CS510 Concurrent Systems
Class 1b

Spin Lock Performance
Introduction

- Shared memory multiprocessors
  - Various different architectures
  - All have hardware support for mutual exclusion
    - Various flavors of atomic read-modify instruction
    - Can be used directly or to build higher level abstractions

- This paper focuses on spin locks
  - Used to protect short critical sections

- The challenge
  - How to implement scalable, low-latency spin locks on multiprocessors
Multiprocessor Architecture Overview

- Two dimensions:
  - Interconnect type (bus or multistage network)
  - Cache coherence strategy

- Six architectures considered:
  - Bus: no cache coherence
  - Bus: snoopy write through invalidation cache coherence
  - Bus: snoopy write-back invalidation cache coherence
  - Bus: snoopy distributed write cache coherence
  - Multistage network: no cache coherence
  - Multistage network: invalidation based cache coherence
Mutual Exclusion and Atomic Instructions

- Example: Test-and-set instruction
  - A lock is a single word variable with two values
    - 0 = FALSE = not locked
    - 1 = TRUE = locked
  - Test-and-set does the following **atomically**:
    - Load the (old) value of lock
    - Store TRUE in lock
    - If the loaded value was FALSE...
      - Then you got the lock (so continue)
    - If the loaded value was TRUE...
      - Then someone else has the lock (so try again)
Using Test-and-Set in a Spin Lock

- Spin on Test-and-Set

  ```
  while(TestAndSet(lock) = BUSY);
  <critical section>
  Lock := CLEAR;
  ```

- Tradeoff: frequent polling gets you the lock faster, but slows everyone else down!
  - Why?

- If you fix this problem using a more complex algorithm latency may become an issue
Spin on Read Approach

- Spin on read (Test-and-Test-and-Set)
  
  ```
  while(lock=BUSY or TestAndSet(lock)=BUSY);
  <critical section>
  lock := CLEAR;
  ```

- Intended for architectures with per-CPU caches
- Why *should* it perform much better?
- Why doesn't it perform much better?
Why Quiescence is Slow for Spin on Read

- When the lock is released its value is modified, hence all cached copies of it are invalidated
- Subsequent reads on all processors miss in cache, hence generating bus contention
- Many see the lock free at the same time because there is a delay in satisfying the cache miss of the one that will eventually succeed in getting the lock next
- Many attempt to set it using TSL
- Each attempt generates contention and invalidates all copies
- All but one attempt fails, causing the CPU to revert to reading
- The first read misses in the cache!
- By the time all this is over, the critical section has completed and the lock has been freed again!
Spin on TSL vs Spin on Read

Fig. 1. Principal performance comparison: elapsed time (second) to execute benchmark (measured). Each processor loops one million/P times: acquire lock, do critical section, release lock, and compute.
Quiescence Time for Spin on Read

Fig. 2. Time to quiesce, spin on read (microseconds).
Strategies for Improving Performance

- Author presents 5 alternative approaches
  - 4 are based on CSMA-CD network strategies
  - Approaches differ by:
    - Where to wait
    - Whether wait time is determined statically or dynamically

- Where to wait
  - Delay only on attempted set
    - spin on read, notice release then delay before setting
  - Delay after every memory access
    - Better for architectures where spin on read generates contention!
Delay Only on Attempted Set

```c
while(lock=BUSY or TestAndSet(lock)=BUSY)
begin
  while (lock=BUSY); /* spin on read without delay */
  delay(); /* delay before TestAndSet */
end;
<critical section>
```

- Cuts contention and invalidations by adding latency between retries
- Performance is good if:
  - Delay is short and there are few other spinners
  - Delay is long but there are many spinners
Delay in Spin on Read (every access)

while (lock=BUSY or TestAndSet(lock)=BUSY)
    delay();
<critical section>

- Basically, just check the lock less frequently
- Good for architectures in which spin on read generates contention
  - I.e. those without cache coherence
How Long to Delay?

- Statically determined
  - There is no single “right” answer
    - Sometimes there are many contending threads and sometimes there are few/none
  - If all processors are given the same delay and they conflict once they will conflict repeatedly!
    - Except that one succeeds in the event of a conflict (unlike CSMA-CD networks!)

- Dynamically determined
  - Based on what?
  - How can we estimate number of contending threads?
Static Delay on Lock Release

- When a processor notices the lock has been released, it waits a fixed amount of time before trying a Test-And-Set
- Each processor is assigned a different static delay (slot)
- Few empty slots means good latency
- Few crowded slots means little contention
- Good performance with:
  - Fewer slots, fewer spinning processors
  - Many slots, more spinning processors
Overhead vs. Number of Slots

Fig. 4. Spin-waiting overhead (seconds) versus number of slots.
Variable Delay

- Like Ethernet backoff
- If processor “collides” with another processor, it backs off for a greater random interval each time
  - Indirectly, processors base backoff interval on the number of spinning processors

```c
while(lock=BUSY or TestAndSet(lock)=BUSY)
    delay();
    delay += randomBackoff();
<critical section>
```
Problems with Backoff

- Both dynamic and static backoff are bad when the critical section is long: they just keep backing off while the lock is being held.
  - Failing in test-and-set is not necessarily a sign of many spinning threads!
- Maximum time to delay should be bounded
- Initial delay on arrival should be a fraction of the last delay
A Different Approach - Queueing

- Delay-based approaches separate contending accesses in time.
- Queueing separates contending accesses in space
- Naïve approach
  - Insert each waiting process into a queue
  - Each process spins on the flag of the process ahead of it
    - All are spinning on different locations!
    - No cache or bus contention
  - But the queue insertion and deletion operations require locks
    - Not good for small critical sections - such as queue ops!
Queueing

- A more efficient approach
  - Each arriving process uses an atomic read and increment instruction to get a unique sequence number
  - On completion of the critical section a process releases the process with the next highest sequence number
    - How?
      - Each process is spinning reading its own flag (in a separate cache line)
      - On release a process sets the flag of the process behind it in the queue
  - ... But you need an atomic read and increment instruction!
Queueing

Init
flags[0] := HAS_LOCK;
flags[1..P-1] := MUST_WAIT;
queueLast := 0;

Lock
myPlace := ReadAndIncrement(queueLast);
while(flags[myPlace mod P]=MUST_WAIT);
<critical section>

Unlock
flags[myPlace mod P] := MUST_WAIT;
flags[(myPlace+1) mod P] := HAS_LOCK;
Queueing Performance

- Works especially well for multistage networks - each flag can be on a separate module, so a single memory location isn’t saturated with requests
- Works less well if there’s a bus without cache coherence, because we still have the problem that each process has to poll for a single value in one place
- Lock latency is increased (overhead), so poor performance when there’s no contention
Costs on different hardware

- Distributed write coherence
  - All processors can share the same global “next” counter

- Invalidation-based coherence
  - All processors should spin in a different cache line

- Non-coherent multistage network
  - Processes should poll locations in different memory modules

- Non-coherent bus
  - Polling can swamp bus
  - Needs a delay, based on how close to the front a process is
Benchmark Spin-lock Alternatives

Fig. 3. Principal performance comparison: spin-waiting overhead (seconds) in executing the benchmark (measured). Each processor loops one million/P times: acquire lock, do critical section, release lock, and compute.
Spin-waiting Overhead for a Burst

Fig. 5. Spin-waiting overhead in achieving barrier, normalized by the number of processors (microseconds per processor).
Network Hardware Solutions

- **Combining Networks**
  - Combine requests to same lock (forward one, return other)
  - Combining increases with increase in contention

- **Hardware Queuing**
  - Blocking enter and exit instructions queue processes at memory module
  - Eliminate polling across the network

- **Goodman’s Queue Links**
  - Stores the name of the next processor in the queue directly in each processor’s cache
  - Inform next processor asynchronously (via inter-processor interrupt?)
Bus Hardware Solutions

- Use additional bus with write broadcast coherence for TSL (push the new value)
- Invalidate cache copies only when Test-and-Set succeeds
- Read broadcast
  - Whenever some other processor reads a value which I know is invalid, I get a copy of that value too (piggyback)
  - Eliminates the cascade of read-misses
- Special handling of Test-and-Set
  - Cache and bus controllers don’t mess with the bus if the lock is busy
Conclusions

- Spin-locking performance doesn’t scale easily
- A variant of Ethernet back-off has good results when there is little lock contention
- Queuing (parallelizing lock handoff) has good results when there is a lot of contention
- A little supportive hardware goes a long way towards a healthy multiprocessor relationship