Announcements

- 2-page project write-ups
  - call to set up an appointment to clarify my comments

Assignments

- HW#2 returned today
  - changed points for extra-credit question from 3 to 2
- HW#3 due today
- HW#4 assigned today, due 11/05
  - place it in my mailbox

Outline

- Last lecture: Scalable Shared Memory Machines
  - scalable support for coherent caches
  - directory-based schemes
    - central issues: what are they?
      - memory versus cache-based schemes
      - invalidation patterns
    - performance concerns: memory space, latency, occupancy
- This lecture:
  - recap: directory-based coherence
  - case study: SGI Cray Origin 2000
  - performance optimizations
    - relaxed memory consistency models
    - programmable protocol processors
      - case study: Stanford FLASH
    - coherent replication in main memory

Recap: Directory-based Cache Coherence

- Snoopy schemes do not scale because they rely on broadcast
  - hierarchical snoopy schemes have root as a bottleneck
- Directory-based schemes allow scaling
  - avoid broadcasts by keeping track of all PEs caching a memory block
  - coherence maintained using point-to-point messages
  - allow flexibility to use any scalable point-to-point network
Recap: Full Bit-Vector Scheme

- Censier and Feautrier, 1978

- Read by PE_i
  - if dirty bit is OFF → \{ read from main memory; turn p[i] ON \}
  - if dirty bit is ON → \{ recall line from dirty PE (cache state to shared); update memory; turn dirty bit OFF; turn p[i] ON; supply recalled data to PE_i \}

- Write by PE_i
  - if dirty bit is OFF → \{ supply data to PE_i; send invalidations to all PEs caching that block; turn dirty bit ON; turn p[i] ON \}
  - if dirty bit is ON → \{ recall line from dirty PE (cache state to invalid); update memory; turn p[i] ON; supply recalled data to PE_i \}

Case Study: SGI Cray Origin 2000

- MIPS R10000 processors
  - 2 per board (for packaging considerations: not a 2-processor SMP)
- Hub chip acts as memory, I/O, and coherence controller
  - L1 block size: 32 B, L2 block size: 128 B

SGI Origin: Cache Coherence Protocol

- States
  - cache: MESI
  - directory: 3 stable (unowned, shared, exclusive), 3 busy, and a poison state
- Protocol actions
  - read request:
    - unowned or shared: set presence bit, and respond with a reply transaction
    - exclusive:
      - set home state to busy-exclusive, modify presence bits to change owner; send intervention request to previous owner, and (speculatively) data to requester; previous owner directly responds to both the requester and the home node
  - write request: can be either RdEx or Upgrade
    - unowned: (if upgrade) NACK, (if RdEx) set presence bit and reply
    - shared:
      - send invalidations to sharers (except requester if upgrade); set new owner; send data (for RdEx) and count to new owner; previous sharers directly acknowledge to requester
    - exclusive: (if upgrade) NACK, (if RdEx) do as in read
SGI Origin: Protocol Actions (contd.)

- Write-back request
  - caused because of cache replacement of a dirty block
    - exclusive: change state to unowned and return an acknowledgement
    - busy:
      - a race condition (intervention and write-back messages cross each other)
      - directory treat write-back as response to intervention; cache ignores the intervention request

- Overall message types
  - 9 requests
  - 6 invalidations and interventions
  - 39 responses
  - extra messages required for correctness
    - write serialization, completion, and atomicity

---

SGI Origin: Directory Structure

- 16- or 64-bit entries
  - 16-bit entry stored in the same DRAM as main memory
  - 64-bit entry stored in an extended directory module (looked up in parallel)

- 3 possible interpretations
  - if the block is in exclusive state
    - pointer contains explicit processor number
      - faster lookup, and resolution between two processors sharing a hub
  - if the block is in shared state
    - directory entry treated as a bit vector
    - presence bit corresponds to a hub (not a processor)
      - broadcast to both processors sharing a hub
    - dynamic choice between bit-per-processor and coarse-vector representation
      - in coarse-vector representation: each bit corresponds to p/64 nodes
      - choice based on if sharers are within same 64-node octant of machine or not

---

SGI Origin: Hub Implementation

---

SGI Origin: Performance Characteristics

- Back-to-back and true unloaded latencies
  - L1 cache: 5.5 ns
  - L2 cache: 56.9 ns
  - local memory: 472 ns (329 ns)
  - remote memory: (4P) 690 ns (564 ns), (16P) 991 ns (862 ns)

- Back-to-back latencies for different initial states of the block
  (latencies for unowned, clean-exclusive, modified states)
  - home=local, owner=local: 472, 707, 1036 ns
  - home=remote, owner=local: 704, 930, 1272 ns
  - home=local, owner=remote: 472, 930, 1159 ns
  - home=remote, owner=remote: 704, 917, 1097 ns
Performance Optimizations

Target two primary causes:
• Coherence protocol implementation
  – excessive synchronization
    • reads need to wait for writes to complete
    • writes need to wait for reads to complete
    ➨ relaxed memory consistency models
  – same protocol used for all data in the system
    • does not take advantage of different sharing behaviors
      – e.g., read-mostly, producer-consumer, etc.
    ➨ programmable protocol processors
• Small caches
  – capacity misses
    • a remote operation is required when working set does not fit in cache
    ➨ cache-only memory architectures (COMA)

Need for Relaxed Consistency Models

• Sequential consistency (SC) requires excessive synchronization
  – program order: completion of previous memory operations
    • requires explicit acknowledgements
  – write atomicity: serialization of writes to the same location
    • requires waiting for all acknowledgements

Consistency Models: Weaker Variations

• Processor consistency (Goodman’89)
  – SC specifies strong ordering of memory accesses in hardware
    • write-to-read ordering implies write-latency cannot be hidden
  – PC allows reads to overtake writes
    • no global ordering of writes
      – violates SC in following example

    P1: A = 1; while ( A==0 ); B = 1; while ( B==0 ); print A;

Consistency Models: Weaker Variants (contd.)

• Weak consistency (Dubois’90)
  – relaxes all program orders
    • read-to-read, read-to-write, write-to-read, and write-to-write
    • models well matched to dynamically scheduled processors
  – insight for WC
    • most programs use synchronization operations to ensure that order of updates is preserved
      – do not assume any order (other than local) for accesses within a critical section
    • so, implementation
      • preserves order only on sync operations
      • other operations are completely unordered
  – burden on programmer to identify sync operations

loads and stores between synchronization operations can be reordered
Consistency Models: Weaker Variants (contd.)

- Release consistency (Gharachorloo’90)
  - WC does not go far enough
  - separation into acquire/release operations permits further optimization
    - LD/ST operations within a block only depend on acquire
      - no way of ensuring this only for protected LD/STs, so all LD/STs and acquires wait for preceding acquire to complete
    - release only needs to wait for LD/ST within the block
      - no way of ensuring this, so waits for all previously issued LD/ST
  - implemented in the Stanford DASH, FLASH, and Origin 2000

Consistency Models: Implementation Freedom

- Sequential consistency
  - reads stall for pending writes
  - writes considered complete on being written to write buffer
- Processor consistency
  - reads can bypass pending writes
- Weak consistency
  - reads can bypass writes, writes can be reordered in the write buffer
  - acquires stall for pending writes and releases
  - releases stall for pending writes
    - first read after release waits for release to perform
- Release consistency
  - reads can bypass writes and pending releases
  - processor issues acquire and stalls for acquire to perform
  - releases stall for pending writes and releases

Consistency Models: Performance Implications

[“Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors”
K. Gharachorloo, A. Gupta, and J. Hennessy, Proceedings of ASPLOS-IV, pp. 245-257.]

weaker consistency models improve performance by overlapping write latency (SC), having read bypass writes (PC), and pipelining writes (WC, RC)

however, taking full advantage of weaker consistency models increases implementation complexity: so the benefits must outweigh costs!

Software Weak-Consistency Models

- Entry consistency (Bershad’93)
  - weaker than release consistency
  - distinguishes among different synchronization variables
    - all LD/ST operations within a synchronization block can be reordered
    - unlike RC, need to wait only for acquire/release operations to the same synchronization variable
  - complicated to implement in hardware
  - naturally supported by concurrent object-oriented programming models
    - method boundary defines acquire/release
    - e.g., Java
Programmer Centric Models

- **Rationale**
  - provide a “safe programming model” with portable assumptions about relationship between source program ordering and allowable interactions
    - avoid concerns about memory reference granularity, what’s a statement, etc.
    - avoid reasoning about the machine operational behavior
  - programmer’s side of the contract
    - mapped to the system-centric consistency model by the vendor’s compiler

- **“Properly Synchronized” programs**
  - all synchronization operations explicitly identified
  - all data accesses ordered through synchronizations
  - several such frameworks
    - Properly-labeled (PL) programs: Gharachorloo’90
    - Data-race-free (DRF) programs: Adve’90
    - Unifying framework (PLpc): Gharachorloo/Adve’92

Need for Programmable Protocol Coprocessors

- **Several sharing patterns in the program**
  - e.g.: read-mostly, producer-consumer, migratory sharing, etc.
  - exploiting specific semantics can reduce required protocol actions
    - e.g., producer-consumer with SC (invalidation-based protocol)
      - 4 messages for each update
        - 1 round-trip for producer to claim ownership
        - 1 round-trip for consumer to read data
      - can reduce to 1 message using an update-based protocol
        - producer sends data to consumer after each update

  ➢ programmable protocol coprocessors
    - allow different ‘objects’ in the system to utilize different coherence protocols
    - primary design issue: degree of customization
      - selection from among a predefined library (e.g., Stanford FLASH)
      - arbitrary user code (e.g., Wisconsin Typhoon)

Stanford FLASH

- **MAGIC chip**
  - integrated memory, network, and cache controller
  - general-purpose programmable engine
    - implements release-consistency using a directory-based protocol
      - directory uses overflow pool (pointer/link store) and is cached in SRAM

Stanford FLASH: MAGIC Microarchitecture

- **Dual-issue, statically scheduled RISC core**
  - no interrupts, exceptions, hardware TLB, memory management
  - subset of DLX instruction set plus
    - bit-field operations (for directory state lookup)
    - branch on bit-set clear
    - interactions with other MAGIC functional units (inbox, outbox, etc.)
Overcoming Capacity Limitations

• Problem
  – in CC-NUMA systems, processor cache replicates data directly on reference
    • no copy in (local) main memory
  – consequence:
    • hardware cache determines capacity of local replication
    • artifactual communication to remote memory on miss

• Solutions
  – tertiary caches
    • a cache for remote accesses
    • e.g., Sequent NUMA-Q, Convex Exemplar
    • already present if nodes of the machine are small-scale SMPs
    • problem:
      – home node stays fixed (even if only one processor accesses block)
      – statically established tertiary cache can be wasteful
  – cache-only memory architectures (COMA)

Cache-Only Memory Architectures (COMA)

• Treat all of main-memory as a hardware-controlled cache
  – e.g, Kendall Square Research (KSR) machines
  – replication capacity is now limited by main-memory size
  – migration is automatic
    • data moves to main memories of nodes which access it
    • attraction memory
  – programmer need not worry about initial data distribution
    • focus on inherent communication and false sharing

• Disadvantage: Hardware complexity
  – storage
    • per-block tags in main memory
    • extra memory overhead needed for replication in the attraction memories
  – complicated coherence protocol
    • no home node: need to find where the data resides
    • how to handle replacement of last copy of a block?

COMA: Performance Trade-Offs

Application characteristics

- Very low miss rate
  - CC-NUMA = COMA

- Mostly coherence misses
  - CC-NUMA > COMA

- Mostly capacity misses and/or poor initial data placement
  - CC-NUMA < COMA

Lecture Summary

• Case study: SGI Cray Origin 2000
• Performance optimizations in scalable shared-memory machines
  – relaxed consistency models
    • PC, WC, RC
  – programmable protocol processors
    • Stanford FLASH
  – cache-only memory architectures
Next Lecture

• Programming for performance
  – common issues: load-balance, synchronization, locality
  – data-parallel programming models
    • data layout
    • compilation strategies