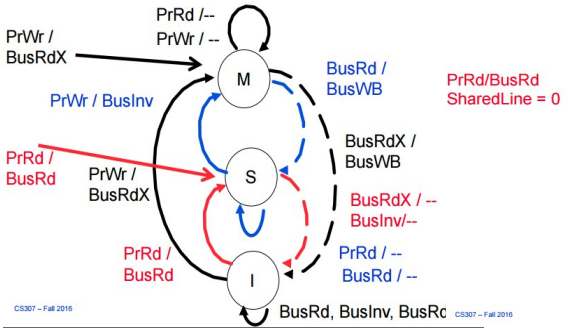
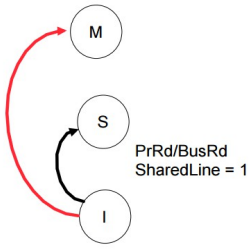
Semaine 3 - Cache Coherence II



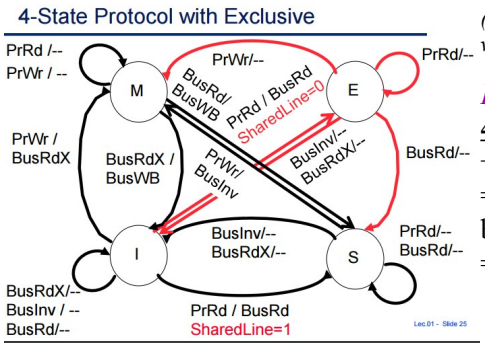


3 states Protocol : PROs and CONs

+ can work with writeback caches
+ *Modified state* for exclusive access
+ *Shared state* for potentially more copies

- *Single-threaded* program
- Private variables in the parallel programs
- Shared variables do not reside in the cache for long (evicted before invalidated by another core)

=> One often reads before a write
=> Lots of I->S->M means lots of BusInv* ! Cache hits take 2 cycles, cache misses (BusInv/BusRd) 6 cycles
=> We can go from I->M on a read if there are *no copies* out there
We use a special signal on the bus : **SharedLine**



*(*Je sais pas trop ce que ça veut dire mais en gros je crois que c'est comme un BusWr c'est quand un processor veut écrire dans la mémoire il envoie un BusInv... Valà :D)*

But, 2 bits can also encode 4 states !

4-State Protocol (with Exclusive State)

+ Exclusive State => indicates a *single copy* in the system !
=> On write, go directly to *Modified* (no BusInv needed) and if evicted, drop the block
=> Used when a block is read : if *SharedLine* = 0, goto *Exclusive*
if *SharedLine* = 1 goto *Shared*

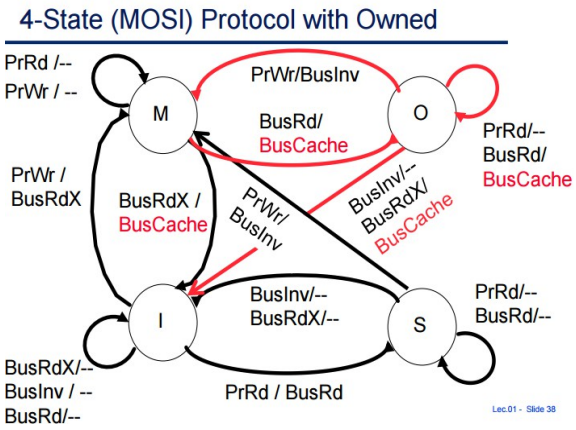
More 4-State Protocol with Owned State

Manycore processors

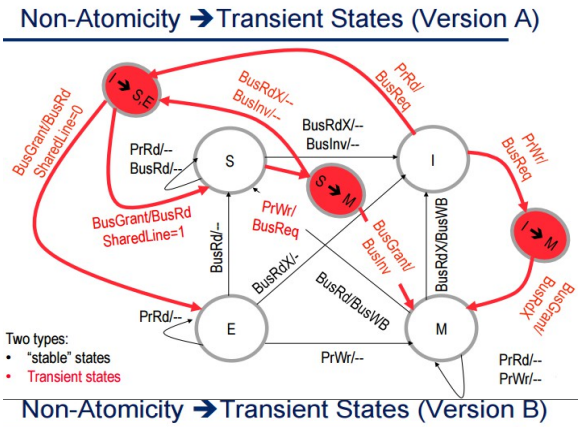
- On-chip much faster than access to off-chip
- Must allows « dirty » copies be shared without updating memory (used cache-to-cache tranfers)

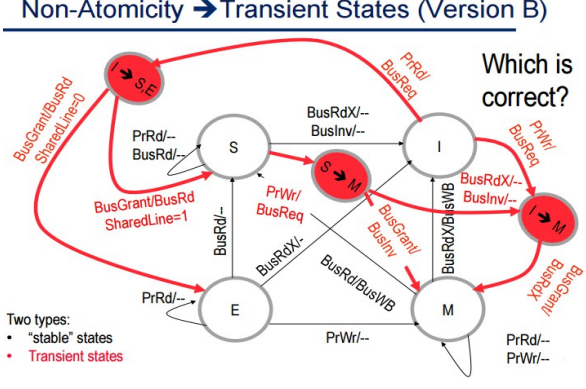
Owned State

- A core in *Modified* State can provide a shared copy
- Cache-to-cache transfer using « BusCache »
- Reader cores move to *Shared* State
- Writer cores move to *Owned* State



Until now, all transitions are assumed atomic, but in reality...Transitions in our current state are **non-atomic** !





What is the reality ?

- Bus requests are not granted immediately
- There can be contention for the bus
- Arbiter chooses who goes first => Others wait !
- Problem if two processors wants to write at the same time

<== So, we add « Transcient States »

Multi-Level Cache Hierarchies

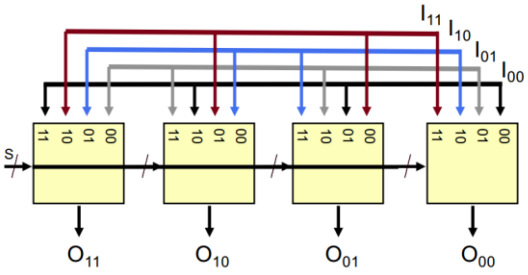
- Data in higher-level is subset of data in lower-level
(if modified in higher level => modified in lower level)
- With inclusion, only need to snoop lowest-level cache (if L2 is in Modified, then L1 is too)
=> Filter traffic at lower level

Instruction Cache

- Regular Cache blocks states are I or S
- Instructions are rarely written/modified
- Coherence in software (but osef)

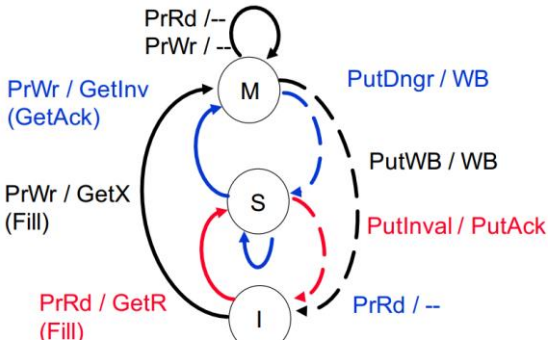
Semaine 4

Issue 1: Busses are fast and cool but 1 bus for 8 CPUs in not enough. The **crossbar** is some mega bus, which connects N to N elements. If 2 inputs wanna go in the same output, we gotta give priority to one.



4x4 crossbar (4 inputs I, 4 outputs O)

Issue 2: When a cache misses, how can it know who has the correct value? By asking the **directory**! A directory takes the requests from the caches and then answer them when it's done. It is connected to the crossbar.



The **MSI directory** protocol is different from usual MSI.

GetInv[C->D]: Tell the dir to invalidate other caches.
PutInval[D->C]: Invalitate the cache.
PutAck[C->D]: Say ok to the dir.
GetAck[D->C]: The dir is done with the cmd.
Fill[D->C]: Give the asked data to C.
PutWB[D->C]: Ask for WB.
PutDngr[D->C]: Ask info to cache C.

The crossbar may include a L2 cache, and in this case, it also updates the L2 while a WB and stores the L2 validity in the directory. It checks the L2 first in the case of a miss.

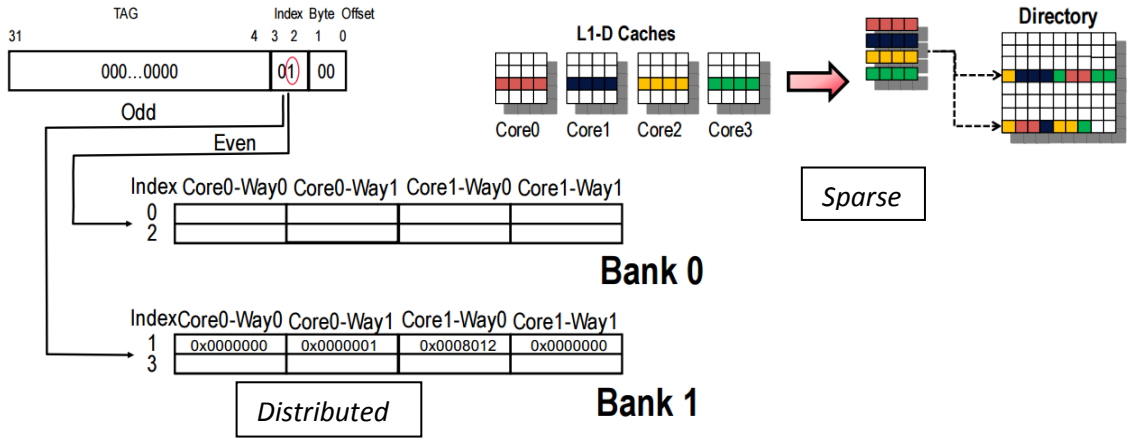
Types of directories

Duplicate tag: The directory can also keep a copy of all the tags(M,S,I) of the L1s. However, a single directory is as slow as a bus.

Distributed directories: We can create multiple directories with a L2 bank for each one (the banks store blocks with address % (number of banks)). But can be slow too!

Sparse directory: Splits L1 sets in directory sets (must also store the #cache). That is, all the caches operations are stored with no rule and too much operation can lead to loss of history (limited dir memory).

Sparse vector: Same as sparse but cache number written as vector. Keeps 1 bit per cache so that we can talk about multiple caches at the same time (cache 1 = 001, cache 2 = 010, caches 1+2 = 011) instead of just talking about 1 cache.



Week 5 : Coherence Misses

True sharing

- Producer-consumer communication pattern
- Happens regardless of the cache block size
- ex:
  - P1 read A - P0 write A
  - P1 write A - P0 read A

False sharing

- Happens if processors update different words in a block
- Updates invalidate the block (but useless because the words are not truly shared)
- ex: (A and B in the same block, but not in the same word)
  - P0 write A - P1 read B
  - P1 read A - P0 write B
  - P0 write A - P1 write B

These communications/sharing affect performance (more sharing if there is more cores)
to reduce these : reduce get and set (access) of the memory value by :

- lock / unlock the value to update (deny other threads to update in the same time)
- keep local until the end

Number of communication misses are affected by:

- Cache size (++cache = less Capacity/Conflict misses, sharing++)
- Number of processors (++processors = sharing++)
- Cache block size (++blocksize = false sharing++, a bit increase of C/C)

Homework 4

1) performance bottleneck => sharing ?
- true sharing : share the array B.
- false sharing: " all the counters fit in the same cache block" = Since the cache block is 64bytes and each counter is 32 bits -> at least 2 counters in the same cache block

2)
false sharing:

P0 write B

P1 write B+4

P0 write B --> false-sharing miss (same block)

true sharing:

P0 write B

P1 write B

P0 write B --> true-sharing miss

3) (cf. algo exo 1.3)
locally update the value before adding it to the array only at the end (less accesses)

static int B[8][BlockSize/IntSize]; // Array B with its elements aligned to the cache block size

barrier(); // A barrier is required to wait for all the threads

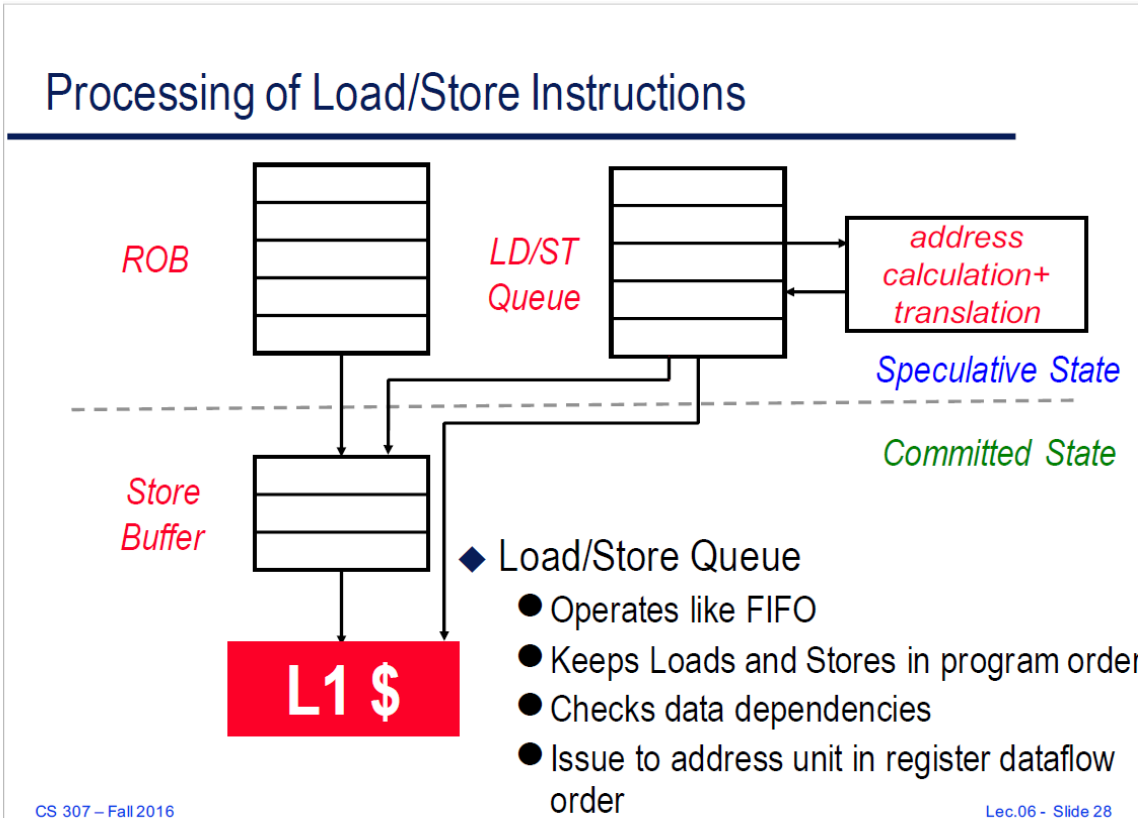
Week 6 : Memory ordering

2 issues :
Single core ordering

- How do we make sure that a load to address A reads the last store in program order?

Multicore/multiprocessor ordering

- If locks guard data, how to we make sure order is preserved?



Stores go first to **Store Buffer**
A few entries to buffer stores before L1
As wide as a Word
Store misses can wait in store buffer
Stores can go to L1 later to open up ports for loads

Loads must check store buffer because it has the latest value
When ports available, stores access L1
Multiple stores at the same address : only the last value is preserved

Uniprocessor Load and Store Semantics

Keep all loads and stores totally in order with respect to each other
However, loads and stores can execute out of order with respect to other types of instructions (while obeying register data-dependence)

Differents memory consistency models :

- Naïve sequential consistency (SC) : no store buffer and memory operations cannot fetch cache blocks while waiting in the load/store queue.
- Optimized sequential consistency : store buffer and entries waiting in the load/store queue can trigger cache block fetches
- Processor consistency (PC) : Optimized SC + allows loads to commit even if there are older stores pending ahead.

Synchronization (L8)

Synchronization event :

1) **Acquire method** : How thread attempts to gain access to protected resource

2) **Waiting algorithm** : How thread waits for access to be granted to shared resource

3) **Release method** : How thread enables other threads to gain resource when its work in the synchronized region is complete

Busy waiting and blocking : `while (condition X not true) {} // after logic assumes X is true`

Blocking synchronization : `if (condition X not true) //` block until true ;

- BEST : Blocking : if waiting time is significant
- BEST : Busy-waiting :
  - Scheduling overhead is larger than expected wait time
  - Processor's resources are not needed for other task

Locks

Desirable lock performance characteristics :

- **Low latency** : If lock is free, and no other processors are trying to acquire it, a processor should be able to acquire the lock quickly
- **Low traffic:** If all processors are trying to acquire lock at once, they should acquire the lock in succession with as little traffic as possible
- **Scalability:** Latency / traffic should scale reasonably with number of processors
- **Low storage cost**
- **Fairness:** Avoid starvation or substantial unfairness
- One ideal: processors should acquire lock in the order they request access to it

contention

Wiki : en informatique, par anglicisme sémantique on parle de contention plutôt qu'utiliser les termes Français de saturation, étranglement ou sous-capacité pour désigner les situations où la capacité d'une ressource à traiter une charge ou des demandes arrive en limite soit par sur sollicitation en provenance d'une entité, soit par plusieurs entités cherchant à accéder concurremment à une ressource partagée.

Test&Set lock (ts)

```
ts R0, mem[addr] // atomically load mem[addr] into R0
// and set mem[addr] to 1
lock:    ts R0, mem[addr] // load word into R0
            bnz lock // if 0, lock obtained
unlock: st mem[addr], #0 // store 0 to address
```

- read-modify-write (rmw)
- When the number of processors increases, the time increases too.

Test&Set lock with back-off

Upon failure to acquire lock, delay for awhile before retrying :

```
void Lock(volatile int* l) {
    int amount = 1;
    while (1) {
      if (Test&Set(*l) == 0) return;
      delay(amount);
      amount *= 2;
}     }
```

- Generates less traffic than Test&Set (not continually attempting to acquire lock)
- Improves scalability (due to less traffic)
- Storage cost unchanged
- Exponential back-off can cause severe unfairness

Test&Test&Set lock (process ≠ instruction)

It avoids useless attempts to acquire a locked lock

```
void Lock(volatile in* lock) {
    while (1) {
        while (*lock != 0) ; // wait while another p has the lock
        if (Test&Set(*lock) == 0) return;  // When released, try to acquire it
}           }
void Unlock(volatile int* lock) {
    *lock = 0;
}
```

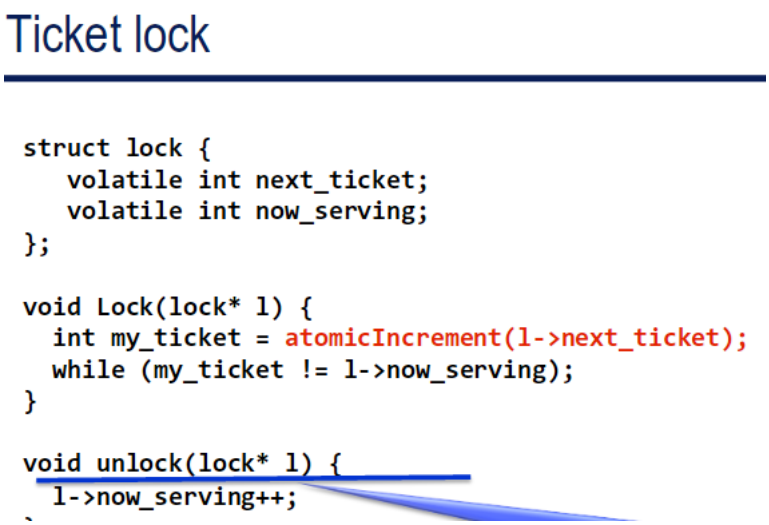
- Higher latency than test & set in uncontended cas
- Generates much less bus trafic → more scalable
- Storage cost unchanged
- Still no provisions for fairness

Ticket lock

Main problem with Test&Set style locks : all waiting processors attempt to acquire lock using test & set

Solution : processors get a ticket and wait their turn. (like in McDo in Renens)

No atomic operation needed to acquire the lock (only a read)



AtomicCAS

```
// atomicCAS: atomic compare and swap
int atomicCAS(int* addr, int compare, int val)
{
   int old = *addr;
   *addr = (old == compare) ? val : old;
   return old;
}
```

List-Based Queue Lock (MCS)

Voir slides : 36-43 (Impossible à décrire) http://parsa.epfl.ch/courses/cs307/lectures/L8-synchronization.pdf

Queue-bases locks behavior (QOLB)

- Acquiring/releasing locks is expensive
- lock is fair
- No acquire/release/spin trafic
- If no contention, acquiring/releasing is expensive
- If contention, fair + little trafic

- **Test&Set** is good with no contention
- **Queue** is best with high contention

Point-to-point event synchronization

- Often use normal variables as flags:

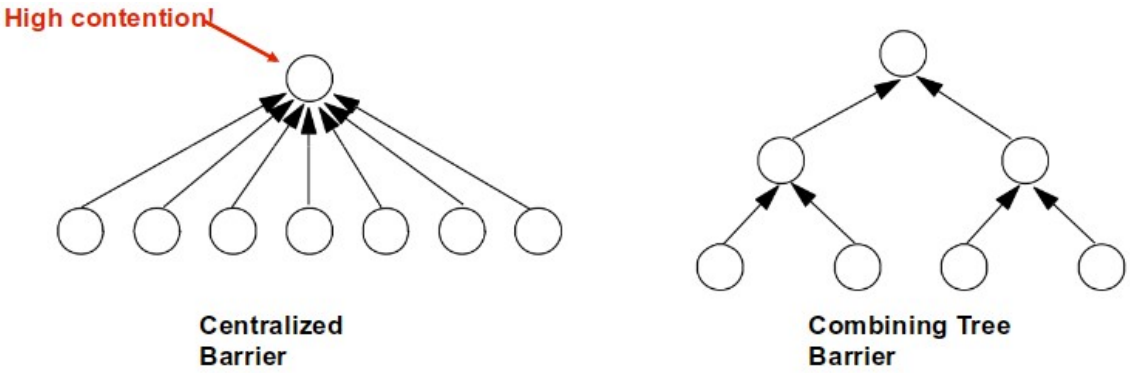
a = f(x);	while (flag == 0);
flag = 1;	b = g(a);

- If we know the initial value of a:

a = f(x)	while (a == 0);
	b = g(a);

Barrriers

En programmation concurrente, une **barrière de synchronisation** permet de garantir qu'un certain nombre de tâches aient passé un point spécifique. Ainsi, chaque tâche qui arrivera sur cette barrière devra attendre jusqu'à ce que le nombre spécifié de tâches soient arrivées à cette barrière.



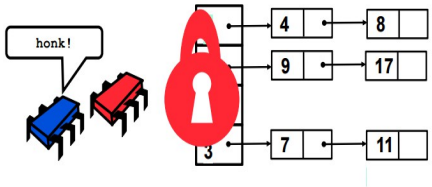
Combining trees make better use of parallelism in interconnect topologies

Semaine 9 – Hardware Lock Elison & Transactional Memory

Part 1 : Concurrent data structures (Hardare Lock Elison)

Coarse grained lock

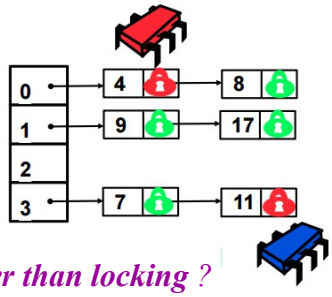
- ✔ Lock large swathes of program code in one big chunk (entire classes, methods or packages)
- ✔ Much easier to understand what's going on in the system
- ✔ Simple implementation => easy to code
- ✖ Bad performance => slow
- ✖ Lots of contention/conflicts !
- ✖ No concurrence/parallelism




```
void editHash(HashTbl tbl, int key) {
   synchronized(tbl) {
       // read objects
       HashObj obj = tbl.get(key);
       // update
       obj.update();
   }
}
```

Fine grained lock

- ✔ Reasonable performance => Faster than Coarse grained lock
- ✔ Lots of concurrence/parallelism
- ✔ Cotention for objects only
- ✖ Hard to code
- ✖ Limited only to object update




```
void editHash(HashTbl tbl, int key) {
  // read objects
  HashObj obj = tbl.get(key);
  synchronized(obj) {
      // update
      obj.update();
  }
}
```

*=> What is **better than locking** ?*

Lock-free data structures & optimistic concurrency

- ✔ Best performance
- ✖ Very hard to code

*=> How about **speculation in hardware** ?*

Hardware Lock Elision

- Optimistically assume no conflicts occur
- Check for conflicts in the data structure
- Evict cache block touched by critical section
- If multiple rollbacks, retry but this time do the lock !

Vocabulary

- A **critical section** is a piece of code that must not be run by multiple threads at once because the code accesses shared resources
- A **mutex** is an algorithm that is used to protect critical sections
- A **rollback** is an operation which returns the database to some previous state
- A **speculative execution** is an optimization technique where a computer performs some task that may not be actually needed

- ✔ **Conservative Locking** : Easier to show correctness, lock more often than necessary
- ✔ **Locking Granularity** : Tradeoff between Performance and Complexity
- ✔ **Thread-unsafe legacy librairies** : Require global locking
- ✖ **Priority Inversion** : When low-priority process holds a lock needed by a high-priority process
- ✖ **Convoying** : Threads holding locks can be de-scheduled, swapped out by OS
- ✖ **Deadlocks/livelocks** : Locking in the different order across threads
  => Need to agree on a system-wide lock acquisition order (if failing to follow order, deadlock !)
  => Deadlocks are hard to resolve/debug
  => Locks break software modularity

Part 2 : Programming with transactions (Transactional Memory)

Transactional Memory (TM)

Concurrency for data residing on disks (cached in memory), in software

- **Atomicity** (all or nothing)
  - Upon transaction commit, all **memory** writes take effect at once
  - On transaction abort, none of the writes appear to take effect
- **Isolation**
  - No other processor can observe **memory** writes before commit
- **Serializability**
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed though

Difference between Transactional Memory and Transactional Processing

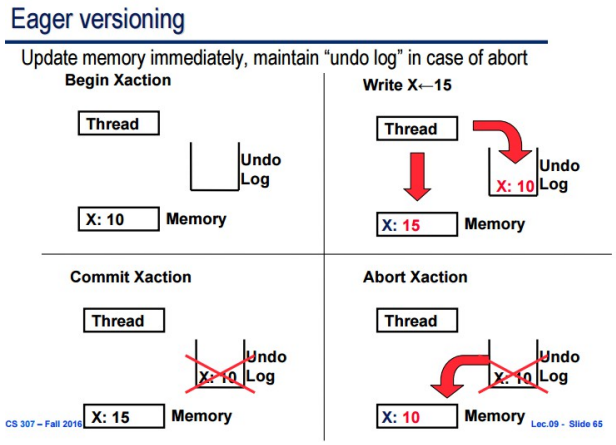
- TM : An atomic & isolated sequence of memory accesses, supported in hardware by processor & cache hierarchy
- TP : Concurrency for data residing on disks (cached in memory), in software

Data Versioning

Until now all speculative data in the pipeline
From now on, speculative data also in the caches & buffers

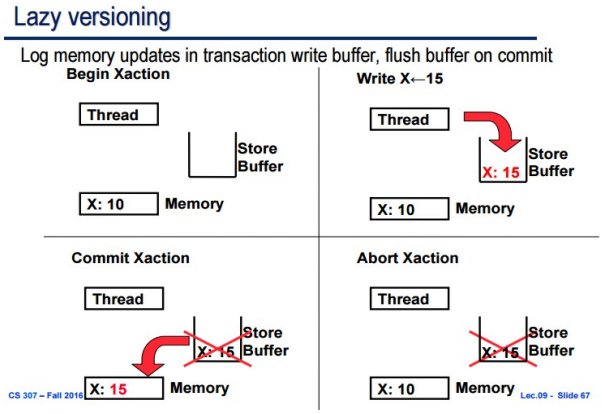
- **Eager versioning (undo-log based)**
  - Update memory location directly
  - Maintain undo info in a log (per store overhead)
    => Either custom buffer
    => Or keep undo log in L1D

✔ Faster commit (data is already in memory)
✖ Slower aborts, fault tolerance
✖ Must read data to store in undo log



- **Lazy versioning (store-buffer based)**
  - Buffer data in a write buffer until commit
  - Update actual memory location on commit

✔ Faster abort (clear log), no fault tolerance issues
✖ Slower commits



Conflict detection

Must detect and handle conflicts between transactions :

- Read-Write, Write-Read conflict
  => A reads addr X, which was written to by pending transaction B
- Write-Write conflict
  => Transactions A & B are pending, both write to addr X

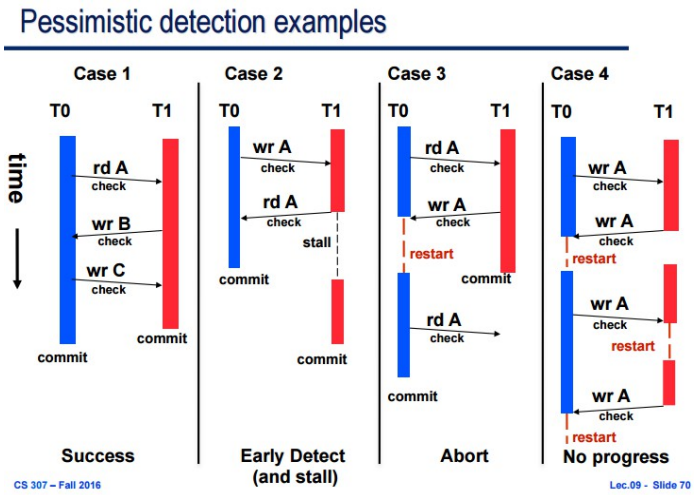
=> ***Two types of conflict detection*** *: Pessismistic Detection and Optimistic Detection*

Pessimistic detection (a.k.a. « eager »)

- Check for conflicts during loads or stores
  => Through coherence actions *(BusRd, BusRdX, BusInv)*
- « **Contention manager** » decides to stall or abort transaction
  => Various priority policies to handle common case fast

✔ Detect conflicts early (undo less work, turn aborts into stalls)
✖ No forward progress guarantees, more aborts in some cases
✖ Fine-grained communication
✖ Detection on critical path

« I suspect conflicts might happen, so let's always check to see if one has occurred upon each coherence operation. »



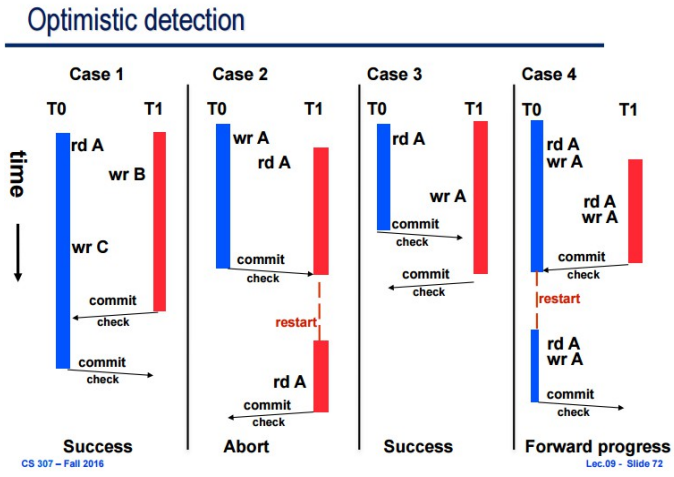
Optimistic detection (a.k.a. « lazy »)

- Detect conflicts when a transaction attempts to commit
  => Collect and validate write set using coherence actions
  => Get exclusive access for cache lines in write set

- On a conflict, give priority to committing transaction
  => Other transactions may abort later on
  => On conflicts between committing transactions, use contention manager to decide priority

✔ Forward progress guarantees
✔ Potentially less conflicts, bulk communication
✖ Detects conflicts late, can still have fairness problems

« Let's hope for the best and sort out all the conflicts only at the time of commit »



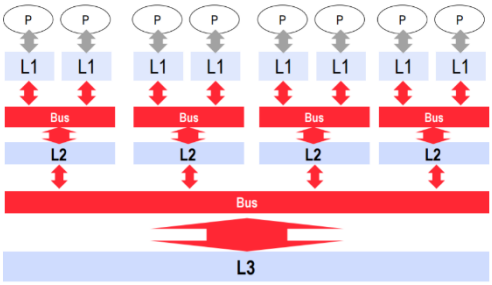
Conclusion

- Speculation is good
  => Synchronization is programmed/placed conservatively
  => Use hardware to avoid waiting
- Expose speculation to software
  => Transactions
  => Use the same hardware
  => Use software to drive the speculation/annotate program

Lecture 10 – Interconnects

Issues:
- Busses are not fast enough to support many CPU's connected at the same time (bus must be used at 40% max).
- Use of hierarchical buses is a solution to that but it slows down the access to memory dramatically. We'd better use crossbars. (Picture on the right).

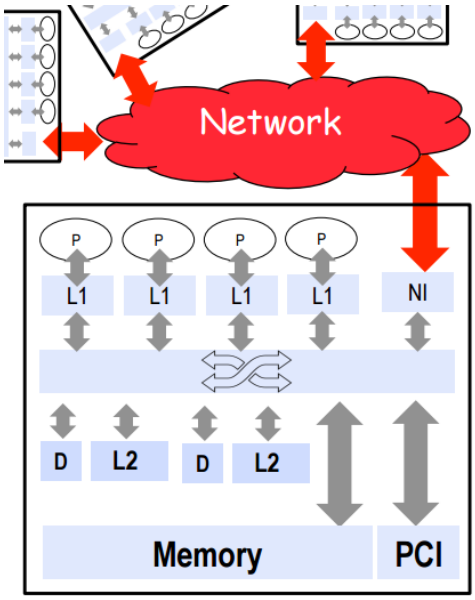
One solution: *Interconnects*



Interconnecting is the concept of linking PCs together through a network. This is an off-chip system, by opposition with what we have seen before.

Interconnect vocabulary:

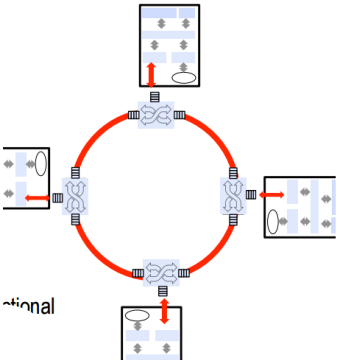
Topology: Physical structure of the communication graph
Routing: Paths through which messages may actually flow
Flow *control*: Control applied to the flow to limit flooding
Switch / *router*: Devices to send the data to the good destination
Channel: Connection between 2 routers
Terminals: PC / server located at each end of a network



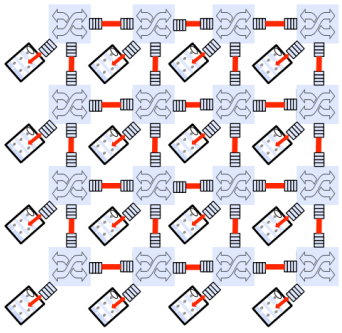
How to use interconnection?
We can connect a network interface NI to the crossbar as if it was a CPU. In the picture on the right, the red Network cloud represents a network of switches / routers. Each switch also contains a crossbar to choose the right route in which to send the packets. It also features buffers at each port for queuing packets.

Types of networks:

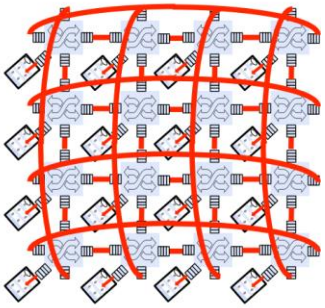
Ring: The terminals are linked in ring.
Switch cost: N, Wire cost: N.
Broadcast is available but no serialization.
Latency: N/2 unilateral, N/4 bidirectional
Bisection bandwidth: 2



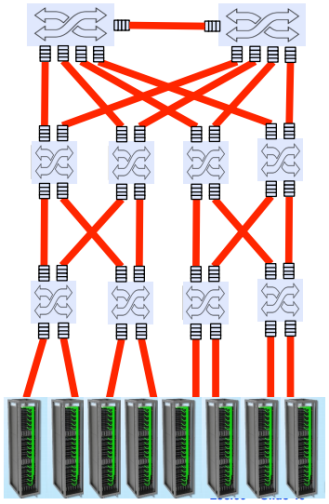
2D Mesh: Linked en trellis.
Switch cost: N, Wire cost: N
No broadcast nor serialization.



2D Torus: Like mesh but extremities are linked as well.
Switch cost: N, Wire cost: 2N
No broadcast nor serialization.
Latency: $0.5N^{0.5}$, Bisection bandwidth: $2N^{0.5}$



Clos: Multi-stage network. Many paths can be used to go to some destination, which allows using good algos.

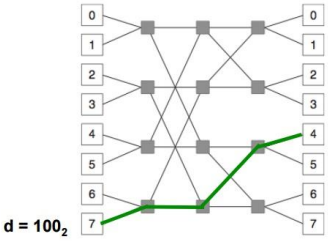


Routing algorithms

In order to use the network links efficiently, we must use some clever algorithm to find a route between a source and a destination.

A routing algorithm: Destination-Tag Routing

We are given an array of number, and when we cross the i^{th} switch, we use the i^{th} number of our array to determine which path to take.



Routing in 2D Mesh/Torus, in general

The algorithm must be agile, depending on the state of the network. It must use local information from each switch to route the packets while avoiding locks. It is not necessarily minimal!

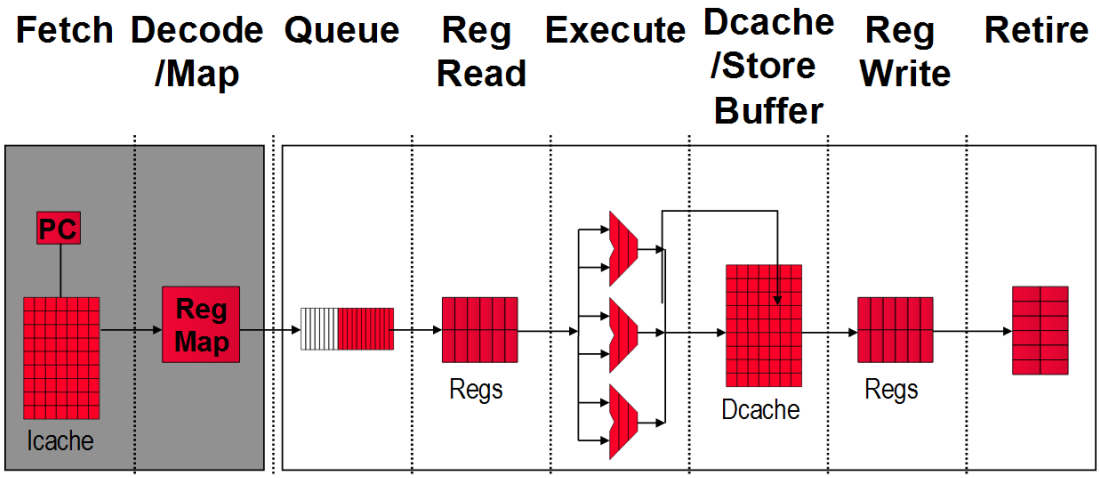
In order to control the flow, each router sends the *flits* (= decomposed packets) to next router when the next router is free or else keeps them in its *buffer*. The routers must communicate in order to tell if they are free or not.

Deadlocks

A *coherence protocol deadlock* can occur when there is a cyclic resource dependency (routers waiting for each other to be freed). We must prevent the formation of cycles in our algo. We divide the buffers into many message classes, one per resource type, so that each part of a potential cycle uses a different resource (then no cycle).

A *routing deadlock* can occur when cycles are inducted by the routing algo. Hopefully, deadlock-free algos exist.

Multithreaded processors



Note :

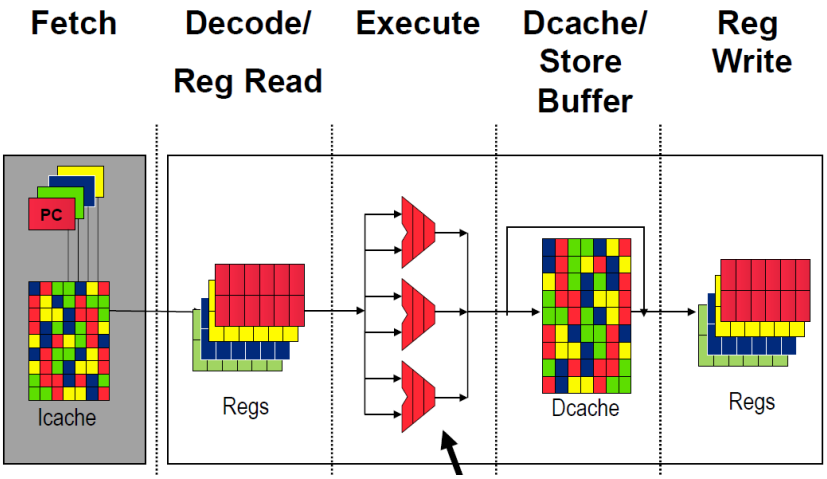
- Superscalar pipeline : can process more than x inst per cycle (x-way)
-

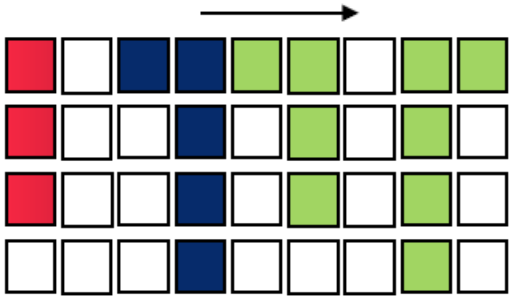
Pipelines suffer from bubbles :

- **Structural hazards :** when you have 2 entries but only one can be processed
- **front-end hazards :**
  - Branch Target Buffer miss (target address need to be decode)
  - branch prediction
  - Instruction cache misses freeze the pipeline
- **data-dependencies**
  - dependent instructions cannot be executed in the same cycle
  - data-cache miss

Blocked Multithreading

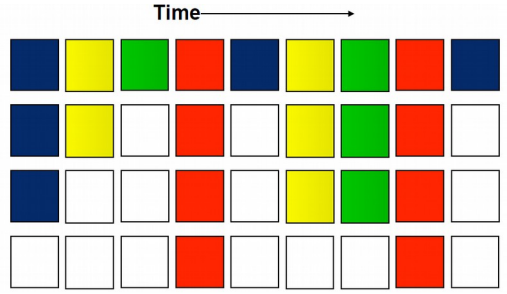
- Execute : one thread running
- Contention in data-cache and instruction-cache
- Question : When to switch thread ???
- GOOD : small changes to existing hardwares
- BAD : single-thread perf. suffers

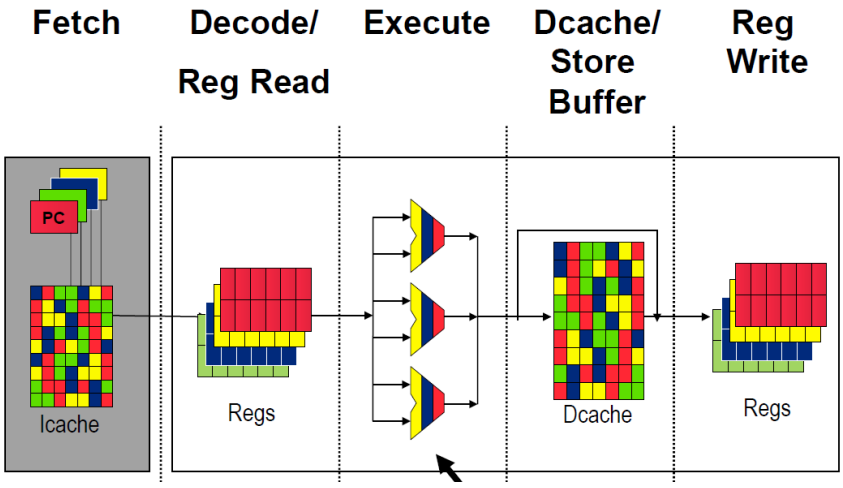




Interleaved/Fined Grained Multithreading

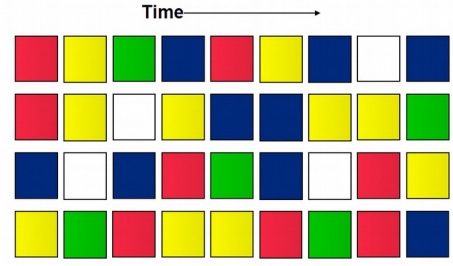
- different threads each cycle
- can interleave thread inst. –> will fill bubbles of some other thread → increase
- in general : ++ threads → - - single thread perf
- GOOD : ++throughput/utilization by reducing impact of depedences, reasonable single thread perf
- BAD : complicated hardwares, multiple contexts, limit single thread perf.
- Execute : multiple threads running

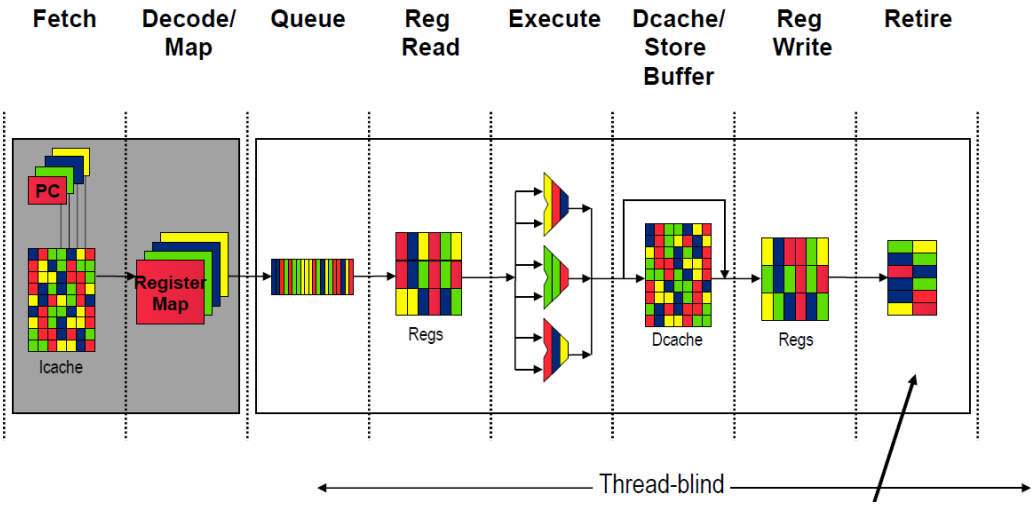




Simultaneous Mutlithreading SMT

- can execute different thread inst in the same cycle
- ++ throughput/utilization
- Retire : not really thread blind
- Question : fetch-interleaving policy ???
- LRU/Round Robin : Modified least recently used
- Icount : thread with fewer inst in pipe. Has priority





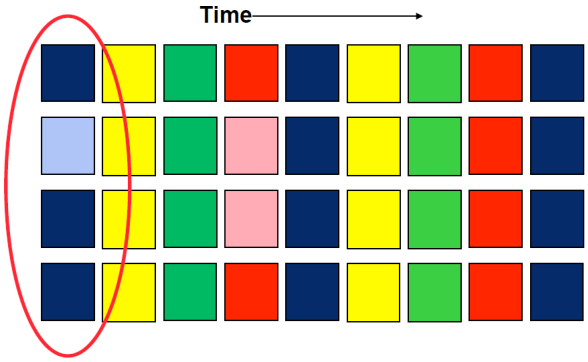
SIMD = Single Instruction, Multiple Data

- Assume wide, multi-item registers
- add rd, rs, rt produces multiple results!
- Fewer instructions, fewer branches!
- Hard for compiler to use, hand code libraries mostly!
- Intel MMX, SSD{1-4}, AVX,…

GPUs

- Make cores simple
- use fine grained multithreading
- warps : groups of threads that run in lockstep → multiple warps can share the pipeline

Thread divergence : threads within warp might exec. diff. Path



Warp: group of threads running in lockstep

--→ Il y a « plein » de détails dans le pdf, mais je ne juge pas nécessaire. Faudra voir le hwk si y'en a un