Advanced Computer Architecture

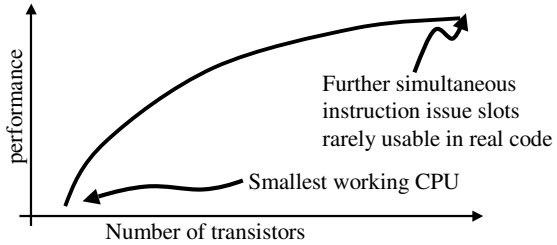
Chapter 7: shared memory and cache consistency

Paul Kelly
Dept of Computing
Imperial College
p.kelly@ic.ac.uk
http://www.doc.ic.ac.uk/~phjk

Overview

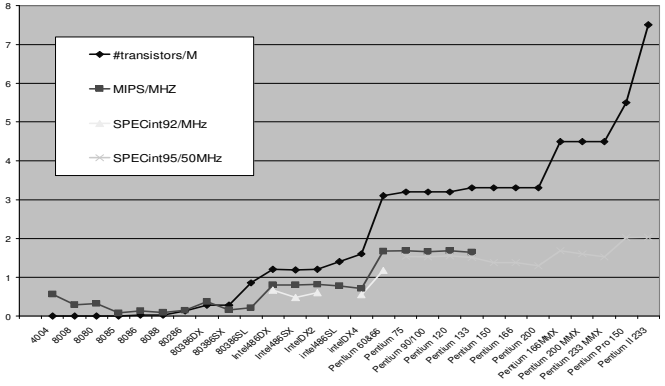
- Why add another processor?
- How should they be connected – I/O, memory bus, L2 cache, registers?
- Cache coherency – the problem
- Coherency – what are the rules?
- Coherency using broadcast – update/invalidate
- Coherency using multicast – SMP vs ccNUMA
- Distributed directories, home nodes, ownership;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

Why add another processor?



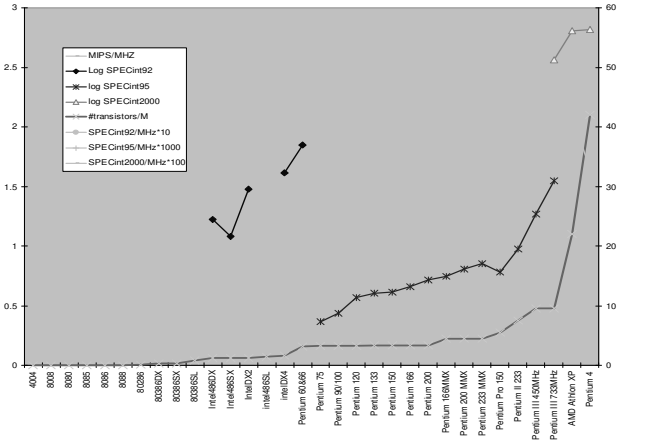
- Increasing the complexity of a single CPU leads to diminishing returns
  - Due to lack of instruction-level parallelism
  - Too many simultaneous accesses to one register file
  - Forwarding wires between functional units too long - inter-cluster communication takes >1 cycle

Architectural effectiveness of Intel processors



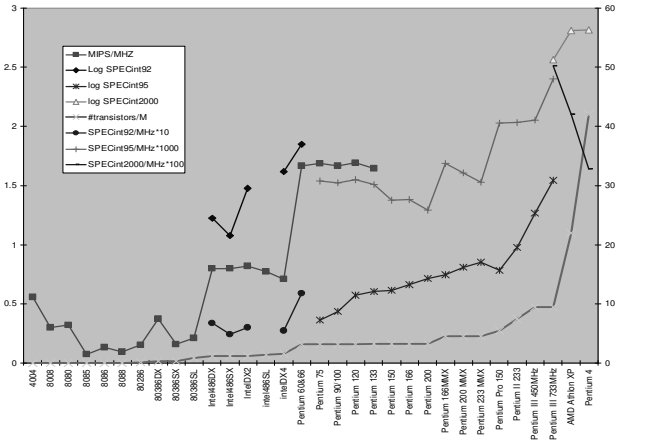
Source: http://www.galaxycentre.com/intelcpu.htm and Intel

Architectural effectiveness of Intel processors



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

Architectural effectiveness of Intel processors



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

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…

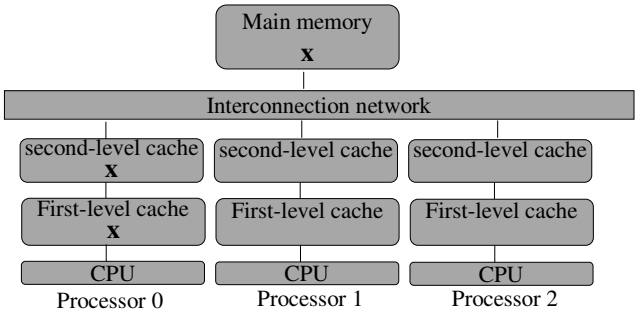
How to add another processor?

- 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)

How to connect processors...

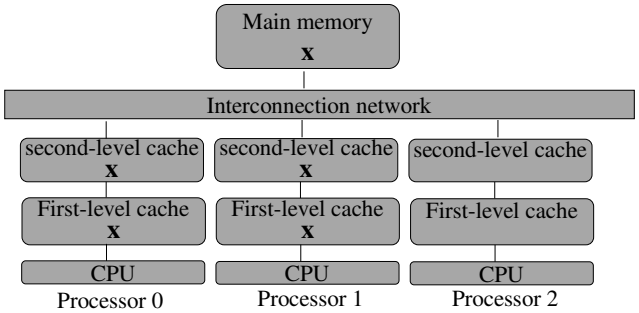
- Tradeoffs:
  - close coupling to minimise delays incurred when processors interact
  - separation to avoid contention for shared resources
- Result:
  - spectrum of alternative approaches based on application requirements, cost, and packaging/integration issues
- Currently:
  - just possible to integrate 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



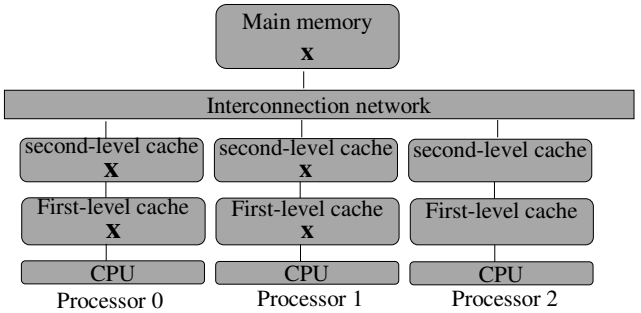
- Suppose processor 0 loads memory location **x**
- **x** is fetched from main memory and allocated into processor 0's cache(s)

Multiple caches… and trouble



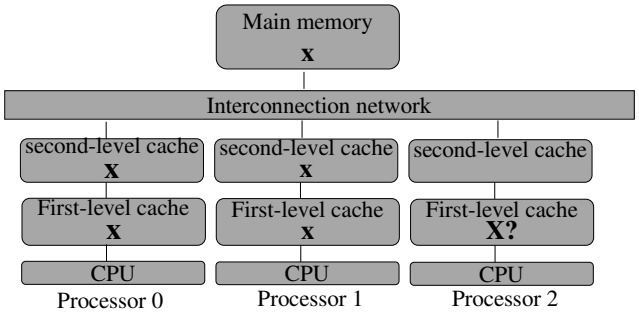
- Suppose processor 1 loads memory location **x**
- **x** is fetched from main memory and allocated into processor 1's cache(s) as well

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)

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

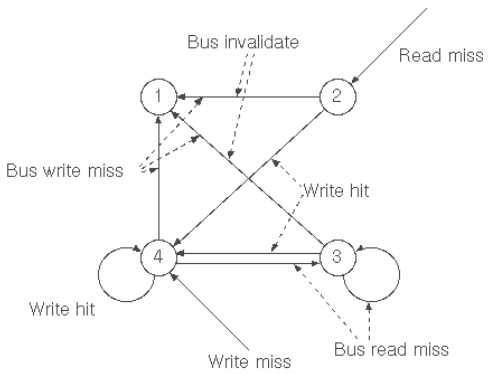
The "Berkeley" Protocol

Four cache line states:

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

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

- **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)

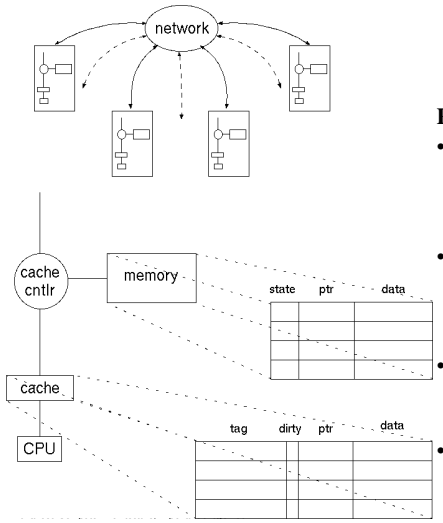
- For every bus transaction, it looks up the directory (cache line state) information for the specified address
  - If this processor holds the only valid data (DIRTY), it responds to a "Bus read miss" by providing the data to the requesting CPU
  - If the memory copy is out of date, one of the CPUs 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

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



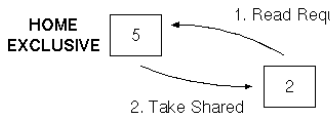
Case study: Sun's S3MP

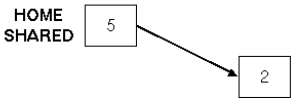
Protocol Basics

- 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

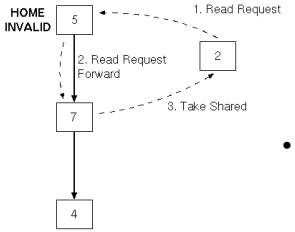
- Simple case: initially only the home has the data:





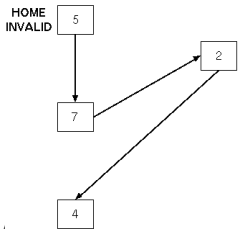
Curved arrows show messages, bold straight arrows show pointers

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

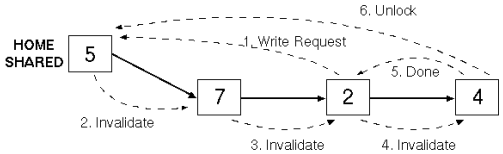


S3MP: Read Requests - remote

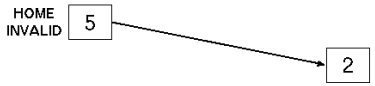
- More interesting case: some other processor has the data



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

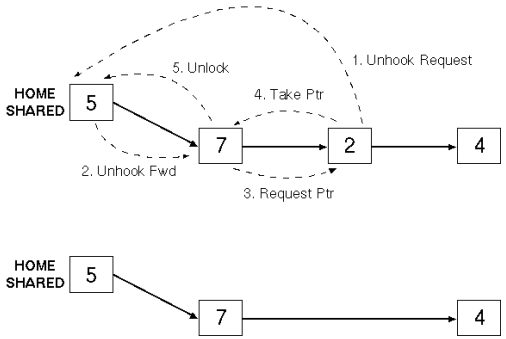


S3MP - Writes



- 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 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 (and many others)
- **Idea**: CPUs share L2/L3 cache
  - Eg IBM Power4
- **Idea**: CPUs share L1 cache
- **Idea**: CPUs share registers, functional units
  - Alpha 21464/EV8, Cray/Tera MTA (multithreaded architecture)

- **Cunning Idea**: do (almost) all the above at the same time
- Eg IBM SP/Power4: 2 CPUs/chip sharing L2, 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 (see http://www.rs6000.ibm.com/resource/features/1999/power4.html)

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