Parallelism (PAR)

Introduction to (shared–memory) parallel architectures

Eduard Ayguadé, Julita Corbalán,
Daniel Jiménez and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2017/18 (Spring semester)

Outline

Uniprocessor parallelism

Symmetric multi–processor architectures

Multicore architectures

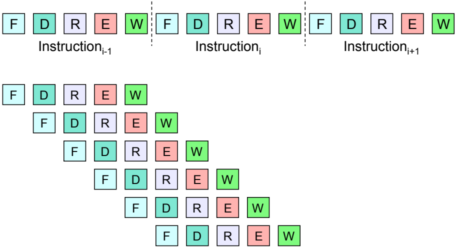
Non-Uniform Memory Architectures

Synchronization mechanisms

The memory consistency problem

Pipelining

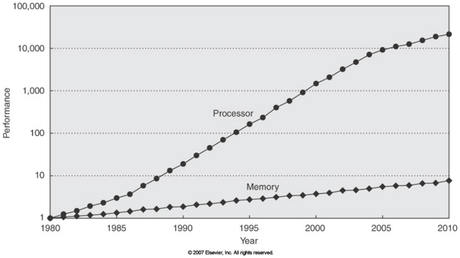
- Execution of single instruction divided in multiple stages
- Overlap the execution of different stages of consecutive instructions
- Ideal: IPC=1 (1 instruction executed per cycle)

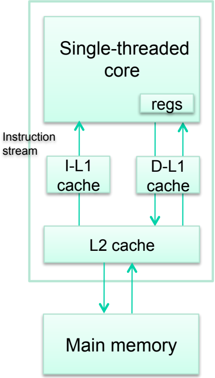


- IPC<1 due to hazards (structural, data, control), preventing the execution of an instruction in its designated clock cycle

Memory hierarchy

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





- Size vs. access time
- Non-blocking design

Memory hierarchy

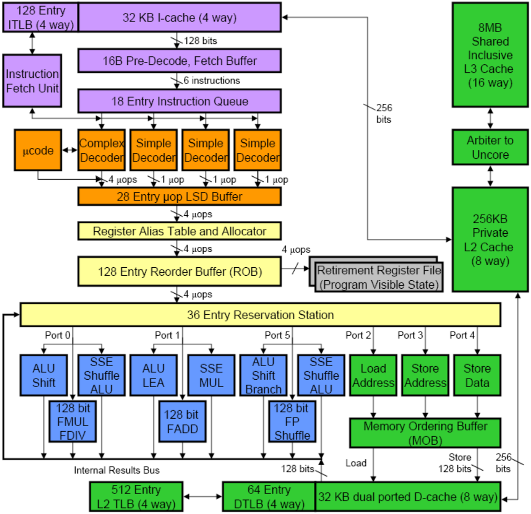
- The principle of locality: if an item is referenced ...
  - Temporal locality: ... it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial locality: ... items whose addresses are close by tend to be referenced soon (e.g., straight line code, array access)
- Line (or block)
  - A number 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

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)
  - Multithreaded architecture[1]: 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

[1] Hyperthreading in Intel terminology

Current uniprocessor architecture: Intel Nehalem i7



Who exploits this uniprocessor parallelism?

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

- Software pipelining to statically schedule ILP
- Unrolling to allow the processor to exploit ILP dynamically
- Data contiguous in memory and aligned to efficiently exploit DLP
- 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

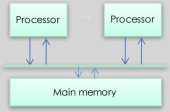
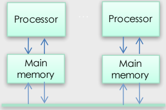
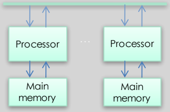
Multicore architectures

Non-Uniform Memory Architectures

Synchronization mechanisms

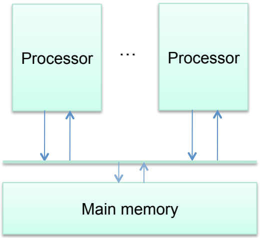
The memory consistency problem

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	• 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	• DSM (Distributed-Shared Memory architecture • NUMA (Non-Uniform Memory Access) architecture
	Multiple separate address spaces		Explicit messages through network interface card	• Message-passing multiprocessor • Cluster Architecture • Multicomputer

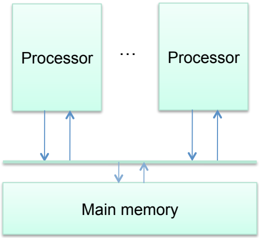
Symmetric multi–processor architectures

- 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)



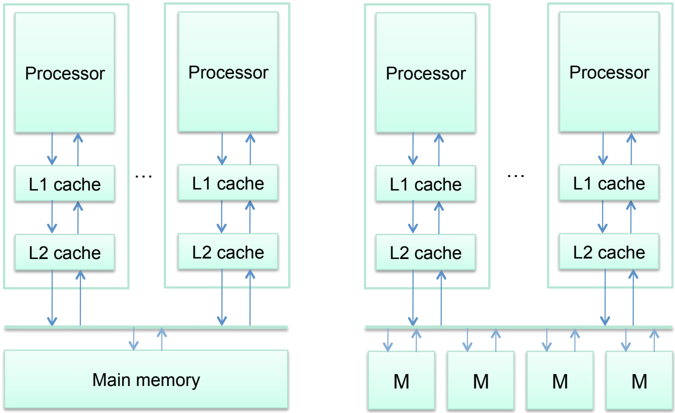
Symmetric multi–processor architectures

- 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

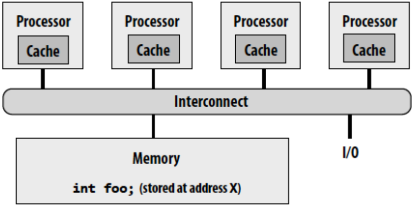


Symmetric multi–processor architectures

Local caches and multi-banked (interleaved) memory



The coherence problem



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

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

(CMU 15-418, Spring 2012)

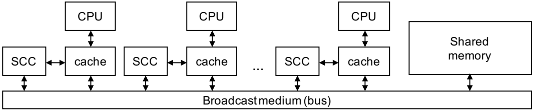
Coherence protocols

- 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 when flushed from cache

Coherence mechanisms

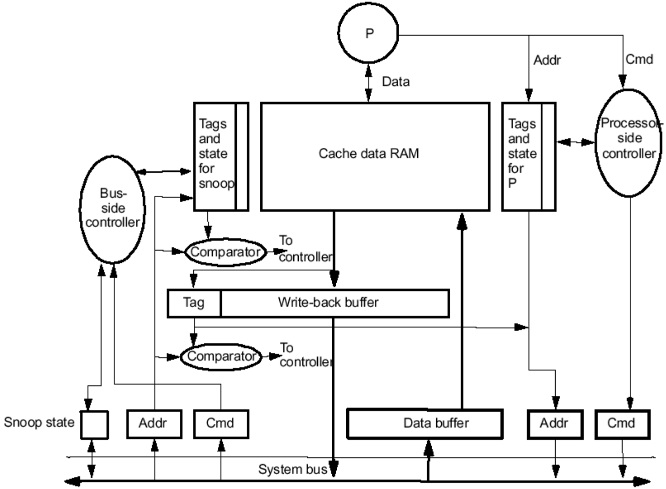
Snooping:

- Every cache that has a copy from a block in 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 or **snoop on the medium** and take action on relevant events (e.g. change status)



Directory-based: the sharing status of each block in memory is kept in just one location (the directory) – to be studied later.

Coherence mechanisms

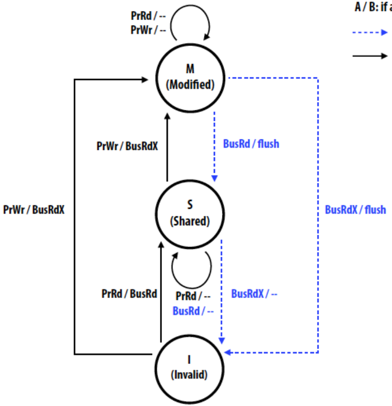


MSI write-invalidate snooping protocol

- States
  - Modified (M): only one copy, written (dirty)
  - Shared (S): one or more copies, all read (clean)
  - Invalid (I): not valid
- 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[2]
  - Flush: puts data on bus (either requested by another cache or voluntarily due to cache replacement – write back)

[2] Sometimes BusUpgr is also included to simply ask for permission to modify.

MSI write-invalidate snooping protocol



A / B: if action A is observed by cache controller, action B is taken

- - - -▸ Broadcast (bus) initiated transaction

⟶ Processor initiated transaction

Alternative state names:

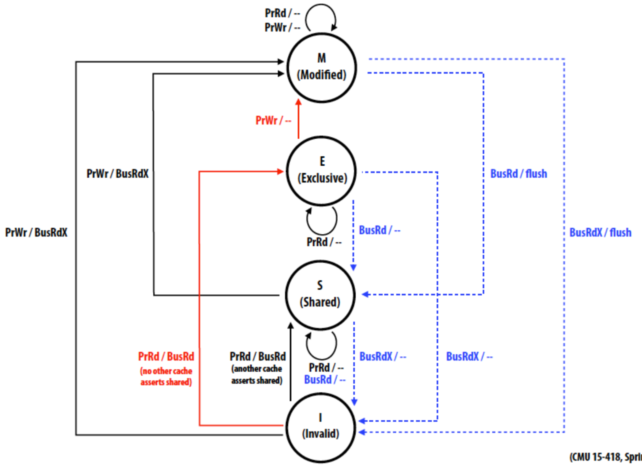
- E (exclusive, read/write access)
- S (potentially shared, read-only access)
- I (invalid, no access)

(CMU 15-418, Spring 2012)

New state for MSI: Exclusive

- MSI requires two bus transactions for the common case of reading data, and later writing to it
  - Transaction 1: BusRd to move from I to S state
  - Transaction 2: BusRdX to move from S to M state
- This inefficiency exists even if application has no sharing at all
- Solution: add additional state E (Exclusive clean)
  - Line not modified, but only this cache has copy
  - Decouples exclusivity from line ownership (line not dirty, so copy in memory is valid copy of data)
  - Upgrade from E to M does not require a bus transaction

MESI write-invalidate snooping protocol



(CMU 15-418, Spring 2012)

MESI: increasing efficiency and complexity

- Does main memory needs to be updated when flushing?
  - **MOESI** protocol adds O (Owned, but not exclusive) state: one cache maintains line in O state, other caches maintain shared line in S state
- Does main memory need to supply data if already in E or S in another cache?
  - No, but if more than one, which cache should provide it?
  - **MESIF** protocol adds F (Forward) state: one cache holds shared line in F state rather than S[3]
- Cache-to-cache transfers: cache in O or F state is responsible for servicing when required by another cache

[3] Usually F state migrates to last cache that loads the line, why?

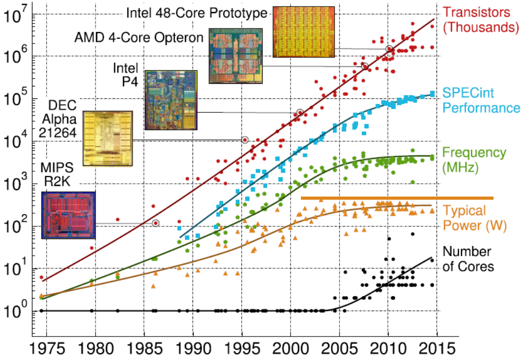
Minimizing sharing

- 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 block may also introduce artefacts: two distinct variables in the same cache block
  - 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

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

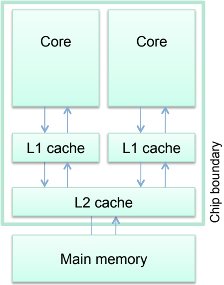
An inflexion point in 2004 ... the power wall[4].



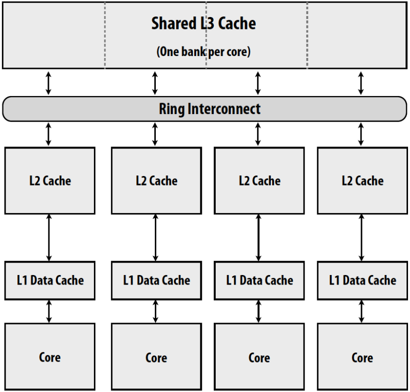
4 Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten.

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 based on Intel Nehalem i7



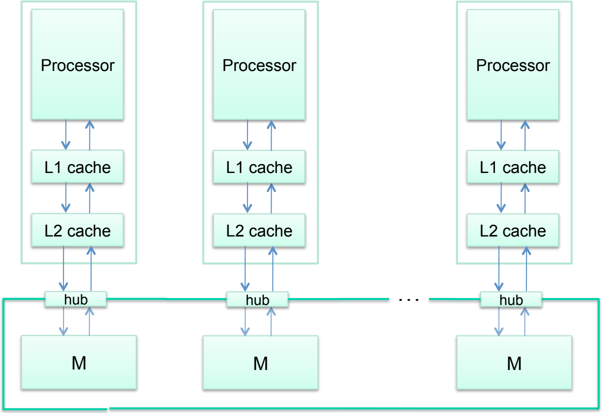
L3: (per chip)
8 MB, inclusive
16-way set associative
32B / clock per bank
26-31 cycle latency

L2: (private per core)
256 KB
8-way set associative, write back
32B / clock, 12 cycle latency
16 outstanding misses

L1: (private per core)
32 KB
8-way set associative, write back
2 x 16B loads + 1 x 16B store per clock
4-6 cycle latency
10 outstanding misses

Outline

Non-Uniform Memory Architectures (NUMA)



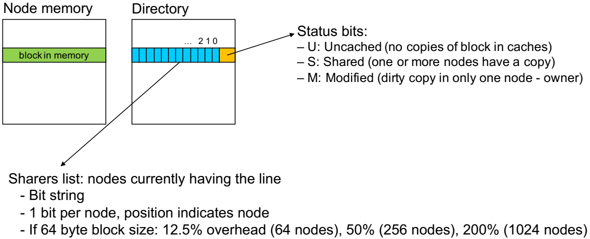
- Hub enables cache-coherent NUMA

Directory-based cache coherency

- Who is involved in maintaining coherence of a memory block?
  - **Home** node: node (memory of the node) where the block is allocated (OS managed, for example first touch)
  - **Remote** nodes: **Owner** node containing **dirty** copy or **Reader** nodes containing **clean** copies of the block
  - **Local** node: node containing the processor requesting the block
- An additional structure is necessary to track the location of copies of memory block in caches: **Directory**
- Coherence is maintained by point-to-point messages (not broadcast) between Local/Remote nodes and the directory in the Home node

Directory-based cache coherency

- Directory structure associated to the node memory: one entry per block 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 block. For small-scale systems, implemented as a bit string



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

Simplified coherency protocol

Possible commands arriving to home node from local node:

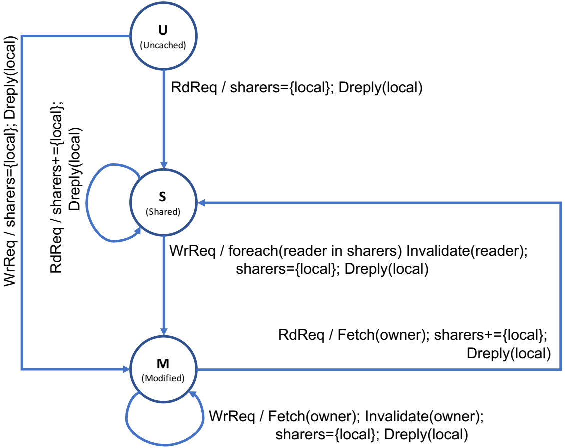
- RdReq: Asks for copy of block with no intent to modify
- WrReq: Asks for copy of block with intent to modify (miss in cache of local node) or simply asks for permission to modify it (hit in cache of local node)
- Dreply: Sends clean copy of block

In response to that, home node[5] may generate other commands to remote nodes:

- Fetch: Asks remote (owner) node for a copy of block
- Invalidate: Asks remote node to invalidate its copy

[5] In other protocols local nodes perform coherence actions based on information provided by the home and owner directly provides data to local.

Simplified coherency protocol



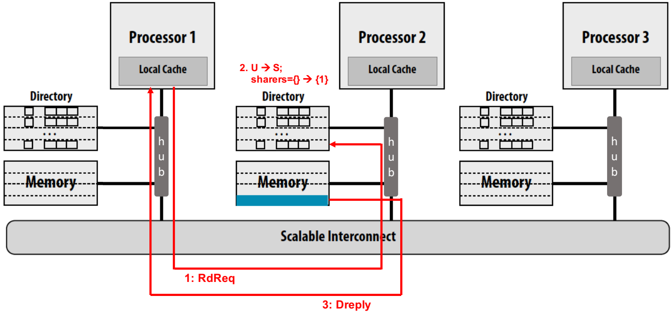
Simplified coherency protocol: additional issues

- **WriteBack** command: send by local node when cache line is replaced due to cache conflict
  - Directory status for block transitions from M to U (block in home node needs to be updated), S to S (one less sharer) or S to U (no sharers left after last one)
- The MSI cache state graph is leveraged to respond to the commands sent from the directory
  - When **Fetch** is received: line status transitions from M to S and line is flushed
  - When **Invalidate** is received: line status transitions from S (or M) to I

Directory-based cache coherency: sequence of actions

Example 1: read miss to uncached block

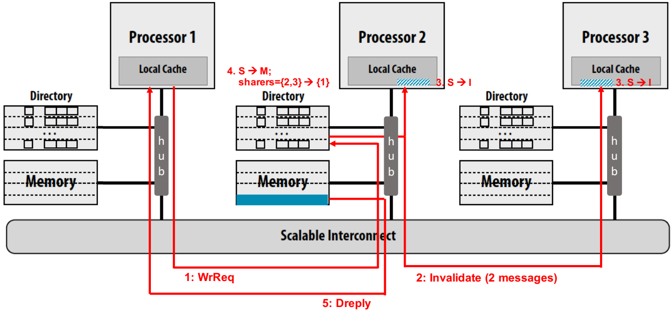
- Local node where the miss request originates: processor 1
- Home node where the memory block resides (clean): processor 2



Directory-based cache coherency: sequence of actions

Example 2: write miss to clean block with two sharers

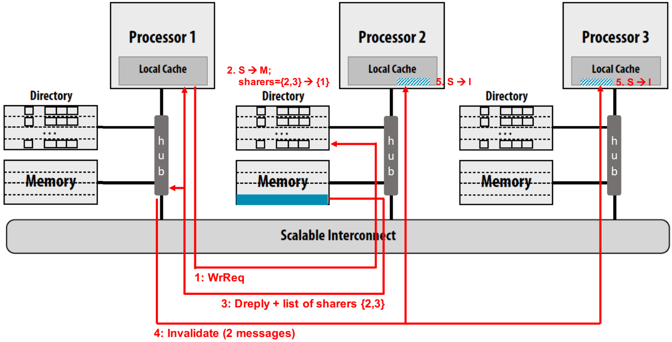
- Local node where the miss request originates: processor 1
- Home node where the memory block resides: processor 2
- Copies of block in caches of processors 2 and 3



Directory-based cache coherency: optimized (optional)

Example 2: write miss to clean block with two sharers

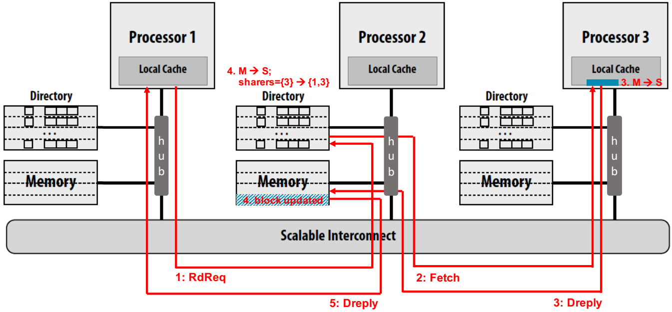
- Local node where the miss request originates: processor 1
- Home node where the memory block resides: processor 2
- Copies of block in caches of processors 2 and 3



Directory-based cache coherency: sequence of actions

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

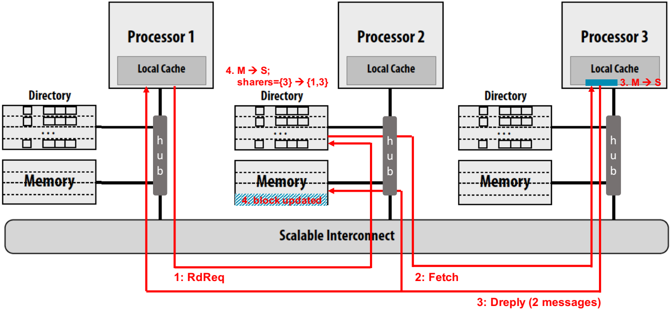
- Local node where the miss request originates: processor 1
- Home node for the memory block: processor 2
- Block dirty currently in cache of processor 3



Directory-based cache coherency: optimized (optional)

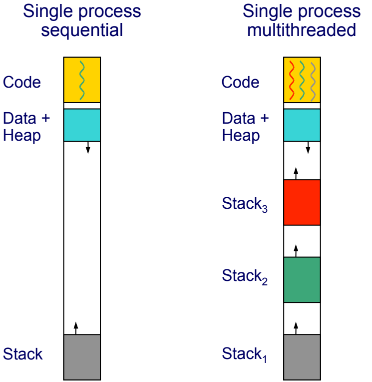
Example 3: read miss to dirty block in remote (owner) node

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



Outline

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

```
        daddui r2, r0, #1 // r0 always equals 0
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)

Support for synchronization at the architecture level

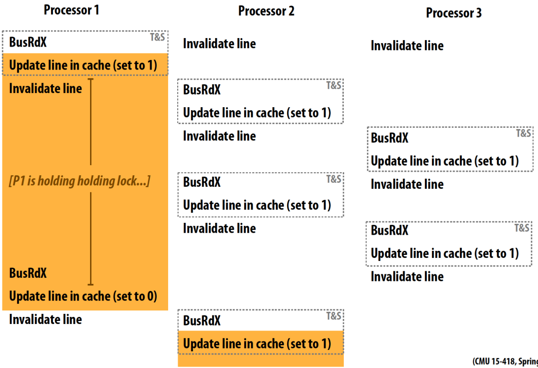
- Atomicity difficult or inefficient in large systems. Alternative: **Load-linked Store-conditional** `ll-sc`
  - `ll` returns the current value of a memory location
  - `sc` stores a new value in that memory location if no updates have occurred to it since the `ll`; otherwise, the store fails
  - `sc` returns success in doing store
- 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
     daddui r3, r2, #1
     sc r3, location
     beqz r3, try
```

test-and-set lock coherence traffic

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



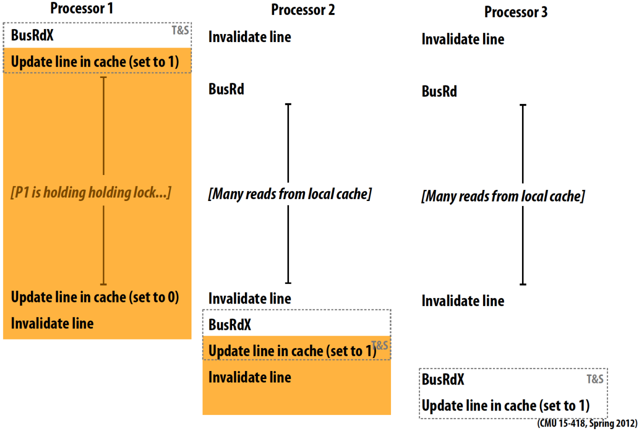
(CMU 15-418, Spring 2012)

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



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
        bnez r2, lock               // test if the lock is free
        daddui r2, r0, #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**[6] can also help to reduce the synchronization cost of high level parallel programs
  - Non optimized version : the synchronization is always done

```
acquire_lock(&lock);
if (value<CONSTANT)        // Test
     value++;              // Set (Assign)
release_lock(&lock);
```

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

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

[6] Note that `acquire_lock` implementation may also use the `test-test-and-set` technique to reduce the synchronization cost

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

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

A=1;                  while (flag==0); /*spin*/
flag=1;               print A;
```

  - 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, Julita Corbalán,
Daniel Jiménez and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2017/18 (Spring semester)