Transactions
Transactional Memory

Chris Rossbach

Outline for Today

- Questions?
- Administrivia
  - Keep working on those labs!
- Agenda
  - Transactions
  - Transactional Memory

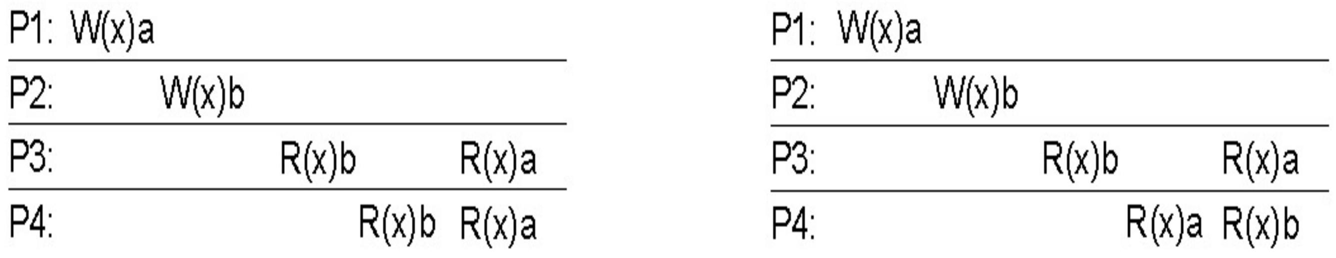
- Acks: Yoav Cohen for some STM slides

Faux Quiz questions

- How are promises and futures related? Since there is disagreement on the nomenclature, don’t worry about which is which—just describe what the different objects are and how they function.
- How does HTM resemble or differ from Load-linked Stored-Conditional?
- What are some pros and cons of HTM vs STM?
- What is Open Nesting? Closed Nesting? Flat Nesting?
- How does 2PL differ from 2PC?
- Define ACID properties: which, if any, of these properties does TM relax?

Sequential Consistency Redux

- All operations are executed in *some* sequential order
- each process issues operations in program order
  - *Any* valid interleaving is allowed
  - All *agree* on the same interleaving
  - Each process preserves its program order

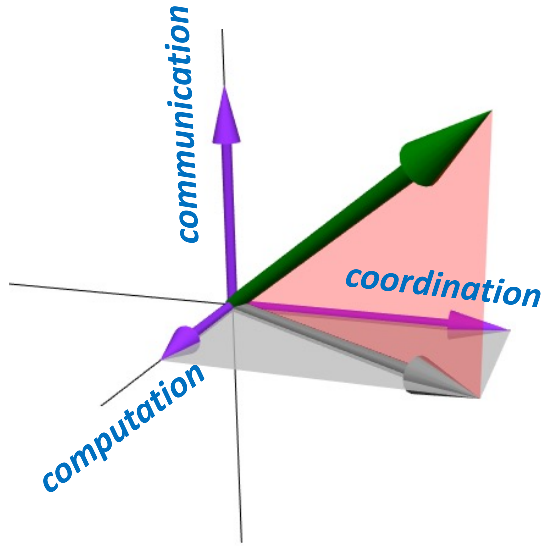


(a) (b)

Are either of these SC?

Transactions and 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*

- Transactions
  - *Mostly about how to deal with shared state*

Transactions

Core issue: multiple updates

Canonical examples:

```
move(file, old-dir, new-dir) {
    delete(file, old-dir)
    add(file, new-dir)
}
```

```
create(file, dir) {
    alloc-disk(file, header, data)
    write(header)
    add (file, dir)
}
```

Problems: crash in the middle / visibility of intermediate state

- Modified data in memory/caches
- Even if in-memory data is durable, multiple disk updates

Problem: Unreliability

- Want reliable update of two resources (e.g. in two disks, machines…)
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA account)
  - Move directory from server A to B
- Machines can crash, messages can be lost

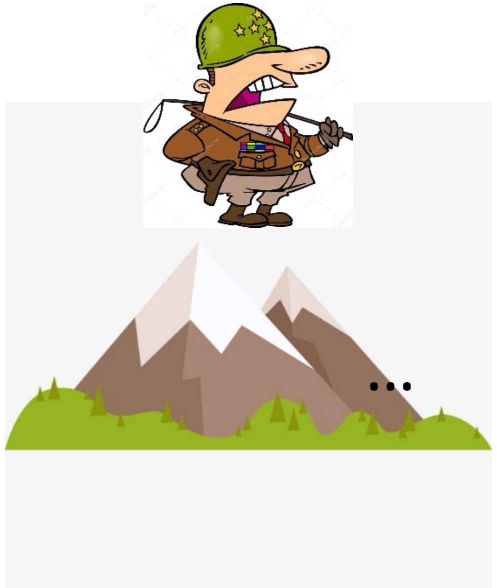
Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?

No.
Not even if all messages get through!

General’s paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

- Even if all messages delivered, can’t assume– maybe some message didn’t get through.
- No solution: one of the few CS impossibility results.



General A → General B: let’s attack at dawn
General B → General A: OK, dawn.
General A → General B: Check. Dawn it is.
General B → General A: Alright already—dawn.

…

Transactions can help

(*but can't solve it*)

- Solves weaker problem:
  - 2 things will either happen or not
  - not necessarily at the same time
- Core idea: one entity has the power to say yes or no for all
  - Local txn: one final update (TxEND) irrevocably triggers several
  - Distributed transactions
    - 2 phase commit
    - One machine has final say for all machines
    - Other machines bound to comply

What is the role of synchronization here?

Transactional Programming Model

```
begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;
```

What has changed from previous programming models?

ACID Semantics

- Atomic – all up
- Consistent – sy   oss updates
- Isolated – no vi
- Durable – once
- Are subsets eve
  - When would ACI be useful?
  - ACD?
  - Isolation only?

What are they?
- A
- C
- I
- D

```
begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;
```

Transactions: Implementation

- Key idea: turn multiple updates into a single one
- Many implementation Techniques
  - Two-phase locking
  - Timestamp ordering
  - Optimistic Concurrency Control
  - Journaling
  - 2,3-phase commit
  - Speculation-rollback
  - Single global lock
  - Compensating transactions

Key problems:
- output commit
- synchronization

Implementing Transactions

```
BEGIN_TXN();
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
COMMIT_TXN();
```

```
BEGIN_TXN() {
    LOCK(single-global-lock);
}
```

```
COMMIT_TXN() {
    UNLOCK(single-global-lock);
}
```

Pros/Cons?

Two-phase locking

- Phase 1: only acquire locks in
- Phase 2: unlock at commit
- avoids deadlock

B commits changes that depend on A's updates

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```

A: grab locks
A: modify x, y,
A: unlock y, x
B: grab locks
B: update x, y
B: unlock y, x
B: COMMIT
A: CRASH

BEGIN_TXN() {
rwset = Union(rset, wset);
rwset = sort(rwset);
forall x in rwset
LOCK(x);
}

COMMIT_TXN() {
forall x in rwset
UNLOCK(x);
}

Pros/Cons?

What happens on failures?

Two-phase commit

- N participants agree or don’t (atomicity)
- Phase 1: everyone “prepares”
- Phase 2: Master decides and tells everyone to actually commit
- What if the master crashes in the middle?

2PC: Phase 1

1. Coordinator sends REQUEST to all participants
2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Example—move: C→S1: delete foo from /, C→S2: add foo to /

Failure case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 decides permission problem
S2 writes/sends VOTE_ABORT

Success case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 writes add foo to /
S2 writes/sends VOTE_COMMIT

2PC: Phase 2

- Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Participants receive decision, write GLOBAL_* to log

2PC corner cases

Phase 1

1. Coordinator sends REQUEST to all participants
2. X Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Phase 2

- Y Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - W Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Z Participants recv decision, write GLOBAL_* to log

- What if participant crashes at X?
- Coordinator crashes at Y?
- Participant crashes at Z?
- Coordinator crashes at W?

2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!
- Can participants ask each other what happened?
- 2PC: always has risk of indefinite blocking
- Solution: (yes) 3 phase commit!
  - Reliable replacement of crashed “leader”
  - 2PC often good enough in practice

Nested Transactions

3 basic flavors:
* **Flat:** subsume inner transactions
* **Closed:** subsume w partial rollback
* **Open:** pause transactional context

- Composition of transactions
  - E.g. interact with multiple organizations, each supporting txns
  - Travel agency: canonical example
- Nesting: view transaction as collection of:
  - actions on unprotected objects
  - protected actions that my be undone or redone
  - real actions that may be deferred but not undone
  - nested transactions that may be undone
- Open Nesting details:
  - Nested transaction returns name and parameters of compensating transaction
  - Parent includes compensating transaction in log of parent transaction
  - Invoke compensating transactions from log if parent transaction aborted
  - Consistent, atomic, durable, but not isolated

Transactional Memory: ACI

Transactional Memory :

- Make multiple memory accesses atomic
- All or nothing – Atomicity
- No interference – Isolation
- Correctness – Consistency
- No durability, for obvious reasons

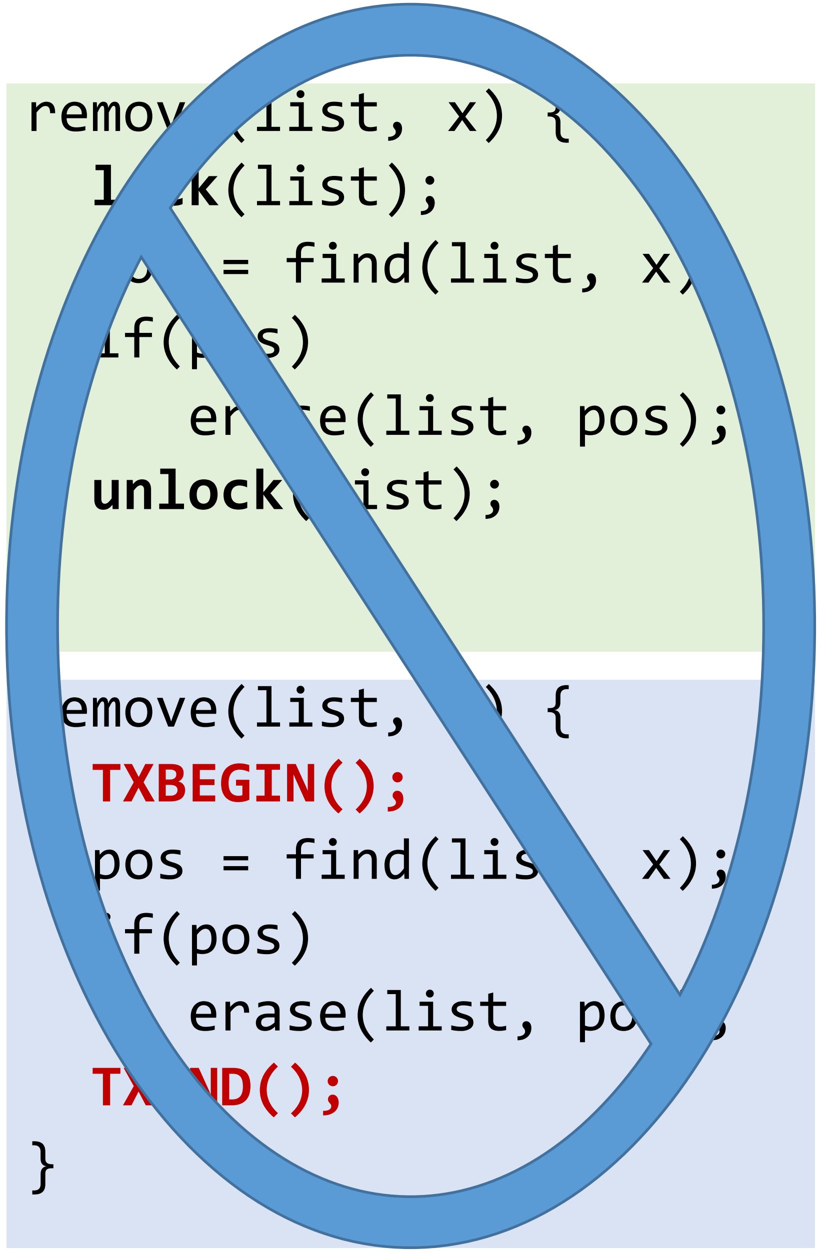
- Keywords : Commit, Abort, Speculative access,
  Checkpoint

```
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
- Transactional Memory: also super-awesome, ***but***:
- Transactions != TM
- TM is an ***implementation technique***
- Often presented as programmer abstraction
- Remember Optimistic Concurrency Control

A Simple TM

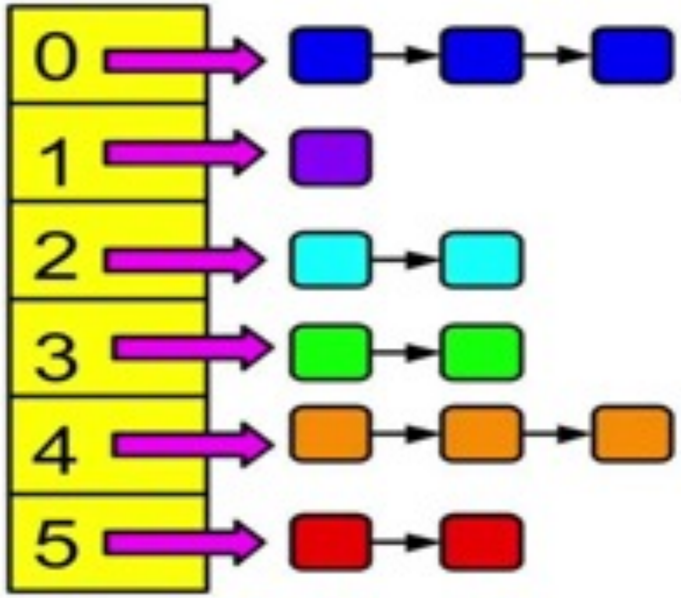
```
remove(list, x) {
  begin_tx();
  pos = find(list, x);
  if(pos)
     erase(list, pos);
  end_tx();
}
```

```
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
}
```

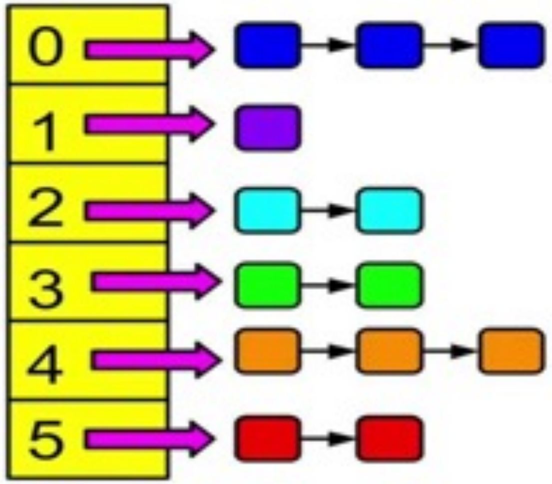
Actually, this works fine…
But how can we improve it?

Concurrency Control Revisited

Consider a hash-table

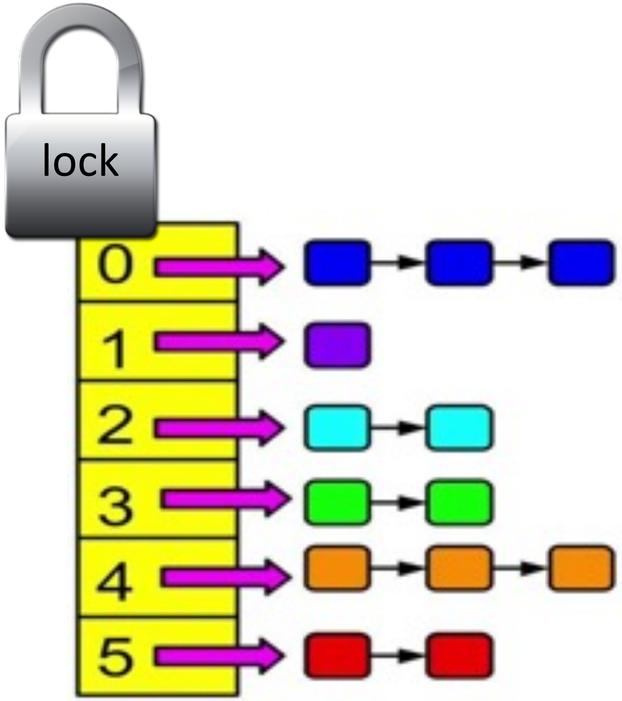


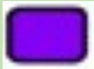
Concurrency Control Revisited



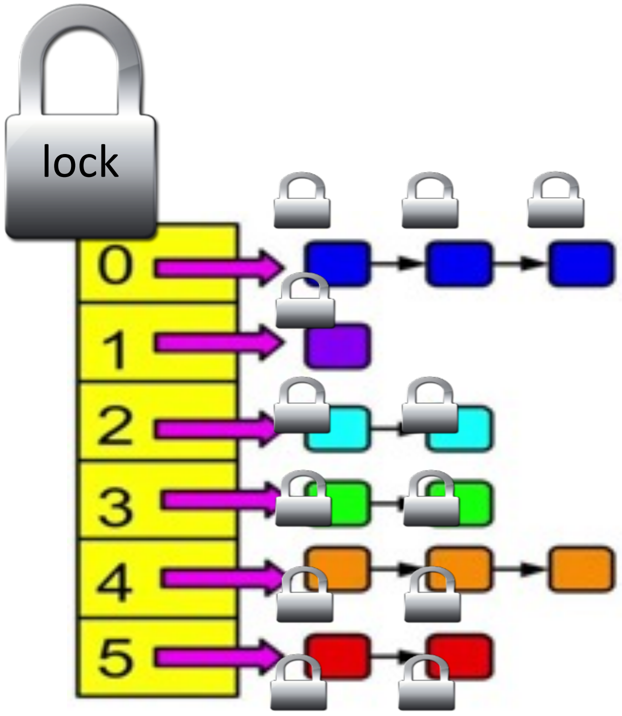
thread T1	thread T2
	ht.add( );
ht.add( );	
if(ht.contains( ))	
	if(ht.contains( ))
	ht.del( );
ht.del( ); ✗	

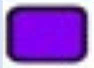
Concurrency Control Revisited



thread T1	thread T2
ht.lock()	**ht.lock();**
ht.add();	ht.add();
if(ht.contains())	
	if(ht.contains())
	ht.del();
ht.del();	
ht.unlock();	**ht.unlock();**

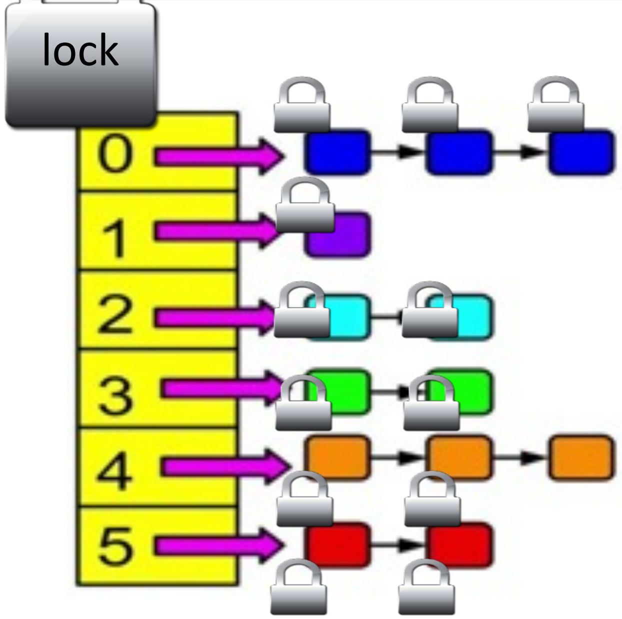
Pessimistic concurrency control



thread T1	thread T2
ht.lock();	**ht.lock();**
ht.add();	ht.add();
if(ht.contains())	if(ht.contains())
ht.del();	ht.del();
ht.unlock();	**ht.unlock();**

Optimistic concurrency control

What do we do when same data is accessed?



thread T1	thread T2
ht.lock();	**ht.lock();**
ht.add();	ht.add();
if(ht.contains())	if(ht.contains())
ht.del();	ht.del();
ht.unlock();	**ht.unlock();**

TM Primer

Key Ideas:

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

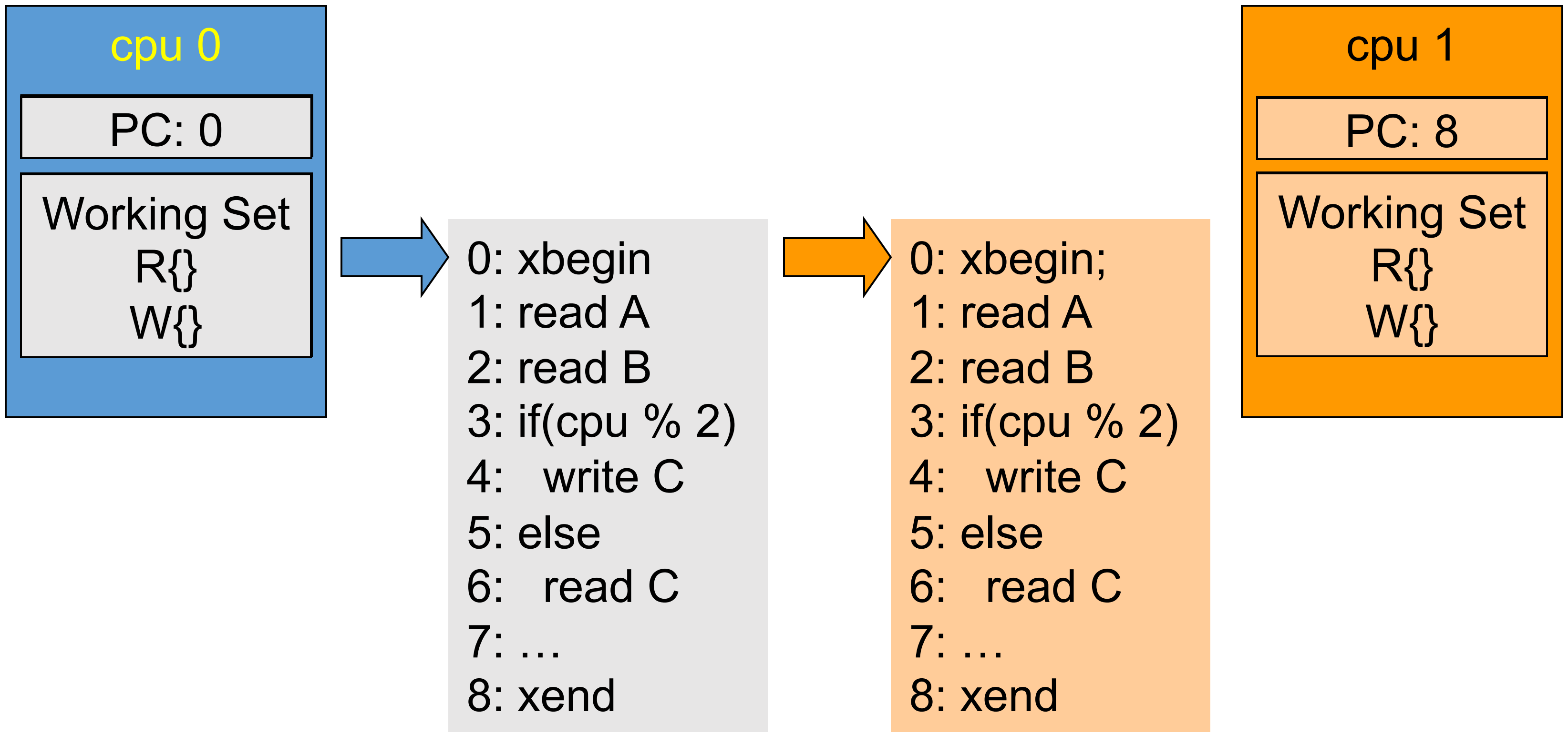
Key Abstractions:

- Primitives
  - **xbegin, xend, xabort**
- Conflict

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

- Contention Manager
  - Need flexible policy

TM basics: example

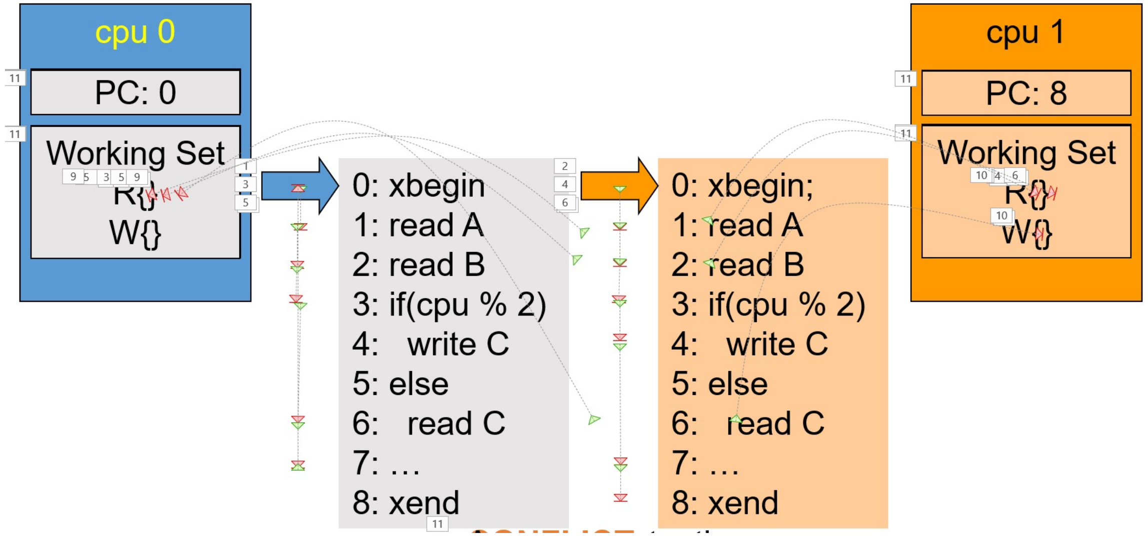


CONFLICT:
C is in the read set of cpu0, and in the write set of cpu1

Assume contention manager decides cpu1 wins:
cpu0 rolls back

cpu1 commits

TM Implementation



Data Versioning

- Eager Versioning
- Lazy Versioning

Conflict Detection and Resolution

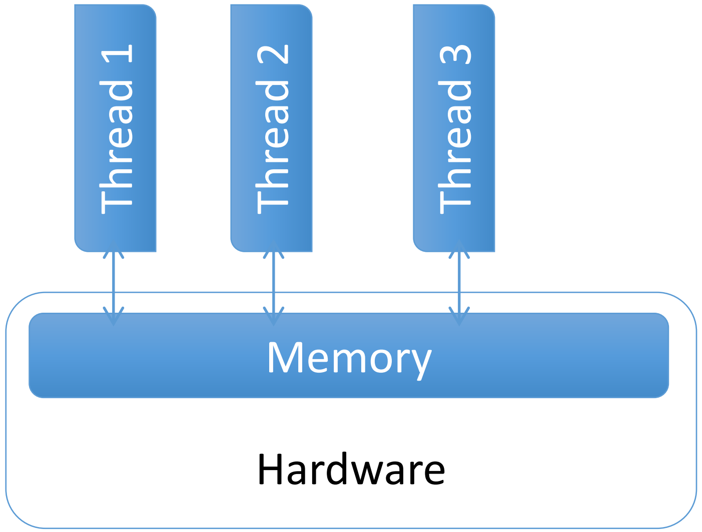
- Pessimistic Concurrency Control
- Optimistic Concurrency Control

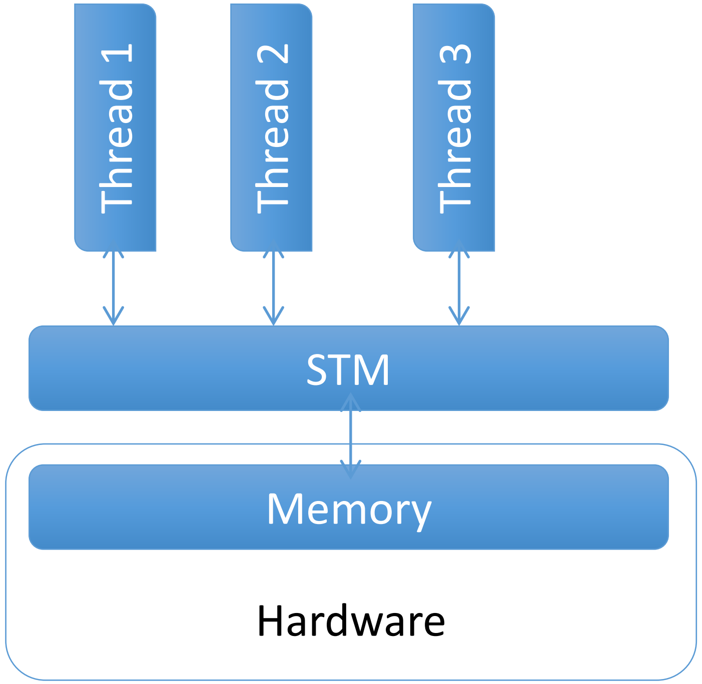
Conflict Detection Granularity

- 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



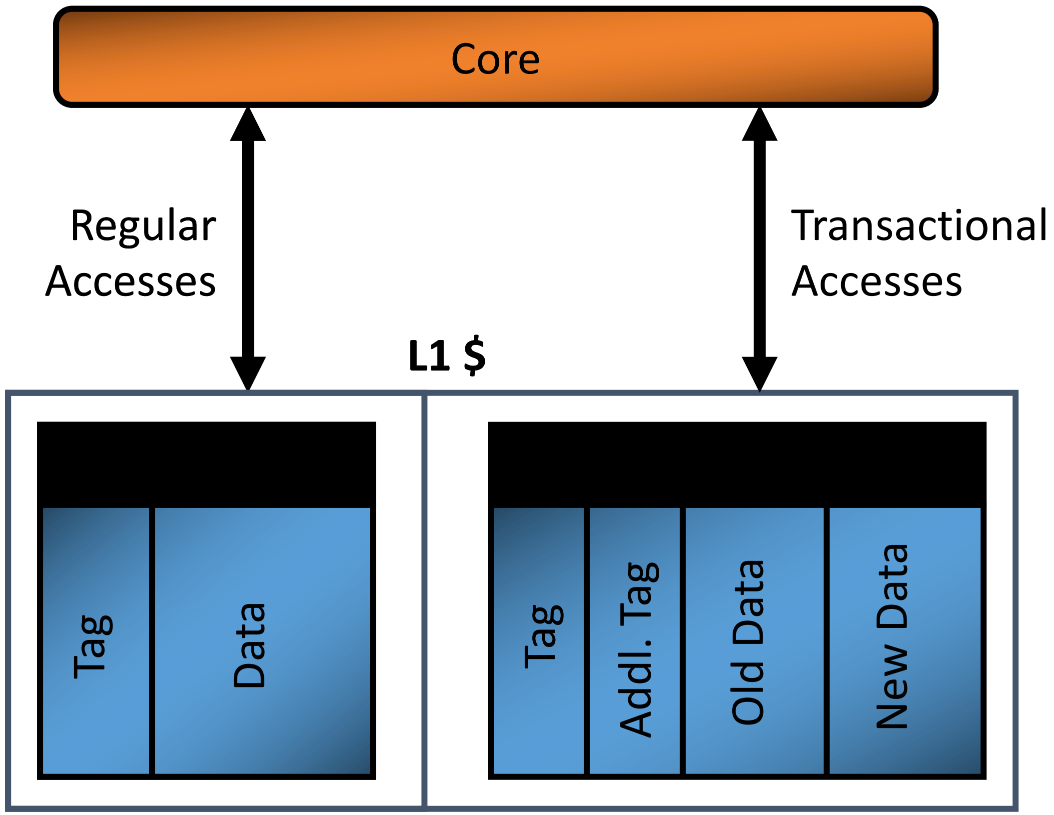


Hardware Transactional Memory

- 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

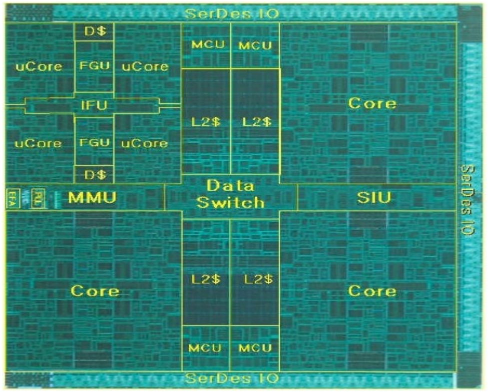


Key ideas

- *Checkpoint architectural state*
- *Caches: 'versioning' for memory*
- *Change coherence protocol*
- *Conflict detection in hardware*
- *'Commit' transactions if no conflict*
- *'Abort' on conflict (or special cond)*
- *'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 / µarchitecture
- Many unexpected causes of aborts
- Rock v1 diagnostics unable to distinguish distinct failure modes

Mask	Name	Description and example cause
0x001	EXOG	**Exogenous** - Intervening code has run: cps register contents are invalid.
0x002	COH	**Coherence** - Conflicting memory operation.
0x004	TCC	**Trap Instruction** - A trap instruction evaluates to "taken".
0x008	INST	**Unsupported Instruction** - Instruction not supported inside transactions.
0x010	PREC	**Precise Exception** - Execution generated a precise exception.
0x020	ASYNC	**Async** - Received an asynchronous interrupt.
0x040	SIZ	**Size** - Transaction write set exceeded the size of the store queue.
0x080	LD	**Load** - Cache line in read set evicted by transaction.
0x100	ST	**Store** - Data TLB miss on a store.
0x200	CTI	**Control transfer** - Mispredicted branch.
0x400	FP	**Floating point** - Divide instruction.
0x800	UCTI	**Unresolved control transfer** - 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?

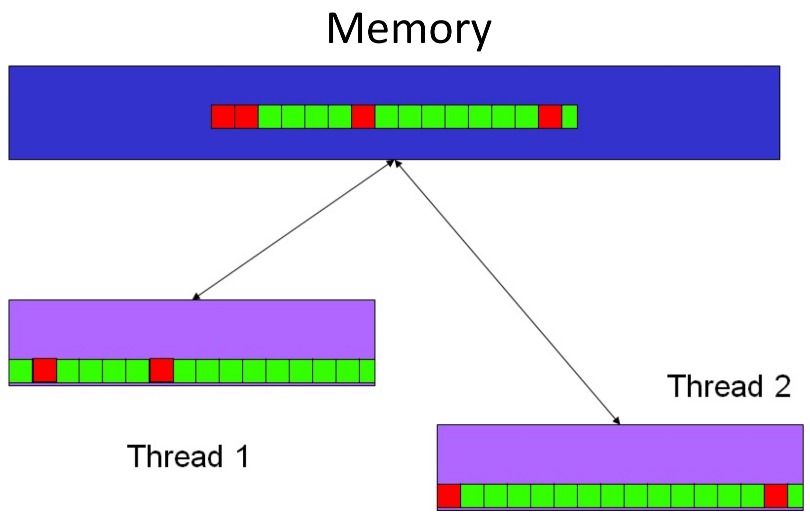
TM is a deep area: consider it for your project!

A Better STM: System Model

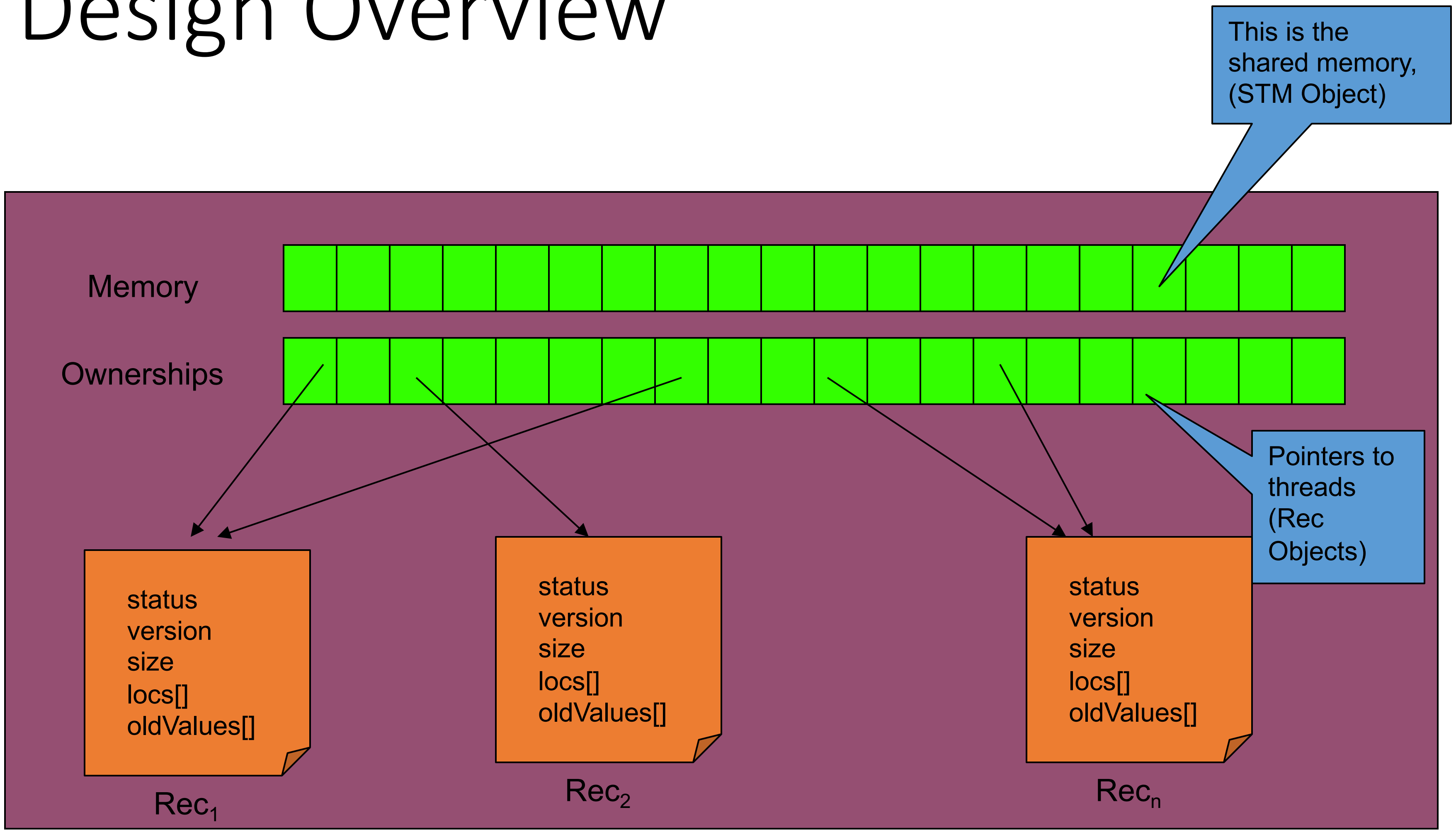
System == <threads, memory>

Memory cell support 4 operations:

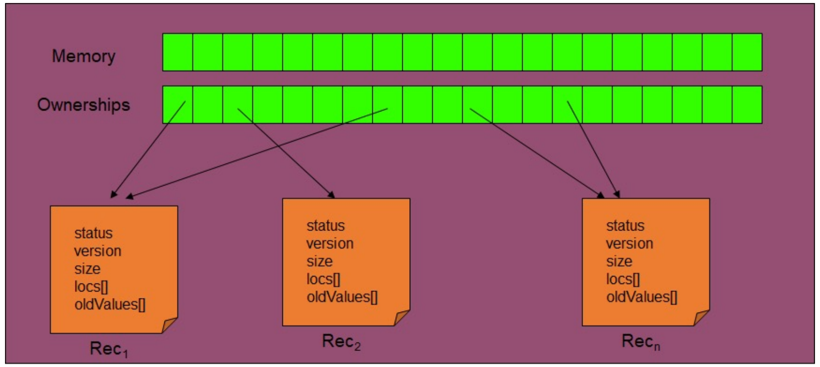
- Write^i(L,v) - *thread i writes v to L*
- Read^i(L,v) - *thread i reads v from L*
- LL^i(L,v) - *thread i reads v from L, marks L read by I*
- SC^i(L,v) - *thread i writes v to L*
  - returns *success* if L is marked as read by i.
  - Otherwise it returns *failure*.



STM Design Overview



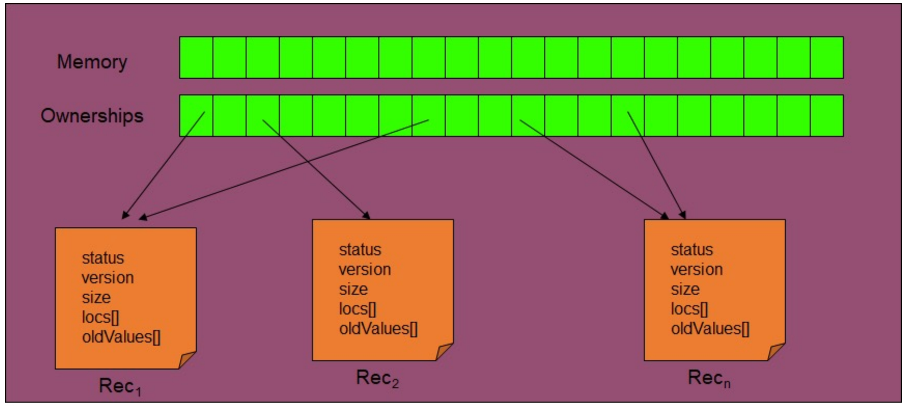
Threads: Rec Objects




```
class Rec {
    boolean stable = false;
    boolean, int status= (false,0);  //can have two values…
    boolean allWritten = false;
    int version = 0;
    int size = 0;
    int locs[] = {null};
    int oldValues[] = {null};
}
```

Each thread → instance of Rec class *(short for record).*

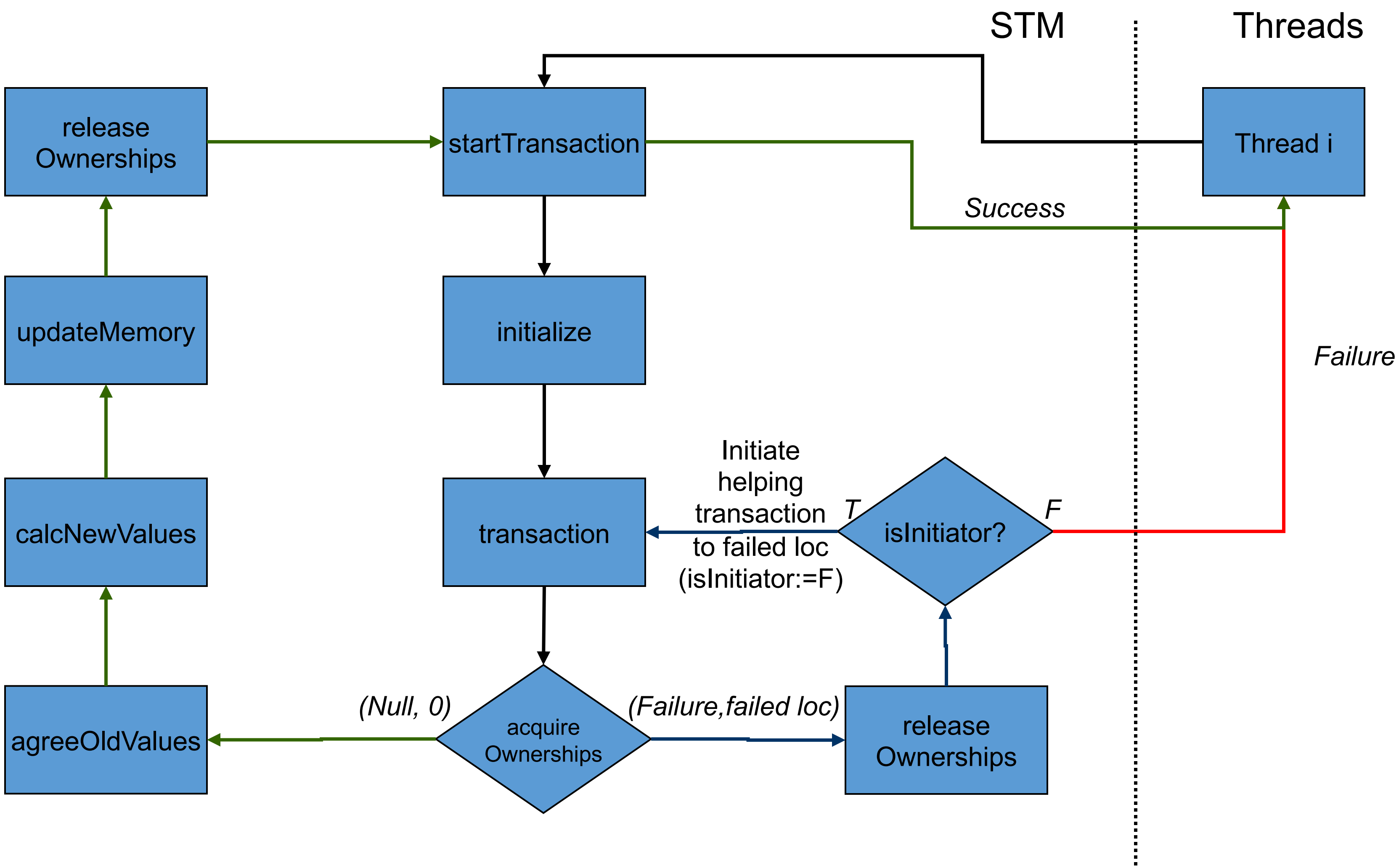
Rec instance defines current transaction on thread



Memory: STM Object

```
public class STM {
    int memory[];
    Rec ownerships[];

    public boolean, int[] startTranscation(Rec rec, int[] dataSet){...};

    private void initialize(Rec rec, int[] dataSet)
    private void transaction(Rec rec, int version, boolean isInitiator) {...};
    private void acquireOwnerships(Rec rec, int version) {...};
    private void releaseOwnershipd(Rec rec, int version) {...};
    private void agreeOldValues(Rec rec, int version) {...};
    private void updateMemory(Rec rec, int version, int[] newvalues) {...};
}
```

Flow of a transaction



Implementation

```
public boolean, int[] startTranscation(Rec rec, int[] dataSet
    initialize(rec, dataSet);
    rec.stable = true;
    transaction(rec, rec.version, true);
    rec.stable = false;
    rec.version++;
    if (rec.status) return (true, rec.oldValues);
    else return false;
}
```

rec – The thread that executes this transaction.
dataSet – The location in memory it needs to own.

This notifies other threads that I can be helped

Implementation

```
private void transaction(Rec rec, int version, boolean isInitiator) {
	acquireOwnerships(rec, version); // try to own locations

	(status, failedLoc) = LL(rec.status);
	if (status == null) {              // success in acquireOwnerships
		if (versoin != rec.version) return;
		SC(rec.status, (true,0));
	}

	(status, failedLoc) = LL(rec.status);
	if (status == true) {              // execute the transaction
		agreeOldValues(rec, version);
		int[] newVals = calcNewVals(rec.oldvalues);
		updateMemory(rec, version);
		releaseOwnerships(rec, version);
	}
	else {                // failed in acquireOwnerships
		releaseOwnerships(rec, version);
		if (isInitiator) {
			Rec failedTrans = ownerships[failedLoc];
			if (failedTrans == null) return;
			else {                // execute the transaction that owns the location you want
				int failedVer = failedTrans.version;
				if (failedTrans.stable) transaction(failedTrans, failedVer, false);
			}
		}
	}
}
```

rec – The thread that executes this transaction.
version – Serial number of the transaction.
isInitiator – Am I the initiating thread or the helper?

Another thread own the locations I need and it hasn't finished its transaction yet.

So I go out and execute its transaction in order to help it.

Implementation

```
private void acquireOwnerships(Rec rec, int version) {
     for (int j=1; j<=rec.size; j++) {
          while (true) do {
               int loc = locs[j];
               if LL(rec.status) != null return;     // transaction completed by some
                  other thread
               Rec owner = LL(ownerships[loc]);
               if (rec.version != version) return;
               if (owner == rec) break;    // location is already mine
               if (owner == null) {        // acquire location
                    if ( SC(rec.status, (null, 0)) ) {
                       if ( SC(ownerships[loc], rec) ) {
                          break;
                       }
                    }
               }
               else {// location is taken by someone else
                    if ( SC(rec.status, (false, j)) ) return;
               }

          }

     }
}
```

If I'm not the last one to read this field, it means that another thread is trying to execute this transaction. Try to loop until I succeed or until the other thread completes the transaction

Implementation

```
private void agreeOldValues(Rec rec, int version) {
      for (int j=1; j<=rec.size; j++) {
            int loc = locs[j];
            if ( LL(rec.oldvalues[loc]) != null ) {
                  if (rec.version != version) return;
                  SC(rec.oldvalues[loc], memory[loc]);
            }
      }
}
```

Copy the dataSet to my private space

```
private void updateMemory(Rec rec, int version, int[] newvalues) {
      for (int j=1; j<=rec.size; j++) {
            int loc = locs[j];
            int oldValue = LL(memory[loc]);
            if (rec.allWritten) return;     // work is done
            if (rec.version != version) return;
            if (oldValue != newValues[j]) SC(memory[loc], newValues[j]);
      }
      if (! LL(rec.allWritten) ) {
         if (rec.version != version) SC(rec.allWritten, true);
      }
}
```

Selectively update the shared memory

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 would you get the best of both?

Hybrid-TM

- Best-effort HTM (use STM for long trx)
- Possible conflicts between HW,SW and HW-SW Trx
  - What kind of conflicts do SW-Trx care about?
  - What kind of conflicts do HW-Trx care about?
- Some initial proposals:
  - HyTM: uses an ownership record per memory location (overhead?)
  - PhTM: HTM-only or (heavy) STM-only, low instrumentation

Questions?