CS302

Concurrency Control

Spring 2025
Arkaprava Basu & Babak Falsafi
parsa.epfl.ch/course-info/cs302

Adapted from slides originally developed by Profs. Falsafi, Fatahalian, Mowry, Wenisch of CMU, Michigan

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr		3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- Concurrency Control
  - Transactional memory
  - Point-to-point synchronization
  - Barriers
- Exercise session
  - Free session
  - Ask doubts and questions to TAs and SAs
- Next Tuesday:
  - Midterm exam

Correction: Vectorized DAXPY Loop

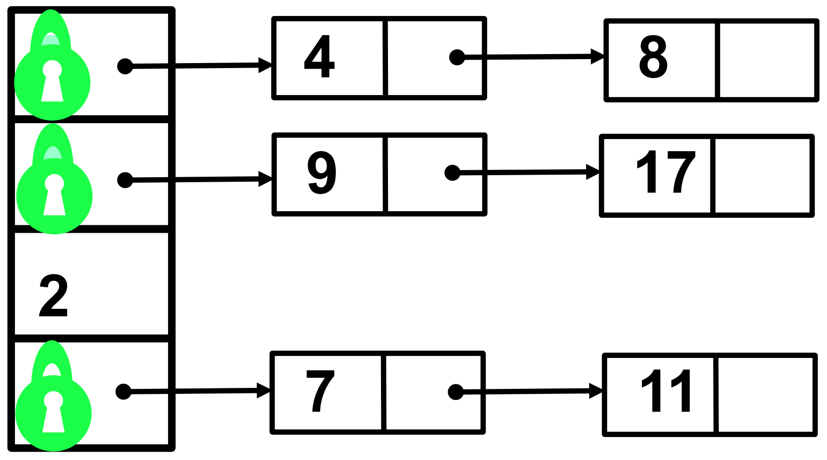
- Assumption:
  - Compute 64 elements at a time
  - Elements are double-precision
- How many serialized cache misses? (w/ 64B cache lines)
  - Vector load brings 64 elements at a time, equivalent to 8 cache lines
  - 64*8B is the total number of bytes loaded, not the number of elements
  - ~~2 * (1024 / (64*8)) = 4~~
  - 2 * (1024 / 64) = 32

```
; x[] -> r2, y[] -> r3
; a -> r4,
; &x[n] -> r5

loop:
  lw.v     v1, 0(r2)    ; load x[i]
  lw.v     v2, 0(r3)    ; load y[i]
  mul.sv   v1, r4, v1   ; a*x[i]
  add.v    v1, v1, v2   ; … + y[i]
  sw.v     v1, 0(r3)    ; store y[i]
  add      r2, r2, 512 ; 64*8
  add      r3, r3, 512
  bne      r2, r5, loop
```

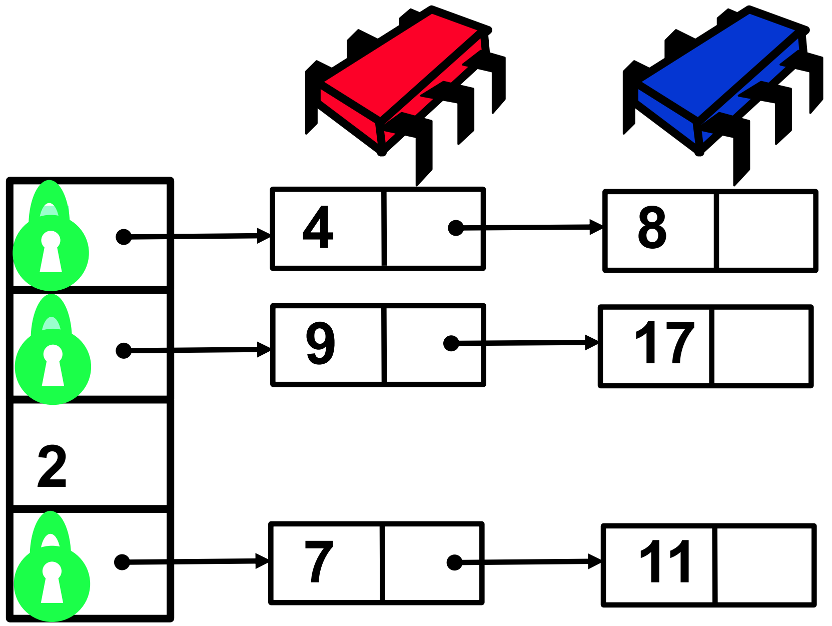
Difficulties of Locking

- Could build lock-free data structures
  - Not easy to reason about and program
- Better idea: hardware to detect contention!
  - No explicit lock ordering in the code
  - Analogy: “I didn’t do it, no one saw me doing it!”
- Connection to Lec.7.1 (slide 15)
  - LL/SC uses cache coherence to detect writes
  - Why not extend it to the whole critical section?

Speculative Lock per Hash Bucket



Simultaneous Modification Without Locking



Declarative Atomicity

- Wouldn't it be nice if…

Instead of writing difficult locking code, simply tell the runtime to make a section atomic?

- Similar principle:
  - Programmer declares a variable so its updates will be atomic, language runtime figures it all out
  - No need for manual memory fences or synch. variables
- Hardware Transactional Memory (HTM) does this

Declarative/Transactional Example

Lock-based Code

```
void editHash(HashTbl tbl,
              int key) {
  synchronized(tbl) {
    // read objects
    HashObj obj = tbl.get(key);
    // update
    obj.update();
  }
}
```

Transactional Code

```
void editHash(HashTbl tbl,
              int key) {
  atomic {
    // read objects
    HashObj obj = tbl.get(key);
    // update
    obj.update();
  }
}
```

- Warning! This is pseudo-code
- Atomic construct (not just a single statement, but a full code section)
  - Code executes atomically
  - All side-effects either entirely visible or not visible at all

Transactional Processing (Databases)

- Used historically in databases (next semester)
  - Concurrency for data on disk (cached in memory)
  - In software
- Atomicity
  - Upon transaction commit, all writes take effect at once
  - On transaction abort, no writes take effect
- Isolation
  - No other processor can observe writes before commit
- Serializability
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed

Transactional Memory (TM)

- Memory (not disk) transactions
  - An atomic & isolated sequence of memory accesses
  - Supported in hardware by processor & cache hierarchy
- Atomicity
  - Upon commit, all memory writes take effect at once
  - On abort, none of the writes take effect
- Isolation
  - No other processor can observe writes before commit
- Serializability
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed

Transactions Are Composable

```
void editHash2(HashTbl tbl, int keyA, int keyB)
{
  atomic {
    // read objects
    HashObj objA = tbl.get(keyA);
    HashObj objB = tbl.get(keyB);
    // update
    objA.update(objB);
    objB.update(objA);
  }
}
```

- Transactions compose gracefully
  - Programmer declares global intent (atomic transfer)
  - No need to know about the implementation
- Outermost transaction defines atomicity boundary

Intel Haswell Transactional Memory

- New instructions for
  - `xbegin`: starts transaction, takes pointer to “fallback address” in case of abort
  - `xend`: ends transaction
  - `xabort`: software-initiated transaction abort
- Similar to LL/SC but can have:
  - A large sequence of instructions (as much as can fit in the pipeline)
  - Multiple memory accesses to various addresses
  - Alternative handler to run when aborting
- Processor commits all memory operations atomically
  - May abort transaction

Hardware Changes for TM

- Assume no conflicts occur
  - Record reads and writes between `xbegin` and `xend`
  - Keep all speculative state in the pipeline (as before)
- Check for conflicts in the data structure
  - Coherence indicating data race (R/W, W/R, W/W)
  - E.g., BusRd matches a ST in the LSQ (R/W)
  - E.g., BusRdX or BusInv matching a LD (W/R) or a ST (W/W) in LSQ
- Other corner cases to roll back
  - Cache eviction for blocks touched by critical section
- On conflict, execute specified recovery code

Recovery Mechanism in TM

- TM does not have locking semantics…

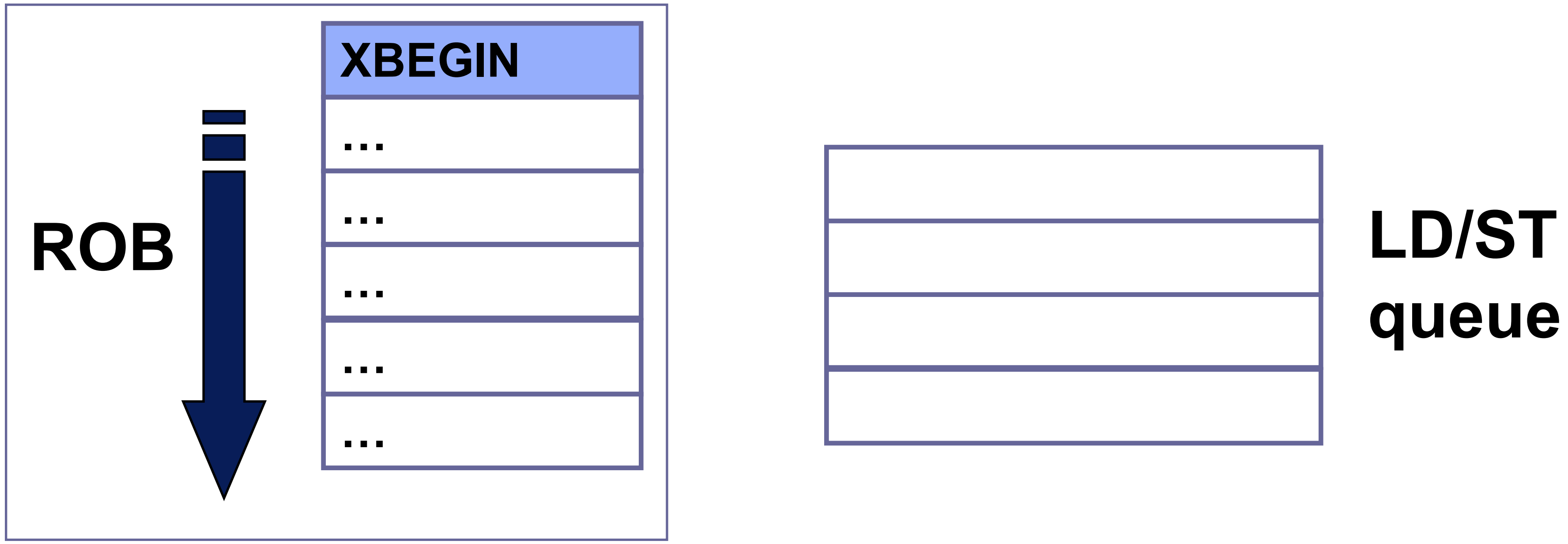
- Therefore, rely on user code for recovery
  - Once again, push behavior to the user

Example With TM

```
xbegin    recover

ld   r1, mem[obj]
addi r1, #10
st   mem[obj], r1

xend // commit trans.


recover:
br   rec_handler
```

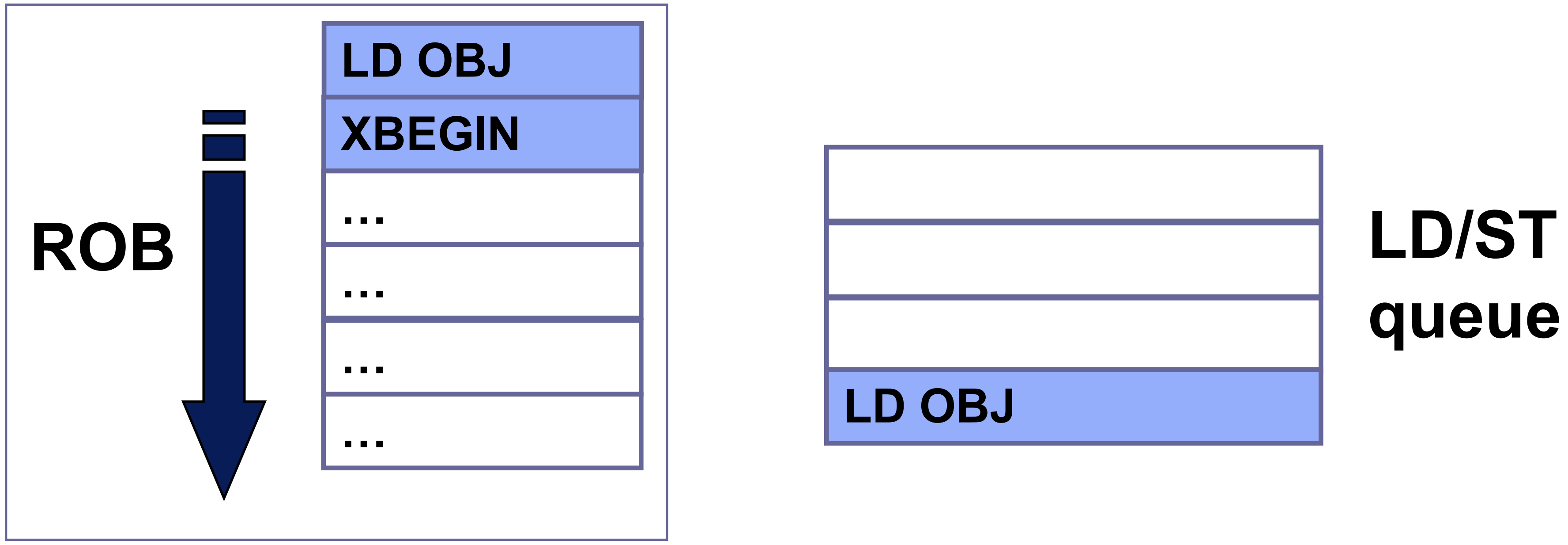
- `xbegin` starts transaction
  - HW begins speculating
- Memory R/W are recorded in the atomic section
- Check for conflicts on `xend`
- Conflicts trap to `recover`

TM



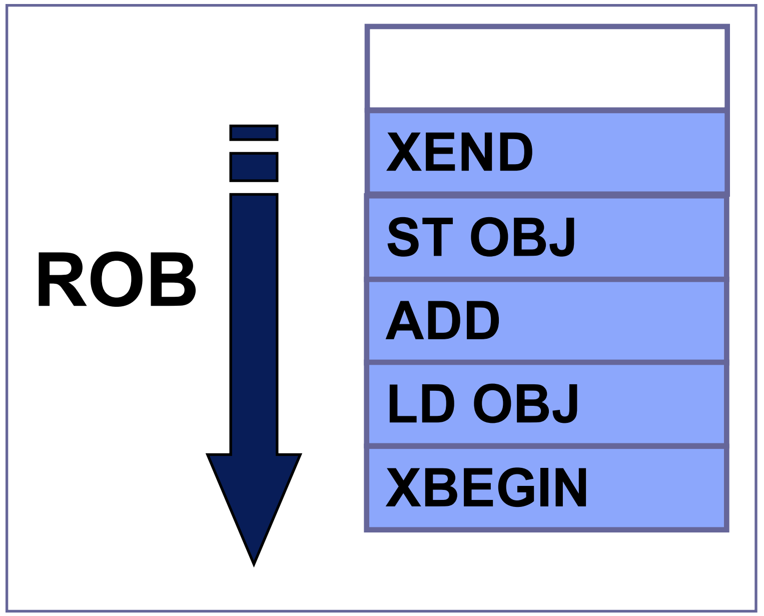
- Processor sees XBEGIN, starts speculating

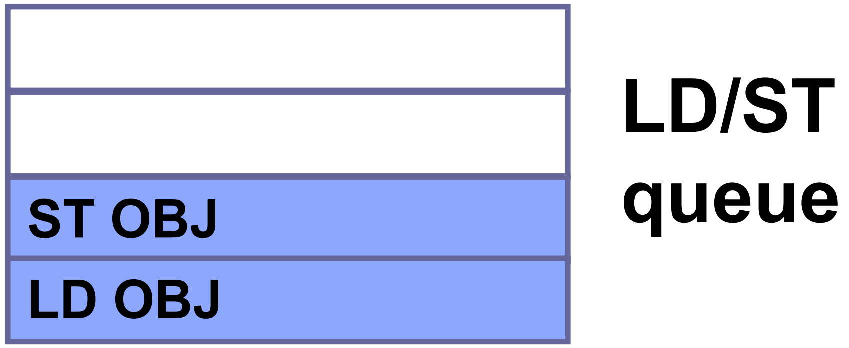
TM



- Processor sees XBEGIN, starts speculating

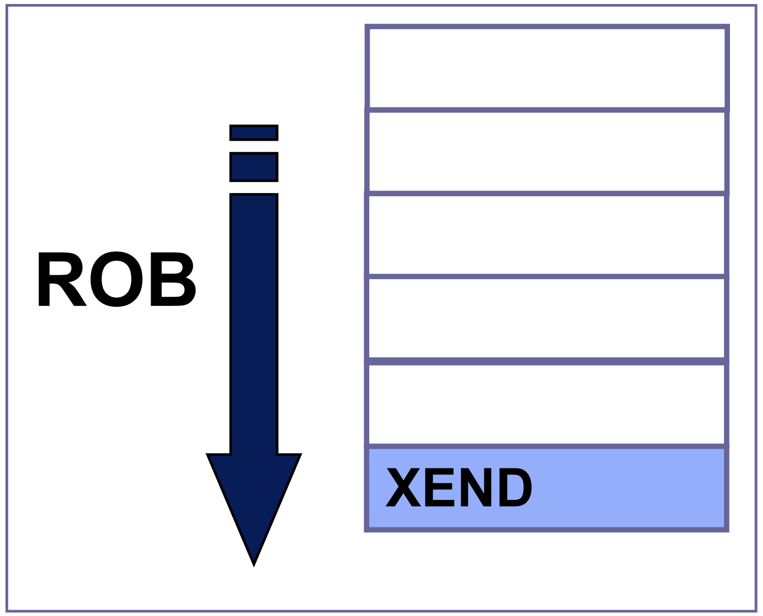
TM

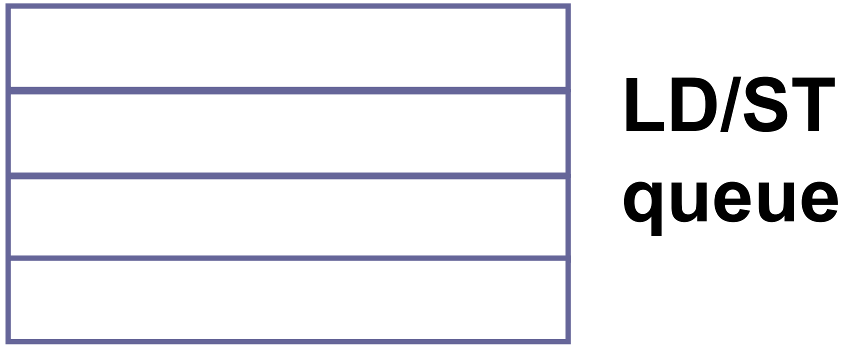




- LD OBJ & ST OBJ hit in the cache
- All state remains in the pipeline
- Wait until we see XEND to start committing

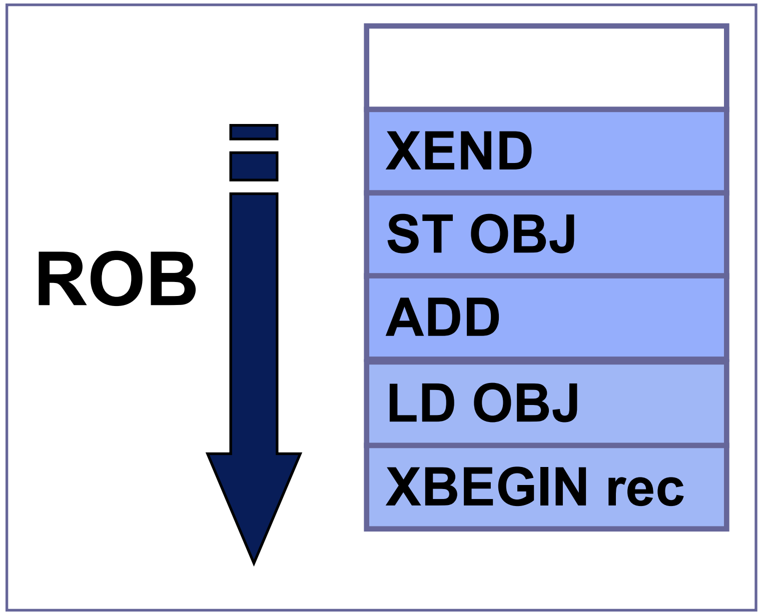
TM

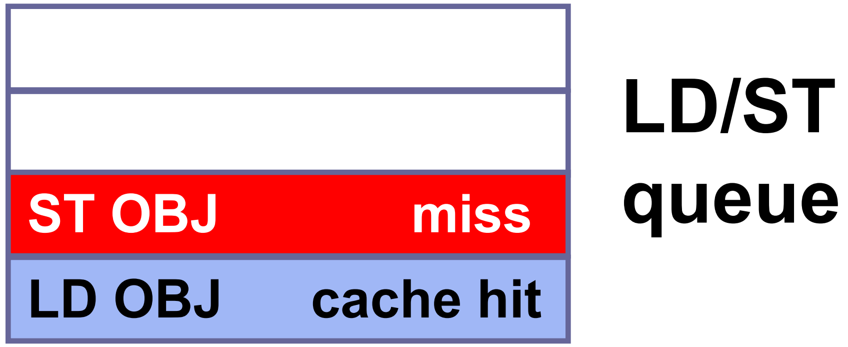




- Pipeline retires if no intervening coherence traffic
  - e.g., BusRdX/BusInv/BusRd for OBJ
  - The entire critical section retires atomically
- When XEND retires, speculation is successful

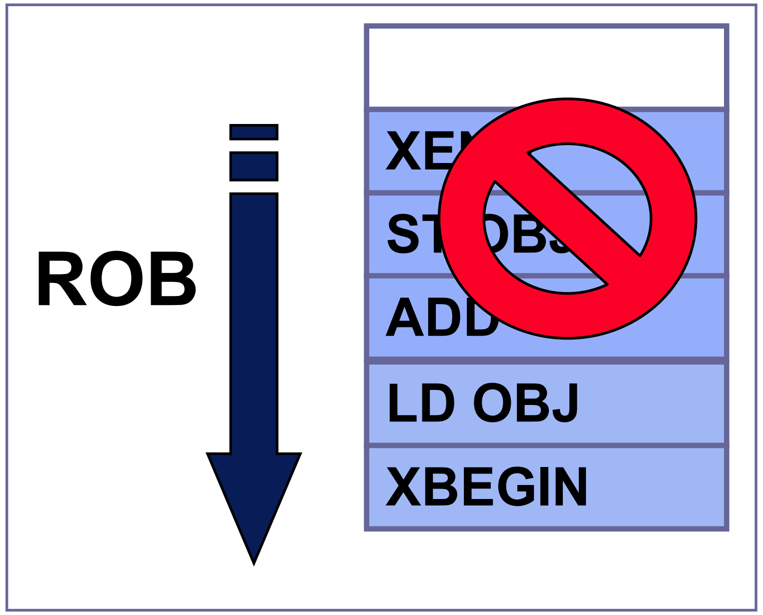
TM Rollback

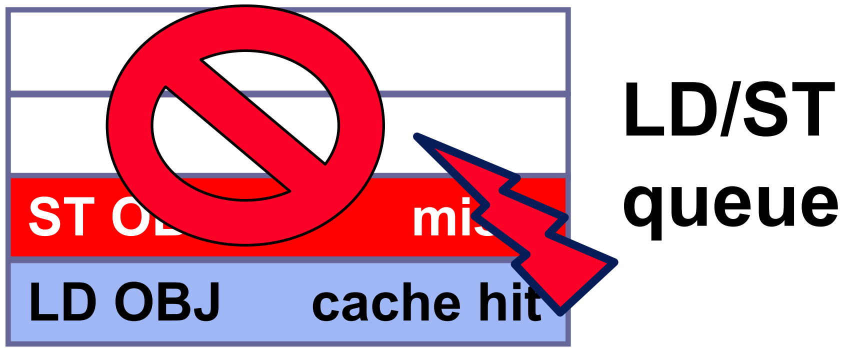




- OBJ is in “S” state
- ST OBJ misses (BusInv to get to “M” state)
- Pipeline blocks waiting for permission to store

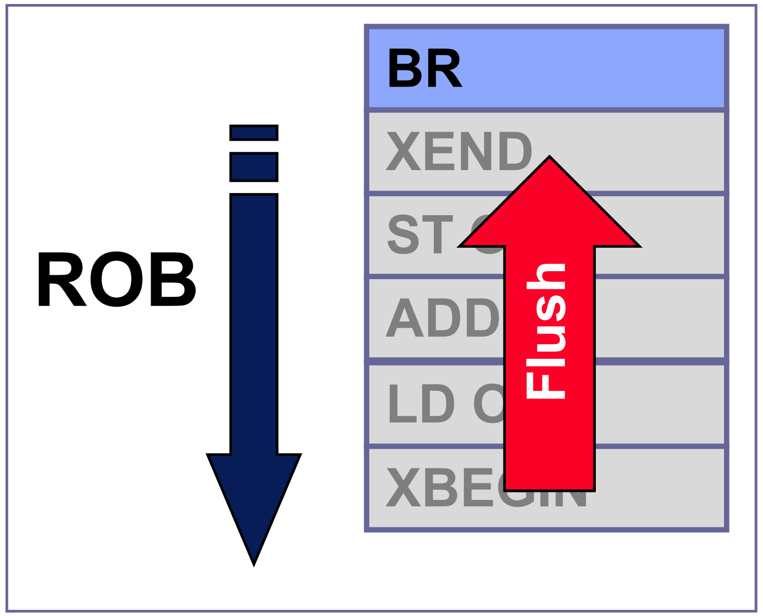
TM Rollback

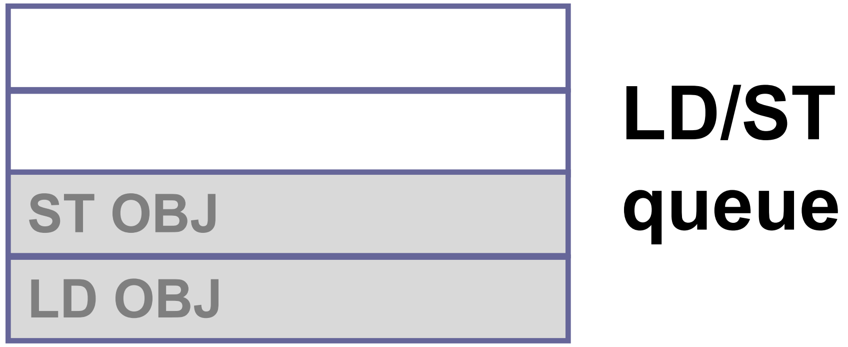




- BusInv arrives for OBJ
  - Another thread is writing to OBJ
- Abort transaction
- Jump to recover routine

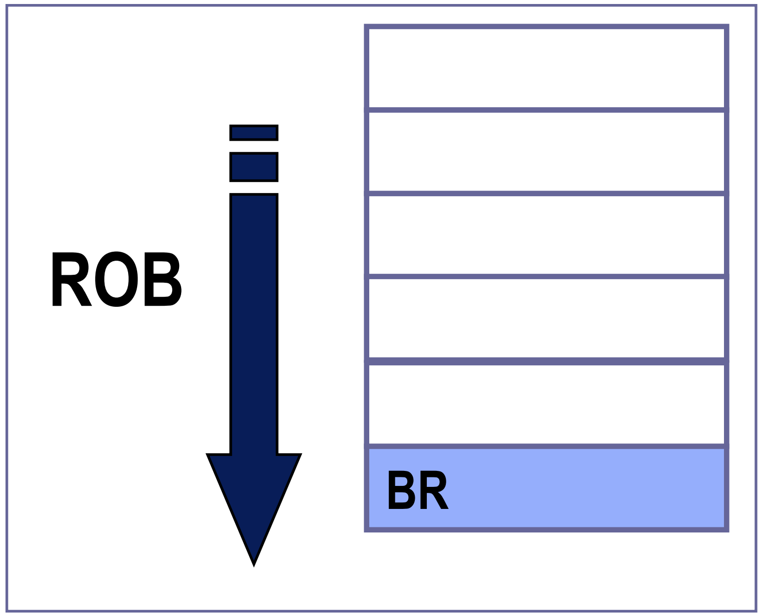
TM Rollback

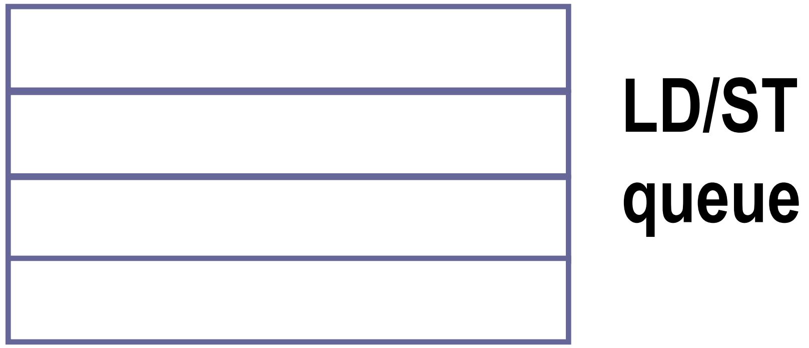




- All instructions flushed
- Execution jumps to label “recover” given by user

TM Recovery





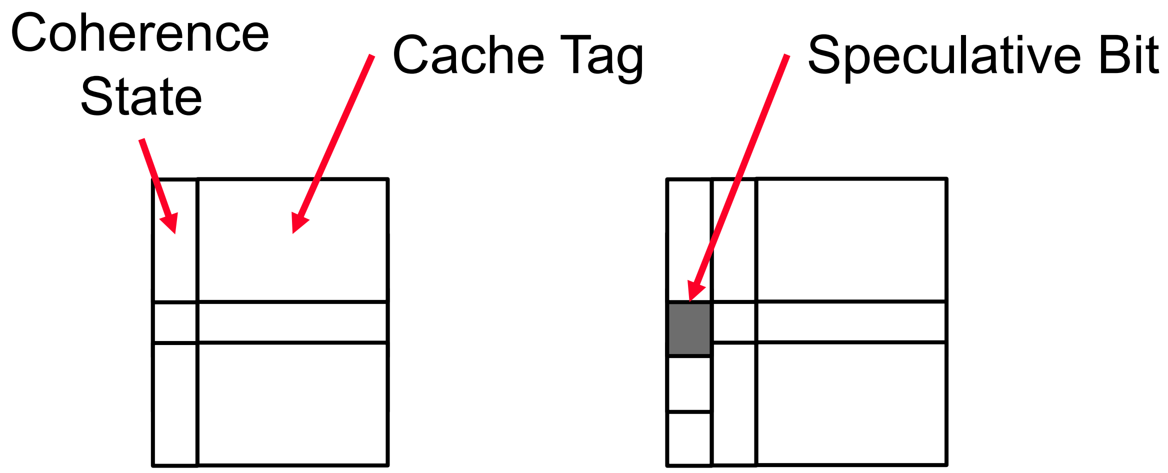
- Run from recovery code

TM Implementations

- Several decades worth of research
  - Originally most done in software
  - Software overhead for detecting conflicts is high
  - Many HW and HW/SW hybrids have emerged
  - See the book by Rajwar & Larus
- Real implementations in Intel, IBM CPUs
  - Keep speculative data in cache hierarchies (not just pipeline)

Extension: Data in Caches

- Recall: we assumed addresses tracked in LSQ
  - How can we extend that to storing it in the caches and store buffer?
- Simple idea: add some bits to mark certain cache lines as speculative
  - Same coherence mechanism to detect conflicts



Detection Policy

- Check for conflict at every operation
  - Use coherence actions (e.g., BusRd, BusRdX, BusInv)
  - Intuition: “I suspect conflicts might happen, so always check to see if one has occurred upon each coherence operation.”

- There are other options, TM open research area

A note on Software Transactional Memory

- SW for speculation, buffering and detection in
  - No hardware support
  - Huge in the academic community
  - Mostly a research testbed before HTM emerged
  - Too slow for real-world deployment

Point to Point Synchronization

Recall: Flag-Based Synchronization

- Example: using a shared variable as a flag

Thread 0

```
// val = done = 0;

val =
expensive_func();
done = 1;
```

Thread 1

```
// done = 0;

while(!done){ }
... = val;
```

- Assumes Sequential Consistency!
    - Recall all of the problems that happen in real systems

Barriers

- Flags mark updates to data, barriers introduce planned “stalls” in threads
  - e.g., Wait for all threads to read their input before start
- Technically equivalent, barriers use flags
  - Flags also have no built-in support for thread blocking
- You likely used these in Assignment 2!
  - e.g., After each time step, threads wait for each other

Implementing Barriers

- Information we need for a centralized barrier:
  - How many threads to wait for
  - Flag to tell threads to proceed or wait
  - And… a lock to protect it all

```
struct Barrier_t {
   Lock aLock;
   int counter;
   int flag;
};
```

Implementing Barriers

- Conceptual algorithm:
  1. Clear the flag, tell all incoming threads to wait
  2. Every arriving thread increments `counter`
  3. Last thread sees `counter == num_threads`
  4. Set the flag, all threads leave

First try: Centralized Barrier

- Exercise: Does this work?
  - Hint: think about two barriers in a row

```
void thrBarrier(Barrier_t* b, int num_threads) {
    lock(b->aLock);
    if(b->counter == 0)
        b->flag = 0;                        // (Step 1)
    int num_waiting = ++(b->counter);       // (Step 2)
    if( num_waiting == num_threads ) {      // (Step 3)
        unlock(b->aLock);
        b->counter = 0;
        b->flag = 1;                          // (Step 4)
    } else {
        unlock(b->aLock);
        while( b->flag == 0 ); // spin
    }
}
```

First try: Centralized Barrier

- Exercise: Does this work?
  - Answer: No! A thread can set the flag, and continue on to the next barrier, which will clear the flag.
  - Not all other threads might have “left” the first one!
    Therefore, this might deadlock.

```
...
thrBarrier(aBarrier, Nthreads);
    // flag is set, but not all threads may see it!
...
thrBarrier(aBarrier, Nthreads);
    // flag cleared, no chance for others to leave
```

Solution: Private Sense Reversal

- The problem was the lead thread resetting `flag` while all others spinning on it
  - Toggled 0 → 1 → 0, but others may have missed that
- Keep a private copy of `flag`, first thread to arrive writes its private copy
  - Prevents the problem of missing the write

```
struct Barrier_t {
   Lock aLock;
   int counter;
   int flag;
};
int local_sense = 0; // private
```

Sense Reversal Implementation

- Recommended: draw this out and convince yourself that it works!

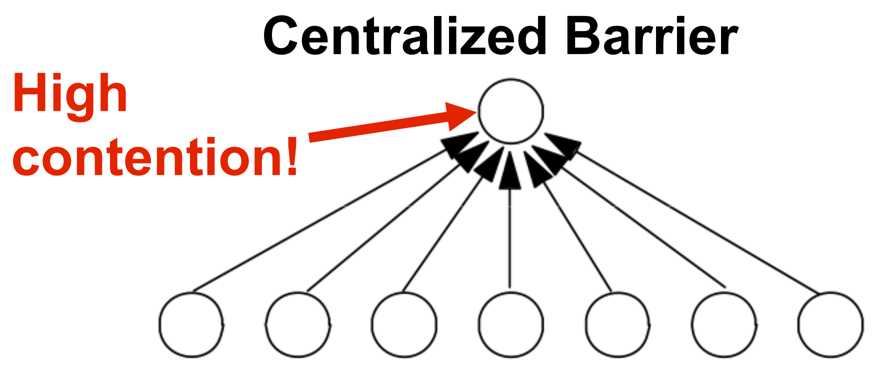
```
void senseBarrier(Barrier_t* b, int num_threads) {
    local_sense = !(local_sense);         // (Step 1)
    lock(b->aLock);
    int num_waiting = ++(b->counter);    // (Step 2)

    if( num_waiting == num_threads ) { // (Step 3)
        unlock(b->aLock);
        b->counter = 0;
        b->flag = local_sense;                // (Step 4)
    } else {
        unlock(b->aLock);
        while( b->flag != local_sense ); // spin
    }
}
```

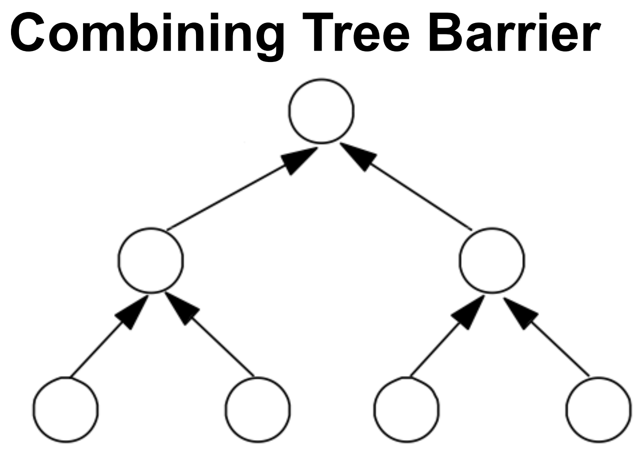
Centralized Barrier Traffic Analysis

- O(P) traffic on a bus:
  - 2P write transactions to obtain barrier lock and update counter
  - 2 write transactions to write flag + reset counter
  - P-1 transactions to read updated flag
- Still we serialize on a single shared variable
  - Latency is O(P)
  - Can we do better?

Combining Trees

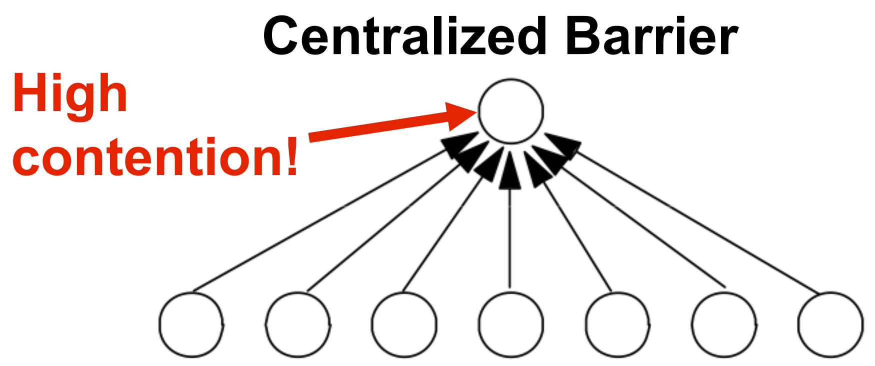
- Combining trees make better use of parallelism in interconnect topologies
  - log(P) latency
  - Strategy makes less sense on a bus
    (all traffic still serialized on single shared bus)

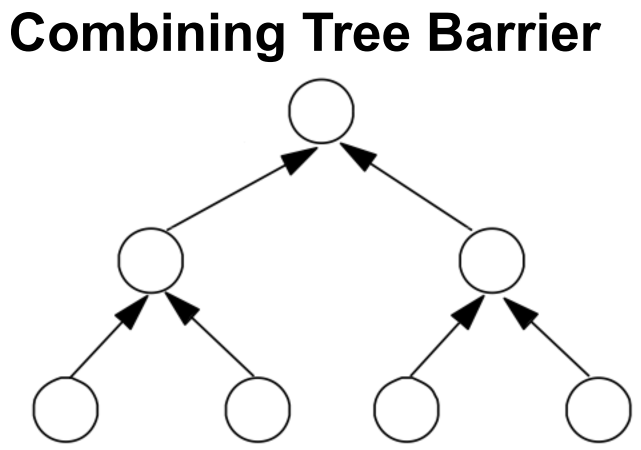




Combining Trees

- Acquire: when processor arrives at barrier, performs atomicIncr() of parent counter
  - Process recurses to root
- Release: beginning from root, notify children of release





Synchronization in OpenMP

- OMP has support for all of the synchronization primitives we talked about in this lecture
  - Locks, atomic operations (L6) and barriers

Critical Sections and Atomics

- Critical Section (using Locks)
  - A portion of code that only 1 thread at a time may execute

```
#pragma omp critical
{
  /* Critical code here */
}
```

- Atomic Execution
  - Protects a single variable update

```
#pragma omp atomic
/* Update statement here */
```

OMP Barriers

- Barrier
  - Performs a barrier synchronization between all the threads in a team at a given point
  - All threads wait at the barrier point and only continue when all threads have reached the barrier point

```
#pragma omp parallel {
   int result = heavy_computation_part1();

   #pragma omp atomic
   sum += result;

   #pragma omp barrier
   heavy_computation_part2(sum);
}
```

OMP Barriers

- Barrier
  - Performs a barrier synchronization between all the threads in a team at a given point
  - All threads wait at the barrier point and only continue when all threads have reached the barrier point

```
#pragma omp parallel {
   int result = heavy_computation_part1();

   #pragma omp atomic
   sum += result;

   #pragma omp barrier
   heavy_computation_part2(sum);
}
```

Single-Threaded Code

- Single-threaded region within a parallel region
  - `master` and `single` directives

```
#pragma omp single
{
  /* Only executed once */
}
```

```
#pragma omp master
{
  /* Only executed by master*/
}
```

OpenMP Locks

- A lock in OpenMP is an object (`omp_lock_t`)
  - At most one thread can hold it at a time
- `omp_init_lock(omp_lock_t *)`
  - Initializes the lock variable passed in
  - The lock is not held by the initializing thread
- `omp_destroy_lock(omp_lock_t*)`
  - Disassociates the given lock variable from any locks

OpenMP Locks

- A lock in OpenMP is an object (`omp_lock_t`)
  - At most one thread can hold it at a time
- `omp_set_lock(omp_lock_t*)`
  - Waits until the lock is available, then acquires the lock
- `omp_unset_lock(omp_lock_t*)`
  - Unsets (releases) the lock
- `omp_test_lock(omp_lock_t*)`
  - Attempts to set a lock, but does not block if the lock is unavailable

OpenMP Locks - Example

```
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel num_threads(4)
{
  int tid = omp_get_thread_num( );
  int i;
  for (i = 0; i < 3; ++i){
    omp_set_lock(&lock);
    printf("T %d: begin locked region\n", tid);
    printf ("T %d: end locked region\n", tid);
    omp_unset_lock(&lock);
  }
}

omp_destroy_lock(&lock);
```

OpenMP Locks - Example

```
T 0: begin locked region
T 0: end locked region
T 0: begin locked region
T 0: end locked region
T 2: begin locked region
T 2: end locked region
T 2: begin locked region
T 2: end locked region
T 1: begin locked region
T 1: end locked region
T 1: begin locked region
T 1: end locked region
T 3: begin locked region
T 3: end locked region
T 3: begin locked region
T 3: end locked region
```

Summary

- HW can enable declarative concurrency control
  - Transforms implementation (locks) into intention (transactions)
  - Programs can more easily manage concurrency

- Barriers, an example of point-to-point synch.
  - Use locks as a part of their implementation
  - Need to be careful of correctness, and traffic