Parallelism: Synchronization Revisited

Todd C. Mowry & Dave Eckhardt

I. Synchronization on a Parallel Machine

II. Transactional Memory

Recall: Intel's `xchg` Instruction

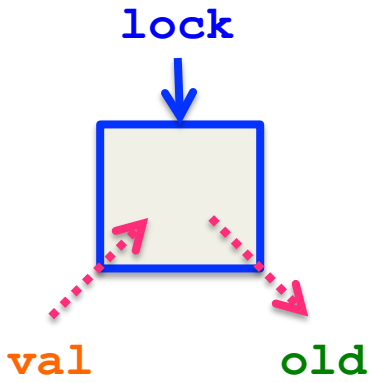
In assembly:

```
xchg (%esi), %edi
```

Functionality:

```
int32 xchg(int32 *lock, int32 val) {
  register int old;
  old = *lock;
  *lock = val;
  return (old);
}
```

} occur *atomically*



- *atomically* read the old value and store a new value
  - at the location pointed to by the lock pointer
- returns the old value
  - (by storing it in the register that contained the new value)

Recall: Using **xchg** to Implement a Lock

- Initialization:

```
int lock_available = 1;        // initially available
```

- Grabbing the lock:
  - "Try-lock" version:

```
i_won = xchg(&lock_available, 0);    // unavailable after this
```

  - Spin-wait version:

```
while (!xchg(&lock_available, 0)     // unavailable after this
    continue;
```

- Unlock:

```
xchg(&lock_available, 1);      // make lock available again
```

How Does **`xchg`** Actually Work?

- Complication:
  - fundamentally, this involves both a load and a store to a memory location
    - and these things can't occur simultaneously!
- How x86 processors handle complex instructions:
  - the hardware translates x86 instructions into simpler µop instructions
  - e.g., "**`add (%esi), %edi`**" actually turns into 3 µops:
    1. load (**`%esi`**) into a hardware register
    2. add **`%edi`** to that hardware register
    3. store result into (**`%esi`**)
- Hence at the µop level, "**`xchg (%esi), %edi`**" turns into:
  1. load (**`%esi`**) into a hardware register
     - *(through hardware register renaming, this eventually ends up in **`%edi`**)*
  2. store **`%edi`** into (**`%esi`**)
- Question: how do you think (MESI) cache coherence handles this sequence?
  - Answer: need to get & hold the block *exclusively* throughout the sequence

If Lock Is Not Available, Should We Spin or Yield?

Spin-Waiting:

```
while (!xchg(&lock_available, 0)
  continue;
```

Yielding (aka Blocking):

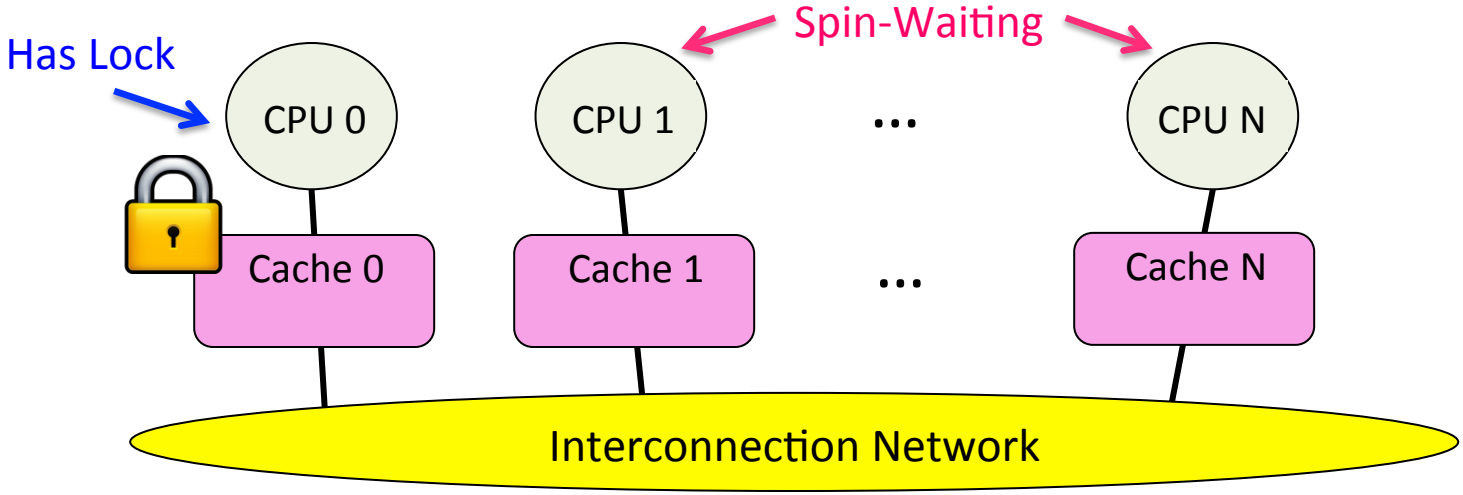
```
while (!xchg(&lock_available, 0)
  // go to sleep until someone does an unlock
  // (or at least yield)
  run_somebody_else();
```

- Uniprocessor (review):
  - Who has the lock? Another thread, that currently is not running!
  - So spinning would be a waste of time.
- Multiprocessor:
  - Who has the lock?
    - Another thread, that is hopefully currently running (on a different processor)!
  - Also, parallel programmers try not to hold locks for very long
    - so hopefully it will become available soon, and we want to grab it ASAP
  - Spin-waiting may be attractive!
    - unless of course the thread holding the lock gets de-scheduled for some reason
    - common approach: spin for a while, and eventually yield
- No simple answer: depends on # of CPUs, and which threads are running on them

Memory System Behavior for High-Contention Locks

- What if all processors are trying to grab the same lock at the same time:

```
while (!xchg(&lock_available, 0)
  continue;
```



- What will the coherence traffic across the interconnection network look like?
- As each processor spin-waits, it repeatedly:
  - requests an exclusive copy of the cache block, invalidating the other caches
  - checks whether the lock is available, and finds that it is not

    → constant stream of cache misses and coherence traffic: very bad!

Improved Version: "Test and Test-and-Set" Lock

```
do {
  while (lock_available == 0)          // "Test" loop
    continue;
} while (!xchg(&lock_available, 0));   // "Test-and-Set" check
```

- *(In the synchronization literature, our* ***xchg*** *lock is called a "test-and-set" lock.)*
- Note that the "test" loop uses a normal memory load (not an **xchg**)
- How does the coherence traffic change with this modification?
  - while the lock is held, the other processors spin locally in their caches
    - using normal read operations in the "test" loop, which hit on the "Shared" block
  - so there is no longer a flood of coherence traffic while the lock is held
- While this reduces traffic while the lock is held, have we solved all problem cases?
  - what happens when the lock is released?

    → a sudden burst of "test-and-set" attempts, with all but one of them failing

Avoiding the Burst of Traffic When a Lock is Released

- One approach: use backoff
  - upon failing to acquire lock, delay for a while before retrying
    - either constant delay or exponential backoff

- The good news:
  - significantly reduces interconnect traffic

- The bad news:
  - exponential backoff can lead to starvation for high-contention locks
    - new requestors back off for shorter times
  - even without starvation, seriously non-FIFO lock acquisition is likely

- Exponential backoff seems to help performance in practice.

Ticket Lock

Two counters:

Both initialized to 0.

next_ticket
(# of requestors)

now_serving
(# of releases that have happened)

Lock:

```
my_ticket = atomic_fetch_and_increment(&next_ticket);
while (now_serving != my_ticket)
   continue;
MFENCE;
```

Can be implemented with Load-Linked/Store-Conditional

Unlock:

```
MFENCE;
now_serving++;
```

Regular (non-atomic) operations.

- What is the coherence traffic like upon an unlock?
  - an invalidation, and then a read miss for each spinning processor
- Possible solution: use delay while spinning (but by how much?)

Ticket Lock Tradeoffs

The good news:

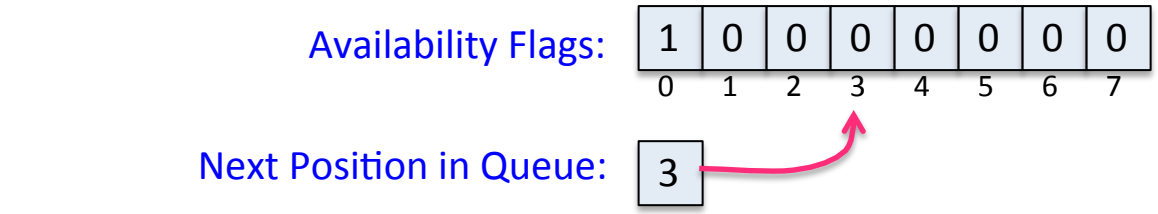
- guaranteed FIFO order
  - so starvation is not possible
- traffic can be quite low

But could it be better still?

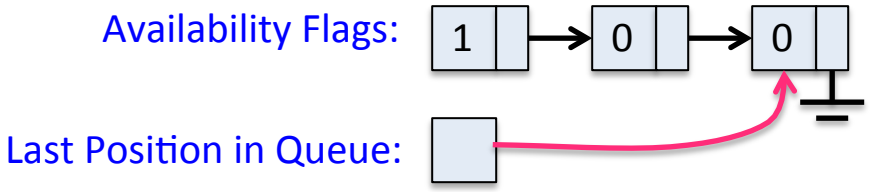
- traffic is not guaranteed to be O(1) per lock handoff

Achieving O(1) Traffic: Queueing Locks

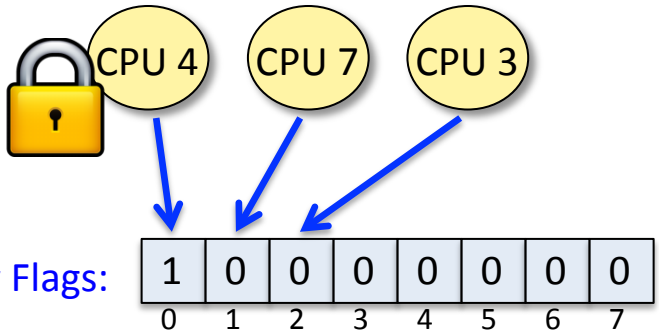
- Basic Idea:
  - pre-determine the order of lock handoff via a queue of waiters
  - during an unlock, the next thread in the queue is directly awakened
    - set a flag variable corresponding to the next waiter
    - each thread stares at a different memory location → spin locally in their caches
- Implementations:
  - Array-Based Queueing Locks:



  - List-Based Queueing Locks:



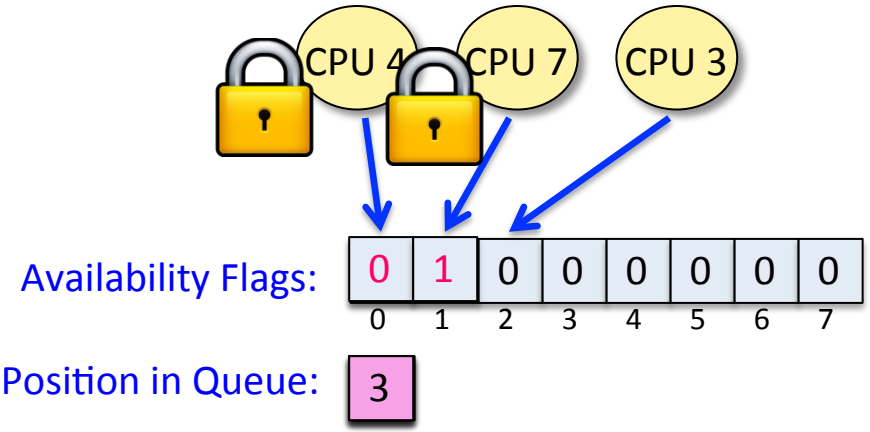
Array-Based Queueing Locks



Lock:

```
my_index = atomic_fetch_and_increment(&next_position)%NUM_PROCESSORS;
while (!lock_available[my_index])
   continue;
MFENCE;
```

Array-Based Queueing Locks



Lock:

```
my_index = atomic_fetch_and_increment(&next_position)%NUM_PROCESSORS;
while (!lock_available[my_index])
   continue;
MFENCE;
```

Unlock:

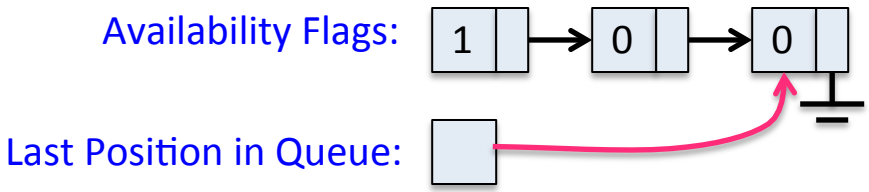
```
MFENCE;
lock_available[my_index] = 0;
next_index = (my_index+1)%NUM_PROCESSORS;
lock_available[next_index] = 1;
```

Tradeoffs:

- Good: FIFO order; O(1) traffic (with cache coherence)
- Bad: requires space per lock proportional to P (x cache line size)

List-Based Queueing Locks

- Proposed by Mellor-Crummey and Scott (called "MCS" locks)



- Same basic idea, but insertions occur at the tail of a linked list.
- Space is allocated on-demand
  - aside from head pointers per lock, need only O(P) space for all locks in total
- Slightly more computation for lock/unlock operations

Which Performs Better: Test-and-Test-and-Set or Queue Locks?

- It depends on the amount of lock contention.
- Low-contention locks:
  - test-and-test-and-set is faster
    - less work to acquire the lock
- High-contention locks:
  - queue-based locks may be faster
    - less communication traffic, especially on large-scale systems

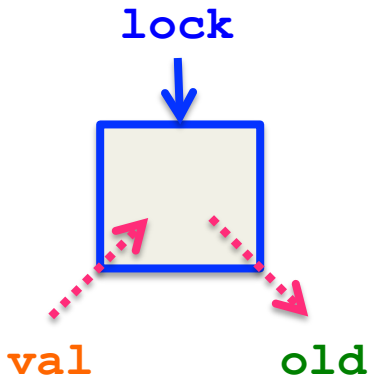
- Hybrid approaches have been proposed
  - switch from one to the other, depending on observed contention

Implementing Atomic Operations in Hardware

- Intel's **xchg** instruction (review):

```
int32 xchg(int32 *lock, int32 val) {
  register int old;
  old = *lock;
  *lock = val;
  return (old);
}
```

} occur *atomically*



  - At the **μop** level, "**xchg (%esi), %edi**" becomes *2 memory operations*:
    1. load (**%esi**) into a hardware register
    2. store **%edi** into (**%esi**)

- Challenges:
  1. Modern pipelines: only perform 1 memory operation per instruction
  2. What if we want slightly fancier functionality?
     - e.g., atomic increment/decrement, compare-and-swap, etc.

Load-Linked / Store Conditional (LL/SC)

- Key Idea:
  - *speculate* that the read-modify-write can occur without getting caught
    - i.e. no other processor could have read/written the block during R-M-W sequence
      - e.g., because the cache block was held in an Exclusive/Dirty state throughout
  - check whether speculation succeeded by monitoring coherence traffic
    - also fails upon context switch, cache line eviction, etc.
  - if speculation fails, then retry
    - Store Conditional (**SC**) returns zero (in source register) if it fails

```
void atomic_add(int *ctr, int delta) {
  do {
    old = LL(ctr);
    new = old + delta;
  } while (!(SC(ctr,new));
}
```

Start tracking `ctr` address

Speculation failed if SC returns zero

Basic Hardware Trick for Implementing Atomicity

1. Bring some data into the cache
2. Perform calculations using that data
3. Store new result to memory

} *Monitor coherence traffic!*

4. Did we get through Steps 1-3 without conflicting remote accesses to the data?
   - If so, then success!
   - If not, then try again.
     - (to avoid livelock, we may eventually retry non-speculatively)

- Observations:
  - Intel's **xchg** does this non-speculatively (for a single memory address)
    - by refusing to give up access to the cache block until it is finished
  - **`LL/SC`** does this speculatively, for a single memory address
  - What if we did this speculatively, for multiple memory addresses?

    → **Transactional Memory**

Wouldn't it be nice if…

- Programmer simply specifies desired outcome:
  - "This code sequence should appear to execute *atomically.*"

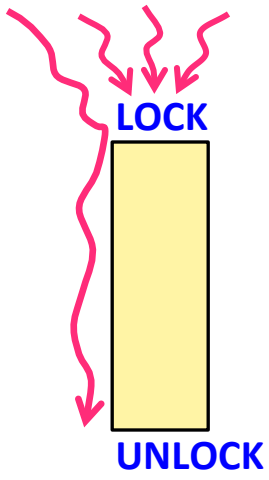
```
void remove_node(Node_type *node) {
    atomic {
        if (node->prev != NULL)
            node->prev->next = node->next;
        if (node->next != NULL)
            node->next->prev = node->prev;
    }
}
```

- The system (e.g., language, run-time software, OS, hardware) makes this happen
  - hopefully *optimistic* (rather than pessimistic) to achieve high performance
  - while enabling *composability* of implementations within abstract objects, etc.

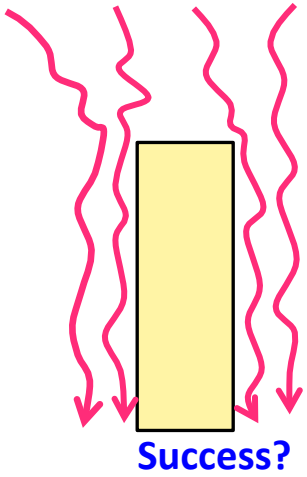
Pessimistic vs. Optimistic Approaches to Atomic Sequences

- Pessimistic approach (e.g., locks, monitors):
  - allow only one thread at a time to execute a potentially-conflicting atomic sequence
- Optimistic approach (e.g., lock-free programming, transactional memory):
  - allow multiple threads to speculatively execute potentially-conflicting atomic sequences; roll-back and retry if speculation fails

Pessimistic



Optimistic



Intel's Transactional Synchronization Extensions (TSX)

Restricted Transactional Memory (RTM):

- **XBEGIN** / **XEND:** specify beginning and end of transaction

```
void remove_node(Node_type *node) {
   atomic {  /* XBEGIN */
      if (node->prev != NULL)
         node->prev->next = node->next;
      if (node->next != NULL)
         node->next->prev = node->prev;
   }          /* XEND */
}
```

- Transactions may abort due to *conflict* or *explicit abort* instruction (**XABORT**)
- If transaction does abort:
  - jump to target specified by the XBEGIN operand
    - abort information is returned in %eax

Source: Ravi Rajwar, Martin Dixon, "Intel Transactional Synchronization Extensions", IDF 2012.

Simple RTM Example: Implementing Locks

acquire_lock(&mutex):

```
Retry:   xbegin Abort  // Enter RTM execution, Abort is fallback path
         cmp mutex, 0  // Check to see if mutex is free
         jz Success
         xabort $0xff  // Abort transactional memory if mutex busy
Abort:
         // check EAX and do retry policy
         // (actually acquire lock or wait to retry)

Success: …
```

release_lock(&mutex):

```
         cmp mutex, 0  // If mutex not free, then was not RTM execution
         jz Commit
         mov mutex, 0  // non-RTM unlock (for compatibility)
Commit:  xend          // commit RTM execution
```

- Can be used for other transactional operations, of course (beyond locks)

Source: Ravi Rajwar, Martin Dixon, "Intel Transactional Synchronization Extensions", IDF 2012.

Major Roles of the Hardware in Transactional Memory

1. Detects Conflicts between Transactions
   - typically done at a cache line granularity within L1 caches
     - leveraging cache coherence messages (in a MESI-like scheme)
   - conflict if at least one transaction writes to a location accessed by another
   - if a conflict is detected, then abort transaction
   - what if an accessed cache block is evicted?
     - in many TM designs: transaction aborts (can no longer track conflicts)
     - in TSX: tracking still occurs
2. Buffers Side-Effects until Transaction either Commits or Aborts
   - held within cache in a special state (not visible to other processors)
   - if transaction commits: these blocks all become visible
   - if transaction aborts: these blocks are all invalidated

The size of a transaction is usually limited by cache capacity and associativity!

Summary

- Implementing locks on parallel machines
  - parallel applications often prefer spin-waiting (carefully!) to yielding
  - BUT naïve spin-waiting can result in devastating coherence traffic
- Improvements over "test-and-set" locks:
  - "test and test-and-set": spin in caches with read hits
    - but still a burst of traffic when lock is released
    - backoff: may avoid burst, but what about starvation?
  - ticket locks: FIFO order
  - queuing locks: O(1) traffic (array or list based)
- Transactional Memory:
  - e.g., Intel's TSX instructions
  - enables atomic sequences involving multiple memory locations
    - (think "handful" of locations, not a huge number)