Required Readings

- Last week & This week
  - H&H Chapters 7.8 and 7.9
Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and array processors, GPUs)
Control Dependence Handling
Recall: 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 = $500/900$
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
    
    TTTTTTTTTTTTNNNNNNNNNNN → 90% accuracy

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

+ Loop branches for loops with large N (number of iterations)
-- Loop branches for loops will small N (number of iterations)

  TNTNTNTNTNTNTNTNTNTNTNTNTN → 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 taken

actually not 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 (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
  - **!taken** 10
  - **!taken** 01
  - **!taken** 00

- **pred**
  - **taken** 10
  - **!taken** 00
  - **!taken** 00

Arrows indicate transitions between states based on the actual taken or !taken conditions.
Hysteresis Using a 2-bit Counter

- "strongly taken"
- "weakly taken"
- actually taken
- actually !taken
- Change prediction after 2 consecutive mistakes
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 = \( \frac{N-1}{N} \)
  TNTNTNTNTNTNTNTNTN \( \Rightarrow \) 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 (1)

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

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

- If first branch not taken, second also not taken

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

- 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

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

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

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

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

Global history/branch predictor

Uses two levels of history (GHR + history at that GHR)

Two Level Global Branch Prediction

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

How Does the Global Predictor Work?

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
Aside: Global Branch Correlation Analysis

- 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

Next Fetch Address

Cache of Target Addresses (BTB: Branch Target Buffer)

PC + inst size
hit?
taken?
Two-Level Global History 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?

target address

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
Two-Level Gshare Branch Predictor

Direction predictor (2-bit counters)

Global branch history

Which direction earlier branches went

Program Counter

Address of the current instruction

Next Fetch Address

Cache of Target Addresses (BTB: Branch Target Buffer)

XOR

taken?

PC + inst size

hit?

target address
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

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

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

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.

Loop closing branch’s history

```
1110 1110 1101 1110 0111 0
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1100
1101
1110
1111
```

PHT
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

Pattern History Table (PHT)

Local history registers

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)

PC + inst size

taken?

hit?

target address

Next Fetch Address
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 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

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

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, 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**

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
- 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:

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

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

Output compared to 0

Training function:

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

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

A successful example of use of machine learning in processor design

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 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 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
State of the Art in Branch Prediction

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

Andre Seznec,
“TAGE-SC-L branch predictors,”
CBP 2014.

Andre Seznec,
“TAGE-SC-L branch predictors again,”
CBP 2016.

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 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:

6 cycles

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:  
Delayed branch code:  
Delayed branch w/ squashing:

```
X:  
  A  
  B  
  C  
  BC X  
  D  
  E  
```

```
X:  
  A  
  B  
  C  
  BC X  
  NOP  
  D  
  E  
```

```
X:  
  A  
  B  
  C  
  BC X  
  D  
  E  
```
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 data independent (RAW, WAW, WAR) instructions does not change program semantics.

For correctness: add a new instruction to the not-taken path?

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

Safe?

Lecture 16 slides (pdf): http://www.ece.cmu.edu/~ece447/s15/ll...
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=PL5PHm2jkkXmi5CxxI7b3JC_L1TWybTDtKq&index=18](https://www.youtube.com/watch?v=isBEVkJgGA&list=PL5PHm2jkkXmi5CxxI7b3JC_L1TWybTDtKq&index=18)

- **Computer Architecture, Spring 2013, Lecture 21**
  - Static Instruction Scheduling (CMU, Spring 2013)
  - [https://www.youtube.com/watch?v=XdDUcWtkRg&list=PL5PHm2jkkXmidJ0d59REog9jDnPDTG6I&index=21](https://www.youtube.com/watch?v=XdDUcWtkRg&list=PL5PHm2jkkXmidJ0d59REog9jDnPDTG6I&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)
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**
  if (a == 5) {b = 4;} else {b = 3;}

  CMPEQ condition, a, 5;
  CMOV condition, b ← 4;
  CMOV !condition, b ← 3;
**Predication (Predicated Execution)**

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

(normal branch code) $\quad$ (predicated code)

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

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

```
eg p1
```

```
(p1)
```

```
add x, b, 1
```

```
add x, b, 1
```
Allen et al., “Conversion of control dependence to data dependence,” POPL 1983.


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

- Eliminates branches → 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.
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

#### 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

---

<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>0 0 0 0 0 0</td>
<td>A S</td>
<td>Rd</td>
<td>Rn</td>
<td>Rs 1 0 0 1</td>
</tr>
<tr>
<td>0 0 0 1 1</td>
<td>0 0 0 1</td>
<td>U A S</td>
<td>RdHi</td>
<td>RdLo</td>
<td>Rs 1 0 0 1</td>
</tr>
<tr>
<td>0 0 0 1 0</td>
<td>0 0 0</td>
<td>B 0 0</td>
<td>Rn</td>
<td>Rd</td>
<td>0 0 0 0 1 0 0 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>P U B W</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset</td>
</tr>
<tr>
<td>1 0 0</td>
<td>P U S W</td>
<td>L</td>
<td>Rn</td>
<td>Register List</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>P U L W</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset 1 1 S H 1</td>
</tr>
<tr>
<td>0 0 0</td>
<td>P U O W</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>0 0 0 0 1 S H 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>L</td>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1 0 1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1 1 0 0 1</td>
<td>Rn</td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>P U N W</td>
<td>L</td>
<td>Rn</td>
<td>CRd</td>
<td>CPNum</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>Op1</td>
<td>CRn</td>
<td>CRd</td>
<td>CPNum</td>
<td>Op2</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>Op1</td>
<td>L</td>
<td>CRn</td>
<td>Rd</td>
<td>CPNum</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>SWI Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conditional Execution in ARM ISA

0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = HS / CS - C set (unsigned higher or same)
0011 = LO / CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (>=)
1011 = LT - N set and V clear, or N clear and V set (>)
1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)
1101 = LE - Z set, or N set and V clear, or N clear and V set (<=)
1110 = AL - always
1111 = NV - reserved.
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.

---

The ARM Instruction Set - ARM University Program - V1.0
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

Predicated Execution

path 1

path 2

CFMerge

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 → > 95% accuracy
Indirect Branch Prediction (I)

- Register-indirect branches have multiple targets

```
<p>| A |
| T |</p>
<table>
<thead>
<tr>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>TARG</td>
</tr>
</tbody>
</table>
```

br.cond TARGET

```
  | A |
    | ? |
  | α |
  | β |
  | δ |
  | ρ |
```

R1 = MEM[R2] branch R1

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

Issues in Branch Prediction (I)

- Need to identify a branch before it is fetched

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

```
PC + inst size
BTB target
Return Address Stack target
Indirect Branch Predictor target
Resolved target from Backend

???

Next Fetch Address
```
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and array processors, GPUs)
VLIW
VLIW Concept

- Superscalar
  - **Hardware** fetches multiple instructions and checks dependencies between them

- VLIW (Very Long Instruction Word)
  - **Software (compiler) packs independent instructions** in a larger “instruction bundle” to be fetched and executed concurrently
  - Hardware fetches and executes the instructions in the bundle concurrently

- **No need for hardware dependency checking** between concurrently-fetched instructions in the VLIW model

- ELI: Enormously longword instructions (512 bits)
A very long instruction word consists of multiple independent instructions packed together by the compiler. Packed instructions can be logically unrelated (contrast with SIMD/vector processors, which we will see soon).

Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction.

Traditional Characteristics
- Multiple functional units
- All instructions in a bundle are executed in lock step
- Instructions in a bundle statically aligned to be directly fed into the functional units
VLIW Performance Example (2-wide bundles)

Ideal IPC = 2

Actual IPC = 2 (6 instructions issued in 3 cycles)
VLIW Lock-Step Execution

- Lock-step (all or none) execution
  - If any operation in a VLIW instruction stalls, all concurrent operations stall

- In a truly VLIW machine:
  - the compiler handles all dependency-related stalls
  - hardware does not perform dependency checking
  - What about variable latency operations? Memory stalls?
VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel

- **RISC** (John Cocke, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple

- **VLIW** (Josh Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design
Commercial VLIW Machines

- **Multiflow TRACE**, Josh Fisher (7-wide, 28-wide)
- **Cydrome Cydra 5**, Bob Rau
- **Transmeta Crusoe**: x86 binary-translated into internal VLIW
- **TI C6000, Trimedia, STMicro (DSP & embedded processors)** and some AMD GPUs
  - Most successful commercially

- **Intel IA-64**
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones
VLIW Tradeoffs

Advantages

+ No need for dynamic scheduling hardware \(\rightarrow\) simple hardware
+ No need for dependency checking within a VLIW instruction \(\rightarrow\) simple hardware for multiple instruction issue + no renaming
+ No need for instruction alignment/distribution after fetch to different functional units \(\rightarrow\) simple hardware

Disadvantages

-- Compiler needs to find N independent operations per cycle
  -- If it cannot, inserts NOPs in a VLIW instruction
  -- Parallelism loss AND code size increase
-- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
-- Lockstep execution causes independent operations to stall
  -- No instruction can progress until the longest-latency instruction completes
VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance
  -- Too many NOPs (not enough parallelism discovered)
  -- Static schedule intimately tied to microarchitecture
    -- Code optimized for one generation performs poorly for next
  -- No tolerance for variable or long-latency operations (lock step)

++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
  - Enable code optimizations
++ VLIW very successful when parallelism is easier to find by the compiler (traditionally embedded markets, DSPs, GPUs)
An Example Work: Superblock

The Superblock: An Effective Technique
for VLIW and Superscalar Compilation

Wen-mei W. Hwu       Scott A. Mahlke     William Y. Chen     Pohua P. Chang
Nancy J. Warter       Roger A. Bringmann   Roland G. Ouellette  Richard E. Hank
Tokuzo Kiyohara      Grant E. Haab        John G. Holm         Daniel M. Lavery *


- Lecture Video on Static Instruction Scheduling
  - https://www.youtube.com/watch?v=isBEVkJgGA
Another Example Work: IMPACT

IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors

Pohua P. Chang  Scott A. Mahlke  William Y. Chen  Nancy J. Warter  Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

The performance of multiple-instruction-issue processors can be severely limited by the compiler’s ability to generate efficient code for concurrent hardware. In the IMPACT project, we have developed IMPACT-I, a highly optimizing C compiler to exploit instruction level concurrency. The optimization capabilities of the IMPACT-I C compiler are summarized in this paper. Using the IMPACT-I C compiler, we ran experiments to analyze the performance of multiple-instruction-issue processors executing some important non-numerical programs. The multiple-instruction-issue processors achieve solid speedup over high-performance single-instruction-issue processors.
Lecture on Static Instruction Scheduling


7,136 views • Feb 26, 2015

Carnegie Mellon Computer Architecture
23K subscribers

Lecture 16: Static Instruction Scheduling
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu/)
Date: Feb 23rd, 2015

Lecture 16 slides (pdf): http://www.ece.cmu.edu/~ece447/s15/ll...

https://www.youtube.com/onurmutlulectures
Lectures on Static Instruction Scheduling

- **Computer Architecture, Spring 2015, Lecture 16**
  - Static Instruction Scheduling (CMU, Spring 2015)
  - [https://www.youtube.com/watch?v=isBEVkIjgGA&list=PL5PHm2jkkXmi5CxxI7b3JC LiTWybTDtKq&index=18](https://www.youtube.com/watch?v=isBEVkIjgGA&list=PL5PHm2jkkXmi5CxxI7b3JC LiTWybTDtKq&index=18)

- **Computer Architecture, Spring 2013, Lecture 21**
  - Static Instruction Scheduling (CMU, Spring 2013)
  - [https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59RE og9jDnPDTG6IJ&index=21](https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59RE og9jDnPDTG6IJ&index=21)
Backup Slides
(for Further Study)
Issues in Fast & Wide Fetch Engines
These Issues Covered in This Lecture…

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/l1...
Lecture 5 slides (ppt): http://www.ece.cmu.edu/~ece740/f15/l1...

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:
    - \( P(b1\ T,\ b2\ NT) + P(b1\ NT,\ b2\ T) \)
    - \( = (85\%*85\%) + (15\%*15\%) = 74.5\% \)
  - Agree predictor:
    - Assume bias bits are set to T (b1) and NT (b2)
    - \( P(b1\ agree,\ b2\ disagree) + P(b1\ disagree,\ b2\ agree) \)
    - \( = (85\%*15\%) + (15\%*85\%) = 25.5\% \)

- 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)


---

**Figure 3. The Microthread Builder**
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
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 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

Block 0100

Block 0101

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

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)

![Dynamic Instruction Stream](image)

- 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

The diagram illustrates the operation of a trace cache. The process starts with a fetch address A, which is directed to the trace cache. The trace cache contains three blocks labeled 1st BB, 2nd BB, and 3rd BB. If the fetch address A matches any of these blocks, a hit occurs, and the corresponding block is output. If there is no match, the next step is to check if a hit in the instruction cache is present. If there is a hit, the output is taken from the trace cache; otherwise, it is taken from the instruction cache. The output is then directed to the instruction latch and finally to the instruction buffers.
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