Announcements

- No penalty for late submission of assignments
  - Highly recommend that you try and submit them by (suggested) due date
  - December 12th is a hard deadline for all assignments/project reports
    - Otherwise, I will be forced to give you an Incomplete in the course

- Homework 4 can be downloaded from the web site
  - No VIA programming, only analysis of expected performance benefits
  - Can be done in groups (same as project groups)

- Feedback about project proposals
  - Please follow up with me (e-mail or office hours)

Outline

- Last lecture
  - Architecture of distributed memory multiprocessors (cont’d)
    - Supporting various programming models
    - Hardware design choices
    - Case studies: Cray T3E, IBM SP
  - Large-scale shared memory machines
    - Extending bus-based coherence
    - Directory-based coherence schemes

- This lecture
  - Directory-based coherence schemes (cont’d)
    - Memory organizations
    - Correctness issues
  - Case Study: SGI Origin 2000
  - Relaxed memory consistency models

(Review) Simple Directory-based Coherence

- Full bit-vector (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 (change state to shared); update memory; turn dirty bit OFF; turn p[i] ON; supply recalled data to PE_i }
(Review) Simple Directory-based Scheme

- Full bit-vector (Censier and Feautrier, 1978)

Write by PE_i
- if dirty bit is OFF
  - 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 (change state to invalid);
  - update memory; turn p[i] ON;
  - supply recalled data to PE_i

Directory Organizations

- Memory-based schemes
  - e.g., Stanford DASH, FLASH, MIT Alewife, SGI Origin 2000
  - directory storage proportional to memory blocks
  - full-map vs. partial-map (limited pointers)
    - main issue here is dealing with overflow
    - dense vs. sparse
      - directory is itself a cache

- Cache-based schemes
  - e.g., SCI (Sequent NUMA-Q, Convex Exemplar)
  - use cache blocks to link together sharing chain
    - storage scales with total amount of cache
    - insertion, deletion, communication
      - single vs. double link
    - built out of SRAM (faster)
    - more messages than memory-based schemes

Full bit-vector
- storage
  - one bit of directory memory per main-memory block per PE
  - memory requirements = \( P \cdot \frac{P \cdot M}{B} \)
    - where \( P \) is the number of processors, \( M \) is main memory per PE, and \( B \) is cache-block size
    - overhead not too large for medium-scale MPs
      - e.g., 256 PEs organized as 64 4-PE clusters, 64 byte cache blocks
      - overhead = 64 bits for each 64-byte block (12.5% overhead)
  - invalidation traffic
    - less than limited pointer schemes
  - one way to reduce overhead is to increase \( B \)
    - can result in false sharing and increased coherence traffic
    - ideally, would like different block sizes based on sharing behavior
      - will become possible with reconfigurable architectures

Analysis of Memory-based Schemes

Limited Pointer Schemes

- Rationale: Few sharers most of the time, so keep few pointers
- Distinguished based on overflow scheme
  - DIR-i-B
    - beyond i-pointers, set invalidate-broadcast bit to ON
    - memory requirements = \( i \cdot \log P \cdot \frac{P \cdot M}{B} \)
    - works well if sharing behavior is in one of two extremes: few sharers, or lots of sharers
  - DIR-i-NB
    - when sharers exceed i, invalidate one of the existing sharers
    - significant degradation expected for widely shared, mostly-read data
  - DIR-i-CV-r
    - when sharers exceed i, use bits allocated to i pointers as a coarse-resolution vector (each bit points to multiple PEs)
    - always results in less traffic than DIR-i-B

- Limitless directories (Alewife): Handle overflow using software traps
Rationale: Since total number of cache blocks is much less than total number of memory blocks, most directory entries are idle most of the time
- e.g., 2 MB cache, 2 GB memory per PE  ▶ >98% idle

Sparse directories reduce memory requirements by
- using single directory entry for multiple memory blocks (as in a cache)
  • directory entry can be freed by invalidating cached copies of a block
  • main problem is the potential for excessive directory entry conflicts
  • solution: associative sparse directories (as in a cache)!

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 invalidation 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 treats 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
Scalable Shared Memory Multiprocessors: Performance Optimizations

Target two primary causes:

- **Small caches**
  - capacity misses
    - a remote operation is required when working set does not fit in cache
    - cache-only memory architectures (COMA)

- **Coherence protocol implementation**
  - 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
  - excessive synchronization
    - reads need to wait for writes to complete
    - writes need to wait for reads to complete
    - relaxed memory consistency models

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
    - Instead of cache 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
    - Broadcast on a ring
    - how to handle replacement of last copy of a block?
    - Swap with processor supplying data

COMA: Performance Trade-Offs

Application characteristics

- **Very low miss rate**
  - CC-NUMA = COMA

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

Mostly coherence misses

- CC-NUMA > COMA

Coarse-grained data access

- CC-NUMA = COMA

Fine-grained data access

- CC-NUMA < COMA

Database applications

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 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)
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

P1
\[ A = 1 \]

while \( A = 0 \);
\[
\begin{align*}
P2 & & P3 \\
M & & M \\
& & M \\
A = 1 & & A = 1
\end{align*}
\]

B = 1; while (B==0);
\[ print A; \]

Consistency Models: Weaker Variants

- 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);
\[ B = 1; \]
\[ print A; \]

P2
\[ A = 0 \]

P3
\[ B = 0 \]

A = 1
\[ while (A==0); \]
\[ B = 1; \]
\[ while (B==0); \]
\[ print A; \]

Consistency Models: Weaker Variants (cont’d)

- 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

Consistency Models: Weak Variants (cont’d)

- 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

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

Consistency Models: Performance Implications


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!
Next Four Lectures

- Scalable parallel machines being built from clusters of commodity PCs
  - Lower communication performance
  - Rare to find hardware support for cache coherence
  - Performance optimizations in message passing and shared memory programming models become essential

- Lecture 8: (November 7)
  - Cluster network interfaces
  - Optimizing message passing programs
    - Portable optimizations
    - Alternative messaging layers

- Lectures 9 and 10: (November 14 and 21)
  - Software support for shared memory models
    - Page-based and object-based shared memory

- Lecture 11: (November 28)
  - High-level Parallel/Distributed Programming Models
  - Future directions