Readings for Next Few Lectures (I)

- P&H Chapter 4.9-4.11

  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts


Readings for Next Few Lectures (II)

Review: 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
Review: Importance of The Branch Problem

- Assume a 5-wide *superscalar* pipeline with 20-cycle branch resolution latency

- How long does it take to fetch 500 instructions?
  - Assume 1 out of 5 instructions is a branch
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy
    - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
    - 40% extra instructions fetched
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched
Can We Do Better?

- Last-time and 2BC predictors exploit “last-time” predictability

- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  - Global branch correlation

- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
  - Local branch correlation
Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch

  
  \[
  \text{if (cond1)} \\
  \text{...} \\
  \text{if (cond1 AND cond2)}
  \]

- If first branch not taken, second also not taken

  \[
  \text{branch Y: if (cond1) a = 2;} \\
  \text{...} \\
  \text{branch X: if (a == 0)}
  \]

- If first branch taken, second definitely not taken
Global Branch Correlation (II)

- branch Y: if (cond1)
- ... 
- branch Z: if (cond2)
- ... 
- branch X: if (cond1 AND cond2)

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken
Eqntott, SPEC 1992

if (aa==2) ;; B1
    aa=0;
if (bb==2) ;; B2
    bb=0;
if (aa!=bb) {
    ;; B3
    ....
}

If B1 is not taken (i.e., aa==0@B3) and B2 is not taken (i.e. bb=0@B3) then B3 is certainly taken
Capturing Global Branch Correlation

- Idea: Associate branch outcomes with “global T/NT history” of all branches
- Make a prediction based on the outcome of the branch the last time the same global branch history was encountered

- Implementation:
  - Keep track of the “global T/NT history” of all branches in a register → Global History Register (GHR)
  - Use GHR to index into a table that recorded the outcome that was seen for each GHR value in the recent past → Pattern History Table (table of 2-bit counters)

- Global history/branch predictor
- Uses two levels of history (GHR + history at that GHR)
Two Level Global Branch Prediction

- First level: Global branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

Pattern History Table (PHT)

How Does the Global Predictor Work?

Intel Pentium Pro Branch Predictor

- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
  - Which pattern history table to use is determined by lower order bits of the branch address
Improving Global Predictor Accuracy

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
  - **Gshare predictor**: GHR hashed with the Branch PC
    - More context information
    - Better utilization of PHT
  -- Increases access latency

One-Level Branch Predictor

Direction predictor (2-bit counters)

Program Counter

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address

PC + inst size

hit?

taken?
Two-Level Global History Branch Predictor

- Global branch history
- Program Counter
- Address of the current instruction

Which direction earlier branches went

Direction predictor (2-bit counters)

- Taken?
- PC + inst size
- Hit?
- Target address

Cache of Target Addresses (BTB: Branch Target Buffer)
Two-Level Gshare Branch Predictor

Which direction earlier branches went

Global branch history

Program Counter

Address of the current instruction

Direction predictor (2-bit counters)

XOR

taken?

PC + inst size

hit?

target address

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
Can We Do Better?

- Last-time and 2BC predictors exploit “last-time” predictability

- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  - Global branch correlation

- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
  - Local branch correlation
Local Branch Correlation

for (i=1; i<=4; i++) { }

If the loop test is done at the end of the body, the corresponding branch will execute the pattern \((1110)^n\), where 1 and 0 represent taken and not taken respectively, and \(n\) is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

Capturing Local Branch Correlation

- **Idea:** Have a per-branch history register
  - Associate the predicted outcome of a branch with “T/NT history” of the same branch

- Make a prediction based on the outcome of the branch the last time the same local branch history was encountered

- Called the local history/branch predictor

- Uses two levels of history (Per-branch history register + history at that history register value)
Two Level Local Branch Prediction

- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

Two-Level Local History Branch Predictor

Which directions earlier instances of *this branch* went

Direction predictor (2-bit counters)

Program Counter

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address

PC + inst size

taken?

hit?

target address
Hybrid Branch Predictors

- Idea: Use more than one type of predictor (i.e., multiple algorithms) and select the “best” prediction
  - E.g., hybrid of 2-bit counters and global predictor

- Advantages:
  + Better accuracy: different predictors are better for different branches
  + Reduced warmup time (faster-warmup predictor used until the slower-warmup predictor warms up)

- Disadvantages:
  -- Need “meta-predictor” or “selector”
  -- Longer access latency

Alpha 21264 Tournament Predictor

- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch

Branch Prediction Accuracy (Example)

- Bimodal: table of 2bc indexed by branch address

Figure 13: Combined Predictor Performance by Benchmark
Biased Branches

- Observation: Many branches are biased in one direction (e.g., 99% taken)

- Problem: These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing “interference” in branch prediction tables and history registers

- Solution: Detect such biased branches, and predict them with a simpler predictor

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
  - **Stall** the pipeline until we know the next fetch address
  - Guess the next fetch address (**branch prediction**)
  - Employ delayed branching (**branch delay slot**)
  - Do something else (**fine-grained multithreading**)
  - Eliminate control-flow instructions (**predicated execution**)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (**multipath execution**)
Review: Predicate Combining (not Predicated Execution)

- Complex predicates are converted into multiple branches
  - if ((a == b) && (c < d) && (a > 5000)) { ... }
    - 3 conditional branches

- Problem: This increases the number of control dependencies

- Idea: Combine predicate operations to feed a single branch instruction
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate

+ Fewer branches in code → fewer mipredictions/stalls

-- Possibly unnecessary work
  -- If the first predicate is false, no need to compute other predicates

- Condition registers exist in IBM RS6000 and the POWER architecture
**Predication (Predicated Execution)**

- **Idea:** Compiler converts control dependence into data dependence → branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (others turned into NOPs)

(normal branch code)  (predicated code)

```c
if (cond) {
    b = 0;
} else {
    b = 1;
}
```

```
p1 = (cond)
branch p1, TARGET

mov b, 1
jmp JOIN

TARGET:
    mov b, 0

add x, b, 1
```

```
p1 = (cond)
(!p1) mov b, 1
(p1) mov b, 0

add x, b, 1
```
Conditional Move Operations

- Very limited form of predicated execution

- CMOV R1 ← R2
  - R1 = (ConditionCode == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)
Review: CMOV Operation

- Suppose we had a Conditional Move instruction...
  - CMOV condition, R1 ← R2
  - R1 = (condition == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

- Code example with branches vs. CMOVs
  if (a == 5) {b = 4;} else {b = 3;}

  CMPEQ condition, a, 5;
  CMOV condition, b ← 4;
  CMOV !condition, b ← 3;
Predicated execution can be high performance and energy-efficient

**Predicated Execution**
Fetch  Decode  Rename  Schedule  RegisterRead  Execute

**Branch Prediction**
Fetch  Decode  Rename  Schedule  RegisterRead  Execute

Pipeline flush!!
Predicated Execution (III)

**Advantages:**
- Eliminates mispredictions for hard-to-predict branches
  - No need for branch prediction for some branches
  - Good if misprediction cost > useless work due to predication
- Enables code optimizations hindered by the control dependency
  - Can move instructions more freely within predicated code

**Disadvantages:**
- Causes useless work for branches that are easy to predict
  - Reduces performance if misprediction cost < useless work
  - Adaptivity: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, program phase, control-flow path.
- Additional hardware and ISA support
- Cannot eliminate all hard to predict branches
  - Loop branches?
Predicated Execution in Intel Itanium

- Each instruction can be separately predicated
- 64 one-bit predicate registers
- Each instruction carries a 6-bit predicate field
- An instruction is effectively a NOP if its predicate is false
Conditional Execution in the ARM ISA

- Almost all ARM instructions can include an optional condition code.

- An instruction with a condition code is executed only if the condition code flags in the CPSR meet the specified condition.
### Conditional Execution in ARM ISA

<table>
<thead>
<tr>
<th>Cond</th>
<th>Opcode</th>
<th>S</th>
<th>Rs</th>
<th>Rd</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
<td>0 0 0 0 0 0</td>
<td>A</td>
<td>S</td>
<td>Rd</td>
<td>Rn</td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>U</td>
<td>A</td>
<td>S</td>
<td>RdHi</td>
<td>RdLo</td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>0 1 1</td>
<td>P</td>
<td>U</td>
<td>B</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>1 0 0</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>0 0 0</td>
<td>P</td>
<td>U</td>
<td>1</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>0 0 0</td>
<td>P</td>
<td>U</td>
<td>0</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>1 0 1</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1 0</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>P</td>
<td>U</td>
<td>N</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>Op1</td>
<td>CRn</td>
<td>CRd</td>
<td>CPNum</td>
<td>Op2</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>Op1</td>
<td>L</td>
<td>CRn</td>
<td>Rd</td>
<td>CPNum</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>SWI</td>
<td>Number</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conditional Execution in ARM ISA

- **0000 = EQ - Z set (equal)**
- **0001 = NE - Z clear (not equal)**
- **0010 = HS / CS - C set (unsigned higher or same)**
- **0011 = LO / CC - C clear (unsigned lower)**
- **0100 = MI - N set (negative)**
- **0101 = PL - N clear (positive or zero)**
- **0110 = VS - V set (overflow)**
- **0111 = VC - V clear (no overflow)**
- **1000 = HI - C set and Z clear (unsigned higher)**
- **1001 = LS - C clear or Z (set unsigned lower or same)**
- **1010 = GE - N set and V set, or N clear and V clear (≥ or =)**
- **1011 = LT - N set and V clear, or N clear and V set (>)**
- **1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)**
- **1101 = LE - Z set, or N set and V clear, or N clear and V set (≤ or =)**
- **1110 = AL - always**
- **1111 = NV - reserved.**
* To execute an instruction conditionally, simply postfix it with the appropriate condition:
  
  - For example an add instruction takes the form:
    
    ```
    ADD r0,r1,r2 ; r0 = r1 + r2 (ADDLAL)
    ```
  
  - To execute this only if the zero flag is set:
    
    ```
    ADDEQ r0,r1,r2 ; If zero flag set then...
                 ; ... r0 = r1 + r2
    ```

* By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.

  - For example to add two numbers and set the condition flags:
    
    ```
    ADDS r0,r1,r2 ; r0 = r1 + r2
                ; ... and set flags
    ```
Conditional Execution in ARM ISA

* Convert the GCD algorithm given in this flowchart into
  1) “Normal” assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.

* The only instructions you need are CMP, B and SUB.
Conditional Execution in ARM ISA

"Normal" Assembler

```
gcd    cmp r0, r1    ;reached the end?
    beq stop
    blt less    ;if r0 > r1
    sub r0, r0, r1 ;subtract r1 from r0
    bal gcd

less  sub r1, r1, r0 ;subtract r0 from r1
    bal gcd

stop
```

ARM Conditional Assembler

```
gcd    cmp    r0, r1    ;if r0 > r1
    subgt r0, r0, r1 ;subtract r1 from r0
    sublt r1, r1, r0 ;else subtract r0 from r1
    bne    gcd       ;reached the end?
```
Idealism

- Wouldn’t it be nice
  - If the branch is eliminated (predicated) when it would actually be mispredicted
  - If the branch were predicted when it would actually be correctly predicted

- Wouldn’t it be nice
  - If predication did not require ISA support
Improving Predicated Execution

- Three major limitations of predication
  1. **Adaptivity**: non-adaptive to branch behavior
  2. **Complex CFG**: inapplicable to loops/complex control flow graphs
  3. **ISA**: Requires large ISA changes

- **Wish Branches** [Kim+, MICRO 2005]
  - Solve 1 and partially 2 (for loops)

- **Dynamic Predicated Execution**
  - **Diverge-Merge Processor** [Kim+, MICRO 2006]
    - Solves 1, 2 (partially), 3
Wish Branches

- The **compiler** generates code (with wish branches) that can be executed **either** as predicated code **or** non-predicated code (normal branch code)
- The **hardware** **decides** to execute predicated code or normal branch code at run-time based on the confidence of branch prediction
- **Easy to predict:** normal branch code
- **Hard to predict:** predicated code
Wish Jump/Join

- **Normal branch code**
  - p1 = (cond) branch p1, TARGET
  - mov b, 1
  - jmp JOIN
  - TARGET: mov b, 0

- **Predicated code**
  - p1 = (cond)
  - (!p1) mov b, 1
  - (p1) mov b, 0

- **High Confidence**
  - wish jump
  - (p1) mov b, 1
  - wish jump p1 JOIN

- **Low Confidence**
  - join !p1
  - mov b, 1
  - wish. join
  - JOIN (1)
  - mov b, 0
  - jmp JOIN

- **Wish jump/join code**
Wish Branches vs. Predicated Execution

- Advantages compared to predicated execution
  - Reduces the overhead of predication
  - Increases the benefits of predicated code by allowing the compiler to generate more aggressively-predicated code
  - Makes predicated code less dependent on machine configuration (e.g. branch predictor)

- Disadvantages compared to predicated execution
  - Extra branch instructions use machine resources
  - Extra branch instructions increase the contention for branch predictor table entries
  - Constrains the compiler’s scope for code optimizations
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
  - **Stall** the pipeline until we know the next fetch address
  - Guess the next fetch address (**branch prediction**)
  - Employ delayed branching (**branch delay slot**)
  - Do something else (**fine-grained multithreading**)
  - Eliminate control-flow instructions (**predicated execution**)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (**multipath execution**)
Multi-Path Execution

- **Idea:** Execute both paths after a conditional branch
  - For a hard-to-predict branch: Use dynamic confidence estimation

- **Advantages:**
  + Improves performance if misprediction cost > useless work
  + No ISA change needed

- **Disadvantages:**
  -- What happens when the machine encounters another hard-to-predict branch? Execute both paths again?
    -- Paths followed quickly become exponential
  -- Each followed path requires its own context (registers, PC, GHR)
  -- Wasted work (and reduced performance) if paths merge
Dual-Path Execution versus Predication

Hard to predict

Dual-path

Predicated Execution
Remember: Branch Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Direction at fetch time</th>
<th>Number of possible next fetch addresses?</th>
<th>When is next fetch address resolved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>Unknown</td>
<td>2</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Unconditional</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Call</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Return</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Indirect</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
</tbody>
</table>

Different branch types can be handled differently
Call and Return Prediction

- Direct calls are easy to predict
  - Always taken, single target
  - Call marked in BTB, target predicted by BTB

- Returns are indirect branches
  - A function can be called from many points in code
  - A return instruction can have many target addresses
    - Next instruction after each call point for the same function
  - Observation: Usually a return matches a call
  - Idea: Use a stack to predict return addresses (Return Address Stack)
    - A fetched call: pushes the return (next instruction) address on the stack
    - A fetched return: pops the stack and uses the address as its predicted target
    - Accurate most of the time: 8-entry stack → > 95% accuracy
**Indirect Branch Prediction (I)**

- Register-indirect branches have multiple targets

![Diagram of indirect branch]

- **Conditional (Direct) Branch**

- **Indirect Jump**

- **Used to implement**
  - Switch-case statements
  - Virtual function calls
  - Jump tables (of function pointers)
  - Interface calls

R1 = MEM[R2]
branch R1
Indirect Branch Prediction (II)

- No direction prediction needed
- Idea 1: Predict the last resolved target as the next fetch address
  + Simple: Use the BTB to store the target address
  -- Inaccurate: 50% accuracy (empirical). Many indirect branches switch between different targets

- Idea 2: Use history based target prediction
  - E.g., Index the BTB with GHR XORed with Indirect Branch PC
  + More accurate
  -- An indirect branch maps to (too) many entries in BTB
    -- Conflict misses with other branches (direct or indirect)
    -- Inefficient use of space if branch has few target addresses
Issues in Branch Prediction (I)

- Need to identify a branch before it is fetched

- How do we do this?
  - BTB hit → indicates that the fetched instruction is a branch
  - BTB entry contains the “type” of the branch

- What if no BTB?
  - Bubble in the pipeline until target address is computed
  - E.g., IBM POWER4
Issues in Branch Prediction (II)

- **Latency:** Prediction is latency critical
  - Need to generate next fetch address for the next cycle
  - Bigger, more complex predictors are more accurate but slower
Complications in Superscalar Processors

- “Superscalar” processors
  - attempt to execute more than 1 instruction-per-cycle
  - must fetch multiple instructions per cycle

- Consider a 2-way superscalar fetch scenario
  (case 1) Both insts are not taken control flow inst
    - nPC = PC + 8
  (case 2) One of the insts is a taken control flow inst
    - nPC = predicted target addr
    - *NOTE* both instructions could be control-flow; prediction based on the first one predicted taken
    - If the 1st instruction is the predicted taken branch
      → nullify 2nd instruction fetched
Multiple Instruction Fetch: Concepts

- Fetch 1 inst/cycle

- Downside:
  - Flynn's bottleneck
    - If you fetch 1 inst/cycle, you cannot finish >1 inst/cycle

- Fetch 4 inst/cycle

Two major approaches

1) VLIW
   - Compiler decides what runs
   - Can be executed in parallel
   - Simple hardware

2) Superscalar
   - Hardware detects dependencies between instructions that are fetched in the same cycle
Review of Last Few Lectures

- Control dependence handling in pipelined machines
  - Delayed branching
  - Fine-grained multithreading
  - Branch prediction
    - Compile time (static)
      - Always NT, Always T, Backward T Forward NT, Profile based
    - Run time (dynamic)
      - Last time predictor
      - Hysteresis: 2BC predictor
      - Global branch correlation → Two-level global predictor
      - Local branch correlation → Two-level local predictor
  - Predicated execution
  - Multipath execution