Shared Memory Multiprocessors

Meta-message today

- powerful high-level abstraction boils down to specific, simple low-level mechanisms
  - each detail has significant implications

- Topic: THE MEMORY ABSTRACTION
  - sequence of reads and writes
    - each read returns the “last value written” to the address

- ILP -> TLP

A take on Moore’s Law

Uniprocessor View

- Performance depends heavily on memory hierarchy
- Managed by hardware
- Time spent by a program
  - $\text{Time}_{\text{prog}}(1) = \text{Busy}(1) + \text{Data Access}(1)$
  - Divide by cycles to get CPI equation
- Data access time can be reduced by:
  - Optimizing machine
    - bigger caches, lower latency...
  - Optimizing program
    - temporal and spatial locality

What is a Multiprocessor?

- A collection of communicating processors
  - Goals: balance load, reduce inherent communication and extra work

- A multi-cache, multi-memory system
  - Role of these components essential regardless of programming model
    - Program model and comm. abstr. affect specific performance tradeoffs
Relationship between Perspectives

- Parallelization step(s) → Performance Issue → Processor time component
- Decomposition/assignment → Load imbalance and communication
- Data-remote → Busy-overhead
- Data-local → Data remote
- Orchestration/communication → Artificial communication and data locality
- Orchestration/mapping → Communication structure

Busy(p) + Data(p) + Synch(p) + Data_remote(p) + Busy_overhead(p)

Back to Basics

- Parallel Architecture = Computer Architecture + Communication Architecture
- Small-scale shared memory
  - extend the memory system to support multiple processors
  - good for multiprogramming throughput and parallel computing
  - allows fine-grain sharing of resources
- Naming & synchronization
  - communication is implicit in store/load of shared physical address
  - synchronization is performed by operations on shared addresses
- Latency & Bandwidth
  - utilize the normal migration within the storage to avoid long latency
  - operations and to reduce bandwidth
- Economical medium with fundamental BW limit
- Focus on eliminating unnecessary traffic

Natural Extensions of Memory System

- Bus-Based Symmetric Shared Memory
  - Dominate the server market
  - Building blocks for larger systems: arriving to desktop
  - Attractive as throughput servers and for parallel programs
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension
- Normal uniprocessor mechanisms to access data
  - Key is extension of memory hierarchy to support multiple processors
- Now Chip Multiprocessors

Caches are Critical for Performance

- Reduce average latency
  - automatic replication closer to processor
- Reduce average bandwidth
- Data is logically transferred from producer to consumer to memory
  - store reg ↔ mem
  - load reg ↔ mem
- Many processors can share data efficiently
- What happens when store & load are executed on different processors?

Example Cache Coherence Problem

- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
- Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!
Caches and Cache Coherence

- Caches play key role in all cases
  - Reduce average data access time
  - Reduce bandwidth demands placed on shared interconnect
- Private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
  => Cache coherence problem
- What do we do about it?
  - Organize the mem hierarchy to make it go away
  - Detect and take actions to eliminate the problem

Advantages

- Cache placement identical to single cache
  - only one copy of any cached block
- Fine-grain sharing
  - Communication latency determined level in the storage hierarchy where the access paths meet
  - 2-10 cycles
  - Cray Xmp has shared registers!
- Potential for positive interference
  - one proc prefetches data for another
- Smaller total storage
  - only one copy of code/data used by both proc.
- Can share data within a line without “ping-pong”
  - long lines without false sharing

Disadvantages

- Fundamental BW limitation
- Increases latency of all accesses
  - X-bar
  - Larger cache
  - L1 hit time determines proc. cycle time !!!
- Potential for negative interference
  - one proc flushes data needed by another
- Many L2 caches are shared today
- CMP makes cache sharing attractive

Intuitive Memory Model

- Reading an address should return the last value written to that address
- Easy in uniprocessors
  - except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical

Snoopy Cache-Coherence Protocols

- Bus is a broadcast medium & Caches know what they have
- Cache Controller “snoops” all transactions on the shared bus
  - relevant transaction if for a block it contains
    - take action to ensure coherence
      - invalidate, update, or supply value
    - depends on state of the block and the protocol

Shared Cache: Examples

- Alliant FX-8
  - early 80’s
  - eight 68020s with x-bar to 512 KB interleaved cache
- Encore & Sequent
  - first 32-bit micros (N32032)
  - two to a board with a shared cache
  - coming soon to microprocessors near you...

Snoopy Cache-Coherence Protocols

- Bus is a broadcast medium & Caches know what they have
- Cache Controller “snoops” all transactions on the shared bus
  - relevant transaction if for a block it contains
    - take action to ensure coherence
      - invalidate, update, or supply value
    - depends on state of the block and the protocol

Intuitive Memory Model

- Reading an address should return the last value written to that address
- Easy in uniprocessors
  - except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical

Advantages

- Cache placement identical to single cache
  - only one copy of any cached block
- Fine-grain sharing
  - Communication latency determined level in the storage hierarchy where the access paths meet
  - 2-10 cycles
  - Cray Xmp has shared registers!
- Potential for positive interference
  - one proc prefetches data for another
- Smaller total storage
  - only one copy of code/data used by both proc.
- Can share data within a line without “ping-pong”
  - long lines without false sharing

Disadvantages

- Fundamental BW limitation
- Increases latency of all accesses
  - X-bar
  - Larger cache
  - L1 hit time determines proc. cycle time !!!
- Potential for negative interference
  - one proc flushes data needed by another
- Many L2 caches are shared today
- CMP makes cache sharing attractive

Intuitive Memory Model

- Reading an address should return the last value written to that address
- Easy in uniprocessors
  - except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical

Snoopy Cache-Coherence Protocols

- Bus is a broadcast medium & Caches know what they have
- Cache Controller “snoops” all transactions on the shared bus
  - relevant transaction if for a block it contains
    - take action to ensure coherence
      - invalidate, update, or supply value
    - depends on state of the block and the protocol
Example: Write-thru Invalidate

Architectural Building Blocks

- Bus Transactions
  - fundamental system design abstraction
  - single set of wires connect several devices
  - bus protocol: arbitration, command/addr, data
  - Every device observes every transaction

- Cache block state transition diagram
  - FSM specifying how disposition of block changes
    - invalid, valid, dirty

Design Choices

- Controller updates state of blocks in response to processor and snoop events and generates bus transactions
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - Write-through vs Write-back
  - Invalidate vs. Update

Architectural Building Blocks

Write-through Invalidate Protocol

- Two states per block in each cache
  - as in uniprocessor
  - state of a block is a p-vector of states
  - Hardware state bits associated with blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Writes invalidate all other caches
  - can have multiple simultaneous readers of block, but write invalidates them

Write-through vs. Write-back

- Write-through protocol is simple
  - every write is observable
- Every write goes on the bus
  - Only one write can take place at a time in any processor
- Uses a lot of bandwidth!

Examples:
- 200 MHz dual issue, CPI = 1, 15% stores of 8 bytes
- 30 M stores per second per processor
- 240 MB/s per processor

1 GB/s bus can support only about 4 processors without saturating

Invalidate vs. Update

- Basic question of program behavior:
  - Is a block written by one processor later read by others before it is overwritten?
- Invalidate.
  - yes: readers will take a miss
  - no: multiple writes without addition traffic
  - also clears out copies that will never be used again
- Update.
  - yes: avoids misses on later references
  - no: multiple useless updates
- even to pack rats

Need to look at program reference patterns and hardware complexity

but first - correctness
Intuitive Memory Model???

- Reading an address should return the last value written to that address
- What does that mean in a multiprocessor?

Coherence?

- Caches are supposed to be transparent
- What would happen if there were no caches
- Every memory operation would go "to the memory location"
  - may have multiple memory banks
  - all operations on a particular location would be serialized
  - all would see THE order
- Interleaving among accesses from different processors
  - within individual processor => program order
  - across processors => only constrained by explicit synchronization
- Processor only observes state of memory system by issuing memory operations!

Definitions

- Memory operation
  - load, store, read-modify-write
- Issues
  - leaves processor’s internal environment and is presented to the memory subsystem (caches, buffers, buses, dram, etc)
- Performed with respect to a processor
  - write: subsequent reads return the value
  - read: subsequent writes cannot affect the value
- Coherent Memory System
  - there exists a serial order of mem operations on each location s.t.
    - operations issued by a process appear in order issued
    - value returned by each read is that written by previous write in the serial order
    => write propagation + write serialization

Is 2-state Protocol Coherent?

- Assume bus transactions and memory operations are atomic, one-level cache
  - all phases of one bus transaction complete before next one starts
  - processor waits for memory operation to complete before issuing next
  - with one-level cache, assume invalidations applied during bus transaction
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (bus order)
  => invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order

Ordering

- Writes establish a partial order
- Doesn’t constrain ordering of reads, though bus will order misses too
  - any order among reads between writes is fine, as long as in program order

Write-Through vs Write-Back

- Write-thru requires high bandwidth
- Write-back caches absorb most writes as cache hits
  => Write hits don’t go on bus
  - But now how do we ensure write propagation and serialization?
  - Need more sophisticated protocols: large design space
- But first, let’s understand other ordering issues
Setup for Mem. Consistency

• Coherence => Writes to a location become visible to all in the same order
• But when does a write become visible?
  - How do we establish orders between a write and a read by different procs?
    - use event synchronization
    - typically use more than one location!

Example

```
P1  P2
/* Assume initial value of A and flag is 0 */
A = 1; while (flag == 0); /* spin idly*/
flag = 1; print A;
```

Memory Consistency Model

• Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it
• Without it, can’t tell much about an SAS program’s execution
• Implications for both programmer and system designer
  - Programmer uses to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
• Contract between programmer and system

Sequential Consistency

• Total order achieved by interleaving accesses from different processes
  - Maintains program order, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - as if there were no caches, and a single memory
• “A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” [Lamport, 1979]

Example WB Snoopy Protocol

• Invalidation protocol, write-back cache
• Each block of memory is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
• Each cache block is in one state (track these):
  - Shared: block can be read
  - OR Exclusive: cache has only copy, its writeable, and dirty
  - OR Invalid: block contains no data
• Read misses: cause all caches to snoop bus
• Writes to clean line are treated as misses

Write-Back State Machine - CPU

• State machine for CPU requests for each cache block
• Non-resident blocks invalid

Conceptual Picture

Write-back cache block
Place write miss on bus

CPU Read hit
Shared read only

CPU Write
Place Write Miss on bus

Cache Block State

Exclusive (read/write)

CPU Write Miss (?)
Write-back cache block
Place write miss on bus
Write-Back State Machine - Bus req

- State machine for bus requests for each cache block

- Invalid state
- Write miss for this block
- Read miss for this block
- Write-back (Block, abort memory access)
- Exclusive (read/write)

Block-replacement

- State machine for CPU requests for each cache block

- Invalid state
- CPU Read hit
- Shared (read-only)
- CPU Write
- Place Write Miss on bus
- Cache Block State
- Exclusive (read/write)
- CPU Write Miss
- Place Write Miss on bus

Write-back State Machine - III

- State machine for CPU requests for each cache block and for bus requests for each cache block

- Invalid state
- Write miss for this block
- Read miss for this block
- Write-back Block (block, abort memory access)
- Exclusive (read/write)

Example

<table>
<thead>
<tr>
<th>Proc.</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>P2</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>

Assumes A1 and A2 map to same cache block, initial cache state is invalid.
**Example**

<table>
<thead>
<tr>
<th>Step</th>
<th>State</th>
<th>Addr</th>
<th>Value</th>
<th>State</th>
<th>Addr</th>
<th>Value</th>
<th>Action</th>
<th>Proc. Addr</th>
<th>Value</th>
<th>Addr Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: Read A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>RdMs</td>
<td>P2 A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shar. A1</td>
<td>10</td>
<td>RdDa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Inv.</td>
<td>A1</td>
<td>20</td>
<td>WrMs</td>
<td>P2 A1</td>
<td>A1 10</td>
<td>20</td>
<td>P2: Write 40 to A2</td>
<td>WrMs</td>
<td>P2 A2</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>P2: Read A1</td>
<td>P1 Write 10 to A1</td>
<td>P2: Write 20 to A1</td>
<td>P2: Write 40 to A2</td>
<td>Assumess A1 and A2 map to same cache block, but A1 != A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Interplay of Protocols and Usage**

- What happens if ... ?
- Collection of sequential programs running on a multiprocessor
- OS starts process up on a different processor?
- Variable written by one and widely shared?
- Variable is flag ping-ponged between threads?
- Two variables touched by different threads fall into the same block?

**MESI State Transition Diagram**

- Detect when read miss is unshared
- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache xfers
- Only one cache flushes data
- MOESI protocol: Owned state: exclusive but memory not valid

**Hardware Support for MESI**

- All cache controllers snoop on BusRd
- Assert 'shared' if present (S? E? M?)
- Issuer chooses between S and E
- How does it know when all have voted?
Dragon Write-back Update Protocol

- 4 states
  - Exclusive-clean or exclusive (E): I and memory have it
  - Shared clean (Sc): I, others, and maybe memory, but I’m not owner
  - Shared modified (Sm): I and others but not memory, and I’m the owner
    - Sm and Sc can coexist in different caches, with only one Sm
  - Modified or dirty (D): I and, none else

- No invalid state
  - If in cache, cannot be invalid
  - If not present in cache, view as being in not-present or invalid state

- New processor events: PrRdMiss, PrWrMiss
  - Introduced to specify actions when block not present in cache

- New bus transaction: BusUpd
  - Broadcasts single word written on bus; updates other relevant caches

Dragon State Transition Diagram

Snooping Cache Variations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Exclusive</td>
<td>Owner Exclusive</td>
<td>Private Dirty</td>
<td>Modified (private,!=Memory)</td>
</tr>
<tr>
<td>Owned</td>
<td>Shared</td>
<td>Shared Clean</td>
<td>Exclusive (private,!=Memory)</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Shared</td>
<td>Shared</td>
<td>Shared (shared,!=Memory)</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

- Owner can update via bus invalidate operation
- Owner must write back when replaced in cache
  - If read sourced from memory, then Private Clean
  - If read sourced from other cache, then Shared
  - Can write in cache if held private clean or dirty

Implementation Complications

- Write Races:
  - Cannot update cache until bus is obtained
  - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - Split transaction bus:
    - Bus transaction is not atomic:
      - can have multiple outstanding transactions for a block
    - Multiple misses can interleave,
      allowing two caches to grab block in the Exclusive state
    - Must track and prevent multiple misses for one block
  - Must support interventions and invalidations

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU just to check:
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
  - block size, associativity of L2 affects L1

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and its dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)
Artifactual Communication

- Accesses not satisfied in local portion of memory hierarchy cause “communication”
  - Inherent communication, implicit or explicit, causes transfers
  - determined by program
  - Artifactual communication
  - determined by program implementation and arch. interactions
  - poor allocation of data across distributed memories
  - unnecessary data in a transfer
  - unnecessary transfers due to system granularities
  - redundant communication of data
- Inherent communication is what occurs with unlimited capacity, small transfers, and perfect knowledge of what is needed.

Fundamental Issues

- 3 issues to characterize parallel machines
  1) Naming
  2) Synchronization
  3) Performance: Latency and Bandwidth

Fundamental Issue #1: Naming

- Naming:
  - what data is shared
  - how it is addressed
  - how processes refer to each other
  - Choice of naming affects code produced by a compiler: via load where just remember address or keep track of processor number and local virtual address for msg. passing
  - Choice of naming affects replication of data: via load in cache memory hierarchy or via SW replication and consistency

Summary: Parallel Framework

- Layers:
  - Programming Model:
    - Multiprogramming: lots of jobs, no communication
    - Message passing: send and receive messages
  - Communication Abstraction:
    - Shared address space: e.g., load, store, atomic swap
    - Message passing: e.g., send, receive
    - Debate over this topic (ease of programming, scaling)
      => many hardware designs 1:1 programming model