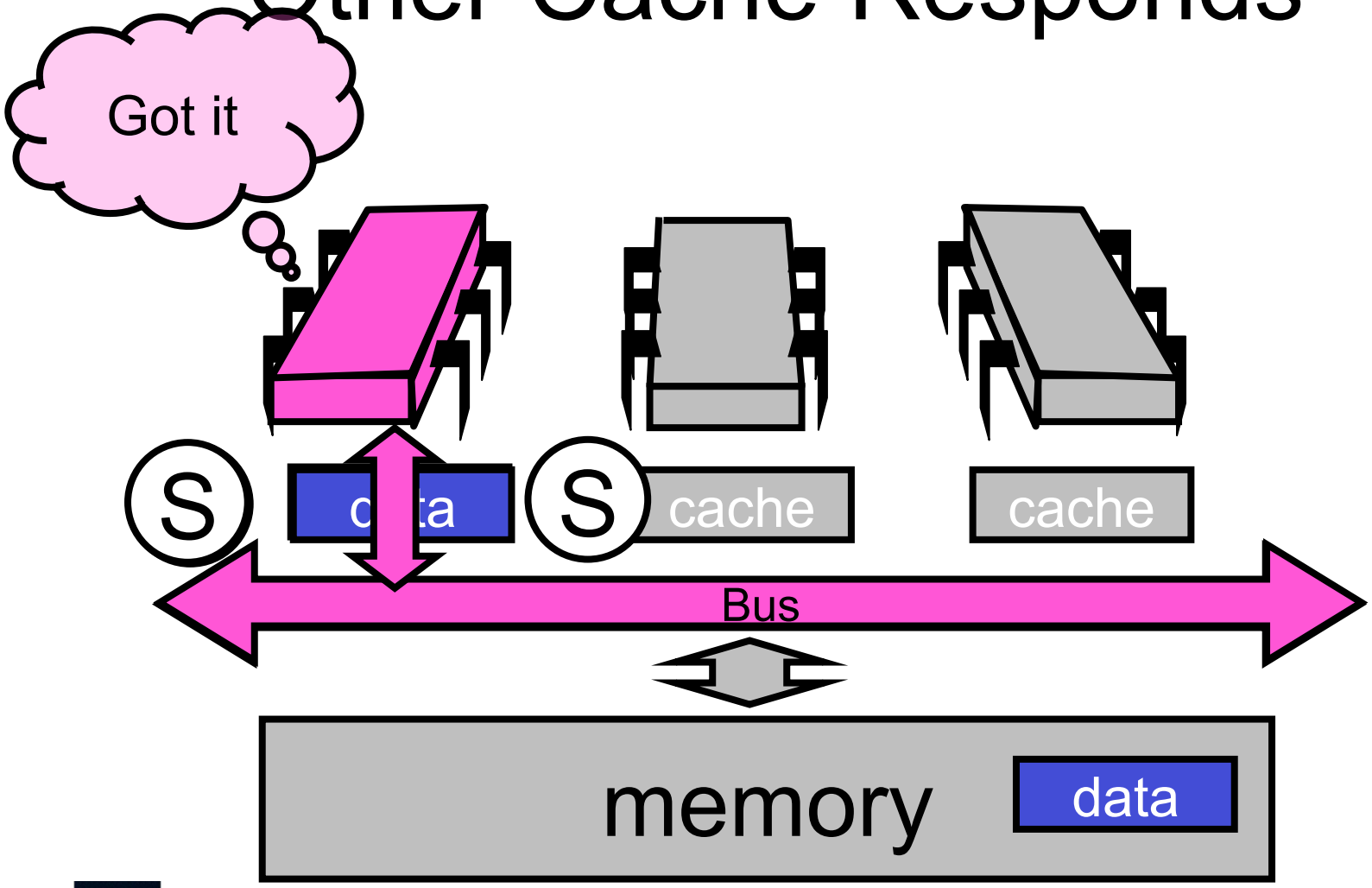


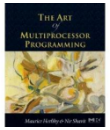
Processor Issues Load Request



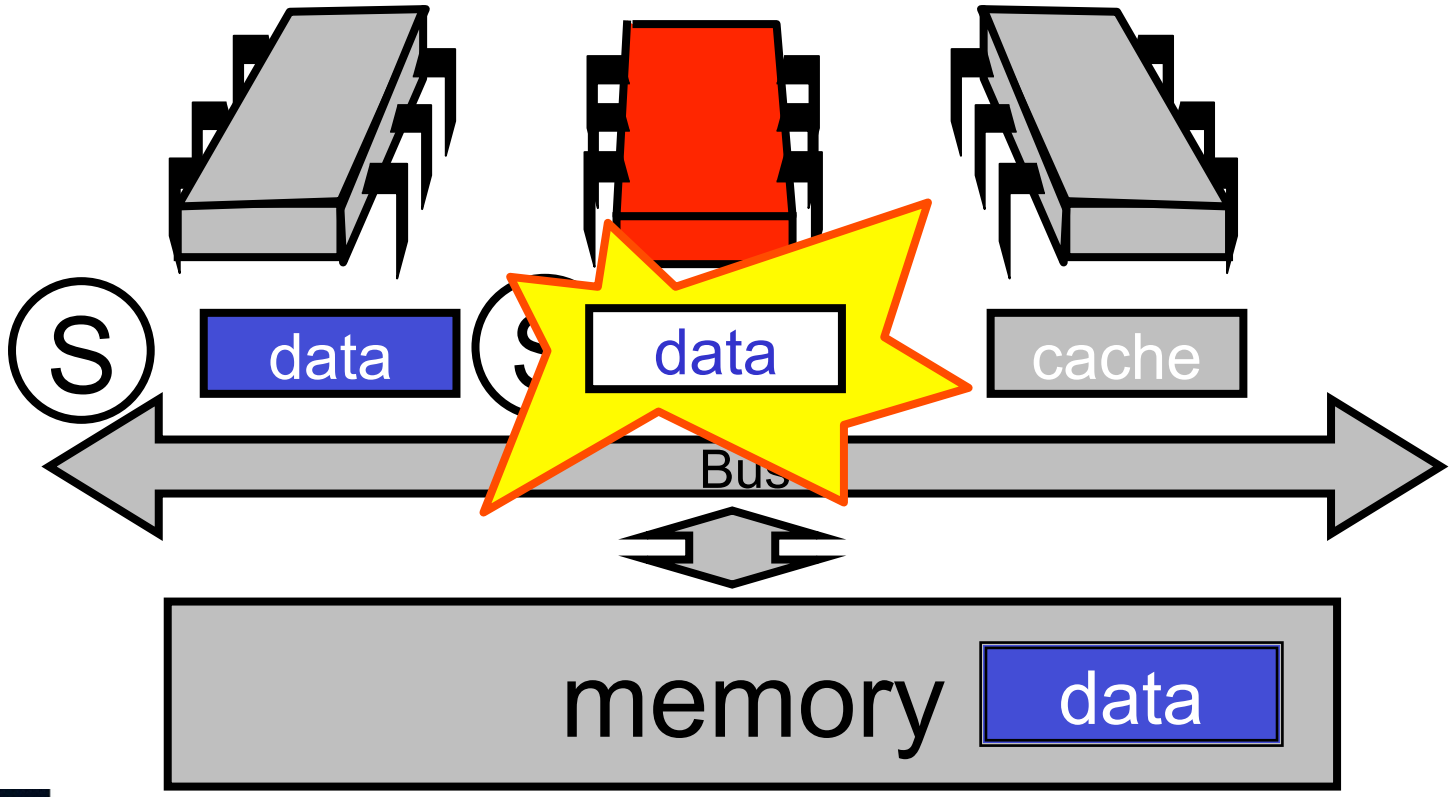


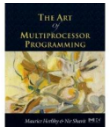
Other Cache Responds



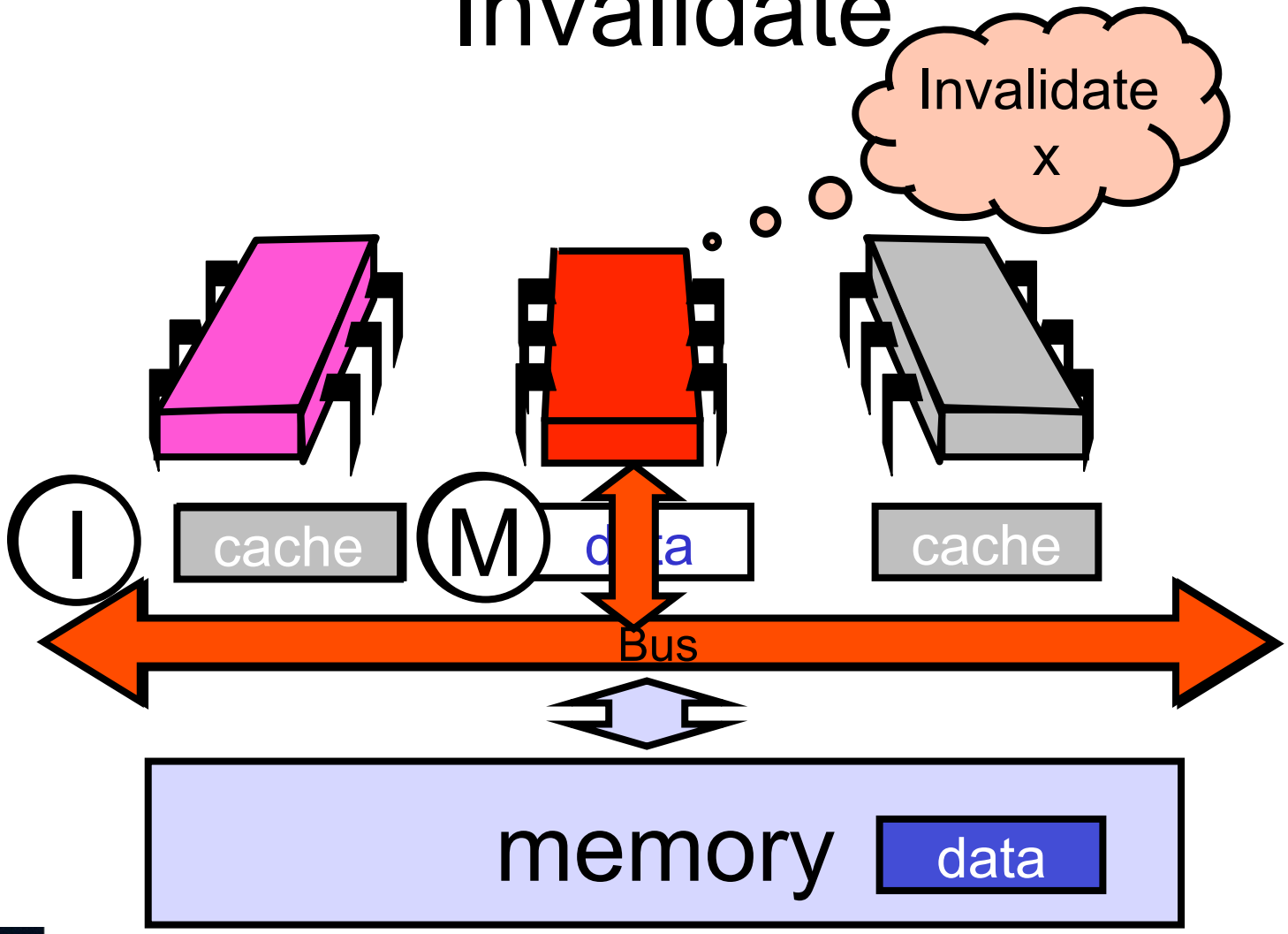


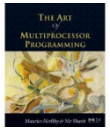
Modify Cached Data



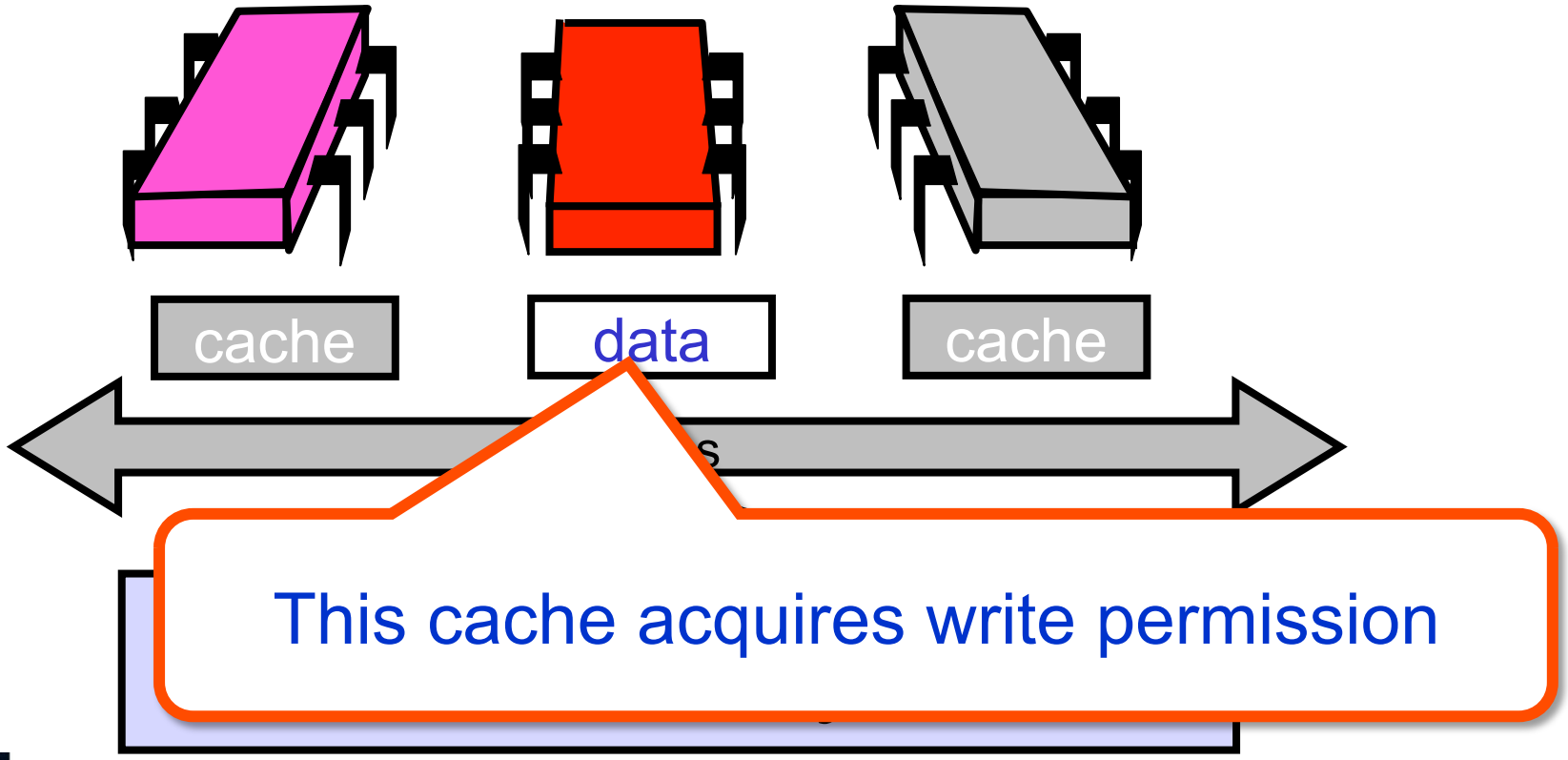


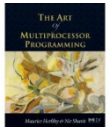
Invalidate



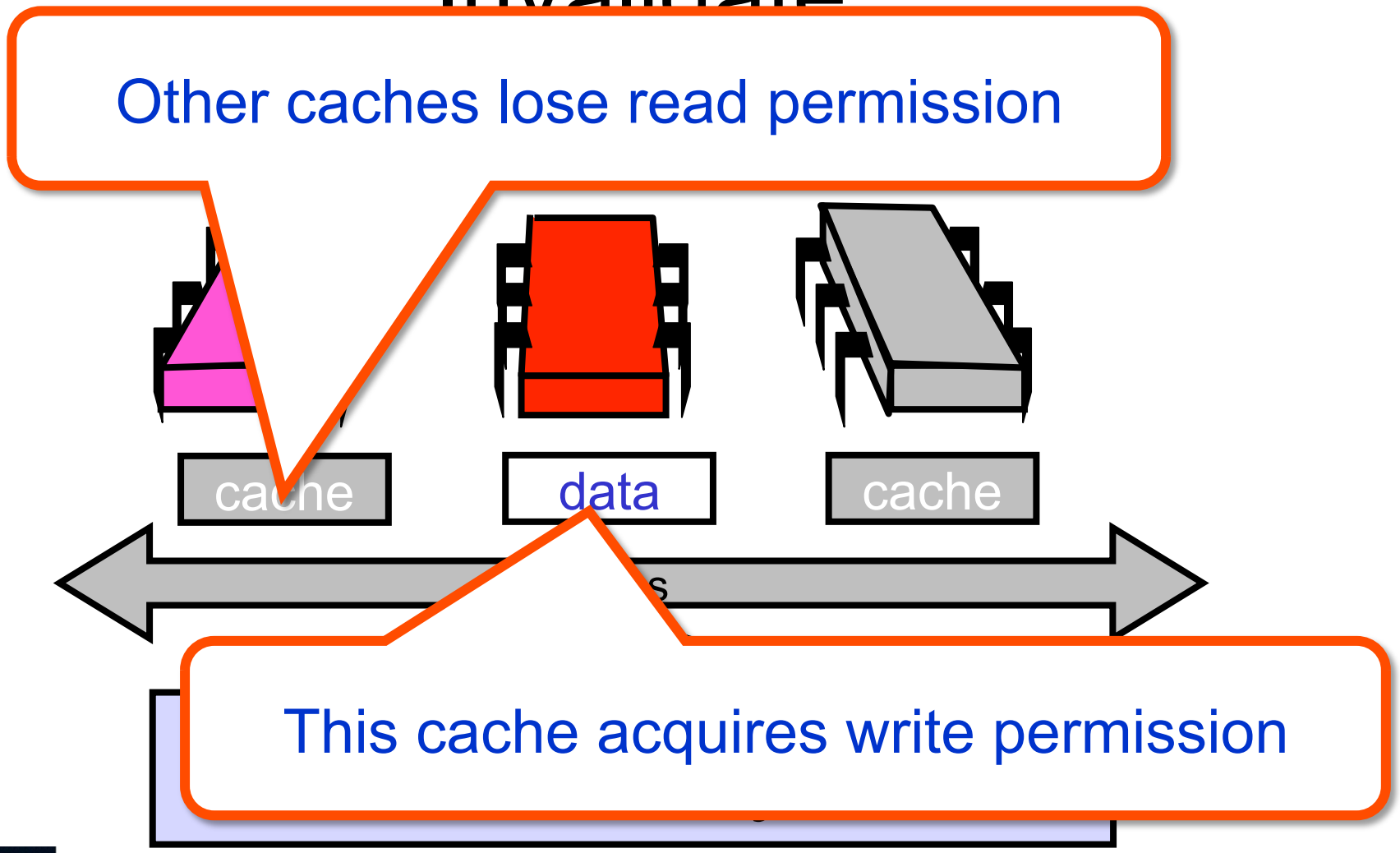


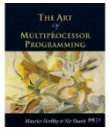
Invalidate



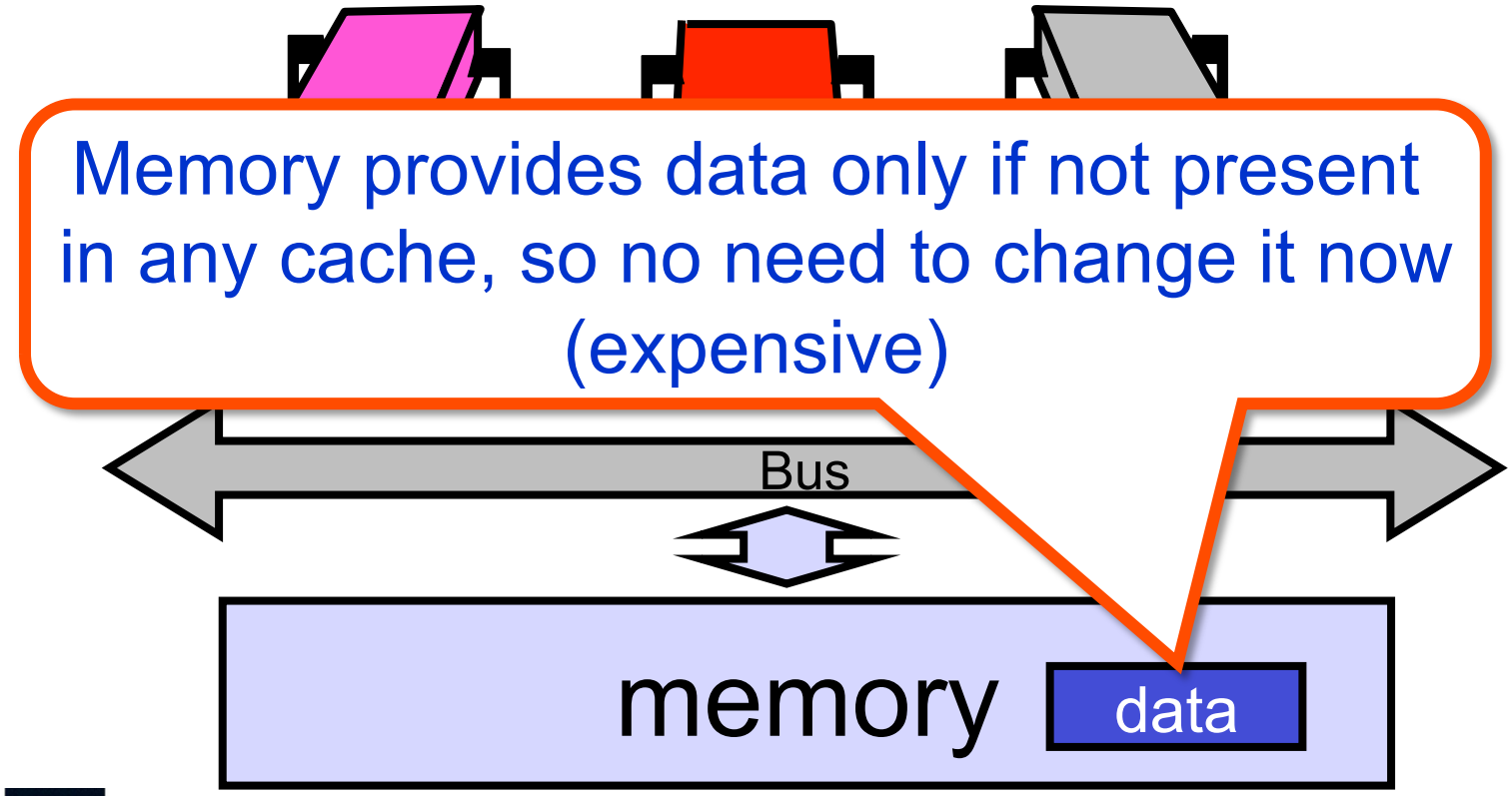


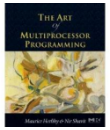
Invalidate


Other caches lose read permission
cache
data
cache
This cache acquires write permission

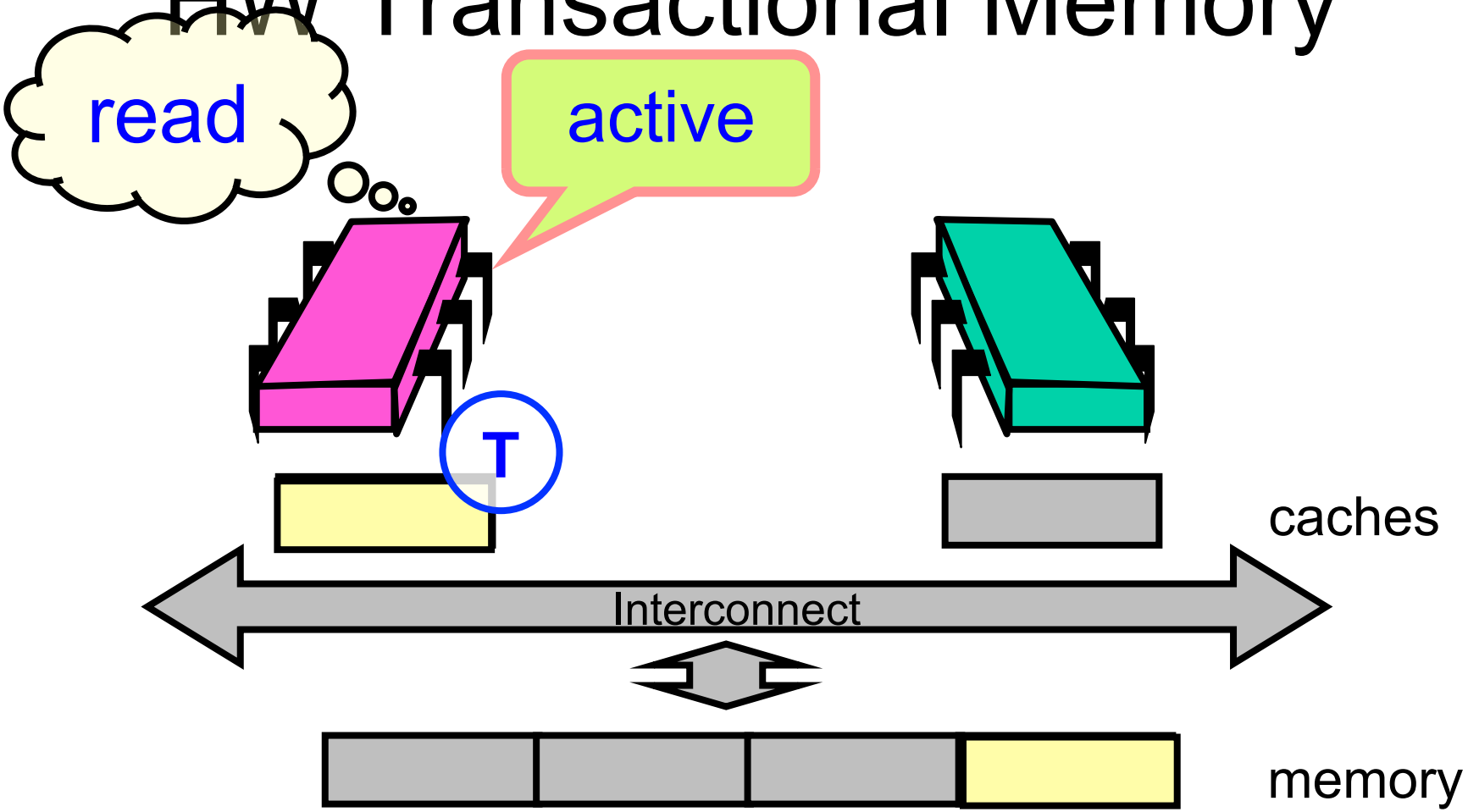


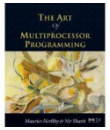
Invalidate


Memory provides data only if not present in any cache, so no need to change it now (expensive)
Bus
memory
data

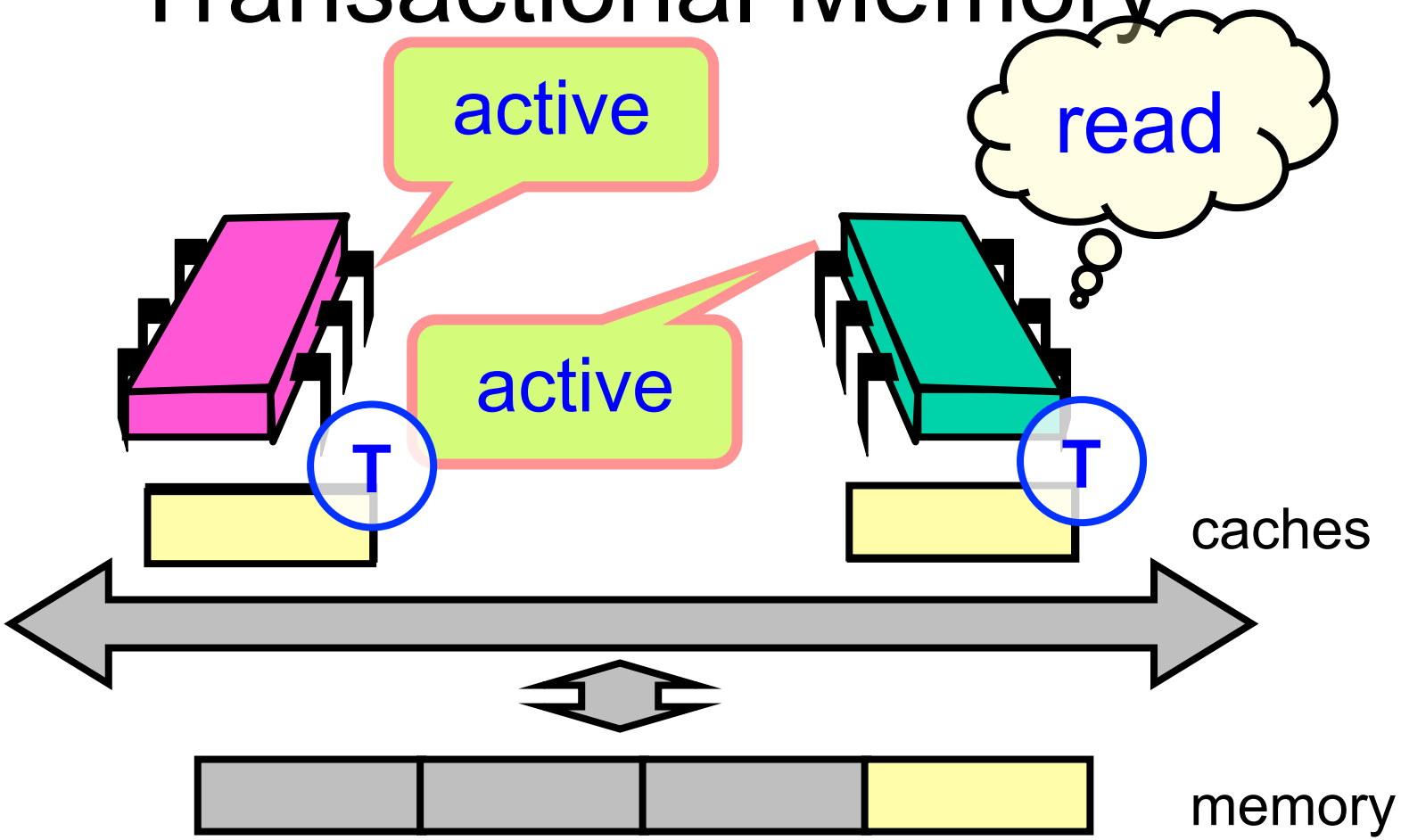


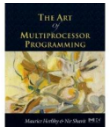
HW Transactional Memory



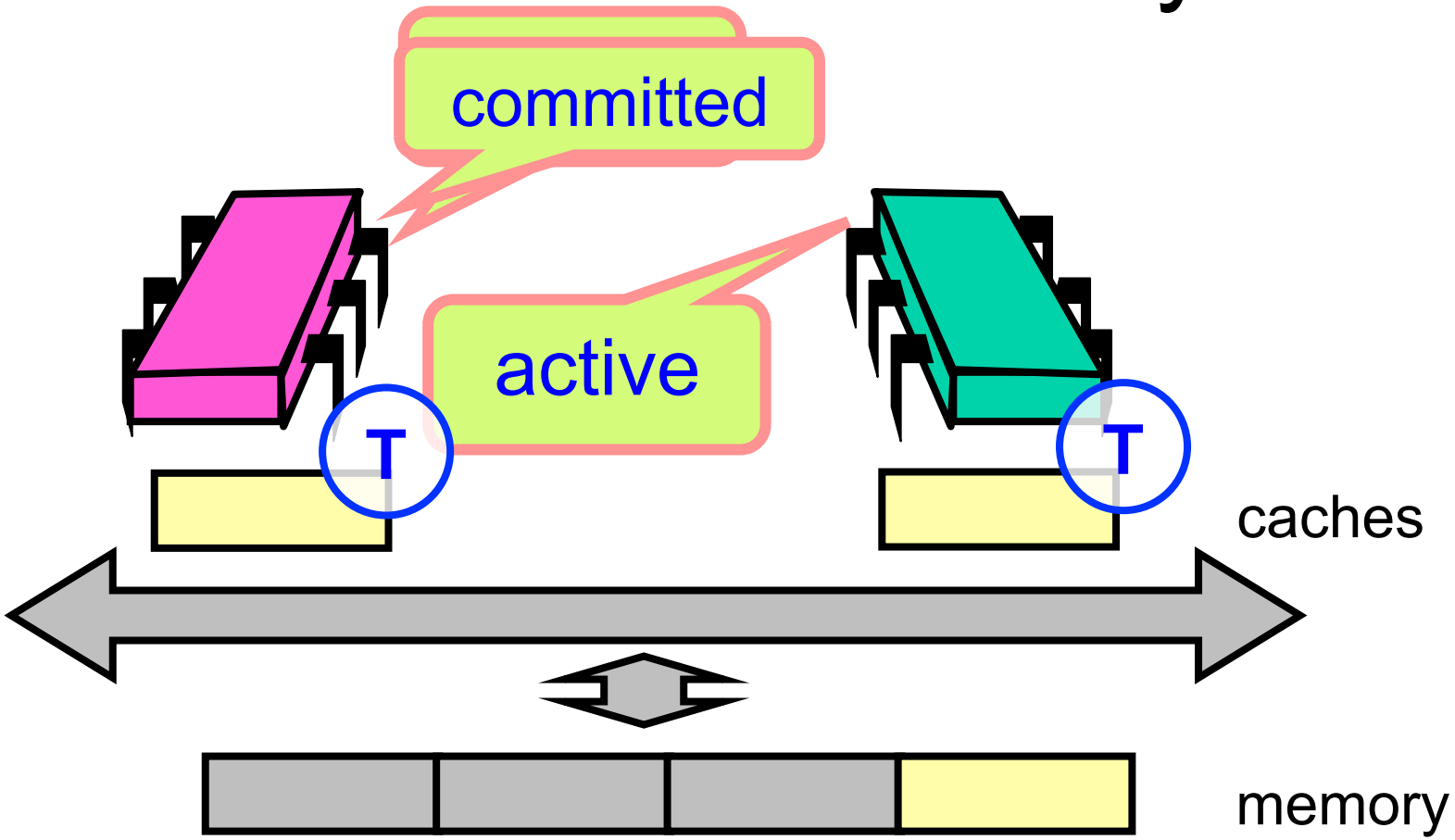


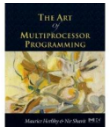
Transactional Memory



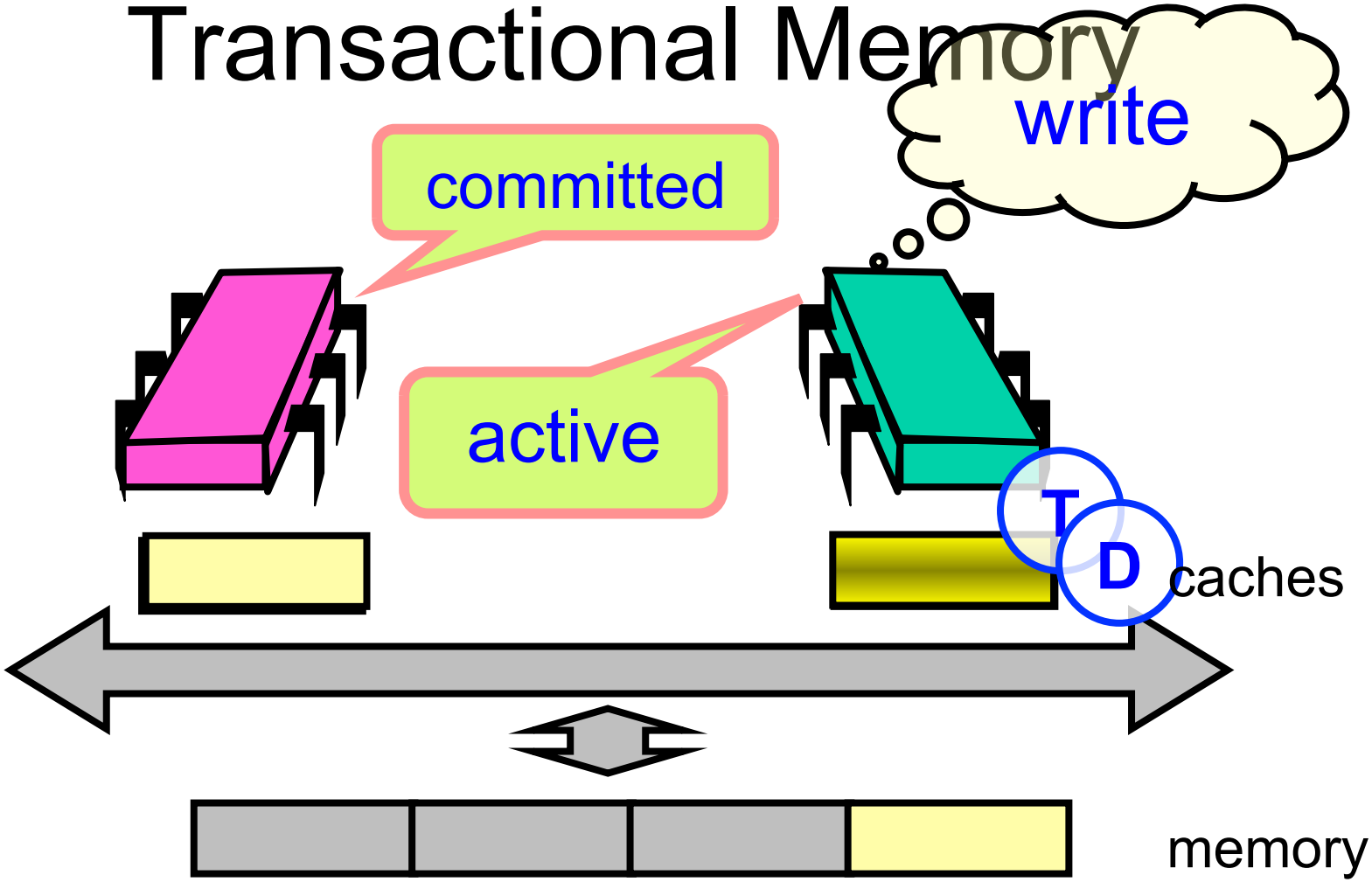


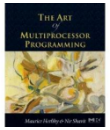
Transactional Memory

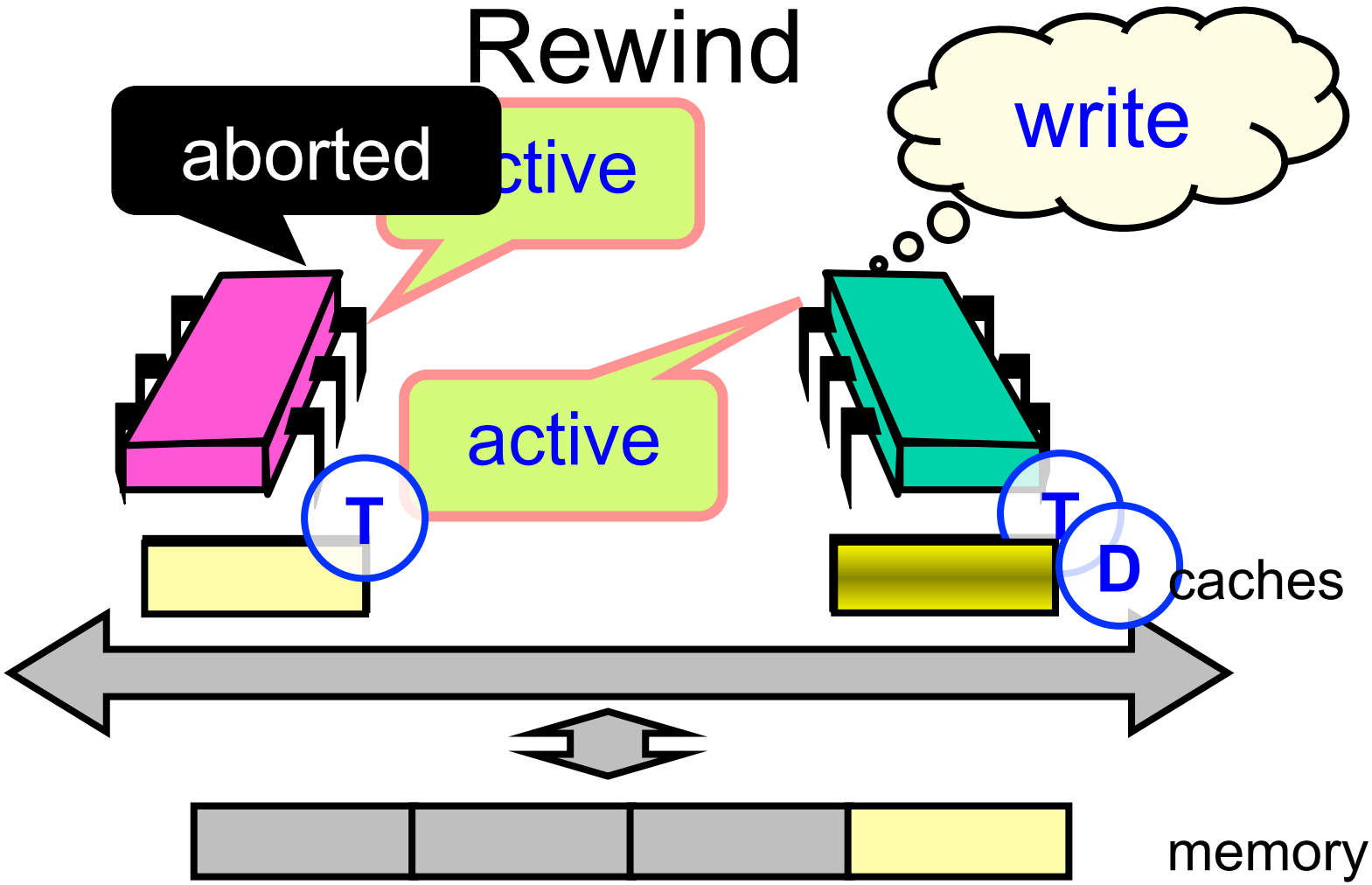


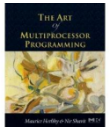


Transactional Memory






Rewind
aborted
ctive
write
active
T
T
D
caches
memory



Transaction Commit

At Commit point …

No cache conflicts? We win.

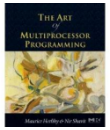
Mark transactional cache entries ….

Was: read-only, Now: valid

Was: modified, Now: dirty

(will be written back)

That's (almost) everything!



Road Map

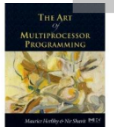
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

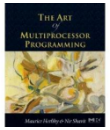
Research Questions



Hardware Transactional Memory (HTM)

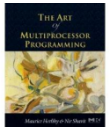
IBM's Blue Gene/Q & System Z & Power8

Intel's Haswell TSX extensions



Intel RTM

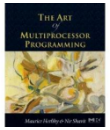
```
if (_xbegin() == _XBEGIN_STARTED) {
   speculative code
   _xend()
} else {
   abort handler
}
```



Intel RTM

```
if (_xbegin() == _XBEGIN_STARTED) {
   speculative code
   _xend()
} else {
   abort handler
}
```

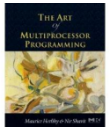
start a speculative transaction



Intel RTM

```
if (_xbegin() == _XBEGIN_STARTED) {
   speculative code
   _xend()
} else {
   abort handler
}
```

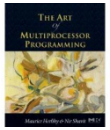
If you see this, you are inside a transaction



Intel RTM

```
if (_xbegin() == _XBEGIN_STARTED) {
   speculative code
    xend()
} else {
   abort handler
}
```

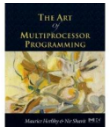
If you see anything else, your transaction aborted



Intel RTM

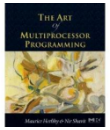
```
if (_xbegin() == _XBEGIN_STARTED) {
   speculative code
   _xend()
} else {
   abort handler
}
```

you could retry the transaction, or take an alternative path



Abort codes

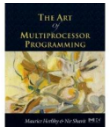
```
if (_xbegin() == _XBEGIN_STARTED) {
   speculative code
} else if (status & _XABORT_EXPLICIT) {
   aborted by user code
} else if (status & _XABORT_CONFLICT) {
   read-write conflict
} else if (status & _XABORT_CAPACITY) {
   cache overflow
} else {
   …
}
```



Abort codes

```
if (_xbegin() == _XBEGIN_STARTED) {
   speculative code
} else if (status & _XABORT_EXPLICIT) {
   aborted by user code
} else if (status & _XABORT_CONFLICT) {
   read-write conflict
} else if (status & _XABORT_CAPACITY) {
   cache overflo
} else {
   …
}
```

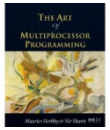
speculative code can call _xabort()



Abort codes

synchronization conflict occurred (maybe retry)

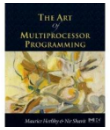
```
if (status & _XABORT_STARTED) {
  s
} else if (status & _XABORT_EXPLICIT) {
  aborted by user code
} else if (status & _XABORT_CONFLICT) {
  read-write conflict
} else if (status & _XABORT_CAPACITY) {
  cache overflow
} else {
  …
}
```



Abort codes

```
if (_xbegin() == _XBEGIN_STARTED) {
   speculative code
} else {
   abort
} else {
   read-write conflict
} else if (status & _XABORT_CAPACITY) {
   cache overflow
} else {
   …
}
```

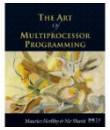
read/write set too big
(maybe don't retry)



Abort codes

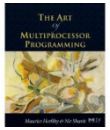
```
if (_xbegin() == _XBEGIN_STARTED) {
   speculative code
} else if (status & _XABORT_EXPLICIT) {
   aborted by user code
} el                                NFLICT) {
   read-write conflict
} else if (status & _XABORT_CAPACITY) {
   cache overflow
} else {
   …
}
```

other abort codes …



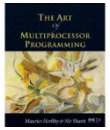
Too Big

Transaction aborts if data set overflows caches, internal buffers

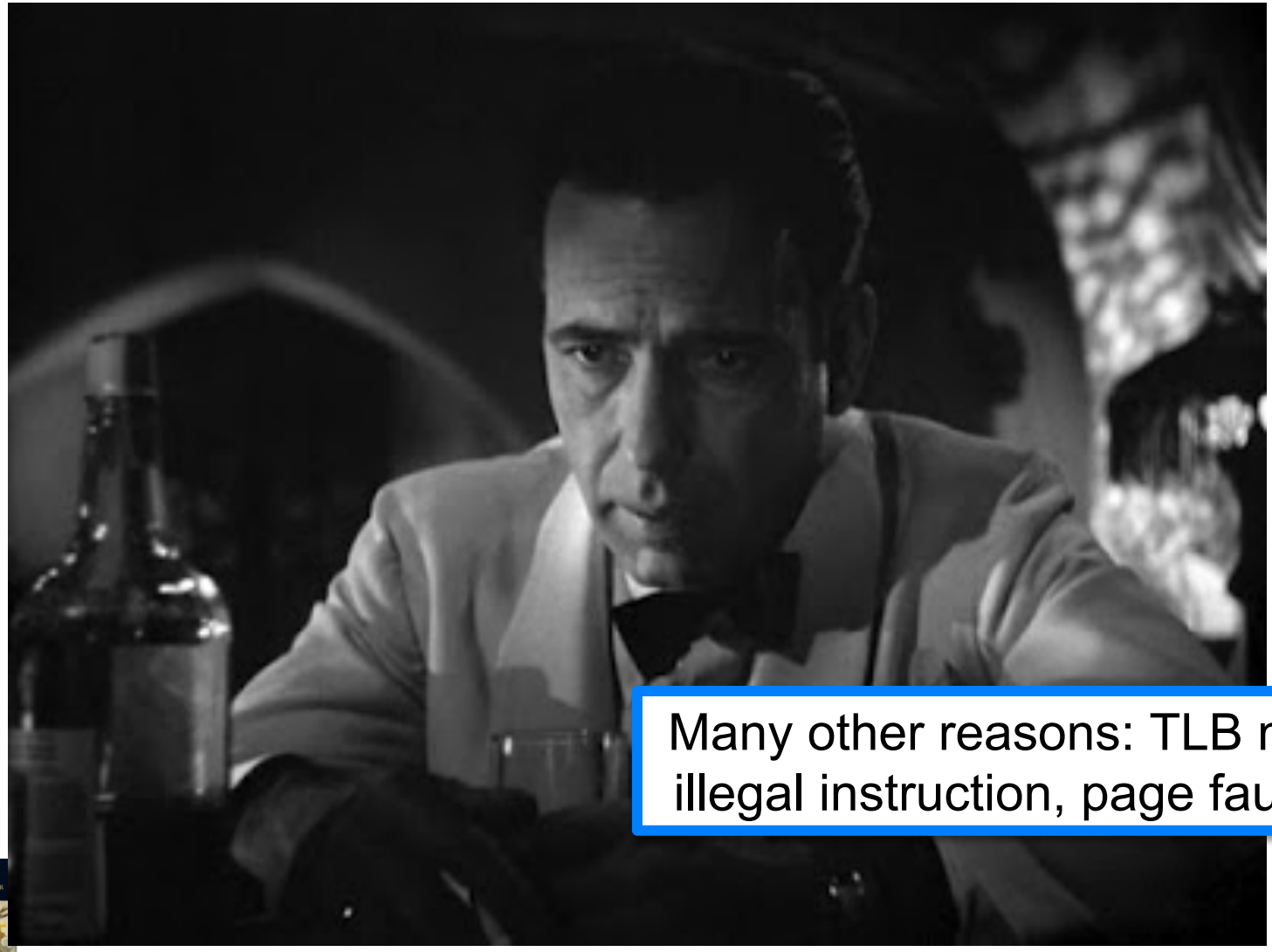


Too Slow

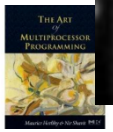
Transaction aborts on timer interrupt



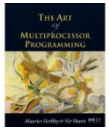
Just Not in the Mood



Many other reasons: TLB miss, illegal instruction, page fault …

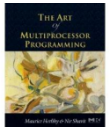


Hybrid Transactional Memory



Non-Speculative Fallback

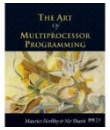
```
if (_xbegin() == _XBEGIN_STARTED) {
   read lock state
   if (lock taken) _xabort();
   work;
   _xend()
} else {
   lock->lock();
   work;
   lock->unlock();
}
```



Non-Speculative Fallback

```
if (_xbegin() == _XBEGIN_STARTED) {
  read lock state
  if (lock taken) _xabort();
  work;
   xend()
} e
  }
```

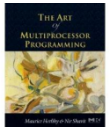
reading lock ensures that transaction will abort if another thread acquires lock



Non-Speculative Fallback

```
if (_xbegin() == _XBEGIN_STARTED) {
  read lock state
  if (lock taken) _xabort();
  work;
  _xend()
} else {
  lo
  wo
  lo
}
```

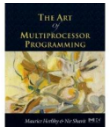
abort if another thread has acquired lock



N

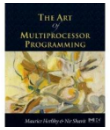
on abort, acquire lock & do work (aborting concurrent speculative transactions)

```
if
   re
   if (lock taken) _xabort();
   work;
    xend()
} else {
   lock->lock();
   work;
   lock->unlock();
}
```



Lock Elision

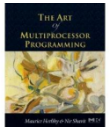
```
<HLE acquire prefix> lock();
do work;
<HLE release prefix> unlock()
```



Lock Elision

```
<HLE acquire prefix> lock();
do work;
<HLE release prefix> unlock()
```

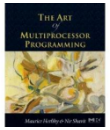
first time around, read lock and execute speculatively



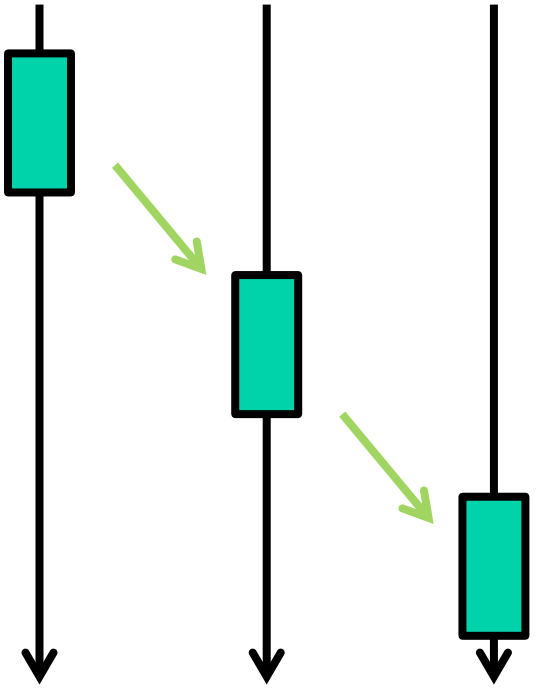
Lock Elision

```
<HLE acquire prefix> lock();
do work;
<HLE release prefix> unlock()
```

**if speculation fails,
no more Mr. Nice Guy,
acquire the lock**



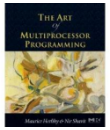
Conventional Locks



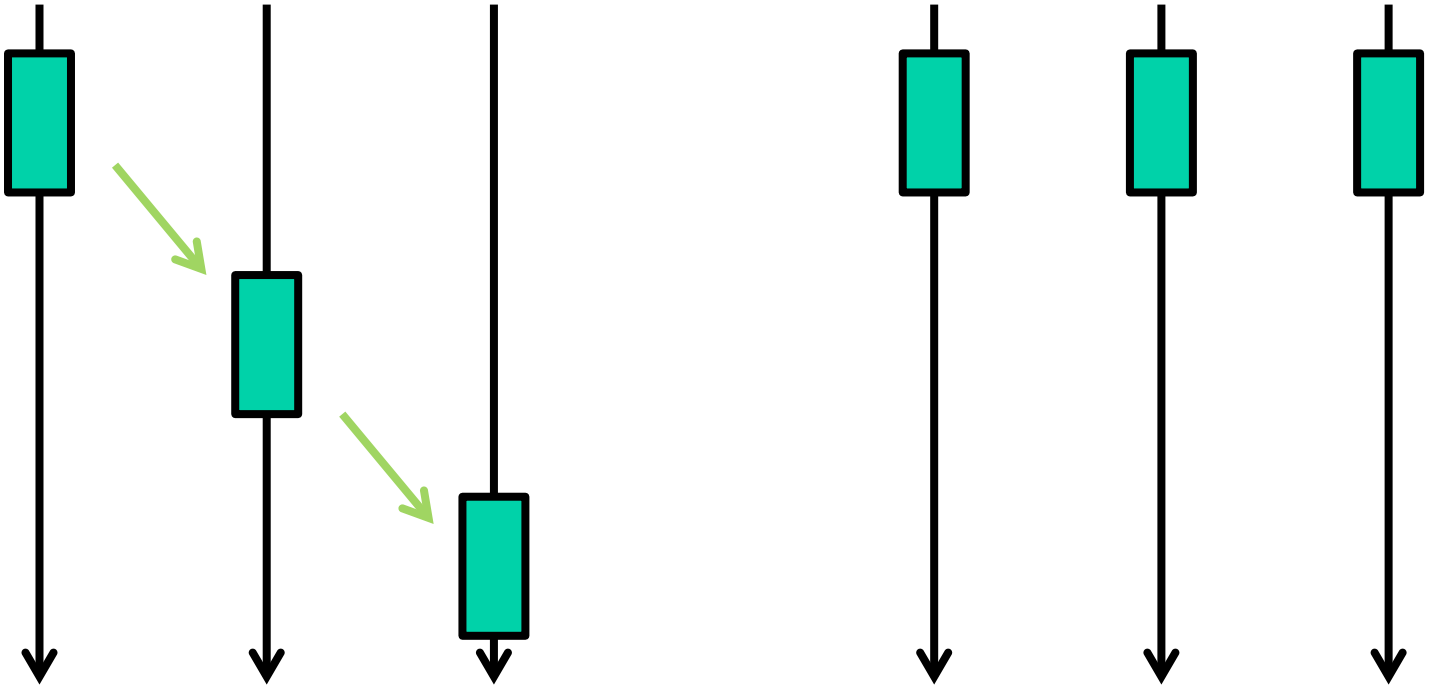
locks

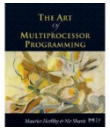
lock transfer latencies

serialized execution

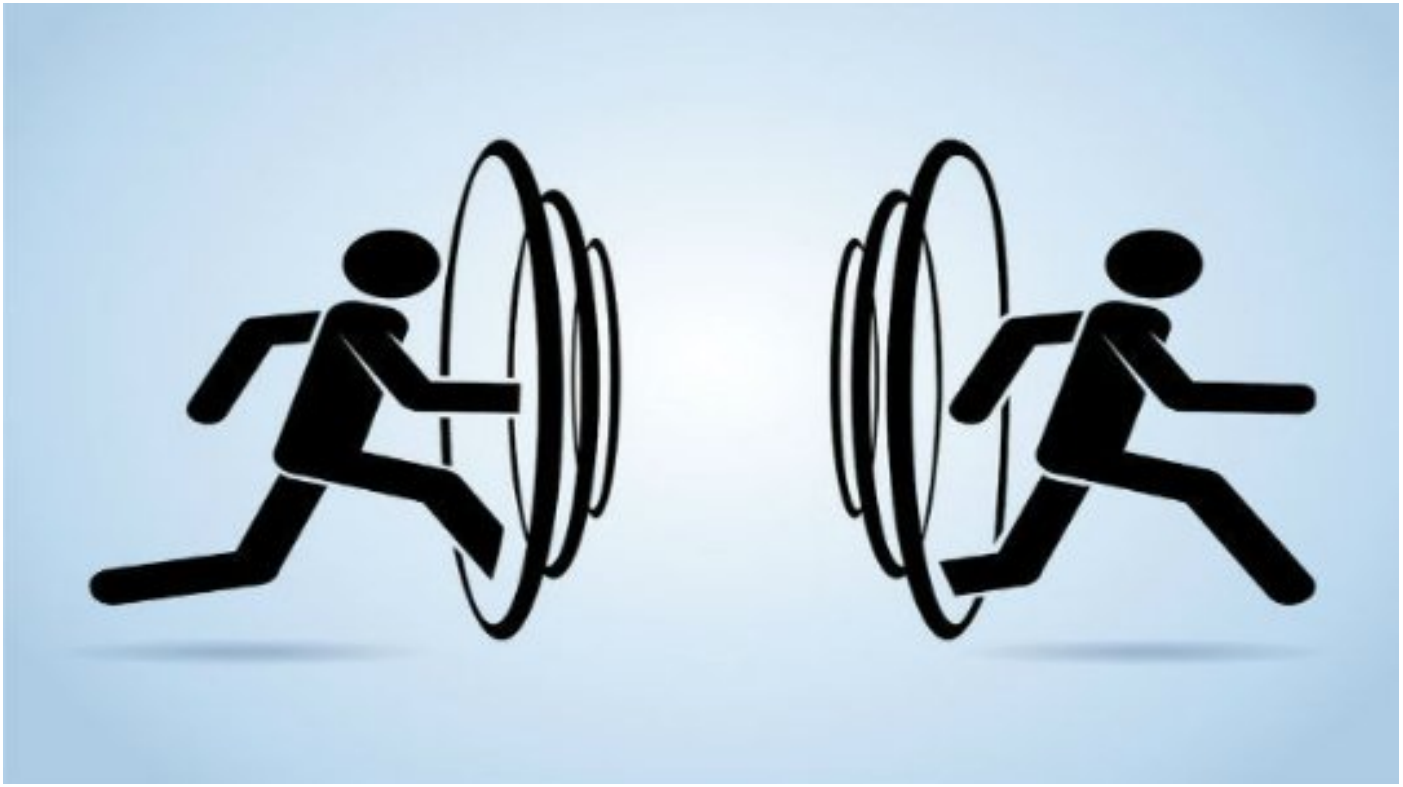


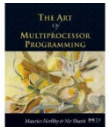
Lock Elision



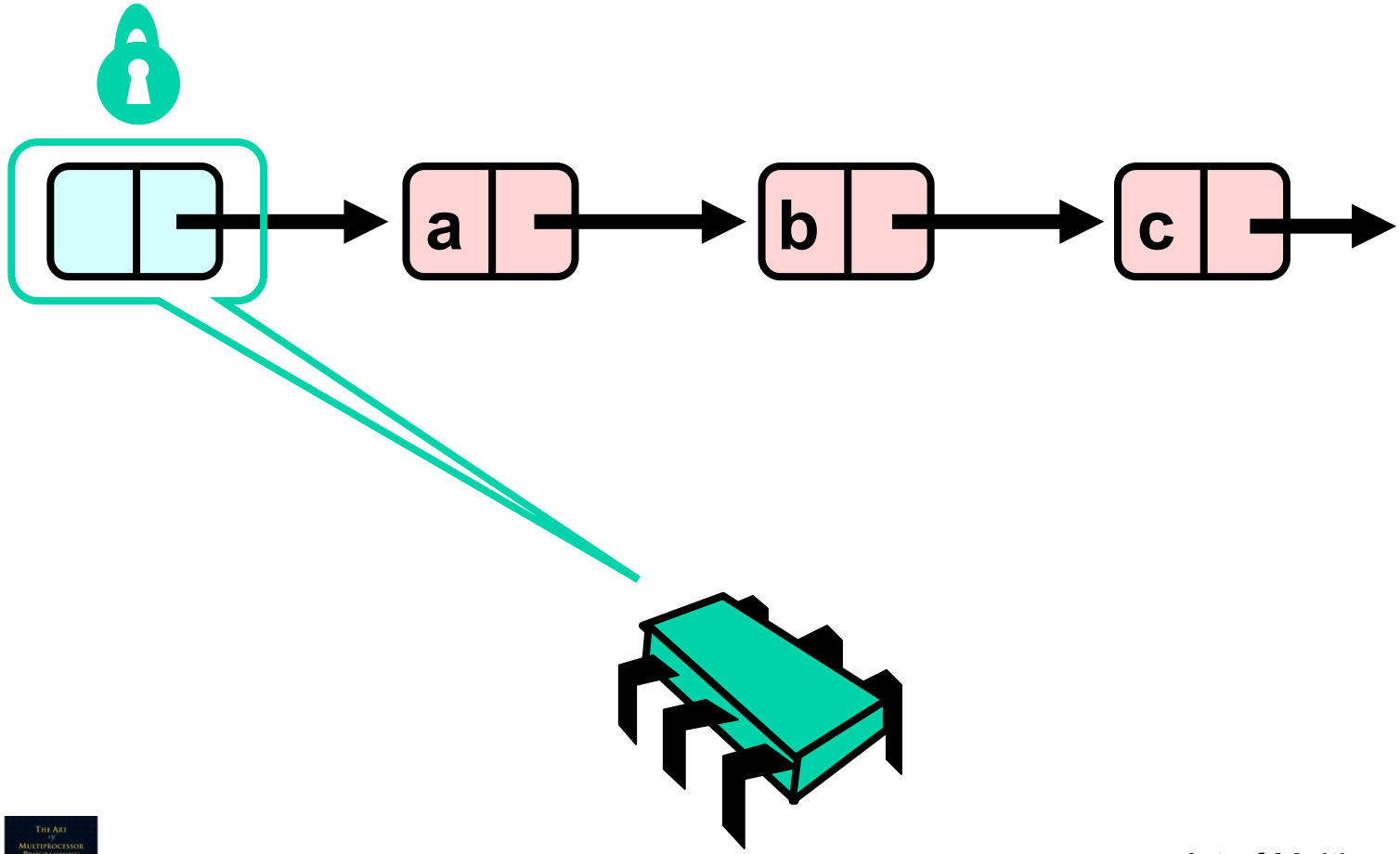


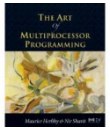
Lock Teleportation



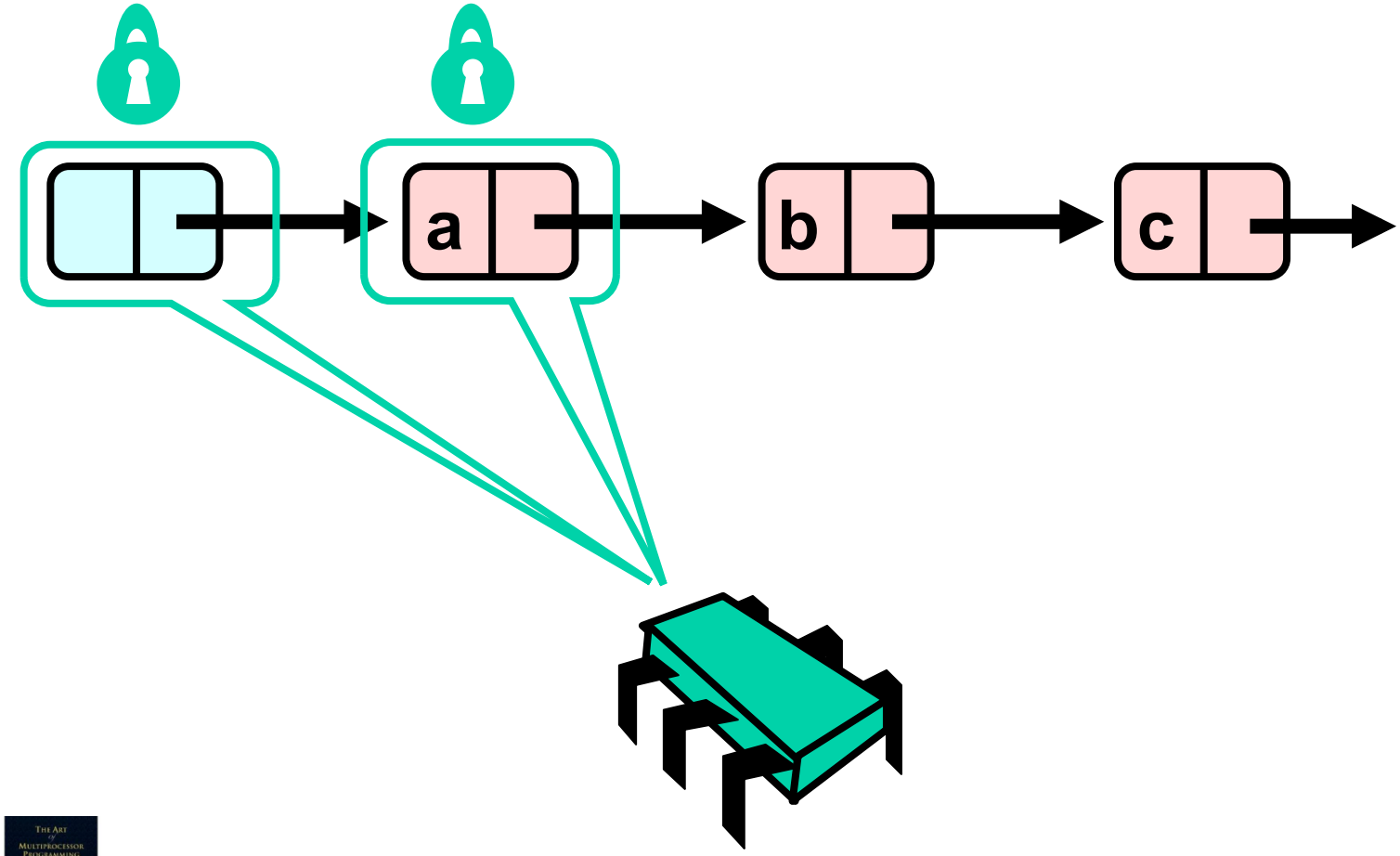


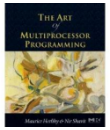
Hand-over-Hand locking



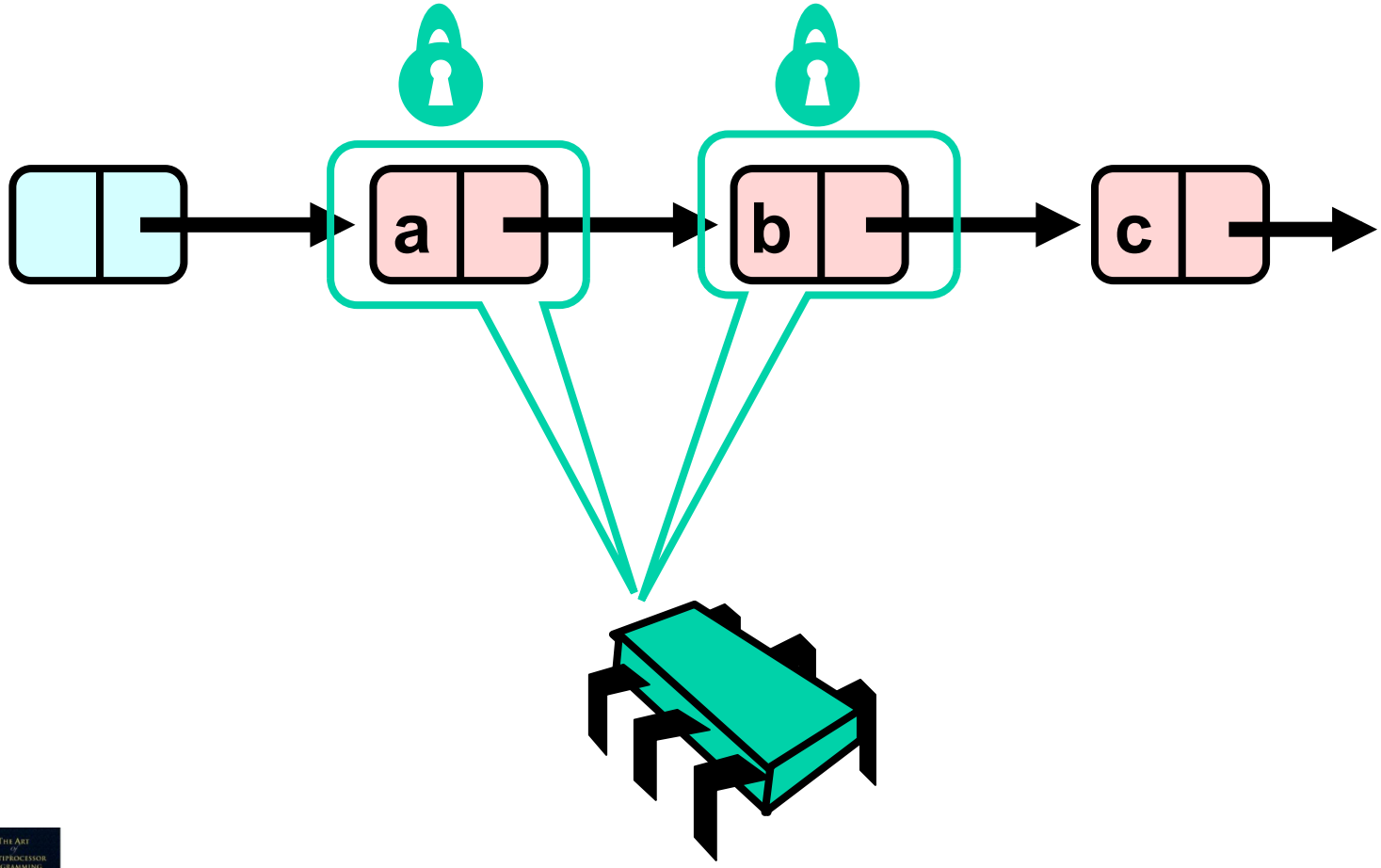


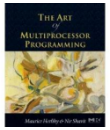
Hand-over-Hand locking



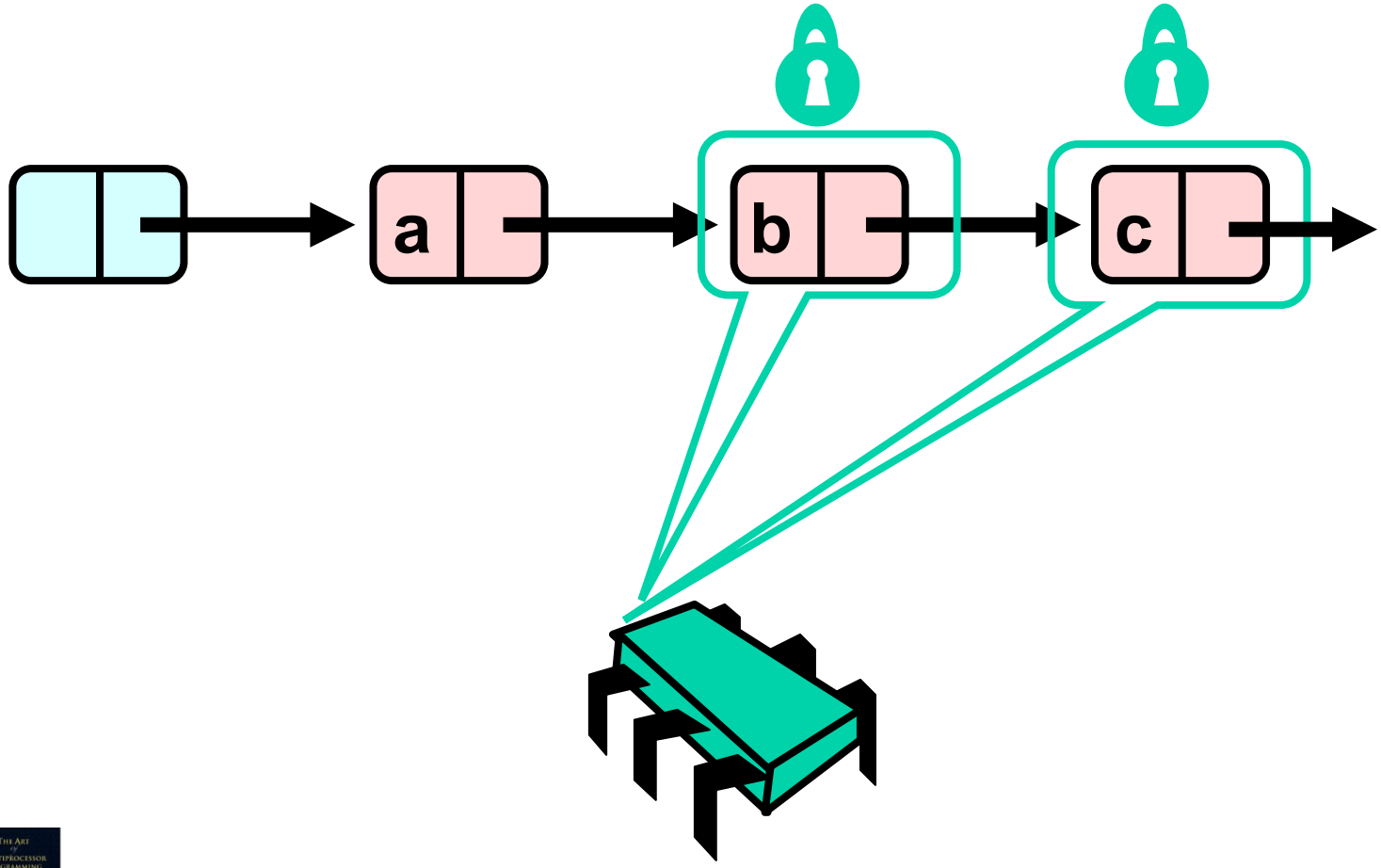


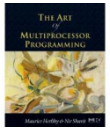
Hand-over-Hand locking



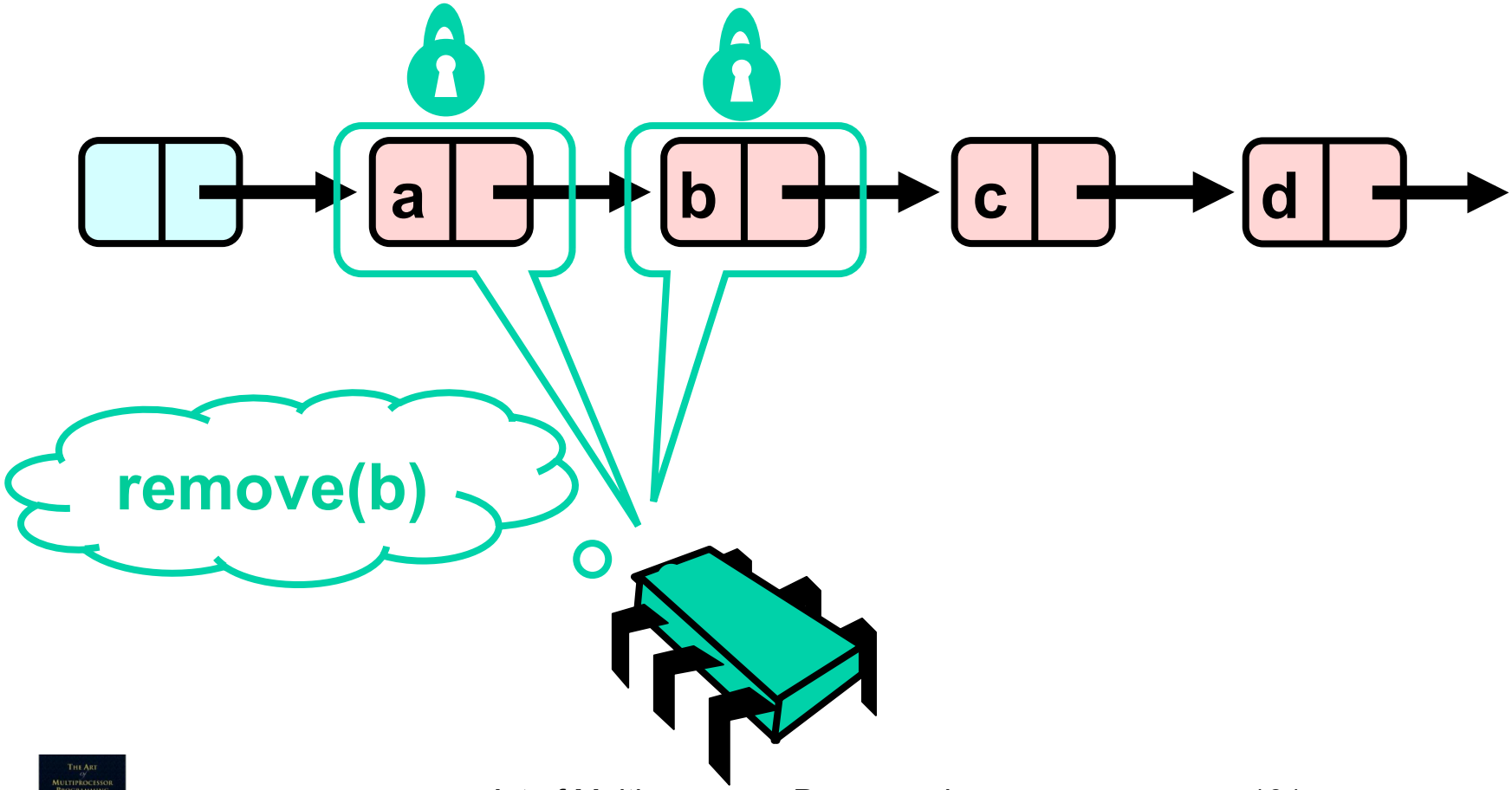


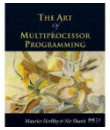
Hand-over-Hand locking



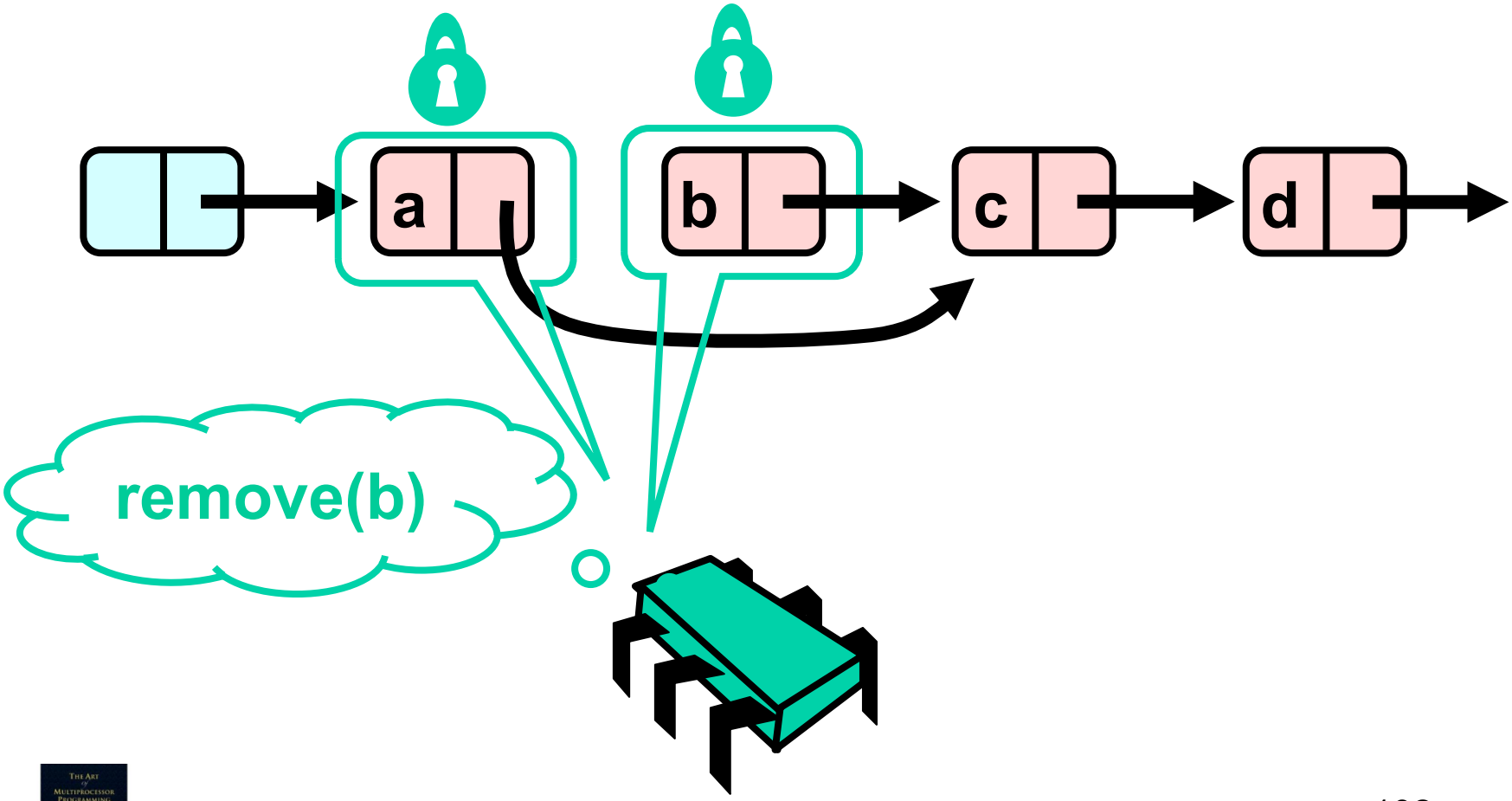


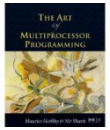
Removing a Node



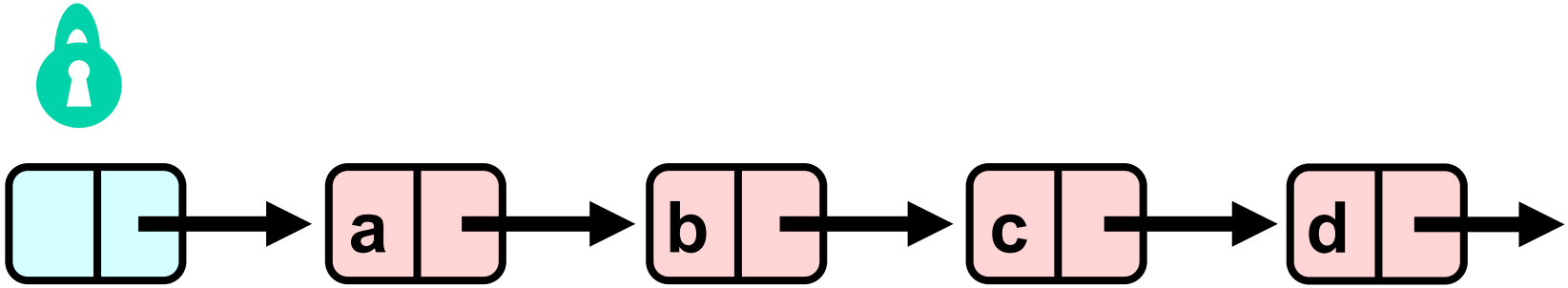


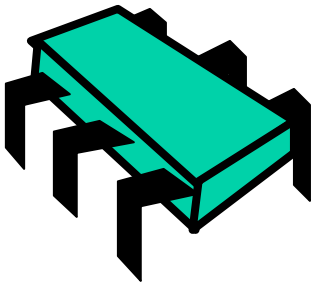
Removing a Node

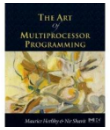




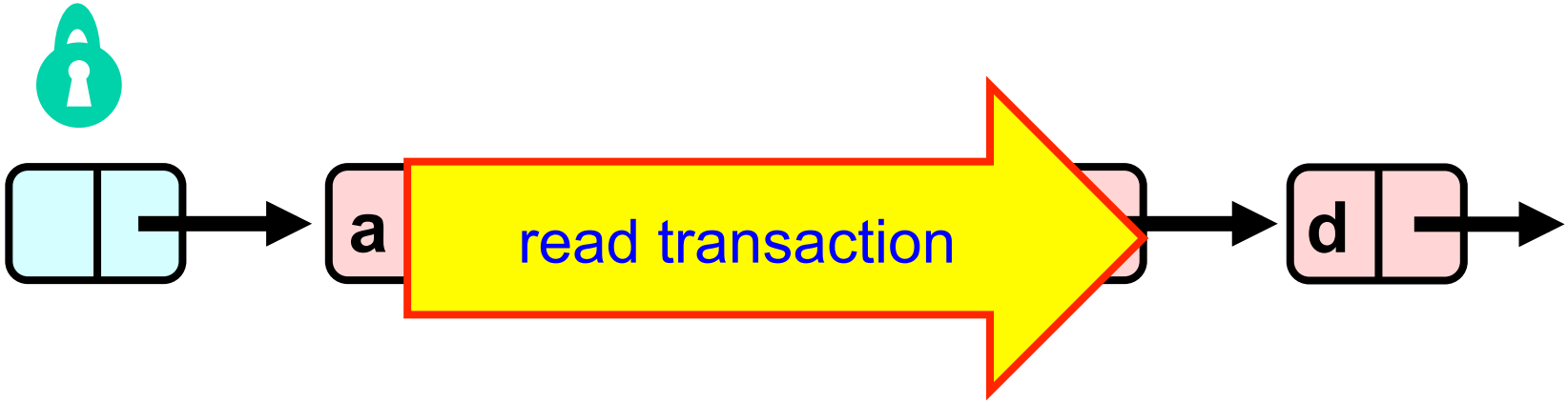
Lock Teleportation

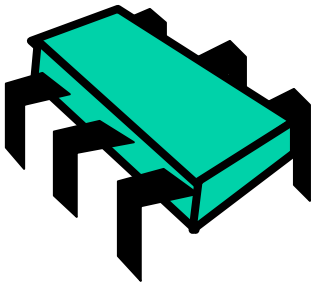


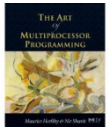




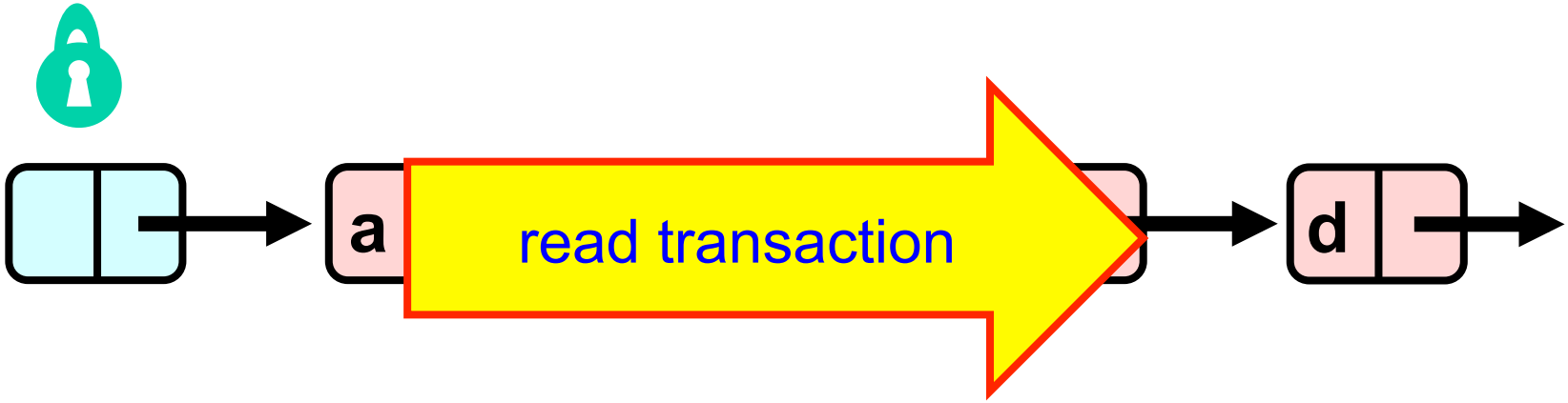
Lock Teleportation

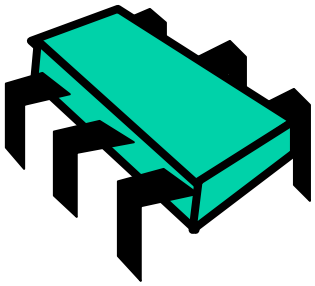


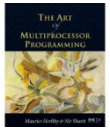




Lock Teleportation

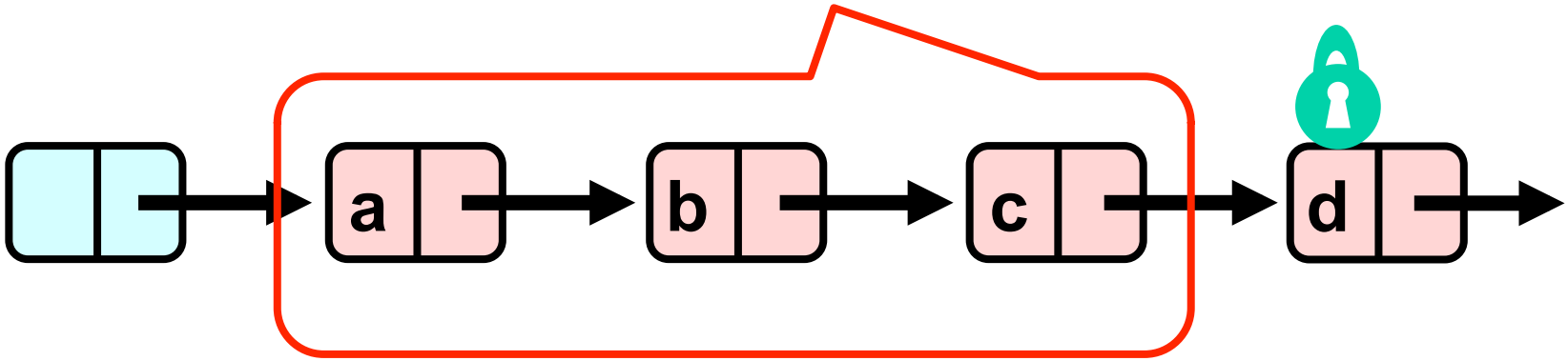


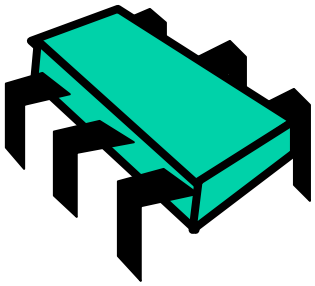


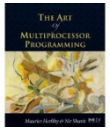


Lock Teleportation

no locks acquired







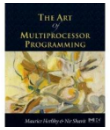
How Far to Teleport?

Too short?

Missed opportunity

Too far?

Transaction aborts, work lost



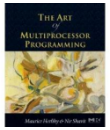
Adaptive Teleportion

On Success:

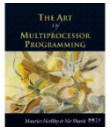
```
limit = limit + 1
```

On Failure:

```
limit = limit / 2
```

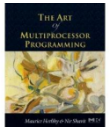


```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    int distance = 0;
    if (xbegin() == _XBEGIN_STARTED) {
      traverse up to teleportLimit nodes
      move lock
      _xend();
      teleportLimit++;
      return pred;
    } else {
      teleportLimit = teleportLimit/2
    }}};
```



```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    i
    i                              BEGIN_STARTED) {
                                   eleportLimit nodes
      move lock
      _xend();
      teleportLimit++;
      return pred;
    } else {
      teleportLimit = teleportLimit/2
    }}};
```

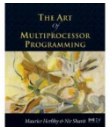
locked node
In sorted list



```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    i
    i                                  BEGI
                                       elep                    s
      move lock
      _xend();
      teleportLimit++;
      return pred;
    } else {
      teleportLimit = teleportLimit/2
    }}};
```

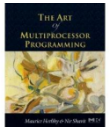
locked node In sorted list

Value to search for



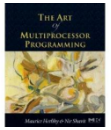
```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    int
    if
      t
      move lock
      _xend();
      teleportLimit++;
      return pred;
    } else {
      teleportLimit = teleportLimit/2
    }}};
```

Returns locked node with value less than or equal to v



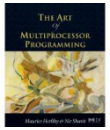
```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    int distance = 0;
    if (xbegin() == _XBEGIN_STARTED) {
      traverse up to teleportLimit nodes
      m
      _
      teleportLimit++;
      return pred;
    } else {
      teleportLimit = teleportLimit/2
    }}};
```

Try for a fixed number of times



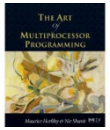
```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    int distance = 0;
    if (xbegin() == _XBEGIN_STARTED) {
      traverse up to teleportLimit nodes
      move lock
      _xend();
      teleportLimit++;
      return pred;
    } else {
      teleport
    }}};
```

Executed as read-only transaction



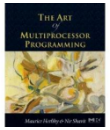
```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    int distance = 0;
    if (xbegin() == _XBEGIN_STARTED) {
      traverse up to teleportLimit nodes
      move lock
      _xend();
      telep
      retur
    } else
      telep
    }}};
```

Thread-local variable that controls how far to traverse the list



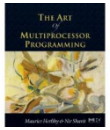
```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    int distance = 0;
    if (xbegin() == _XBEGIN_STARTED) {
      traverse up to teleportLimit nodes
      move lock
      _xend();
      telep
      retur
    } else
      telep
    }}};
```

Stop if either (1) we find value `v`, or (2) we traverse `teleportLimit` nodes



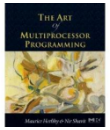
```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    int distance = 0;
    if (xbegin() == _XBEGIN_STARTED) {
      traverse up to teleportLimit nodes
      move lock
      _xend();
      teleportLimit++;
                                    nit/2
    }}};
```

Unlock starting node, lock final node



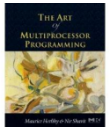
```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    int distance = 0;
    if (xbegin() == _XBEGIN_STARTED) {
      traverse up to teleportLimit nodes
      move lock
      _xend();
      teleportLimit++;
      return pred;
    }}};
```

Try to commit transaction



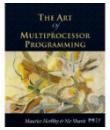
```
Node* teleport(Node* start, T v) {
  int retries
  while (--ret
    int distan
    if (xbegin
      traverse list up to threshold
      move lock
       xend();
      teleportLimit++;
      return last node;
    } else {
      teleportLimit = teleportLimit/2
    }}};
```

On commit, advance `teleportLimit` by 1, and return locked node



```
Node* teleport(Node* start, T v) {
  int retries = RETRY_THRESHOLD;
  while (--retries) {
    int distance = 0;
    if (xbegin() == _XBEGIN_STARTED) {
      traverse
      move lock
      _xend();
      teleportLimit++;
      return last node;
    } else {
      teleportLimit = teleportLimit/2
    }}};
```

On abort, cut `teleportLimit` in half

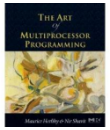


Lock-Based STMs

STMs come in different forms:

Lock-Free

Lock-based

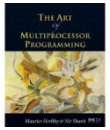


Lock-Based STM

But, didn't you just say that locks are evil?

For applications, yes!

For run-time systems written by experts, maybe not ….



Lock-Based STMs

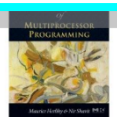
Each transaction keeps

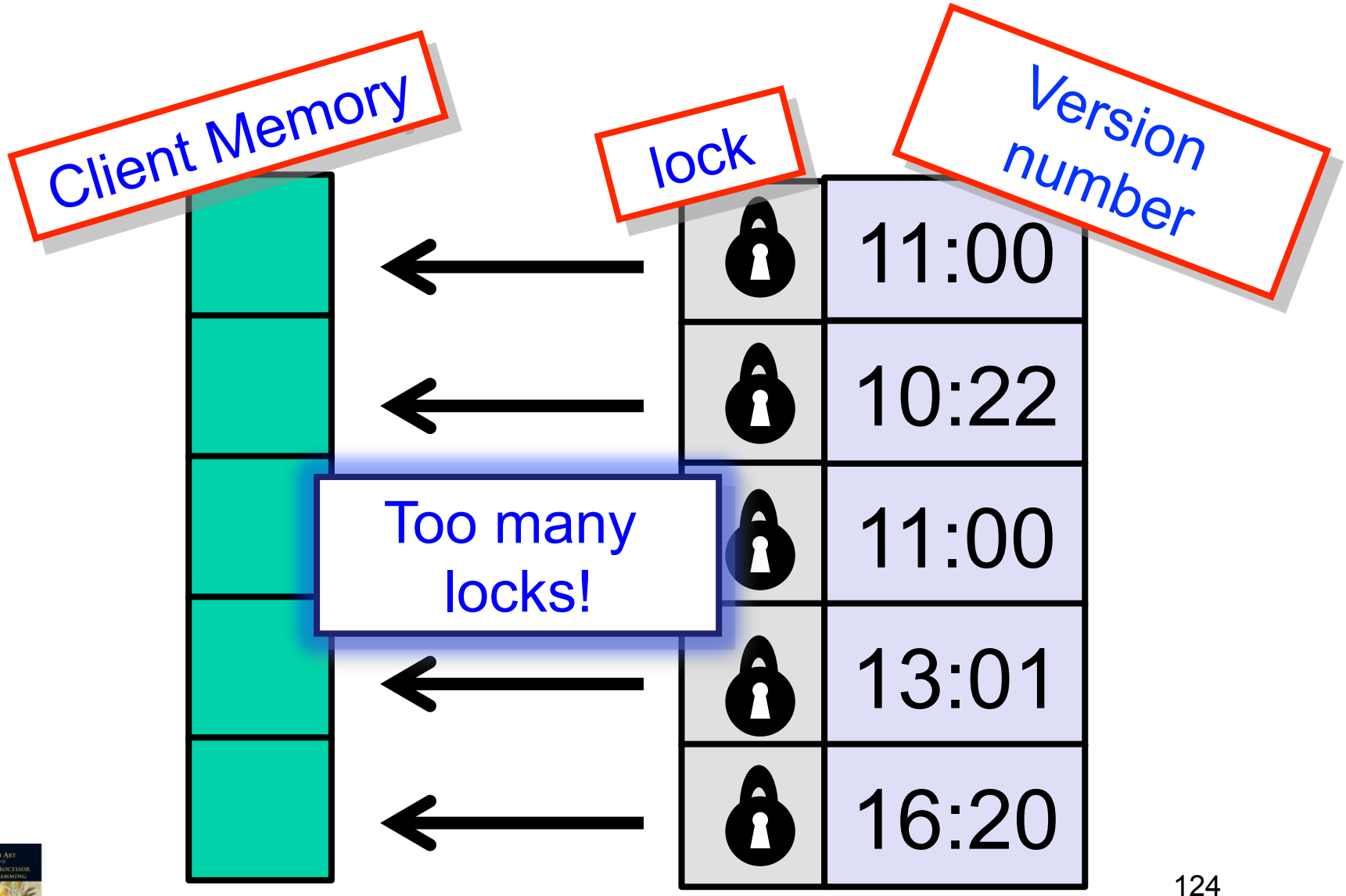
Read Set: locations and values read

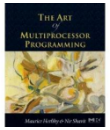
Write Set: locations and values written

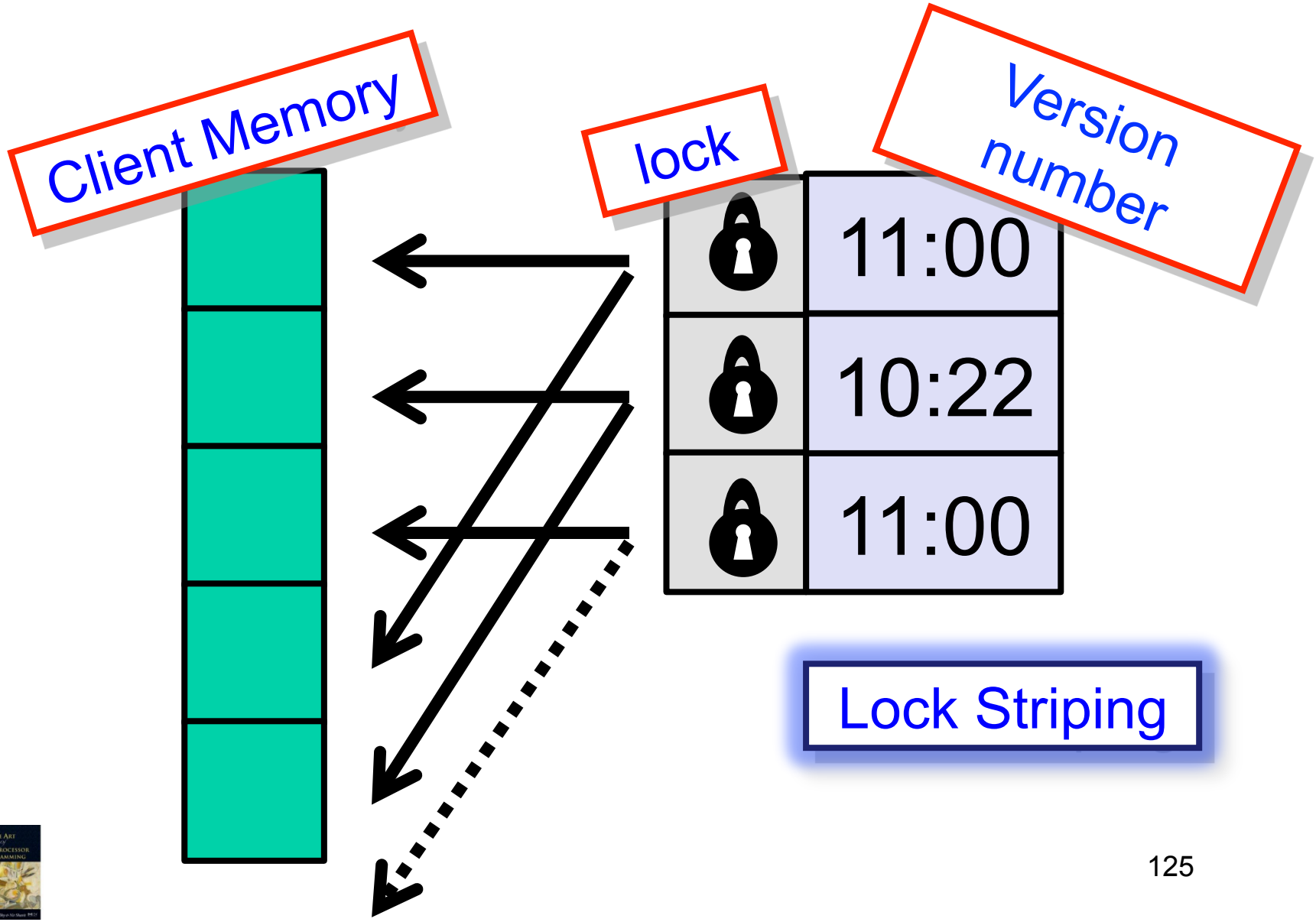
Changes installed at commit

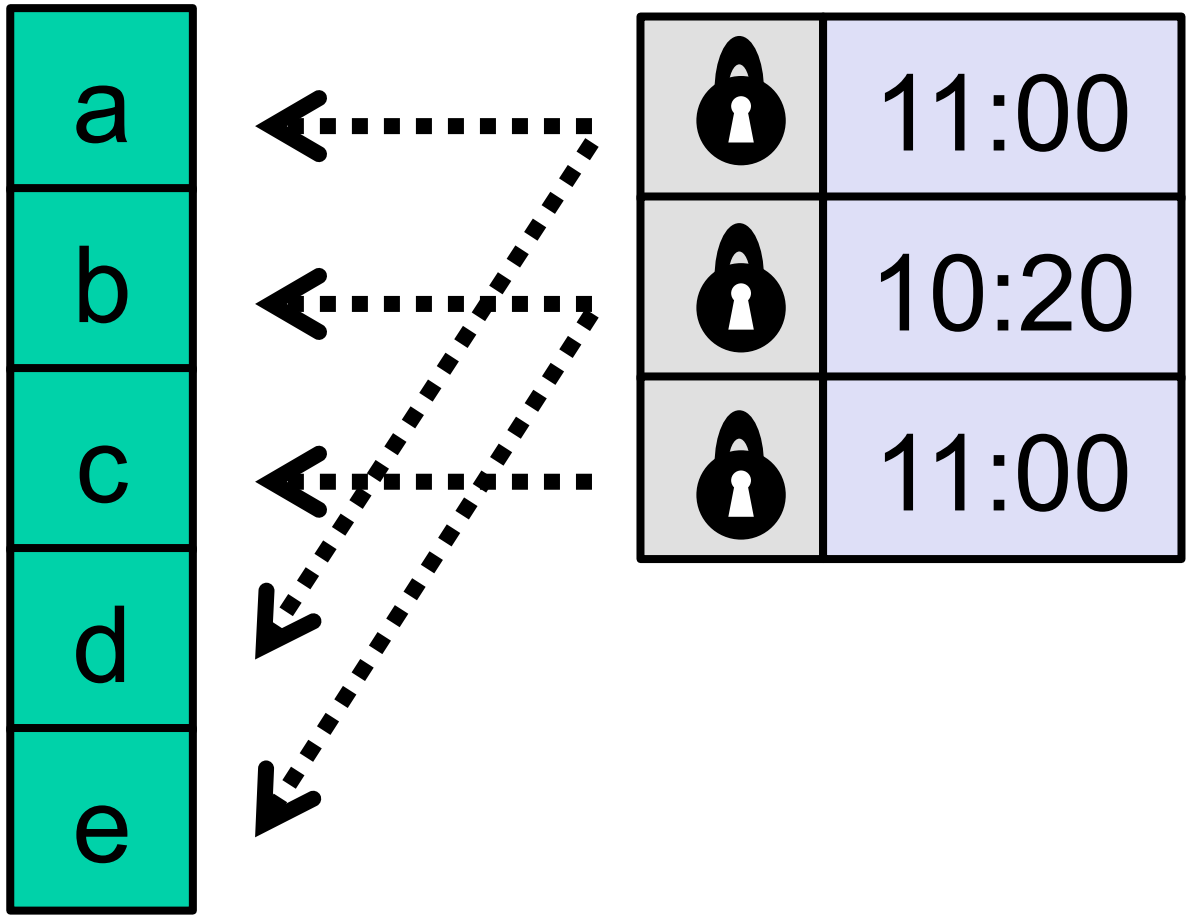
Conflicts detected at comit

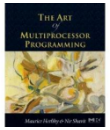


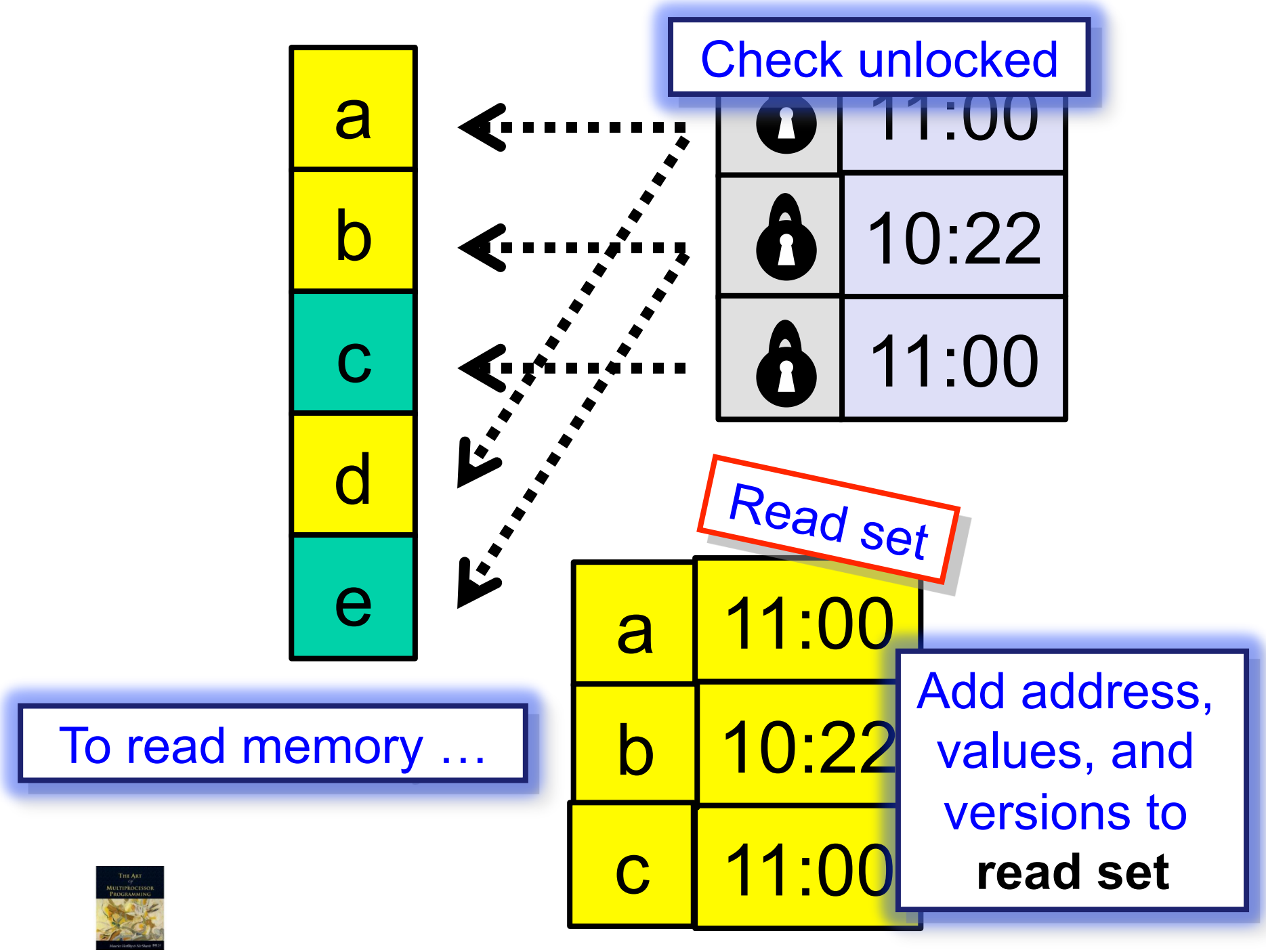

Client Memory
lock
Version number
11:00
10:22
11:00
13:01
16:20
Too many locks!

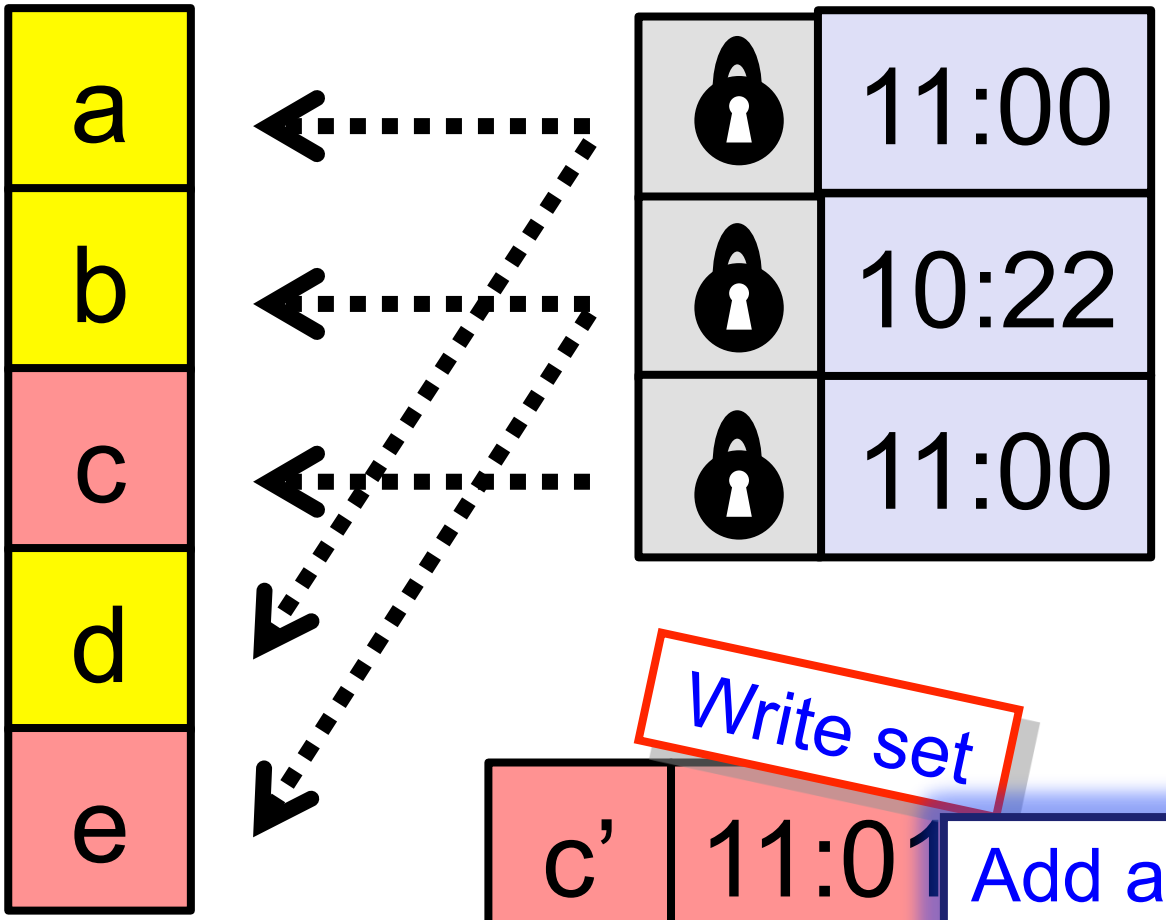


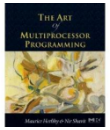

Client Memory
lock
Version number
11:00
10:22
11:00
Lock Striping

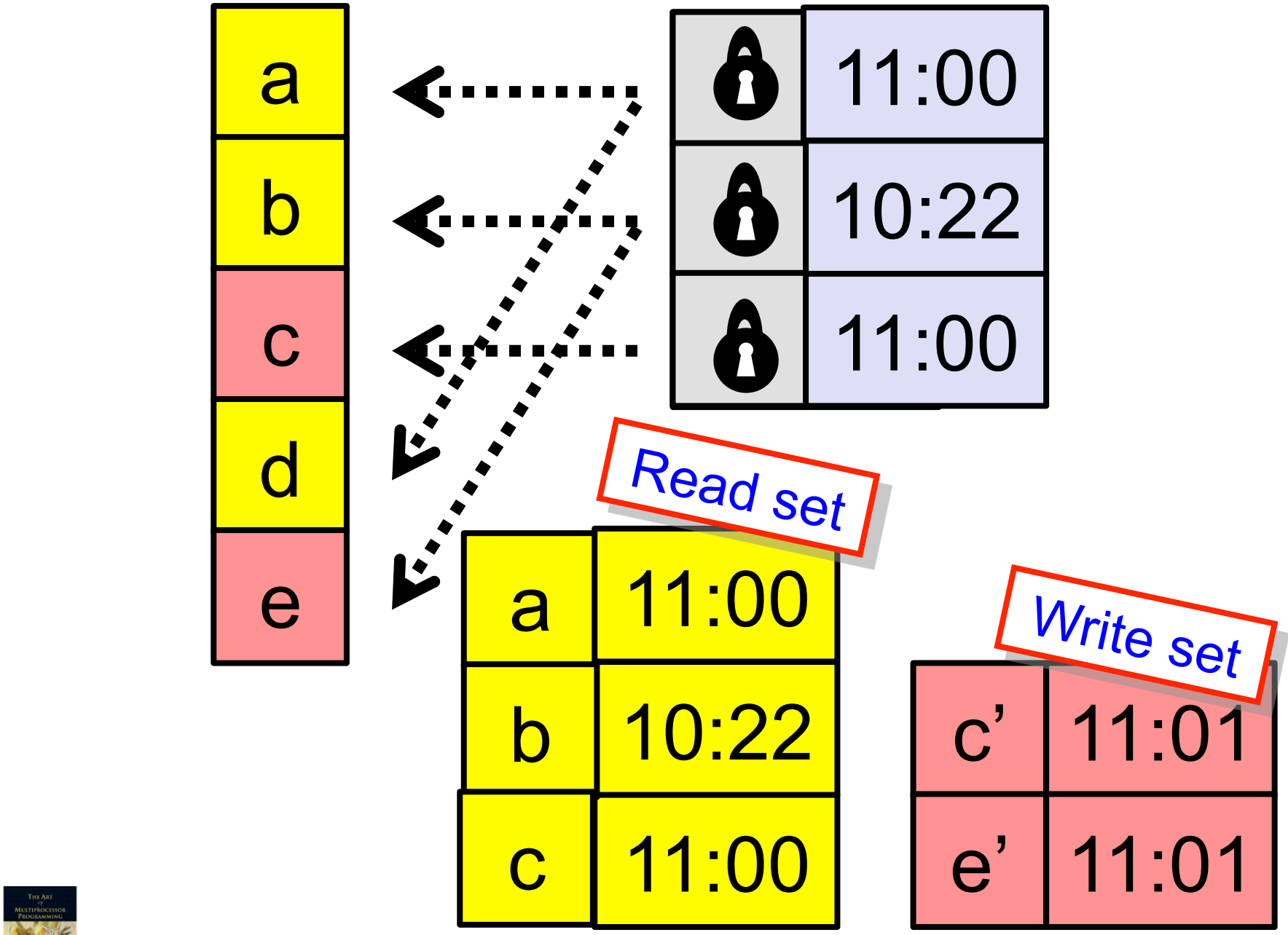

a
b
c
d
e
11:00
10:20
11:00

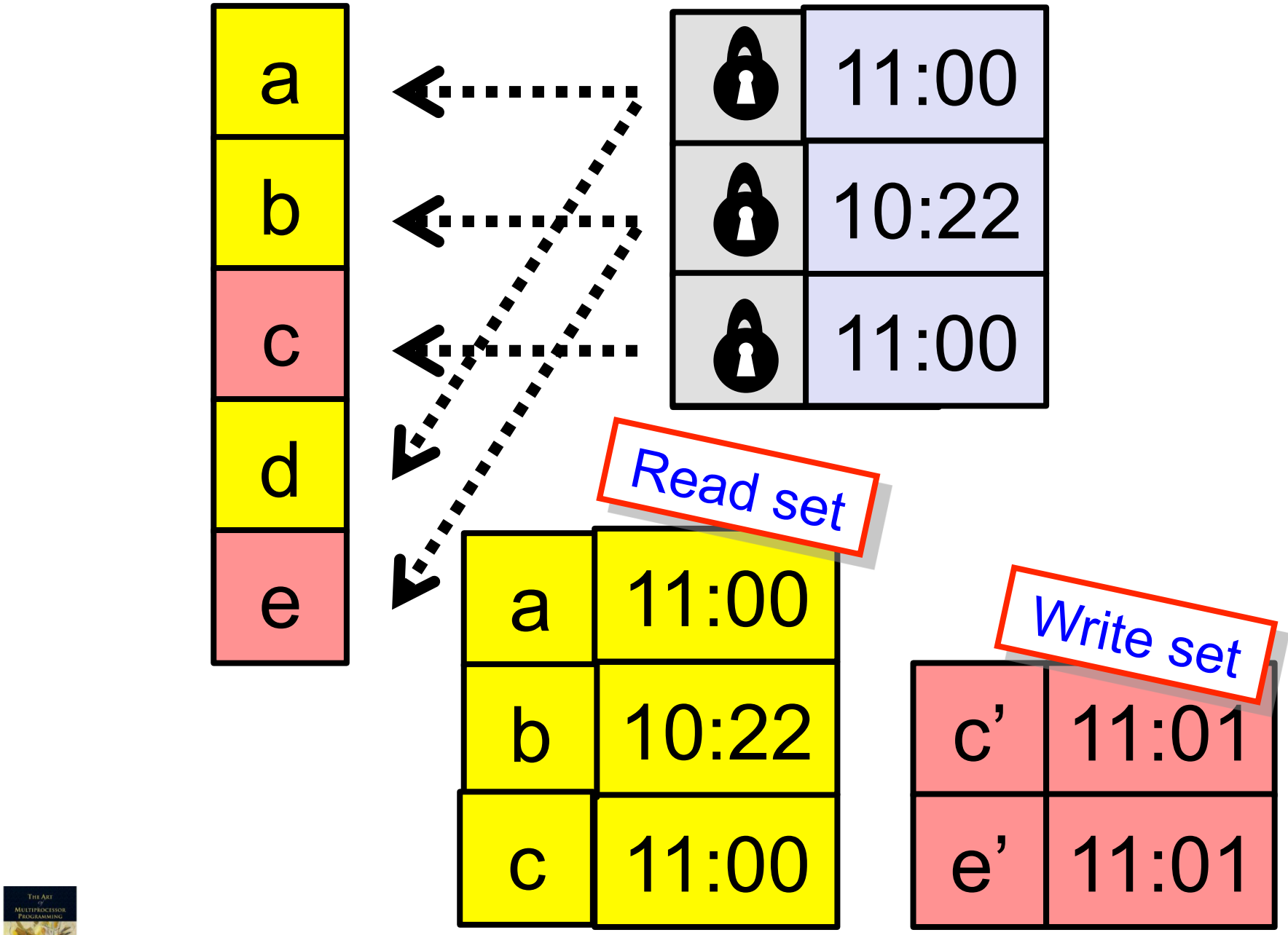


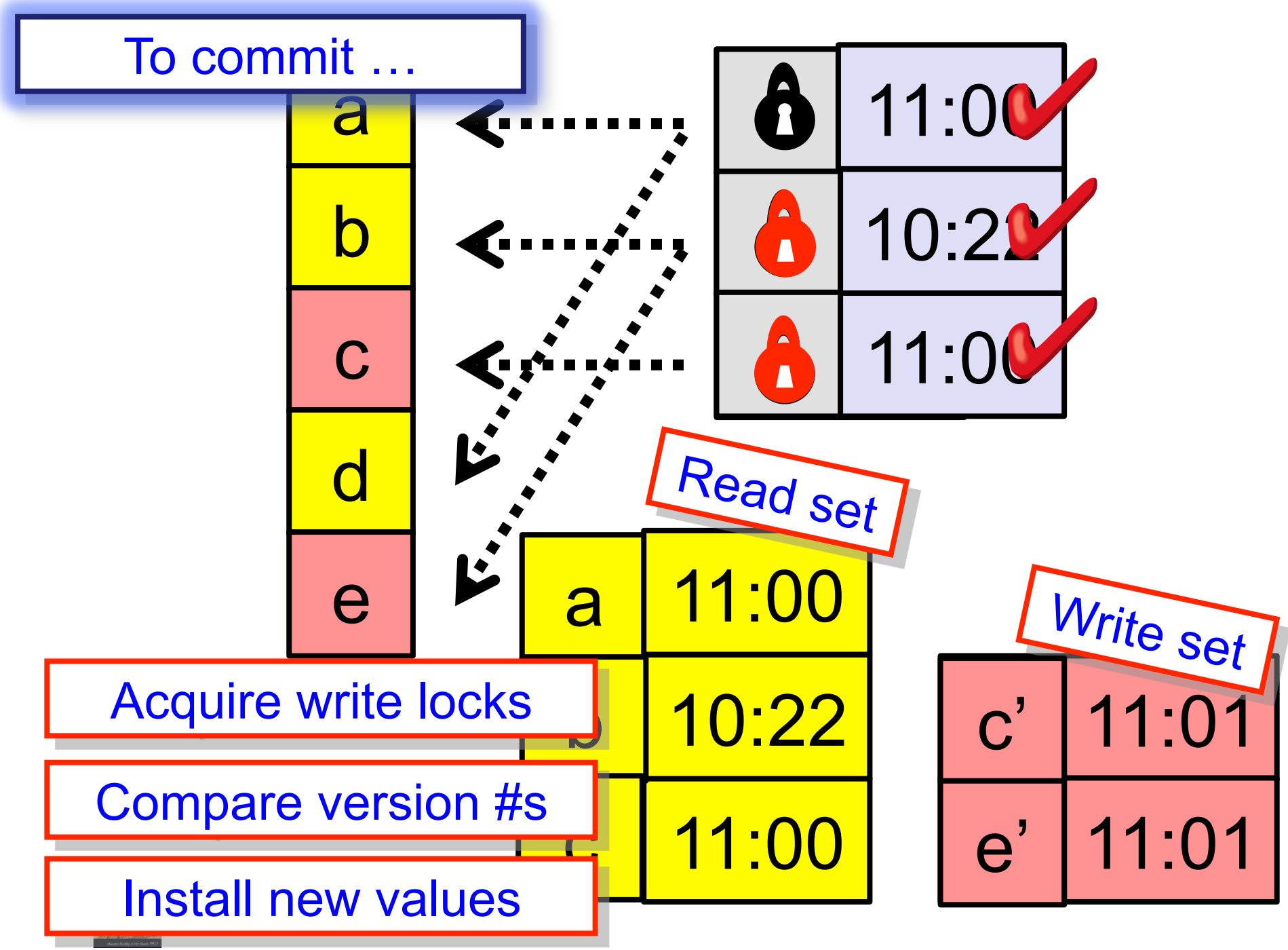

Check unlocked
a
b
c
d
e
11:00
10:22
11:00
Read set
a
11:00
b
10:22
c
11:00
To read memory …
Add address, values, and versions to **read set**

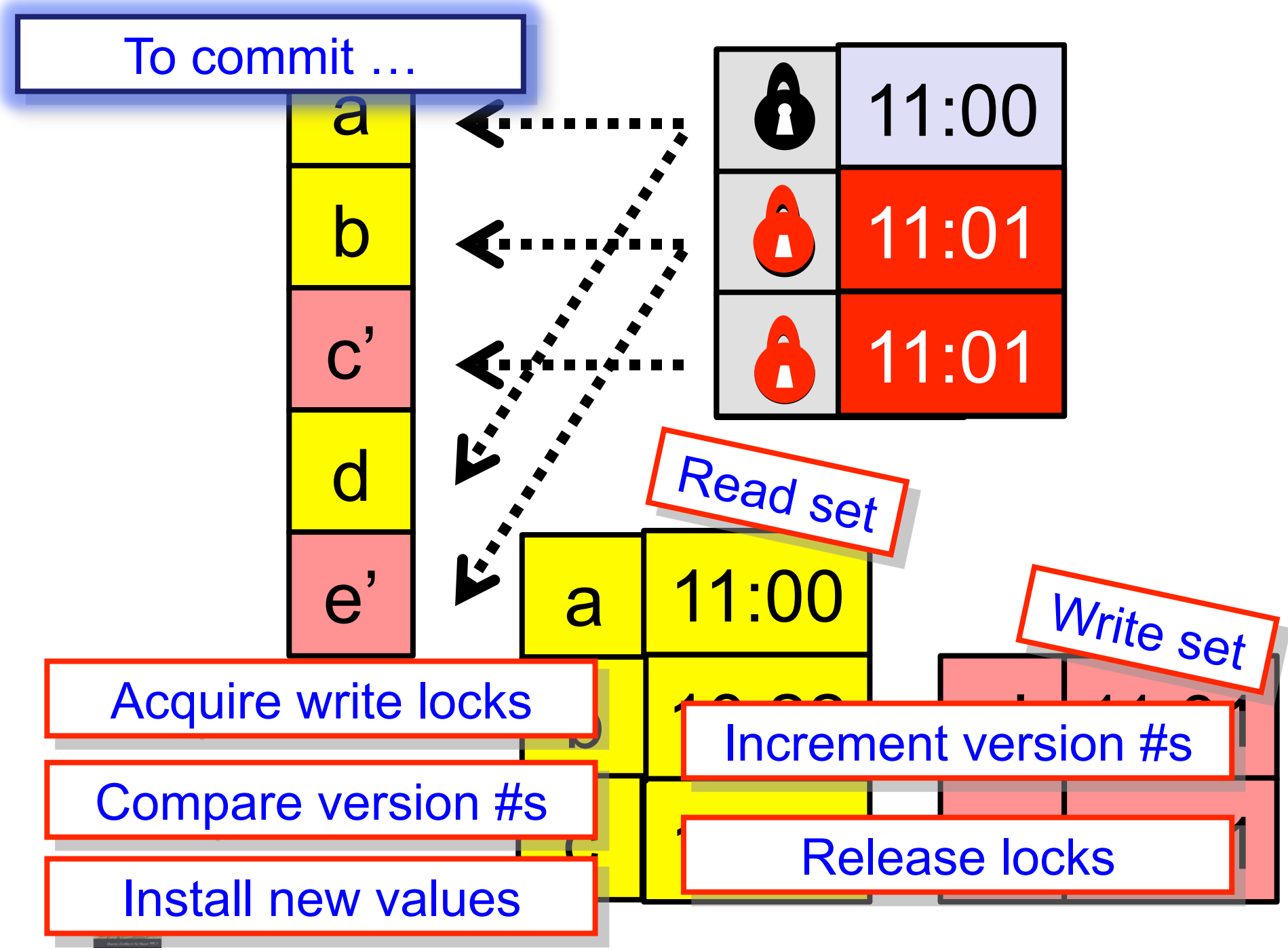

a
b
c
d
e
11:00
10:22
11:00
Write set
c’
11:01
e’
11:01
Add address, new values and versions to **write set**
To write memory …




a
b
c
d
e
11:00
10:22
11:00
Read set
a
11:00
b
10:22
c
11:00
Write set
c’
11:01
e’
11:01


a
b
c
d
e
11:00
10:22
11:00
Read set
a
11:00
b
10:22
c
11:00
Write set
c’
11:01
e’
11:01


To commit …
a
b
c
d
e
11:00
10:22
11:00
Read set
a
11:00
10:22
11:00
Write set
c’
11:01
e’
11:01
Acquire write locks
Compare version #s
Install new values

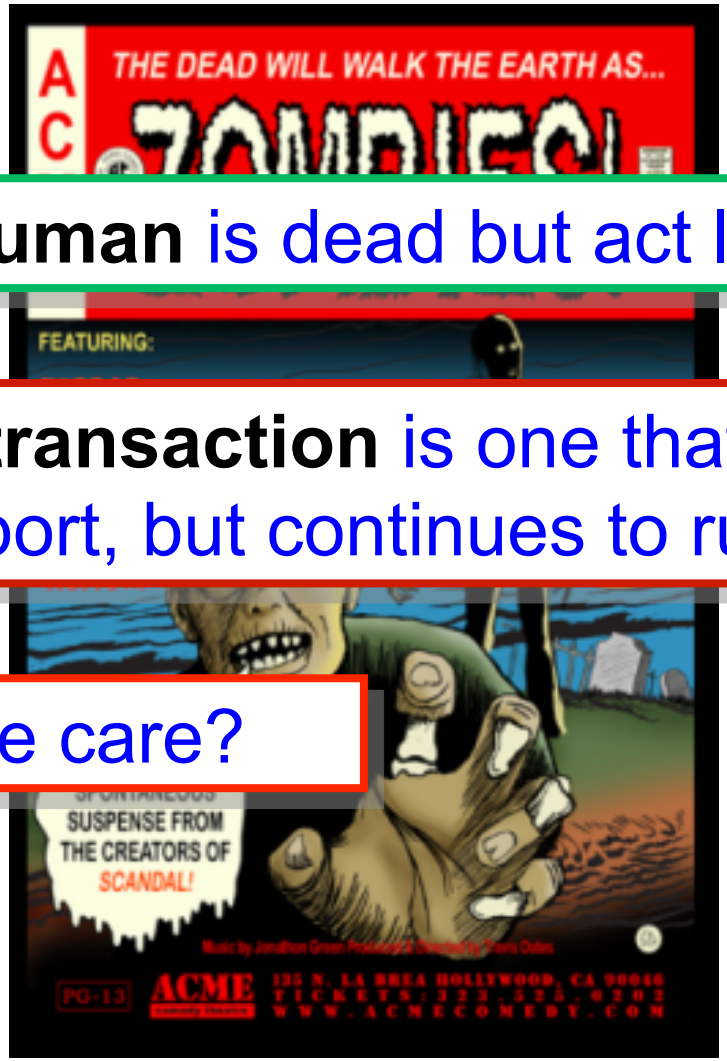

To commit …
a
b
c’
d
e’
11:00
11:01
11:01
Read set
a
11:00
Write set
Acquire write locks
Compare version #s
Install new values
Increment version #s
Release locks

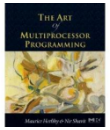
Zombie Transactions

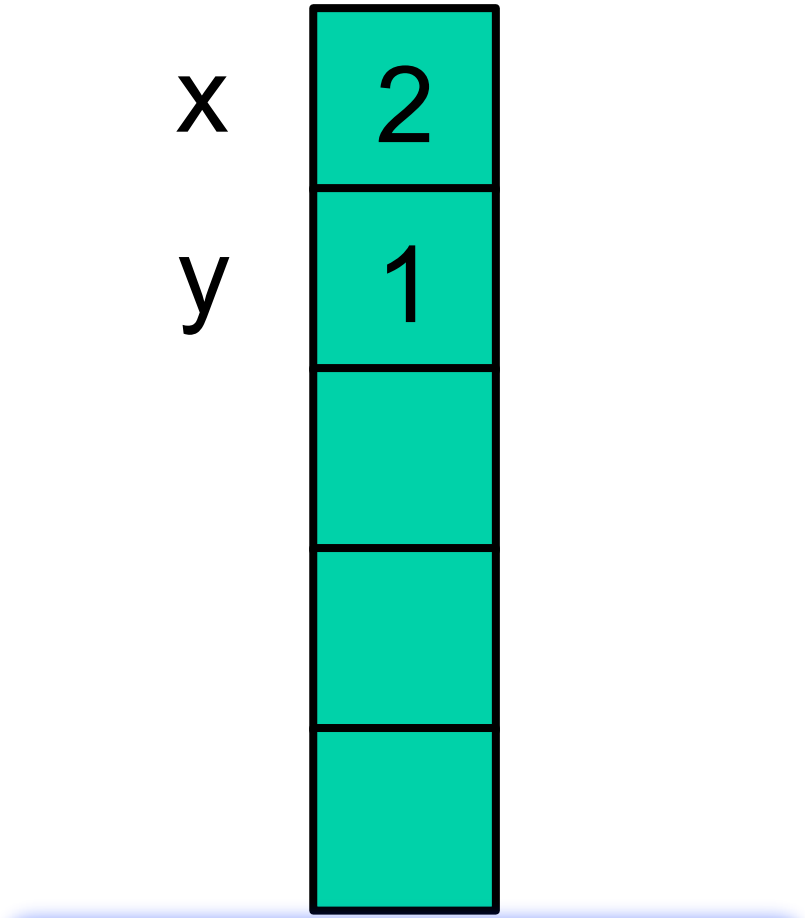
A **zombie human** is dead but act like it is alive …

A **zombie transaction** is one that will certainly abort, but continues to run …

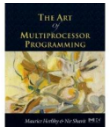
Why do we care?

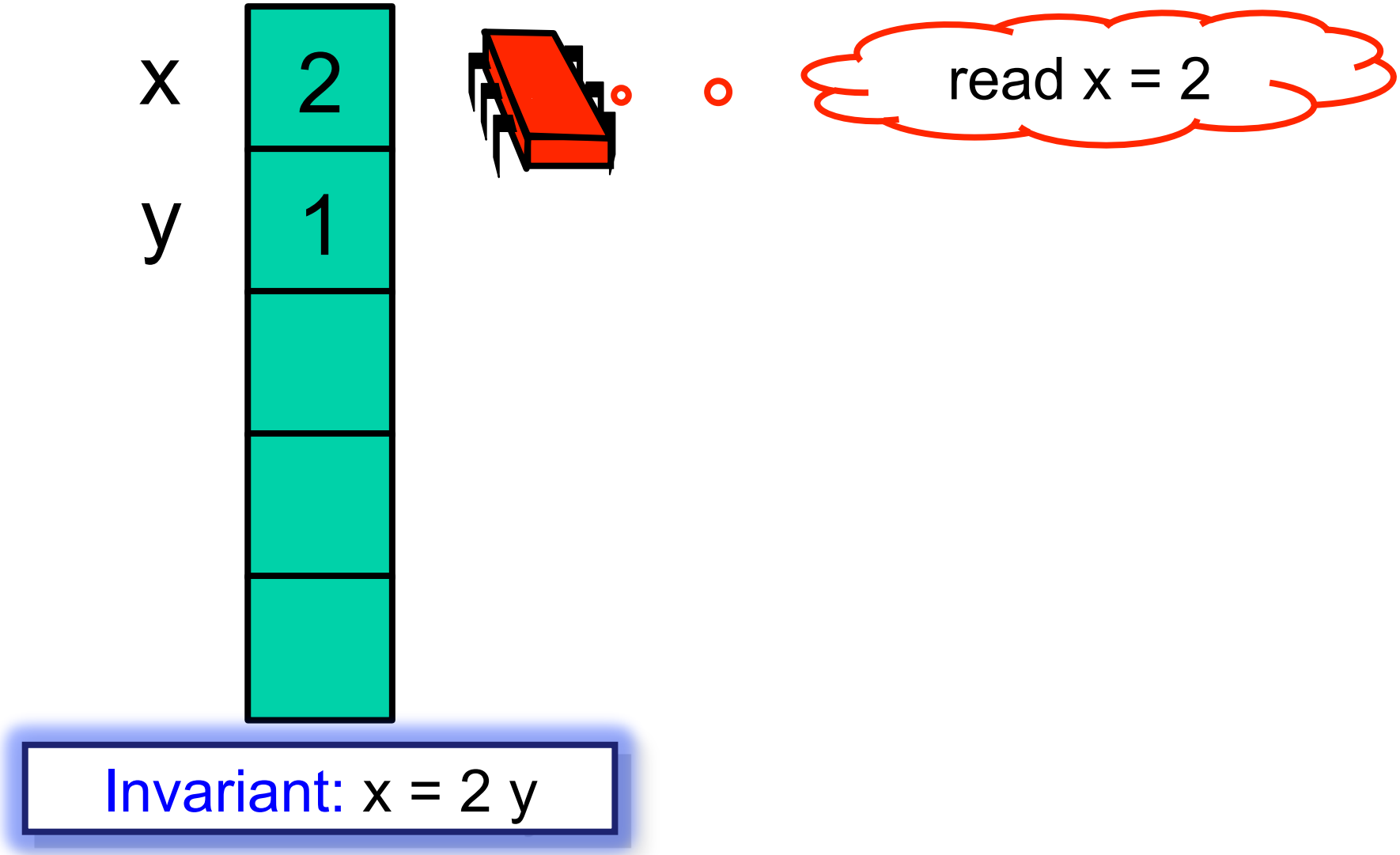


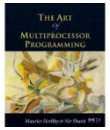


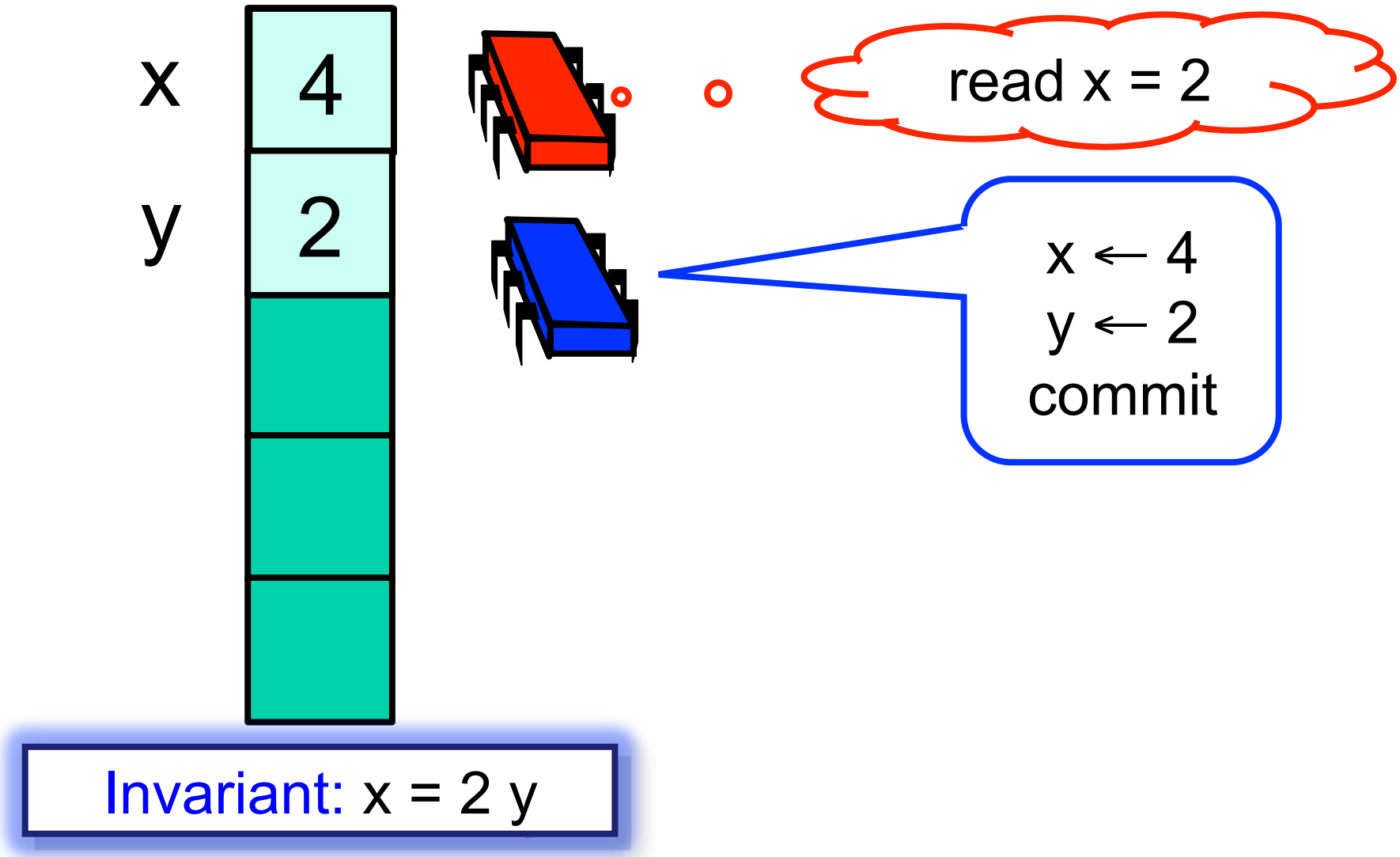


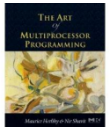
Invariant: x = 2 y

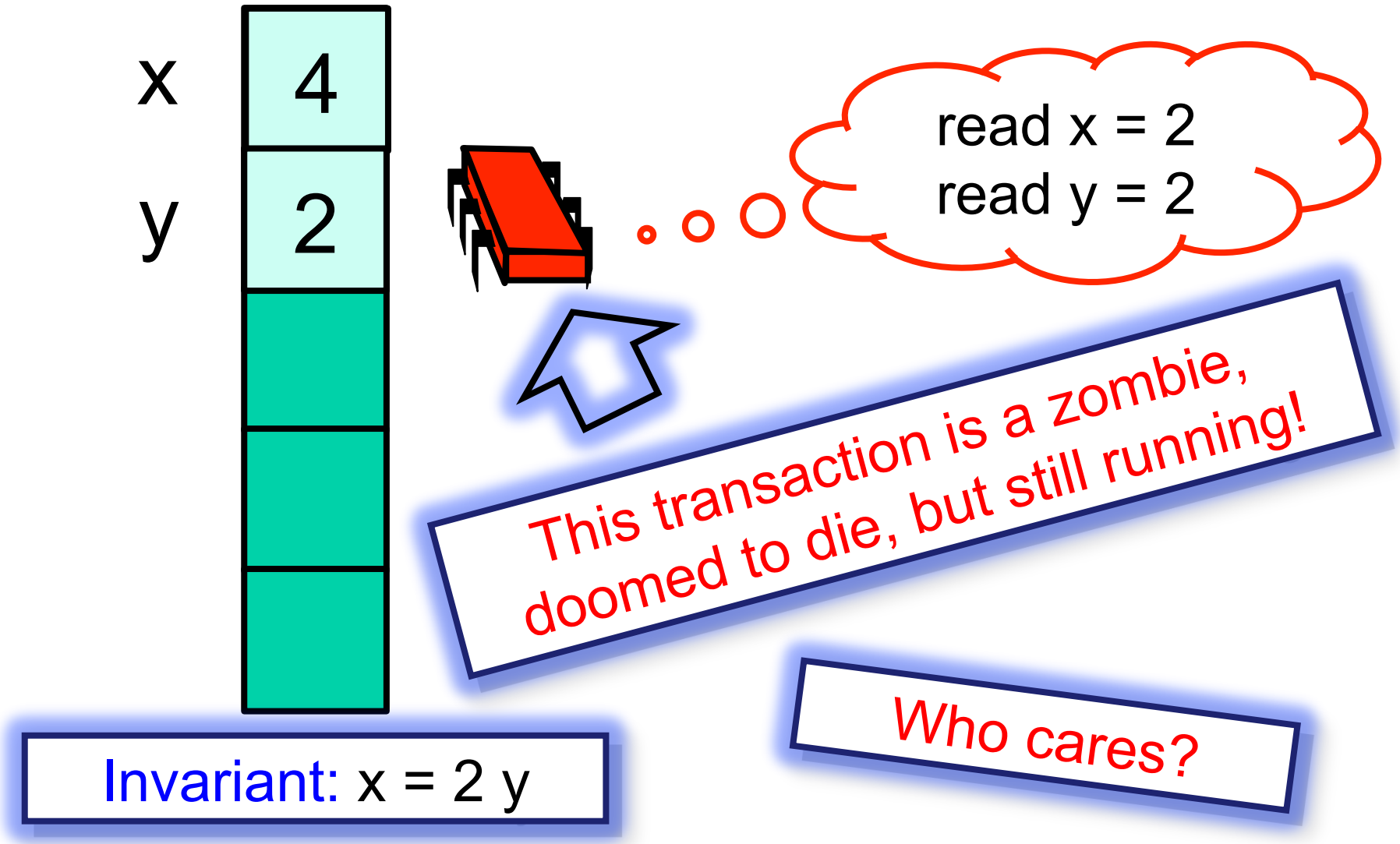


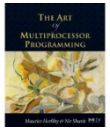

x
2
y
1
read x = 2
Invariant: x = 2 y

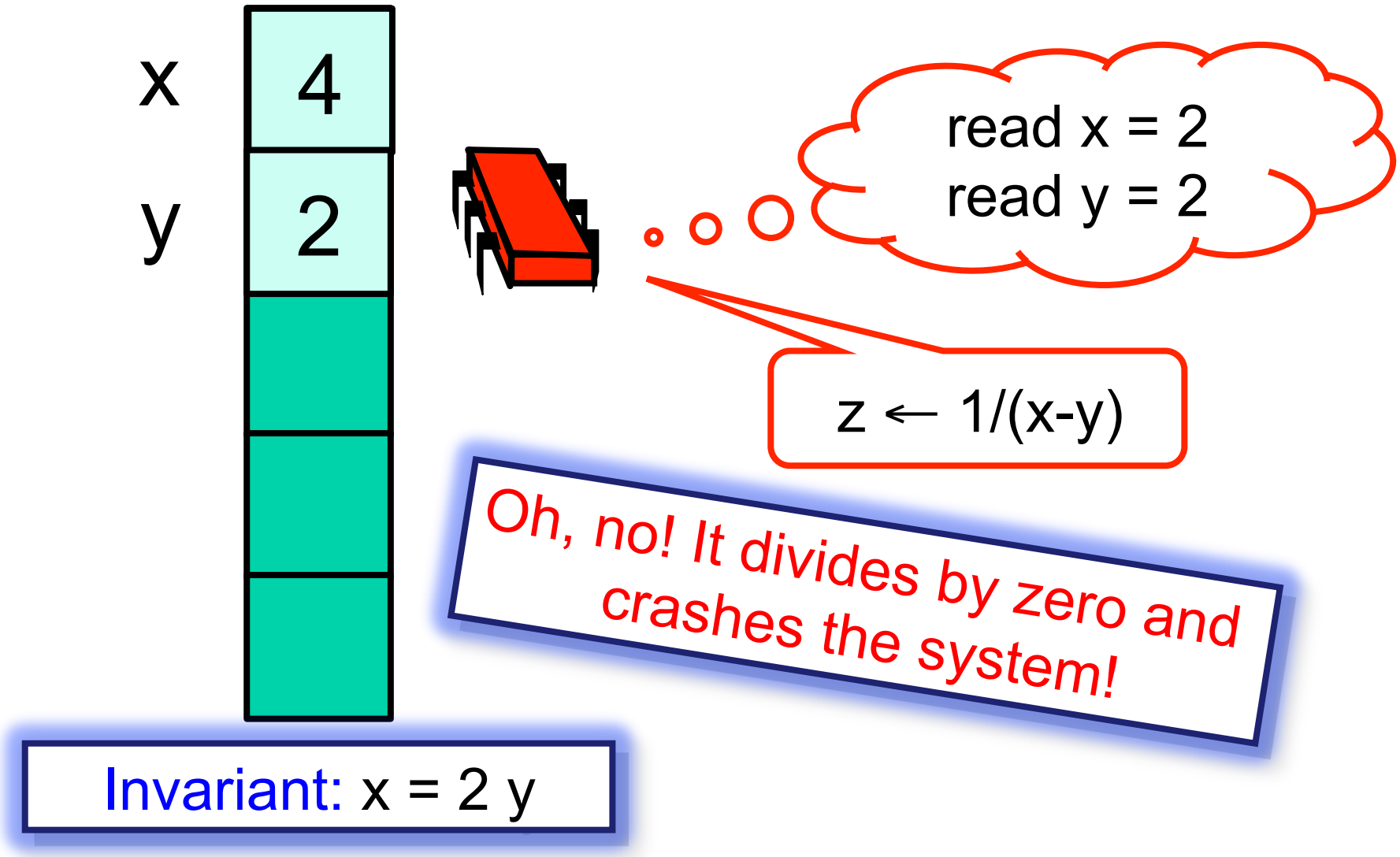


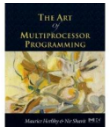

x
4
y
2
read x = 2
x ← 4
y ← 2
commit
Invariant: x = 2 y

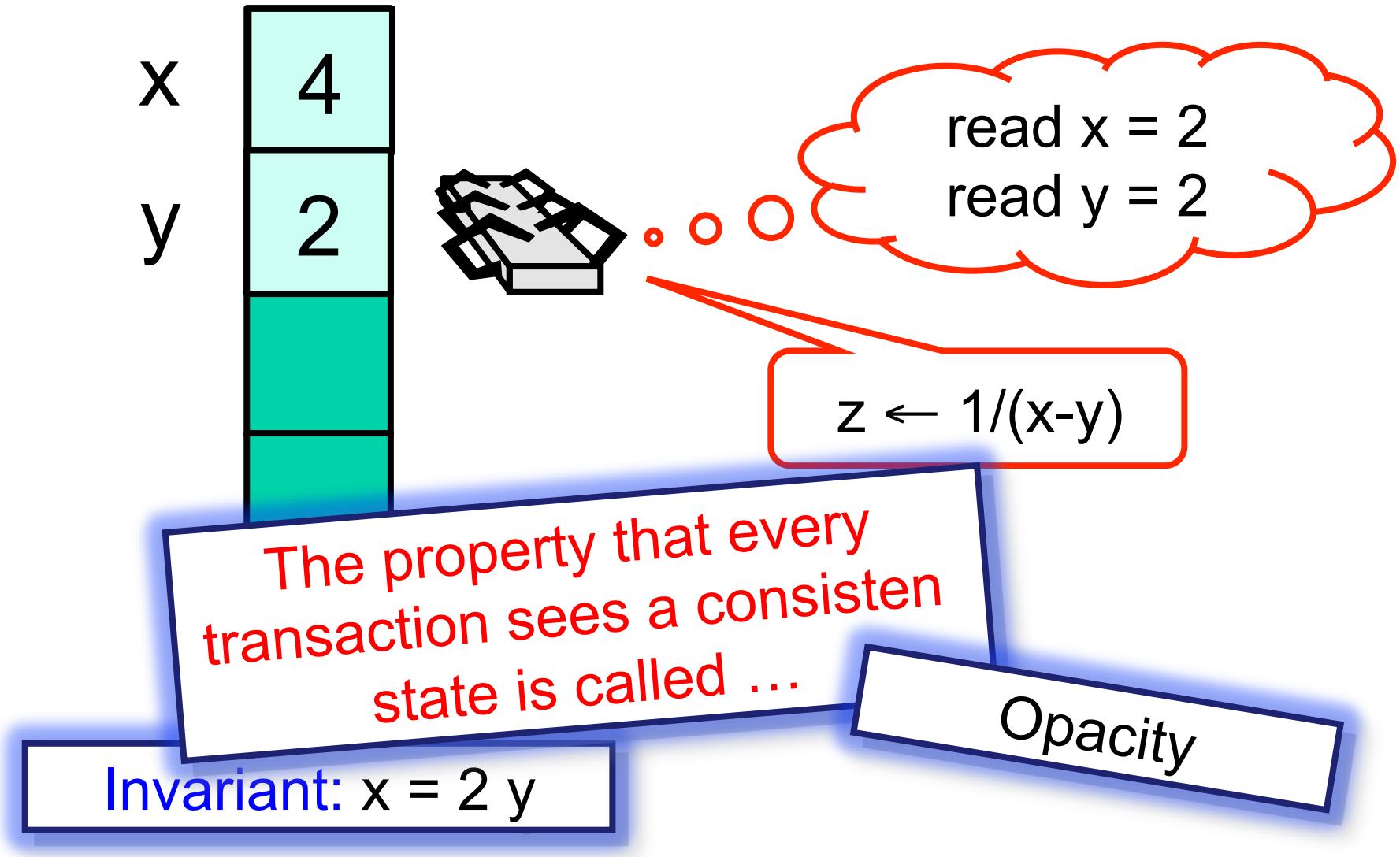


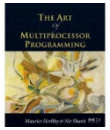

x
4
y
2
read x = 2
read y = 2
This transaction is a zombie, doomed to die, but still running!
Who cares?
Invariant: x = 2 y



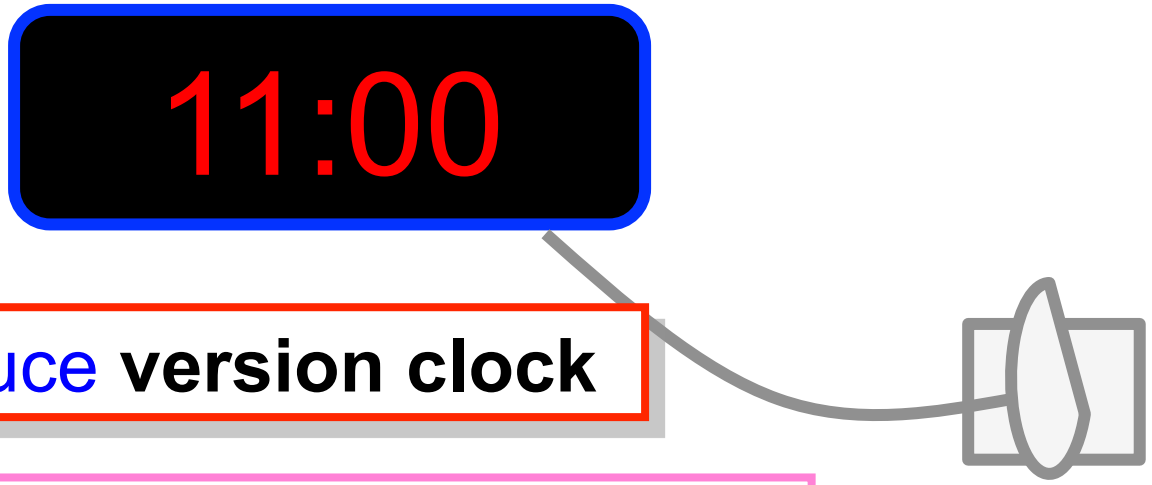

x
4
y
2
read x = 2
read y = 2
z ← 1/(x-y)
Oh, no! It divides by zero and crashes the system!
Invariant: x = 2 y




x
4
y
2
read x = 2
read y = 2
z ← 1/(x-y)
The property that every transaction sees a consisten state is called …
Opacity
Invariant: x = 2 y

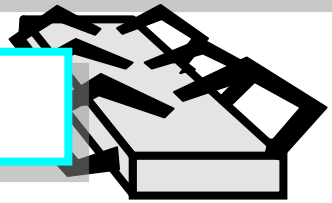


Version Clock



Introduce **version clock**

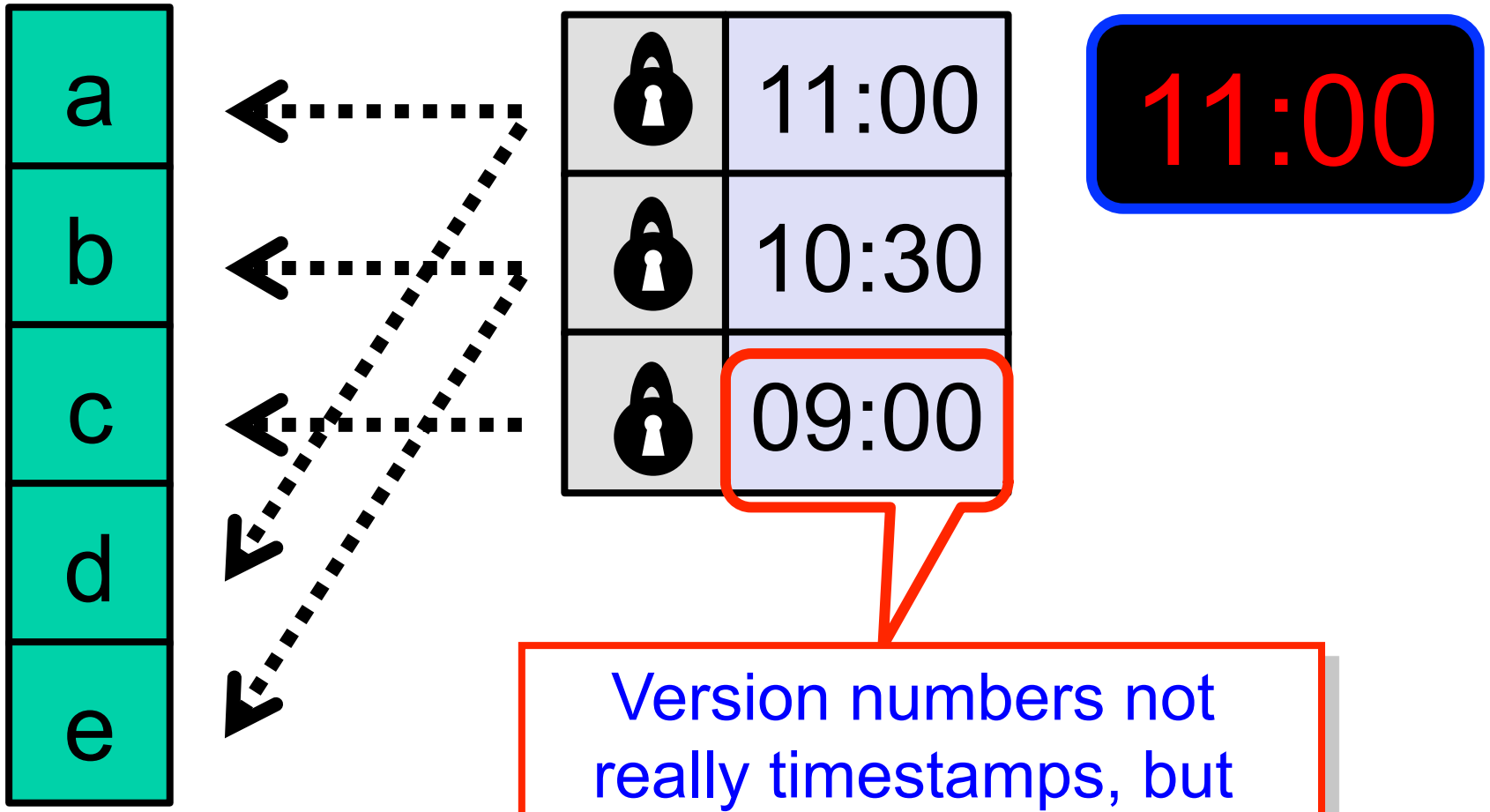
Incremented by (some) writers

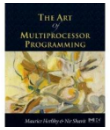


Read by everyone

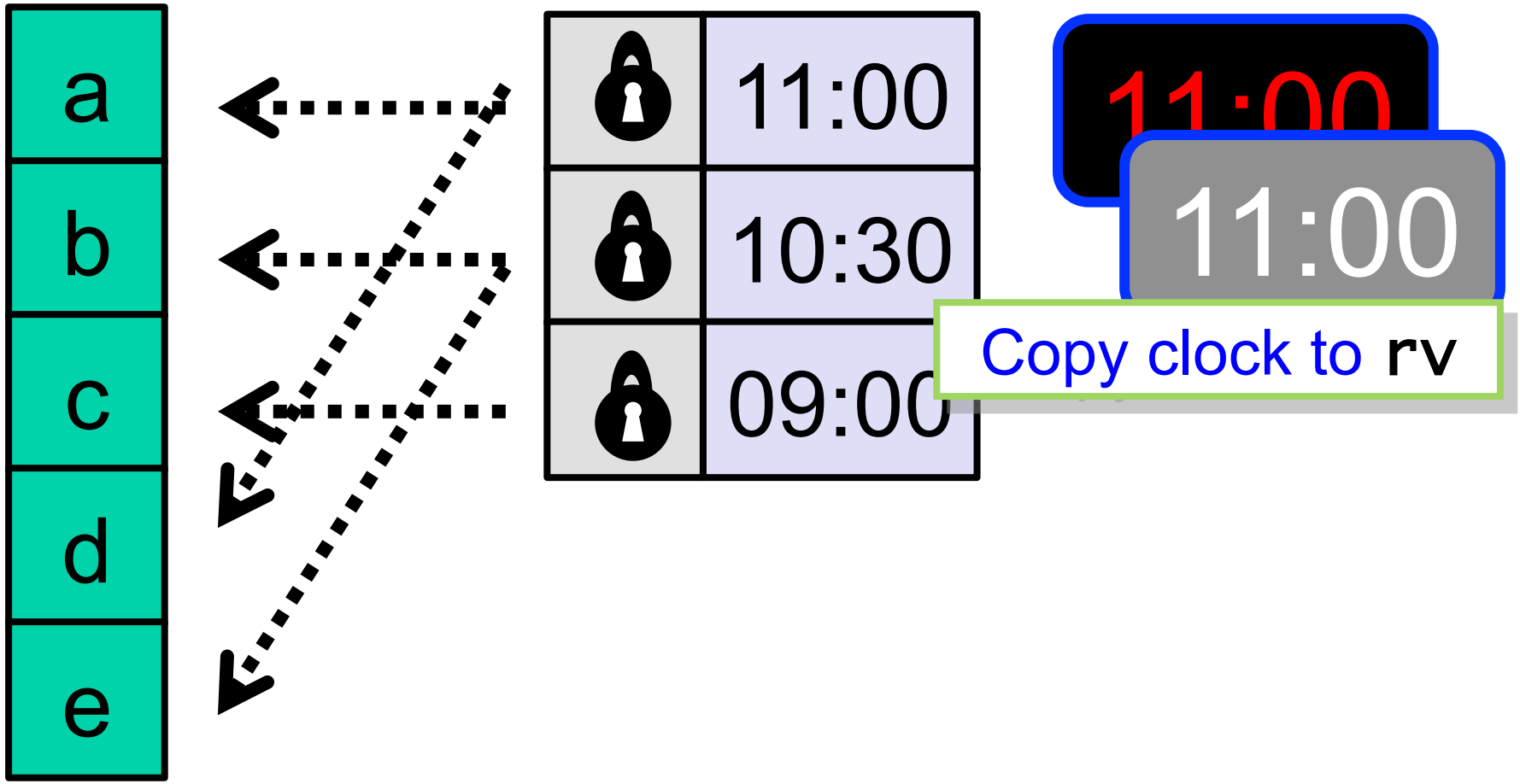
Guarantees **opacity**

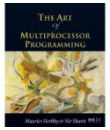
Transactin

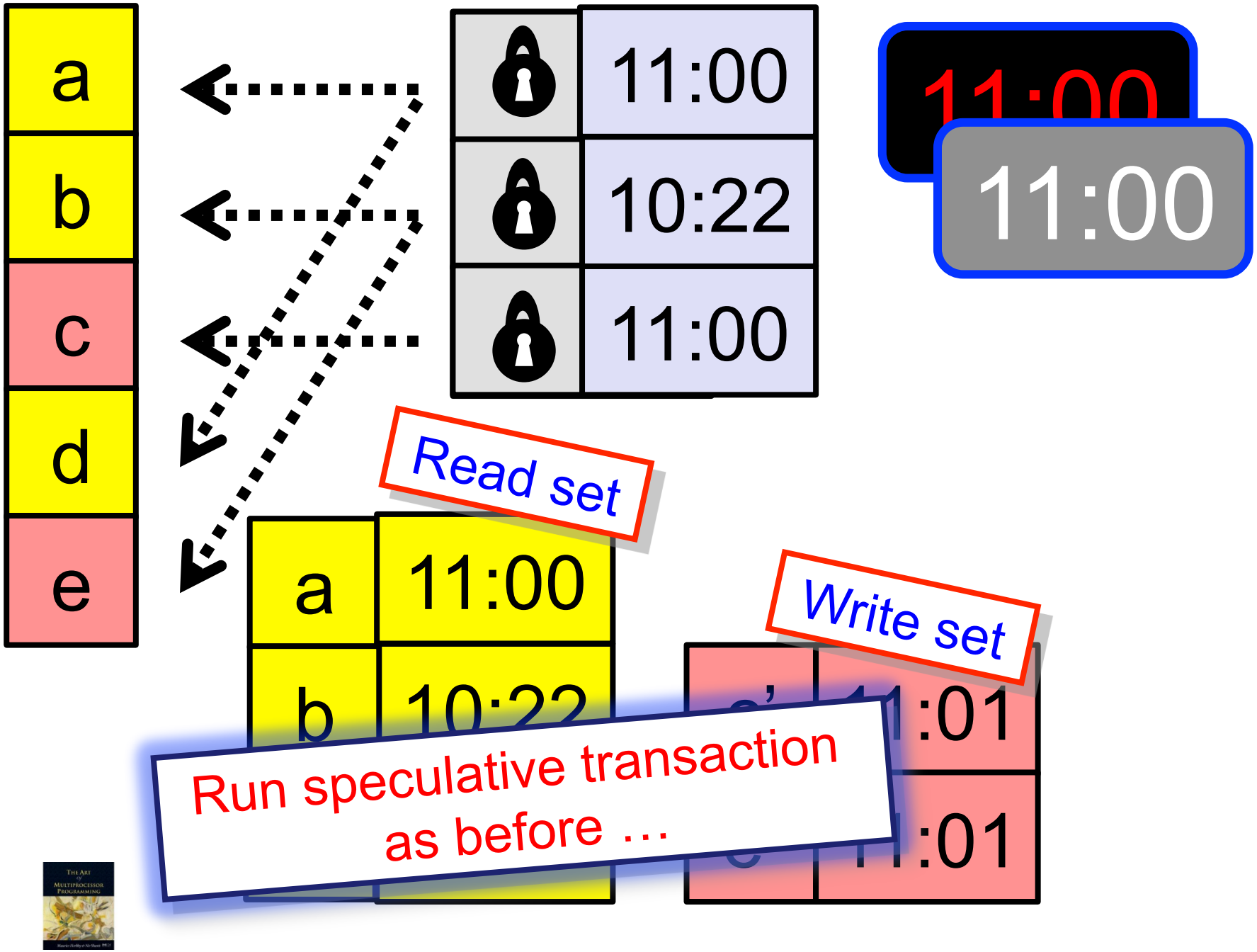

a
b
c
d
e
11:00
10:30
09:00
11:00
Version numbers not really timestamps, but useful to pretend

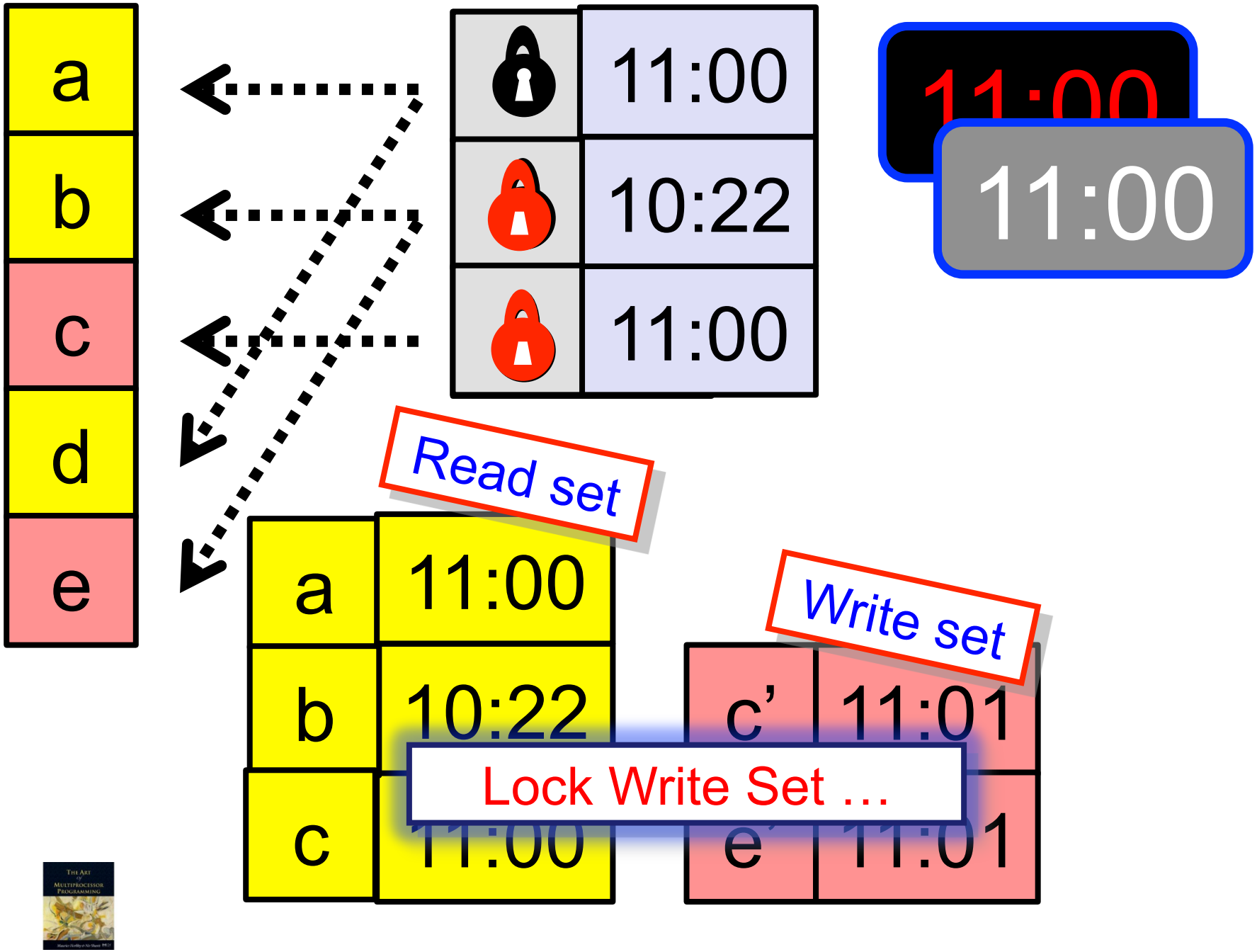


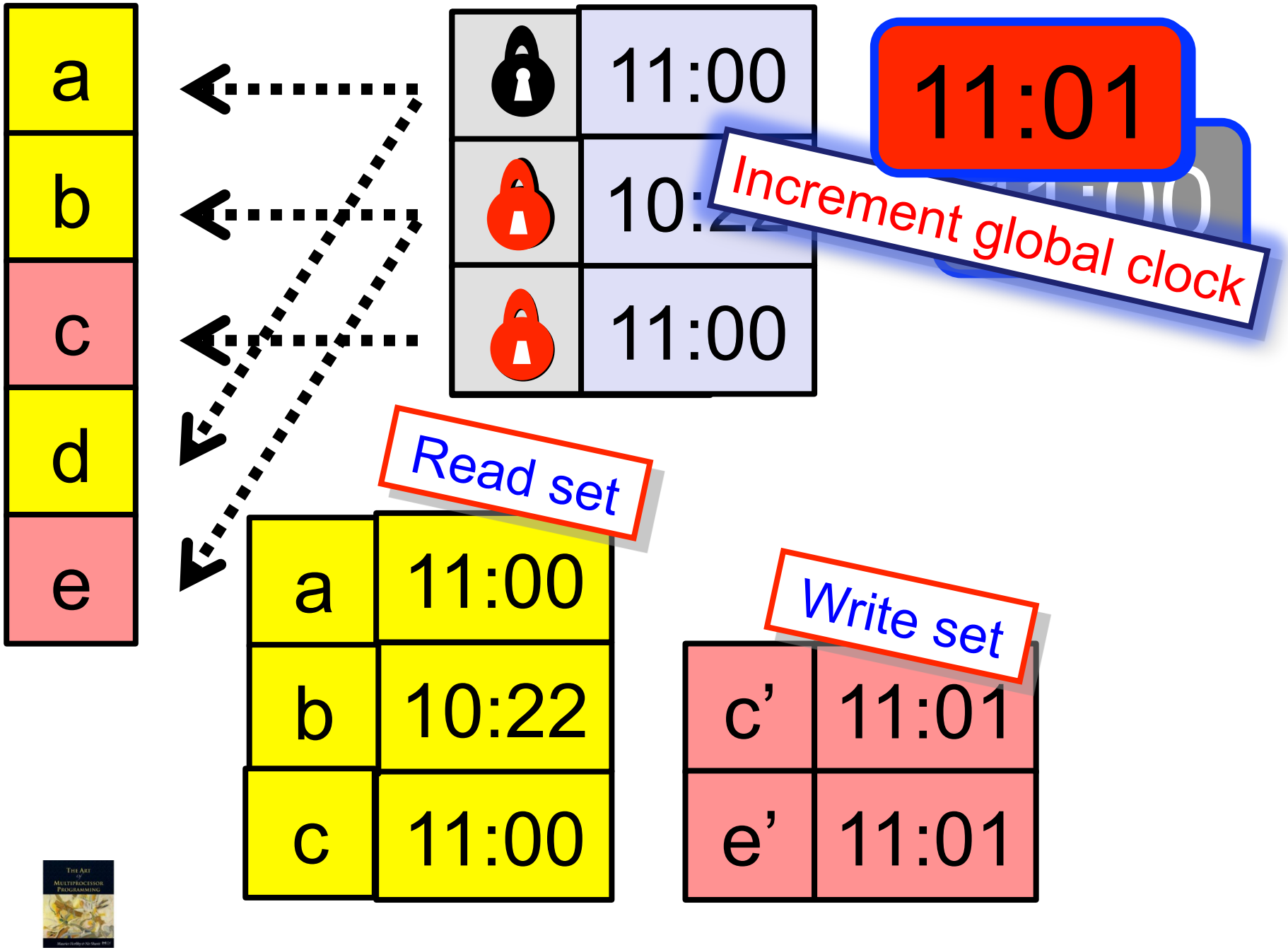
Transactions

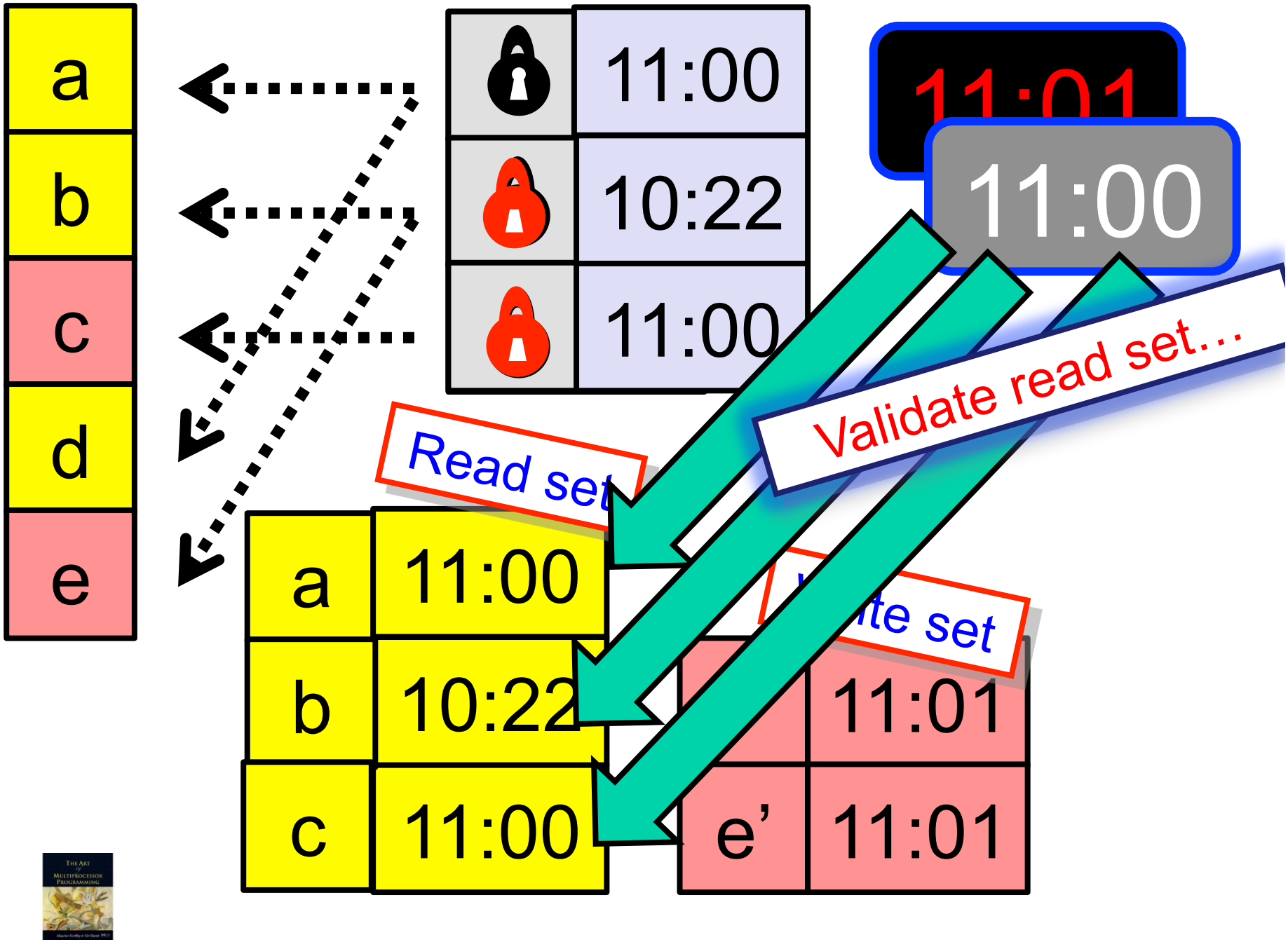


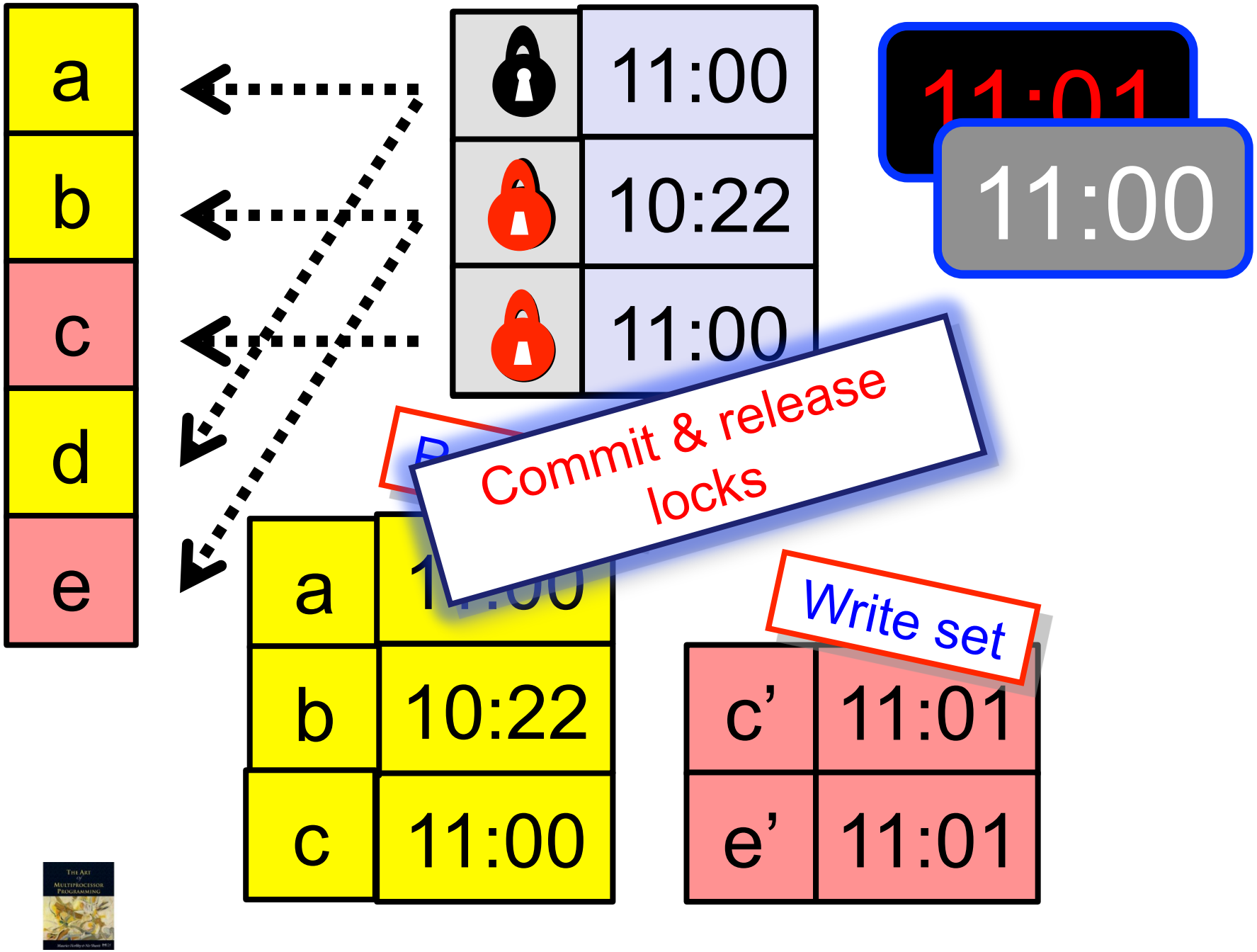


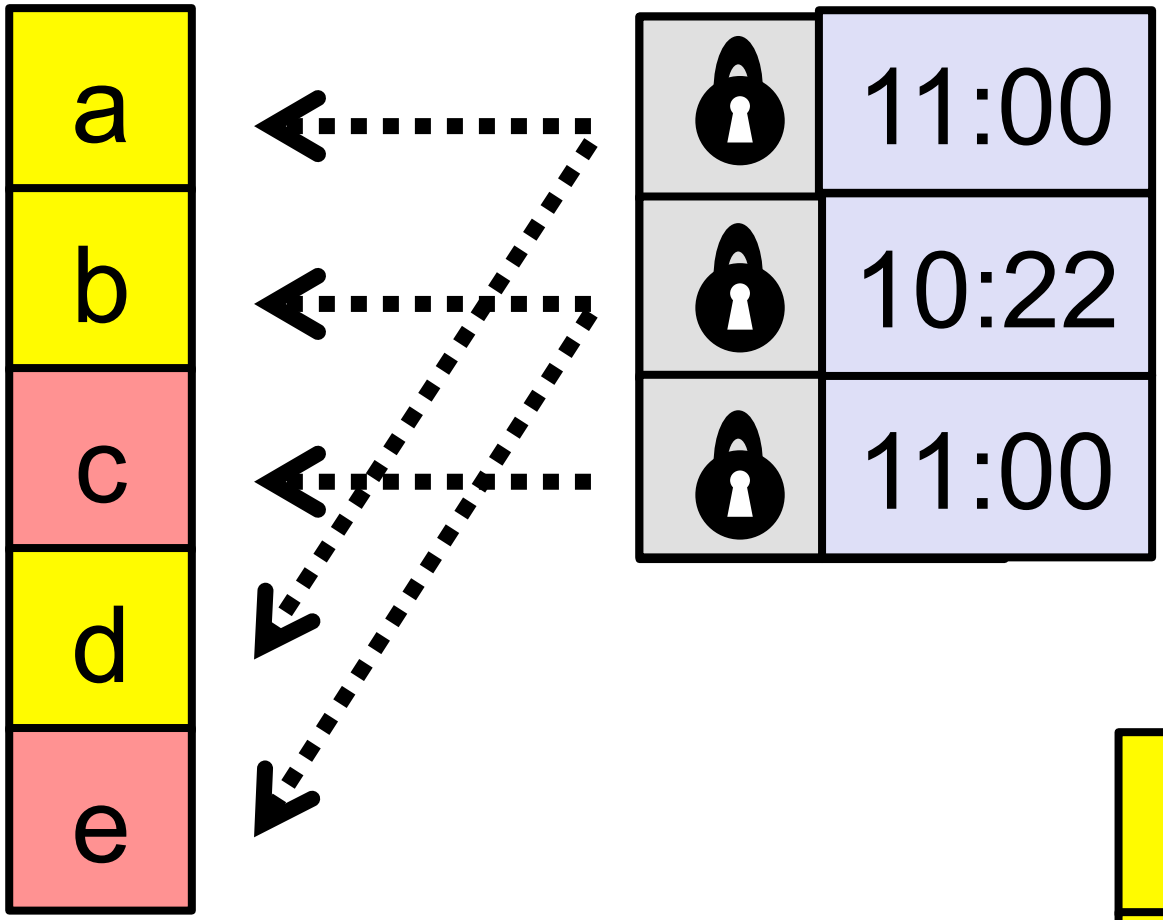

a
b
c
d
e
11:00
10:22
11:00
11:00
11:00
Read set
a
11:00
b
10:22
Write set
:01
:01
Run speculative transaction
as before …


a
b
c
d
e
11:00
10:22
11:00
11:00
11:00
Read set
a
11:00
b
10:22
c
11:00
Write set
c’
11:01
e’
11:01
Lock Write Set …


a
b
c
d
e
11:00
11:00
11:00
11:01
Increment global clock
Read set
a 11:00
b 10:22
c 11:00
Write set
c’ 11:01
e’ 11:01


a
b
c
d
e
11:00
10:22
11:00
11:01
11:00
Validate read set…
Read set
a 11:00
b 10:22
c 11:00
11:01
e’ 11:01

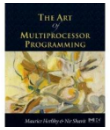

a
b
c
d
e
11:00
10:22
11:00
11:01
11:00
Commit & release locks
a
b
10:22
c
11:00
Write set
c’
11:01
e’
11:01

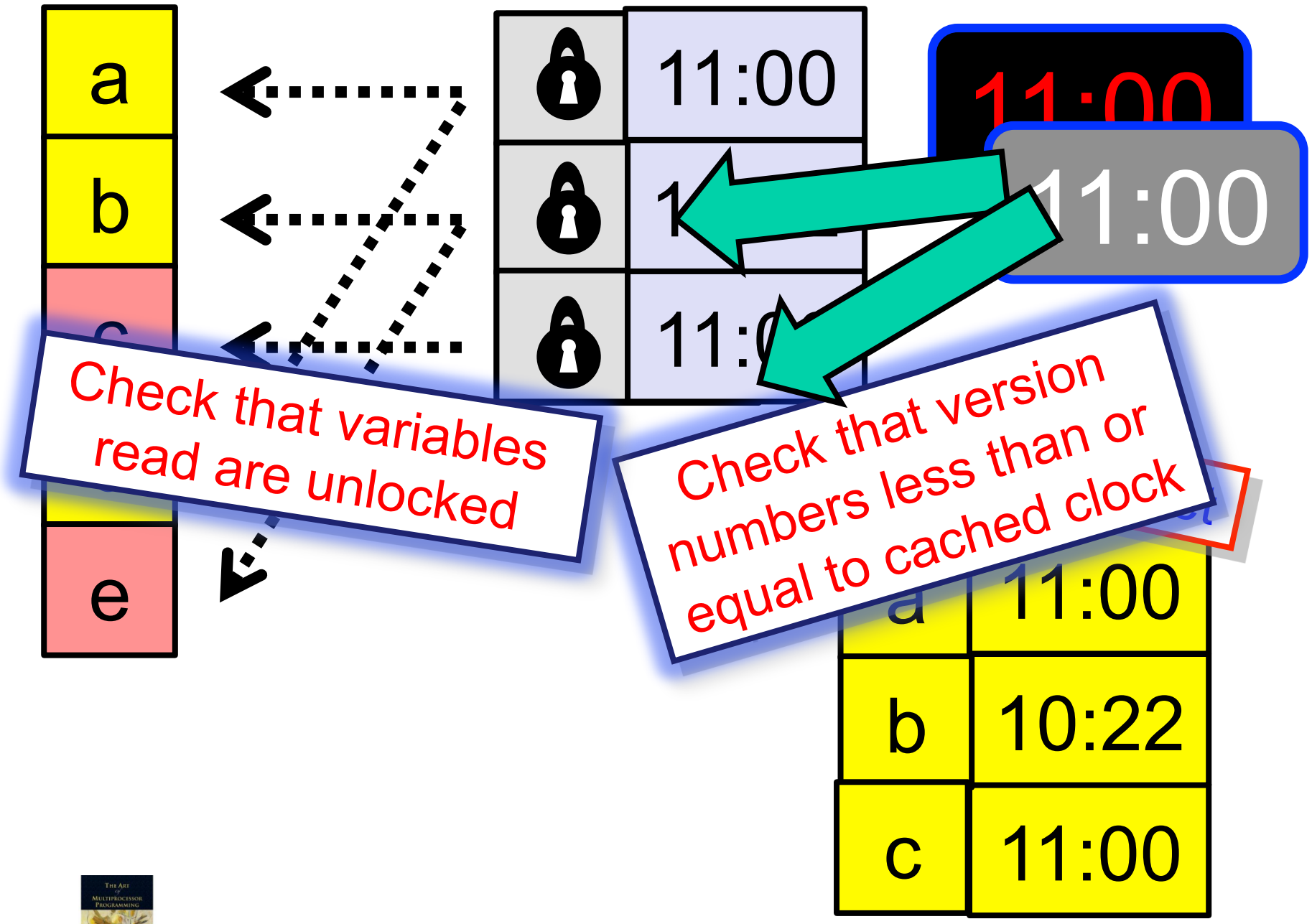


Read set

a	11:00
b	10:22
c	11:00

Read-only transactions?




a
b
c
e
11:00
11:00
11:00
Check that variables read are unlocked
Check that version numbers less than or equal to cached clock
a 11:00
b 10:22
c 11:00

Road Map

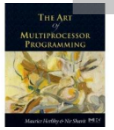
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

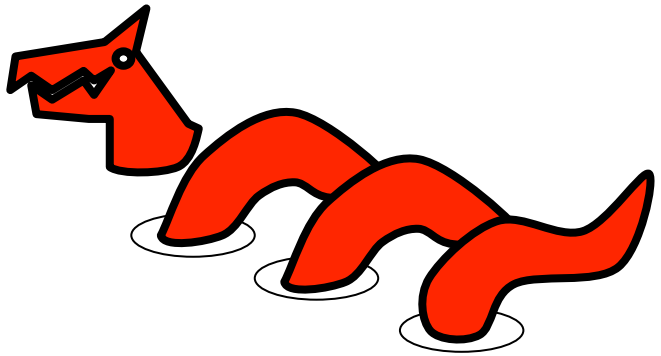
Software Transactional Memory

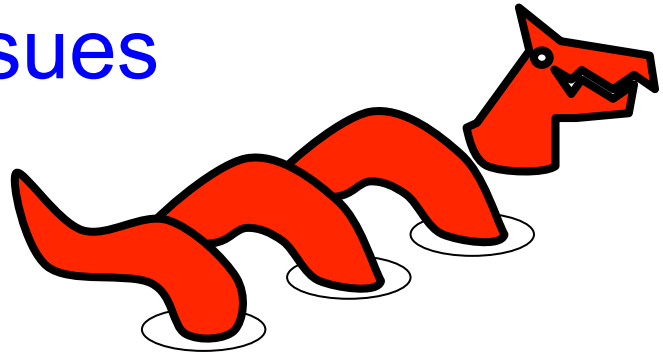
Research Questions

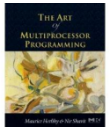


TM Design Issues

- Implementation choices
- Language design issues
- Semantic issues

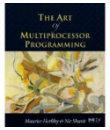






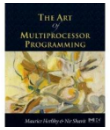
Granularity

- 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



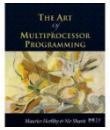
Direct/Deferred Update

- *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



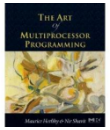
Conflict Detection

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



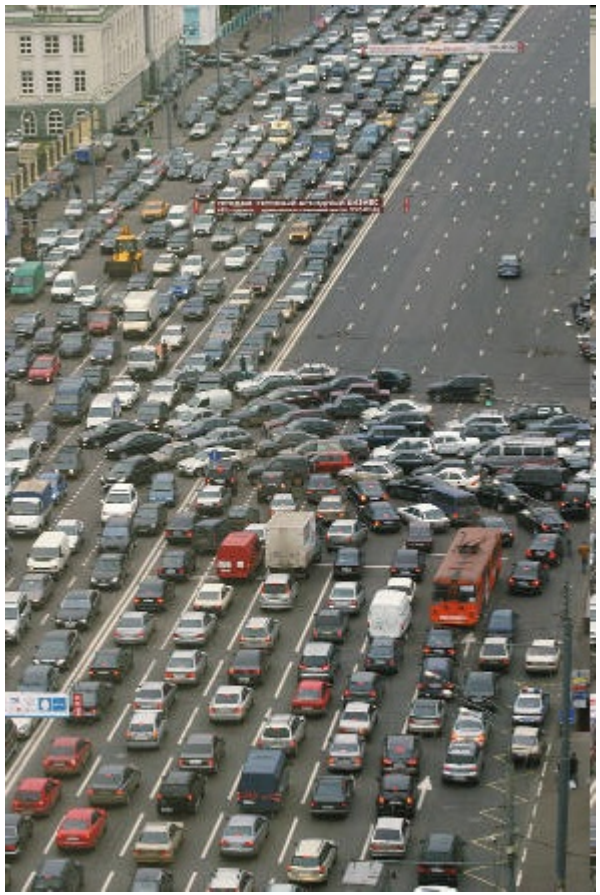
Conflict Detection

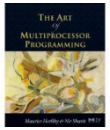
- Eager detection may abort transactions that could have committed.
- Lazy detection discards more computation.



Contention Management & Scheduling

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

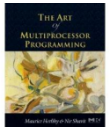




Contention Manager Strategies

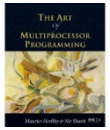
- Exponential backoff
- Priority to
  - Oldest?
  - Most work?
  - Non-waiting?
- None Dominates
- But needed anyway

Judgment of Solomon



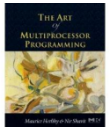
I/O & System Calls?

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



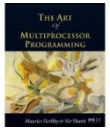
I/O & System Calls

- 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

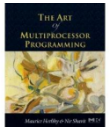
```
int i = 0;
try {
  atomic {
    i++;
    node = new Node();
  }
} catch (Exception e) {
  print(i);
}
```



Exceptions

Throws OutOfMemoryException!

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

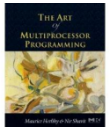


Exceptions

Throws OutOfMemoryException!

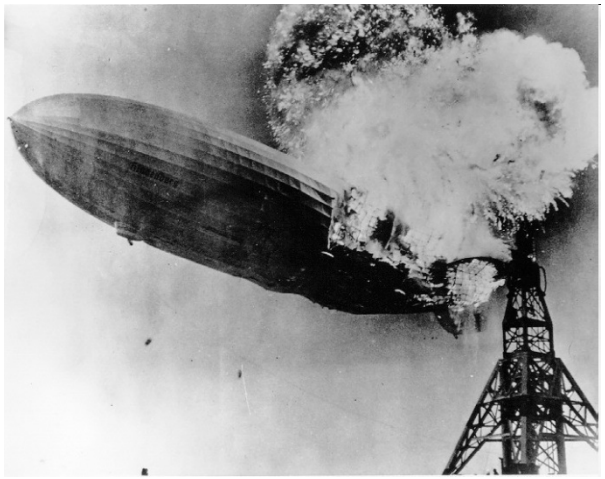
```
int i = 0;
try {
  atomic {
    i++;
    node = new Node();
  }
} catch (Exception e) {
  print(i);
}
```

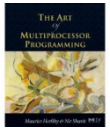
What is printed?



Unhandled Exceptions

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

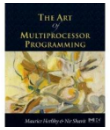




Nested Transactions

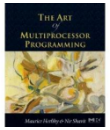
```
atomic void foo() {
  bar();
}

atomic void bar() {
  …
}
```

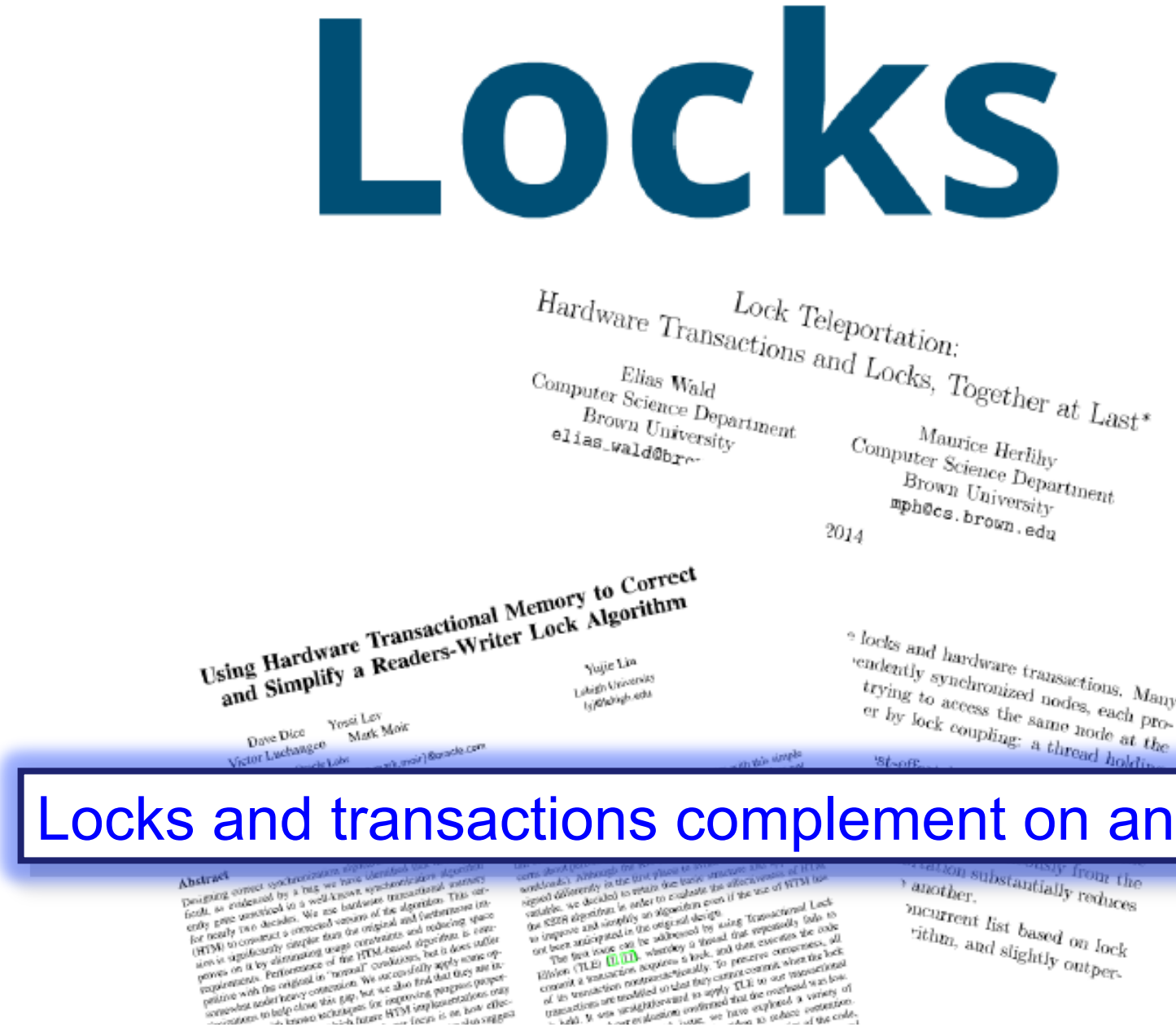


Nested Transactions

- 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



Locks



Lock Teleportation:
Hardware Transactions and Locks, Together at Last*

Elias Wald
Computer Science Department
Brown University
elias_wald@bro-

Maurice Herlihy
Computer Science Department
Brown University
mph@cs.brown.edu

2014

Using Hardware Transactional Memory to Correct and Simplify a Readers-Writer Lock Algorithm

Yujie Liu
Lehigh University
lyj@lehigh.edu

Dave Dice Yossi Lev
Victor Luchangco Mark Moir

locks and hardware transactions. Many
endently synchronized nodes, each pro-
trying to access the same node at the
er by lock coupling: a thread holding

Abstract

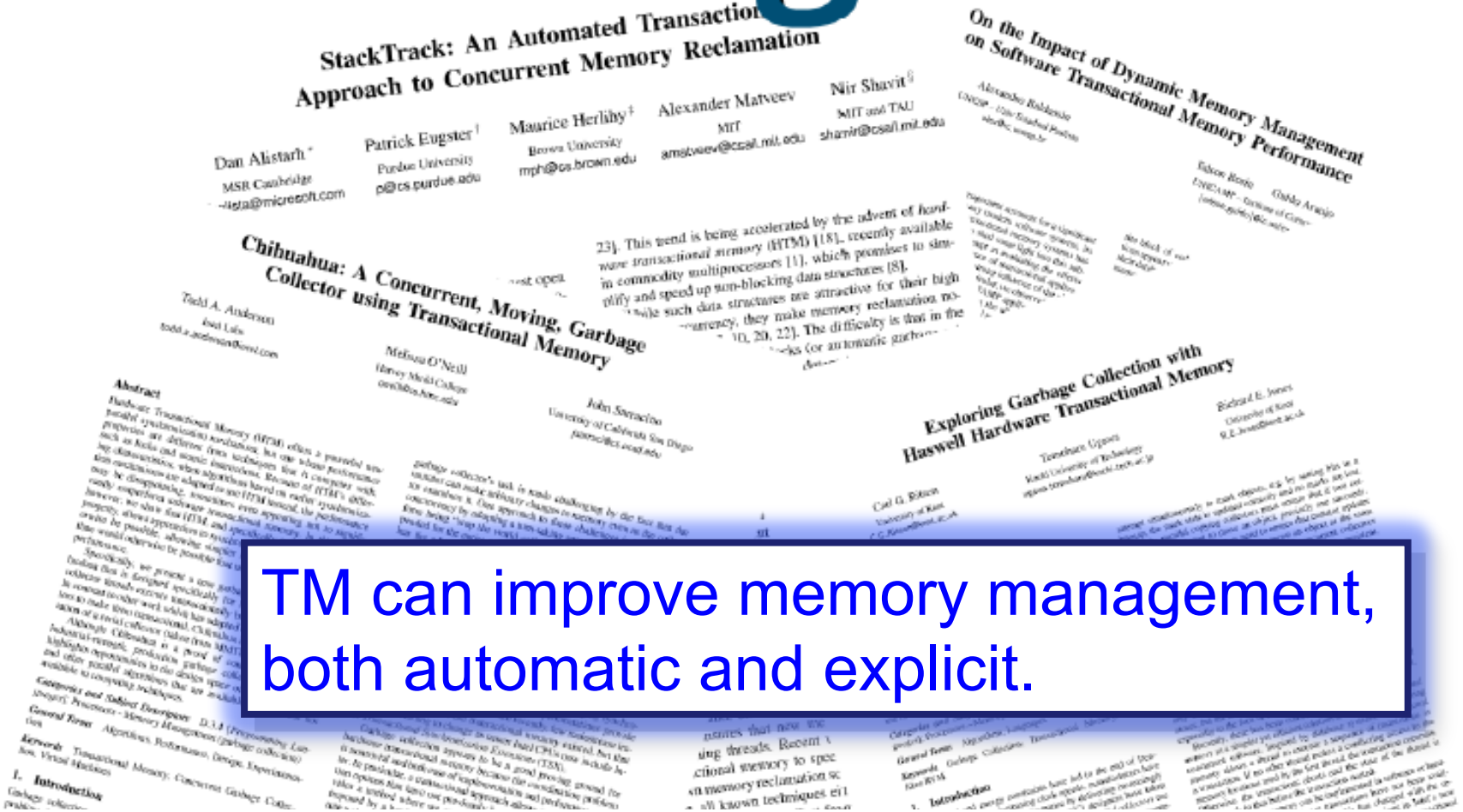
Designing correct synchronization algorithms is difficult, as evidenced by a bug we have identified that recently gave attendees at a well-known synchronization conference fits. We use hardware transactional memory for nearly two decades. We use hardware transactional memory (HTM) to construct a concise version of the algorithm. This version is significantly simpler than the original and furthermore improves on it by eliminating unsafe constructs and reducing space usage. Performance of the HTM-based algorithm is competitive with the original on "normal" workloads, but it does suffer somewhat under heavy contention. We also find that they are unable to help close this gap, but we also find that they are unable to improving progress properties for known techniques for improving progress properties of HTM implementations using future HTM implementations using

Locks and transactions complement on another

another.

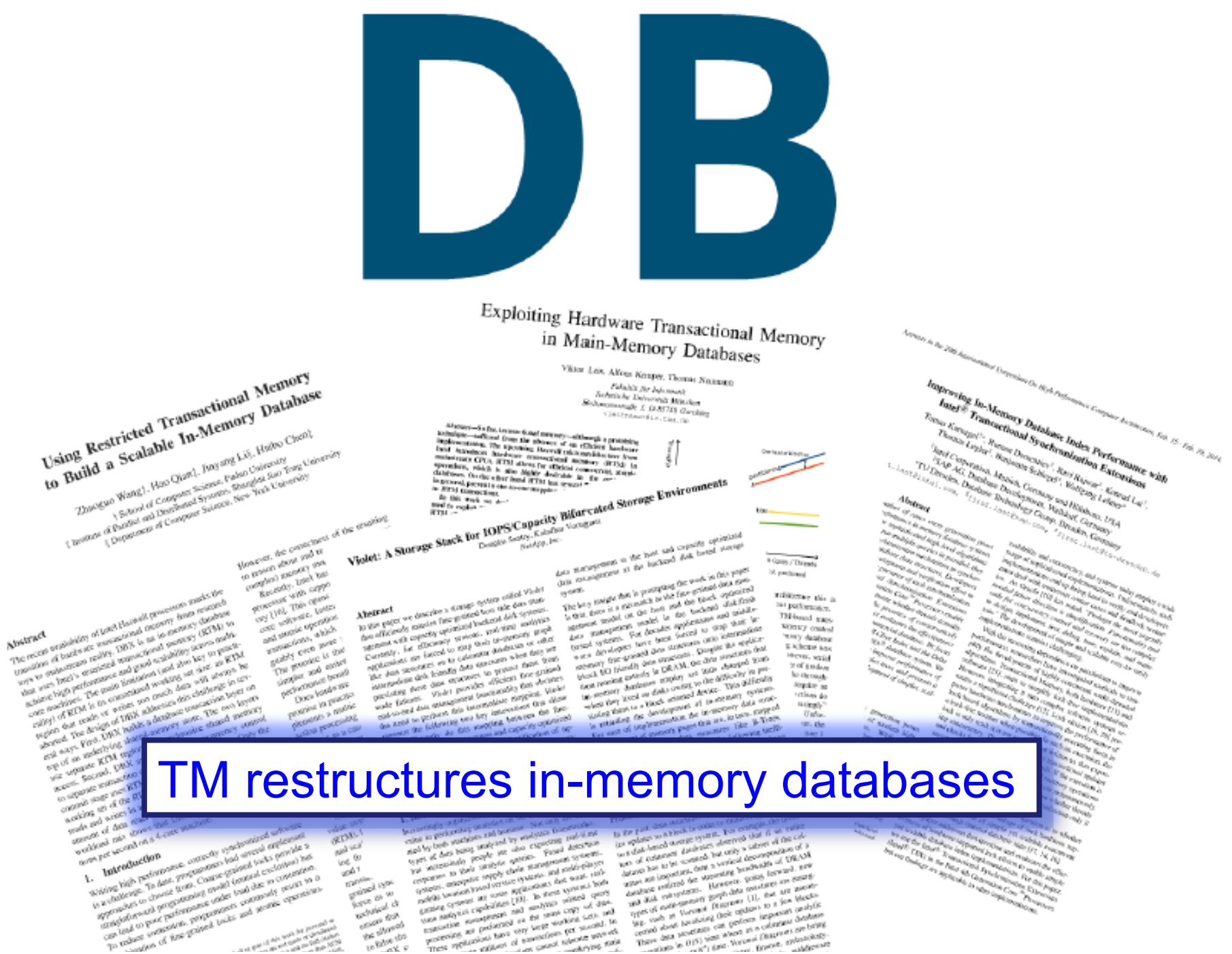
ncurrent list based on lock
rithm, and slightly outper-

Memory Management

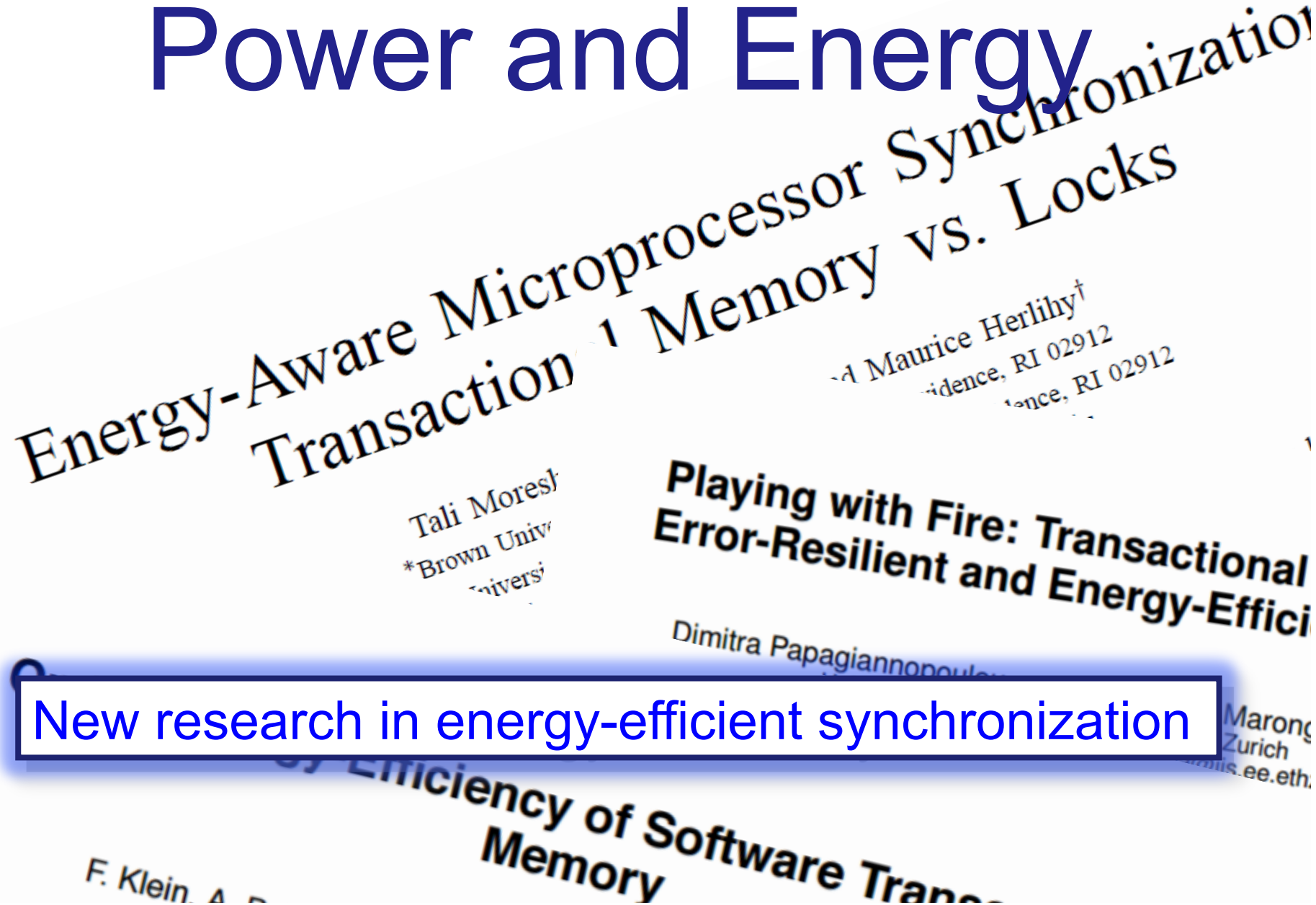


TM can improve memory management, both automatic and explicit.

DB

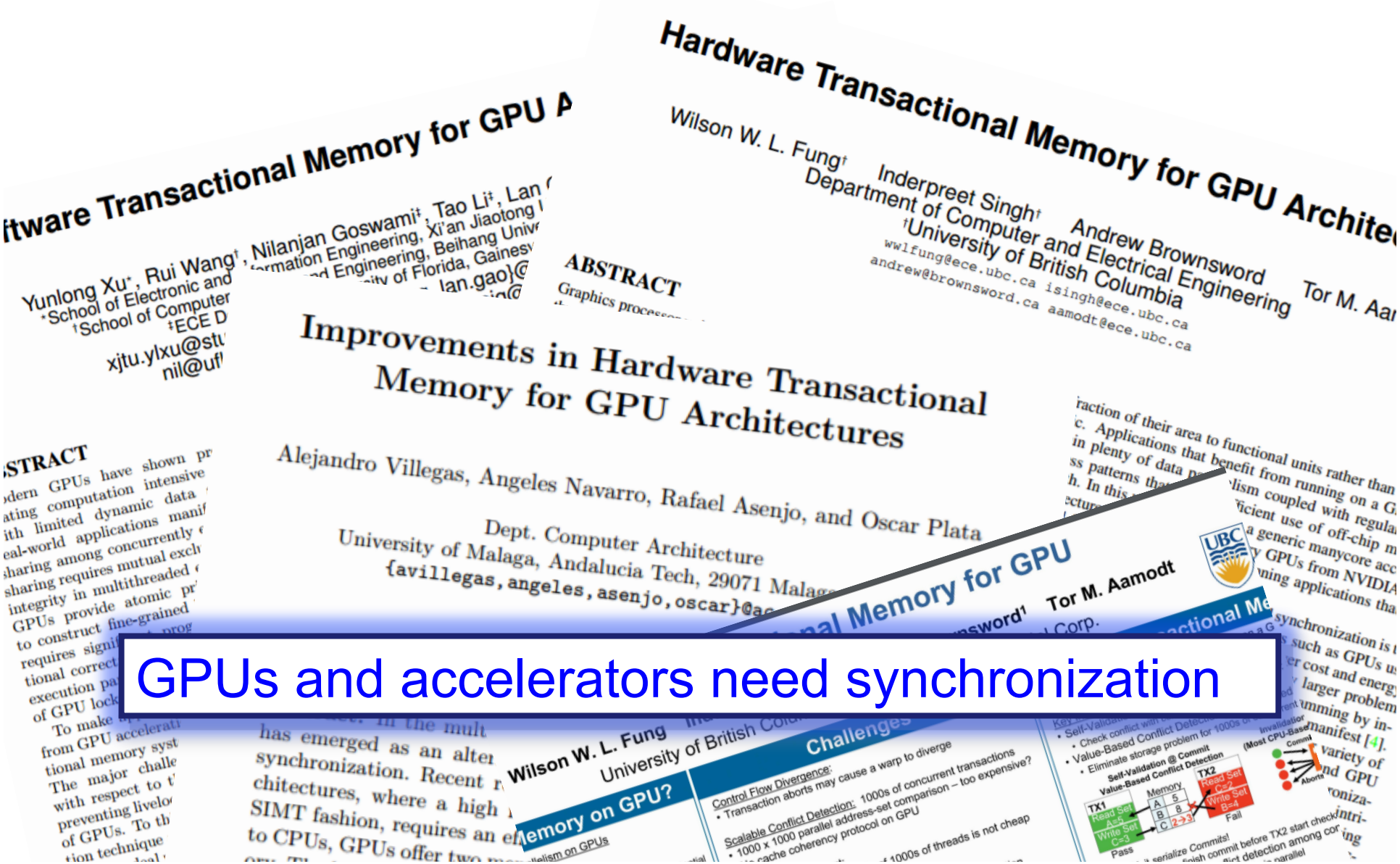

Using Restricted Transactional Memory to Build a Scalable In-Memory Database
Exploiting Hardware Transactional Memory in Main-Memory Databases
Violet: A Storage Stack for IOPS/Capacity Bifurcated Storage Environments
Improving In-Memory Database Index Performance with Intel® Transactional Synchronization Extensions
TM restructures in-memory databases

Power and Energy



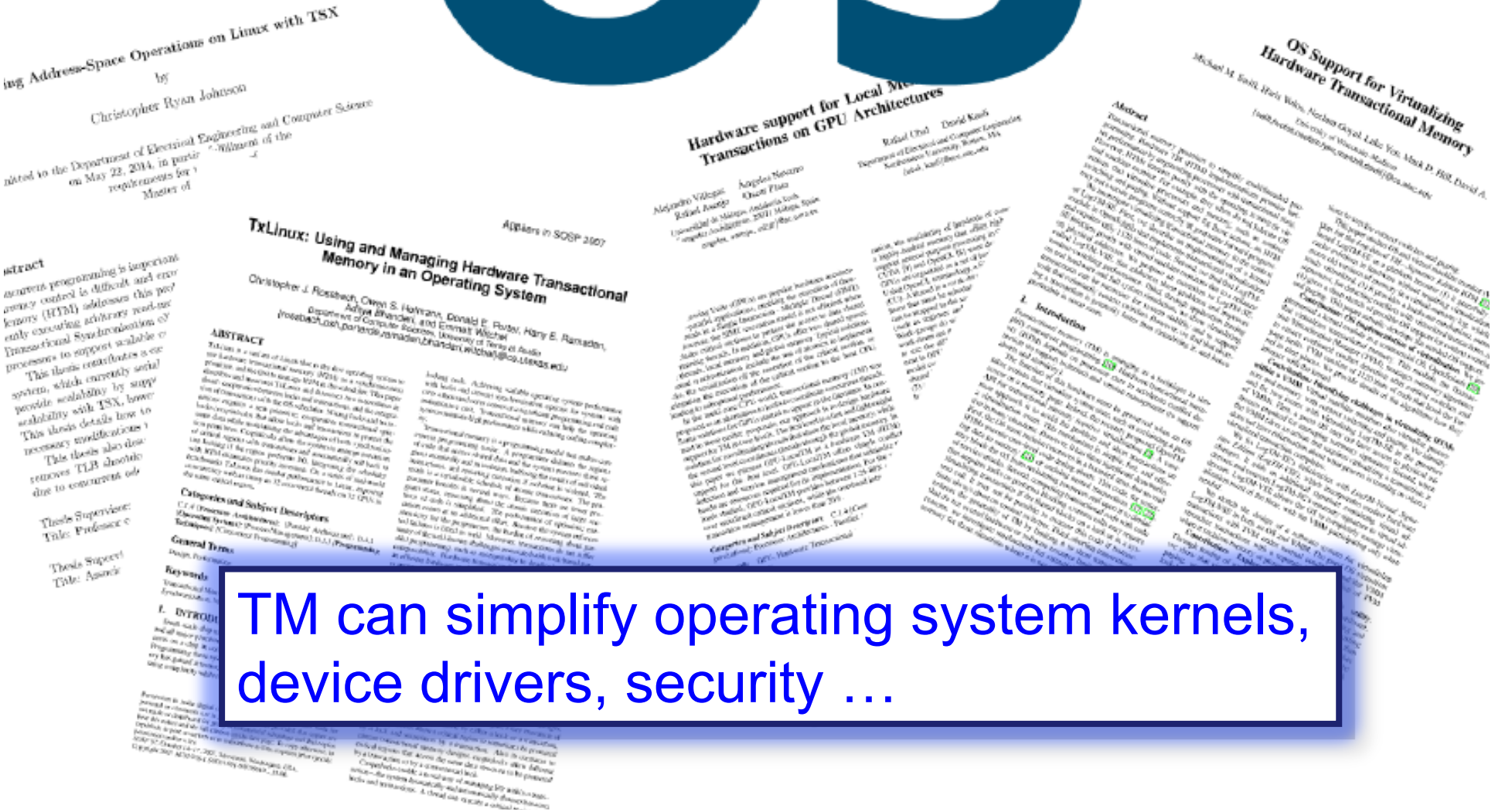
New research in energy-efficient synchronization

GPUs, etc.



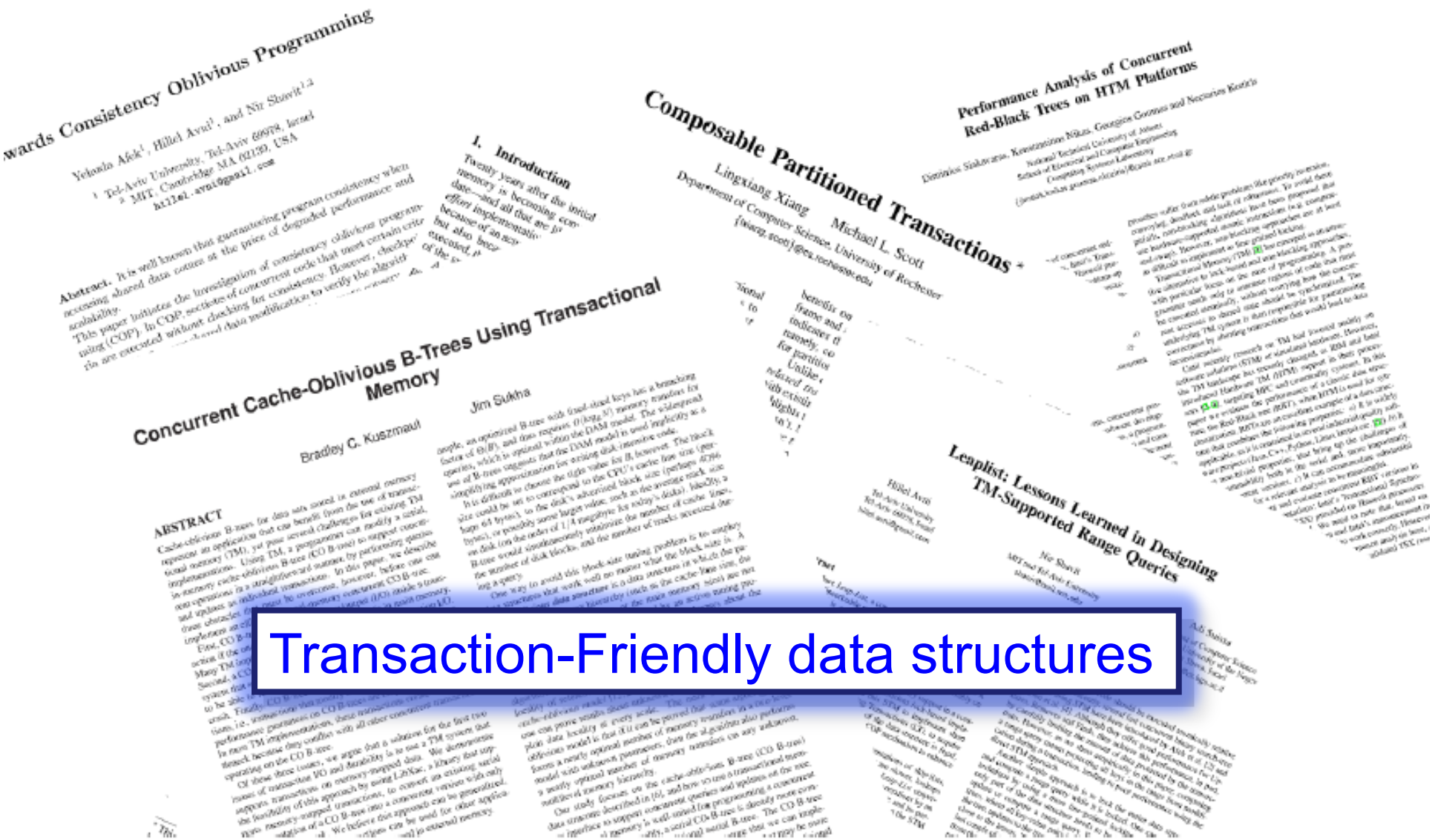
GPUs and accelerators need synchronization

OS



TM can simplify operating system kernels, device drivers, security …

Data Structures



Theory

8th Workshop on the Theory of Transactional Memory
WTTM 2016

July 25, 2016, Chic

International Conference on Computer Aided Verification
CAV 2009: Computer Aided Verification pp 1-15

Transactional Memory: Glimmer of a Theory

ed Paper)

ors and affiliations

The Theory of Tra Memory

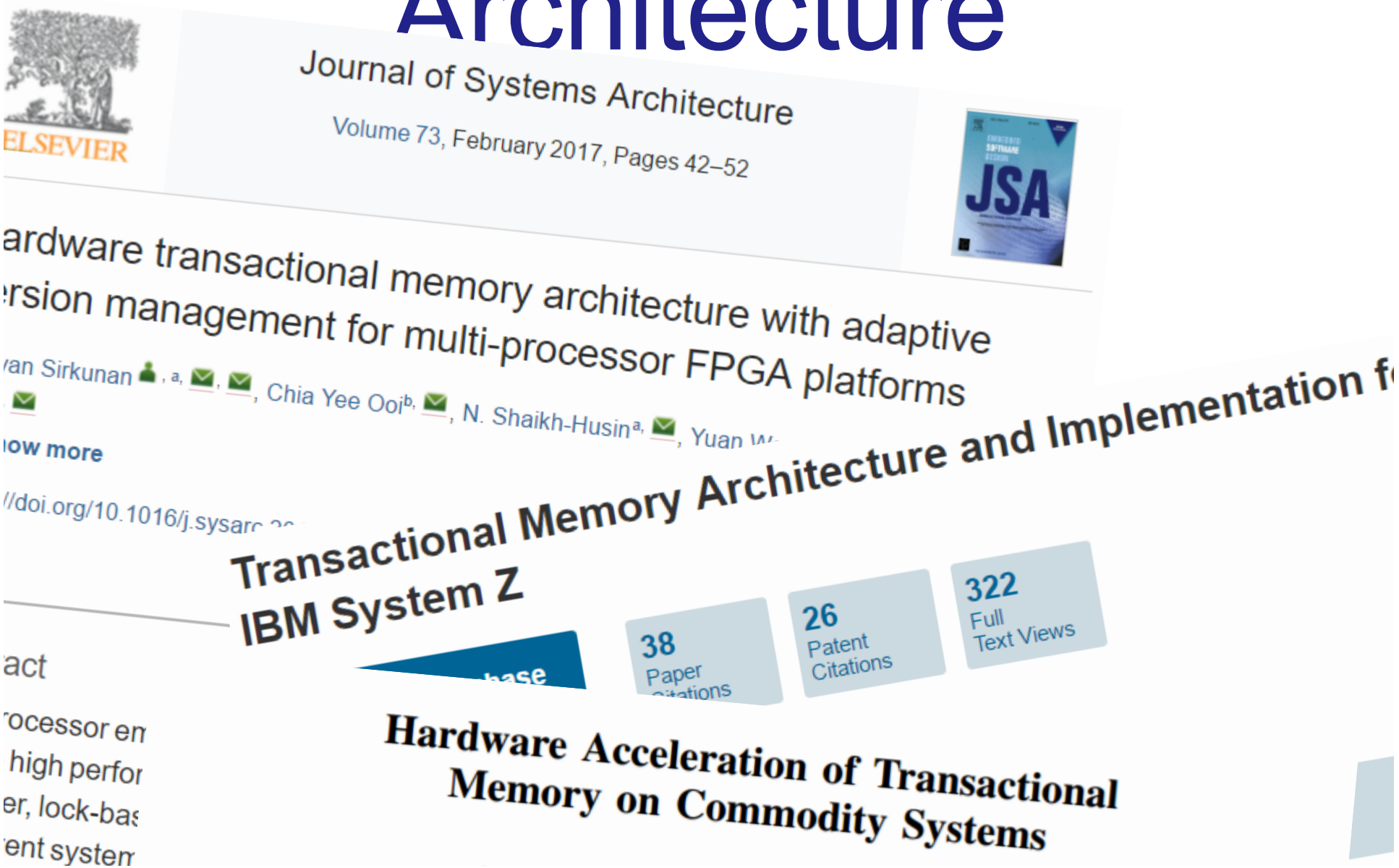
Guerraoui, Rachid; Kapalka, Michal

Published in: Bulletin of the EATCS, num. 97

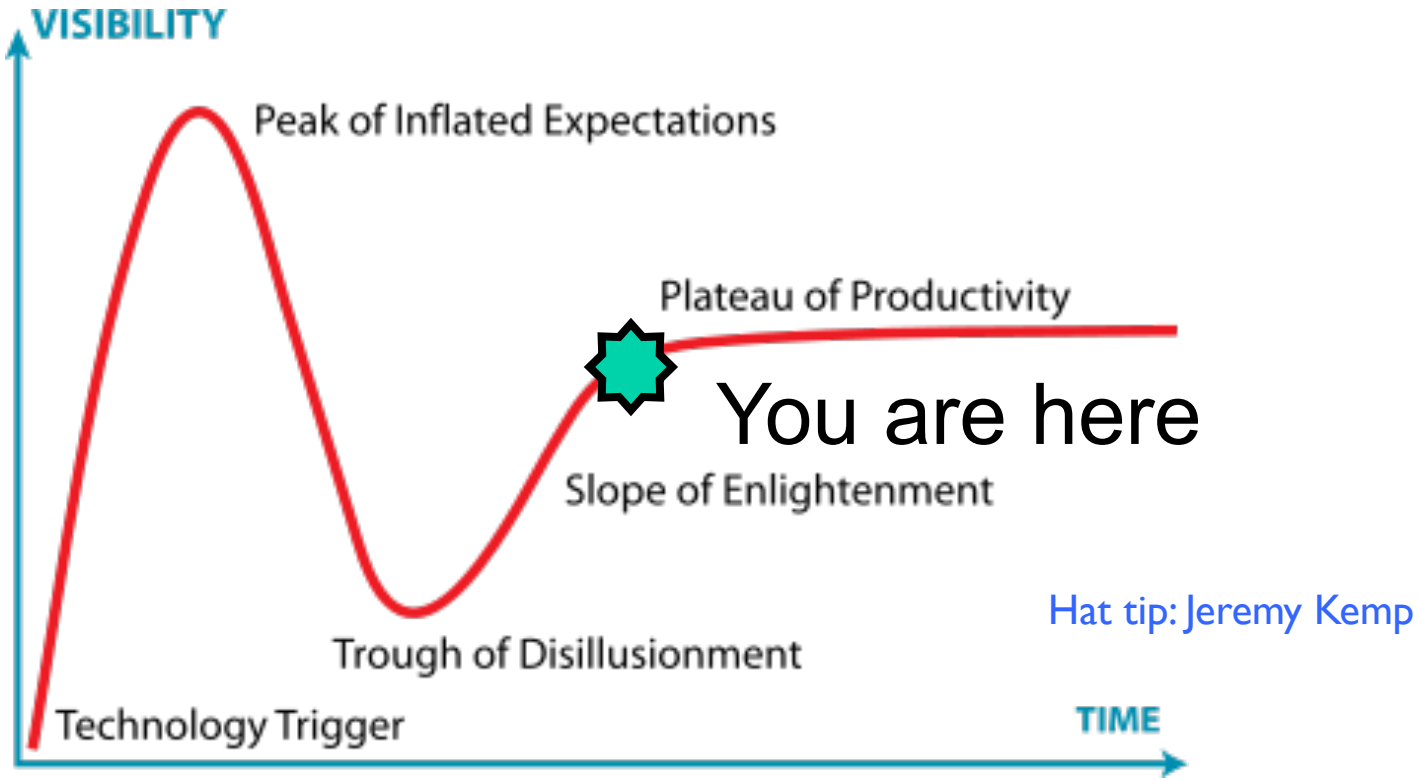
Publication date: 2009

nsactional memory (TM) is a promising paradigr
ramming. This paper is an overview of our
of TM. We first present a correctne
isting TM implementations. T
acterize the two main
ck-based TMs
ent powe

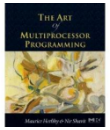
Architecture



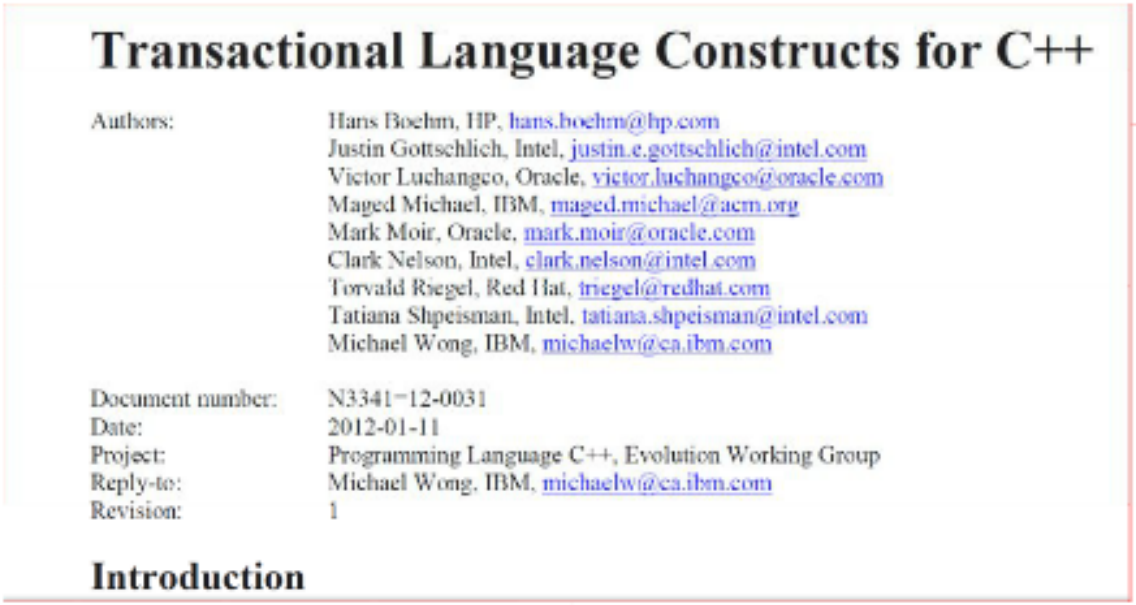
Gartner Hype Cycle



Hat tip: Jeremy Kemp



Transactions are Here to Stay



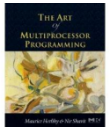
Transactional Language Constructs for C++

Authors: Hans Boehm, HP, hans.boehm@hp.com
Justin Gottschlich, Intel, justin.e.gottschlich@intel.com
Victor Luchangco, Oracle, victor.luchangco@oracle.com
Maged Michael, IBM, maged.michael@acm.org
Mark Moir, Oracle, mark.moir@oracle.com
Clark Nelson, Intel, clark.nelson@intel.com
Torvald Riegel, Red Hat, triegel@redhat.com
Tatiana Shpeisman, Intel, tatiana.shpeisman@intel.com
Michael Wong, IBM, michaelw@ca.ibm.com

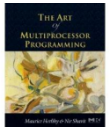
Document number: N3341=12-0031
Date: 2012-01-11
Project: Programming Language C++, Evolution Working Group
Reply-to: Michael Wong, IBM, michaelw@ca.ibm.com
Revision: 1

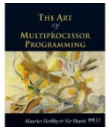
Introduction

Intel® Architecture
Instruction Set Extensions
Programming Reference



Спасибо!




The Art of
Multiprocessor
Programming