DEVIN MAYA 3/7/24

# CSE 120 Homework 5

**Q1 (10 pts)** Your program consists of arithmetic instructions, load/store instructions, branch instructions, and unconditional jumps in the given ratio: 50%, 20%, 20%, and 10%. For the branch instructions, assume that 60% of all branches are Taken. Assume CPI=1 for arithmetic instructions, CPI=6 for Load/Store, base CPI=1 for branch instructions, and CPI=1 for unconditional jumps. Your processor can support 3 control hazard strategies, and you are supposed to choose one for the given program:

1. Strategy 1: Unconditional stall – Stalls pipeline for 1 cycle until the branch is resolved.
2. Strategy 2: Branch prediction – Predicts Taken by default; Stall pipeline for 2 cycles if the branch is Not Taken.
3. Strategy 3: Branch prediction – Predicts Not Taken by default; Stall pipeline for 1 cycle if the branch is Taken.

Which strategy would result in the fastest runs for the given program?

## Q1 (10 pts)

## Given Data

The instruction mix and CPI values are as follows:

- Arithmetic instructions: 50% of the total, with CPI = 1
- Load/Store instructions: 20% of the total, with CPI = 6
- Branch instructions: 20% of the total, with CPI = 1 (base CPI) plus additional stalls based on the strategy
  - 60% branches are Taken, 40% are Not Taken
- Unconditional jumps: 10% of the total, with CPI = 1

## Calculating Average CPI

The average CPI for the program can be calculated using the formula:

$$\text{Average CPI} = \sum(\text{Percentage of Instructions} \times \text{CPI of Instruction Type})$$

For branch instructions, we adjust the base CPI based on the strategy and the outcome of the branch (Taken or Not Taken).

### Strategy 1: Unconditional Stall

Every branch instruction, regardless of outcome, adds 1 cycle of stall:

$$\text{CPI}_{\text{branch, S1}} = \text{CPI}_{\text{base}} + 1$$

### Strategy 2: Branch Prediction (Predicts Taken)

- Taken branches do not incur additional cycles since the prediction is correct.
- Not Taken branches incur 2 additional stall cycles:

$$\text{CPI}_{\text{branch, S2}} = 0.60 \times 1 + 0.40 \times (1 + 2) = 1.8$$

### Strategy 3: Branch Prediction (Predicts Not Taken)

- Not Taken branches do not incur additional cycles since the prediction is correct.
- Taken branches incur 1 additional stall cycle:

$$\text{CPI}_{\text{branch, S3}} = 0.40 \times 1 + 0.60 \times (1 + 1) = 1.6$$

## Final Calculation

Substituting the given percentages and calculating the overall average CPI for each strategy, we derive the explicit CPI values for each strategy.

Using a more mathematically rigorous approach, the calculations for each strategy's branch CPI and the resulting average CPI for the entire program are as follows, expressed in LaTeX notation:

### Strategy 2: Branch Prediction (Predicts Taken)

For Strategy 2, the branch prediction assumes branches are Taken by default. Therefore, for Taken branches (60%), there are no additional stall cycles since the prediction is correct, but Not Taken branches (40%) incur a 2 cycle stall because the prediction is incorrect:

$$\text{CPI}_{\text{branch, S2}} = (0.60 \times 1) + (0.40 \times (1 + 2)) = 0.60 + 1.20 = 1.8$$

### Strategy 3: Branch Prediction (Predicts Not Taken)

For Strategy 3, the branch prediction assumes branches are Not Taken by default. Thus, for Not Taken branches (40%), there are no additional stall cycles since the prediction is correct, and Taken branches (60%) incur a 1 cycle stall because the prediction is incorrect:

$$\text{CPI}_{\text{branch, S3}} = (0.40 \times 1) + (0.60 \times (1 + 1)) = 0.40 + 1.20 = 1.6$$

## Conclusion

Given the calculations:

- The average CPI for Strategy 2 is 2.16, with the CPI for branch instructions being 1.8.
- The average CPI for Strategy 3 is 2.12, with the CPI for branch instructions being 1.6.

Therefore, **Strategy 3** is the most efficient with the lowest average CPI of **2.12**

**Q2 (10 pts)** Given a program with the following repeating pattern (e.g., a loop) of branch outcomes: T, T, T, NT, NT. (T=branch taken, NT=branch not taken)

1. What is the accuracy (lec 13.pptx) of the 1-bit predictor for the given sequence of branch outcomes, provided we encounter this branch outcome sequence only once? Assume that the predictor starts off in the "Not Taken" state ("NT")? (2 pts)
2. Using the 2-bit predictor for the given sequence of branch outcomes: T, T,NT, T, NT, what is the worst choice for the starting state (choose one from SNT, WNT, WT, and ST)? (5 pts)
3. What is the overall accuracy ("score card" in lec 15-16) of the 2-bit predictor if the given pattern is repeated forever? i.e., T, T, NT, NT, NT, T, T, NT, NT, NT, .... to infinity and beyond? Assume that the predictor starts off the first iteration in the "Strongly Not Taken" state ("SNT"). Subsequent iterations continue from the branch predictor state at the end of the previous iteration. (3 pts)

## Question 2 Analysis

Given a program with a specific pattern of branch outcomes, we analyze the accuracy of 1-bit and 2-bit predictors for the given sequences.

### 1. 1-bit Predictor Accuracy

Given the initial state of "Not Taken" (NT) and the sequence of branch outcomes T, T, T, NT, NT, we calculate the accuracy of the 1-bit predictor.

**Accuracy Calculation:**

- The initial state is NT, so the first T (branch taken) is incorrectly predicted.
- After the first T, the predictor switches to T, correctly predicting the next two Ts.
- However, it incorrectly predicts the last two NTs, as the state was last set to T after the third T.

- Total correct predictions: 2 out of 5.
- Accuracy: $\frac{2}{5} = 40\%$.

## 2. Worst Starting State for 2-bit Predictor

Considering the sequence T, T, NT, T, NT and the 2-bit predictor's dynamics, we identify the worst starting state for accuracy.

**Analysis:**

- Starting in "Strongly Taken" (ST) would likely result in incorrect predictions for the NTs due to the predictor's confidence in the taken branch.
- Conversely, starting in "Strongly Not Taken" (SNT) would result in incorrect predictions for the initial Ts.
- Given the short sequence, starting in ST is less adaptable to the NTs, making it a worse starting state for this particular sequence.

**Conclusion:** The worst starting state for the given sequence is "Strongly Taken" (ST).

## 3. Overall Accuracy of 2-bit Predictor

With the infinite repeating pattern T, T, NT, NT, NT, starting from the "Strongly Not Taken" (SNT) state, we evaluate the overall accuracy of the 2-bit predictor.

**Accuracy Calculation:**

- For each 5-cycle pattern, the predictor will initially make one incorrect prediction, but will adjust correctly for the subsequent patterns, leading to 4 correct predictions out of 5.
- Thus, over an infinite number of cycles, the accuracy stabilizes at $\frac{4}{5} = 80\%$.

**Conclusion:** The overall accuracy of the 2-bit predictor, starting in the "SNT" state with the given repeating pattern, is 80%.

**Q3(10 pts)**

1. In a 16-way associative cache of size 512KB, with 16B blocks, what are the tag, index, and offset (in hexadecimal) for the memory address 0x172f3d4c? (2 pts)
2. In a fully associative 16KB cache with 256B blocks, what are the tag, index, and offset (in hexadecimal) for the memory address 0x000fc128? (2 pts)
3. For the given code snippets, predict the cache hit rate. The cache is 128KB, direct mapped, and has 64-byte cache lines. There are two implementations for the for loops (v1 and v2) which are executed separately in isolation. Assume that the:

(a) long variables total sum and array are doublewords

(b) the cache is empty before the loops start executing.

(c) If you need any other assumptions to solve this question, state them in your answer.

```
// Version v1
for(int i=0; i<256; i++) {
    for (int j=0; j<256; j++) {
        total_sum += arr[i][j];
    }
}

// Version v2
for(int i=0; i<256; i++) {
    for (int j=0; j<256; j++) {
        total_sum += arr[j][i];
    }
}
```

## Q3(10 pts)

### Cache Address Decomposition and Hit Rate Prediction

Given scenarios include cache address decomposition for a 16-way associative cache and a fully associative cache, and prediction of cache hit rates for two code versions accessing an array.

1. For the memory address `0x172f3d4c` in a 16-way associative cache of size 512KB with 16B blocks:

   - Block size = 16 bytes ⇒ Offset = 4 bits ($\log_2(16)$).
   - Total number of blocks = $\frac{512\times1024}{16}$ = 32768 blocks.
   - Number of sets = $\frac{32768}{16}$ = 2048 sets ⇒ Index = 11 bits ($\log_2(2048)$).
   - Decomposition of `0x172f3d4c`: Offset = `0xC`, Index (next 11 bits), Tag (remaining higher order bits).

2. For the memory address `0x000fc128` in a fully associative 16KB cache with 256B blocks:

   - Fully associative ⇒ Index = 0.
   - Offset = 8 bits ($\log_2(256)$).

- Tag = Remaining higher order bits after the offset.

3. Cache hit rate prediction for code snippets:

   Assumptions:

   - `long` variables (total sum and array elements) are doublewords (8 bytes).
   - The cache is empty before execution.

   **Version v1** (row-wise access) likely exhibits a higher cache hit rate due to better spatial locality.

   **Version v2** (column-wise access) may suffer from a lower cache hit rate due to poor spatial locality and direct-mapped cache limitations.

   **Conclusion: Version v1** is expected to have a higher cache hit rate compared to **Version v2**.

**Q4 (10 pts)** In lecture, we went through an example of filling in a set associative cache. However, there, we used block addresses instead of byte addresses (we deal with byte addresses in RISC-V memory references). Thus, there was no need for offset bits in this class example. The following exercise gives a more practical example of memory reference in a set-associative cache. However, as in the class example, only the tag and set field values are sufficient to verify if the memory reference exists in the cache. Suppose you have a 2-way associative cache that is 8KB, with 256B block size. Assume we implement an additional MRU bit to indicate which of the 2 blocks/ways in a set was most recently referenced. You send a sequence of memory references to the cache for read; each is a 32-bit byte-addressable memory address in hexadecimal: 0x1000, 0x100B, 0x1018, 0x11c0, 0x2200, 0x21cc, 0x2206.

1. For each of the byte-addressable memory references, fill in the corresponding set index number for lookup, the tag number, the byte offset, and whether the reference is a hit or a miss in the table below. You may enter these values in hexadecimal. Assume that the cache is empty at the start of the first memory address and is not reset for subsequent memory references. (8 pts)
2. How many times did we have to implement the LRU policy in accessing a memory reference from the cache, only for the given set of memory addresses? (2 pts)

## Q4 (10 pts)

In this exercise, we analyze a set-associative cache's behavior given a sequence of byte-addressable memory references. The cache configuration and the memory addresses are provided, and we aim to decompose each address into its corresponding set index, tag, and offset. Additionally, we assess cache hits or misses and determine the application of the LRU policy.

## Cache Analysis for Memory References

We consider a 2-way associative cache with the following specifications:

- Cache size: 8KB
- Block size: 256B
- Associativity: 2-way
- Number of sets: $8 \times 1024/(256 \times 2) = 16$

The task involves analyzing a sequence of memory references for their set index, tag, byte offset, and determining each reference as a hit or miss. An MRU bit is assumed for managing cache blocks within each set.

## Memory Reference Analysis

Each memory reference's details are to be calculated as follows:

- Set index number
- Tag number
- Byte offset
- Hit or Miss status

## Memory Reference Table

| **Memory Address** | **Set Index** | **Tag** | **Byte Offset** | **Hit/Miss** |
|---|---|---|---|---|
| 0x1000 | 0x0 | 0x1 | 0x0 | Miss |
| 0x100B | 0x0 | 0x1 | 0xB | Hit |
| 0x1018 | 0x0 | 0x1 | 0x18 | Hit |
| 0x11C0 | 0x1 | 0x1 | 0xC0 | Miss |
| 0x2200 | 0x2 | 0x2 | 0x0 | Miss |
| 0x21CC | 0x1 | 0x2 | 0xCC | Miss |
| 0x2206 | 0x2 | 0x2 | 0x6 | Hit |

## LRU Policy Implementation Count

the LRU policy was implemented **0 times** for the given set of memory addresses.