EC 513
Computer Architecture

Branch Prediction

Prof. Michel A. Kinsky
Instruction Execution Phases

1. **Fetch:** Instruction bits retrieved from cache
2. **Decode:** Instructions placed in appropriate issue (aka “dispatch”) stage buffer
3. **Execute:** Instructions and operands sent to execution units
   - When execution completes, all results and exception flags are available
4. **Commit:** Instruction irrevocably updates architectural state (aka “graduation” or “completion”)
Branch Penalty

- How many instructions need to be killed on a misprediction?
  - Modern processors may have > 10 pipeline stages between next pc calculation and branch resolution!
Average Branches Distance

- Average Run-Length between Branches
  - Average dynamic instruction mix from SPEC92:

<table>
<thead>
<tr>
<th></th>
<th>SPECint92</th>
<th>SPECfp92</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>39 %</td>
<td>13 %</td>
</tr>
<tr>
<td>FPU Add</td>
<td></td>
<td>20 %</td>
</tr>
<tr>
<td>FPU Mult</td>
<td></td>
<td>13 %</td>
</tr>
<tr>
<td>load</td>
<td>26 %</td>
<td>23 %</td>
</tr>
<tr>
<td>store</td>
<td>9 %</td>
<td>9 %</td>
</tr>
<tr>
<td>branch</td>
<td>16 %</td>
<td>8 %</td>
</tr>
<tr>
<td>other</td>
<td>10 %</td>
<td>12 %</td>
</tr>
</tbody>
</table>

SPECint92: compress, eqntott, espresso, gcc, li
SPECfp92: doduc, ear, hydro2d, mdijdp2, su2cor
Branches and Jumps

- Each instruction fetch depends on one or two pieces of information from the preceding instruction:
  1. Is the preceding instruction a taken branch?
  2. If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JAL/JALR</td>
<td>After Inst. Decode</td>
<td>After Reg Fetch</td>
</tr>
<tr>
<td>BEQ/BNE</td>
<td>After Reg Fetch</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>
**Branches and Jumps**

- Each instruction fetch depends on one or two pieces of information from the preceding instruction:
  1. Is the preceding instruction a taken branch?
  2. If so, what is the target address?

<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>
Branch Penalties in Modern Pipelines

- UltraSPARC-III instruction fetch pipeline stages (in-order issue, 4-way superscalar, 750MHz, 2000)

<table>
<thead>
<tr>
<th>A</th>
<th>PC Generation/Mux</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>

Branch Target Address Known

Branch Direction & Jump Register Target Known

Remainder of execute pipeline (+ another 6 stages)
Control Dependencies

- Branches are very frequent
  - Approx. 20% of all instructions
- Can not wait until we know where it goes
  - Long pipelines
    - Branch outcome known after $x$ cycles
    - No scheduling past the branch until outcome known
  - Superscalars (e.g., 4-way)
    - Branch every cycle or so!
    - One cycle of work, then bubbles for $\sim x$ cycles?
Control Flow Penalty

- Assume a 4-wide superscalar pipeline with 20-cycle branch resolution latency
- How long does it take to fetch 400 instructions?
  - Assume no fetch breaks and 1 out of 4 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
Reducing Control Flow Penalty

- **Software solutions**
  - Eliminate branches - loop unrolling increases the run length
  - Reduce resolution time - instruction scheduling compute the branch condition as early as possible (of limited value)

- **Hardware solutions**
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address
    - Speculate - branch prediction speculative execution of instructions beyond the branch
  - 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
Branch Prediction

- **Motivation:**
  - Branch penalties limit performance of deeply pipelined processors
  - Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

- **Required hardware support:**
  - Prediction structures: branch history tables, branch target buffers, etc.
  - Mispredict recovery mechanisms:
    - Keep result computation separate from commit
    - Kill instructions following branch in pipeline
    - Restore state to state following branch
Branch Prediction Techniques

- 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

- Overall probability a branch is taken is ~60-70% but:

  - ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
    - bne0 (preferred taken)  beq0 (not taken)
  - ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
    - Typically reported as ~80% accurate
Static Branch Prediction

- **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 branch: target address lower than branch PC

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

- 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
Dynamic Branch Prediction

- **Key Idea:** Predict branches based on the dynamic execution behavior of the program
  - 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

- **Disadvantages**
  - More complex architecture (requires additional hardware)
Dynamic Branch Prediction

- Learning based on past behavior
- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)
Dynamic Branch Prediction

- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)
    - So we also need to predict the next fetch address (to be used in the next cycle)

- Target address remains the same for a conditional direct branch across dynamic instances
  - Store the target address from previous instance and access it with the PC
  - Branch Target Buffer (BTB) or Branch Target Address Cache
Pentium 4: A Superscalar CISC Architecture

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| TC Nxt IP | TC Fetch | Drive | Alloc | Rename | Que | Sch | Sch | Disp | Disp | RF | RF | Ex | Flgs | BrCk | Drive |

3.2 GB/s System Interface

L2 Cache and Control

BTB

Trace Cache

Rename/Alloc

μop Queues

Schedulers

FP RF

L1 D-Cache and D-TLB

Integer RF

FP store

FP move

Fmul

Fadd

MMX

SSE

Load AGU

Store AGU

μCode ROM

Decoder

BTB & I-TLB

L1 D-Cache and D-TLB
Fetch Stage with Branch Target Buffer

- Program Counter
- Taken?
- Hit?
- PC + Inst size (e.g., 4)
- Target address
- Next Fetch Address
Dynamic Prediction

Prediction as a feedback control process

Operations
- Predict
- Update
Predictor Primitive

- Indexed table holding values
- Operations
  - Predict
  - Update
- Algebraic notation
  - Prediction = P[Width, Depth](Index; Update)
One-bit Predictor

- $A21064(\text{PC}; \text{T}) = P[1, 2K](\text{PC}; \text{T})$
- What happens on loop branches?
  - At best, mispredicts twice for every use of loop
Last Time Predictor

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
  - TTTTTTTTTTTNNNNNNNNNNN $\rightarrow$ 90% accuracy
- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = $(N-2)/N$
  - Works well for for loops with a large number of iterations
  - Works poorly for loops with small number of iterations or non-correlated branches
  - TNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTN $\rightarrow$ 0% accuracy
Implementation of the 1-bit Predictor

Tag Table

BTB

N-bit

One Bit Per Branch

Taken?

PC+4

Next PC

1 0
State Machine of the 1-bit Predictor

- **Predict not Taken**
  - Actually not taken
  - Actually taken

- **Predict Taken**
  - Actually taken
  - Actually not taken
Two-bit Predictor

- Counter\([W,D](I; T) = P[W, D]\)
  - \((I; \text{if } T \text{ then } P+1 \text{ else } P-1)\)
- \(A21164(\text{PC}; T) = \text{MSB}(\text{Counter}[2, 2K](\text{PC}; T))\)
### Branch Prediction Bits

- Assume 2 BP bits per instruction
- Use saturating counter

<table>
<thead>
<tr>
<th>On taken</th>
<th>On taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>Strongly taken</td>
</tr>
<tr>
<td>1 0</td>
<td>Weakly taken</td>
</tr>
<tr>
<td>0 1</td>
<td>Weakly ¬taken</td>
</tr>
<tr>
<td>0 0</td>
<td>Strongly ¬taken</td>
</tr>
</tbody>
</table>
Two-bit Saturation Predictor

Pred Taken

Pred !Taken

Pred Taken

Pred !Taken
Forward Two-bit Predictor

- Pred Taken → actually taken → Pred Taken
- Pred !Taken → actually !taken → Pred !Taken
- Pred Taken → actually taken → Pred !Taken
- Pred !Taken → actually !taken → Pred !Taken
Branch History Table

**Fetch PC**

I-Cache

**Instruction**

Opcode

Offset

Branch?

Target PC

**BHT Index**

$2^k$-entry BHT, 2 bits/entry

Taken/¬Taken?

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions
History Register

- History(PC, T) = P(PC; P ⊕ T)
Exploiting Spatial Correlation

```c
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

- If first condition false, second condition also false
- History register, H, records the direction of the last N branches executed by the processor
Two-level Predictor

\[
\text{GHist}(;T) = \text{Counter}(\text{History}(0, T); T)
\]

\[
\text{Ind-Ghist}(\text{PC};T) = \text{Counter}(\text{PC} \parallel \text{Hist}(\text{GHist}(;T); T))
\]
Tournament Predictor

- Alpha 21264
  - 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
Hybrid Predictor

- Pentium-M
  - Hybrid, but uses tag-based selection mechanism
RISC-V Pipeline with a BTB
The pipeline Front-End with the BTB

- **IF**
  - Inst Mem gets PC and fetches new inst
  - BTB gets PC and looks it up
  - IF/ID latch loaded with new inst
  - Branch?
    - yes: IF/ID latch loaded with pred inst
    - no: IF/ID latch loaded with seq. inst
  - PC \( \leftarrow \text{pred addr} \)
  - PC \( \leftarrow \text{PC + 4} \)

- **ID**
  - BTB Hit?
    - yes
    - Br taken?
      - yes
      - no
    - no
  - PC \( \leftarrow \text{PC + 4} \)

- **EXE**
Next Class

- SIMD and Vector Processors