Announcements

- Project
  - 2-page proposal due today

- Assignments
  - MPI installation on griffin.cs.nyu.edu
  - any other questions about HW 3?

Outline

- Last lecture: Scalable Distributed Memory Machines
  - introduction
    - need for scalability
    - main design distinction: level of integration
  - hardware choices and support for programming models
  - case study: Cray T3E

- This 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
  - case study: SGI Cray Origin 2000

[Culler/Singh/Gupta, Chapter 8 (Sections 8.1-8.5)]

Caching Remote Data

- Recap: Physical shared address space

- Natural to fetch and cache entire line
  - however, coherence protocols cannot do broadcast and match (snoop)!
Extending Bus-based Coherence Schemes

- **Motivation**
  - leverage SMP building blocks, packaging, optimized technology
  - preserve tight cluster interaction
- **Issues**
  - can these systems be put together (without changing anything inside)
  - what kind of overheads do these systems incur?
- **Several examples**
  - Encore Gigamax, Convex Exemplar, Sequent NUMA-Q

Hierarchy of Busses

- **Snoopy cache protocol**
  - on B1 with L2 acting as memory, on B2 with L2 acting as processor
- **Scaling considerations**
  - number of processors: limited by packaging
  - memory latency and bandwidth: single traversal to root
  - coherency protocol bandwidth: many B1, one B2
  - locality/placement: sharing locality avoids B2 broadcasts

Cluster-based Hierarchies (Encore Gigamax)

- Main memory distributed among clusters
  - L2 cache can be replaced by a tag-only router-coherence switch
- **Scaling considerations**
  - number of processors: limited by packaging
  - memory latency and bandwidth: multiple, fast local access
  - coherency protocol bandwidth: many B1, one B2
  - locality/placement: important

Cache Coherence in Encore Gigamax

- **Operation**
  - write to B1 is passed to B2 if
    - reference to a remote memory word
    - reference to a local memory word, but present in some remote cache
  - read to B1 is passed to B2 if
    - reference to a remote memory word (and not in cluster cache)
    - reference to a local memory word, but dirty in some remote cache
  - write to B2 is passed to B1 if
    - reference to a local memory word
    - data belongs to remote memory, but the block is dirty in a local cache
  - many race conditions possible: write-back going out as request coming in
Ring-based Cache Coherence (KSR)

• Buses can be replaced by rings
  – any media capable of a broadcast
• Scaling considerations
  – number of processors: many
  – memory latency and bandwidth: linear in \( P \)
  – coherency protocol bandwidth:
    – broadcast medium, but no global serialization
    – many concurrent transactions, out of phase (many, many states)
  – locality/placement: within B1, all same on R2

Lessons from Hierarchical Coherence Schemes

• Why does bus-based coherence work?
  – FSM sequences with effectively atomic transitions ensure consensus on status of memory block and therefore coherence
• Why do extensions to bus-based schemes work?
  – layers extend these FSM transitions, delaying additional accesses until a global decision can be enforced
  – scaling limitations, but the overall scheme works because of global agreement
• General formulation: “directory-based” structure
  – associate an explicit state with each memory block
  – query and update this state using atomic transitions
    – memory consistency is ensured by restricting what this state can be

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

A Simple Directory-based Coherence Scheme

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

  - if dirty bit is OFF \( \rightarrow \{ \text{read from main memory; turn } p[i] \text{ ON} \} \)
  - if dirty bit is ON \( \rightarrow \{ \text{recall line from dirty PE (cache state to shared); update memory; turn dirty bit OFF; turn } p[i] \text{ ON; supply recalled data to } PE_i \} \)
A Simple Directory-based Scheme (contd.)

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

  - 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\(_i\) }

Directory-based Coherence: Key Issues

- What information is needed to achieve coherence?
  - a means to trap accesses from processors, and force these global memory system operations to happen correctly
  - a means to purge/revoke other processors local copies
    - keep track of where the copies are
    - multicast invalidation requests
  - global state transitions for the block, similar consistency as with the bus-based approach

- Scaling considerations
  - number of processors: many
  - memory latency and bandwidth: scalable network
  - coherence protocol bandwidth: ?
    - function of how many concurrent invalidates (updates), number of copies
  - locality/placement: moderately important
    - local network transactions

Cache Invalidation Patterns

- Hypothesis
  - on a write to a shared location, with high probability, only a small number of caches need to be invalidated

- If the above were not true, directory schemes would offer little advantage over snooping schemes!

- Empirical study (see Culler/Singh/Gupta: Section 8.3.1)
  - SPLASH-2 benchmarks running on 64 processors
  - infinite capacity, fully associative caches
    - cache replacement might reduce number of sharers

Invalidation Patterns
Invalidate Pattern Summary

- Different categories of sharing (Gupta and Weber, IEEE TOC, July 1992)
  - code and read-only objects (e.g., A and B matrices in matrix multiply)
  - no problem since never written
  - migratory objects (e.g., global sum onto which PEs add their partial sums)
  - only a single invalidation generated per write, independent of P
  - mostly-read objects (e.g., a bound variable in branch-and-bound TSP)
  - invalidations are large but infrequent: so little impact on performance
  - frequently read/written objects (e.g., task queues)
  - invalidations usually remain small, though frequent
  - producer-consumer sharing (e.g., near-neighbor interactions in eqn. solver)
    - typically, few invalidations
  - synchronization objects
    - low-contention locks result in few invalidations
    - high-contention locks need special support
      - hardware combining (NYU Ultracomputer)
      - software trees, queuing locks

Performance Issues

- How long do each of the protocol operations take?
  - read a global directory, locate and purge copies, change state, move copies of data
  - typical solutions:
    - exploit concurrency across operations (multiple invalidations)
    - relaxed consistency models
      - reduce need for protocol operations
      - pipeline requests and acknowledgements
    - more details in Lecture 7
- How much memory is needed for all these states/protocols?
  - store block’s consistency state, locate copies, state for ongoing transactions
  - is memory proportional to physical memory, cache sizes, or something else?

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

Analysis of Memory-based Schemes

- Full-bit vector
  - storage
    - one bit of directory memory per main-memory block per PE
    - memory requirements = P * \( \frac{P * M}{B} \)
    - 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
**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
    - 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**

**Sparse Directories**

- **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., 256 KB cache, 16 MB 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)

**Implementing Cache-Coherent Shared Memory**

- Remote pseudo-processor performs directory update and memory access
  - write: [ pseudo-memory sends RdEx to remote home; pseudo-processor sends invalidates to sharers and replies with block; invalidations acks returned to home node or requester ]

**Correctness: Write Serialization for Coherence**

- **Recap:**
  - all processors must see writes to a given location as having happened in the same order
  - in a bus-based system: serialization enforced by the global bus

- **In directory-based schemes:** serialize all requests at the home node
  - typical approach for serialization at a location: **busy** (or **pending**) states
    - buffer at the home node (e.g., MIT Alewife)
    - buffer at the requestors (e.g., SCI protocol)
    - NACK and retry (e.g., Origin 2000)
    - forward to the dirty node (e.g., Stanford DASH)

  - however, just serializing at a location is not sufficient
    - also need to be careful to ensure serialization among ongoing transactions
      - e.g., update-based protocols in a network that does not preserve delivery order (two processors may see updates in a different order than at the home node)
      - and how incoming transactions are applied to a pending block
Correctness: Write Completion and Write Atomicity

- Recap
  - sequential consistency (SC) requires write completion and write atomicity

- In directory-based systems:
  - write completion requires explicit acknowledgements
  - write atomicity requires waiting for all acknowledgements
    - relaxed consistency models (Lecture 7) address this problem

\[
\begin{align*}
P1 & \quad\quad A = 1; \quad\quad \text{while (A==0);} \\
P2 & \quad\quad B = 1; \quad\quad \text{while (B==0);} \\
P3 & \quad\quad \text{print A;}
\end{align*}
\]

Correctness: Deadlock, Livelock, and Starvation

- Deadlock
  - primary cause: filling up of a finite input buffer
  - solutions
    - provide enough buffer space (e.g., MIT Alewife)
    - use NACKs
    - provide separate request/response networks
    - hybrid of two or more of the above
      - start with a request/reply protocol, evolve into NACK-based approach

- Livelock
  - primary cause: NACK based approach
  - solution: associate priorities with retried requests (based upon no. of NACKs)

- Starvation
  - primary cause: unfair buffering at serialization point
  - solutions:
    - hope that this does not happen (due to randomness in machine delays)
    - NACKs + random delay before retry

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
Lecture Summary

- Scalable shared-memory parallel machines
  - hierarchical coherence protocols
  - directory-based coherence schemes
    - key idea
      - associate state (directory) with each memory block
      - update state using atomic transactions to ensure correct memory consistency
      - benefit: no broadcasts, arbitrary network topology, tolerable hardware overheads
    - design choices
      - memory versus cache-based schemes
      - directory organization
    - performance concerns: memory space, latency, occupancy
  - case study: SGI Cray Origin 2000

Next Lecture

- Scalable Shared Memory Machines (continued)
  - relaxed memory consistency models
    - weak ordering
    - release consistency
    - entry consistency
  - coherent replication in main memory
  - programmable protocol processors
  - case study: Stanford FLASH