Module 14 – Cache Coherency

(ECE M116C- CS M151B) Computer Architecture Systems

Nader Sehatbakhsh
Department of Electrical and Computer Engineering
University of California, Los Angeles

Quiz 2

- Branch prediction
- Out-of-order execution
- Cache design

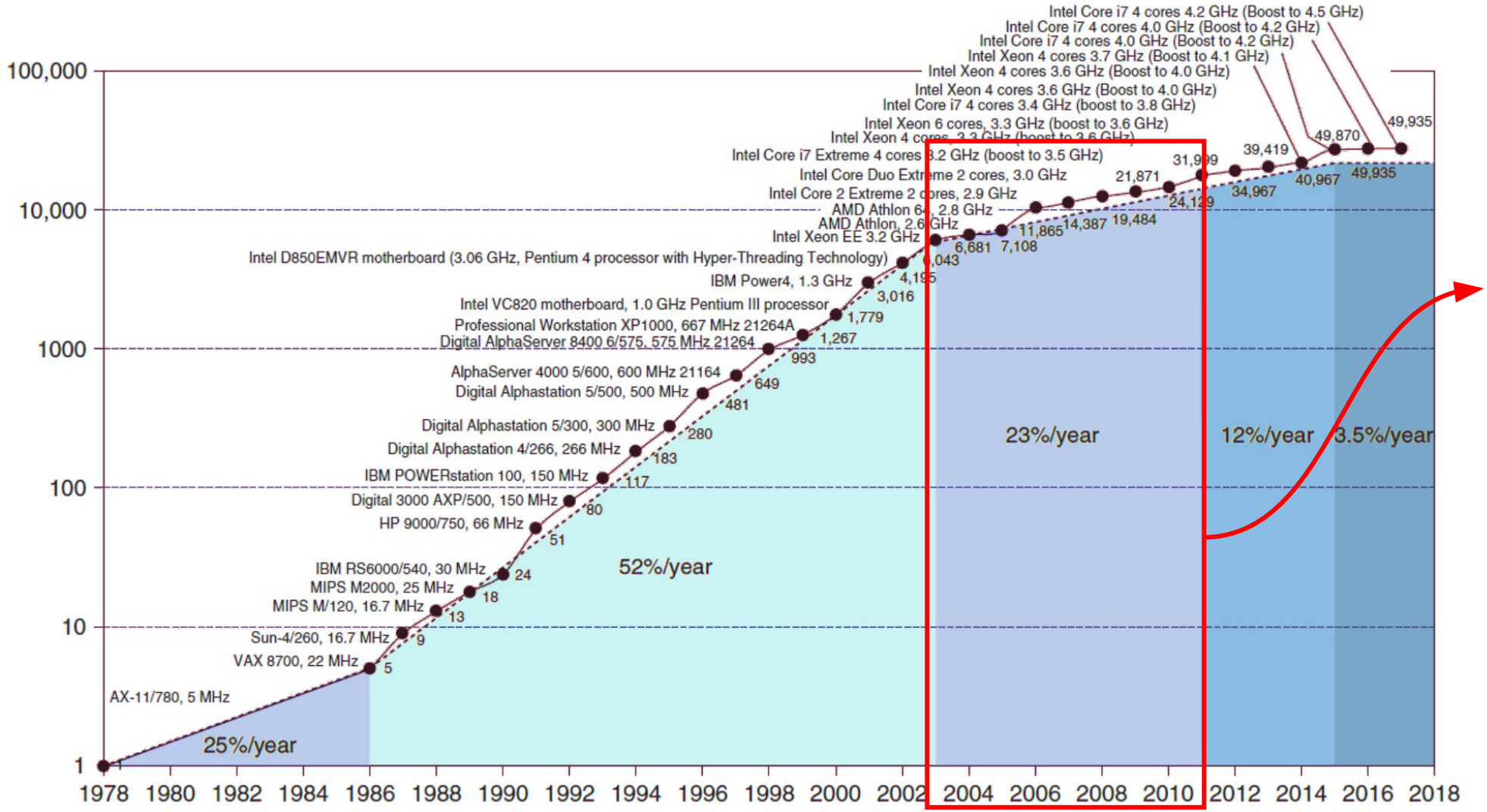
→ Same format as Quiz 1

CA 2

- Great job everyone!

- Huge congratulations to our top three (mostly orders of few MBs):
  - Simon Traub → TAG with special hashing and replacement
  - Cyrus Asasi → An ensemble of perceptron predictors
  - Saahas Kohli → Piecewise linear branch predictor

Where are we now?



End of Dennard's scaling.
Hitting the ***power wall.***

*Plot was taken from [1].

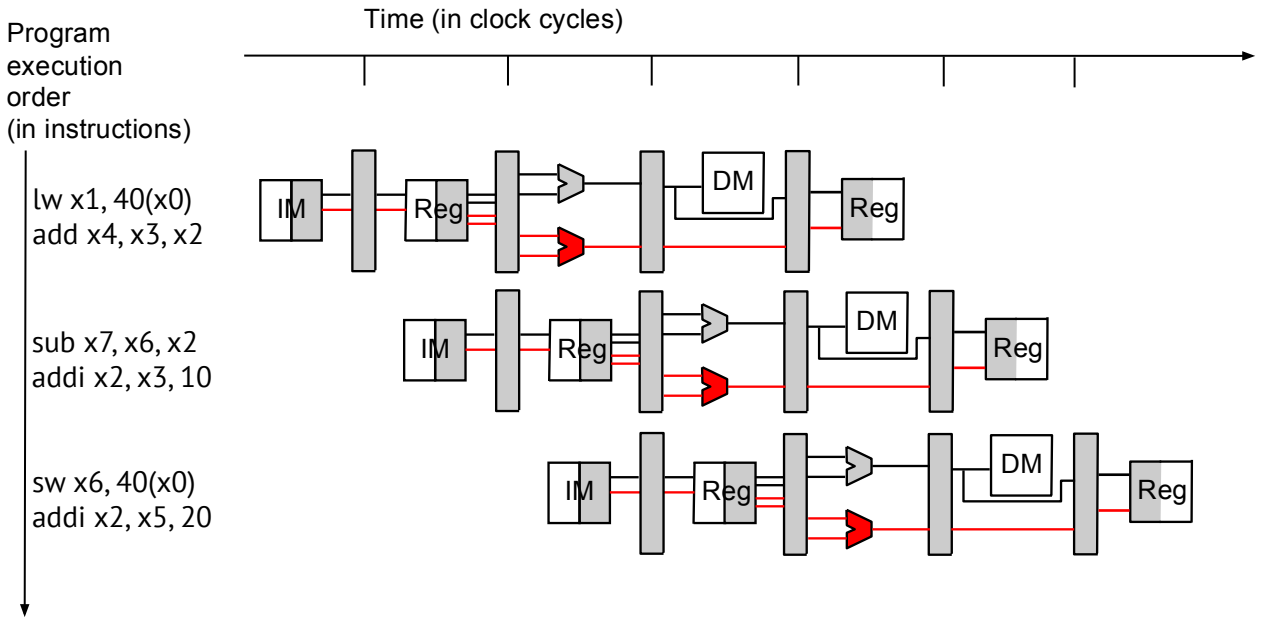
What now?

UniProcessor

- Threads
  - "Per-thread" state
    - Context state: PC, registers
    - Stack (per-thread local variables)
  - "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.

Long time ago…

- Instruction-Level Parallelism (ILP)



ILP

- ***Limitations***
  - Data Dependency
  - Poor Branch Prediction
    - Lots of flushes!
  - Memory Latency (aka memory "wall")

ILP

- ***Limitations***
  - Data Dependency
  - Poor Branch Prediction
    - Lots of flushes!
  - Memory Latency (aka memory "wall")

★ ***Out of order execution!***

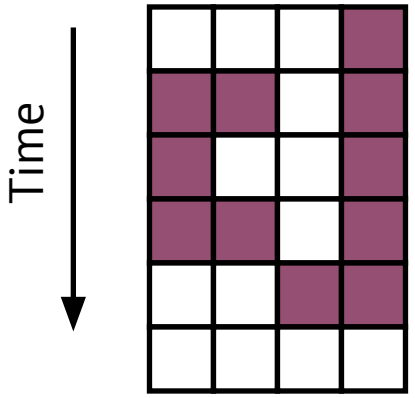
- *It can only solve some problems!*

ILP Limitations

Due to the limitations described in the previous slide, superscalar pipelines are typically ***underutilized.***

FU being used.

FU is idle.

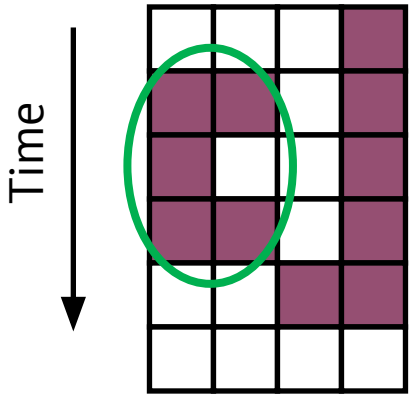


ILP Limitations

Due to the limitations described in the previous slide, superscalar pipelines are typically ***underutilized.***

FU being used.

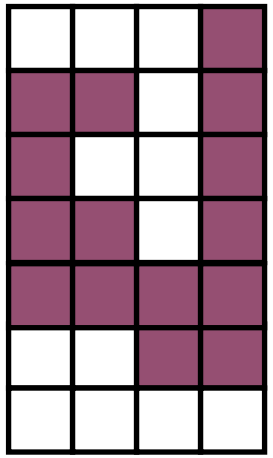
FU is idle.

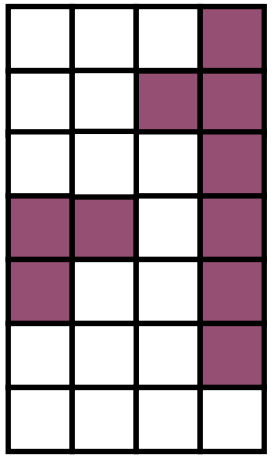


What can we do?

What can we do?

★ *Interleave programs!*





Simultaneous Multithreading (SMT)

- Execute multiple programs (threads) on the same pipeline.

➔ *SMT 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.*

SMT Datapath

- Example:

Multi-Tasking and Context Switch

- Running different processes/tasks on the same core.

- *Different from SMT!*

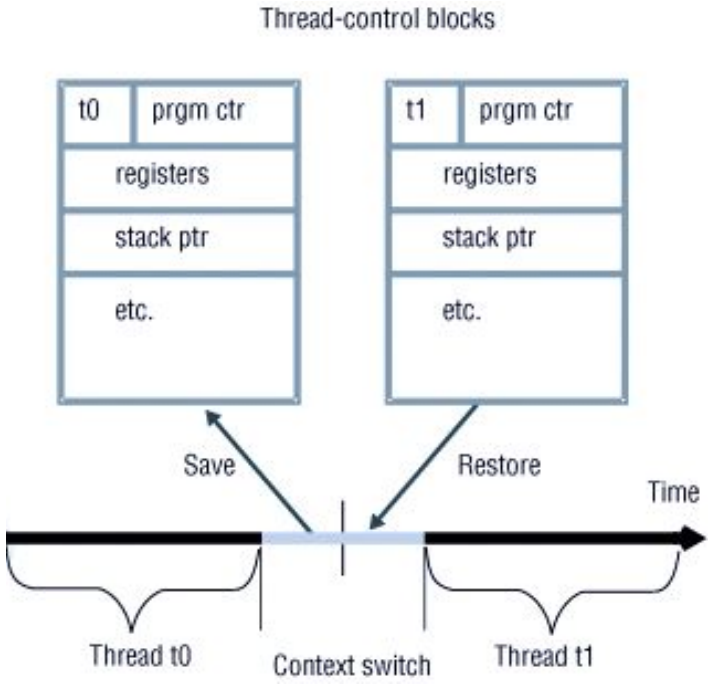


Image taken from IBM Knowledge Center [2].

Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Normal vs.CMT vs. FMT. vs. SMT

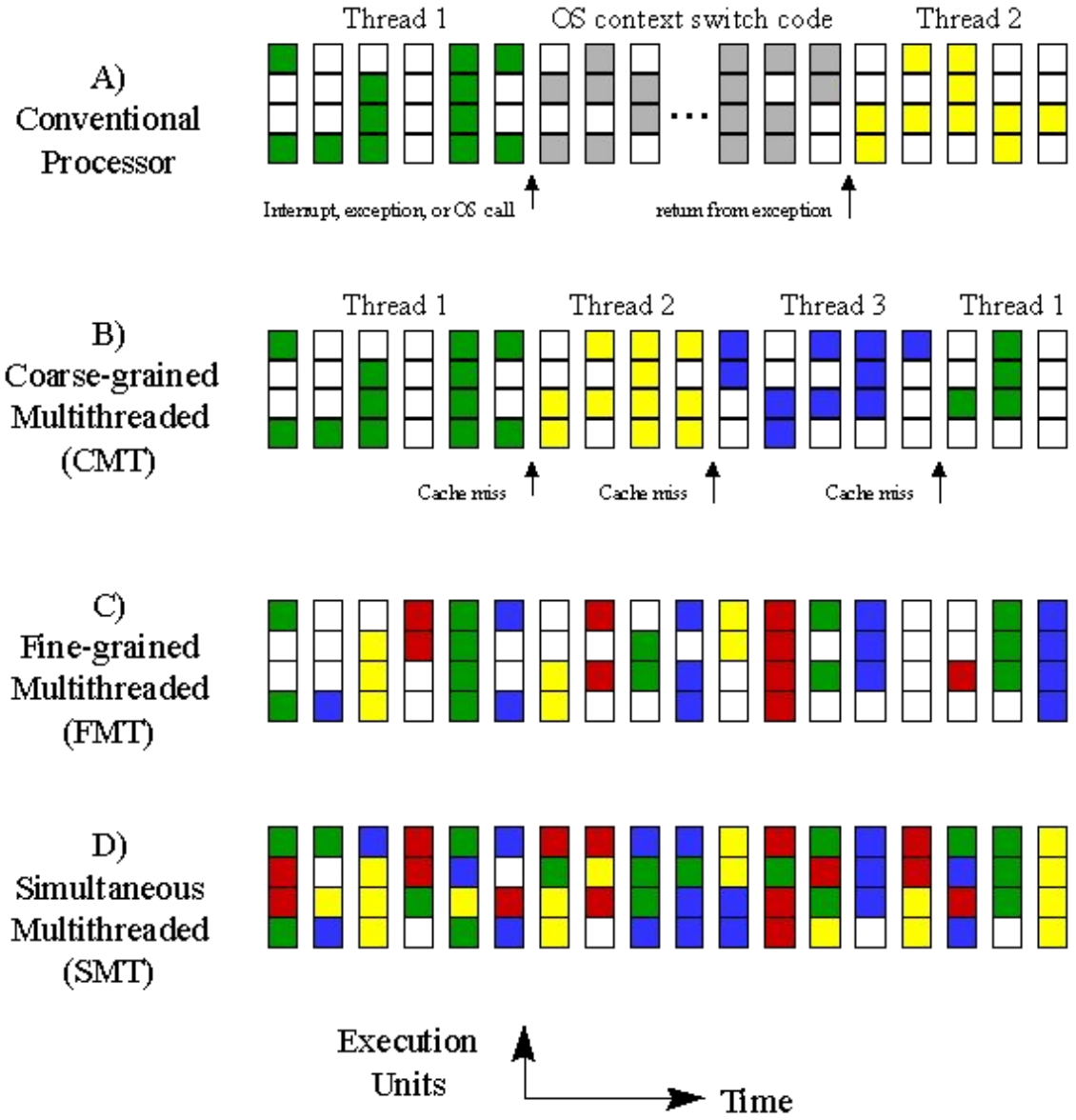


Image taken from https://www.realworldtech.com/alpha-ev8-smt/

Can we do better?

Multi-Core Systems

- How to leverage multiple cores
- Design choices
- Challenges

Multi-Core Systems

*Image was taken from: http://tomstechnicalblog.blogspot.com/2016/02/rxjava-understanding-observeon-and.html

Real-World Systems

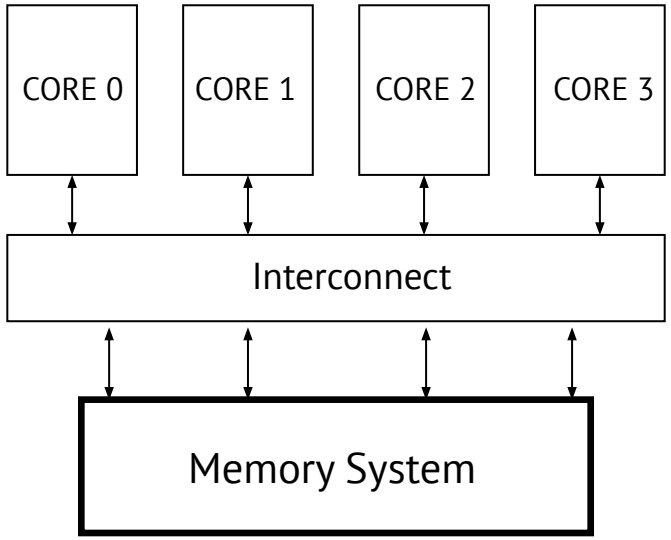
- *Server vs. Personal vs. Embedded*

- 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)

-- Intel uses codenames for each generation of its microarchitecture:

- *Skylake, Broadwell, Haswell, Comet Lake, …*

Multi-Core System



- *Interconnect*

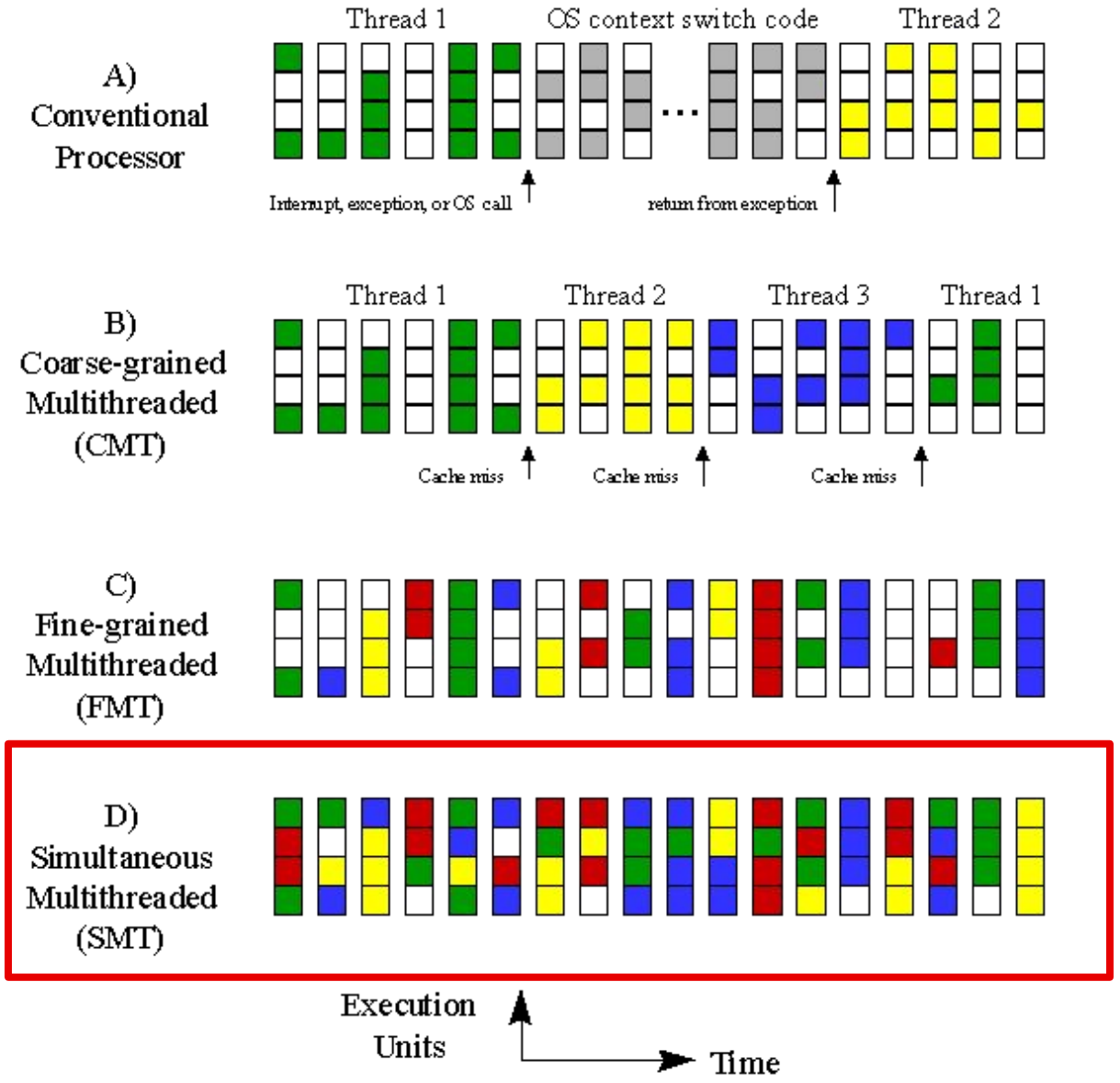
How multi-core helps?

1- Multitasking

2- Thread-Level Parallelism

- *There are not the same!*
- *Amdahls's law*

Multitasking



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)

- What we will mainly focus on

Thread-Level Parallelism

Example:

- Sum 64,000 numbers on 64 processors
  - Each processor has ID: 0 ≤ Pn ≤ 63
  - Partition 1000 numbers per processor
  - Initial summation on each processor

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

  - Now need to add these partial sums
    - Reduction: divide and conquer
    - Half the processors add pairs, then quarter, …

Thread-Level Parallelism

- *Difficult to extract (and write)!*
  - *Parallel algorithms and programming:* *MPI, Pthread, …*

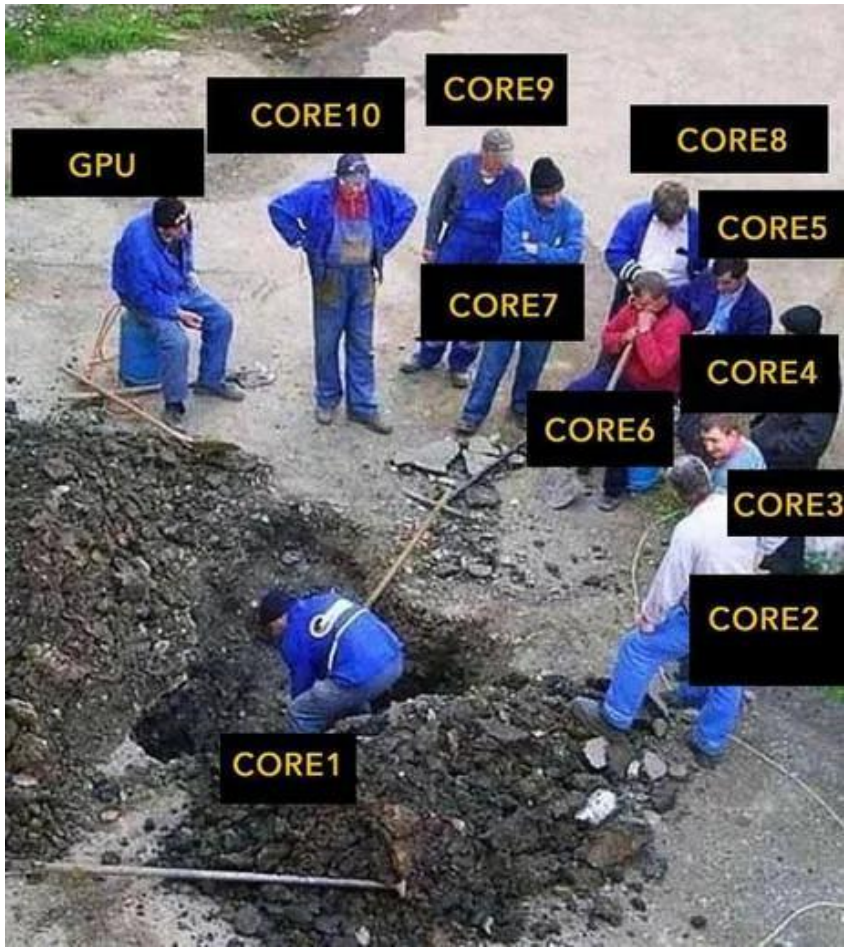
- *Communication overhead and sequential parts*

Thread-Level Parallelism

- *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.*


GPU
CORE10
CORE9
CORE8
CORE5
CORE7
CORE4
CORE6
CORE3
CORE2
CORE1

How to improve?

- Better algorithms
- Faster communication

→ Parallel Programming

How to build a multicore system?

What Changes?

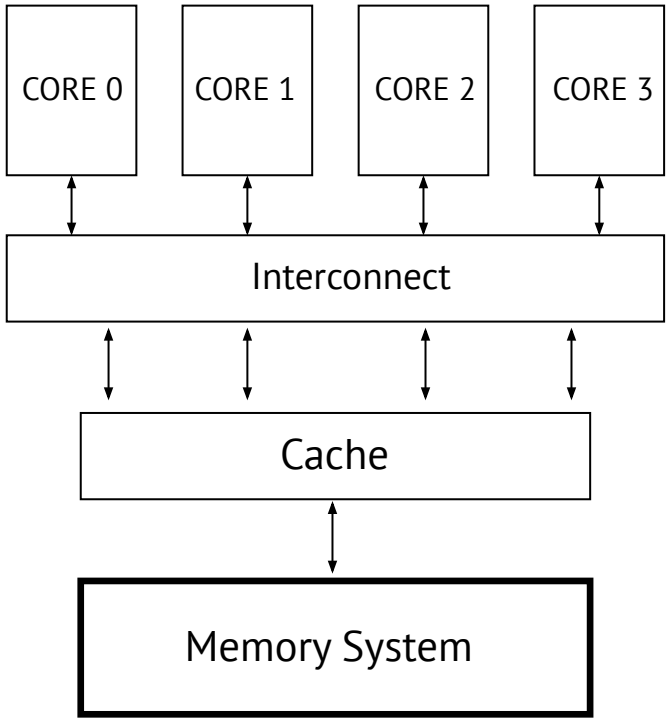
- Interaction with Memory!
- Multiple cores try to talk to the same memory!
- Data needs to be shared, otherwise, what is the point!

Multi-Core Design Choices

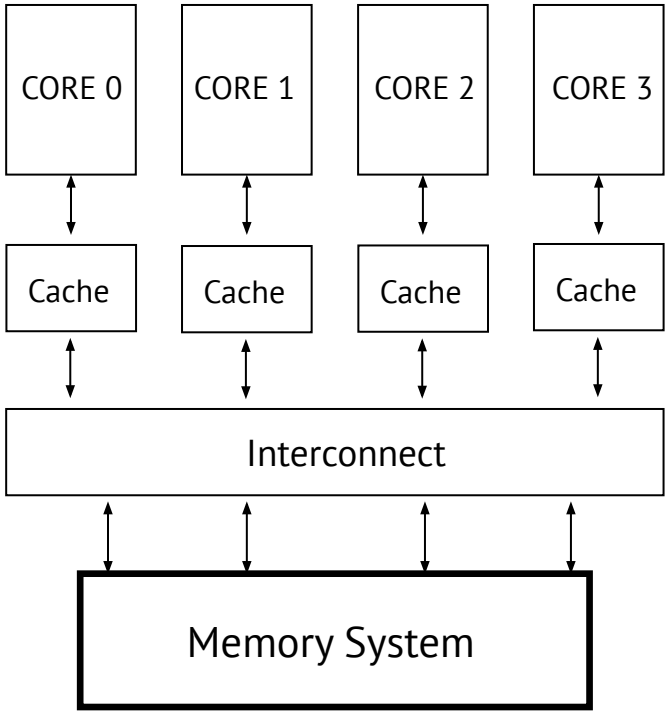
➔ *How to share the main memory between cores?*

➔ *What about caches?*

Shared

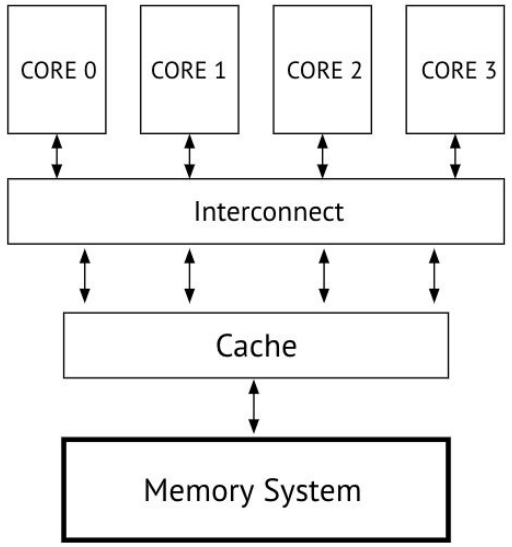


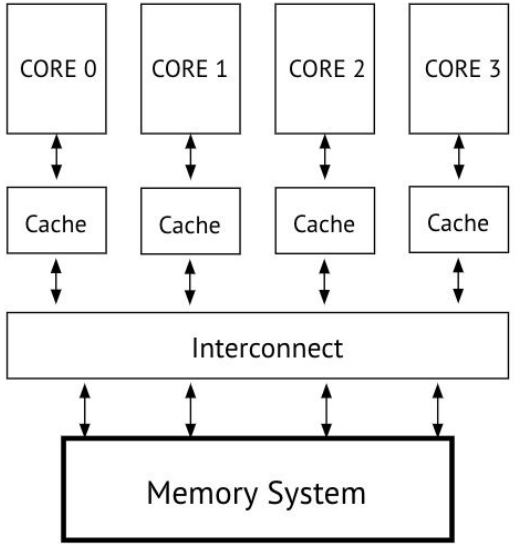
Private


CORE 0
CORE 1
CORE 2
CORE 3
Cache
Cache
Cache
Cache
Interconnect
Memory System

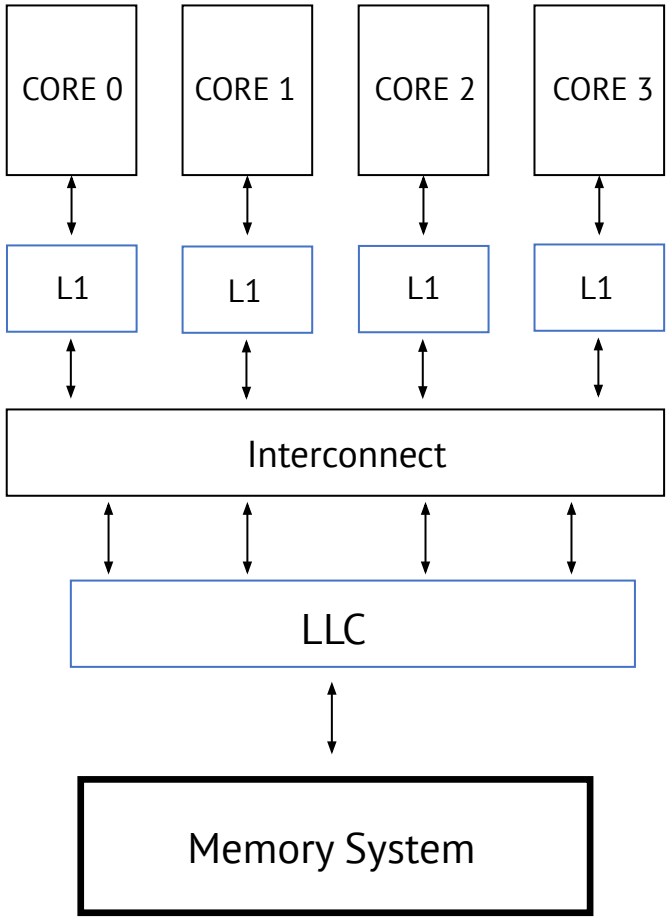
Shared vs. Private

- Hit time?
  - P: S:
- Miss rate?
  - Two scenarios:
    - Cache-friendly→ P: S:
    - Streaming application→ P: S:
- Miss Penalty?!
- What about sharing??





Hybrid Approach



- Best of both worlds!

Great! Let's use the hybrid and move on! Is that it?

Challenges

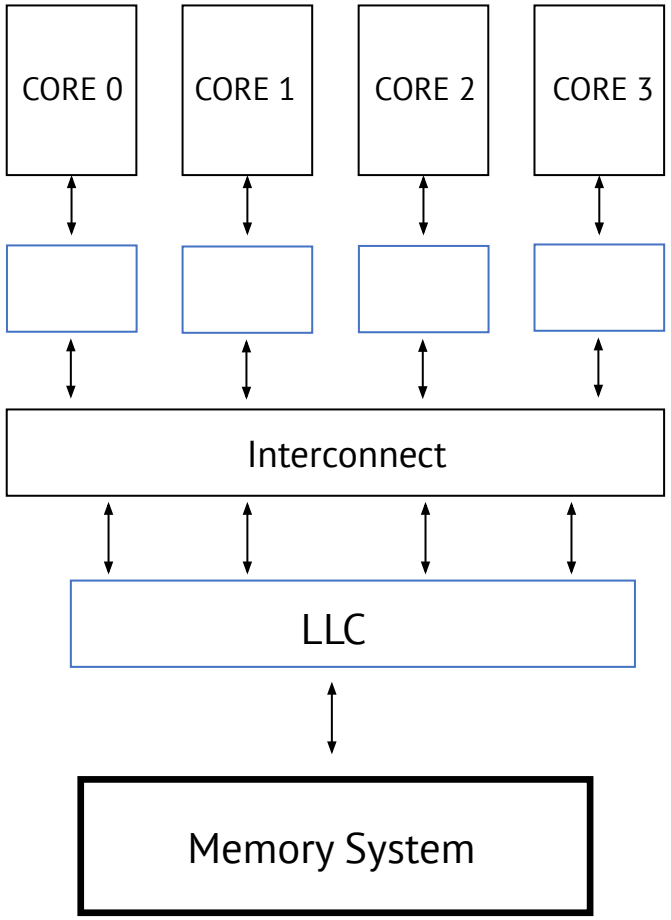
- Cache and memory is shared between cores and is simultaneously utilized by them.

1. This could lead to correctness issues! HOWW???
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*).

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
```

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?

Issues in Multicore

- Correctness
  - Coherency
  - Consistency

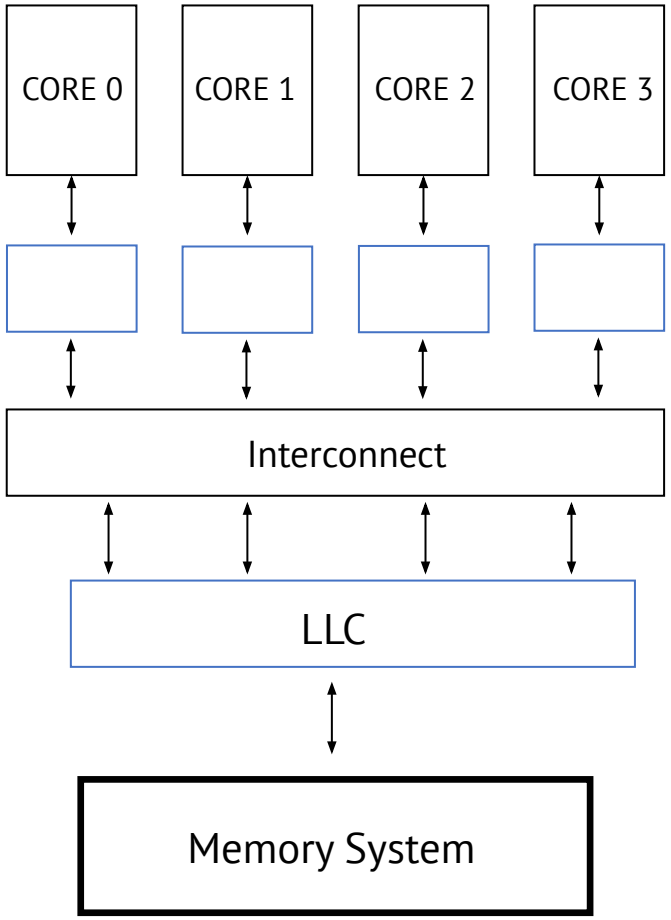
- Performance
  - Will talk about it later!

Issues in Multicore

- Correctness
  - Coherency
  - Consistency

- Performance
  - Will talk about it later!

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
```

Cache Coherency

- Uniformity of shared resource data that ends up stored in multiple local caches.

  (why we have shared data?)

- What if no sharing? What if no private cache?

Cache Coherency

- A memory system is coherent if:

  1. Read what is written: A read of X on P1 returns the value written by the most recent write to X on 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!

Can we prevent the cache coherence problem in software?

– *NO!*

- Cache is transparent (invisible) to software.

➔ *What about ISA?*

Can we prevent the cache coherence problem in software?

~~***NO!***~~ → ***maybe***

- Cache is transparent (invisible) to software.

➔ *What about ISA?*

- Special instruction(s) to flush a line and/or entire cache.
- *Very inefficient* (needed for every single access!)

Can we prevent the cache coherence problem in software?

~~***NO!***~~ → ***maybe***

- Cache is transparent (invisible) to software.

➔ *What about ISA?*

- Special instruction(s) to flush a line and/or entire cache.
- *Very inefficient* (needed for every single access!)

➔ *OS?*

Solution?

Solution?

- We need cache coherency protocols!
  - What should happen after each load or store for each core!
  - See something say something!

Solution?

★ *Use hardware to track each cache line!*

Solution?

★ *Use hardware to track each cache line!*

➔ *How?*

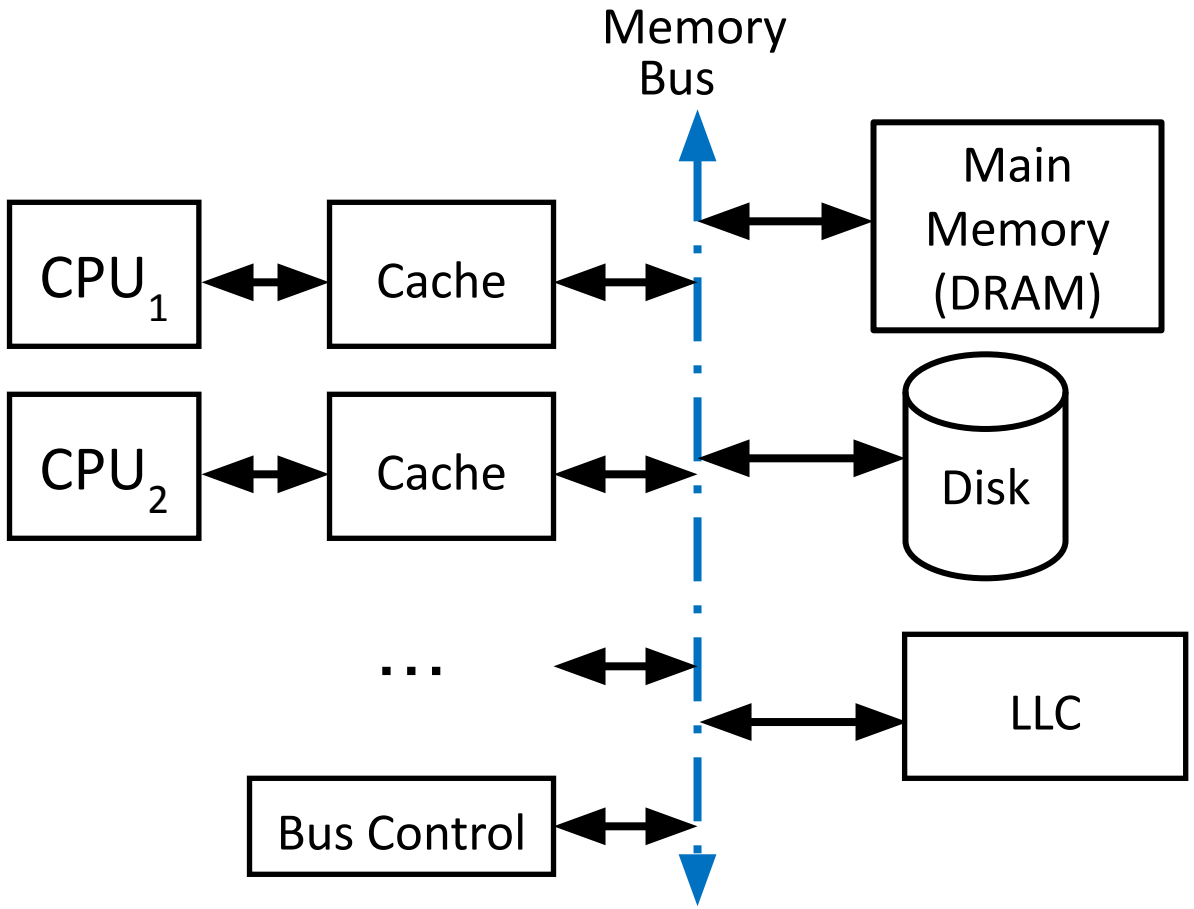
Solution?

★ *Use hardware to track each cache line!*

➔ *How?*

★ *"Snoop" the cache for each read/write.* ***Invalida*** *each copy if a new write comes.*

Snooping Cache



Each unit *"broadcasts"* its action to the bus.

Where do we store these?

- With lines! (metadata)
- Recall LRU states …

Snoopy Cache Coherence

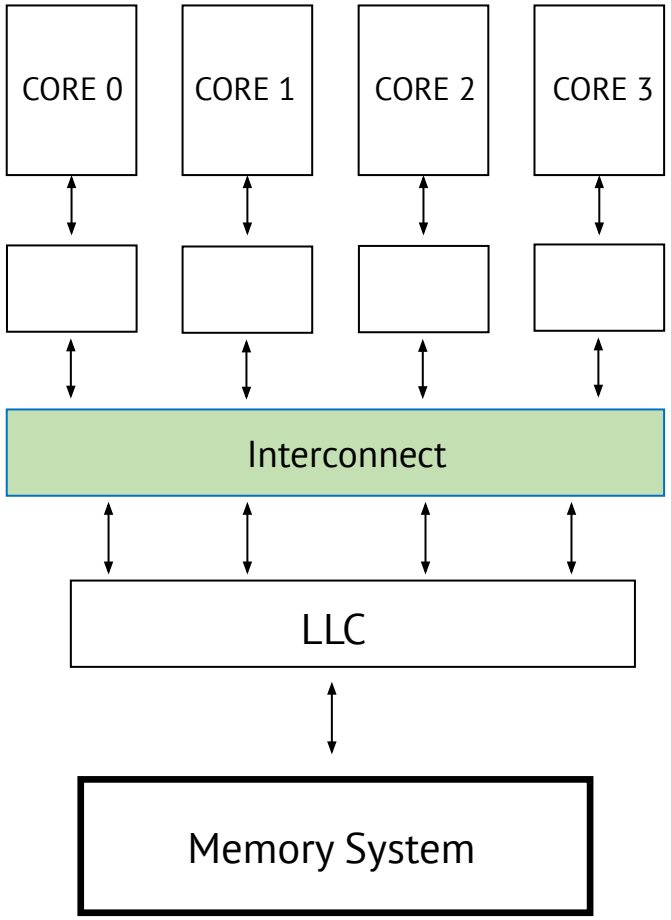
- *Write miss:*
  - The address is invalidated in all other caches before the write is performed.

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.
  - Otherwise, just read from the memory.

Cache Coherence Problem

with Snooping



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)

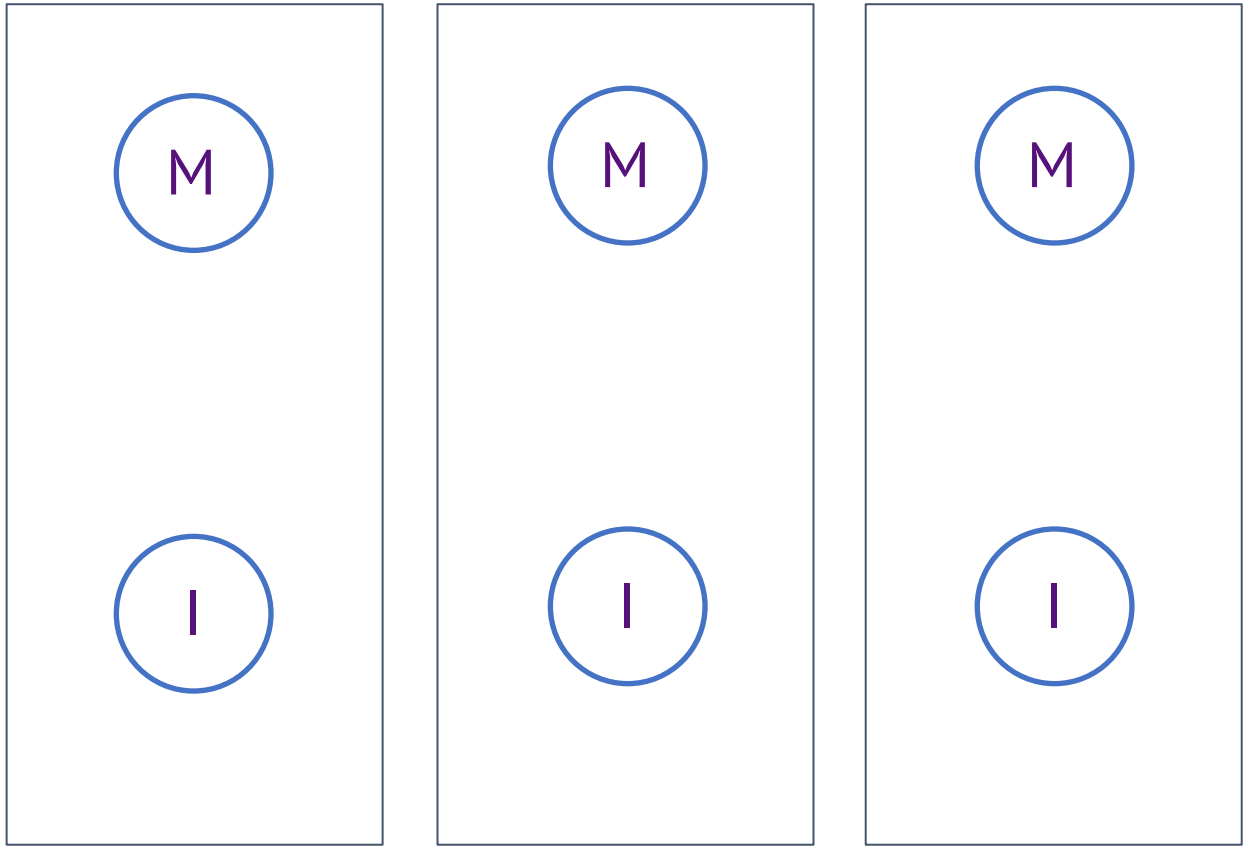
Coherence States

- 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

MI Coherence Protocol

All for same address



Core 1:
Core 2:
Core 3:

- Sequence:

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***
  - ***Modified***
  - ***Invalid***

Cache States

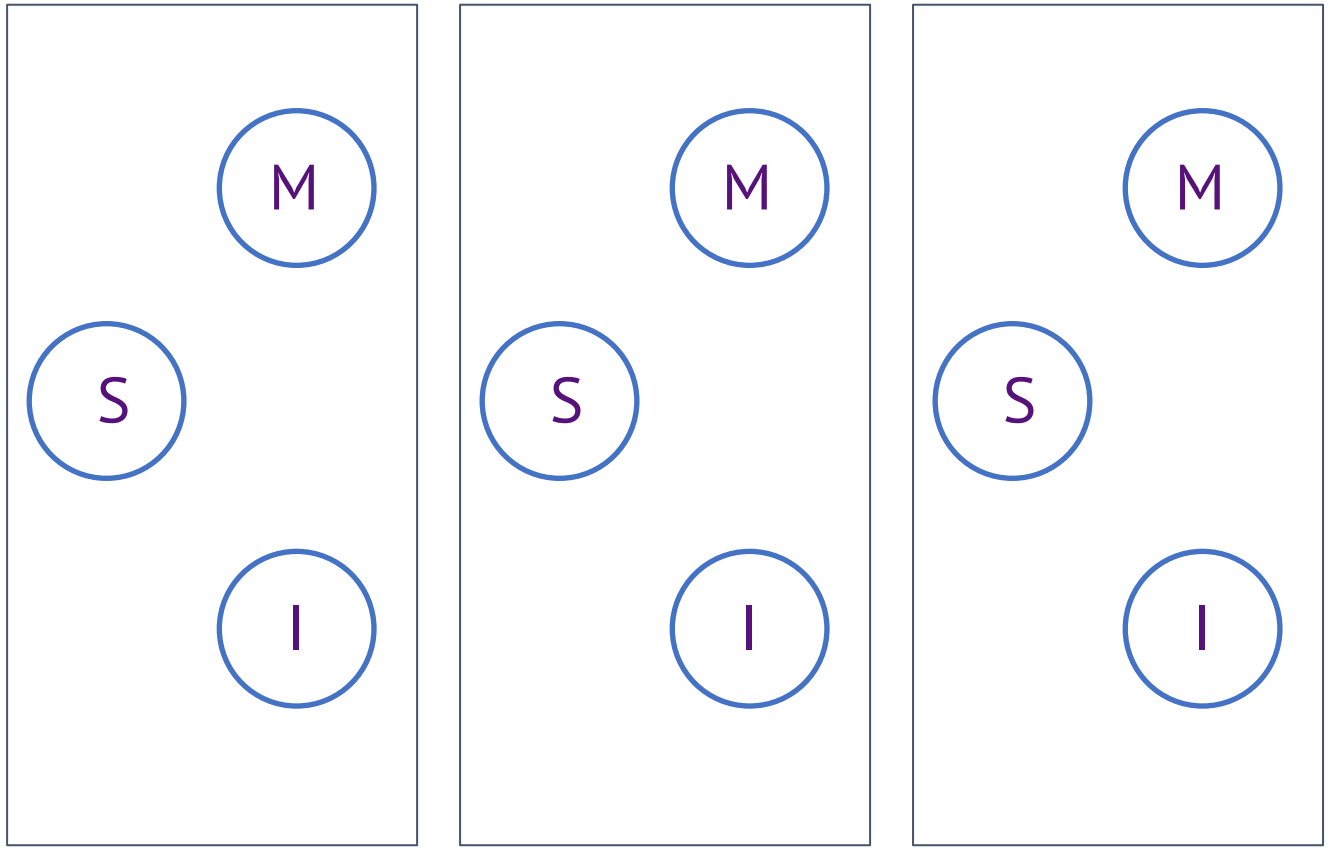
- Using snooping and broadcasting, each cache line (in each core) can have one of three possible ***states:***
  - ***Shared***
    - This line is a fresh and "clean" copy of main memory.
  - ***Modified***
    - This core wants to write to this line, thus it contains "dirty" copy of the line.
  - ***Invalid***
    - 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

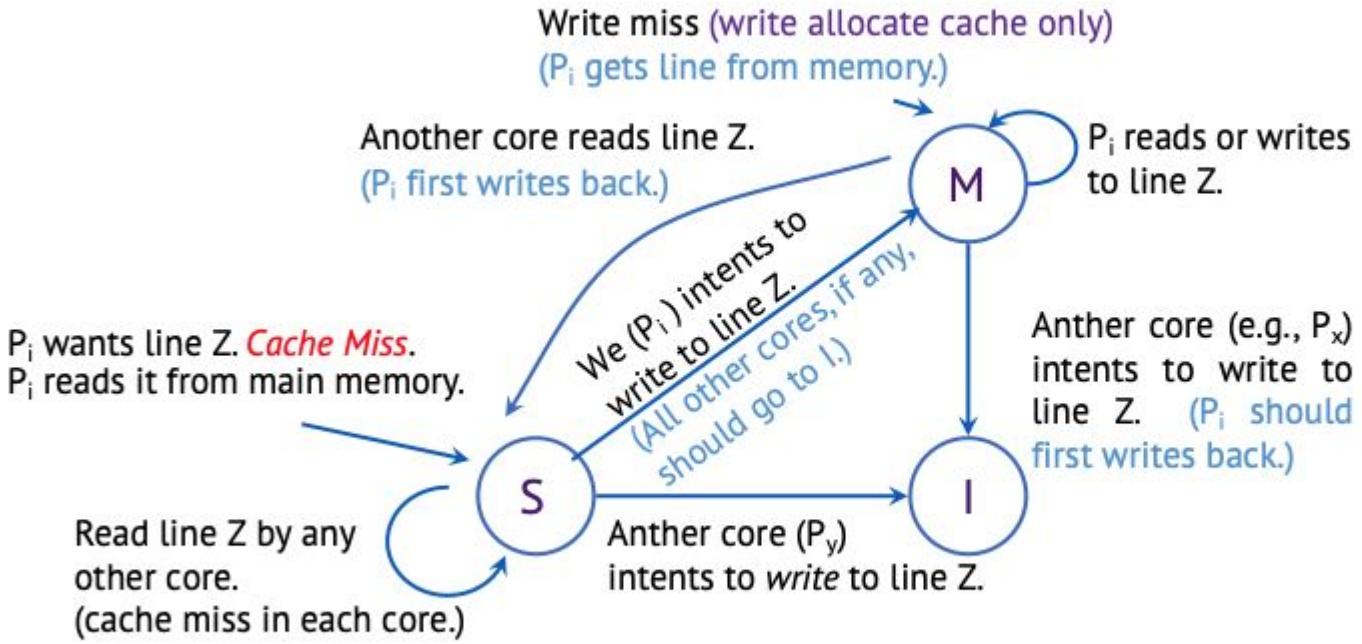
All for same address



Core 1:
Core 2:
Core 3:

- Sequence:

MSI (per core)


Write miss (write allocate cache only)
(P_i gets line from memory.)
Another core reads line Z.
(P_i first writes back.)
P_i reads or writes to line Z.
M
We (P_i) intents to write to line Z.
(All other cores, if any, should go to I.)
P_i wants line Z. *Cache Miss*.
P_i reads it from main memory.
Anther core (e.g., P_x) intents to write to line Z. (P_i should first writes back.)
S
I
Read line Z by any other core.
(cache miss in each core.)
Anther core (P_y) intents to *write* to line Z.

Optimizing MSI

➔ *How can we make MSI better?*

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).

MESI Example

Core 1:

Core 2:

Core 3:

- Sequence:

Optimizing MSI

➔ *How can we make MSI better?*

- *One core in M (others in I), a new core wants to read, who provides the data? Main memory or the core in M?*

Optimizing MSI

➔ *How can we make MSI better?*

- *One core in M (others in I), a new core wants to read, who provides the data? Main memory or the core in M?*

- How to decide?
  - Add a new state called O.

MOSI Protocol

- Same as MSI except:
  - If on M and a new core wants to read, it goes to O and sends the data to the other core (no need for write back and no need for a memory read).
  - If on O (or M) and the other core want to write, it writes back and invalidates.

MOSI Example

Core 1:

Core 2:

Core 3:

- Sequence:

MOESI

- We can have both E and O at the same time!
- Benefit from both E and O states simultaneously!

Optimizing MSI

➔ *How can we make MSI better?*

- *Multiple cores in S, a new core wants to read, who provides the data?*

Optimizing MSI

➔ *How can we make MSI better?*

- *Multiple cores in S, a new core wants to read, who provides the data?*

- Add a new state called forwarder (F)
  - *Instead of going to S, from E we can go to F.*

MOESIF

- Putting the all together!
  - I⟶ E ⟶ F
  - I⟶ M ⟶ O
  - I⟶ E ⟶ M (silent)

MOESIF Example

Core 1:

Core 2:

Core 3:

- Sequence:

Optimizing MSI – *Summary*

➔ *How can we make MSI better?*

- Unnecessary broadcasts.
- Extra write-backs to main memory.
- Cache-to-cache transfers.

False Sharing

- A cache line contains more than one block.
- We need blocks but cache-coherence is done at the ***line-level!***

False Sharing

- False sharing *misses* when a line is invalidated because some block in the line, other than the one being read, is written into.

- This is called **coherence miss** (recall 3Cs).

- Example:
  - 200 and 202

Is snooping the only option?

– NO!

- The main issue with snooping is *scalability* (each core has to broadcast to every core).

➔ *What can we do instead?*

Directory-Based Coherence Protocol

- Have a directory in each cache!
  - Keeps track of copies of cached lines and their states, who has it?
  - On a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary.
  - No need for snooping and broadcasting.

End of Presentation

Acknowledgement

- This course is partly inspired by the following courses created by my colleagues:
  - CS152, Krste Asanovic (UCB)
  - 18-447, James C. Hoe (CMU)
  - CSE141, Steven Swanson (UCSD)
  - CIS 501, Joe Devietti (Upenn)
  - CS4290, Tom Conte (Georgia Tech)
  - 252-0028-00L, Onur Mutlu (ETH)