Lect. 8: Synchronization

- Synchronization is necessary to ensure that operations in a parallel program happen in the correct order
  - Condition synchronization
  - Mutual exclusion
- Different primitives are used at different levels of abstraction
  - High-level (e.g. monitors, parallel sections and loops): supported in languages themselves or language extensions (e.g, Java threads, OpenMP)
  - Middle-level (e.g., locks, barriers, and condition variables): supported in libraries (e.g., POSIX threads)
  - Low-level (e.g., compare&swap, test&set, load-link & store-conditional, transactional memory): supported in hardware
- Higher level primitives can be constructed from lower level ones
- Things to consider: deadlock, livelock, starvation

Example: Sync. in Java Threads

- Synchronized Methods
  - Concurrent calls to the method on the same object have to be serialized
  - All data modified during one call to the method becomes atomically visible to all calls to other methods of the object
  - E.g.:

```
public class SynchronizedCounter {
  private int c = 0;

  public synchronized void increment() {
    c++;
  }
}

SynchronizedCounter myCounter;
```

  - Can be implemented with locks

Example: Sync. in OpenMP

- Doall loops
  - Iterations of the loop can be executed concurrently
  - After the loop, all processors have to wait and a single one continues with the following code
  - All data modified during the loop is visible after the loop
  - E.g.:

```
#pragma omp parallel for \
        private(i,s) shared (A,B)\
        schedule(static)
for (i=0; i<N; i++) {
  s = …

  A[i] = B[i] + s;
}
```

  - Can be implemented with barrier

Example: Sync. in POSIX Threads

- Locks
  - Only one thread can own the lock at any given time
  - Unlocking makes all the modified data visible to all threads and locking forces the thread to obtain fresh copies of all data
  - E.g.:

```
pthread_mutex_t mylock;

pthread_mutex_init(&mylock, NULL);
pthread_mutex_lock(&mylock);

Count++;

pthread_mutex_unlock(&mylock);
```

  - Can be implemented with hardware atomic RMW (e.g. test&set)

Example: Building Locks from Ld/St?

- E.g., Peterson's algorithm

```
            Processor 0                                   Processor 1
int A, B, C;
Int flag[2], turn;

flag[0]=0; flag[1]=0;
turn = 0;
-------------------------------------------------------------------------
/* lock */                                  /* lock */
flag[0] = 1; turn = 1;                      flag[1] = 1; turn = 0;
While(flag[1]&&turn==1);                    While(flag[0]&&turn==0);


/* unlock */                                /* unlock*/
flag[0] = 0;                                flag[1] = 0;
```

initialization

parallel

Example: Building Locks from Ld/St?

- Requires SC.
  - – Relaxed models need to use fences.
- Works for only 2 processors
- A general N processor solution (for e.g. Bakery algorithms) requires O(N) flag variables and it also slow.

Building Locks with Hdw. Primitives

- Example: Test&Set

```
int lock(int *mylock) {

  int value;

  value = test&set(mylock,1);
  if (value)
    return FALSE;
  else
    return TRUE;
}

void unlock(int *mylock) {
  *mylock = 0;
   return;
}
```

Hardware Primitives

- Hardware's job is to provide atomic memory operations, which involves read and write to a memory location atomically.
- Also called as a Read-Modify-Write (RMW) instructions.
- Implemented in the IS, but usually encapsulated in library function calls by manufacturers
- At a minimum, hardware must provide an atomic swap (or test&set), but there are more sophisticated ones

Compare&Swap

– Compare&Swap (e.g., Sun Sparc): if value in memory is equal to value in register R2 then swap memory value with the value in R3

```
CAS   (R1),R2,R3 : if (MEM[R1]==R2)
                        MEM[R1]<->R3;
```

- Can implement more complex conditions for synchronization
- The compare and the swap must be performed atomically

```
 int compare_and_swap(int *addr, int value, int new_value)
{
        ATOMIC_BEGIN();
        int old_value = *addr;
        if(old_value == value)   *addr = new_value;
        ATOMIC_END();
        return old_value;
}
```

Fetch&Add

– Fetch&Increment (e.g., Intel x86) (in general Fetch&Op): increment the value in memory and return the old value in register

```
lock; ADD   (R1),R2, R3 : R3 = MEM[R1]; MEM[R1]=MEM[R1]+R2
```

- Less flexible than Compare&Swap
- The fetch and addition must be done atomically.

```
 int fetch_and_add(int *addr, int increment)
{
        ATOMIC_BEGIN();
        int old_value = *addr;
        *addr = *addr + increment;
        ATOMIC_END();
        return old_value;
}
```

Test&Set (or Swap)

- Swap (test-and-set): swap the values in memory and in a register
  - Less flexible of all
  - swap must be performed atomically

```
lock; ADD   (R1),R2, R3 : R3 = MEM[R1]; MEM[R1]=R2
```

```
 int test_and_set(int *addr, int new_value)
{
        ATOMIC_BEGIN();
        int old_value = *addr;
        *addr = new_value;
        ATOMIC_END();
        return old_value;
}
```

Why a bunch of Read-modify-Write instr.?

- Are each of these instructions equal in "power" in synchronisation situations?
- Or are some instructions more powerful than others?

Consensus

- Bunch of threads from 1..n, each thread proposes a value, propose[i]
- Consensus problem: can the threads agree on a value?
    - Need to select a winner thread
    - The winner must know.
    - The losers must also know the identity of the winner
    - Abstracts the mutual exclusion problem
- With Compare&Swap:
    - The winner can swap their own thread id into consensus variable.
    - Losers can't modify the consensus variable.
    - The losers will know who won.
    - The swap must happen only for the winner (and hence CAS works)

Implementing RMWs

- Need to guarantee atomicity of R and W
- Lock the bus until the R and W performs
  - Early implementation.
  - No other processor can issue memory requests until the RMW completes.
  - Slow (impacts other processors too)
- Cache line locking
  - Obtain exclusive access by doing a Read exclusive (i.e invalidates other cache lines and obtain block in modified state).
  - Deny coherence requests to that line until W completes

Implementing RMWs

- Need to guarantee atomicity of R and W
- Cache line locking
  - Obtain exclusive access by doing a Read exclusive (i.e invalidates other cache lines and obtain block in modified state).
  - Deny coherence requests to that line until W completes
- Interaction with the memory model?
  - E.g. TSO, there might be older writes in the write buffer.
  - Typically the write buffer is flushed before the RMW
  - Cost of RMW includes a fence-like write buffer drain.

Building Locks with Hdw. Primitives

- Example: Test&Set

```
int lock(int *mylock) {

  int value;

  value = test&set(mylock,1);
  if (value)
    return FALSE;
  else
    return TRUE;
}

void unlock(int *mylock) {
  *mylock = 0;
   return;
}
```

What If the Lock is Taken?

- Spin-wait lock

```
while (!lock(&mylock));
…
unlock(&mylock);
```

  – Each call to lock invokes the hardware primitive, which involves an expensive memory operation and takes up network bandwidth

- Spin-wait on cache: Test-and-Test&Set
  – Spin on cached value using normal load and rely on coherence protocol

```
while (TRUE) {
  if (!lock(&mylock))
    while (!mylock);
  else break;
}
…
unlock(&mylock);
```

  – Still, all processors race to memory, and clash, once the lock is released

What If the Lock is Taken?

- Software solution: Blocking locks and Backoff

```
while (TRUE) {
  if (!lock(&mylock)) wait (time);
  else break;
}
…
unlock(&mylock);
```

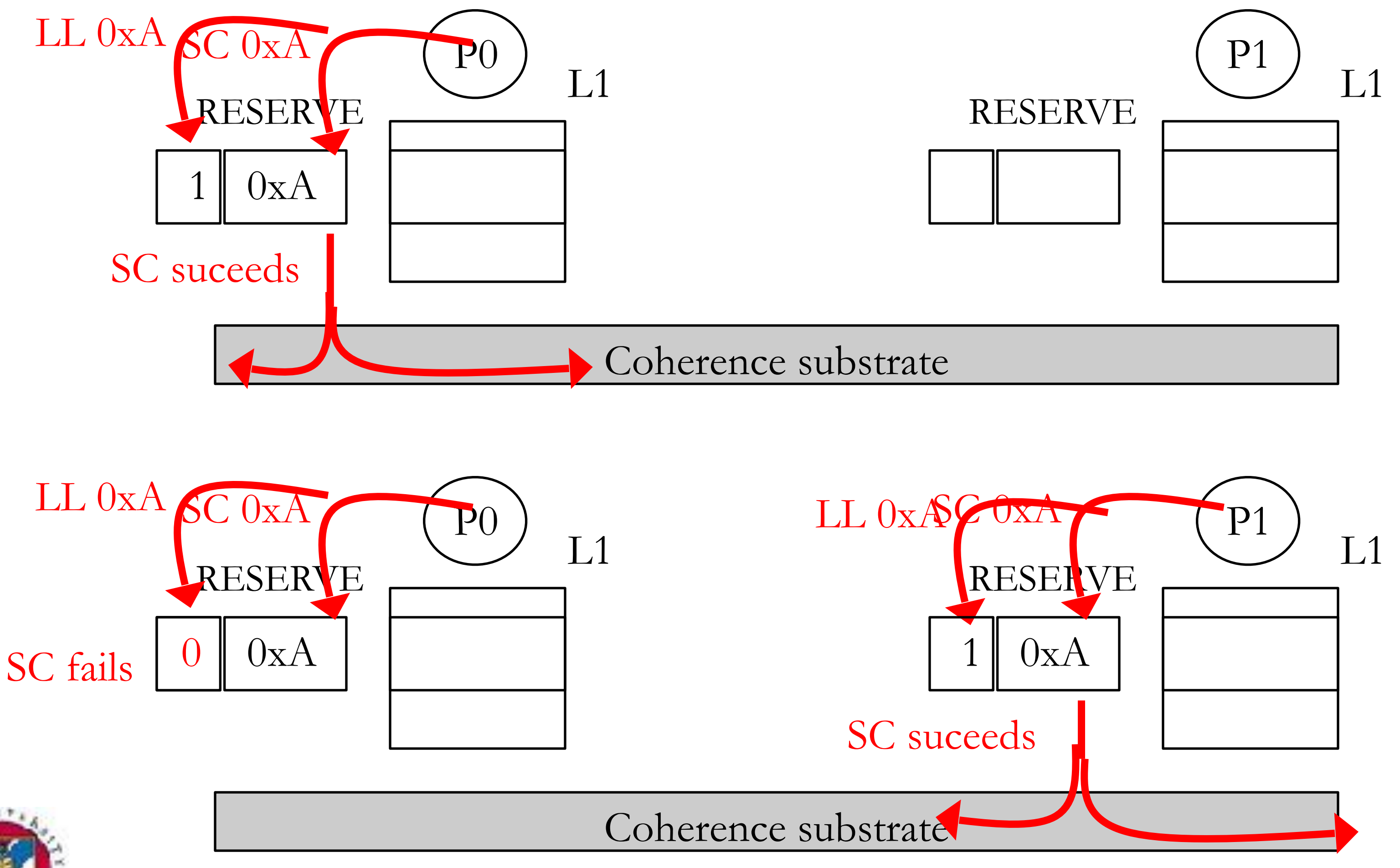
  - Wait can be implemented in the application itself (backoff) or by calling the OS to be put to sleep (blocking)
  - The waiting time is usually increased exponentially with the number of retries
  - Similar to the backoff mechanism adopted in the Ethernet protocol

LL/SC

- Load-link and Store-conditional
  - Implement atomic memory operation as two operations
  - Load-link (LL):
    - Registers the intention to acquire the lock
    - Returns the present value of the lock
  - Store-conditional (SC):
    - Only stores the new value if no other processor attempted a store between our previous LL and now
    - Returns 1 if it succeeds and 0 if it fails
  - Relies on the coherence mechanism to detect conflicting SC's
  - All operation is done locally at the cache controllers or directory, no need for complex blocking operation in memory
  - Introduced in the MIPS processor, now also used in PowerPC and ARM

Another Hardware Primitive

- Load-link and Store-conditional operation



Building Locks with LL/SC

- E.g., spin-wait with attempted swap

```
try:   MOV  R2, lock
       LL   R1, location  ; value of lock loaded
       BNZ  R1, try       ; try again if lock taken
       SC   R2, location  ; try to store conditionally
       BEQZ R2, try       ; branch if SC failed
       RET
```

Advantages:

- Lesser complexity on coherence system – does not suffer from deadlocks
- Failing SC does not send invalidates
- Lends itself naturally to test-and-test&set like implementation

Transactional Memory (TM)*

- Coarse-grain locking is easy but limits concurrency
- Fine-grain locking is efficient but hard to get right
- Can we get the performance of the latter with the programmability of the former?
- TM
  - Atomic read-modify-writes for sections of code (think ll/sc for a bunch of memory addresses)
  - First proposed in the context of database transactions
  - First proposed as replacement for locks by Herlihy and Moss in 1993
  - Intel Haswell architecture has TM implementation

With thanks to Christos Kozyrakis for some of the content

HashMap

```
get(Object key)
{
   int id = hash (key);
   HashEntry e = buckets[id];
   while(e!=NULL)
   {
       if (key == e.key) return
    e.value;
      e = e.next;
   }
}
```

```
  put(Object key, Type value)
{
  int id = hash (key);
  HashEntry e = buckets[id];
  while(e!=NULL)
  {
      prev = e;
      e = e.next;
      if (key == e.key) {
         e.value = value;
         return success;
      }
  }
  add_entry(e, key, value);
}
```

- Given key returns value if found
- Not thread-safe. Why?

Making HashMap thread safe

- Grab a mutex before entering get, put
  - Coarse-grain locking
  - Easy to program
  - Limits concurrency
- Redesign using per-bucket locks
  - Fine-grain locking
  - Error prone and complex
- Use TM
  - Simply enclose get and put within Atomic
  - System ensures atomicity.

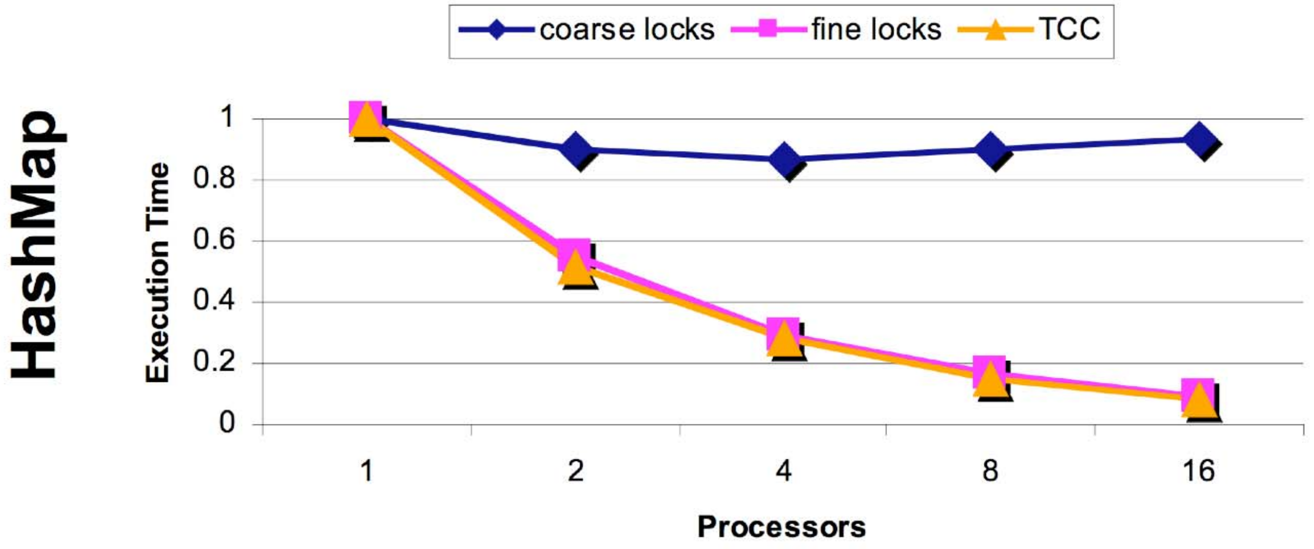
HashMap using TM

```
get(Object key)
{
 ATOMIC
 {
   int id = hash (key);
  HashEntry e = buckets[id];
   while(e!=NULL)
   {
       if (key == e.key) return e.value;
      e = e.next;
   }
  }
}
```

```
  put(Object key, Type value)
{
 ATOMIC
 {
  int id = hash (key);
  HashEntry e = buckets[id];
  while(e!=NULL)
  {
      prev = e;
      e = e.next;
      if (key == e.key) {
         e.value = value;
         return success;
      }
  }
  add_entry(e, key, value);
 }
}
```

Performance

TCC: A Hardware TM system



- Hardware TM as good as fine-grained locking!

What does TM guarantee?

- Atomicity
  - If and when a transaction commits, all writes appear to take effect at once
  - If and when a transaction aborts, none of the writes appear to take effect.
- Isolation
  - No other code can observe writes, until the transaction completes successfully.
- Serializability
  - Transactions must appear to commit in a single serial order.
  - Transactions must appear in this order in program order.

Advantages of TM

- Programmability
  - As easy to use as coarse-grained locks.
  - Programmer does not need to worry about how to enforce atomicity.
  - Composability: Safe to compose transactions.
  - Failure atomicity: No explicit undo necessary on exception, simply abort.

- Performance
  - Allows for fine-grained concurrency
  - Performs as well as fine-grained locks.

TM Caveats

- All locks may not be converted to atomic transactions

```
//Thread 1                       //Thread 2
synchronized(lock1){             synchronized(lock2){
  …                                …
  flagB = true;                    flagA = true;
  while (flagA==0);                while (flagB==0);
  …                                …
}                                }
```

- Hard to Undo output and Redo Input

- Semantics of interaction of transactional and non-transactional code tricky

How system ensures Atomicity?

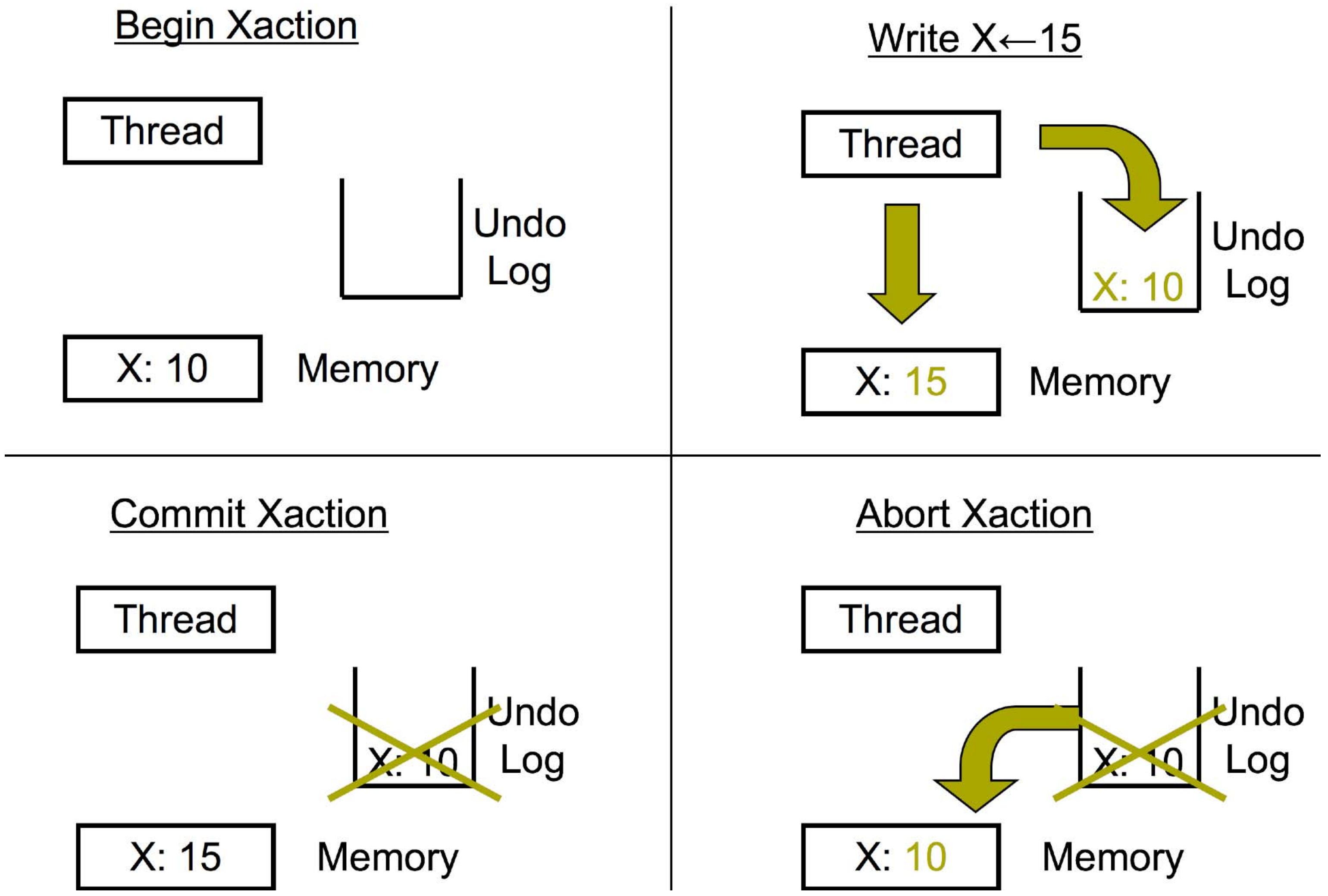
- TM implementation must provide
  - Versioning: the ability to recover in case transaction does not succeed by either buffering new values or logging old values.
  - Conflict detection: the ability to detect if two transactions are conflicting (if one modifies locations that are read/modified by the other).
  - HTM: If versioning and conflict detection in HW
  - STM: If versioning and conflict detection in SW

Versioning

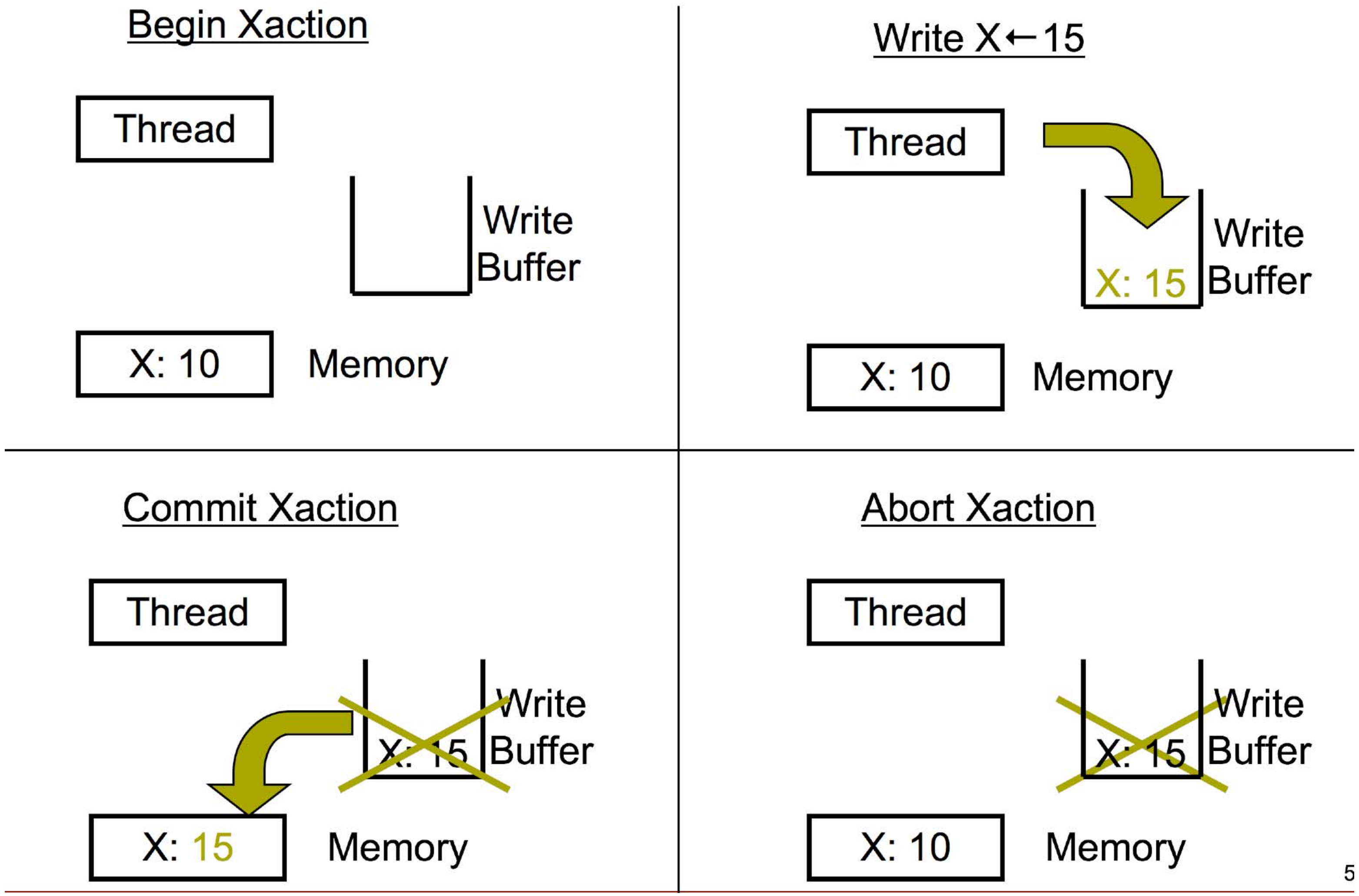
- Eager Versioning (undo-log based)
  - Update memory directly.
  - Maintain old values in a log.
  - Faster commits (discard log), direct reads (relevant in STM)
  - But slower aborts.

- Lazy Versioning (write-buffer based)
  - Buffer data until commit in a write-buffer
  - Update actual memory on commit
  - Fast aborts (discard write buffer)
  - But slower commits and indirect reads (STM)

Eager Versioning Example



Lazy Versioning Example

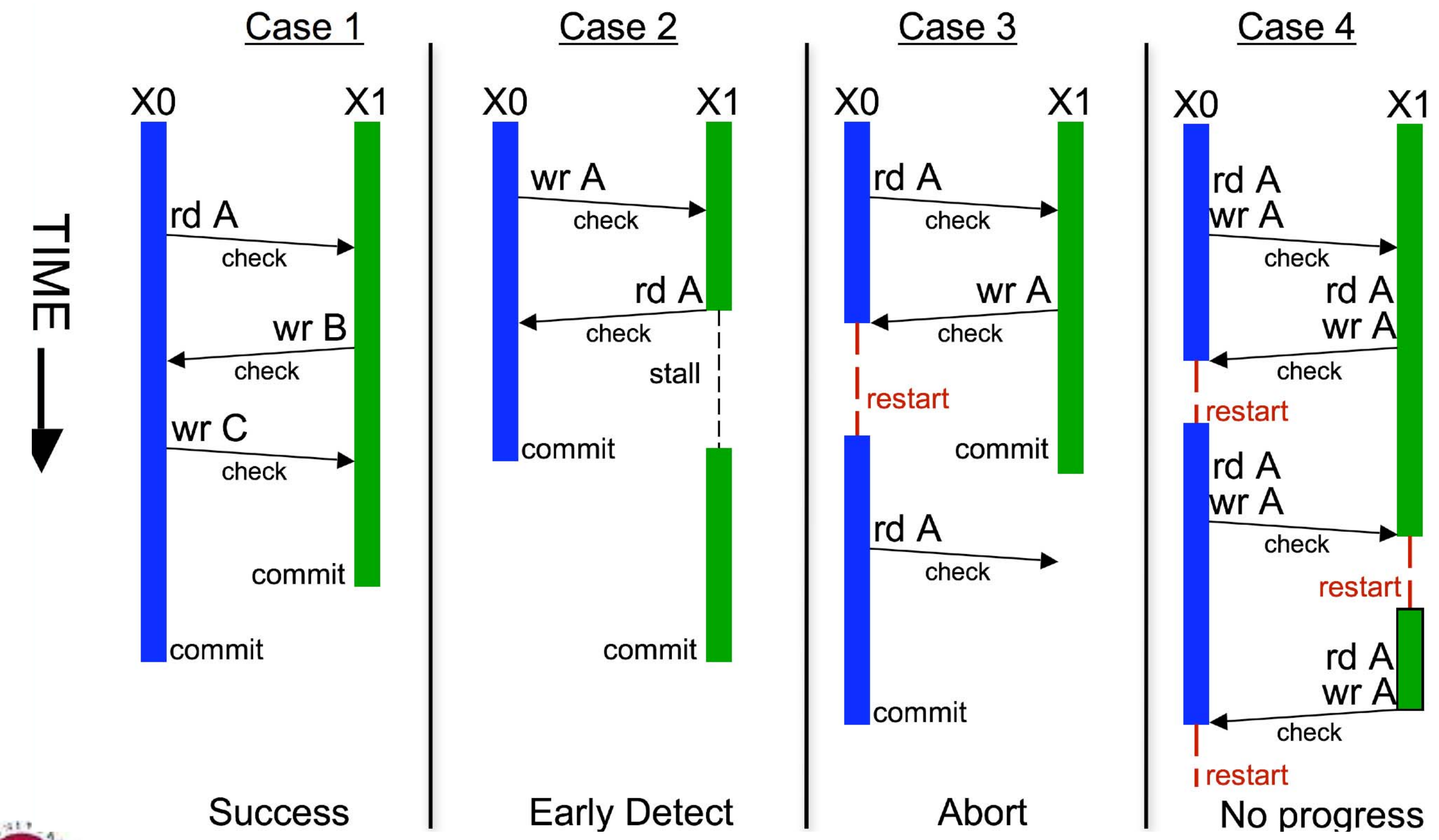


Conflict detection

- Detect and handle conflicts between transactions
  - R/W and W/W conflicts
  - Must keep track of read-set (addresses read) and write-set (addresses written) of transactions

- Pessimistic (or eager) detection
  - Check for conflicts during loads or stores
    - STM: by instrumenting loads and stores with locks and version numbers
    - HTM: leverage coherence protocol.
  - Contention manager: upon detecting a conflict, either stall or abort

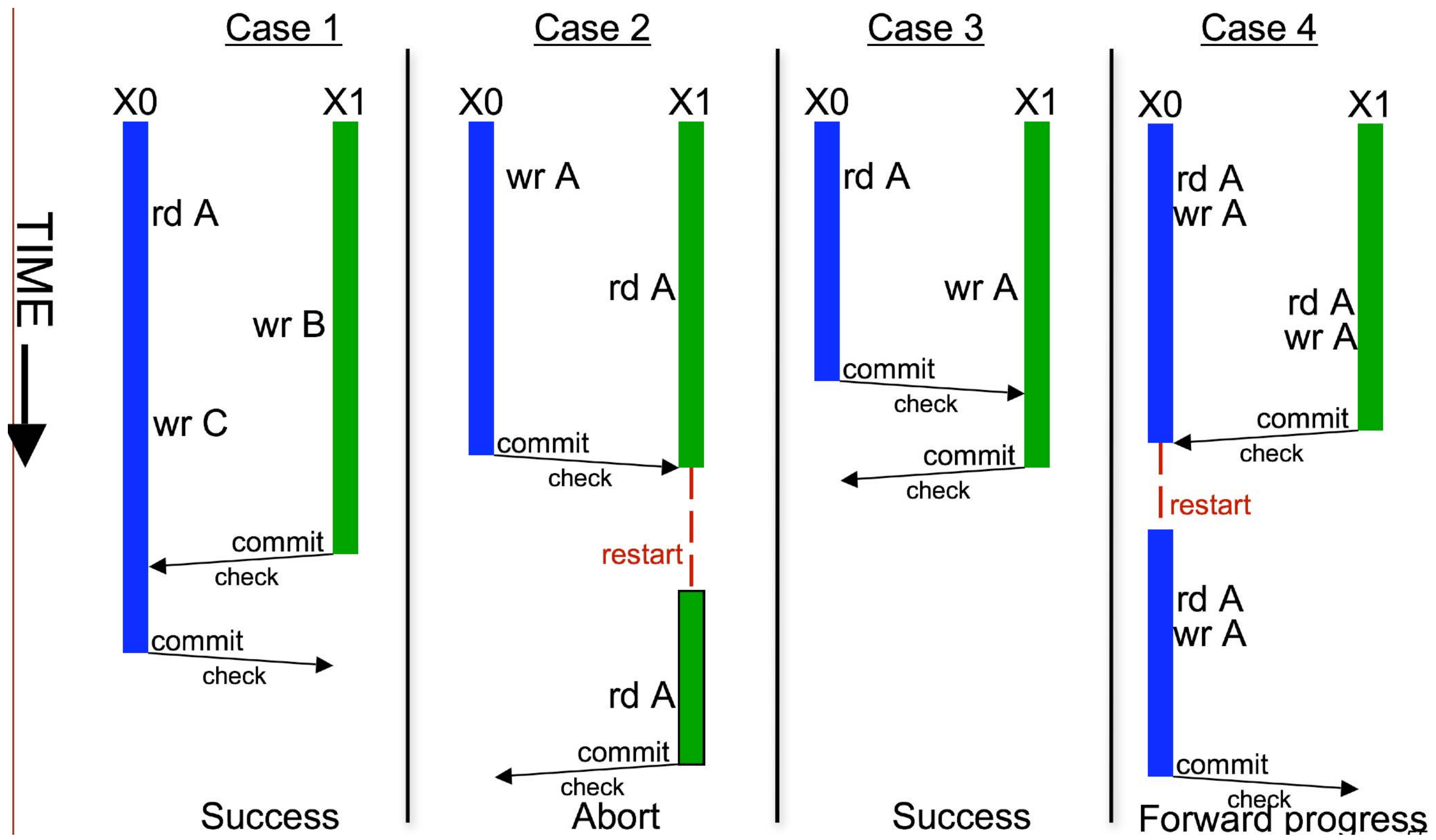
Pessimistic Detection Example



Conflict detection

- Optimistic (or lazy) detection
  - Check for conflicts when a transaction is about to commit
    - STM: by instrumenting loads and stores with locks and version numbers
    - HTM: leverage coherence protocol; validate write set by obtaining exclusive access to write-set.
  - Contention manager: upon detecting a conflict, give priority to committing transaction.

Optimistic Detection Example



HTM

- Data versioning in caches
  - Cache the write-buffer or the undo log
  - Cache metadata to track read and write set.
    - Read/write bits for each cache line set on loads/stores
    - Gang cleared on transaction commit or abort
    - Replacements cause an abort!

- Conflict detection through coherence protocol
  - Coherence lookups detects conflicts.
    - Requests check R/W bits
  - Works for both snooping and directory protocols.

Intel's HTM: RTM

- Restricted Transactional Memory
  - xbegin: Begin transaction (also provide offset to fallback instr.)
  - xend: End transaction
  - xabort: User controlled abort of transaction
    - On abort, control transfers to fallback
    - E.g. At fallback PC, programmer can have coarse-grained lock version
- Implementation
  - Write-buffer based (L1 cache)
  - Conflict detection using coherence protocol.

References

- Transactional memory

  James Larus and Christos Kozyrakis. 2008. Transactional memory. Commun. ACM 51, 7 (July 2008), 80-88.

- A commercial machine with Full/Empty bits:

  "The Tera Computer System", R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, Intl. Symp. on Supercomputing, June 1990.

- Performance evaluations of synchronization for shared-memory:

  "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", T. Anderson, IEEE Trans. on Parallel and Distributed Systems, January 1990.

  "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", J. Mellor-Crummey and M. Scott, ACM Trans. on Computer Systems, February 1991.