HY425 Lecture 17: Snoopy coherence protocols

Dimitrios S. Nikolopoulos

University of Crete and FORTH-ICS

December 17, 2010

Outline

Building blocks

Example protocol
- Implementation issues

False sharing

Architectural building blocks

Cache block state transition diagram

- Finite state machine showing how state of block changes between states (e.g. invalid, dirty, shared)
- State indicates if cache block is owned by processor only or shared with other processors

Transactions on broadcast medium (bus or switch)

- Arbitration for shared medium
- Commands (invalidate/update, write back, read)
- All devices attached to bus observe all transactions

Architectural building blocks

Cache block state transition diagram

- Finite state machine showing how state of block changes between states (e.g. invalid, dirty, shared)
- State indicates if cache block is owned by processor only or shared with other processors

Transactions on broadcast medium (bus or switch)

- Arbitration for shared medium
- Commands (invalidate/update, write back, read)
- All devices attached to bus observe all transactions

Architectural building blocks

Serialization

- Write serialization enforced by shared medium
- First processor to write invalidates other copies of block
- Write can not be completed until processor obtains rights to access bus
- Coherence requires serialization of accesses to same cache block. Accomplished through shared medium

Up-to-date copy

- Locate which processor has it, or, force update in memory

Architectural building blocks

Serialization

- Write serialization enforced by shared medium
- First processor to write invalidates other copies of block
- Write can not be completed until processor obtains rights to access bus
- Coherence requires serialization of accesses to same cache block. Accomplished through shared medium

Up-to-date copy

- Locate which processor has it, or, force update in memory

Locating the up-to-date copy

- Write-through policy guarantees that up-to-date copy is in memory
- Write-through is simple to use if there is enough BW
- Write-back harder because most recent copy may be in a cache
- Using the snooping mechanism:
  - Snoop every address placed on bus
  - If processor has latest copy of requested cache block, it provides it in response to a read request and writes back
  - Can be implemented with a cache-to-cache transfer
  - Less BW, scalable to more processors or cores

Locating the up-to-date copy

- Write-through policy guarantees that up-to-date copy is in memory
- Write-through is simple to use if there is enough BW
- Write-back harder because most recent copy may be in a cache
- Using the snooping mechanism:
  - Snoop every address placed on bus
  - If processor has latest copy of requested cache block, it provides it in response to a read request and writes back
  - Can be implemented with a cache-to-cache transfer
  - Less BW, scalable to more processors or cores

Locating the up-to-date copy

- Write-through policy guarantees that up-to-date copy is in memory
- Write-through is simple to use if there is enough BW
- Write-back harder because most recent copy may be in a cache
- Using the snooping mechanism:
  - Snoop every address placed on bus
  - If processor has latest copy of requested cache block, it provides it in response to a read request and writes back
  - Can be implemented with a cache-to-cache transfer
  - Less BW, scalable to more processors or cores

Locating the up-to-date copy

- Write-through policy guarantees that up-to-date copy is in memory
- Write-through is simple to use if there is enough BW
- Write-back harder because most recent copy may be in a cache
- Using the snooping mechanism:
  - Snoop every address placed on bus
  - If processor has latest copy of requested cache block, it provides it in response to a read request and writes back
  - Can be implemented with a cache-to-cache transfer
  - Less BW, scalable to more processors or cores

Cache resources for snooping with write-back

- Normal cache tags can be used for snooping addresses
- Valid bit per block is used by invalidations
- Read misses rely on snooping to locate up-to-date copy in memory or other processor's cache
- Writes need to know if other copies of the same block are cached
  - If no other copies, no need to place write miss on the bus (or other shared medium)
  - If other copies, need to place invalidate message on the bus (or other shared medium)

Cache resources for snooping with write-back

- Normal cache tags can be used for snooping addresses
- Valid bit per block is used by invalidations
- Read misses rely on snooping to locate up-to-date copy in memory or other processor's cache
- Writes need to know if other copies of the same block are cached
  - If no other copies, no need to place write miss on the bus (or other shared medium)
  - If other copies, need to place invalidate message on the bus (or other shared medium)

Cache resources for snooping with write-back

- Normal cache tags can be used for snooping addresses
- Valid bit per block is used by invalidations
- Read misses rely on snooping to locate up-to-date copy in memory or other processor's cache
- Writes need to know if other copies of the same block are cached
  - If no other copies, no need to place write miss on the bus (or other shared medium)
  - If other copies, need to place invalidate message on the bus (or other shared medium)

Cache resources for snooping with write-back

- Normal cache tags can be used for snooping addresses
- Valid bit per block is used by invalidations
- Read misses rely on snooping to locate up-to-date copy in memory or other processor's cache
- Writes need to know if other copies of the same block are cached
  - If no other copies, no need to place write miss on the bus (or other shared medium)
  - If other copies, need to place invalidate message on the bus (or other shared medium)

Tracking if block is shared

- Extra state bit associated with each cache block, complements valid bit and dirty bit
  - Write to shared block places invalidate message on bus and marks cache block as private in some coherence protocols
  - No further invalidations will be sent for that block
  - The processor that writes is the owner of the block
  - Owner changes state of cache block from shared to exclusive

Cache behavior in response to bus transactions

- Every bus transaction must check the cache-address tags
  - Every bus transaction must check the cache-address tags
  - May interfere with simultaneous cache accesses from the processor
- Interference can be reduced by duplicating tags
  - Every bus transaction must check the cache-address tags
  - One set for cache accesses, other set for bus accesses
- Interference can be reduced by using L2 tags
  - Every bus transaction must check the cache-address tags
  - L2 cache is less heavily used than L1
  - Every entry in L1 cache must be present in L2 cache, i.e. the cache should preserve the inclusion property
  - If transaction hits in the L2 cache, it must arbitrate for accessing the L1 cache and potentially updating the block, or updating the state of the block (invalidate), or retrieve the block. This requires stalling the processor

Cache behavior in response to bus transactions

- Every bus transaction must check the cache-address tags
  - Every bus transaction must check the cache-address tags
  - May interfere with simultaneous cache accesses from the processor
- Interference can be reduced by duplicating tags
  - Every bus transaction must check the cache-address tags
  - One set for cache accesses, other set for bus accesses
- Interference can be reduced by using L2 tags
  - Every bus transaction must check the cache-address tags
  - L2 cache is less heavily used than L1
  - Every entry in L1 cache must be present in L2 cache, i.e. the cache should preserve the inclusion property
  - If transaction hits in the L2 cache, it must arbitrate for accessing the L1 cache and potentially updating the block, or updating the state of the block (invalidate), or retrieve the block. This requires stalling the processor

Cache behavior in response to bus transactions

- Every bus transaction must check the cache-address tags
  - Every bus transaction must check the cache-address tags
  - May interfere with simultaneous cache accesses from the processor
- Interference can be reduced by duplicating tags
  - Every bus transaction must check the cache-address tags
  - One set for cache accesses, other set for bus accesses
- Interference can be reduced by using L2 tags
  - Every bus transaction must check the cache-address tags
  - L2 cache is less heavily used than L1
  - Every entry in L1 cache must be present in L2 cache, i.e. the cache should preserve the inclusion property
  - If transaction hits in the L2 cache, it must arbitrate for accessing the L1 cache and potentially updating the block, or updating the state of the block (invalidate), or retrieve the block. This requires stalling the processor

Outline

Building blocks

Example protocol
Implementation issues

False sharing

Example protocol

- Finite-state controller on each processor
- Logically, a separate FSM controller per cache block
  - Requests for different blocks can proceed independently
- In practical implementations, FSM controller allows requests to different blocks to proceed in a pipelined fashion
  - The processing of a request may be initiated before the processing of another request is completed, even though one cache access or one bus access is allowed at a time

Example protocol

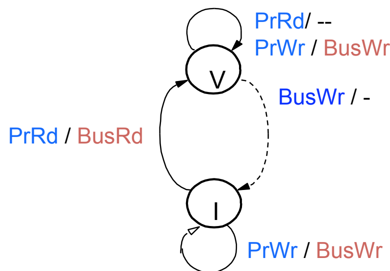
- Finite-state controller on each processor
- Logically, a separate FSM controller per cache block
  - Requests for different blocks can proceed independently
- In practical implementations, FSM controller allows requests to different blocks to proceed in a pipelined fashion
  - The processing of a request may be initiated before the processing of another request is completed, even though one cache access or one bus access is allowed at a time

Example protocol

- Finite-state controller on each processor
- Logically, a separate FSM controller per cache block
  - Requests for different blocks can proceed independently
- In practical implementations, FSM controller allows requests to different blocks to proceed in a pipelined fashion
  - The processing of a request may be initiated before the processing of another request is completed, even though one cache access or one bus access is allowed at a time

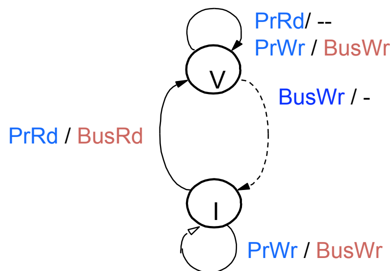
Write-through invalidate protocol

- Two states per cache block
  - As in uniprocessor: V/I
  - State of a block is a p-vector of states with as many elements as the number of processors
  - State bits associated with blocks that are resident in the cache, other blocks considered "invalid"
- Writes invalidate all other cached copies (i.e. all simultaneous readers)



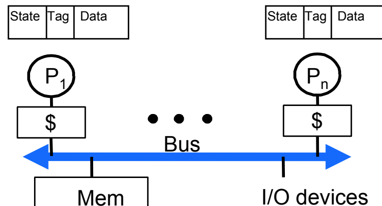
Write-through invalidate protocol

- Two states per cache block
  - As in uniprocessor: V/I
  - State of a block is a p-vector of states with as many elements as the number of processors
  - State bits associated with blocks that are resident in the cache, other blocks considered “invalid”
- Writes invalidate all other cached copies (i.e. all simultaneous readers)



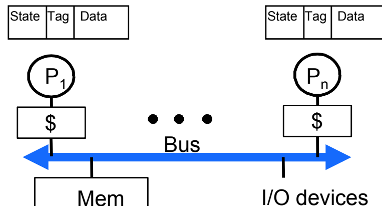
Write-through invalidate protocol

- Two states per cache block
  - As in uniprocessor: V/I
  - State of a block is a p-vector of states with as many elements as the number of processors
  - State bits associated with blocks that are resident in the cache, other blocks considered “invalid”
- Writes invalidate all other cached copies (i.e. all simultaneous readers)



Write-through invalidate protocol

- Two states per cache block
  - As in uniprocessor: V/I
  - State of a block is a p-vector of states with as many elements as the number of processors
  - State bits associated with blocks that are resident in the cache, other blocks considered “invalid”
- Writes invalidate all other cached copies (i.e. all simultaneous readers)



Coherence of 2-state protocol

- Processor tracks state of memory system by issuing loads and stores
- If bus transactions are atomic and system has only one level of cache
  - All steps of a bus transaction complete before next bus transaction starts
  - Processor waits for load/store to complete before issuing next
  - Invalidations applied with bus transactions directly to L1 cache
- Write serialization: All writes serialized in bus and all invalidations applied to caches in bus order.
- Reads: Processor sees writes through reads and reads may happen without appearing on bus

Coherence of 2-state protocol

- Processor tracks state of memory system by issuing loads and stores
- If bus transactions are atomic and system has only one level of cache
  - All steps of a bus transaction complete before next bus transaction starts
  - Processor waits for load/store to complete before issuing next
  - Invalidations applied with bus transactions directly to L1 cache
- Write serialization: All writes serialized in bus and all invalidations applied to caches in bus order.
- Reads: Processor sees writes through reads and reads may happen without appearing on bus

Coherence of 2-state protocol

- Processor tracks state of memory system by issuing loads and stores
- If bus transactions are atomic and system has only one level of cache
  - All steps of a bus transaction complete before next bus transaction starts
  - Processor waits for load/store to complete before issuing next
  - Invalidations applied with bus transactions directly to L1 cache
- Write serialization: All writes serialized in bus and all invalidations applied to caches in bus order.
- Reads: Processor sees writes through reads and reads may happen without appearing on bus

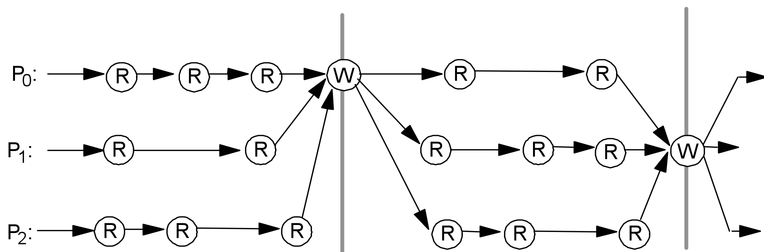
Coherence of 2-state protocol

- Processor tracks state of memory system by issuing loads and stores
- If bus transactions are atomic and system has only one level of cache
  - All steps of a bus transaction complete before next bus transaction starts
  - Processor waits for load/store to complete before issuing next
  - Invalidations applied with bus transactions directly to L1 cache
- Write serialization: All writes serialized in bus and all invalidations applied to caches in bus order.
- Reads: Processor sees writes through reads and reads may happen without appearing on bus

Coherence of 2-state protocol

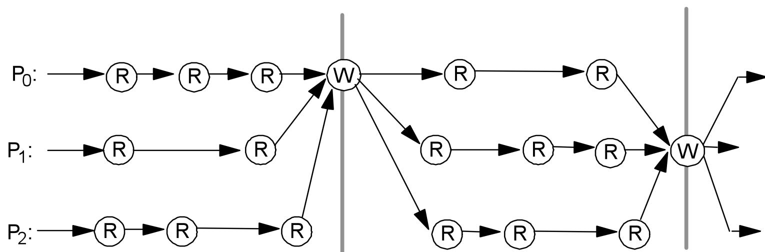
- Processor tracks state of memory system by issuing loads and stores
- If bus transactions are atomic and system has only one level of cache
  - All steps of a bus transaction complete before next bus transaction starts
  - Processor waits for load/store to complete before issuing next
  - Invalidations applied with bus transactions directly to L1 cache
- Write serialization: All writes serialized in bus and all invalidations applied to caches in bus order.
- Reads: Processor sees writes through reads and reads may happen without appearing on bus

Ordering



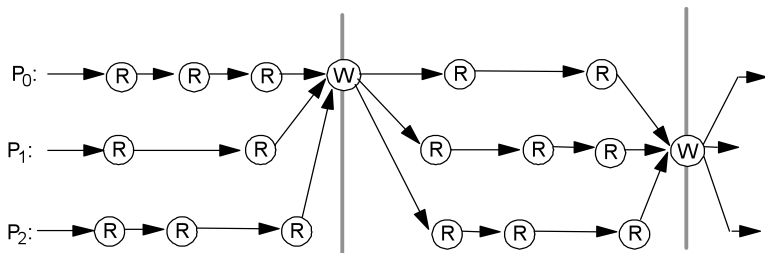
- Writes establish partial order

Ordering



- Writes establish partial order
- Order of writes does not constrain order of reads, however the bus will serialize reads that miss

Ordering



- Writes establish partial order
- Order of writes does not constrain order of reads, however the bus will serialize reads that miss
- Any order of reads between writes is OK

Write-back snoopy protocol

- Invalidation protocol, with write-back caches
  - Cache controller snoops every address on shared medium
  - If cache has dirty copy of requested block, it provides the block in response to a read request
- Memory block has one state:
  - Shared: Clean in all caches and up-to-date in memory
  - Exclusive: Dirty in exactly one cache
  - Uncached: Not present in any cache

Write-back snoopy protocol

- Invalidation protocol, with write-back caches
  - Cache controller snoops every address on shared medium
  - If cache has dirty copy of requested block, it provides the block in response to a read request
- Memory block has one state:
  - Shared: Clean in all caches and up-to-date in memory
  - Exclusive: Dirty in exactly one cache
  - Uncached: Not present in any cache

Write-back snoopy protocol

- Cache block has one state:
  - Shared: block can be read
  - Exclusive: cache has exclusive copy, which is dirty and writable
  - Invalid: block contains no data
- Read misses are transmitted on shared medium and cause all caches to snoop shared medium
- Writes to clean blocks are treated as misses – they are transmitted on shared medium and snooped to invalidate stale copies

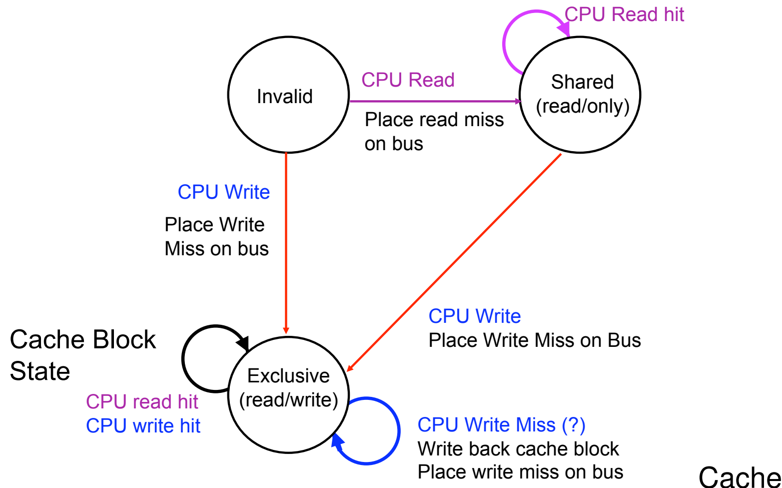
Write-back snoopy protocol

- Cache block has one state:
  - Shared: block can be read
  - Exclusive: cache has exclusive copy, which is dirty and writable
  - Invalid: block contains no data
- Read misses are transmitted on shared medium and cause all caches to snoop shared medium
- Writes to clean blocks are treated as misses – they are transmitted on shared medium and snooped to invalidate stale copies

Write-back snoopy protocol

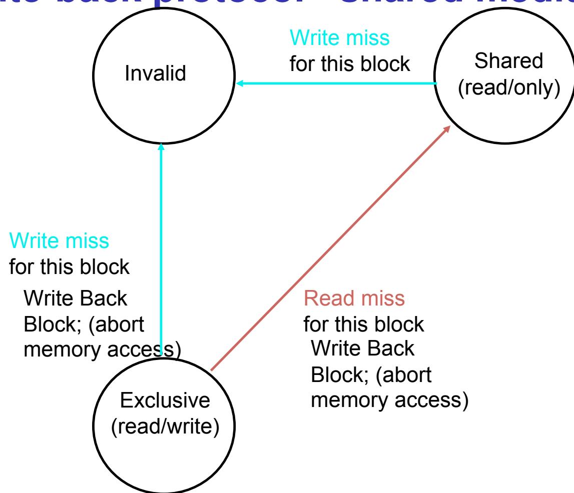
- Cache block has one state:
  - Shared: block can be read
  - Exclusive: cache has exclusive copy, which is dirty and writable
  - Invalid: block contains no data
- Read misses are transmitted on shared medium and cause all caches to snoop shared medium
- Writes to clean blocks are treated as misses – they are transmitted on shared medium and snooped to invalidate stale copies

Write-back protocol - CPU requests



Cache block state transitions upon requests from the CPU

Write-back protocol - shared medium requests



Cache block state transitions upon requests from the shared medium (i.e. misses from other CPUs)

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assume A1 and A2 map to the same cache block and initial cache state is invalid.

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assume A1 and A2 map to the same cache block and initial cache state is invalid.

Example

	P1			P2			Bus				Memory	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1 Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assume A1 and A2 map to the same cache block and initial cache state is invalid.

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

Assume A1 and A2 map to the same cache block and initial cache state is invalid.

Example

step	P1 State	Addr	Value	P2 State	Addr	Value	Bus Action	Proc.	Addr	Value	Memory Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

Assume A1 and A2 map to the same cache block and initial cache state is invalid.

Example

step	P1 State	Addr	Value	P2 State	Addr	Value	Bus Action	Proc.	Addr	Value	Memory Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2							WrMs	P2	A2		A1	10
				Excl.	A2	40	WrBk	P2	A1	20	A1	20

Assume A1 and A2 map to the same cache block and initial cache state is invalid.

Implementation issues

- Write races:
  - Can not update cache until processor reserves the bus
    - Otherwise other processor may get bus first and write the same cache block
- Two-step process:
  - Arbitrate for the bus
  - Place miss on bus and complete operation
- If miss occurs for block while waiting for the bus handle the miss (invalidate) and 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 acquire block in exclusive state
  - Must track and prevent multiple misses for a cache block
- Must support interventions and invalidations

Implementation issues

- Write races:
  - Can not update cache until processor reserves the bus
    - Otherwise other processor may get bus first and write the same cache block
- Two-step process:
  - Arbitrate for the bus
  - Place miss on bus and complete operation
- If miss occurs for block while waiting for the bus handle the miss (invalidate) and 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 acquire block in exclusive state
  - Must track and prevent multiple misses for a cache block
- Must support interventions and invalidations

Implementation issues

- Write races:
  - Can not update cache until processor reserves the bus
    - Otherwise other processor may get bus first and write the same cache block
- Two-step process:
  - Arbitrate for the bus
  - Place miss on bus and complete operation
- If miss occurs for block while waiting for the bus handle the miss (invalidate) and 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 acquire block in exclusive state
  - Must track and prevent multiple misses for a cache block
- Must support interventions and invalidations

Implementation issues

- Write races:
  - Can not update cache until processor reserves the bus
    - Otherwise other processor may get bus first and write the same cache block
- Two-step process:
  - Arbitrate for the bus
  - Place miss on bus and complete operation
- If miss occurs for block while waiting for the bus handle the miss (invalidate) and 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 acquire block in exclusive state
  - Must track and prevent multiple misses for a cache block
- Must support interventions and invalidations

Implementation issues

- Write races:
  - Can not update cache until processor reserves the bus
    - Otherwise other processor may get bus first and write the same cache block
- Two-step process:
  - Arbitrate for the bus
  - Place miss on bus and complete operation
- If miss occurs for block while waiting for the bus handle the miss (invalidate) and 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 acquire block in exclusive state
  - Must track and prevent multiple misses for a cache block
- Must support interventions and invalidations

Implementing snooping caches

- Multiple processors access both addresses and data trough the bus
- Bus needs commands for coherence in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, invalidate or update
- Since every bus transaction checks cache tags, cache tag checking may interfere with CPU tag checking
  - Use duplicate tags in L1 and coherence control to allow parallel tag checks
  - Use inclusion between L2 and L1 so that coherence checks are done in L2
    - Block size and associativity of L2 defines block size and associativity of L1

Implementing snooping caches

- Multiple processors access both addresses and data trough the bus
- Bus needs commands for coherence in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, invalidate or update
- Since every bus transaction checks cache tags, cache tag checking may interfere with CPU tag checking
  - Use duplicate tags in L1 and coherence control to allow parallel tag checks
  - Use inclusion between L2 and L1 so that coherence checks are done in L2
    - Block size and associativity of L2 defines block size and associativity of L1

Implementing snooping caches

- Multiple processors access both addresses and data trough the bus
- Bus needs commands for coherence in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, invalidate or update
- Since every bus transaction checks cache tags, cache tag checking may interfere with CPU tag checking
  - Use duplicate tags in L1 and coherence control to allow parallel tag checks
  - Use inclusion between L2 and L1 so that coherence checks are done in L2
    - Block size and associativity of L2 defines block size and associativity of L1

Implementing snooping caches

- Multiple processors access both addresses and data trough the bus
- Bus needs commands for coherence in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, invalidate or update
- Since every bus transaction checks cache tags, cache tag checking may interfere with CPU tag checking
  - Use duplicate tags in L1 and coherence control to allow parallel tag checks
  - Use inclusion between L2 and L1 so that coherence checks are done in L2
    - Block size and associativity of L2 defines block size and associativity of L1

Optimization: MESI protocol

- Assume a cache block is read by only one processor
- Reading processor should be able to write to the block without notifying other processors, if other processors do not have copies of the block in their cache
- Solution: split the exclusive state into modified (both exclusive and dirty) or exclusive (unmodified)
- Bus read requests to exclusive block makes block shared
- Modified blocks are written back upon replacement
- Block in shared or exclusive state has up-to-date value in memory

Optimization: MESI protocol

- Assume a cache block is read by only one processor
- Reading processor should be able to write to the block without notifying other processors, if other processors do not have copies of the block in their cache
- Solution: split the exclusive state into modified (both exclusive and dirty) or exclusive (unmodified)
- Bus read requests to exclusive block makes block shared
- Modified blocks are written back upon replacement
- Block in shared or exclusive state has up-to-date value in memory

Optimization: MESI protocol

- Assume a cache block is read by only one processor
- Reading processor should be able to write to the block without notifying other processors, if other processors do not have copies of the block in their cache
- Solution: split the exclusive state into modified (both exclusive and dirty) or exclusive (unmodified)
- Bus read requests to exclusive block makes block shared
- Modified blocks are written back upon replacement
- Block in shared or exclusive state has up-to-date value in memory

Optimization: MESI protocol

- Assume a cache block is read by only one processor
- Reading processor should be able to write to the block without notifying other processors, if other processors do not have copies of the block in their cache
- Solution: split the exclusive state into modified (both exclusive and dirty) or exclusive (unmodified)
- Bus read requests to exclusive block makes block shared
- Modified blocks are written back upon replacement
- Block in shared or exclusive state has up-to-date value in memory

Optimization: MESI protocol

- Assume a cache block is read by only one processor
- Reading processor should be able to write to the block without notifying other processors, if other processors do not have copies of the block in their cache
- Solution: split the exclusive state into modified (both exclusive and dirty) or exclusive (unmodified)
- Bus read requests to exclusive block makes block shared
- Modified blocks are written back upon replacement
- Block in shared or exclusive state has up-to-date value in memory

Optimization: MESI protocol

- Assume a cache block is read by only one processor
- Reading processor should be able to write to the block without notifying other processors, if other processors do not have copies of the block in their cache
- Solution: split the exclusive state into modified (both exclusive and dirty) or exclusive (unmodified)
- Bus read requests to exclusive block makes block shared
- Modified blocks are written back upon replacement
- Block in shared or exclusive state has up-to-date value in memory

Optimization: MOESI protocol

- Assume block in modified state in MESI is requested by other processor
- Block must be written back to memory and transitions to shared in cache
- MOESI protocol avoids the write-back to memory
  - Block in modified moves to owned state
  - Block is transferred to requesting processor through a cache-to-cache transfer
  - Owned state is like shared, except that the block is not up-to-date in memory

Optimization: MOESI protocol

- Assume block in modified state in MESI is requested by other processor
- Block must be written back to memory and transitions to shared in cache
- MOESI protocol avoids the write-back to memory
  - Block in modified moves to owned state
  - Block is transferred to requesting processor through a cache-to-cache transfer
  - Owned state is like shared, except that the block is not up-to-date in memory

Optimization: MOESI protocol

- Assume block in modified state in MESI is requested by other processor
- Block must be written back to memory and transitions to shared in cache
- MOESI protocol avoids the write-back to memory
  - Block in modified moves to owned state
  - Block is transferred to requesting processor through a cache-to-cache transfer
  - Owned state is like shared, except that the block is not up-to-date in memory

Limitations of bus-based cache coherence

- Single memory serves all CPUs
  - Need multiple memory banks
- Bus must support coherence traffic and normal memory traffic
  - Solution: multiple buses or interconnection networks
- Example: AMD Opteron
  - Memory connected directly to each multi-core processor
  - Point-to-point connections for up to 4 multi-core processors
  - Remote memory access latency close to local memory access latency

Limitations of bus-based cache coherence

- Single memory serves all CPUs
  - Need multiple memory banks
- Bus must support coherence traffic and normal memory traffic
  - Solution: multiple buses or interconnection networks
- Example: AMD Opteron
  - Memory connected directly to each multi-core processor
  - Point-to-point connections for up to 4 multi-core processors
  - Remote memory access latency close to local memory access latency

Limitations of bus-based cache coherence

- Single memory serves all CPUs
  - Need multiple memory banks
- Bus must support coherence traffic and normal memory traffic
  - Solution: multiple buses or interconnection networks
- Example: AMD Opteron
  - Memory connected directly to each multi-core processor
  - Point-to-point connections for up to 4 multi-core processors
  - Remote memory access latency close to local memory access latency

Outline

Building blocks

Example protocol
Implementation issues

False sharing

3 Cs to 4 Cs

- 3 Cs model describes uniprocessor cache miss traffic
- Fourth C refers to misses caused by cache coherence
  - Invalidations that result in subsequent cache misses

3 Cs to 4 Cs

- 3 Cs model describes uniprocessor cache miss traffic
- Fourth C refers to misses caused by cache coherence
  - Invalidations that result in subsequent cache misses

Coherence misses

- True sharing misses arise because two processors need the same word in the same block and one writes
  - Write invalidates block on other processor
  - Read by other processor incurs a miss
  - Miss would always occur regardless of block size

Coherence misses

- False sharing arise because two processors need different words in the same block and one writes
  - Write invalidates block on other processor
  - Read by other processor incurs a miss
  - Miss would not occur if the block size were one word

True vs. false sharing

Time	P1	P2	True, False, Hit, Why?
1	Write X1		True miss; invalidate X1 on P2
2		Read X2	
3	Write X1		
4		Write X2	
5	Read X2		

Assume x1 and x2 in same cache block and P1 and P2 have the block in cache in shared state.

True vs. false sharing

Time	P1	P2	True, False, Hit, Why?
1	Write X1		True miss; invalidate X1 on P2
2		Read X2	False miss; X1 irrelevant to P2
3	Write X1		
4		Write X2	
5	Read X2		

Assume x1 and x2 in same cache block and P1 and P2 have the block in cache in shared state.

True vs. false sharing

Time	P1	P2	True, False, Hit, Why?
1	Write X1		True miss; invalidate X1 on P2
2		Read X2	False miss; X1 irrelevant to P2
3	Write X1		False miss; X1 irrelevant to P2
4		Write X2	
5	Read X2		

Assume x1 and x2 in same cache block and P1 and P2 have the block in cache in shared state.

True vs. false sharing

Time	P1	P2	True, False, Hit, Why?
1	Write X1		True miss; invalidate X1 on P2
2		Read X2	False miss; X1 irrelevant to P2
3	Write X1		False miss; X1 irrelevant to P2
4		Write X2	False miss; X1 irrelevant to P2
5	Read X2		

Assume x1 and x2 in same cache block and P1 and P2 have the block in cache in shared state.

True vs. false sharing

Time	P1	P2	True, False, Hit, Why?
1	Write X1		True miss; invalidate X1 on P2
2		Read X2	False miss; X1 irrelevant to P2
3	Write X1		False miss; X1 irrelevant to P2
4		Write X2	False miss; X1 irrelevant to P2
5	Read X2		True miss; invalid X2 on P1

Assume x1 and x2 in same cache block and P1 and P2 have the block in cache in shared state.