CE 431
Parallel Computer Architecture
Spring 2019

Branches
Nikos Bellas

Electrical and Computer Engineering Department
University of Thessaly
Control Hazards

The 5-stage microarchitecture executes `beq/bne` instructions in EX stage.

BUT:
Three instructions from the wrong path have already made it to the pipeline if the `beq` instruction is TAKEN.
Stalls due to Mispredicted Branches

- We can delay the pipeline by three cycles each time we decode a branch (delay till branch execution)

- Wasting three cycles for each branch is quite a lot
Performance Analysis

- NOT TAKEN (NT) \(\Rightarrow\) No penalty
- TAKEN (T) \(\Rightarrow\) 3 cycles penalty
- 20\% of all instructions are branches
- 50\% of branches are TAKEN
- Base CPI = 1
  - CPI = \( 1 + (0.20 \times 0.50) \times 3 \) =
  - = \( 1 + 0.1 \times 3 \) = 1.3 (30\% slow down)

Need to reduce both these terms
The Branch Problem

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

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

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

• Assume \( N = 20 \) stages, \( W = 5 \) (5 wide fetch)
  – Assume: 1 out of 5 instructions is a branch (each 5 instruction-block ends with a branch)

• How long does it take to fetch 500 useful instructions (i.e. from the correct path)?
  – 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
Branches in real programs

```c
if ((*ptr1==0) && ((*ptr2=*ptr3)==1) && (*ptr4>2))
    val5++;
else
    val6++;```

```assembly
lw $t0, ptr1($0)
B1:
bnez $t0, L_val6
lw $t1, ptr3($0)
sw $t1, ptr2($0)
B2:
bne $t1, $t8, L_val6
lw $t1, ptr4($0)
B3:
ble $t1, $t9, L_val6
lw $s0, val5($0)
addi $s0, $s0, 1
sw $s0, val5($0)
j Exit
L_val6:
lw $s0, val6($0)
addi $s0, $s0, 1
sw $s0, val6($0)
Exit:
```
Branch prediction

• Pentium 4 (2003) has 20 pipe stages
• Branches are executed in stage 15
• There can be tens of instructions somewhere in the pipeline at any time
• Multiple branch instructions at the same time
• Need to predict correctly all of them
  – Otherwise, pipeline flush and large performance penalties
1. Reduce Misprediction Penalty

• Branch execution (\texttt{beq rs, rt, Label}) requires two operations:
  1. Decide if the branch is TAKEN or NOT TAKEN
  2. Compute target address: PC <= Label or PC <= PC+4

• Both these need to be executed earlier to reduce misprediction penalty
  • Transfer (2) from the EX to the ID stage (this is easy)
  • Need extra hardware in the ID stage to check if the two registers \texttt{rs} and \texttt{rt} are equal (1)
  • Can do that with 32 XOR gates. But it may increase the clock period
New pipeline with reduced penalty

What if we have the following instr.
sub  $2, $1, $3
beq  $4, $2, L

More complex forwarding and an extra stall cycle
2. Reduce Misprediction Penalty (Delayed Branches)

- **Idea**: let us try to fetch $N$ useful instructions (from the fall through path) before branch is executed
  - Initial architecture $N=3$
  - New architecture $N=1$

- The $N$ instructions after the branch are executed as usual
  - No pipeline flush

- This technique is called *Delayed Branch*

- $N$ should be small and is architecture dependent
Delayed Branch

Compiler should be able to detect N useful instructions and place them after the branch. Assume N=1 in the following examples.

This is not always possible. If not possible, just put nops

Typically N should be small (e.g. N=1)

Easy

May need code restructure by the compiler

May need code restructure by the compiler
3. Reduce Probability of Wrong Path
Branch prediction

- Try to predict the next instruction to fetch (predict the outcome of the branch)
  - Easiest approach: always predict NOT TAKEN
  - Increment PC <= PC+4 and fetch next instruction from memory
- If we are correct, there is no branch penalty
Branch prediction

• But if we are wrong (Branch NOT TAKEN), we have fetched an instruction \( I \) from the wrong path.
  1. Need to flush that instruction \( I \) from the pipeline and place a \texttt{nop} to register IF/ID
  2. Need to change PC so that we immediately start fetching from the Label.

• We pay \( N \) cycles penalty in case of misprediction

\[ \text{beq } \$2, \$3, \text{Label} \]

\[ \text{next instruction 1} \]

\[ \text{Label: } \ldots \]
Static branch prediction

- Each branch is predicted statically, i.e. at compile time. This prediction (T/NT) never changes at execution time.

- Methods for static branch prediction:
  1. All branched are predicted as NOT TAKEN
  2. Backward taken, forward not taken (BTFN):
     - All forward branches are predicted as NOT TAKEN
     - All backward branches as TAKEN
  3. Profile-driven
     - Use profiling data from previous code runs to understand the behavior of branches. Assign prediction semantics to the branches according to that,
  4. User-specified branch prediction:
     - if (likely(x)) { ... }
     - if (unlikely(error)) { ... }
Dynamic branch prediction

- In modern superscalar and superpipeline CPUs with a large number of stages (~20), the penalty in case of misprediction is higher
  - High Branch misprediction can severely slow down such a CPU
- Dynamic branch prediction attempts to predict branch direction from past program behavior
Dynamic branch prediction

• **Rule 1:** The execution of a branch (T / NT) is related on previous executions of that branch
  – Local Branch correlation
Dynamic branch prediction
1-bit prediction scheme

- The simplest dynamic prediction algorithm: predict that the branch will do the same as its last execution.

```
outer: ...
   ...
inner: ...
   ...
   beq ..., ..., inner
   ...
   beq ..., ..., outer
```

- Simple but somewhat unstable (changes prediction “too easily”)
- `beq` of the inner loop θα will be mispredicted twice for each execution of the outer loop.
1-bit prediction scheme

- Predict taken
  - Taken
  - Not taken
- Predict not taken
  - Not taken
  - Taken
2-bit prediction scheme

- A 4-state FSM is used to implement the 2-bit prediction scheme.
- A prediction will have to miss twice before it is changed.
- It makes predictions more "stable" by adding hysteresis.
- 2-bit (or sometimes 3-bit descriptors) are the basis of more advanced predictors used in modern CPUs.
Dynamic branch prediction

• **Rule 1:** The execution of a branch (T / NT) is related on previous executions of that branch
  – Local Branch correlation

• **Rule 2:** The execution of a branch (T / NT) is also related to previous executions of neighboring branches.
  – Global Branch correlation
Dynamic branch prediction

Recent executions of one branch are related to the execution of a subsequent branch

if (cond1)
...
if (cond1 AND cond2)

if cond1 is False,
then the second conditional is False

branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)

If branch Y is True,
then branch X is False
Dynamic branch prediction

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

if (B1 == True) && (B2 == True) then B3 = False

NT NT \rightarrow T
Correlating branch prediction

• Use both Local and Global Branch Predictions to improve prediction rate
  – Correlating branch predictors
• Make a prediction based on the outcome of the branch the last time the same global branch history was encountered
• Use Global history/branch predictor
  – Use an m-bit Global History Table (GHR) to capture the history of the most recent m branches
• Uses two levels of history (GHR + history at that GHR for the specific branch)
(m, n) branch predictor

- m = #bits of GHR
- n = #bits of for the predictor of each branch (n-bit counter)

Size of predictor =

\[ 2^m \times n \times \text{Number of prediction entries selected by branch address} \]

(0,2) : predictor with no global history and 2-bit counters

GHR: m-bit recent global branch history
Gshare Predictors

- Gshare predictors is a type of \((m, n)\) predictor in which GHR is combined with the Branch PC (using a hash function).

- Pattern History Table (PHT)
  - Contains \(n\)-bit saturating counters for prediction.
  - Indexed by a Hash function which can be as simple as an XOR.
Can We Do Even Better?

- Predictability of branches varies

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

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

- Idea: Exploit that heterogeneity by designing heterogeneous branch predictors
Tournament Branch Predictors

• **Idea:** Use more than one type of predictor (i.e., multiple algorithms) and select dynamically the “best” prediction

• Tournament predictors combine Local and Global history adaptively

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

• **Disadvantages:**
  -- Need “meta-predictor” or “selector”
  -- Longer access latency
Prediction Using Multiple History Lengths

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

• Idea: Have multiple PHTs indexed with a GHRs/branchPC with different history lengths and intelligently allocate PHT entries to different branches

• PHT are tagged like caches

TAGE: Tagged & prediction by the longest history matching entry

Tagless base predictor

prediction
TAGE: Which Table to Use?

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

• Special case:
  – Many mispredictions on newly allocated entries: weak Ctr

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

TAGE is the basis of the state-of-art branch predictor (branch prediction championship)
Hybrid predictors performance
Branch Confidence Estimation

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

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

• An example estimator:
  – Keep a record of correct/incorrect outcomes for the past N instances of the “branch”
  – Based on the correct/incorrect patterns, guess if the current prediction will likely be correct/incorrect
What to Do With Confidence Estimation?

- An example application: Pipeline Gating
  - Throttle instruction fetching/execution when too many branches are predicted with low confidence

Branch Target Buffer (BTB)

• Require three things for branches:
  1. Predict the branch direction (we covered that already)
  2. Predict the new target address
  3. Identify that the instruction is a branch BEFORE it is decoded (IF stage)

• Branch target buffer (BTB) covers 2 and 3
  – Cache memory that stores the predicted address of the next instruction after a branch
  – Contains the predicted-taken branches (since in an untaken branch we can just fetch the next sequential address)
  – It is accessed at the same time as the I-Cache using the PC (IF stage)
Branch Target Buffer (BTB)

- If the PC of the fetched instruction matches an address in the BTB, the predicted PC becomes the new PC of the CPU.
- Instruction fetching begins immediately at that address.
Branch Target Buffer (BTB)

- **Two sources of penalties**
  - Branch predicted Taken (BTB Hit), but ends up Not Taken
  - Branch predicted Not Taken (BTB Miss), but ends up Taken
- **In each case, we need to flush the pipeline**
  - Flush all non-committed instructions in the ROB after the mispredicted branch
  - Performance penalty can be severe
Branch Target Buffer (BTB) in the 5 stage pipeline
Integration of Branch Prediction and BTB

Branch predictor

Program Counter

Address of the current instruction

BTB: Branch Target Buffer

taken?

PC + inst size

hit?

target address

Next Fetch Address
4. Speculation

- If branch prediction confidence is low, select both paths (T and NT)
  - This is widely used in high performance CPUs
  - Higher hardware/higher complexity, potentially higher energy
  - Need to be able to limit speculation at run time

![Diagram of Speculation](image)
Speculation

• How much to speculate
  – Mis-speculation degrades performance and power relative to no speculation
    • May cause additional misses (cache, TLB)
  – Need to prevent speculative code from causing higher cost misses (e.g. L2)

• Speculating through multiple branches
  – Complicates speculation recovery

• Speculation and energy efficiency
  – Note: speculation is only energy efficient when it significantly improves performance
How much to speculate

SPEC2000 benchmark suite

Fraction of instructions that are executed as result of mis-speculation is much higher for integer benchmarks.
6. Predication

- Completely get rid of branch instructions
- Convert all control hazards to data hazards
- A lot of modern ISA support predication
  - ARM
  - x86 partially
if (a!='c) {
    b = 0;
} else {
    b = 1;
}

Extra predicate operand for each instr.
If p1 is TRUE, execute instruction
If p1 is FALSE, instruction becomes NOP
ARM processor is fully predicated

- Instructions are predicated using the appropriate postfix
  - `add r0, r1, r2` ; Regular ADD: `r0 = r1 + r2`
  - `addeq r0, r1, r2` ; `r0 = r1 + r2` only if conditional flag ZERO is set

- Conditional move (CMOV), the most well known x86 predicated instruction

  - `if (A==0) {S=T}`
  
  - `p1 = (R1 == 0)`
  - `CMOV R2, R3, <p1>`
Takeaways

• Branches are the second largest culprit for CPU performance slowdown (memory instructions are the first)
  – Wrong branch prediction has large penalties
  – A lot of in-flight instructions have to flushed from the pipeline

• All modern CPUs use dynamic branch prediction and BTBs
  – Accurate branch prediction (>95%) are very important to achieve high performance and instruction throughput.
  – Branch prediction heuristics are rather complex