Multi -threading

Part 1

Three basic techniques of computer systems design

- Locality of access
- Parallelism
- Speculation/prediction

Three basic techniques of computer systems design

- Locality of access
- Parallelism
- Speculation/prediction

Types of parallelism

- Instruction level parallelism (ILP)

- Memory level parallelism (MLP)

Types of parallelism

- Instruction level parallelism (ILP)
  - **Implicit** parallelism in a **single** stream of instruction
    - Independent instructions in a stream
    - Extracted by compiler or hardware (e.g., OoO hardware)
- Memory level parallelism (MLP)
  - Implicit parallelism among memory instructions
  - Exploited by non-blocking caches

Implicit Parallelism

- HW (or compiler) needs to find parallelism
  - HW presented with a single stream of sequential instructions
    - To software it should look like the instructions executed one after another
    - HW needs to find out which instructions/operations can be executed in parallel without breaking the sequential semantics

Implicit Parallelism

- HW (or compiler) needs to find parallelism
  - HW presented with a single stream of sequential instructions
    - To software it should look like the instructions executed one after another
    - HW needs to find out which instructions/operations can be executed in parallel without breaking the sequential semantics
- Good for software: Users got “free” performance just by buying a new chip
  - No change needed to the program (same ISA)
  - Higher frequency (smaller, faster transistors)
  - Higher IPC (different micro-arch)
  - But this was not sustainable…

Implicit parallelism not enough

- OoO superscalars extract ILP from sequential programs
  - Hardly more than 1-2 IPC on real workloads
- In practice, IPC is limited by:
  - ILP available in the program (single thread)
    - True data dependences
    - Coming from algorithm and compiler

Implicit parallelism not enough

- OoO superscalars extract ILP from sequential programs
  - Hardly more than 1-2 IPC on real workloads
- In practice, IPC is limited by:
  - ILP available in the program (single thread)
    - True data dependences
    - Coming from algorithm and compiler
  - Limited BW
    - From memory and cache
    - Fetch/commit bandwidth
    - Renaming (must find dependences among all insns dispatched in a cycle)

Implicit parallelism not enough

- OoO superscalars extract ILP from sequential programs
  - Hardly more than 1-2 IPC on real workloads
- In practice, IPC is limited by:
  - ILP available in the program (single thread)
    - True data dependences
    - Coming from algorithm and compiler
  - Limited BW
    - From memory and cache
    - Fetch/commit bandwidth
    - Renaming (must find dependences among all insns dispatched in a cycle)
  - Limited HW resources
    - # ROB, RS and LSQ entries, functional units

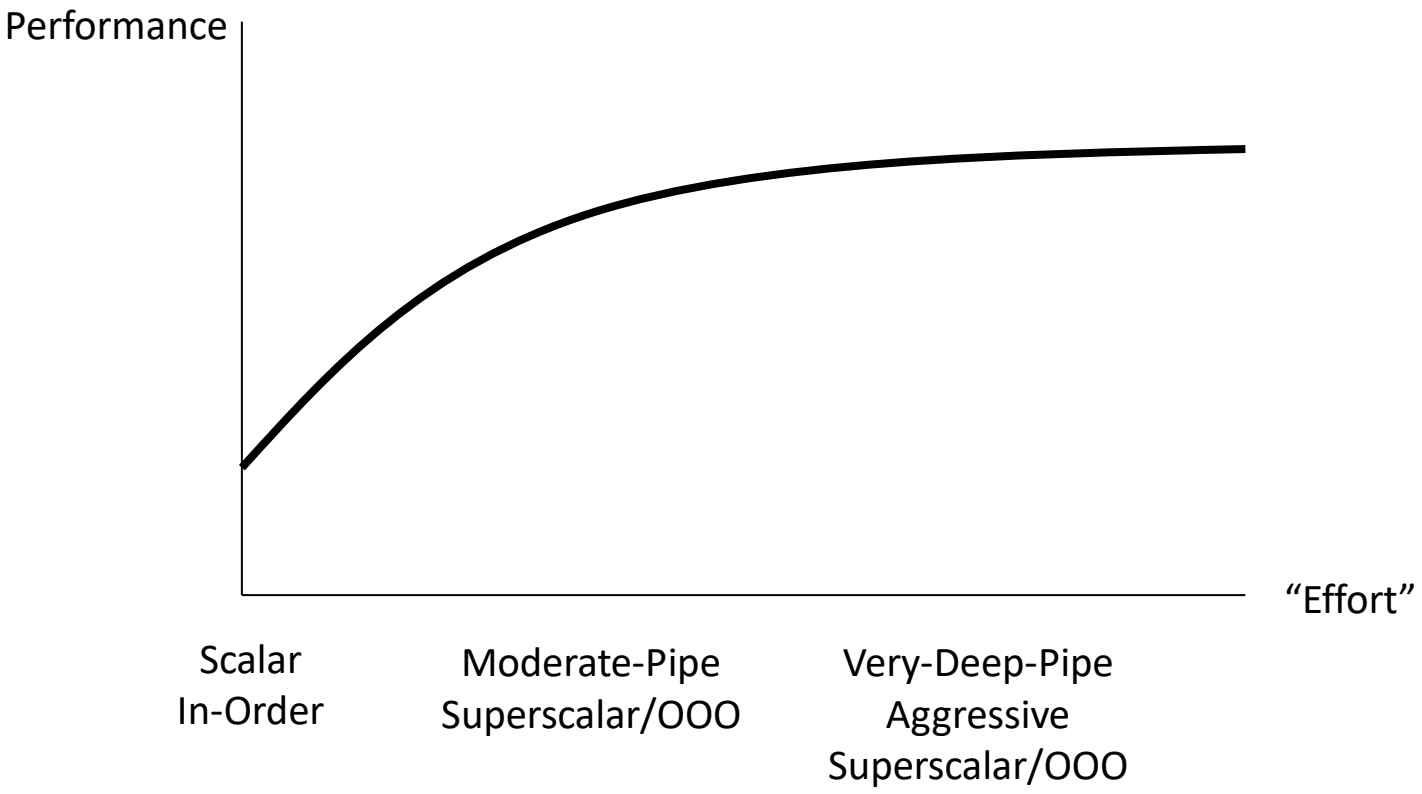
Implicit parallelism not enough

- OoO superscalars extract ILP from sequential programs
  - Hardly more than 1-2 IPC on real workloads
- In practice, IPC is limited by:
  - ILP available in the program (single thread)
    - True data dependences
    - Coming from algorithm and compiler
  - Limited BW
    - From memory and cache
    - Fetch/commit bandwidth
    - Renaming (must find dependences among all insns dispatched in a cycle)
  - Limited HW resources
    - # ROB, RS and LSQ entries, functional units
  - Branch prediction accuracy
  - Imperfect memory disambiguation

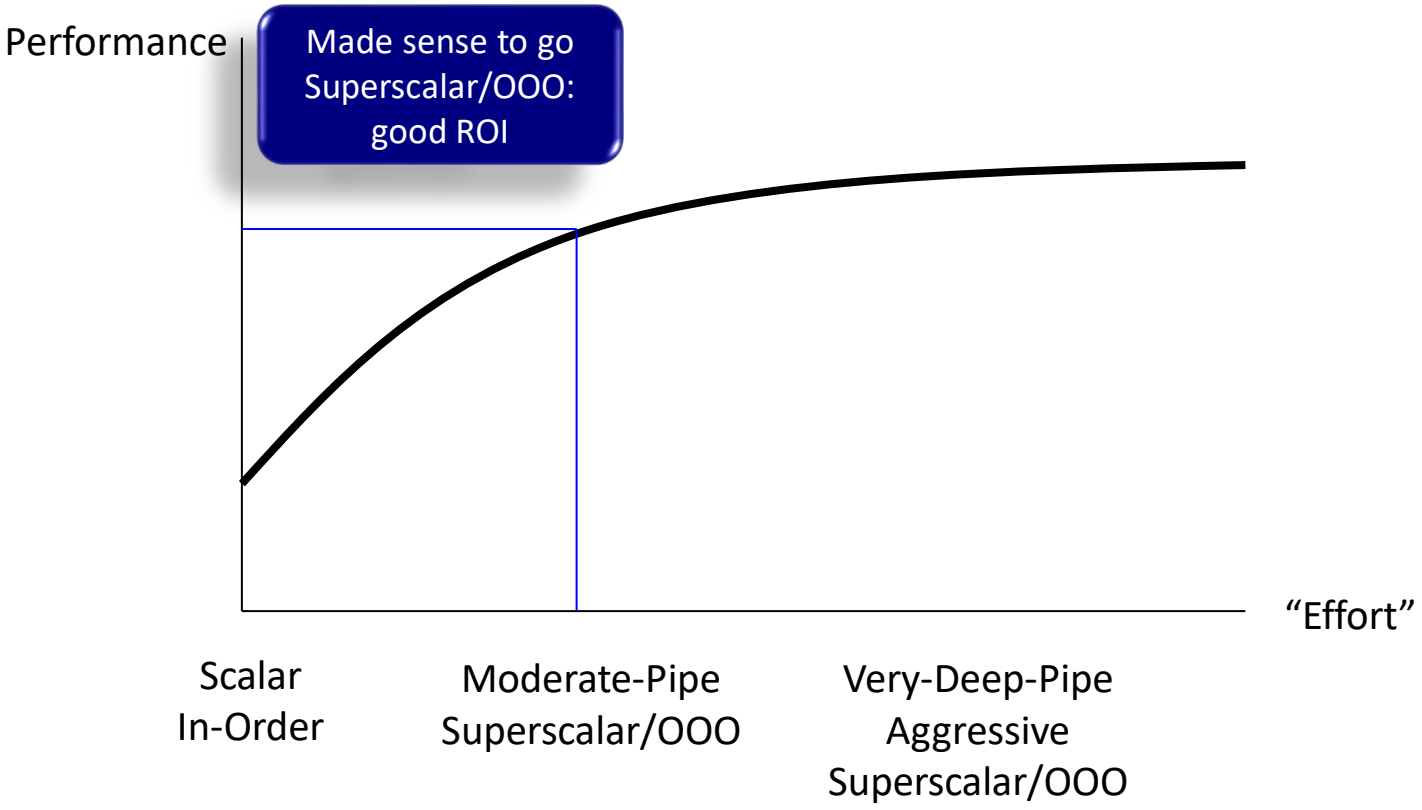
Implicit parallelism not enough

- To get more performance, we can keep pushing IPC and/or frequency
  - Design complexity (time to market)
  - Cooling (cost)
  - Power delivery (cost)
  - …
- But it is too costly for the marginal improvements gained

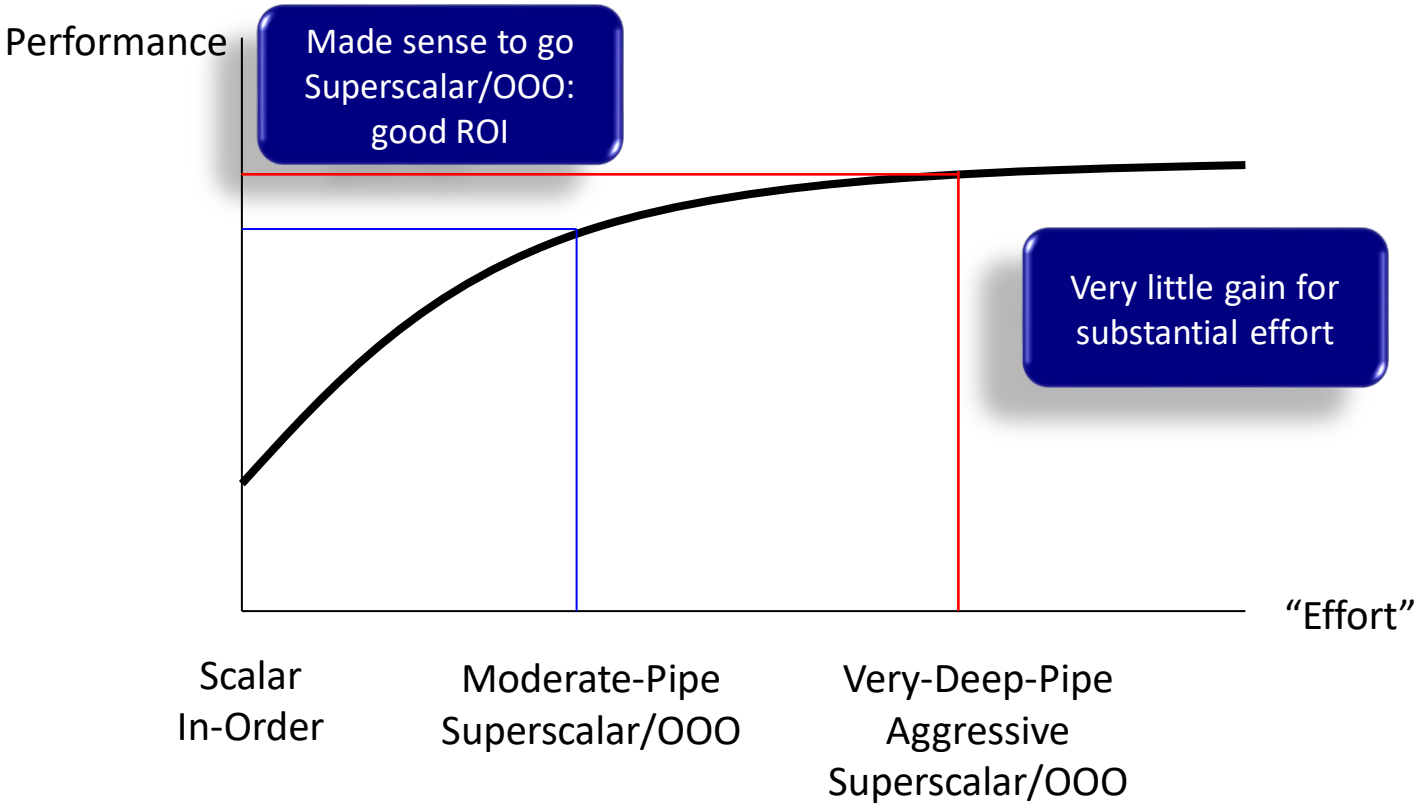
Diminishing return on more complexity



Diminishing return on more complexity



Diminishing return on more complexity



Need to increase utilization

- Adding more functional units (e.g., ALU) is relatively easy
  - Floating point units are more costly but transistors have been becoming smaller and cheaper

Need to increase utilization

- Adding more functional units (e.g., ALU) is relatively easy
  - Floating point units are more costly but transistors have been becoming smaller and cheaper
- The challenge is in finding enough parallelism in keeping all of them busy at a time
  - Almost impossible beyond a point, if there is a single sequential stream of instructions.

Types of parallelism

- Instruction level parallelism (ILP)
  - **Implicit** parallelism in a **single** stream of instruction
    - Independent instructions in a stream
    - Extracted by compiler or hardware (e.g., OoO hardware)
- Memory level parallelism (MLP)
  - Implicit parallelism among memory instructions
  - Exploited by non-blocking caches

Implicit parallelism

Types of parallelism

- Instruction level parallelism (ILP)
  - **Implicit** parallelism in a **single** stream of instruction
    - Independent instructions in a stream
    - Extracted by compiler or hardware (e.g., OoO hardware)
- Memory level parallelism (MLP)
  - Implicit parallelism among memory instructions
  - Exploited by non-blocking caches
- Thread level parallelism (TLP)
  - **Multiple independent** stream of instructions
    - **Explicitly** specified by the application

Implicit parallelism

Types of parallelism

- Instruction level parallelism (ILP)
  - **Implicit** parallelism in a **single** stream of instruction
    - Independent instructions in a stream
    - Extracted by compiler or hardware (e.g., OoO hardware)
- Memory level parallelism (MLP)
  - Implicit parallelism among memory instructions
  - Exploited by non-blocking caches
- Thread level parallelism (TLP)
  - **Multiple independent** stream of instructions
    - **Explicitly** specified by the application

Implicit parallelism

Explicit parallelism

Explicit Parallelism

- **HW user** (programmer, compiler) responsible for finding and expressing parallelism
  - HW does **not need** to allocate resources to find parallelism
    → Simpler, more efficient HW

Explicit Parallelism

- **HW user** (programmer, compiler) responsible for finding and expressing parallelism
  - HW does **not need** to allocate resources to find parallelism → Simpler, more efficient HW
- Common forms
  - ***Thread-Level Parallelism (TLP)***: Hardware Multithreading, Multiprocessors

Explicit Parallelism

- **HW user** (programmer, compiler) responsible for finding and expressing parallelism
  - HW does **not need** to allocate resources to find parallelism → Simpler, more efficient HW
- Common forms
  - ***Thread-Level Parallelism (TLP)***: Hardware Multithreading, Multiprocessors
  - ***Data-Level Parallelism (DLP)***: Vector processors, SIMD extensions, GPUs
  - ***Request-Level Parallelism (RLP)***: Data centers

Sources of TLP

- Different applications
  - MP3 player in background while you work in Office
  - Other background tasks: OS/kernel, virus check, etc…
  - Piped applications
    - gunzip -c foo.gz | grep bar | perl some-script.pl

Multiprogramming

Sources of TLP

- Different applications
  - MP3 player in background while you work in Office
  - Other background tasks: OS/kernel, virus check, etc…
  - Piped applications
    - gunzip -c foo.gz | grep bar | perl some-script.pl
- Threads within the same application
  - Explicitly coded multi-threading
    - pthreads
  - Parallel languages and libraries
    - OpenMP, Cilk, TBB, etc…

Multiprogramming

Multi-threading

Architectures to Exploit TLP

- ***Hardware Multithreading (MT)***: Multiple threads **share the same processor** pipeline
  - Coarse-grained MT (CGMT)
  - Fine-grained MT (FMT)
  - Simultaneous MT (SMT)

Architectures to Exploit TLP

- ***Hardware Multithreading (MT)***: Multiple threads **share the same processor** pipeline
  - Coarse-grained MT (CGMT)
  - Fine-grained MT (FMT)
  - Simultaneous MT (SMT)
- ***Multiprocessors (MP):*** Different threads run on **different processors**
  - Multiple processor chips
  - Chip Multiprocessors (CMP), a.k.a. Multicore processors
  - A combination of the above

Latency vs Throughput

- **MT trades (single-thread) latency for throughput**
  - – Sharing processor degrades latency of individual threads
  - + But improves aggregate throughput
  - + Improves utilization
- Example
  - Thread A: individual latency=10s, latency with thread B=15s
  - Thread B: individual latency=20s, latency with thread A=25s
  - Sequential latency (first A then B or vice versa): 30s
  - Parallel latency (A and B simultaneously): 25s
  - – MT slows each thread by 5s
  - + But finishes both work 5s earlier
- **Different workloads have different parallelism**
  - SpecFP has lots of ILP (can even use an 8-wide machine)
  - Server workloads have TLP (can use multiple threads)

MT Implementations: Sharing

- How do multiple threads share a single processor?
  - Different sharing mechanisms for different kinds of structures
  - Depend on what kind of state structure stores

MT Implementations: Sharing

- How do multiple threads share a single processor?
  - Different sharing mechanisms for different kinds of structures
  - Depend on what kind of state structure stores

- **No state**: ALUs
  - Dynamically shared
- **Persistent hard state (aka "context")**: PC, registers
  - Anything exposed to the s/w
  - Replicated
- **Persistent soft state**: caches, branch predictor
  - Dynamically partitioned (like on a multi-programmed uni-processor)
    - TLBs need ASIDs, caches/bpred tables don't
  - Exception: **ordered "soft" state** (BHR, RAS) is replicated
- **Transient state**: ROB, RS
  - Partitioned … somehow (will see later)

MT Implementations: New design issues

- Main question: **thread scheduling policy**
  - When to switch from one thread to another?
- Related question: **pipeline partitioning**
  - How exactly do threads share the pipeline itself?

- Choice depends on
  - What kind of latencies you want to tolerate
  - How much single thread performance you are willing to sacrifice

MT Implementations: New design issues

- Main question: **thread scheduling policy**
  - When to switch from one thread to another?
- Related question: **pipeline partitioning**
  - How exactly do threads share the pipeline itself?

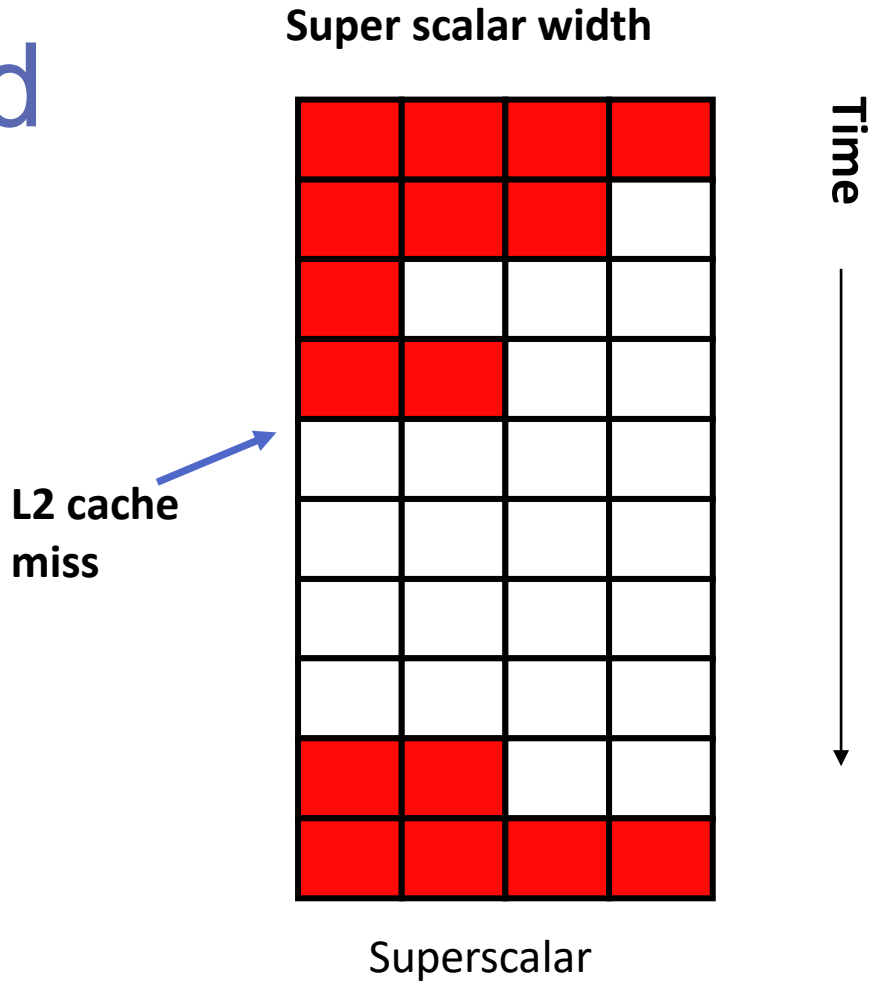
- Choice depends on
  - What kind of latencies you want to tolerate
  - How much single thread performance you are willing to sacrifice

- Three designs
  - Coarse-grain multithreading (CGMT)
  - Fine-grain multithreading (FGMT)
  - Simultaneous multithreading (SMT)

Thread 1 Thread 2

Coarse -Grained Multithreading

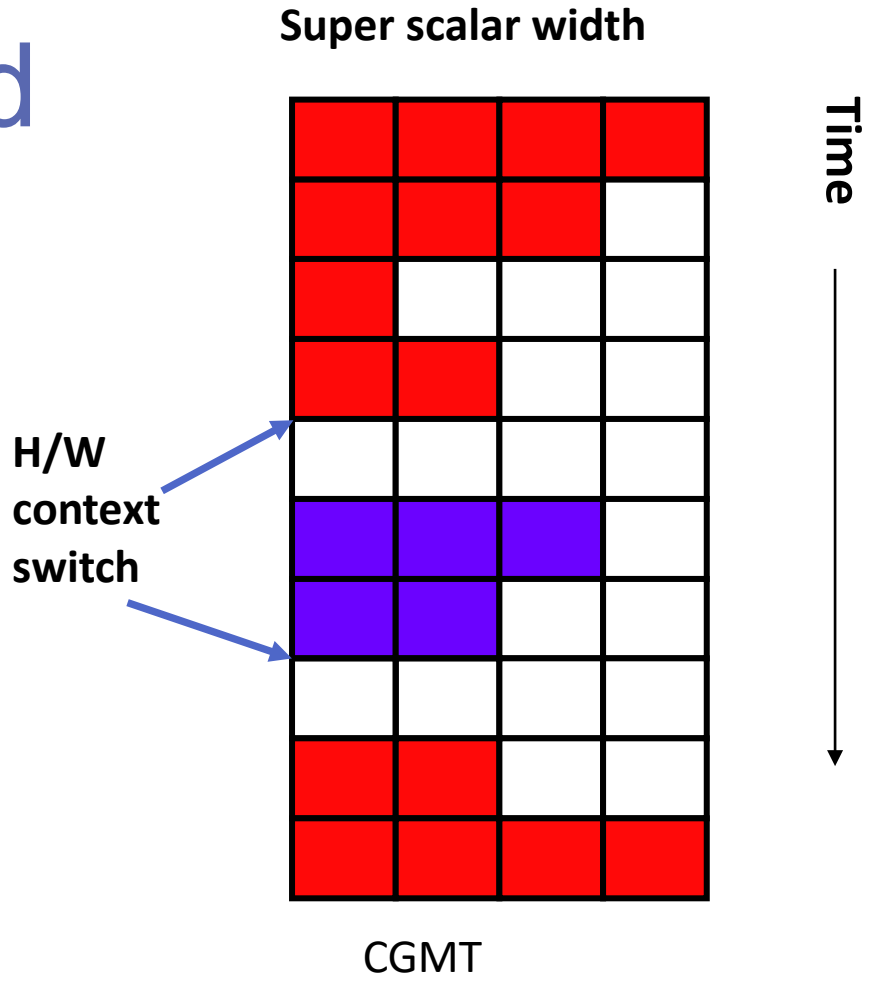
- The OS should have already scheduled both threads on the CPU
  - **But only one thread in the pipeline at any time**
- Hardware switches to another thread when current thread stalls on a long latency event
  - E.g., L2 miss
- Example: IBM Northstar/Pulsar



Thread 1 Thread 2

Coarse -Grained Multithreading

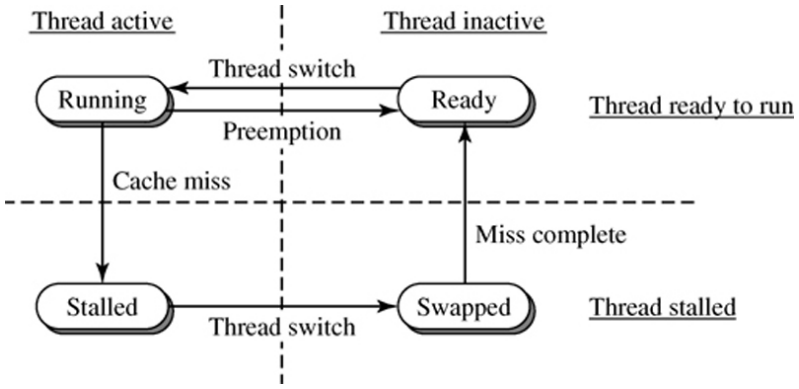
- The OS should have already scheduled both threads on the CPU
  - **But only one thread in the pipeline at any time**
- Hardware switches to another thread when current thread stalls on a long latency event
  - E.g., L2 miss
- Example: IBM Northstar/Pulsar



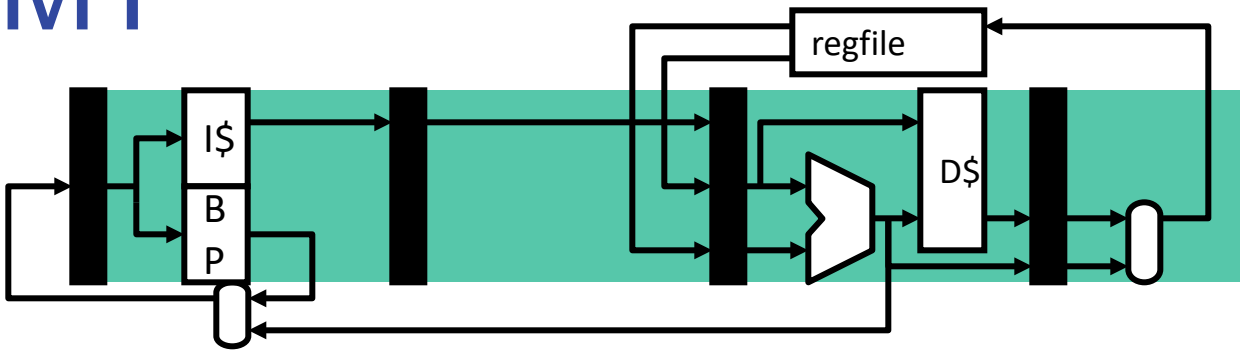
Coarse -Grained Multithreading

- Needs HW “preemption” and “priority” mechanisms to ensure fairness and high utilization
  - Different from OS preemption and priority
  - e.g., HW “preempts” long running threads with no L2 miss
  - High “priority” means thread should not be preempted
    - e.g., when in a critical section
    - Priority changes communicated using special instructions

Thread State Transition Diagram in a CGMT Processor

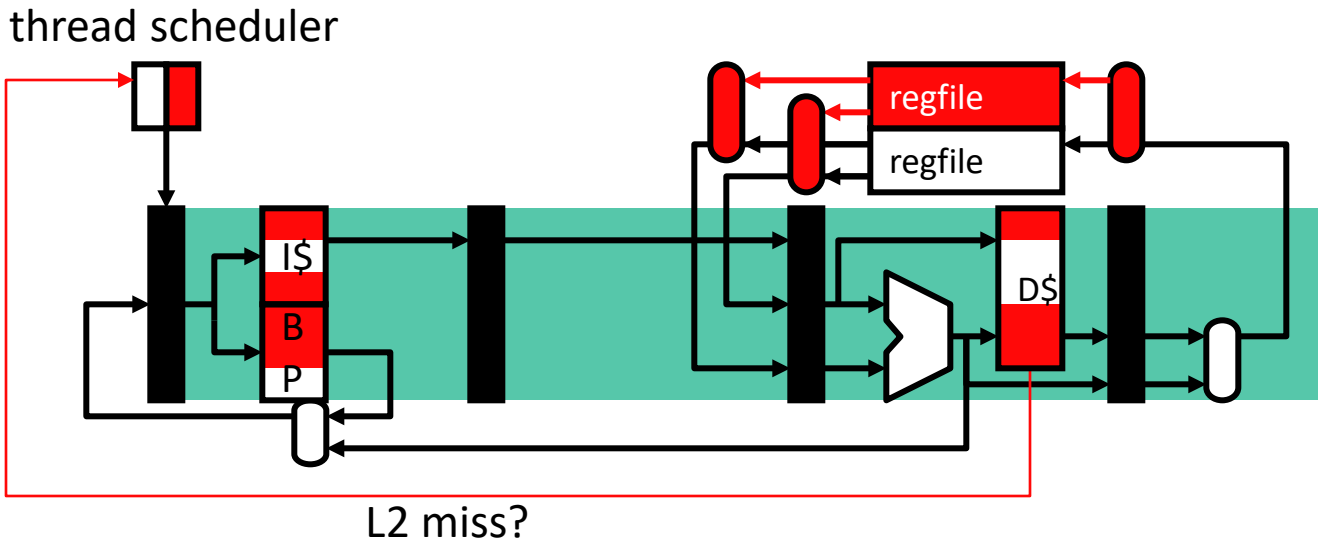


CGMT



Normal pipeline

- CGMT



Coarse -Grained Multithreading

- ✓ ✗ Sacrifices a little single thread performance
- ✗ Tolerates only long latencies (e.g., L2 misses)
  - ✗ Only eliminating some of the vertical waste

Coarse-Grained Multithreading

✓✗ Sacrifices a little single thread performance

✗ Tolerates only long latencies (e.g., L2 misses)

- ✗ Only eliminating some of the vertical waste

- A possible thread scheduling policy
  - Designate a "preferred" thread (e.g., thread A)
  - Switch to thread B on thread A L2 miss
  - Switch back to A when A L2 miss returns

Coarse-Grained Multithreading

✓✗ Sacrifices a little single thread performance

✗ Tolerates only long latencies (e.g., L2 misses)

- ✗ Only eliminating some of the vertical waste

- A possible thread scheduling policy
  - Designate a "preferred" thread (e.g., thread A)
  - Switch to thread B on thread A L2 miss
  - Switch back to A when A L2 miss returns

- Pipeline partitioning
  - None, flush on switch
  - Need short in-order pipeline for good performance
    - High switch cost otherwise

Coarse-Grained Multithreading

✓✗ Sacrifices a little single thread performance

✗ Tolerates only long latencies (e.g., L2 misses)

- ✗ Only eliminating some of the vertical waste

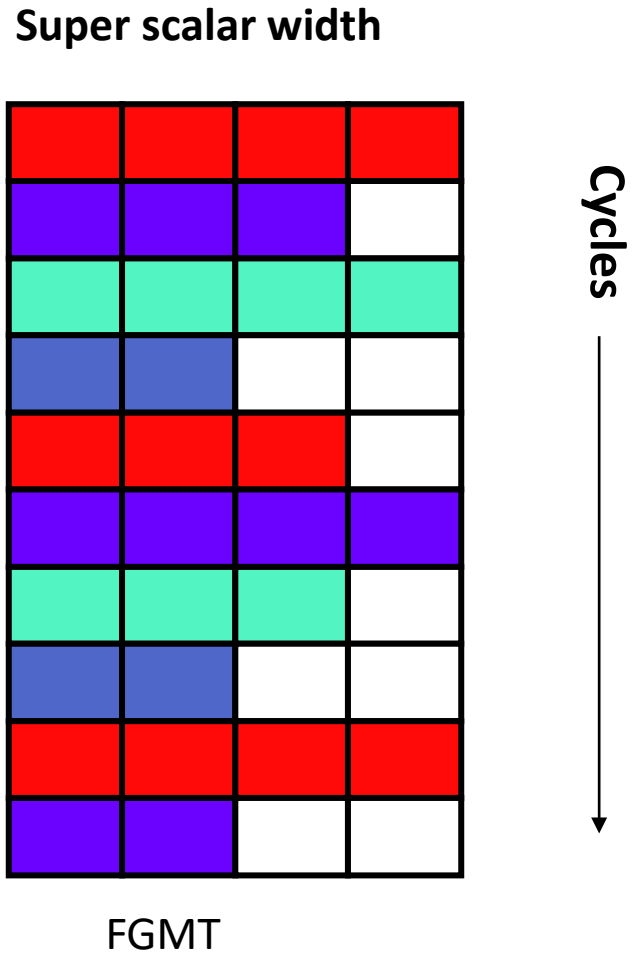
- A possible thread scheduling policy
  - Designate a "preferred" thread (e.g., thread A)
  - Switch to thread B on thread A L2 miss
  - Switch back to A when A L2 miss returns

- Pipeline partitioning
  - None, flush on switch
  - Need short in-order pipeline for good performance
    - High switch cost otherwise

What is the advantage over OS context switch?

Fine -Grained Multithreading

- Every cycle, a different thread fetches and issues instructions
  - Irrespective of whether there is any stall
- (Many) more threads
- **Multiple threads in pipeline at once**



FGMT

- (Many) more threads
- Multiple threads in pipeline at once