Digital Design & Computer Arch.
Lecture 18: Branch Prediction II

Prof. Onur Mutlu

ETH Zürich
Spring 2022
5 May 2022
We Are Almost Done w/ This

- Microarchitecture Fundamentals
  - Single-cycle Microarchitectures
  - Multi-cycle Microarchitectures

- Pipelining & Precise Exceptions
  - Pipelining
  - Pipelined Processor Design
    - Control & Data Dependence Handling
    - Precise Exceptions: State Maintenance & Recovery

- Out-of-Order & Superscalar Execution
  - Out-of-Order Execution
  - Dataflow & Superscalar Execution
  - Branch Prediction
Readings for Today

- H&H Chapters 7.8 and 7.9
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)
Recall: 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 * 1 (wrong path) = 120 cycles
  - 20% extra instructions fetched; IPC = 500/120

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

- **60% accuracy**
  - 100 (correct path) + 20 * 40 (wrong path) = 900 cycles
  - 800% extra instructions fetched; IPC = 500/900
Recall: Fetch Stage with BTB and Direction Prediction

Direction predictor (taken?)

Program Counter

Address of the current branch

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
  - Advanced algorithms (e.g., using perceptrons, geometric history)
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 → 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
Static Branch Prediction (II)

- 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) → accurate if profile is representative!

-- Requires hint bits in the branch instruction format

-- Accuracy depends on dynamic branch behavior:

TTTTTTTTTTTNNNNNNNNN → 50% accuracy
TNTNTNTNTNTNTNTNTNTN → 50% accuracy

-- Accuracy depends on representativeness of profile input set

TTTTTTTTTTTTTTTTTTTNNN → 90% accuracy (or 10%?)
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)

- Ball and Larus, ”Branch prediction for free,” PLDI 1993.
  - 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
    - Java JIT (just in time) compiler, Microsoft CLR (common lang. runtime)
Recall: Rosetta 2 Binary Translator

Rosetta 2 [ edit ]

In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the Mac transition to Apple silicon. The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.[2][8]

In addition to the just-in-time (JIT) translation support, Rosetta 2 offers ahead-of-time compilation (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.[9]

Rosetta 2's performance has been praised greatly.[10][11] In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 memory ordering in Apple M1 SOC.[12]

Although Rosetta 2 works for most software, some software doesn't work at all[13] or is reported to be "sluggish".[14] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes assembly language code, or that generates machine code), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.[15]
The Secret of Denver: Binary Translation & Code Optimization

As we alluded to earlier, NVIDIA's decision to forgo a traditional out-of-order design for Denver means that much of Denver's potential is contained in its software rather than its hardware. The underlying chip itself, though by no means simple, is at its core a very large in-order processor. So it falls to the software stack to make Denver sing.

Accomplishing this task is NVIDIA's dynamic code optimizer (DCO). The purpose of the DCO is to accomplish two tasks: to translate ARM code to Denver's native format, and to optimize this code to make it run better on Denver. With no out-of-order hardware on Denver, it is the DCO's task to find instruction level parallelism within a thread to fill Denver's many execution units, and to reorder instructions around potential stalls, something that is no simple task.
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, geometric history)
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
    \[TTTTTTTTTTNNNNNNNNNN \rightarrow 90\% \text{ accuracy}\]

- 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 N (number of iterations)
  \(\sim 100\% \text{ accuracy}\)
-- Loop branches for loops with small N (number of iterations)
  \[TNTNTNTNTNTNTNNTNTN \rightarrow 0\% \text{ accuracy}\]
Implementing the Last-Time Predictor (I)

Direction predictor (taken last time?)

Cache of Target Addresses (BTB: Branch Target Buffer)

Program Counter

Address of the current branch
The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch.
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 each for $T$ or $NT$ instead of 1 state for each

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter (2BC)
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

- Also called bimodal prediction
Counter using *saturating arithmetic*
- Arithmetic with maximum and minimum values

**State Machine for 2-bit Saturating Counter**

- pred taken 11
- pred !taken 01
- pred taken 10
- pred !taken 00

Arrows indicate transitions between states.
Hysteresis Using a 2-bit Counter

Change prediction after 2 consecutive mistakes
Two-Bit Counter Based Prediction

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

- Accuracy for a loop with N iterations = \((N-1)/N\)

TNTNTNTNTNTNTNTNTNTNTNTNTNTNT → 50% accuracy

(assuming counter initialized to weakly taken)

+ Better prediction accuracy

-- More hardware cost (but counter can be part of a BTB entry)
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

\[
\text{if (x<1) . . .} \\
\text{branch Y: if (cond1) a = 2;} \\
\text{if (x>1) . . .} \\
\text{branch X: if (a == 0)}
\]

- 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 workload
  - Generates truth table from Boolean expression

```java
if (aa==2) {
    aa=0;
}
if (bb==2) {
    bb=0;
}
if (aa!=bb) {
    ....
}
```

If B1 is taken (i.e., aa=0@B3) and B2 is taken (i.e. bb=0@B3) then B3 is not 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?

```plaintext
for (i=0; i<100; i++)
    for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

<table>
<thead>
<tr>
<th>test</th>
<th>value</th>
<th>GR</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>j&lt;3</td>
<td>j=1</td>
<td>1101</td>
<td>taken</td>
</tr>
<tr>
<td>j&lt;3</td>
<td>j=2</td>
<td>1011</td>
<td>taken</td>
</tr>
<tr>
<td>j&lt;3</td>
<td>j=3</td>
<td>0111</td>
<td>not taken</td>
</tr>
<tr>
<td>i&lt;100</td>
<td></td>
<td>1110</td>
<td>usually taken</td>
</tr>
</tbody>
</table>

This branch tests i
Last 3 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

- First widely commercially successful out-of-order execution machine
  - Out-of-order + superscalar + 2-level branch prediction + precise exceptions using reorder buffer
Intel Pentium Pro (1995)

- Processor chip
- Level 2 cache chip
- Multi-chip module package
Aside: 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 used for prediction
  + Better utilization of the two-bit counter array
  -- 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

hit?

taken?

Next Fetch Address

target address
Two-Level Global History Branch Predictor

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

Diagram:
- Flow from Global branch history to Direction predictor
- Flow from Direction predictor to Cache of Target Addresses
- Flow from Cache of Target Addresses to Next Fetch Address
- Logic gates for prediction and decision-making
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)

taken?

PC + inst size

hit?

Next Fetch Address

Cache of Target Addresses (BTB: Branch Target Buffer)
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 $\langle 1110 \rangle^n$ 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?
Aside: Two-Level Predictor Taxonomy

- 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 Better?
Can We Do Even Better?

- Predictability of branches varies
- Some branches are more predictable using local history
- Some branches are more predictable using global
- For others, a simple two-bit counter is enough
- Yet for others, a single 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 (hybrid) 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 *warmup* time (faster-warmup predictor used until the slower-warmup predictor warms up)

- **Disadvantages:**
  -- Need “meta-predictor” or “selector” to decide which predictor to use
  -- Longer access latency

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

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, ...)

Are We Done w/ Branch Prediction?

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

- Some “difficult” workloads still suffer a lot, though!
  - E.g., gcc
  - Max IPC with tournament prediction: 9
  - Max IPC with perfect prediction: 35
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.

Figure 2: The Loop Detector logic

Figure 3: The Indirect Branch Predictor logic


- **Micro-Ops Fusion** - fuses operations together to enable faster execution of instructions at lower power
- **Advanced Branch Prediction** - fewer re-dos for increased performance
- **1MB Power Optimized L2 Cache** - enables higher CPU performance
- **77 Million Transistors**
- **Streaming SIMD Extensions II** - compatible with Pentium® 4 Processor optimized software
- **Dedicated Stack Management** - faster instruction at lower power levels
- **Enhanced Intel SpeedStep® Technology** - Multiple voltages & frequency operating points
- **400 MHz Power Optimized System Bus** - faster system bus to enhance performance at lower power levels

[Click here for more information](https://www.anandtech.com/show/1083/3)
More Advanced Branch Prediction
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
    - Represents how much the bit is correlated with the direction of the branch
    - Positive correlation: large positive $+$ weight
    - Negative correlation: large negative $-$ 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:

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

Output compared to 0

Bias weight
(bias of branch, independent of the history)

Training function:

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

- **Advantages**
  - + More sophisticated learning mechanism → better accuracy
  - + Enables long branch history lengths → better accuracy

- **Disadvantages**
  -- Complexity (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

A successful example of use of machine learning in processor design

Evolution of the Samsung Exynos CPU Microarchitecture

Industrial Product


bgrayson@ieee.org, jrupley@austin.rr.com, gzuraskijr@gmail.com, eric.quinell@gmail.com, djimenez@acm.org, Tarun.Nakra@amd.com, pkitchin@gmail.com, ryan.hensley@ieee.org, nedbrek@gmail.com, sinhavk@gmail.com, and mascot26@gmail.com

*SiFive †Centaur ‡Independent Consultant §ARM ¶Texas A&M University ‡‡AMD **Nuvia ‡‡Goodix

Abstract—The Samsung Exynos family of cores are high-performance “big” processors developed at the Samsung Austin Research & Design Center (SARC) starting in late 2011. This paper discusses selected aspects of the microarchitecture of these cores - specifically perceptron-based branch prediction, Spectre v2 security enhancements, micro-operation cache algorithms, prefetcher advancements, and memory latency optimizations. Each micro-architecture item evolved over time, both as part of continuous yearly improvement, and in reaction to changing mobile workloads.

Index Terms—microprocessor, superscalar, branch prediction, prefetching

- Deep technical details within the microarchitecture.

The remainder of this paper discusses several aspects of front-end microarchitecture (including branch prediction microarchitecture, security mitigations, and instruction supply) as well as details on the memory subsystem, in particular with regards to prefetching and DRAM latency optimization. The overall generational impact of these and other changes is presented in a cross-workload view of IPC.

II. METHODOLOGY
**AMD Piledriver/Zen/Zen2 (2012-Present)**

- Employ a perceptron branch predictor

---

[Diagram: Fetch]

- Improved branch prediction
- New TAGE branch predictor
- Larger BTBs
  - L0 BTB, 16 entries
  - L1 BTB, 512 entries (up from 256)
  - L2 BTB, 7K entries (up from 4K)
- Larger 1K indirect target array (ITA)
- ~30% lower mispredict rate target
- Optimized 32KB, 8-way L1I cache
  - Higher associativity
  - Improved prefetching
  - Improved utilization

Another Idea: TAGE
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

Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

TAGE Branch Predictor

- **Advantages**
  - Chooses the “best” history length to predict each branch \(\rightarrow\) better accuracy
  - Enables long branch history lengths \(\rightarrow\) better accuracy

- **Disadvantages**
  - Hardware (design) complexity is not low
  - Need to choose good hash functions and table sizes to maximize accuracy and minimize latency

A successful recent idea that is used in many modern processor designs
**AMD Zen2 (2019)**

- A multi-level branch predictor
  - Perceptron (L1)
  - TAGE (L2)

---

**FETCH**

- Improved branch prediction
- New TAGE branch predictor
- Larger BTBs
  - L0 BTB, 16 entries
  - L1 BTB, 512 entries (up from 256)
  - L2 BTB, 7K entries (up from 4K)
- Larger 1K indirect target array (ITA)
- ~30% lower mispredict rate target
- Optimized 32KB, 8-way L1I cache
  - Higher associativity
  - Improved prefetching
  - Improved utilization

[Diagram showing branch prediction model](https://fuse.wikichip.org/news/2458/a-look-at-the-amd-zen-2-core/)
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.
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/table 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 predicated execution
    - ...

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

Other Ways of Handling Branches
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)
Delayed Branching (I)

- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles

- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.

- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot
Delayed Branching (II)

Normal code:  

Timeline:  

Delayed branch code:  

Timeline:  

Normal code:

A
B
C
BC X
D
E
F
G

Timeline:

if
ex

A
B
A
C
B
BC
C
--
BC
G
--

6 cycles

Delayed branch code:

A
C
BC X
B
D
E
BC
C
F

Timeline:

if
ex

A
C
A
BC
C
B
BC
G
B

5 cycles
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC ISA
  - Semantics: If the branch falls through (i.e., it is not taken), the delay slot instruction is not executed
  - Why could this help?

Normal code:

![Normal code diagram]

Delayed branch code:

![Delayed branch code diagram]

Delayed branch w/ squashing:

![Delayed branch w/ squashing diagram]
Delayed Branching (IV)

- **Advantages:**
  + Keeps the pipeline full with useful instructions in a simple way assuming
    1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves
    2. All delay slots can be filled with useful instructions

- **Disadvantages:**
  -- Not easy to fill the delay slots (even with a 2-stage pipeline)
    1. Number of delay slots increases with pipeline depth, superscalar execution width
    2. Number of delay slots should be variable with variable latency operations. Why?
  -- Ties ISA semantics to hardware implementation
    -- SPARC, MIPS, HP-PA: 1 delay slot
    -- What if pipeline implementation changes with the next design?
An Aside: Filling the Delay Slot

reordering independent instructions does not change program semantics

within same basic block For correctness: add a new instruction to the not-taken path? Safe?

For correctness: add a new instruction to the taken path?

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Lecture on Static Instruction Scheduling
Lectures on Static Instruction Scheduling

- Computer Architecture, Spring 2015, Lecture 16
  - Static Instruction Scheduling (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=isBEVkJgGA&list=PL5PHm2jkkXmi5CxxI7b3JCtKq&index=18

- Computer Architecture, Spring 2013, Lecture 21
  - Static Instruction Scheduling (CMU, Spring 2013)
  - https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=21
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)
Other Branch Solutions Covered in This Lecture…

https://www.youtube.com/onurmutlulectures
Other Branch Solutions Covered in This Lecture…

- Computer Architecture, Spring 2015, Lecture 5
  - Advanced Branch Prediction (CMU, Spring 2015)
    - [https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4](https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4)

- See Backup Slides
Digital Design & Computer Arch.

Lecture 18: Branch Prediction II

Prof. Onur Mutlu

ETH Zürich
Spring 2022
5 May 2022
Backup Slides

(These are for Your Further Study)
Other Branch Solutions Covered in This Lecture…

Superblock Code Optimization Example

Original Code

Code After Superblock Formation

Code After Common Subexpression Elimination

18-740 Computer Architecture - Advanced Branch Prediction - Lecture 5
4,696 views - Sep 23, 2015

Carnegie Mellon Computer Architecture
25K subscribers

Lecture 5: Advanced Branch Prediction
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu/)
Date: September 16, 2014.

Lecture 5 slides (pdf): http://www.ece.cmu.edu/~ece740/f15/ii...
Lecture 5 slides (ppt): http://www.ece.cmu.edu/~ece740/f15/ii...

https://www.youtube.com/onurmutlulectures
Other Branch Solutions Covered in This Lecture…

- **Computer Architecture, Spring 2015, Lecture 5**
  - Advanced Branch Prediction (CMU, Spring 2015)
  - [https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4](https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4)

- **See Backup Slides**

[SAFARI](https://www.youtube.com/onurmutlulectures)
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)
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 instead of having one branch for each
  - 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:** Convert control dependence to data dependence

- **Simple example:** 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**
  
  ```c
  if (a == 5) {b = 4;} else {b = 3;}
  ```

  ```c
  CMPEQ condition, a, 5;
  CMOV condition, b ← 4;
  CMOV !condition, b ← 3;
  ```
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 become NOPs)

(normal branch code)  (predicated code)

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

```
if (cond) {
    p1 = (cond)
    branch p1, TARGET
}
```

```
if (!p1)
    mov b, 1
    jmp JOIN
```

```
TARGET:
    mov b, 0
```

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

```
    (p1) mov b, 0
```

```
    add x, b, 1
```
Predicated Execution References


Conditional Move Operations

- Very limited form of predicated execution

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

- Eliminates branches $\rightarrow$ enables straight line code (i.e., larger basic blocks in code)

**Advantages**
- Eliminates hard-to-predict branches
- Always-not-taken prediction works better (no branches)
- Compiler has more freedom to optimize code (no branches)
  - control flow does not hinder inst. reordering optimizations
  - code optimizations hindered only by data dependencies

**Disadvantages**
- Useless work: some instructions fetched/executed but discarded (especially bad for easy-to-predict branches)
- Requires additional ISA (and hardware) support
- Can we eliminate all branches this way?
Predicated Execution vs. Branch Prediction

+ Eliminates mispredictions for hard-to-predict branches
  + No need for branch prediction for some branches
  + Good if misprediction cost > useless work due to predication

-- 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.
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 could include an optional condition code
  - Prior to ARM v8

- An instruction with a condition code is executed **only if**
  - the condition code flags in the CPSR (Current Program Status Register) meet the specified condition
### Conditional Execution in ARM ISA

<table>
<thead>
<tr>
<th>Cond</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
<td>00000000</td>
<td>A</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>000010</td>
<td>B</td>
<td>0 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>00010</td>
<td>0</td>
<td>0 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>1 0 0 0 0 0 1</td>
<td>I</td>
<td>P</td>
<td>U</td>
<td>B</td>
<td>W</td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>000010</td>
<td>0</td>
<td>0 0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>0 0 0 0 1</td>
<td>000010</td>
<td>U</td>
<td>W</td>
<td>L</td>
<td>Rn</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>0 0 1 0 0 1 0</td>
<td>1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>000010</td>
<td>P</td>
<td>U</td>
<td>N</td>
<td>W</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 0 0 1</td>
<td>Op1</td>
<td>CRn</td>
<td>CRd</td>
<td>CPNum</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 0 0 1</td>
<td>Op1</td>
<td>L</td>
<td>CRn</td>
<td>Rd</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>000010</td>
<td>Op1</td>
<td>1 1 1 1</td>
<td>SWI Number</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction type
- Data processing / PSR Transfer
- Multiply
- Long Multiply (v3M / v4 only)
- Swap
- Load/Store Byte/Word
- Load/Store Multiple
- Halfword transfer : Immediate offset (v4 only)
- Halfword transfer: Register offset (v4 only)
- Branch
- Branch Exchange (v4T only)
- Coprocessor data transfer
- Coprocessor data operation
- Coprocessor register transfer
- Software interrupt
Conditional Execution in ARM ISA

<table>
<thead>
<tr>
<th>Cond</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ - Z set (equal)</td>
</tr>
<tr>
<td>0001</td>
<td>NE - Z clear (not equal)</td>
</tr>
<tr>
<td>0010</td>
<td>HS / CS - C set (unsigned higher or same)</td>
</tr>
<tr>
<td>0011</td>
<td>LO / CC - C clear (unsigned lower)</td>
</tr>
<tr>
<td>0100</td>
<td>MI - N set (negative)</td>
</tr>
<tr>
<td>0101</td>
<td>PL - N clear (positive or zero)</td>
</tr>
<tr>
<td>0110</td>
<td>VS - V set (overflow)</td>
</tr>
<tr>
<td>0111</td>
<td>VC - V clear (no overflow)</td>
</tr>
<tr>
<td>1000</td>
<td>HI - C set and Z clear (unsigned higher)</td>
</tr>
<tr>
<td>1001</td>
<td>LS - C clear or Z (set unsigned lower or same)</td>
</tr>
<tr>
<td>1010</td>
<td>GE - N set and V set, or N clear and V clear (&gt; or =)</td>
</tr>
<tr>
<td>1011</td>
<td>LT - N set and V clear, or N clear and V set (&gt; )</td>
</tr>
<tr>
<td>1100</td>
<td>GT - Z clear, and either N set and V set, or N clear and V set (&gt;)</td>
</tr>
<tr>
<td>1101</td>
<td>LE - Z set, or N set and V clear, or N clear and V set (&lt;, or =)</td>
</tr>
<tr>
<td>1110</td>
<td>AL - always</td>
</tr>
<tr>
<td>1111</td>
<td>NV - reserved.</td>
</tr>
</tbody>
</table>
Conditional Execution in ARM ISA

* 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 (ADDA)`
  
  - 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?
```
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 execution:
- Path 1: A → C → D → E → F
- Path 2: A → B → D → E → F

Predicated execution:
- Path 1: A → C → CFMerge
- Path 2: A → B → CFMerge
Handling Other Types of Branches
### 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>

**How can we predict an indirect branch with many target addresses?**
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 $\rightarrow$ > 95% accuracy
Indirect Branch Prediction (I)

- Register-indirect branches have multiple targets

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

Conditional (Direct) Branch

Indirect Jump

\[ \text{R1} = \text{MEM[R2]} \]

\[ \text{branch R1} \]
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
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.

![Diagram](image1)

**Figure 2: The Loop Detector logic**

![Diagram](image2)

**Figure 3: The Indirect Branch Predictor logic**

Gochman et al.,
“The Intel Pentium M Processor: Microarchitecture and Performance,”
Issues in Branch Prediction (I)

- Need to identify a branch before it is fetched

How do we do this?
- BTB hit $\rightarrow$ indicates that the fetched instruction is a branch
- BTB entry contains the “type” of the branch
- Pre-decoded “branch type” information stored in the instruction cache identifies type of branch

What if no BTB?
- Bubble in the pipeline until target address is computed
- E.g., IBM POWER4
**Latency of Branch Prediction**

- **Latency:** Prediction is latency critical
  - Need to generate next fetch address for the next cycle
  - Bigger, more complex predictors are more accurate but slower
Issues in Fast & Wide Fetch Engines
These Issues Covered in This Lecture…

Superblock Code Optimization Example

Original Code

Code After Superblock Formation

Code After Common Subexpression Elimination

18-740 Computer Architecture - Advanced Branch Prediction - Lecture 5
4,696 views • Sep 23, 2015

Carnegie Mellon Computer Architecture
25K subscribers

Lecture 5: Advanced Branch Prediction
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu/)
Date: September 16, 2014.

Lecture 5 slides (pdf): http://www.ece.cmu.edu/~ece740/f15/ii...
Lecture 5 slides (ppt): http://www.ece.cmu.edu/~ece740/f15/ii...

https://www.youtube.com/onurmutlulectures
These Issues Covered in This Lecture…

- Computer Architecture, Spring 2015, Lecture 5
  - Advanced Branch Prediction (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4

https://www.youtube.com/onurmutlulectures
Interference in Branch Predictors
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:
    - \( P(b1 \text{ T}, b2 \text{ NT}) + P(b1 \text{ NT}, b2 \text{ T}) \)
    - \( = (85\% \times 85\%) + (15\% \times 15\%) = 74.5\% \)
  - Agree predictor:
    - Assume bias bits are set to T (b1) and NT (b2)
    - \( P(b1 \text{ agree}, b2 \text{ disagree}) + P(b1 \text{ disagree}, b2 \text{ agree}) \)
    - \( = (85\% \times 15\%) + (15\% \times 85\%) = 25.5\% \)

- 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 to mispredict them again

■ Alpha EV8 (21464) branch predictor
Another Direction: Helper Threading

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

Issues in Wide & Fast Fetch
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: Get the predicted 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.
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”? 

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 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

**Code After Superblock Formation**
- opA: mul r1<-r2,3
- opB: add r2<-r2,1
- opC: mov r3<-r1

**Code After Common Subexpression Elimination**
- opA: mul r1<-r2,3
- opB: add r2<-r2,1
- 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 branch outcomes of control transfer instructions.
- Traces repeat: programs have frequently executed paths.
- Trace cache idea: Store the dynamic instruction sequence in the same physical location.

(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)

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

Trace Cache: Example

- Fetch Address A
- Instruction Cache
  - 1st BB
  - 2nd BB
  - 3rd BB

- Trace Cache
  - 1st BB
  - 2nd BB
  - 3rd BB

- Line-Fill Buffer

- Instruction Latch
  - 0
  - 1

- To Instruction Buffers

hit?
Take output from trace cache if trace cache hit; otherwise, take output from instruction cache.
An Example Trace Cache Based Processor

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) → 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?
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