Parallelism (PAR)

Introduction to (shared–memory) parallel architectures

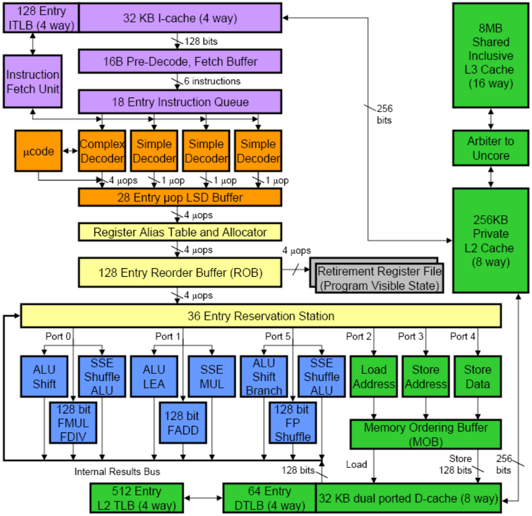
Eduard Ayguade, Josep Ramon Herrero and Daniel Jiménez
({eduard,josepr,djimenez}@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya

2014/15-Fall

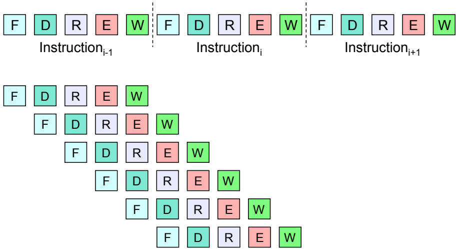
Outline

Current uniprocessor architecture: Intel Nehalem i7



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

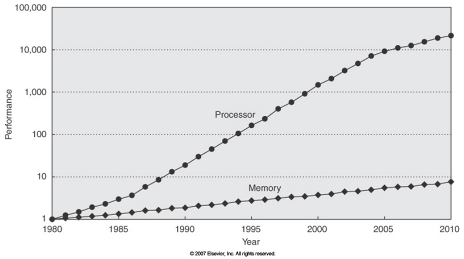
Sources of parallelism in uniprocessors

- 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)
  - 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

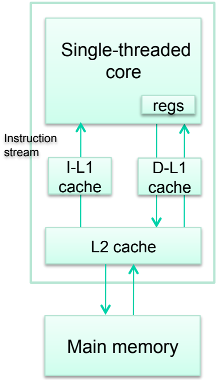
[1] Hyperthreading in Intel terminology

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

Who exploits this uniprocessor parallelism?

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

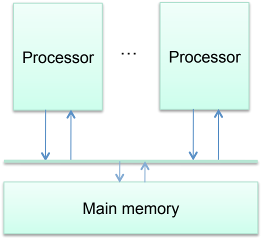
- Software pipelining to statically schedule ILP
- Unrolling to allow the processor to exploit ILP dynamically
- Data contiguous in memory and aligned to efficiently exploit DLP
- Blocking (or tiling) to define a problem that fits in register/L1-cache/L2-cache (temporal locality)
- ...

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

Outline

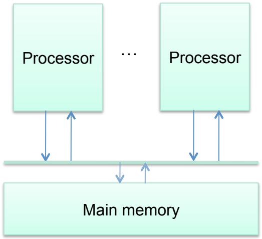
Symmetric multi–processor architectures

- 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)



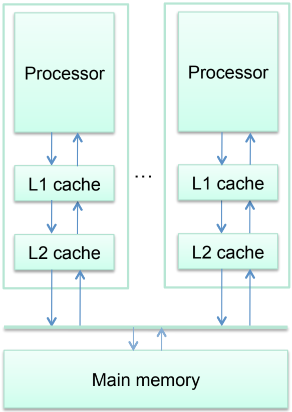
Symmetric multi–processor architectures

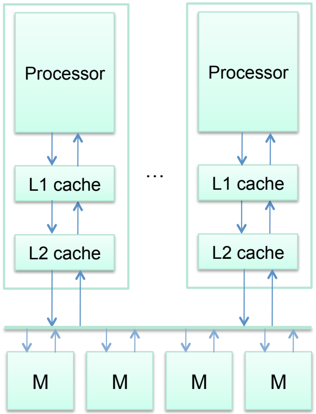
- Uniform Memory Access (UMA)
  - Access to shared data with load/store instructions
  - Access time to a memory location is independent of which processor makes the request or which memory chip contains the data
- The bottleneck in the scalability of SMP is the 'bandwidth' of the interconnection network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



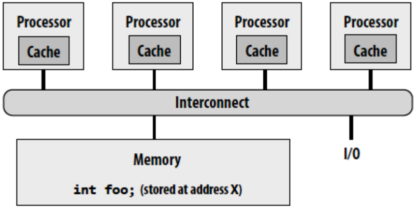
Symmetric multi–processor architectures

Local caches and multi-banked (interleaved) memory





The coherence problem



Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	$\boxed{0}$ miss				0
P2 load X	0	$\boxed{0}$ miss			0
P1 store X	1	0			0
P3 load X	1	0	$\boxed{0}$ miss		0
P3 store X	1	0	2		0
P2 load X	1	$\boxed{0}$ 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 **

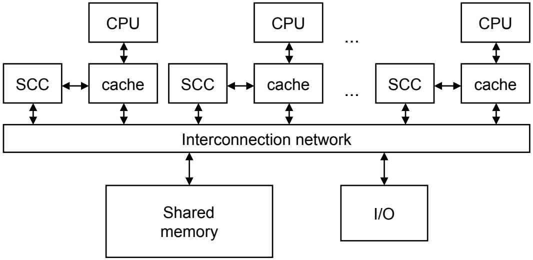
** Assumes write-back cache behavior

(CMU 15-418, Spring 2012)

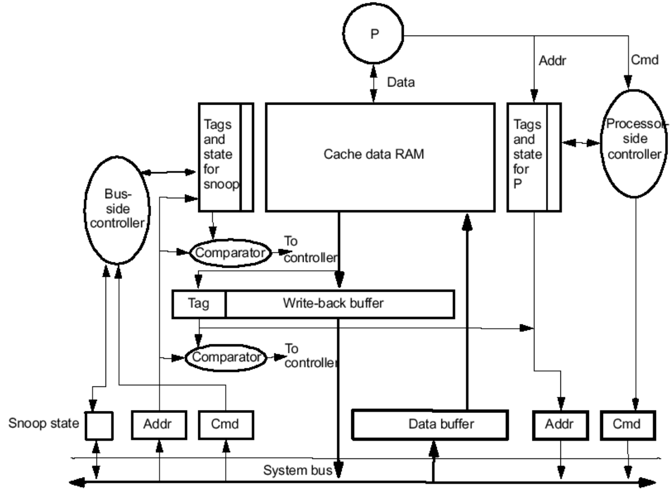
Coherence protocols

Snooping-based coherence protocol: bus is a broadcast medium

- Transactions on bus are visible to all processors
- Processors or their representatives can snoop (monitor) the bus and take action on relevant events (e.g. change state)



Coherence protocols



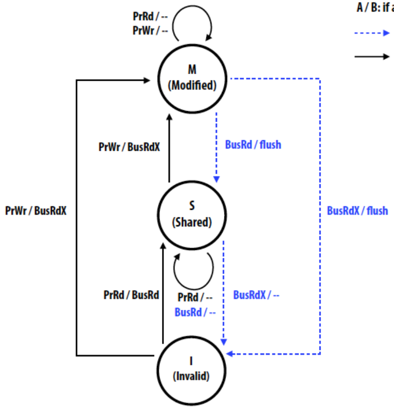
Coherence protocols

- Write-update: writing processor broadcasts the new value and forces all others to update their copies
  - New data appears sooner in caches and main memory ...
  - ... but, higher bus traffic
- Write-invalidate: writing processor forces all others to invalidate their copies
  - Data is updated when flushed from cache or requested
  - Dirty in cache state now indicates exclusive ownership:
    - Exclusive: only cache with a valid copy (subsequent writes to same block do not need to broadcast invalidate)
    - Owner: responsible for supplying block upon a request for it
- To be studied in detail in later courses

Example: write-invalidate, Snoopy with MSI protocol

- States
  - Dirty or Modified (M): one only
  - Shared (S): one or more
  - Invalid (I) or not exists (miss)
- CPU events
  - PrRd (Processor read)
  - PrWr (Processor write)
- Bus transactions
  - BusRd: asks for copy with no intent to modify
  - BusRdX: asks for copy with intent to modify (or invalidates other copies in case it is already available)
  - Flush: updates memory

Example: write-invalidate, Snoopy with MSI protocol



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

- - - -▶ Broadcast (bus) initiated transaction

——▶ Processor initiated transaction

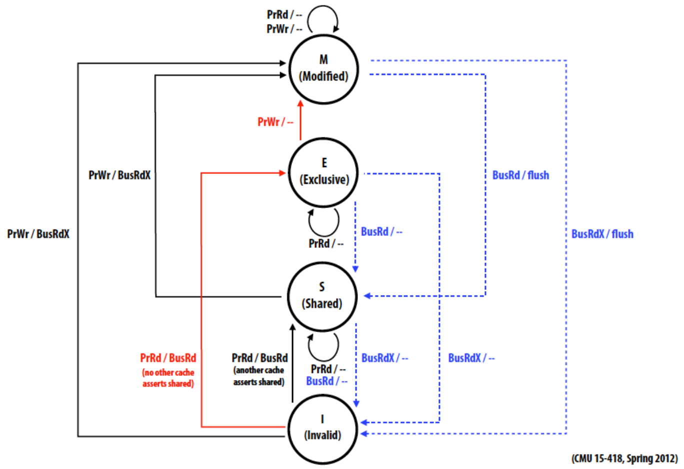
Alternative state names:
- E (exclusive, read/write access)
- S (potentially shared, read-only access)
- I (invalid, no access)

(CMU 15-418, Spring 2012)

Example: write-invalidate, Snoopy with MSI protocol

- 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

Example: MESI protocol



MESI: cache–to–cache transfers

- Who should supply data on a cache miss when line is in the E or S state of another cache?
  - Can get data from memory or can get data from another cache
  - If more than one, which cache should provide it?
- Cache-to-cache transfers add complexity, but commonly used today to reduce both latency of access and memory bandwidth requirements

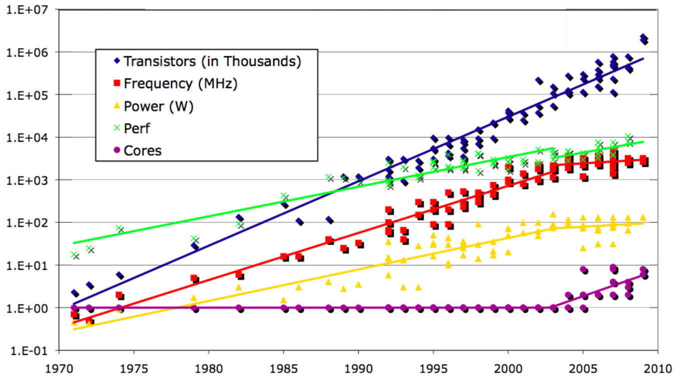
Minimizing sharing

- 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 artifacts: 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!

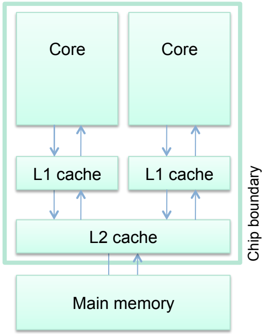
An inflexion point in 2004 ... the power wall[2].



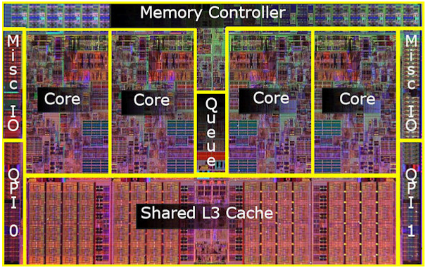
[2] Data from K. Olukotun, L. Hammond, H. Sutter, B. Smith, C. Batten, and K. Asanovic.

Multicore era

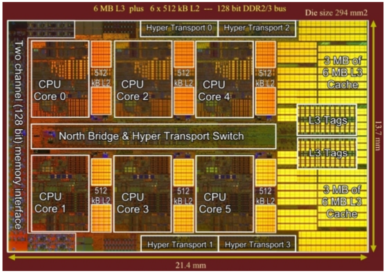
- The number of transistors on a chip is continuing to increase to accommodate multiple processors (cores) on a single chip
- Multicore = Chip Multi-Processor (CMP)



Multicore era (examples)

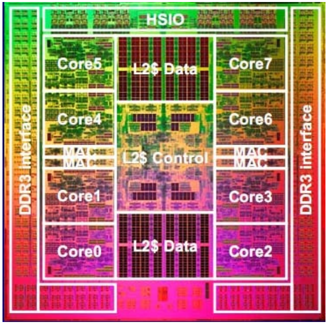


Intel's Nehalem i7 4-core (2 threads/core), 32/32 KB L1 and 256 KB L2 per core. Unified 4 or 8 MB L3

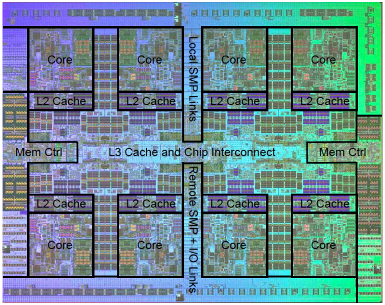


AMD's Istanbul 6-core, 512KB L2 per core and 6 MB shared L3

Multicore era (examples)



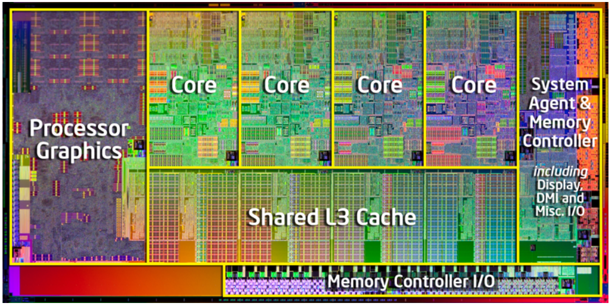
Sun's SPARC64-VIIIfx 8-core, 32/32K L1 per core and 5 MB shared L2. 2 GHz.. 760 M transistors



IBM's Power7 8-core (4 threads/core), 32/32K L1 and 256KB L2 per core. 32 MB shared L3. 2.4 GHz. 1.2 billion tansistors

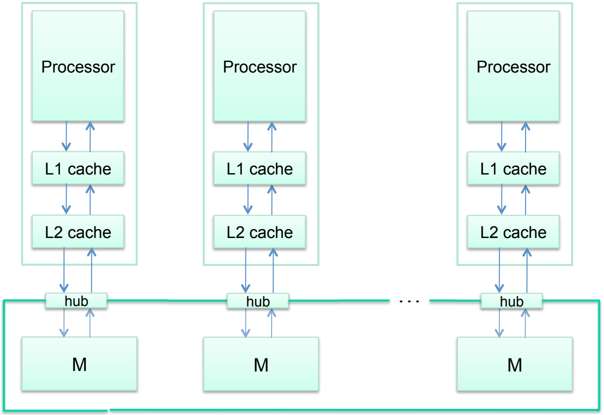
Multicore era (examples)

- Homogeneous multi-core systems include only identical cores, heterogeneous multi-core systems have cores with different functionalities, performances, ... (e.g. Intel Sandy Bridge)



Outline

Non-Uniform Memory Architectures (NUMA)



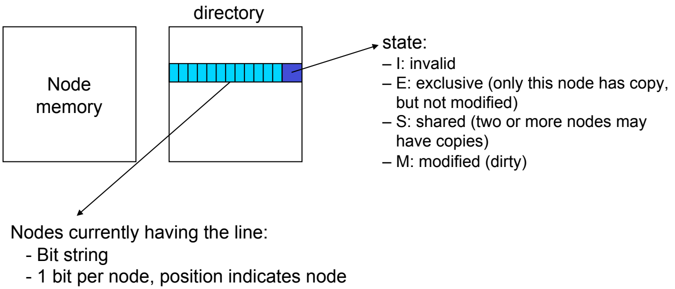
- Hub enables cache-coherent NUMA

Directory-based cache coherency protocol

- Scalability: coherence is maintained by point-to-point messages between the caches, not by broadcast mechanisms
  - **Home** node of a line: node (memory of the node) where the line is allocated (OS managed, for example first touch)
  - **Local** (requesting) node: node containing processor requesting a block
  - Other nodes may be involved: **Owner** node containing **dirty** copy or **Reader** nodes containing **clean** copies
- An additional structure is necessary to track the location of cache copies: **directory**

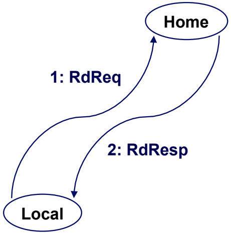
Directory-based cache coherency protocol

- Directory structure associated to each node: one directory entry per cache line of memory
  - State bits: they track the state of cache lines in its memory
  - Presence bits: if a block is shared, it must track which processors have it
  - If a block is valid only in one node, it should know which one


directory
Node memory
state:
– I: invalid
– E: exclusive (only this node has copy, but not modified)
– S: shared (two or more nodes may have copies)
– M: modified (dirty)
Nodes currently having the line:
- Bit string
- 1 bit per node, position indicates node

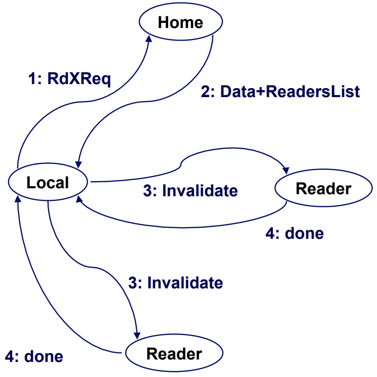
Directory-based cache coherency

- Example 1: read miss to clean line
  - Local node where the miss request originates
  - Home node where the cache line resides

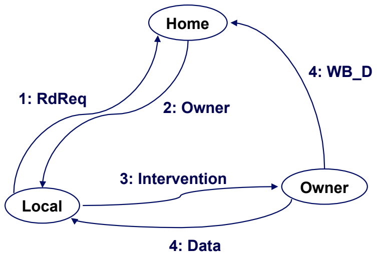


Directory-based cache coherency

Example 2: Access to clean copy, with two readers, for write

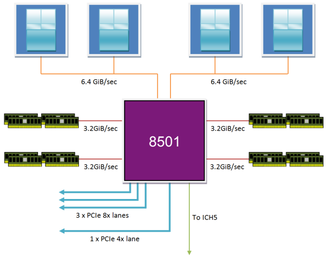


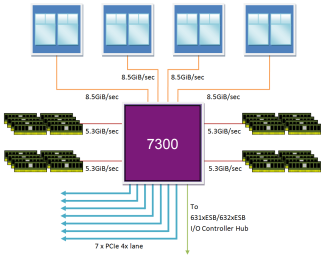
Example 3: Access to dirty block for read



Front-Side Bus (FSB)

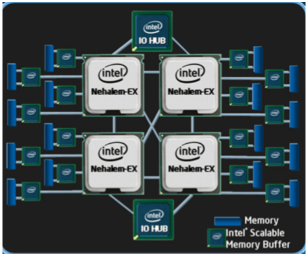
- Processor-northbridge (memory and I/O Hubs) connection (64-bit, 1-4 transfers/cycle, i.e. 6.4 GB/s @ 200 MHz)
- A number of processors can be connected to a single FSB sharing its total bandwidth
- Northbridge with several FSB ports ("snoop filters" to isolate FSB ports)





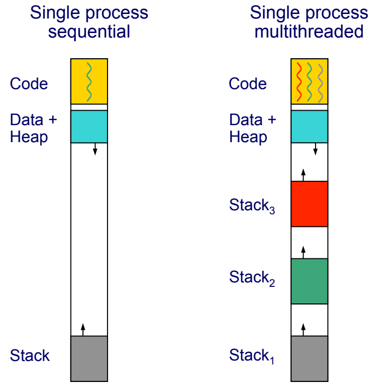
Quick Path Interconnect (QPI) and Hypertransport (HT)

- Point-to-point processor interconnects (16-bit data, 2 transfers/cycle, bidirectional, i.e. 25.6 GB/s @ 3.2 GHz)
- Connect several processors with integrated memory controllers and/or chipsets (IO): non-uniform memory access (NUMA)



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 to signal a certain event
- Components:
  - Acquire method: acquire the right to do the synchronization
  - Waiting policy: busy wait, block/awake, wait for a while and then block, ...
  - Release method: allow others to proceed
- What's wrong with ...? (assume flag initialized to 0)

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

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

```
        lock:  t&s r2, flag
               bnez r2, lock     // already locked?
```

- Atomic exchange: interchange of a value in a register with a value in memory
- `fetch-and-op`: read value in location and replace with increment/decrement

Support for synchronization at the architecture level

- Atomicity difficult or inefficient in large systems. Idea: assume optimistic atomic access
- 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 `sc` is guaranteed to fail

```
atomic: ll r1, location
      r2 = f (r1)                 // f can be: "r2=1", "r2=r1", "r2=r1+1", ...
      sc location, r2
      beqz  atomic                // Z flag set to 1 if atomicity has occurred
```

Other synchronization primitives

- How to implement a barrier synchronization primitive?
  - Processors block until all have reached it
  - Structure with fields {lock, counter, flag}

```
      ...
lock: t&s r2, barrier.lock        // acquire lock
      bnez r2, lock
      if (barr.counter == 0)
         barr.flag = 0            // reset flag if first
      mycount =  barr.counter++;
      barr.lock = 0               // release lock
      if (mycount == P) {         // last to arrive?
         barr.counter =0          // reset for next barrier
         barr.flag = 1            // release waiting processors
      } else
         while (barr.flag == 0)   // busy wait for release
      ...
```

Reducing Synchronization Cost: `test-test-and-set`

- Acquire lock using `test-and-set` based synchronization

```
      ...
lock: t&s r2, barrier.lock        // test and acquire lock if free
      bnez r2, lock
      ...
```

- `test-test-and-set` technique reduces the necessary memory bandwidth and coherence protocol operations required by a pure `test-and-set` based synchronization:
  - First, wait using a regular load instruction (lock will be cached)
  - Second, use `test-and-set` operation

```
      ...
lock: ld r2, barrier.lock         // first test with regular load
                                  // lock is cached meanwhile it is not updated
      bnez r2, lock               // test if the lock is free
      t&s r2, barrier.lock        // second test and acquire lock if STILL free
      bnez r2, lock
      ...
```

Reducing Synchronization Cost: `test-test-and-set`

- `test-test-and-set` **idea**[3] 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
     value++;             // Set (Assign)
  release_lock(&lock);
}
```

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

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

- 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?

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 Ayguade, Josep Ramon Herrero and Daniel Jiménez
({eduard,josepr,djimenez}@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya

2014/15-Fall