

Dependence-Aware Transactional Memory for Increased Concurrency

Hany E. Ramadan,
Christopher J. Rossbach,
Emmett Witchel
University of Texas, Austin

Concurrency Conundrum

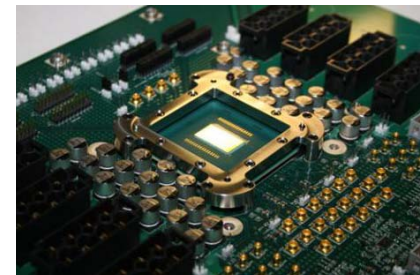
- Challenge: CMP ubiquity
- Parallel programming with locks and threads is difficult
 - deadlock, livelock, convoys...
 - lock ordering, poor composability
 - performance-complexity tradeoff
- Transactional Memory (HTM)
 - simpler programming model
 - removes pitfalls of locking
 - coarse-grain transactions can perform well ***under low-contention***



2 cores







16 cores



80 cores
(neat!)

High contention: optimism warranted?

- **TM performs poorly with write-shared data**
 - Increasing core counts make this worse
- **Write sharing is common**
 - Statistics, reference counters, lists...
- **Solutions complicate programming model**
 - open-nesting, boosting, early release, ANTs...

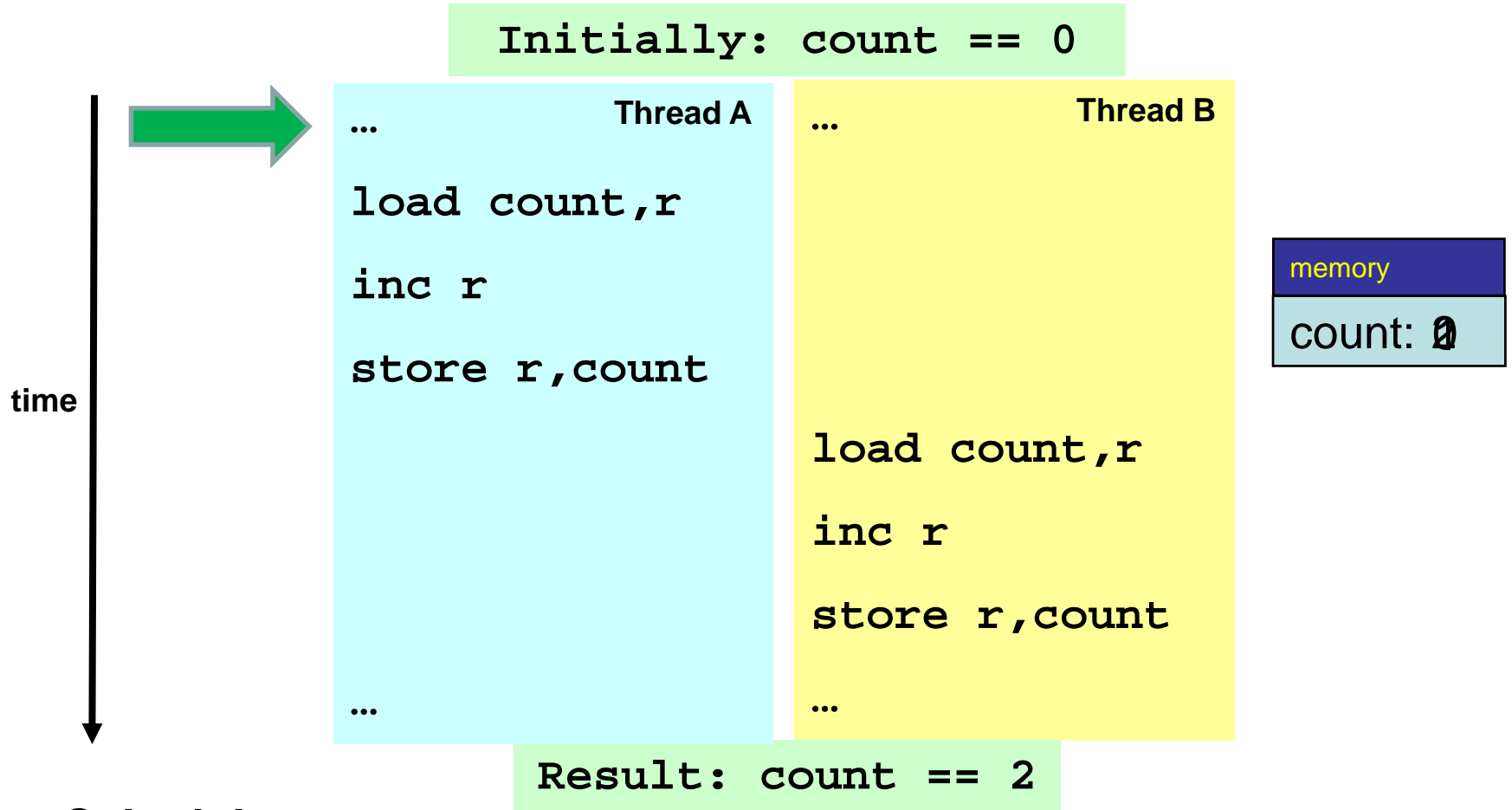
	Locks	Transactions
Read-Sharing		
Write-Sharing		

Dependence-Awareness
can help the programmer
(transparently!)

Outline

- Motivation
- Dependence-Aware Transactions
- Dependence-Aware Hardware
- Experimental Methodology/Evaluation
- Conclusion

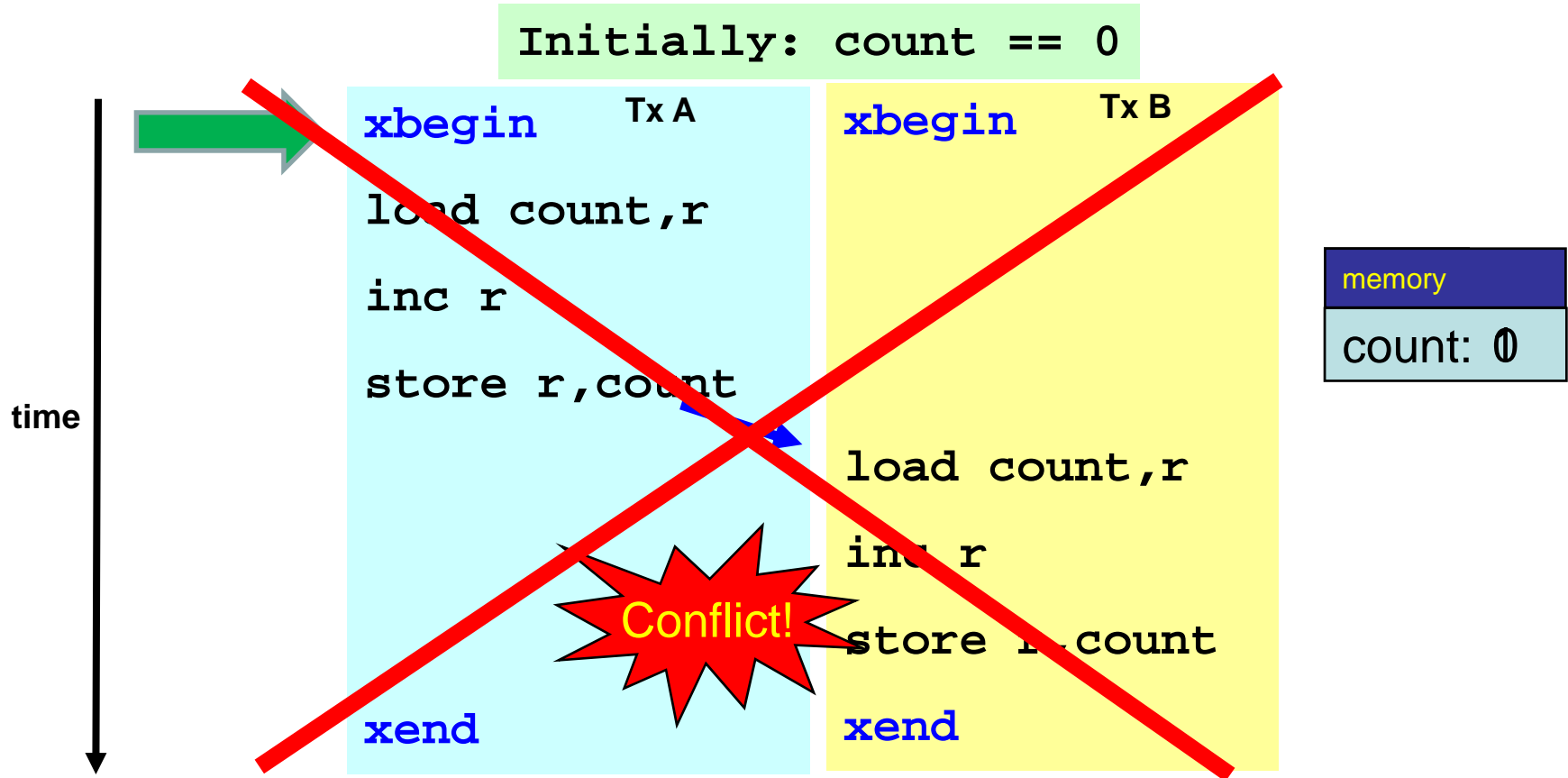
Two threads sharing a counter



Schedule:

- dynamic instruction sequence
- models concurrency by interleaving instructions from multiple threads⁵

TM shared counter



Conflict: Current HTM designs cannot accept such schedules,

- both transactions access a datum, at least one is a write
- despite the fact that they yield the correct result!
- intersection between read and write sets of transactions

Does this really matter?

xbegin Critsec A

<lots of work>

```
load count,r  
inc r  
store r,count
```

xend

xbegin Critsec B

<lots of work>

```
load count,r  
inc r  
store r,count
```

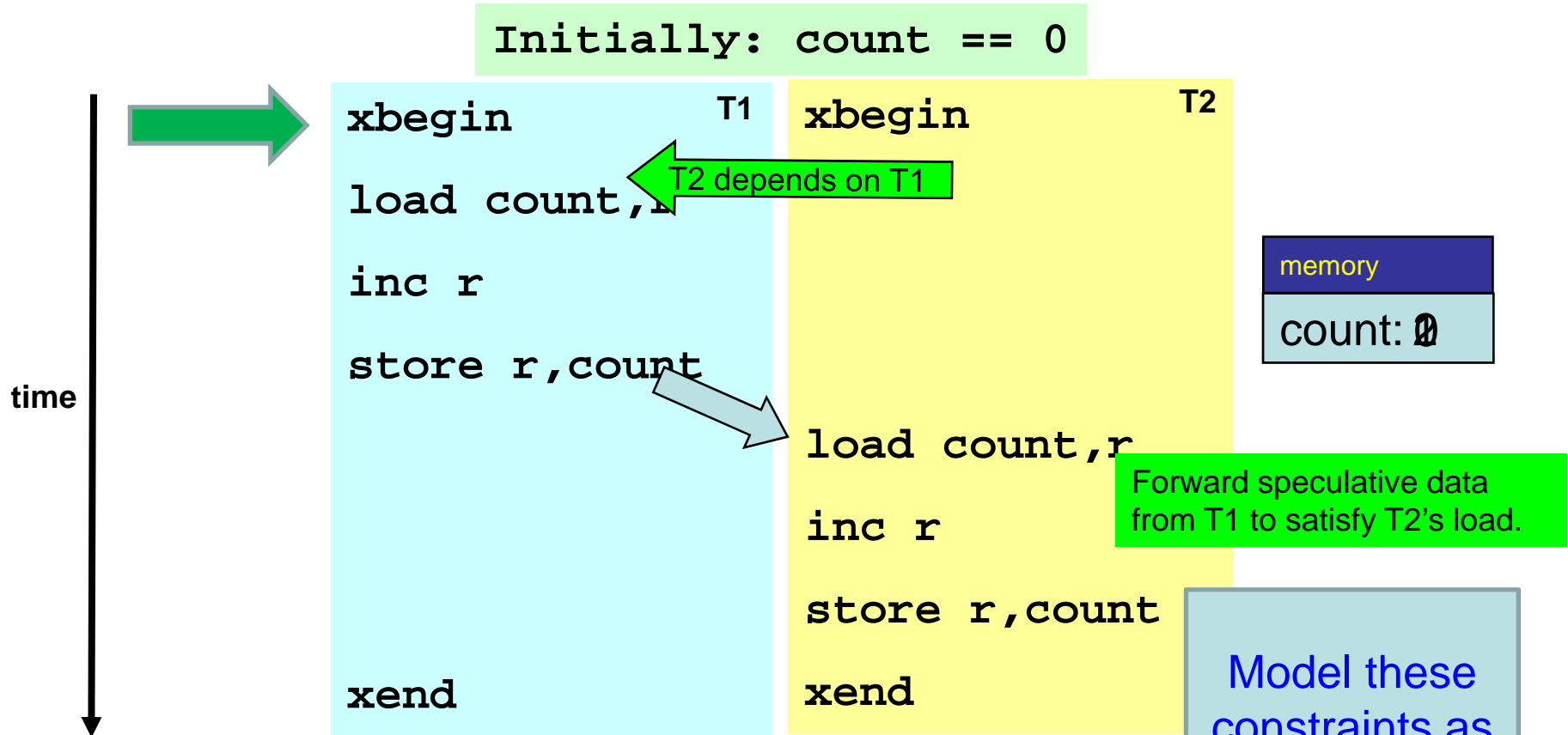
xend

Common Pattern!

- Statistics
- Linked lists
- Garbage collectors

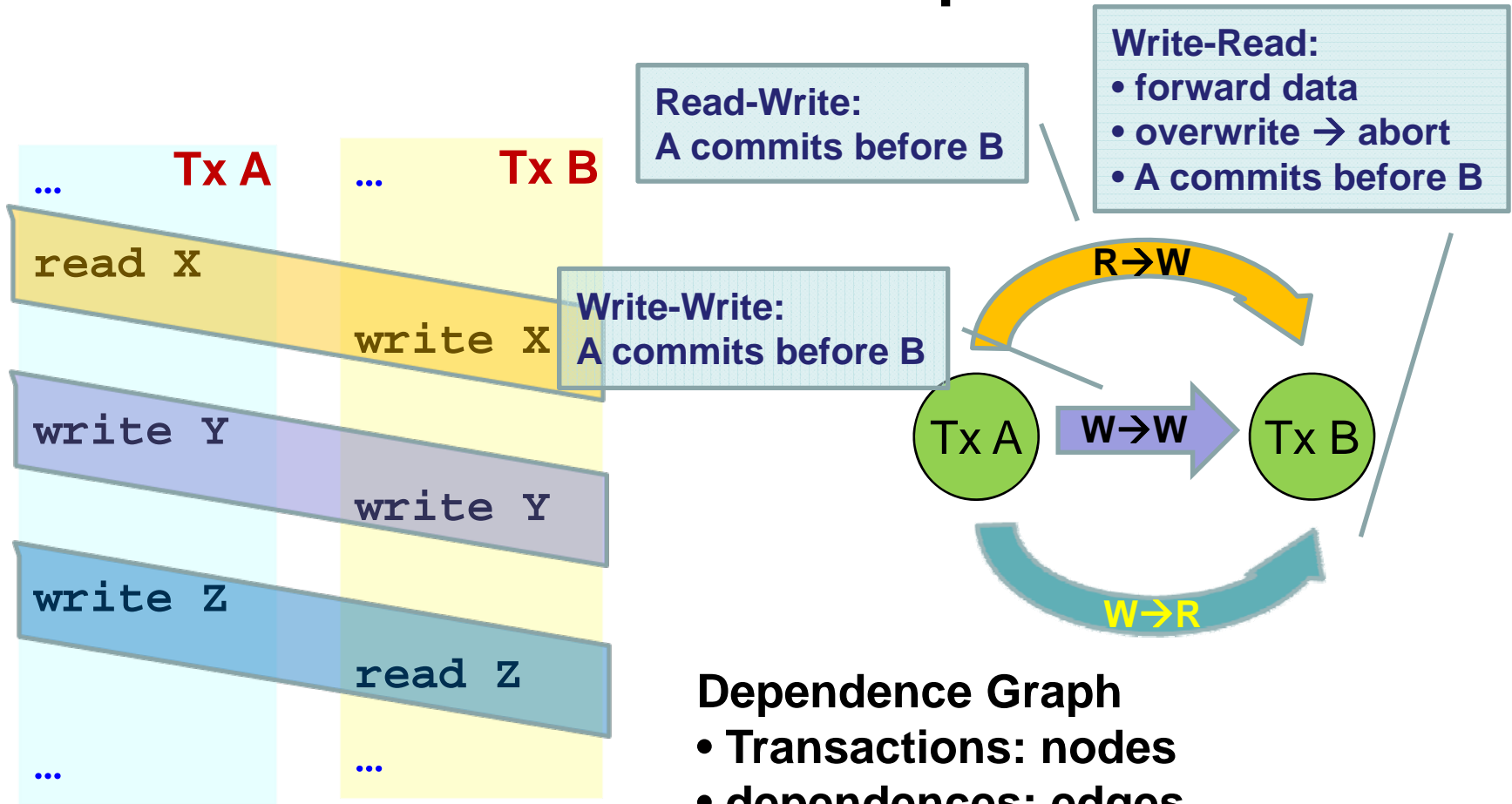
DATM: use dependences to commit conflicting transactions

DATM shared counter



- T1 must commit before T2
- If T1 aborts, T2 must also abort
- If T1 overwrites `count`, T2 must abort

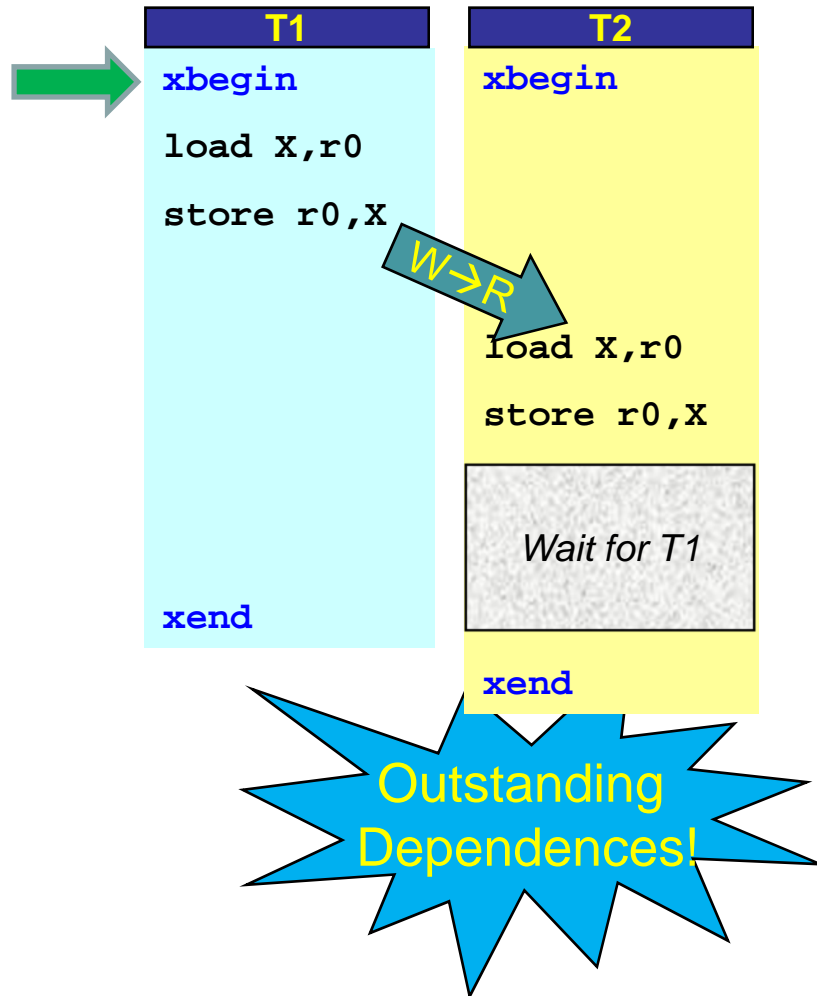
Conflicts become Dependencies



Dependence Graph

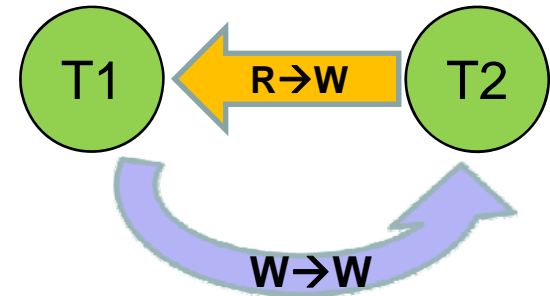
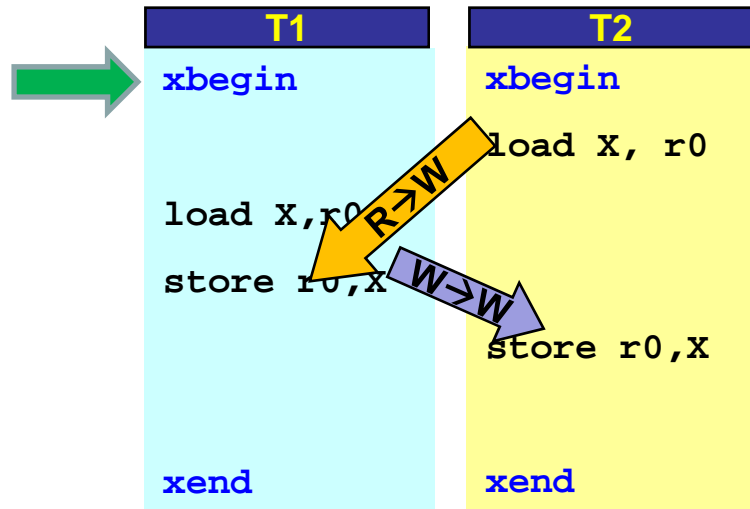
- Transactions: nodes
- dependencies: edges
- *edges dictate commit order*
- *no cycles → conflict serializable*

Enforcing ordering using Dependence Graphs



T1 must serialize before T2

Enforcing Consistency using Dependence Graphs



- cycle → not conflict serializable
- restart some tx to break cycle
- invoke contention manager

Theoretical Foundation

Correctness
and optimality
Proof: See
[PPoPP 09]

results of
leaving equivalent

Serializable

- Co
- Co
- Co
- 2-p
- imp

Dependence	Forward	Restart
$W_0 \rightarrow W_1$	No	If in cycle
$R_0 \rightarrow W_1$	No	If in cycle
$W_0 \rightarrow R_1$	Yes	If in cycle, and T_1 must if either: a) T_0 does. b) T_0 overwrites forwarded data with new value.

implementation of CS
(LogTM, TCC, RTM, MetaTM, OneTM....)

transactions, and $active(h)$ the set of active (not committed or aborted) transactions. If x is a variable, $prev(x, h)$ is the set of active transactions that have read or written x in h , $active(x, h)$ the active or committed transactions that most recently wrote x in h , and $val(x, h)$ the value x wrote.

Definition A.1. A history h is atomic if $committed(h)$ is equivalent to a legal failure-free serial history.

Let $serial(h)$ be the serial history equivalent to $committed(h)$ in which transactions appear in the order of their commit events in h .

A consistency control mechanism can be thought of as an automaton that accepts concurrent histories. The mechanism is correct if these histories are atomic. We now define dependence-aware transactional summary as an automaton that accepts atomic histories. The automaton keeps the following state. The history h is the history accepted so far. For each T , we keep track of $active(T)$, the set of transactions that must commit before T can commit. We also keep track of $notActive(T)$, the set of transactions that must commit or abort before T can commit. Transactions are given by pre- and post-conditions, where X^+ denotes the new state of component X .

An active transaction can always abort:

- Pre: $T \in active(h)$.
- Post: $T \in h$, $T \notin active(h)$.

This transition captures the effects of deadlock detection, either event or timeout (that is, timeout).

When a transaction T reads, we track the write and dependency on the transaction whose value it read.

- Pre: $T \in active(h)$.
- Post: $T \in active(h)$.
- Pre: $W \in h$, $W \in read(active(h))$.
- $active(T) := active(T) \cup \{active(w, h)\}$.
- When a transaction T reads, we track its read-write and write-write dependencies on the active transactions that read or wrote that variable.
- Pre: $T \in active(h)$.
- Post: and also if $T \in active(h)$.
- $W \in h$, $W \in read(active(h))$.
- $notActive(T) := notActive(T) \cup \{prev(x, h)\}$.
- A transaction can commit only if every value it read was written by a non-committed transaction, and every value it wrote was previously read or written by a non-committed or non-aborted transaction.
- Pre: $T \in active(h)$.
- $active(T) \subseteq committed(h) \cup \{T\}$.
- $notActive(T) \subseteq committed(h) \cup \{aborted(h) \cup \{T\}$.
- Post: $h \in h$, $T \in committed$.

The dependence-aware algorithm satisfies this simple invariant: the committed transactions, modification does not in-order read and writes to the same variable.

Lemma A.1. Let u_0 be a write event by T_0 , and v_1 either a read or write event by T_1 , both to a variable x in h . If u_0 precedes v_1 in $serial(h)$, then u_0 precedes v_1 in h .

Proof. Suppose instead that v_1 precedes u_0 in h . Because they were incident, T_0 and T_1 were both active when u_0 was appended to history h . It follows that $T_1 \in prev(x, h)$, so $T_1 \in notActive(T_0)$, implying that T_1 must commit before T_0 , making v_1 before u_0 in $serial(h)$, a contradiction. \square

Lemma A.2. Let h be a history containing u_0 , a write event by T_0 , v_1 , a read event by T_1 , that returns the value written by u_0 , and v_0 , a write event by T_0 that follows u_0 in h . We claim that $T_1 \in notActive(T_0)$.

Proof. Because v_1 returns the value written by u_0 , there are no writes between u_0 and v_1 . Therefore v_0 follows v_1 in h , and the claim is established when v_1 is appended to the history. \square

If v_1 is a read event in h , let $readFrom(x)$ be the write event whose value v_1 returns. That is, for every read event v in $serial(h)$,

- $readFrom(x, serial(h)) = readFrom(x, h)$.

The property holds vacuously in the original case. Suppose, by way of contradiction, the property becomes violated at some step. That step must be the commit of a transaction T_0 , because the other step leaves $serial(h)$ unchanged. Let u_0 be a read step in h/T_0 that violates the property, let $v_0 = readFrom(x, h)$, and let T_1 be the transaction that executed v_0 .

First, v_0 must be in $serial(h)$ because $T_1 \in active(T_0)$, and a precondition for T_0 to commit is that T_1 be committed. Second, there can be no v_0 by T_1 between u_0 and v_0 in $serial(h)$. If v_0 occurs after u_0 in $serial(h)$, then by Lemma A.1, v_0 occurs after v_0 in h , so by Lemma A.2, $T_1 \in notActive(T_0)$, implying that T_1 could not have committed after T_0 .

It follows that when a transaction commits, the new serial history is legal, because every read event returns the value written by the most recent write event in $serial(h)$.

We have shown that our implementation is safe: all histories accepted are atomic. We now show on a stronger claim: this automaton accepts all conflict-serializable [11] histories. By contrast, most TM systems, whether hardware or software, accept only those histories simulated by a single-phase locking a strictly smaller set.

Some care is needed when interpreting this claim. As noted, no active transaction can be aborted at any time, for any reason. In practice, an implementation will abort a transaction only if it detects, or suspects, a deadlock resulting from a cyclic dependency. Our automaton accepts all conflict-serializable histories, but an actual implementation

Outline

- Motivation/Background
- Dependence-Aware Model
- Dependence-Aware Hardware
- Experimental Methodology/Evaluation
- Conclusion

DATM Requirements

Mechanisms:

- Maintain global dependence graph
 - Conservative approximation
 - Detect cycles, enforce ordering constraints
- Create dependences
- Forwarding/receiving
- Buffer speculative data

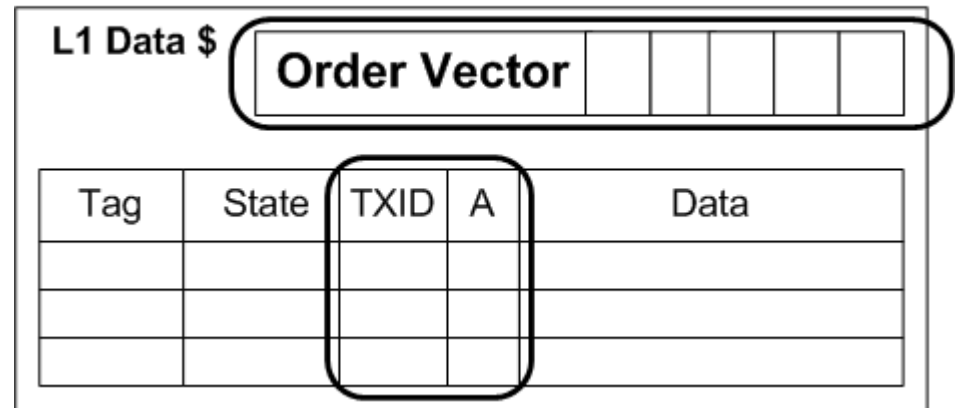
Implementation:

coherence, L1, per-core HW structures

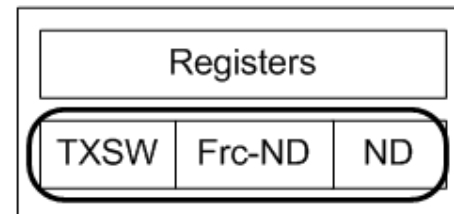
Dependence-Aware Microarchitecture

- **Order vector:**
dependence graph
- **TXID**, access bits:
versioning, detection
- **TXSW**: transaction status
- **Frc-ND + ND** bits:
disable dependences

L1 Cache Controller



Processor Core

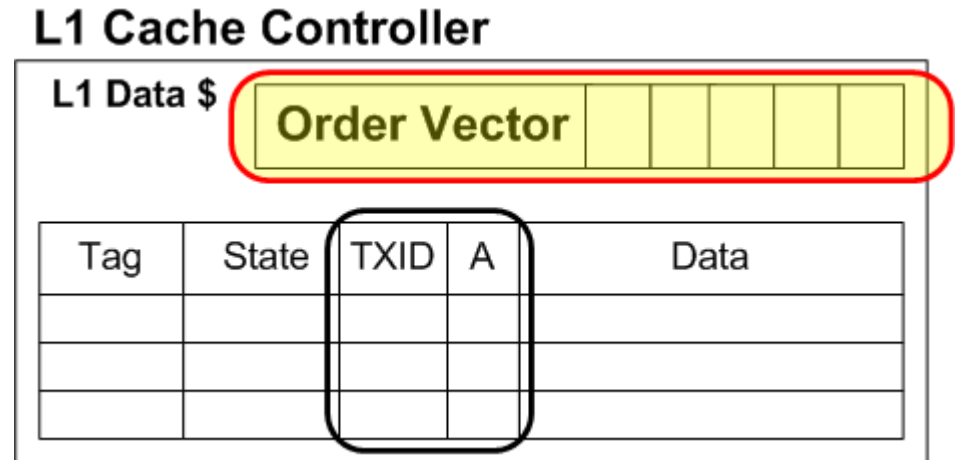


These are not programmer-visible!

Dependence-Aware Microarchitecture

Order vector:

- dependence graph
- topological sort
- conservative approx.



Processor Core

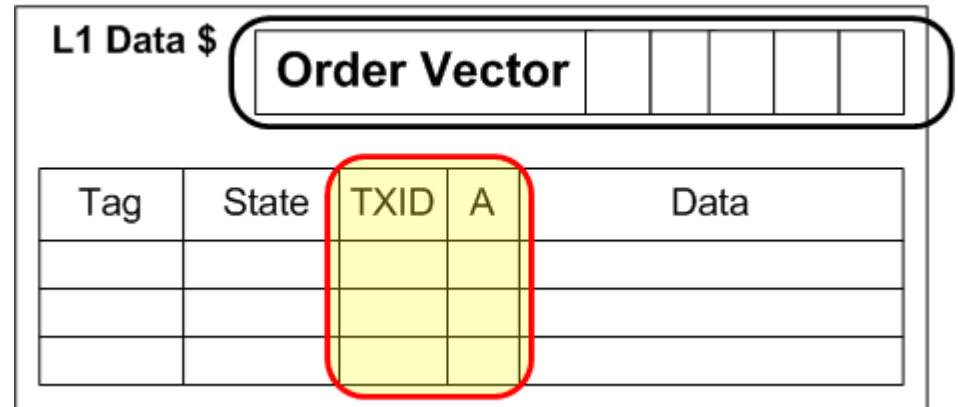


These are not programmer-visible!

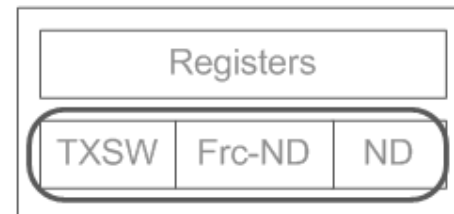
Dependence-Aware Microarchitecture

- **Order vector:**
dependence graph
- **TXID**, access bits:
versioning, detection
- **TXSW**: transaction status
- **Frc-ND + ND** bits: disable
dependences

L1 Cache Controller



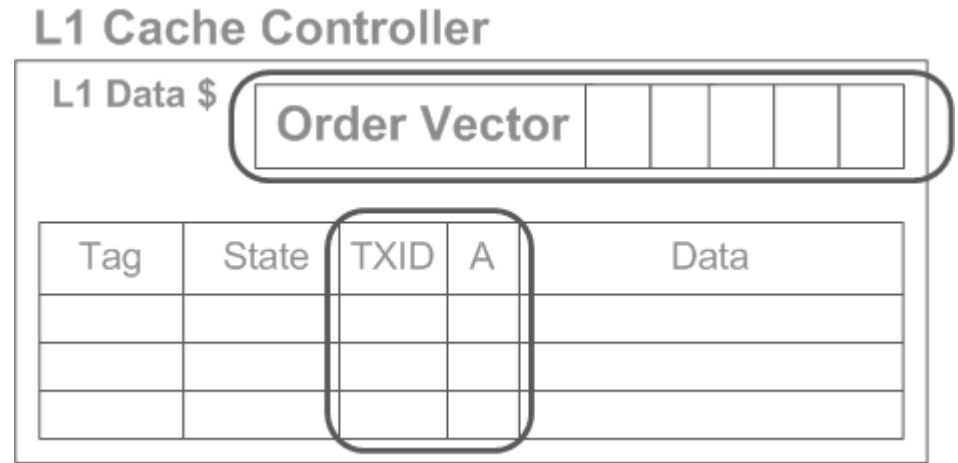
Processor Core



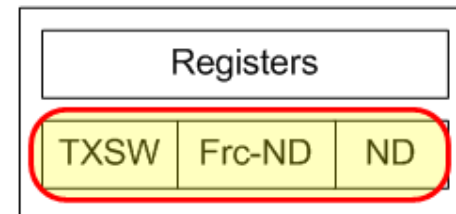
These are not programmer-visible!

Dependence-Aware Microarchitecture

- **Order vector:**
dependence graph
- **TXID**, access bits:
versioning, detection
- **TXSW**: transaction status
- **Frc-ND + ND** bits:
disable dependences, no
active dependences



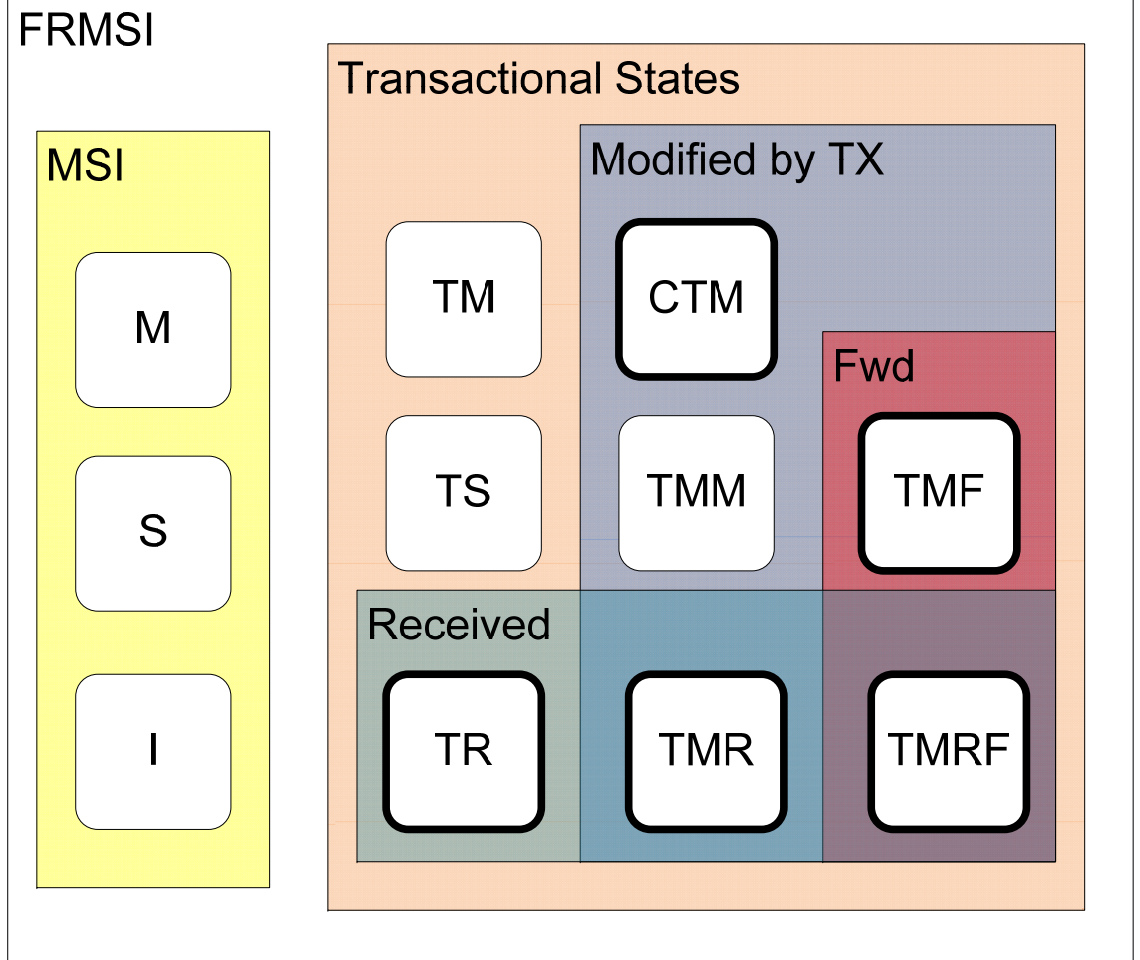
Processor Core



These are not programmer-visible!

FRMSI protocol: forwarding and receiving

- MSI states
- TX states (T^*)
- Forwarded: (T^*F)
- Received: (T^*R^*)
- Committing (CTM)
- Bus messages:
 - **TXOVW**
 - **xABT**
 - **xCMT**



FRMSI protocol: forwarding and receiving

- **MSI states**

- TX states (T^*)
- Forwarded: (T^*F)
- Received: (T^*R^*)
- Committing (CTM)
- Bus messages:
 - TXOVW
 - xABT
 - xCMT

FRMSI

MSI

M

S

I

Transactional States

Modified by TX

TM

CTM

TS

TMM

Fwd

TMF

Received

TR

TMR

TMRF

FRMSI protocol: forwarding and receiving

- MSI states
- **TxMSI states (T*)**
- Forwarded: (T*F)
- Received: (T*R*)
- Committing (CTM)
- Bus messages:
 - TXOVW
 - xABT
 - xCMT

FRMSI

MSI

M

S

I

Transactional States

Modified by TX

TM

CTM

TS

TMM

Fwd

TMF

Received

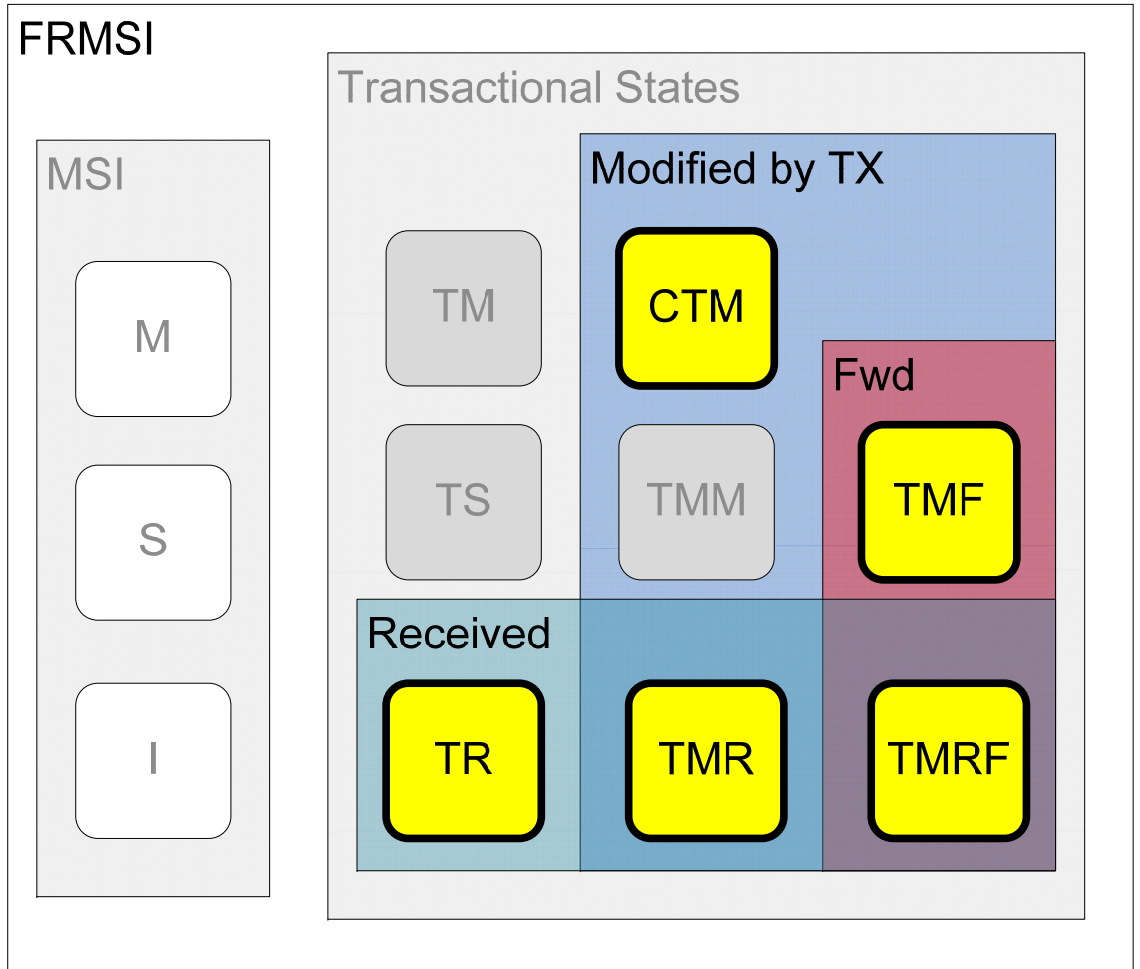
TR

TMR

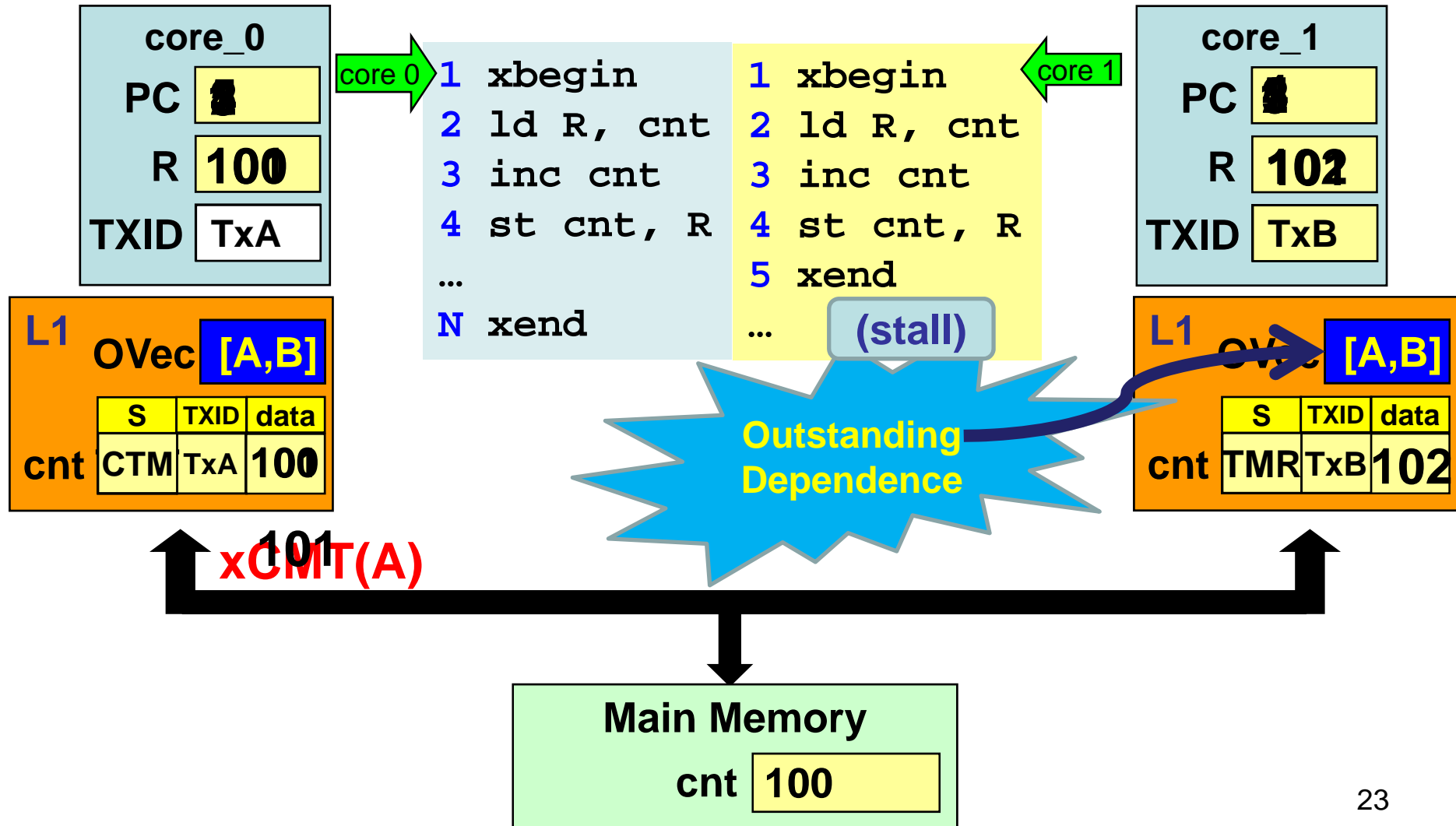
TMRF

FRMSI protocol: forwarding and receiving

- MSI states
- TX states (T^*)
- **Forwarded: (T^*F)**
- **Received: (T^*R^*)**
- **Committing (CTM)**
- Bus messages:
 - TXOVW
 - xABT
 - xCMT



Converting Conflicts to Dependences



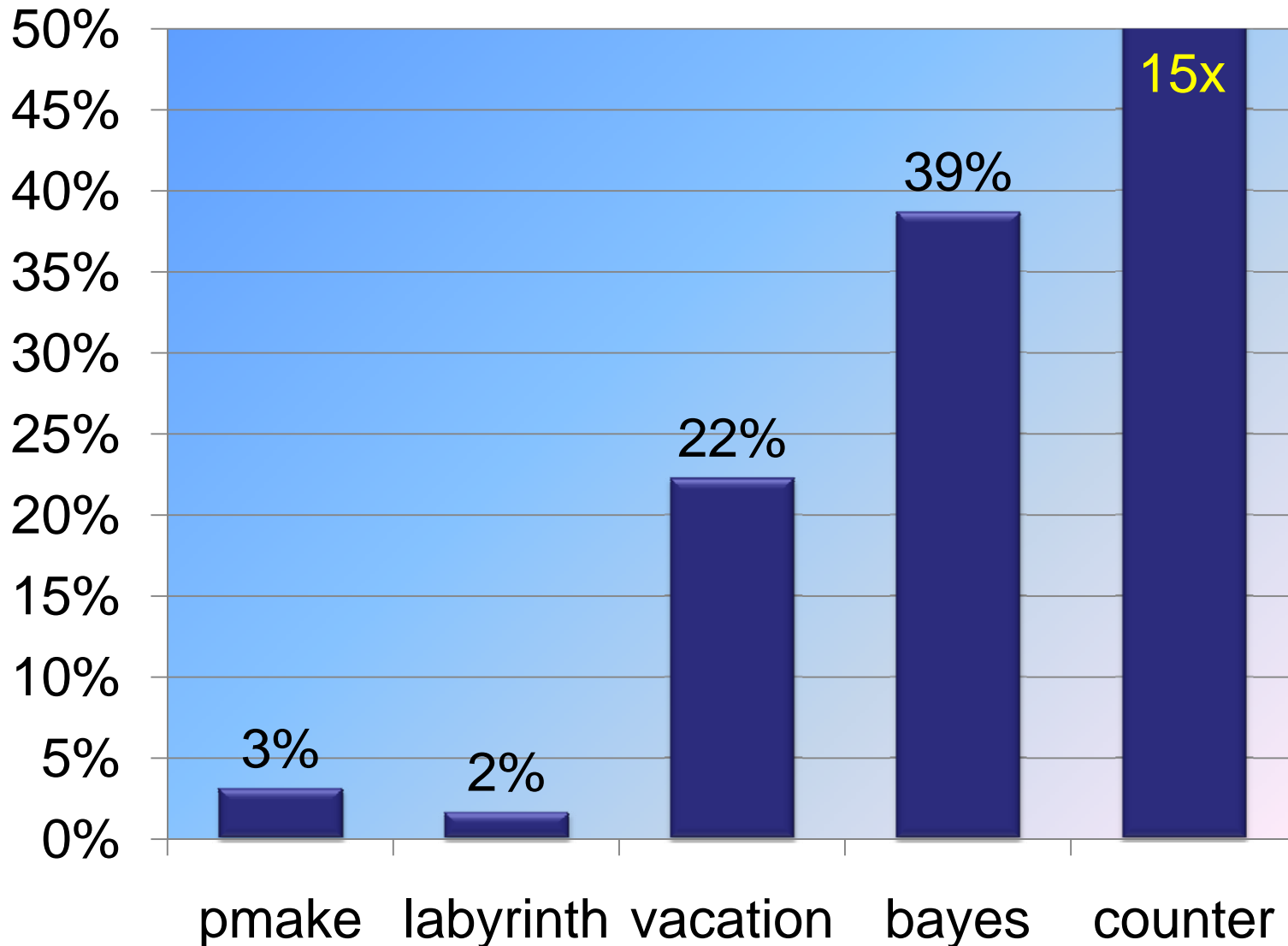
Outline

- Motivation/Background
- Dependence-Aware Model
- Dependence-Aware Hardware
- **Experimental Methodology/Evaluation**
- **Conclusion**

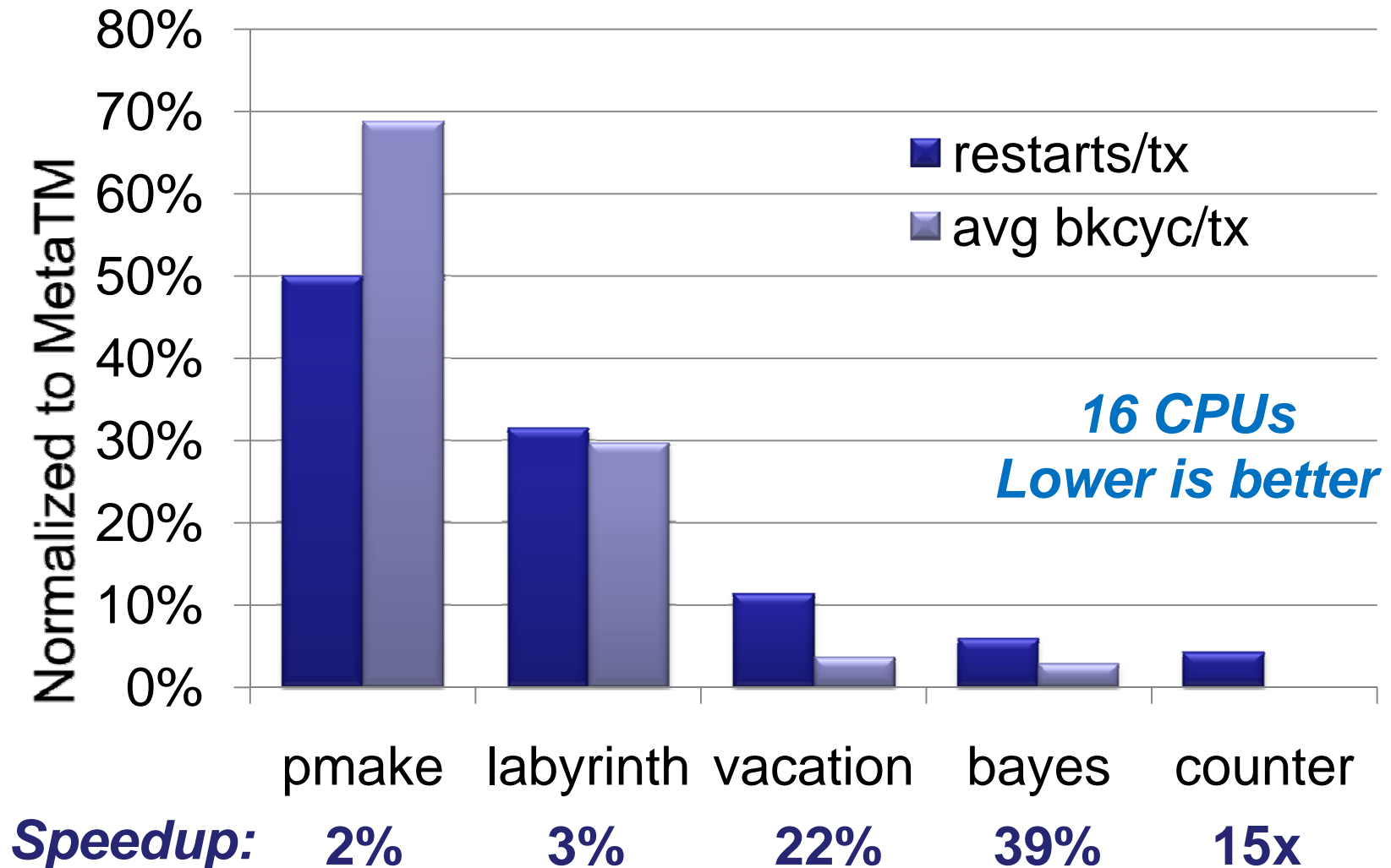
Experimental Setup

- Implemented DATM by extending MetaTM
 - Compared against MetaTM: [Ramadan 2007]
- Simulation environment
 - Simics 3.0.30 machine simulator
 - x86 ISA w/HTM instructions
 - 32k 4-way tx L1 cache; 4MB 4-way L2; 1GB RAM
 - 1 cycle/inst, 24 cyc/L2 hit, 350 cyc/main memory
 - 8, 16 processors
- Benchmarks
 - **Micro-benchmarks**
 - **STAMP [Minh ISCA 2007]**
 - **TxLinux: Transactional OS [Rossbach SOSP 2007]**

DATM speedup, 16 CPUs

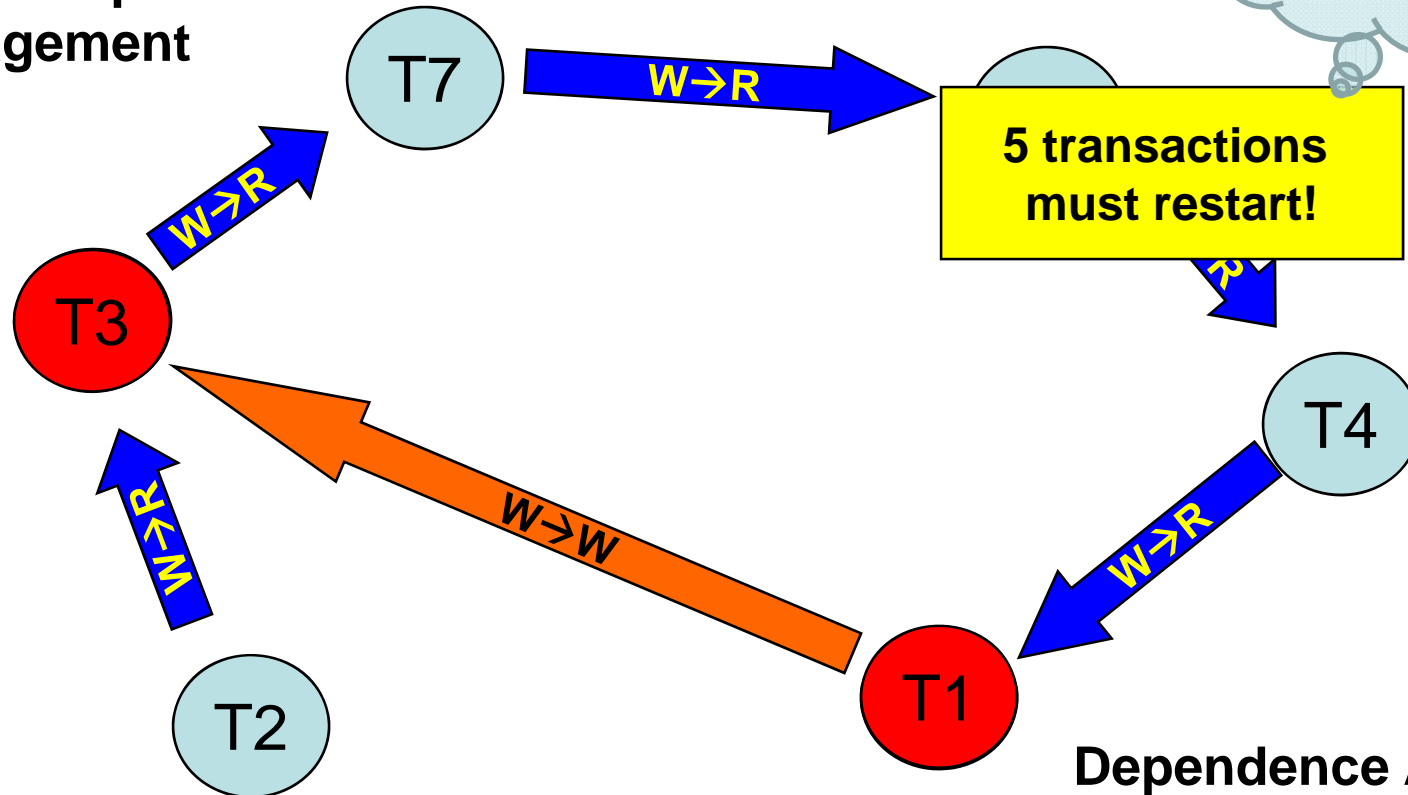


Eliminating wasted work



Dependence Aware Contention Management

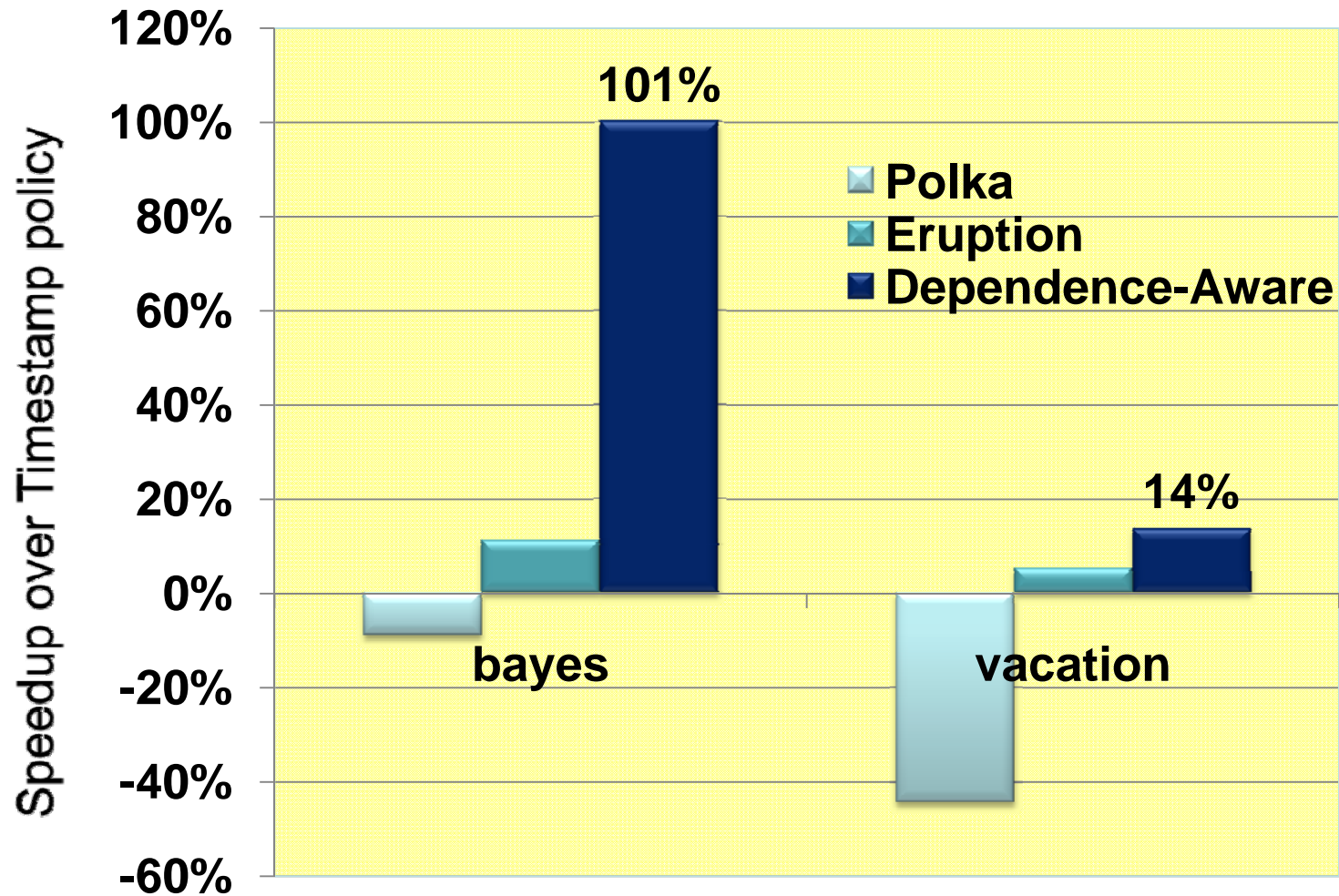
Timestamp contention management



Dependence Aware

Order Vector T2, T3, T7, T6, T4

Contention Management



Outline

- Motivation/Background
- Dependence-Aware Model
- Dependence-Aware Hardware
- Experimental Methodology/Evaluation
- **Conclusion**

Related Work

- **HTM**
 - TCC [Hammond 04], LogTM[-SE] [Moore 06], VTM [Rajwar 05], MetaTM [Ramadan 07, Rossbach 07], HASTM, PTM, HyTM, RTM/FlexTM [Shriraman 07,08]
- **TLS**
 - Hydra [Olukotun 99], Stampede [Steffan 98,00], Multiscalar [Sohi 95], [Garzaran 05], [Renau 05]
- **TM Model extensions**
 - Privatization [Spear 07], early release [Skare 06], escape actions [Zilles 06], open/closed nesting [Moss 85, Menon 07], Galois [Kulkarni 07], boosting [Herlihy 08], ANTs [Harris 07]
- **TM + Conflict Serializability**
 - Adaptive TSTM [Aydonat 08]

Conclusion

- DATM can commit conflicting transactions
- Improves write-sharing performance
- Transparent to the programmer
- DATM prototype demonstrates performance benefits

Source code available!
www.metatm.net 32

Broadcast and Scalability

**Yes.
We broadcast.
But it's less than you might
think...**

**Because each node maintains
all deps, this design uses
broadcast for:**

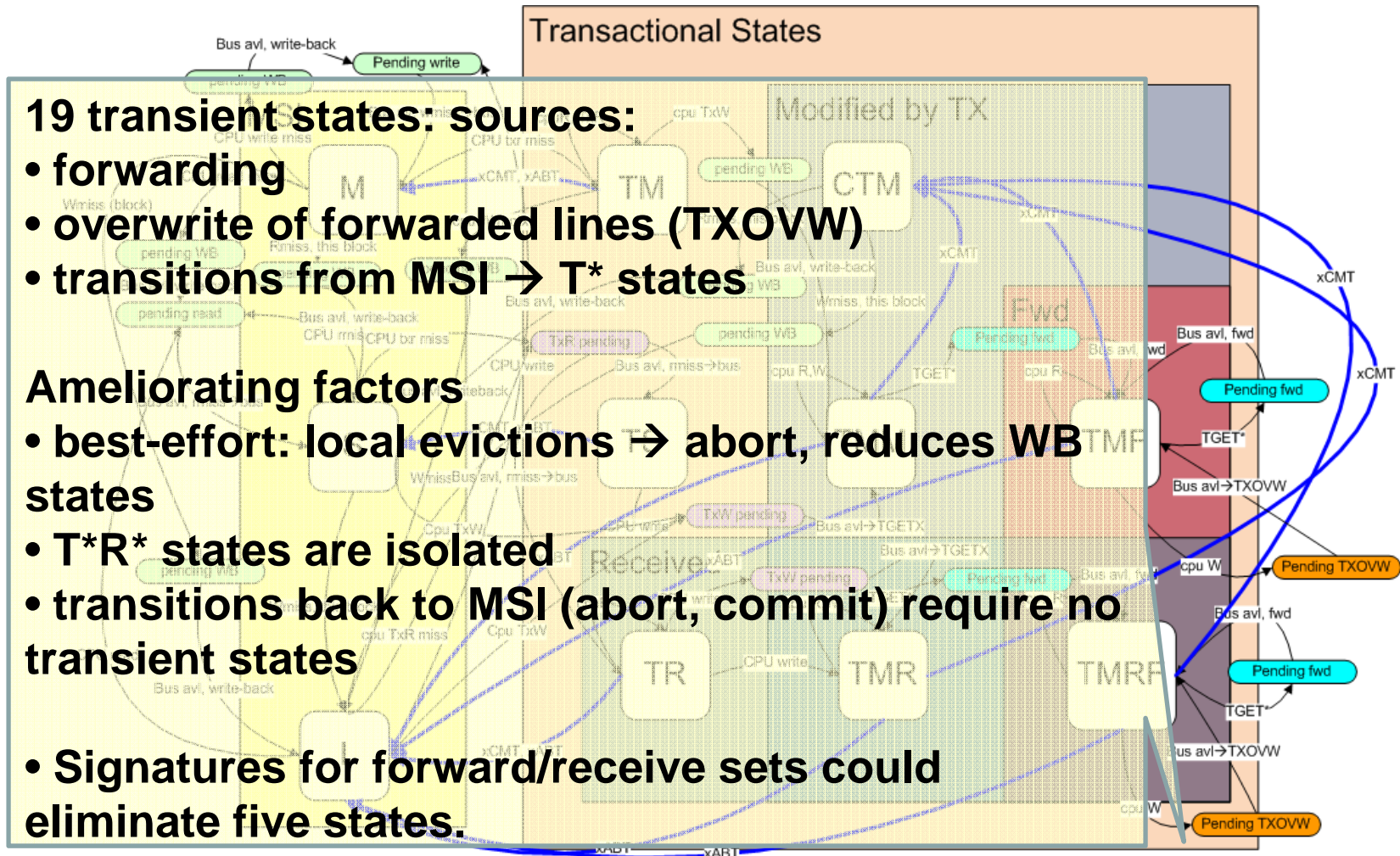
- **Commit**
- **Abort**
- **TXOVW (broadcast writes)**
- **Forward restarts**
- **New Dependences**

benchmark	tx	broadcast w	forw rest
bayes	762	1	0.4%
config(8p)	4698136	10,132	19.7%
counter	160000	0	0%
counter-tt	16000	0	0%
genome	352376	104	0%
kmeans	436986	40,723	0%
labyrinth	128	4	0%
list(8p)	78586	86	3.3%
pmake(8p)	251844	10,009	12.9%
ssca2	47304	0	0%
vacation	20000	143	0.4%













**These sources of broadcast
could be avoided in a directory
protocol:**

**keep only the relevant subset of the
dependence graph at each node**

FRMSI Transient states



Why isn't this TLS?

	TLS	DATM	TM
Forward data between threads			
Detect when reads occur too early			
Discard speculative state after violations			
Retire speculative Writes in order			
Memory renaming			
Multi-threaded workloads			
Programmer/Compiler transparency			

1. Txns can commit in arbitrary order. TLS has the daunting task of maintaining program order for epochs
2. TLS forwards values when they are the globally committed value (i.e. through memory)
3. No need to detect “early” reads (before a forwardable value is produced)
4. Notion of “violation” is different—we must roll back when CS is violated, TLS, when epoch ordering
5. Retiring writes in order: similar issue—we use CTM state, don't need speculation write buffers. ³⁵
6. Memory renaming to avoid older threads seeing new values—we have no such issue

Doesn't this lead to cascading aborts?

	WR deps	RW deps	forward restarts	cascade aborts
bayes	3.8%	8.5%	0.4%	0%
config (8p)	0.3%	0.2%	19.7%	2.3%
counter	90.0%	80.7%	0%	0%
counter-tt	99.9%	0.3%	0%	100.0%
genome	0.1%	0.1%	0%	14.2%
kmeans	8.9%	6.0%	0%	3.1%
labyrinth	32.0%	32.0%	0%	0.1%
list	14.3%	3.8%	3.3%	0.2%
pmake (8p)	0.5%	0.5%	12.9%	6.5%
ssca2	0.1%	0.1%	0%	0%
vacation	35.2%	7.9%	0.4%	3.5%

Rare in most workloads

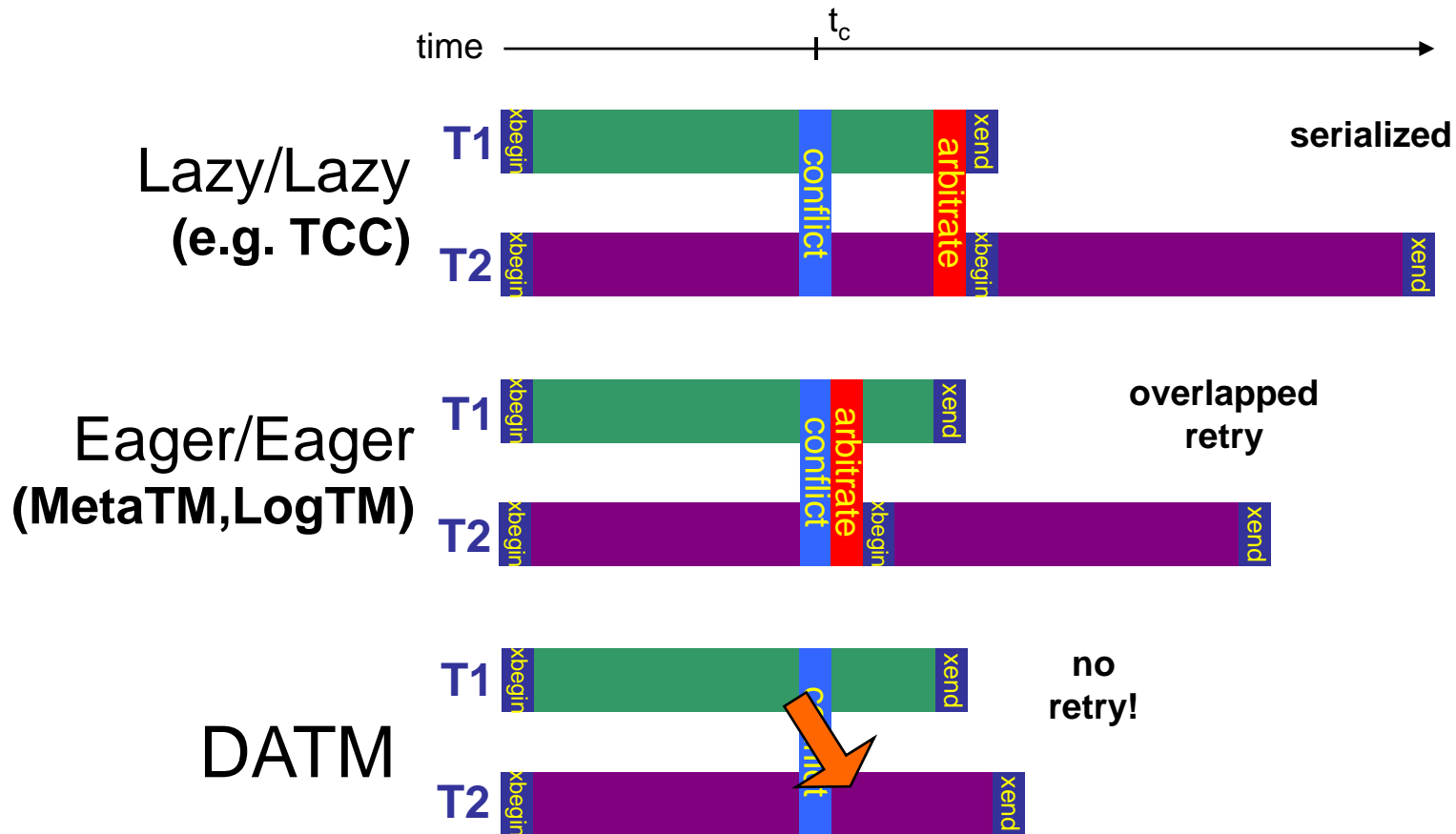
Inconsistent Reads

	incons. reads
bayes	3
config (8p)	1
counter	0
counter-tt	0
genome	1
kmeans	0
labyrinth	1
list	0
pmake (8p)	10
ssca2	0
vacation	34

OS modifications:

- **Page fault handler**
- **Signal handler**

Increasing Concurrency

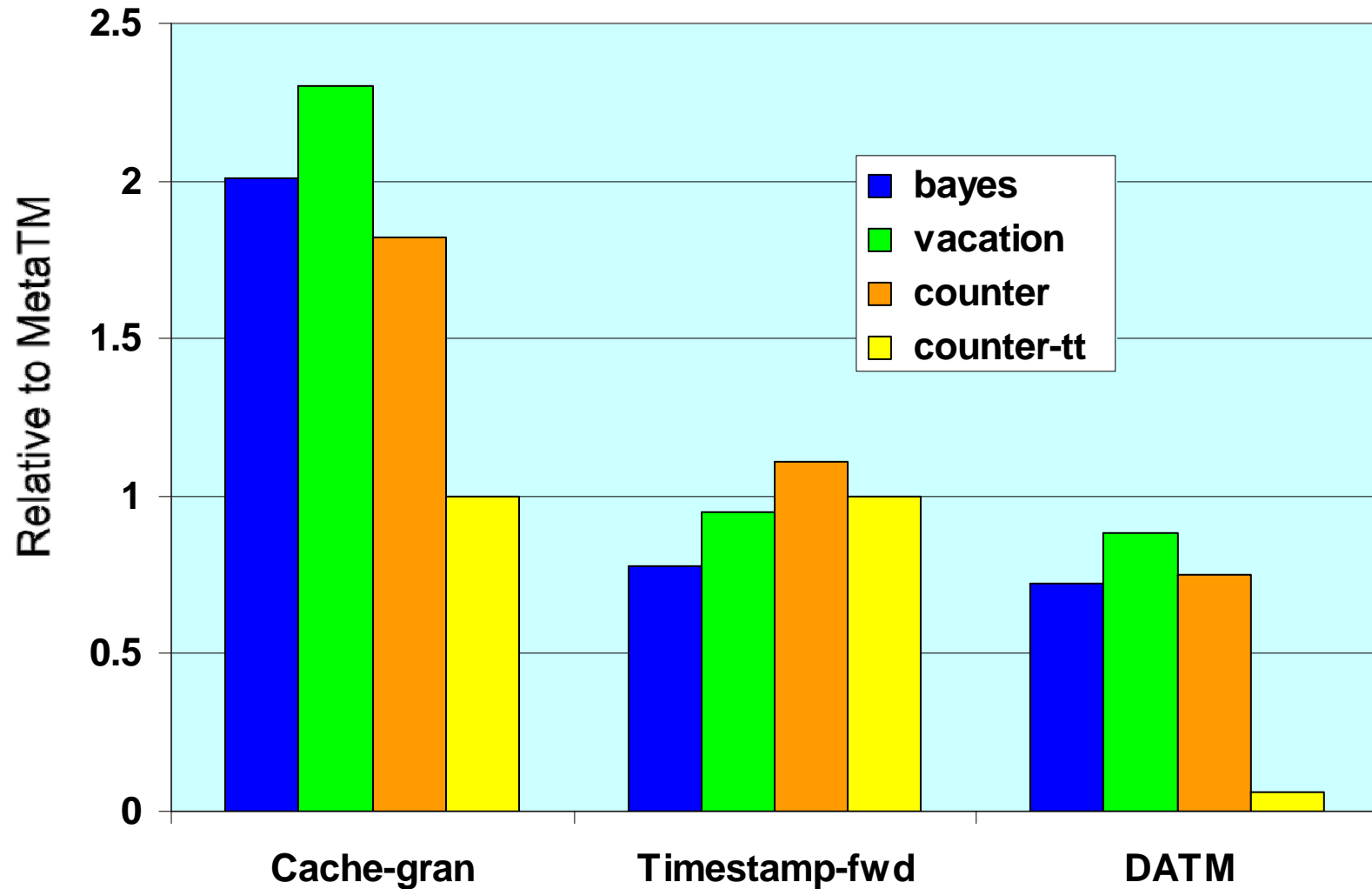


(Eager/Eager: Updates buffered, conflict detection at commit time)

Design space highlights

- Cache granularity:
 - eliminate per word access bits
- Timestamp-based dependences:
 - eliminate Order Vector
- Dependence-Aware contention management

Design Points



Hardware TM Primer

Key Ideas:

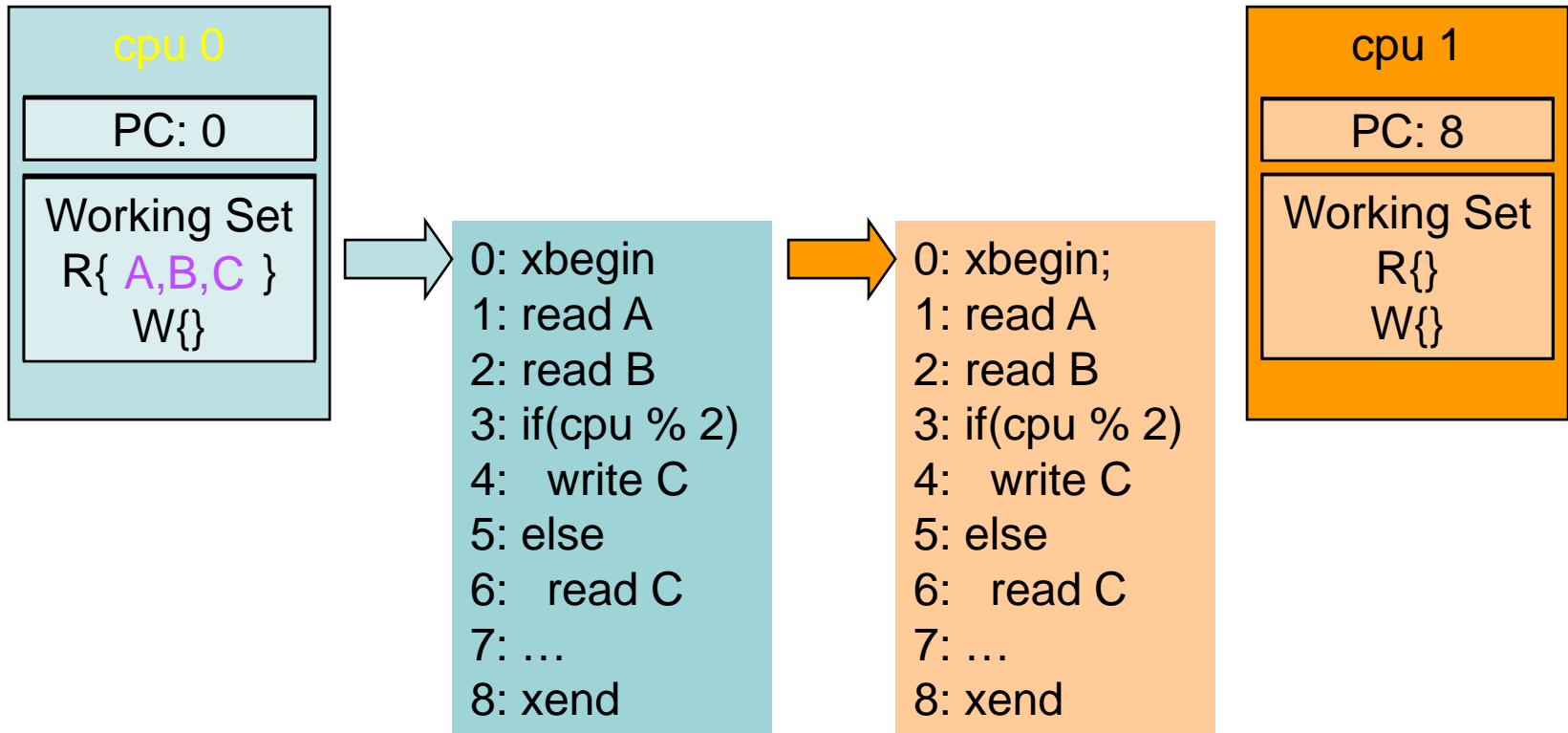
- ▶ Critical sections execute concurrently
- ▶ Conflicts are detected dynamically
- ▶ If conflict serializability is violated, rollback

Key Abstractions:

- Primitives
 - **xbegin, xend, xretry**
- Conflict
$$\emptyset \neq \{W_a\} \cap \{R_b \cup W_b\}$$
- Contention Manager
 - Need flexible policy

“Conventional Wisdom Transactionalization”:
Replace locks with transactions

Hardware TM basics: example



CONFLICT
Assume Contention
manager decides cpu1
wins:
C is in the read set of
cpu0, and in the write
set of cpu1
cpu0 rolls back
cpu1 commits