

Parallelism (PAR)

Introduction to (shared-memory) parallel architectures

Eduard Ayguadé, José Ramón Herrero and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2019/20 (Fall semester)

Additional learning material for this lesson

- ▶ Atenea: Unit 4.1 Introduction to parallel architectures I
 - ▶ Video lessons covering the cache coherence problem in UMA architectures and two snooping-based solutions (write update and invalidate)
- ▶ Atenea: Unit 4.2 Introduction to parallel architectures II
 - ▶ Video lessons covering directory-based coherence in NUMA architectures
- ▶ Atenea: Unit 4.3 Introduction to parallel architectures III
 - ▶ Video lesson covering true vs. false sharing
- ▶ Quizzes after video lessons
- ▶ Collection of Exercises: problems in Chapter 4

Outline

Uniprocessor parallelism

Symmetric multi-processor architectures

Multicore architectures

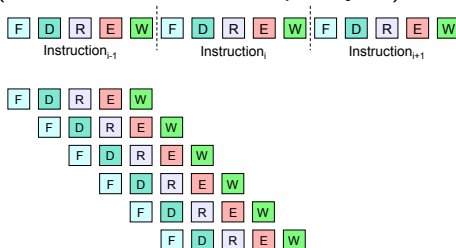
Non-Uniform Memory Architectures

Synchronization mechanisms

The memory consistency problem

Pipelining

- ▶ Execution of single instruction divided in multiple stages
- ▶ Overlap the execution of different stages of consecutive instructions
- ▶ Ideal: $IPC=1$ (1 instruction executed per cycle)



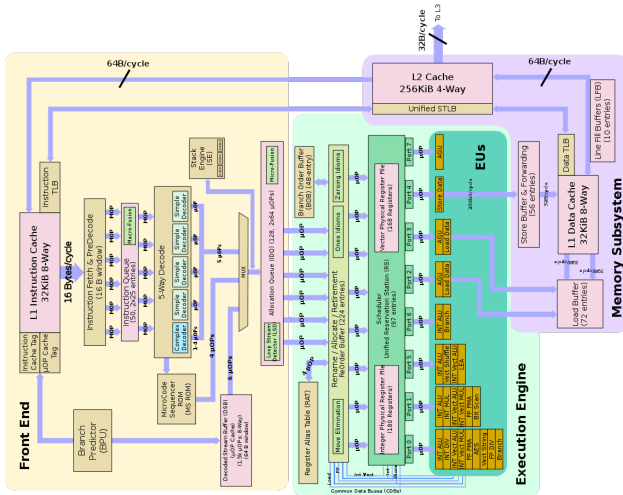
- ▶ $IPC < 1$ due to hazards (structural, data, control), preventing the execution of an instruction in its designated clock cycle

Sources of parallelism in uniprocessors

- ▶ ILP (Instruction-level parallelism)
 - ▶ Superscalar architecture: multiple issue slots (functional units)
 - ▶ Execution of multiple instructions, from the same instruction flow, per cycle
- ▶ TLP (thread-level parallelism)
 - ▶ Hardware multithreading¹: fill the pipeline with instructions from multiple instruction flows
 - ▶ Latency hiding (cache misses, non-pipelined FP, ...)
- ▶ DLP (data-level parallelism)
 - ▶ SIMD architecture: single-instruction executed on multiple-data in a single word
 - ▶ Vector functional unit

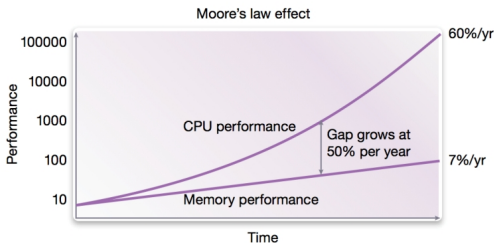
¹Hyperthreading in Intel terminology

Current uniprocessor architecture: Intel Skylake

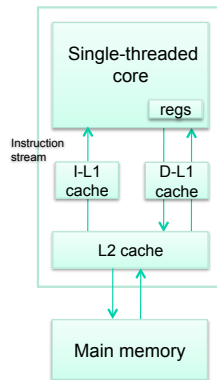


Memory hierarchy

- ▶ Addressing the yearly increasing gap between CPU cycle and memory access times



- ▶ Size vs. access time



- ▶ Non-blocking design

Memory hierarchy

- ▶ The principle of locality: if an item is referenced ...
 - ▶ Temporal locality: ... it will tend to be referenced again soon (e.g., loops, reuse)
 - ▶ Spatial locality: ... items whose addresses are close by tend to be referenced soon (e.g., straight line code, array access)
- ▶ Line (or block)
 - ▶ A number of consecutive words in memory (e.g. 32 bytes, equivalent to 4 words x 8 bytes)
 - ▶ Unit of information that is transferred between two levels in the hierarchy
- ▶ On an access to a level in the hierarchy
 - ▶ Hit: data appears in one of the lines in that level
 - ▶ Miss: data needs to be retrieved from a line in the next level

Elements of cache design

- ▶ Organization
 - ▶ Single vs. multilevel cache, Unified vs. split (instruction/data)
 - ▶ Cache size and line size
 - ▶ Addressing: logical vs. physical
- ▶ Placement algorithm
 - ▶ Direct, associative, set associative
- ▶ Replacement algorithm
 - ▶ Random, Least Recently Used (LRU), First-in First-out (FIFO), Least Frequently Used (LFU)
- ▶ Write (on hit) Policy
 - ▶ Write-through, Write-back
- ▶ Write (on miss) Policy
 - ▶ Write-allocate, write-no-allocate

Who exploits this uniprocessor parallelism and memory organization?

In theory, the compiler understands all of this ... but in practice the compiler may need your help, for example:

- ▶ Software pipelining to statically schedule ILP
- ▶ Vectorization to efficiently exploit SIMD vector units
- ▶ Data contiguous in memory and aligned to cache lines
- ▶ Blocking (or tiling) to define a problem that fits in register/L1-cache/L2-cache (temporal locality)

Reasons and techniques explored in detail in PCA course
(Architecture-Conscious Programming)

Outline

Uniprocessor parallelism

Symmetric multi-processor architectures

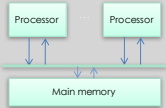
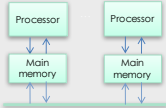
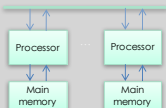
Multicore architectures

Non-Uniform Memory Architectures

Synchronization mechanisms

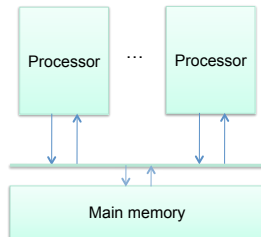
The memory consistency problem

Classification of multi-processor architectures

Memory architecture	Address space(s)	Connection	Model for data sharing	Names
(Centralized) Shared-memory architecture	Single shared address space, uniform access time		Load/store instructions from processors	<ul style="list-style-type: none"> SMP (Symmetric Multi-Processor) architecture UMA (Uniform Memory Access) architecture
Distributed-memory architecture	Single shared address space, non-uniform access time		Load/store instructions from processors	<ul style="list-style-type: none"> DSM (Distributed-Shared Memory) architecture NUMA (Non-Uniform Memory Access) architecture
	Multiple separate address spaces		Explicit messages through network interface card	<ul style="list-style-type: none"> Message-passing multiprocessor Cluster Architecture Multicomputer

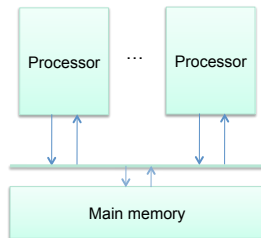
Symmetric multi-processor architectures

- ▶ Abbreviated SMP
 - ▶ Two or more identical processors are connected to a single shared main memory
 - ▶ Interconnection network: any processor can access to any memory location
- ▶ Symmetric multiprocessing: a single OS instance on the SMP
 - ▶ Asymmetric multiprocessor (e.g. high/low ILP processors, ...) and/or multiprocessing (e.g. some processors running OS, others user code)



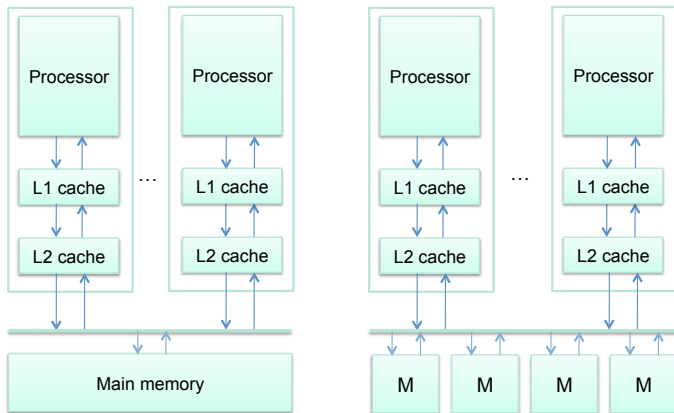
Symmetric multi-processor architectures

- ▶ Uniform Memory Access (UMA)
 - ▶ Access to shared data with load/store instructions
 - ▶ Access time to a memory location is independent of which processor makes the request or which memory chip contains the data
- ▶ The bottleneck in the scalability of SMP is the 'bandwidth' of the interconnection network and the memory



Symmetric multi-processor architectures

Local caches and multi-banked (interleaved) memory



The coherence problem

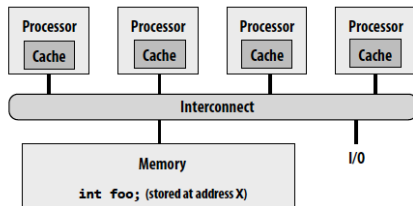


Chart shows value of `foo` (variable stored at address X) stored in main memory and in each processor's cache **

** Assumes write-back cache behavior

Action	P1 \$	P2 \$	P3 \$	P4 \$	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (say this load causes eviction of foo)		0	2		1

(CMU 15-418, Spring 2012)

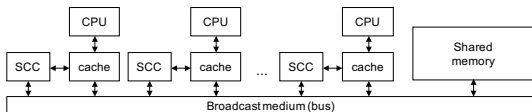
Coherence protocols

- ▶ Write-update:
 - ▶ Writing processor broadcasts the new value and forces all others to update their copies
 - ▶ Higher bus traffic
- ▶ Write-invalidate:
 - ▶ Writing processor forces all others to invalidate their copies
 - ▶ The new value is provided to others when requested or when flushed from cache

Coherence mechanisms

Broadcast-based:

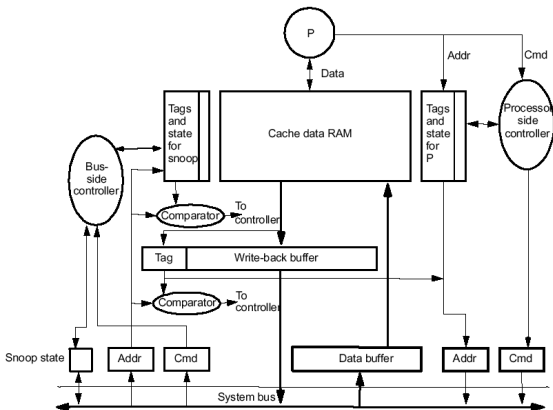
- ▶ Every cache that has a copy of a line from physical memory keeps its sharing status (status distributed)
- ▶ Broadcast medium (e.g. a bus) used to make all transactions visible to all caches and define **ordering**
- ▶ Caches monitor (snoop on) the medium and take action on relevant events (SCC: snoop cache controllers)



Directory-based: the sharing status of each line in memory is kept centralised in just one location (the directory) – studied later

Dual-ported caches to support coherence

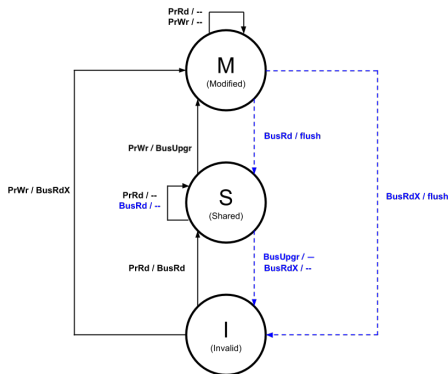
Listen to commands both from processor and from broadcast medium (e.g. bus)



Simple write-invalidate snooping protocol (MSI)

- ▶ A line in a cache memory can be in three different states:
 - ▶ Modified (M): dirty copy of the line
 - ▶ Shared (S): clean copy of the line
 - ▶ Invalid (I): invalidated copy of the line (not valid), or it does not exist in cache
- ▶ CPU events
 - ▶ PrRd (Processor read)
 - ▶ PrWr (Processor write)
- ▶ Bus transactions (caused by cache controllers)
 - ▶ BusRd: asks for copy with no intent to modify
 - ▶ BusRdX: asks for copy with intent to modify
 - ▶ BusUpgr: asks for permission to modify existing line, causes invalidation of other copies
 - ▶ Flush: puts line on bus, either because requested or voluntarily when dirty line in cache is replaced (WriteBack)

Simple write-invalidate snooping protocol (MSI)



- ▶ Who provides the line when requested via BusRd or BusRdX?
 - ▶ If line in S or I in other caches then main memory provides it
 - ▶ If line in M in another cache then this cache provides it (Flush)

Optional: MSI optimizations. Cache-to-cache transfers

- ▶ Does main memory need to be updated when flushing?
MOSI protocol adds O (Owned) state:
 - ▶ When flushing, state in cache for the line transitions from M to O (dirty since main memory is not updated)
 - ▶ Cache with line in O state is responsible for providing data when requested (not main memory)
 - ▶ Other caches maintain shared line in S state
 - ▶ Main memory updated when line in O is replaced from cache

Optional: MSI optimizations. Cache-to-cache transfers

- ▶ Does main memory need to supply data if already shared in another cache? **MSIF** protocol adds F (Forward) state:
 - ▶ In MSI which cache should provide the line if several copies?
 - ▶ Cache with line in F state is responsible for providing data when requested (not main memory)
 - ▶ Last cache asking for line transitions to F state (temporal locality), others transition/keep it S

Optional: MSI optimizations. Thread-private lines

- ▶ MSI requires two bus transactions for the common case of read followed by write, both from the same processor (no sharing at all)
 - ▶ Transaction 1: BusRd to move from I to S state
 - ▶ Transaction 2: BusUpgr to move from S to M state
- ▶ **MESI** protocol adds E (Exclusive) clean state:
 - ▶ Cache line in E if only one clean copy of the line
 - ▶ If write access by the same processor, the upgrade from E to M does not require a bus transaction (BusUpgr)
 - ▶ If line in E and another cache requests it then cache line state changes from E to S
- ▶ Combined use possible (MESIF/MOESI/MOESIF)

Minimizing sharing

- ▶ True sharing
 - ▶ Frequent writes to a variable can create a bottleneck
 - ▶ Sometimes multiple copies of the value, one per processor, are possible (e.g. the data structure that stores the freelist/heap for malloc/free)
- ▶ False sharing
 - ▶ Cache line may also introduce artefacts: two distinct variables in the same cache line
 - ▶ Technique: allocate data used by each processor contiguously, or at least avoid interleaving in memory
 - ▶ Example problem: an array of ints, one written frequently by each processor (many ints per cache line)

Outline

Uniprocessor parallelism

Symmetric multi-processor architectures

Multicore architectures

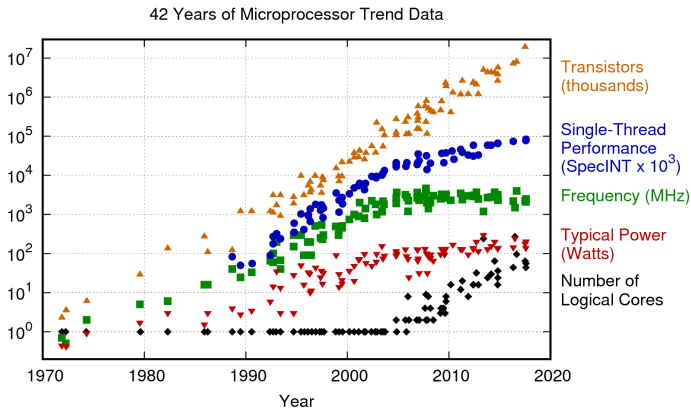
Non-Uniform Memory Architectures

Synchronization mechanisms

The memory consistency problem

Transistors, frequency, power, performance and ... cores!

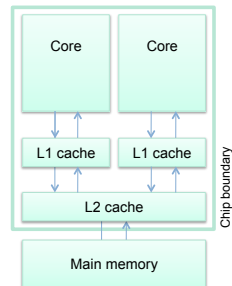
An inflexion point in 2004 ... the power wall



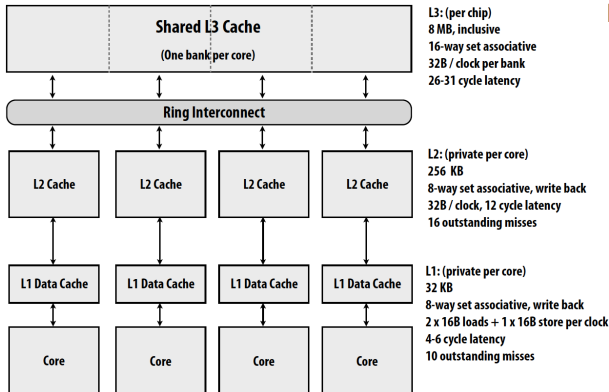
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Multicores

- ▶ The increasing number of transistors on a chip is used to accommodate multiple processors (cores) on a single chip
- ▶ Usually private caches (up to a certain cache level) and one last-level cache (LLC)
- ▶ Coherence maintained at the LLC level
- ▶ Chip or socket boundary, access to main memory
- ▶ Multicore = Chip Multi-Processor (CMP)

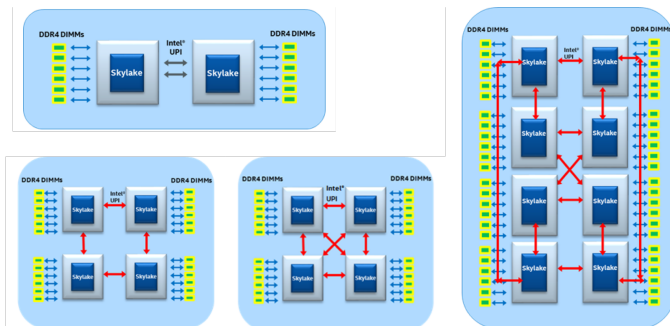


Example: multicore socket based on Intel Nehalem i7



Example: scalable multi-socket systems

Each socket is a multicore processor with a number of cores inside (e.g. SKL up to 24) and connected to memory (DDR DIMMs)



UPI/QPI ports to interconnect sockets and provide cache-coherent shared memory (but not uniform access time anymore!)

Outline

Uniprocessor parallelism

Symmetric multi-processor architectures

Multicore architectures

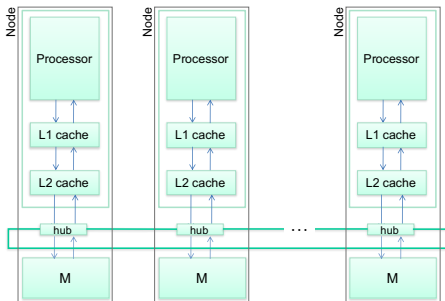
Non-Uniform Memory Architectures

Synchronization mechanisms

The memory consistency problem

Non-Uniform Memory Architectures (NUMA)

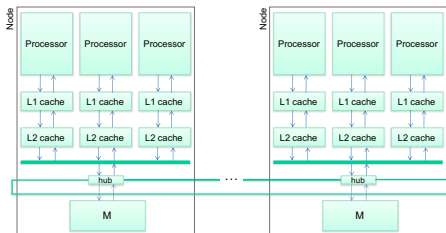
- ▶ Memory physically distributed but logically shared (coherent) by hardware; penalty: access time to memory not uniform
- ▶ Each node has a portion of the address space



- ▶ Hub: allows the interchange of data and coherence commands through an interconnection network

Non-Uniform Memory Architectures (NUMA)

- ▶ And in current systems each node may have several processors



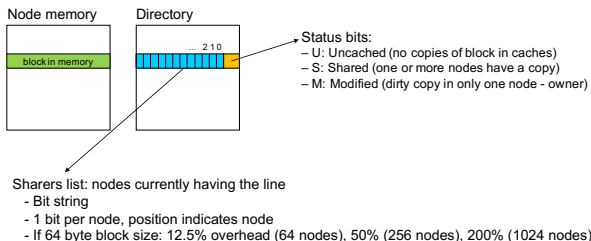
- ▶ Who is involved in maintaining coherence of a memory line?
 - ▶ **Home** node: node (memory of the node) where the line is allocated (OS managed, for example first touch)
 - ▶ **Remote** nodes: **Owner** node containing **dirty** copy or **Reader** nodes containing **clean** copies of the line
 - ▶ **Local** node: node with the processor requesting the line

Scaling of the broadcast mechanism

- ▶ Snooping schemes broadcast coherence messages to determine the state of a line in the other caches
 - ▶ Local node sends command to ALL nodes (home and remote, having or not copy of the line)
 - ▶ Could be extended to support coherence in small NUMA systems, but does not scale to large number of nodes (excessive coherence traffic)
- ▶ Alternative: avoid broadcast by storing information about the status of the line in one place: **Directory** in the Home node
 - ▶ The directory tracks the location of copies of memory line in caches, local node ONLY sends command to the home
 - ▶ Coherence is maintained by point-to-point messages between the home node (directory) and local/remote nodes

Directory-based cache coherency

- ▶ Directory structure associated to the node memory: one entry per line of memory
 - ▶ **Status bits:** they track the state of cache lines in its memory
 - ▶ **Sharers list:** tracks the list of remote nodes having a copy of a line. For small-scale systems, implemented as a bit string



- ▶ Directory is the centralised structure that "orders" the accesses to each line

Simplified coherency protocol

Possible commands arriving to home node from local node:

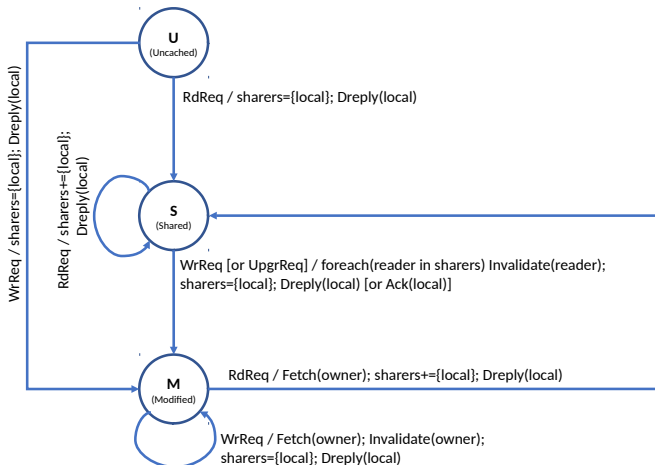
- ▶ **RdReq**: asks for copy of line with no intent to modify
- ▶ **WrReq**: asks for copy of line with intent to modify
- ▶ **UpgrReq**: asks for permission to modify an existing line, invalidating all other copies

As a result of **RdReq** and **WrReq** the home node sends clean copy of line (**Dreply** command to local node). For **UpgrReq** it sends an acknowledgment (**Ack** command).

If needed the home node may generate other commands to remote nodes:

- ▶ **Fetch**: asks remote (owner) node for a copy of line (**Dreply**)
- ▶ **Invalidate**: asks remote (reader) node to invalidate its copy, remote sends confirmation to home (**Ack**)

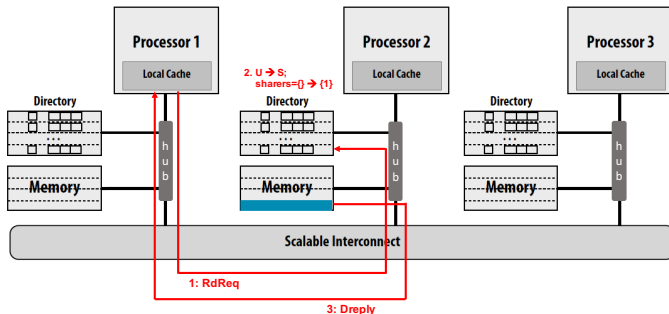
Simplified coherency protocol



Directory-based cache coherency: sequence of actions

Example 1: read miss to uncached line

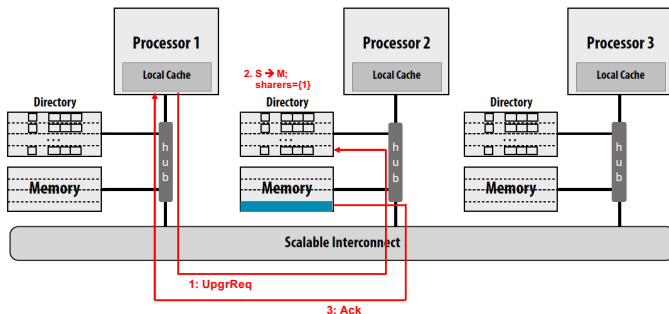
- ▶ Local node where the miss request originates: processor 1
- ▶ Home node where the memory line resides (clean): processor 2



Directory-based cache coherency: sequence of actions

Example 1 (cont.): write hit to previously cached (clean) line

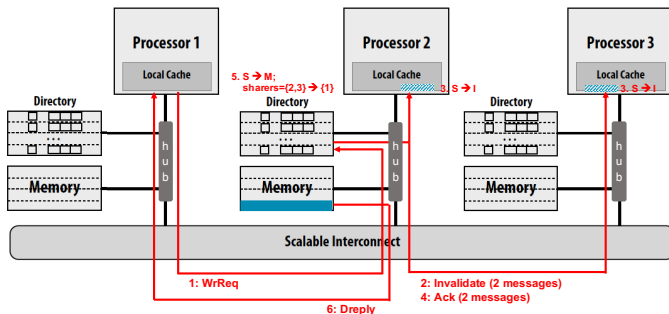
- ▶ Local node where the upgrade request originates: processor 1
- ▶ Home node where the memory line resides: processor 2



Directory-based cache coherency: sequence of actions

Example 2: write miss to clean line with two sharers

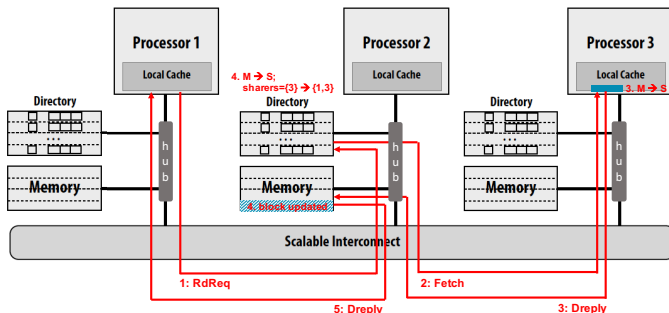
- ▶ Local node where the miss request originates: processor 1
- ▶ Home node where the memory line resides: processor 2
- ▶ Copies of line in caches of processors 2 and 3



Directory-based cache coherency: sequence of actions

Example 3: read miss to dirty line in remote (owner) node

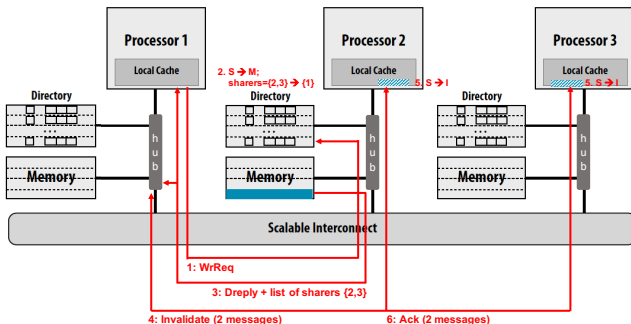
- ▶ Local node where the miss request originates: processor 1
- ▶ Home node for the memory line: processor 2
- ▶ Line dirty currently in cache of processor 3



Optional: directory-based protocol optimizations

Optimized protocols allow local nodes to perform coherence actions based on information provided by the home

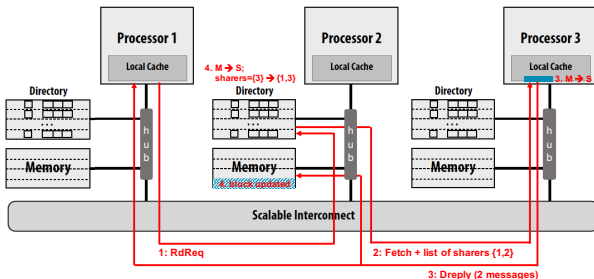
- Example 2: write miss to clean line with two sharers



Optional: directory-based protocol optimizations

Optimized protocols also allow owner node to directly provide data to local node.

- Example 3: read miss to dirty line in remote (owner) node

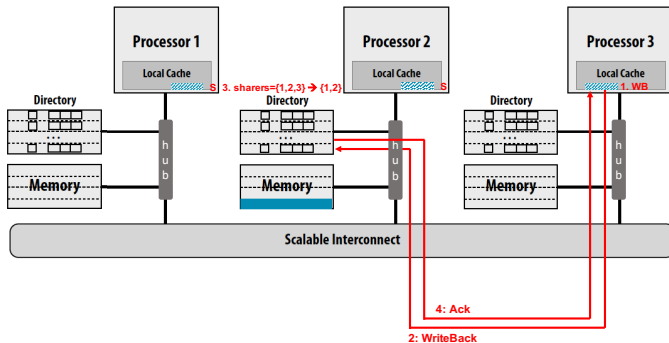


Simplified coherency protocol: additional issues (1)

- ▶ **WriteBack** command: send by local node when cache line is eliminated by its local cache replacement algorithm
 - ▶ If the line is clean, the directory still needs to be notified ...
 - ▶ ... so directory status for line transitions S to S (one less sharer in the list) or S to U (no sharers left after last one)
 - ▶ If the line is dirty, the home node will request the local node (Fetch) for the valid copy of line and update main memory ...
 - ▶ ... causing directory status for line to transition from M to U
 - ▶ In both cases, home node sends confirmation to local node (**Ack**)

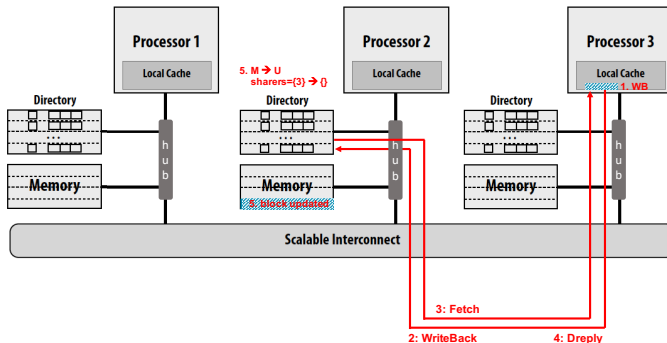
Directory-based cache coherency: sequence of actions

Example 4: replacement of a clean line in cache, multiple sharers



Directory-based cache coherency: sequence of actions

Example 5: writeback of a dirty line



Simplified coherency protocol: additional issues (2)

- ▶ If nodes have snoopy-based coherence, then the hub becomes an additional agent that interacts with the home (directory) nodes for the cache lines copied in the node
 - ▶ When BusRd, BusRdX or BusUpgr is snooped on the bus the hub sends RdReq, WrReq or UpgrReq, respectively, to the home
 - ▶ When Fetch is received by hub from home: BusRd placed on the bus, causing cache line status to transition from M to S and line to be flushed (Flush)
 - ▶ When Invalidate is received by hub from home: BusUpgr is placed on the bus, causing cache line status to transition from S to I
- ▶ Examples in the Collection of Exercises and/or Exams with Solutions

Outline

Uniprocessor parallelism

Symmetric multi-processor architectures

Multicore architectures

Non-Uniform Memory Architectures

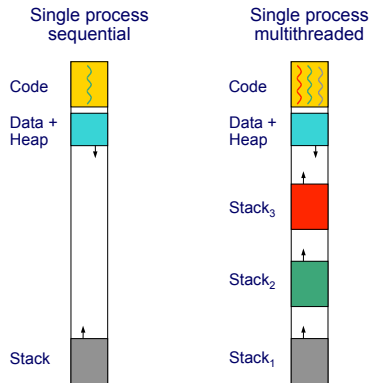
Synchronization mechanisms

The memory consistency problem

Shared memory: address space

Programmer needs

- ▶ Distribute work
- ▶ All threads can access data, heap and stacks
- ▶ Memory is not flat in a NUMA system
 - ▶ True and false sharing even more important
 - ▶ Data allocation and initialization sets the home node
 - ▶ Perform work according to data allocation to minimize data traffic
- ▶ Use synchronization mechanisms to avoid data races



Why synchronization?

- ▶ Needed to guarantee safety in the access to a shared-memory location or shared resource (e.g. mutual exclusion) or to signal a certain event (e.g. barrier)
- ▶ Components:
 - ▶ Acquire method: how thread attempts to gain access to shared location/resource
 - ▶ Waiting policy: how thread waits for access to be granted to shared location/resource: busy wait, block/awake, wait for a while and then block, ...
 - ▶ Release method: how thread enables other threads to gain access to location/resource once its access completes

Example: a simple, but incorrect, lock

- ▶ What's wrong with ...?
(assume flag initialized to 0, i.e. lock is free; flag equals one means lock is taken)

P1	P2
...	...
lock: ld r1, flag	lock: ld r1, flag
bnez r1, lock	bnez r1, lock
st flag, #1	st flag, #1
... // safe access	... // safe access
unlk: st flag, #0	unlk: st flag, #0
...	...

- ▶ Problem: data race because sequence load–test–store is not atomic!

Support for synchronization at the architecture level

- ▶ Need hardware support to guarantee atomic (indivisible) instruction to fetch and update memory
 - ▶ User-level synchronization operations (e.g. locks, barriers, point-to-point, ...) using these primitives

- ▶ test-and-set: read value in location and set to 1
Example: test-and-set based lock implementation

```
lock:  t&s r2, flag
      bnez r2, lock    // already locked?
      ...
unlock: st flag, #0    // free lock
```

Support for synchronization at the architecture level

- ▶ Atomic exchange: interchange of a value in a register with a value in memory

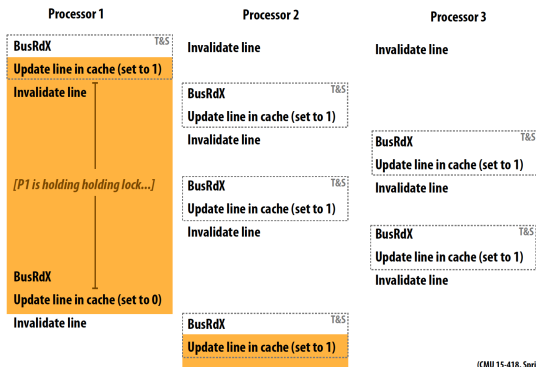
Example: atomic exchange based lock implementation

```
        mov r2, #1
lock:   exch r2, flag    // atomic exchange
        bnez r2, lock    // already locked?
        ...
unlock: st flag, #0      // free lock
```

- ▶ fetch-and-op: read value in location and replace with result after simple arithmetic operation (usually add, increment, sub or decrement)

test-and-set lock coherence traffic

```
lock:  t&s r2, flag      // test and acquire lock if free
       bnez r2, lock    // do it again if already locked
       ...
unlock: st flag, #0      // free the lock
```



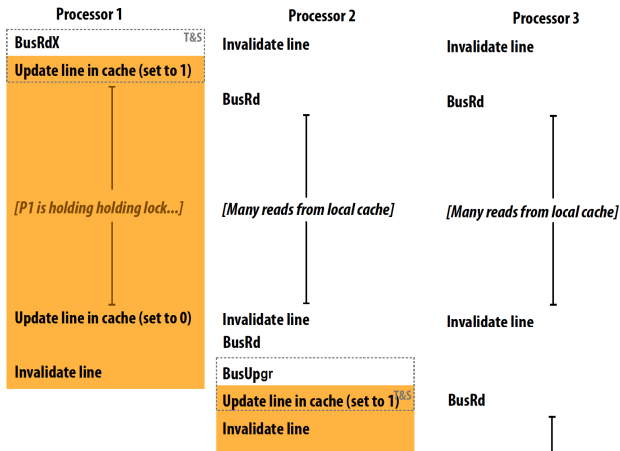
(CMU 15-418, Spring 2012)

Reducing synchronization cost: test-test-and-set

- ▶ test-test-and-set technique reduces the necessary memory bandwidth and coherence protocol operations required by a pure test-and-set based synchronization:
 - ▶ Wait using a regular load instruction (lock will be cached)
 - ▶ When lock is released, try to acquire using test-and-set

```
lock:  ld r2, flag           // test with regular load
                                   // lock is cached meanwhile it is not updated
        bnez r2, lock        // test if the lock is free
        t&s r2, flag         // test and acquire lock if STILL free
        bnez r2, lock
        ...
unlock: st flag, #0          // free the lock
```

test-test-and-set lock coherence traffic



Support for synchronization at the architecture level

- ▶ Atomicity difficult or inefficient in large systems. Alternative: **Load-linked Store-conditional ll-sc**
 - ▶ ll returns the current value of a memory location
 - ▶ sc stores a new value in that memory location if no updates have occurred to it since the ll; otherwise, the store fails
 - ▶ sc returns success (1) or failure (0)
- ▶ Examples implementing atomic exchange (left) and fetch-and-increment (right):

```
try: mov r3, r4
      ll r2, location
      sc r3, location
      beqz r3, try
      mov r4, r2
```

```
try: ll r2, location
      add r3, r2, #1
      sc r3, location
      beqz r3, try
```

Reducing synchronization cost: test-test-and-set

- ▶ test-test-and-set technique can also be implemented with ll-sc
 - ▶ First, wait using load linked instruction ll (lock will be cached)
 - ▶ Second, use store conditional sc operation to test if someone else did it first

```
lock:  ll r2, flag           // first test with load linked
                                     // lock is cached meanwhile it is not updated
                                     // test if the lock is free
        bnez r2, lock
        mov r2, #1
        sc r2, flag          // try to store 1
        beqz r2, lock        // repeat if someone else did it before me
        ...
unlock: st flag, #0          // free the lock
```

Reducing synchronization cost: test-test-and-set

- ▶ test-test-and-set **idea** can also help to reduce the synchronization cost of high level parallel programs
 - ▶ Non optimized version : the synchronization is always done

```
omp_set_lock(&lock);  
if (value<CONSTANT)    // Test  
    value++;           // Set (Assign)  
omp_unset_lock(&lock);
```

- ▶ Optimized version : the synchronization is done if any chance of doing "Set" operation

```
if (value<CONSTANT) {    // Test  
    omp_set_lock(&lock); // lock cost is only paid if necessary  
    if (value<CONSTANT)  // Test again  
        value++;         // Set (Assign)  
    omp_unset_lock(&lock);  
}
```

Other synchronization primitives

- ▶ How to implement a barrier synchronization primitive?
 - ▶ Threads arriving wait until all have reached the barrier
 - ▶ Structure with fields {lock, counter, flag}

```
barrier:
    acquire_lock(&barr.lock);
    if (barr.counter == 0)
        barr.flag = 0        // reset flag if first
    mycount = barr.counter++;
    release_lock(&barr.lock);

    if (mycount == P) {       // last to arrive?
        barr.counter = 0     // reset counter for next barrier
        barr.flag = 1       // release waiting processors
    } else
        while (barr.flag == 0) // busy wait for release
        ...
```

- ▶ Does it work when consecutive barriers appear? Try to solve it

Outline

Uniprocessor parallelism

Symmetric multi-processor architectures

Multicore architectures

Non-Uniform Memory Architectures

Synchronization mechanisms

The memory consistency problem

Consistency

- ▶ In current systems, the compiler and hardware can freely reorder operations to different memory locations, as long as data/control dependences in sequential execution are guaranteed. This enables:
 - ▶ Compiler optimizations such as register allocation, code motion, loop transformations, ...
 - ▶ Hardware optimizations, such as pipelining, multiple issue, write buffer bypassing and forwarding, and lockup-free caches, ...

all of which lead to overlapping and reordering of memory operations

Consistency: example 1

- ▶ Will writes to different locations be seen in an order that makes sense, according to what is written in the source code?
- ▶ Example: two processors are synchronizing on a variable called flag. Assume A and flag are both initialized to 0

P1	P2
<pre>A=1; flag=1;</pre>	<pre>while (flag==0); /*spin*/ print A;</pre>

- ▶ What value does the programmer expect to be printed?

Consistency: example 2

- ▶ Will writes from one core be seen in a different core, according to what is written in the source code?
- ▶ For example, synchronisation through a shared variable (`next` is implicitly shared):

```
int next = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    for (int end = 0; end == 0; ) {
        ...
        next++;
        if (next==N) end=1;
    }
}
```

```
#pragma omp task
{
    int mynext = 0;
    for (int end = 0; end == 0; ) {
        while (next <= mynext);
        ...
        mynext++;
        if (mynext==N) end=1;
    }
}
```


Consistency: example 2 (cont.)

- ▶ Will writes from one core be seen in a different core, according to what is written in the source code?
- ▶ For example, synchronisation through a shared variable (`next` is implicitly shared):

```
int next = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    for (int end = 0; end == 0; ) {
        ...
        next++;
        #pragma omp flush(next)
        if (next==N) end=1;
    }
}
```

```
#pragma omp task
{
    int mynext = 0;
    for (int end = 0; end == 0; ) {
        while (next <= mynext) {
            #pragma omp flush(next)
            ;
        }
        ...
        mynext++;
        if (mynext==N) end=1;
    }
}
```

Memory consistency model

The memory consistency model ...

- ▶ Provides a formal specification of how the memory system will appear to the programmer ...
- ▶ ... by placing restrictions on the reordering of shared-memory operations

Sequential consistency, easy to understand but it may disallow many hardware and compiler optimizations that are possible in uniprocessors by enforcing a strict order among shared memory operations.

Memory consistency model

Relaxed consistency (weak), specifying regions of code within which shared-memory operations can be reordered

- ▶ fence machine instruction to force all pending memory operations to complete
- ▶ `#pragma omp flush` and other implicit points in OpenMP language

Different possibilities and implementations to be studied in *Multiprocessors* course

Parallelism (PAR)

Introduction to (shared-memory) parallel architectures

Eduard Ayguadé, José Ramón Herrero and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2019/20 (Fall semester)