

# Transactional Memory

Chris Rossbach and Calvin Lin

cs380p

# Outline

Background

Transactional Memory

*Acknowledgements: Yoav Cohen for some STM slides*

# Transactional Memory

## 3 Programming Model Dimensions:

How to specify computation

How to specify communication

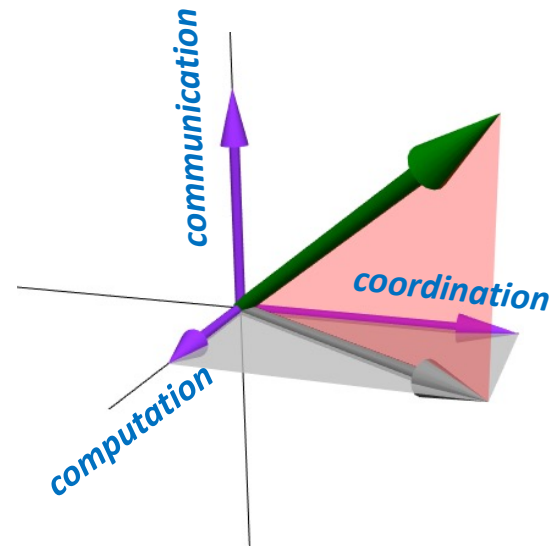
How to specify coordination/control transfer

Threads, Futures, Events etc.

*Mostly about how to express control*

Transactional Memory

*Shared state: synchronization through memory*



# TM: Motivation

- Threads/Locks have a *\*lot\** of down-sides:
  - Tuning parallelism for different environments
  - Load balancing/assignment brittle
  - Shared state requires locks →
    - Priority inversion
    - Deadlock
    - Incorrect synchronization
  - ...
- TM: *restructure programming model* → *no locks!*



# Transactional Memory: ACI

Transactional Memory :

Make multiple memory accesses atomic

All or nothing – Atomicity

No interference – Isolation

Correctness – Consistency

No durability, for obvious reasons

Keywords : Commit, Abort,

Speculative access, Checkpoint

```
remove(list, x) {  
    lock(list);  
    pos = find(list, x);  
    if(pos)  
        erase(list, pos);  
    unlock(list);  
}
```

```
remove(list, x) {  
    TXBEGIN();  
    pos = find(list, x);  
    if(pos)  
        erase(list, pos);  
    TXEND();  
}
```

# The Real Goal

```
remove(list, x) {  
  atomic {  
    pos = find(list, x);  
    if(pos)  
      erase(list, pos);  
  }  
}
```

- Transactions: super-awesome
- TM: also super-awesome, **but**:
  - Transactions != TM
  - TM → **implementation technique**
  - Often presented as programmer abstraction

```
remove(list, x) {  
  lock(list);  
  pos = find(list, x);  
  if(pos)  
    erase(list, pos);  
  unlock(list);  
}
```

```
remove(list, x) {  
  TXBEGIN();  
  pos = find(list, x);  
  if(pos)  
    erase(list, pos);  
  TXEND();  
}
```

# A Simple TM

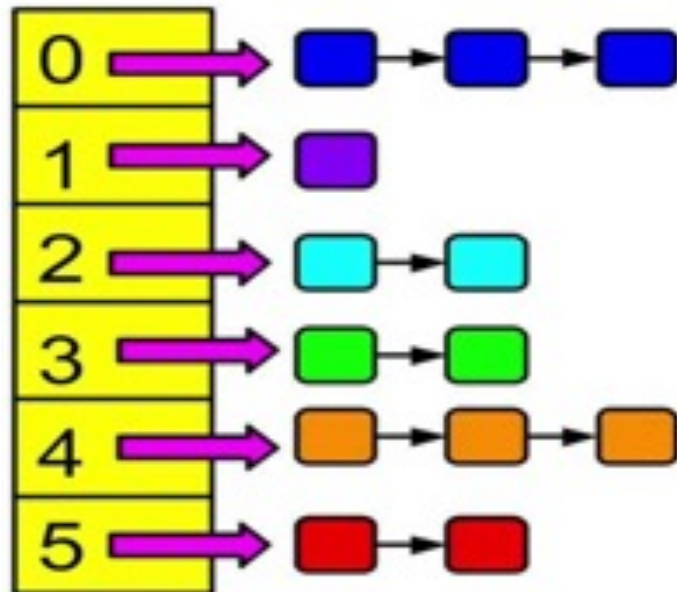
```
pthread_mutex_t g_global_lock;  
  
begin_tx() {  
    pthread_mutex_lock(g_global_lock);  
}  
  
end_tx() {  
    pthread_mutex_unlock(g_global_lock);  
}  
  
abort() {  
    // can't happen  
}
```

```
remove(list, x) {  
    begin_tx();  
    pos = find(list, x);  
    if(pos)  
        erase(list, pos);  
    end_tx();  
}
```

Actually, this works fine...  
But how can we improve it?

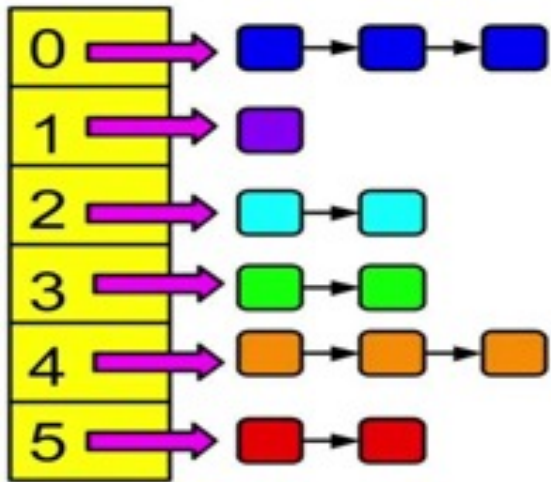
# Concurrency Control Revisited








Consider a hash-table



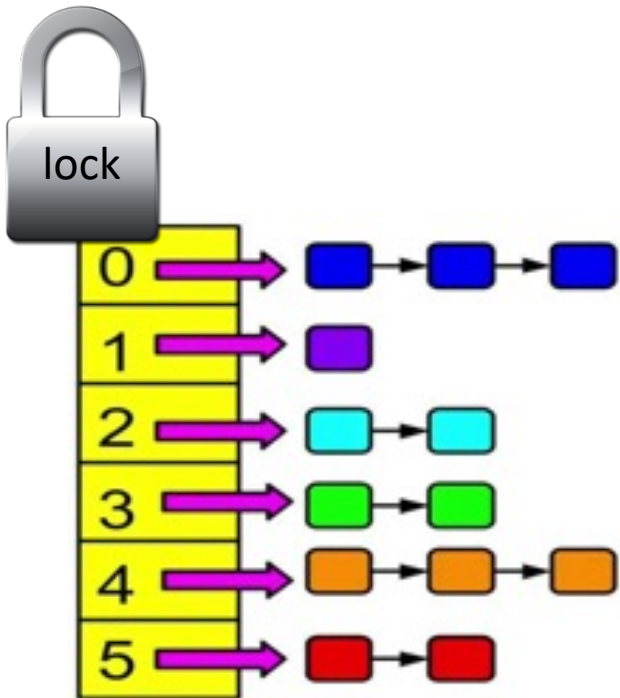








# Concurrency Control Revisited



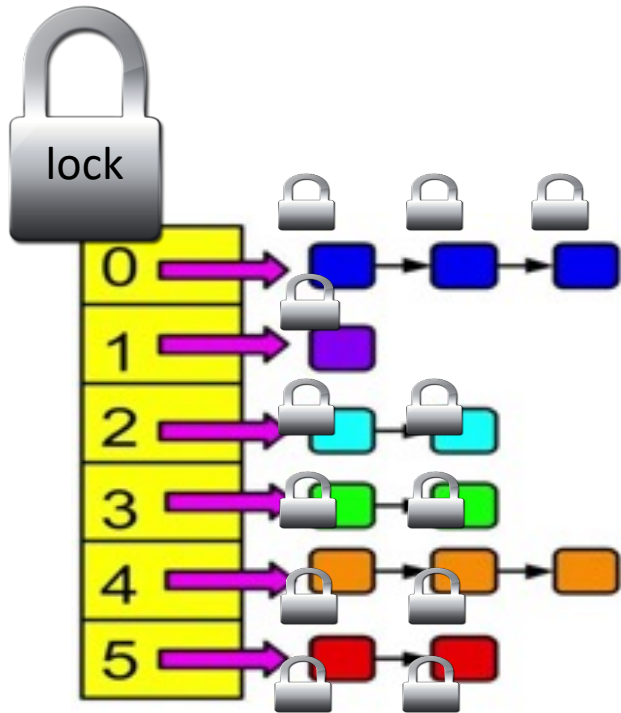
thread T1	thread T2
<pre>ht.add();</pre>	<pre>ht.add();</pre>
<pre>if(ht.contains())</pre>	<pre>if(ht.contains())</pre>
<pre>ht.del();</pre>	<pre>ht.del();</pre>
	







# Concurrency Control Revisited



thread T1	thread T2
<pre>ht.lock(); ht.add();  if(ht.contains())  ht.del(); ht.unlock();</pre>	<pre>ht.lock(); ht.add();  if(ht.contains()) ht.del(); ht.unlock();</pre>

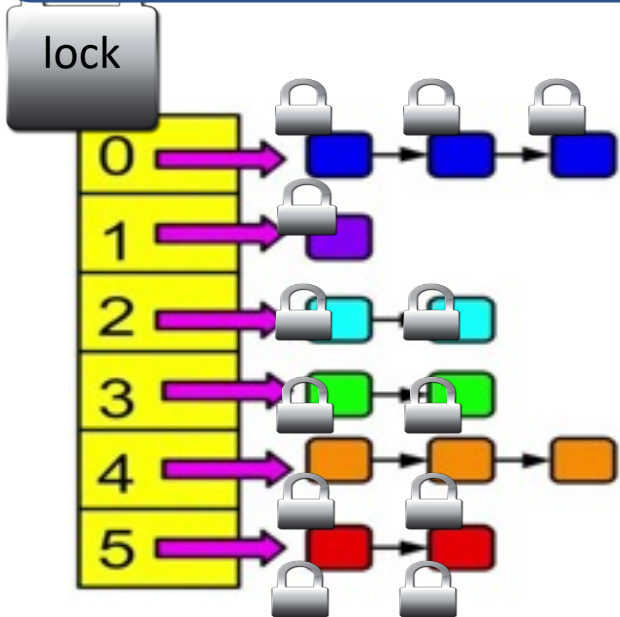
# Pessimistic concurrency control









thread T1	thread T2
<pre>ht.lock(); ht.add();  if(ht.contains())   ht.del(); ht.unlock();</pre>	<pre>ht.lock(); ht.add();  if(ht.contains())   ht.del(); ht.unlock();</pre>

# Optimistic concurrency control

What do we do when same data is accessed?



thread T1	thread T2
<pre>ht.lock(); ht.add();  if(ht.contains())   ht.del(); ht.unlock();</pre>	<pre>ht.lock(); ht.add();  if(ht.contains())   ht.del(); ht.unlock();</pre>

# TM Primer

## Key Ideas:

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

## Key Abstractions:

Primitives

**xbegin, xend, xabort**

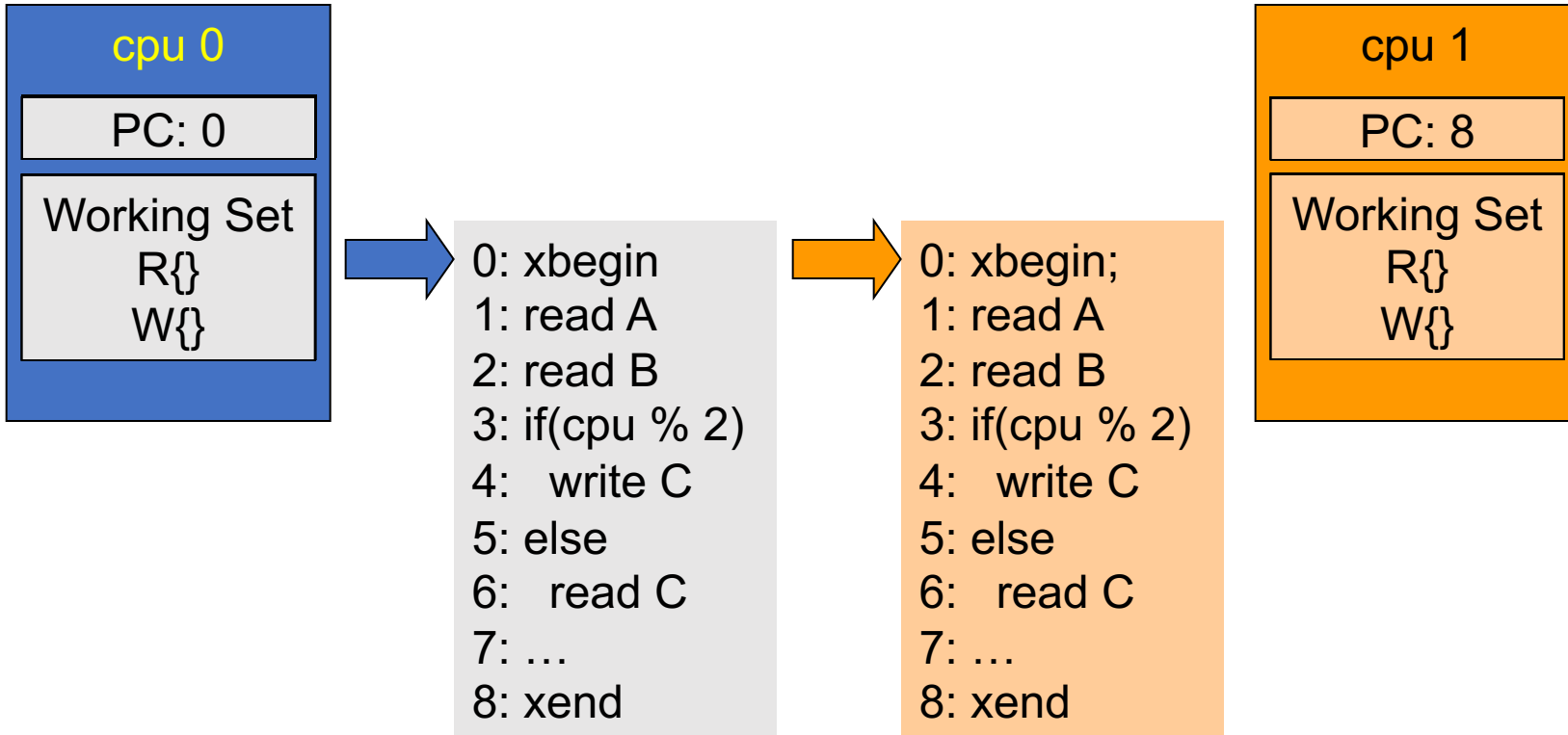
Conflict

$$\emptyset \neq \{W_a\} \cap \{R_b \cup W_b\}$$

Contention Manager

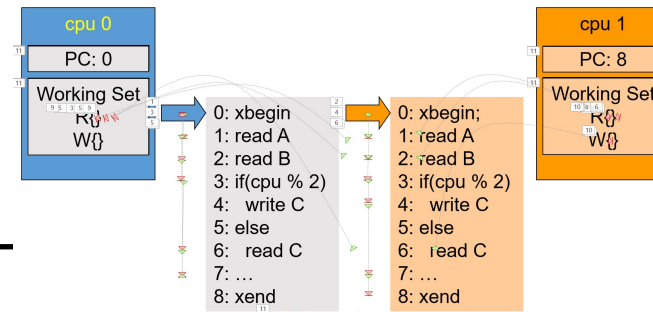
Need flexible policy

# 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

# TM Implementation



## Data Versioning

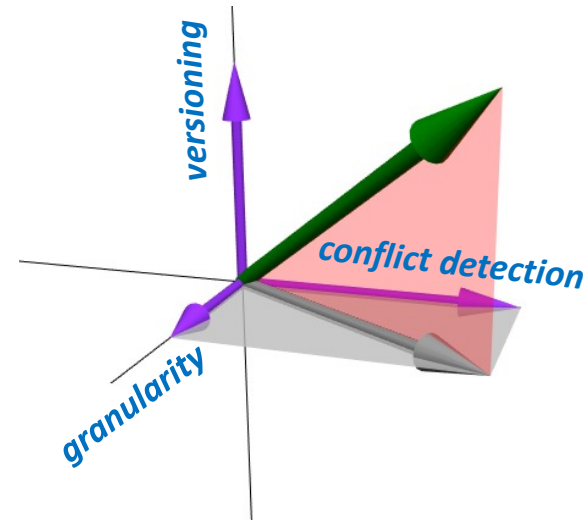
- *How to manage uncommitted state?*
- Eager Versioning
- Lazy Versioning

## Conflict Detection and Resolution

- *How to tell when same data are touched?*
- Pessimistic Concurrency Control
- Optimistic Concurrency Control

## Conflict Detection Granularity

- *What is the unit of protected state?*
- Object Granularity
- Word Granularity
- Cache line Granularity



# TM Design Alternatives

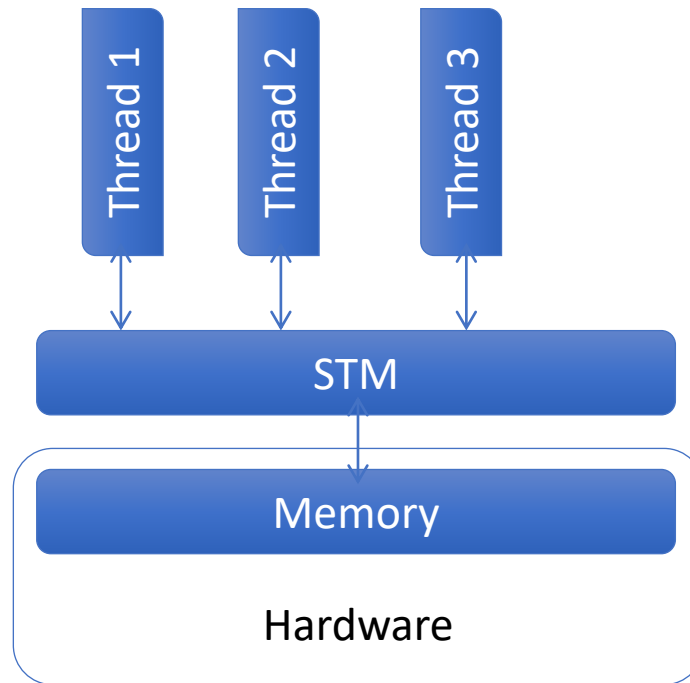
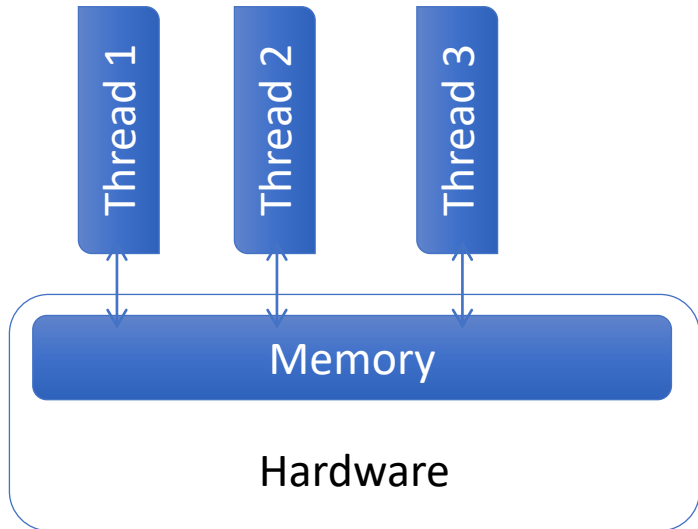
## Hardware (HTM)

Caches track RW set, HW speculation/checkpoint

## Software (STM)

Instrument RW

Inherit TX Object





# Hardware Transactional Memory

Idea: Track read / write sets in HW  
commit / rollback in hardware as well

Cache coherent hardware already manages much of this

Basic idea: cache == speculative storage

HTM ~= smarter cache

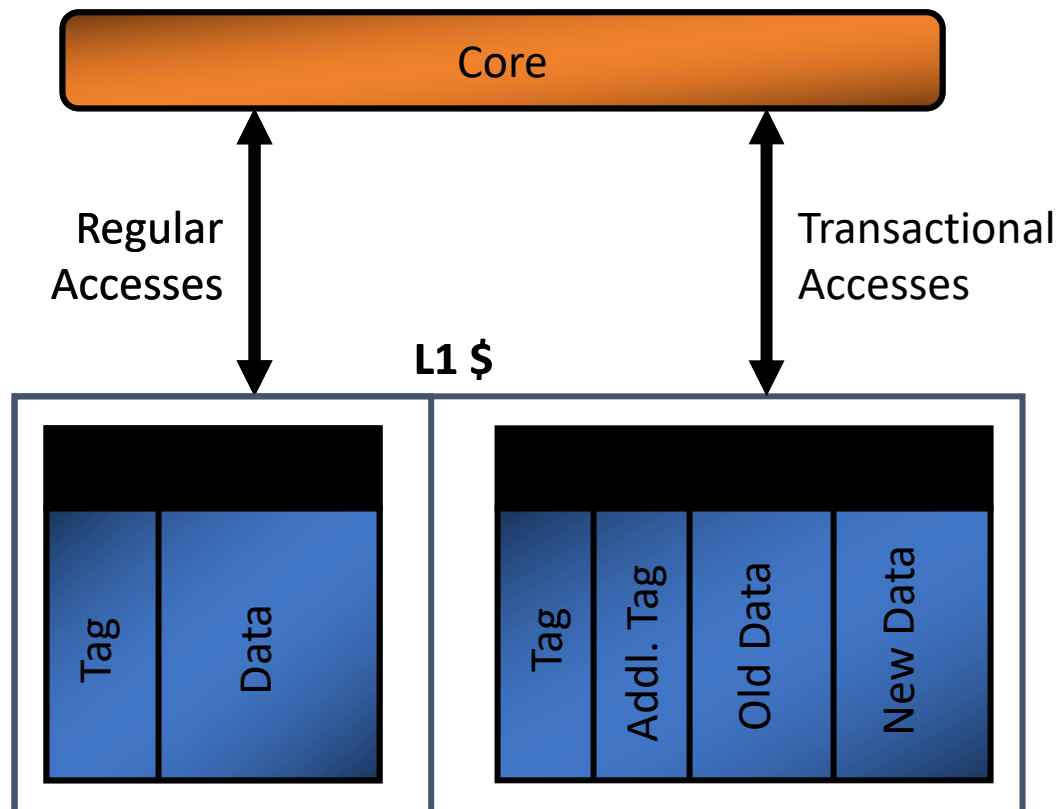
Can support many different TM paradigms

Eager, lazy

optimistic, pessimistic

# Hardware TM

“Small” modification to cache

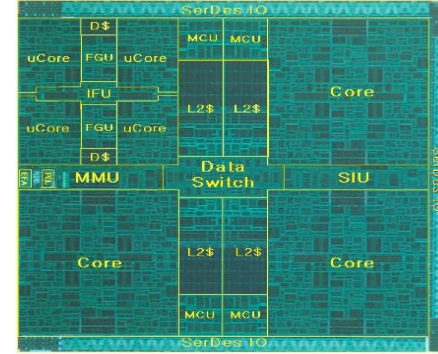


## Key ideas

- *Checkpoint architectural state*
- *Caches: ‘versioning’ for memory*
- *Change coherence protocol*
- *Conflict detection in hardware*
- *‘Commit’ tx if no conflict*
- *‘Abort’ on conflict*
- *‘Retry’ aborted transaction*

Pros/Cons?

# Case Study: SUN Rock



Major challenge: diagnosing cause of Transaction aborts

Necessary for intelligent scheduling of transactions

Also for debugging code

debugging the processor architecture /  $\mu$ architecture

Many unexpected causes of aborts

Rock v1 diagnostics unable to distinguish distinct failure modes

Mask	Name	Description and example cause
0x001	EXOG	<b>Exogenous</b> - Intervening code has run: cps register contents are invalid.
0x002	COH	<b>Coherence</b> - Conflicting memory operation.
0x004	TCC	<b>Trap Instruction</b> - A trap instruction evaluates to "taken".
0x008	INST	<b>Unsupported Instruction</b> - Instruction not supported inside transactions.
0x010	PREC	<b>Precise Exception</b> - Execution generated a precise exception.
0x020	ASYN	<b>Async</b> - Received an asynchronous interrupt.
0x040	SIZ	<b>Size</b> - Transaction write set exceeded the size of the store queue.
0x080	LD	<b>Load</b> - Cache line in read set evicted by transaction.
0x100	ST	<b>Store</b> - Data TLB miss on a store.
0x200	CTI	<b>Control transfer</b> - Mispredicted branch.
0x400	FP	<b>Floating point</b> - Divide instruction.
0x800	UCTI	<b>Unresolved control transfer</b> - branch executed without resolving load on which it depends

Table 1. cps register: bit definitions and example failure reasons that set them.

# A Simple STM

```
pthread_mutex_t g_global_lock;  
  
begin_tx() {  
    pthread_mutex_lock(g_global_lock);  
}  
  
end_tx() {  
    pthread_mutex_unlock(g_global_lock);  
}  
  
abort() {  
    // can't happen  
}
```

```
remove(list, x) {  
    begin_tx();  
    pos = find(list, x);  
    if(pos)  
        erase(list, pos);  
    end_tx();  
}
```

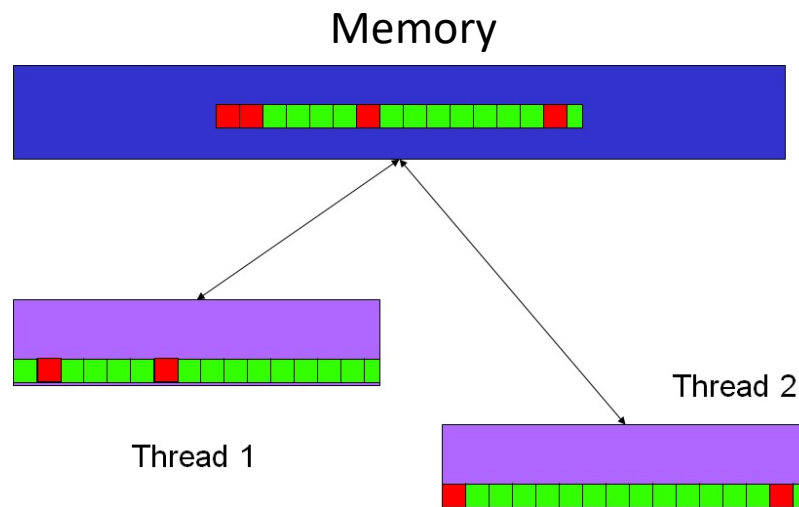
Is this Transactional  
Memory?  
Yes...just not optimistic

# A Better STM: System Model

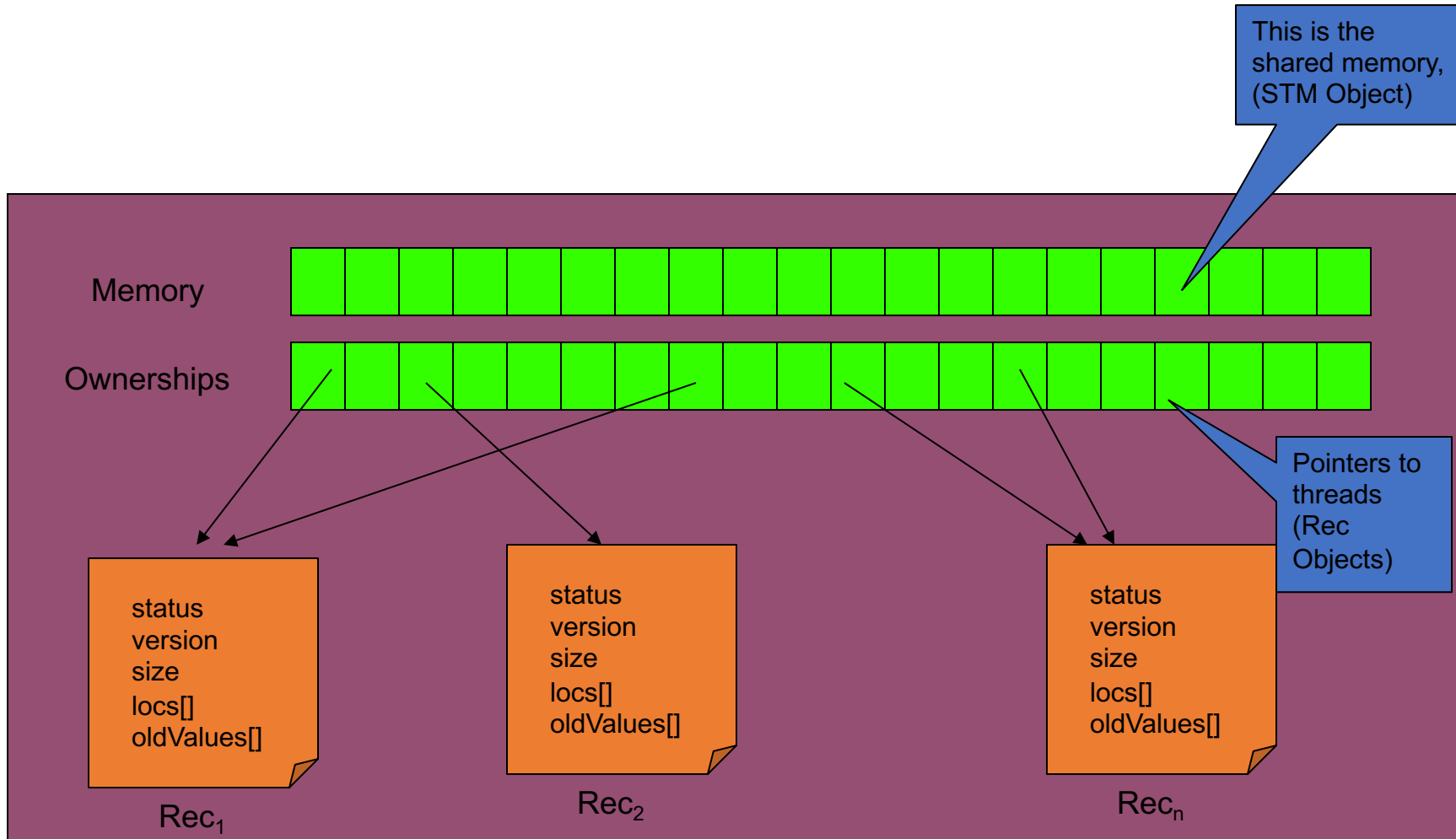
System == <threads, memory>

Memory cell supports TM operations:

- $Write^i(L,v)$  - thread  $i$  writes  $v$  to  $L$
- $Read^i(L,v)$  - thread  $i$  reads  $v$  from  $L$
- $LL^i(L,v)$  - thread  $i$  reads  $v$  from  $L$ , marks  $L$  read by  $i$
- $SC^i(L,v)$  - thread  $i$  writes  $v$  to  $L$ 
  - returns *success* if  $L$  is marked as read by  $i$ .
  - Otherwise it returns *failure*.



# STM Design Overview



# HTM vs. STM

Hardware	Software
Fast (due to hardware operations)	Slow (due to software validation/commit)
Light code instrumentation	Heavy code instrumentation
HW buffers keep amount of metadata low	Lots of metadata
No need of a middleware	Runtime library needed
Only short transactions allowed (why?)	Large transactions possible

How could you get the best of both?

# Hybrid-TM

Best-effort HTM (use STM for long txns)

Possible conflicts between HW,SW and HW-SW Txn

What kind of conflicts do SW-Txns care about?

What kind of conflicts do HW-Txns care about?

Some initial proposals:

HyTM: uses an ownership record per memory location (overhead?)

PhTM: HTM-only or (heavy) STM-only, low instrumentation

**Current HW essentially requires something like this**



# Concluding Remarks

- Transactions: a great abstraction
- Solve reliability and concurrency problems
- Transactional Memory: an implementation
  - Solves only concurrency problems
  - Implementable in many ways (HW, SW, hybrid,...)