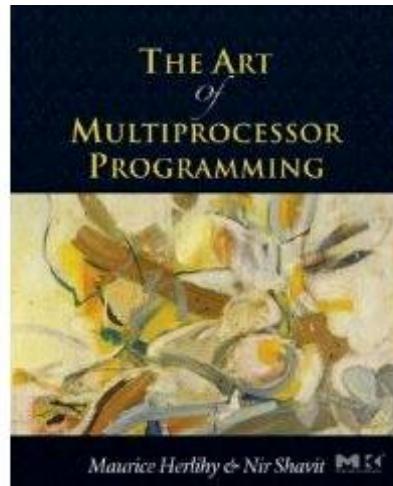
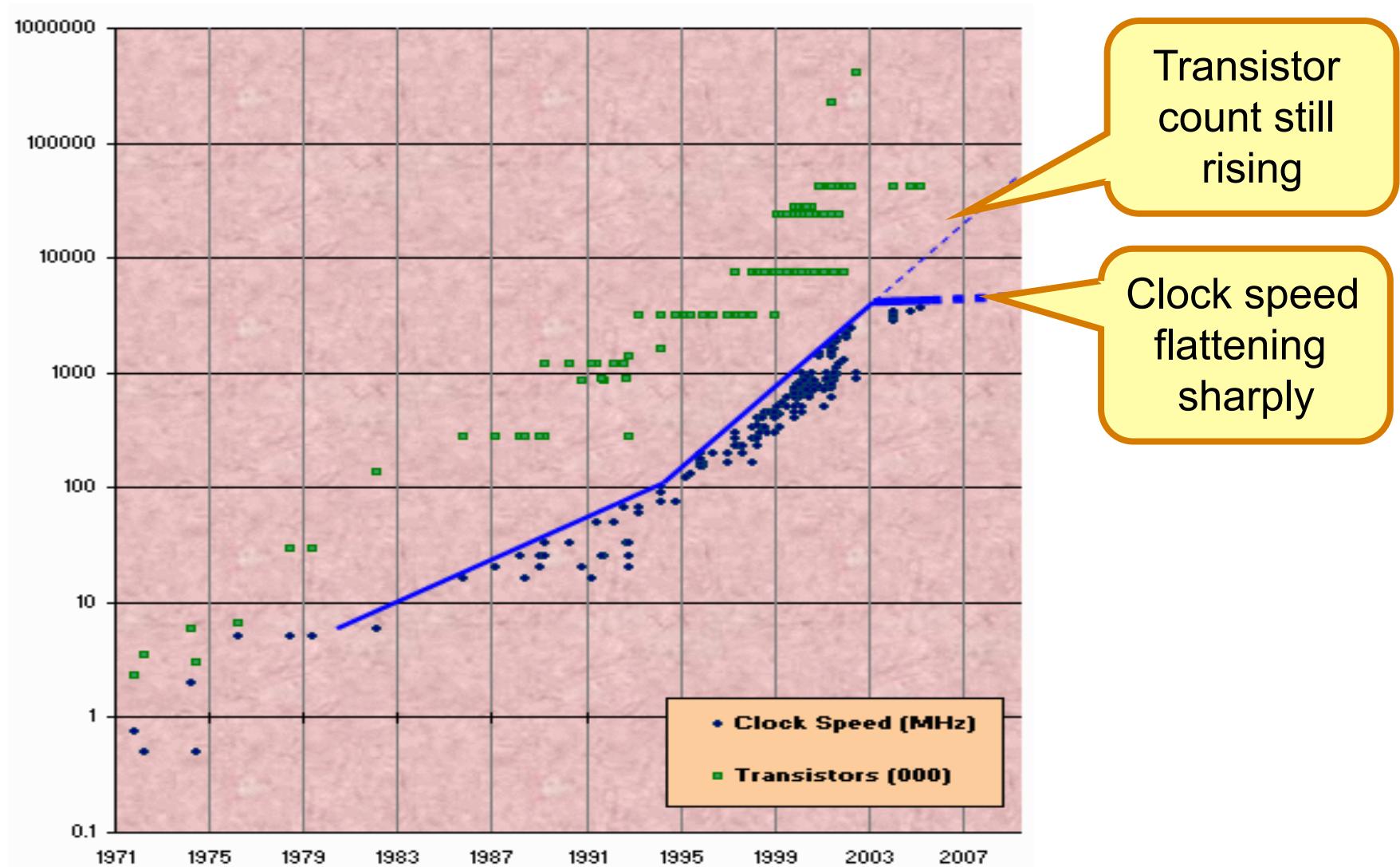


Transactional Memory

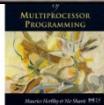


Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

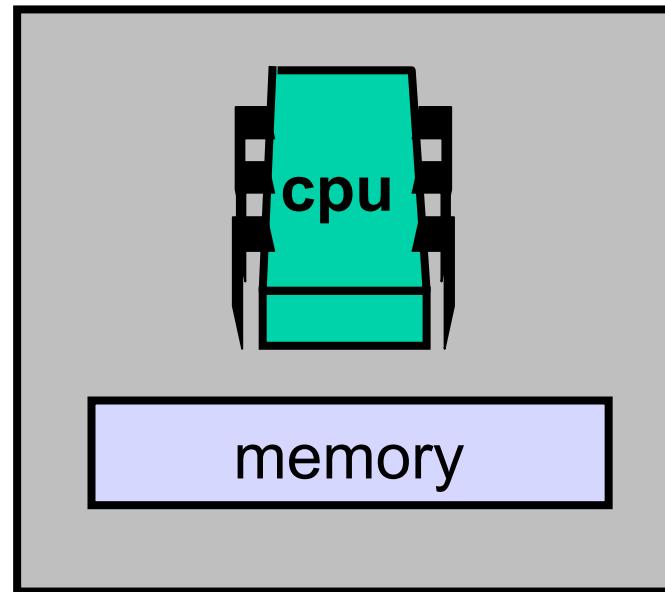
Moore's Law



Moore's Law (in practice)

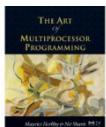
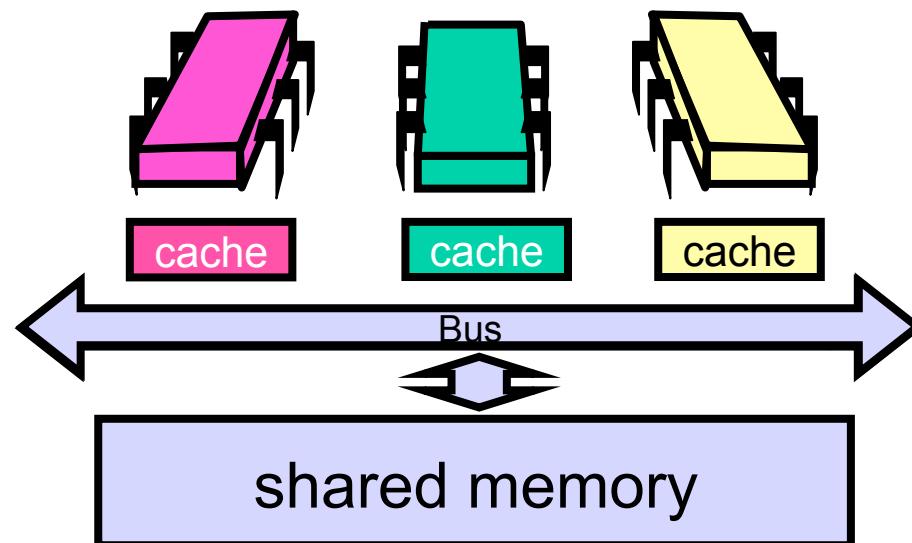


Nearly Extinct: the Uniprocessor



Art of Multiprocessor
Programming

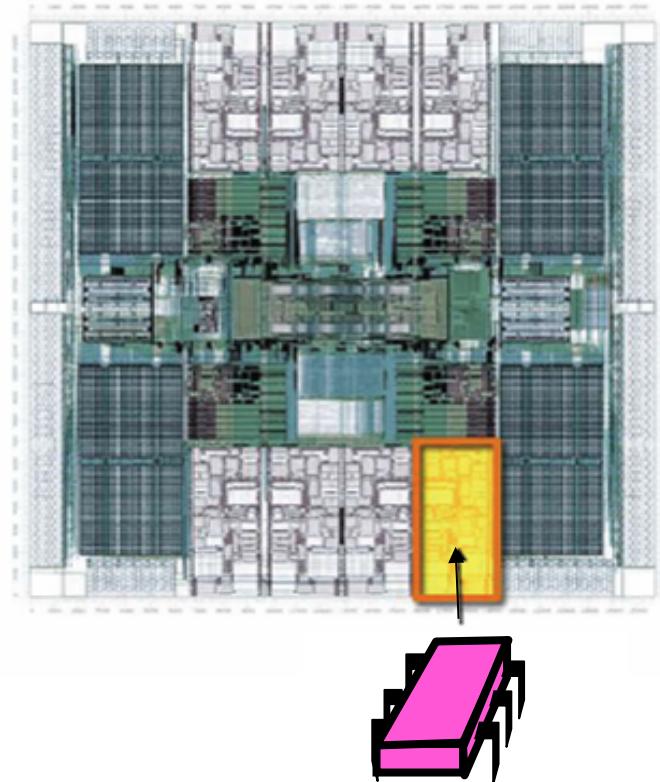
Endangered: The Shared Memory Multiprocessor (SMP)



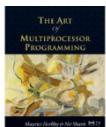
Art of Multiprocessor
Programming

The New Boss: The Multicore Processor (CMP)

All on the
same chip

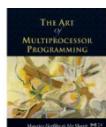
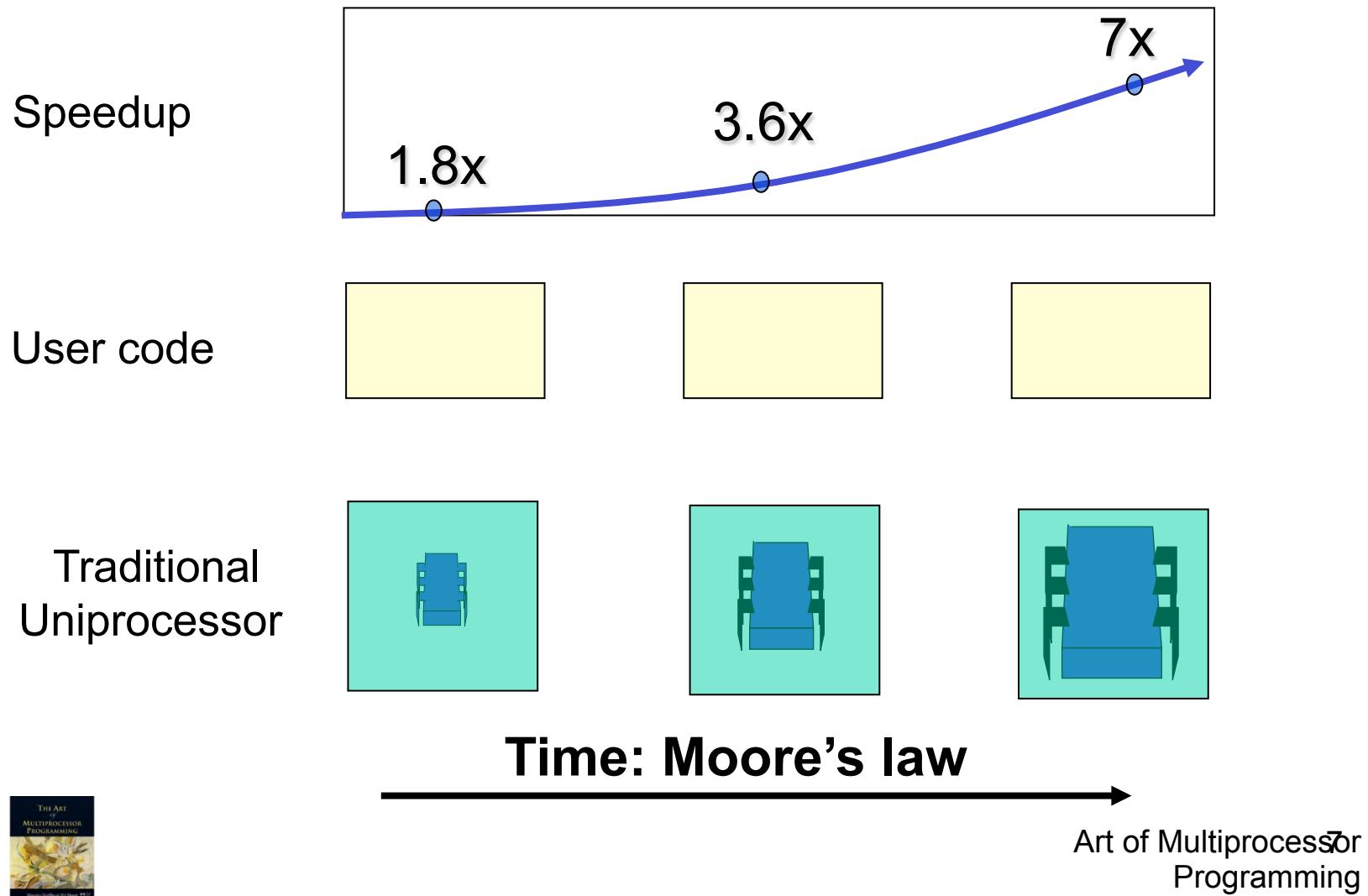


**Sun
T2000
Niagara**

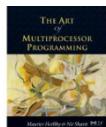
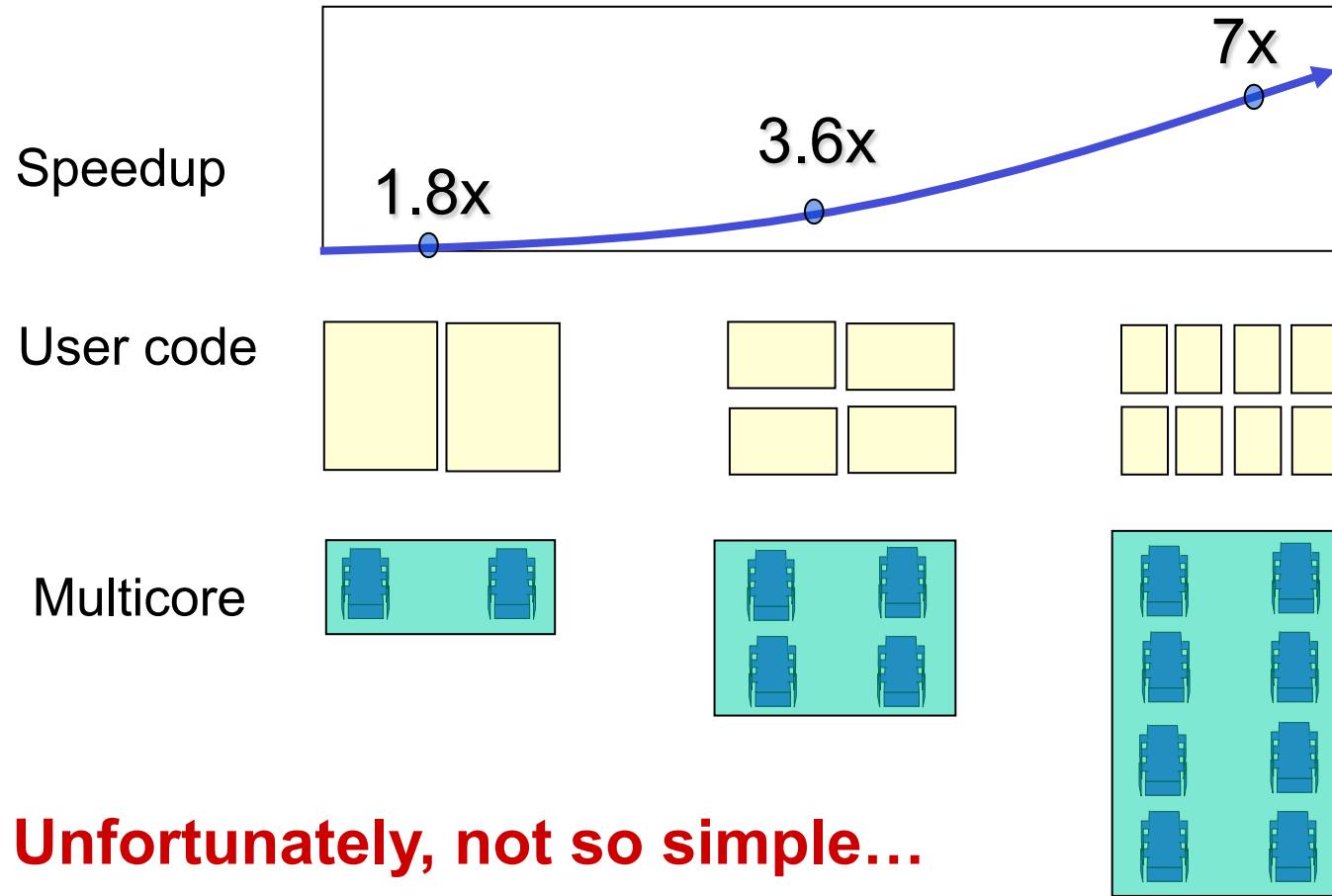


Art of Multiprocessor
Programming

Traditional Scaling Process

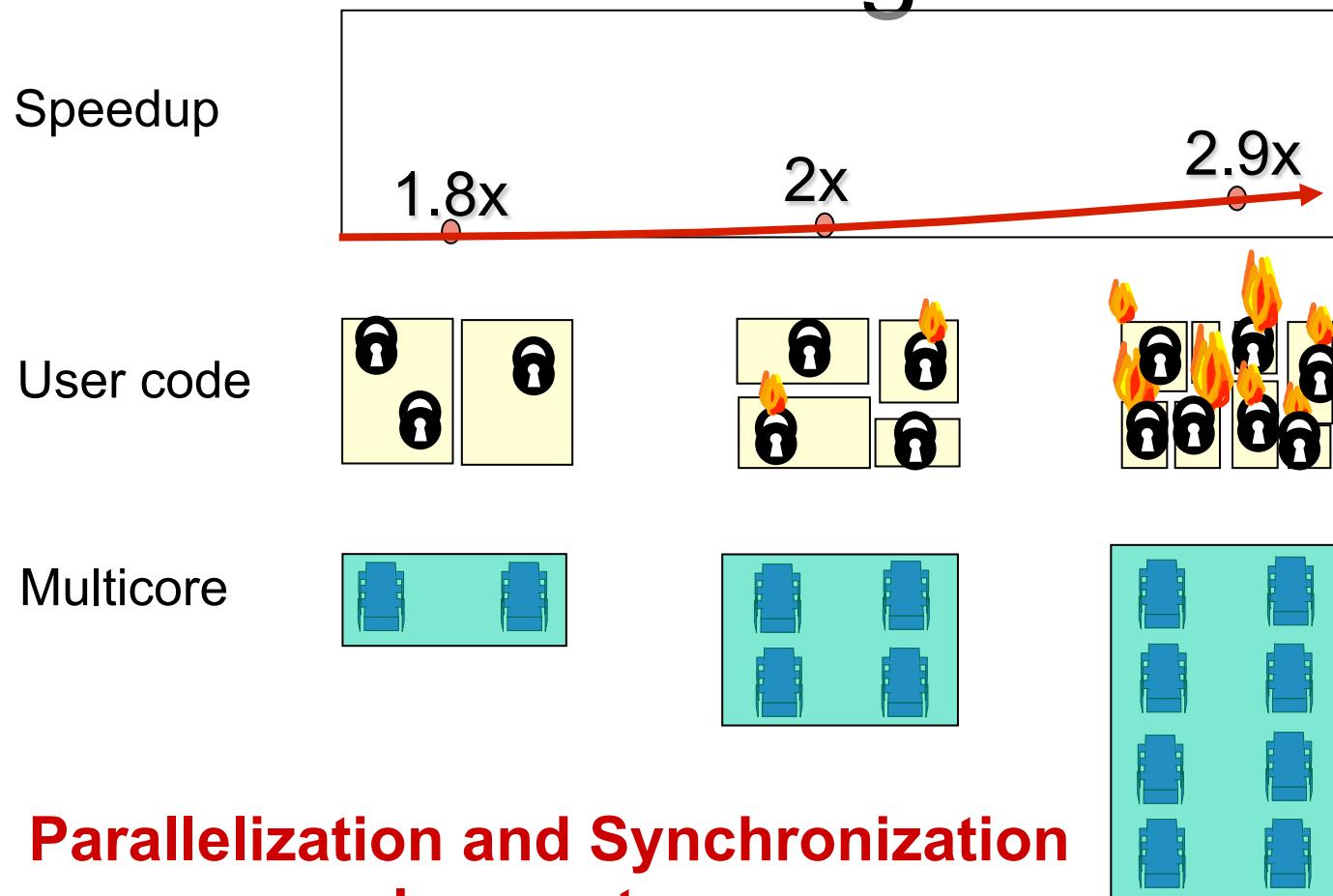


Ideal Scaling Process

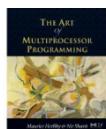


Art of Multiprocessor
Programming

Actual Scaling Process



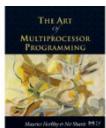
**Parallelization and Synchronization
require great care...**



Art of Multiprocessor
Programming

Amdahl's Law

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

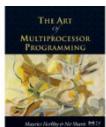


Art of Multiprocessor
Programming

Amdahl's Law

$$1/(1+p+n)$$

Speedup=

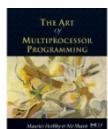
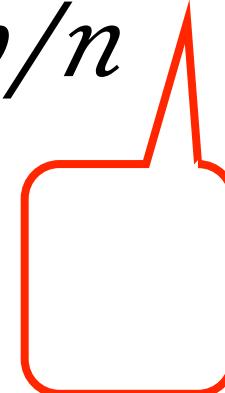


Art of Multiprocessor
Programming

Amdahl's Law

Speedup =

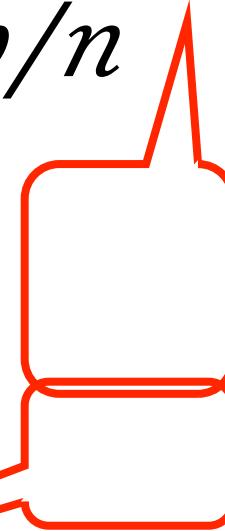
$$1/(1+p+p/n)$$



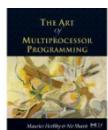
Amdahl's Law

Speedup =

$$1/(1+p+p/n)$$



Number of
threads



Amdahl's Law

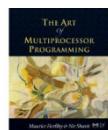
**sequential
fraction**

Speedup=

$$1/(1+p+p/n)$$

**parallel
fraction**

**Number of
threads**



Amdal's Law



Bad synchronization ruins everything

Example

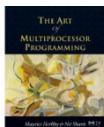
You buy a 10-core machine ...

Your application is:

60% concurrent

40% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine ...

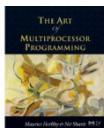
Your application is:

60% concurrent

40% sequential

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

How close to a 10-fold speedup?



Example

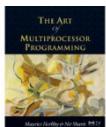
You buy a 10-core machine ...

Your application is:

80% concurrent

20% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine ...

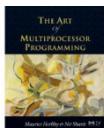
Your application is:

80% concurrent

20% sequential

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

How close to a 10-fold speedup?



Example

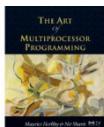
You buy a 10-core machine ...

Your application is:

90% concurrent

10% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine ...

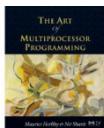
Your application is:

80% concurrent

20% sequential

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

How close to a 10-fold speedup?



Example

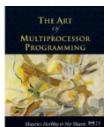
You buy a 10-core machine ...

Your application is:

99% concurrent

01% sequential

How close to a 10-fold speedup?



Example

You buy a 10-core machine ...

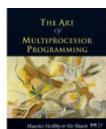
Your application is:

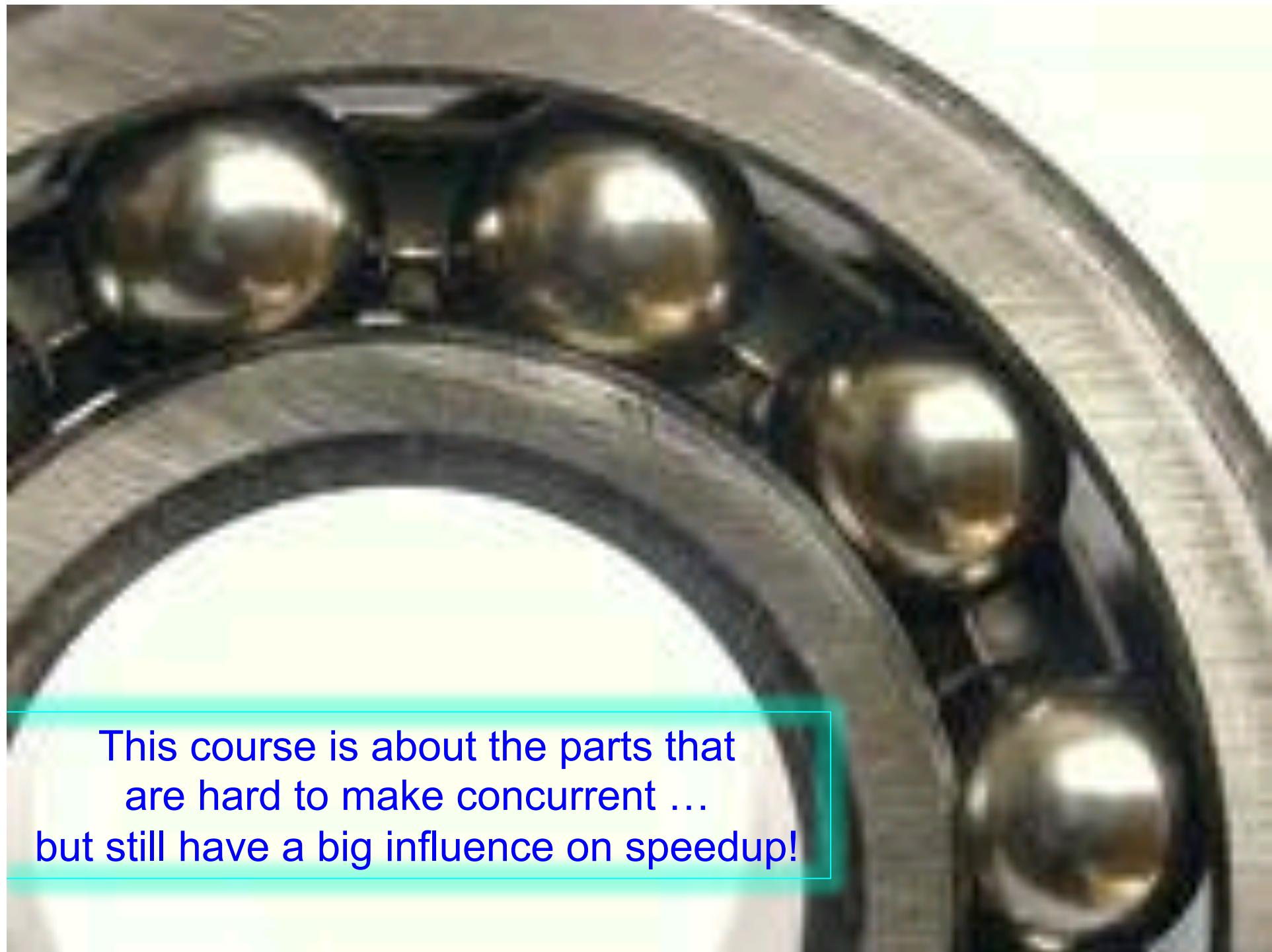
80% concurrent

20% sequential

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

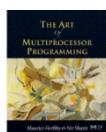
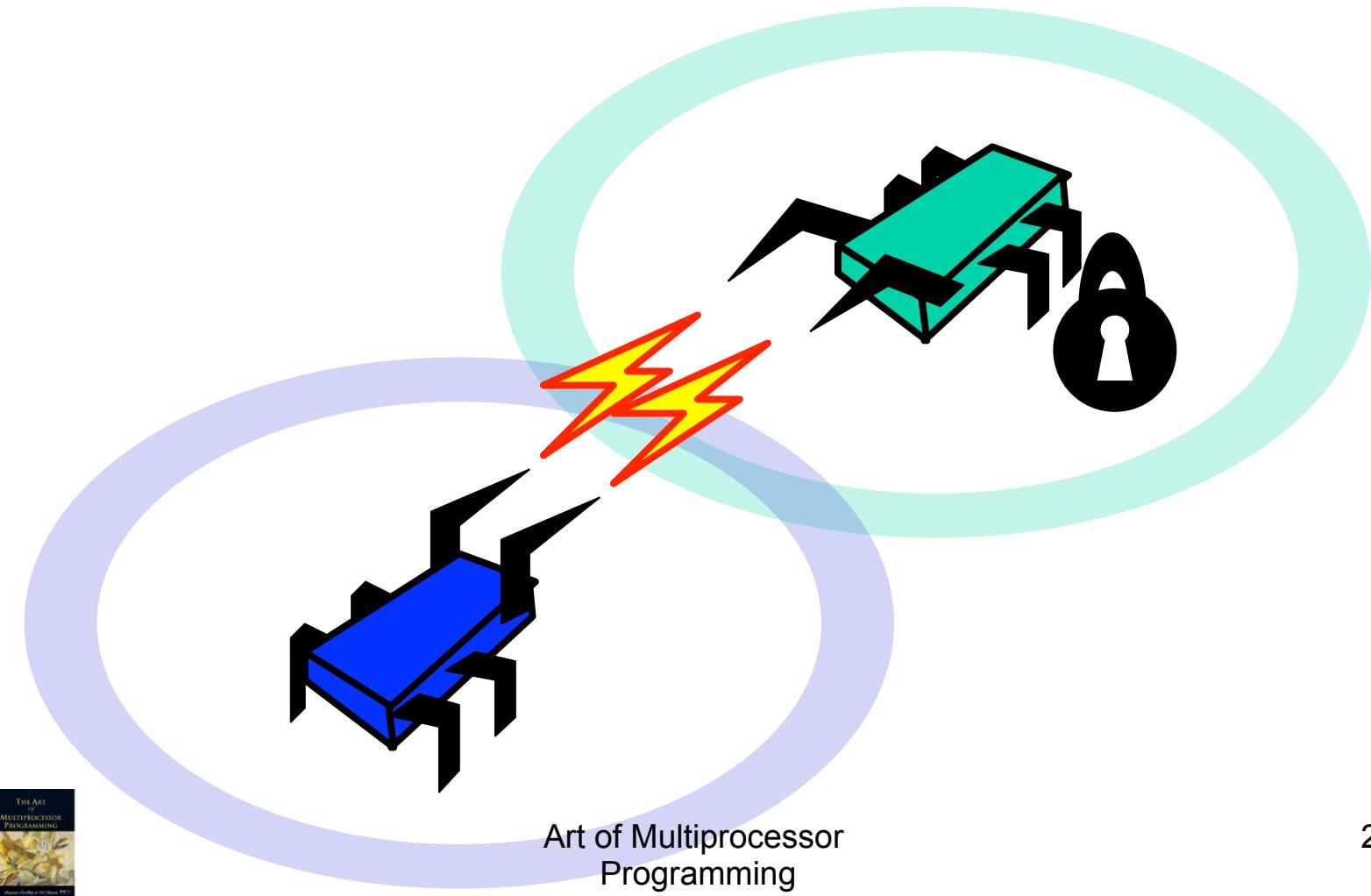
How close to a 10-fold speedup?





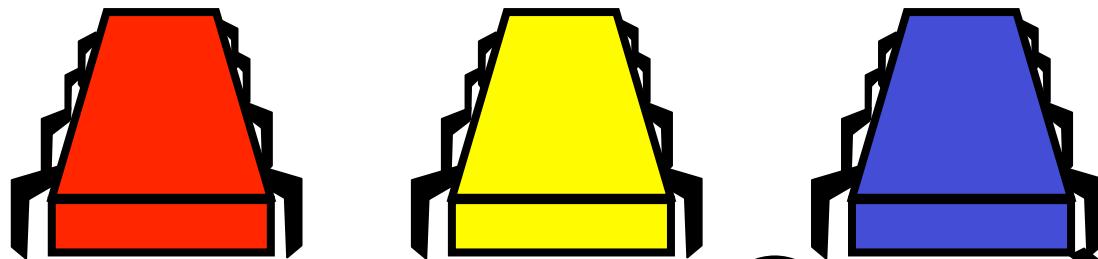
This course is about the parts that
are hard to make concurrent ...
but still have a big influence on speedup!

Locking

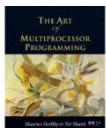
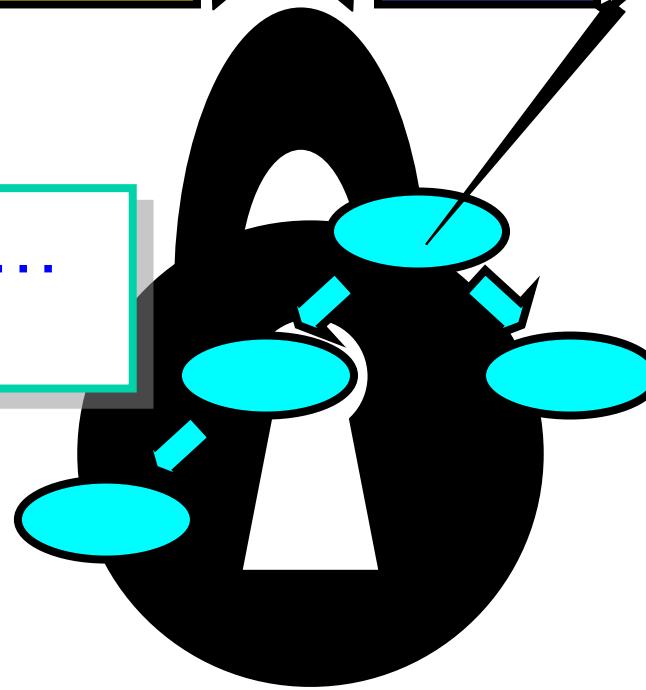


Art of Multiprocessor
Programming

Coarse-Grained Locking

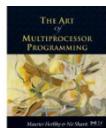
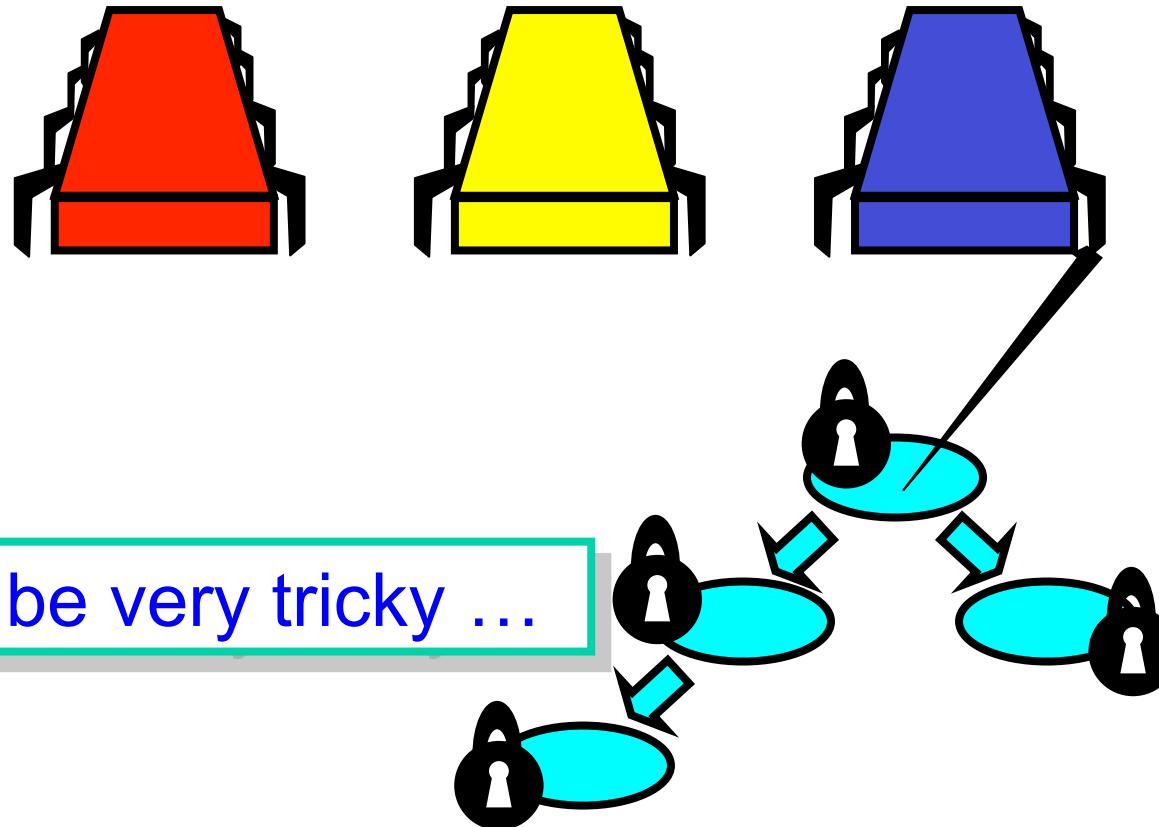


Easily made correct ...
But not scalable.



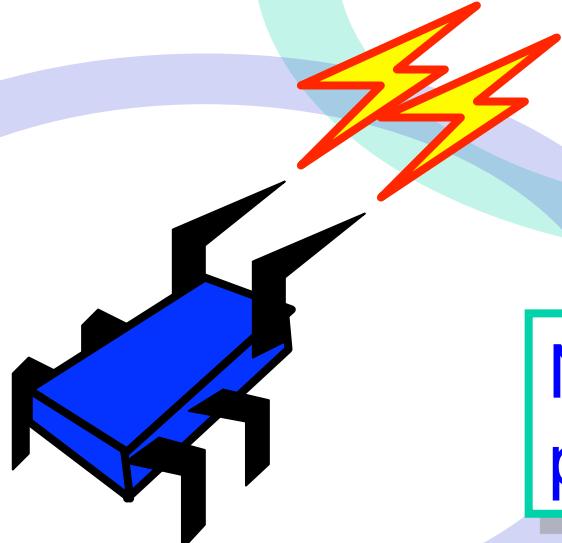
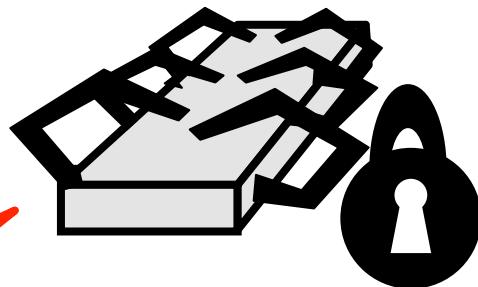
Art of Multiprocessor
Programming

Fine-Grained Locking

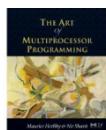


Locks are not Robust

If a thread holding
a lock is delayed ...



No one else can make
progress



Locking Relies on Conventions

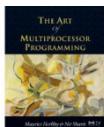
Relation between ...

Lock data and object data

Exists only in programmer's

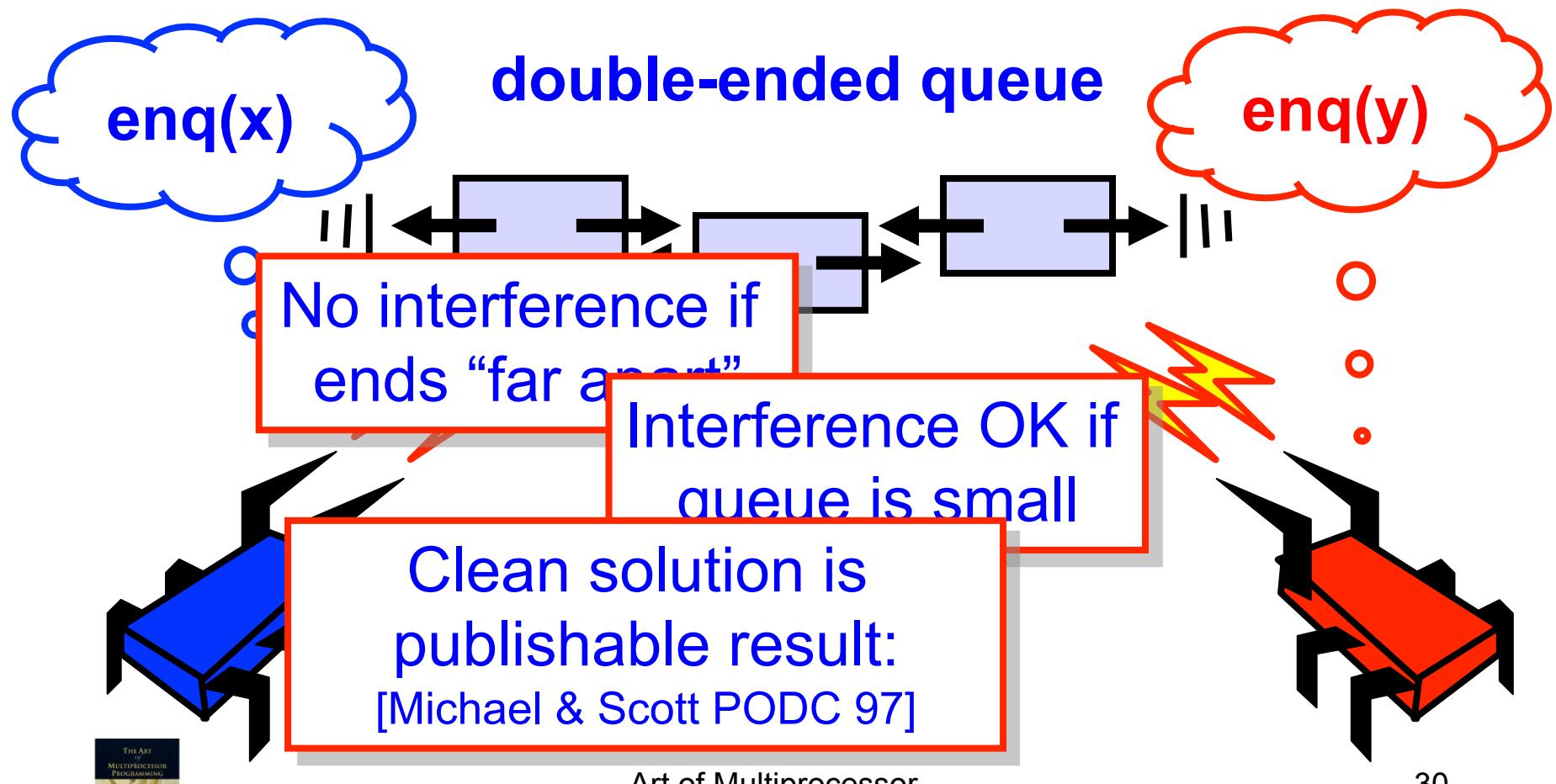
Actual comment
from Linux Kernel
(hat tip: Bradley Kuszmaul)

```
/*
 * 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.
 */
```

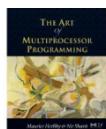
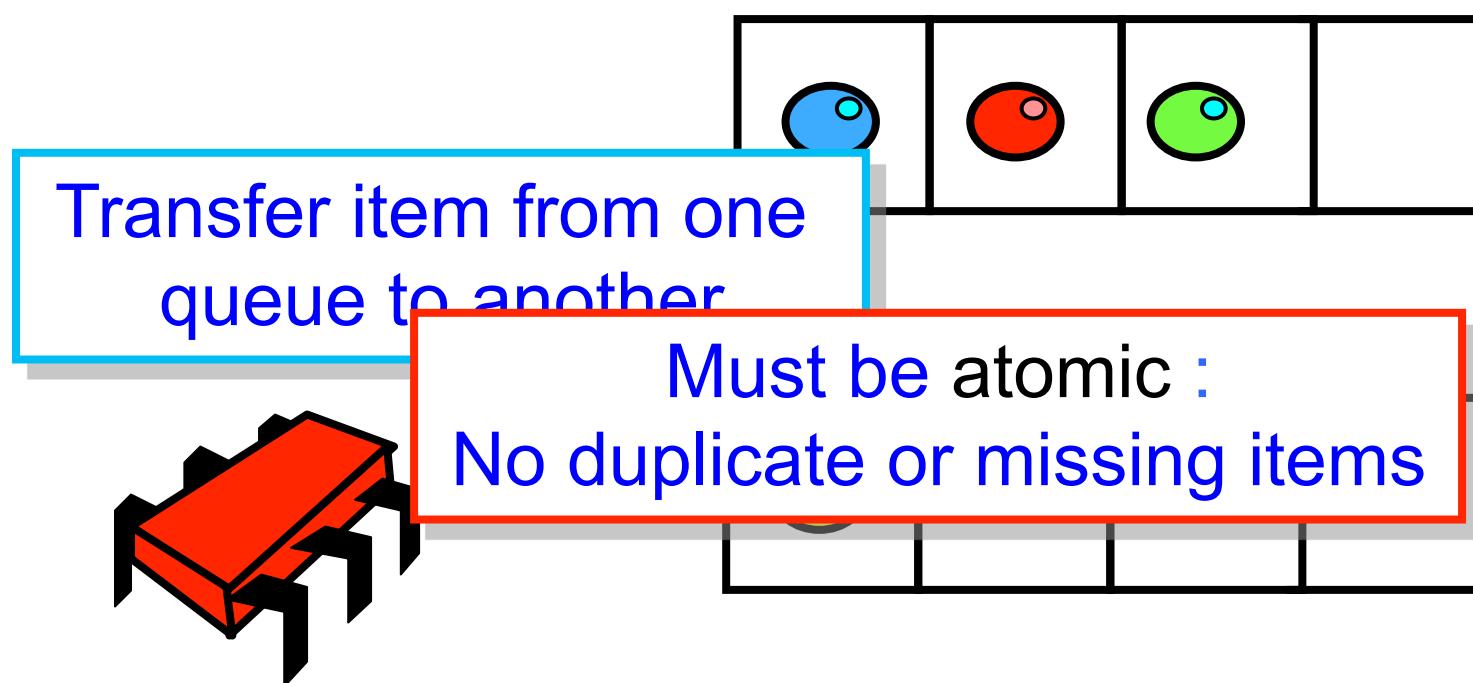


Art of Multiprocessor
Programming

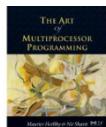
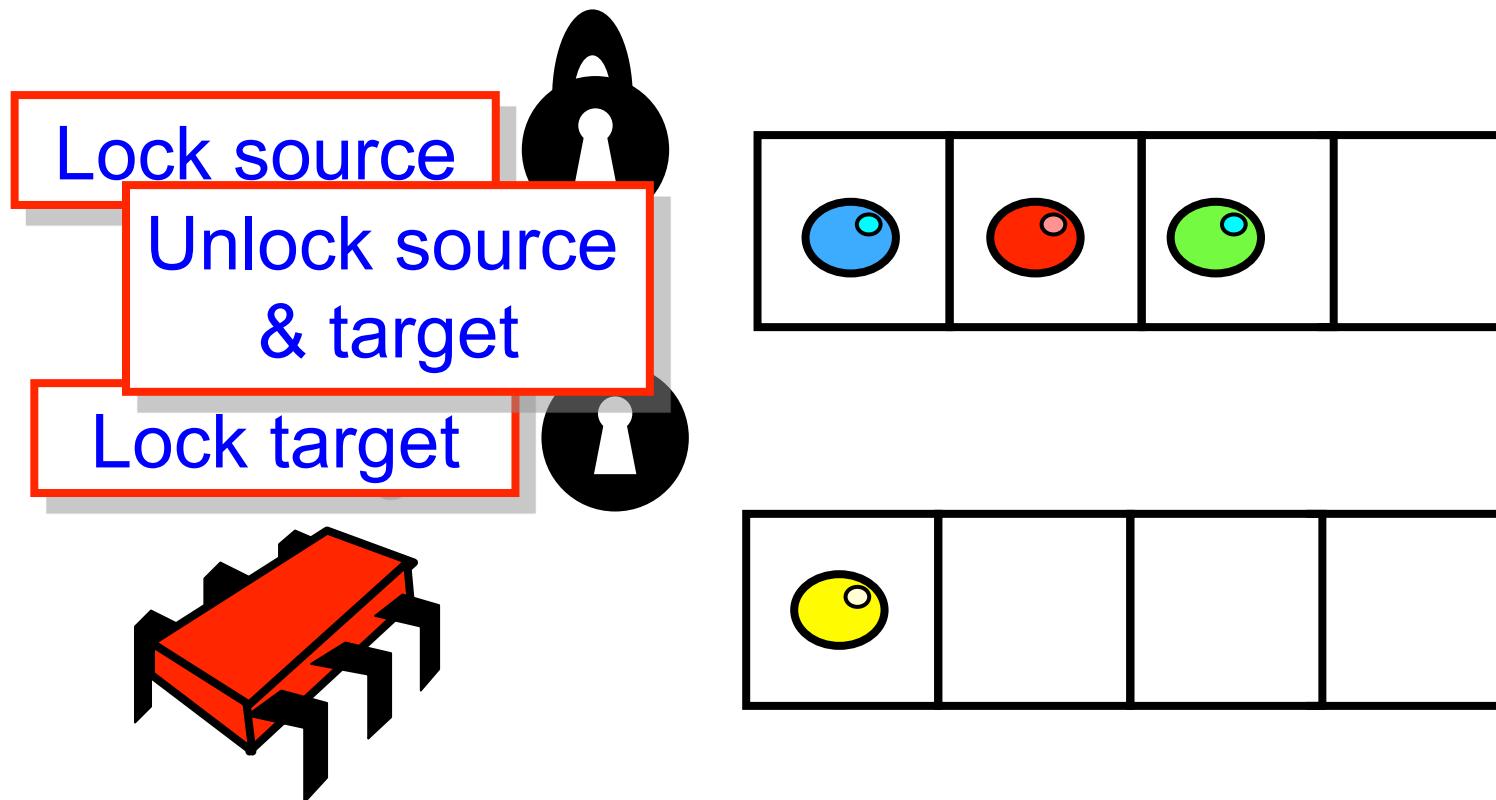
Simple Problems are hard



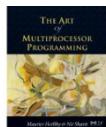
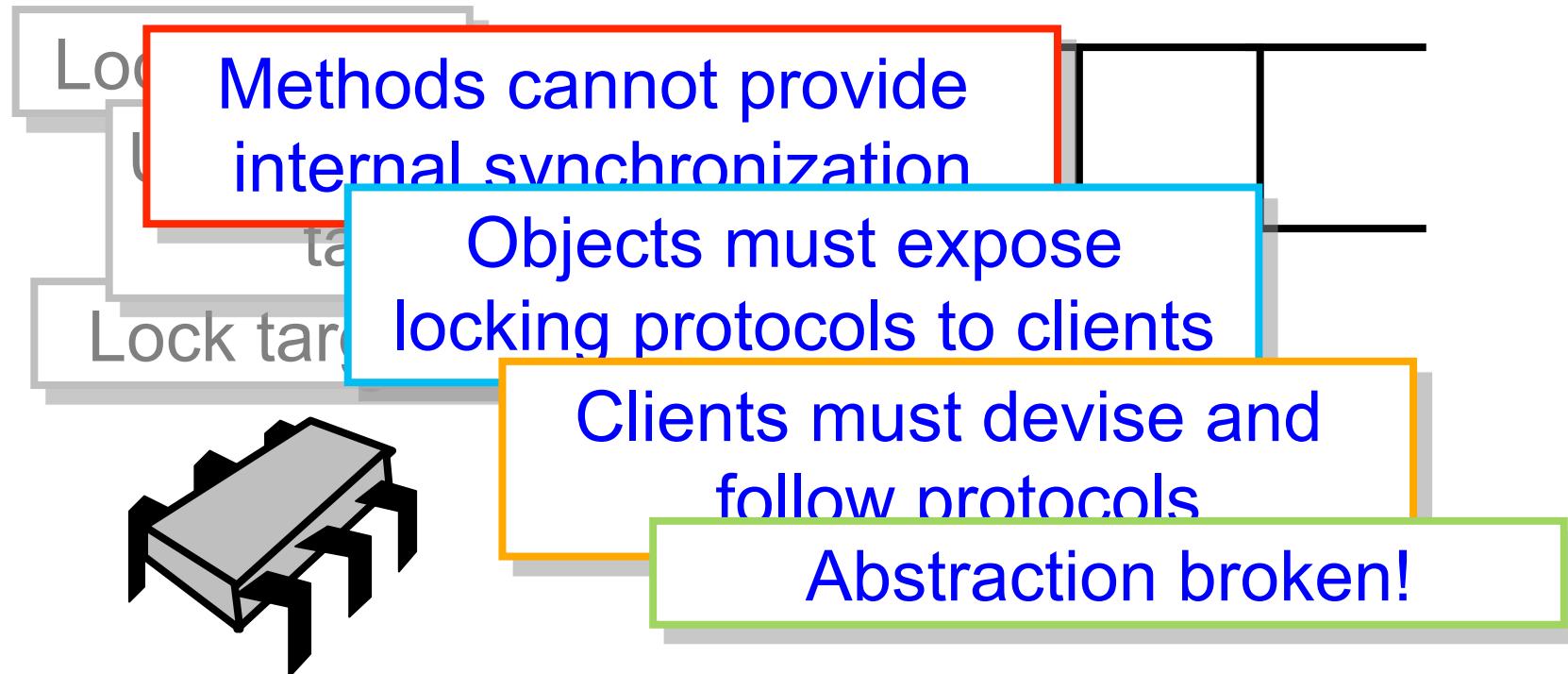
Locks Not Composable



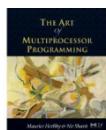
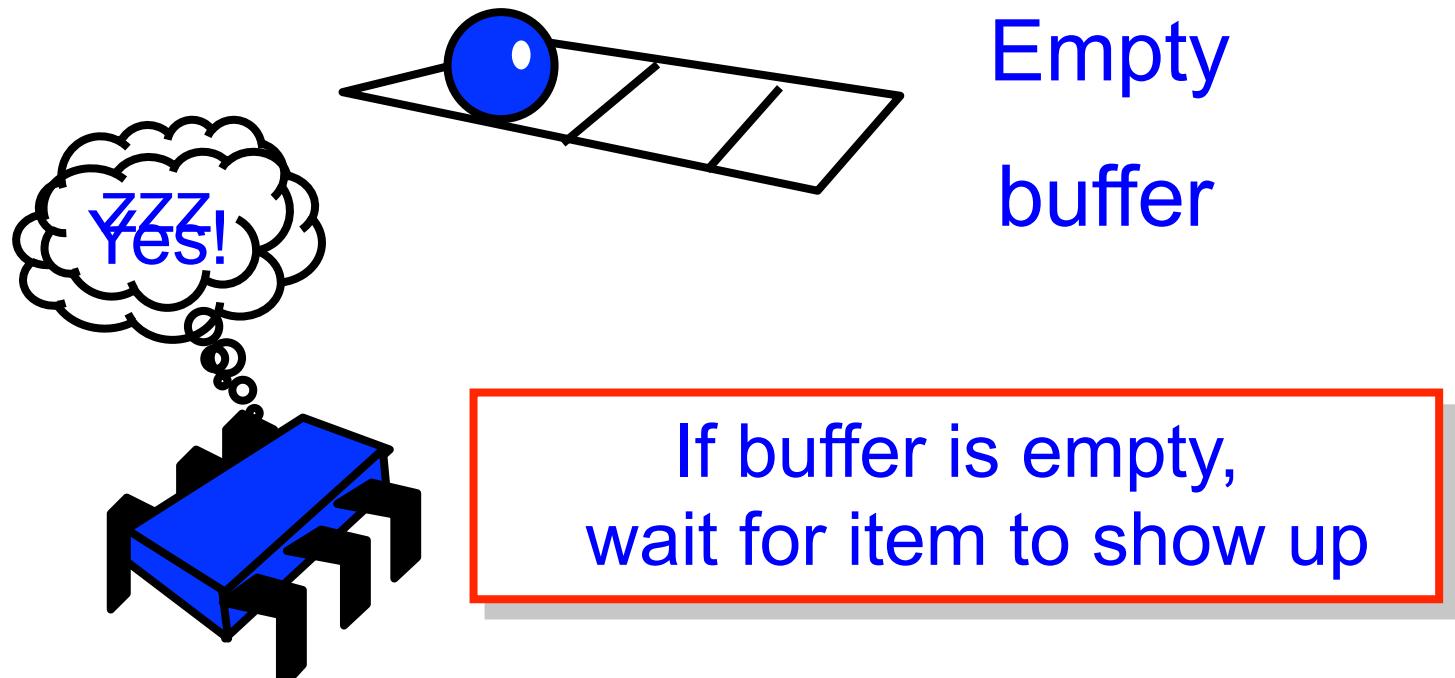
Locks Not Composable



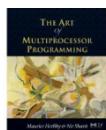
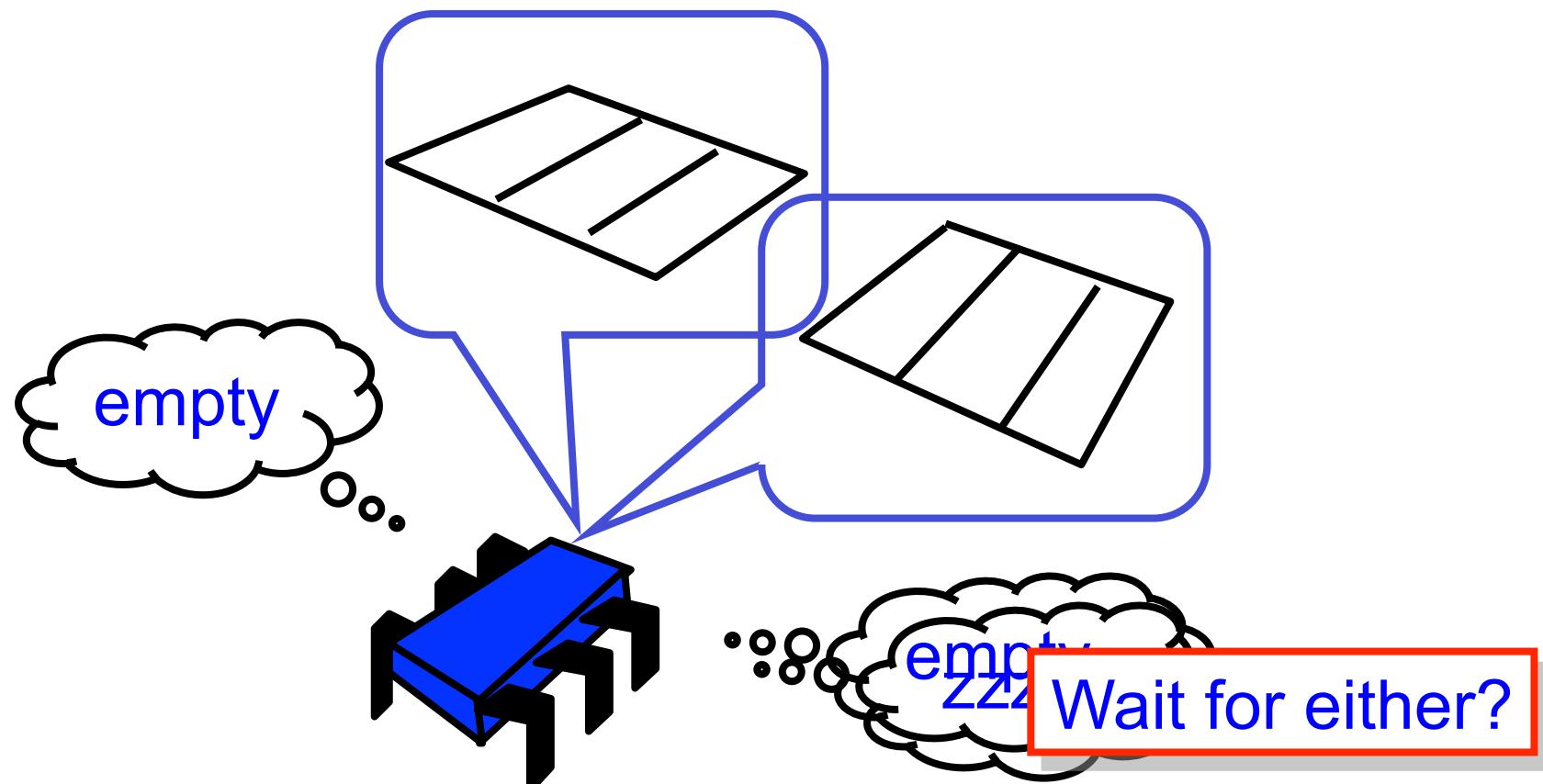
Locks Not Composable



Monitor Wait and Signal

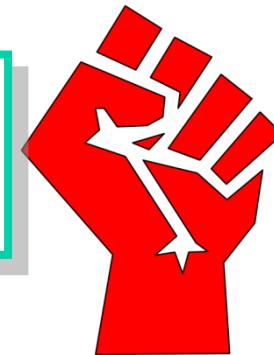


Wait and Signal do not Compose



The Transactional Manifesto

Much modern programming practice
inadequate for multicore world

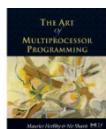


Agenda

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

Design languages and libraries

Implement efficient run-times



Road Map

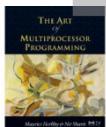
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

Research Questions



Road Map

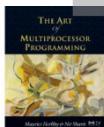
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

Research Questions



Transactions

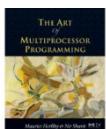
Block of code

Atomic: appears to happen
instantaneously

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

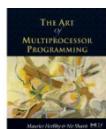
Commit: takes effect
(atomically)

Abort: has no effect
(typically restarted)



Atomic Blocks

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

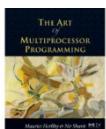


Atomic Blocks

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

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

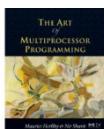
No data race



A Double-Ended Queue

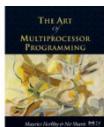
```
public void LeftEnq(item x) {  
    Qnode q = new Qnode(x);  
    q.left = left;  
    left.right = q;  
    left = q;  
}
```

Write sequential Code



A Double-Ended Queue

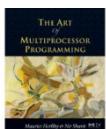
```
public void LeftEnq(item x)
atomic {
    Qnode q = new Qnode(x);
    q.left = left;
    left.right = q;
    left = q;
}
```



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

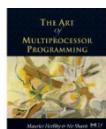
Not always this simple!

Conditional waits?

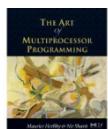
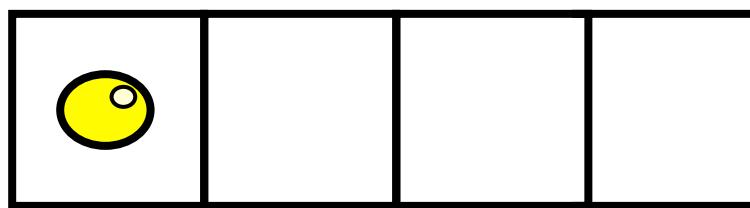
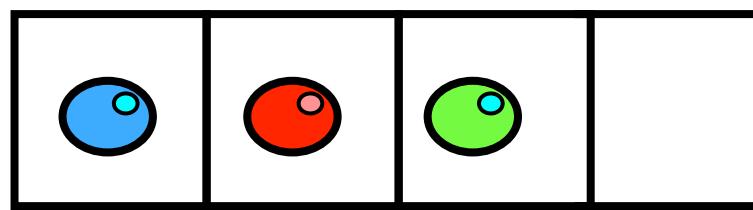
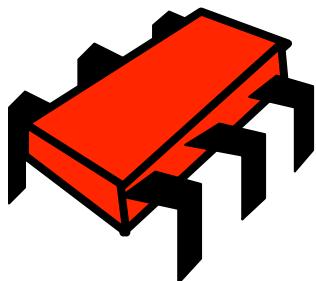
False conflicts?

Resource limits?

Better problems to have ...



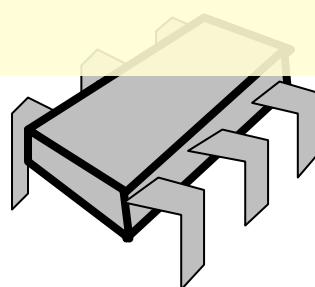
Composition?



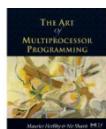
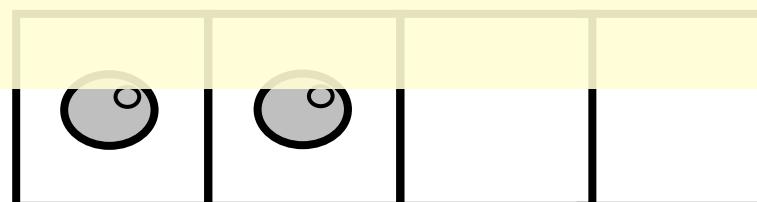
Art of Multiprocessor
Programming

Composition?

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



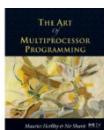
Trivial or what?



Conditional Waiting

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

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



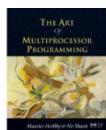
Composable Conditional Waiting

```
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

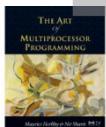
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

Research Questions



Hardware Transactional Memory

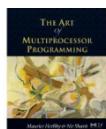
Exploit standard “cache coherence”



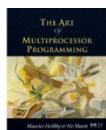
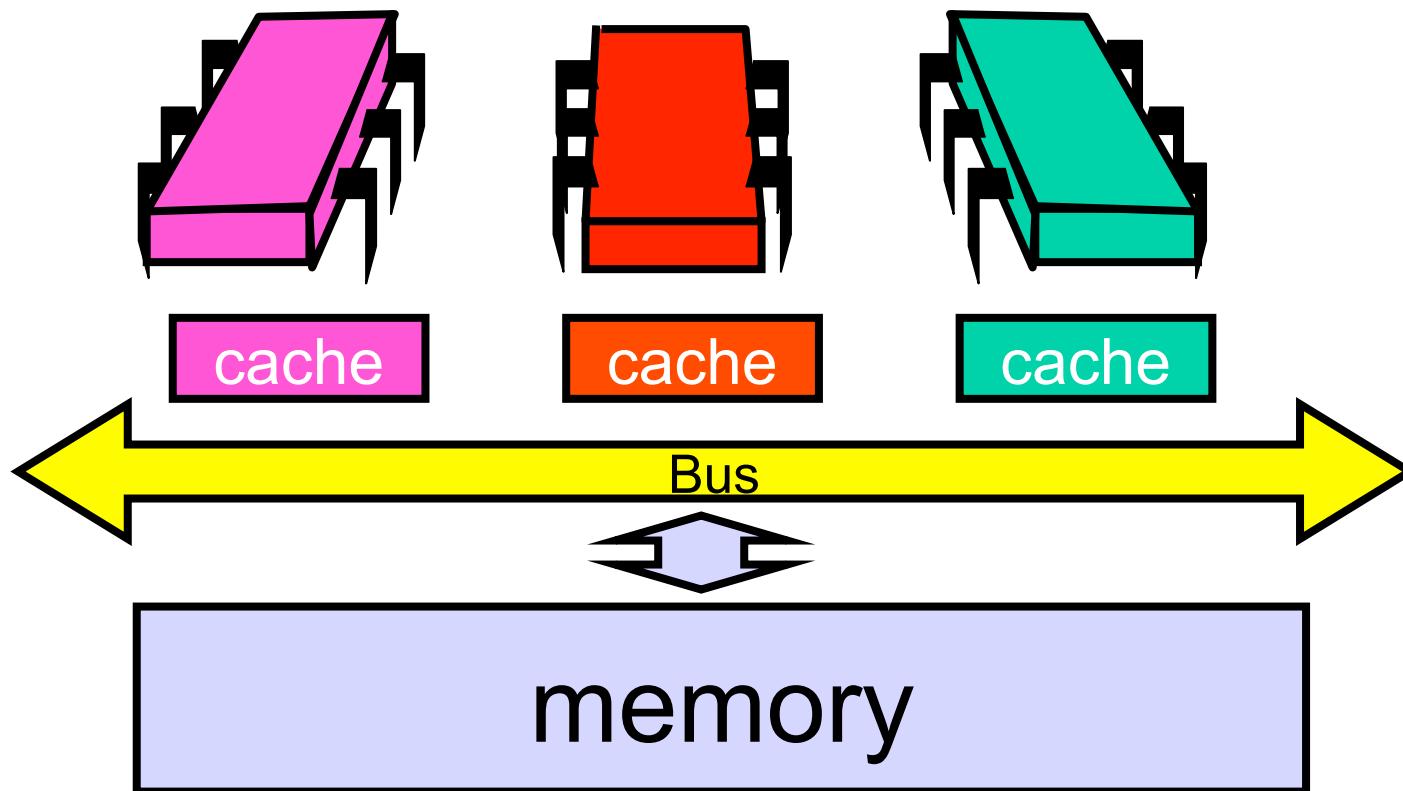
Detect synchronization conflicts ...



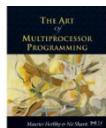
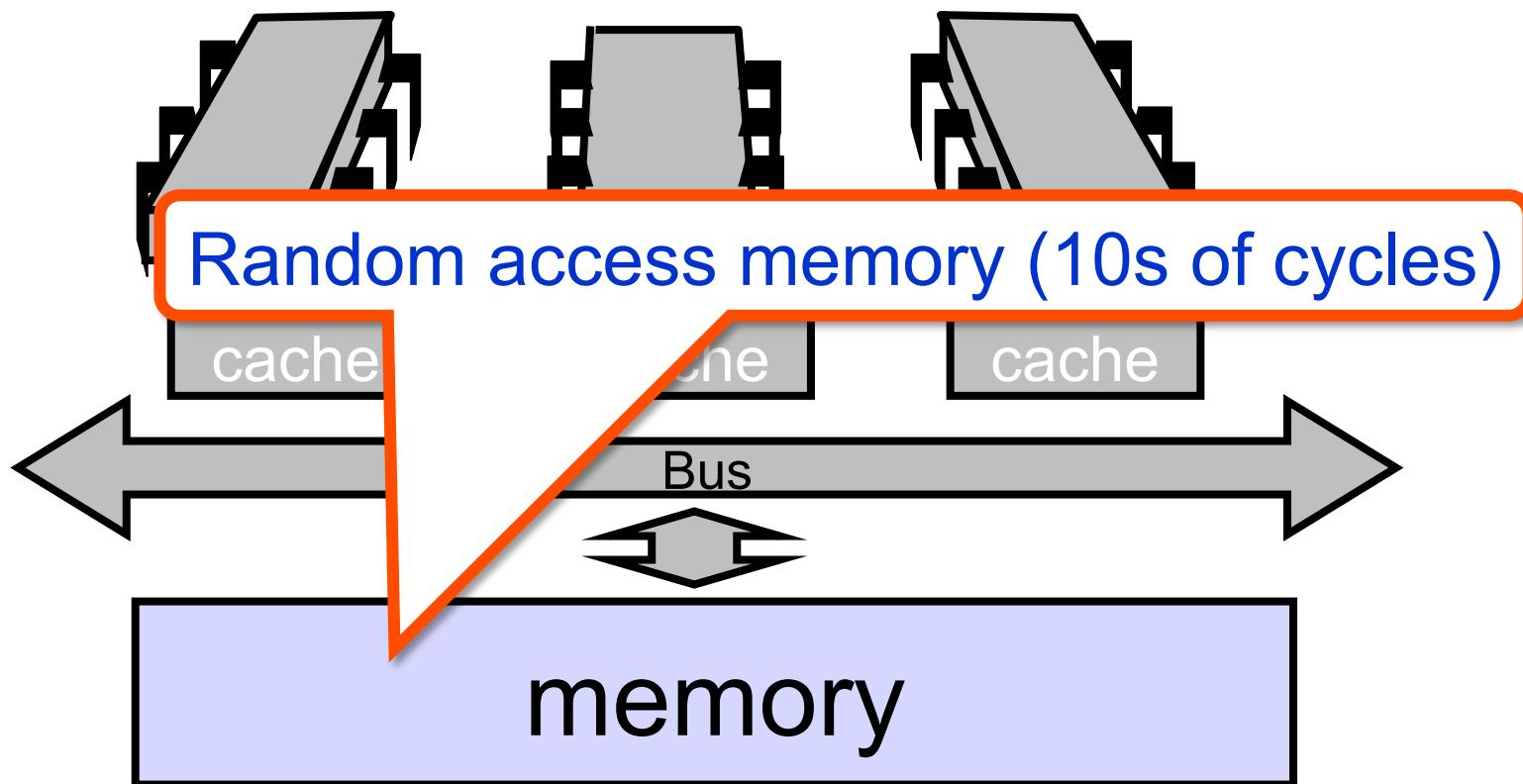
Invalidate cached copies of data.



Standard Cache Coherence



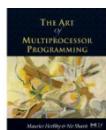
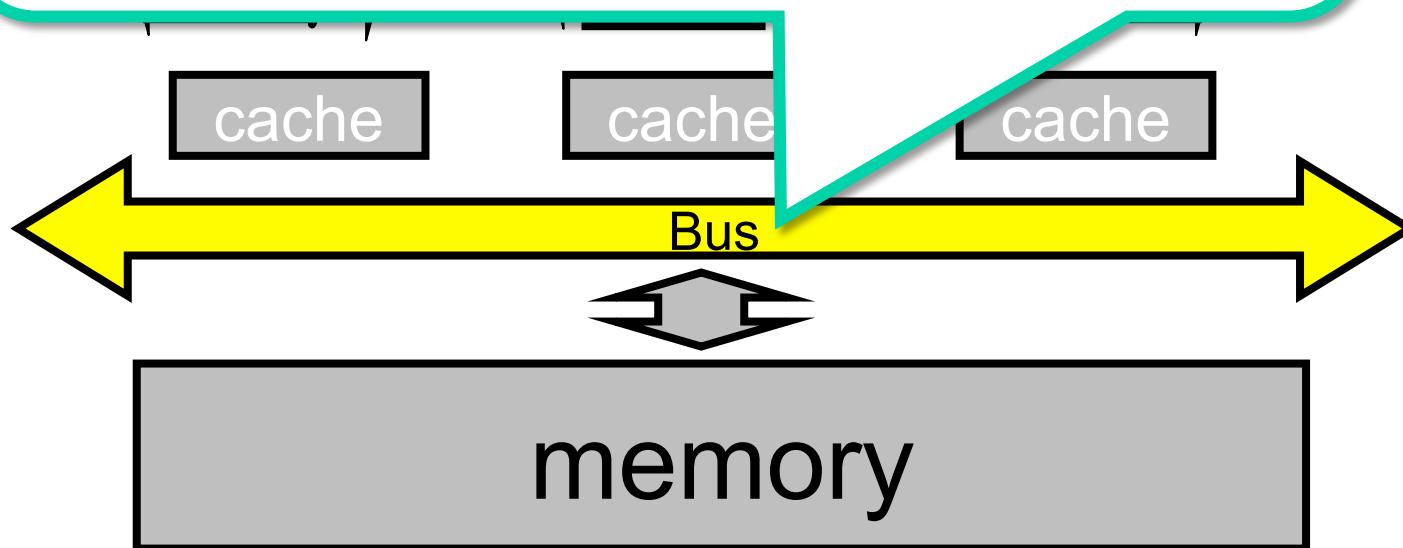
Standard Cache Coherence



Standard Cache Coherence

Shared Bus

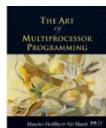
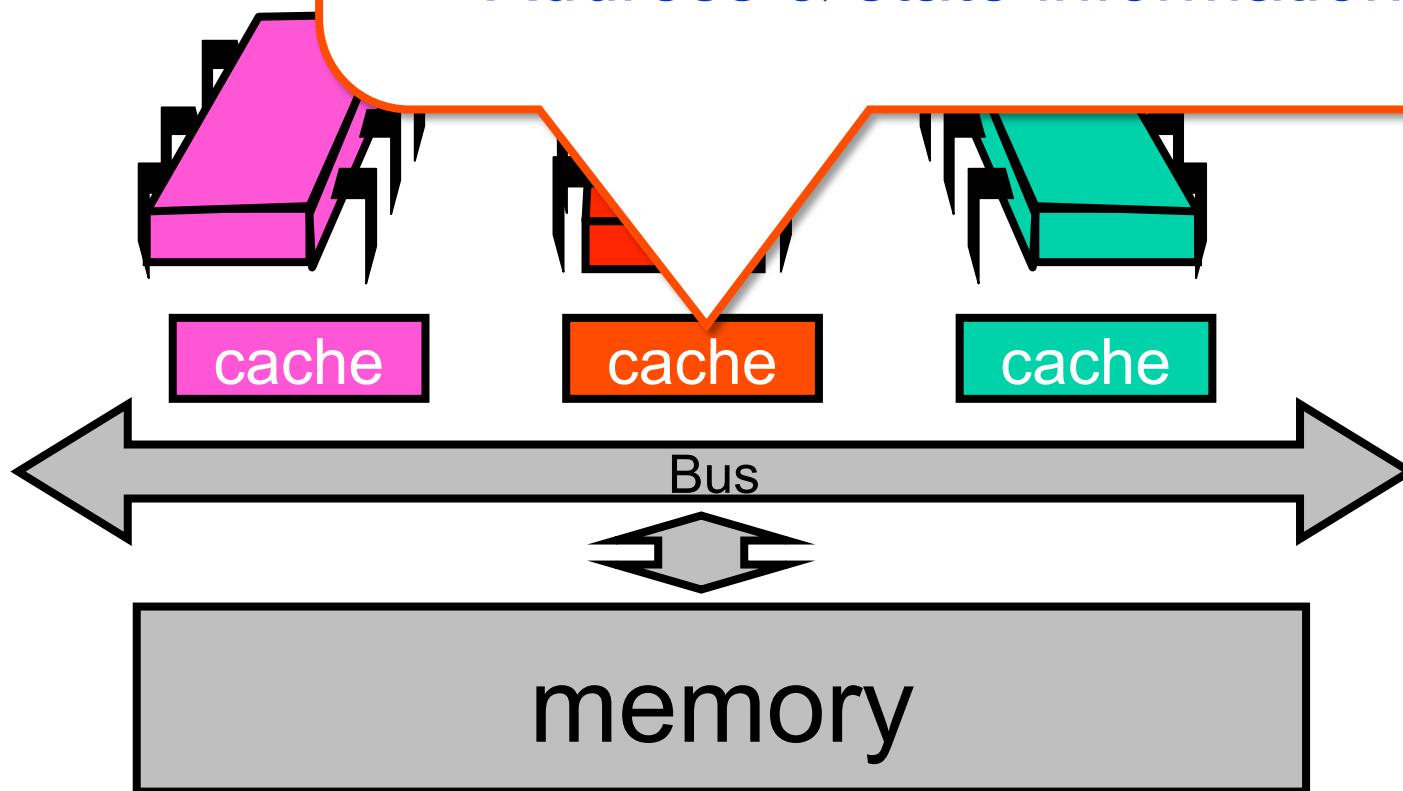
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”



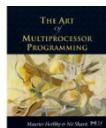
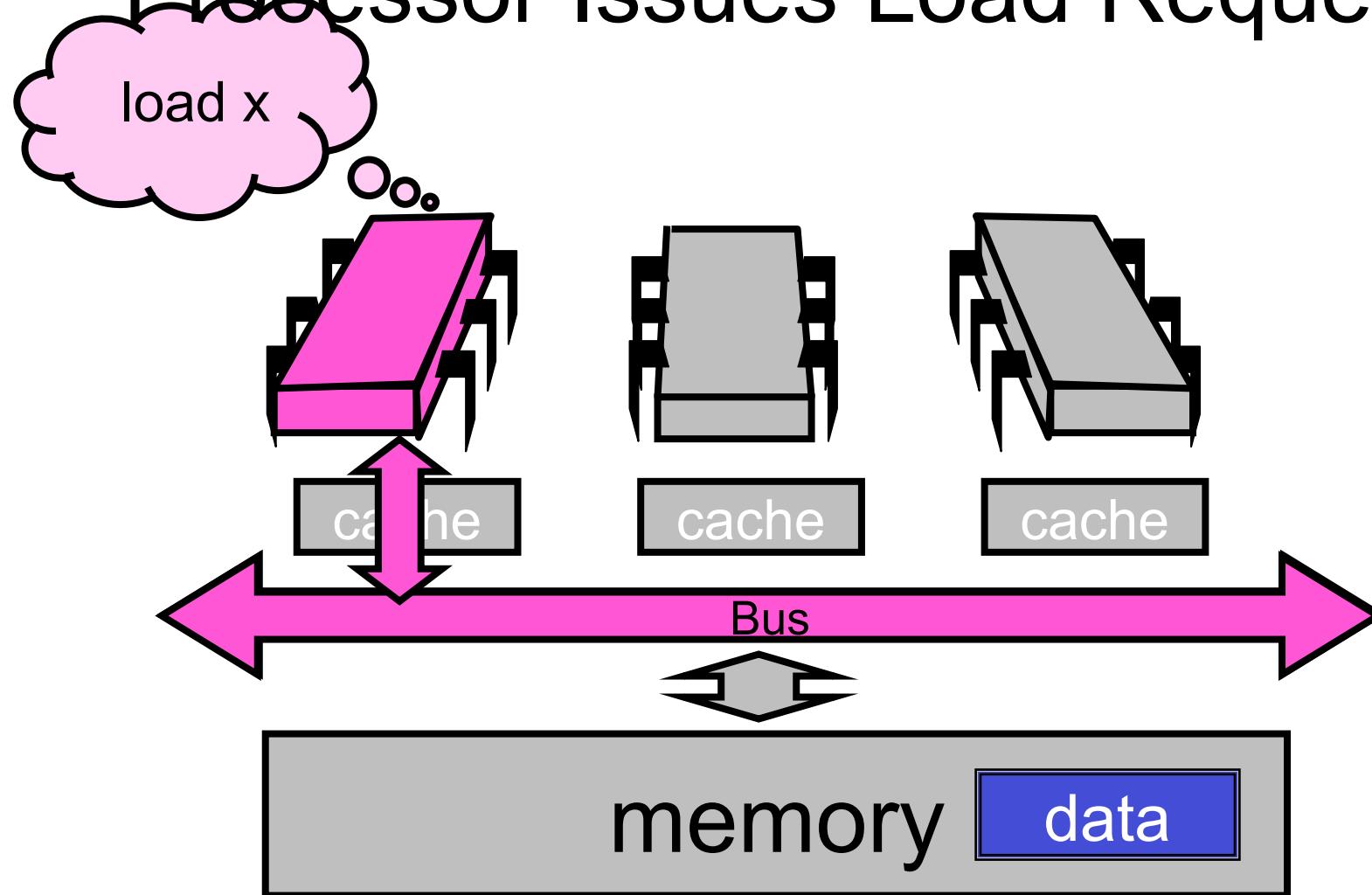
Standards

Per-Processor Caches

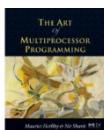
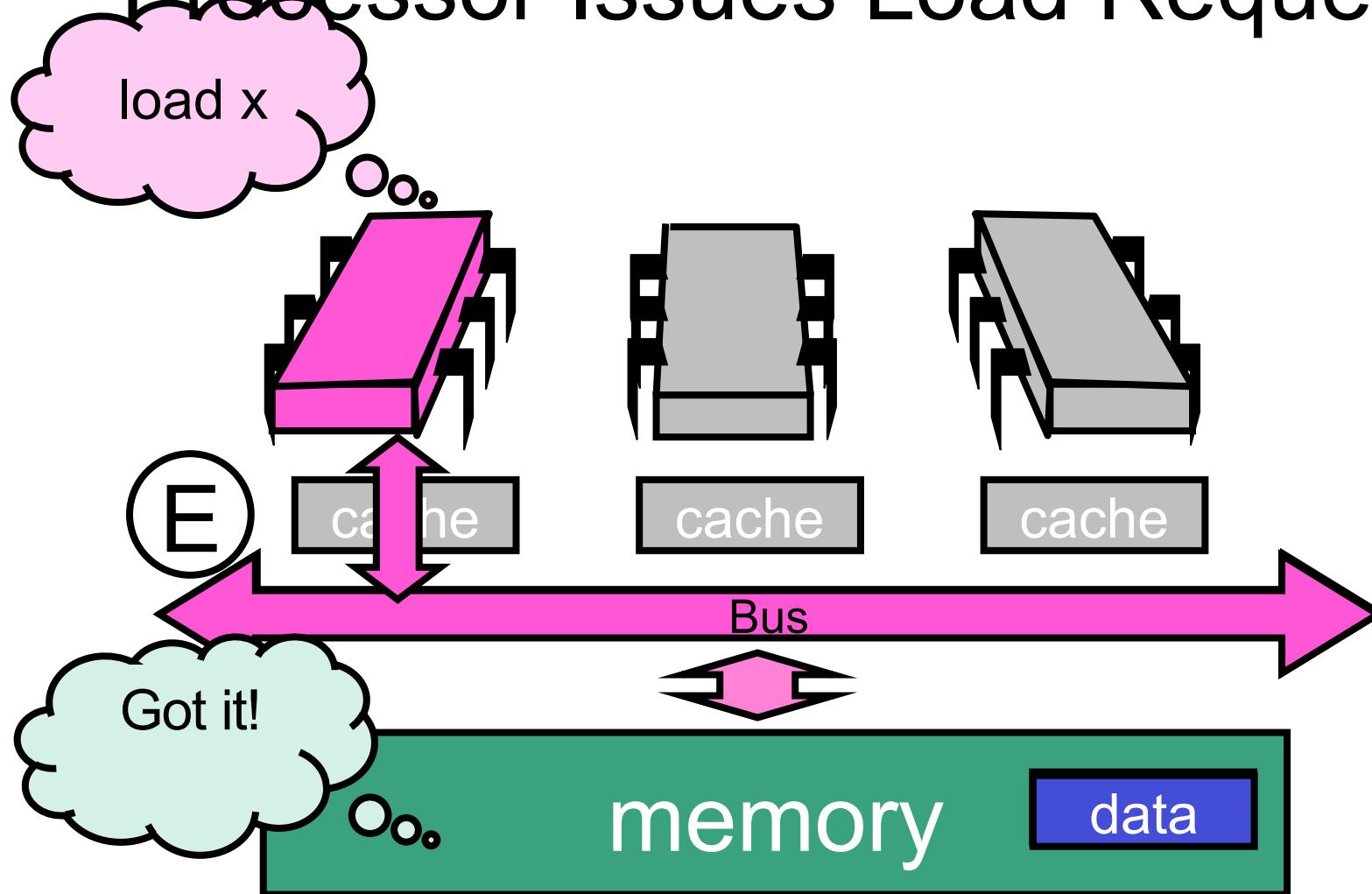
- Small
- Fast: 1 or 2 cycles
- Address & state information



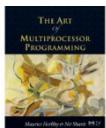
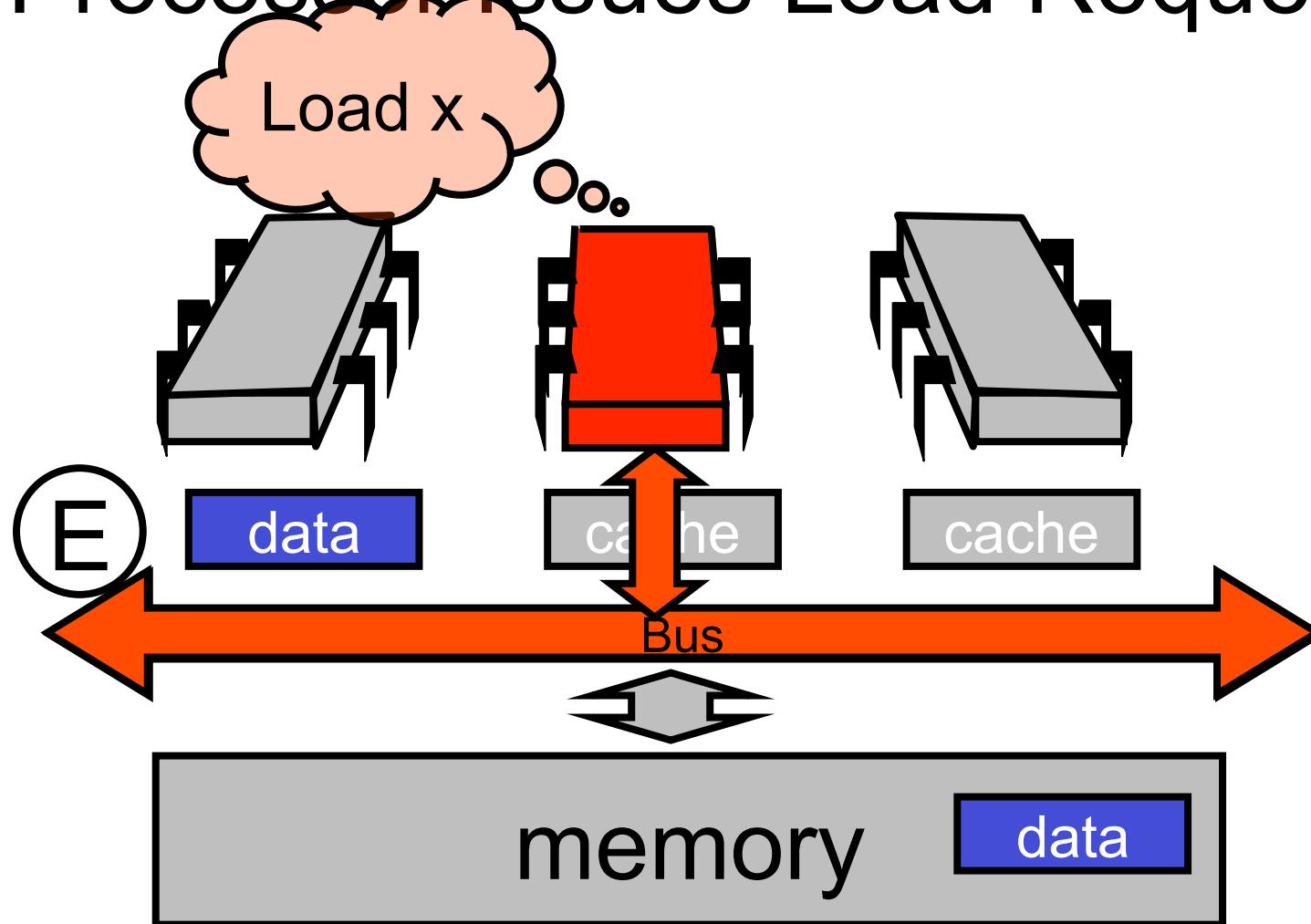
Processor Issues Load Request



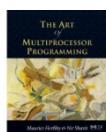
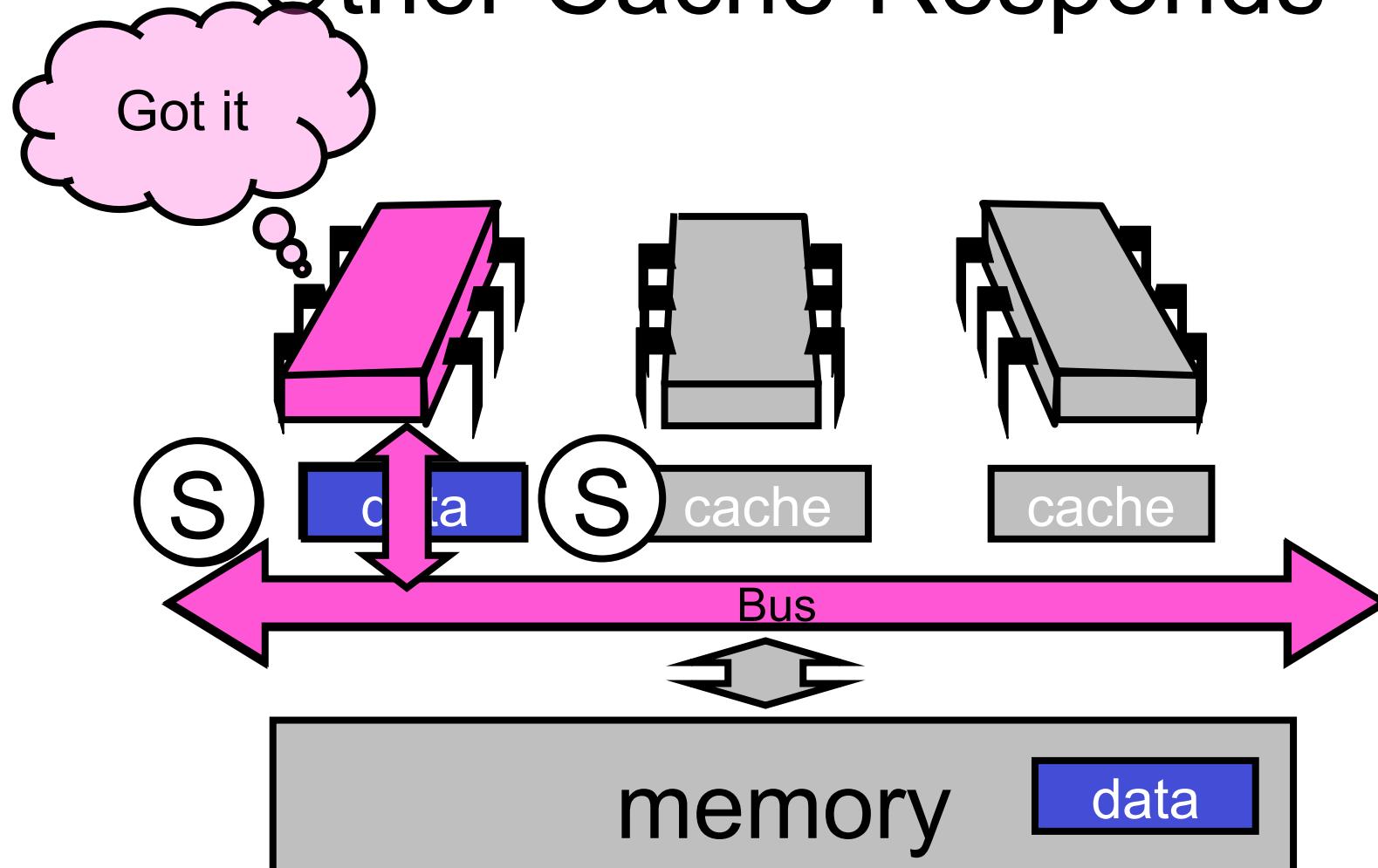
Processor Issues Load Request



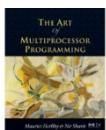
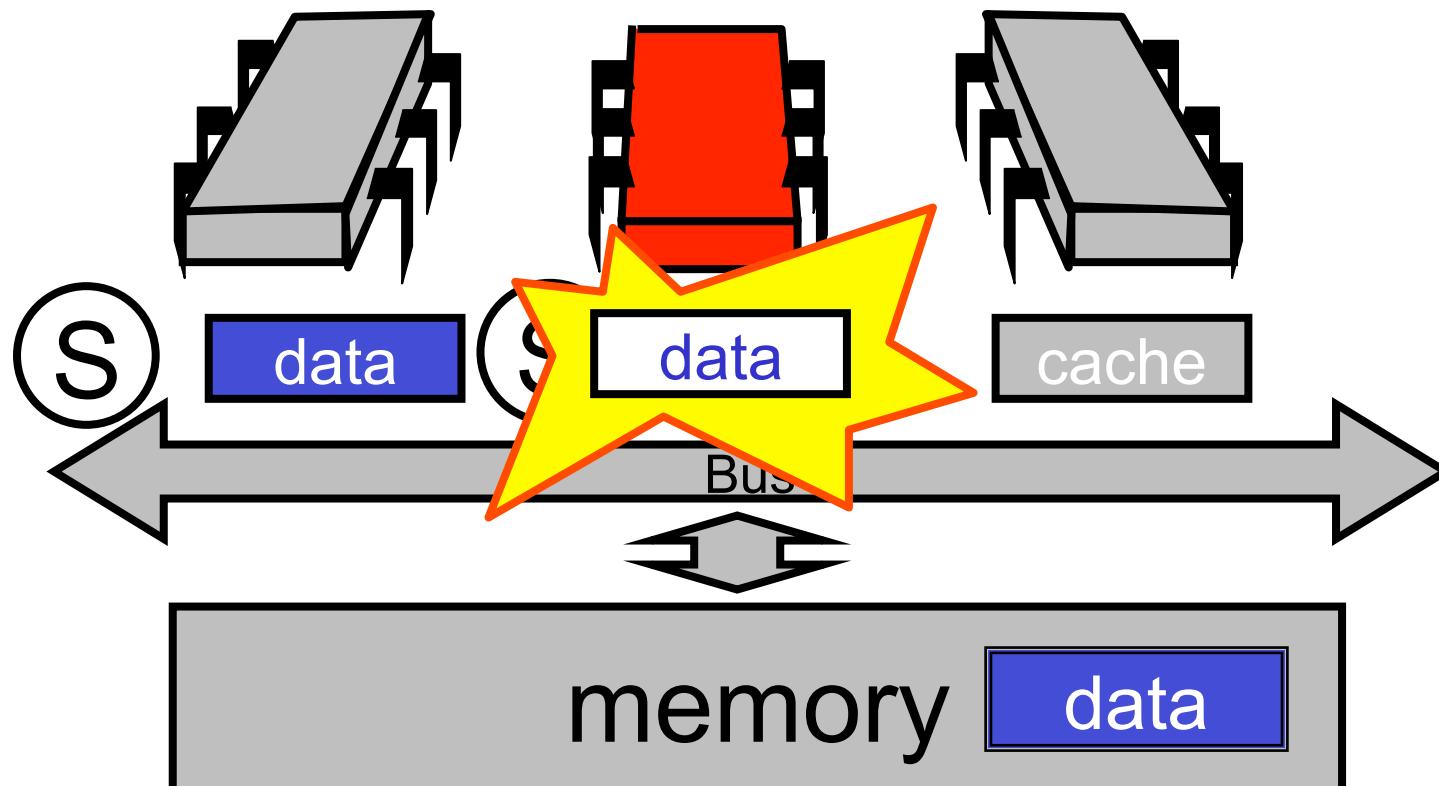
Processor Issues Load Request



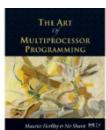
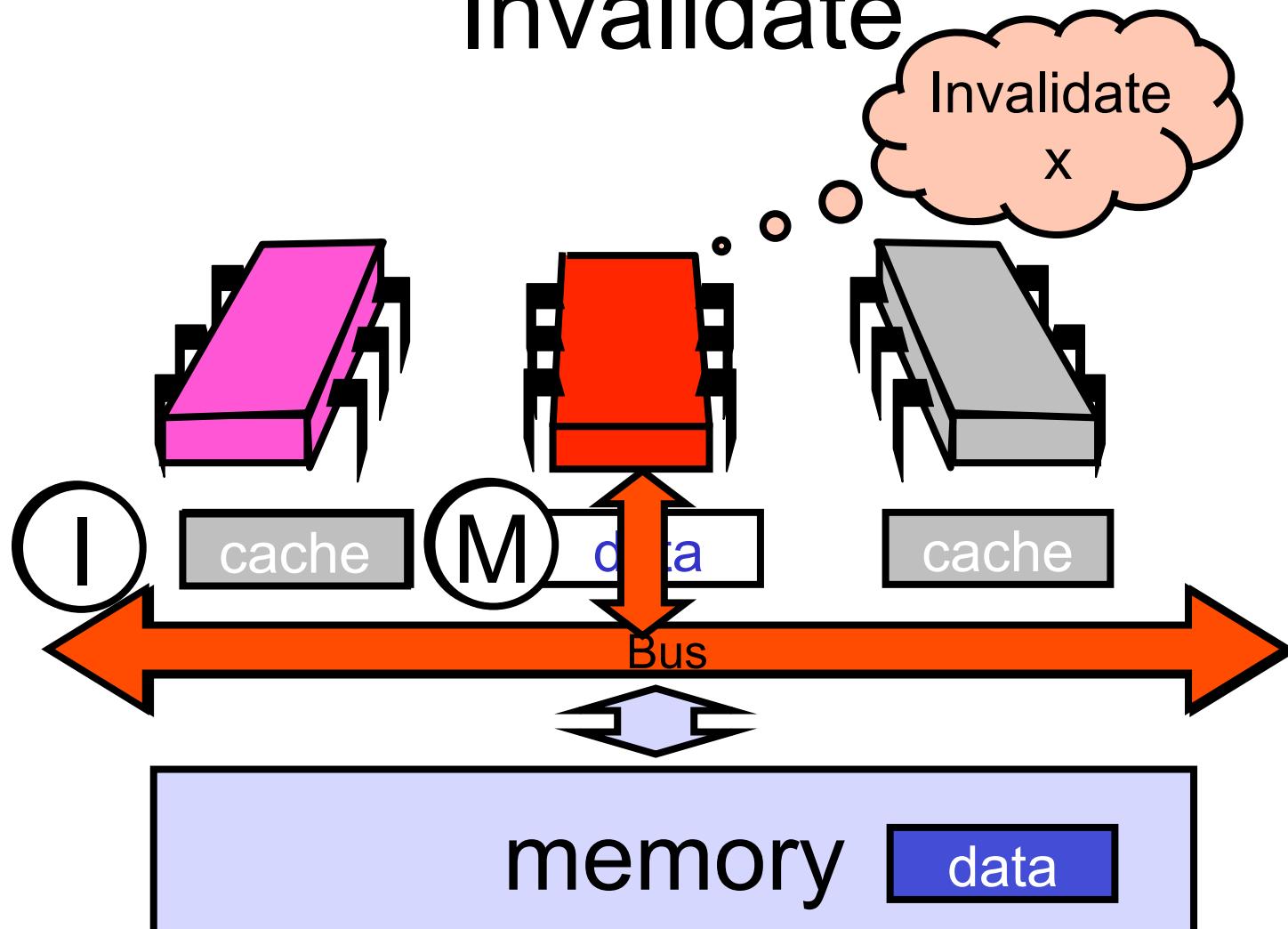
Other Cache Responds



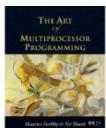
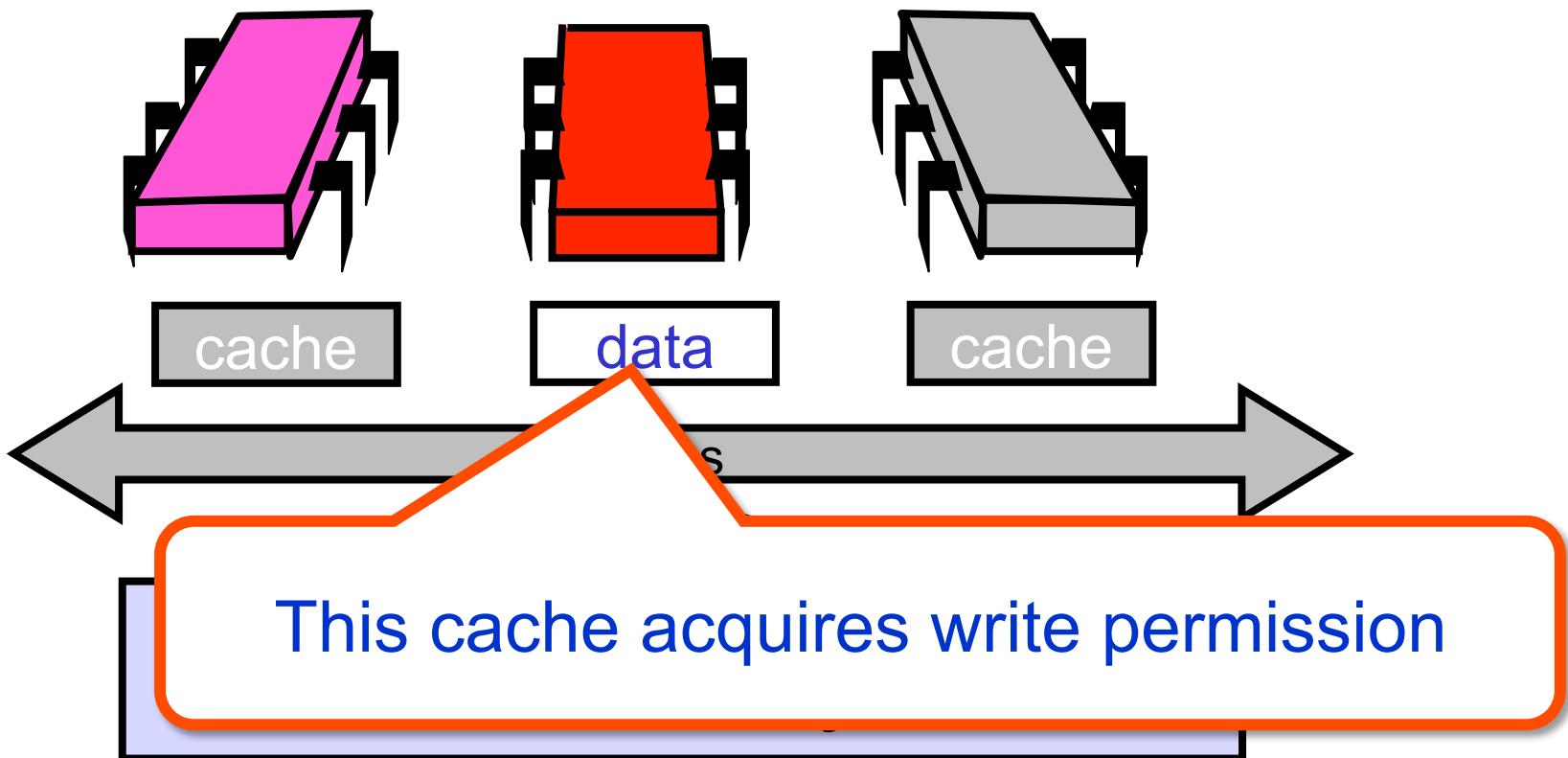
Modify Cached Data



Invalidate

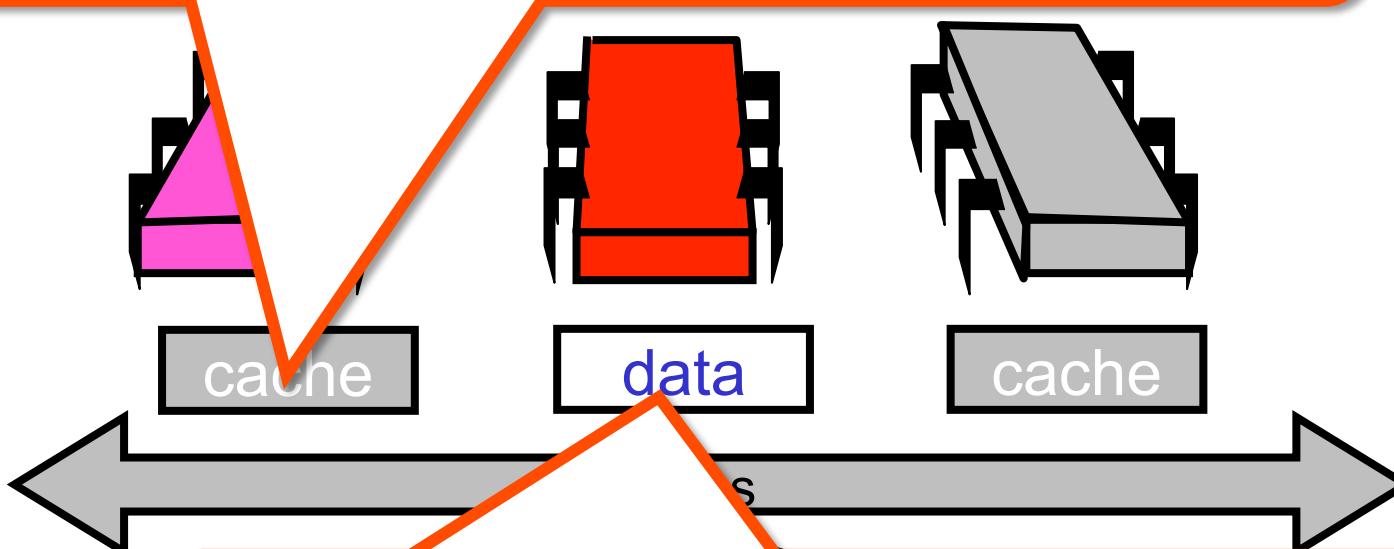


Invalidate

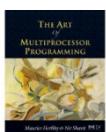


Invalidate

Other caches lose read permission

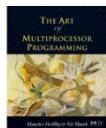
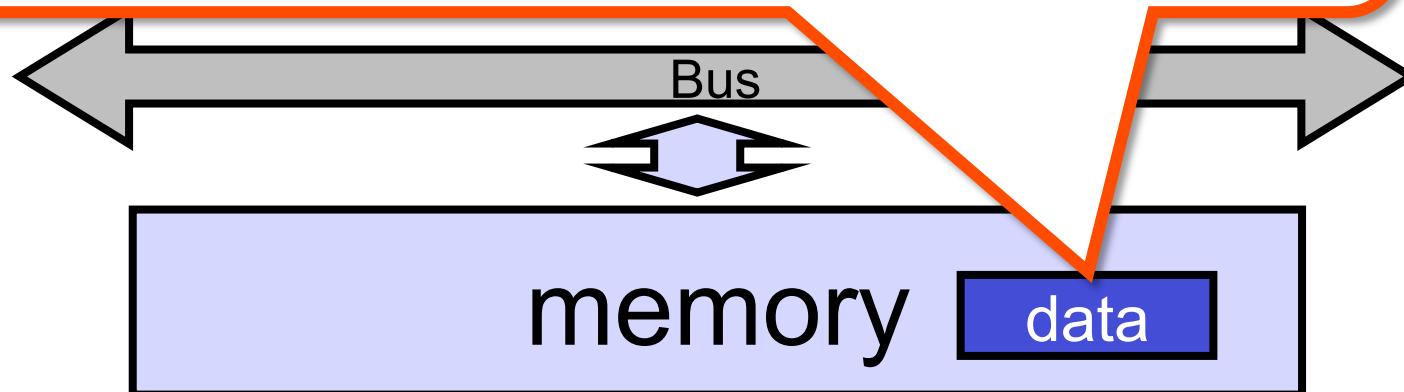


This cache acquires write permission

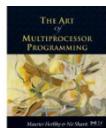
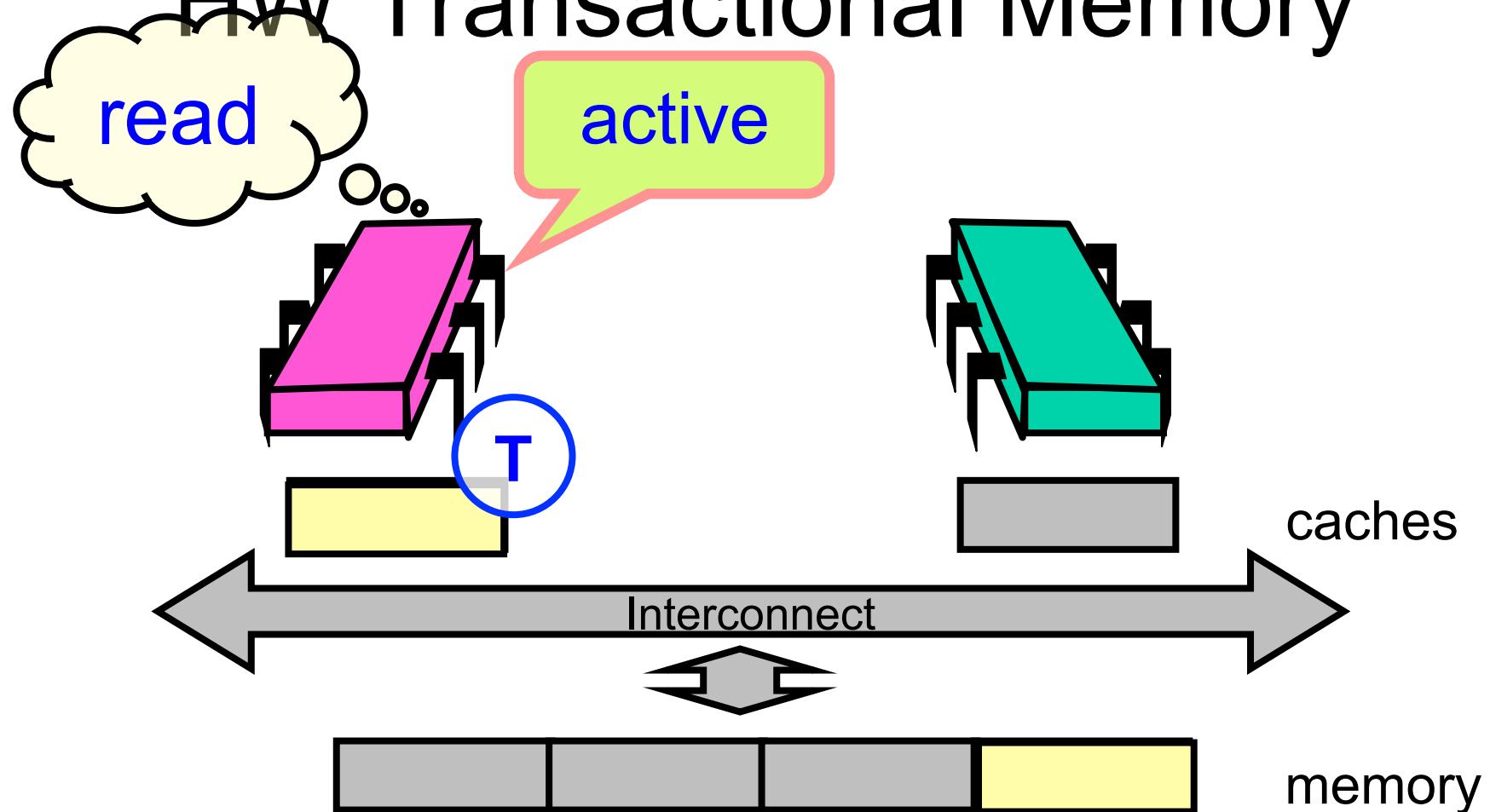


Invalidate

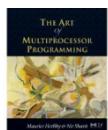
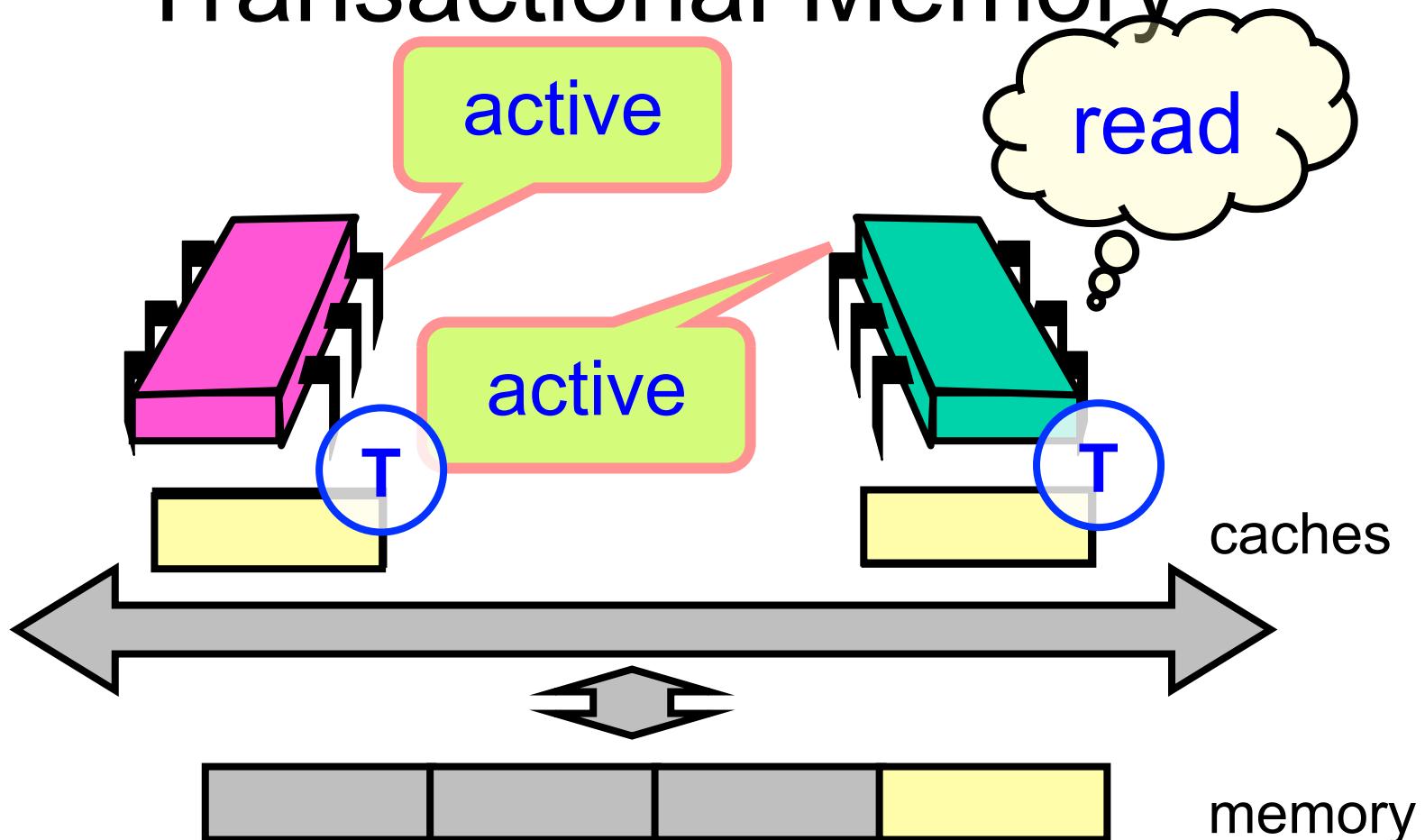
Memory provides data only if not present
in any cache, so no need to change it now
(expensive)



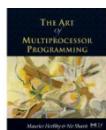
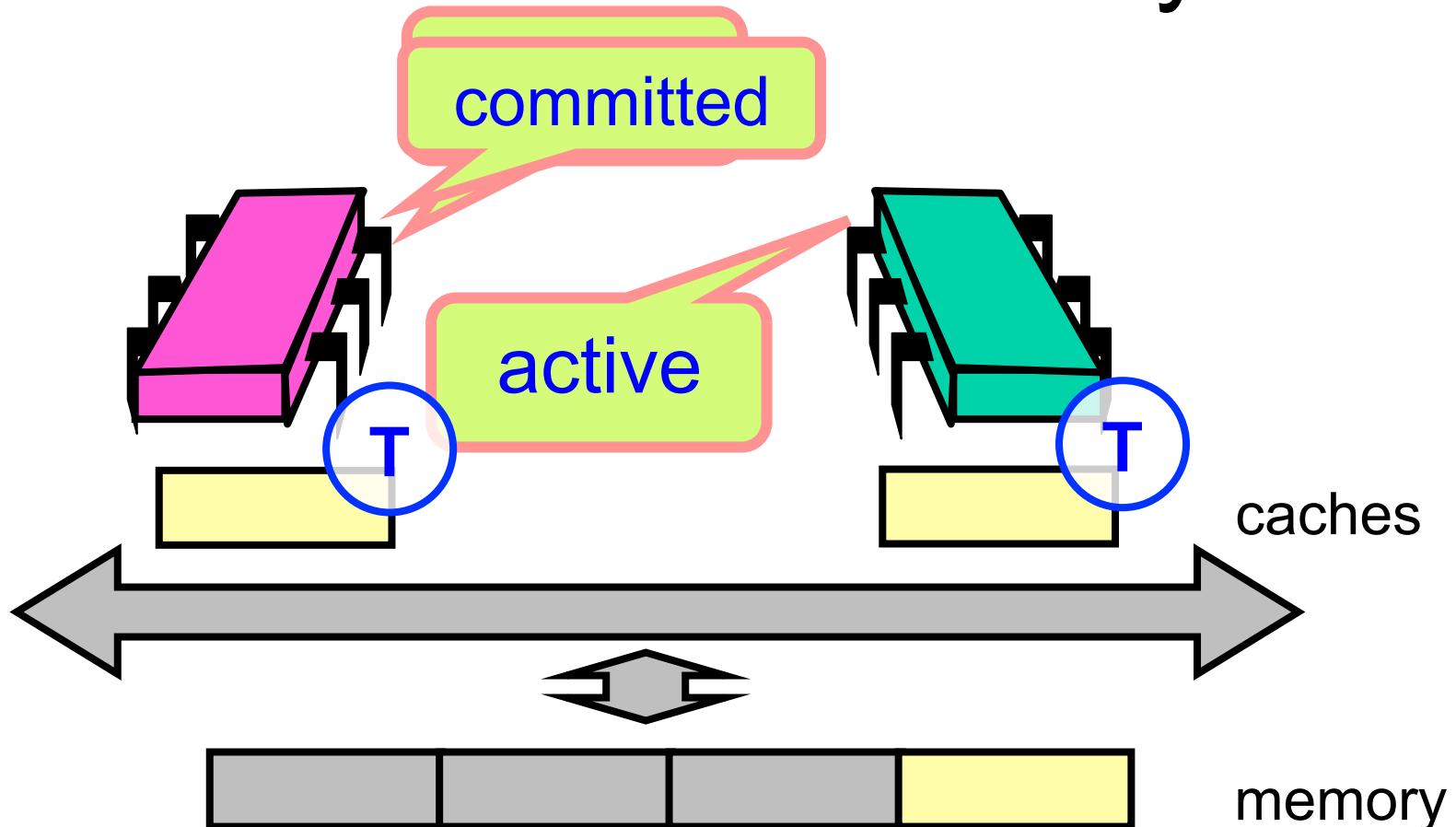
HW Transactional Memory



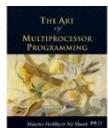
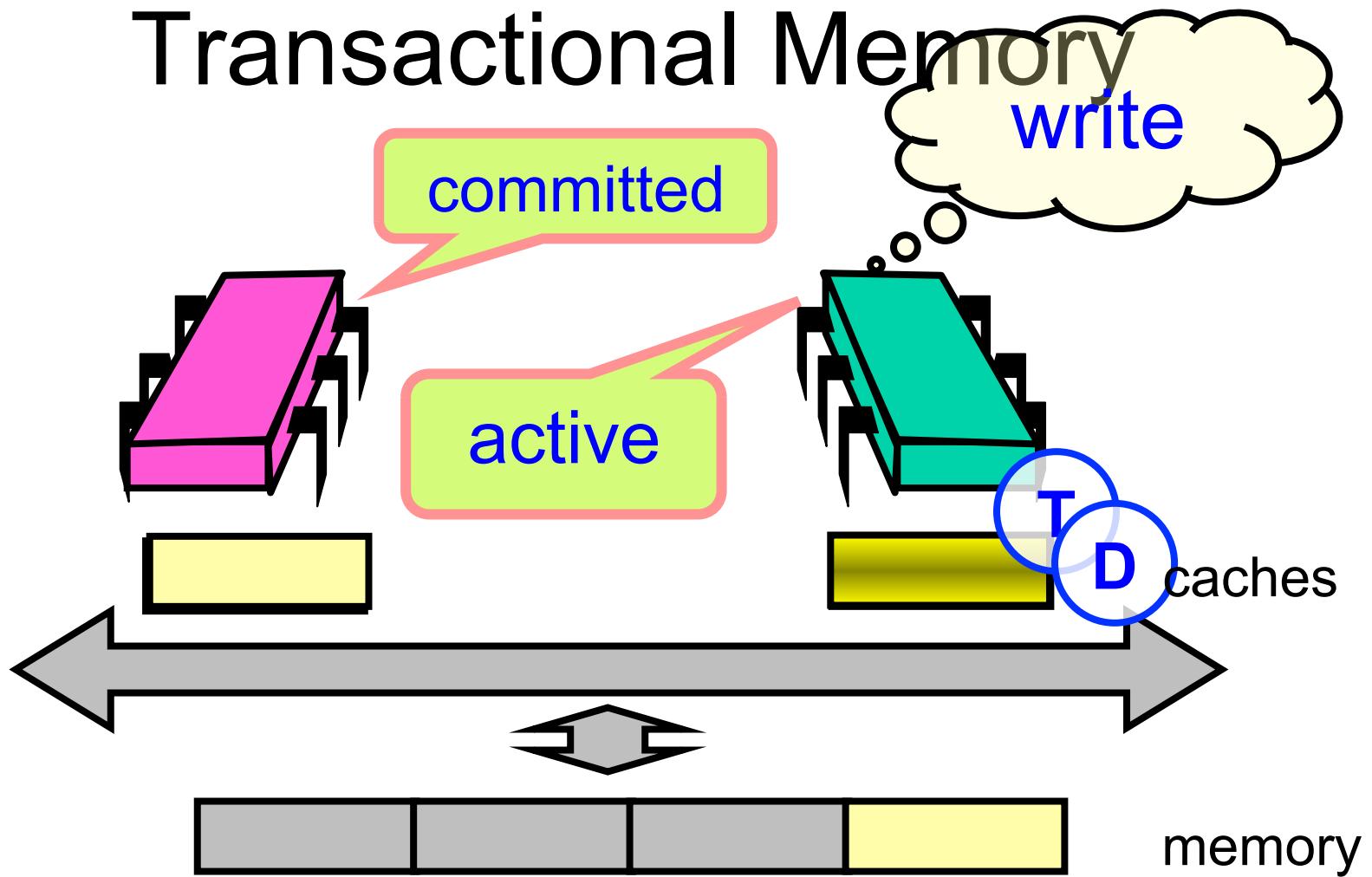
Transactional Memory

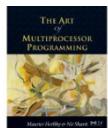
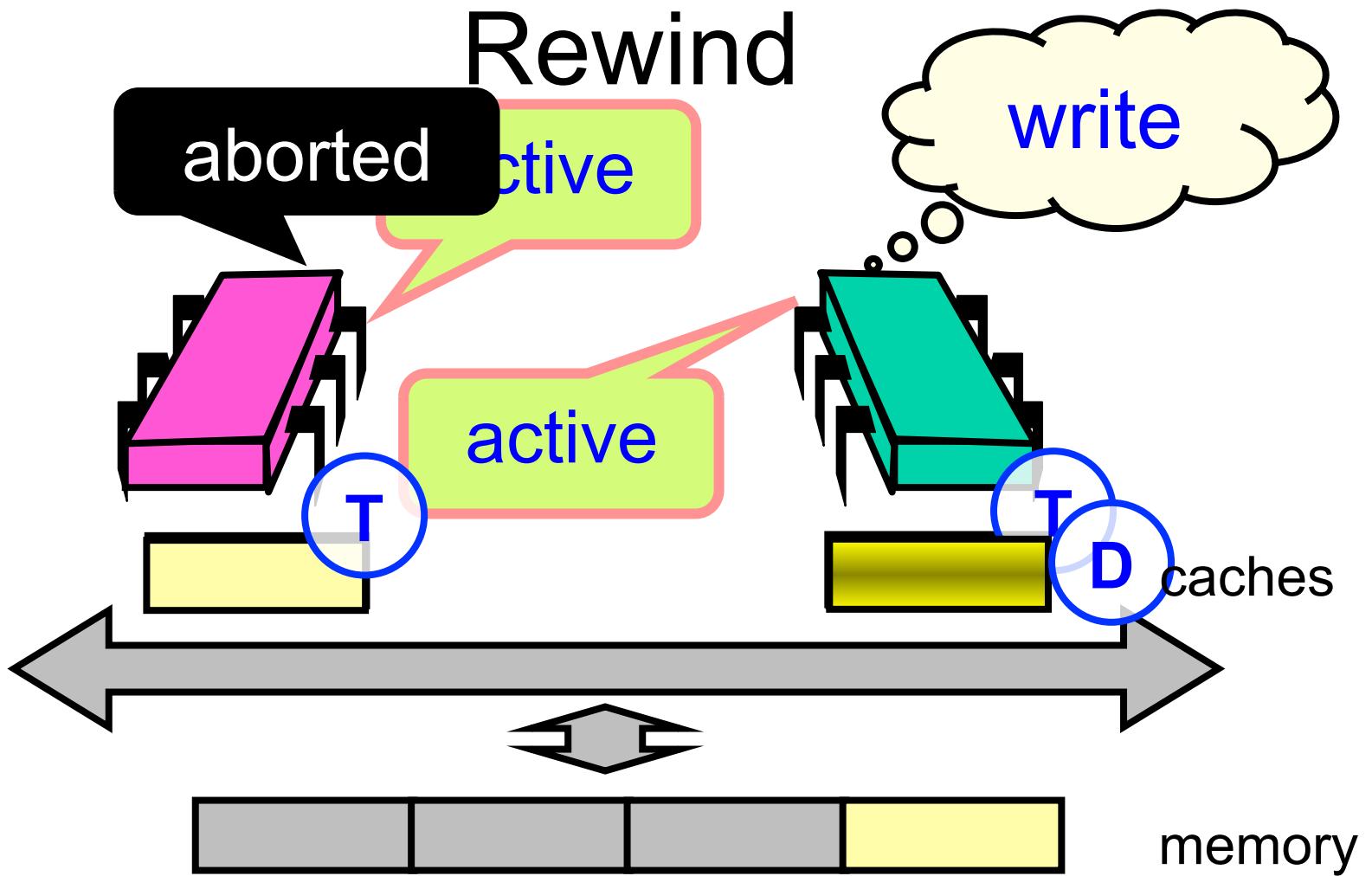


Transactional Memory



Transactional Memory





Transaction Commit

At Commit point ...

No cache conflicts? We win.

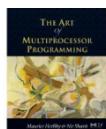
Mark transactional cache entries

Was: read-only, Now: valid

Was: modified, Now: dirty

(will be written back)

That's (almost) everything!



Road Map

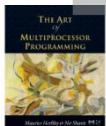
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

Software Transactional Memory

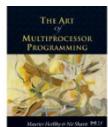
Research Questions



Hardware Transactional Memory (HTM)

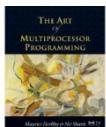
IBM's Blue Gene/Q & System Z & Power8

Intel's Haswell TSX extensions



Intel RTM

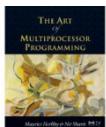
```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
    _xend()  
} else {  
    abort handler  
}
```



Intel RTM

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

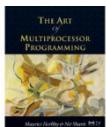
start a speculative transaction



Intel RTM

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

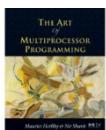
**If you see this, you are
inside a transaction**



Intel RTM

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

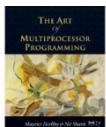
If you see anything else,
your transaction aborted



Intel RTM

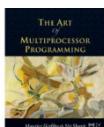
```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
    _xend()  
} else {  
    abort handler  
}
```

you could retry the
transaction, or take an
alternative path



Abort codes

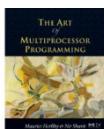
```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
} else if (status & _XABORT_EXPLICIT) {  
    aborted by user code  
} else if (status & _XABORT_CONFLICT) {  
    read-write conflict  
} else if (status & _XABORT_CAPACITY) {  
    cache overflow  
} else {  
    ...  
}
```



Abort codes

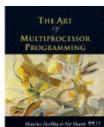
```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
} else if (status & _XABORT_EXPLICIT) {  
    aborted by user code  
} else if (status & _XABORT_CONFLICT) {  
    read-write conflict  
} else if (status & _XABORT_CAPACITY) {  
    cache overflow  
} else {  
    ...  
}
```

**speculative code can call
_xabort()**



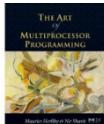
Abort codes

```
if synchronization conflict occurred (maybe retry)
    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 {
    ...
}
```



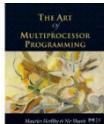
Abort codes

```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
} else if (status & _XABORT_READWRITE) {  
    read/write set too big  
    (maybe don't retry)  
} else if (status & _XABORT_READWRITE_CONFLICT) {  
    abort due to a read-write conflict  
} else if (status & _XABORT_CAPACITY) {  
    read-write conflict  
    cache overflow  
} else {  
    ...  
}
```



Abort codes

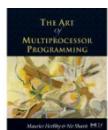
```
if (_xbegin() == _XBEGIN_STARTED) {  
    speculative code  
} else if (status & _XABORT_EXPLICIT) {  
    aborted by user code  
} else if (status & _XABORT_CONFLICT) {  
    other abort codes ...  
    read write conflict  
} else if (status & _XABORT_CAPACITY) {  
    cache overflow  
} else {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



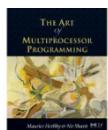
Too Big



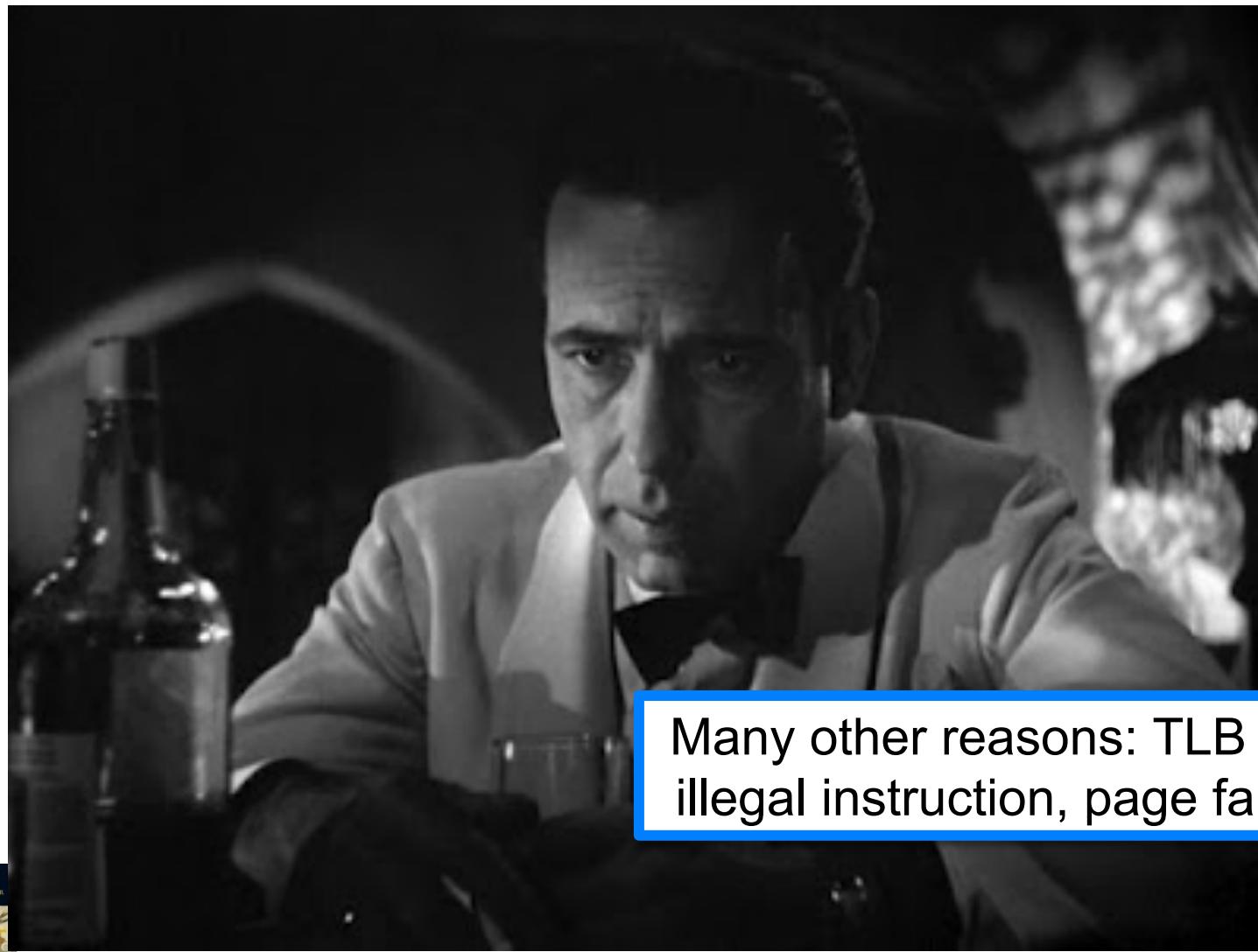
Transaction aborts if data set overflows caches, internal buffers



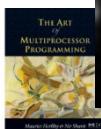
Too Slow



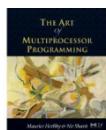
Just Not in the Mood



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

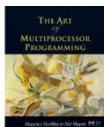


Hybrid Transactional Memory



Non-Speculative Fallback

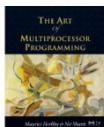
```
if (_xbegin() == _XBEGIN_STARTED) {
    read lock state
    if (lock taken) _xabort();
    work;
    _xend()
} else {
    lock->lock();
    work;
    lock->unlock();
}
```



Non-Speculative Fallback

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

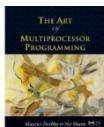
reading lock ensures that transaction will abort if another thread acquires lock



Non-Speculative Fallback

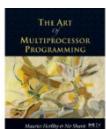
```
if (_xbegin() == _XBEGIN_STARTED) {  
    read lock state  
    if (lock taken) _xabort();  
    work;  
    _xend()  
} else {  
    lock state  
    work;  
    lock state  
}  
}
```

abort if another thread has acquired lock



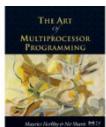
on abort, acquire lock & do work
(aborting concurrent speculative transactions)

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



Lock Elision

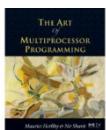
```
<HLE acquire prefix> lock();  
do work;  
<HLE release prefix> unlock()
```



Lock Elision

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

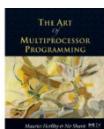
**first time around,
read lock and
execute speculatively**



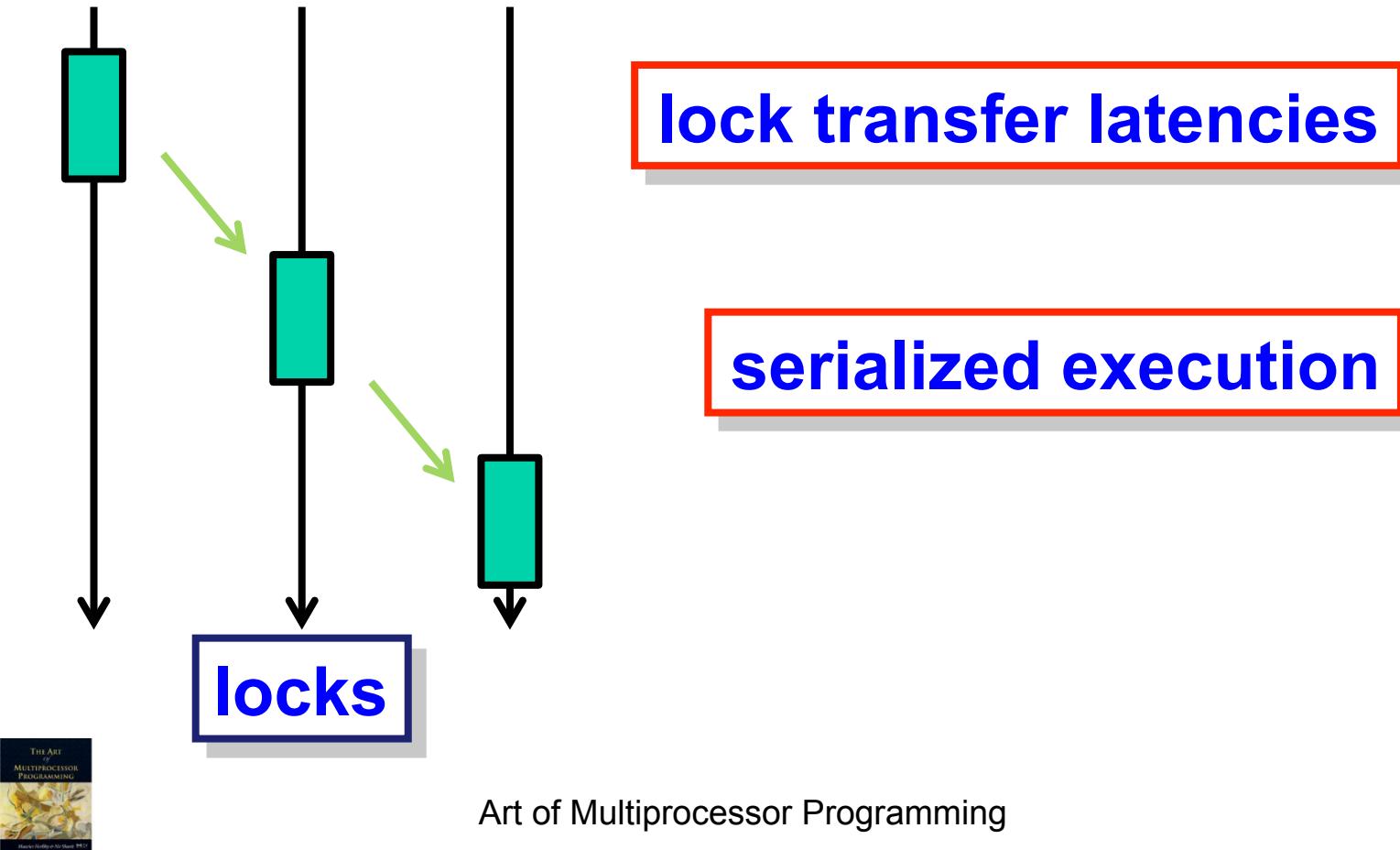
Lock Elision

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

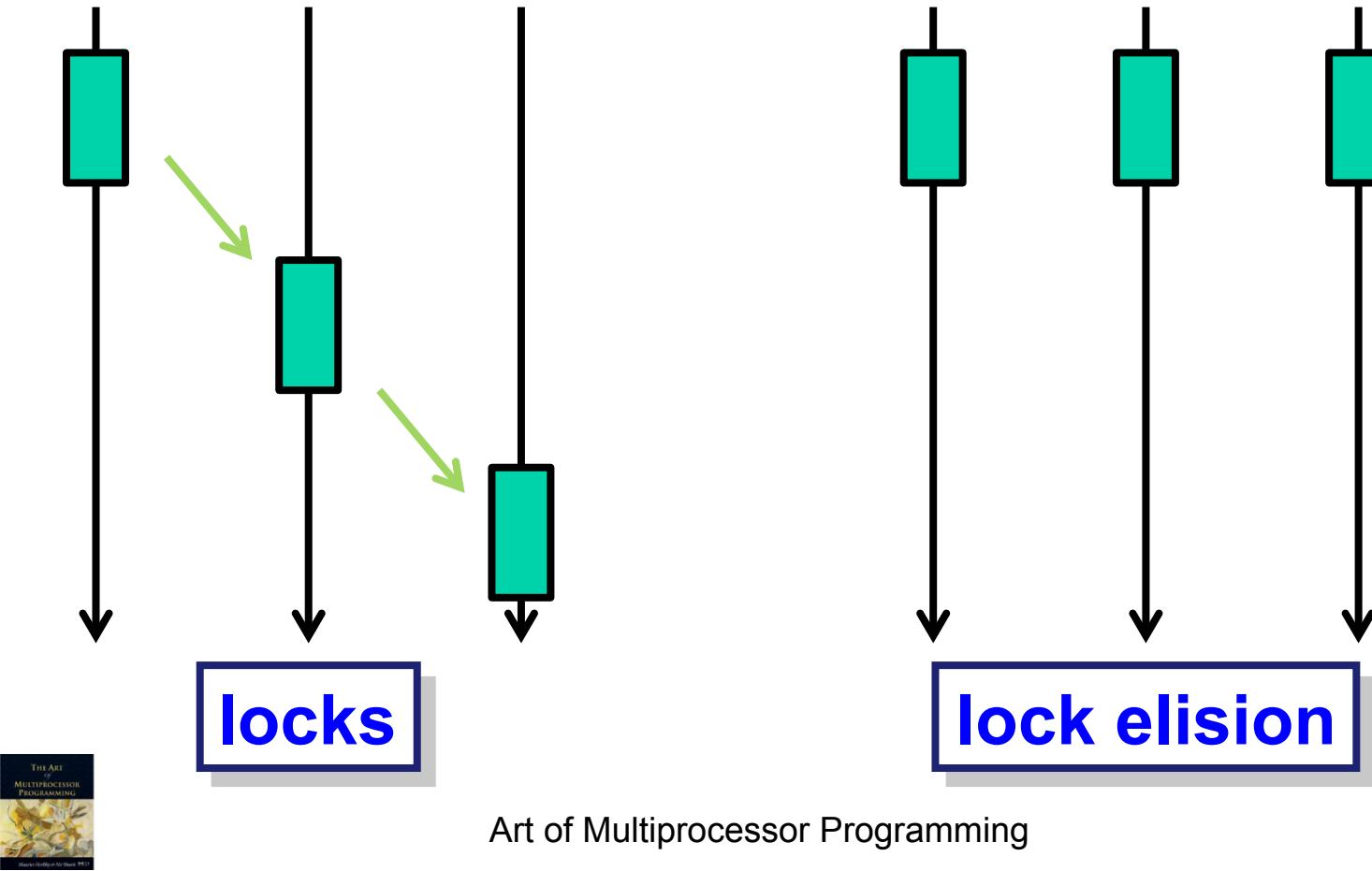
**if speculation fails,
no more Mr. Nice Guy,
acquire the lock**



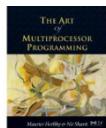
Conventional Locks



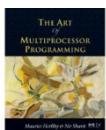
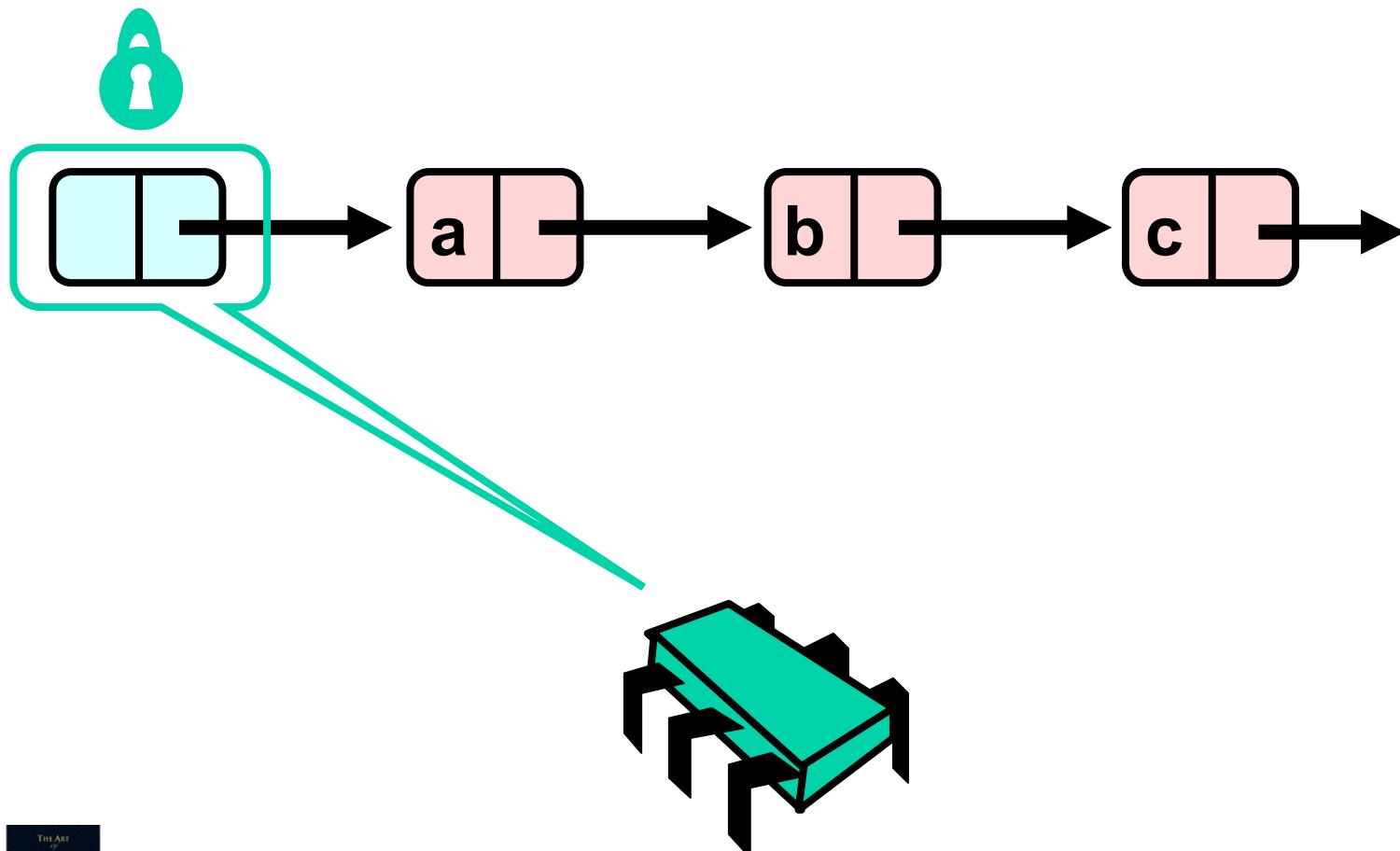
Lock Elision



Lock Teleportation



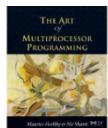
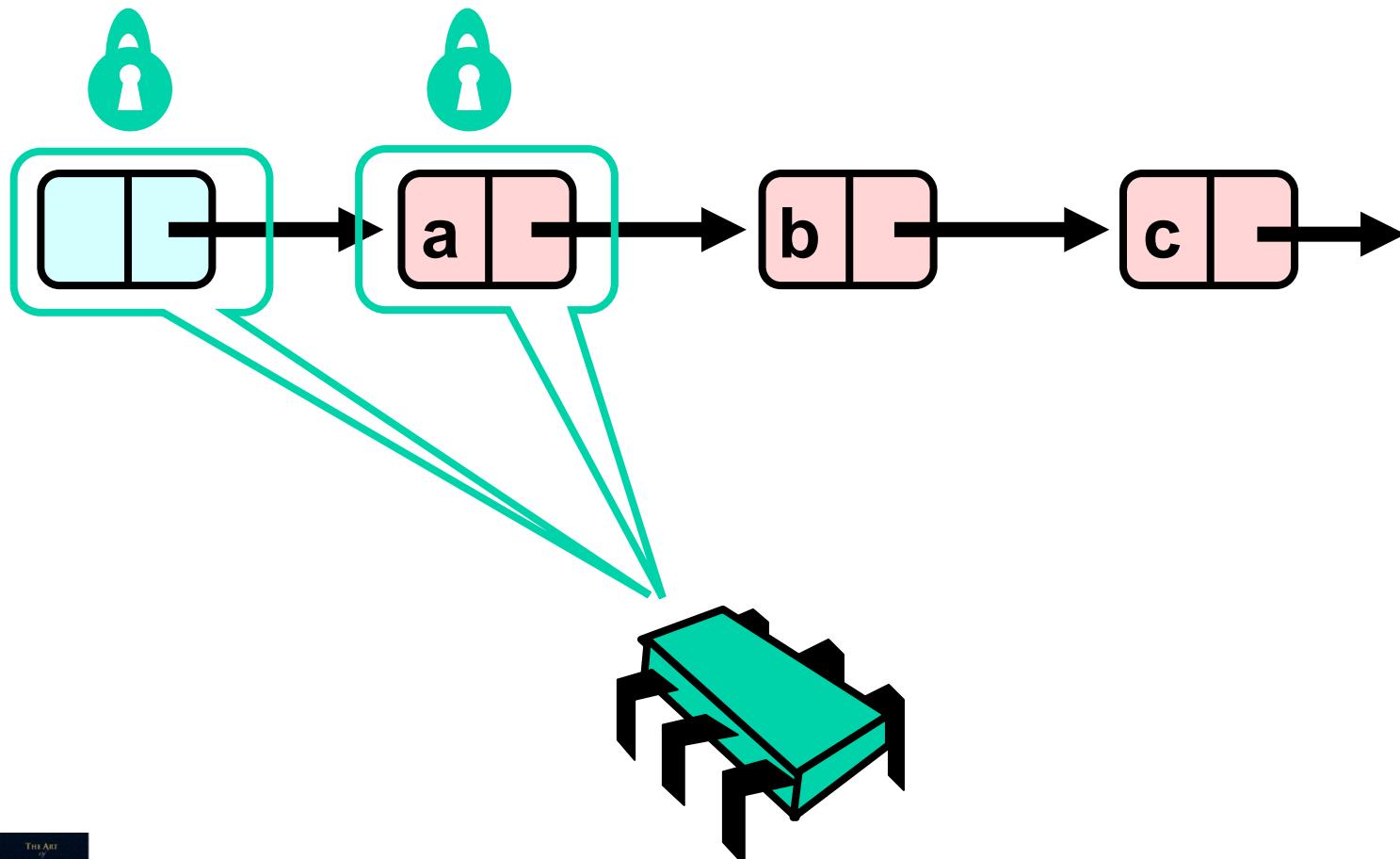
Hand-over-Hand locking



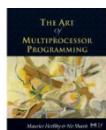
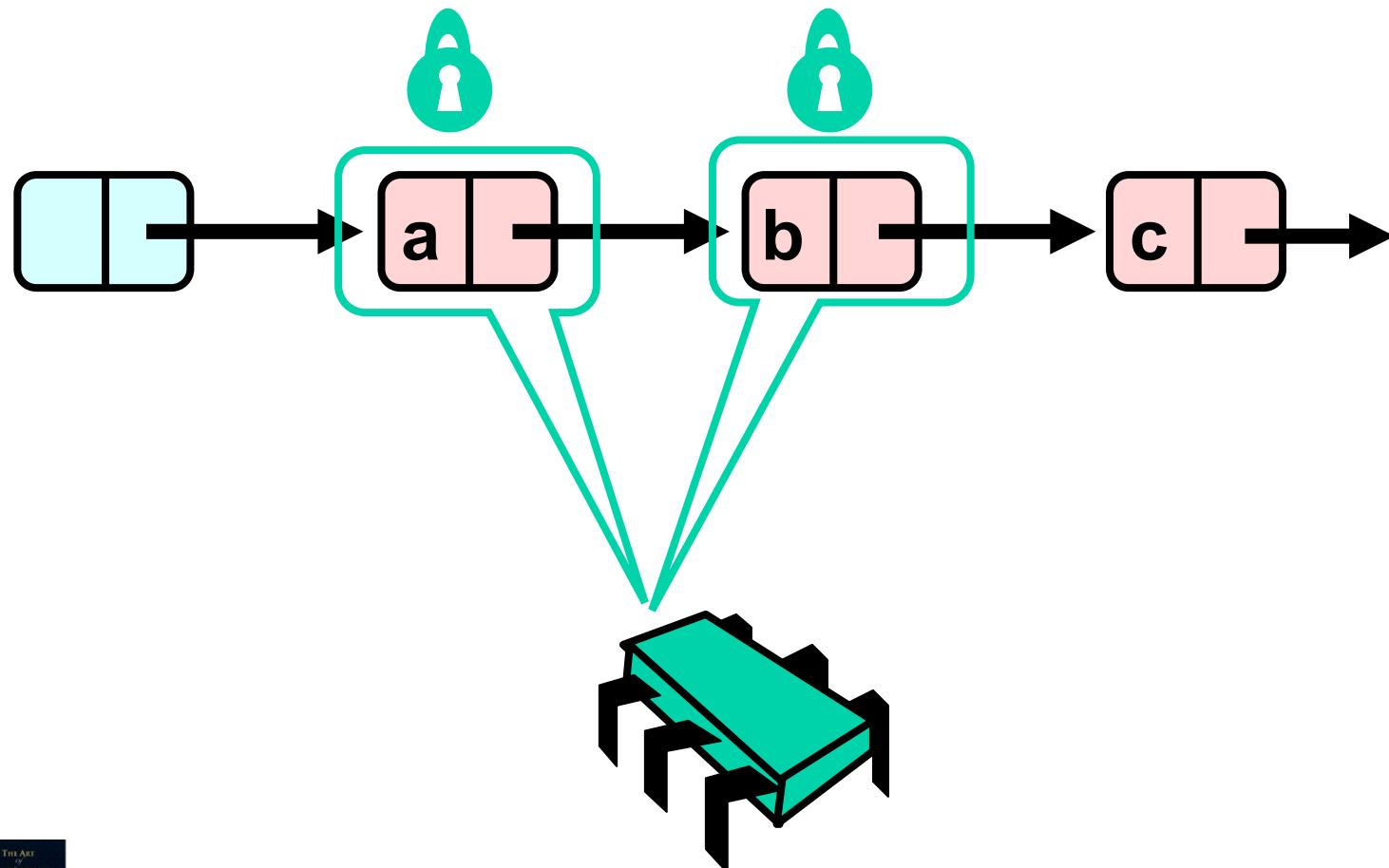
Art of Multiprocessor Programming

Art of Multiprocessor
Programming 97

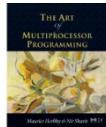
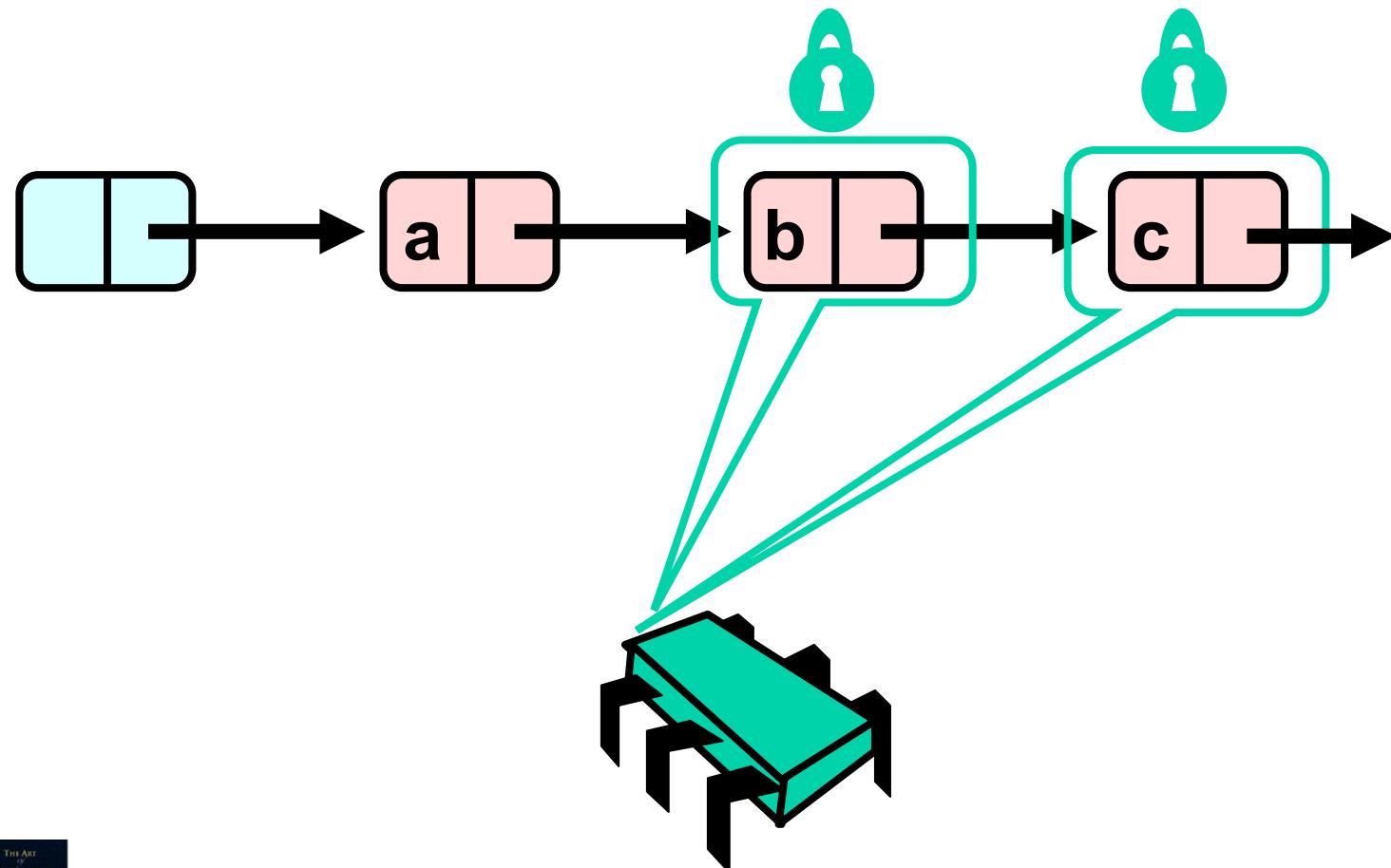
Hand-over-Hand locking



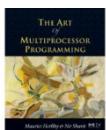
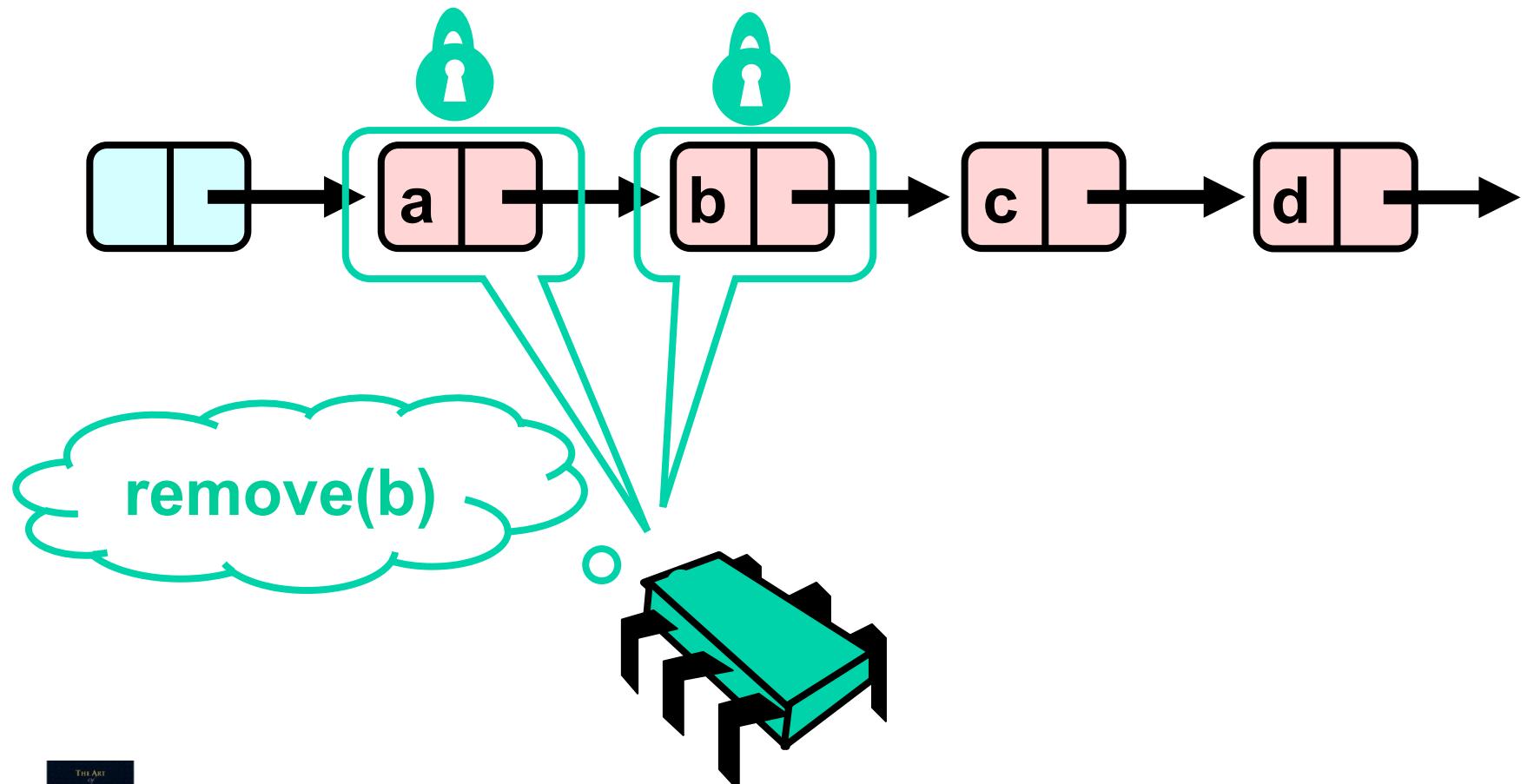
Hand-over-Hand locking



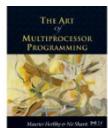
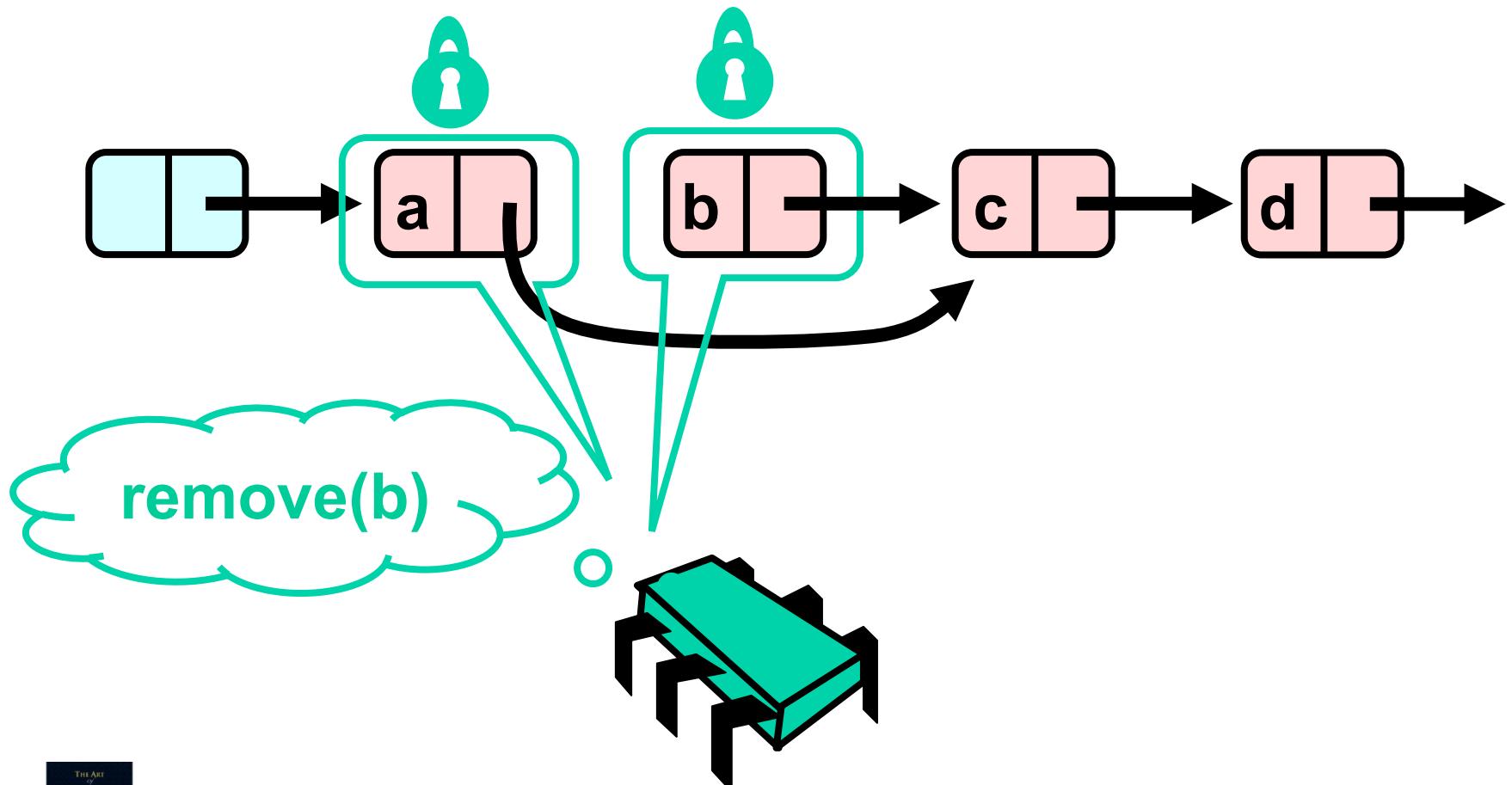
Hand-over-Hand locking



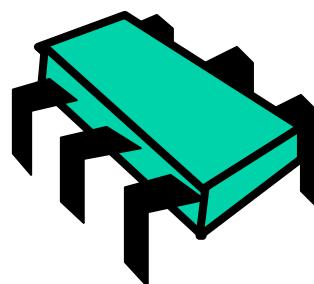
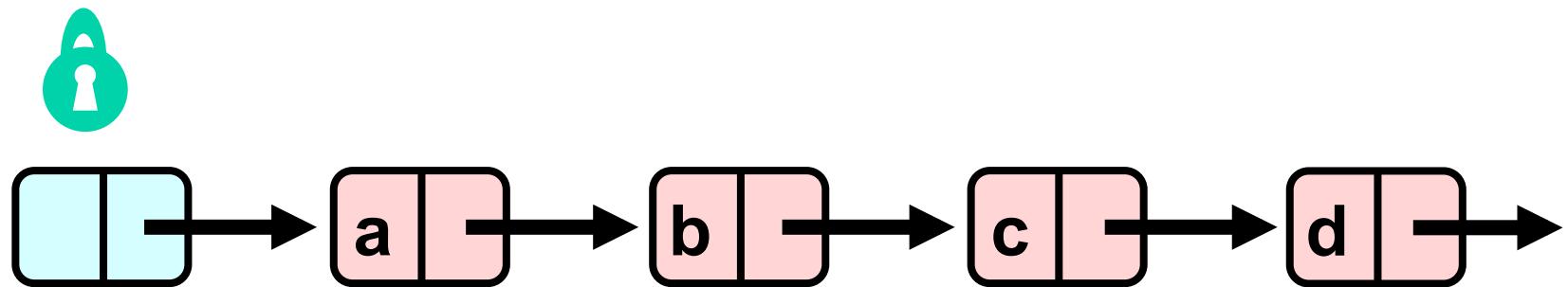
Removing a Node



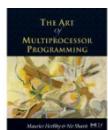
Removing a Node



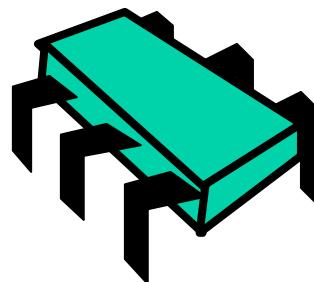
Lock Teleportation



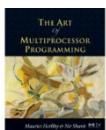
Art of Multiprocessor Programming



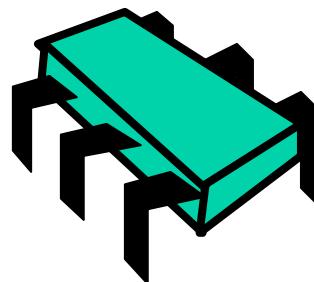
Lock Teleportation



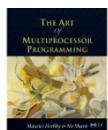
Art of Multiprocessor Programming



Lock Teleportation

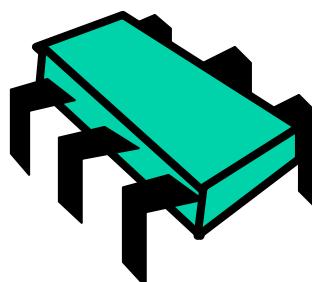
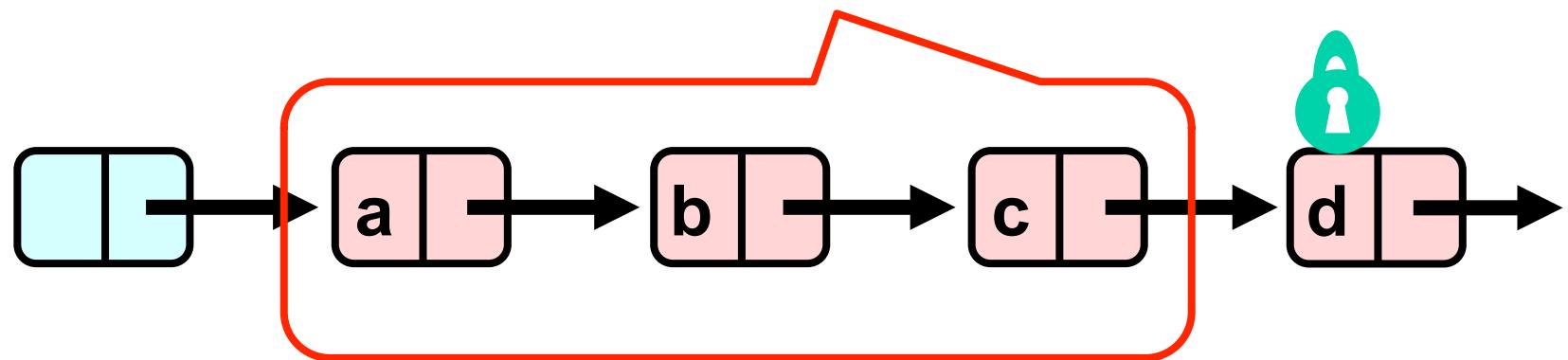


Art of Multiprocessor Programming

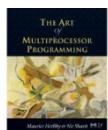


Lock Teleportation

no locks acquired



Art of Multiprocessor Programming



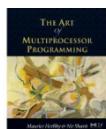
How Far to Teleport?

Too short?

Missed opportunity

Too far?

Transaction aborts, work lost



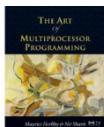
Adaptive Teleportation

On Success:

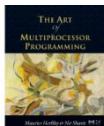
```
limit = limit + 1
```

On Failure:

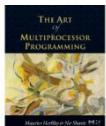
```
limit = limit / 2
```



```
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
        } } };
```



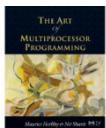
```
Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        if (locked_node_in_sorted_list(v)) {
            if (v->state == BEGIN_STARTED) {
                teleportLimit nodes
                move lock
                _xend();
                teleportLimit++;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



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

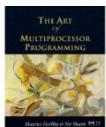
**locked node
In sorted list**

**Value to
search for**



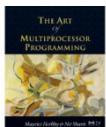
```
Node* teleport(Node* start, T v) {  
    int retries = RETRY_THRESHOLD;  
    while (--retries) {  
        int  
        if  
            +  
        move lock  
            _xend();  
            teleportLimit++;  
            return pred;  
    } else {  
        teleportLimit = teleportLimit/2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

Returns locked node with
value less than or equal to v



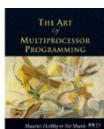
```
Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            ...
            teleportLimit++;
        }
        return pred;
    } else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

Try for a fixed number of times



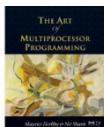
```
Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            teleportLimit++;
            return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

Executed as read-only transaction



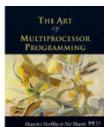
```
Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            telep
            return
       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

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



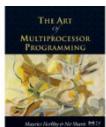
```
Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            telep
            return
      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

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



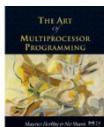
```
Node* teleport(Node* start, T v) {  
    int retries = RETRY_THRESHOLD;  
    while (--retries) {  
        int distance = 0;  
        if (xbegin() == _XBEGIN_STARTED) {  
            traverse up to teleportLimit nodes  
            move lock  
            _xend();  
            _teleportLimit++;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

**Unlock starting node, lock
final node**



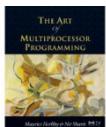
```
Node* teleport(Node* start, T v) {
    int retries = RETRY_THRESHOLD;
    while (--retries) {
        int distance = 0;
        if (xbegin() == _XBEGIN_STARTED) {
            traverse up to teleportLimit nodes
            move lock
            _xend();
            teleportLimit++;
            return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

Try to commit transaction



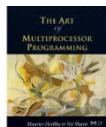
```
Node* teleport(Node* start, T v) {
    int retries;
    while (--retries > 0) {
        int distance;
        if (xbegin() == 0) {
            traverse list up to threshold
            move lock
            xend();
            teleportLimit++;
            return last node;
        } else {
            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

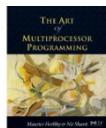


Lock-Based STMs

STMs come in different forms:

Lock-Free

Lock-based

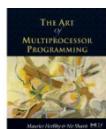


Lock-Based STM

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

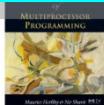
Each transaction keeps

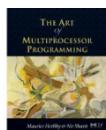
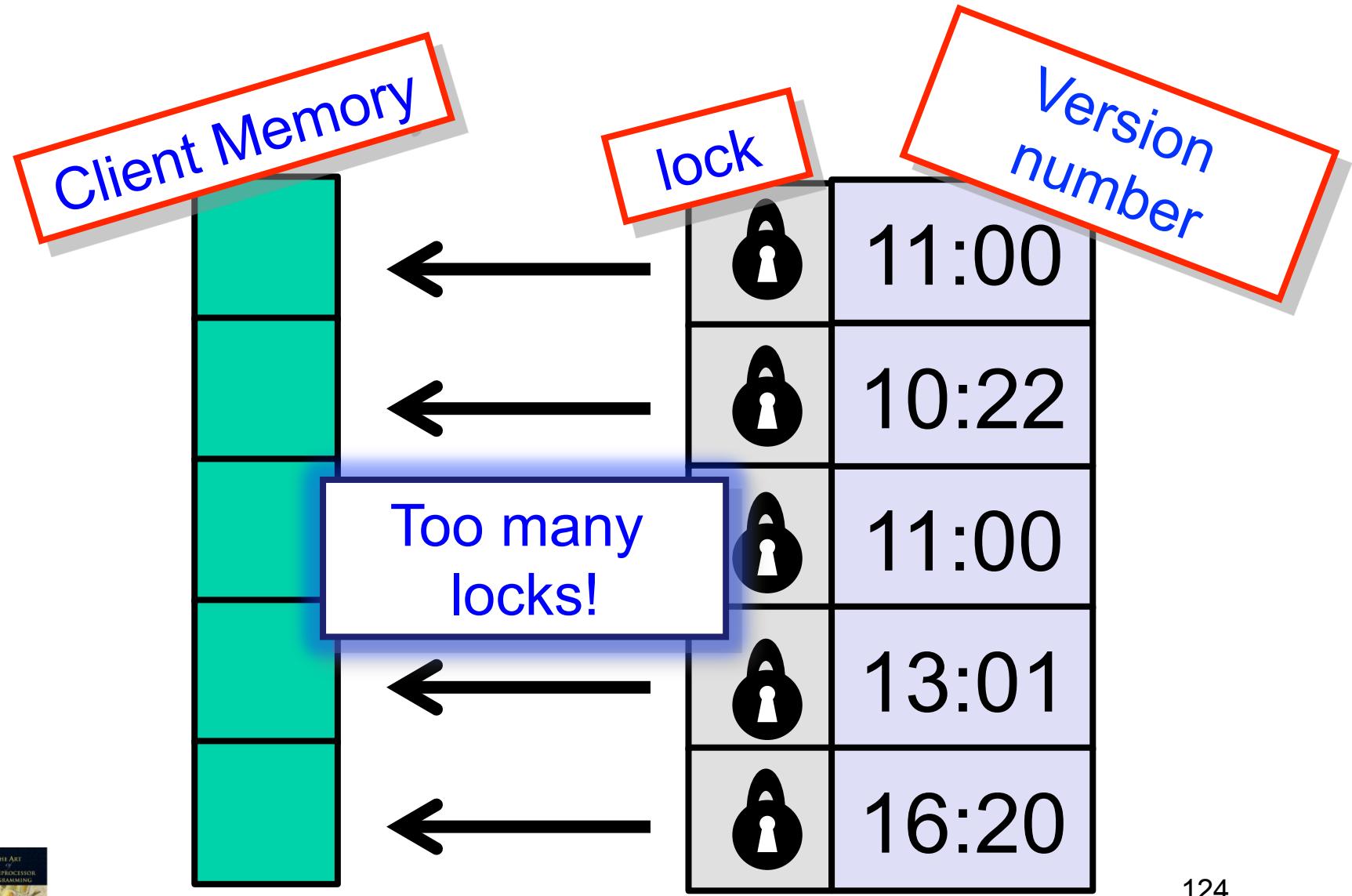
Read Set: locations and values read

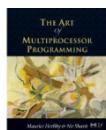
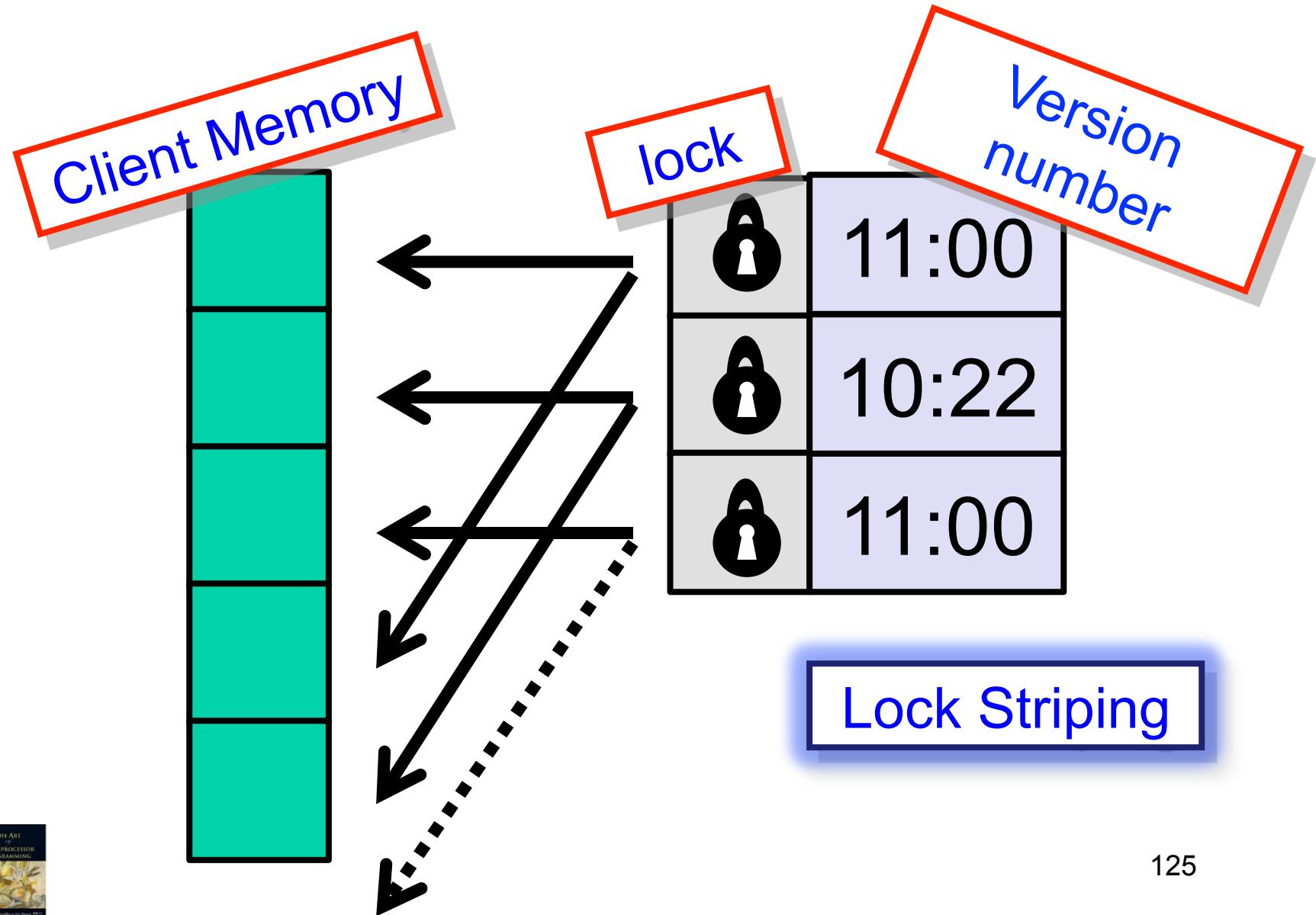
Write Set: locations and values written

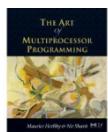
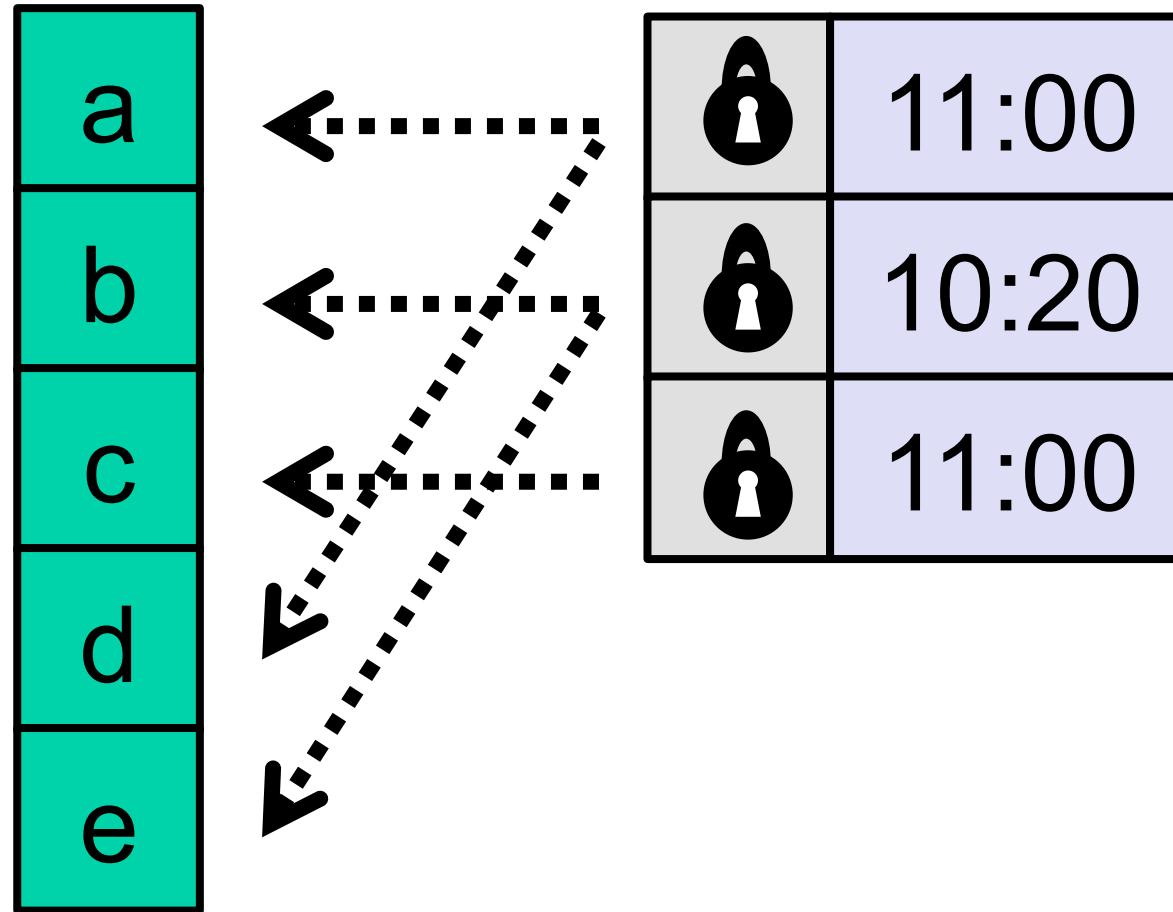
Changes installed at commit

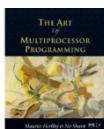
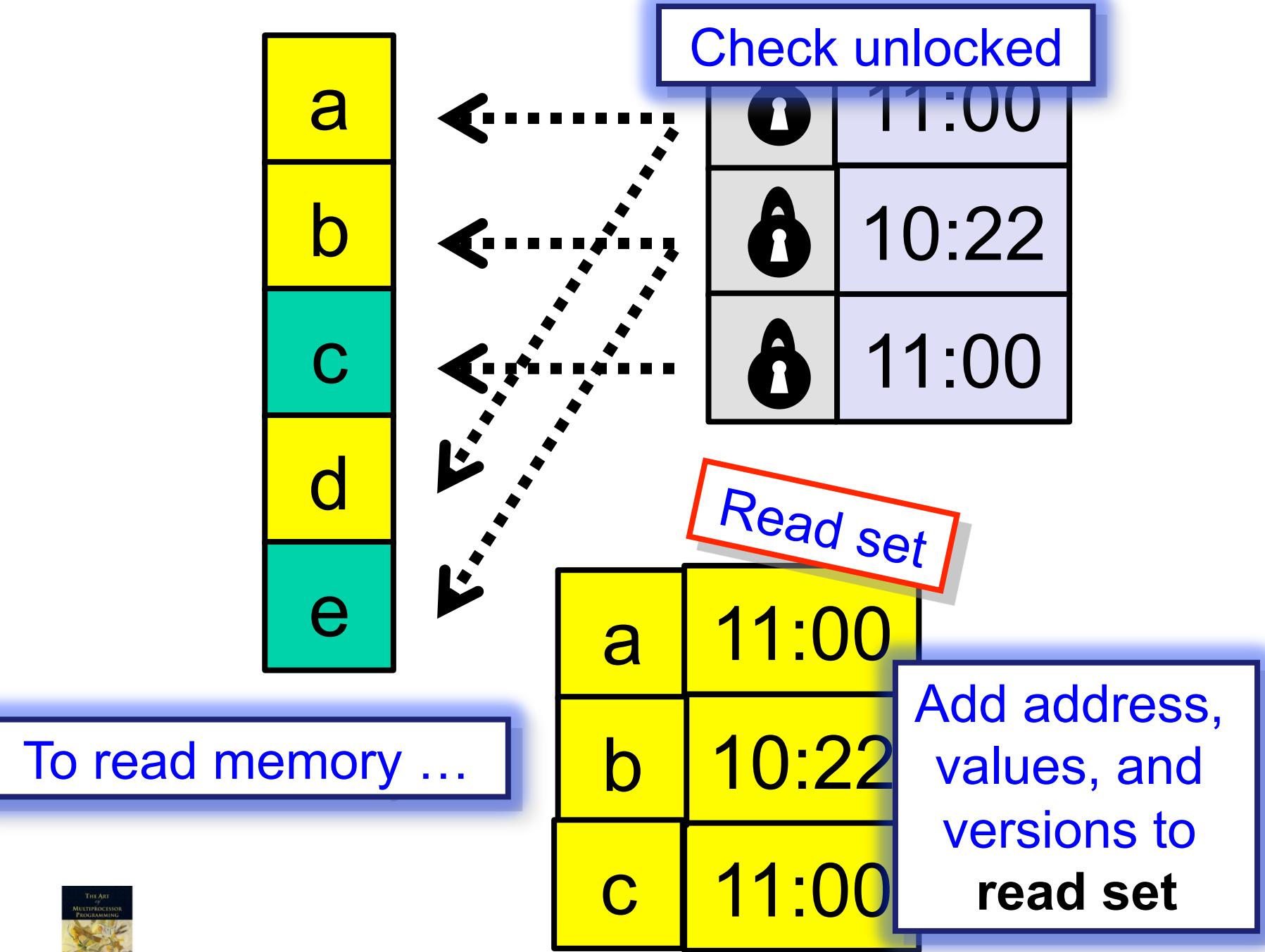
Conflicts detected at comit

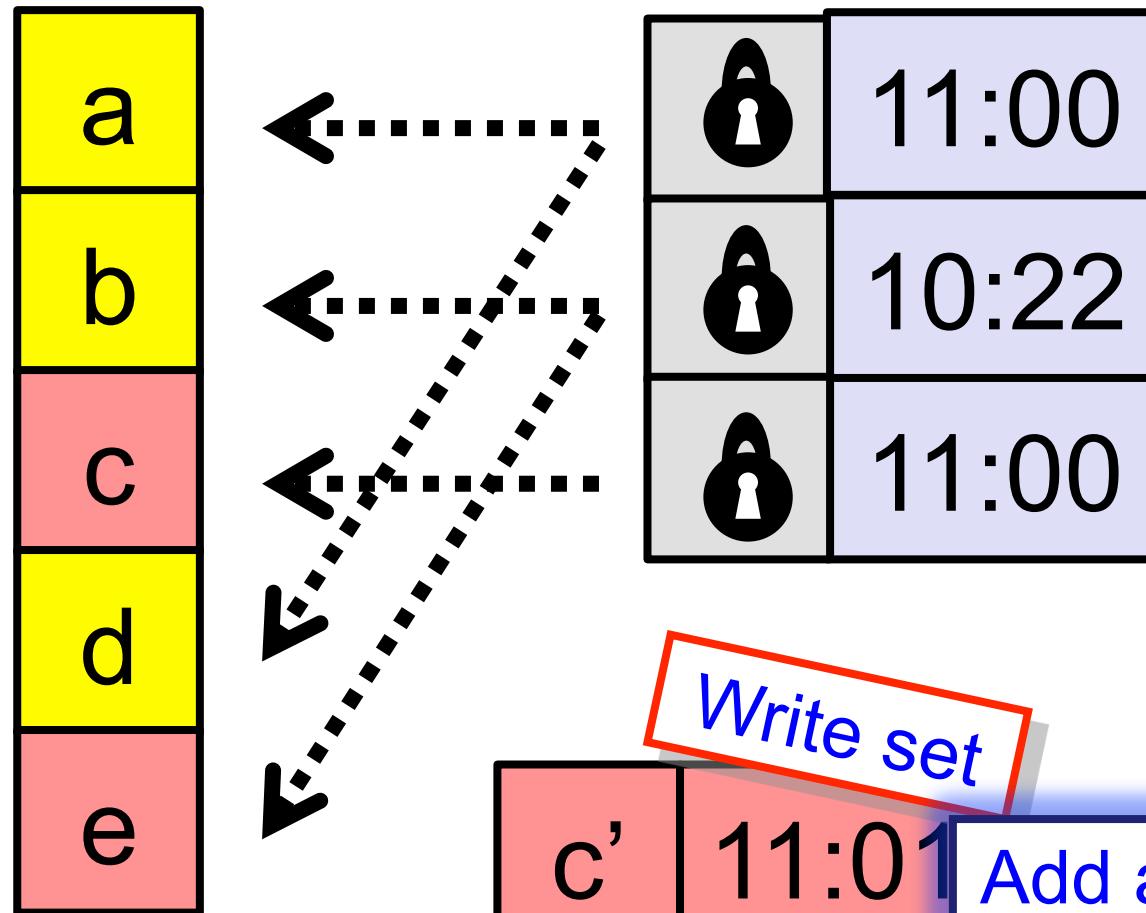






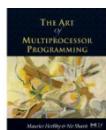


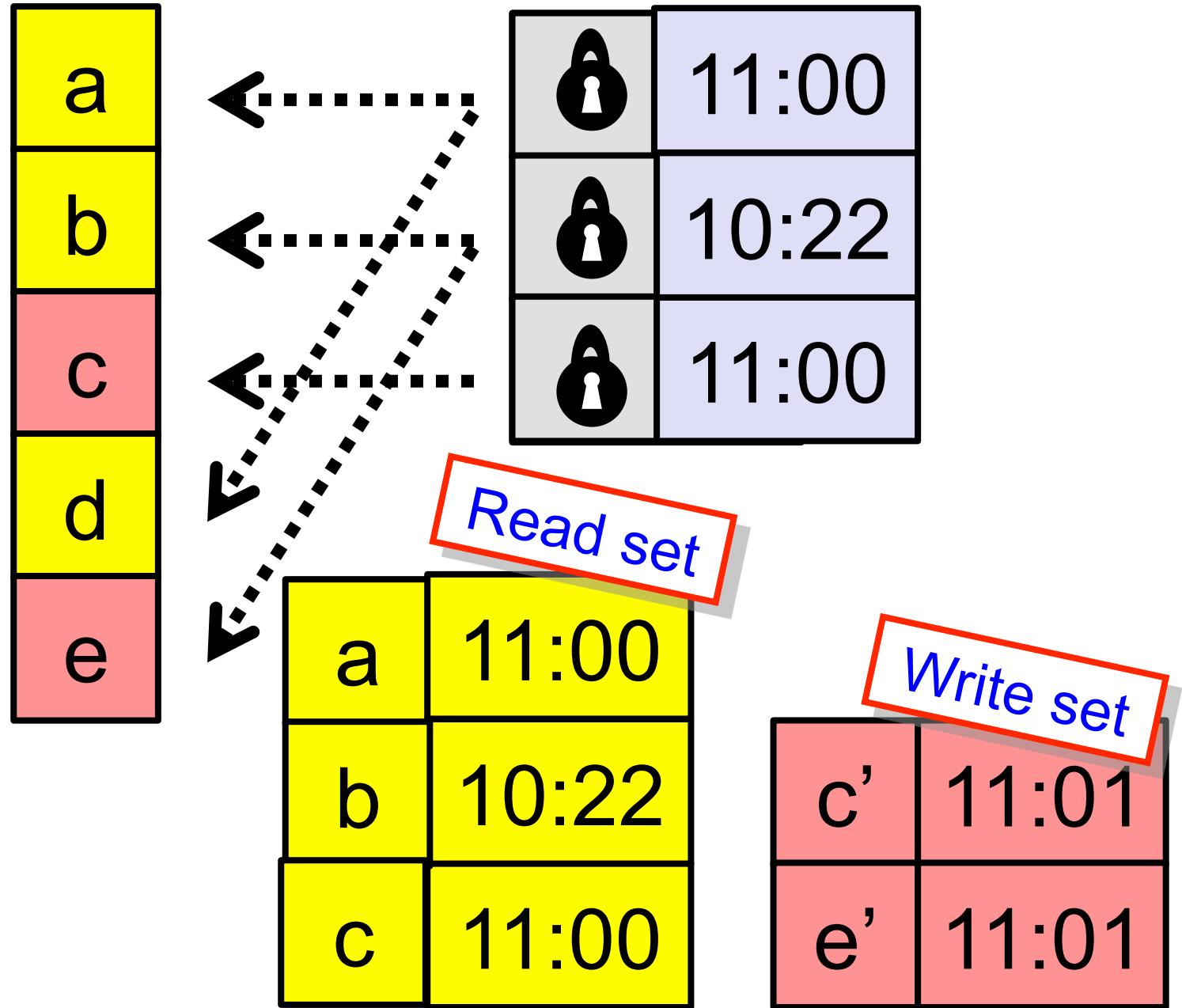
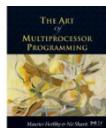


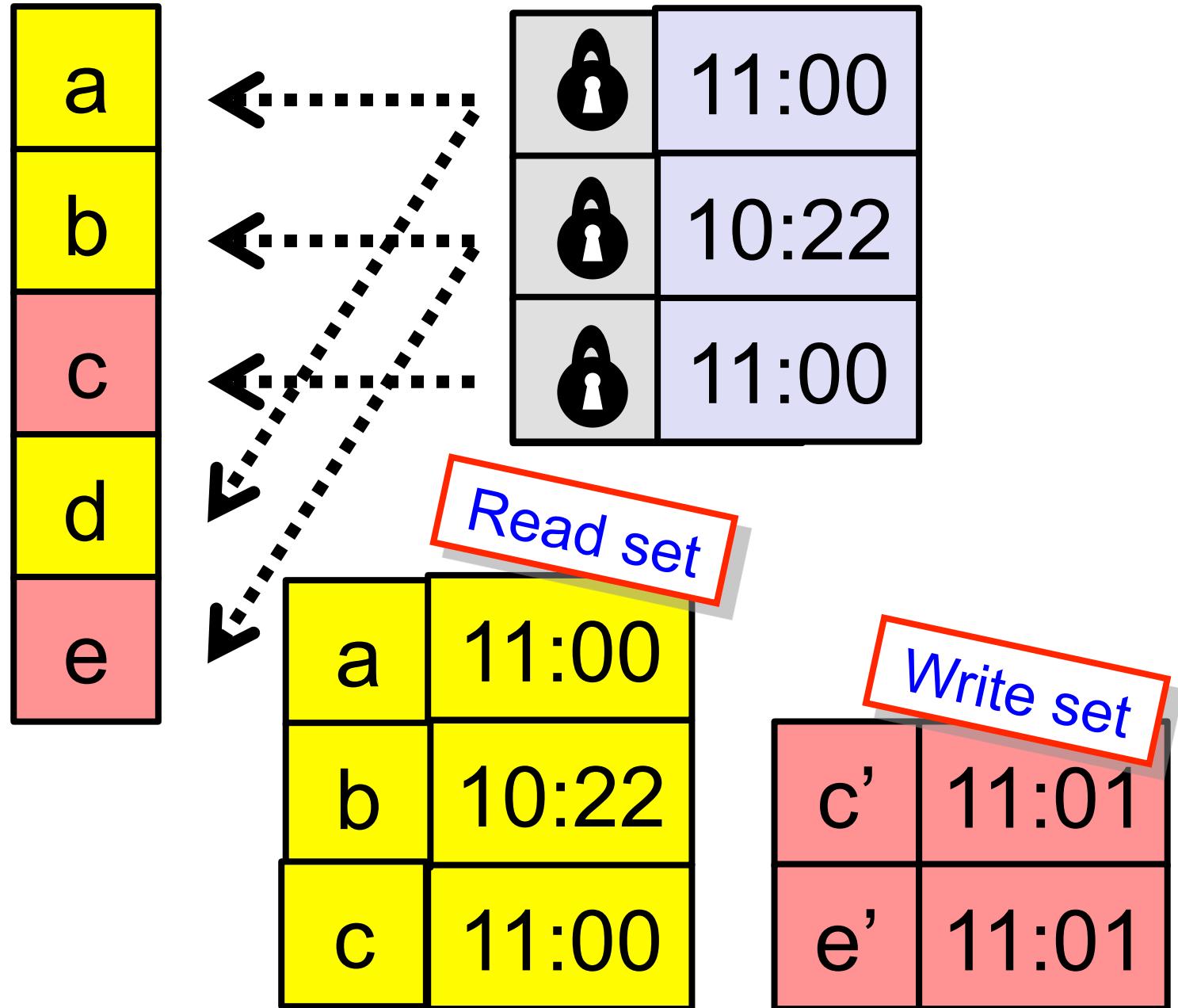
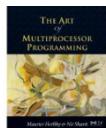


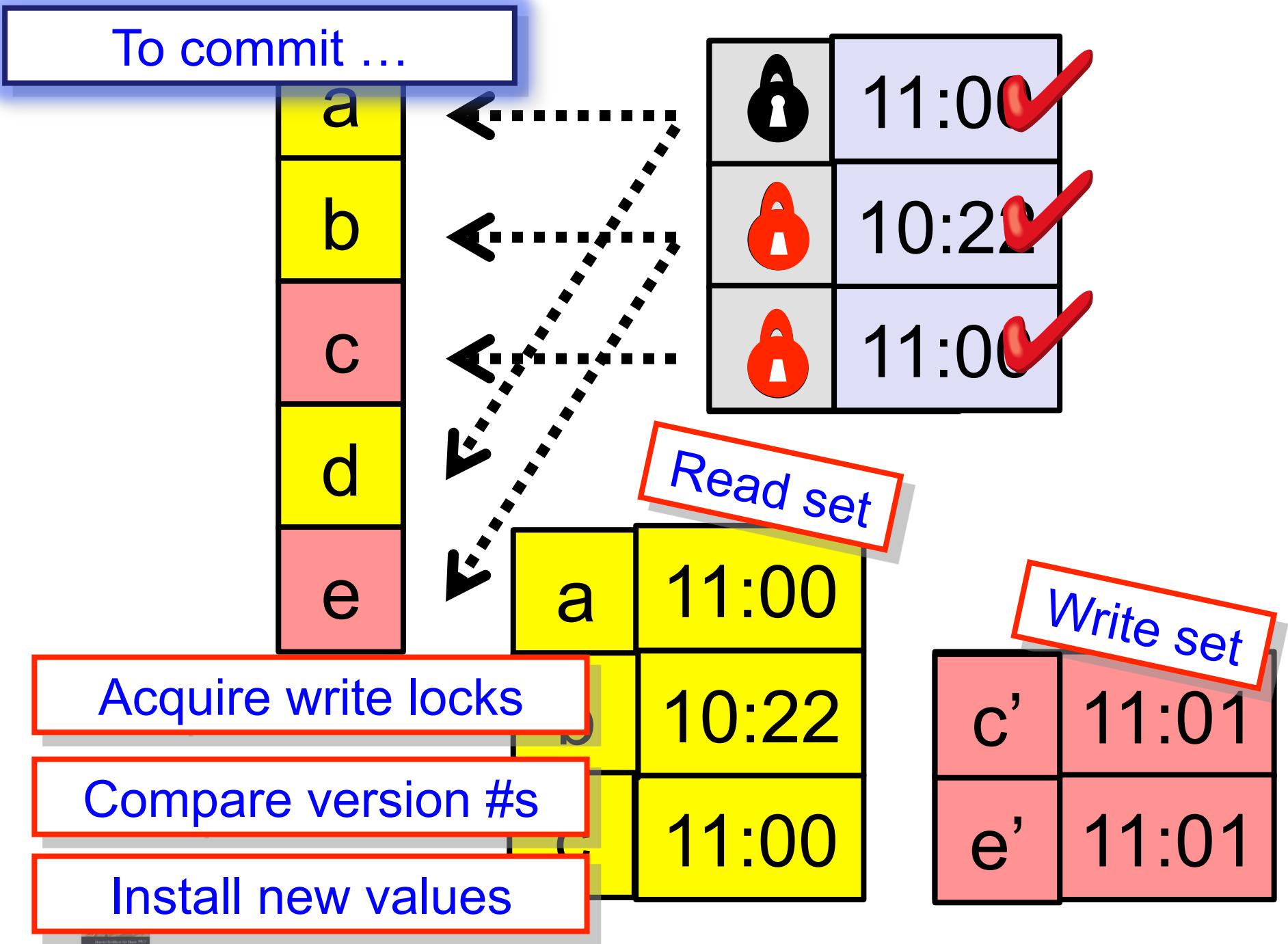
To write memory ...

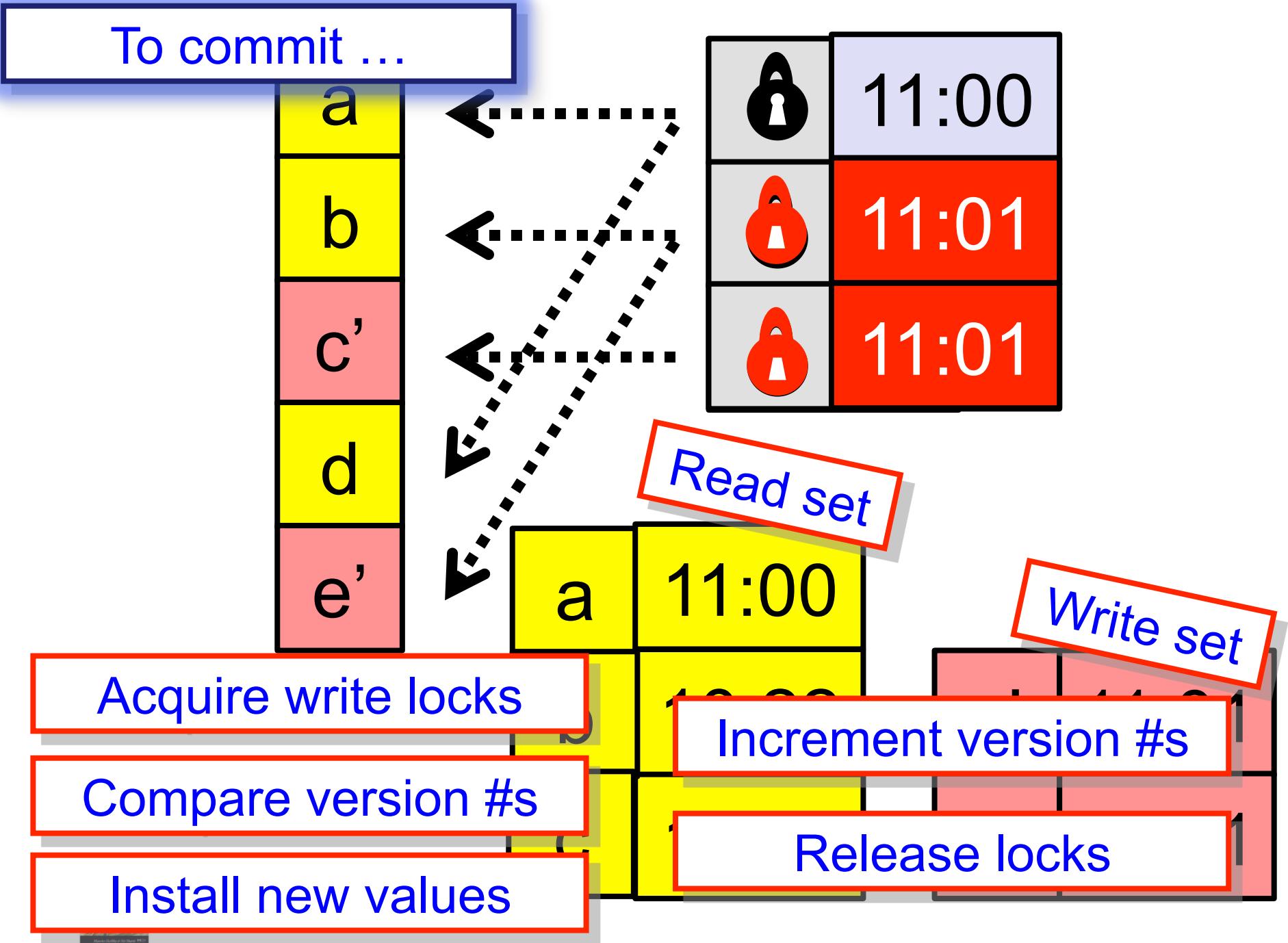
Add address,
new values
and versions
to write set











Zombie Transactions

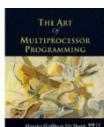


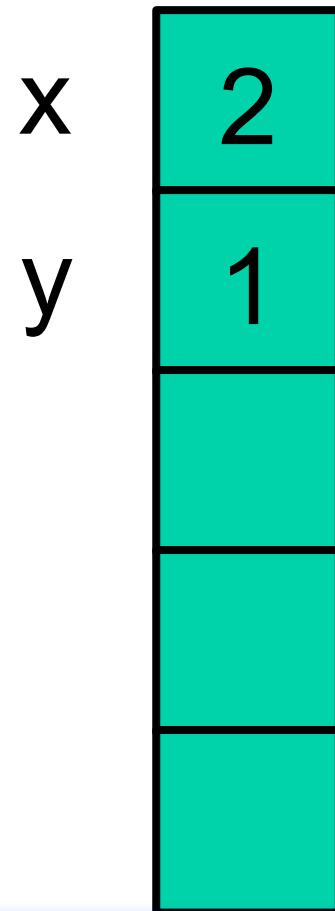
A **zombie human** is dead but act like it is alive ...

FEATURING:

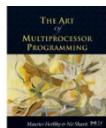
A **zombie transaction** is one that will certainly abort, but continues to run ...

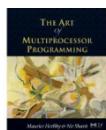
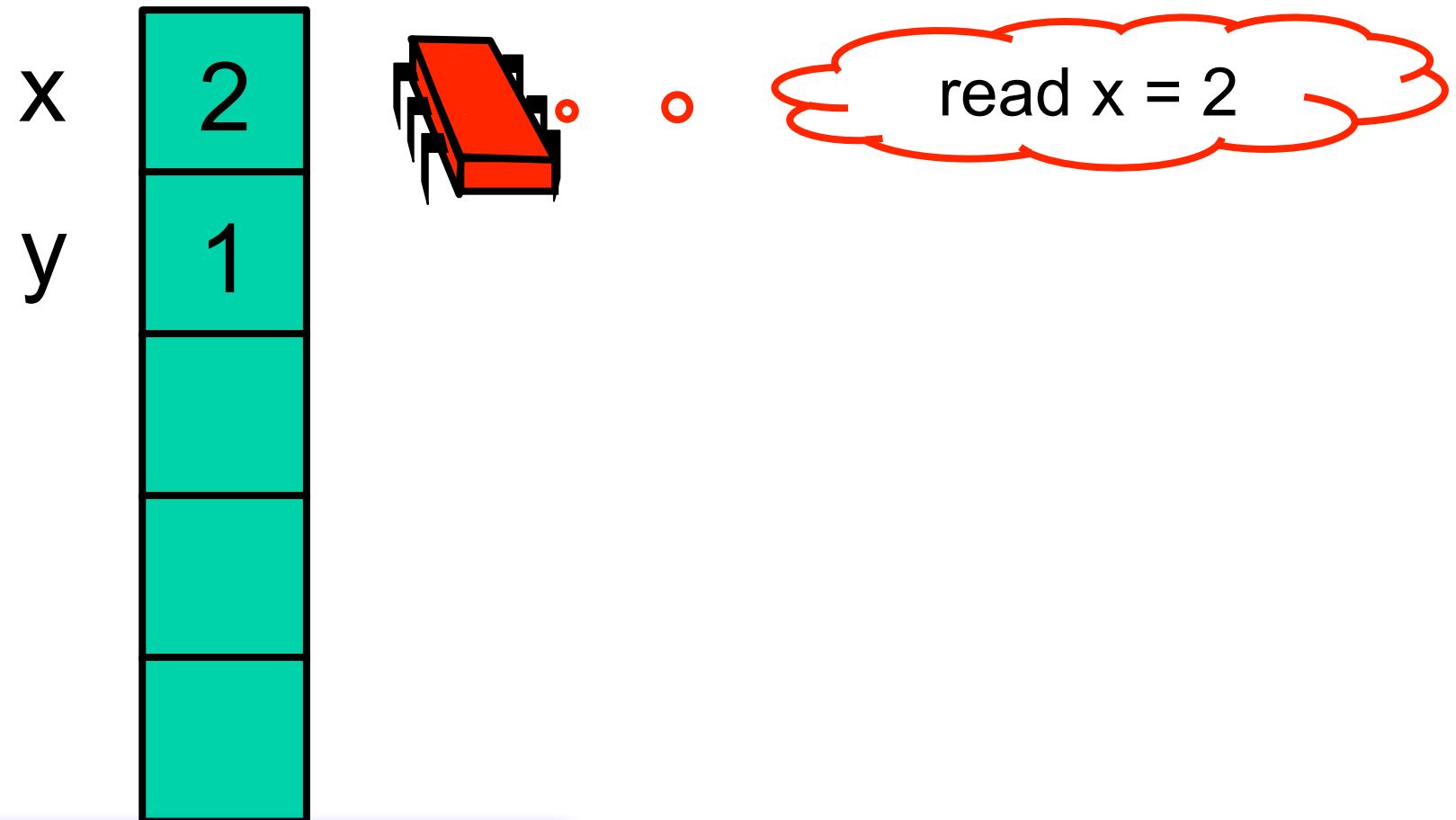
Why do we care?

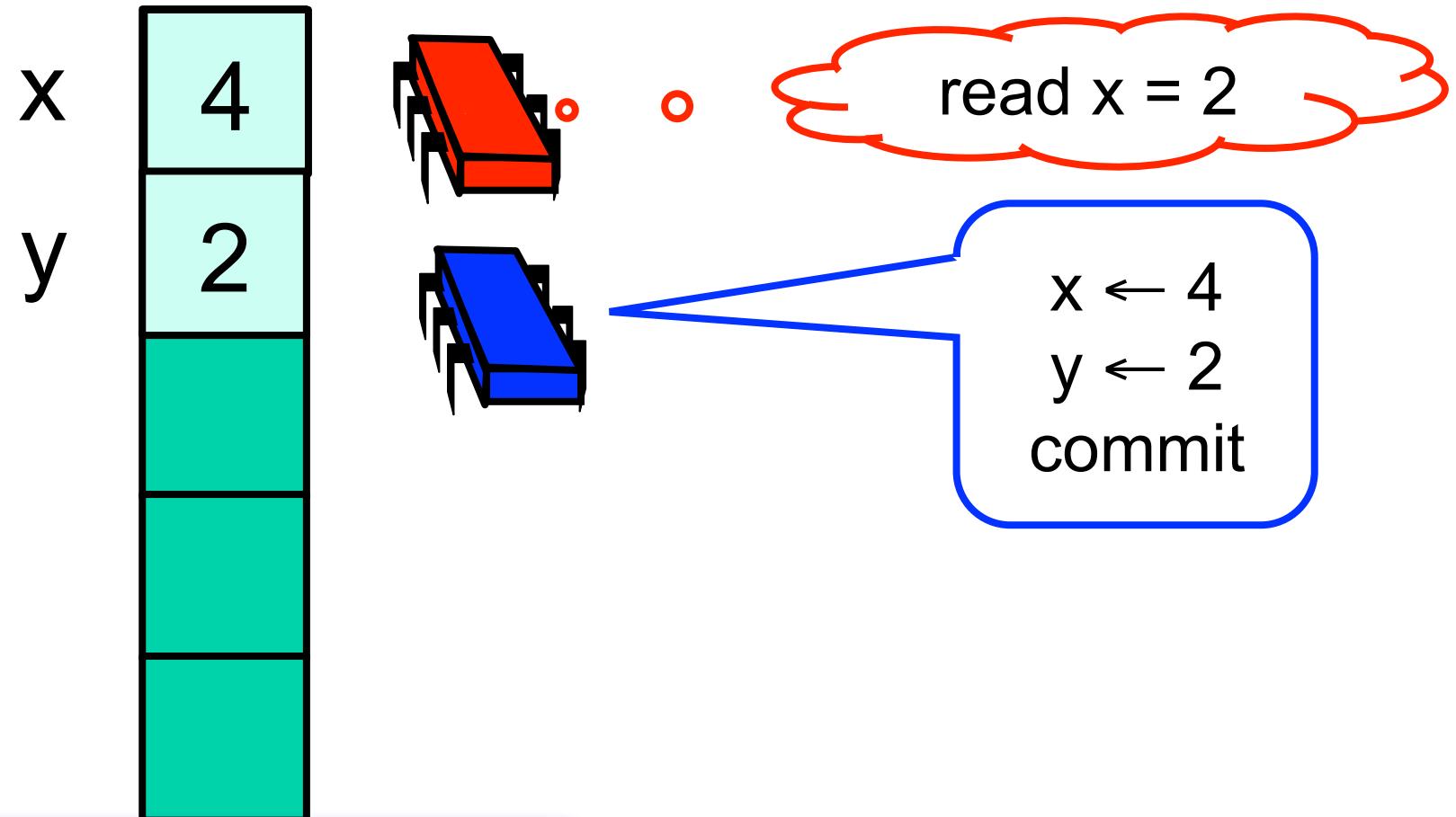




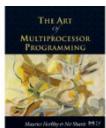
Invariant: $x = 2 \ y$

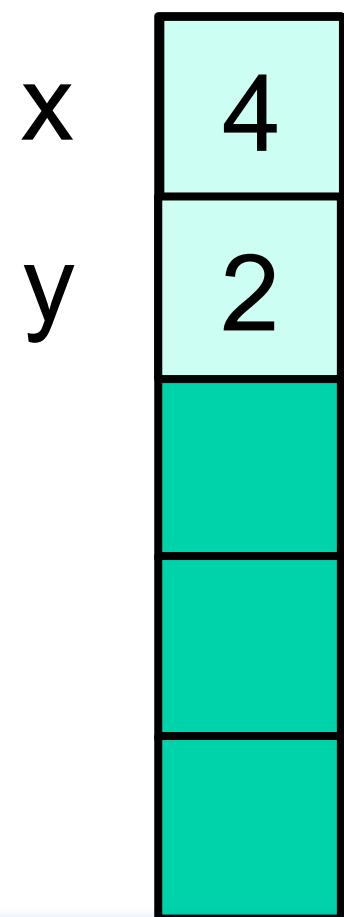




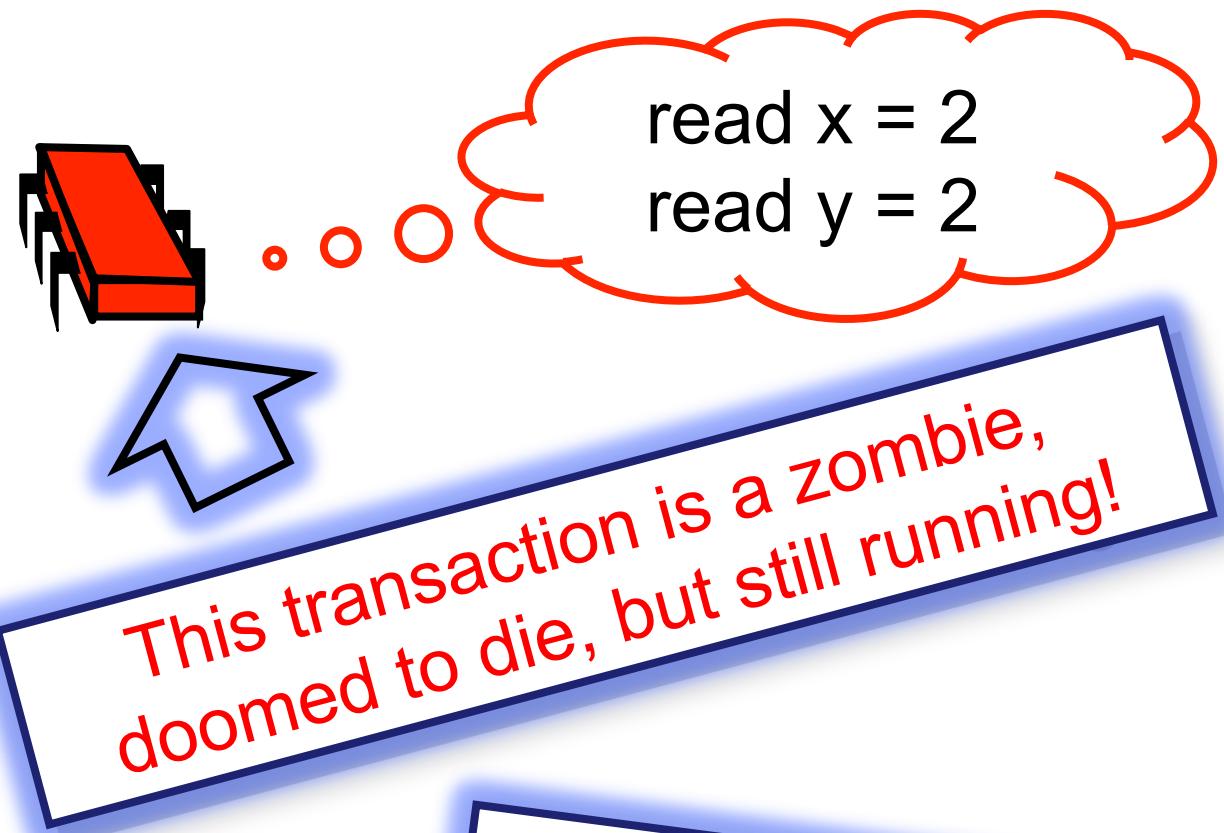


Invariant: $x = 2$ y



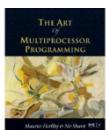


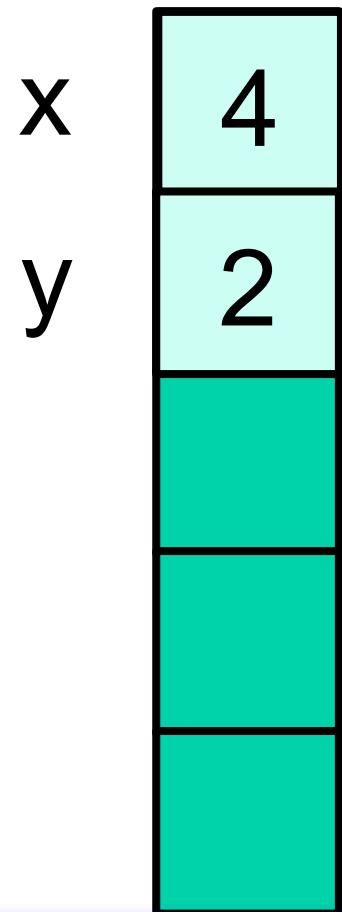
Invariant: $x = 2$ y



This transaction is a zombie,
doomed to die, but still running!

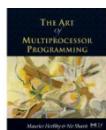
Who cares?

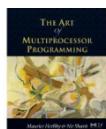
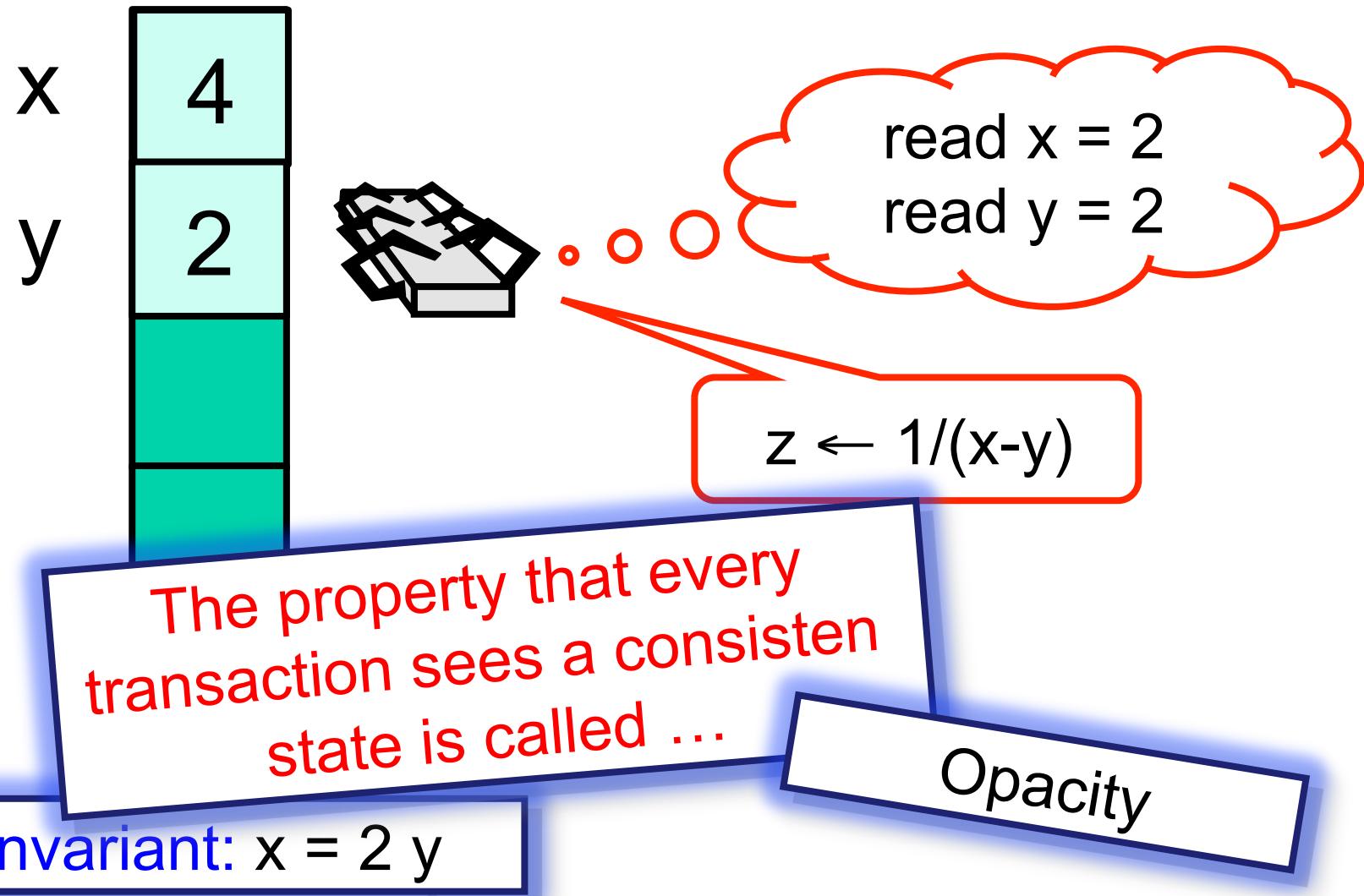




*Oh, no! It divides by zero and
crashes the system!*

Invariant: $x = 2 y$

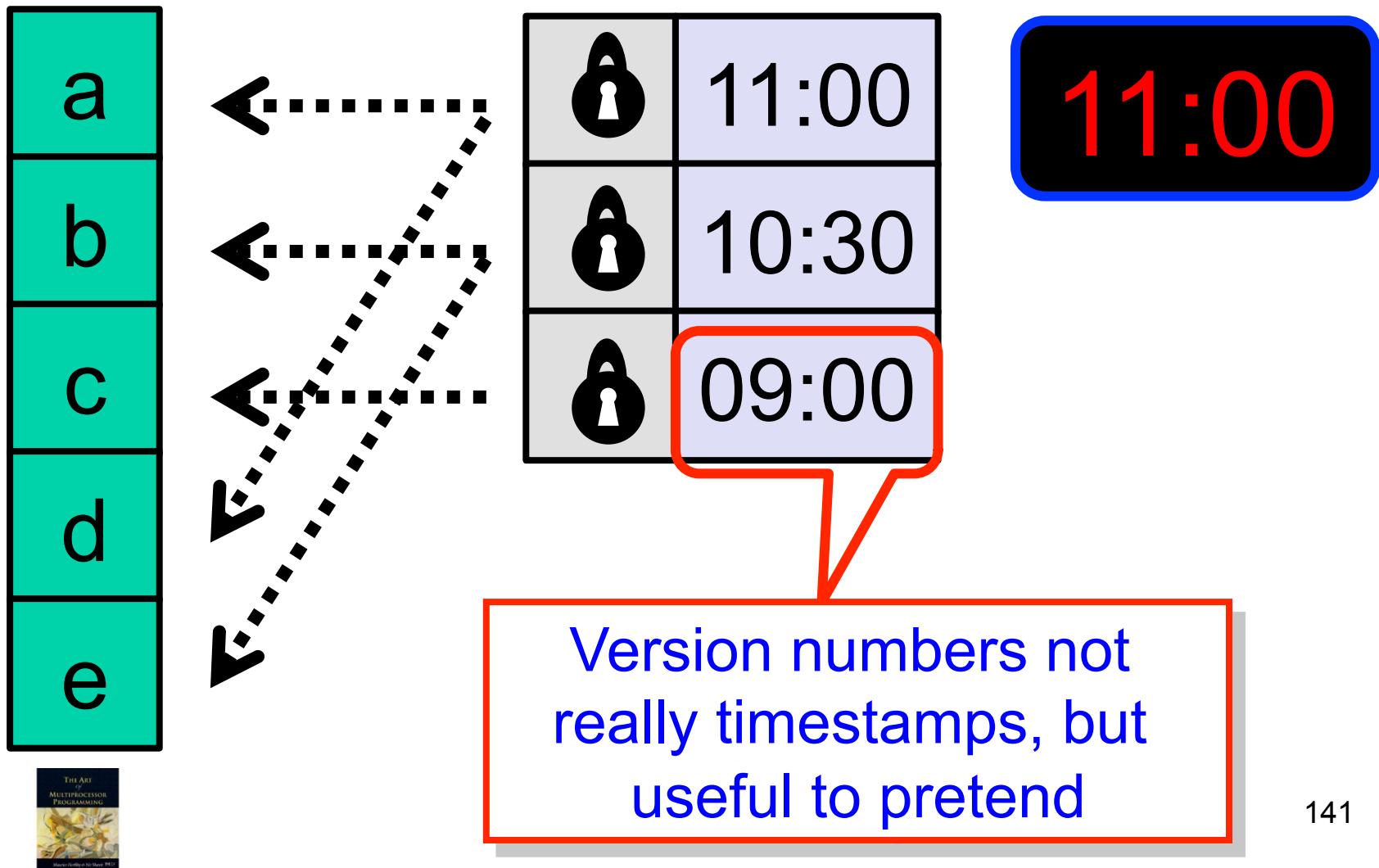




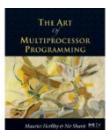
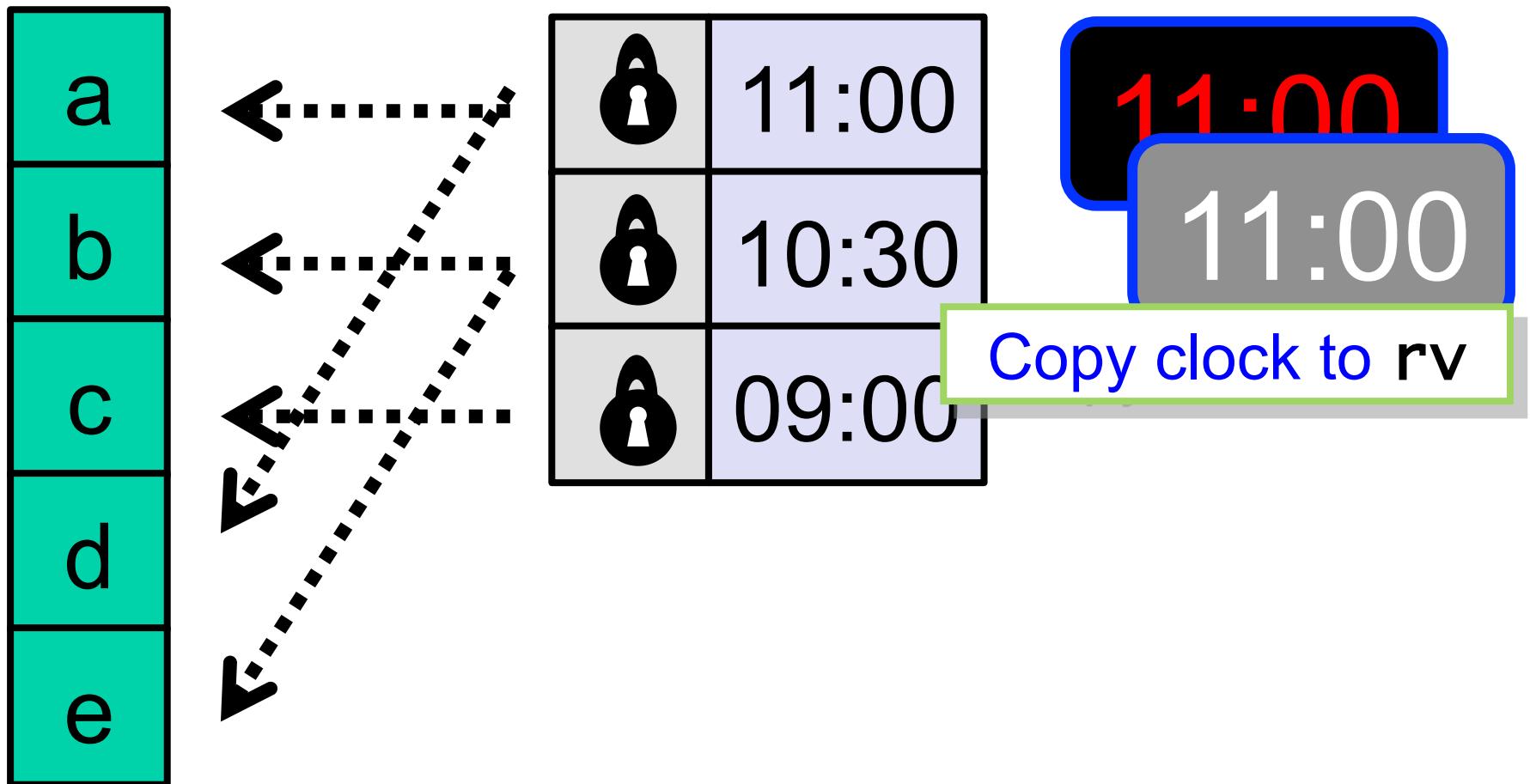
Version Clock

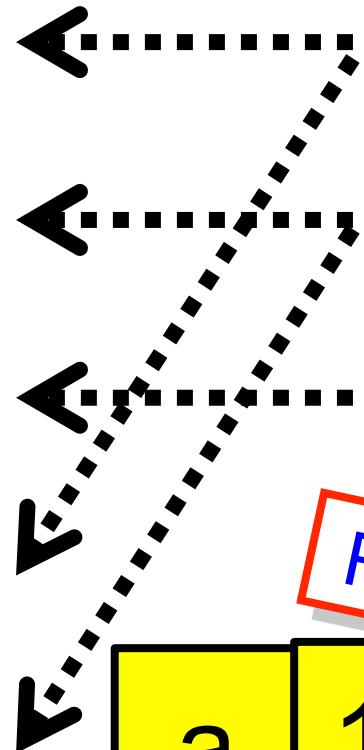
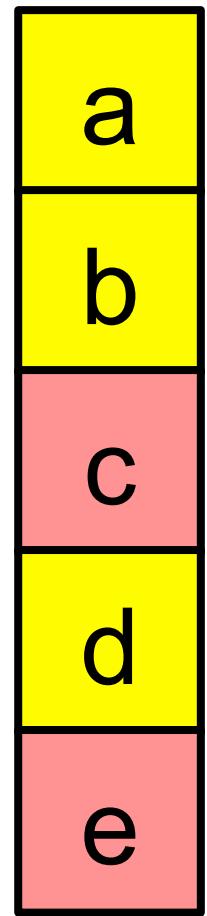


Transactin



Transactions





	11:00
	10:22
	11:00



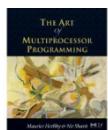
Read set

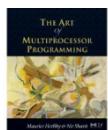
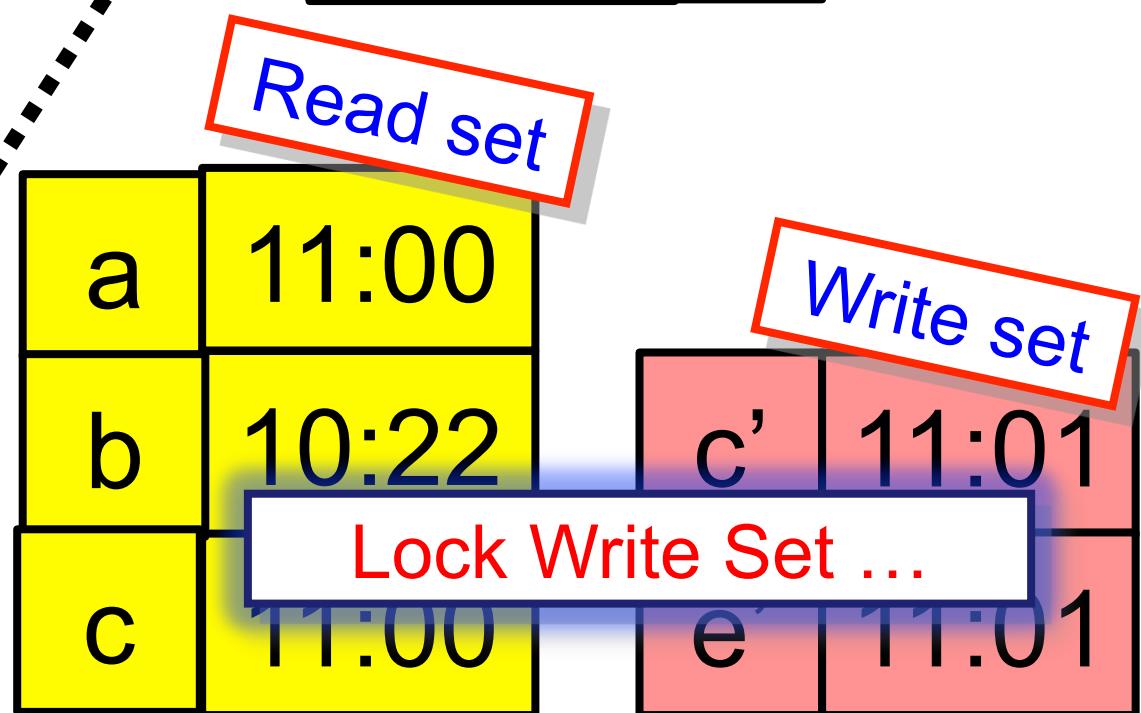
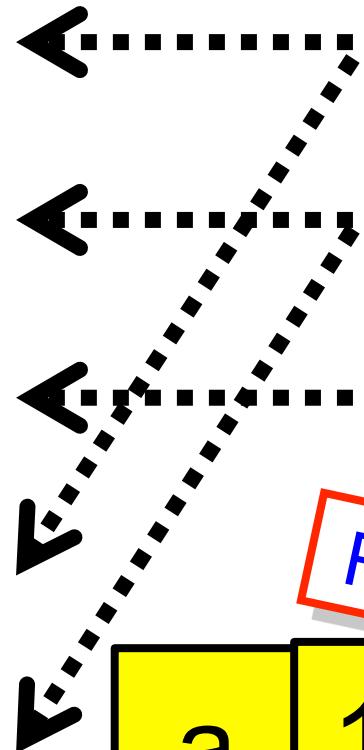
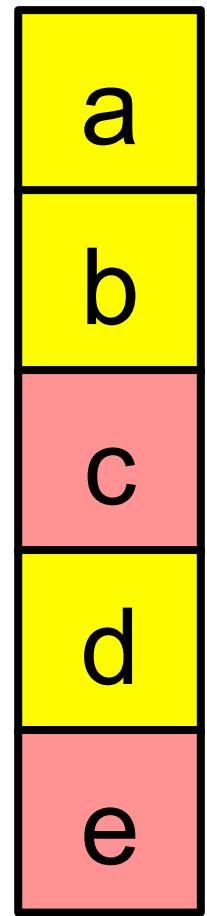
a	11:00
b	10:22

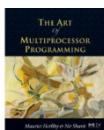
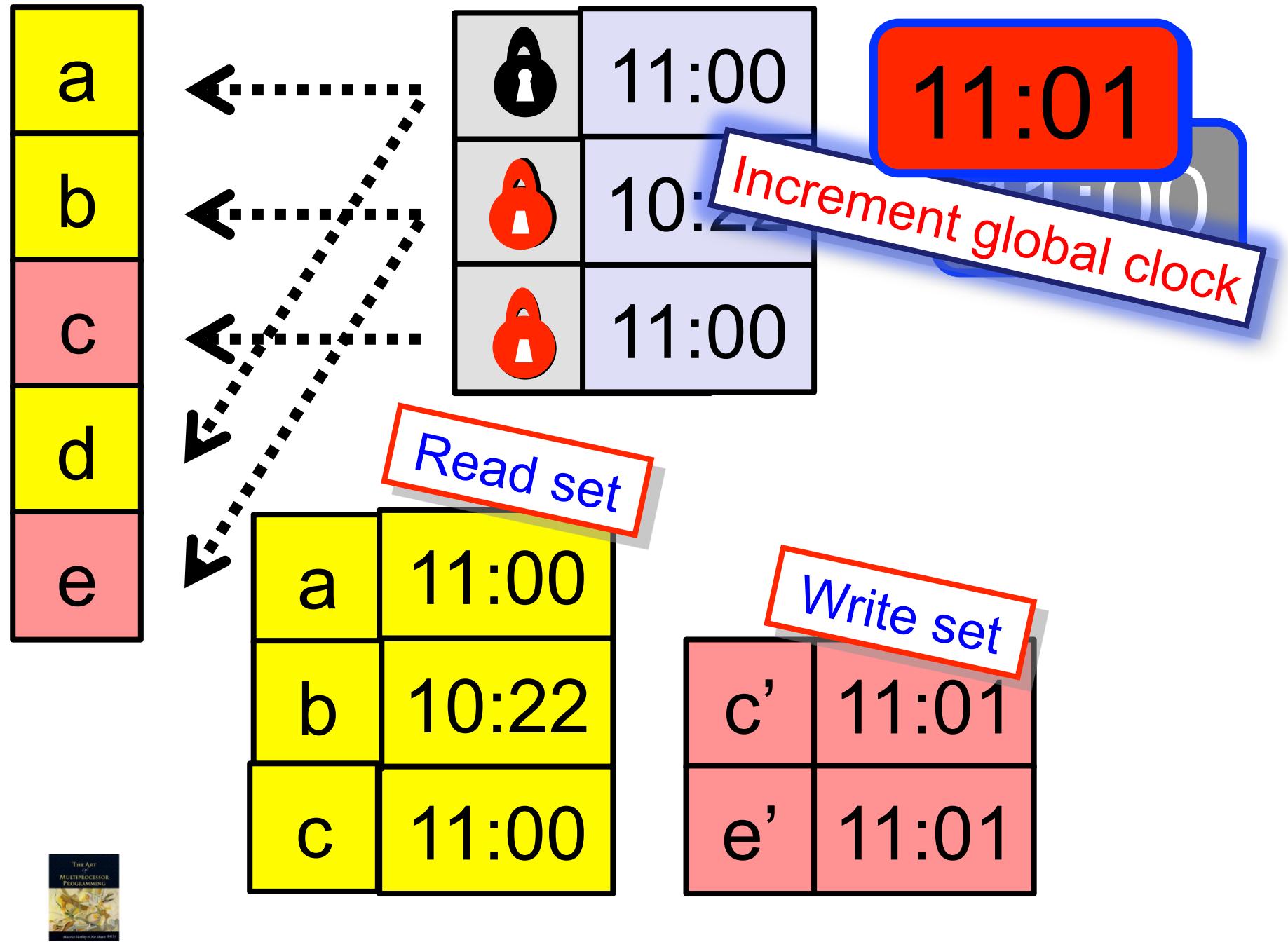
Write set

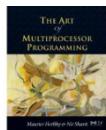
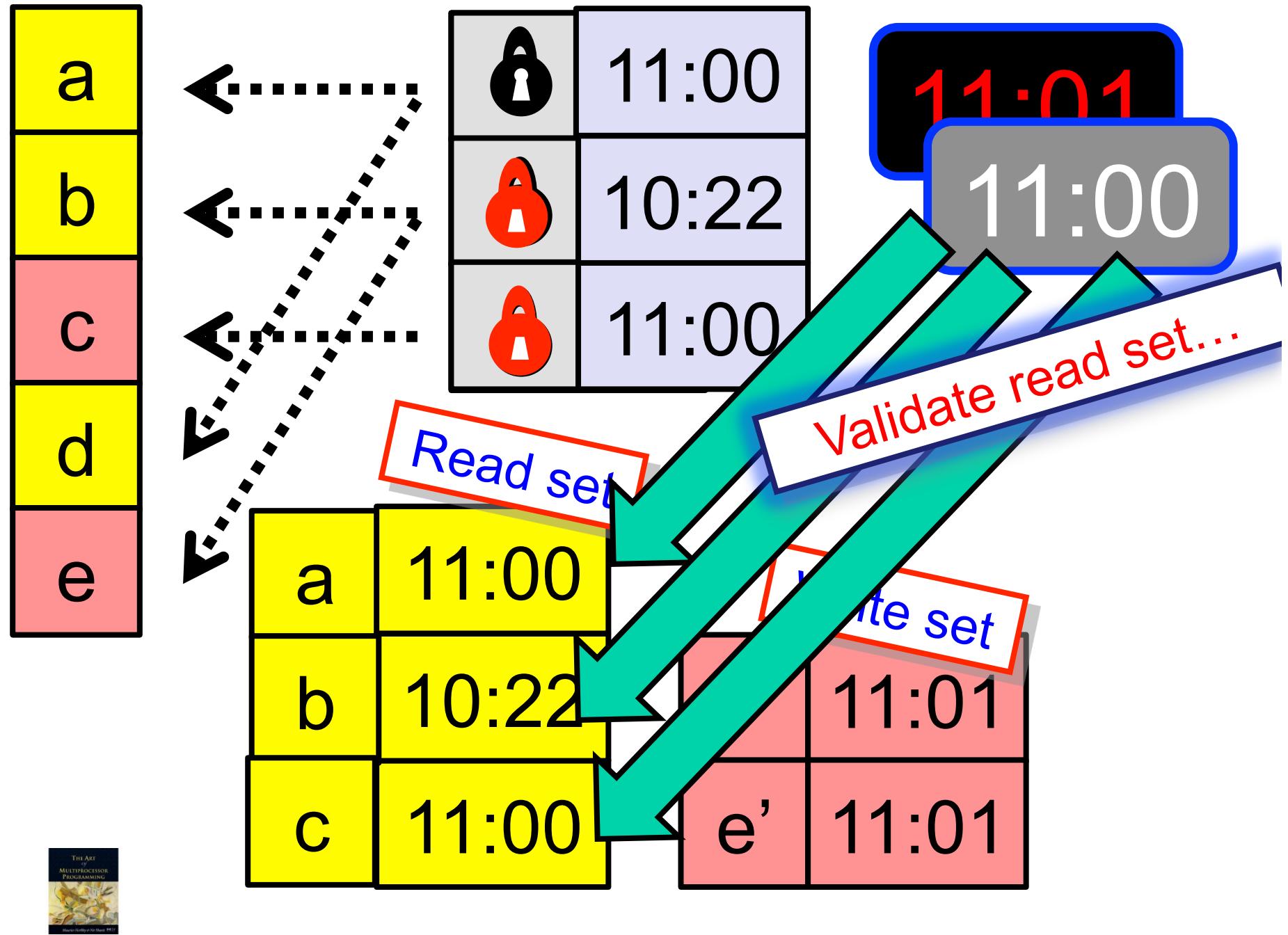
c'	1:01
d	1:01

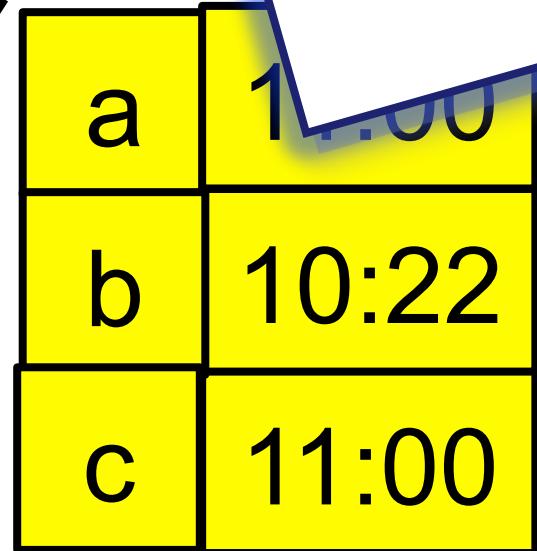
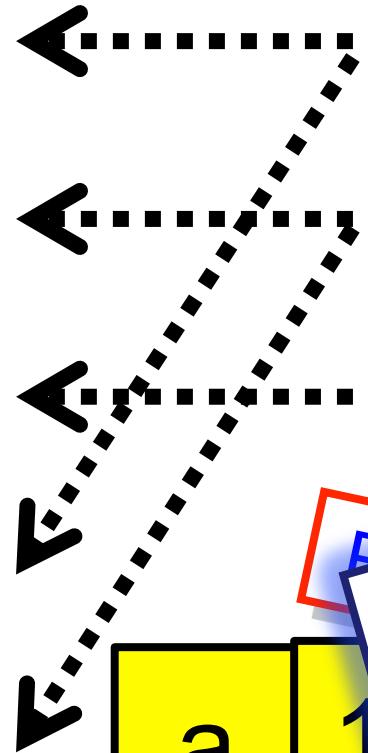
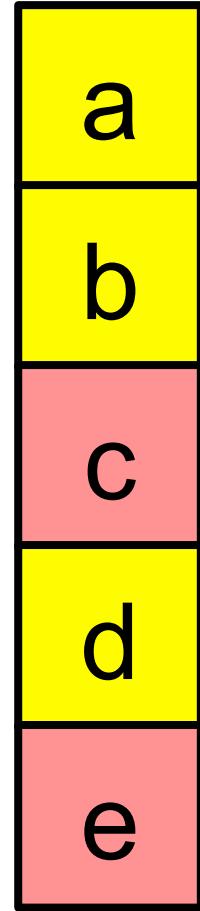
Run speculative transaction
as before ...



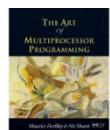
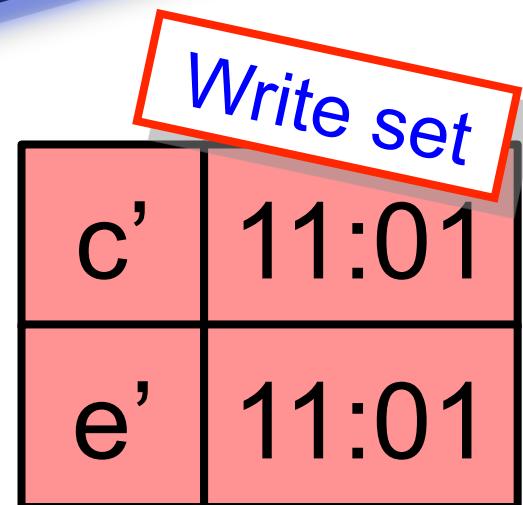




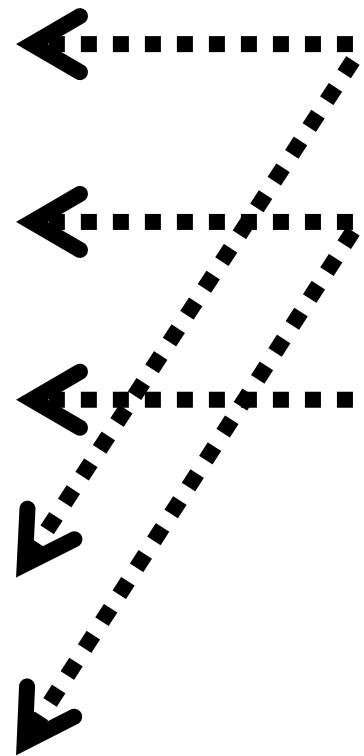




P Commit & release
locks



a
b
c
d
e



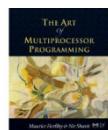
11:00
10:22
11:00

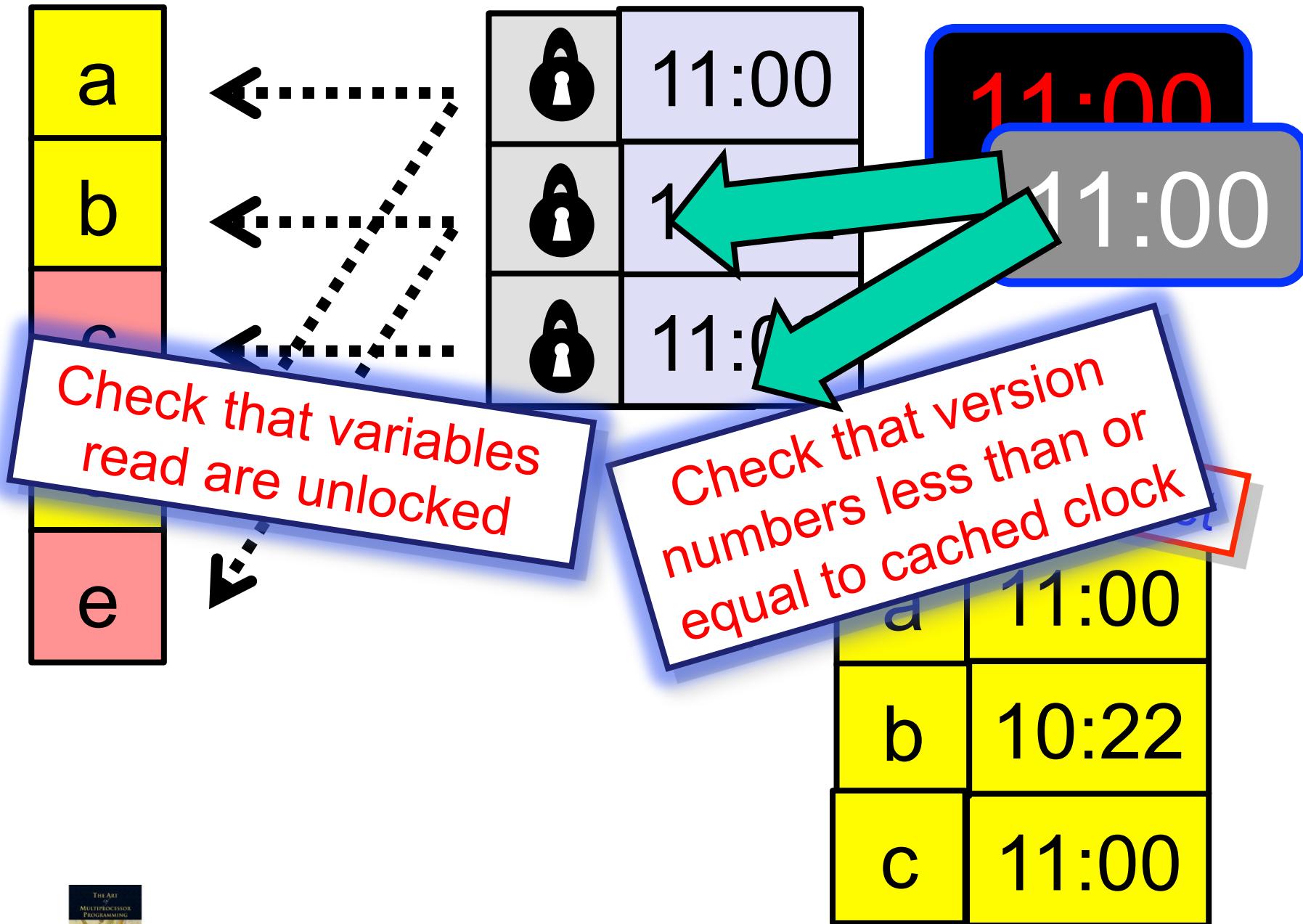


Read-only
transactions?

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

Read set





Road Map

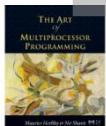
Transactional Memory

Hardware Transactional Memory

Hybrid Transactional Memory

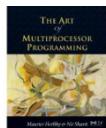
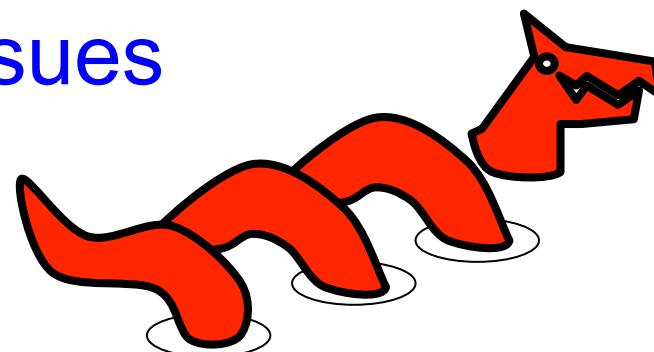
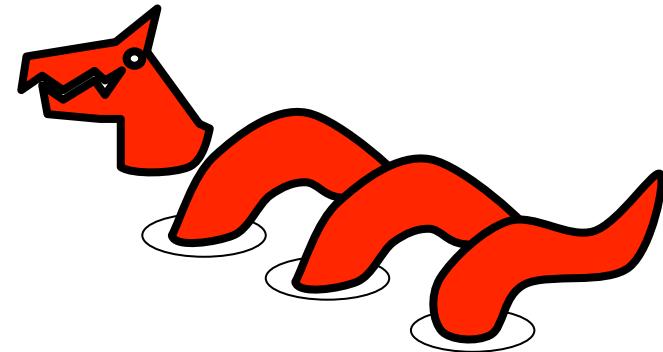
Software Transactional Memory

Research Questions



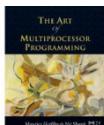
TM Design Issues

- Implementation choices
- Language design issues
- Semantic issues



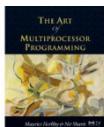
Granularity

- 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



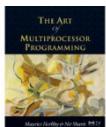
Direct/Deferred Update

- *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



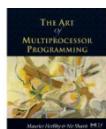
Conflict Detection

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



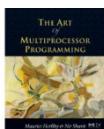
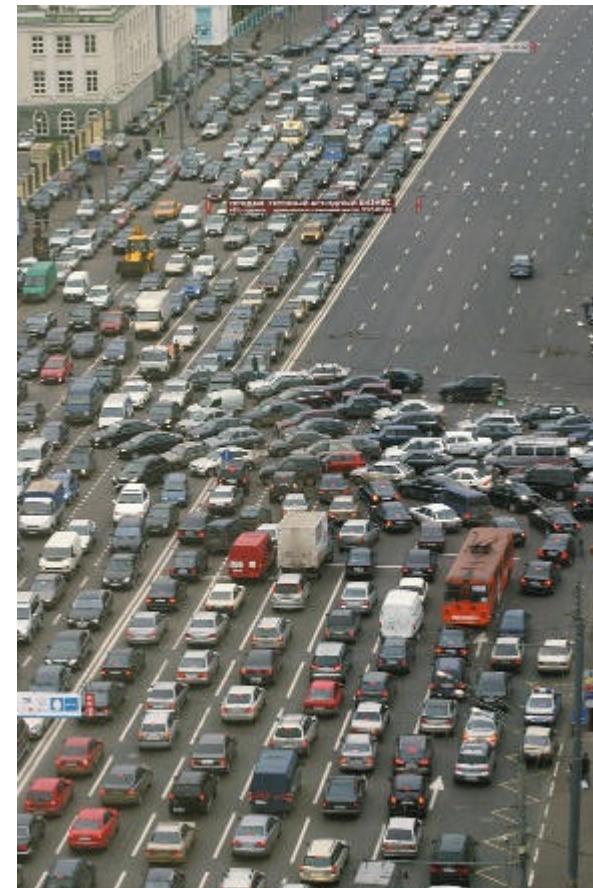
Conflict Detection

- 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



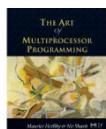
Art of Multiprocessor
Programming

Contention Manager Strategies

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

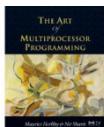


Judgment of Solomon



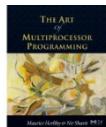
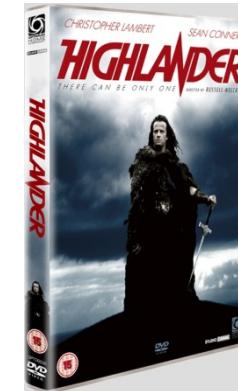
I/O & System Calls?

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



I/O & System Calls

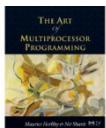
- One solution: make transaction irrevocable
 - If transaction tries I/O, switch to irrevocable mode.
- There can be only one ...
 - Requires serial execution
- No explicit aborts
 - In irrevocable transactions



Exceptions



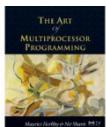
```
int i = 0;  
try {  
    atomic {  
        i++;  
        node = new Node();  
    }  
} catch (Exception e) {  
    print(i);  
}
```



Exceptions

Throws OutOfMemoryException!

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



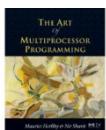
Exceptions

Throws OutOfMemoryException!

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

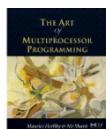
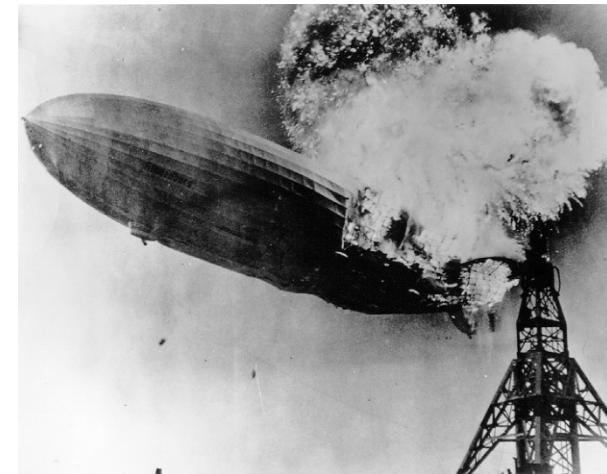
```
    print(i);
}
```

What is
printed?



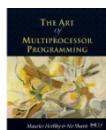
Unhandled Exceptions

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



Nested Transactions

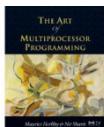
```
atomic void foo() {  
    bar();  
}  
  
atomic void bar() {  
    ...  
}
```



Art of Multiprocessor
Programming

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@brown.edu

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

Dave Dice Yossi Levy
Victor Luchangco Mark Moir
<http://csail.mit.edu/~dice/> yossi.levy@mit.edu
<http://csail.mit.edu/~victor/> mark.moir@mit.edu

Yujie Liu
Loyalty University
lyj@loyalh.edu

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

Locks and transactions complement on another

Abstract

Designing correct synchronization algorithms is a difficult task, as evidenced by a long history of literature on synchronization bugs. One of the most well-known synchronization algorithms is the RSRB algorithm, which has been used for nearly two decades. We use hardware transactional memory (HTM) to construct a corrected version of the algorithm. This correction is significantly simpler than the original and furthermore improves performance by eliminating usage constraints and reducing space requirements. Performance of the HTM-based algorithm is competitive with the original in “normal” conditions, but it does suffer somewhat under heavy contention. We successfully apply some optimizations to help close this gap, but we also find that they are not known techniques for improving parallel performance. In addition, future HTM implementations may have different characteristics than current ones, so it is important to investigate how the RSRB algorithm would fare in such a setting.

Our approach is to use HTM to implement the RSRB algorithm differently in the first phase to avoid contention on shared variables. Although the RSRB algorithm is designed to work with multiple threads, we decided to retain the basic structure after the first phase. This is because the RSRB algorithm is designed to handle the case of HTM failure, and we wanted to keep the original design as much as possible. To handle the case of HTM failure, we propose a new algorithm called TLE (Transactional Lock Elimination). TLE is a thread that repeatedly tries to commit a transaction acquire a lock, and that executes the code of its transaction non-interactively. To preserve consistency, all transactions are modified so that they cannot commit when the lock is held. We are straightforward to apply TLE to our implementation, and our evaluation confirmed that the overhead was low. In addition, we have explored a variety of ways to reduce contention in the RSRB algorithm, and we have evaluated a variety of ways to reduce contention in the TLE algorithm.

Locks and transactions complement on another. Locks can be used to ensure consistency from the transactional list based on lock elimination substantially reduces contention, and slightly outper-

Memory Management

StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation

Dan Alistarh*
 MSR Cambridge
da@microsoft.com

Patrick Eugster
Purdue University
e@cs.psu.edu

Maurice Hirsch
Brown University
mh@cs.brown.edu

Alexander Matveev
MFF

Nir Shavit
MIT and TAU
shavit@csail.mit.edu

On the Impact of Dynamic Memory Management on Software Transactional Memory Performance

*Sabon Rodo - Gólio Aracy
UNICAMP - Currículo de
Investigación*

Chihuahua: A Concurrent, Moving, Garbage Collector using Transactional Memory

Abstracts

Abstract
 Hardware Transactional Memory (HTM) offers a promising approach to parallel synchronization. In addition, for one class of memory operations, it is different from techniques like *lock-free* or *atomic*. Because of HTM's inherent properties, when algorithms based on earlier mechanisms such as locks and atomic operations are adapted to HTM based systems, big differences may appear. In this paper, we propose a new mechanism for distributed memory transactional memory. The main idea is to use a lock-free algorithm for each transaction. We also propose a new mechanism for distributed memory transactional memory. The main idea is to use a lock-free algorithm for each transaction.

T
b

TM can improve memory management, both automatic and explicit.

Exploring Garbage Collection with
Haswell Hardware Transactional Memory
Tobias Unger
University of Technology
<http://hbw.kit.edu/~tunger/>

Brian E. Wren,
Director of Risk
& Insurance, AIA

target simultaneously in mark objects, e.g. by setting this as a target, the user would upload content and no marks would be added, instead, targeting multiple mark objects, it is then possible to add multiple targets to a single file. In general, an open piece, previously, has been considered as a document that contains specific applications, such as Microsoft Word or Excel, whereas now it can be thought as a document that contains multiple applications.

names that new the
ung threads. Recent
functional memory to spe-
- cial memory reclamation so-
- - al known techniques etc.

Keywords: Osteopetrosis; Trauma; Bone
E-mail: Kwiatkowska@med.poznan.pl

Recently, the U.S. government has developed a system designed to detect and identify about 10 percent of all exports made by U.S. companies.

DB

Exploiting Hardware Transactional Memory in Main-Memory Databases

Viktor Lrix, Alfons Kasper, Thomas Nussbaumer
Fakultät für Informatik
Technische Universität München
Schellingstrasse 4, D-80718 München
e-mail: nussbaumer@in.tum.de

Summary—So far, Lewin's field memory—although a promising technique—suffered from the absence of an efficient hardware implementation. The upcoming Harvard radix memory will have been implemented. Address-based memory (HTM) is a more efficient CPU-level solution for efficient communication, and operation is also highly efficient in the cache databases. On the other hand, HTM has several disadvantages, primarily in terms of performance and complexity.

Stack for IOPS/Capacity Bifurcated Storage Env.
Dedicated Storage, Kudos for Virtugard.
StarApp, Inc.

• A Storage Stack for 10IPS/Capacity Bifurcated Storage Environment
Douglas Society, Kishorabai Vartaknagar,
NetApp, Inc.

Using Restricted Transactional Memory to Build a Scalable In-Memory Database

Zhiqiao Wang¹, Hao Quan², Juyang Li³, Haibo Chen¹,
¹School of Computer Science, Fudan University
²Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
³Department of Computer Science, New York University

SYNTHETIC

Abstract. The recent availability of fast Ethernet precision marks the transition of local area networking from research labs to end-user facilities. DMA is an inexpensive hardware device that uses local-area networking technology (Ethernet) to achieve high performance and good availability across a wide range of environments. The main limitation has also been overcome by DMA in its concentrated working mode (as opposed to relay), that scales up to 100 nodes per switch data with about 100 Mbytes/s. The design of DMA addresses this challenge in several ways. First, DMA builds a distributed backbone in top of an underlying shared memory fabric. Second, DMA uses separate DMA regions to separate management, control, and switching functions. Third, DMA works on the RMAC standard and scales to 10 Gbytes/s. The fourth feature of DMA is its high performance, which is 10 times per second on a 4-port machine.

www.FCC.gov

1. Introduction
Writing high performance, correctly is a challenge. To date, practitioners had several approaches to choose from. Change-oriented tools provide a straightforward programming model that can lead to poor performance unless handled carefully. In addition, some of these tools do not support the notion of reuse (planned, tricks and serials, open-source).

U.S.
U.S.

globalization
and increased competition.
Global markets provide a
broad base in comparison
to a company's regional
or domestic opportunities.

370

In this paper we describe a new, more efficient method for partitioning hard disk drives with capacity optimized bootstrap disk systems. Currently, for efficiency reasons, most system applications are forced to rely on their own proprietary graphical user interfaces to manage disk volumes or other information disk layout. Data structures which are generated by these user interfaces are generally not shared with file systems. When a provider installs one of these tools, it provides its own graphical user interface, usually a command-line interface, to access the data managed by that tool. This makes it difficult to perform file system management. Under the following sections we will introduce our new system and compare it with currently optimized

Digitized by srujanika@gmail.com

uctures in-
to the system. The system is designed to provide statistical information to management, and to help management make better decisions. The system is based on a combination of statistical methods and computer technology. It provides management with a wide range of information, including financial data, market research, and operational data. The system is designed to be user-friendly and easy to use. It is also designed to be flexible, so that it can be customized to meet the specific needs of different organizations.

4. Grids/Three-Dimensional

and pattern. DB-based memory management has been forced to stop their development. Many data stations will be forced to use memory-resident memory manager as well as changed from using them to a cache-oriented to the difficulty it is posing in the development of device drivers. This difficulty is mitigating the developments of memory systems. One way of implementation of the memory system is to use a memory manager program that is to have a mapped memory space. The memory manager will be able to handle the memory space and the memory access requests.

In the first, static DRAM, the bank controller is responsible for managing the memory system. This controller is a dedicated memory system, too. In contrast, in a dual-channel memory system, there are two sets of memory controllers, one for each channel. The bank controller is responsible for managing the memory system. This controller is a dedicated memory system, too. In contrast, in a dual-channel memory system, there are two sets of memory controllers, one for each channel.

Abstract A study of open access preservation policies in 100 academic libraries in the United States and 100 in Germany was conducted. The results show that while most libraries in both countries have adopted some form of open access preservation, there are significant differences between the two countries in terms of the types of open access preservation adopted and the reasons for doing so. The findings also indicate that while most libraries in both countries believe that open access preservation is important, they are not fully aware of its implications for their collections and services.

the development of the brain and nervous system. The brain and spinal cord are composed of billions of nerve cells, called neurons, which transmit information through electrical signals. The brain is divided into several regions, each with specific functions. The cerebrum, the largest part of the brain, is responsible for higher-level functions such as language, memory, and problem-solving. The cerebellum coordinates movement and balance. The brain is surrounded by a protective layer called the meninges, and is cushioned by cerebrospinal fluid. The brain is connected to the rest of the body via the spinal cord, which carries signals between the brain and the rest of the body.

Databases

TM restructures in-memory databases

Abstract

Improving In-Memory Database Index Performance with
Just-in-Time Transactional Synchronization Techniques

Tomasz Karczewski¹, Raman Dasgupta², Wolfgang Lehner³,
Thomas Leyler⁴, Bernhard Grotzsch and Michael Hähnlein¹,
SiAP AG, Database Technology Group, Braunschweig, Germany
¹SiAP AG, Database Technology Group, Braunschweig, Germany
²SiAP AG, Database Technology Group, Braunschweig, Germany
³TU Dresden, Dresden, Germany
⁴SiAP AG, Database Technology Group, Braunschweig, Germany

Power and Energy

Energy-Aware Microprocessor Synchronization
Transactions vs. Memory vs. Locks

A Maurice Herlihy[†]
Providence, RI 02912
Dimitra Papagiannouli[‡]
Providence, RI 02912

**Playing with Fire: Transactional
Error-Resilient and Energy-Effici**

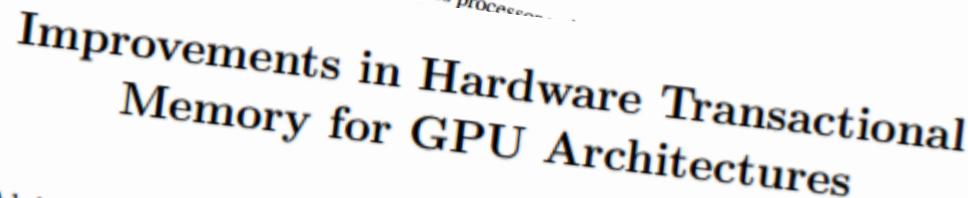
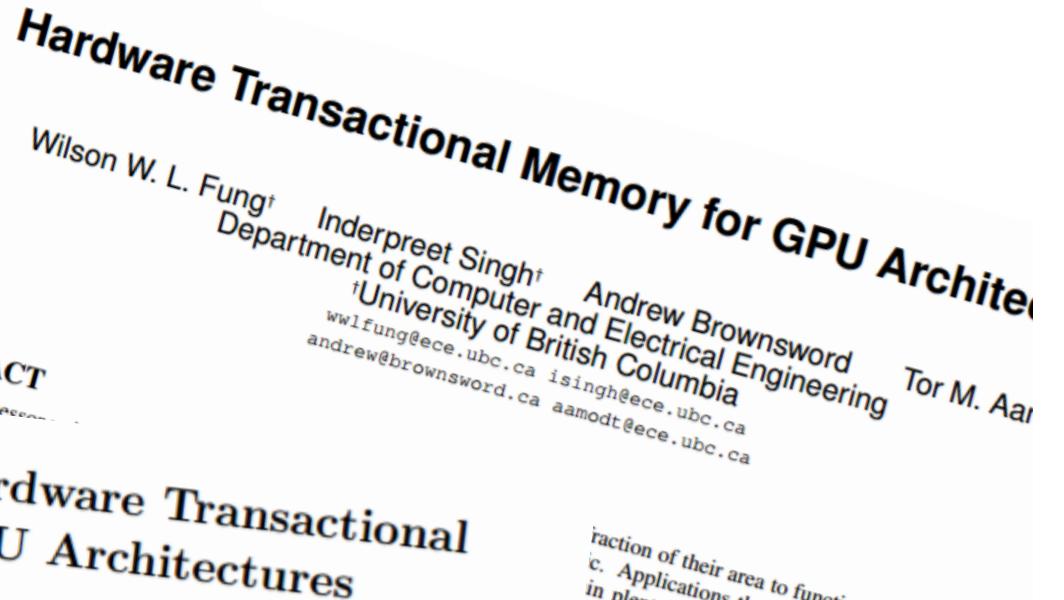
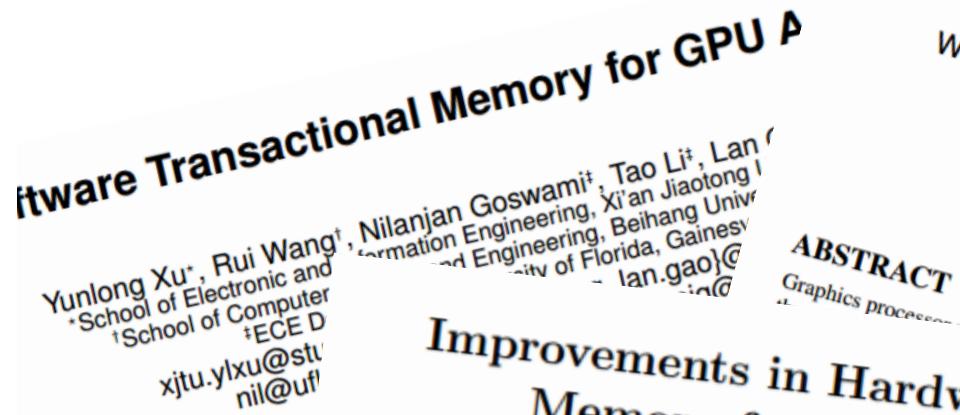
New research in energy-efficient synchronization

F. Klein

Efficiency of Software Transactions
Memory

Marongiu
Zurich
see.ethz.ch

GPIs, etc.



GP

To make the best use of GPU acceleration, it is important to understand the memory system and the challenges associated with it. The major challenge is preventing livelock in the memory system of GPUs. To this end, various techniques have been proposed.

GPUs and accelerators need synchronization





8th Workshop on the Theory of Transactional
Memory
WTTM 2016
July 25, 2016, Chicago

[International Conference on Computer Aided Verification](#)
[CAV 2009: Computer Aided Verification pp 1-15
\(Full Paper\)](#)

Theory | The Theory of Transactional Memory

Guerraoui, Rachid; Kapalka, Michal
Published in: Bulletin of the EATCS, num. 97
Publication date: 2009

Transactional memory (TM) is a promising paradigm for concurrent programming. This paper is an overview of our work on TM. We first present a corrective approach to existing TM implementations. Then we characterize the two main classes of clock-based TMs in terms of their power.

Architecture



Journal of Systems Architecture
Volume 73, February 2017, Pages 42–52



Hardware transactional memory architecture with adaptive
version management for multi-processor FPGA platforms

Ian Sirkunam^a, Chia Yee Ooi^b, N. Shaikh-Husain^a, Yuan Wu^a

Show more

<https://doi.org/10.1016/j.sysarc.2016.11.001>

Transactional Memory
IBM System Z

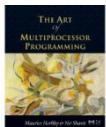
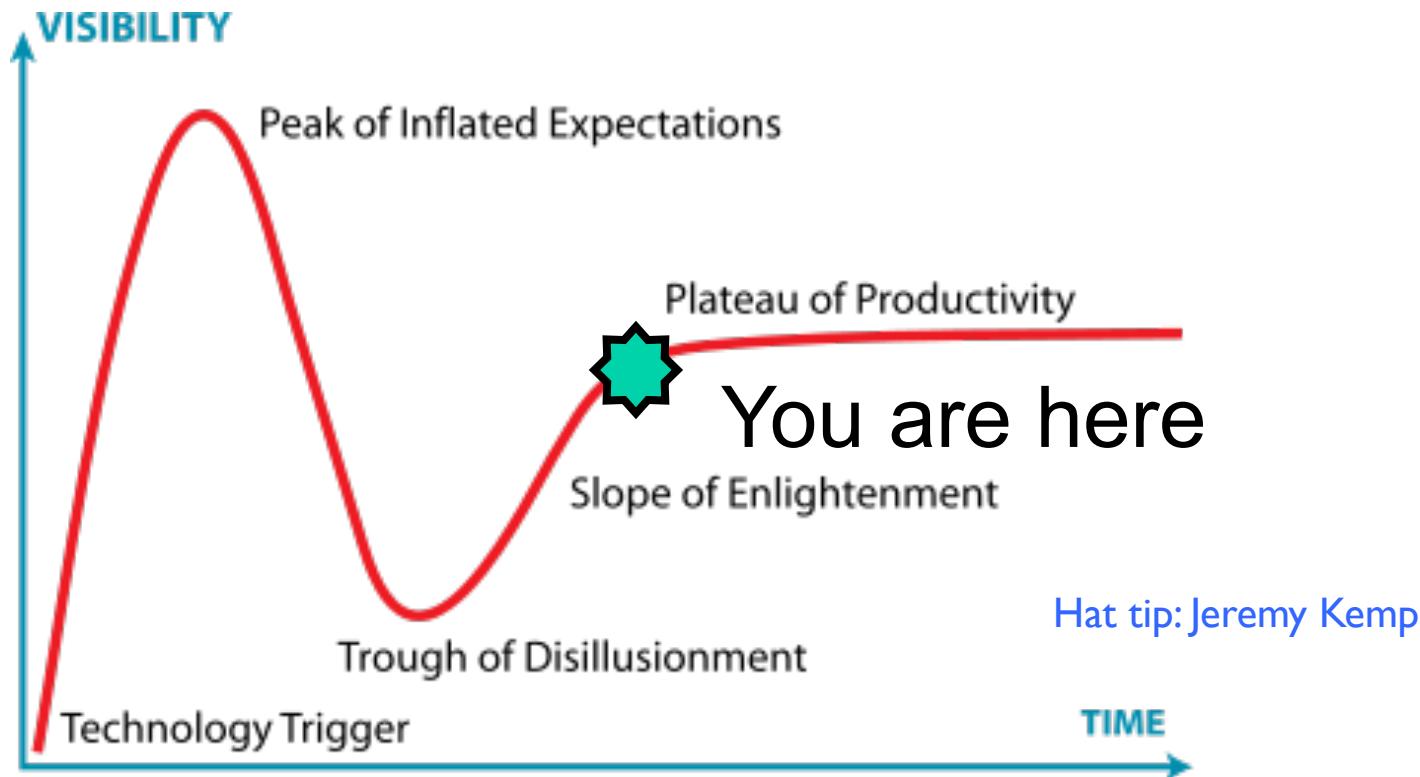
Hardware Acceleration of Transactional
Memory on Commodity Systems

38
Paper Citations

26
Patent Citations

322
Full Text Views

Gartner Hype Cycle



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

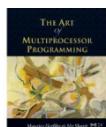
Revision:

1

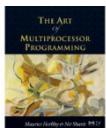
Introduction



Intel® Architecture Instruction Set Extensions Programming Reference



Спасибо!



Art of Multiprocessor
Programming

177

