Outline

- Last lecture
  - Cluster network interfaces
    - Fast Ethernet, Myrinet, VIA
  - Optimizing message passing programs
    - Portable optimizations
    - Alternative messaging layers
      - Active Messages, Put/Get

- This lecture
  - Optimizing shared memory programs
    - Improved synchronization primitives
  - Software support for shared memory
    - Page based shared memory
      - Ivy, LRC (TreadMarks), HLRC

Performance of Shared Memory Programs

- Division of responsibility between hardware and programmer
  - hardware provides efficient data access and coherence primitives
  - programmer writes program to take advantage of these primitives

- Performance determined by two factors
  - memory access costs
    - does the program utilize the cache structure effectively?
    - temporal locality: small working sets so they fit in the cache
      - blocked algorithms as in the Equation Solver, LU examples
    - spatial locality: reduce fragmentation and false sharing
  - synchronization costs
    - how much time do threads spend waiting
      - for another thread (waiting for an event)
      - for entry into a critical section

Reducing Synchronization Costs

Two degrees of freedom for reducing synchronization costs

- Restructure program to reduce synchronization overheads
  - reduce the size of critical sections
  - partition/fuse critical sections
  - implicit synchronization (data ownership)
  - reader-writer locks

- Improve performance of lock primitives
  - Reduce effects of contention
  - Reduce number of atomic operations
  - Cope with slow/preempted processes
  - Examples:
    - Ticket locks
    - Array locks
    - Wait-free data structures
Better Lock Primitives:

Ticket Lock

- Works like waiting line at deli or bank
  - two counters per lock (next_ticket, now_serving)
  - acquire: fetch&inc next_ticket; wait for now_serving to equal it
    • atomic op upon arrival at lock, not when it’s free (so less contention)
  - release: increment now-serving

Benefits
  - FIFO order
  - low latency for low-contention if fetch&inc cacheable

Costs
  - $O(p)$ read misses at release, since all spin on same variable
  - can be difficult to find a good amount to delay on backoff
    • exponential backoff not a good idea due to FIFO order
    • backoff proportional to (now-serving - next-ticket) may work well

Array-based Queuing Locks

- Waiting processes poll on different locations in an array of size $p$
  - acquire
    • fetch&inc to obtain address on which to spin (next array element)
    • ensure that these addresses are in different cache lines or memories
  - release
    • set next location in array, thus waking up process spinning on it

Benefits
  - $O(1)$ traffic per acquire with coherent caches
  - FIFO ordering, as in ticket lock

Costs
  - $O(p)$ space per lock

Better Lock Primitives:

Non-blocking Locks

- Built on top of non-blocking universal atomic primitives
  - LL/SC: SC succeeds only if no intervening SC
  - CAS: compare-and-swap

Overall strategy
  - if a process falls behind, some other process completes the operation for it
  - provides automatic tolerance to preemption-and scheduling behavior
  - however, primitives are data-structure specific
    • e.g., concurrent queues, heaps, etc.
  - [discuss code in handout]
Software Support for Shared Memory

- **Rationale**
  - Shared memory model offers programmability advantages
  - However, hardware support for shared memory is expensive, not scalable
  - **Solution:** Emulate shared memory in software
  - **Challenge:** Overcoming overheads of communication, coherence

- **Coherence operations in CC-NUMA machines take advantage of hardware support for**
  - **Access control**
    - Detecting that a cache block is not in the desired state
    - Required for initiating appropriate coherence operations
  - **Per-block tags and state (directory)**
  - **Protocol processing**
    - Including interventions in the processor cache
  - **Messages sent between nodes**

---

Page-Based Access Control

- **Problem:** Need an efficient mechanism for detecting when a memory operation (LD/ST) references a block in an incorrect mode
  - Required for initiating coherence actions
  - In the CC-NUMA case, this logic is built into the cache hardware
    - Incurs zero cost

- **Observation:** Modern-day processors/machines provide support for virtual memory, which has very similar semantics
  - Each memory access checks the page protection bits
    - Page can be mapped to be INVALID, READ_ONLY, or READ_WRITE
    - Where is this support?

- **Basic Idea:** Use a page as the granularity of sharing
  - Protection bits serve to identify the state of the block
  - Access violations trigger protection faults (protocol processing)
  - Shared blocks are kept in local memory (still cacheable)

---

Page-based Shared Memory: Ivy

- **Proposed by Li and Hudak (Yale) in 1986**
  - Protocol processing in page-fault handlers
  - Implements a basic invalidation-based protocol
    - Sequential consistency
    - No notion of a "home copy"
    - Pages associated with version numbers to avoid "double fault" overhead
  - Needs additional synchronization operations

---

Software Coherence Processing

- **Invoked upon detecting an access violation**
  - Block tags and directory state maintained in software
  - Explicit issuing/synchronization on network transactions (messages) sent to other nodes
  - Explicit memory management: storage for cached blocks

- **Challenge:** How to overcome higher processing overheads?

- **Advantage:** Flexibility
  - Everything is in software, so can do more complicated things
  - Relaxed memory-consistency models
  - Different protocols for different sharing patterns
    - Read-only, producer-consumer, write-mostly, migratory sharing, etc.
  - A very active area of research
Problems with Basic Page-Based Shared Memory

- Large sharing granularity implies more fragmentation + false sharing
  - ping-pong of pages between nodes
  - expensive because it results in
    - more page-faults: interrupt + multiple kernel-user space crossings
    - substantial communication overhead to maintain SC
- Solution: Use more sophisticated protocols

Using Relaxed Memory Consistency

- Reduce the required amount of communication
  - release consistency (RC) allows invalidations to not be propagated until synchronization points
    - different flavor from use of RC in hardware for reducing write stalls
- however, propagating invalidations on release still results in more communication than is strictly required
  - a process may not do an acquire (so it does not need to see effect of write)
  - may invalidate data earlier than necessary, causing additional false sharing
  - separate messages are required for invalidations and acquires

Relaxed Memory Consistency Models (cont’d)

**Eager release consistency:** Propagate write-notices on release

**Lazy release consistency:** Propagate write-notices on acquire

Lazy Release Consistency: Vector Timestamps

- Concept
  - Establish a system of per-process virtual time
    - Time incremented every time a synchronization event happens
  - Each node keeps track, for each node, the virtual time up to which it has seen write notices from that node
    - Acquirer sends this list to current lock owner
    - Releaser compares with own list and sends write notices that it has seen but the acquirer has not
      - So, in our example, P₂ learns about P₀’s write on z from P₁
  - Why should this work?
Vector Timestamps (cont’d): Implementation

- Execution of a process divided into intervals
  - Defined by synchronization events (acquires and releases)
  - Different processes may be executing different numbered intervals
    - Total order within the same process
    - Partial order defined by synchronization handoff (or a chain of the same)
- Each process maintains a vector time stamp
  - $V_P^i$: Vector time stamp for interval $i$ on process $P$
  - Contains one entry for each process
  - Entry indicates most recent interval from that process that $P$ has received and applied write notices for ($P$’s entry is just $i$)
  - Releaser sends acquirer write notices for unseen intervals (conservative: why?)
    - Acquirer replaces own time stamp with the larger of its own and the releaser’s time stamp

Implications
- A node may get called to send a write notice after an arbitrarily long time
- Storage overheads, garbage collection

Multiple Writer Protocols

- Lazy Release Consistency solves problems of false sharing involving multiple readers and a single writer but invalidations are still required if there are multiple writers
  - Likely because of the page granularity
- Solution
  - Initially write-protect a page
  - On first write, a protection violation occurs
    - System makes copy of the page (called a twin)
    - Unprotect the page to allow further writes
  - Associate a diff (page - twin) with the write-notice

Interaction between Lazy Release Consistency and Multiple-Writer Protocols

- Each write notice is associated with a diff
- A processor can be called upon to supply this diff at any time
  - So, need to store multiple (possibly out-of-date) diffs for the same page
    - Requester needs to retrieve all of the diffs and apply them in turn
      - Contrast with just getting the page
  - Further increases storage requirements

Page-based Shared Memory (2): TreadMarks

- Developed by Pete Keleher, Alan Cox, Willy Zwaenepoel and others at Rice University (1993 – current day)
- Incorporates support for
  - Lazy release consistency
  - Multiple writer protocols
  - Fairly successful in making software shared memory implementations achieve reasonable performance
    - Protocols are significantly more complicated than their hardware counterparts
- Now a commercial product
  - Being incorporated into an OpenMP compiler from Kuck and Associates (now Intel)
  - Available for academic use for a small fee
Alternative Methods for Propagating Writes

- Problem with earlier solution (particularly with LRC):
  - processing overheads: diff creation
    - Direct cost: Run-length encoding
    - Indirect cost: Diff computation destroys cache locality
  - memory overhead: diffs and write-notices have arbitrary lifetime
    - since not possible to predict when a processor would do an acquire
    - need distributed garbage collection

- Tradeoff between computation/communication
  - made sense when communication had to be avoided at all costs (c. 1993)
  - Modern-day cluster interconnects are much better in this regard

- Main reason diffs are required is because there is no home node
  - Vector timestamps obviate need for explicit directory structures
  - Write notices and diffs distributed across the system

Page-based Shared Memory (3): HLRC

- HLRC: Home-based Lazy Release Consistency
  - Example of recent solutions that bring back notion of a home node
    - one node maintains master copy of page
    - diffs are eagerly folded into this copy
    - on an acquire
      - requesting processor gets write-notices from releasing processor
      - As before
      - fetches whole page from home node
        (only one round-trip required to update page, instead of multiple messages)
  - Tradeoff: Cost of propagating diffs to home node vs.
    - maintaining it locally (for an unpredictable time)
    - Forcing the requesters to collect diffs from various nodes
  - diff preparation/propagation costs can be further reduced by relying on additional
    hardware support
    - Princeton SHRIMP: Allows write propagation in the background
    - VIA: allows diff propagation using one network transaction (REMOTE_WRITE)

Page-Based Shared Memory: Implications

- LRC + multiple writer protocols improve performance dramatically
- However, still a wide gap as compared to hardware shared memory for applications with fine-grained sharing
  - false sharing
  - extra communication and processing overhead
  - page faults and fetches are expensive to satisfy
  - synchronization through software messages: dilates critical sections
  - scalability problems because of auxiliary data structures

=> Research challenge
- can one ever match performance of hardware shared memory using software-only approaches?
- Key: software approaches can take advantage of protocol flexibility
  - customize protocols to application behavior
  - more processing but reduced communication (favored by architectural trends)

Next Lecture

- Software support for shared memory (cont’d)
  - Object-based shared memory
  - Application-specific coherence protocols