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 2023

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

What you should get from this

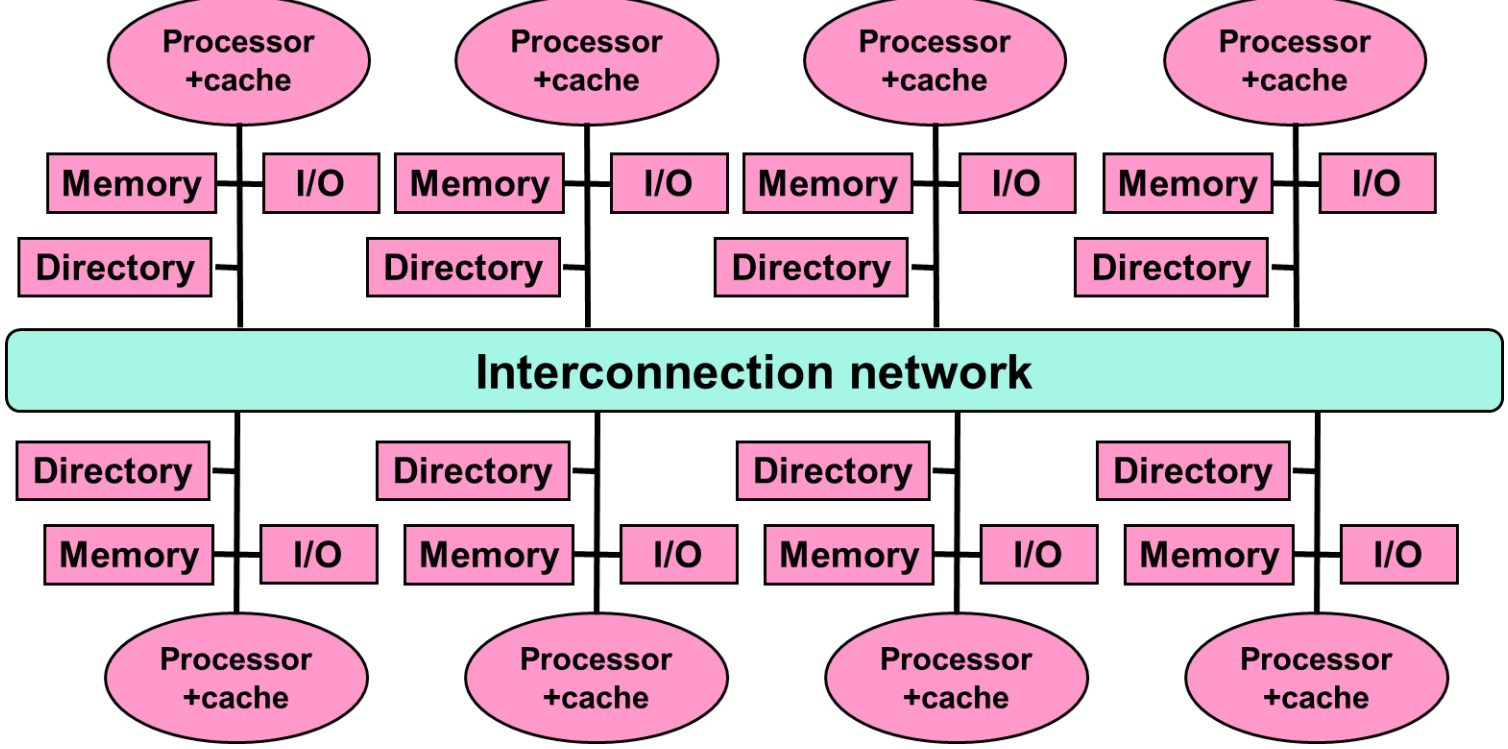
Parallel systems architecture is a vast topic, and we can only scratch the surface. The critical things I hope you will learn from this very brief introduction are:

- 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

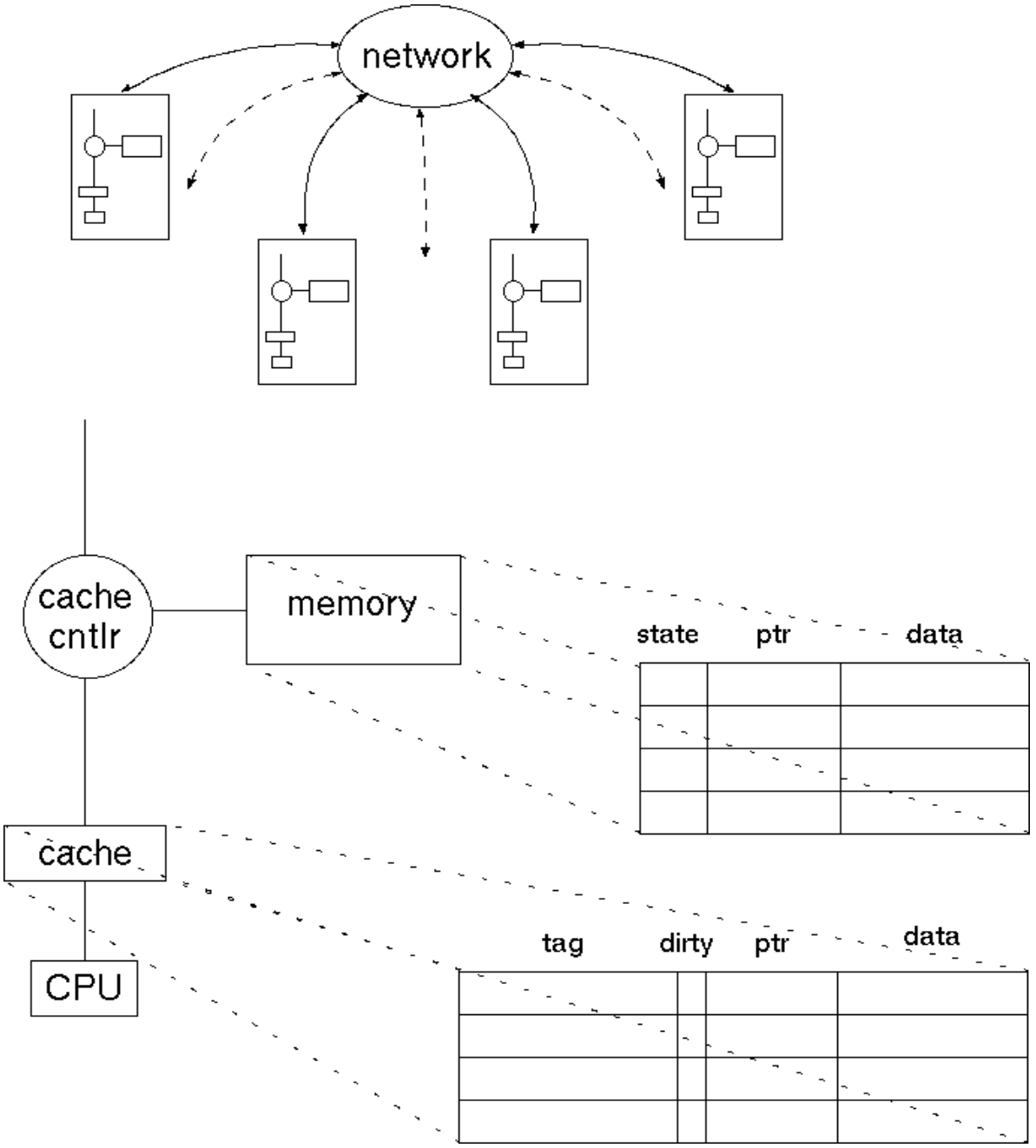
- Snooping cache coherency protocols rely on a bus:
  - For broadcasting invalidations and read requests
  - To establish global ordering on events
- The bus inevitably becomes a bottleneck when many processors are used
  - So snooping does not work
  - So we need to use a more general interconnection 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

Case study: Sun's S3MP



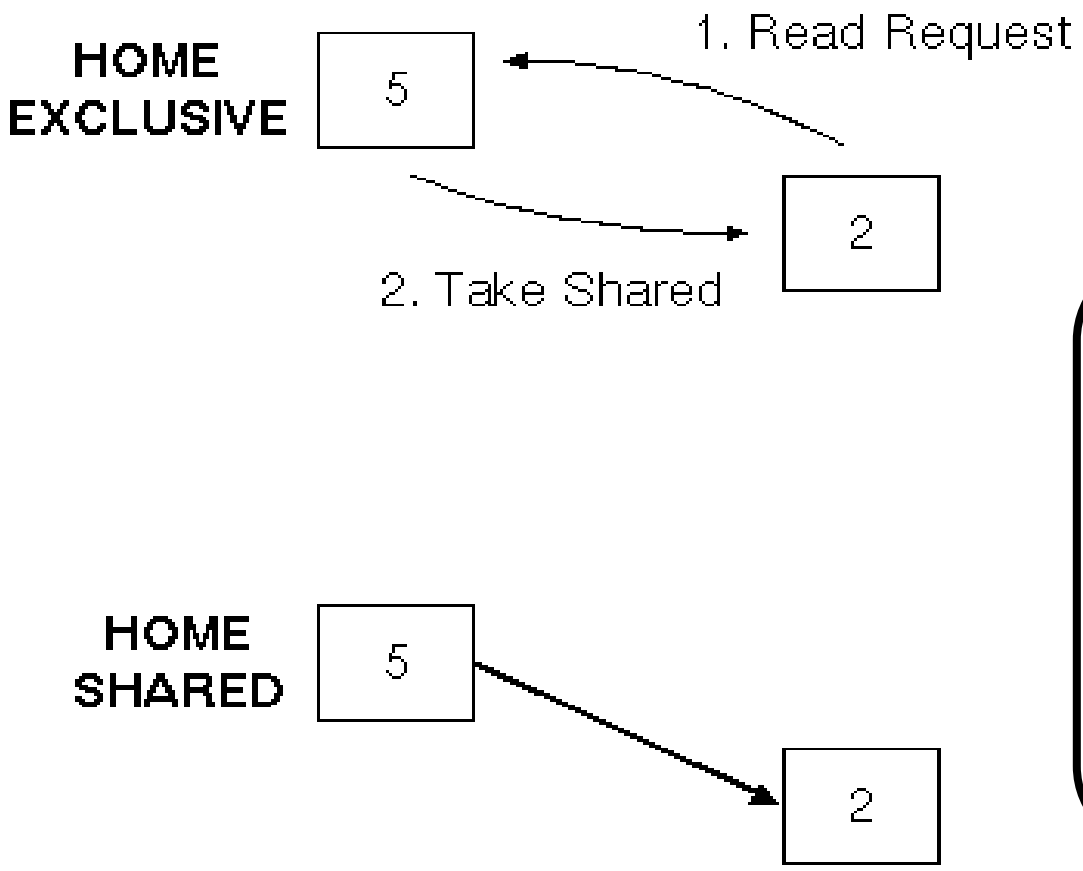
Protocol Basics

- 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

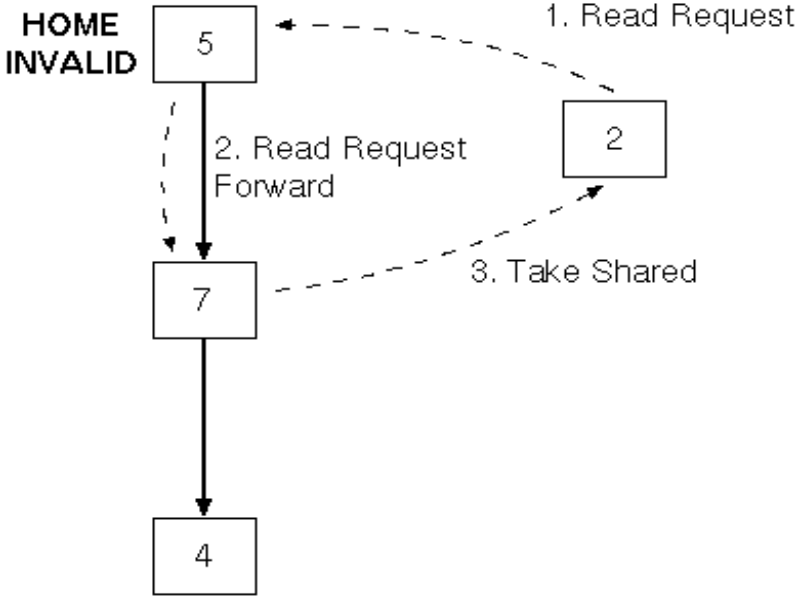
S3MP: Read Requests

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

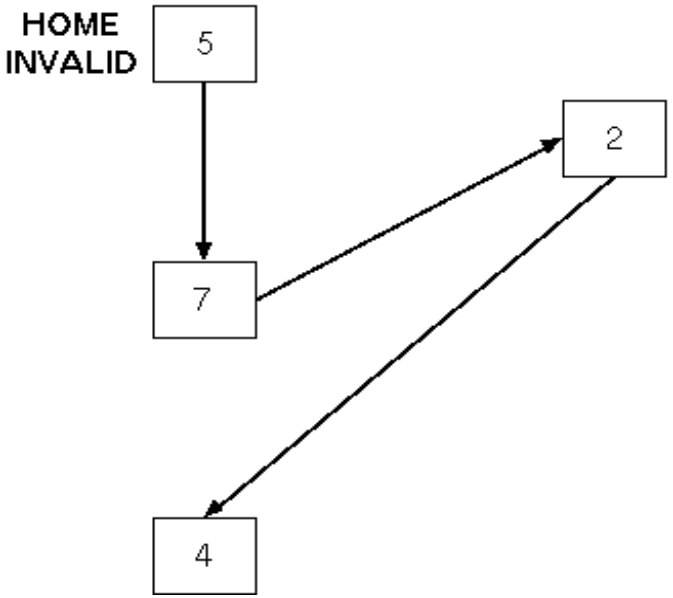


- **Home replies with the data, creating a sharing chain containing just the reader**

S3MP: Read Requests - remote

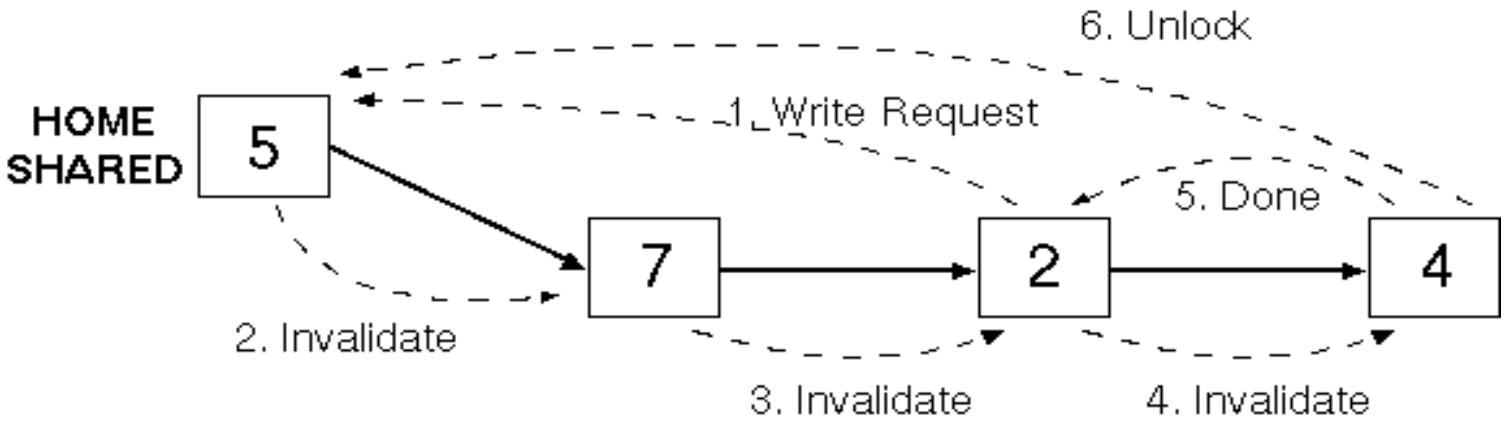


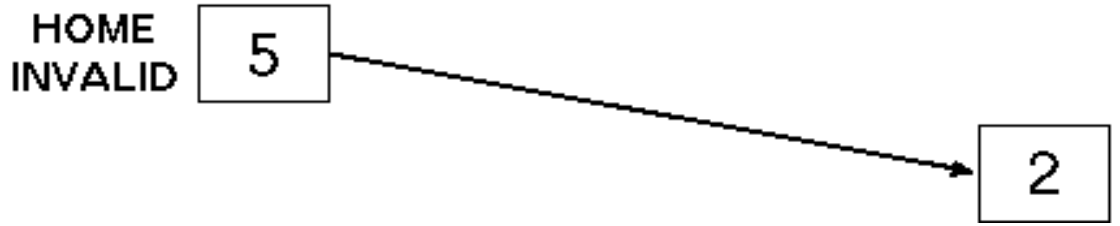
- More interesting case: some other processor has the data



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

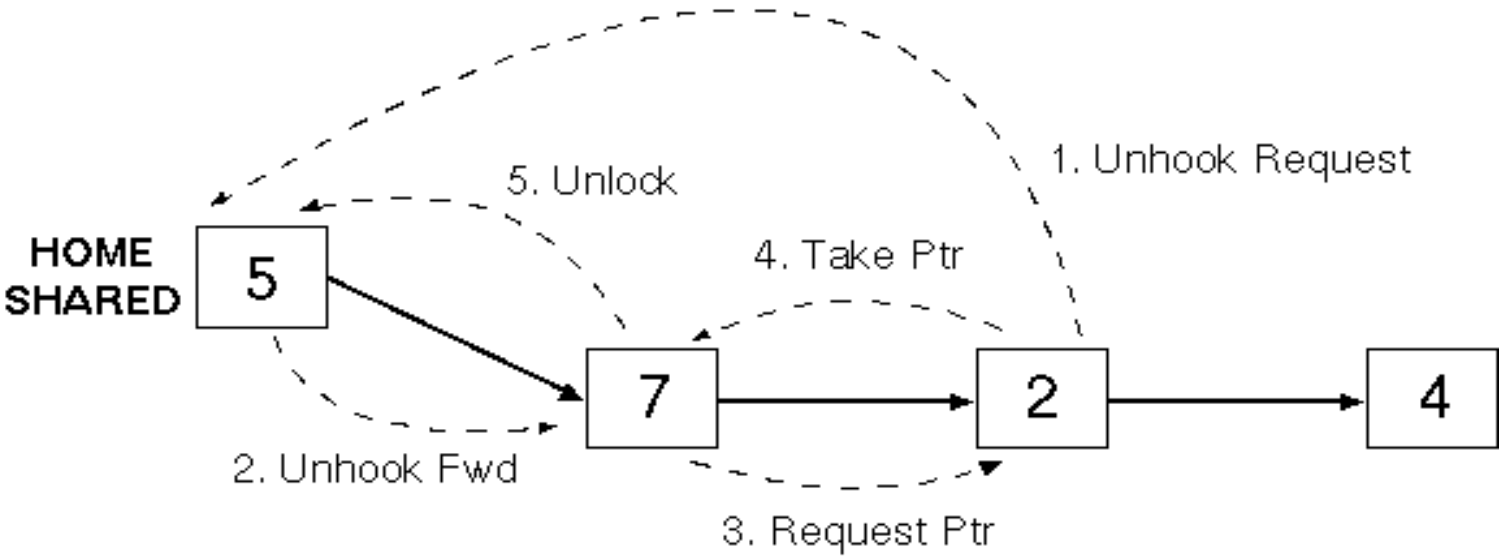
S3MP - Writes

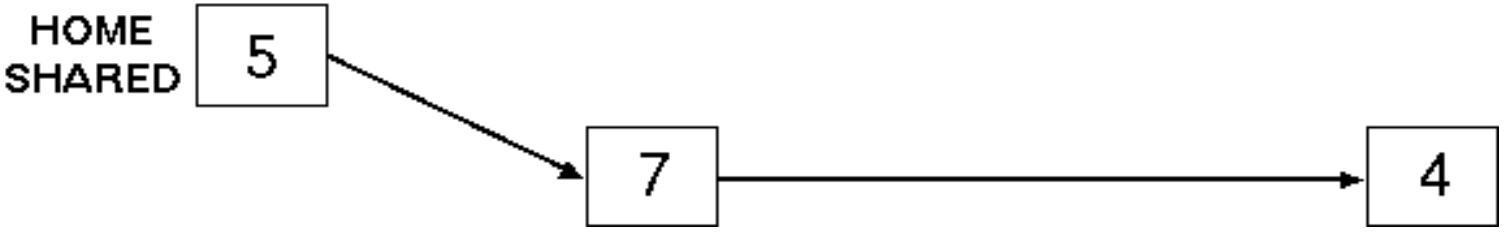




- If the line is exclusive (i.e. dirty bit is set) no message is required
- Else send a write-request to the home
  - Home sends an invalidation message down the chain
  - Each copy is invalidated (other than that of the requester)
  - Final node in chain acknowledges the requester and the home
- Chain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

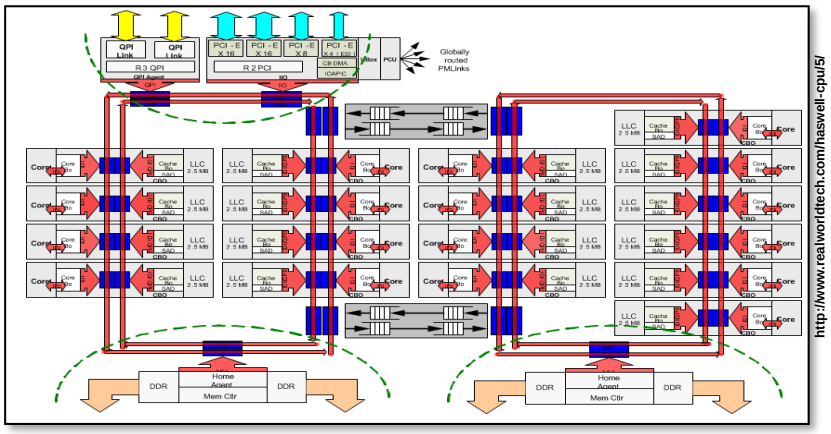
Finding your data

- 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

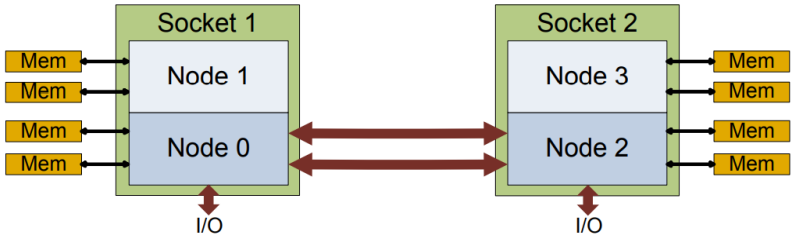
ccNUMA summary

- 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!

ccNUMA in real life…

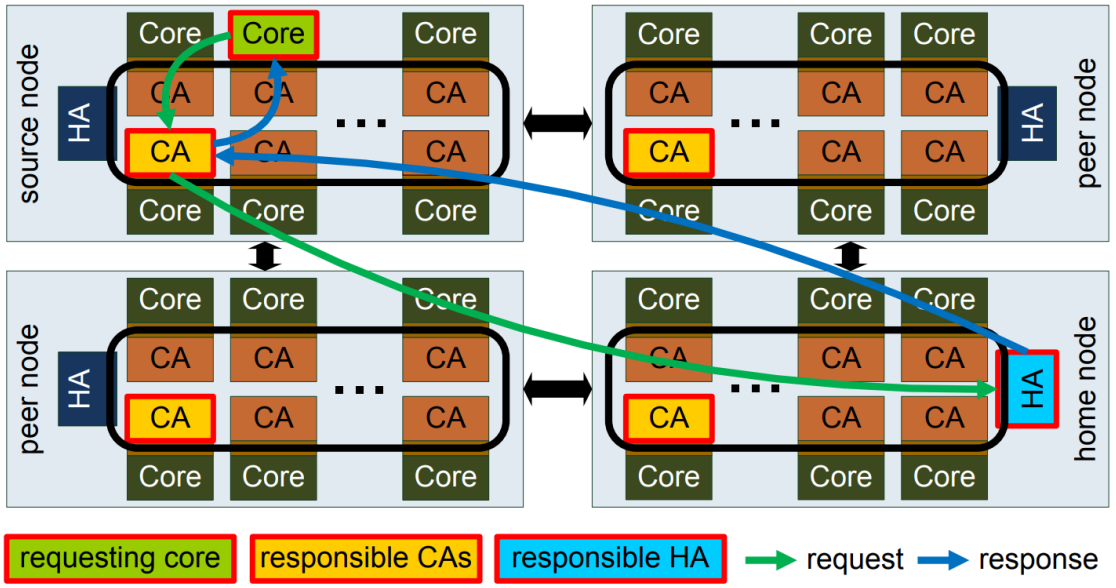


18-core chip with two rings



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)

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)

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

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

Sarah A. M. Talbot, Paul H. J. Kelly:
Adaptive Proxies: Handling Widely-Shared Data in Shared-Memory Multiprocessors (Research Note). 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