COM SCI M151B Week 8

Aidan Jan

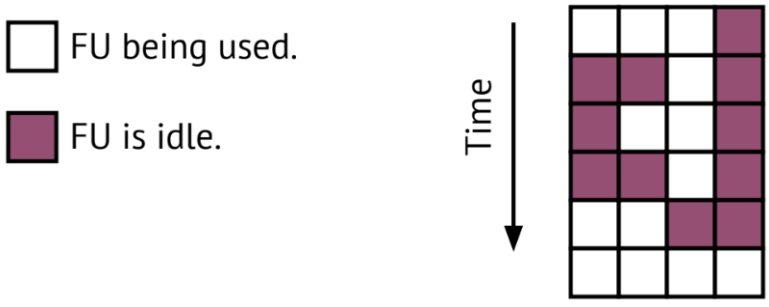
November 19, 2024

UniProcessor

- Threads
  - "Per-thread" state
    - Context state: PC, registers
    - Stack (per-thread local variable)
  - "Shared" state: global, heap, etc.
  - Threads generally share the same memory space.
- A process is like a thread, but with its own memory space.
- Generally, system software (the O.S.) manages threads.
  - "Thread scheduling", "context switching"
- In single-core system, all threads share one processor
  - Hardware timer interrupt occasionally triggers O.S.
  - Quickly swapping threads gives the illusion of concurrent execution.

Instruction-Level Parallelism (ILP)

- We introduced Instruction-Level Parallelism (ILP, superscalar) a long time ago
  - 2 ALUs, and you can fetch two instructions at a time, etc.
- Limitations
  - Data dependency
  - Poor branch prediction (Lots of flushes!)
  - Memory Latency (aka memory "wall")
- Due to the limitations, superscalar pipelines are typically underutilized.



- What can we do?

Simultaneous Multithreading (SMT)

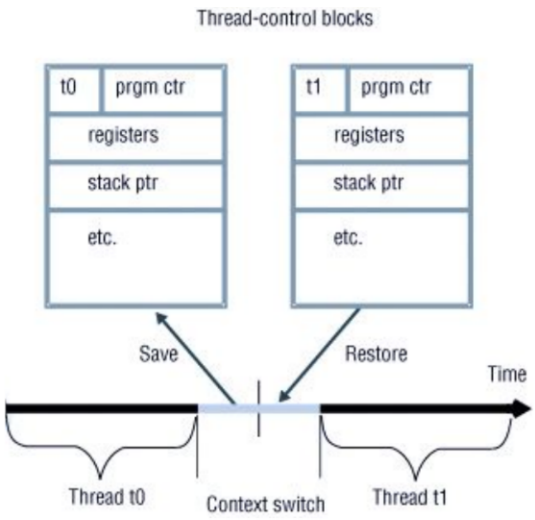
- Execute multiple programs (threads) on the same pipeline. (Two separate programs typically have no dependencies on each other!)
- This allows us to utilize the pipeline width more efficiently.
- Processes and threads
- Different from context switch
- Fine-grained vs. Coarse-grained.

How to implement SMT?

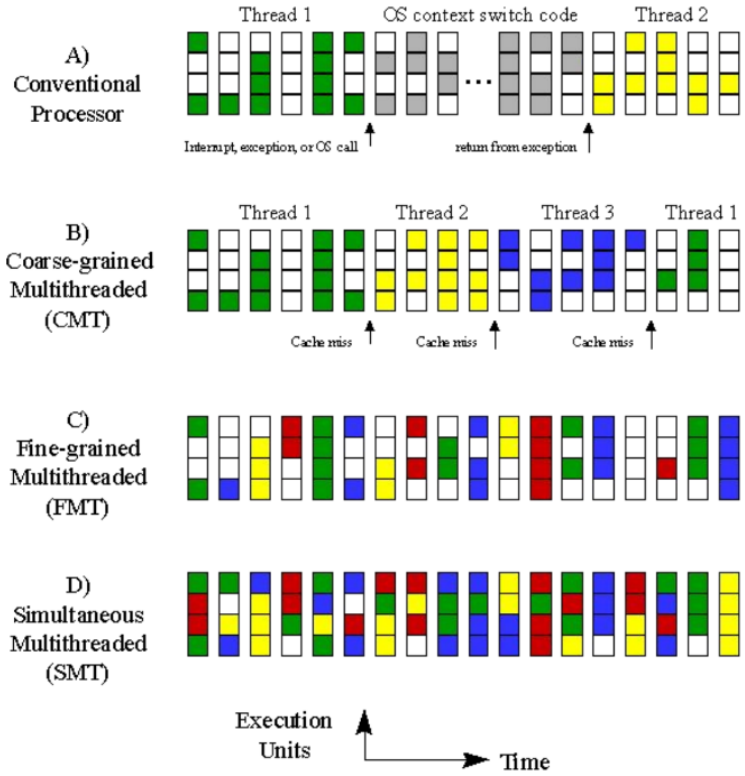
- Each thread requires its own:
  - PC
  - Register file
  - Logic for virtual address translation
  - Exception Handling mechanism
  - (No need for extra memory or separate ROB, RS, etc.)
- Example: [fill]

Multi-Tasking and Context Switch

- Running different processes/tasks on the same core.
- Different from SMT!



Normal vs. CMT vs. FMT vs. SMT



- Can we do better?

Multi-Core Systems

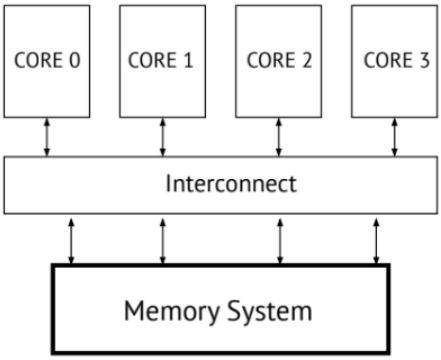
- How to leverage multiple cores
- Design choices
- Challenges

Real World Systems

Server vs. Personal vs. Embedded:

- PC:
  - 4-8 core systems
  - 4-8-wide superscalar, out-of-order
  - SMT
  - 3-level cache
- Server:
  - Intel Xeon, 20-50 cores
- Embedded:
  - ARM Cortex (M and A series)
  - In-order

– No OS and no virtual address



How multi-core helps?

- Multitasking
- Thread-Level Parallelism
- (They are not the same!)
- (Amdahl's law)

Multitasking

- Scheduling
- Power vs. Performance
- Heterogeneous systems
- ...

Thread-Level Parallelism

- How to improve the speed of one program?
  - This can give us improvement in performance (hence faster computers)

Example:

- Sum 64000 numbers on 64 processors
  - Each processor has ID: $0 \leq Pn \leq 63$
  - Partition 1000 numbers per processor
  - Initial summation on each processor
  - Now need to add these partial sums

```
sum[Pn] = 0;
for (int i = 1000 * Pn; i < 1000 * (Pn + 1); i++) {
    sum[Pn] += A[i];
}
```

    * Reduction: divide and conquer
    * Half the processors add pairs, then quarter, ...

Problems:

- Difficult to extract (and write)!
  - Parallel algorithms and programming: MPI, Pthread, ...
- Communication overhead and sequential parts
- N cores won't give us N times faster system.

How do we improve this?

- Better algorithms
- Faster communication
- → Parallel Programming

How to build a multicore system?

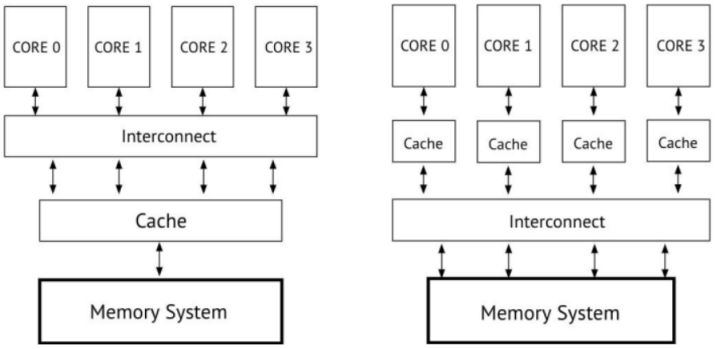
What changes?

- Interaction with Memory!
- Multiple cores try to talk to the same memory!
- Data needs to be shared, otherwise, what is the point!?

Design Choices

- How to share the main memory between cores?
- What about caches?

Shared vs. Private

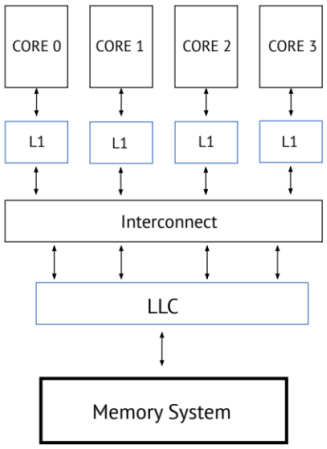


Left: Shared, Right: Private

- Hit time?
  - P: 3 S: 7
  - Shared places CPU further from cache.
- Miss rate?
  - Two scanarios:
    - Cache-friendly → P: 7 S: 3
    - Streaming application → P: 3 S: 7
- Miss penalty?!
- What about sharing?

Hybrid Approach

Best of both worlds!



- Place L1 cache private, the others are shared.

Challenges

- Cache and memory is shared between cores and is simultaneously utilized by them.
  1. This could lead to correctness issues! (See below.)
  2. This could also lead to performance issues!
     - Cache is not efficiently shared and utilized (e.g., one program can hurt others...)

Correctness Issues in Multicore

1. We may end up with incorrect copies in each private cache (called *cache coherency*)
2. Multiple cores are reading from/writing to memory concurrently, thus we may end up with incorrect ordering of reads/writes (called *memory consistency*).

Synchronization Problem (Who arrives first?)

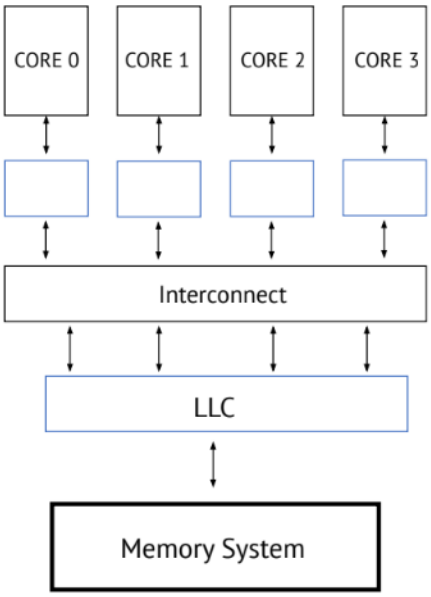
```
Core1: lw x1, x0, Z     Core2: lw x2, x0, Y
...                     ...
Core1: sw x7, x0, Y     Core2: sw x8, x0, Z
```

x8 = 100, x7 = 200

x1 =? x2=?

Synchronization Problem: What order memory should see?

Cache Coherence Problem



Mem[Z] = 0 and Z is an address (e.g., 0x200)
(x5=10)

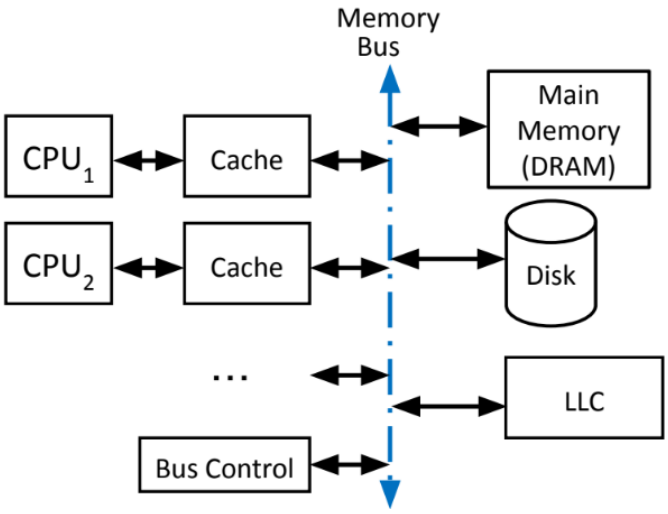
```
Core1: lw x1, x0,  Z
...
Core2: lw x3, x0, Z
...
Core1: sw x5, x0, Z
...
Core2: lw x6, x0, Z
```

- Uniformity of *shared* resource data that ends up stored in multiple *local caches*.
- What if no sharing? What if no private cache?
- A memory system is *coherent* if:
  1. Read what is written: A read of X on P1 returns the value written the most recent write to X and P1 if no other processor has written to X in between.
  2. Coherent writes happen eventually: If P1 writes to X and P2 reads X after a sufficient time, and there are no other writes to X in between, P2's read returns the value written by P1's write.
  3. Causality of writes: Writes to the same location are serialized: two writes to location X are seen in the same order by all processors.
- Can we prevent the cache coherence problem in software?
  - No!
  - Cache is transparent to software.
  - What about ISA? (No → Maybe)
    - Special instruction(s) to flush a line and/or entire cache
    - Very inefficient (needed for every single access!)
  - What about in the OS?

Solution to cache coherency

- We need cache coherency protocols!
  - What should happen after each load or store for each core?
  - See something say something!
- Use hardware to track each cache line!
  - "Snoop" the cache for each read/write/ Invalidate each copy if a new write comes.

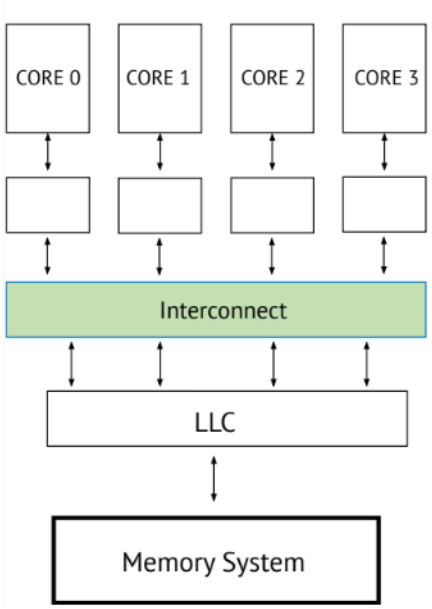
Snoopy Cache



- Where do we store these?
  - With lines! (metadata)
  - Recall LRU states

Snoopy Cache Coherence

- Write miss:
  - The address is invalidated in all other caches before the write is performed
- Read miss:
  - If a *dirty copy* is found in another cache, a **write-back** is performed before the memory is read (e.g., a broadcast is made to all the cores whenever a value is updated).
  - Otherwise, just read from the memory.

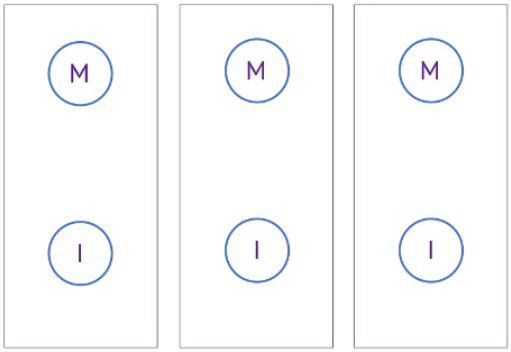


Mem[Z] = 0 and Z is an address (e.g., 0x200)
(x5=10)

```
Core1: lw x1, x0,  Z
...
Core2: lw x3, x0, Z
...
Core1: sw x5, x0, Z
...
Core2: lw x6, x0, Z
```

Coherence States

- The minimum we need:
  a. We have the valid copy, and it is most updated!
  b. We do not have the copy (or we had it but no longer valid)
- MI Coherence protocol
  - M = Modified state
  - I = Invalid state



  - Core 1: I, M, I, I, M
  - Core 2: I, I, M, I*, I
  - Core 3: I, I, I, M, I
  - Sequence: R_1, W_2, R_3, R_1

Adding a Shared State

- No need to invalidate if the other processor wants to read (still the same copy)
- → Unnecessary invalidation create overhead and hurt the performance

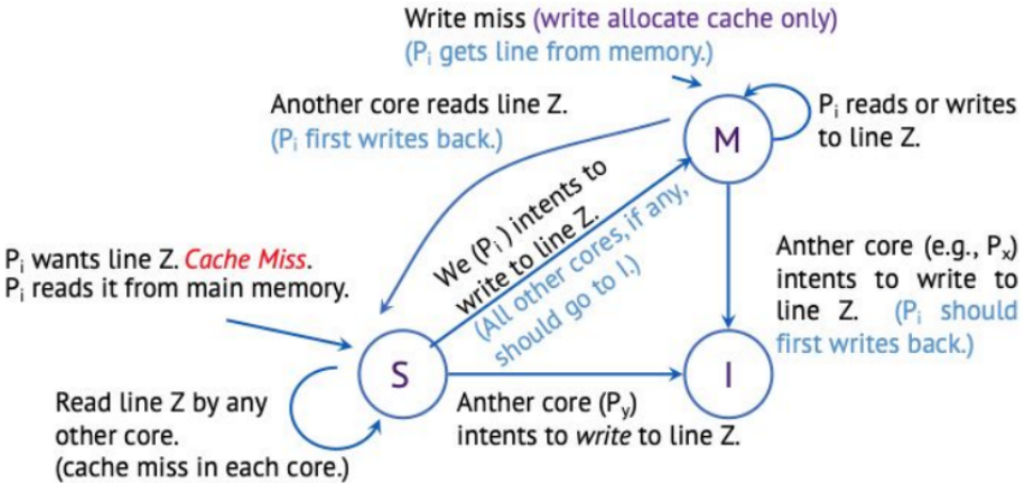
Cache States

- Using snooping and broadcasting, each cache line (in each core) can have one of three possible **states**.
  - Shared - This line is a fresh and "clean" copy of main memory
  - Modified - This core wants to write to this line, thus it contains "dirty" copy of the line.
  - Invalid - Another core wants to write to this line, so the current copy is no longer valid.

Dirty Line

- Write back vs. Write Through
- Dirty bit

MSI (Per Core)



Optimizing MSI

How can we make MSI better?

- A core is in S, and issues a write (store), it needs to broadcast that:
  - But if this core is the only one that has this line (exclusive), why does it have to broadcast?
  - It can silently upgrade from S to M, no broadcast needed.
- How to distinguish between shared and exclusive?
  - Need a new state called E.

MESI Protocol

- Same as MSI, except:
  - On a read miss, a processor can go from I to E if this is the only core that needs that line.
  - If on E, and if another core wants that line, it changes from E to S, the other core also goes to S.
  - In E, if the same core wants to write, it silently goes to M (others are all in I).