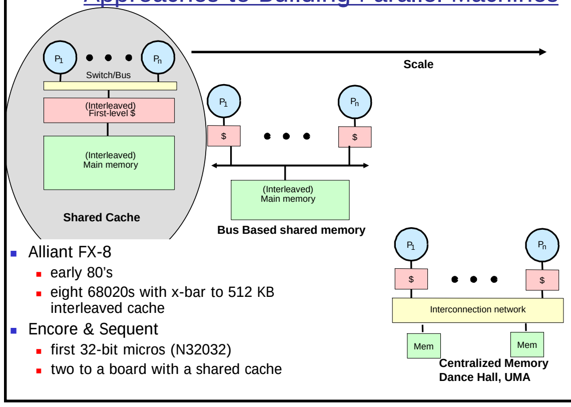
Shared Memory Architectures

Arvind Krishnamurthy
Fall 2004

Approaches to Building Parallel Machines

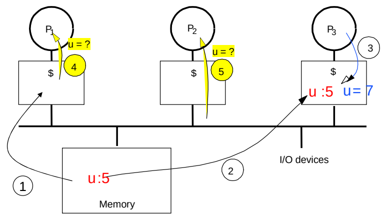


- Alliant FX-8
  - early 80's
  - eight 68020s with x-bar to 512 KB interleaved cache
- Encore & Sequent
  - first 32-bit micros (N32032)
  - two to a board with a shared cache

Shared Cache Architectures

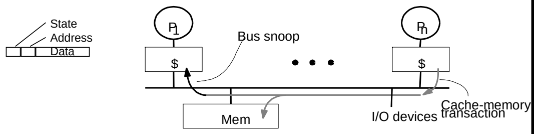
- What are the advantages and disadvantages?

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!

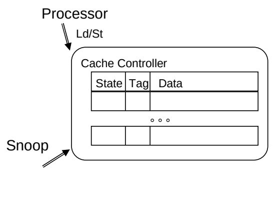
Snoopy Cache-Coherence Protocols



- Bus is a broadcast medium & caches know what they have
- Cache Controller "snoops" all transactions on the shared bus
  - A transaction is a relevant transaction if it involves a cache block currently contained in this cache
  - take action to ensure coherence
    - invalidate, update, or supply value
  - depends on state of the block and the protocol

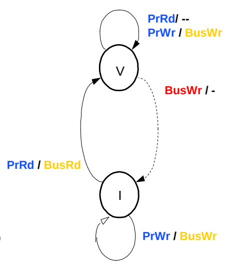
Design Choices

- Update state of blocks in response to processor and snoop events
  - Valid/invalid
  - Dirty (inconsistent with memory)
  - Shared (in another caches)
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - Write-through vs Write-back
  - Invalidate vs. Update



Write-through Invalidate Protocol

- Two states per block in each cache
  - Invalid, Valid
  - as in uniprocessor
- Cache check:
  - Compute position(s) to check based on address
  - Check whether valid
  - If valid, check whether tag matches required address
- If present:
  - If read ➔ just use the value
  - If write ➔ update value, send update to bus
- 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!

Example: 3GHz processor, CPI = 1, 10% stores of 8 bytes

=> 300 M stores per second per processor

=> 2400 MB/s per processor

4 GB/s bus can support only about 1-2 processors without saturating

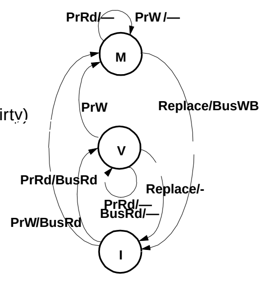
Invalidate vs. Update

- When does one prefer:
  - Invalidation based scheme?
  - Update based scheme?

=> Need to look at program reference patterns and hardware complexity, but first: correctness

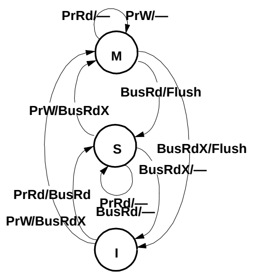
Write-back Caches (Uniprocessor)

- 3 Processor operations:
  - read
  - write
  - replace
- 3 states:
  - Invalid, valid(clean), modified(dirty)
- 2 bus transactions:
  - Read, write-back



Write-back MSI (multi-processors)

- Treat valid as “shared”
- Treat modified as “exclusive”
- Introduce new bus operation
  - Read-exclusive: read for later modifications (read to own)
  - BusRdx causes others to invalidate
  - BusRdx even if write-hit in S
  - Read obtains block in “shared”



Lower Level Protocol Choices

- How does memory know whether or not to supply data on BusRd?
- BusRd observed in M state: transition to make?
  - M → I
  - M → S
  - Depends on expectation of access patterns
- BusRdX could be replaced by BusUpgr without data transfer
- Read-Write is 2 bus transactions, even if no sharing
  - BusRd (I→S) followed by BusRdX or BusUpgr
  - What happens on sequential programs? Performance degrades

Update Protocols

- If data is to be communicated between processors, invalidate protocols seem inefficient
- Consider shared flag:
  - P0 waits for it to be zero, then does work and sets it one
  - P1 waits for it to be one, then does work and sets it zero
- How many transactions?
  - P0: Read shared
  - P1: Read Exclusive
  - P1: Write 0
  - P0: Read
  - P1: Read shared
  - P0: Read Exclusive
  - P0: Write 1
  - P1: Read...

Shared Memory Systems

- Two variants:
  - Shared cache systems
  - Separate cache, bus-based access to shared memory
- Variants:
  - Write-through vs. write-back systems
  - Invalidation-based vs. update-based systems

Write-Back Update Protocol

- Let's have a system where:
  - Write-backs happen when cache line is replaced
  - All writes result in updates of other caches caching the value
- Let's design the simplest write-back update protocol:
  - How many states should it have?
  - What are the significance of the states?

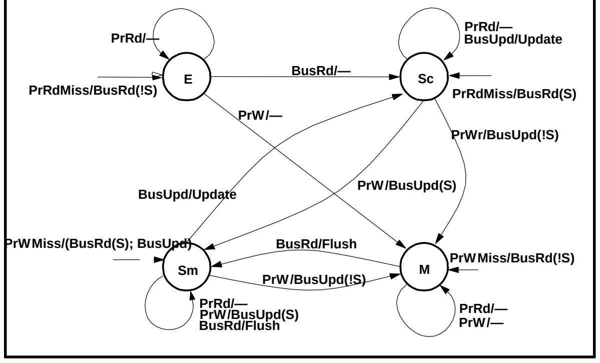
Dragon Write-back Update Protocol

- 4 states
  - Exclusive-clean (E): Myproc & Memory have it
  - Shared clean (Sc): Myproc and other procs may have it
  - Shared modified (Sm): Myproc and other procs may have it, memory does not have updated value (Myproc's responsibility to update memory)
  - Modified(M): Myproc has it, no one else
- Cache block can be:
  - M state on one cache and no one has the same cache block
  - E state on one cache and no one has the same cache block
  - Sc on one or more caches
  - Sm on one cache, Sc on zero or more caches
- No invalid state
  - If in cache, cannot be invalid (but still need to deal with tag mismatches)
- New bus transaction: BusUpd
  - Broadcasts single word written on bus, updates other relevant caches
  - Bandwidth savings

Questions:

- How can we recognize which state should be currently associated with a cache line?
- How do we know that a cache line should be stored in:
  - Exclusive state?
  - Modified state?
  - Shared clean state?
  - Shared modified state?

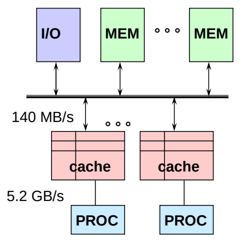
Dragon State Transition Diagram



Lower-level Protocol Choices

- Can shared-modified state be eliminated?
  - If memory is updated on BusUpd transactions (DEC Firefly)
  - Dragon protocol doesn't (assumes DRAM memory slow to update)
- Should replacement of an Sc block be broadcast?
  - Would allow last copy to go to E state (or M state) and not generate updates
    - More overhead on replacements
    - Less overhead for updates

Limits of Bus-based Shared Memory



Assume:
  3 GHz processor w/o cache
=> 12 GB/s inst BW per proc. (32-bit)
=> 3.6 GB/s data BW at 30% load-store

Suppose 98% inst hit rate and 95% data hit rate
=> 240 MB/s inst BW per processor
=> 180 MB/s data BW per processor
⇒ 420 MB/s combined BW

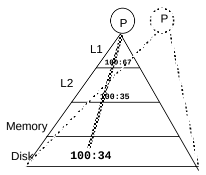
Assuming 4 GB/s bus bandwidth
∴ 10 processors will saturate bus

What is the solution to this scalability problem?

Sharing: a performance problem

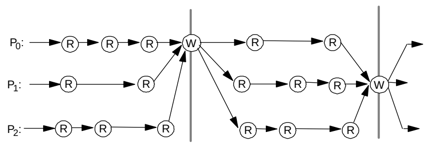
- True sharing
  - Frequent writes to a variable can create a bottleneck
  - OK for read-only or infrequently written data
  - Technique: make copies of the value, one per processor, if this is possible in the algorithm
  - Example problem: the data structure that stores the freelist/heap for malloc/free
- False sharing
  - Cache block may also introduce artifacts
  - Two distinct variables in the same cache block
  - Technique: allocate data used by each processor contiguously, or at least avoid interleaving
  - Example problem: an array of ints, one written frequently by each processor

Intuitive Memory Model



- Reading an address should return the last value written to that address
- Uniprocessor: Different levels can have different values
  - Made consistent at "replacement" time
- Multiprocessor:
  - Approach 1: Use shared top level caches
  - Approach 2: Use separate caches but keep them consistent through snooping

Ordering of operations on single variable



- Bus establishes a total ordering on writes
  - Assuming atomic bus transactions
  - Later we will look at split-phase transactions
- Writes establish a partial order on operations from different processors
  - Doesn't constrain ordering of reads, though bus will order read misses too

Ordering of operations on multiple variables

P_1	P_2
/*Assume initial values of A and B are 0 */	
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

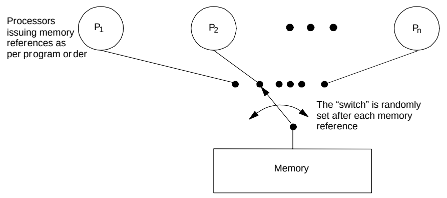
- Question: which of these outputs are valid?
  - [0 0]
  - [0 1]
  - [2 0]
  - [2 1]
- Question: what might cause these kinds of outputs on real machines?
- What's the intuition? This is the memory consistency model

Related Example

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

- Intuition not guaranteed by coherence
- Coherence: preserves order to a single variable
  - Coherence is not enough!
- Also expect memory to respect order between accesses to *different* locations issued by a given process

Sequential Consistency


Processors issuing memory references as per program order
P1
P2
Pn
The "switch" is randomly set after each memory reference
Memory

- Total order achieved by *interleaving* accesses:
  - Maintains *program order;* memory operations, from all processes, appear to [issue, execute, complete] atomically
  - 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]**

Implementing Sequential Consistency

- Sufficient Conditions
  - every process issues memory operations in program order
  - after a read is issued, the issuing process waits for the read to complete before issuing its next operation
  - after a write operation is issued, the issuing process waits for the write to complete (update other caches if necessary) before issuing next memory operation
- How can architectural enhancements violate SC?
  - Out-of-order execution
  - Write-buffers
  - Non-blocking operations
- How can compilers violate SC?
  - Reordering of operations
  - Compilers designed to generate correct uniprocessor code and not correct multiprocessor code

Summary

- Basic shared memory design
  - Shared cache
  - Separate caches: results in coherence problems
  - Snoopy cache: write back vs. write-through, update vs. invalidate
    - State machines for coherence logic
    - Cache coherence implements state machine
    - Scales to about 10's of processors
- Coherence is not enough
  - Require to specify memory consistency model
  - Abstract model for accesses to multiple variables