15-740/18-740
Computer Architecture
Lecture 14: Runahead Execution

Prof. Onur Mutlu
Carnegie Mellon University
Fall 2011, 10/12/2011
Reviews

- Due Today
Announcements

- **Milestone I**
  - Due this Friday (Oct 14)
  - Format: 2-pages
  - Include results from your initial evaluations. We need to see good progress.
  - Of course, the results need to make sense (i.e., you should be able to explain them)

- **Midterm I**
  - Postponed to October 24

- **Milestone II**
  - Will be postponed. Stay tuned.
Course Checkpoint

- **Homeworks**
  - Look at solutions
  - These are not for grades. These are for you to test your understanding of the concepts and prepare for exams.
  - Provide succinct and clear answers.

- **Review sets**
  - Concise reviews

- **Projects**
  - Most important part of the course
  - Focus on this
  - If you are struggling, talk with the TAs
Course Feedback

- Fill out the forms and return
Last Lecture

- Tag broadcast, wakeup+select loop
- Pentium Pro vs. Pentium 4 designs: buffer decoupling
- Consolidated physical register files in Pentium 4 and Alpha 21264
- Centralized vs. distributed reservation stations
- Which instruction to select?
Today

- Load related instruction scheduling
- Runahead execution
Review: Centralized vs. Distributed Reservation Stations

- **Centralized (monolithic):**
  + Reservation stations not statically partitioned (can adapt to changes in instruction mix, e.g., 100% adds)
  -- All entries need to have all fields even though some fields might not be needed for some instructions (e.g. branches, stores, etc)
  -- Number of ports = issue width
  -- More complex control and routing logic

- **Distributed:**
  + Each RS can be specialized to the functional unit it serves (more area efficient)
  + Number of ports can be 1 (RS per functional unit)
  + Simpler control and routing logic
  -- Other RS’s cannot be used if one functional unit’s RS is full (static partitioning)
Review: Issues in Scheduling Logic (I)

- What if multiple instructions become ready in the same cycle?
  - Why does this happen?
    - An instruction with multiple dependents
    - Multiple instructions can complete in the same cycle
  - Which one to schedule?
    - Oldest
    - Random
    - Most dependents first
    - Most critical first (longest latency dependency chain)

- Does not matter for performance unless the active window is very large
Issues in Scheduling Logic (II)

- When to schedule the dependents of a multi-cycle execute instruction?
  - One cycle before the completion of the instruction
  - Example: IMUL, Pentium 4, 3-cycle ADD

- When to schedule the dependents of a variable-latency execute instruction?
  - A load can hit or miss in the data cache
  - Option 1: Schedule dependents assuming load will hit
  - Option 2: Schedule dependents assuming load will miss

- When do we take out an instruction from a reservation station?
Scheduling of Load Dependents

- Assume load will hit
  + No delay for dependents (load hit is the common case)
  -- Need to squash and re-schedule if load actually misses

- Assume load will miss (i.e. schedule when load data ready)
  + No need to re-schedule (simpler logic)
  -- Significant delay for load dependents if load hits

- Predict load hit/miss
  + No delay for dependents on accurate prediction
  -- Need to predict and re-schedule on misprediction

What to Do with Dependents on a Load Miss? (I)

- A load miss can take hundreds of cycles
- If there are many dependent instructions on a load miss, these can clog the scheduling window
- Independent instructions cannot be allocated reservation stations and scheduling can stall
- How to avoid this?

**Idea:** Move miss-dependent instructions into a separate buffer

- Example: Pentium 4’s “scheduling loops”
But, dependents still hold on to the physical registers

Cannot scale the size of the register file indefinitely since it is on the critical path

Possible solution: Deallocate physical registers of dependents

Questions

- Why is OoO execution beneficial?
  - What if all operations take single cycle?
  - Latency tolerance: OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently

- What if an instruction takes 500 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
  - What limits the latency tolerance scalability of Tomasulo’s algorithm?
    - Active/instruction window size: determined by register file, scheduling window, reorder buffer, store buffer, load buffer
Small Windows: Full-window Stalls

- When a long-latency instruction is not complete, it blocks retirement.

- Incoming instructions fill the instruction window.

- Once the window is full, processor cannot place new instructions into the window.
  - This is called a full-window stall.

- A full-window stall prevents the processor from making progress in the execution of the program.
Small Windows: Full-window Stalls

8-entry instruction window:

- LOAD R1 ← mem[R5]
- BEQ R1, R0, target
- ADD R2 ← R2, 8
- LOAD R3 ← mem[R2]
- MUL R4 ← R4, R3
- ADD R4 ← R4, R5
- STOR mem[R2] ← R4
- ADD R2 ← R2, 64

L2 Miss! Takes 100s of cycles.

- Independent of the L2 miss, executed out of program order, but cannot be retired.

- Younger instructions cannot be executed because there is no space in the instruction window.

- The processor stalls until the L2 Miss is serviced.

- L2 cache misses are responsible for most full-window stalls.
Impact of L2 Cache Misses

512KB L2 cache, 500-cycle DRAM latency, aggressive stream-based prefetcher
Data averaged over 147 memory-intensive benchmarks on a high-end x86 processor model
Impact of L2 Cache Misses

500-cycle DRAM latency, aggressive stream-based prefetcher
Data averaged over 147 memory-intensive benchmarks on a high-end x86 processor model
The Problem

- Out-of-order execution requires large instruction windows to tolerate today’s main memory latencies.

- As main memory latency increases, instruction window size should also increase to fully tolerate the memory latency.

- Building a large instruction window is a challenging task if we would like to achieve
  - Low power/energy consumption (tag matching logic, ld/st buffers)
  - Short cycle time (access, wakeup/select latencies)
  - Low design and verification complexity
Efficient Scaling of Instruction Window Size

- One of the major research issues in out of order execution
- How to achieve the benefits of a large window with a small one (or in a simpler way)?

  - **Runahead execution**?
    - Upon L2 miss, checkpoint architectural state, speculatively execute only for prefetching, re-execute when data ready

  - **Continual flow pipelines**?
    - Upon L2 miss, deallocate everything belonging to an L2 miss dependent, reallocate/re-rename and re-execute upon data ready

  - **Dual-core execution**?
    - One core runs ahead and does not stall on L2 misses, feeds another core that commits instructions
Runahead Execution (I)

- A technique to obtain the memory-level parallelism benefits of a large instruction window

- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode

- In runahead mode:
  - Speculatively pre-execute instructions
  - The purpose of pre-execution is to generate prefetches
  - L2-miss dependent instructions are marked INV and dropped

- Runahead mode ends when the original miss returns
  - Checkpoint is restored and normal execution resumes

Runahead Example

*Perfect Caches:*

Load 1 Hit  Load 2 Hit

Compute   Compute

*Small Window:*

Load 1 Miss  Load 2 Miss

Compute   Stall   Compute   Stall

Miss 1  Miss 2

*Runahead:*

Load 1 Miss  Load 2 Miss  Load 1 Hit  Load 2 Hit

Compute   Runahead   Compute

Miss 1  Miss 2

Saved Cycles
Benefits of Runahead Execution

Instead of stalling during an L2 cache miss:

- Pre-executed loads and stores independent of L2-miss instructions generate very accurate data prefetches:
  - For both regular and irregular access patterns

- Instructions on the predicted program path are prefetched into the instruction/trace cache and L2.

- Hardware prefetcher and branch predictor tables are trained using future access information.
Runahead Execution Mechanism

- Entry into runahead mode
  - Checkpoint architectural register state

- Instruction processing in runahead mode

- Exit from runahead mode
  - Restore architectural register state from checkpoint
Instruction Processing in Runahead Mode

Runahead mode processing is the same as normal instruction processing, EXCEPT:

- It is purely speculative: *Architectural (software-visible)* register/memory state is NOT updated in runahead mode.

- L2-miss dependent instructions are identified and treated specially.
  - They are quickly removed from the instruction window.
  - Their results are not trusted.
L2-Miss Dependent Instructions

- Two types of results produced: INV and VALID

- INV = Dependent on an L2 miss

- INV results are marked using INV bits in the register file and store buffer.

- INV values are not used for prefetching/branch resolution.
Removal of Instructions from Window

- Load 1 Miss

- Oldest instruction is examined for **pseudo-retirement**
  - An INV instruction is removed from window immediately.
  - A VALID instruction is removed when it completes execution.

- Pseudo-retired instructions free their allocated resources.
  - This allows the processing of later instructions.

- Pseudo-retired stores communicate their data to dependent loads.
A pseudo-retired store writes its data and INV status to a dedicated memory, called *runahead cache*.

Purpose: Data communication through memory in runahead mode.

A dependent load reads its data from the runahead cache.

Does not need to be always correct → Size of runahead cache is very small.
Branch Handling in Runahead Mode

- **INV branches cannot be resolved.**
  - A mispredicted INV branch causes the processor to stay on the wrong program path until the end of runahead execution.

- **VALID branches are resolved and initiate recovery if mispredicted.**
Runahead Execution (III)

**Advantages:**
- Very accurate prefetches for data/instructions (all cache levels)
- Follows the program path
- Simple to implement, most of the hardware is already built in
- Uses the same thread context as main thread, no waste of context
- No need to construct a pre-execution thread

**Disadvantages/Limitations:**
- Extra executed instructions
- Limited by branch prediction accuracy
- Cannot prefetch dependent cache misses. Solution?
- Effectiveness limited by available “memory-level parallelism” (MLP)
- Prefetch distance limited by memory latency

**Implemented in IBM POWER6, Sun “Rock”**