(Review)
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
  - more about this later in the lecture …

Generic Distributed-Memory Multiprocessor

2 levels of switches
- intranode switch is typically a bus
- internode switches provide independent communication paths between nodes so that bandwidth increases as nodes are added
Key Distinction: Level of Integration

- **chip-level (processor bus)**
  - J-machine, Alewife, nCUBE/2
  - 10 GB/s, 1-5 cycles

- **board-level (memory bus)**
  - CM-5, Paragon, T3E, Origin, SX1
  - 1 GB/s, 20-50 cycles

- **system-level (I/O bus)**
  - SP, PCs/Myrinet, PCs/Fast Ethernet
  - 1-5 cycles, ~100 MB/s, 500-1000 cycles

---

Distributed Memory Multiprocessors

- **Key primitive:** A network transaction
  - one way transfer + some action at the destination

- **Differences from a bus transaction**
  - source and destination of a transaction are uncoupled
  - no direct wires, no global arbitration of resources
  - no global information available to all modules at the same instant
  - huge number of concurrent transactions

---

HW Design Choices (1): Physical DMA

- **HW does no interpretation on the information within a transaction**
  - representative of early message-passing machines
    - nCUBE10, nCUBE/2, Intel iPSC, iPSC/2, iPSC860, Delta, Ametek, SP-1
  - physical DMA to/from network
    - use of physical addresses
    - sending/receiving requires OS intervention

---

HW Design Choices (2): User-level Access

- **HW distinguishes between system and user messages**
  - user messages both injected and received without OS intervention
  - more recent machines: CM-5, Cray T3D/T3E, VIA

---
HW Design Choices (3): Dedicated Message Processing

- HW dedicates resources to interpret information in transactions
  - protocol processing can be off-loaded to communication processor
  - example machines: Meiko CS-2, Intel Paragon
  - differentiated by whether CP is a symmetric processor, or an embedded processor (with its own path to the network interface)

HW Design Choices (4): Shared Physical Address Space

- HW provides support for loads, stores, atomic operations
  - dancehall: NYU Ultracomputer, BBN Butterfly, IBM RP-3
  - distributed memory: Cray T3D, Cray T3E, Origin

Clusters and Networks of Workstations

- Building scalable machines using commodity components
  - Systems: processors, memory, disk (e.g., a PC)
  - High-performance networks (e.g., Myrinet, VIA)
    - connects to the system on the peripheral bus
  - Cost-effective, but needs to overcome several disadvantages
    - Loose coupling of network to node
    - Independent task scheduling
    - ...

Case Study (1): Cray T3E (c. 1998)

- 2048 processors, 3D Torus
  - DEC Alpha 21164
  - 300-600 MHz
  - 8KB L1 instr and data
  - 96 KB L2 unified
  - 2 outstanding memory refs
  - 128 bits wide
  - 1.2 GB/s
  - 75 MHz
  - Router, 3D interconnect
  - 600 MB/s per link in each direction
  - 4 x 4B x 75MHz = 1.2 GB/s

User-level latency:
- ~10 µs
- Bandwidth:
- ~2 GBps

User/system
Cray T3E: The “Glue” Logic

- CPU
  - Stream Buffers
  - Address Translation
  - E-registers
  - Messaging
  - Atomic Ops
  - Network

- Low-overhead synchronization
- Enhances local memory bandwidth
- Pipelining of remote memory requests (split-phase)
- Address translation
- E-registers
- Atomic ops

Cray T3E: E-Registers

- Central mechanism of the “glue” logic
  - 512 user + 128 system registers (64-bit)
  - Memory-mapped into non-cacheable I/O space
    - Bit 39 of physical address distinguishes cacheable/noncacheable space
  - 2 types of operations
    - Direct loads and stores
      - Implicitly synchronized using full/empty flags
    - Global E-register operations (gets and puts) use I/O space stores
      - Address encodes command (put/get), and src/dest E-register
      - Data encodes pointer to block of E-registers and address index
      - Flexible data distribution
      - Split-phase operations: 4 put/get operations every 2 (75 MHz) cycles
      - Strided puts/gets

- Benefits
  - Pipelined remote memory operations
  - High memory-to-memory bandwidth
    - 60 MB/s versus 340 MB/s for random gather

Cray T3E: Synchronization Operations

- Atomic memory operations (AMO) against arbitrary locations
  - Build upon E-register support
    - Specify AMO command on address bus
    - Operands taken from aligned block of base/index E-registers (data bus)
    - Result returned to E-register specified on address bus
  - Operations
    - Fetch&Inc (hardware merge support)
    - Fetch&Add
    - Compare&Swap
    - Masked Swap
  - Performance
    - Sustained rate: 40ns per F&I, 222ns per op for the others
    - Latency: ~1.5µs
  - Hardware barrier support
    - Single cycle check

Cray T3E: Case Study (2): IBM SP

- Relatively loosely coupled
  - Five types of nodes
    - “Thin” and “Wide” 332 MHz PowerPC 604e nodes
      - 2-4 CPUs per board
    - “Thin”, “Wide”, and “High” 375 MHz Power 3 nodes
      - 2-4 CPUs per board in the first two (difference in no. of PCI slots)
      - 16 per board in the last one
  - Different network options
    - Ethernet, Token Ring, FDDI, high-performance SP-2 interconnect
    - SP-2 switch performance
      - Latency: 20 µs
      - Bandwidth: 140 – 350 MB/s
Supporting a Physical Shared Address Space on Distributed Memory Multiprocessors

- Natural to fetch and *cache* entire line
- However, need to keep cached data *coherent*
  - protocols cannot do broadcast and match (snoop!)

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)
  
  **Read by PE_\text{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_\text{i}

A Simple Directory-based Scheme (cont’d)

- Full bit-vector (Censier and Feautrier, 1978)
  
  **Write by PE_\text{i}:**
  - if dirty bit is OFF  
    - supply data to PE_\text{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 (change state to invalid); update memory; turn p[i] ON; supply recalled data to PE_\text{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!
  - Might as well serialize every transaction

- Empirical study
  - SPLASH-2 benchmarks running on 64 processors
  - infinite capacity, fully associative caches
    - cache replacement might reduce number of sharers

Invalidation Patterns

  - 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
      - different memory blocks, multiple invalidations
    - relaxed consistency models (later in the lecture)
      - reduce need for protocol operations
      - pipeline requests and acknowledgments
- 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
  - 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 = $P \times \frac{P \times M}{B}$
    - 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
      - memory requirements = $i \times \log P \times \frac{P \times M}{B}$
    - 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-NB
- Limitless directories (Alewife): Handle overflow using software traps
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: 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

Improving Coherence Performance:
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

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); print A;
      P2: B = 1; while (B==0);
      P3: A = 1

Load
Load
Store
Store
Load
Store

loads and stores between synchronization operations can be reordered

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

- Very similar to total order store ordering (TSO) used in SPARC v9 processors
  - can enforce SC using fence or memory barrier instructions

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: Weaker 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 acquire/relases 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 2000/3000

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

End of Lecture Material

Course Summary

- Lecture 1: Fundamentals
  - Technology trends
  - CPU performance equation
- Lecture 2: Instruction set architectures
  - SIMD extensions
- Lectures 3-8.5: Processor Core
  - 5-stage RISC pipeline
  - Hazards: Structural, Data, Control
  - Branch prediction
  - Dynamic scheduling
    - Scoreboarding, Tomasulo’s
  - Speculative execution
  - Multiple instruction issue
  - VLIW/EPIC processors
  - Limits of ILP
- Lecture 8.5-11: Memory Hierarchy
  - Caches
    - Structure
    - Performance
  - Memory
    - Organization-level performance
    - Chip-level performance
- Lecture 12-14: Multiprocessors
  - Road to convergence
    - Data parallel, systolic, data flow
  - Snoopy cache coherence protocols
  - Distributed memory computers
  - Directory-based cache coherence
  - Memory consistency models
- Plus, the five assignments …