Advanced Computer Architecture

Chapter 10 – Multicore, parallel, and cache coherency

Part 4:

Scalable shared-memory – directory-based cache coherency protocols



COSMOS: UK National Cosmology Supercomputer. SGI Altix UV 2000 with 1536 cores and 12.2TB RAM, globally accessible (delivered 2012)

November 2026

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd, 4th and 5th eds)*, and on the lecture slides of David Patterson, John Kubiatowicz and Yujia Jin at Berkeley

Hennessy and Patterson 6th ed Section 5.4 pp404

Overview

- Why add another processor?
- What are *large* parallel machines used for?
- How do you program a parallel machine?
- How should they be connected – I/O, memory bus, L2 cache, registers?
- Cache coherency – the problem
- Coherency – what are the rules?
- Coherency using broadcast – update/invalidate
- Coherency using multicast – SMP vs ccNUMA
- Distributed directories, home nodes, ownership; the ccNUMA design space
- Beyond ccNUMA; COMA and Simple COMA
- Hybrids and clusters

What you should get from this

Parallel systems architecture is a vast topic, and we can only scratch the surface. The critical things I hope you will learn from this very brief introduction are:

- Why power considerations motivate multicore
- Why is shared-memory parallel programming attractive?
  - How is dynamic load-balancing implemented?
  - Why is distributed-memory parallel programming harder but more likely to yield robust performance?
- What is the cache coherency problem
  - There is a design-space of “snooping” protocols based on broadcasting invalidations and requests
- How are atomic operations and locks implemented?
  - Eg load-linked, store conditional
- What is sequential consistency?
  - Why might you prefer a memory model with weaker consistency?
- For larger systems, some kind of “directory” is needed to avoid/reduce the broadcasting

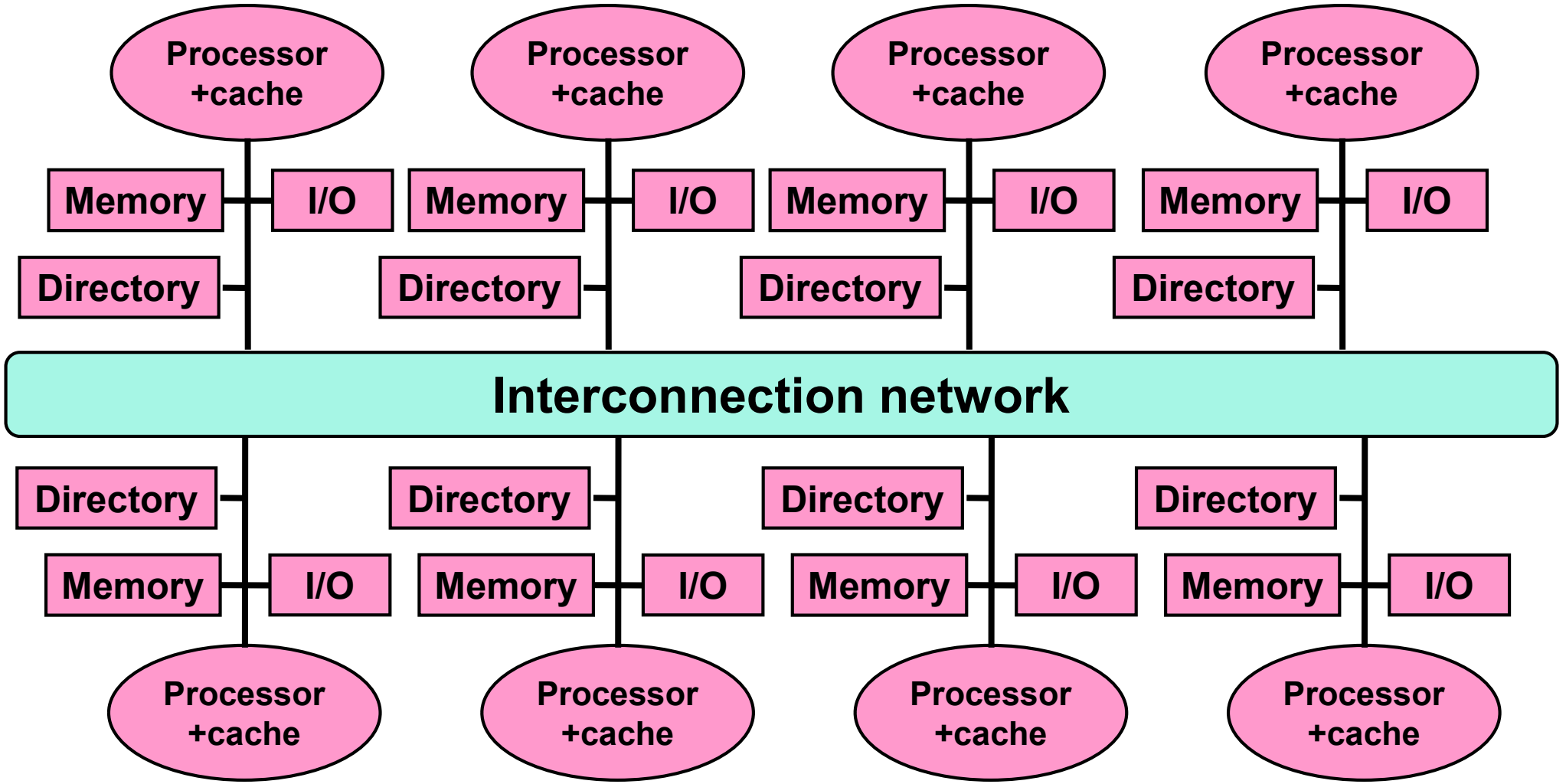
Large-Scale Shared-Memory Multiprocessors: Directory-based cache coherency protocols

- Snooping cache coherency protocols rely on a bus:
  - For broadcasting invalidations and read requests
  - To establish global ordering on events
- The bus inevitably becomes a bottleneck when many processors are used
  - So snooping does not work
  - So we need to use a more general interconnection network
- DRAM memory is also distributed (*Non-Uniform Memory Architecture, NUMA*)
  - Each node allocates space from local DRAM
  - Copies of remote data are made in cache
- Major design issues:
  - How to find and represent the "directory" of each line?
  - How to find a copy of a line?

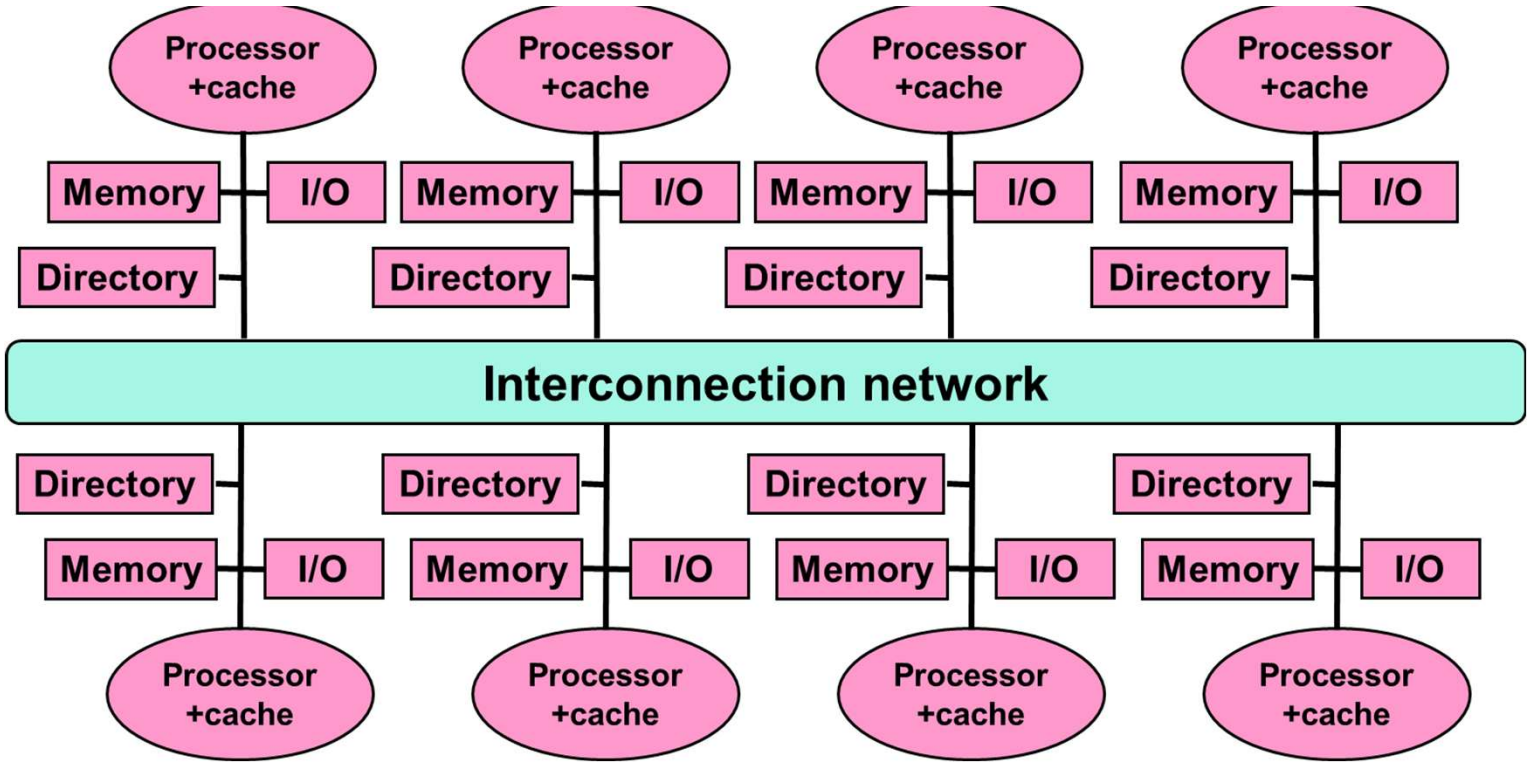
Larger shared-memory multiprocessors

- Separate Memory per Processor
- Local or Remote access via memory controller
- Directory per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/one location)
  - MINUS: In memory => directory is *f*(memory size) vs. *f*(cache size)
- How do we prevent the directory being a bottleneck? distribute directory entries with memory, each keeping track of which cores have copies of their blocks

Distributed Directory MPs



Larger shared-memory multiprocessors

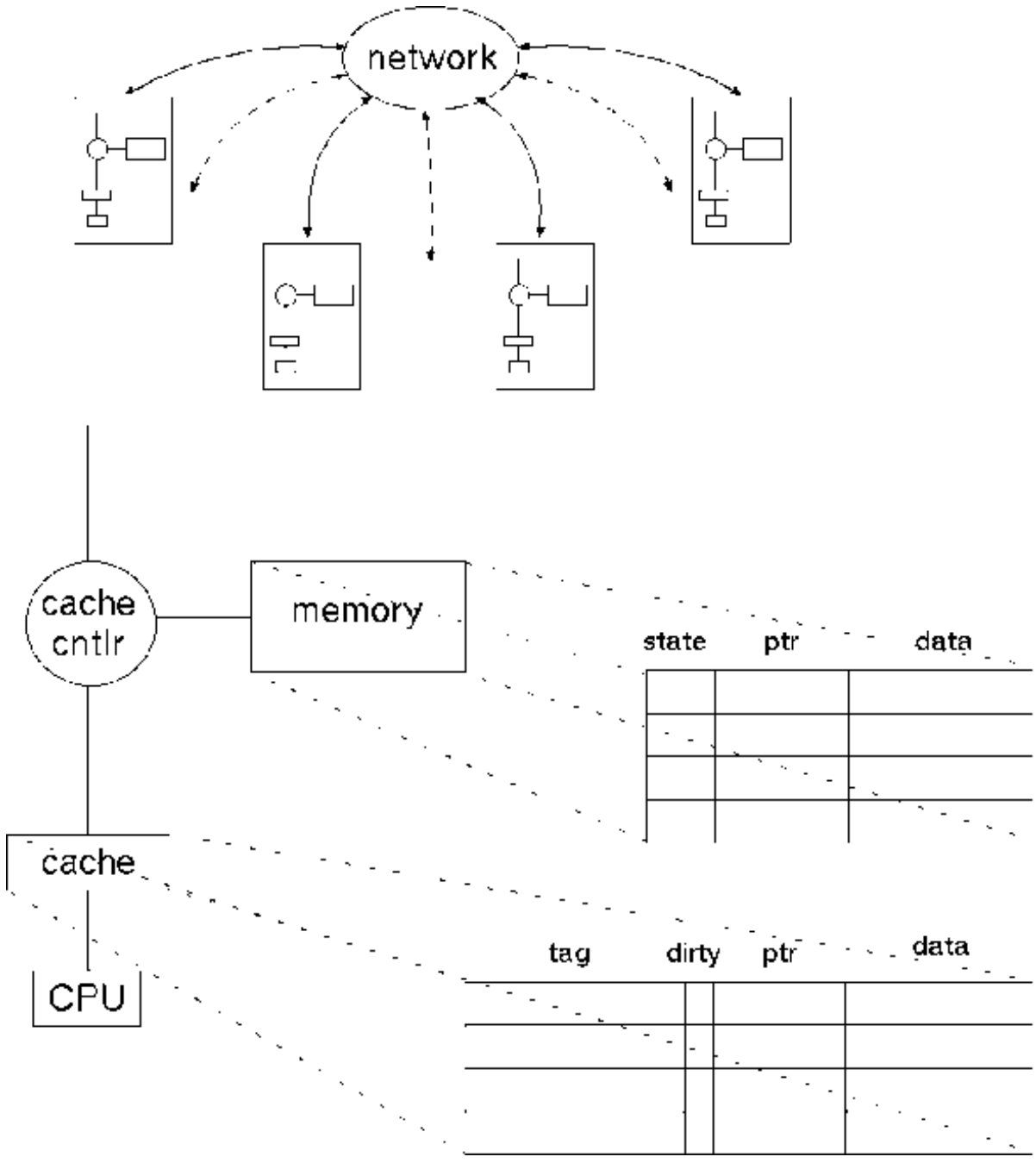


- Separate Memory per Processor, Local or Remote access via memory controller
- Directory per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/one location)
  - MINUS: In memory => directory is f(memory size) vs. f(cache size)
- How do we prevent the directory being a bottleneck?
  Distribute directory entries with memory, each keeping track of which cores have copies of their blocks

Summary

- Caches contain all information on state of cached memory blocks
- Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping => uniform memory access)
- Directory has extra data structure to keep track of state of all cache blocks
- Distributing directory => scalable shared address multiprocessor
  => Cache coherent, Non uniform memory access

Case study: Sun's S3MP



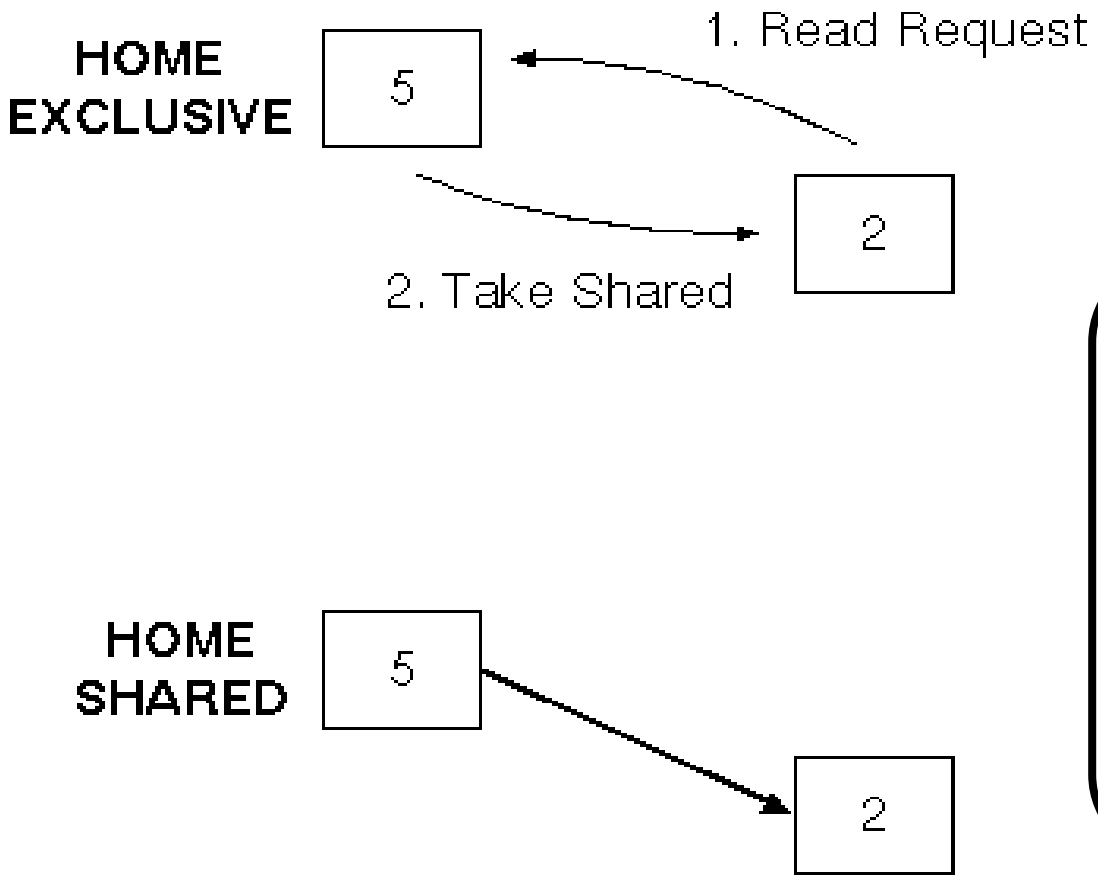
Protocol Basics

- S3.MP uses distributed singly-linked sharing lists, with static homes
- Each line has a "home" node, which stores the root of the directory
- Requests are sent to the home node
- Home either has a copy of the line, or knows a node which does

A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Browne, G. Aybay, and D. Lee. 1993. The S3.mp architecture: a local area multiprocessor. In Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures (SPAA '93). ACM, New York, NY, USA, 140-141. DOI=http://dx.doi.org/10.1145/165231.165249

S3MP: Read Requests

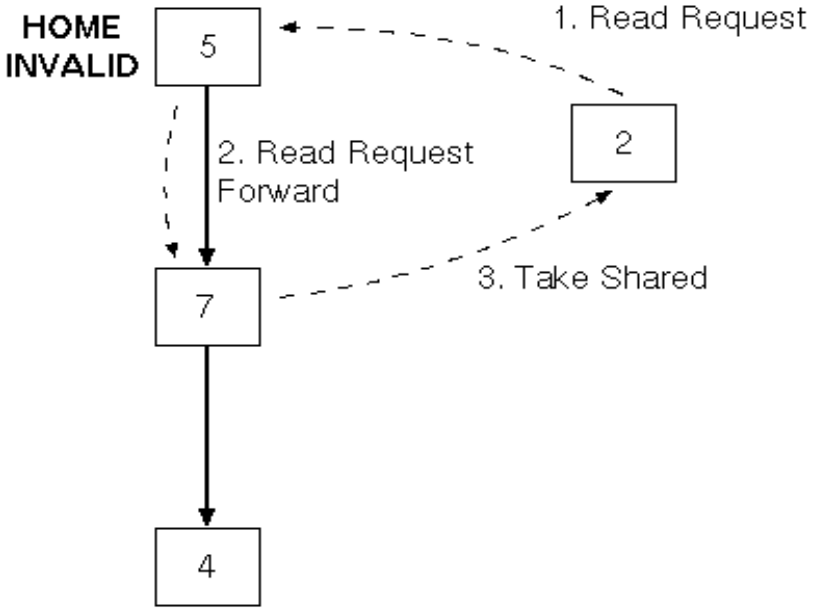
- Simple case: initially only the home has the data:



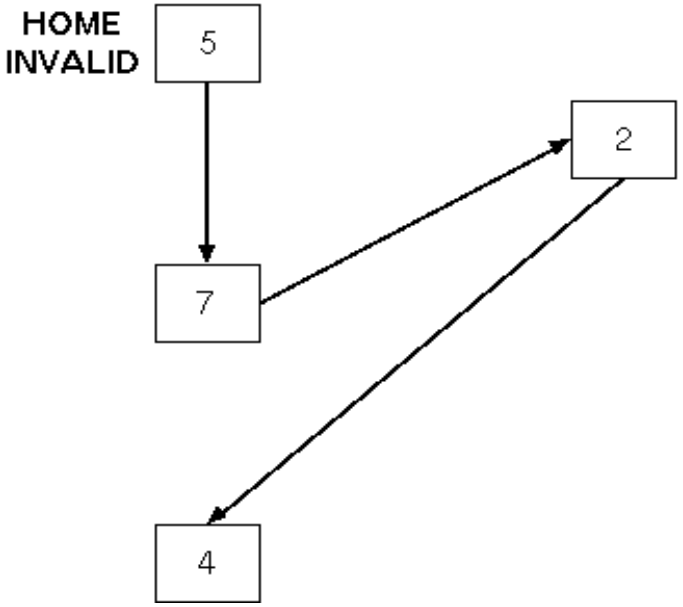
Curved arrows show messages, bold straight arrows show pointers

- **Home replies with the data, creating a sharing chain containing just the reader**

S3MP: Read Requests - remote

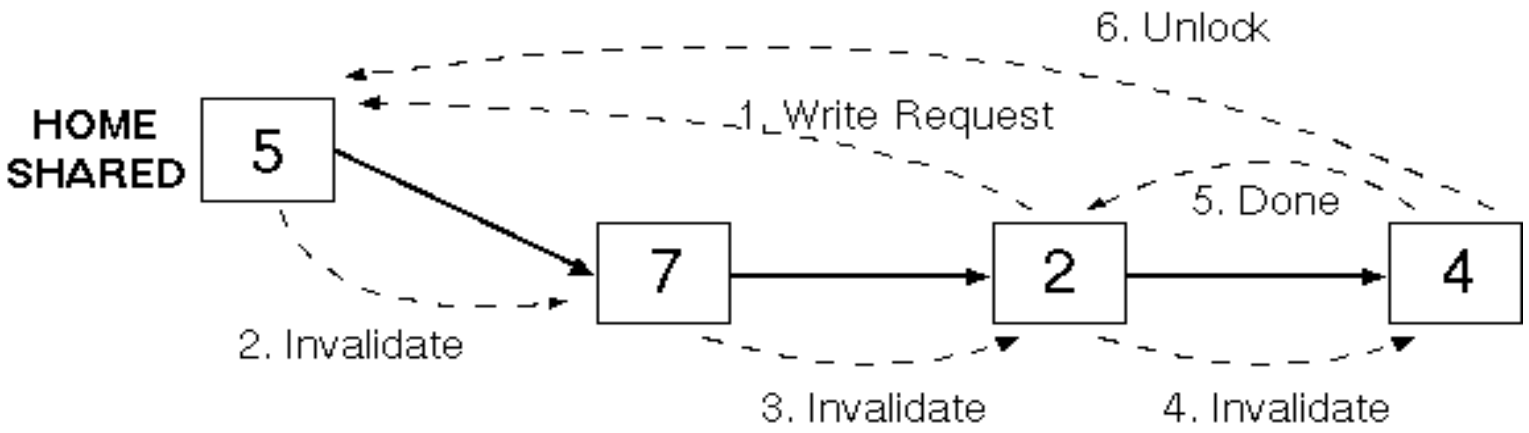


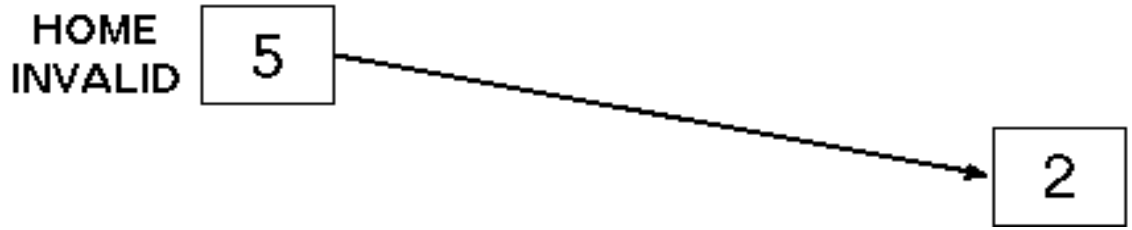
- More interesting case: some other processor has the data



- Home passes request to first processor in chain, adding requester into the sharing list

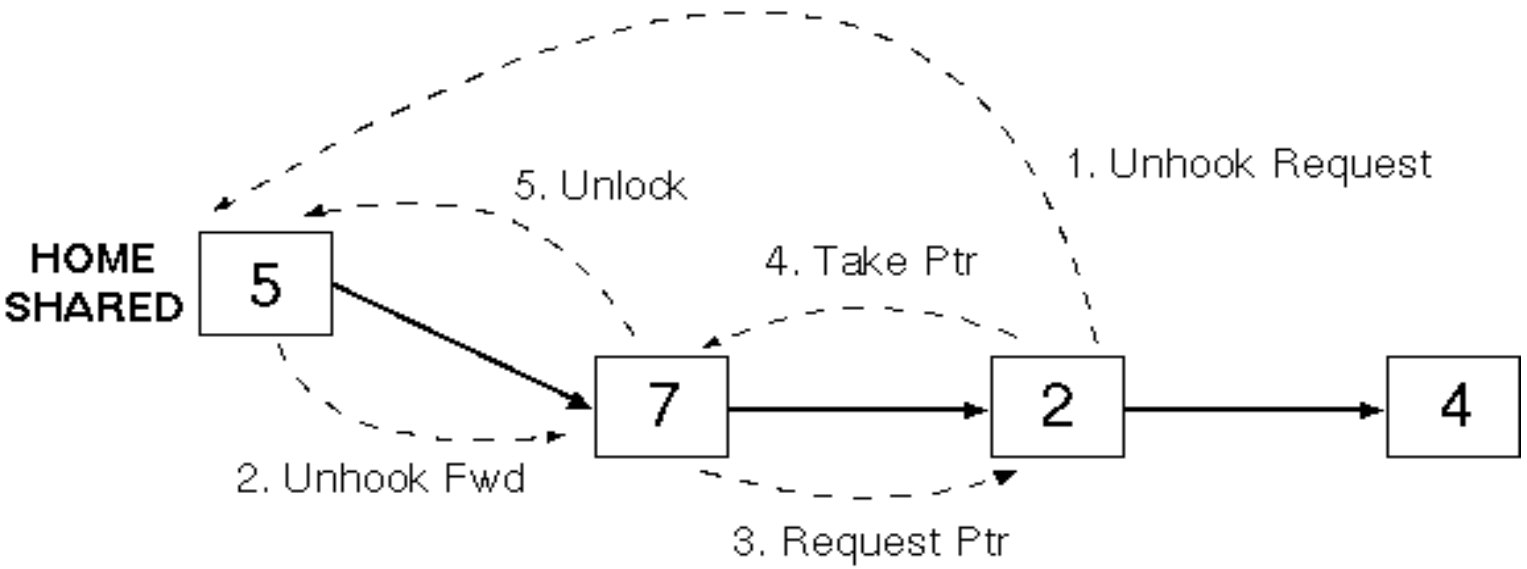
S3MP - Writes

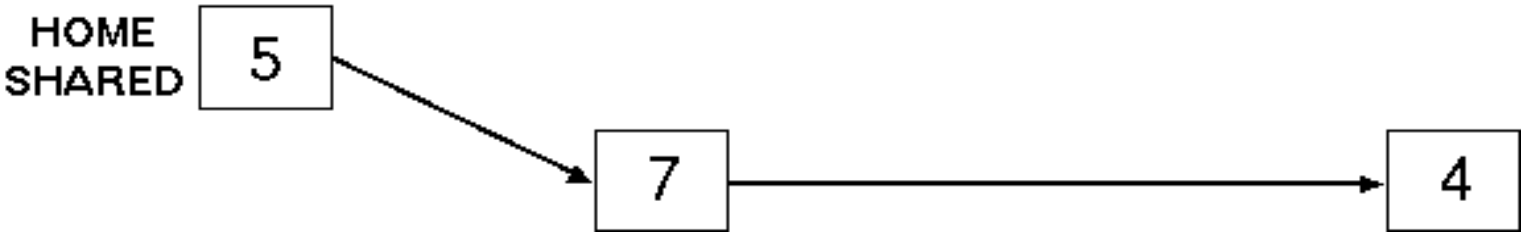




- If the line is exclusive (i.e. dirty bit is set) no message is required
- Else send a write-request to the home
  - Home sends an invalidation message down the chain
  - Each copy is invalidated (other than that of the requester)
  - Final node in chain acknowledges the requester and the home
- Chain is locked for the duration of the invalidation

S3MP - Replacements





- When a read or write requires a line to be copied into the cache from another node, an existing line may need to be replaced
- Must remove it from the sharing list
- Must not lose last copy of the line

Finding your data

- How does a CPU find a valid copy of a specified address's data?
  1. Translate virtual address to physical
  2. Physical address includes bits which identify "home" node
  3. Home node is where DRAM for this address resides
  4. But current valid copy may not be there – may be in another CPU's cache
  5. Home node holds pointer to sharing chain, so always knows where valid copy can be found

ccNUMA summary

- S3MP's cache coherency protocol implements strong consistency
  - Many recent designs implement a weaker consistency model…
- S3MP uses a singly-linked sharing chain
  - Widely-shared data – long chains – long invalidations, nasty replacements
  - "Widely shared data is rare"
- In real life:
  - IEEE Scalable Coherent Interconnect (SCI): doubly-linked sharing list
  - SGI Origin 2000: 64-bit vector sharing list
    - Origin 2000 systems were deliverered with 256 CPUs
  - Sun E10000: hybrid multiple buses for invalidations, separate switched network for data transfers
  - Multi-node and multi-socket SMP clusters –
    - Next slide!

Beyond ccNUMA

- COMA: cache-only memory architecture
  - Data migrates into local DRAM of CPUs where it is being used
  - Handles very large working sets cleanly
  - Replacement from DRAM is messy: have to make sure *someone* still has a copy
  - Scope for interesting OS/architecture hybrid
  - System slows down if total memory requirement approaches RAM available, since space for replication is reduced
- Examples: DDM, KSR-1/2

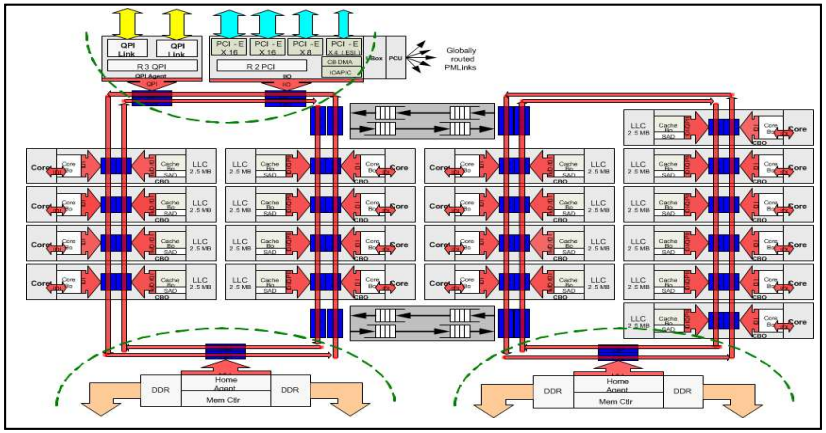
Clustered architectures

- **Idea**: systems linked by network connected via I/O bus
  - Eg PC cluster with Myrinet or Quadrics interconnection network
  - Eg Quadrics+PCI-Express: 900MB/s (500MB/s for 2KB messages), 3us latency
- **Idea**: systems linked by network connected via memory interface
  - Eg Cray Seastar
- **Idea**: CPU/memory packages linked by network connecting main memory units
  - Eg SGI Origin, nVidia Tesla?
- **Idea**: CPUs share L2/L3 cache
  - Eg IBM Power4,5,6, Intel Core2, Nehalem, AMD Opteron, Phenom
- **Idea**: CPUs share L1 cache
- **Idea**: CPUs share registers, functional units
  - IBM Power5,6, PentiumIV(hyperthreading), Intel Atom, Alpha 21464/EV8, Cray/Tera MTA (multithreaded architecture), nVidia Tesla

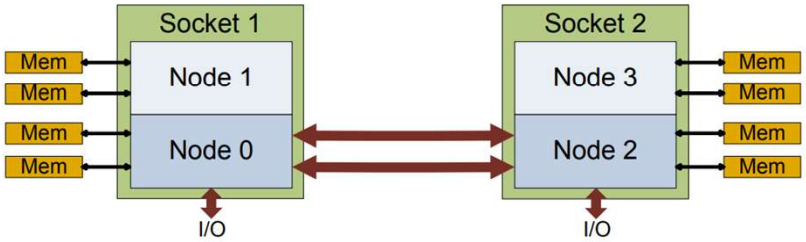
- **Cunning Idea: do (almost) all the above at the same time**
- **Eg IBM SP/Power6: 2 CPUs/chip, 2-way SMT, "semi-shared" 4M/core L2, shared 32MB L3, multichip module packages/links 4 chips/node, with L3 and DRAM for each CPU on same board, with high-speed (ccNUMA) link to other nodes – assembled into a cluster of 8-way nodes linked by proprietary network**

http://www.realworldtech.com/page.cfm?ArticleID=RWT101606194731

ccNUMA in real life…

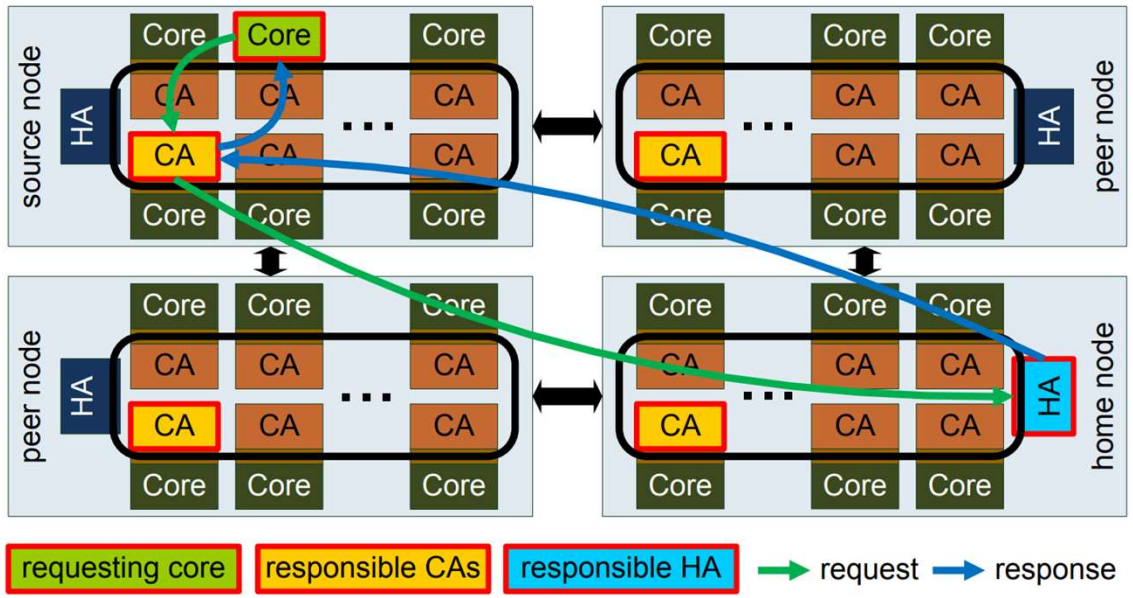


18-core chip with two rings



Two-socket configuration

(“cluster-on-die” mode)



HA: “home agent”

CA: “cache agent”

- Each cache line has 2 bits of directory indicating whether the line is held in other nodes: *remote-invalid, snoop-all (*potentially modified copy exists), or *shared* (multiple clean copies exist)
- On L2 miss, core sends request to a Cache Agent on its node (based on physical address)
- The Cache Agent checks for a local L3 hit – but if miss, passes request to Home Agent
- Invalidations and read requests are propagated to other nodes accordingly by the Home Agent
- Directory information for frequently-exchanged lines are cached in the Home Agent (8 bits)

- **Recall: Intel Haswell e5 2600 v3**
- **A complex hybrid coherency scheme**

Daniel Molka, Daniel Hackenberg, Robert Schone, and Wolfgang E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. ICPP2015 (2015_ICPP_authors_version.pdf (tu-dresden.de))

Which cache should the cache controller control?

- L1 cache is already very busy with CPU traffic
- L2 cache also very busy…
- L3 cache doesn't always have the current value for a cache line
  - Although L1 cache is often write-through, L2 is normally write-back
  - Some data may bypass L3 (and perhaps L2) cache (eg when stream-prefetched, or if L3 is managed as a victim cache)
- Eg in IBM Power4 and others, cache controller manages L2 cache – all external invalidations/requests
- L3 cache improves access to DRAM for accesses both from CPU and from network (essentially a *victim* cache)

Summary and Conclusions

- Caches can be used to form the basis of a parallel computer supporting a single, shared address space
- Bus-based multiprocessors do not scale well due to broadcasts and the need for each cache controller to snoop all the traffic
- Larger-scale shared-memory multiprocessors require a cache *directory* to track where copies are held
  - Hierarchical and hybrid schemes can work, with snooping within a cluster of cores, and a directory scheme at the cluster level
- ccNUMA: each node has a fragment of the system's DRAM, every physical address has a unique "home" node
- COMA: each node (sometimes called a NUMA domain) has a fragment of the system's DRAM, but data is migrated between NUMA domains adaptively
- NUCA: cache is distributed, so access latency is non-uniform (and management may include dynamic/adaptive placement strategies)

Appendix: more on directory-based coherency protocols

These slides (from Hennessy and Patterson) introduce and illustrate a directory-based cache coherency protocol using a bit-vector directory representation, as found in the SGI Origin multiprocessor

Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data;
    memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (for example bit vector, 1 if processor has copy)
- Keep it simple(r):
  - Writes to non-exclusive data
    => write miss
  - Processor blocks until access completes
  - We will assume that messages are received and acted upon in order sent
    - This is not necessarily the case in general

Directory Protocol

- No bus and don't want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether exclusive or shared
- Example messages on next slide:
  P = processor number, A = address

Directory Protocol Messages

Message type	*Source*	*Destination*	*Msg Content*
Read miss	Local cache	Home directory	P, A

- *Processor P reads data at address A; make P a read sharer and arrange to send data back*

Write miss	Local cache	Home directory	P, A

- *Processor P writes data at address A; make P the exclusive owner and arrange to send data back*

Invalidate	Home directory	Remote caches	A

- *Invalidate a shared copy at address A.*

Fetch	Home directory	Remote cache	A

- *Fetch the block at address A and send it to its home directory*

Fetch/Invalidate	Home directory	Remote cache	A

- *Fetch the block at address A and send it to its home directory; invalidate the block in the cache*

Data value reply	Home directory	Local cache	Data

- *Return a data value from the home memory (read miss response)*

Data write-back	Remote cache	Home directory	A, Data

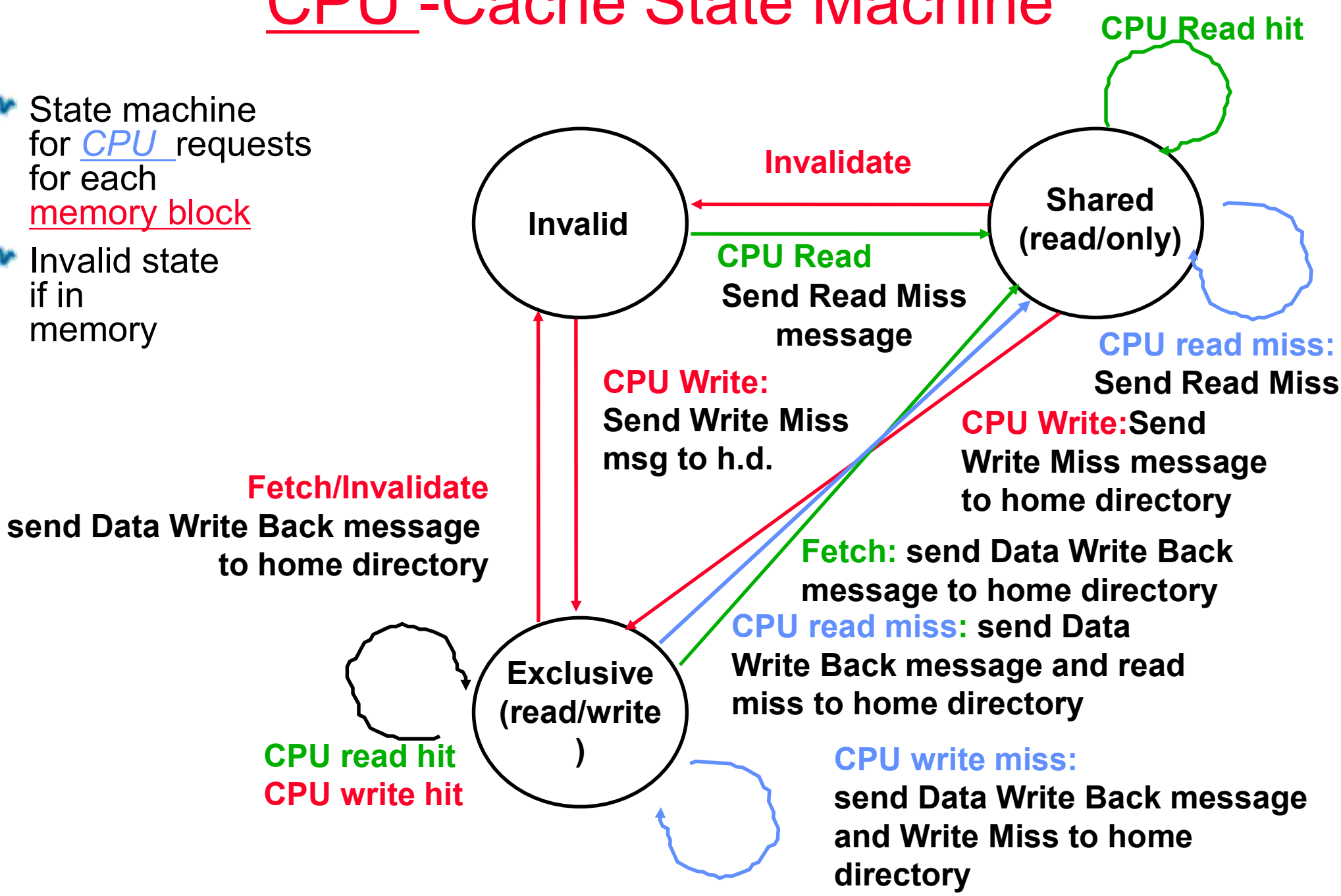
- *Write-back a data value for address A (invalidate response)*

State Transition Diagram for an Individual Cache Block in a Directory Based System

- States identical to snoopy case; transactions very similar.
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss msg to home directory.
- Write misses that were broadcast on the bus for snooping => explicit invalidate & data fetch requests.
- Note: on a write, a cache block is bigger, so need to read the full cache block

CPU -Cache State Machine

- State machine for *CPU* requests for each memory block
- Invalid state if in memory

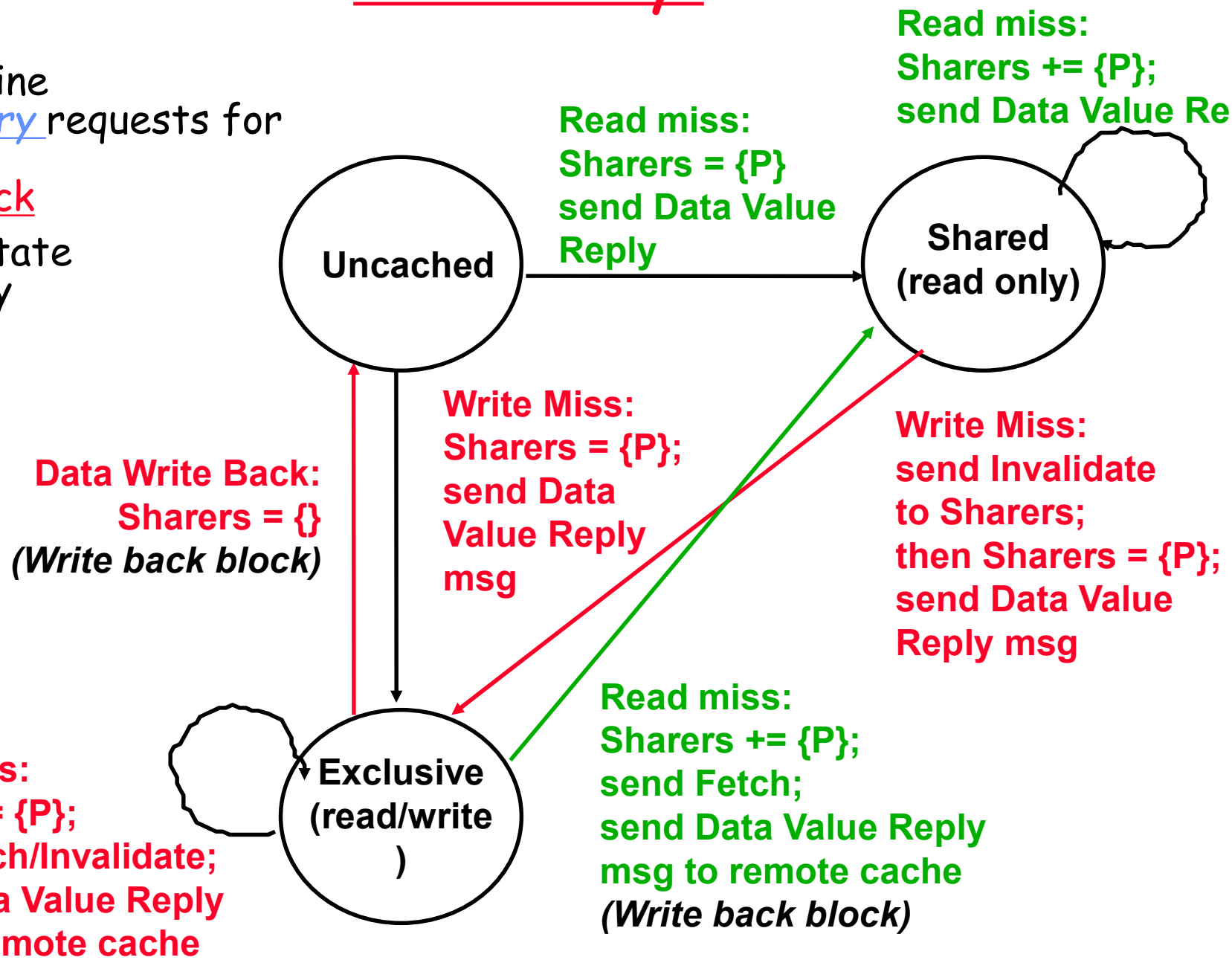


State Transition Diagram for the Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send msgs to satisfy requests
- Tracks all copies of memory block.
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message.

Directory State Machine

- State machine for *Directory* requests for each memory block
- Uncached state if in memory



Example Directory Protocol

- Message sent to directory causes two actions:
  - Update the directory
  - More messages to satisfy request
- Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  - Read miss: requesting processor sent data from memory &requestor made only sharing node; state of block made Shared.
  - Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.
- Block is Shared => the memory value is up-to-date:
  - Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  - Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

Example Directory Protocol

- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:
  - Read miss: owner processor sent data fetch message, causing state of block in owner's cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor. Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.
  - Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date
    (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
  - Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

Example

Processor 1 Processor 2 Interconnect Directory Memory

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

A1 and A2 map to the same cache block

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Directory Addr	Directory State	Directory {Procs}	Memory Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2							WrMs	P2	A2		A2	Excl.	{P2}	0
							WrBk	P2	A1	20	A1	Unca.	{}	20
				Excl.	A2	40	DaRp	P2	A2	0	A2	Excl.	{P2}	0

A1 and A2 map to the same cache block

Implementing a Directory

- We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network (see H&P 3rd ed Appendix E)

- Optimizations:
  - read miss or write miss in Exclusive: send data directly to requestor from owner vs. 1st to memory and then from memory to requestor

Notes for questions

Directories

- A directory in a cache coherency protocol is a mechanism to track which remote caches need to be invalidated when a store is executed
- A cache requires invalidation if it might contain a copy of the cache line targeted by the store
- One idea might be to keep (with every cache line that we own) a bit vector, with a bit set for each destination to which an invalidation should be sent
  - Eg in SGI's Origin2000, every cache line has a 64-bit directory
- In S3MP the directory is represented as a singly-linked list, pointed to by a field in the main-memory location when the cache line lives
- There are alternatives. For example we could keep a small number of bits with each cache line, indicating whether there might be a copy of the line
  - In another cache on this chip
  - In another cache in this socket
    - When the remote chip receives the invalidation message, it may use a "snoop filter" to track which caches within this chip require

Under what circumstances might there be contention at a cache controller?

```
##################################################
# using Gaussian elimination, find x where A*x = b #
##################################################
PROC in situ gaussian elimination = (REF MAT a, b)REF MAT: (
# Note: a and b are modified "in situ", and b is returned as x #

  FOR diag TO UPB a-1 DO
    INT pivot row := diag; SCAL pivot factor := ABS a[diag,diag];
    FOR row FROM diag + 1 TO UPB a DO # Full pivoting #
      SCAL abs a diag = ABS a[row,diag];
      IF abs a diag>=pivot factor THEN
        pivot row := row; pivot factor := abs a diag FI
    OD;
  # now we have the "best" diag to full pivot, do the actual pivot #
    IF diag NE pivot row THEN
    # a[pivot row,] =:= a[diag,]; XXX: unoptimised # #DB#
      a[pivot row,diag:] =:= a[diag,diag:]; # XXX: optimised #
      b[pivot row,] =:= b[diag,] # swap/pivot the diags of a & b #
    FI;

    IF ABS a[diag,diag] <= near min scal THEN
      raise value error("singular matrix") FI;
    SCAL a diag reciprocal := 1 / a[diag, diag];

    FOR row FROM diag+1 TO UPB a DO
      SCAL factor = a[row,diag] * a diag reciprocal;
    # a[row,] -:= factor * a[diag,] XXX: "unoptimised" # #DB#
      a[row,diag+1:] -:= factor * a[diag,diag+1:];# XXX: "optimised" #
      b[row,] -:= factor * b[diag,]
    OD
  OD;
```

Gaussian elimination with partial pivoting:

- We iterate along the diagonal of the matrix
- At each step we pick the best row to perform an elimination step
  - The row least likely to cause rounding errors
- Then we do the elimination in parallel

The pivot row is picked on each iteration

Then *every* processor reads it

- So every processor requests data from the cache controller holding the pivot row

https://rosettacode.org/wiki/Gaussian_elimination#ALGOL_68

Under what circumstances might there be contention at a cache controller?

```
##################################################
# using Gaussian elimination, find x where A*x = b #
##################################################
PROC in situ gaussian elimination = (REF MAT a, b)REF MAT: (
# Note: a and b are modified "in situ", and b is returned as x #

  FOR diag TO UPB a-1 DO
    INT pivot row := diag; SCAL pivot factor := ABS a[diag,diag];
    FOR row FROM diag + 1 TO UPB a DO # Full pivoting #
      SCAL abs a diag = ABS a[row,diag];
      IF abs a diag>=pivot factor THEN
        pivot row := row; pivot factor := abs a diag FI
    OD;
  # now we have the "best" diag to full pivot, do the actual pivot #
    IF diag NE pivot row THEN
    # a[pivot row,] =:= a[diag,]; XXX: unoptimised # #DB#
      a[pivot row,diag:] =:= a[diag,diag:]; # XXX: optimised #
      b[pivot row,] =:= b[diag,] # swap/pivot the diags of a & b #
    FI;

    IF ABS a[diag,diag] <= near min scal THEN
      raise value error("singular matrix") FI;
    SCAL a diag reciprocal := 1 / a[diag, diag];

    FOR row FROM diag+1 TO UPB a DO
      SCAL factor = a[row,diag] * a diag reciprocal;
    # a[row,] -:= factor * a[diag,] XXX: "unoptimised" # #DB#
      a[row,diag+1:] -:= factor * a[diag,diag+1:];# XXX: "optimised" #
      b[row,] -:= factor * b[diag,]
    OD
  OD;
```

Gaussian elimination with partial pivoting:

- We iterate along the diagonal of the matrix
- At each step we pick the best row to perform an elimination step
  - The row least likely to cause rounding errors
- Then we do the elimination in parallel

The pivot row is picked on each iteration

Then *every* processor reads it

- So every processor requests data from the cache

Sarah A. M. Talbot, Paul H. J. Kelly: **Adaptive Proxies: Handling Widely-Shared Data in Shared-Memory Multiprocessors,** Euro-Par 2000: 567-572 https://link.springer.com/content/pdf/10.1007/BFb0024734.pdf

https://rosettacode.org/wiki/Gaussian_elimination#ALGOL_68

Can you think of an example of a program that creates long sharing chains which are frequently invalidated?

```
##################################################
# using Gaussian elimination, find x where A*x = b #
##################################################
PROC in situ gaussian elimination = (REF MAT a, b)REF MAT: (
# Note: a and b are modified "in situ", and b is returned as x #

  FOR diag TO UPB a-1 DO
    INT pivot row := diag; SCAL pivot factor := ABS a[diag,diag];
    FOR row FROM diag + 1 TO UPB a DO # Full pivoting #
      SCAL abs a diag = ABS a[row,diag];
      IF abs a diag>=pivot factor THEN
        pivot row := row; pivot factor := abs a diag FI
    OD;
  # now we have the "best" diag to full pivot, do the actual pivot #
    IF diag NE pivot row THEN
    # a[pivot row,] =:= a[diag,]; XXX: unoptimised # #DB#
      a[pivot row,diag:] =:= a[diag,diag:]; # XXX: optimised #
      b[pivot row,] =:= b[diag,] # swap/pivot the diags of a & b #
    FI;

    IF ABS a[diag,diag] <= near min scal THEN
      raise value error("singular matrix") FI;
    SCAL a diag reciprocal := 1 / a[diag, diag];

    FOR row FROM diag+1 TO UPB a DO
      SCAL factor = a[row,diag] * a diag reciprocal;
    # a[row,] -:= factor * a[diag,] XXX: "unoptimised" # #DB#
      a[row,diag+1:] -:= factor * a[diag,diag+1:];# XXX: "optimised" #
      b[row,] -:= factor * b[diag,]
    OD
  OD;
```

Gaussian elimination with partial pivoting:

- We iterate along the diagonal of the matrix
- At each step we pick the best row to perform an elimination step
  - The row least likely to cause rounding errors
- Then we do the elimination in parallel

The pivot row is picked on each iteration

Then *every* processor reads it

- So every processor requests data from the cache controller holding the pivot row
- So now cache copies of the pivot row are everywhere
- If the pivot row is overwritten later, they all have to be invalidated

https://rosettacode.org/wiki/Gaussian_elimination#ALGOL_68

NUMA and its relatives

- NUMA: Non-Uniform Memory Architecture
  - Any machine where some memory is nearer than other memory
  - Eg two-socket shared-memory machine with DRAM attached to both sockets
- CC-NUMA: cache-coherent NUMA
  - The “home” of each physical address is in a fixed physical location, possibly nearer, possibly further away
- COMA: cache-only memory architecture
  - The home of a physical address might be dynamically migrated to be nearer where it is being used
- S3MP is a NUMA machine – data might be in your core’s local DRAM, or remote

- **Objective: make sure every processor that tries to claim the lock eventually succeeds**
- **When a thread attempts to claim the lock, it is assigned a number to wait for**

```
1  ticketLock_init(int *next_ticket, int *now_serving)
2  {
3      *now_serving = *next_ticket = 0;
4  }
5
6  ticketLock_acquire(int *next_ticket, int *now_serving)
7  {
8      my_ticket = fetch_and_inc(next_ticket);
9      while (*now_serving != my_ticket) {}
10 }
11
12 ticketLock_release(int *now_serving)
13 {
14     ++*now_serving;
15 }
```

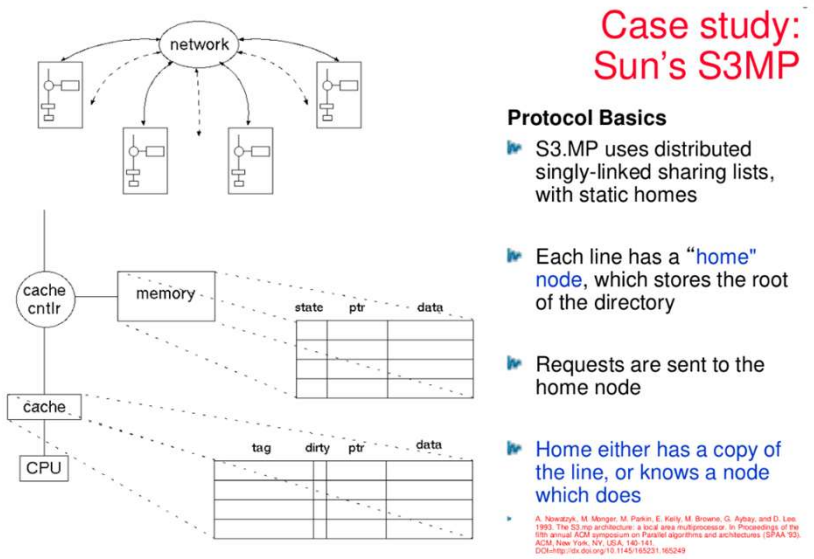
https://en.wikipedia.org/wiki/Ticket_lock

Four Processor Ticket Lock Example

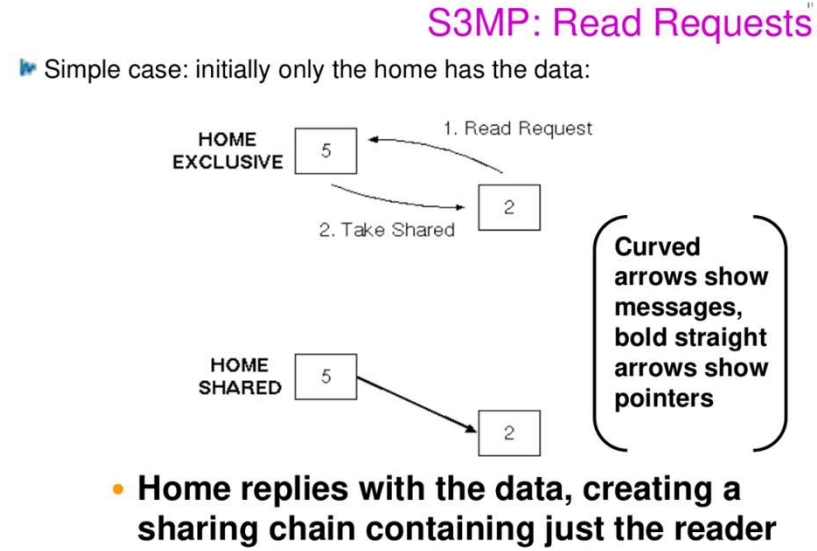
Row	Action	next_ticket	now_serving	P1 my_ticket	P2 my_ticket	P3 my_ticket	P4 my_ticket
1	Initialized to 0	0	0	-	-	-	-
2	P1 tries to acquire lock (succeed)	1	0	0	-	-	-
3	P3 tries to acquire lock (fail + wait)	2	0	0	-	1	-
4	P2 tries to acquire lock (fail + wait)	3	0	0	2	1	-
5	P1 releases lock, P3 acquires lock	3	1	0	2	1	-
6	P3 releases lock, P2 acquires lock	3	2	0	2	1	-
7	P4 tries to acquire lock (fail + wait)	4	2	0	2	1	3
8	P2 releases lock, P4 acquires lock	4	3	0	2	1	3
9	P4 releases lock	4	4	0	2	1	3
10	...	4	4	0	2	1	3

Student question: S3MP sharing chains

I am still quite confused about the `ptr` in the following slide.

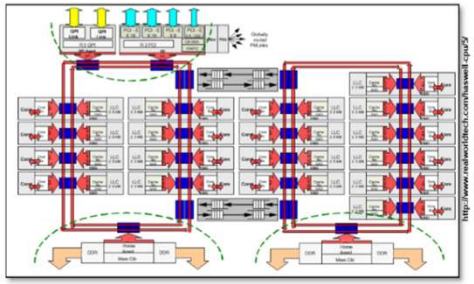


I would like to know. Where the pointers are pointing to in the processors 5 and 2 in the following example:

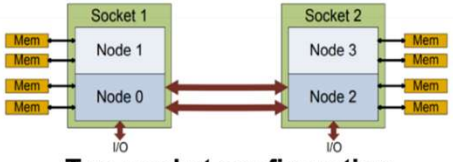


- The ptr field stored with a cache block in *memory* points to the head of the sharing chain - it tells you where to find the first cache controller that has a copy of this cache block (the "state" field tells you whether the data held in the memory is in fact up-to-date, or whether, instead, the up-to-date data is found in the cache pointed to by ptr.
- The ptr field in the *cache* implements the linked sharing list. If that cache is the end of the list, its ptr will be null. If there is another cache that also has a copy of the block, ptr will point to the next cache in the chain.
- In the example in the second slide:
- **Initially**, the *memory* copy of the block in node 5 carries a null ptr. The "Home exclusive" state tells us that the only valid copy of this block is the copy in node 5's memory. Node 2 does not have a copy of this cache block (none of the blocks in the indexed set has a matching tag).
- **After the read-request/take-shared transaction:** there are now two copies of the data - in the memory at node 5, and in the cache at node 2. There is a one-element sharing chain. The cache block's ptr field in node 5's memory holds the id of the cache controller 2 where the first (and in this case the only) cache line in the sharing chain resides. The cache block stored in node 2's cache has a ptr field which is null, because that's the end of the sharing chain.

Student question: directories in Intel multicore CPUs

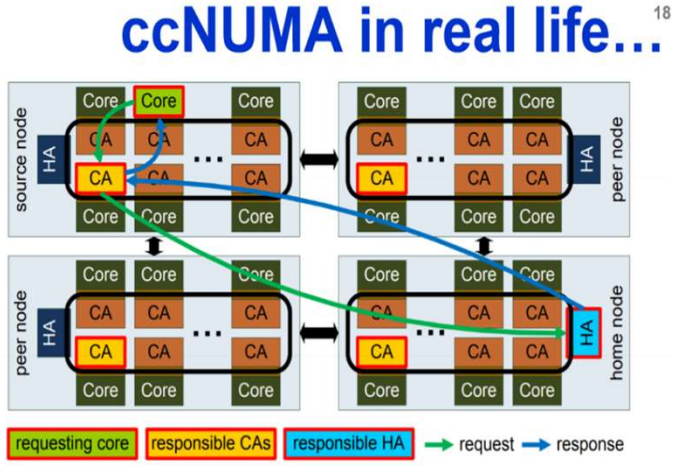


18-core chip with two rings



Two-socket configuration

("cluster-on-die" mode)



- Each cache line has 2 bits of directory indicating whether the line is held in other nodes: *remote-invalid, snoop-all* (potentially modified copy exists), or *shared* (multiple clean copies exist)
- On L2 miss, core sends request to a Cache Agent on its node (based on physical address)
- The Cache Agent checks for a local L3 hit – but if miss, passes request to Home Agent
- Invalidations and read requests are propagated to other nodes accordingly by the Home Agent
- Directory information for frequently-exchanged lines are cached in the Home Agent (8 bits)

- **Recall: Intel Haswell e5 2600 v3**
- **A complex hybrid coherency scheme**

Daniel Molka, Daniel Hackenberg, Robert Schone, and Wolfgang E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. ICPP2015 (2015_ICPP_authors_version.pdf (tu-dresden.de))

in this example, I was wondering how this is implementing snooping? I understood snooping to be monitoring the interconnection bus for reads/writes etc. and causing the state transition from this. This implementation above seems mainly to be directory based as far as I can tell

- Within a single ring, snooping works: every cache controller sees every core's invalidations and read-requests.
- So the question is what happens *between* rings. We need some kind of directory - to know that a copy of the data might exist in another ring, and (in the two-socket case) which rings might have a copy.
- There is a detailed but more-or-less comprehensible explanation of how this was done in Haswell in this article:
- 2015_ICPP_authors_version.pdf (tu-dresden.de)
  https://tu-dresden.de/zih/forschung/ressourcen/dateien/abgeschlossene-projekte/benchit/2015_ICPP_authors_version.pdf?lang=en
- Section IV, pages 3-4.
- Very briefly: basically each cache line has a state bit that indicates whether a copy exists in *any* remote ring. If it does, invalidations and read-requests are sent to the "home agent" of *all* remote rings.
- However this is accelerated for highly-shared cache lines using a "directory cache" - if a line is in the directory cache then more refined directory information is available, which allows invalidations and read-requests to be sent to a refined subset of remote caches.
- If you like this sort of thing, you might enjoy the slides included in this blog: John McCalpin's blog » Cache Coherence Protocols (utexas.edu)
  https://sites.utexas.edu/jdm4372/category/computer-architecture/cache-coherence-protocols/ (scroll down for the slides). The talk is about tracking down a performance bug in a large supercomputer, caused by associativity conflicts in the snoop filter (basically the "directory cache" referred to above). Leading to a 28% performance improvement.

Student question: shared memory or message-passing?

- **Q:** I am trying to understand the real difference between shared memory and distributed memory.
  In shared memory, the hardware still has to move cache lines between cores when data is accessed in a different pattern, so communication still exists, it's just at the hardware level.
  In distributed memory, the programmer moves the data explicitly through MPI send/recv.
  So where is the convenience of shared memory? Is it just programmer convenience or there may be an actual performance benefit? Am I misunderstanding something?

- **Q:** Some considerations:
- (1) with shared memory, communication is initiated by a load instruction - there is no other software overhead
- *[you might wonder whether we could implement send/recv in the ISA; note that send & recv work on vectors of data [you could say they are vectorised]]*
- (2) with shared memory, a thread can load from any address without needing to know which other thread produced the value store there. In contrast, with MPI send/recv, the core that holds the data needs to send it - so it needs to know when and where that data will be needed. A partial solution is to use MPI's Remote Memory Access API (often called "one-sided communication"). But with RMA you still need to know which processor to read it from (see https://scalable.uni-jena.de/opt/pc/chapters/assignment_mpi_rma.html).
- *[You might observe that send/recv avoids the need for things the cache coherency protocol does for you. Actually so does RMA.]*
- (3) Suppose one thread writes some data, and then every core needs to read that data. In shared memory this can be a mess, because the read requests might all have to be served by one cache controller. In contrast, in MPI you would use a broadcast (MPI_Bcast) - which would use a highly-optimised network-topology-aware tree algorithm.
- *[In the S3MP protocol described in the lecture, this really is an acute problem for some applications (Gaussian elimination with row pivoting is the famous example - every thread needs the row that was selected to be the pivot). However in a clustered design, the problem can be mitigated to some extent, I think, using snoop filters - see Ch10 part 4 slide 46. I actually had a PhD student, Sarah Talbot, work on this problem https://link.springer.com/content/pdf/10.1007/BFb0057968.pdf]*
- Note that implementing MPI requires careful attention to minimising copying - we don't want to copy data into intermediary buffers. And also cache coherency - an MPI_send needs to invalidate (or update) the receiver's cached data at the specified address.
- MPI implementations are finely-honed and message passing can be better than shared memory even within a single node of a distributed-memory machine. This talk covers the topic https://www.archer2.ac.uk/training/courses/220727-mpi-openmp-performance-vt/ . Our own experience in our https://www.firedrakeproject.org/ project is that MPI beats MPI+OpenMP, but your mileage will vary.
- Note that in a distributed-memory multi-node GPU system, you want the GPUs to be able to communicate directly to each other, without having the host CPU as intermediary. See https://developer.nvidia.com/nccl .