CSCI-564 Advanced Computer Architecture

Lecture 9: Handling Branches 1

Bo Wu
Colorado School of Mines
We talked about two methods

- Predict next instruction (PC+4) will be executed
  - If prediction is wrong, flush the pipeline

- Schedule the instructions to fill up the branch delay slots
  - Need ISA support
Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlaps the execution of other threads’ instructions

+ No logic needed for handling control and data dependences within a thread
-- Single thread performance suffers
-- Extra logic for keeping thread contexts
-- Does not overlap latency if not enough threads to cover the whole pipeline
Fine-grained Multithreading: History

• 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 (Heterogeneous Element Processor)
  – 120 threads/processor
  – available queue vs. unavailable (waiting) queue for threads
  – each thread can have only 1 instruction in the processor pipeline; each thread independent
  – to each thread, processor looks like a non-pipelined machine
  – system throughput vs. single thread performance tradeoff


Multithreaded Pipeline Example

Slide credit: Joel Emer
Sun Niagara Multithreaded Pipeline

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
  - Some dependency checking logic between threads remains (load/store)
Branch Prediction: Guess the Next Instruction to Fetch

PC 0x0000

12 cycles

8 cycles

Branch prediction

LD R1, MEM[R0]
ADD R2, R2, #1
BRZERO 0x0001
ADD R3, R2, #1
MUL R1, R2, R3
LD R2, MEM[R2]
LD R0, MEM[R2]
Misprediction Penalty

LD R0, MEM[R2]
LD R2, MEM[R2]
BR
ZERO 0x0001
LD R1, MEM[R0]
ADD R2, R2, #1
ADD R3, R2, #1
MUL R1, R2, R3
LD R2, MEM[R2]
LD R0, MEM[R2]
Misprediction Penalty
I-$ RF
Flush!! PC
DEC
WB
D-$
Branch Prediction

• Processors are pipelined to increase concurrency
• How do we keep the pipeline full in the presence of branches?
  – Guess the next instruction when a branch is fetched
  – Requires guessing the direction and target of a branch
Branch Prediction: Always PC+4

When a branch resolves
- branch target (Inst_k) is fetched
- all instructions fetched since inst_h (so called “wrong-path” instructions) must be flushed
Pipeline Flush on a Misprediction

$t_0$  $t_1$  $t_2$  $t_3$  $t_4$  $t_5$

Inst$_h$  IF$_{PC}$  ID  ALU  MEM  WB

Inst$_i$  IF$_{PC+4}$  ID  ID  killed

Inst$_j$  IF$_{PC+8}$  ID  killed

Inst$_k$  IF$_{target}$  ID  ALU  WB

Inst$_l$  IF  ID  ALU  WB

Inst$_h$ is a branch
Performance Analysis

- correct guess $\Rightarrow$ no penalty
- incorrect guess $\Rightarrow$ 2 bubbles
- Assume
  - no data hazards
  - 20% control flow instructions
  - 70% of control flow instructions are taken
- $\text{CPI} = [1 + (0.20 \times 0.7) \times 2] = [1 + 0.14 \times 2] = 1.28$

Can we reduce either of the two penalty terms?
Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early

CPI = \[ 1 + (0.2 \times 0.7) \times 1 \] = 1.14
Branch Prediction (Enhanced)

• Idea: Predict the next fetch address (to be used in the next cycle)

• Requires three things to be predicted at fetch stage:
  – Whether the fetched instruction is a branch
  – (Conditional) branch direction
  – Branch target address (if taken)

• Observation: Target address remains the same for a conditional direct branch across dynamic instances
  – Idea: Store the target address from previous instance and access it with the PC
  – Called Branch Target Buffer (BTB) or Branch Target Address Cache
Always taken CPI = \[ 1 + (0.20 \times 0.3) \times 2 \] = 1.12 (70% of branches taken)
More Sophisticated Branch Direction Prediction

- Global branch history
- Program Counter
- Which direction earlier branches went
- Address of the current branch
- Direction predictor (2-bit counters)
- XOR
- PC + inst size
- hit?
- taken?
- target address
- Cache of Target Addresses (BTB: Branch Target Buffer)
- Next Fetch Address
Simple Branch Direction Prediction Schemes

• Compile time (static)
  – Always not taken
  – Always taken
  – BTFN (Backward taken, forward not taken)
  – Profile based (likely direction)

• Run time (dynamic)
  – Last time prediction (single-bit)
More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid
Static Branch Prediction (I)

- **Always not-taken**
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the “not-taken” path

- **Always taken**
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken

- **Backward taken, forward not taken (BTFN)**
  - Predict backward (loop) branches as taken, others not-taken
Static Branch Prediction (II)

- **Profile-based**
  - Idea: Compiler determines likely direction for each branch using profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous slide) \(\rightarrow\) accurate if profile is representative!

-- Requires hint bits in the branch instruction format

-- Accuracy depends on dynamic branch behavior:
- TTTTTTTTTTNNNNNNNNNN \(\rightarrow\) 50% accuracy
- TNTNTNTNTNTNTNTNTNTN \(\rightarrow\) 50% accuracy

-- Accuracy depends on the representativeness of profile input set
Static Branch Prediction (III)

- Program-based (or, program analysis based)
  - Idea: Use heuristics based on program analysis to determine statically-predicted direction
  - Example opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Example loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)
  - Pointer and FP comparisons: Predict not equal

+ Does not require profiling

-- Heuristics might be not representative or good
-- Requires compiler analysis and ISA support (ditto for other static methods)

  - 20% misprediction rate
Static Branch Prediction (III)

- **Programmer-based**
  - Idea: *Programmer provides the statically-predicted direction*
  - Via *pragmas* in the programming language that qualify a branch as likely-taken versus likely-not-taken

+ Does not require profiling or program analysis
+ Programmer may know some branches and their program better than other analysis techniques
  -- Requires programming language, compiler, ISA support
  -- Burdens the programmer?
Aside: Pragmas

- Idea: **Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy**

- if (likely(x)) { ... }
- if (unlikely(error)) { ... }

- Many other hints and optimizations can be enabled with pragmas
  - E.g., whether a loop can be parallelized
  - #pragma omp parallel
  - **Description**
    - The omp parallel directive explicitly instructs the compiler to parallelize the chosen segment of code.
Static Branch Prediction

• All previous techniques can be combined
  – Profile based
  – Program based
  – Programmer based

• What are common disadvantages of all three techniques?
  – Cannot adapt to dynamic changes in branch behavior
    • This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its overheads...)
Dynamic Branch Prediction

• Idea: Predict branches based on dynamic information (collected at run-time)

• Advantages
  + Prediction based on history of the execution of branches
  + It can adapt to dynamic changes in branch behavior
  + No need for static profiling: input set representativeness problem goes away

• Disadvantages
  -- More complex (requires additional hardware)
Last Time Predictor

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    TTTTTTTTTTNNNNNNNNNN $\rightarrow$ 90% accuracy

- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with $N$ iterations $= (N-2)/N$

+ Loop branches for loops with large number of iterations
-- Loop branches for loops with small number of iterations
  TNTNTNTNTNTNTNTNTNTNTNTNTNTN $\rightarrow$ 0% accuracy

Last-time predictor CPI $= \left[ 1 + (0.20 \times 0.15) \times 2 \right] = 1.06$ (Assuming 85% accuracy)
Implementing the Last-Time Predictor

The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch.
State Machine for Last-Time Prediction

- **predict not taken**
  - actually not taken
  - actually taken

- **predict taken**
  - actually taken
  - actually not taken
Improving the Last Time Predictor

• Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  – even though the branch may be mostly taken or mostly not taken

• Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  – Use two bits to track the history of predictions for a branch instead of a single bit
  – Can have 2 states for T or NT instead of 1 state for each