Advanced Computer Architecture
高级计算机体系结构

第10讲：Thread-Level Parallelism (2)

张献伟

xianweiz.github.io

DCS5367, 12/7/2021

Review Questions

- Q1: SMP?

  Symmetric (shared-memory) multiprocessor

- Q2: what is NUMA?

  Non-uniform memory access, occurring in DSM

- Q3: What is cache coherence issue?

  Processors see different values of the same data.

- Q4: two classes of protocols.

  Snooping, directory.

- Q5: write invalidation protocol.

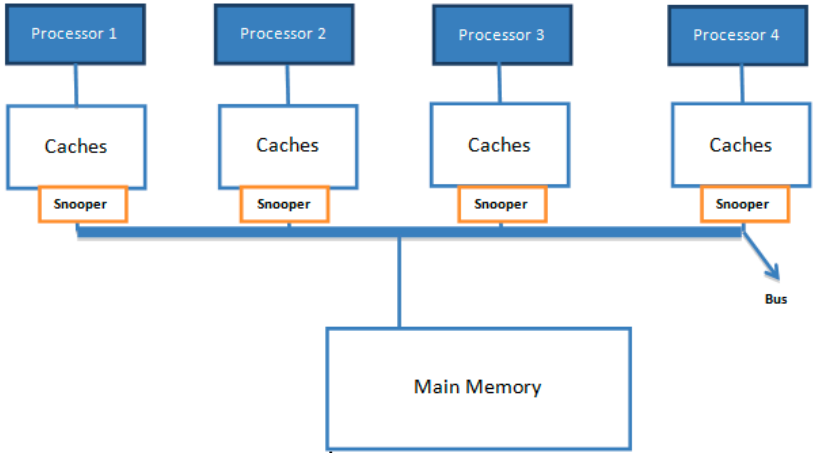
  On write, invalidate all other copies.

- Q6: how to handle miss in snoopy protocol?

  Write-through: from memory, write-back: memory or cache

Review: Snoopy Implementation[窥探实现]

- Key is to use bus, or another broadcast medium, to perform invalidates
- To perform an invalidate
  - The processor simply acquires bus access and broadcasts the address to be invalidated on the bus[获得总线，广播地址]
  - All processors continuously snoop on the bus, watching the addresses[窥探总线，收听地址]
  - The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache is invalidated[核对地址，作废数据]



Review: Snoopy Implementation (cont.)

- When a write to a block that is shared occurs,[写到共享块]
  - The writing processor must acquire bus access to broadcast its invalidation
- If two processors attempt to write shared blocks at the same time,[两个处理器同时写到共享块]
  - Their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus[串行‘无效’操作]
  - The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated[作废数据]
  - If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes[串行写操作]

Review: Snoopy Implementation (cont.)

- Locate a data item when a cache miss occurs,[找到数据]
  - For write-through cache, easy to find the recent value[写通]
    - All written data are always sent to the memory
  - For write-back cache, harder to find the most recent value[写回]
    - The newest value can be in a private cache rather than in the shared cache or memory
- Happily, write-back caches can use the same snooping scheme both for cache misses and for writes[同样窥探]
  - Each processor snoops every address placed on the shared bus[每个处理器窥探每个地址]
  - If a processor finds that it has a dirty copy of the requested cache block, it provides that block in response to the read request and causes the memory (or L3) access to be aborted[某个处理器拥有脏数据→ 响应]

Snoopy Implementation (cont.)

- Normal cache tags can be used to implement snooping, and the valid bit for each block makes invalidation easy to implement
  - Read misses, whether generated by an invalidation or by other events, are simply relying on the snooping capability
  - For writes, we'd like to know whether any other copies of the block are cached, because
    - If no other copies, then the write need not be placed on the bus
- Add an extra bit to track whether a block is shared
  - The bit is used to decide whether a write must generate an invalidate
    - Write to shared: invalidate, then mark block as "exclusive"
    - Sole copy of a cache block is normally called "owner"

Example Protocol

- Invalidation protocol for write-back caches
- Each data block can be[数据块状态]
  - Uncached: not in any cache
  - Clean in one or more caches and up-to-date in memory, or
  - Dirty in exactly one cache

Dirty in more caches???

- Correspondingly, we record the coherence state of each block in a cache as[一致性状态]
  - Invalid: block contains no valid data
  - Shared: a clean block (can be shared by other caches), or
  - Modified/Exclusive: a dirty block (cannot be in any other cache)

MSI protocol = Modified/shared/invalid

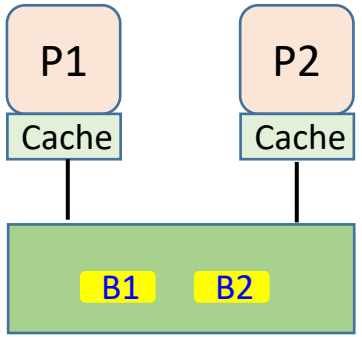
Makes sure that if a block is dirty in one cache, it is not valid in any other cache and that a read request gets the most updated data

Example Protocol (cont.)

- A read miss to a block in a cache, C1, generates a bus transaction
  - If another cache, C2, has the block "modified", it has to write back the block before memory supplies it
    - C1 gets data from the bus and the block becomes "shared" in both caches
- A write hit to a shared block in C1 forces an "Invalidate"
  - Other caches that have the block should invalidate it – the block then becomes "modified" in C1
- A write hit to a modified block does not generate "Invalidate" or change of state
- A write miss (to an invalid block) in C1 generates a bus transaction
  - If a cache, C2, has the block as "shared", it invalidates it
  - If a cache, C2, has the block in "modified", it writes back the block and changes it state in C2 to "invalid"
  - If no cache supplies the block, the memory will supply it
  - When C1 gets the block, it sets its state to "modified"

Example

- Assume that
  - Blocks *B1* and *B2* map to the same cache location *L*
  - Initially neither *B1* or *B2* is cached
  - Block size = one word



Event	In P1's cache	In P2's cache
	L = invalid	**L = invalid**
P1 writes 10 to B1 (write miss)	L <- B1 = 10 (modified)	L = invalid
P1 reads B1 (read hit)	L <- B1 = 10 (modified)	L = invalid
P2 reads B1 (read miss)	B1 is written back L <- B1 = 10 (shared)	L <- B1 = 10 (shared)
P2 writes 20 to B1 (write hit)	L = invalid	Put invalidate B1 on bus L <- B1 = 20 (modified)
P2 writes 40 to B2 (write miss)	L = invalid	B1 is written back L <- B2 = 40 (modified)
P1 reads B1 (read miss)	L <- B1 = 20 (shared)	L <- B2 = 40 (modified)

Example (cont.)

- When an invalidate or a write miss is placed on the bus, any cores whose private caches have copies of the block invalidate it
- For a write miss, if the block is exclusive in just one private cache, that cache also writes back the block
  - Otherwise, the data can be read from the shared cache or memory

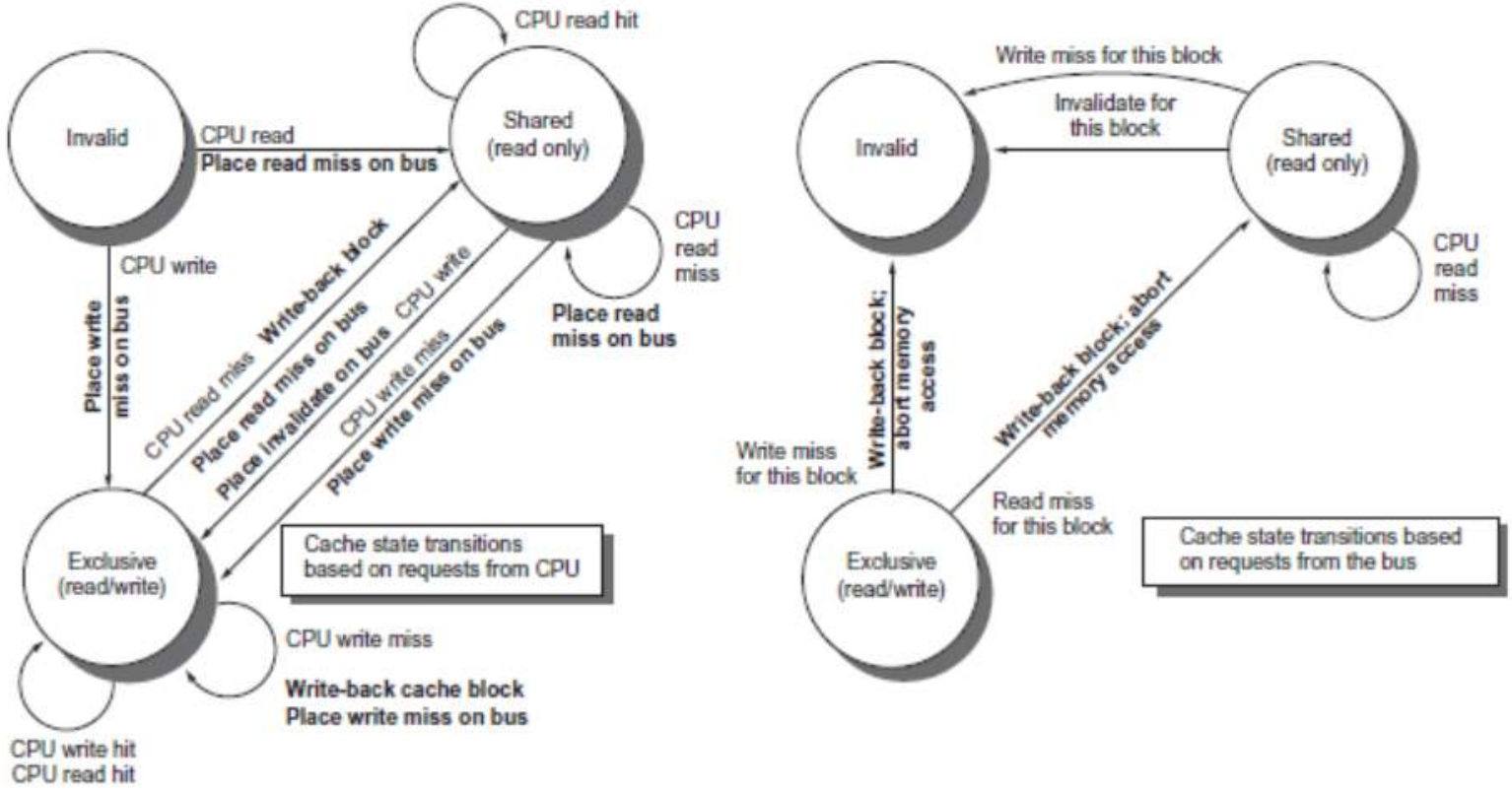
Event	In P1's cache	In P2's cache
	L <- B1 = 20 (shared)	**L <- B2 = 40 (modified)**
P1 writes 30 to B1 (write hit)	Put invalidate B1 on bus L <- B1 = 30 (modified)	L <- B2 = 40 (modified)
P2 writes 50 to B1 (write miss)	B1 is written back L = invalid	B2 is written back L <- B1 = 50 (modified)
P1 reads B1 (read miss)	L <- B1 = 50 (shared)	B1 is written back L <- B1 = 50 (shared)
P2 reads B2 (read miss)	L <- B1 = 50 (shared)	L <- B2 = 40 (shared)
P1 writes 60 to B2 (write miss)	L <- B2 = 60 (modified)	L = invalid

The Protocol

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, because they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to read shared data: place cache block on bus, write-back block, and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Formal Specification[形式化定义]

- Finite state transition diagram for a single private cache block[状态转换图]
  - Transitions based on processor and bus requests, respectively



MSI Issues & Extensions[扩展]

- Complications for the basic MSI protocol
  - Operations are not atomic[非原子操作]
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
  - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- MSI: always invalidate before writing
- Extensions
  - Adding additional states and transitions, which optimize certain behaviors, possibly resulting in improved performance
  - Two common extensions
    - **MESI**: new 'Exclusive'
    - **MOESI**: new 'Exclusive' and 'Owner'

MESI and MOESI

- MESI adds state Exclusive
  - Shared: Exclusive (only one cache) + Shared (2 or more caches)
  - Indicate when a cache block is resident only in a single cache but is clean
  - A subsequent write to a block in *E* state by the same core need not acquire bus access or generate an invalidate
- MOESI further adds state Owner
  - Shared: Shared Modified (O) + Shared Clean (S)
  - Indicate that the associated block is owned by that cache and out-of-date in memory
  - In MSI/MESI, when sharing a block in *M* state, the state is changed to *S*, and the block must be written back to memory
  - In MOESI, the block can be changed from *M* to *O* without writing it to memory

Performance of SMPs: Misses

- In a multicore using a snooping coherence protocol, overall cache performance is a combination of
  - The behavior of uniprocessor cache miss traffic
  - The traffic caused by communication, resulting in invalidations and subsequent cache misses
- Three C's classification of uniprocessor misses
  - Capacity(容量), compulsory(冷启动), conflict(地址冲突)
- Coherence misses caused by interprocessor communication[一致性缺失]
  - **True sharing misses**: directly arise from the sharing of data among processors
  - **False sharing misses**: the miss would not occur if the block size were a single word

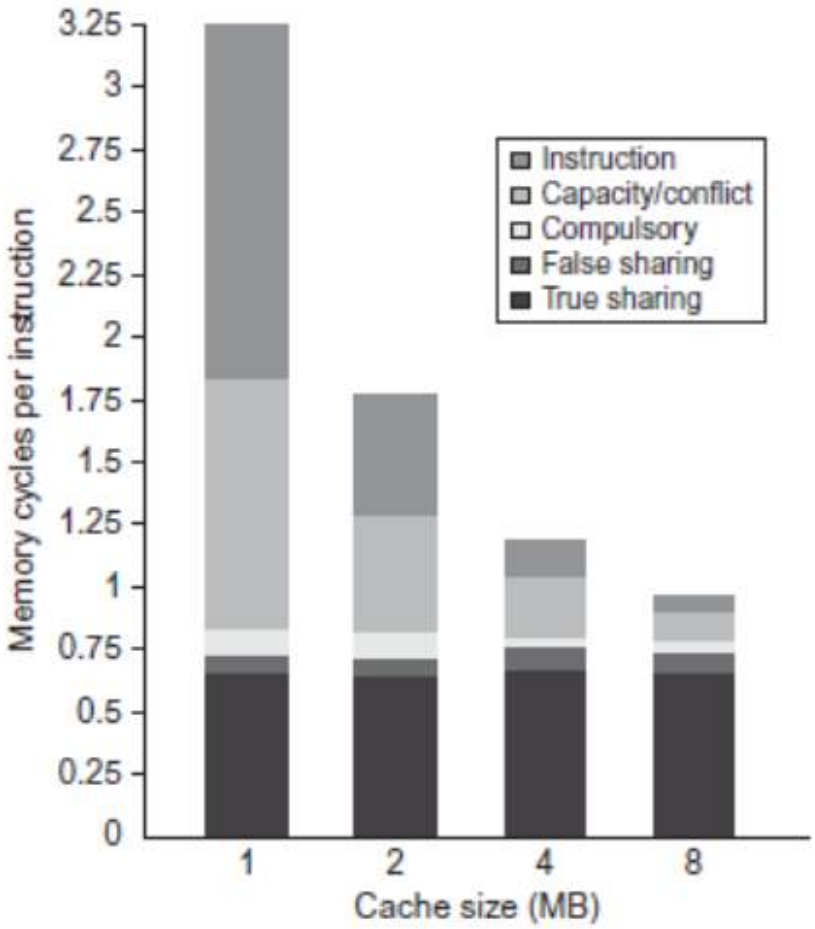
Performance of SMPs: Misses (cont.)

- **True sharing misses**, in an invalidation-based protocol
  - The first write by a processor to a shared block causes an invalidation to establish ownership of that block (invalidate all)
  - When another processor tries to read a modified word in that block, a miss occurs and the resultant block is transferred (invalidated by the store)
- **False sharing misses**
  - Caused by the coherence alg. with a single valid bit per block
  - Occurs when a block is invalidated (and a subsequent reference causes a miss)
    - Some word in the block, other than the one being read, is written into

Time	P1	P2
1	Write x1	
2		Read x2

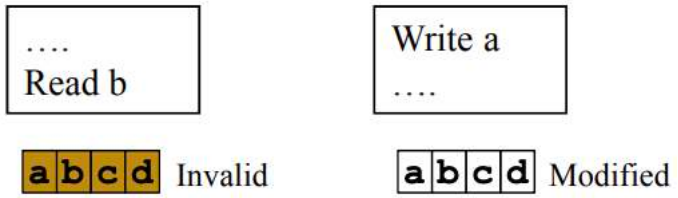
False sharing miss: x2 was invalidated by 'write x1' in P1

Performance of SMPs: Result



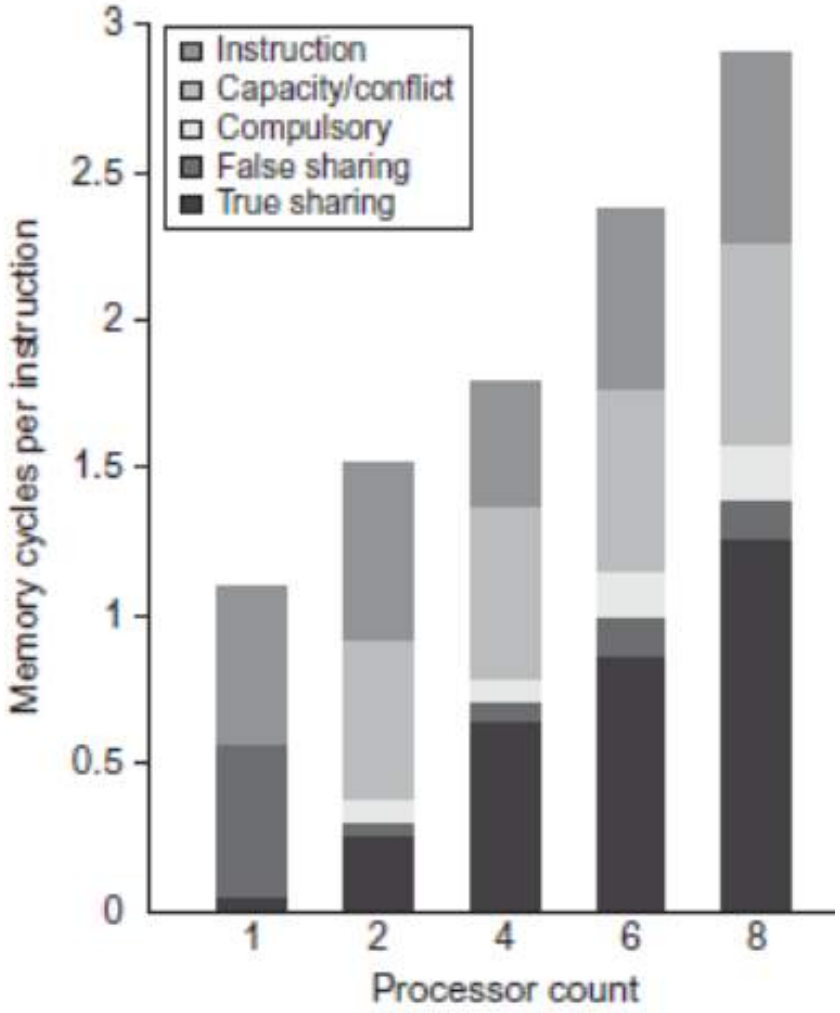
CPI for commercial benchmarks

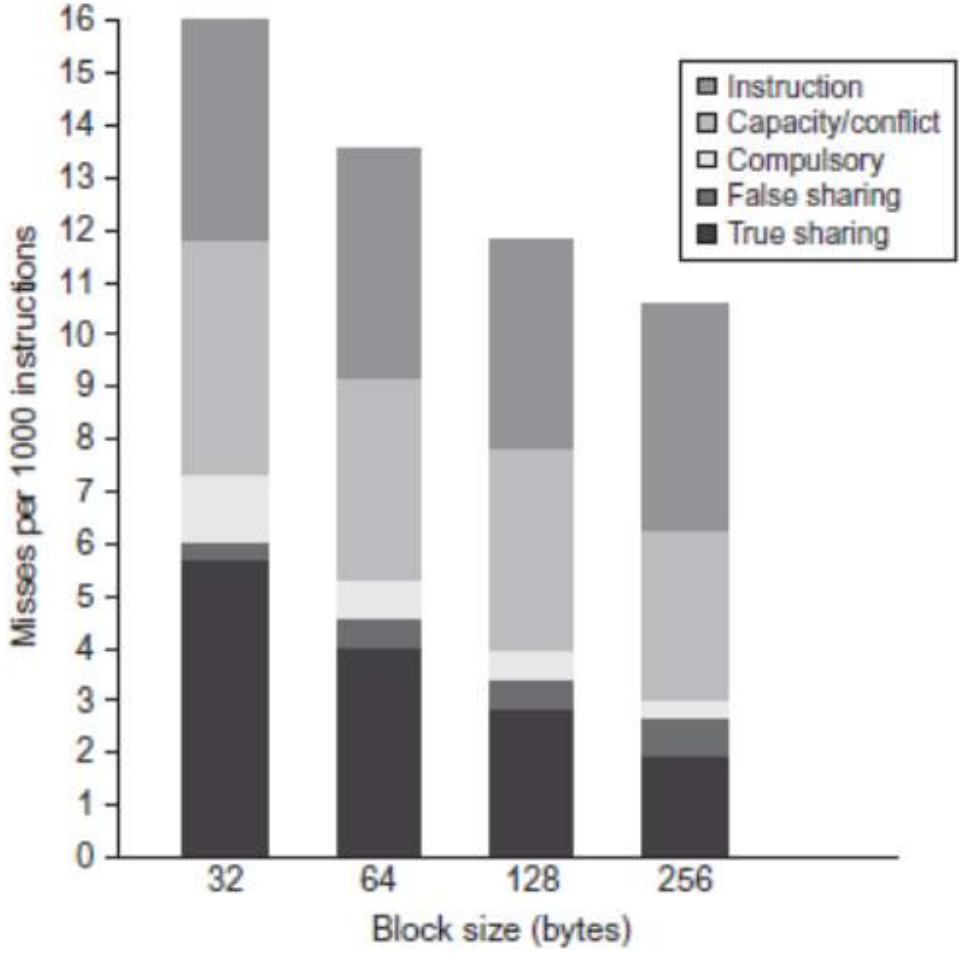
- Coherence misses:
  - True sharing misses
    - Write to a shared block
    - Read an invalid block
  - False sharing misses
    - Read an unmodified word in an invalidated block



Increasing the cache size eliminates most of the uniprocessor misses while leaving the multiprocessor misses untouched.

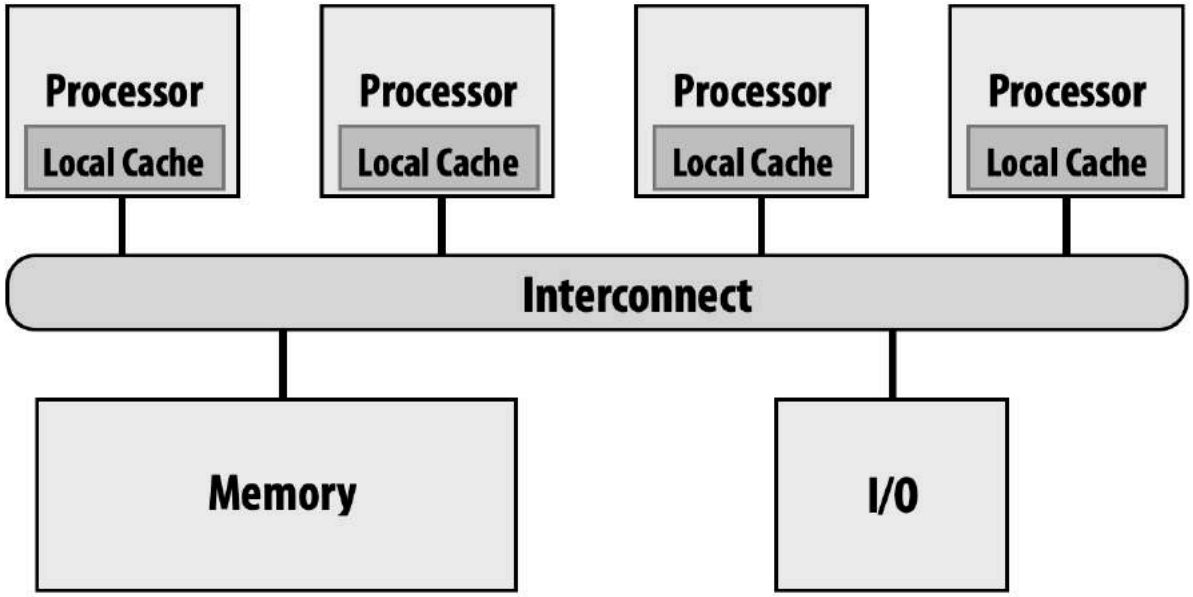
Performance of SMPs: Result (cont.)





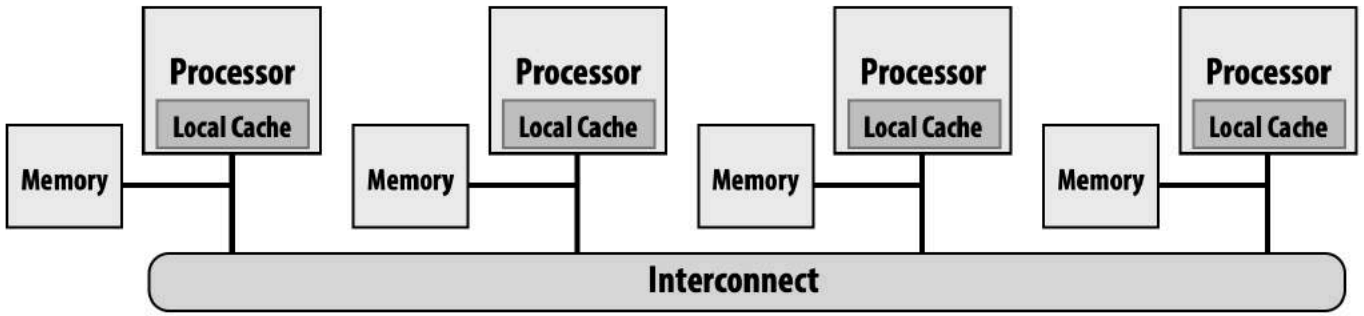
Limits of Snooping Protocol[限制]

- Snooping cache coherence protocols rely on broadcasting coherence info to all processors over the chip inter-connect
  - Cache miss occurred, triggering cache communicated with all other caches



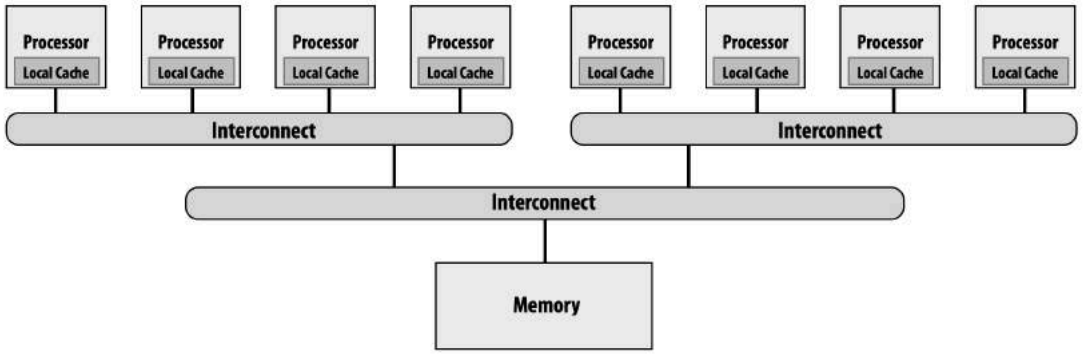
Limits of Snooping Protocol (cont.)

- On a non-uniform memory access (NUMA) shared memory system, regions of memory are located near the processors increases scalability
  - Yield higher aggregate bandwidth and reduced latency
- NUMA does little good if the coherence protocol can't be scaled
  - Processor can access nearby memory, but need to broadcast to all other processors (overhead)

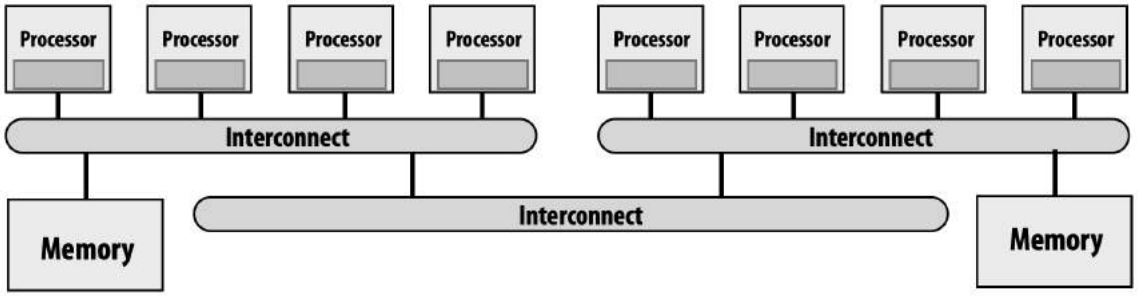


Scaling Cache Coherence to Large Machines

- One possible solution: **hierarchical snooping**[层级化]
  - Use snooping coherence at each level

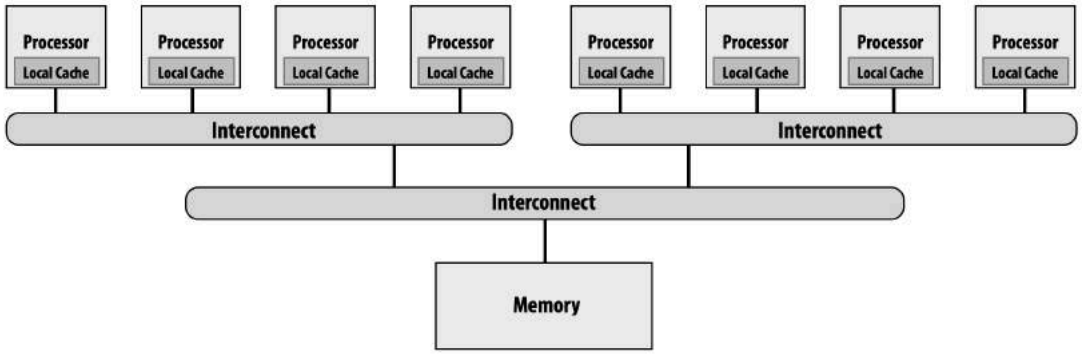


- Another: with **memory localized** with the groups of processors, rather than centralized



Scaling Cache Coherence (cont.)

- One possible solution: hierarchical snooping
  - Use snooping coherence at each level



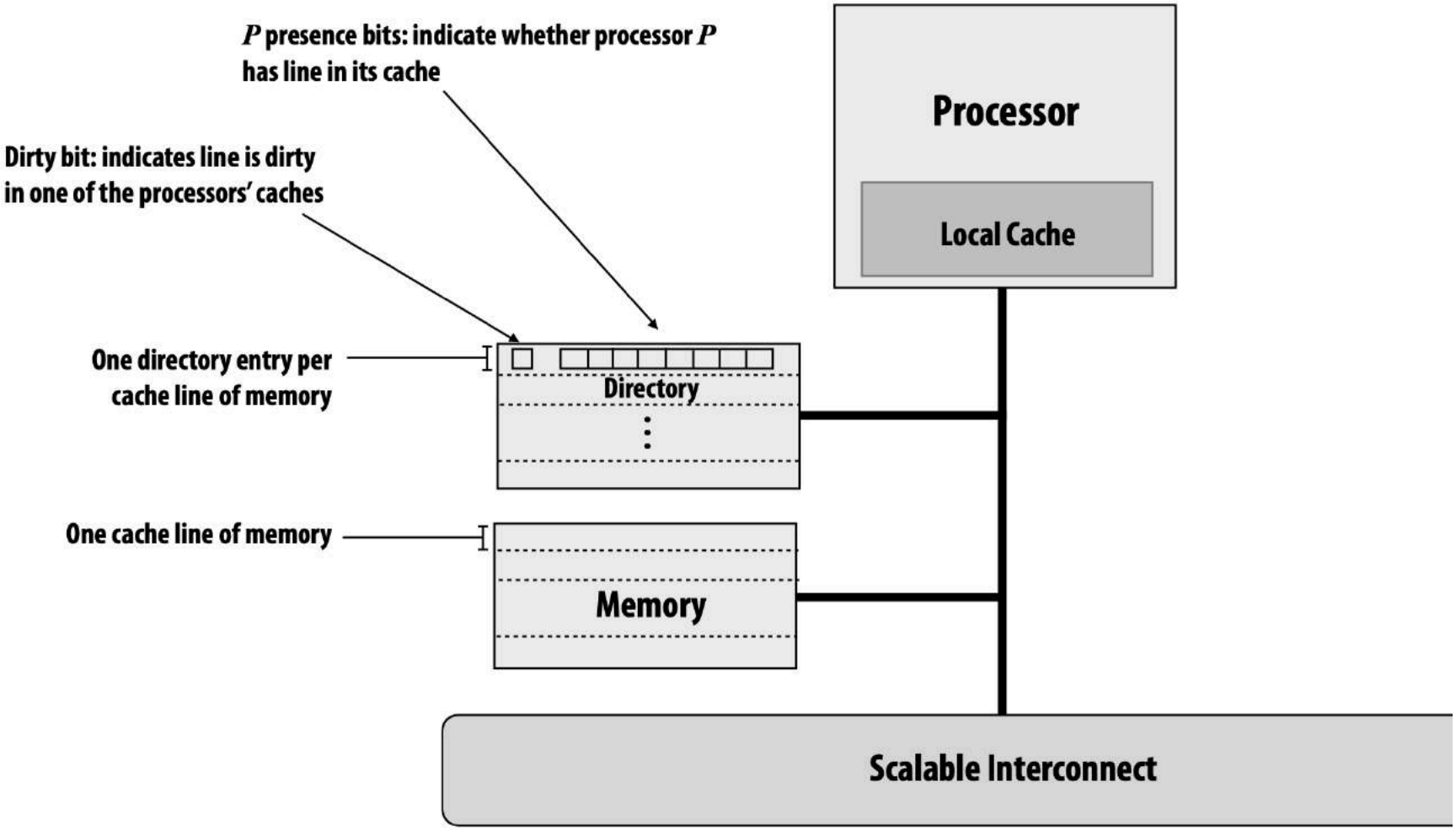
  - Advantages
    - Relatively simple to build (already have to deal with similar issues due to multi-level caches)
  - Disadvantages
    - The root of network may become a performance bottleneck
    - Larger latencies than direct communication
    - Doesn't apply to more general network topologies

Scalable Coherence using Directories

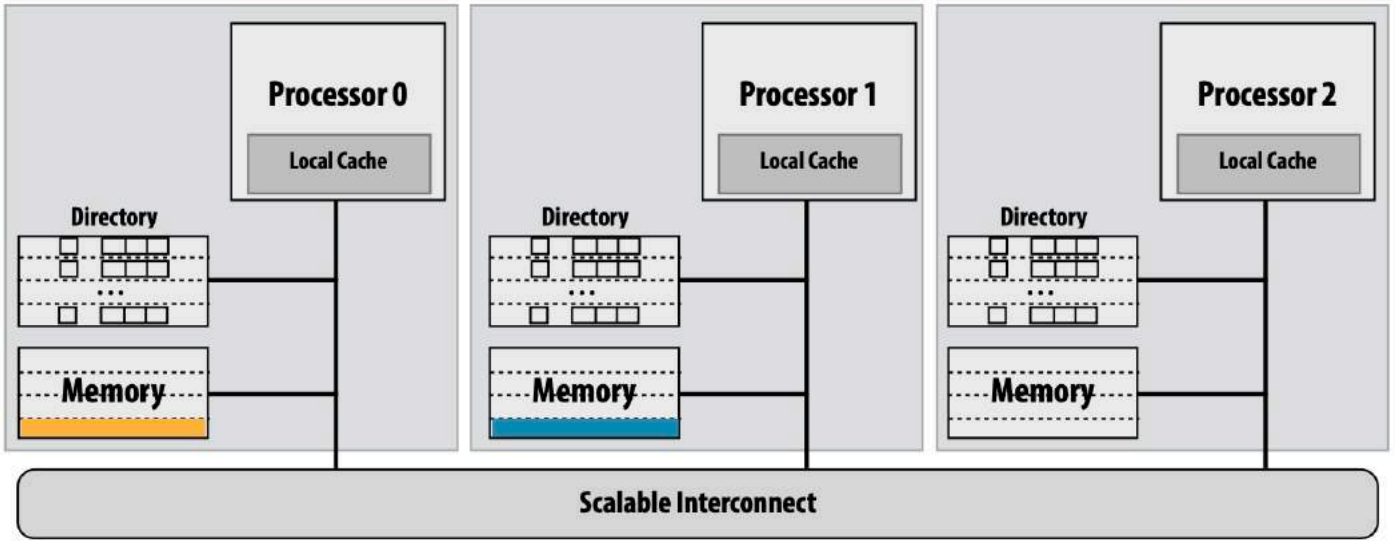
- To avoid broadcast by storing info about status of the line in one place: directory
  - The directory entry for a cache line contains information about the state of the cache line in all caches
  - Caches look up information from the directory as necessary
  - Cache coherence is maintained by point-to-point messages between the caches (not by broadcast mechanisms)

- Theoretical advantages of directory-based approach
  - The root of network won't be the performance bottleneck
  - Can apply to more general network topologies(e.g. meshes, cubes)

Simple Directory Protocol Impl.



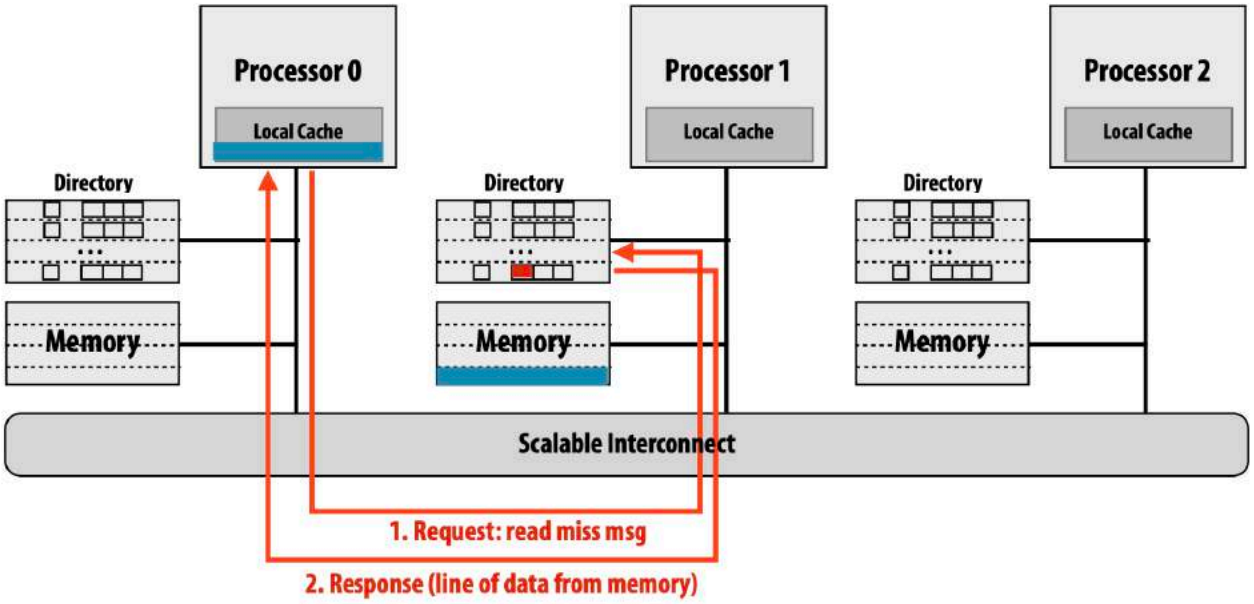
Distributed Directory: Partition



- Directory partition is co-located with memory it describes
- "Home node" of a line: node with memory holding the corresponding data for the line
  - For example: node 0 is the home node of orange line, node 1 is the home node of blue node
- "Requesting node": node containing processor requesting line

Example: read miss to clean line

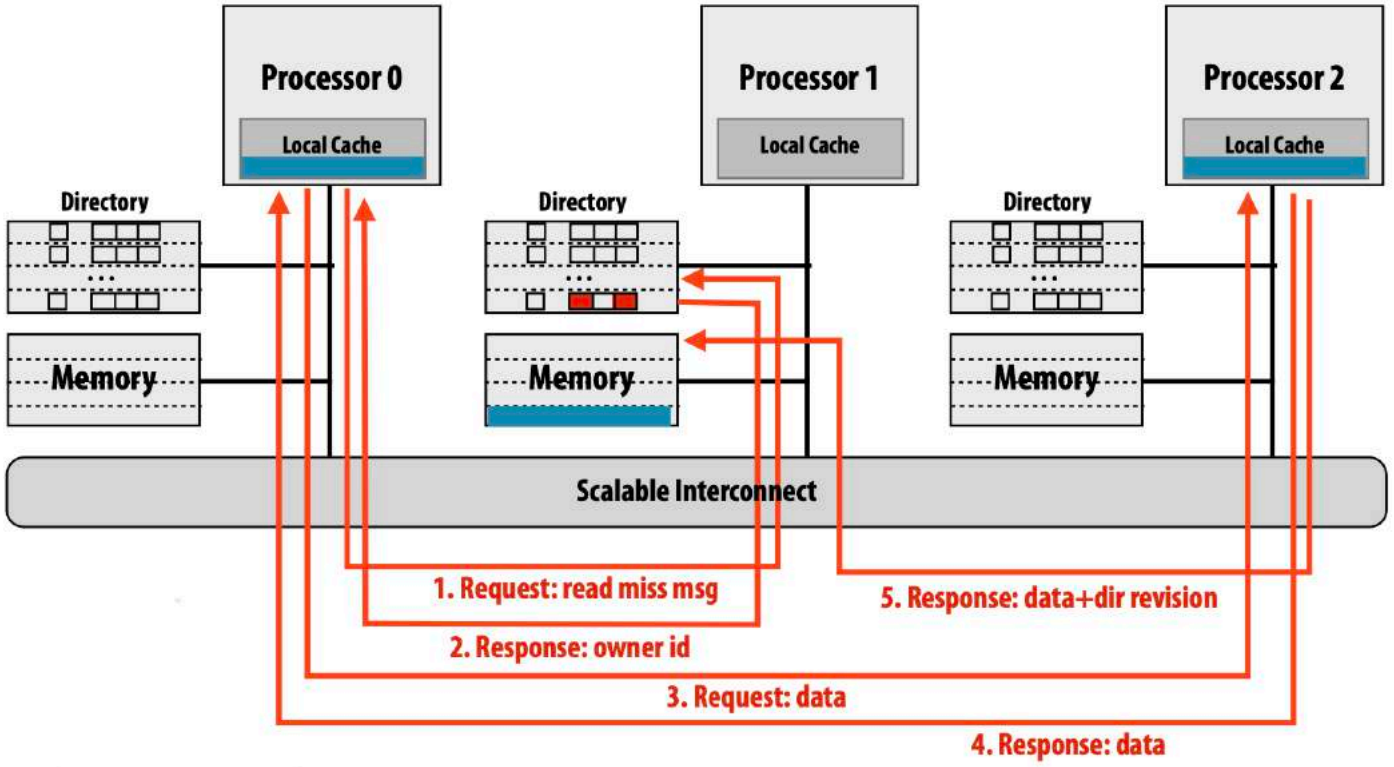
Read from main memory by processor 0 of blue line(not dirty)



- Read miss message sent to home node of requested line
- Home directory checks entry for line
  - If dirty bit of line is OFF, respond with contents from memory, set presence[0] to true(to indicate line is cached by processor 0)

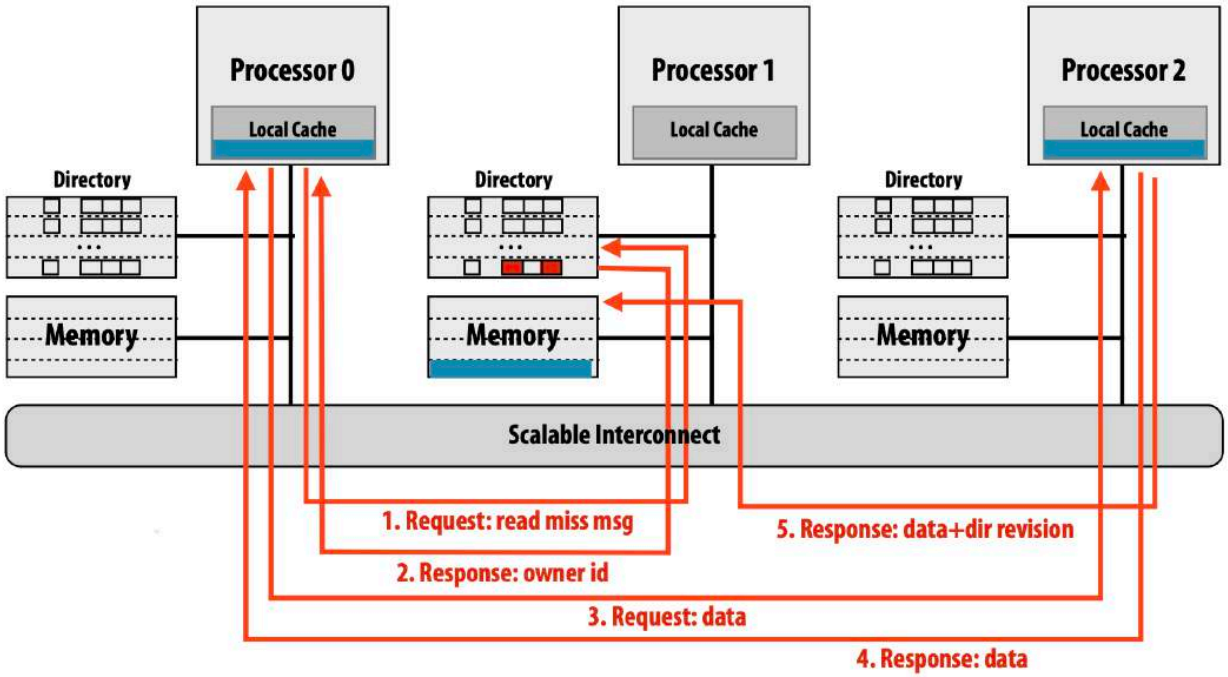
Example: read miss to dirty line

- Read from main memory by processor 0 of blue line
  - Dirty and its content is in P2's cache



Example: read miss to dirty line (cont.)

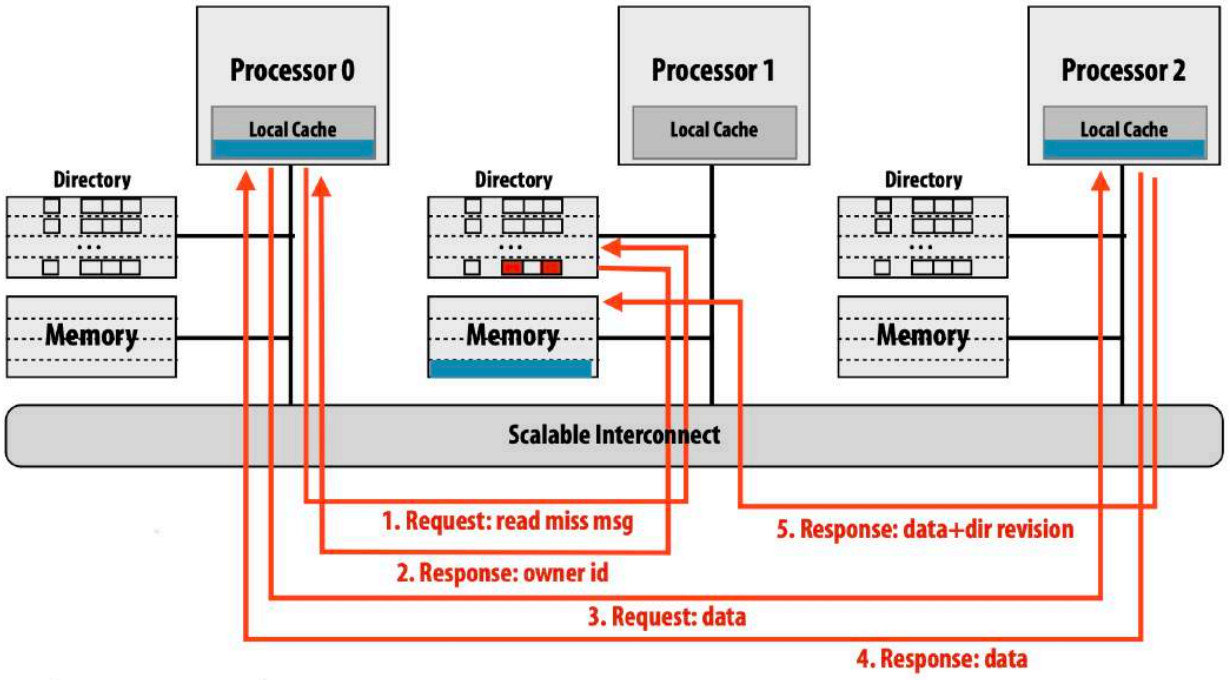
Procedure



1. If dirty bit is ON, data must be sourced by another processor
2. Home node responds with id of line owner
3. Requesting node requests data from owner

Example: read miss to dirty line (cont.)

Procedure

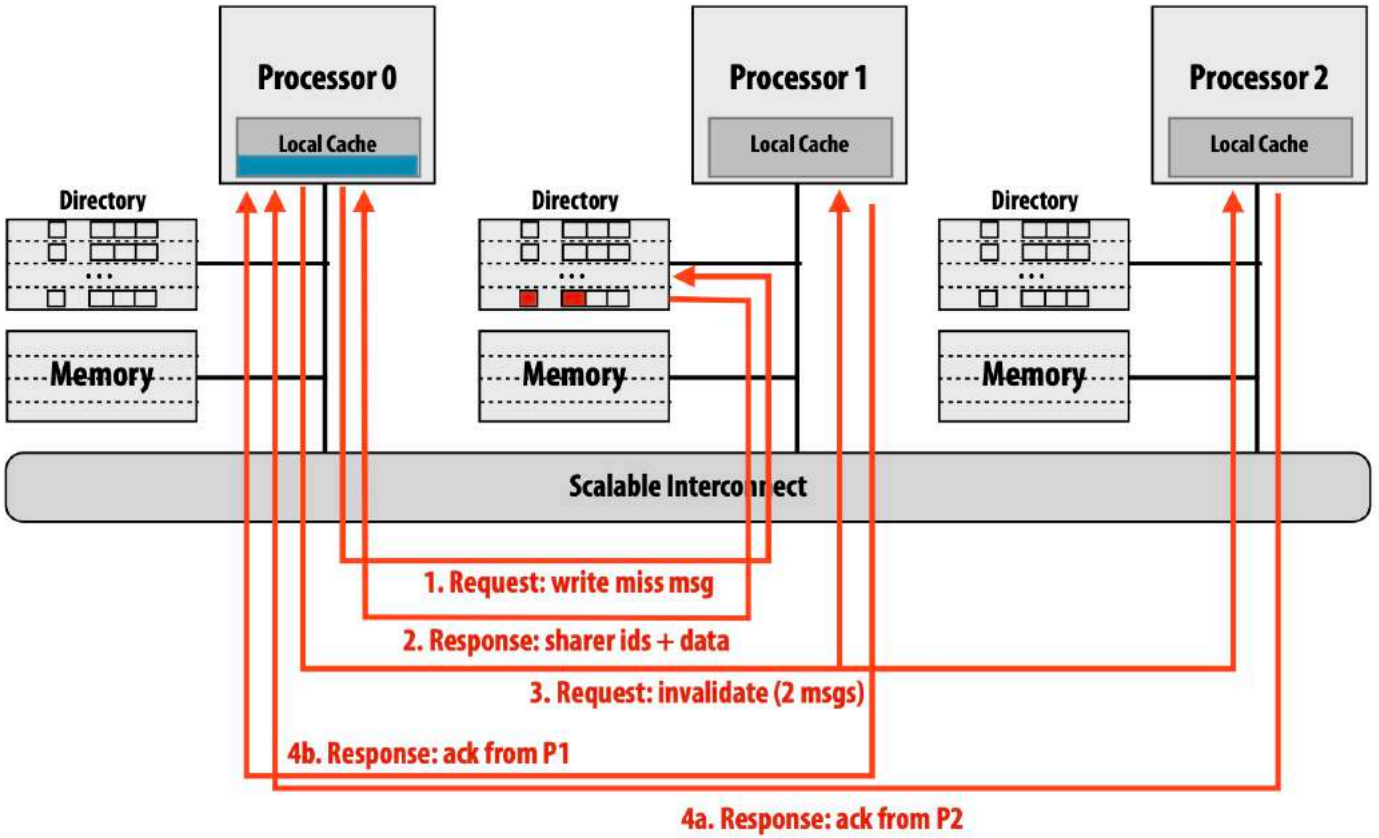


4. Owner responds to requesting node, changes state in cache to SHARED (read only)

5. Owner also responds to home node, home clears dirty, updates presence bits, updates memory

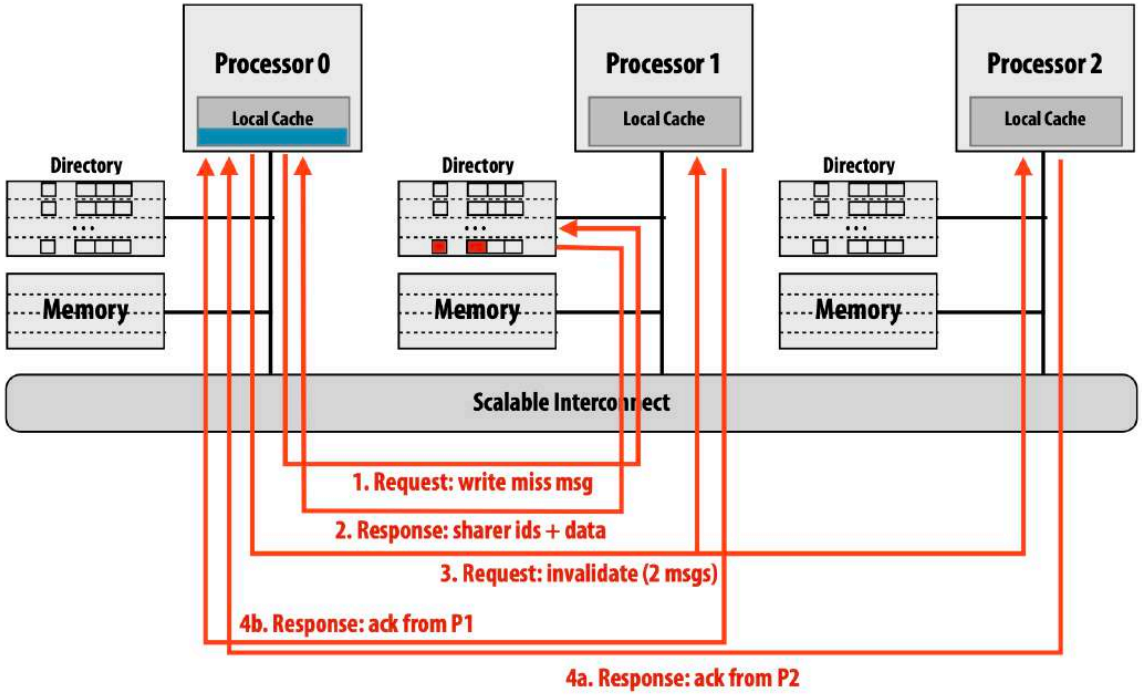
Example: write miss

- Write to memory by processor 0
  - Line is clean, but resident in P1's and P2's caches



Example: write miss (cont.)

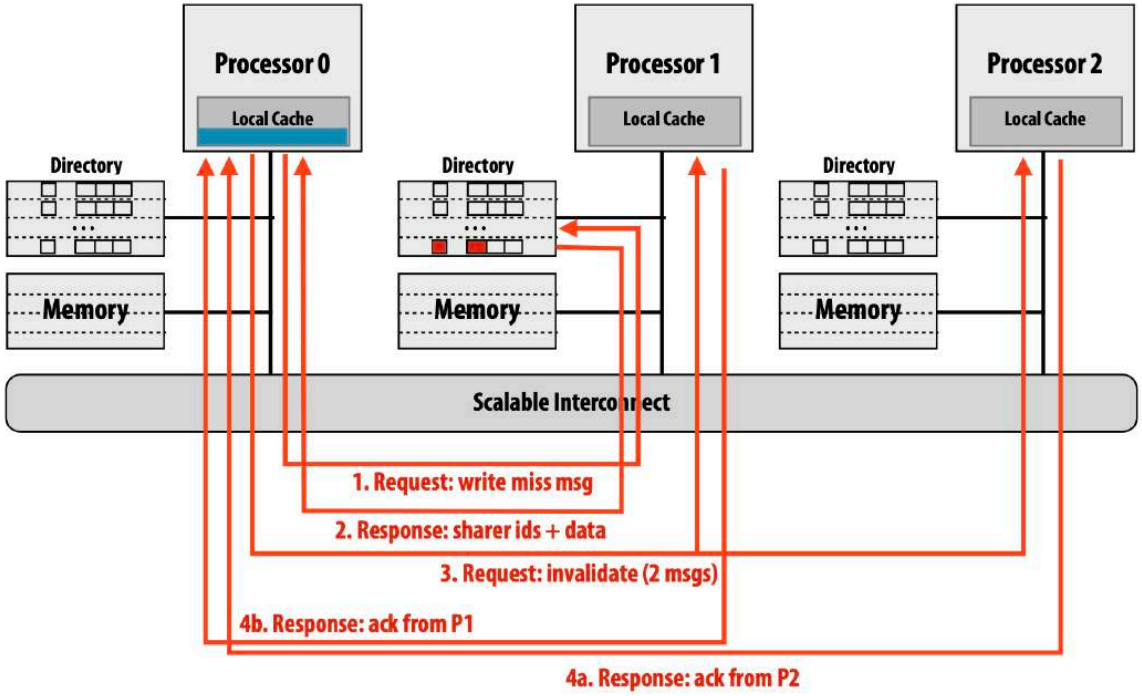
Procedure



1. If dirty bit is ON, data muse be modified on another processor
2. Home node responds with id of node containing this data(sharer) and data

Example: write miss (cont.)

Procedure



3. Requesting sharer to invalidate corresponding data

4. Get response from P1 and P2

After receiving both invalidation acks, P0 can write

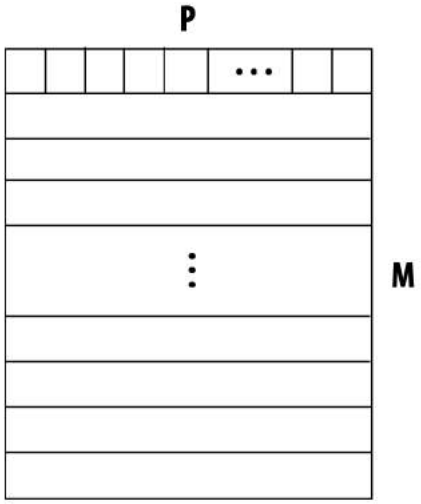
Pros of Directory Protocol

- On reads, directory tells requesting node exactly where to get the line from
  - Either from home node (if the line is clean)
  - Or from the owning node (if the line is dirty)
  - Either way, retrieving data involves only point-to-point communication

- On writes, the advantages of directories depends on the number of sharers
  - In the limit, if all caches are sharing data, all caches must be communicated with (just like broadcast in a snooping protocol)

Cons of Directory Protocol

- Full bit vector directory representation
  - One presence bit per node



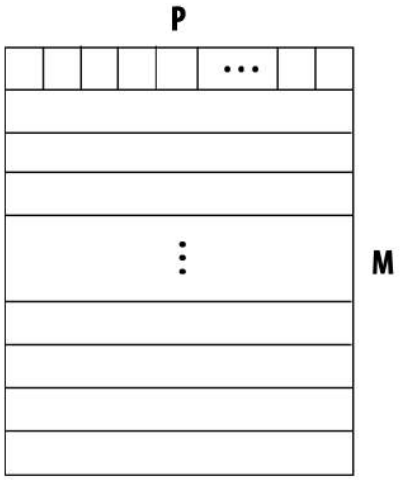
- Storage proportional to P * M
  - P = number of nodes(e.g., processors)
  - M = number of lines in memory
- Storage overhead rises with P
  - Assume 64 byte cache line size (512 bits)
  - 64 nodes (P=64) -> 12% overhead
  - 256 nodes (P=256) -> 50% overhead
  - 1024 nodes (P=1024) -> 200% overhead

Reducing Storage Overhead

- Optimizations on full-bit vector scheme
  - Increase cache line size (reduce M term)
  - Group multiple processors into a single directory "node" (reduce P term)
    - Need only one directory bit per node, not one bit per processor
    - Hierarchical: could use snooping protocol to maintain coherence among processors in a node, directory across nodes

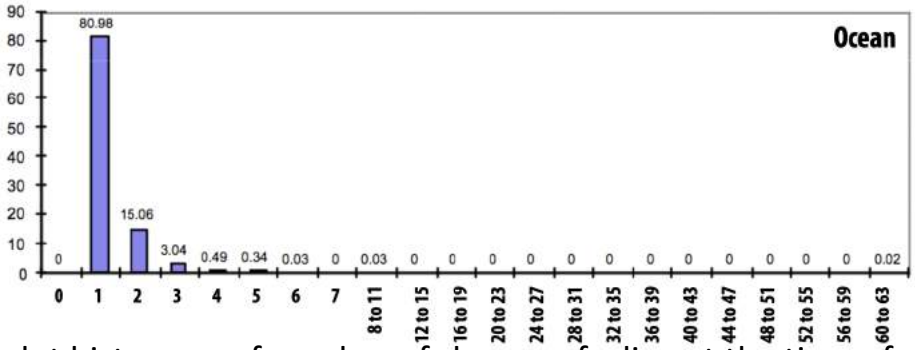
- Two alternative schemes
  - **Limited pointer** schemes (reduce P)
  - **Sparse directories** (reduce M)

- Reduce number of messages transferred



Limited Pointer Schemes

- Since data is expected to only be in a few caches at once, storage for a limited number of pointers per directory entry should be sufficient (only need a list of the nodes holding a valid copy of the line)
  - Example:
    - In a 1024 processor system
    - Full bit vector scheme needs 1024 bits per line
    - Using limited pointer scheme, 1024 bits can store approximately 100 pointers to nodes holding the line (log(1024) = 10 bits per pointer)
    - In practice, we can get by with far less than this.(20-80 principle)



Graphs plot histogram of number of sharers of a line at the time of a write

http://15418.courses.cs.cmu.edu/spring2017/lecture/directorycoherence

Managing Overflow in Limited Pointer Schemes

If too many pointers (sharers) are required…

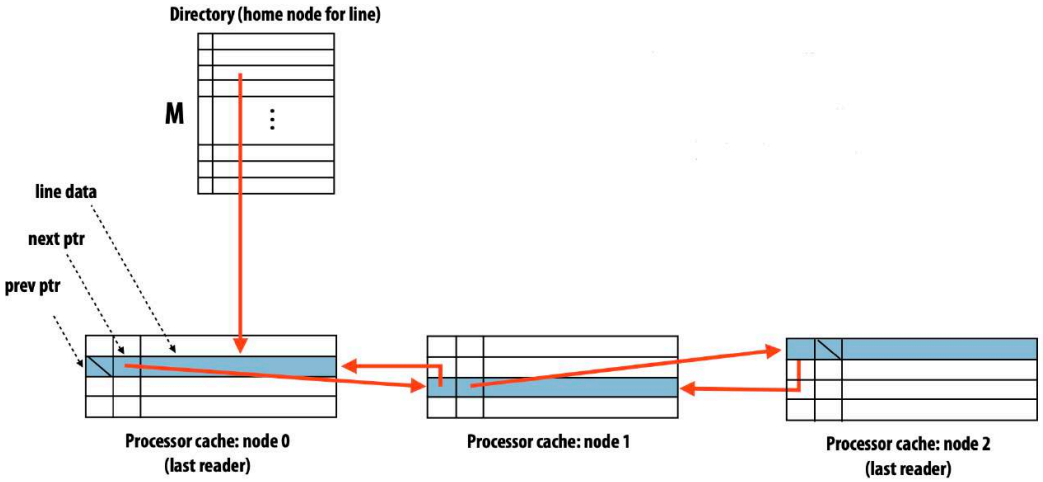
- Fallback to broadcast (if broadcast mechanism exists)
  - When more than max number of sharers, revert to broadcast
- If no broadcast mechanism present on machine
  - Don't allow more than a max number of sharers
  - On overflow, newest sharer replaces an existing one (must invalidate line in the old sharer's cache)
- Coarse vector fallback
  - Revert to bit vector representation
  - Each bit corresponds to K nodes
  - On write, invalidate all nodes a bit corresponds to

Summary of Limited Pointer Schemes

- Limited pointer schemes reduce directory storage overhead caused by large P
  - By adopting a compact representation of a list of shares
- But do we really need to maintain storage for a list for each cache-line chunk of data in memory?
- Key observation: the majority of memory is NOT resident in cache. And to carry out coherence protocol the system only needs sharing information for lines that are currently in cache
  - Most directory entries are empty most of the time
  - 1 MB cache, 1 GB memory per node -> 99.9% of directory entries are idle

Sparse Directories

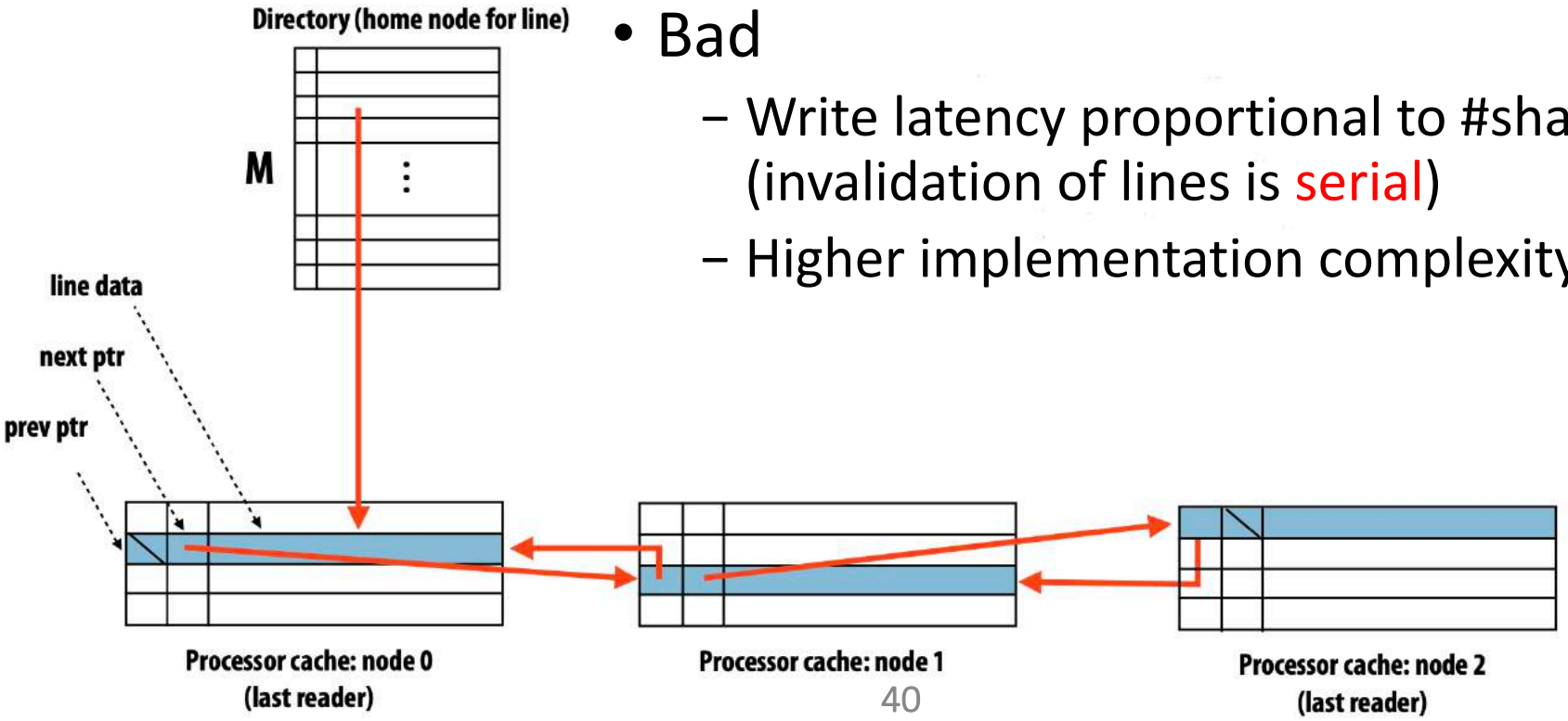
- Directory at home node maintains pointer to only one node caching line (not a list of sharers)
- Pointer to next node in list is stored as extra information in the cache line (like the line's tag, dirty bits, etc.)
- On read miss: add requesting node to head of list
- On write miss: propagate invalidations along list
- On evict: need to patch up list (linked list removal)



http://15418.courses.cs.cmu.edu/spring2017/lecture/directorycoherence

Scaling Properties of Sparse Directories

- Good
  - Low memory storage overhead (one pointer to list head per line)
  - Additional directory storage is proportional to cache size (the list stored in SRAM)
  - Traffic on write is still proportional to number of sharers

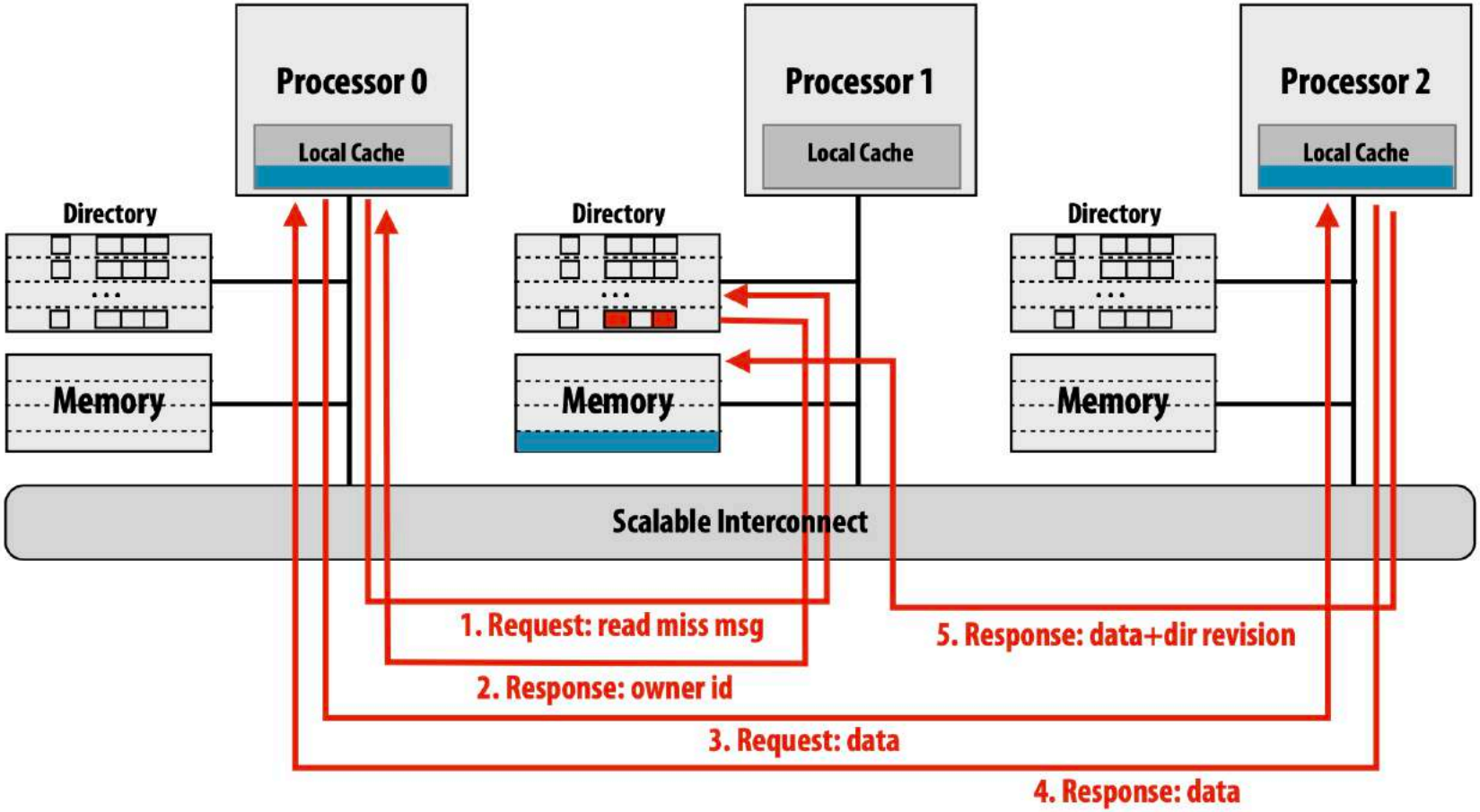


- Bad
  - Write latency proportional to #sharers (invalidation of lines is serial)
  - Higher implementation complexity

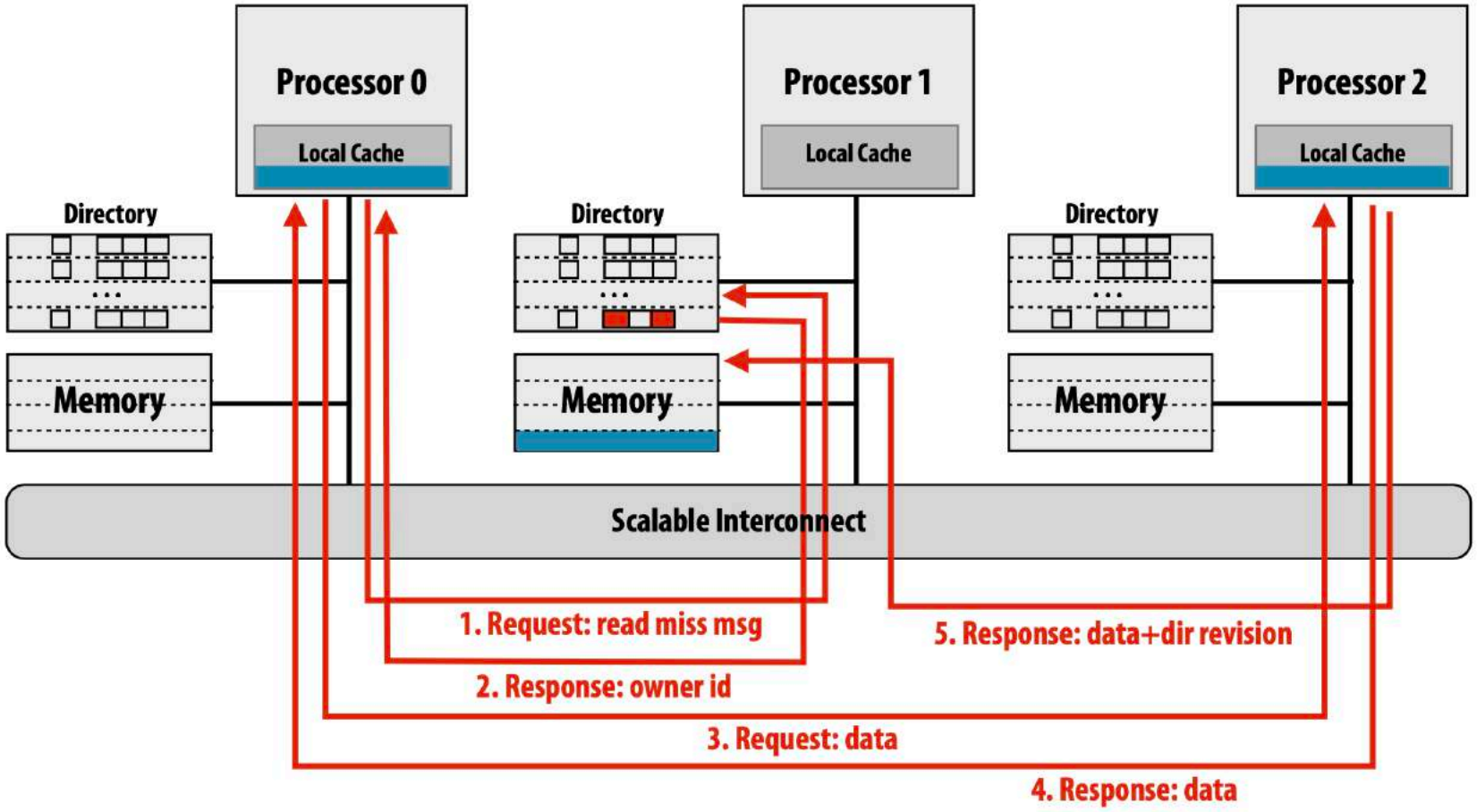
http://15418.courses.cs.cmu.edu/spring2017/lecture/directorycoherence

Reduce number of msg. sent

Read from main memory by P0 of the blue line: line is dirty (contained in P2's cache)



Reduce number of msg. sent (cont.)

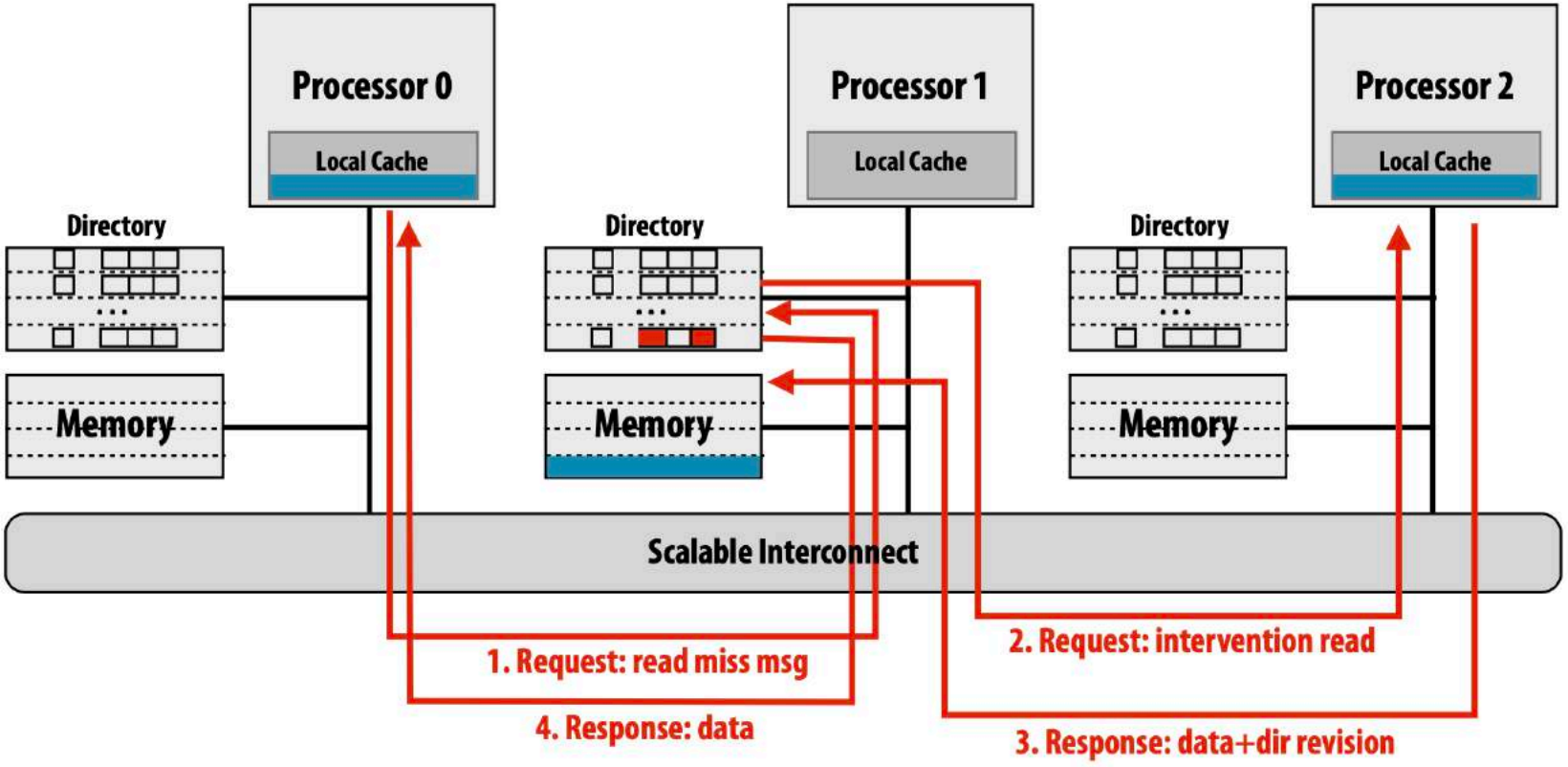


Five network transactions in total

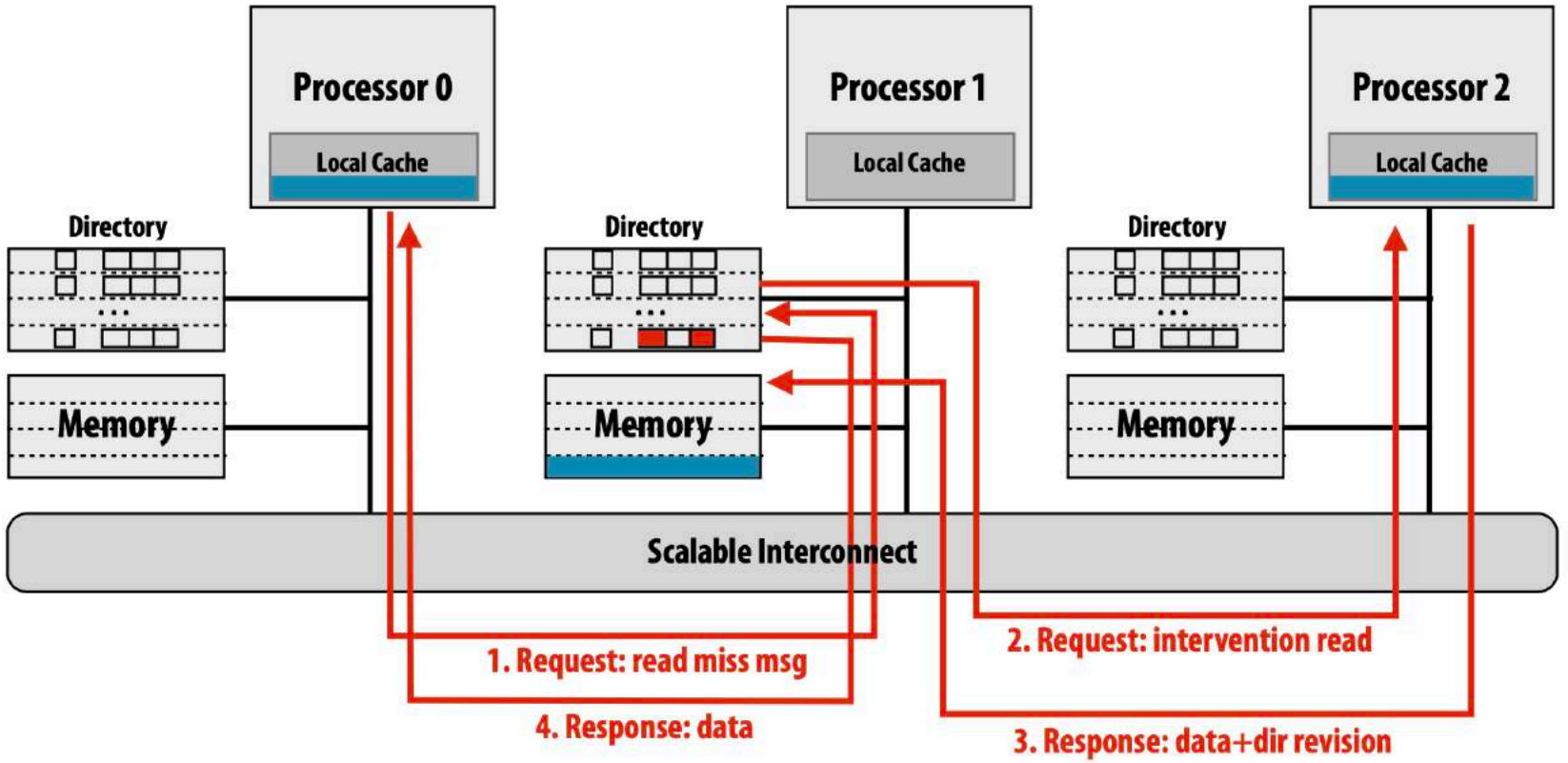
Four of them are sequential (transaction 4&5 can parallel)

Intervention Forwarding

Read from main memory by P0 of the blue line: line is dirty (contained in P2's cache)

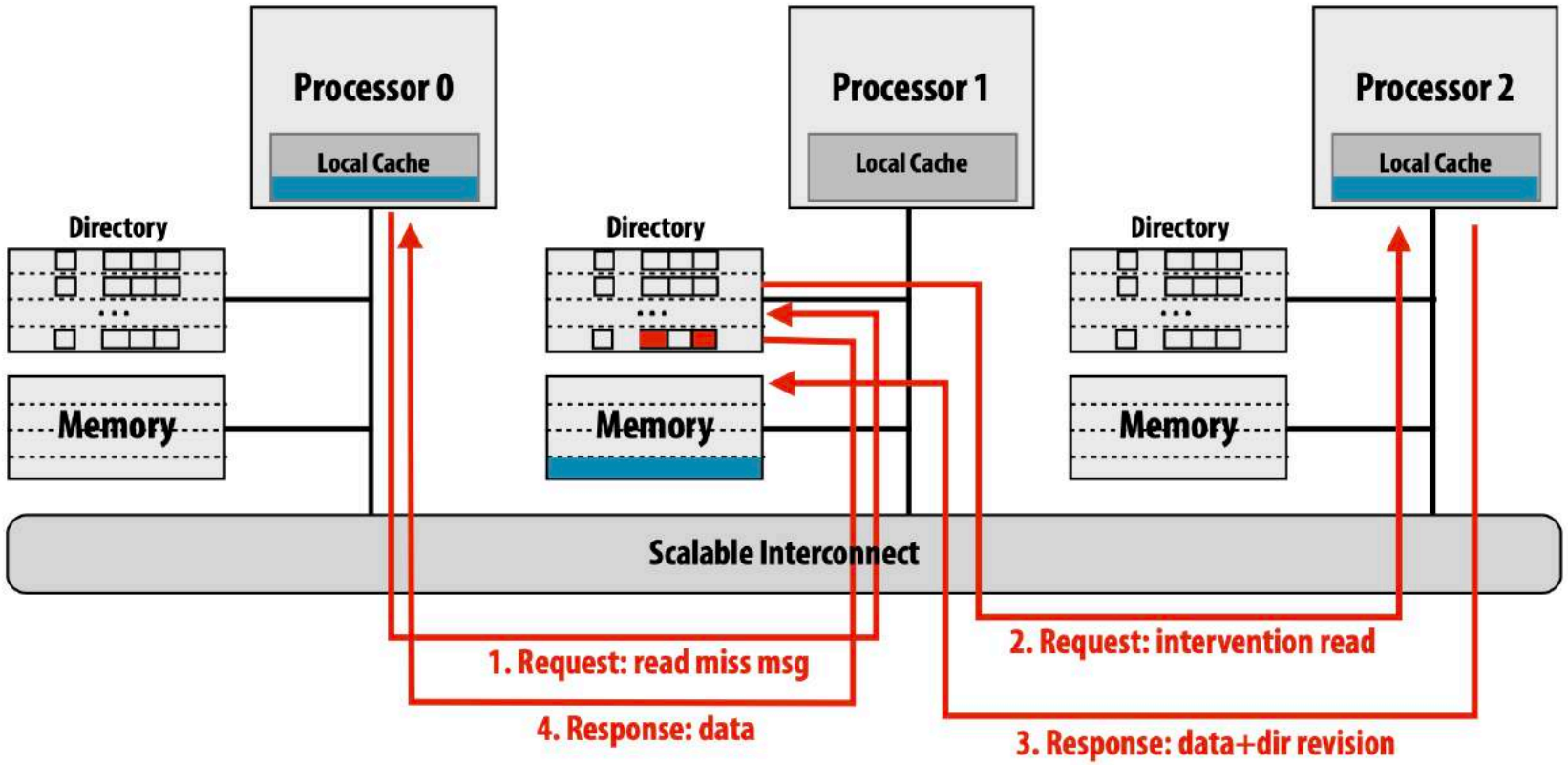


Intervention Forwarding (cont.)



1. Requests to read miss message on home node (P1)
2. Home node requests data from owner node (P2)

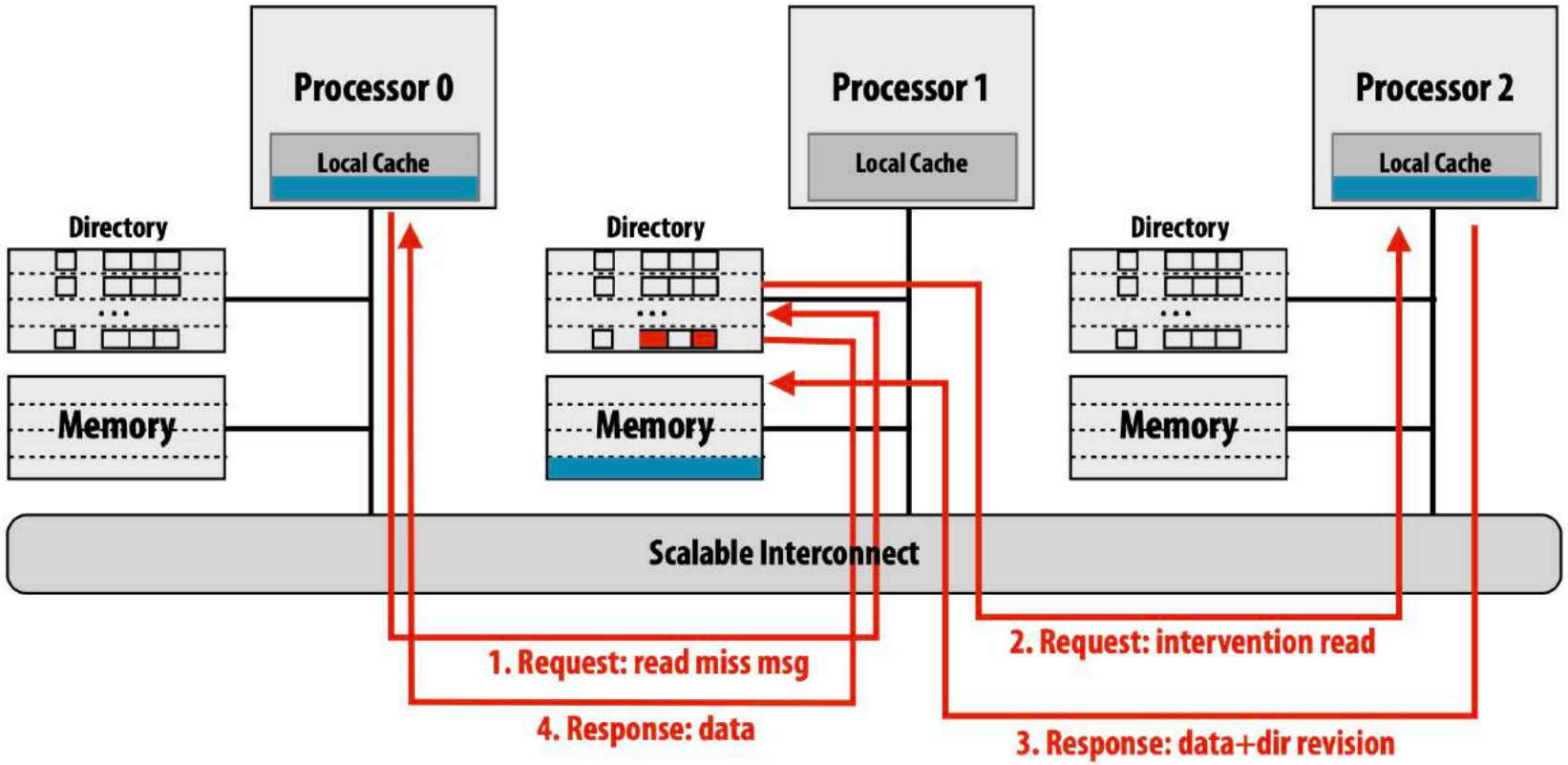
Intervention Forwarding (cont.)



3. Owning node response

4. Home node updates directory, responds to requesting node with requested data

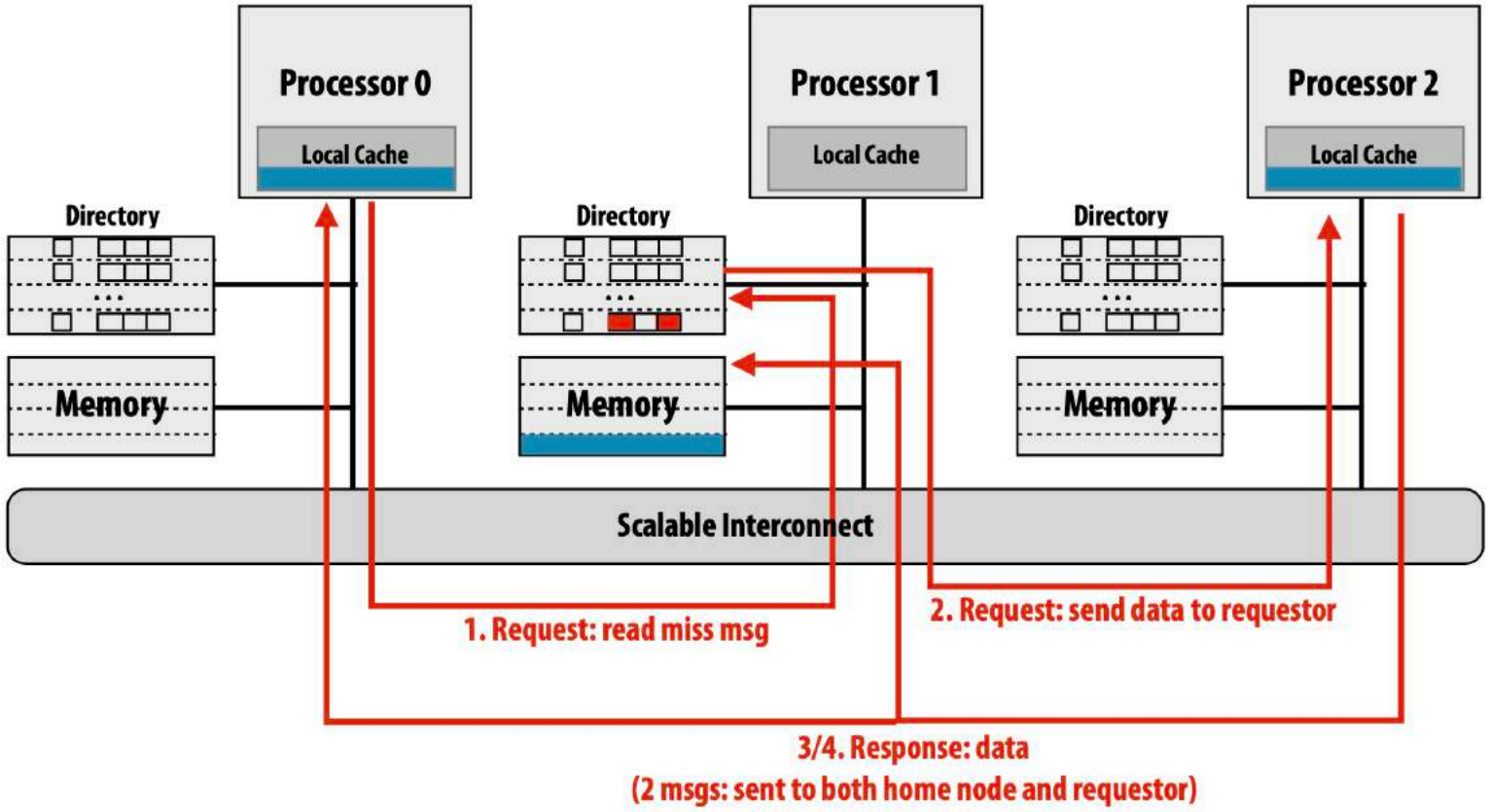
Intervention Forwarding (cont.)



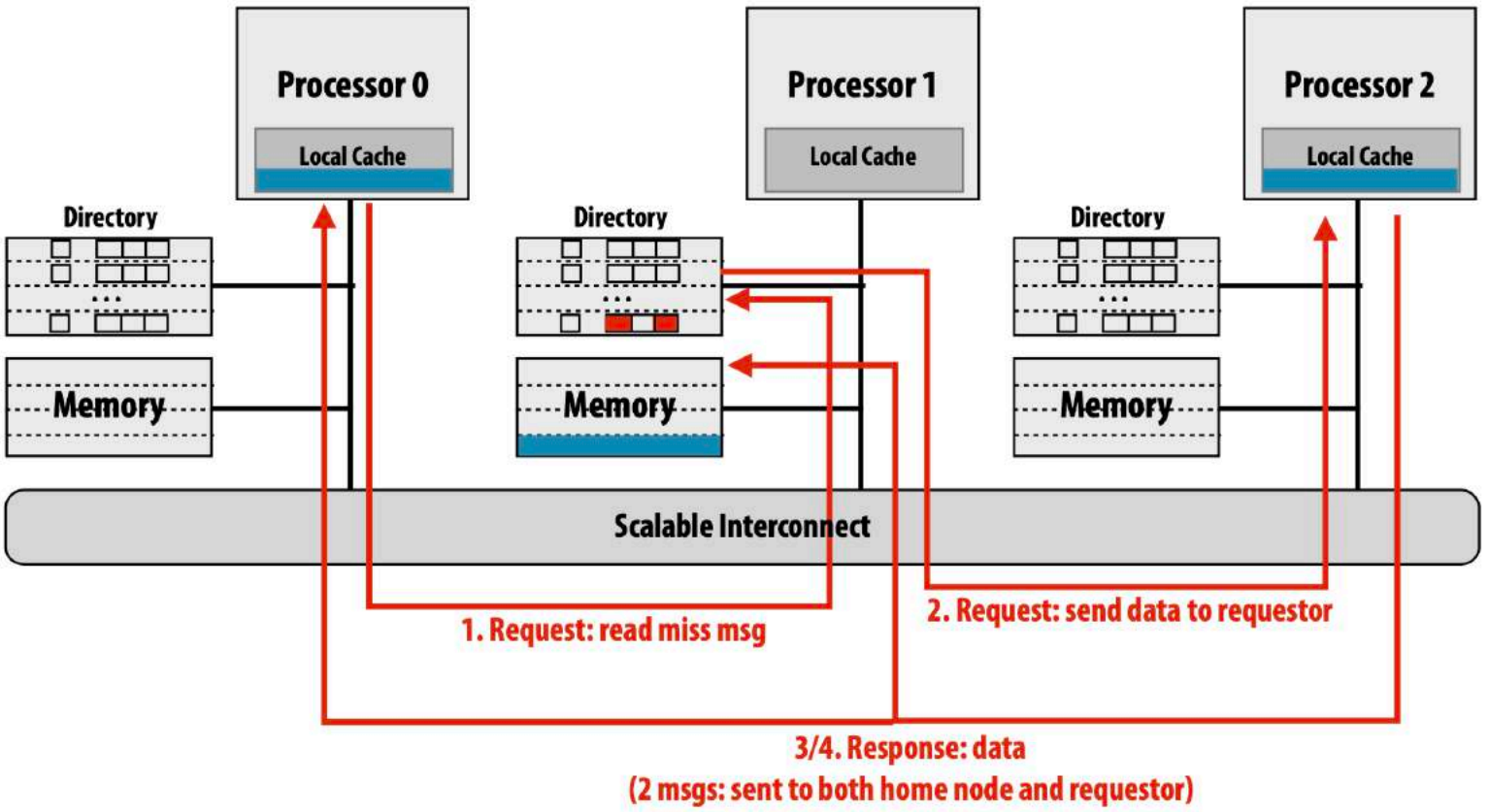
Total 4 transactions are needed.
But all of them are sequential, can they be parallel?

Request Forwarding

Read from main memory by P0 of the blue line: line is dirty (contained in P2's cache)

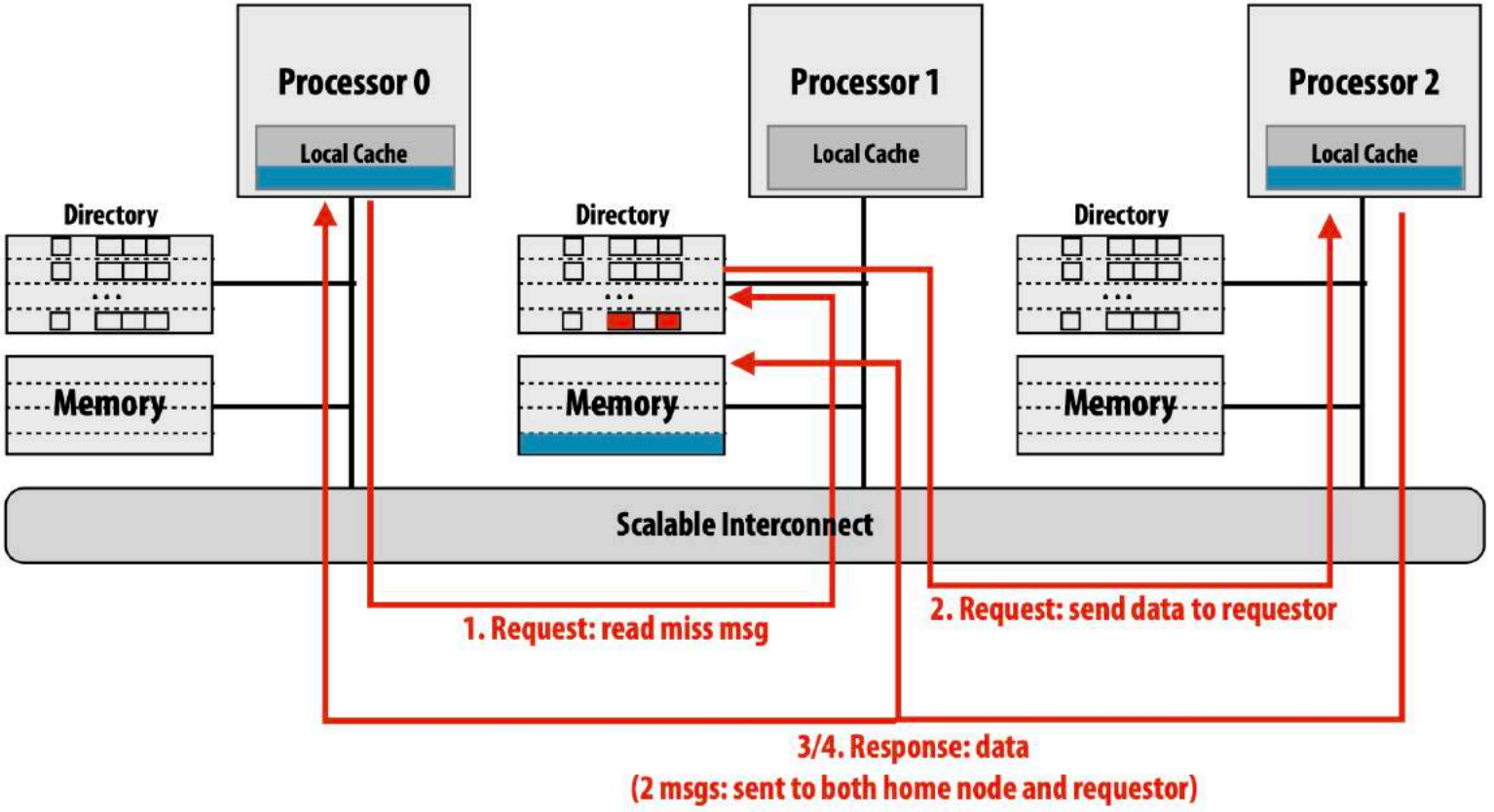


Request Forwarding (cont.)



1. Requests to read miss message on home node (P1)
2. Home node send target data to owner

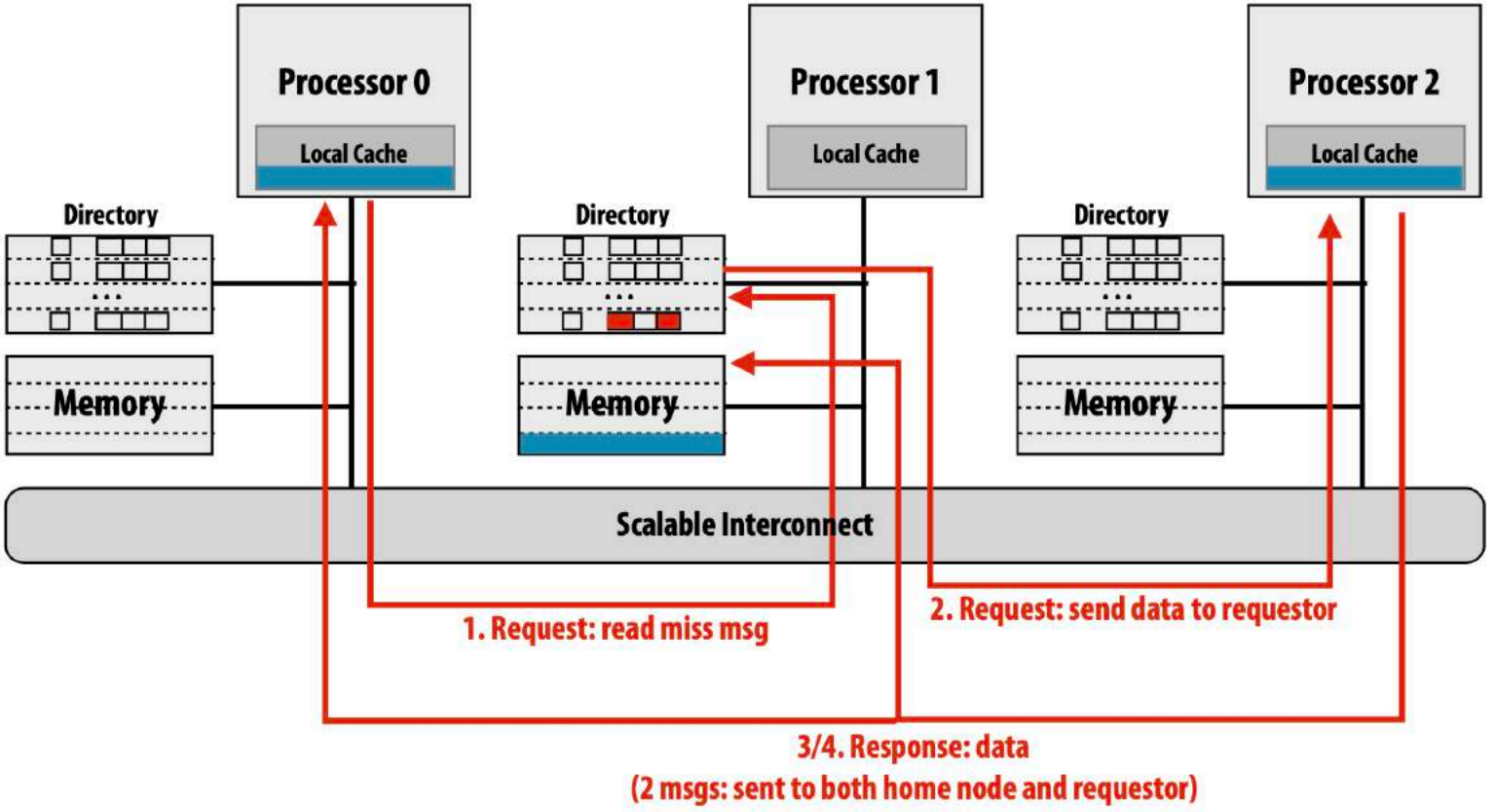
Request Forwarding (cont.)



3. Owning node responses data to the home node

4. Owning node responses data to the requesting node

Request Forwarding (cont.)



Only 3 transactions are in serial

Transaction 3 & 4 can be parallel

Summary of Directory-base Coherence

- Primary observation: broadcast doesn't scale, but we don't need to broadcast to ensure coherence because often the number of caches containing a copy of a line is small
- Instead of snooping, just store the list of sharers in a directory and check the list when necessary
- One challenge
  - Use hierarchies of processors or larger cache size
  - Limited pointer schemes: exploit fact that most processors not sharing line
  - Sparse directory schemes: exploit fact that most lines not in line
- Another challenge
  - Reduce messages sent (traffic) and parallelize transactions (latency)