332
Advanced Computer Architecture
Chapter 7

Parallel architectures, shared memory, and cache coherency

February 2009

Paul H J Kelly

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

Overview

- Why add another processor?
- 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

London e-Science Centre's "Mars" cluster

408 CPUs:

- 72 dual 1.8GHz Opteron processors, 2Gb memory, 80Gb S-ATA disk, Infiniband
- 40 dual 1.8GHz Opteron processors, 4Gb memory, 80Gb S-ATA disk, Gigabit Ethernet
- 88 dual 1.8GHz Opteron processors, 2Gb memory, 80Gb S-STA disk, Gigabit Ethernet
- 4 dual 2.2GHz Opteron processors, 4Gb memory, 36Gb SCSI disk, Gigabit Ethernet
- 1 Mellanox MTS9600 Infiniband switch configured with 72 ports
- 5 Extreme Networks Summit 400-48t 48 port Gigabit Switches

NPACI Blue Horizon, San Diego Supercomputer Center

- **144** eight-processor SMP High Nodes perform the primary compute functions.
- **12** two-processor SMP High Nodes perform service functions.
- **1,152** Power3 processors in the compute nodes run at 375 MHz. Each processor has a peak performance of 1.500 Mflops (millions of floating-point operations per second)
- **576** gigabytes of total system memory, at 4 GB per compute node
- **5.1** terabytes -- 5,100 gigabytes -- of associated disk

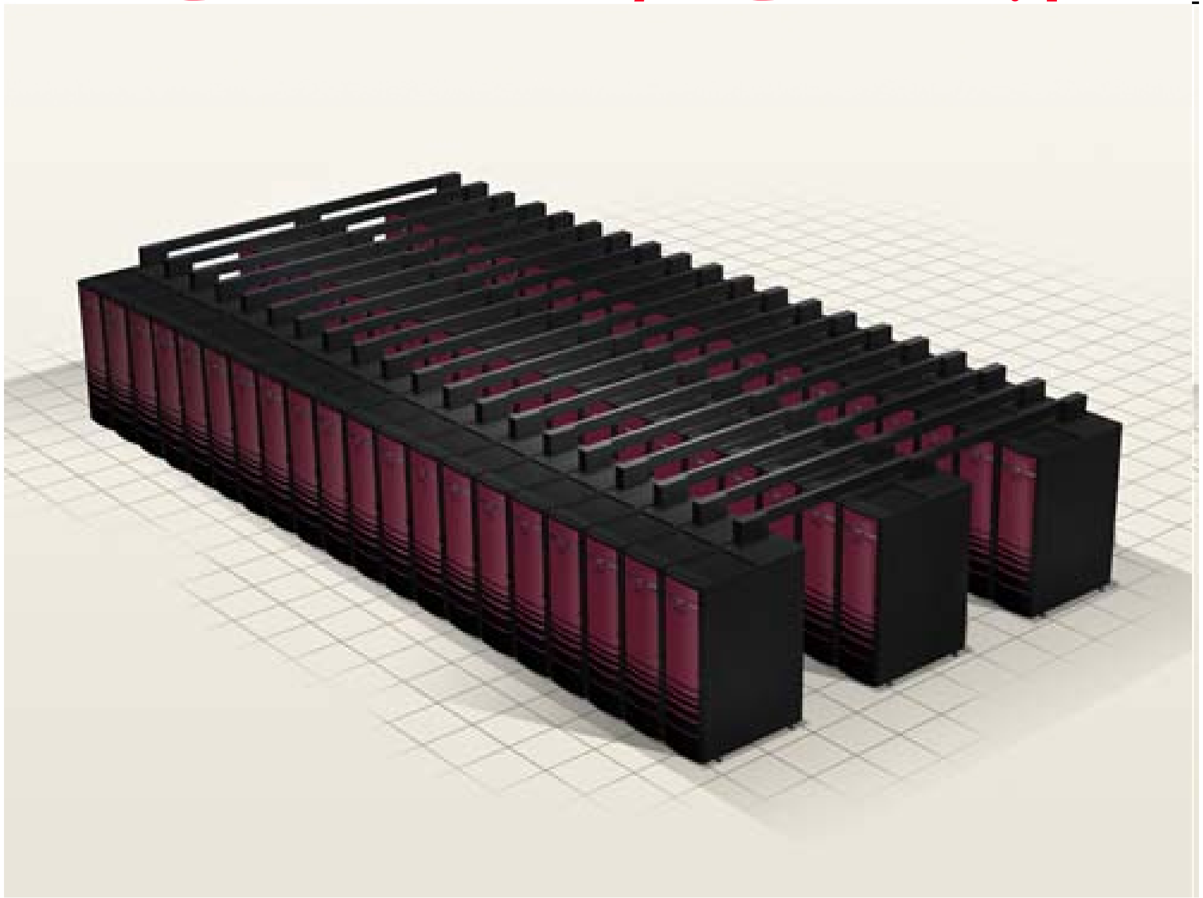
- **42 towers, or frames, house the compute and service nodes. Four racks house the disk.**
- **1,500 square feet**
- **Programmed using MPI ("Message Passing Interface")**

HPCx – Supercomputing service for UK science

- Located at Daresbury Labs, Cheshire, operated by University of Edinburgh Parallel Computing Centre
- IBM eServer 575 nodes, each 16-way SMP Power5
- 96 nodes for compute jobs for users, a total 1536 processors

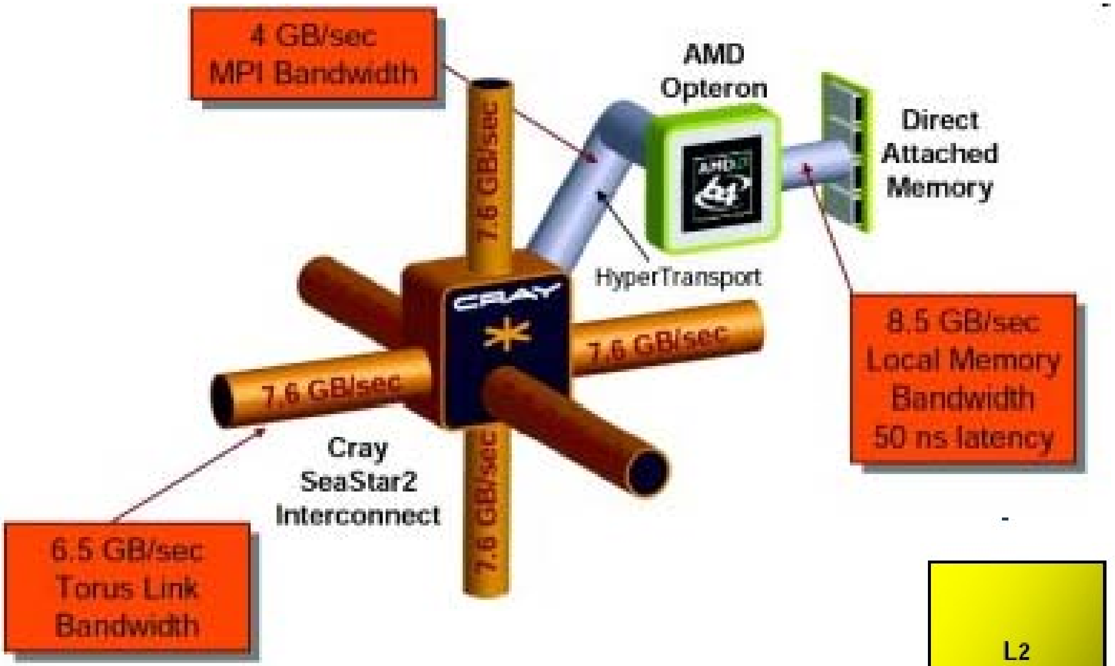
- Four chips (8 processors) are integrated into a multi-chip module (MCM)
- Two MCMs (16 processors) comprise one frame
- Each MCM is configured with 128 MB of L3 cache and 16 GB of main memory
- Total main memory of 32 GB per frame shared between 16 processors of frame
- frames connected via IBM's High Performance Switch (HPS)

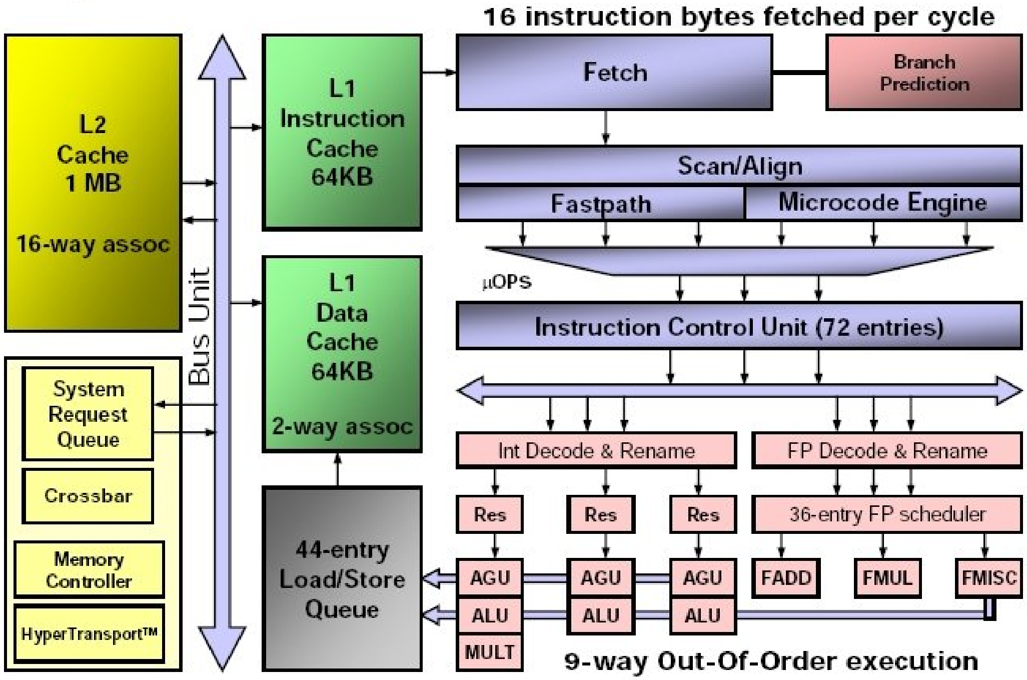
HECToR – "High-End Computing Terascale Resource"



- Successor to HPCx
- Located at University of Edinburgh, operated on behalf of UK science community
- Six-year £133M contract, includes mid-life upgrades (60->250 TeraFLOPS)
- Cray XT4: 5664 nodes with 2 cores each, so 11328 AMD Opteron cores (2.8GHz)
- 6GB RAM per node (3GB per core, 33TB in the machine)
- Cray "Seastar" interconnect (20 x 12 x 24 3D torus), directly connected to HyperTransport memory interface. Peak bi-directional bandwidth of 7.6 GB/s
- 12 I/O nodes connected to 576TB (soon to be increased) of RAID disk storage
- OS: Compute Node Linux kernel for compute nodes, full linux for I/O
- HECToR also includes a large vector processor extension, using CRAY X2 technology

http://www.hector.ac.uk





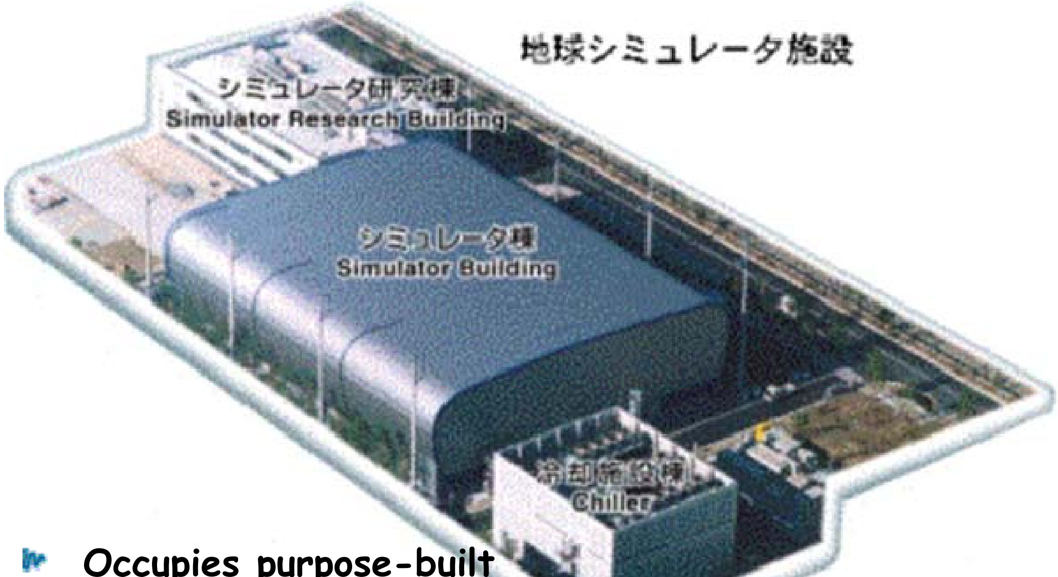
- 36 entry FPU instruction scheduler
- 64-bit/80-bit FP Realized throughput (1 Mul + 1 Add)/cycle: 1.9 FLOPs/cycle
- 32-bit FP Realized throughput (2 Mul + 2 Add)/cycle: 3.4+ FLOPs/cycle

SGI Origin 3800 at SARA, Netherlands

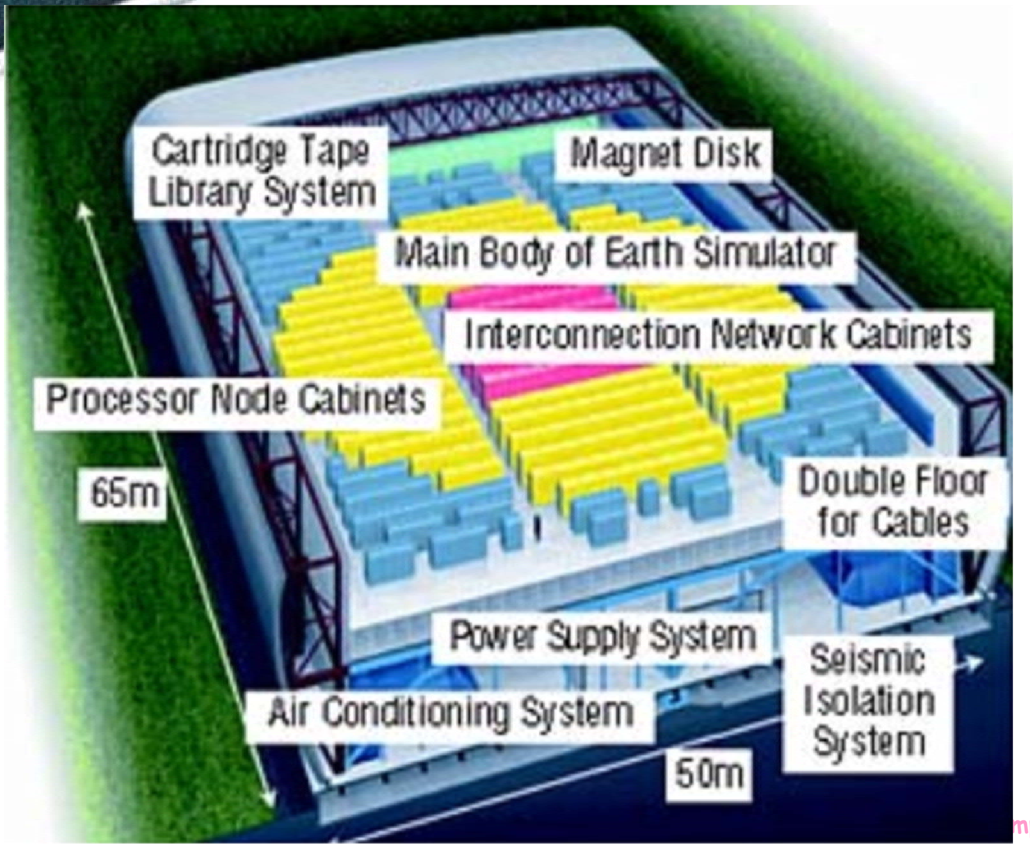
- 1024-CPU system consisting of two 512-CPU SGI Origin 3800 systems. Peak performance of 1 TFlops (10^{12} floating point operations) per second. 500MHz R14000 CPUs organized in 256 4-CPU nodes
- 1 TByte of RAM. 10 TByte of on-line storage & 100 TByte near-line storage
- 45 racks, 32 racks containing CPUs & routers, 8 I/O racks & 5 racks for disks
- Each 512-CPU machine offers application program a single, shared memory image

The “Earth Simulator”

- 5,120 (640 8-way nodes) 500 MHz NEC CPUs
- 8 GFLOPS per CPU (41 TFLOPS total)
- 2 GB (4 512 MB FPLRAM modules) per CPU (10 TB total)
- shared memory inside the node
- 640 × 640 crossbar switch between the nodes
- 16 GB/s inter-node bandwidth
- 20 kVA power consumption per node



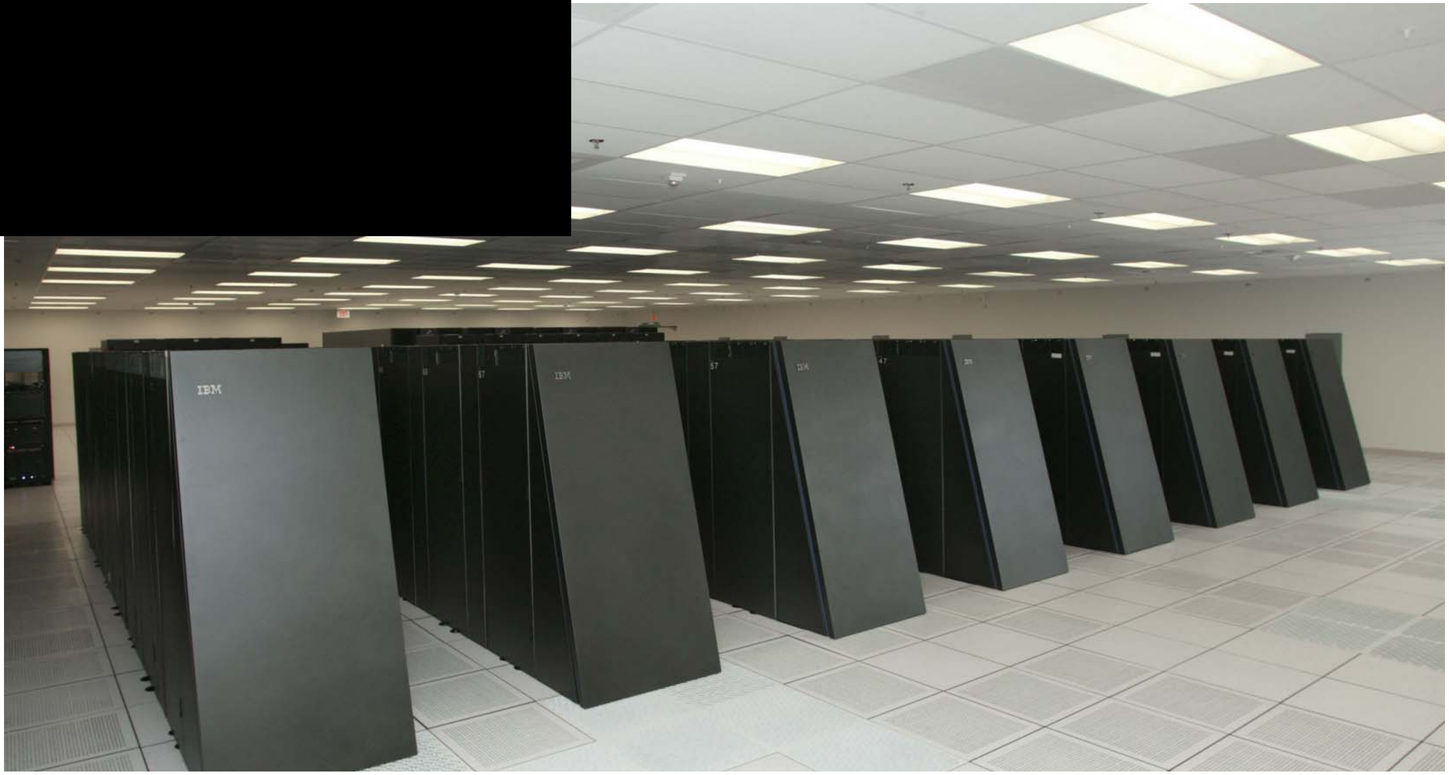
- Occupies purpose-built building in Yokohama, Japan
- Operational late 2001/early 2002
- Vector CPU using 0.15 µm CMOS process, descendant of NEC SX-5
- Runs Super-UX OS



- CPUs housed in 320 cabinets, 2 8-CPU nodes per cabinet. The cabinets are organized in a ring around the interconnect, which is housed in another 65 cabinets
- Another layer of the circle is formed by disk array cabinets.
- The whole thing occupies a building 65 m long and 50 m wide

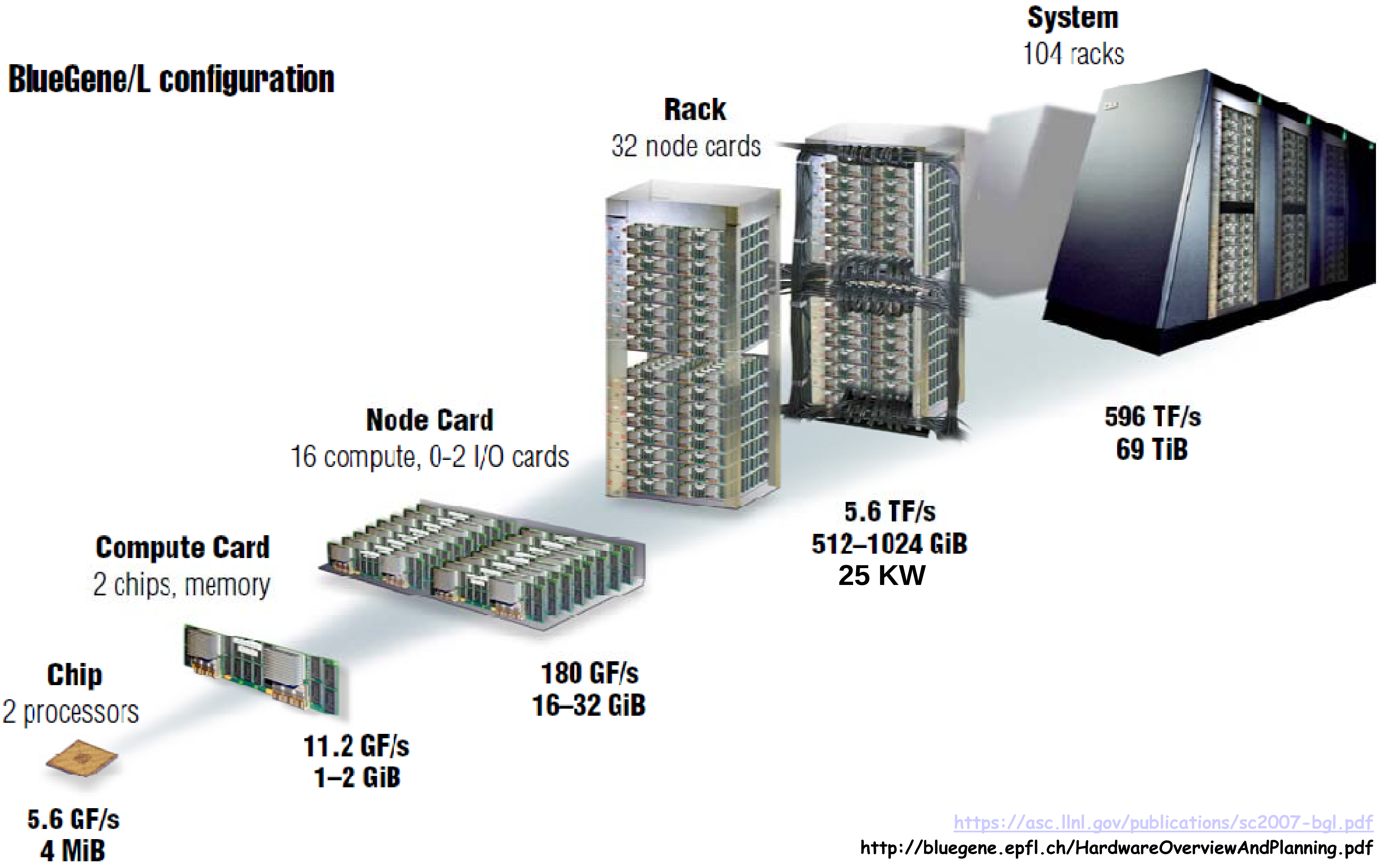
Bluegene/L at LLNL

- 1024 nodes/cabinet, 2 CPUs per node
- 106,496 nodes in total, 69TiB RAM
- 32 x 32 x 64 3D torus interconnect
- 1,024 gigabit-per-second links to a global parallel file system to support fast input/output to disk
- 1.5-3.0 MWatts



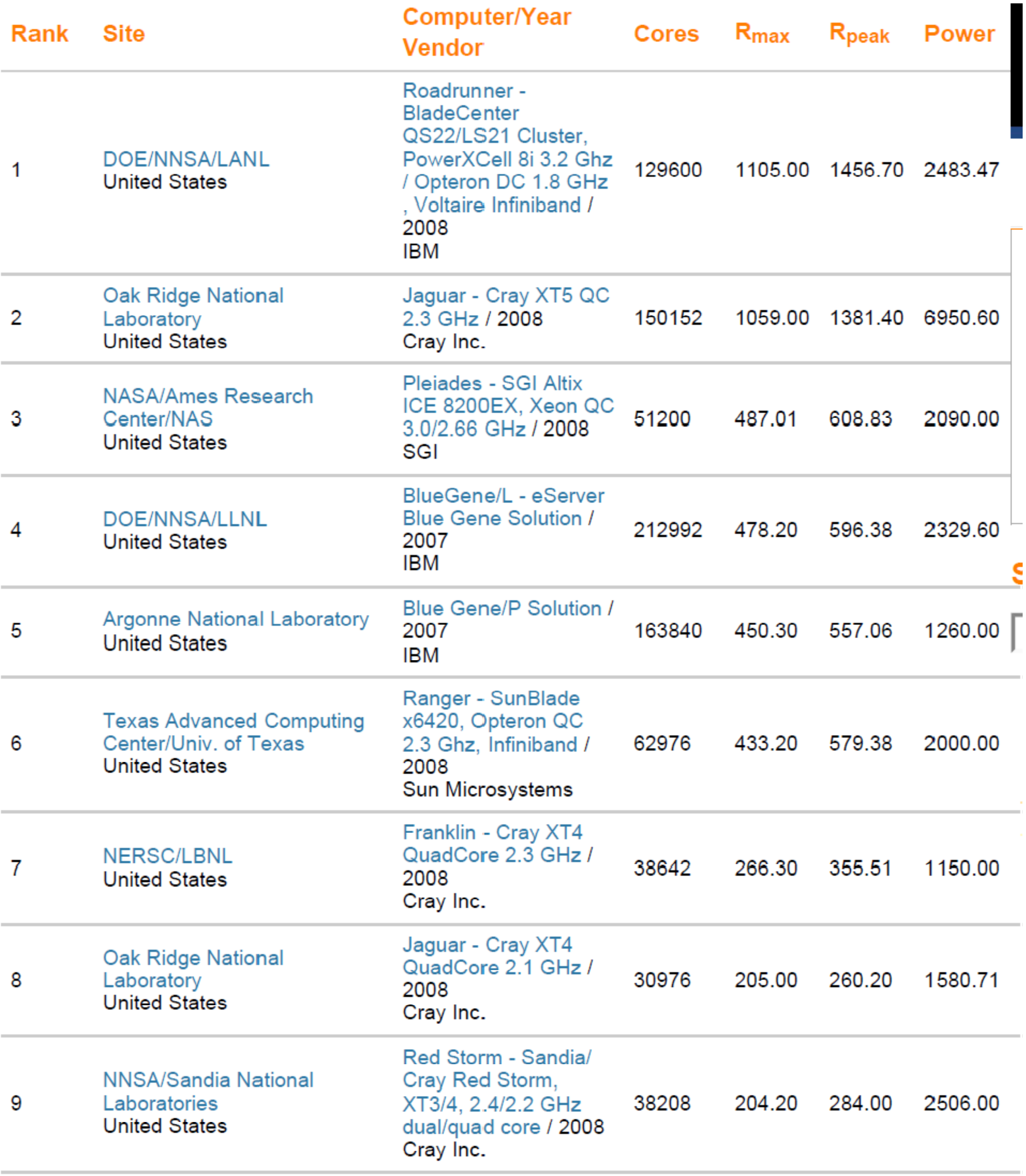
- Time-lapse movie of installation of 212,992-CPU system at Lawrence Livermore Labs
- "classified service in support of the National Nuclear Security Administration's stockpile science mission"

BlueGene/L



~~https://asc.llnl.gov/publications/sc2007-bgl.pdf~~

http://bluegene.epfl.ch/HardwareOverviewAndPlanning.pdf



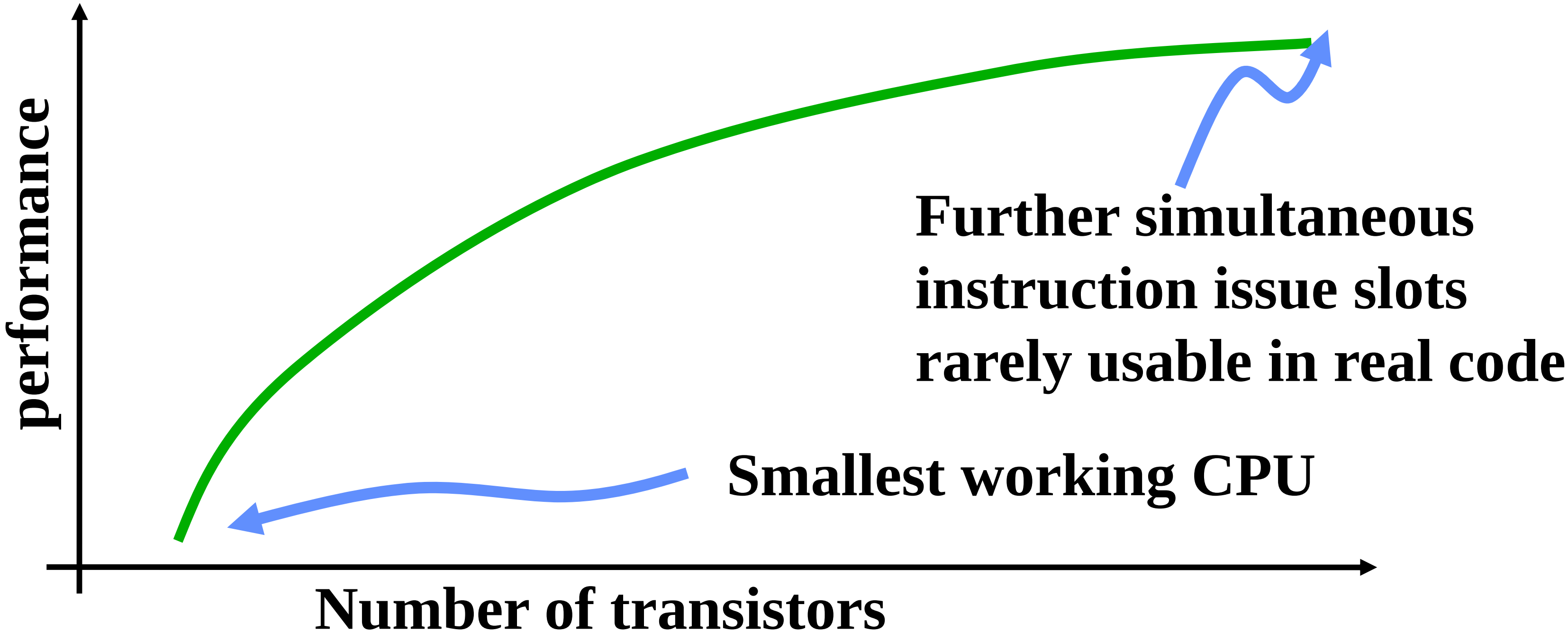
Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz , Voltaire Infiniband / 2008 IBM	129600	1105.00	1456.70	2483.47
2	Oak Ridge National Laboratory United States	Jaguar - Cray XT5 QC 2.3 GHz / 2008 Cray Inc.	150152	1059.00	1381.40	6950.60
3	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz / 2008 SGI	51200	487.01	608.83	2090.00
4	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution / 2007 IBM	212992	478.20	596.38	2329.60
5	Argonne National Laboratory United States	Blue Gene/P Solution / 2007 IBM	163840	450.30	557.06	1260.00
6	Texas Advanced Computing Center/Univ. of Texas United States	Ranger - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband / 2008 Sun Microsystems	62976	433.20	579.38	2000.00
7	NERSC/LBNL United States	Franklin - Cray XT4 QuadCore 2.3 GHz / 2008 Cray Inc.	38642	266.30	355.51	1150.00
8	Oak Ridge National Laboratory United States	Jaguar - Cray XT4 QuadCore 2.1 GHz / 2008 Cray Inc.	30976	205.00	260.20	1580.71
9	NNSA/Sandia National Laboratories United States	Red Storm - Sandia/ Cray Red Storm, XT3/4, 2.4/2.2 GHz dual/quad core / 2008 Cray Inc.	38208	204.20	284.00	2506.00

- TOP500 List - Nov 2008 (1-9)
- ranked by their performance on the LINPACK Benchmark.
- Rmax and Rpeak values are in GFlops. For more details about other fields, check the TOP500 description.
- Task is "to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine"
- http://www.top500.org

What are parallel computers used for?

- Executing loops in parallel
  - Improve performance of single application
  - Barrier synchronisation at end of loop
  - Iteration i of loop 2 may read data produced by iteration i of loop 1 – but perhaps also from other iterations
  - Example: NaSt3DGP
- High-throughput servers
  - Eg. database, transaction processing, web server, e-commerce
  - Improve performance of single application
  - Consists of many mostly-independent transactions
  - Sharing data
  - Synchronising to ensure consistency
  - Transaction roll-back
- Mixed, multiprocessing workloads

Why add another processor?

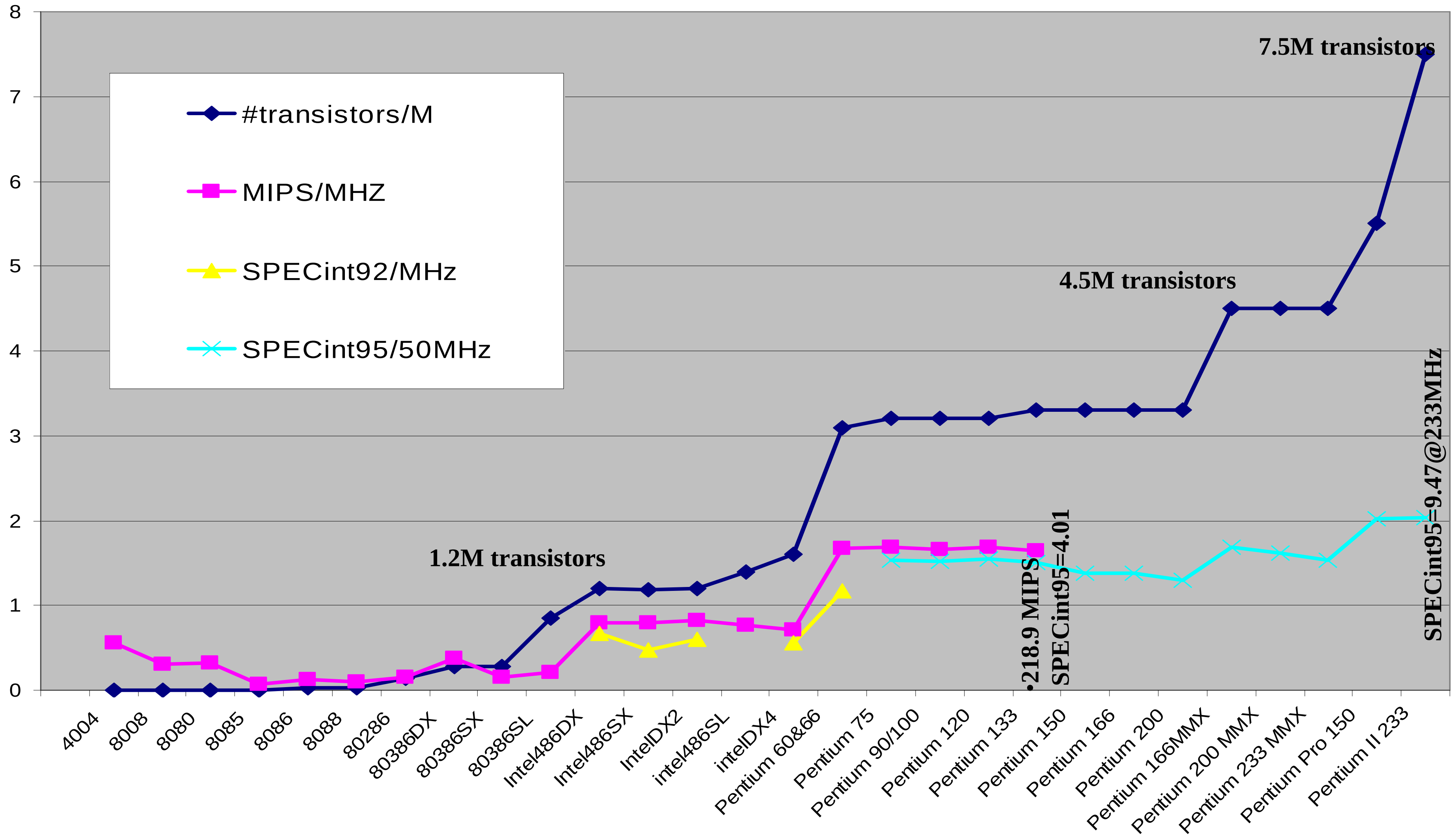


Further simultaneous instruction issue slots rarely usable in real code

Smallest working CPU

- Increasing the complexity of a single CPU leads to diminishing returns
  - Due to lack of instruction-level parallelism
  - Too many simultaneous accesses to one register file
  - Forwarding wires between functional units too long - inter-cluster communication takes >1 cycle

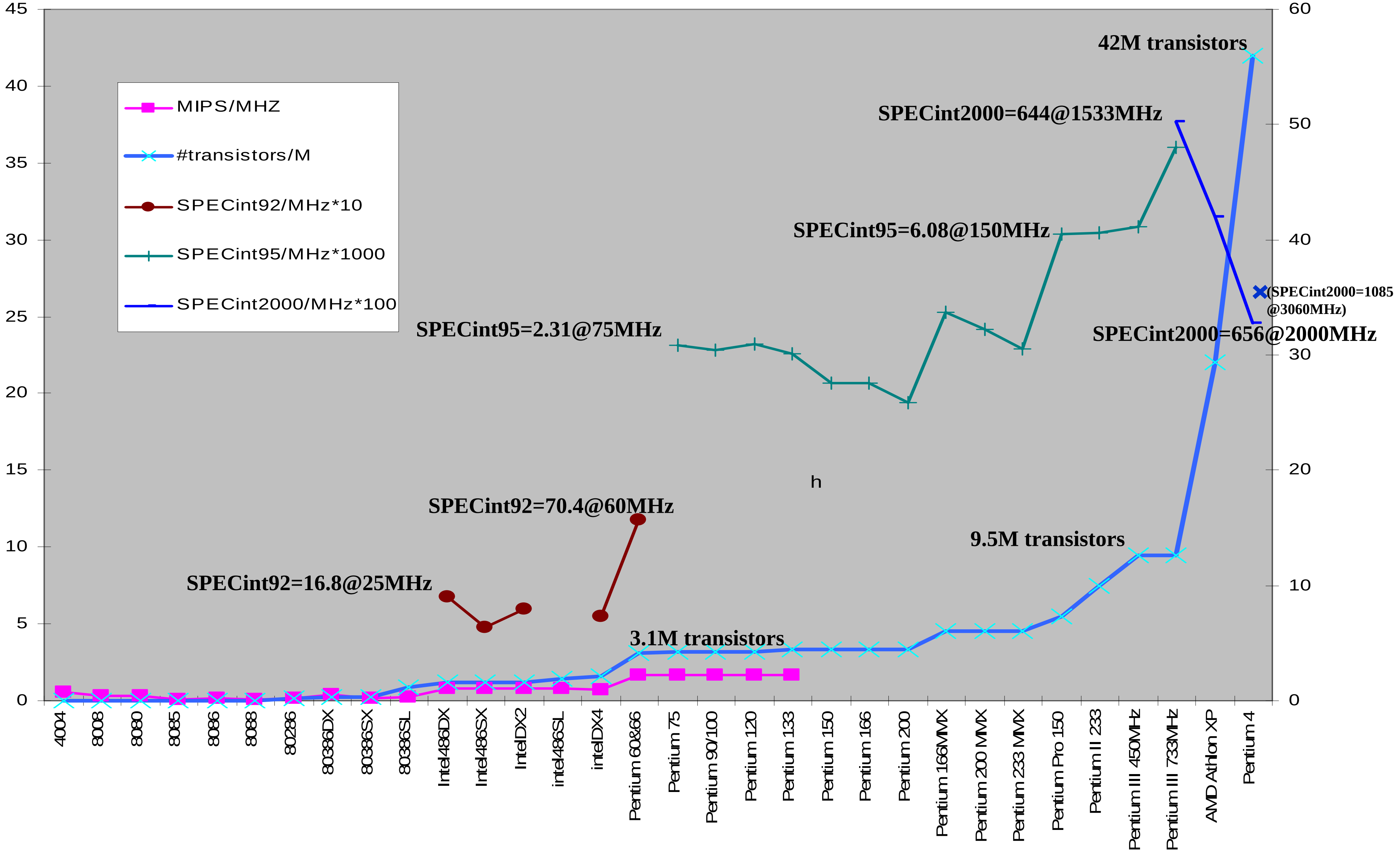
Architectural effectiveness of Intel processors



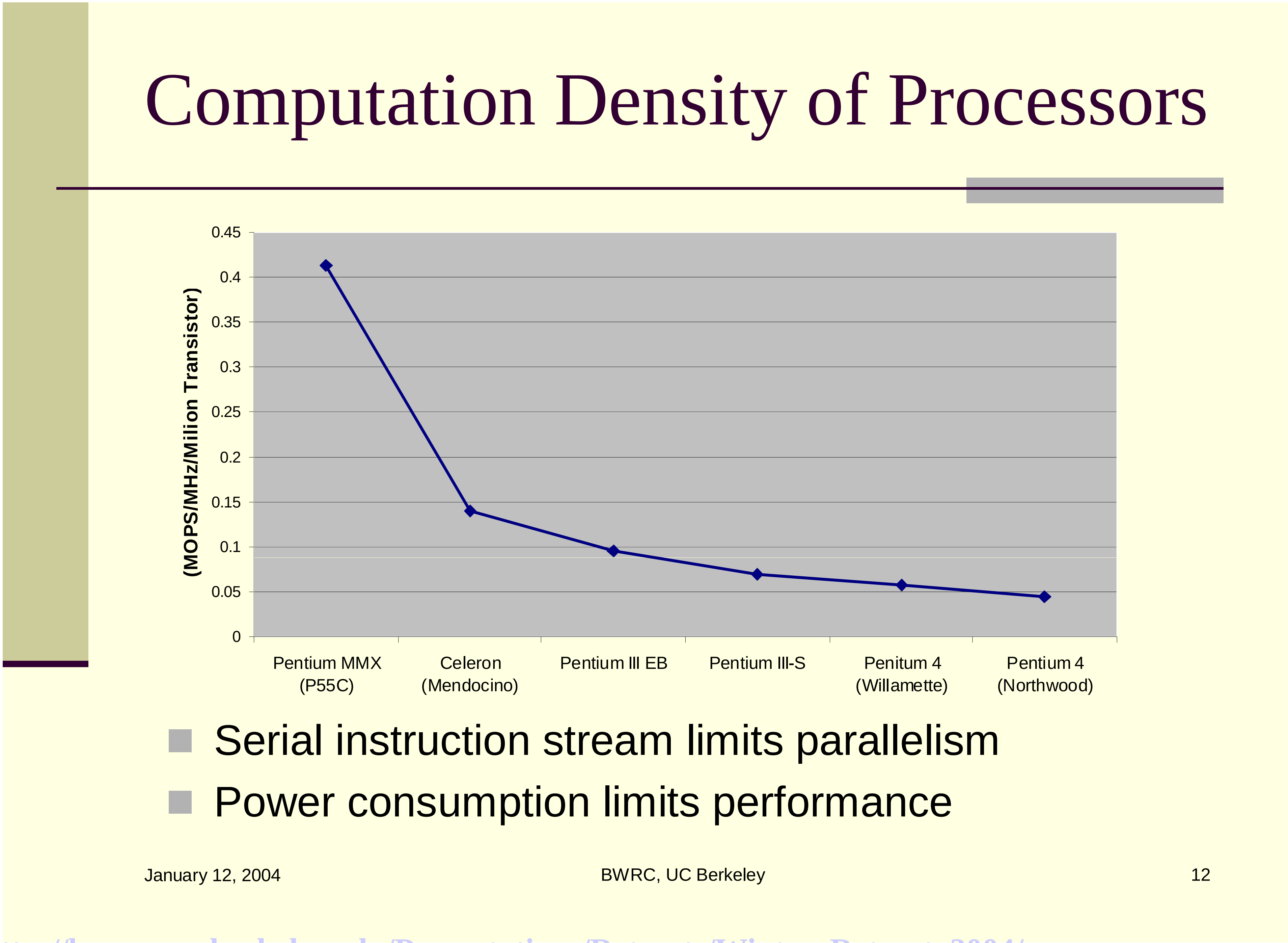
- Architectural effectiveness shows performance gained through architecture rather than clock rate
- Extra transistors largely devoted to cache, which of course is essential in allowing high clock rate

Source: http://www.galaxycentre.com/intelcpu.htm **and Intel**

Architectural effectiveness of Intel processors



Sources: http://www.galaxycentre.com/intelcpu.htm http://www.specbench.org/ www.sandpile.org **and Intel**



Computation Density of Processors

- Serial instruction stream limits parallelism
- Power consumption limits performance

January 12, 2004 BWRC, UC Berkeley 12

http://bwrc.eecs.berkeley.edu/Presentations/Retreats/Winter_Retreat_2004/

How to add another processor?

- Idea: instead of trying to exploit more instruction-level parallelism by building a bigger CPU, build two - or more
- This only makes sense if the application parallelism exists…
- Why might it be better?
  - No need for multiported register file
  - No need for long-range forwarding
  - CPUs can take independent control paths
  - Still need to synchronise and communicate
  - Program has to be structured appropriately…

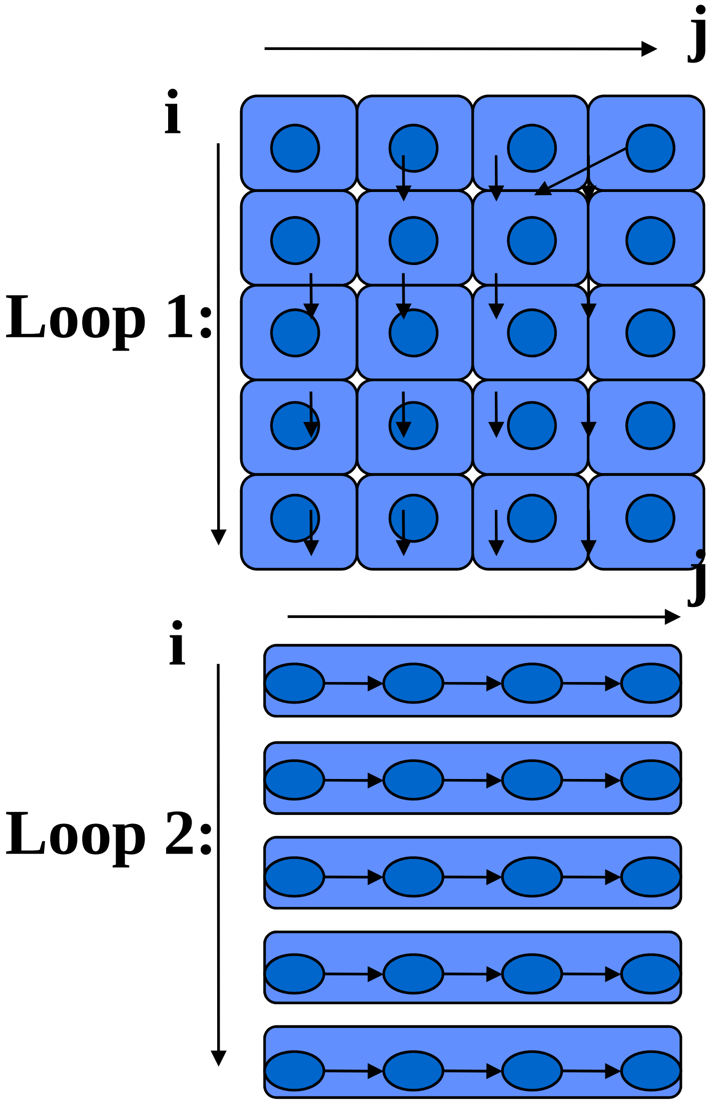
How to add another processor?

- How should the CPUs be connected?
- **Idea**: systems linked by network connected via I/O bus
  - Eg Fujitsu AP3000, Myrinet, Quadrics
- **Idea**: CPU/memory packages linked by network connecting main memory units
  - Eg SGI Origin
- **Idea**: CPUs share main memory
  - Eg Intel Xeon SMP
- **Idea**: CPUs share L2/L3 cache
  - Eg IBM Power4
- **Idea**: CPUs share L1 cache
- **Idea**: CPUs share registers, functional units
  - Cray/Tera MTA (multithreaded architecture), Symmetric multithreading (SMT), as in Hyperthreaded Pentium 4, Alpha 21464, etc

How to program a parallel computer?

- Shared memory makes parallel programming much easier:

```
for(i=0; I<N; ++i)
  par_for(j=0; j<M; ++j)
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; I<N; ++i)
  for(j=0; j<M; ++j)
    A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```



- First loop operates on rows in parallel
- Second loop operates on columns in parallel
- With distributed memory we would have to program message passing to transpose the array in between
- With shared memory… no problem!

Shared-memory parallel - OpenMP

- OpenMP is a standard design for language extensions for shared-memory parallel programming
- Language bindings exist for Fortran, C and to some extent (eg research prototypes) for C++, Java and C#
- Implementation requires compiler support

- Example:

```
for(i=0; I<N; ++i)
   #pragma omp parallel for
   for(j=0; j<M; ++j)
     A[i,j] = (A[i-1,j] + A[i,j])*0.5;
#pragma omp parallel for
par_for(i=0; I<N; ++i)
   for(j=0; j<M; ++j)
     A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

Implementing shared-memory parallel loop

```
for (i=0; i<N; i++) {
  C[i]  = A[i] + B[i];
}
```

```
if (myThreadId() == 0)
  i = 0;
barrier();
// on each thread
while (true) {
  local_i = FetchAndAdd(&i);
  if (local_i >= N) break;
  C[local_i] = 0.5*(A[local_i] + B[local_i]);
}
barrier();
```

Barrier(): block until all threads reach this point

- "self-scheduling" loop
- FetchAndAdd() is atomic operation to get next un-executed loop iteration:

```
Int FetchAndAdd(int *i) {
  lock(i);
  r = I;
  i = i+1;
  unlock(i);
  return(r);
}
```

Optimisations:

- Work in chunks
- Avoid unnecessary barriers
- Exploit "cache affinity" from loop to loop

There are smarter ways to implement FetchAndAdd….

More OpenMP

```
#pragma omp parallel for \
   default(shared) private(i) \
   schedule(static,chunk) \
   reduction(+:result)
for (i=0; i < n; i++)
   result = result + (a[i] * b[i]);
```

- **default(shared) private(i):**

  All variables except i are shared by all threads.

- **schedule(static,chunk):**

  Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the "team"

- **reduction(+:result):**

  performs a reduction on the variables that appear in its argument list

  - A private copy for each variable is created for each thread. At the end of the reduction, the reduction operator is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

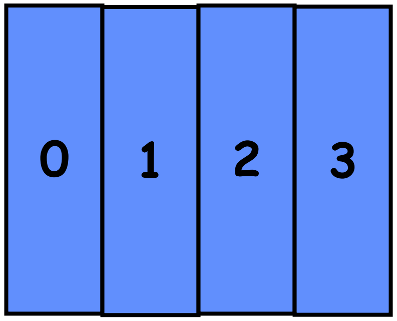
http://www.llnl.gov/computing/tutorials/openMP/#REDUCTION

Distributed-memory parallel - MPI

- MPI (“Message-passing Interface) is a standard API for parallel programming using message passing
- Usually implemented as library
- Six basic calls:
  - MPI_Init - Initialize MPI
  - MPI_Comm_size - Find out how many processes there are
  - MPI_Comm_rank - Find out which process I am
  - MPI_Send - Send a message
  - MPI_Recv - Receive a message
  - MPI_Finalize - Terminate MPI
- Key idea: collective operations
  - MPI_Bcast - broadcast data from the process with rank "root" to all other processes of the group.
  - MPI_Reduce – combine values on all processes into a single value using the operation defined by the parameter op.

MPI Example: initialisation

- SPMD
  - "Single Program, Multiple Data"
  - Each processor executes the program
  - First has to work out what part it is to play
  - "myrank" is index of this CPU
  - "comm" is MPI "communicator" – abstract index space of p processors
  - In this example, array is partitioned into slices

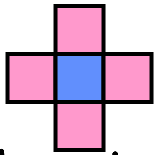


http://www.netlib.org/utk/papers/mpi-book/node51.html

```
! Compute number of processes and myrank
   CALL MPI_COMM_SIZE(comm, p, ierr)
   CALL MPI_COMM_RANK(comm, myrank, ierr)
! compute size of local block
   m = n/p
   IF (myrank.LT.(n-p*m)) THEN
            m = m+1
   END IF
! Compute neighbors
   IF (myrank.EQ.0) THEN
            left = MPI_PROC_NULL
   ELSE left = myrank - 1
   END IF
   IF (myrank.EQ.p-1)THEN
            right = MPI_PROC_NULL
   ELSE right = myrank+1
   END IF
! Allocate local arrays
   ALLOCATE (A(0:n+1,0:m+1), B(n,m))
```

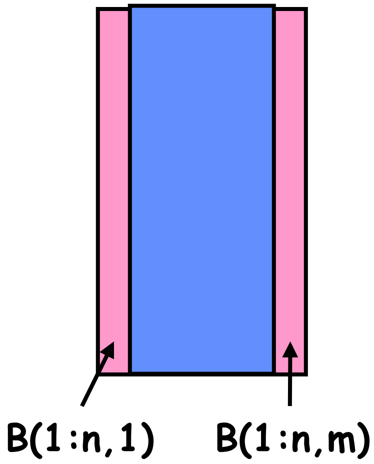
Example: Jacobi2D

- Sweep over A computing moving average of neighbouring four elements



- Compute new array B from A, then copy it back into B
- This version tries to overlap communication with computation

```
!Main Loop
DO WHILE(.NOT.converged)
    ! compute boundary iterations so they're ready to be sent right away
    DO i=1, n
        B(i,1)=0.25*(A(i-1,j)+A(i+1,j)+A(i,0)+A(i,2))
        B(i,m)=0.25*(A(i-1,m)+A(i+1,m)+A(i,m-1)+A(i,m+1))
    END DO
    ! Communicate
    CALL MPI_ISEND(B(1,1),n, MPI_REAL, left, tag, comm, req(1), ierr)
    CALL MPI_ISEND(B(1,m),n, MPI_REAL, right, tag, comm, req(2), ierr)
    CALL MPI_IRECV(A(1,0),n, MPI_REAL, left, tag, comm, req(3), ierr)
    CALL MPI_IRECV(A(1,m+1),n, MPI_REAL, right, tag, comm, req(4), ierr)
    ! Compute interior
    DO j=2, m-1
        DO i=1, n
            B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
        END DO
    END DO
    DO j=1, m
        DO i=1, n
            A(i,j) = B(i,j)
        END DO
    END DO
    ! Complete communication
    DO i=1, 4
        CALL MPI_WAIT(req(i), status(1.i), ierr)
    END DO
END DO
```

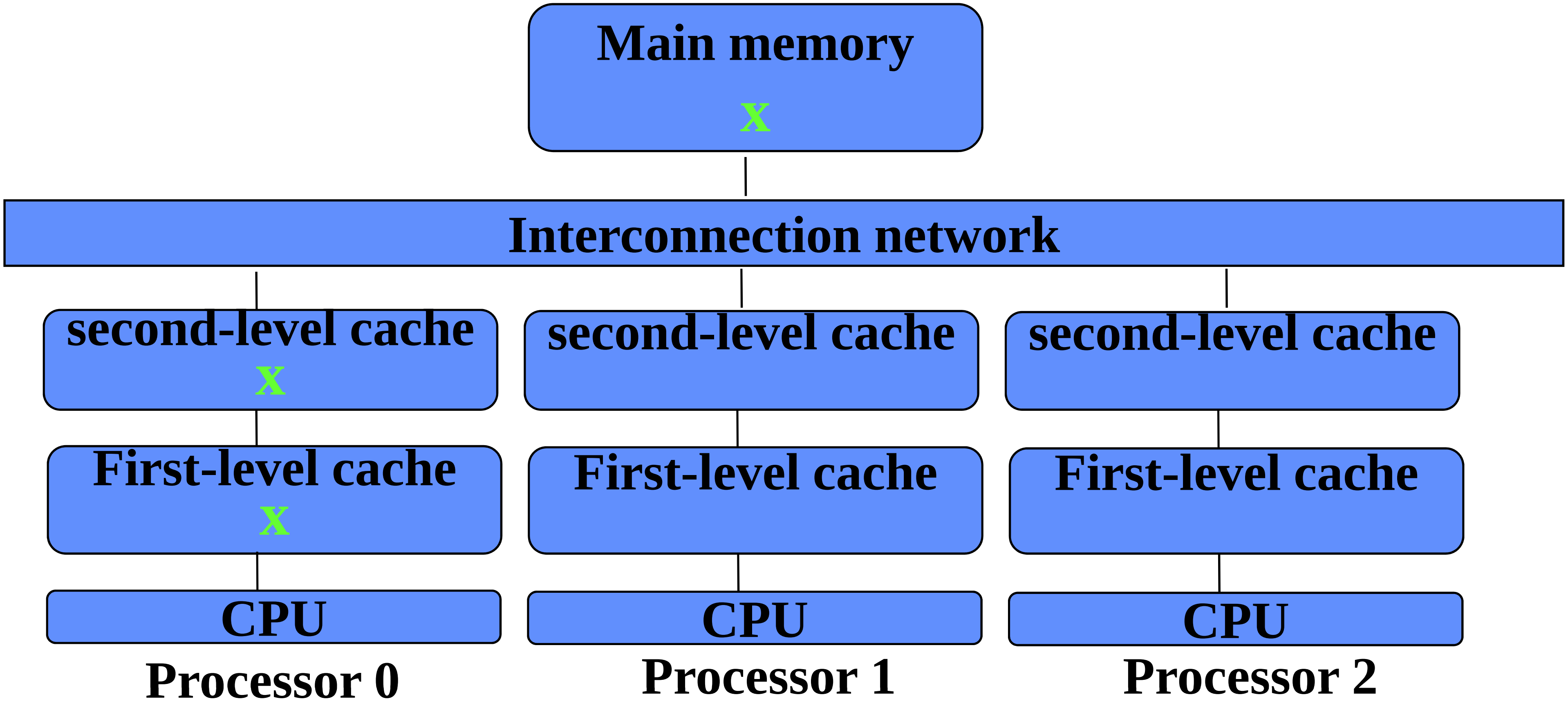


http://www.netlib.org/utk/papers/mpi-book/node51.html

How to connect processors...

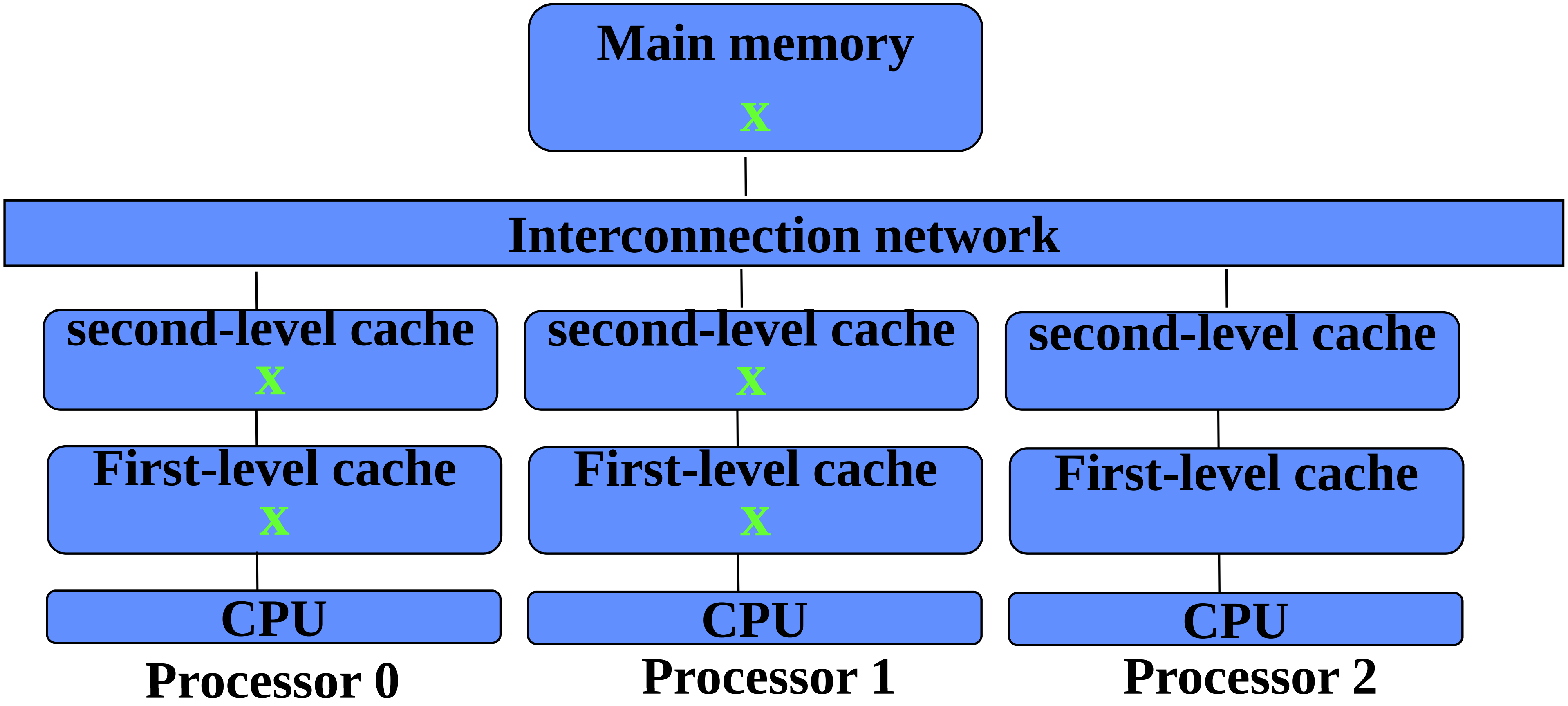
- Tradeoffs:
  - close coupling to minimise delays incurred when processors interact
  - separation to avoid contention for shared resources
- Result:
  - spectrum of alternative approaches based on application requirements, cost, and packaging/integration issues
- Currently:
  - just possible to integrate 2 full-scale CPUs on one chip together with large shared L2 cache
  - common to link multiple CPUs on same motherboard with shared bus connecting to main memory
  - more aggressive designs use richer interconnection network, perhaps with cache-to-cache transfer capability

Multiple caches… and trouble



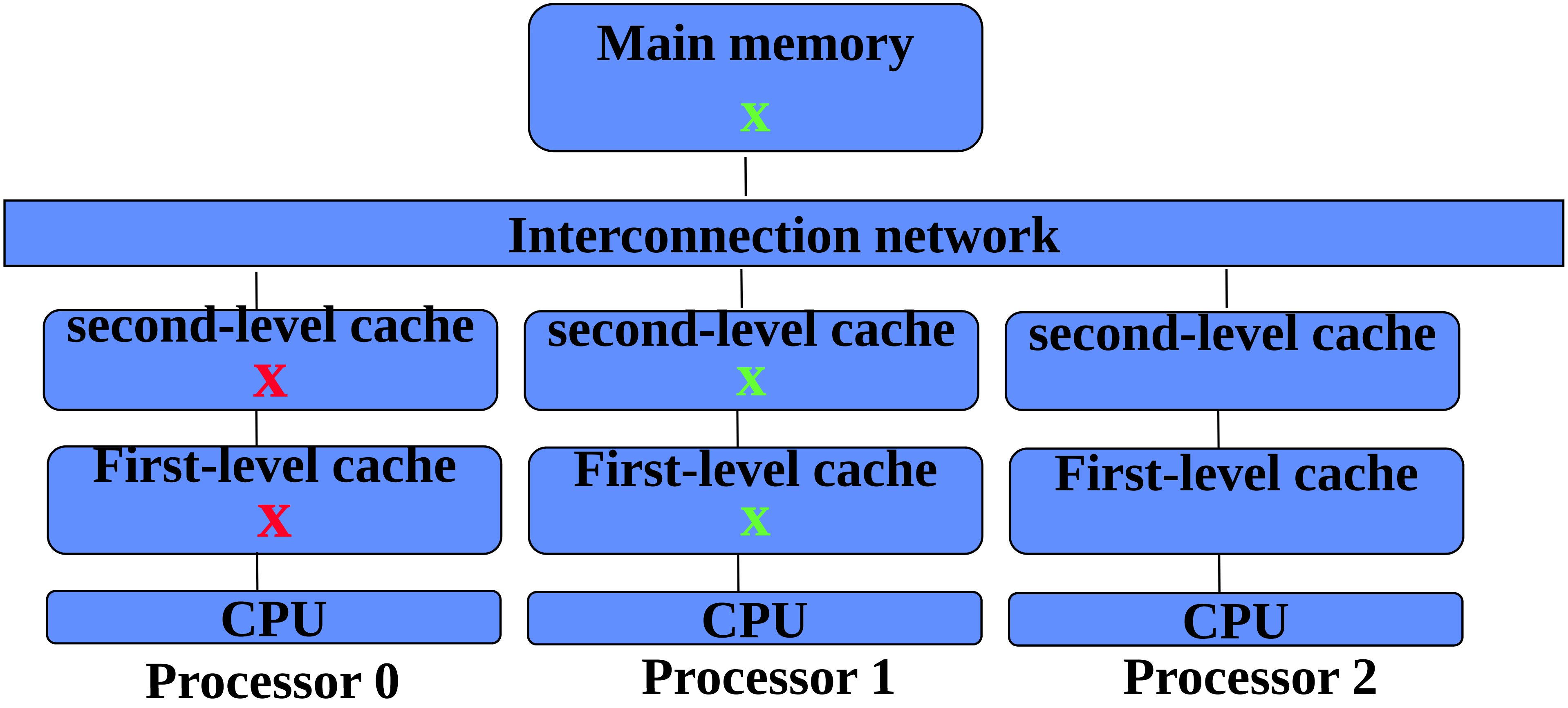
- Suppose processor 0 loads memory location x
- x is fetched from main memory and allocated into processor 0's cache(s)

Multiple caches… and trouble



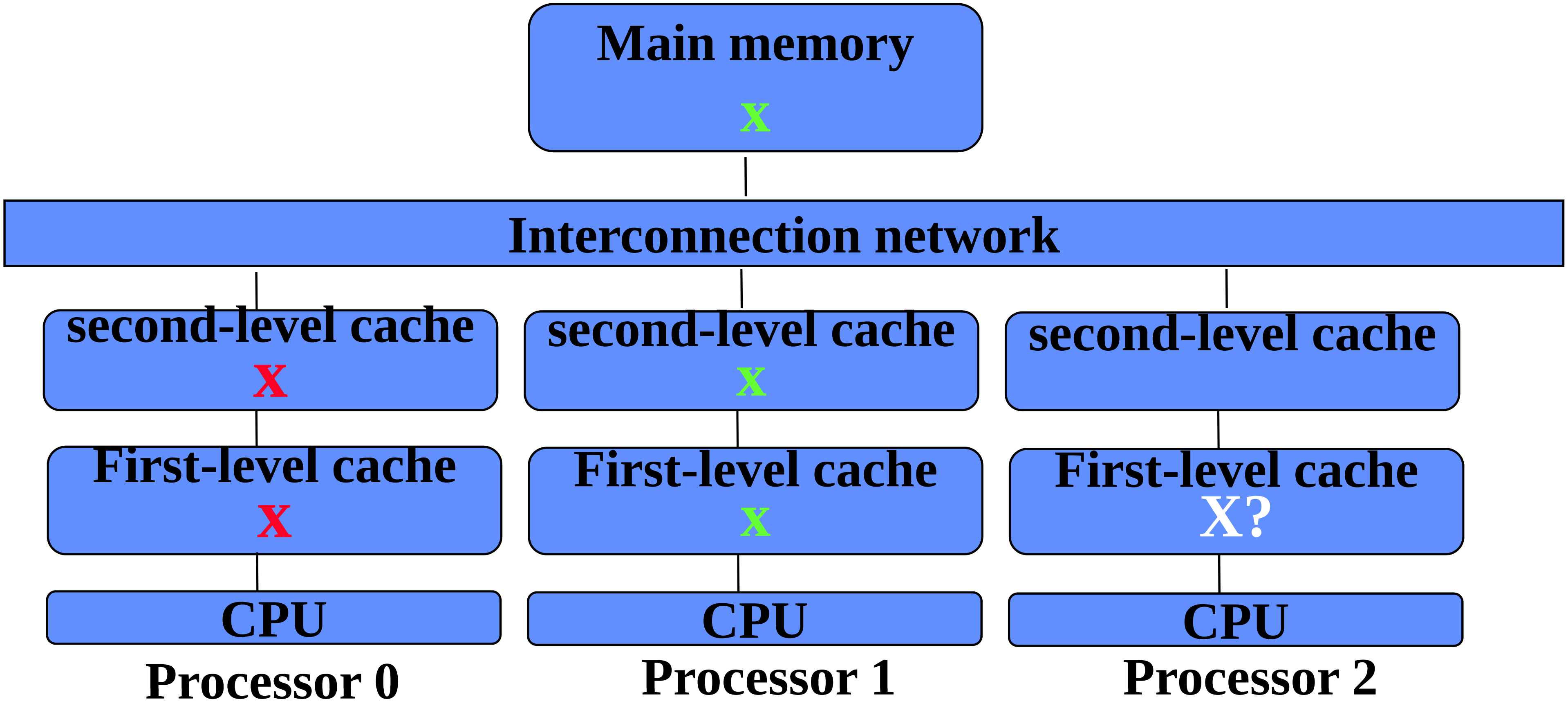
- Suppose processor 1 loads memory location x
- x is fetched from main memory and allocated into processor 1's cache(s) as well

Multiple caches… and trouble



- Suppose processor 0 stores to memory location x
- Processor 0's cached copy of x is updated
- Processor 1 continues to used the old value of x

Multiple caches… and trouble



- Suppose processor 2 loads memory location **x**
- How does it know whether to get x from main memory, processor 0 or processor 1?

Implementing distributed, shared memory

- Two issues:

1. How do you know where to find the latest version of the cache line?
2. How do you know when you can use your cached copy – and when you have to look for a more up-to-date version?

We will find answers to this after first thinking about what a distributed shared memory implementation is supposed to do…

Cache consistency (aka cache coherency)

- Goal (?):
  - "Processors should not continue to use out-of-date data indefinitely"
- Goal (?):
  - "Every load instruction should yield the result of the most recent store to that address"
- Goal (?): (definition: **Sequential Consistency**)
  - "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"

(Leslie Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs" (IEEE Trans Computers Vol.C-28(9) Sept 1979)

Implementing Strong Consistency: update

- Idea #1: when a store to address x occurs, **update** all the remote cached copies
- To do this we need either:
  - To broadcast every store to every remote cache
  - Or to keep a list of which remote caches hold the cache line
  - Or at least keep a note of whether there are *any* remote cached copies of this line
- But first…how well does this update idea work?

Implementing Strong Consistency: update…

Problems with update

1. What about if the cache line is several words long?
   - Each update to each word in the line leads to a broadcast
2. What about old data which other processors are no longer interested in?
   - We'll keep broadcasting updates indefinitely…
   - Do we really have to broadcast *every* store?
   - It would be nice to know that we have exclusive access to the cacheline so we don't have to broadcast updates…

A more cunning plan… invalidation

- Suppose instead of ***updating*** remote cache lines, we *invalidate* them all when a store occurs?
- After the first write to a cache line we know there are no remote copies – so subsequent writes don't lead to communication
- Is invalidate always better than update?
  - Often
  - But not if the other processors really need the new data as soon as possible
- Note that to exploit this, we need a bit on each cache line indicating its sharing state
- (analogous to write-back vs write-through caches)

Update vs invalidate

Update:

- May reduce latency of subsequent remote reads
- if any are made

Invalidate:

- May reduce network traffic
- e.g. if same CPU writes to the line before remote nodes take copies of it

The “Berkeley" Protocol

Four cache line states:

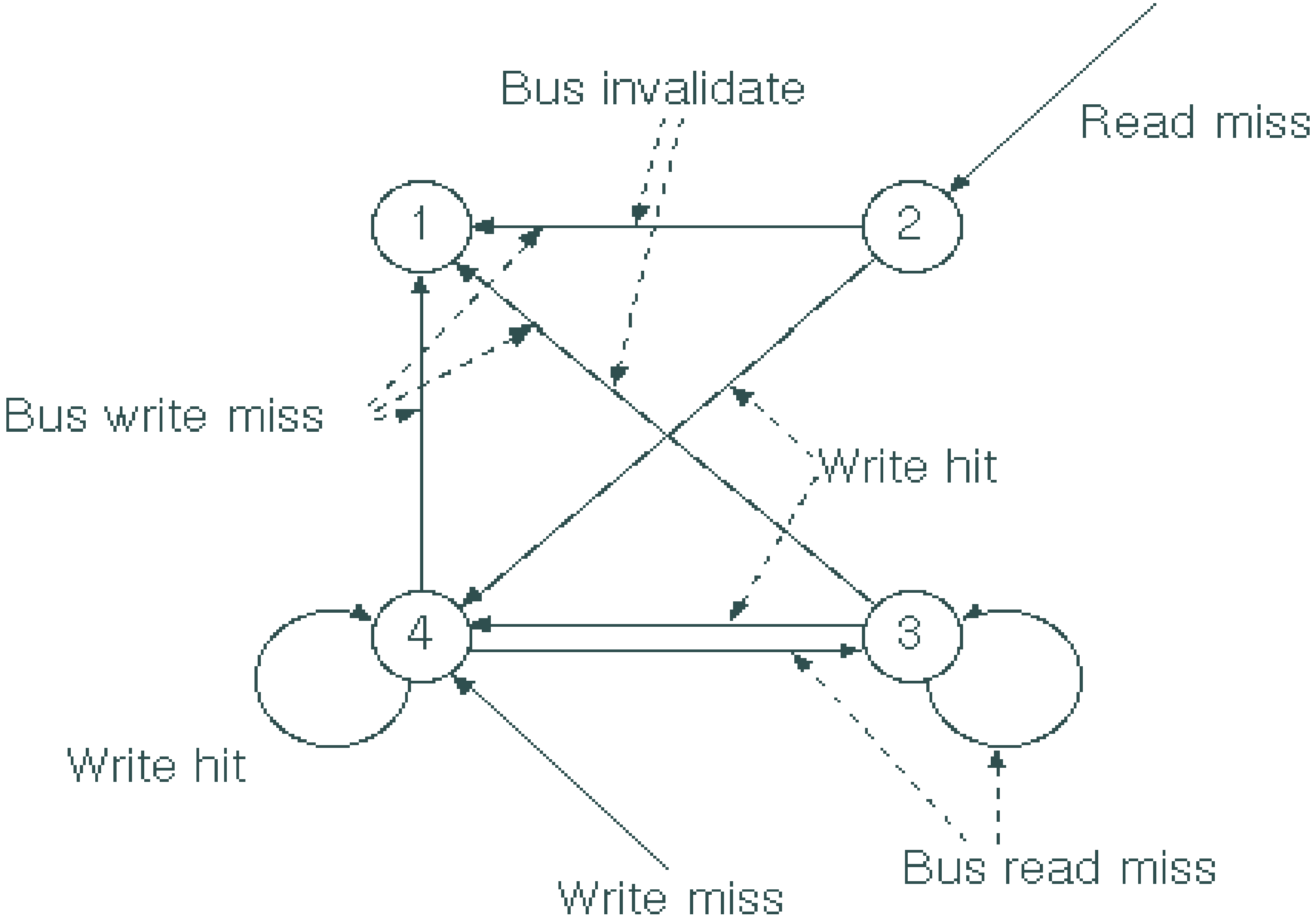
Broadcast invalidations on bus unless cache line is exclusively “owned” (DIRTY)

- **INVALID**
- **VALID : clean, potentially shared, unowned**
- **SHARED-DIRTY : modified, possibly shared, owned**
- **DIRTY : modified, only copy, owned**

- **Read miss:**
  - **If another cache has the line in SHARED-DIRTY or DIRTY,**
    - **it is supplied**
    - **changing state to SHARED-DIRTY**
  - **Otherwise**
    - **the line comes from memory. The state of the**
    - **line is set to VALID**

- **Write hit:**
  - **No action if line is DIRTY**
  - **If VALID or SHARED-DIRTY,**
    - **an invalidation is sent, and**
    - **the local state set to DIRTY**
- **Write miss:**
  - **Line comes from owner (as with read miss).**
  - **All other copies set to INVALID, and line in requesting cache is set to DIRTY**

Berkeley cache coherence protocol: state transition diagram



1. INVALID
2. VALID: clean, potentially shared, unowned
3. SHARED-DIRTY: modified, possibly shared, owned
4. DIRTY: modified, only copy, owned

The Berkeley protocol is representative of how typical bus-based SMPs work

Q: What has to happen on a "Bus read miss"?

The job of the cache controller - snooping

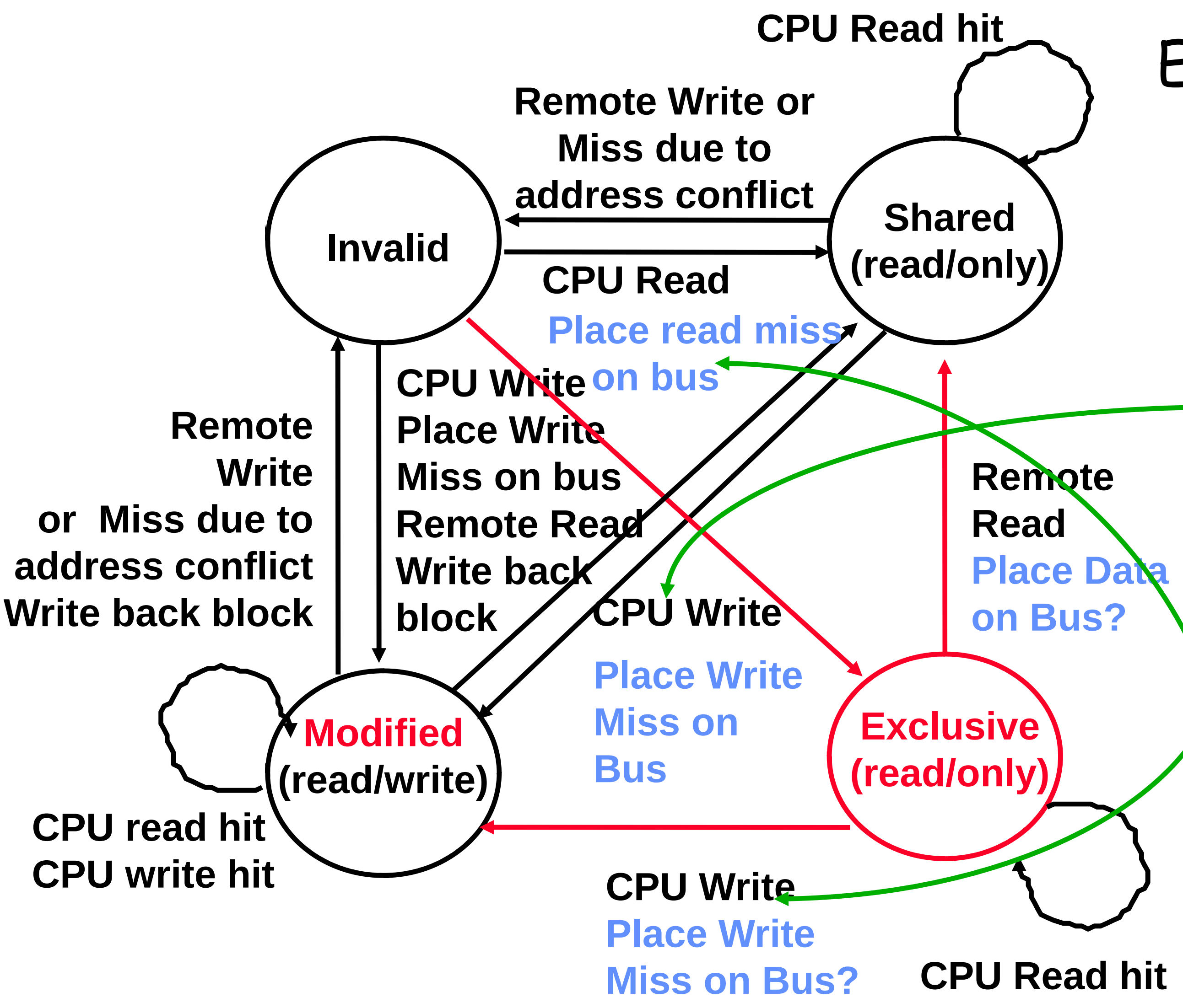
- The protocol state transitions are implemented by the cache controller – which "snoops" all the bus traffic
- Transitions are triggered either by
  - the bus (Bus invalidate, Bus write miss, Bus read miss)
  - The CPU (Read hit, Read miss, Write hit, Write miss)

- For every bus transaction, it looks up the directory (cache line state) information for the specified address
  - If this processor holds the only valid data (DIRTY), it responds to a "Bus read miss" by providing the data to the requesting CPU
  - If the memory copy is out of date, one of the CPUs will have the cache line in the SHARED-DIRTY state (because it updated it last) – so must provide data to requesting CPU
  - State transition diagram doesn't show what happens when a cache line is displaced…

Berkeley protocol - summary

- Invalidate is usually better than update
- Cache line state "DIRTY" bit records whether remote copies exist
  - If so, remote copies are invalidated by broadcasting message on bus – cache controllers snoop all traffic
- Where to get the up-to-date data from?
  - Broadcast read miss request on the bus
  - If this CPUs copy is DIRTY, it responds
  - If no cache copies exist, main memory responds
  - If several copies exist, the CPU which holds it in "SHARED-DIRTY" state responds
  - If a SHARED-DIRTY cache line is displaced, … need a plan
- How well does it work?
  - See extensive analysis in Hennessy and Patterson

Snoop Cache Extensions



Extensions:

- Fourth State: Ownership
- Shared-> Modified, need invalidate only (upgrade request), don't read memory
  Berkeley Protocol
- Clean exclusive state (no miss for private data on write)
  MESI Protocol
- Cache supplies data when shared state (no memory access)
  Illinois Protocol

Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
	Owned Exclusive	Private Dirty	Modified (private,!=Memory)
Exclusive	Owned Shared	Private Clean	eXclusive (private,=Memory)
Shared	Shared	Shared	Shared (shared,=Memory)
Invalid	Invalid	Invalid	Invalid

Owner can update via bus invalidate operation
Owner must write back when replaced in cache

If read sourced from memory, then Private Clean
If read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

Implementation Complications

- Write Races:
  - Cannot update cache until bus is obtained
    - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then 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 grab block in the Exclusive state
    - Must track and prevent multiple misses for one block
- Must support interventions and invalidations

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU just to check:
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
    - block size, associativity of L2 affects L1

Implementing Snooping Caches

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write-back cache, may have the desired copy and it’s dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)

Large-Scale Shared-Memory Multiprocessors

Bus inevitably becomes a bottleneck when many processors are used

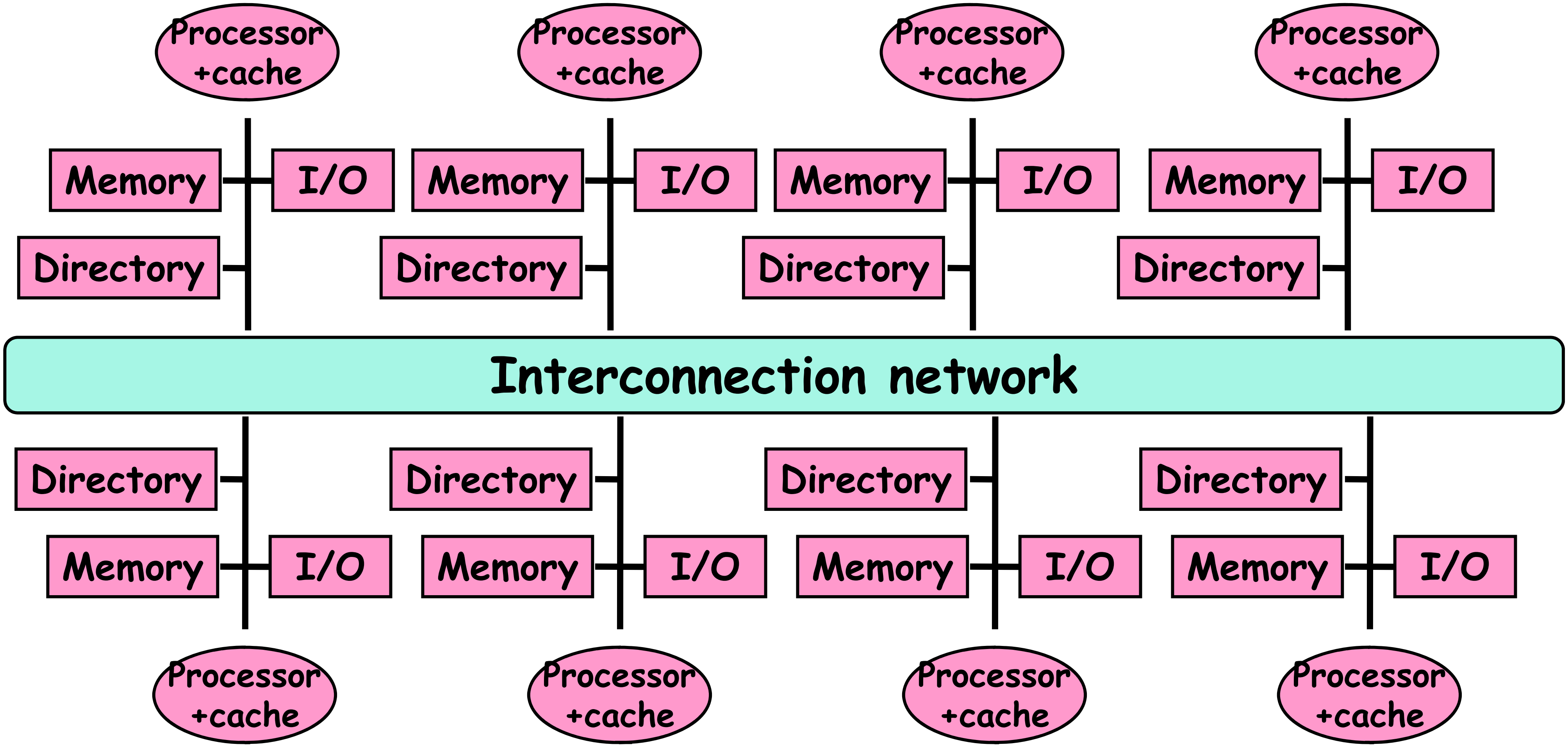
- Use a more general interconnection network
- So snooping does not work

- DRAM memory is also distributed
  - 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?
- As a case study, we will look at S3.MP (Sun's Scalable Shared memory Multi-Processor, a CC-NUMA (cache-coherent non-uniform memory access) architecture

Larger MPs

- Separate Memory per Processor
- Local or Remote access via memory controller
- 1 Cache Coherency solution: non-cached pages
- Alternative: 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)
- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which Procs have copies of their blocks

Distributed Directory MPs



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 (usually bit vector, 1 if processor has copy)
- Keep it simple(r):
  - Writes to non-exclusive data
    => write miss
  - Processor blocks until access completes
  - Assume messages received
    and acted upon in order sent

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

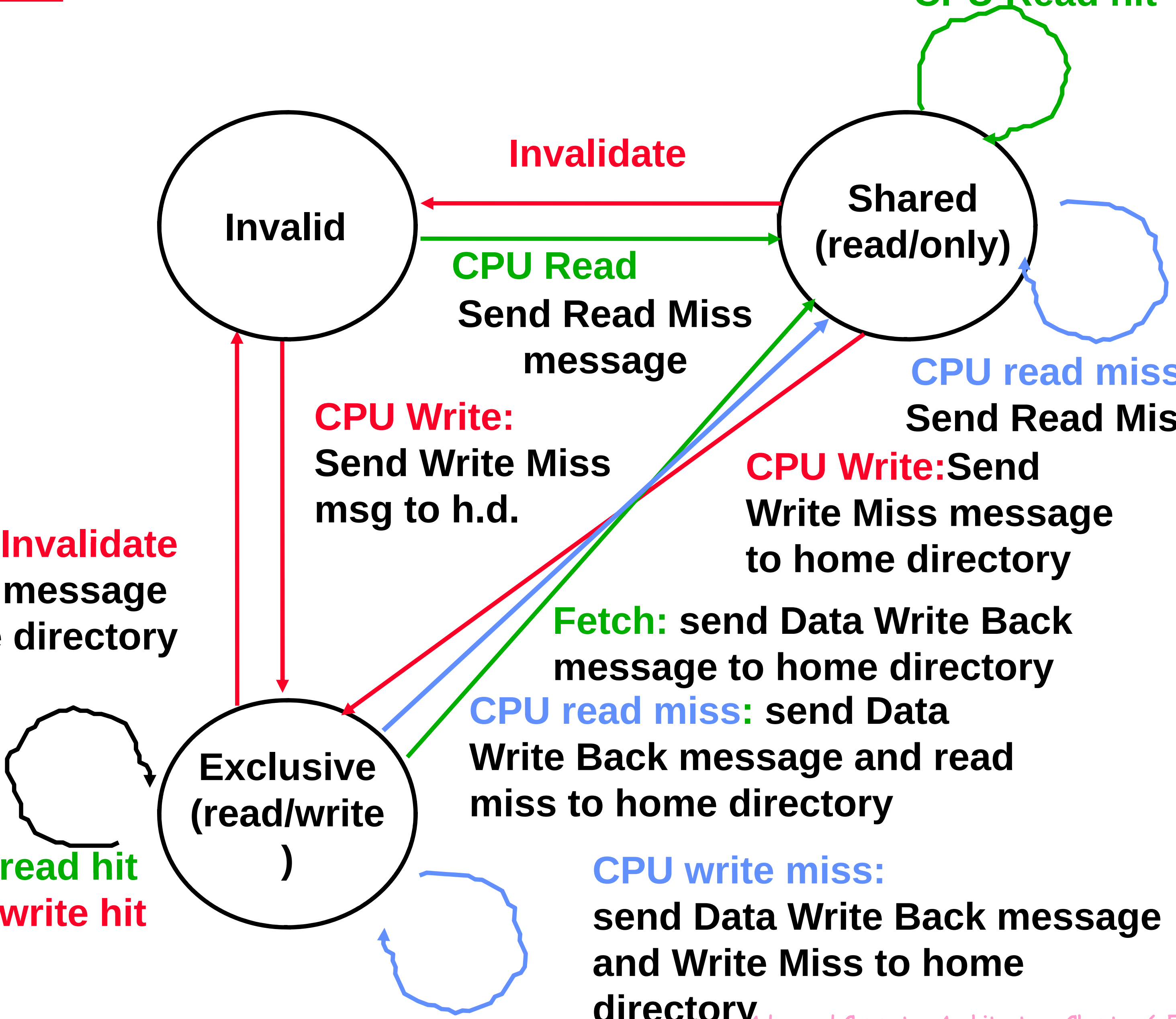
<table>
<tr><th><em>Message type</em></th><th><em>Source</em></th><th><em>Destination</em></th><th><em>Msg Content</em></th></tr>
<tr><td>Read miss</td><td>Local cache</td><td>Home directory</td><td>P, A</td></tr>
<tr><td colspan="4">➧ <em>Processor P reads data at address A;
make P a read sharer and arrange to send data back</em></td></tr>
<tr><td>Write miss</td><td>Local cache</td><td>Home directory</td><td>P, A</td></tr>
<tr><td colspan="4">➧ <em>Processor P writes data at address A;
make P the exclusive owner and arrange to send data back</em></td></tr>
<tr><td>Invalidate</td><td>Home directory</td><td>Remote caches</td><td>A</td></tr>
<tr><td colspan="4">➧ <em>Invalidate a shared copy at address A.</em></td></tr>
<tr><td>Fetch</td><td>Home directory</td><td>Remote cache</td><td>A</td></tr>
<tr><td colspan="4">➧ <em>Fetch the block at address A and send it to its home directory</em></td></tr>
<tr><td>Fetch/Invalidate</td><td>Home directory</td><td>Remote cache</td><td>A</td></tr>
<tr><td colspan="4">➧ <em>Fetch the block at address A and send it to its home directory; invalidate the block in the cache</em></td></tr>
<tr><td>Data value reply</td><td>Home directory</td><td>Local cache</td><td>Data</td></tr>
<tr><td colspan="4">➧ <em>Return a data value from the home memory (read miss response)</em></td></tr>
<tr><td>Data write-back</td><td>Remote cache</td><td>Home directory</td><td>A, Data</td></tr>
<tr><td colspan="4">➧ <em>Write-back a data value for address A (invalidate response)</em></td></tr>
</table>

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



CPU Read hit

Invalid

Shared (read/only)

Invalidate

CPU Read

Send Read Miss message

CPU read miss: Send Read Miss

CPU Write: Send Write Miss msg to h.d.

CPU Write:Send Write Miss message to home directory

Fetch/Invalidate send Data Write Back message to home directory

Fetch: send Data Write Back message to home directory

CPU read miss: send Data Write Back message and read miss to home directory

Exclusive (read/write)

CPU read hit

CPU write hit

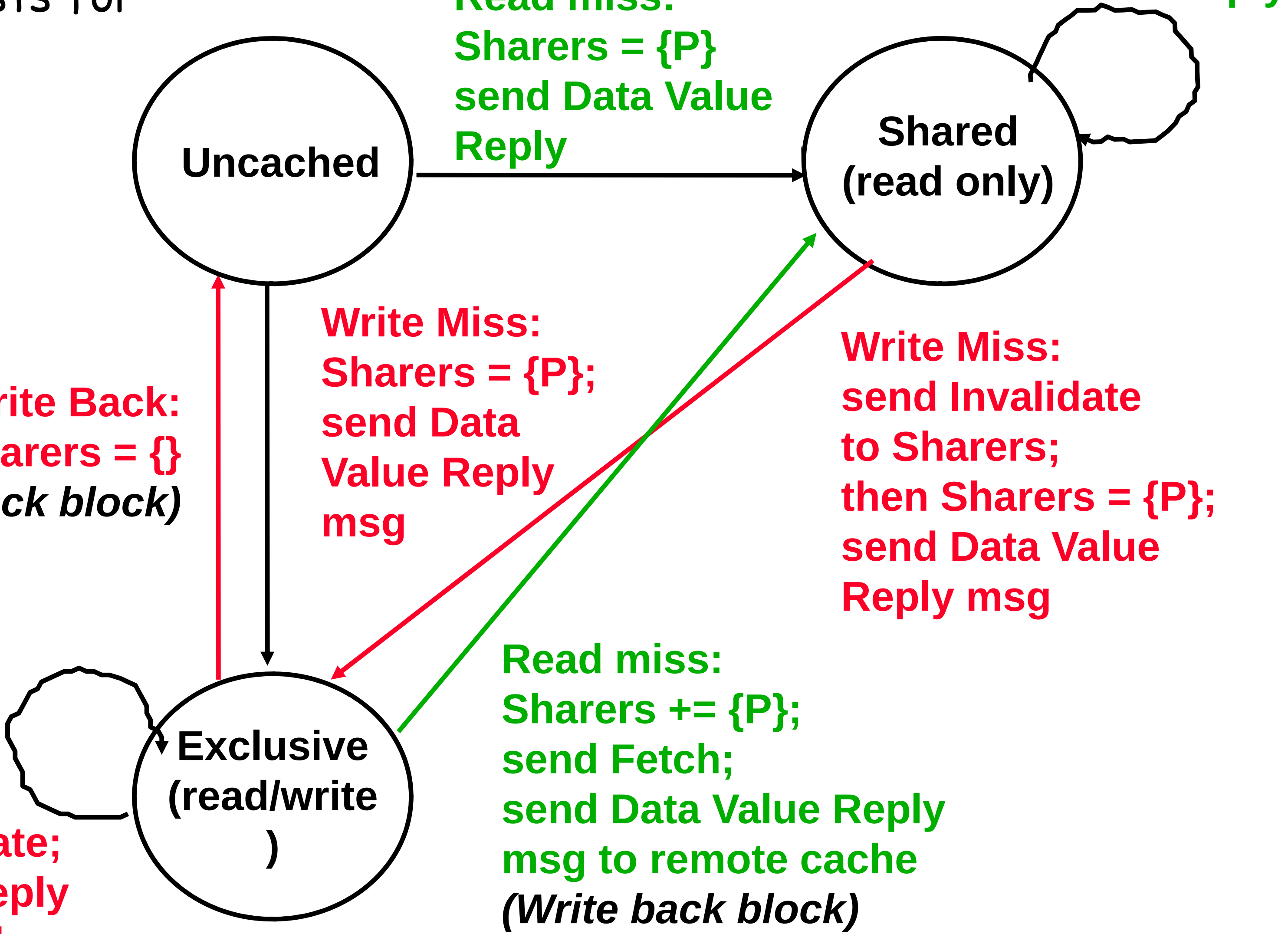
CPU write miss: send Data Write Back message and Write Miss to home directory

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 statisfy 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



Read miss:
Sharers = {P}
send Data Value Reply

Read miss:
Sharers += {P};
send Data Value Reply

Write Miss:
Sharers = {P};
send Data Value Reply msg

Data Write Back:
Sharers = {}
(Write back block)

Write Miss:
send Invalidate to Sharers;
then Sharers = {P};
send Data Value Reply msg

Read miss:
Sharers += {P};
send Fetch;
send Data Value Reply msg to remote cache
(Write back block)

Write Miss:
Sharers = {P};
send Fetch/Invalidate;
send Data Value Reply msg to remote cache

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			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														
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														10
														10
P2: Write 40 to A2														10

Write Back

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														10

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							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

Synchronization

- Why Synchronize? Need to know when it is safe for different processes to use shared data

- Issues for Synchronization:
  - Uninterruptable instruction to fetch and update memory (atomic operation);
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptable Instruction to Fetch and Update Memory

- Atomic exchange: interchange a value in a register for a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    - 0 if you succeeded in setting the lock (you were first)
    - 1 if other processor had already claimed access
  - Key is that exchange operation is indivisible
- Test-and-set: tests a value and sets it if the value passes the test
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - 0 => synchronization variable is free

Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceeding load) and 0 otherwise
- Example doing atomic swap with LL & SC:

```
try:   mov      R3,R4              ; mov exchange value
       ll       R2,0(R1); load linked
       sc       R3,0(R1); store conditional
       beqz     R3,try    ; branch store fails (R3 = 0)
       mov      R4,R2              ; put load value in R4
```

- Example doing fetch & increment with LL & SC:

```
try:   ll       R2,0(R1); load linked
       addi     R2,R2,#1           ; increment (OK if reg-reg)
       sc       R2,0(R1)           ; store conditional
       beqz     R2,try    ; branch store fails (R2 = 0)
```

A
l
p
h
a
,

A

R

User Level Synchronization—Operation Using this Primitive

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock
-
```
                li      R2,#1
lockit:         exch    R2,0(R1)         ;atomic exchange
                bnez    R2,lockit        ;already locked?
```

- What about MP with cache coherency?
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables
- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic
- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:            li      R2,#1
lockit:lw       R3,0(R1) ;load var
                bnez    R3,lockit        ;not free=>spin
                exch    R2,0(R1) ;atomic exchange
                bnez    R2,try           ;already locked?
```

Another MP Issue: Memory Consistency Models

- What is consistency? When must a processor see the new value? e.g., seems that

```
P1:     A = 0;                    P2:     B = 0;
        .....                             .....
        A = 1;                            B = 1;
L1:     if (B == 0) ...           L2:     if (A == 0) ...
```

- Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?
- Memory consistency models:
  what are the rules for such cases?
- Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
  - SC: delay all memory accesses until all invalidates done

Memory Consistency Model

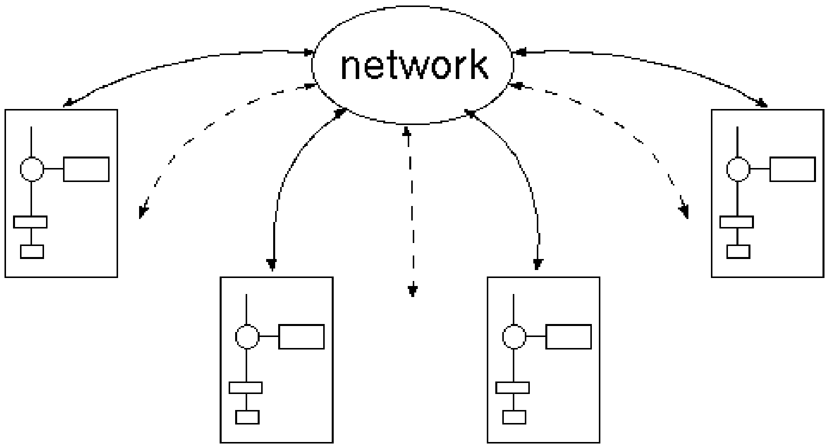
- Weak consistency schemes offer faster execution than sequential consistency
- Several processors provide fence instructions to enforce sequential consistency when an instruction stream passes such a point. Expensive!
- Not really an issue for most programs;
  they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations

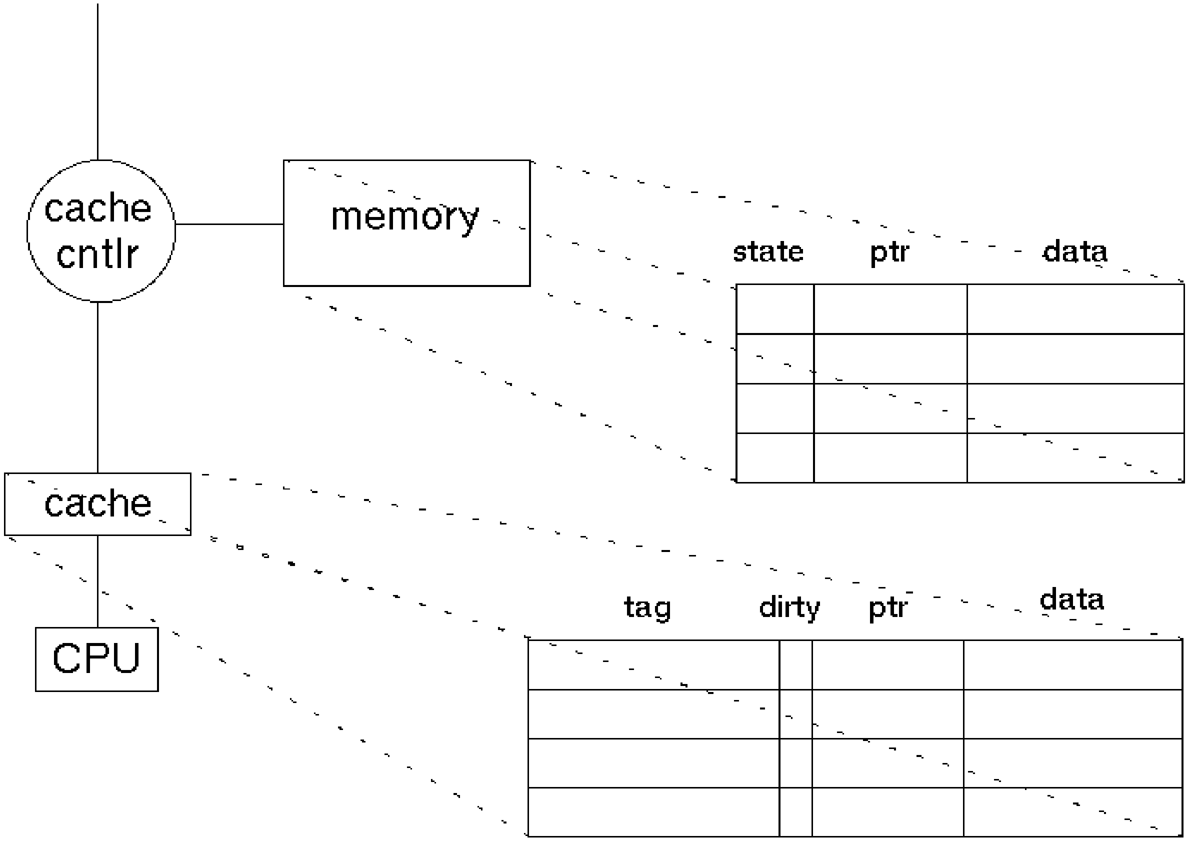
    write (x)
    ...
    release (s) *{unlock}*
    ...
    acquire (s) *{lock}*
    ...
    read(x)
- Only those programs willing to be nondeterministic are not synchronized: "data race": outcome f(proc. speed)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW
  to different addresses

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



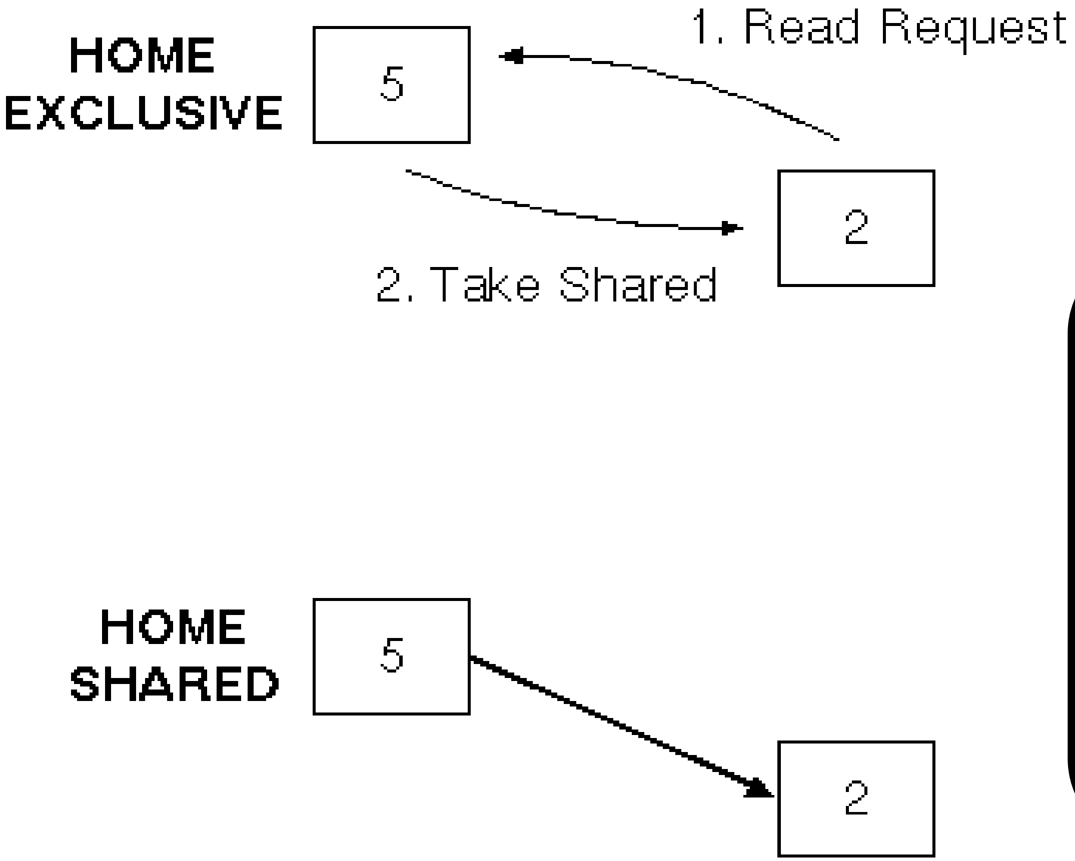


Protocol Basics

- 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

S3MP: Read Requests

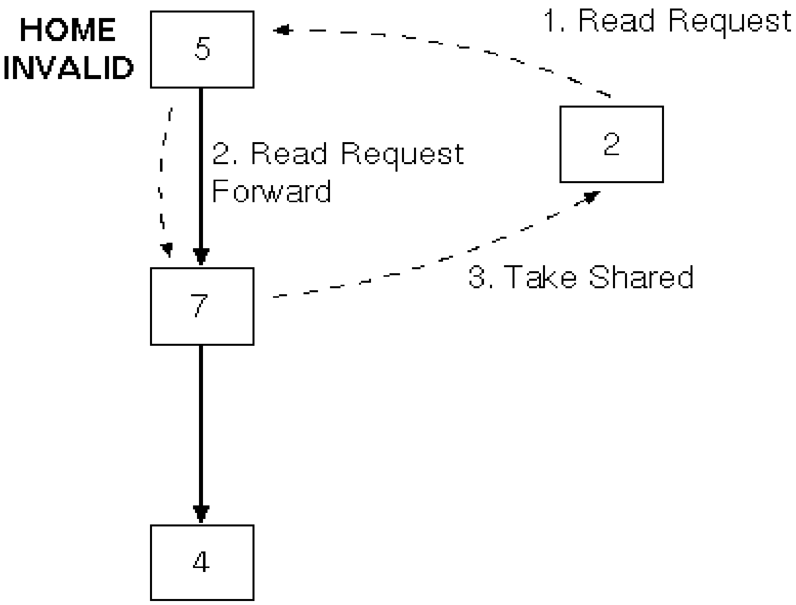
- Simple case: initially only the home has the data:



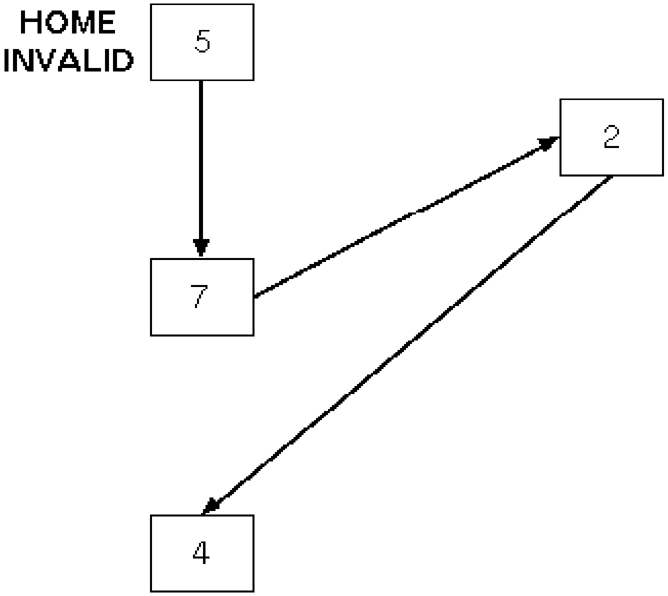
Curved arrows show messages, bold straight arrows show pointers

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

S3MP: Read Requests - remote

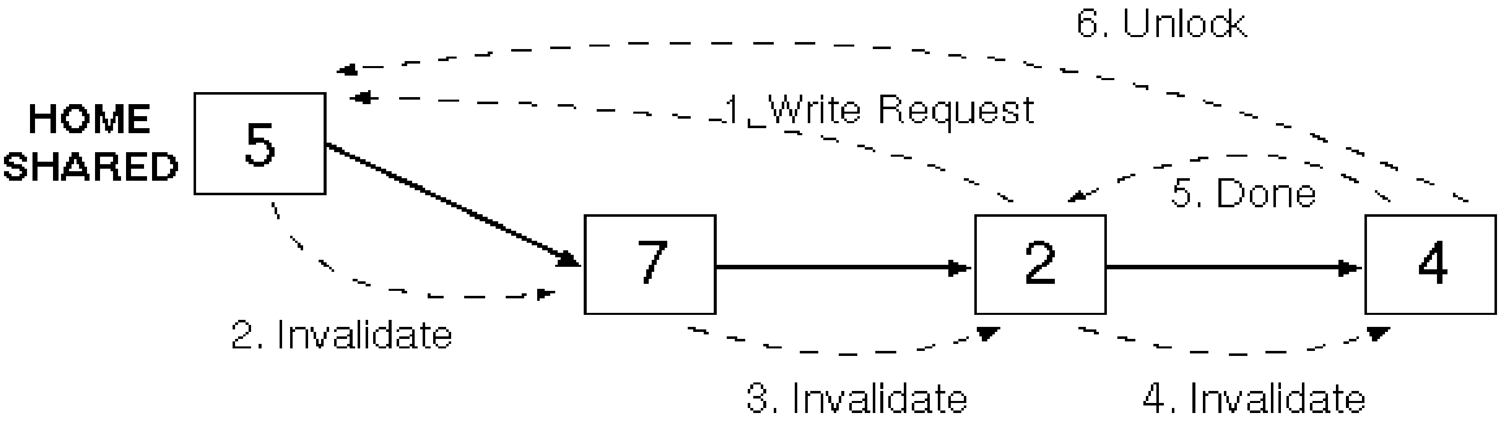


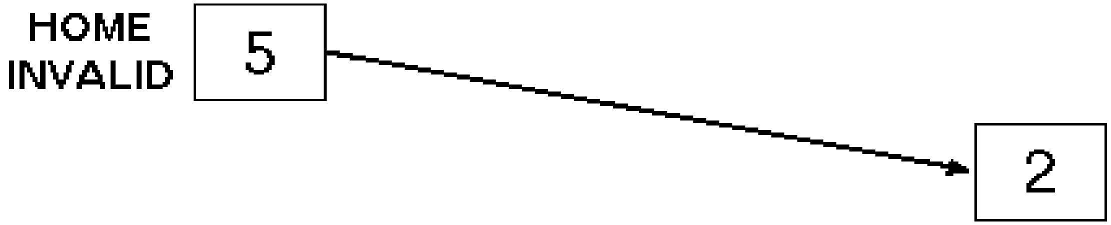
- More interesting case: some other processor has the data



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

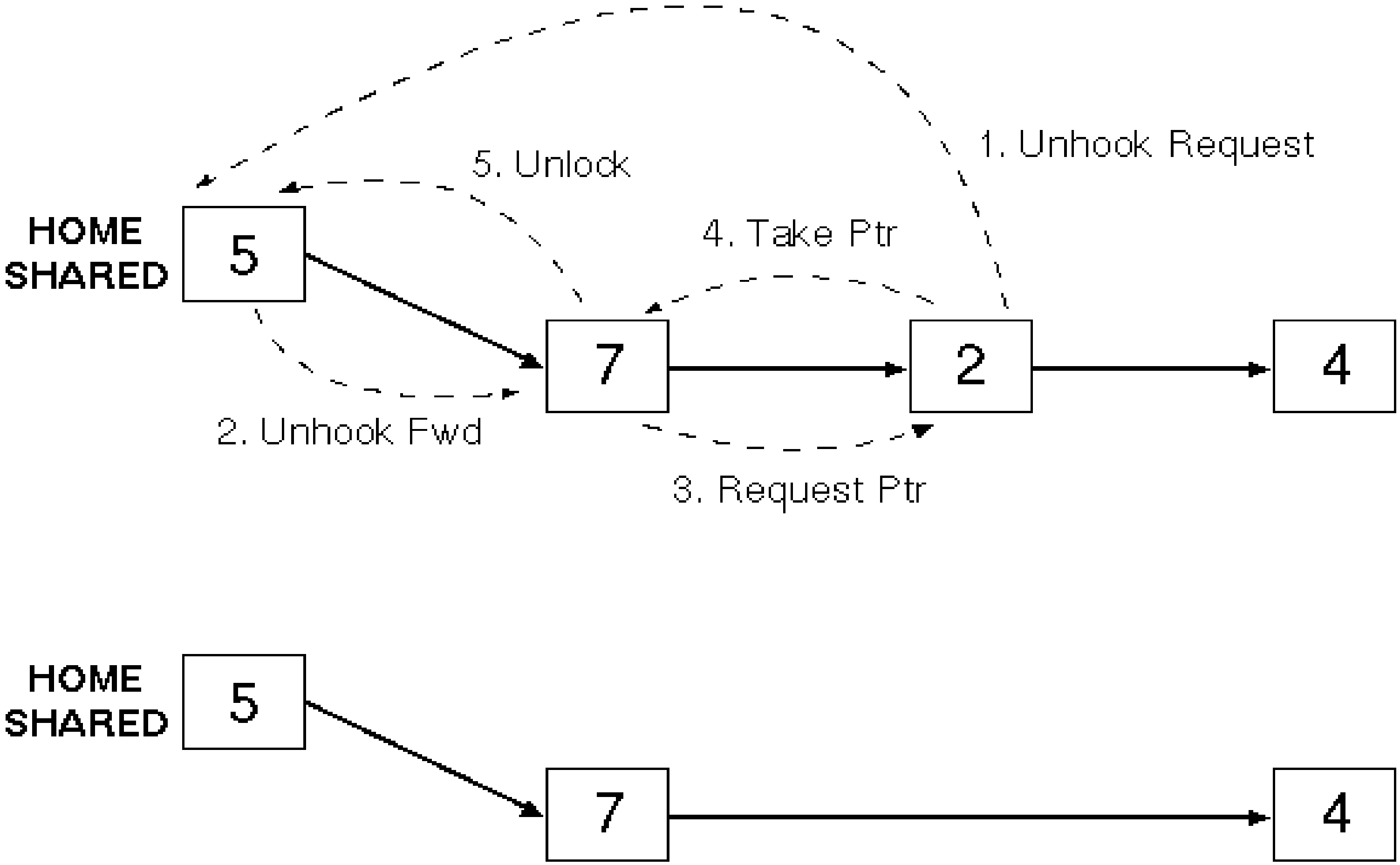
S3MP - Writes





- 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: bit vector sharing list
    - Real Origin 2000 systems in service with 256 CPUs
  - Sun E10000: hybrid multiple buses for invalidations, separate switched network for data transfers
    - Many E10000s in service, often with 64 CPUs

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, rumours from IBM…

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

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
  1. Although L1 cache is normally write-through, L2 is normally write-back
  2. Some data may bypass L3 (and perhaps L2) cache (eg when stream-prefetched)
- In Power4, cache controller manages L2 cache – all external invalidations/requests
- L3 cache improves access to DRAM for accesses both from CPU and from network

Summary and Conclusions

- Caches are essential to gain the maximum performance from modern microprocessors
- The performance of a cache is close to that of SRAM but at the cost of DRAM
- Caches can be used to form the basis of a parallel computer
- Bus-based multiprocessors do not scale well: max < 10 nodes
- Larger-scale shared-memory multiprocessors require more complicated networks and protocols
- CC-NUMA is becoming popular since systems can be built from commodity components (chips, boards, OSs) and use existing software
- e.g. HP/Convex, Sequent, Data General, SGI, Sun, IBM