Mid-Semester Exam

- November 8
- In class
- All covered material in the course can be part of the exam
High-Level Summary of Last Few Lectures

- DRAM
  - Cutting-edge Issues in Latency, Retention and New Uses

- SIMD Processing
  - Array Processors
  - Vector Processors
  - SIMD Extensions
  - Graphics Processing Units
    - GPU Architecture

- Intro to Genome Analysis
  - Algorithm-Architecture Co-Design
Agenda for Today (& Tomorrow)

- Control Dependence Handling
  - Problem
  - Six solutions

- Branch Prediction

- Other Methods of Control Dependence Handling
Required Readings

  Required

  - MICRO Test of Time Award Winner (after 24 years)
  - Required
Recommended Readings

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

  - **Recommended**
Recall: Pipelining
Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage

- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations

- Handling exceptions, interrupts

- Advanced: Improving pipeline throughput
  - Minimizing stalls
Causes of Pipeline *Stalls*

- **Stall**: A condition when the pipeline stops moving
- **Resource contention**
- **Dependences** *(between instructions)*
  - Data
  - Control
- **Long-latency (multi-cycle) operations**
Dependences and Their Types

- Also called “dependency” or less desirably “hazard”

- Dependences dictate ordering requirements between instructions

- Two types
  - Data dependence
  - Control dependence

- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately
Control Dependence Handling
Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?

- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?
## 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
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)*
Stall Fetch Until Next PC is Known: Good Idea?

This is the case with non-control-flow and unconditional br instructions!
The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions

- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  - N cycles: (minimum) branch resolution latency

- If we are fetching W instructions per cycle (i.e., if the pipeline is W wide)
  - A branch misprediction leads to $N \times W$ wasted instruction slots
Importance of The Branch Problem

- Assume $N = 20$ (20 pipe stages), $W = 5$ (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work; IPC = $500/100$
  - 99% accuracy
    - 100 (correct path) + $20 \times 1$ (wrong path) = 120 cycles
    - 20% extra instructions fetched; IPC = $500/120$
  - 90% accuracy
    - 100 (correct path) + $20 \times 10$ (wrong path) = 300 cycles
    - 200% extra instructions fetched; IPC = $500/300$
  - 60% accuracy
    - 100 (correct path) + $20 \times 40$ (wrong path) = 900 cycles
    - 800% extra instructions fetched; IPC = $500/900$
Branch Prediction
Branch Prediction: Guess the Next Instruction to Fetch

Branch Prediction: Guess the Next Instruction to Fetch

Branch prediction

Stall fetch

PC 0x0001

12 cycles

8 cycles

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

Flush!!
PC
I-$\rightarrow$ DEC
RF
D-$\rightarrow$ WB
WB

0x0001
0x0002
0x0003
0x0004
0x0005
0x0006
0x0007
Simplest: Always Guess \( \text{NextPC} = \text{PC} + 4 \)

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of \textit{next fetch address prediction} (and branch prediction)

How can you make this more effective?

Idea: \textbf{Maximize the chances that the next sequential instruction is the next instruction to be executed}

- Software: Lay out the control flow graph such that the “likely next instruction” is on the not-taken path of a branch
  - Profile guided code positioning \( \rightarrow \) Pettis & Hansen, PLDI 1990.
- Hardware: ??? (how can you do this in hardware...)
  - Cache traces of executed instructions \( \rightarrow \) Trace cache
Guessing $\text{NextPC} = \text{PC} + 4$

- How else can you make this more effective?

- Idea: Get rid of control flow instructions (or minimize their occurrence)

- How?
  1. Get rid of unnecessary control flow instructions $\rightarrow$ combine predicates (predicate combining)
  2. Convert control dependences into data dependences $\rightarrow$ predicated execution
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

Inst_h is a branch
Performance Analysis

- correct guess $\Rightarrow$ no penalty $\sim 86\%$ of the time
- incorrect guess $\Rightarrow$ 2 bubbles

Assume

- no data dependency related stalls
- 20\% control flow instructions
- 70\% of control flow instructions are taken
- $\text{CPI} = \left[ 1 + (0.20 \times 0.7) \times 2 \right] = \left[ 1 + 0.14 \times 2 \right] = 1.28$

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

- Resolve branch condition and target address early

\[ \text{CPI} = \left[ 1 + (0.2 \times 0.7) \times 1 \right] = 1.14 \]

Is this a good idea?
Branch Prediction (A Bit More 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
Fetch Stage with BTB and Direction Prediction

- Program Counter
- Address of the current branch
- Direction predictor (taken?)
- PC + inst size
- Next Fetch Address
- Cache of Target Addresses (BTB: Branch Target Buffer)

- Taken?
- Hit?
More Sophisticated Branch Direction Prediction

- Global branch history
- Program Counter
- Which direction earlier branches went
- Address of the current branch
- XOR
- Direction predictor (taken?)
- PC + inst size
- hit?
- target address
- Cache of Target Addresses (BTB: Branch Target Buffer)
- Next Fetch Address
Three Things to Be Predicted

- Requires three things to be predicted at fetch stage:
  1. Whether the fetched instruction is a branch
  2. (Conditional) branch direction
  3. Branch target address (if taken)

- Third (3.) can be accomplished using a BTB
  - Remember target address computed last time branch was executed

- First (1.) can be accomplished using a BTB
  - If BTB provides a target address for the program counter, then it must be a branch
  - Or, we can store “branch metadata” bits in instruction cache/memory → partially decoded instruction stored in I-cache

- Second (2.): How do we predict the direction?
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
  - Advanced algorithms (e.g., using perceptrons)
Static Branch Prediction (I)

- **Always not-taken**
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40% (for conditional branches)
  - Remember: Compiler can layout code such that the likely path is the “not-taken” path $\rightarrow$ more effective prediction

- **Always taken**
  - No direction prediction
  - Better accuracy: ~60-70% (for conditional branches)
    - 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
Profile-based

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

+ Per branch prediction (more accurate than schemes in previous slide) \(ightarrow\) accurate if profile is representative!
-- Requires hint bits in the branch instruction format
-- Accuracy depends on dynamic branch behavior:
  TTTTTTTTTTTNNNNNNNNNN \(\rightarrow\) 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 (IV)

- **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?
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

- How would you do that?

- What is the common disadvantage 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...)
  - What is a Dynamic Compiler?
    - A compiler that generates code at runtime: Remember Transmeta?
    - Java JIT (just in time) compiler, Microsoft CLR (common lang. runtime)
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
  - Advanced algorithms (e.g., using perceptrons)
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
    TTTTTTTTTTTNNNNNNNNNNN → 90% accuracy

- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = \( \frac{N-2}{N} \)

+ Loop branches for loops with large N (number of iterations)
-- Loop branches for loops with small N (number of iterations)
  TNTNTNTNTNTNTNTNTNTNTNTNTNTN → 0% accuracy
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 not taken
  - Actually taken
Improving the Last Time Predictor

- **Problem:** A last-time predictor changes its prediction from $T \rightarrow NT$ or $NT \rightarrow T$ 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

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

- Accuracy for a loop with N iterations = \( \frac{N-1}{N} \)
  TNTNTNTNTNTNTN \( \rightarrow \) 50% accuracy
  (assuming counter initialized to weakly taken)

+ Better prediction accuracy
-- More hardware cost (but counter can be part of a BTB entry)
State Machine for 2-bit Saturating Counter

- Counter using *saturating arithmetic*
  - Arithmetic with maximum and minimum values

```
actually taken
  pred taken
  11
```
```
actually !taken
  actually taken
  pred taken
  10
```
```
actually !taken
  actually taken
  pred !taken
  01
```
```
actually !taken
  actually taken
  pred !taken
  00
```
```
Hysteresis Using a 2-bit Counter

Change prediction after 2 consecutive mistakes
Is This Good Enough?

- ~85-90% accuracy for many programs with 2-bit counter based prediction (also called bimodal prediction)

- Is this good enough?

- How big is the branch problem?
Let’s Do the Exercise Again

- Assume $N = 20$ (20 pipe stages), $W = 5$ (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

How long does it take to fetch 500 instructions?

- **100% accuracy**
  - 100 cycles (all instructions fetched on the correct path)
  - No wasted work; IPC = 500/100

- **90% accuracy**
  - 100 (correct path) + 20 * 10 (wrong path) = 300 cycles
  - 200% extra instructions fetched; IPC = 500/300

- **85% accuracy**
  - 100 (correct path) + 20 * 15 (wrong path) = 400 cycles
  - 300% extra instructions fetched; IPC = 500/400

- **80% accuracy**
  - 100 (correct path) + 20 * 20 (wrong path) = 500 cycles
  - 400% extra instructions fetched; IPC = 500/500
Can We Do Better: Two-Level Prediction

- 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 are 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
Global Branch Correlation (III)

- Eqntott, SPEC’92: Generates truth table from Boolean expr.

```c
if (aa==2) {
    aa=0;
    ;; B1
}

if (bb==2) {
    bb=0;
    ;; B2
}

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

How Does the Global Predictor Work?

This branch tests $i$

Last 4 branches test $j$

History: TTTN

Predict taken for $i$

Next history: TTNT

(shift in last outcome)

Intel Pentium Pro Branch Predictor

- Two level global 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
Global Branch Correlation Analysis

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

- Only 3 past branches’ directions *really* matter
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

Review: One-Level Branch Predictor

Direction predictor (2-bit counters)

Program Counter

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

PC + inst size

taken?

Next Fetch Address

hit?

target address
Two-Level Global History Branch Predictor

- Global branch history
- Program Counter
- Address of the current instruction
- Direction predictor (2-bit counters)
- Cache of Target Addresses (BTB: Branch Target Buffer)

Which direction earlier branches went

Next Fetch Address

PC + inst size

hit?

taken?

target address
Two-Level Gshare Branch Predictor

- Global branch history
- Program Counter
- Which direction earlier branches went
- Address of the current instruction
- Direction predictor (2-bit counters)
- Cache of Target Addresses (BTB: Branch Target Buffer)
- Next Fetch Address
- Taken?
- PC + inst size
- Hit?
- Target address

- XOR

The diagram illustrates the flow of information through the branch prediction mechanism. The program counter (PC) and instruction size are XOR-ed with the global branch history to determine the direction predictor's inputs. The prediction is then checked against the cache of target addresses (BTB), and the outcome is used to fetch the next address.
An Issue: Interference in the PHTs

- Sharing the PHTs between histories/branches leads to interference
  - Different branches map to the same PHT entry and modify it
  - Interference can be positive, negative, or neutral

- Interference can be eliminated by dedicating a PHT per branch
  -- Too much hardware cost

- How else can you eliminate or reduce interference?
Reducing Interference in PHTs (I)

- Increase size of PHT

- Branch filtering
  - Predict highly-biased branches separately so that they do not consume PHT entries
  - E.g., static prediction or BTB based prediction

- Hashing/index-randomization
  - Gshare
  - Gskew

- Agree prediction
Biased Branches and Branch Filtering

- **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 (e.g., last time, static, ...)

Reducing Interference: Gshare

- Idea 1: Randomize the indexing function into the PHT such that probability of two branches mapping to the same entry reduces
  - Gshare predictor: GHR hashed with the Branch PC
  - Better utilization of PHT
  - More context information
  - Increases access latency

Reducing Interference: Agree Predictor

- Idea 2: Agree prediction
  - Each branch has a “bias” bit associated with it in BTB
    - Ideally, most likely outcome for the branch
  - High bit of the PHT counter indicates whether or not the prediction agrees with the bias bit (not whether or not prediction is taken)
  - + Reduces negative interference (Why???)
  - -- Requires determining bias bits (compiler vs. hardware)

Why Does Agree Prediction Make Sense?

- Assume two branches have taken rates of 85% and 15%.
- Assume they conflict in the PHT

Let’s compute the probability they have opposite outcomes

- Baseline predictor:
  - $P (b_1 \text{ T}, b_2 \text{ NT}) + P (b_1 \text{ NT}, b_2 \text{ T})$
  - $= (85\% \times 85\%) + (15\% \times 15\%) = 74.5\%$

- Agree predictor:
  - Assume bias bits are set to T (b1) and NT (b2)
  - $P (b_1 \text{ agree}, b_2 \text{ disagree}) + P (b_1 \text{ disagree}, b_2 \text{ agree})$
  - $= (85\% \times 15\%) + (15\% \times 85\%) = 25.5\%$

- Works because most branches are biased (not 50% taken)
Reducing Interference: Gskew

- Idea 3: Gskew predictor
  - Multiple PHTs
  - Each indexed with a different type of hash function
  - Final prediction is a majority vote
  - Distributes interference patterns in a more randomized way (interfering patterns less likely in different PHTs at the same time)
    - More complexity (due to multiple PHTs, hash functions)


More Techniques to Reduce PHT Interference

- The bi-mode predictor
  - Separate PHTs for mostly-taken and mostly-not-taken branches
  - Reduces negative aliasing between them

- The YAGS predictor
  - Use a small tagged “cache” to predict branches that have experienced interference
  - Aims to not mispredict them again

- Alpha EV8 (21464) branch predictor
Can We Do Better: Two-Level Prediction

- Last-time and 2BC predictors exploit only “last-time” predictability for a given branch

- 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 (in addition to 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)\textsuperscript{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.

More Motivation for Local History

- To predict a loop branch “perfectly”, we want to identify the last iteration of the loop.
- By having a separate PHT entry for each local history, we can distinguish different iterations of a loop.
- Works for “short” loops.
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

hit?

taken?

target address
BHR can be global (G), per set of branches (S), or per branch (P)

PHT counters can be adaptive (A) or static (S)

PHT can be global (g), per set of branches (s), or per branch (p)

Can We Do Even Better?

- Predictability of branches varies
  - Some branches are more predictable using local history
  - Some using global
  - For others, a simple two-bit counter is enough
  - Yet for others, a bit is enough

- Observation: There is heterogeneity in predictability behavior of branches
  - No one-size fits all branch prediction algorithm for all branches

- Idea: Exploit that heterogeneity by designing heterogeneous branch predictors
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 \textit{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

Are We Done w/ Branch Prediction?

- Hybrid branch predictors work well
  - E.g., 90-97% prediction accuracy on average

- Some “difficult” workloads still suffer, though!
  - E.g., gcc
  - Max IPC with tournament prediction: 9
  - Max IPC with perfect prediction: 35
Are We Done w/ Branch Prediction?

Some Other Branch Predictor Types

- **Loop branch detector and predictor**
  - Loop iteration count detector/predictor
  - Works well for loops with small number of iterations, where iteration count is predictable
  - Used in Intel Pentium M

- **Perceptron branch predictor**
  - Learns the *direction correlations* between individual branches
  - Assigns weights to correlations

- **Hybrid history length based predictor**
  - Uses different tables with different history lengths
The advanced branch prediction in the Pentium M processor is based on the Intel Pentium® 4 processor’s [6] branch predictor. On top of that, two additional predictors to capture special program flows, were added: a Loop Detector and an Indirect Branch Predictor.

Gochman et al.,
“The Intel Pentium M Processor: Microarchitecture and Performance,”
Perceptrons for Learning Linear Functions

- A perceptron is a simplified model of a biological neuron
- It is also a simple **binary classifier**

- A perceptron maps an input vector X to a 0 or 1
  - Input = Vector X
  - Perceptron learns the linear function (if one exists) of how each element of the vector affects the output (stored in an internal Weight vector)
  - Output = Weight.X + Bias > 0

- In the branch prediction context
  - Vector X: Branch history register bits
  - Output: Prediction for the current branch

---

**Perceptron Branch Predictor (I)**

- **Idea:** Use a perceptron to learn the correlations between branch history register bits and branch outcome

- **A perceptron learns a target Boolean function of N inputs**
  
  Each branch associated with a perceptron

  A perceptron contains a set of weights $w_i$
  
  - Each weight corresponds to a bit in the GHR
  - How much the bit is correlated with the direction of the branch
  - Positive correlation: large + weight
  - Negative correlation: large - weight

  Prediction:
  
  - Express GHR bits as 1 (T) and -1 (NT)
  - Take dot product of GHR and weights
  - If output > 0, predict taken

Perceptron Branch Predictor (II)

Prediction function:

\[ y = w_0 + \sum_{i=1}^{n} x_i w_i \]

Dot product of GHR and perceptron weights

Bias weight (bias of branch independent of the history)

Output compared to 0

Training function:

\[
\begin{align*}
    \text{if } \text{sign}(y_{out}) &\neq t \text{ or } |y_{out}| \leq \theta \text{ then} \\
    \text{for } i := 0 \text{ to } n \text{ do} \\
    &w_i := w_i + t x_i \\
    \text{end for} \\
    \text{end if}
\end{align*}
\]
Perceptron Branch Predictor (III)

- Advantages
  - + More sophisticated learning mechanism \(\rightarrow \) better accuracy

- Disadvantages
  - -- Hard to implement (adder tree to compute perceptron output)
  - -- Can learn only linearly-separable functions
    - e.g., cannot learn XOR type of correlation between 2 history bits and branch outcome
Prediction Using Multiple History Lengths

- Observation: Different branches require different history lengths for better prediction accuracy.

- Idea: Have multiple PHTs indexed with GHRs with different history lengths and intelligently allocate PHT entries to different branches.

TAGE: Tagged & prediction by the longest history matching entry

Tagless base predictor

TAGE: Multiple Tables

Altpred: Alternative prediction

TAGE: Which Table to Use?

- General case:
  - Longest history-matching component provides the prediction

- Special case:
  - Many mispredictions on newly allocated entries: weak Ctr

On many applications, **Altpred** more accurate than **Pred**
- Property dynamically monitored through 4-bit counters

A Tagged Table Entry

- **Ctr**: 3-bit prediction counter
- **U**: 1 or 2-bit counters
  - Was the entry recently useful?
- **Tag**: partial tag

State of the Art in Branch Prediction

- See the Branch Prediction Championship
  - [https://www.jilp.org/cbp2016/program.html](https://www.jilp.org/cbp2016/program.html)


Figure 1. The TAGE-SC-L predictor: a TAGE predictor backed with a Statistical Corrector predictor and a loop predictor
Another Direction: Helper Threading

- Idea: Pre-compute the outcome of the branch with a separate, customized thread (i.e., a helper thread)

Figure 3. The Microthread Builder

Branch Confidence Estimation

- **Idea:** Estimate if the prediction is likely to be correct
  - i.e., estimate how “confident” you are in the prediction

- **Why?**
  - Could be very useful in deciding how to speculate:
    - What predictor/PHT to choose/use
    - Whether to keep fetching on this path
    - Whether to switch to some other way of handling the branch, e.g. dual-path execution (eager execution) or dynamic predication
    - ...

How to Estimate Confidence

An example estimator:

- Keep a record of correct/incorrect outcomes for the past $N$ instances of the “branch”
- Based on the correct/incorrect patterns, guess if the current prediction will likely be correct/incorrect

What to Do With Confidence Estimation?

- An example application: Pipeline Gating

Another application: Statistical Correction of Prediction


We did not cover the following slides in lecture. These are for your benefit.
Issues in Fast & Wide Fetch Engines
I-Cache Line and Way Prediction

- **Problem:** Complex branch prediction can take too long (many cycles)

- **Goal**
  - Quickly generate (a reasonably accurate) next fetch address
  - Enable the fetch engine to run at high frequencies
  - Override the quick prediction with more sophisticated prediction

- **Idea:** Predict the next cache line and way at the time you fetch the current cache line

- **Example Mechanism (e.g., Alpha 21264)**
  - Each cache line tells which line/way to fetch next (prediction)
  - On a fill, line/way predictor points to next sequential line
  - On branch resolution, line/way predictor is updated
  - If line/way prediction is incorrect, one cycle is wasted
Figure 3. Alpha 21264 instruction fetch. The line and way prediction (wrap-around path on the right side) provides a fast instruction fetch path that avoids common fetch stalls when the predictions are correct.


Alpha 21264 Line & Way Prediction
Issues in Wide Fetch Engines

- Wide Fetch: Fetch multiple instructions per cycle
- Superscalar
- VLIW
- SIMT (GPUs’ single-instruction multiple thread model)

Wide fetch engines suffer from the branch problem:
- How do you feed the wide pipeline with useful instructions in a single cycle?
- What if there is a taken branch in the “fetch packet”?
- What is there are “multiple (taken) branches” in the “fetch packet”?
Fetching Multiple Instructions Per Cycle

- Two problems

1. **Alignment** of instructions in I-cache
   - What if there are not enough (N) instructions in the cache line to supply the fetch width?

2. **Fetch break**: Branches present in the fetch block
   - Fetching sequential instructions in a single cycle is easy
   - What if there is a control flow instruction in the N instructions?
   - Problem: *The direction of the branch is not known but we need to fetch more instructions*

- These can cause effective fetch width < peak fetch width
Wide Fetch Solutions: Alignment

- **Large cache blocks**: Hope N instructions are contained in the block

- **Split-line fetch**: If address falls into second half of the cache block, fetch the first half of next cache block as well
  - Enabled by banking of the cache
  - Allows sequential fetch across cache blocks in one cycle
  - Intel Pentium and AMD K5
Split Line Fetch

Cache Banking

Memory Map

Cache

Need alignment logic:
Short Distance Predicted-Taken Branches

First Iteration (Branch B taken to E)

Second Iteration (Branch B fall through to C)
Techniques to Reduce Fetch Breaks

- Compiler
  - Code reordering (basic block reordering)
  - Superblock

- Hardware
  - Trace cache

- Hardware/software cooperative
  - Block structured ISA
Basic Block Reordering

- Not-taken control flow instructions not a problem: no fetch break: make the likely path the not-taken path
- Idea: Convert taken branches to not-taken ones
  - i.e., reorder basic blocks (after profiling)
  - Basic block: code with a single entry and single exit point

- Code Layout 1 leads to the fewest fetch breaks
Basic Block Reordering


- Advantages:
  + Reduced fetch breaks (assuming profile behavior matches runtime behavior of branches)
  + Increased I-cache hit rate
  + Reduced page faults

- Disadvantages:
  -- Dependent on compile-time profiling
  -- Does not help if branches are not biased
  -- Requires recompilation
Superblock

- **Idea**: Combine frequently executed basic blocks such that they form a single-entry multiple-exit larger block, which is likely executed as straight-line code

  + Helps wide fetch
  + Enables aggressive compiler optimizations and code reordering within the superblock

  -- Increased code size
  -- Profile dependent
  -- Requires recompilation

Superblock Formation (I)

Is this a superblock?
Tail duplication: duplication of basic blocks after a side entrance to eliminate side entrances

→ transforms a trace into a superblock.
Superblock Code Optimization Example

Original Code

opA: mul r1<-r2,3

opB: add r2<-r2,1

opC: mul r3<-r2,3

99

Code After Superblock Formation

opA: mul r1<-r2,3

opB: add r2<-r2,1

opC: mov r3<-r1

opC’: mul r3<-r2,3

99

Code After Common Subexpression Elimination

opA: mul r1<-r2,3

opB: add r2<-r2,1

opC: mov r3<-r1

opC’: mul r3<-r2,3
Techniques to Reduce Fetch Breaks

- Compiler
  - Code reordering (basic block reordering)
  - Superblock

- Hardware
  - Trace cache

- Hardware/software cooperative
  - Block structured ISA
Trace Cache: Basic Idea

- A trace is a **sequence of executed instructions**.
- It is specified by a start address and the outcomes of control transfer instructions within the trace.
- **Traces repeat**: programs have frequently executed paths.
- **Trace cache idea**: Store a dynamic instruction sequence in the same physical location so that it can be fetched in unison.

(a) Instruction cache.

(b) Trace cache.
Reducing Fetch Breaks: Trace Cache

- Dynamically determine the basic blocks that are executed consecutively
- Trace: Consecutively executed basic blocks
- Idea: Store consecutively-executed basic blocks in physically-contiguous internal storage (called trace cache)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>. . .</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
</table>

Dynamic Instruction Stream

- **Basic trace cache operation:**
  - Fetch from consecutively-stored basic blocks (predict next trace or branches)
  - Verify the executed branch directions with the stored ones
  - If mismatch, flush the remaining portion of the trace

 Rotenberg et al., “Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching,” MICRO 1996. **Received the MICRO Test of Time Award 20 years later**
Trace Cache: Example
An Example Trace Cache Based Processor

Example Multiple Branch Predictor

What Does A Trace Cache Line Store?

- 16 slots for instructions. Instructions are stored in decoded form and occupy approximately five bytes for a typical ISA. Up to three branches can be stored per line. Each instruction is marked with a two-bit tag indicating to which block it belongs.

- Four target addresses. With three basic blocks per segment and the ability to fetch partial segments, there are four possible targets to a segment. The four addresses are explicitly stored allowing immediate generation of the next fetch address, even for cases where only a partial segment matches.

- Path information. This field encodes the number and directions of branches in the segment and includes bits to identify whether a segment ends in a branch and whether that branch is a return from subroutine instruction. In the case of a return instruction, the return address stack provides the next fetch address.

Trace Cache: Advantages/Disadvantages

+ Reduces fetch breaks (assuming branches are biased)
+ No need for decoding (instructions can be stored in decoded form)
+ Can enable dynamic optimizations within a trace

-- Requires hardware to form traces (more complexity) \rightarrow called fill unit
-- Results in duplication of the same basic blocks in the cache
-- Can require the prediction of multiple branches per cycle
   -- If multiple cached traces have the same start address
   -- What if XYZ and XYT are both likely traces?
Trace Cache Design Issues: Example

- **Branch promotion**: promote highly-biased branches to branches with static prediction
  + Larger traces
  + No need for consuming branch predictor BW
  + Can enable optimizations within trace

-- Requires hardware to determine highly-biased branches

**Without Branch Promotion**

**With Branch Promotion**
How to Determine Biased Branches

Figure 6.19: Diagram of the branch bias table.
Fill Unit Optimizations

- Fill unit constructs traces out of decoded instructions
- Can perform optimizations across basic blocks
  - Branch promotion: promote highly-biased branches to branches with static prediction
  - Can treat the whole trace as an atomic execution unit
    - All or none of the trace is retired (based on branch directions in trace)
    - Enables many optimizations across blocks
- Dead code elimination
- Instruction reordering
- Reassociation

```
ADDI Rx ← Ry + 4
ADDI Rx ← Ry + 4
ADDI Rz ← Rx + 4
ADDI Rz ← Ry + 8
```

Remember This Optimization?

Original Code

```
opA: mul r1<-r2,3
    opB: add r2<-r2,1
    opC: mul r3<-r2,3
```

Part of Trace in Fill Unit

```
opA: mul r1<-r2,3
    99
    opC: mov r3<-r1
    opC: mul r3<-r2,3
```

Optimized Trace

```
opA: mul r1<-r2,3
    99
    opC: mov r3<-r1
```
Intel Pentium 4 Trace Cache

- A 12K-uop trace cache replaces the L1 I-cache
- Trace cache stores decoded and cracked instructions
  - Micro-operations (uops): returns 6 uops every other cycle
- x86 decoder can be simpler and slower

Diagram:
- Front End BTB: 4K Entries
- ITLB & Prefetcher
- x86 Decoder
- Trace Cache BTB: 512 Entries
- Trace Cache: 12K uop’s
- L2 Interface