18-742 Spring 2011
Parallel Computer Architecture
Lecture 15: Multithreading

Prof. Onur Mutlu
Carnegie Mellon University
Announcements

- Project Milestone Meetings – Wednesday, March 2
  - 2:30-4:30pm, HH-B206
  - 20 minutes per group at most
  - Sign up on the 742 website

- Meeting format:
  - 10 minute presentation
  - 10 minute Q&A and feedback
  - Anyone can attend the meeting

- No class Friday (March 4) – Spring Break
Reviews

- Due Yesterday (Feb 27)

- Due March 13
Last Lectures

- **Interconnection Networks**
  - Introduction & Terminology
  - Topology
  - Buffering and Flow control
  - Routing
  - Router design
  - Network performance metrics
  - On-chip vs. off-chip differences
  - Livelock, deadlock, the turn model

- **Research on NoCs and packet scheduling**
  - The problem with packet scheduling
  - Application-aware packet scheduling
  - Aergia: Latency slack based packet scheduling
  - Bufferless routing and livelock issues
Reminder: Some Questions

- What are the possible ways of handling contention in a router?
- What is head-of-line blocking?
- What is a non-minimal routing algorithm?
- What is the difference between deterministic, oblivious, and adaptive routing algorithms?
- What routing algorithms need to worry about deadlock?
- What routing algorithms need to worry about livelock?
- How to handle deadlock?
- How to handle livelock?
- What is zero-load latency?
- What is saturation throughput?
- What is an application-aware packet scheduling algorithm?
Today

- Multithreading, finally...
Multithreading
Readings: Multithreading

Required


Recommended

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 ...)

9
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
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

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
  - 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 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 \(\rightarrow\) better ability to tolerate memory latency

Thread state per processor
- 128 x 32 general purpose registers
- 128 x 1 thread status registers
Threads move to/from different pools as an instruction execute
- 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)

  - Explicit switch on event
Coarse-grained Multithreading in APRIL


- 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 reduces the overhead of switching
Intel Montecito


- 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?
Fairness in Coarse-grained Multithreading


- How can you solve the below problem?

Figure 1. Intuitive example of unfair execution in SOE. $Ex_1$ marks execution of instructions from thread 1, $Ex_2$ 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
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
Simultaneous Multithreading

- Fine-grained and coarse-grained multithreading can start execution of instructions from a single thread at a given cycle

- Execution unit (or pipeline stage) utilization can be low if there are not enough instructions from a thread to “dispatch” in one cycle
  - In a machine with multiple execution units (i.e., superscalar)

- Idea: **Dispatch instructions from multiple threads in the same cycle (to keep multiple execution units utilized)**
Data dependencies reduce functional unit utilization in pipelined processors
Functional unit utilization becomes lower in superscalar, OoO machines. Finding 4 instructions in parallel is not always possible.
Predicated Execution

- Idea: Convert control dependencies into data dependencies
- Improves FU utilization, but some results are thrown away
Chip Multiprocessor

- Idea: Partition functional units across cores
- Still limited FU utilization within a single thread; limited single-thread performance
Fine-grained Multithreading

- Still low utilization due to intra-thread dependencies
- Single thread performance suffers
Simultaneous Multithreading

- Idea: Utilize functional units with independent operations from the same or different threads

Time

- [Diagram of time with different colored squares illustrating the concept of simultaneous multithreading]
Horizontal vs. Vertical Waste

- Why is there horizontal and vertical waste?
- How do you reduce each?
Simultaneous Multithreading

- Reduces both horizontal and vertical waste
- Required hardware
  - Multiple contexts and ability to dispatch from multiple threads simultaneously

- Superscalar, OoO processors already have this machinery
  - Dynamic instruction scheduler searches the scheduling window to wake up and select ready instructions
    - As long as dependencies are correctly tracked (via renaming and memory disambiguation), scheduler can be thread-agnostic
Basic Superscalar OoO Pipeline
SMT Pipeline

- Register file needs to become larger. Why?
Changes to Pipeline for SMT

- Replicated resources
  - Program counter
  - Register map
  - Return address stack
  - Global history register

- Shared resources
  - Register file (size increased)
  - Instruction queue
  - First and second level caches
  - Translation lookaside buffers
  - Branch predictor