Computer Architecture: Multithreading (I)

Prof. Onur Mutlu
Carnegie Mellon University

A Note on This Lecture

- These slides are partly from 18-742 Fall 2012, Parallel Computer Architecture, Lecture 9: Multithreading

- Video of that lecture:
  - http://www.youtube.com/watch?v=k5_yW_iNlvg&list=PL5PHm2jkkXmh4cDkC3s1VBB7-njlgiG5d&index=9

Readings: Multithreading

- Required
  - Spracklen and Abraham, "Chip Multithreading: Opportunities and Challenges," HPCA Industrial Session, 2005.
  - Kalla et al., "IBM Power5 Chip: A Dual-Core Multithreaded Processor," IEEE Micro 2004.
  - Tullsen et al., "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," ISCA 1996.
  - Eyerman and Eeckhout, "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors," HPCA 2007.

- Recommended
  - Hirata et al., "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," ISCA 1992
  - Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
  - Gabor et al., "Fairness and Throughput in Switch on Event Multithreading," MICRO 2006.
  - Agarwal et al., "APRIL: A Processor Architecture for Multiprocessing," ISCA 1990.

Multithreading (Outline)

- Multiple hardware contexts
- Purpose
- Initial incarnations
  - CDC 6600
  - HEP
  - Tera
- Levels of multithreading
  - Fine-grained (cycle-by-cycle)
  - Coarse grained (multitasking)
    - Switch-on-event
  - Simultaneous
- Uses: traditional + creative (now that we have multiple contexts, why do we not do …)

Multithreading: Basics

- Thread
  - Instruction stream with state (registers and memory)
  - Register state is also called "thread context"

- Threads could be part of the same process (program) or from different programs
  - Threads in the same program share the same address space (shared memory model)

- Traditionally, the processor keeps track of the context of a single thread
- Multitasking: When a new thread needs to be executed, old thread's context in hardware written back to memory and new thread's context loaded

Hardware Multithreading

- General idea: Have multiple thread contexts in a single processor
  - When the hardware executes from those hardware contexts determines the granularity of multithreading

- Why?
  - To tolerate latency (initial motivation)
    - Latency of memory operations, dependent instructions, branch resolution
    - By utilizing processing resources more efficiently
  - To improve system throughput
    - By exploiting thread-level parallelism
    - By improving superscalar/OoO processor utilization
  - To reduce context switch penalty

Initial Motivations

- Tolerate latency
  - When one thread encounters a long-latency operation, the processor can execute a useful operation from another thread

- CDC 6600 peripheral processors
  - I/O latency: 10 cycles
  - 10 I/O threads can be active to cover the latency
  - Pipeline with 100ns cycle time, memory with 1000ns latency
  - Idea: Each I/O "processor" executes one instruction every 10 cycles on the same pipeline
  - Thornton, "Design of a Computer: The Control Data 6600," 1970.
  - Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.

Hardware Multithreading

- Benefit

  + Latency tolerance

  + Better hardware utilization (when?)

  + Reduced context switch penalty

- Cost

  - Requires multiple thread contexts to be implemented in hardware (area, power, latency cost)

  - Usually reduced single-thread performance

    - Resource sharing, contention

    - Switching penalty (can be reduced with additional hardware)

Types of Multithreading

- Fine-grained
  - Cycle by cycle
- Coarse-grained
  - Switch on event (e.g., cache miss)
  - Switch on quantum/timeout
- Simultaneous
  - Instructions from multiple threads executed concurrently in the same cycle

Fine-grained Multithreading

- Idea: Switch to another thread every cycle such that no two instructions from the thread are in the pipeline concurrently

- Improves pipeline utilization by taking advantage of multiple threads
- Alternative way of looking at it: Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

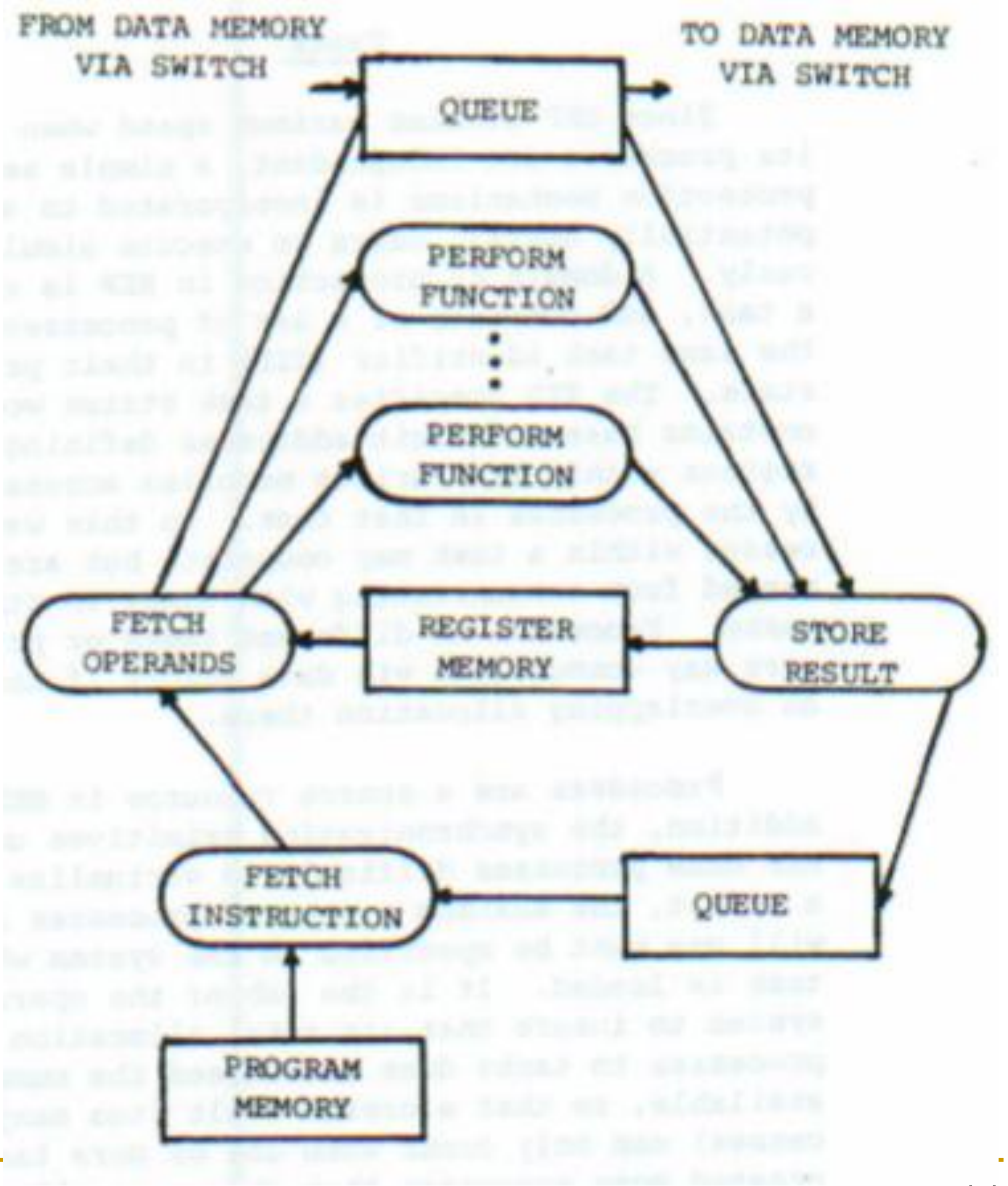
Fine-grained Multithreading

- CDC 6600's peripheral processing unit is fine-grained multithreaded
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles

- Denelcor HEP
  - Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
  - 120 threads/processor
    - 50 user, 70 OS functions
  - available queue vs. unavailable (waiting) queue
  - each thread can only have 1 instruction in the processor pipeline; each thread independent
  - to each thread, processor looks like a sequential machine
  - throughput vs. single thread speed

Fine-grained Multithreading in HEP

- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
  - assuming no memory access



Fine-grained Multithreading

- Advantages

  + No need for dependency checking between instructions
  (only one instruction in pipeline from a single thread)

  + No need for branch prediction logic

  + Otherwise-bubble cycles used for executing useful instructions from different threads

  + Improved system throughput, latency tolerance, utilization

- Disadvantages

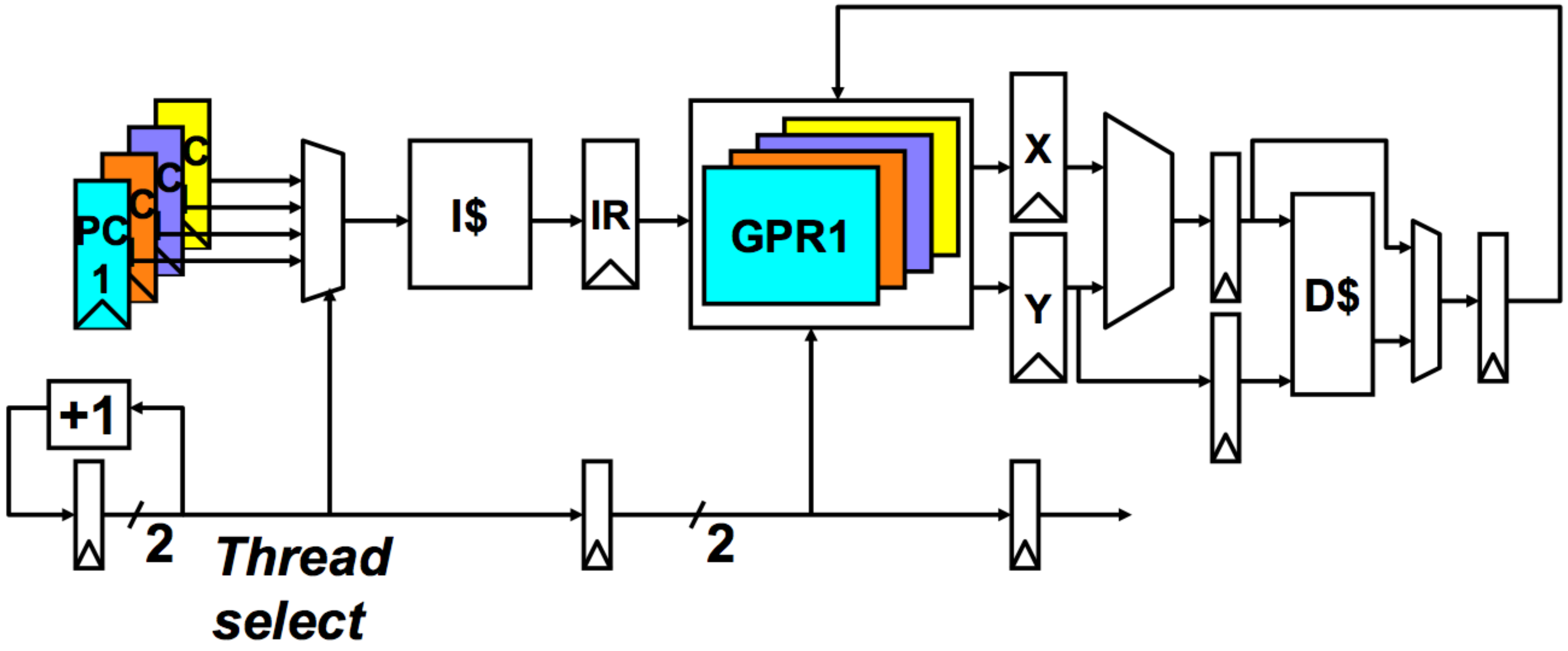
  - Extra hardware complexity: multiple hardware contexts, thread selection logic

  - Reduced single thread performance (one instruction fetched every N cycles)

  - Resource contention between threads in caches and memory

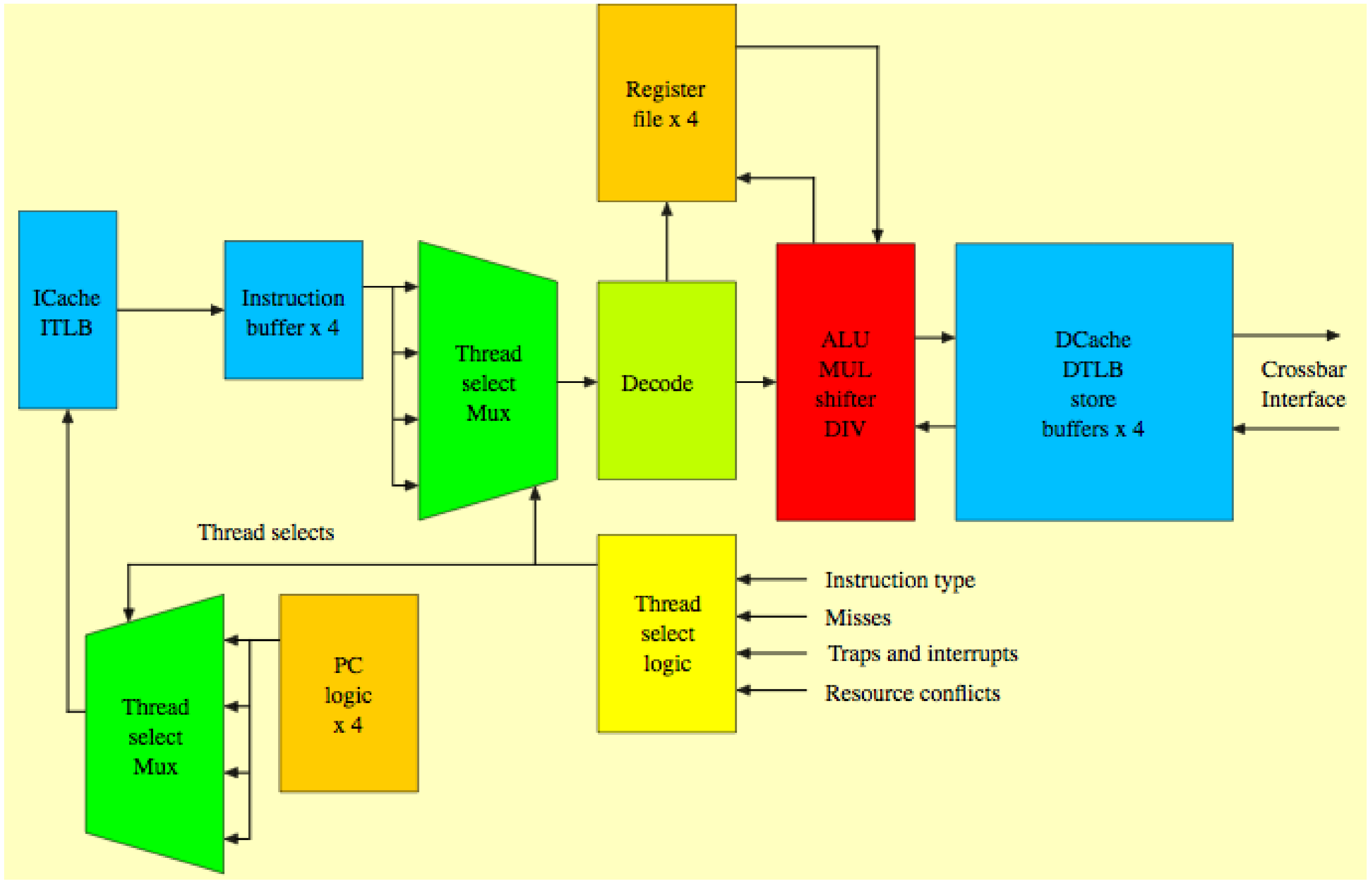
  - Dependency checking logic between threads remains (load/store)

Multithreaded Pipeline Example



- Slide from Joel Emer

Sun Niagara Multithreaded Pipeline



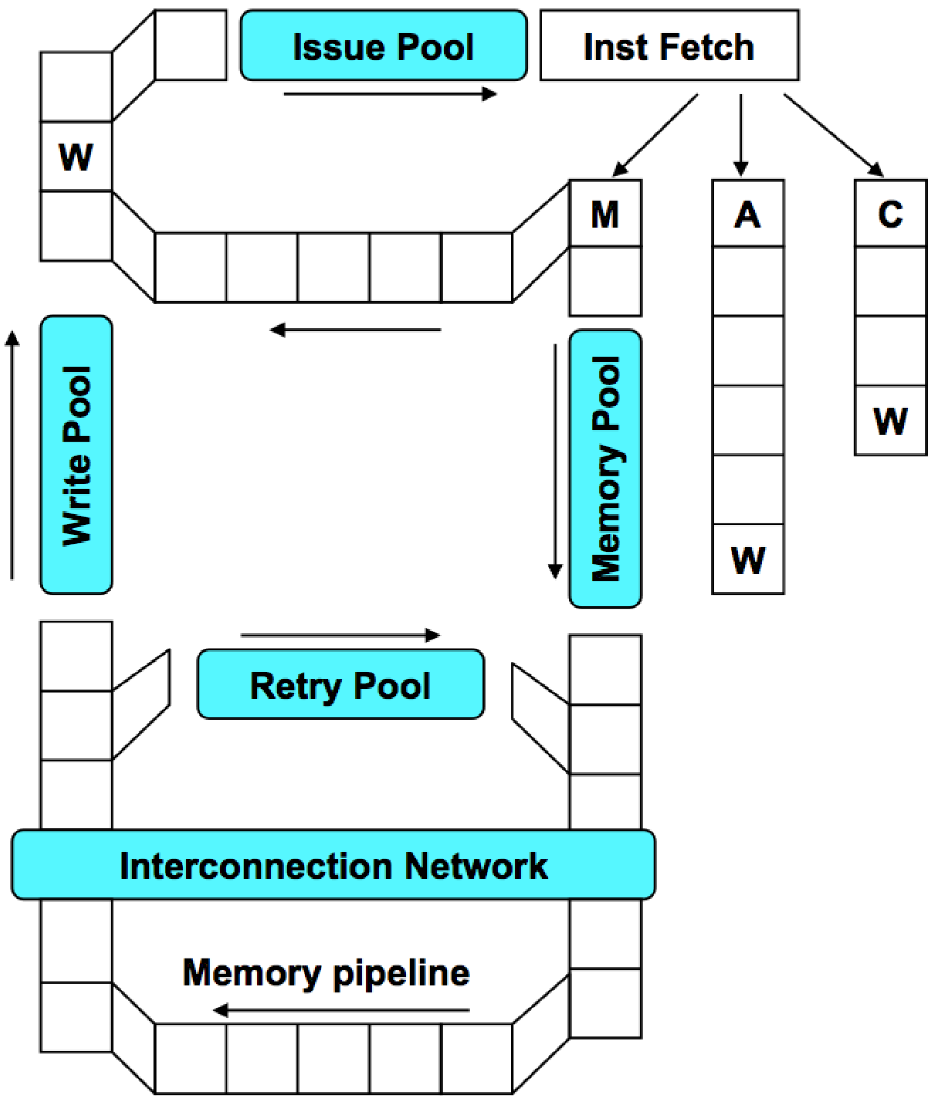
Tera MTA Fine-grained Multithreading

- 256 processors, each with a 21-cycle pipeline
- 128 active threads
- A thread can issue instructions every 21 cycles
  - Then, why 128 threads?

- Memory latency: approximately 150 cycles
  - No data cache
  - Threads can be blocked waiting for memory
  - More threads → better ability to tolerate memory latency

- Thread state per processor
  - 128 x 32 general purpose registers
  - 128 x 1 thread status registers

Tera MTA Pipeline



- Threads move to/from different pools as an instruction executes
  - More accurately, thread IDs are kept in each pool

Coarse-grained Multithreading

- Idea: When a thread is stalled due to some event, switch to a different hardware context
  - Switch-on-event multithreading

- Possible stall events
  - Cache misses
  - Synchronization events (e.g., load an empty location)
  - FP operations

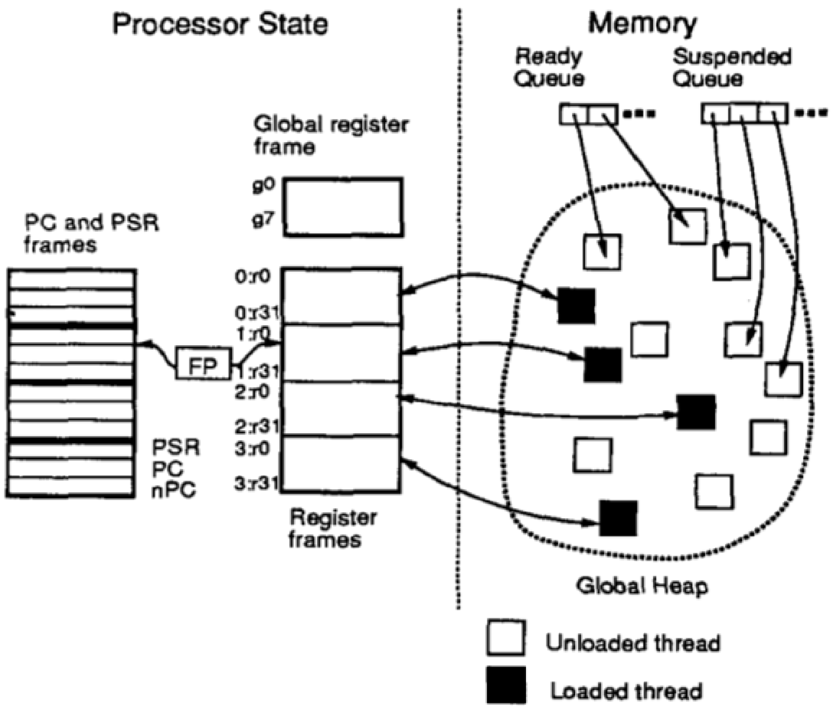
- HEP, Tera combine fine-grained MT and coarse-grained MT
  - Thread waiting for memory becomes blocked (un-selectable)
- Agarwal et al., "APRIL: A Processor Architecture for Multiprocessing," ISCA 1990.
    - Explicit switch on event

Coarse-grained Multithreading in APRIL

- Agarwal et al., "APRIL: A Processor Architecture for Multiprocessing," ISCA 1990.

- 4 hardware thread contexts
  - Called "task frames"

- Thread switch on
  - Cache miss
  - Network access
  - Synchronization fault



- How?
  - Empty processor pipeline, change frame pointer (PC)

Fine-grained vs. Coarse-grained MT

- Fine-grained advantages
  + Simpler to implement, can eliminate dependency checking, branch prediction logic completely
  + Switching need not have any performance overhead (i.e. dead cycles)
    + Coarse-grained requires a pipeline flush or a lot of hardware to save pipeline state
      → Higher performance overhead with deep pipelines and large windows

- Disadvantages
  - Low single thread performance: each thread gets 1/Nth of the bandwidth of the pipeline

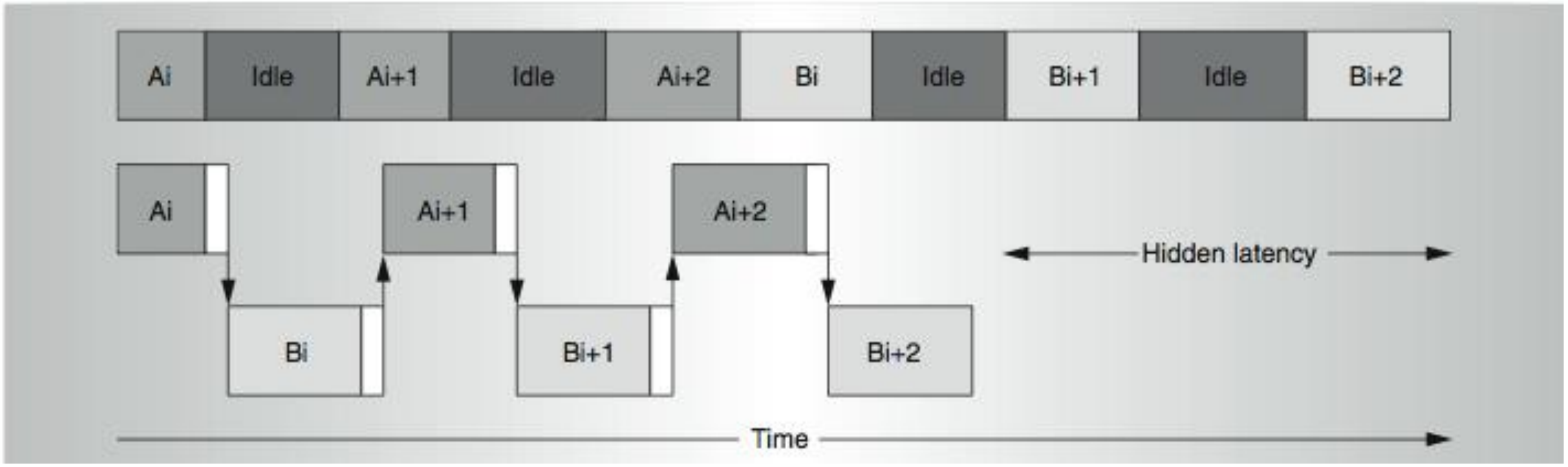
IBM RS64-IV

- 4-way superscalar, in-order, 5-stage pipeline
- Two hardware contexts
- On an L2 cache miss
  - Flush pipeline
  - Switch to the other thread

- Considerations
  - Memory latency vs. thread switch overhead
  - Short pipeline, in-order execution (small instruction window) reduces the overhead of switching

Intel Montecito

- McNairy and Bhatia, "Montecito: A Dual-Core, Dual-Thread Itanium Processor," IEEE Micro 2005.



- Thread switch on
  - L3 cache miss/data return
  - Timeout – for fairness
  - Switch hint instruction
  - ALAT invalidation – synchronization fault
  - Transition to low power mode
- <2% area overhead due to CGMT

Fairness in Coarse-grained Multithreading

- Resource sharing in space and time always causes fairness considerations
  - Fairness: how much progress each thread makes

- In CGMT, the time allocated to each thread affects both fairness and system throughput
  - When do we switch?
  - For how long do we switch?
    - When do we switch back?
  - How does the hardware scheduler interact with the software scheduler for fairness?
  - What is the switching overhead vs. benefit?
    - Where do we store the contexts?

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Fairness in Coarse-grained Multithreading

- Gabor et al., "Fairness and Throughput in Switch on Event Multithreading," MICRO 2006.
- How can you solve the below problem?

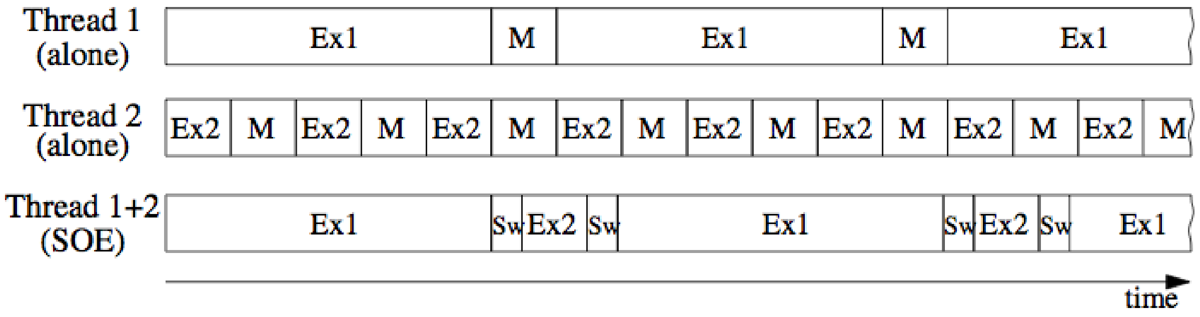


Figure 1. Intuitive example of unfair execution in SOE. *Ex1* marks execution of instructions from thread 1, *Ex2* from thread 2, *M* marks last level cache misses and *Sw* denotes thread switch overheads. When both threads run together using SOE (bottom), the 2nd thread runs extremely slowly while the 1st thread's performance is hardly affected by the multithreading.

Fairness vs. Throughput

- Switch not only on miss, but also on data return

- Problem: Switching has performance overhead
  - Pipeline and window flush
  - Reduced locality and increased resource contention (frequent switches increase resource contention and reduce locality)

- One possible solution
  - Estimate the slowdown of each thread compared to when run alone
  - Enforce switching when slowdowns become significantly unbalanced
  - Gabor et al., "Fairness and Throughput in Switch on Event Multithreading," MICRO 2006.

Thread Switching Urgency in Montecito

- Thread urgency levels
  - 0-7
- Nominal level 5: active progress
- After timeout: set to 7
- After ext. interrupt: set to 6
- Reduce urgency level for each blocking operation
  - L3 miss
- Switch if urgency of foreground lower than that of background

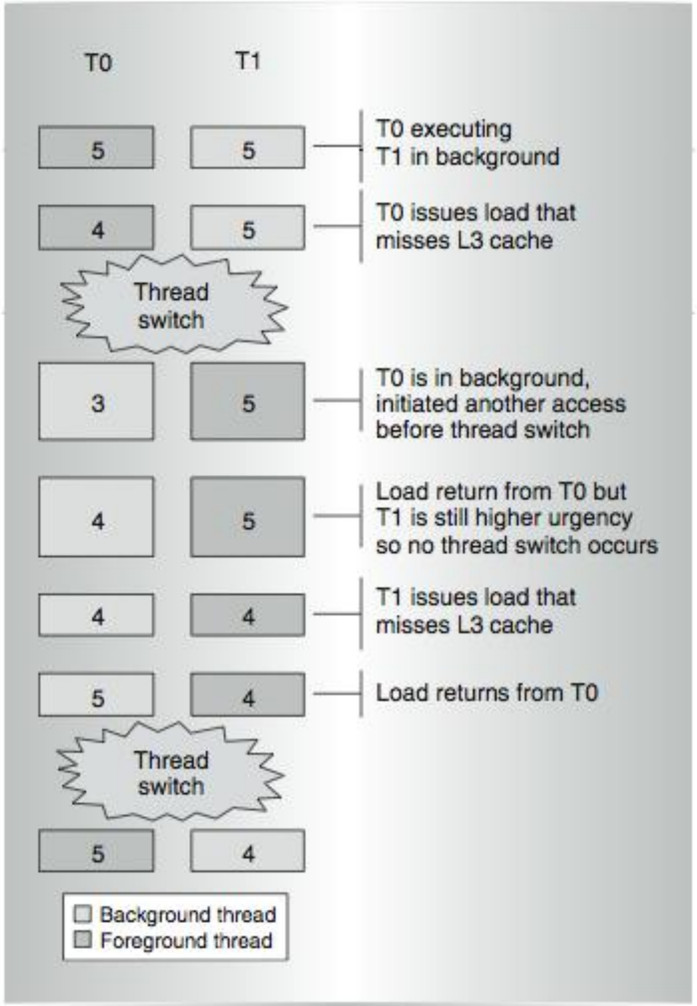


Figure 4. Urgency and thread switches on the Montecito processor.