CS3350B
Computer Architecture
Winter 2015

Lecture 7.1: Multicore: Basics and Key Issues

Marc Moreno Maza

www.csd.uwo.ca/Courses/CS3350b

[Adapted from lectures on
Computer Organization and Design,
Patterson & Hennessy, 4th or 5th edition, 2011]

Why We need Multiprocessors?

- Uniprocessor performance
  - 25% annual improvement rate from 1978 to 1986
  - 52% annual improvement rate from 1986 to 2002
    - Profound impact of **RISC**, **x86**
  - 20% annual improvement rate from 2002 to present
    - **Power wall**: solutions for higher ILP are power-inefficient
    - **ILP wall**: hard to exploit more ILP
    - **Memory wall**: ever-increasing memory latency relative to processor speed

Beyond Implicit Parallelism

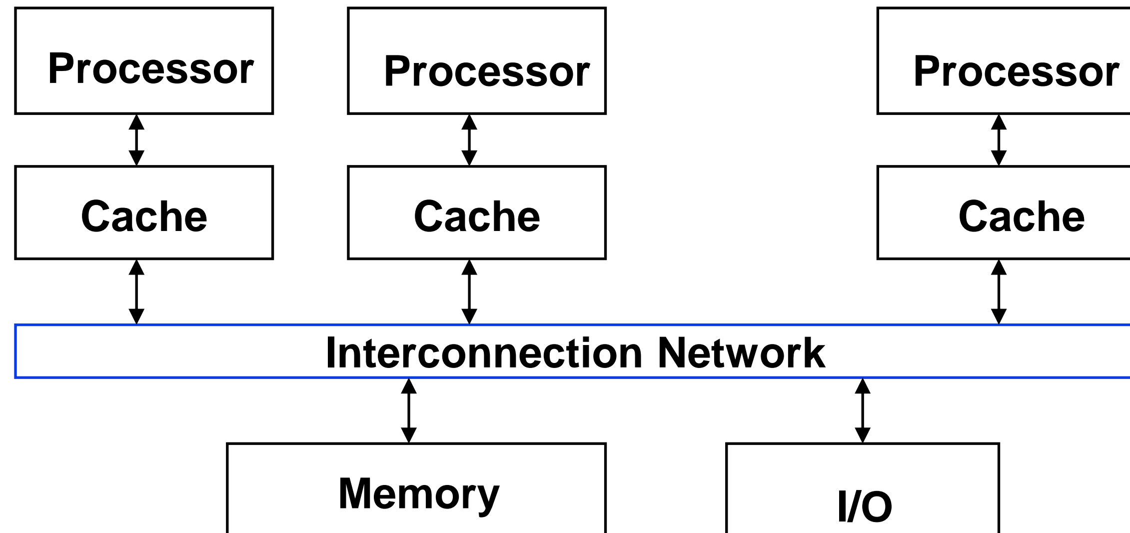
- Consider "`atxpy`":

```
double a, x[SIZE], y[SIZE], z[SIZE];
void atxpy(){
    for (i = 0; i < SIZE; i++)
        z[i] = a*x[i] + y[i];
}
```

- Lots of instruction-level parallelism (ILP)
  - Great!
  - But how much can we really exploit? 4-issue? 8-issue?
    - Limits to (efficient) super-scalar execution
- But, if SIZE is 10,000 the loop has 10,000-way parallelism!
  - How do we exploit it?

Where are We Now?

- Multiprocessor – a computer system with at least two processors



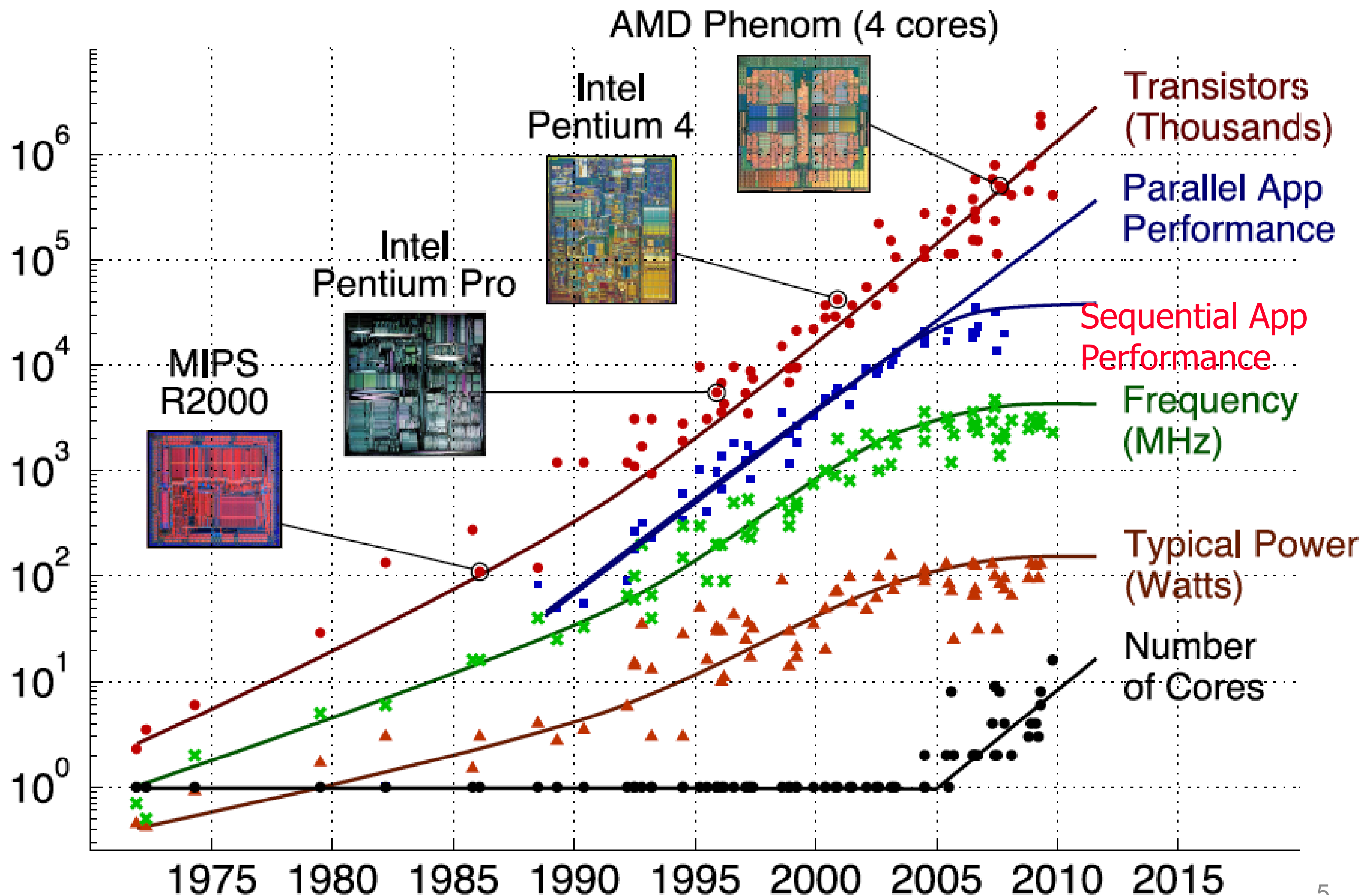
  - Can deliver high throughput for independent jobs via job-level parallelism or process-level parallelism
  - And improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program

Multicores Now Common

- The power challenge has forced a change in the design of microprocessors
  - Since 2002 the rate of improvement in the response time of programs has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year
- Today's microprocessors typically contain more than one core – Chip Multicore microProcessors (CMPs) – in a single IC
  - The number of cores is expected to double every two years

Product	AMD Barcelona	Intel Nehalem	IBM Power 6	Sun Niagara 2
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~2.5 GHz?	4.7 GHz	1.4 GHz
Power	120 W	~100 W?	~100 W?	94 W

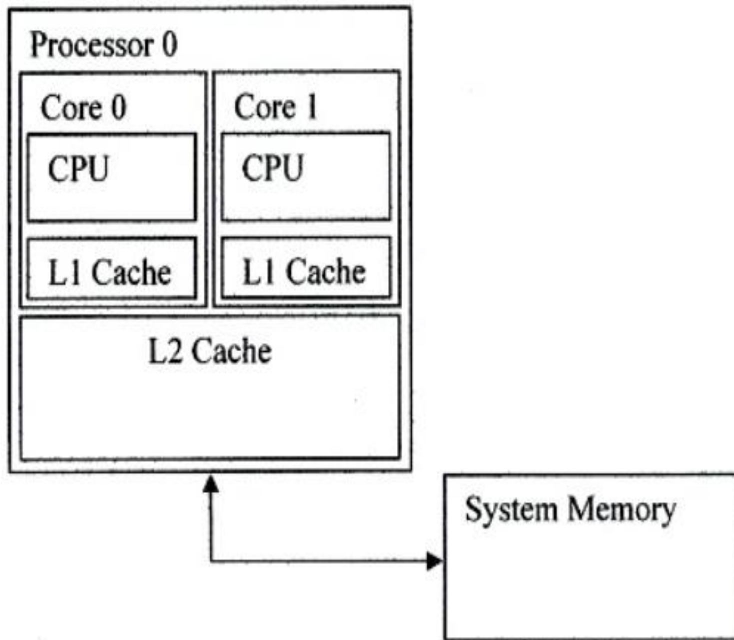
Transition to Multicore



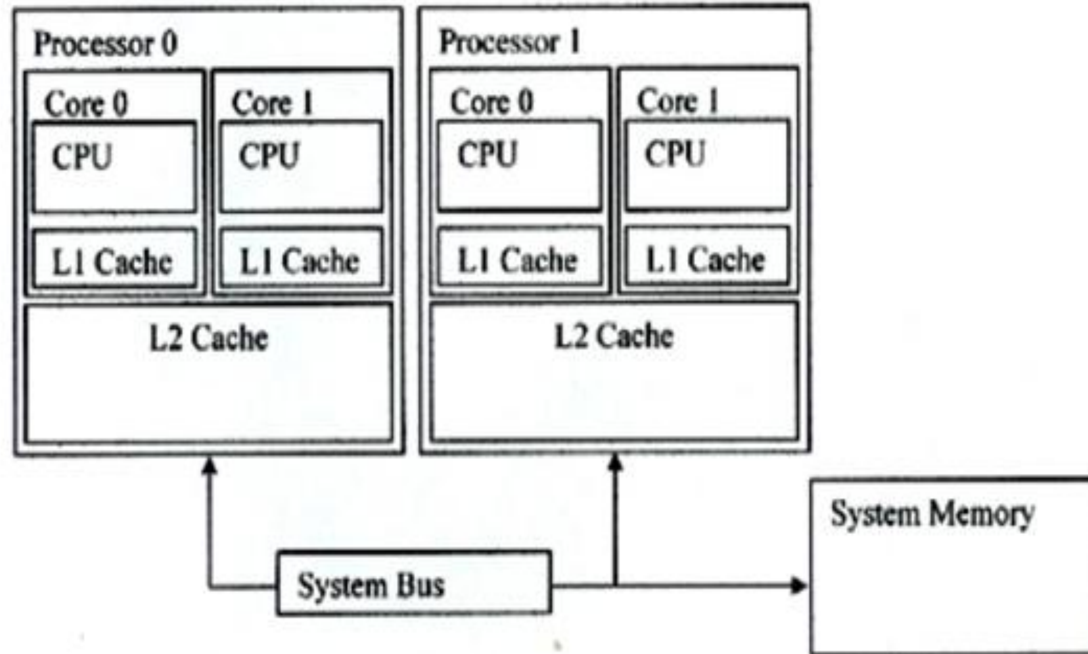
Other Multiprocessor Basics

- Some of the problems that need higher performance can be handled simply by using a **cluster** – a set of independent servers (or PCs) connected over a local area network (LAN) functioning as a single large multiprocessor
  - Search engines, Web servers, email servers, databases, …
- A key challenge is to craft **parallel** (concurrent) programs that have high performance on multiprocessors as the number of processors increase – i.e., that **scale**
  - Scheduling, load balancing, time for synchronization, overhead for communication

Multicore vs Multi-processor

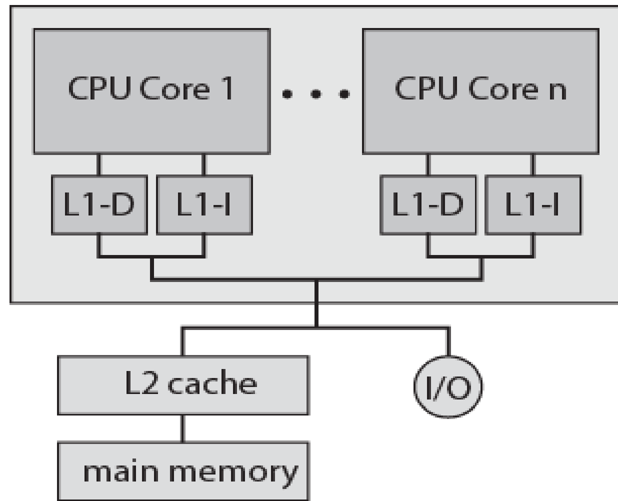


Multicore Processor with Shared L2 Cache

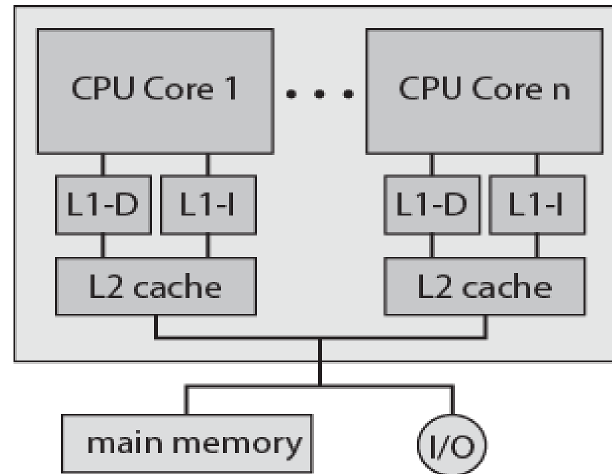


Multi-Processor System with Cores that share L2 Cache

Multicore Organization Alternatives



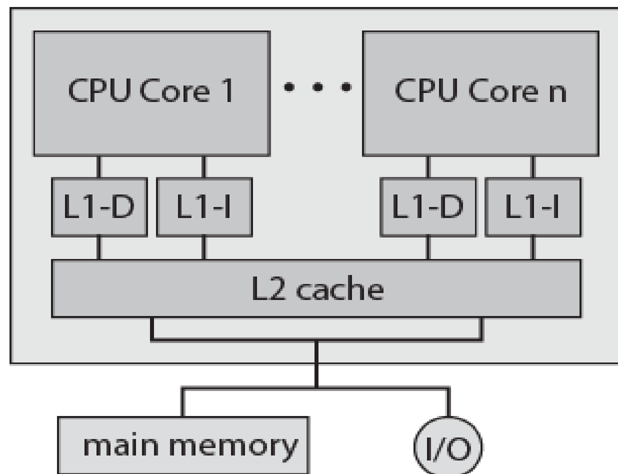
(a) Dedicated L1 cache



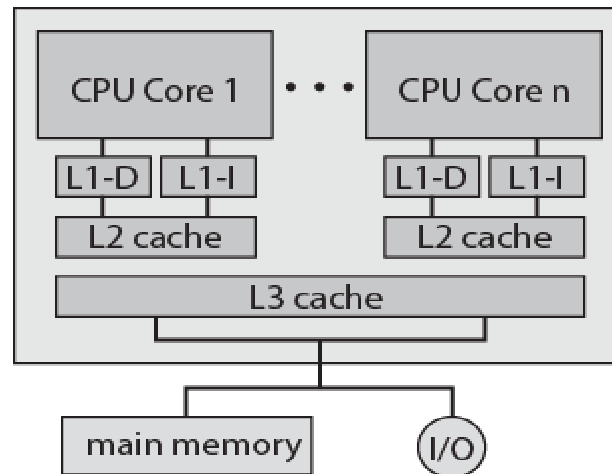
(b) Dedicated L2 cache

(a) ARM11 MPCore

(b) AMD Opteron



(c) Shared L2 cache

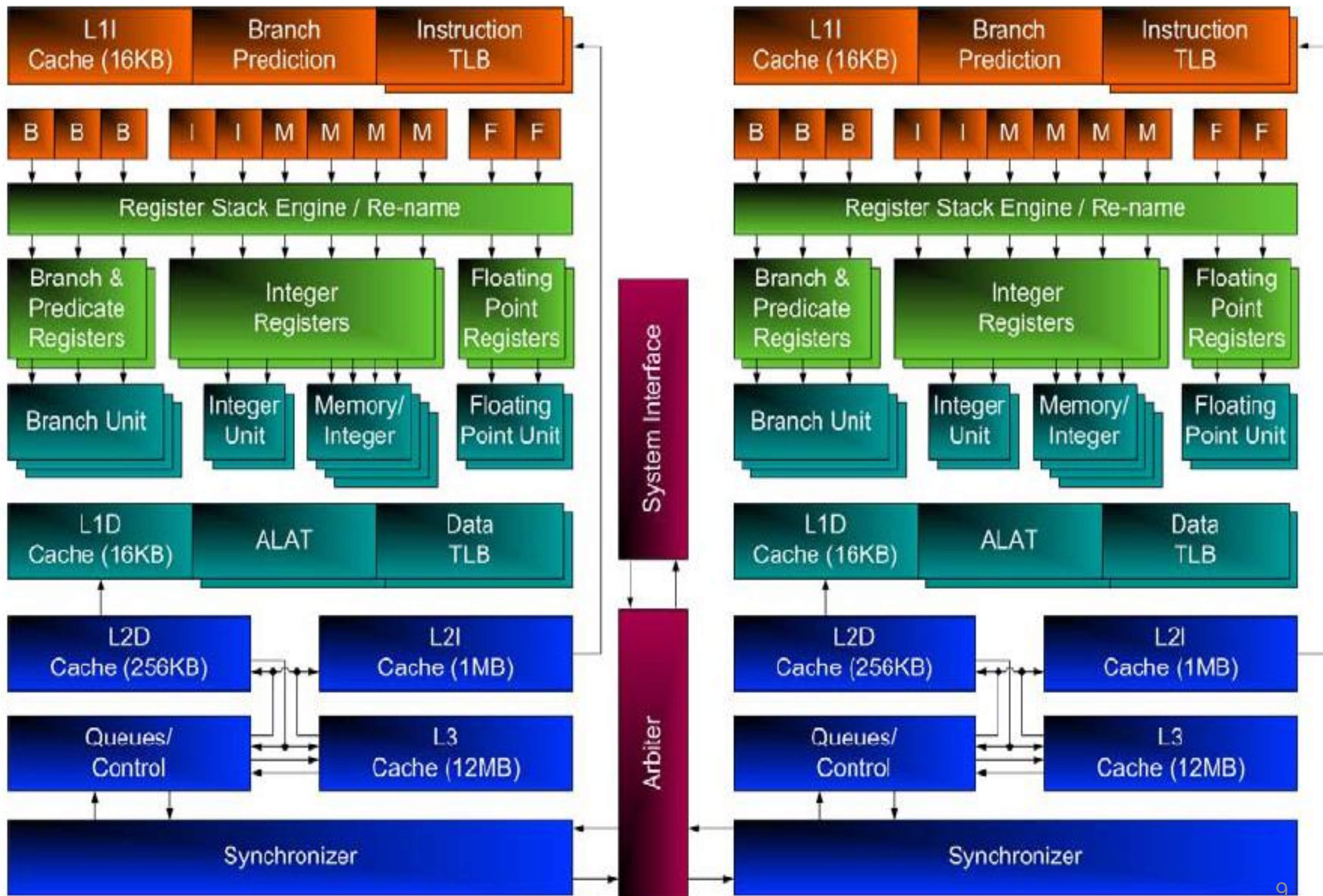


(d) Shared L3 cache

(c) Intel Core Duo

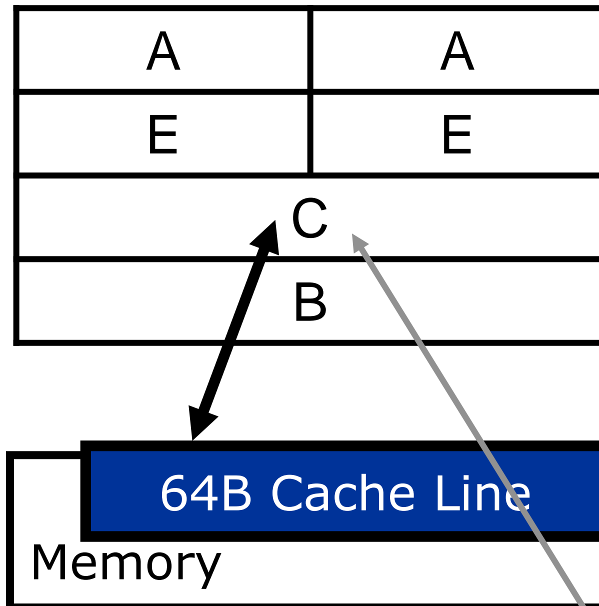
(d) Intel Core i7

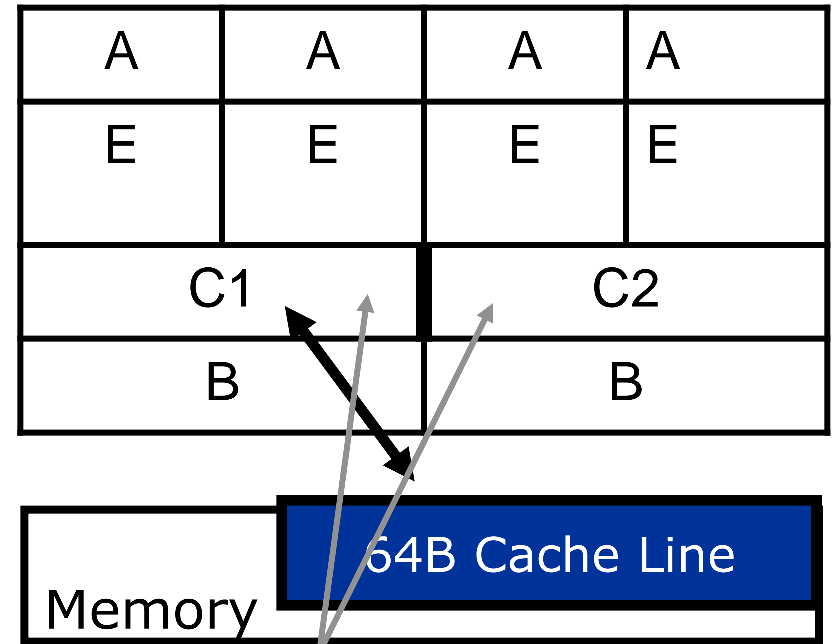
Itanium 2 Dual Core



High Level Multicore Architectural view

Intel® Core™ Microarchitecture – Memory Sub-system

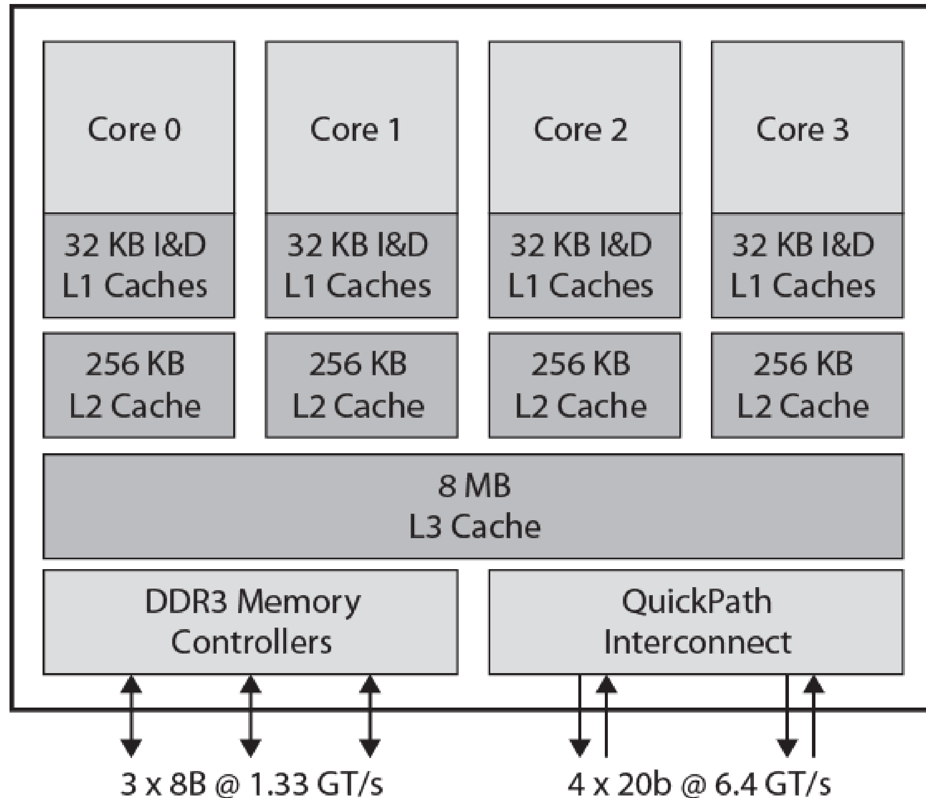




Dual Core has shared cache, Quad core has both shared And separated cache

A = Architectural State
C = 2nd Level Cache
E = Execution Engine & Interrupt
B = Bus Interface connects to main memory & I/O

Intel Core i7 Block Diagram



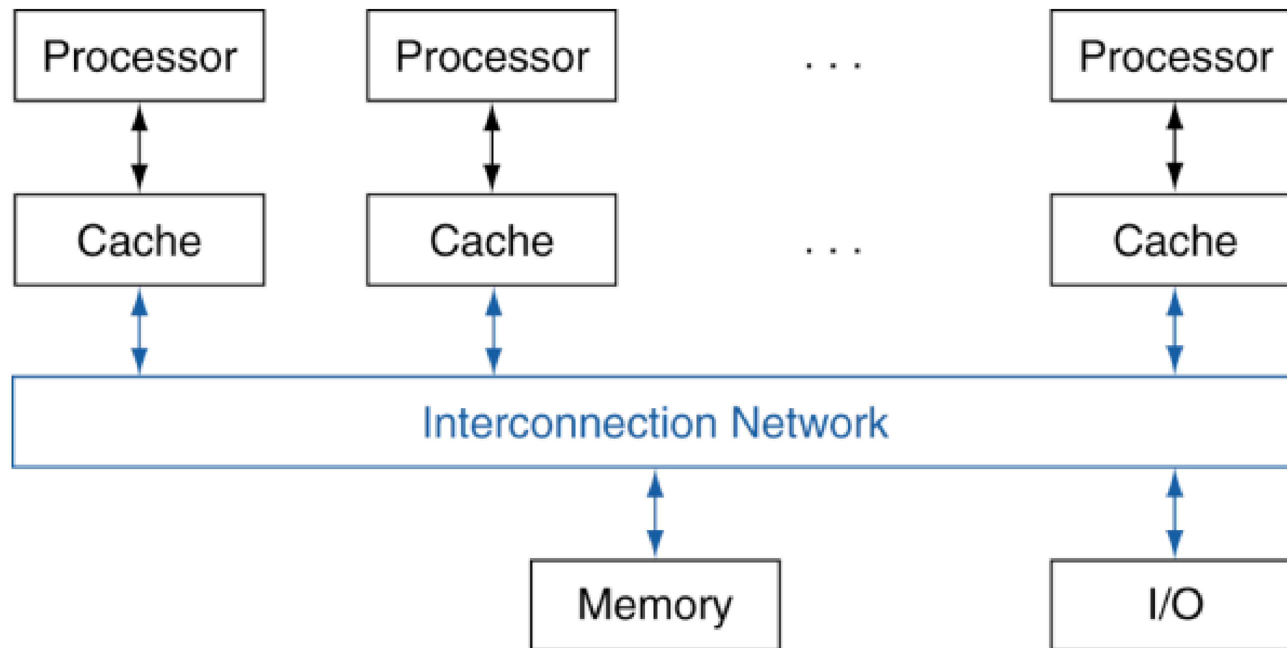
- November 2008
- Four x86 **SMT** processors
- Dedicated L2, shared L3 cache
- Speculative pre-fetch for caches
- On chip DDR3 memory controller
  - Three 8 byte channels (192 bits) giving 32GB/s
  - No front side bus
- QuickPath Interconnection
  - Cache coherent point-to-point link
  - High speed communications between processor chips
  - 6.4G transfers per second, 16 bits per transfer
  - Dedicated bi-directional pairs
  - Total bandwidth 25.6GB/s

Simultaneous Multithreading (SMT)

- A variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor (superscalar) to exploit both program ILP and thread-level parallelism (TLP)
  - Most SS processors have more machine level parallelism than most programs can effectively use (i.e., than have ILP)
  - With **register renaming** and **dynamic scheduling**, multiple instructions from independent threads can be issued without regard to dependencies among them
    - Need separate rename tables for each thread or need to be able to indicate which thread the entry belongs to
    - Need the capability to commit from multiple threads in one cycle

- Intel's Pentium 4 SMT is called **hyperthreading**
  - Supports just two threads (doubles the architecture state)

SMT



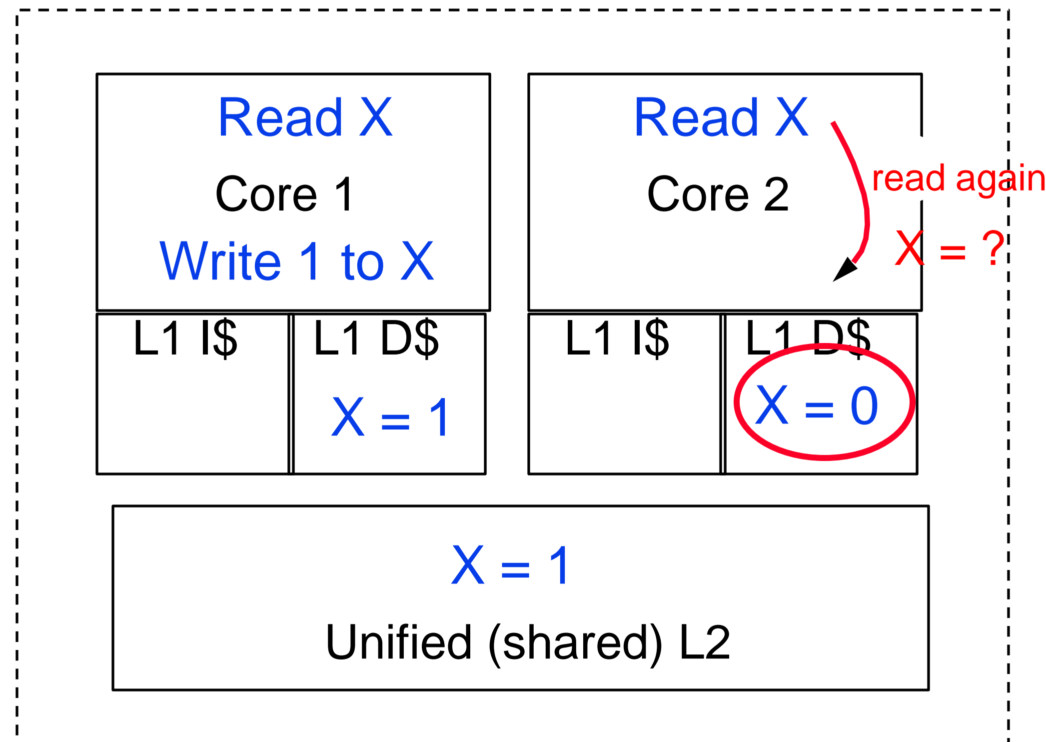
- Caches are (equally) helpful with multi-core
  - Reduce access latency, reduce bandwidth requirements
  - For both private and shared data across cores
- But caches introduce the problems of coherence & consistency

Cache Coherence

- What is the coherence problem?
  - Core writes to a location in its L1 cache
  - Other L1 caches may hold shared copies - these will be immediately out of date
- The core may either
  - Write through to L2 cache and/or memory
  - Copy back only when cache line is rejected
- In either case because each core may have its own copy, it is not sufficient just to update L2 and/or memory

Cache Coherence in Multicores

- In multicore processors its likely that the cores will **share** a common physical address space, causing a cache coherence problem



A Coherent Memory System

- Any read of a data item should return the most recently written value of the data item
  - **Coherence** – defines what values can be returned by a read
    - Writes to the same location are serialized (two writes to the same location must be seen in the same order by all cores)
  - **Consistency** – determines when a written value will be returned by a read
- To enforce coherence, caches (hardware) must provide
  - Replication of shared data items in multiple cores' caches
    - Replication reduces both latency and contention for a read shared data item
  - Migration of shared data items to a core's local cache
    - Migration reduced the latency of the access the data and the bandwidth demand on the shared memory (L2 in our example)

Snooping Protocols

- Schemes where every core knows **which** other core has a copy of its cached data are far too complex.
- So each core (cache system) ‘snoops’ (i.e. watches continually) for activity concerned with data addresses which it has cached.
- This has normally been implemented with a bus structure which is ‘global’, i.e. all communication can be seen by all
- Snooping Protocols can be implemented without a bus, but for simplicity the next slides assume a shared bus.
- There are ‘directory based’ coherence schemes but we will not consider them this year.

Snooping Protocols

Write Invalidate

1. A core wanting to write to an address, grabs a bus cycle and sends a ‘**write invalidate**’ message which contains the address
2. All snooping caches invalidate their copy of appropriate cache line
3. The core writes to its cached copy (assume for now that it also writes through to memory)
4. Any shared read in other cores will now miss in cache and re-fetch the new data.

Snooping Protocols

Write Update

1. A core wanting to write grabs bus cycle and broadcasts address & new data as it updates its own copy
2. All snooping caches update their copy

- Note that in both schemes, the problem of simultaneous writes is taken care of by **bus arbitration** - only one core can use the bus at any one time.

Update or Invalidate?

- Update looks the simplest, most obvious and fastest, but:
  - Multiple writes to the same word (no intervening read) need only one invalidate message but would require an update for each
  - Writes to same block in (usual) multi-word cache block require only one invalidate but would require multiple updates.

Update or Invalidate?

- Due to both spatial and temporal locality, the previous cases occur often.
- **Bus bandwidth** is a precious commodity in shared memory multi-cores chips
- Experience has shown that **invalidate protocols** use significantly less bandwidth.
- We will only consider implementation details only of the invalidate protocols.

  All commercial machines use write-invalidate as their standard coherence protocol

Implementation Issues

- In both schemes, knowing if a cached value is not shared (no copies in another cache) can avoid sending any messages.
- Invalidate description assumed that a cache value update was written through to memory. If we used a ‘copy back’ scheme (usual for high performance) other cores could re-fetch incorrect old value on a cache miss.
- We need a protocol to handle all this.

MESI Protocol (1)

- A practical multi-core invalidate protocol which attempts to minimize bus usage.
- Allows usage of a ‘copy back’ scheme - i.e. L2/main memory is not updated until a ‘dirty’ cache line is displaced
- Extension of the usual cache tags, i.e. invalid tag and ‘dirty’ tag in normal copy back cache.
- To make the description simpler, we will ignore L2 cache and treat L2/main memory as a single main memory unit

MESI Protocol (2)

Any cache line can be in one of 4 states (2 bits)

- **Modified** – The cache line has been modified and is different from main memory – This is the only cached copy. (cf. ‘dirty’)
- **Exclusive** – The cache line is the same as main memory and is the only cached copy
- **Shared** - Same value as main memory but copies may exist in other caches.
- **Invalid** - Line data is not valid (as in simple cache)

MESI Protocol (3)

- Cache line state changes are a function of memory access events.

- Events may be either
  - Due to local core activity (i.e. cache access)
  - Due to bus activity - as a result of snooping

- Each cache line has its own state affected only if the address matches

MESI Protocol (4)

- Operation can be described informally by looking at actions in a local core
  - Read Hit
  - Read Miss
  - Write Hit
  - Write Miss

- More formally by a state transition diagram

MESI Local Read Hit

- The line must be in one of MES
- This must be the correct local value (if M it must have been modified locally)
- Simply return value
- No state change

MESI Local Read Miss

A core makes read request to main memory upon a read miss: detailed action depends on copies in other cores

Case 1: One cache has an E copy

- The snooping cache puts the copy value on the bus
- The memory access is abandoned
- The local core caches the value
- Both lines are set to S

Case 2: No other copy in caches

- The core waits for a memory response
- The value is stored in the cache and marked E

MESI Local Read Miss

Case 3: Several caches have a copy (S)

- One snooping cache puts the copy value on the bus (arbitrated)
- The memory access is abandoned
- The local core caches the value and sets the tag to S
- Other copies remain S

MESI Local Read Miss

Case 4: One cache has M (modified) copy

- The snooping cache puts its copy of the value on the bus
- The memory access is abandoned
- The local core caches the value and sets the tag to S
- **The source (M) value is copied back to memory**
- The source value changes its tag from M to S

MESI Local Write Hit

Line must be one of MES

- M
  - line is exclusive and already ‘dirty’
  - Update local cache value
  - no state change
- E
  - Update local cache value
  - Change E to M
- S
  - Core broadcasts an invalidate on bus
  - Snooping cores with an S copy change S to I
  - The local cache value is updated
  - The local state changes from S to M

MESI Local Write Miss

Detailed action depends on copies in other cores

Case 1: No other copies

- Local copy state set to M

Case 2: Other copies, either one in state E or more in state S

- Value read from memory to local cache - bus transaction marked RWITM (read with intent to modify)
- The snooping cores see this and set their tags to I
- The local copy is updated and sets the tag to M

MESI Local Write Miss

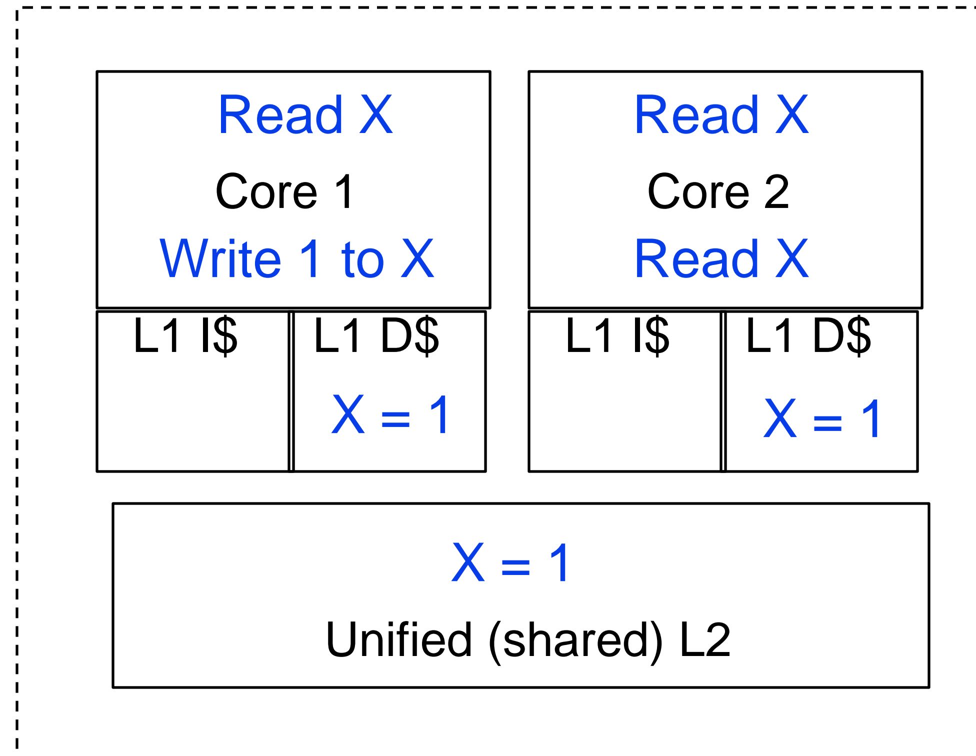
Case 3: Another copy in state M

- Core issues bus transaction marked RWITM
- The snooping core sees this
  - Blocks the RWITM request
  - Takes control of the bus
  - Writes back its copy to memory
  - Sets its copy state to I
- The original local core re-issues RWITM request
- This is now simply a no-copy case
  - Value read from memory to local cache
  - Local copy value updated
  - Local copy state set to M

Comments on MESI Protocol

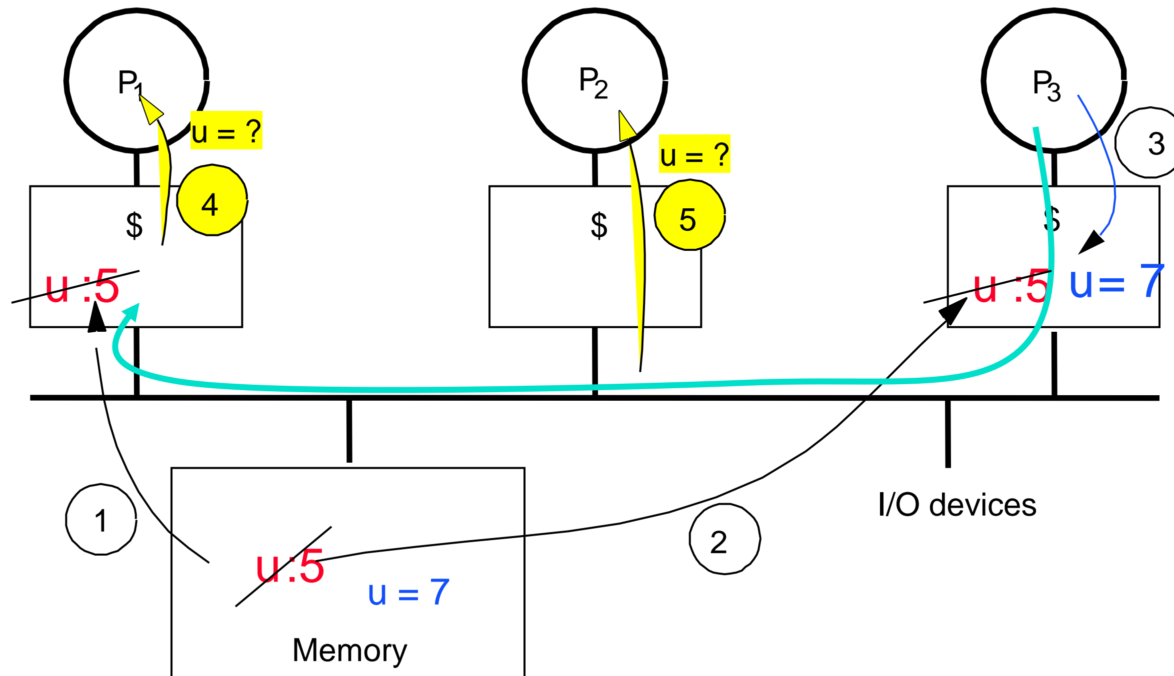
- Relies on global view of all memory activity – usually implies a **global bus**
- Bus is a limited shared resource
- As number of cores increases
  - Demands on bus bandwidth increase – more total memory activity
  - The bus gets slower due to increased capacitive load
- General consensus is that bus-based systems cannot be extended beyond a small number (8 or 16?) cores

Example of Snooping Invalidation



- When the second miss by Core 2 occurs, Core 1 responds with the value canceling the response from the L2 cache (and also updating the L2 copy)

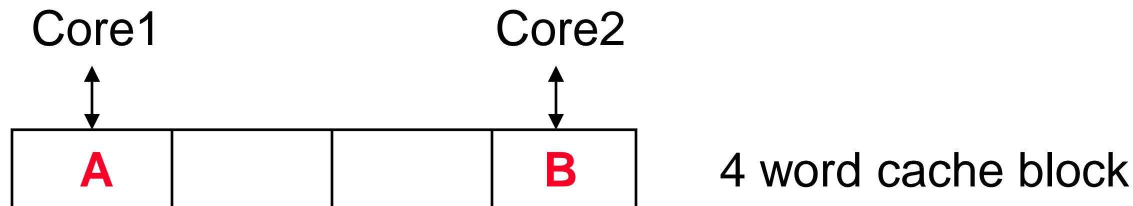
Another Example for Write Invalidate Snooping



- Must invalidate before step 3
- All recent MPUs use **write invalidate** (invalidate protocols use significantly less bandwidth)

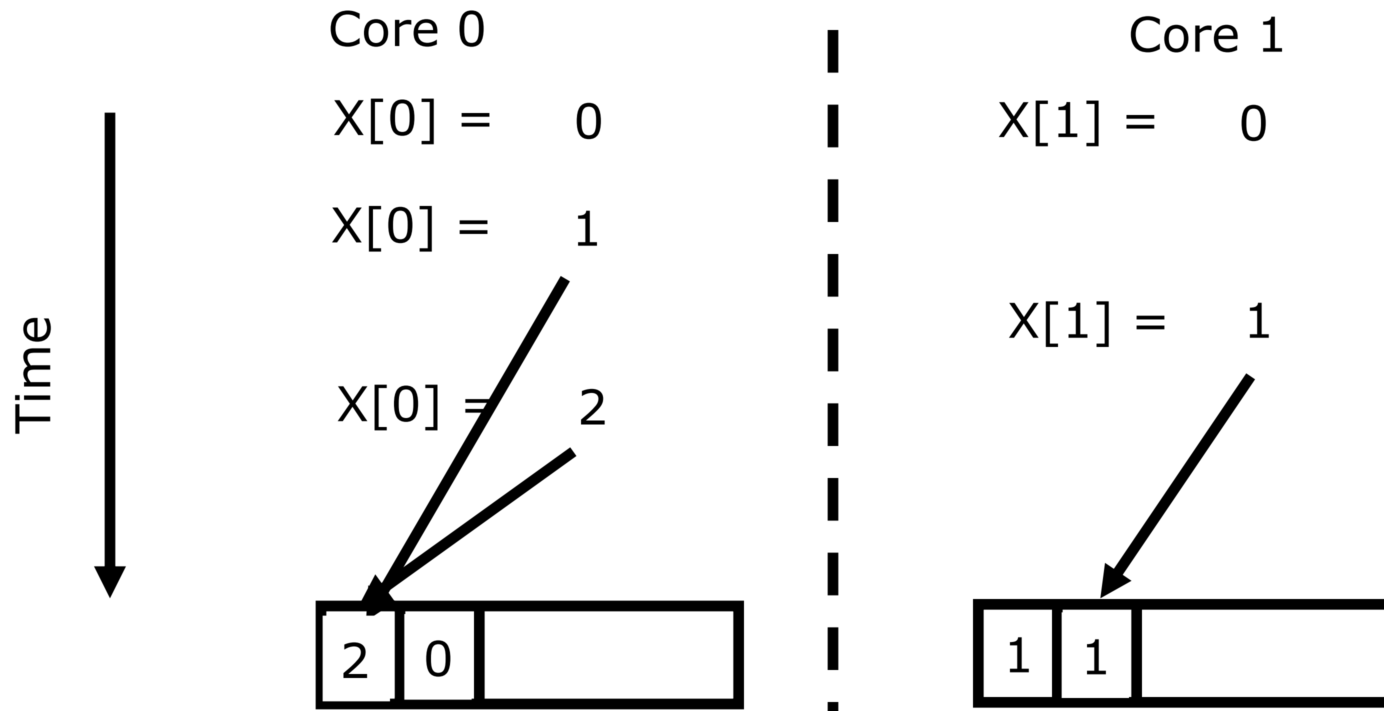
Block Size Effects on Multicores

- Writes to one word in a multi-word block mean that the full block is invalidated
- **Multi-word blocks** can also result in **false sharing**: when two cores are writing to two different variables that happen to fall in the same cache block
  - With **write-invalidate,** false sharing increases cache miss rates



False Sharing

- Performance issue in programs where cores may write to different memory addresses BUT in the same cache lines
- Known as Ping-Ponging – Cache line is shipped between cores



False Sharing is not an issue in shared cache.
It is an issue in separated cache

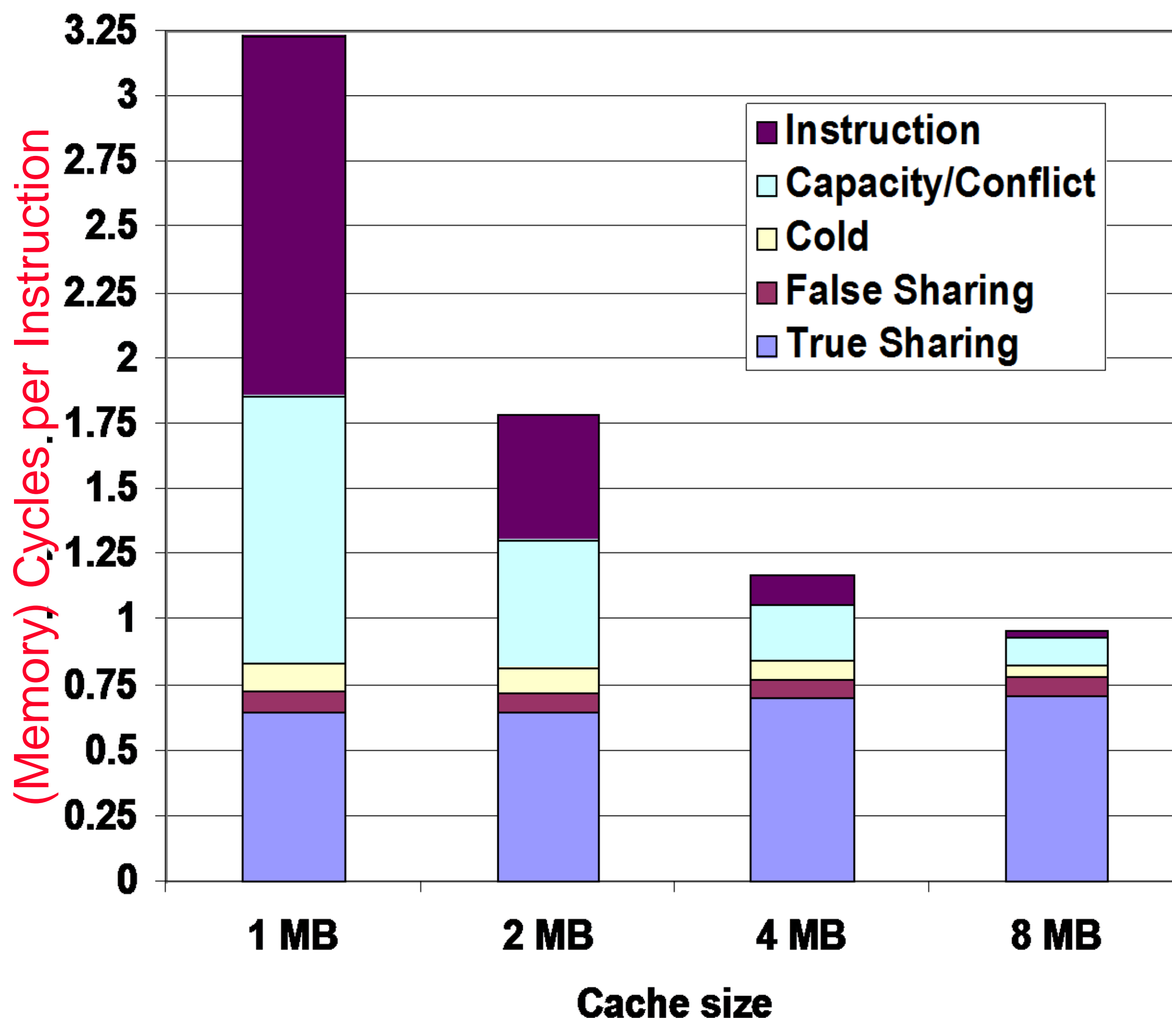
Coherency Misses

1. True sharing misses arise from the communication of data through the cache coherence mechanism
   - Invalidates due to 1st write to shared block
   - Reads by another CPU of modified block in different cache
   - Miss would still occur if block size were 1 word
2. False sharing misses when a block is invalidated because some word in the block, other than the one being read, is written into
   - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
   - Block is shared, but no word in block is actually shared $\Rightarrow$ miss would not occur if block size were 1 word

MP Performance for Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

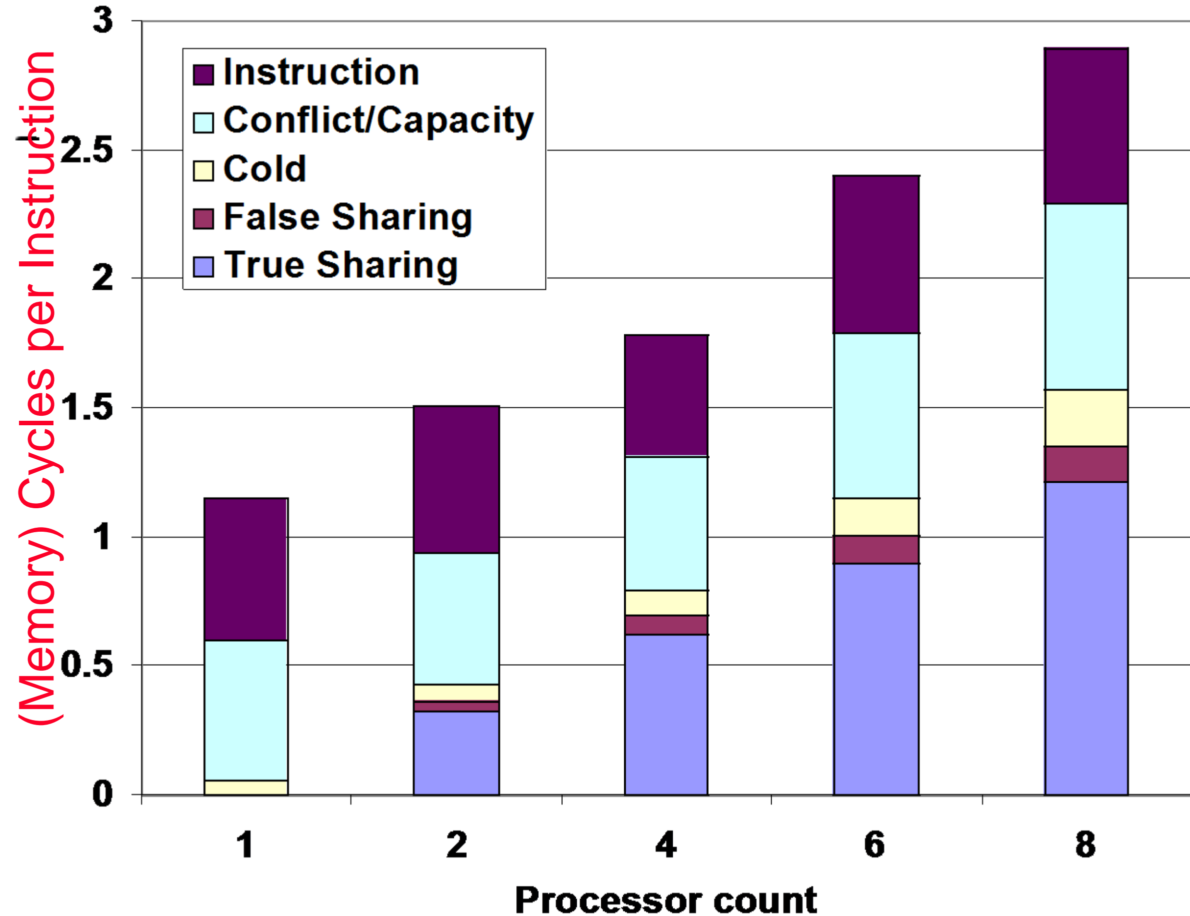
- **Uniprocessor** cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)

- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)



MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

True sharing, false sharing increase going from 1 to 8 CPUs



Avoiding False Sharing

Change either

- Algorithm
  - adjust the implementation of the algorithm (the loop stride) to access data in different cache line for each thread

Or

- Data Structure:
  - add some “padding” to a data structure or arrays (just enough padding generally less than cache line size) so that threads access data from different cache lines.

Summary

- Multicores are common multiprocessors nowadays
- Offer computing resources to improve throughput by process and thread level parallelism in addition to ILP
- Dedicated on-chip caches and shared lower level caches
- Key issues
  - Cache coherence
    - Popular protocol: snooping invalidation
  - False-sharing

Example: True v. False Sharing v. Hit?

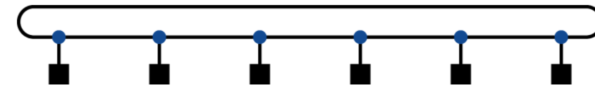
Assume x1 and x2 in same cache block.
P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

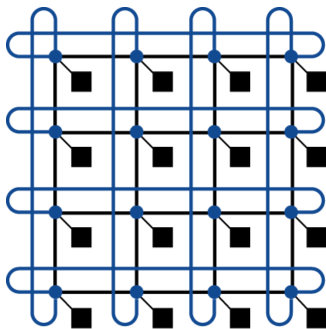
Aside: Interconnection Networks

- Network topologies
  - Arrangements of processors, switches, and links

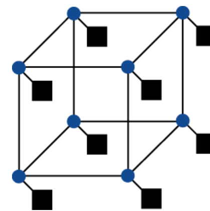
Bus



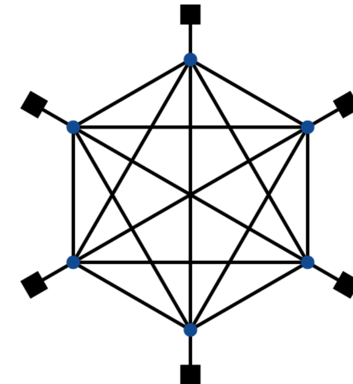
Ring



2D Mesh

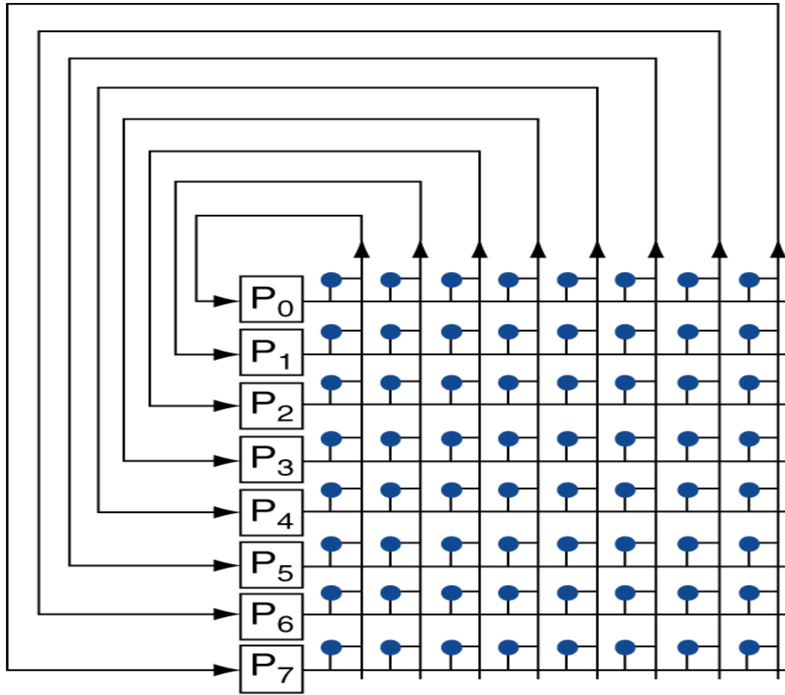


N-cube (N = 3)

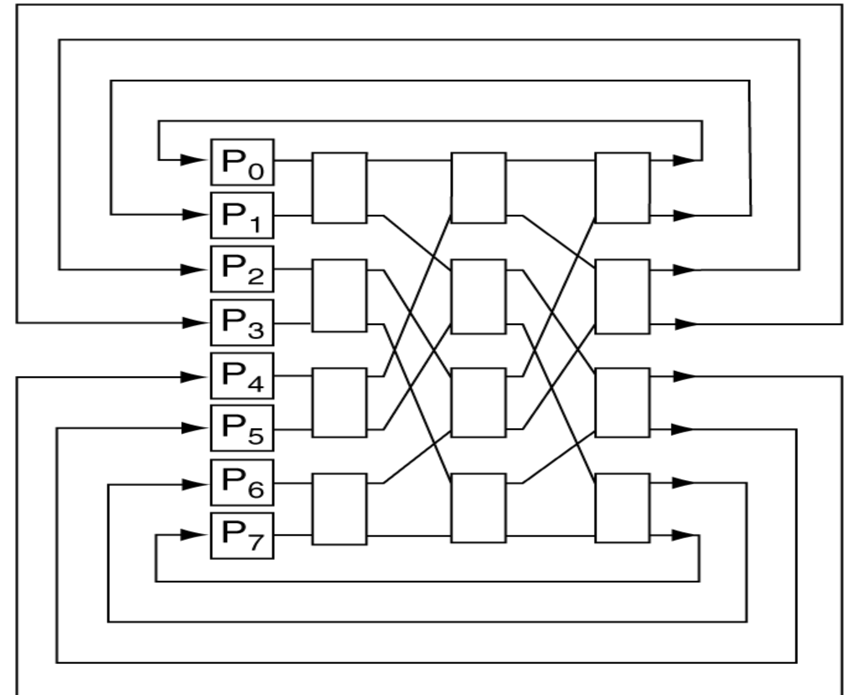


Fully connected

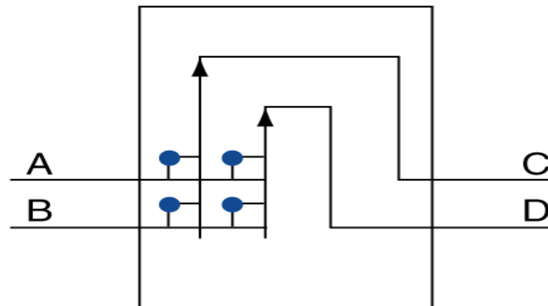
Aside: Multistage Networks



a. Crossbar



b. Omega network



c. Omega network switch box

Aside: Network Characteristics

- Performance
  - Latency per message (unloaded network)
  - Throughput
    - Link bandwidth
    - Total network bandwidth
    - Bisection bandwidth
  - Congestion delays (depending on traffic)
- Cost
- Power
- Routability in silicon

Next: Exploit TLP on Multicores

- Basic idea: Processor resources are expensive and should not be left idle
- Long memory latency to memory on cache miss?
- Hardware switches threads to bring in other useful work while waiting for cache miss
- Cost of thread context switch must be much less than cache miss latency
- Put in redundant hardware so don’t have to save context on every thread switch:
  - PC, Registers, L1 caches?
- Attractive for apps with abundant TLP
  - Commercial multi-user workloads

```
struct foo {
    int x;
    int y;
};

static struct foo f;

/* The two following functions are running concurrentlyby two threads: */

int sum_a(void) {
    int s = 0;
    int i;
    for (i = 0; i < 1000000; ++i)
        s += f.x;
    return s;
}

void inc_b(void) {
    int i;
    for (i = 0; i < 1000000; ++i)
        ++f.y;
}
```