# COMP25212 Ex2 Branch Prediction

Aqib Faruqui

March 2025

## Contents

# 1 Phase 1: Profiling and Statistics

## 1.1 The 90/10 Rule

1. To explore the proportion of *instr_profile* in which the code spends 90% of its execution time, I first separated the number of times each instruction was fetched and decoded.

   ```
   awk '{print $1}' instr_profile > counts.txt
   ```

   Then I calculated the total number of instruction executions to find that the **11339** instructions were executed **3423933** times.

   ```
   total=$(awk '{sum += $1} END {print sum}' counts.txt)
   echo "Total instruction executions: $total"
   Total instruction executions: 3423933
   ```

   After using the Linux *sort* utility, I ran *src/execution_proportion.cpp* to find:

   ```
   Instructions needed to reach 90% threshold: 1941
   Proportion accounting for 90% of execution time: 17.1179%
   ```

## 1.2 Basic Blocks

2. Counting all lines that contain 'taken' returns the number of branch encounters, regardless of whether or not they were taken, so subtracting the number of lines that contain 'not taken' gives us the number of times a branch is taken.

   ```
   t=$(grep -c 'taken' block_profile)
   nt=$(grep -c 'not taken' block_profile)
   echo "Number of times a branch is taken: $((t-nt))"
   Number of times a branch is taken: 428065
   ```

3. The *block_profile* contains all non-sequential PC changes, including branches, returns and direct PC loads. Therefore, the average number of instructions between branches is given by comparing the number of instructions to the number of taken branches:

   ```
   instructions=3424177 && taken=$((t - nt))
   echo "scale=4; $instructions / $taken" | bc
   Average instructions between PC reloads: 7.9991
   ```

4. As found in question 2, there are **428065** taken branches. For each, there are two discarded cycles. We are also given that **3424177** instructions are fetched. So under the assumption that each instruction takes a single cycle, the proportion of wasted cycles gives the proportion of execution time wasted due to control hazard avoidance.

   ```
   wasted=$((428065 * 2))
   total=$((3424177 + wasted))
   echo "scale=4; 100 * $wasted / $total" | bc
   Proportion of execution time wasted: 20.0016%
   ```

## 1.3 Branch Target Buffer (BTB)

5. As mentioned, the total number of lines containing 'taken' returns the number of branch encounters. As we are not focussing on the taken/not taken status here, it is perhaps more clear to count the number of branch encounters through the line count of *block_profile*, also returning **667867** encounters. Of these encounters, **596271** are of type 'B'.

   ```
   total_branches=$(wc -l < block_profile)
   B_type_branches=$(grep -c '^B' block_profile)
   echo "scale=4; 100 * $B_type_branches / $total_branches" | bc
   Proportion of B type branches encountered: 89.2799%
   ```

6. Given a Branch Target Buffer (BTB) that correctly predicts all invariant branches, only the taken variant branches now incur a 2 cycle penalty. To find the number of taken variant branches, we can find the number of taken B type branches and subtract this from the **428065** taken branches, as found in question 2.

   ```
   not_taken_b=$(grep '^B' block_profile | grep 'not taken' | wc -l)
   total_b=$(grep -c '^B' block_profile)
   taken_b=$((total_b - not_taken_b))
   taken_variant=$((taken - taken_b))
   ```

   Of the taken branches we find that **45859** are variant, so only these require flushing the pipeline and incurring a 2 cycle penalty.

   ```
   wasted=$((taken_variant * 2))
   total=$((3424177 + wasted))
   echo "scale=4; 100 * $wasted / $total" | bc
   Overhead with accurate invariant branch prediction: 2.6086%
   ```

7. The Branch Target Buffer (BTB) is a cache structure, so naturally experiences compulsory misses for branches not yet encountered. This extends to capacity misses where the working set of branches exceeds the size of the BTB.

   Another reason for imperfect branch prediction is loop behaviour. For static compile-time predictors (e.g. backward taken, forward not taken), the loop entry and loop exit will be mispredicted in steady state. A dynamic run-time predictor (e.g. 2-bit counter) adds hysteresis to prevent a strong prediction changing with a single opposing event. However, this still fails to predict a branch not taken upon the loop exit.

8. Of the **667867** branches encountered, **428065** are taken and **239802** are not taken. So the ratio of branches taken to not taken is **1.7850:1**, or approximately **9:5** adjusted and rounded to one significant figure.

   ```
   branches=$(grep -c 'taken' block_profile)
   not_taken=$(grep -c 'not taken' block_profile)
   taken=$((branches - not_taken))
   echo "scale=4; $taken / $not_taken" | bc
   1.7850
   ```

9. Of the **512043** conditional branches encountered, **272241** are taken and **239802** are not taken. So the ratio of conditional branches taken to not taken is **1.1352:1**, or approximately **9:8** adjusted and rounded to one significant figure.

   ```
   cond=$(grep -c '?' block_profile)
   not_taken_cond=$(grep -c '? not taken' block_profile)
   taken_cond=$((cond - not_taken_cond))
   echo "scale=4; $taken_cond / $not_taken_cond" | bc
   1.1352
   ```

10. Diving deeper into the conditional branches, we find that forward branches have a ratio of taken to not taken of **0.7777:1**, or approximately **7:9**. On the other hand, backward branches are more likely to be taken at a ratio of **2.4566:1**, or approximately **5:2**.

    ```
    forward_taken=$(grep -c 'F ? taken' block_profile)
    forward_not=$(grep -c 'F ? not taken' block_profile)
    backward_taken=$(grep -c 'B ? taken' block_profile)
    backward_not=$(grep -c 'B ? not taken' block_profile)
    echo "scale=4; $forward_taken / $forward_not" | bc
    0.7777
    echo "scale=4; $backward_taken / $backward_not" | bc
    2.4566
    ```

11. Evidently, forward branches are less likely to be taken than not whereas backward branches are more likely to be taken than not. This points us to a common static branch prediction strategy of *backwards taken / forwards not taken.* This holds as forward branches are typically conditionals and backward branches are typically loops.

# 2 Phase 2: Modelling

## 2.1 Associativity and Cache Lines

12. A Branch Target Buffer (BTB) typically operates on a small working set, and so is itself a small cache. As such, it does not benefit as much as larger caches from the reduced lookup delay offered by set associativity. Therefore, a fully associative BTB structure is ideal to completely eliminate conflict misses.

    This especially helps nested loop structures, as their branch instruction addresses are likely to be close to each other and thus more likely to map to the same tag fields in a cache with lower associativity.

    Moreover, loop branches typically exhibit a high degree of temporal locality, making it extremely costly to run a sequence of branch instructions that would undergo cache thrashing in a set associative cache.

13. Spatial locality typically does not hold in the case of caching branch target addresses, as a branch instruction does not imply that its immediate neighbours are also branch instructions. As such, the BTB should implement single element cache lines to simplify lookup logic, not waste storage and allow the BTB entries to be fully utilised.

## 2.2 Implementation of Branch Target Buffer

14. In line with maximising accuracy for small cache sizes, I modelled a fully associative Branch Target Buffer (BTB). This structure eliminates conflict misses while efficiently handling the temporal locality of branch instructions. Each BTB entry stores a single (source, target) pair since branches lack spatial locality, representing the branch instruction address mapping to the branch target address.

    The model uses a Least Recently Used (LRU) replacement strategy to further exploit the temporal locality of branch instructions. By evicting the least recently accessed branch when the BTB is full, we are more likely to maintain our working set of branches and prevent cache thrashing. This is implemented with a hashmap and doubly linked list for O(1) operations in all cases of updating the cache.

    To avoid cold-start misses, all invariant ('B' type) branches are cached, regardless of whether they are taken or not on their first encounter. While this may increase cache thrashing, to further exploit temporal locality sees the hit rate increase from **62.0974%** when only caching taken branches to **77.5748%** when caching all invariant branches for a BTB with 32 entries.

    Looking closer at what branch instructions are cached, I found that the *block_profile* trace has significantly more forward invariant branches than backwards. This meant using a *backwards taken / forwards not taken* prediction strategy significantly reduced the BTB's coverage over all branch instructions, unexpectedly leading to a drastic fall in hit rate from **77.5748%** to **34.9576%** for the same BTB considered above.

    To run the simulation, compile and execute with:

    ```
    g++ src/main.cpp src/btb.cpp src/profiler.cpp -o main
    ./main
    Size: 32 entries
    Hits: 462556
    Misses: 133715
    Hit Rate: 77.5748%
    ```

15. Exploring cache sizes of 8, 16, 32, 64, 128 and 256 shows a consistent decrease in overhead as the cache size increases. However, the logarithmic scale undermines the diminishing returns of increasing cache size, effectively highlighting how the curve flattens for larger cache sizes.

    Therefore, *Figure 1* supports our earlier notion that a Branch Target Buffer typically works on a small working set at any one time, and suggests that smaller cache sizes offer the best balance between performance and hardware cost.

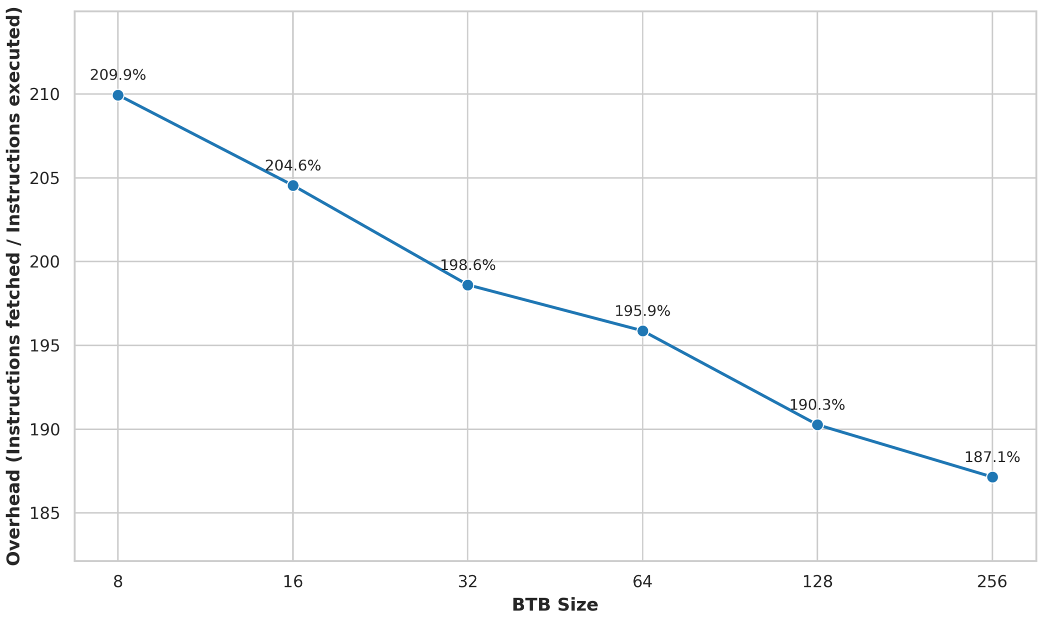

Figure 1: BTB size vs Overhead

# 3 Phase 3: Sophistication

## 3.1 Dynamic Branch Prediction with 2-bit Counter

16. Figure 2 compares two implementations of our Branch Target Buffer: the original BTB, predicting taken for all cached branches; and an improved BTB, incorporating a 2-bit saturated counter to dilate the effect of a single event on a cache entry. Both show the same trend of decreasing overhead as the cache size increases, but the 2-bit counter yields significantly lower overhead across all sizes.

    Similar to *Figure 1*, the logarithmic scale hides diminishing returns for larger caches. Despite this, the relative improvement from the 2-bit counter is still apparent. All in all, this suggests that a BTB benefits more from improving prediction accuracy than increasing capacity. And thus, smaller, smarter BTBs offer the best balance between performance and hardware cost.

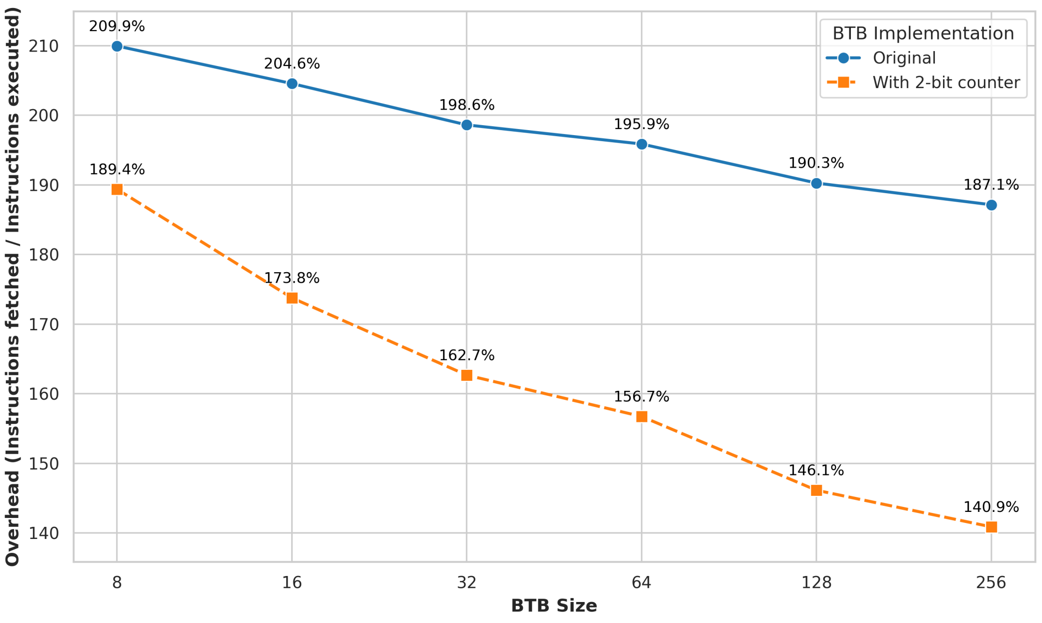

Figure 2: BTB size vs Overhead Improvement with 2-bit Counter

## 3.2 Research

17. Each branch prediction is independent, so the chances of getting the next 5 predictions correct with 90% accurate prediction is given by:

$$0.9^5 * 100 = 59.1\%(3sf)$$

18. For invariant branches, static prediction techniques make all decisions at compile-time, so may fail to accurately reflect a program's behaviour over its lifetime. As such, modern branch predictors rely more on dynamic approaches to predict whether branches are taken or not.

    As seen, an improvement on the 'same as last time' branch predictor is to add hysteresis in the form of a 2-bit counter. Beyond this, modern branch predictors improve prediction accuracy by using multiple sources of historical data and more advanced decision-making procedures. For example, global and local history tracking recognises patterns across multiple branch executions, rather than relying on a recent sequence of outcomes.

Expanding on this, loop branches can be improved upon with loop predictors to prevent mispredictions in the entry and exit of a loop. Interestingly, a tagged geometric (TAGE) predictor captures short-term and long-term trends for more complex branch patterns. Often, modern processors combine multiple approaches in a hybrid fashion to significantly reduce the number, and thus penalty, of mispredictions.

19. Variant branches have non-determinstic target addresses, often used in virtual function calls, switch-case statements and other runtime-dependent jumps. As their target addresses vary dynamically, the problem of branch prediction shifts from predicting a taken/not taken status to predicting a target address.

    Similar to that of invariant branch prediction, a Branch Target Buffer (BTB) is still used but with additional logic to track multiple targets per branch. In addition, tagged geometric history (TAGE) applies to variant branch prediction to correlate indirect jumps with context. This context may make use of preceding branches or the state of the call stack, with an aim to improve target resolution.

    A different way in which modern processors handle variant branch prediction is with a dedicated Return Address Stack (RAS) to predict function return addresses. This operates similarly to a process stack, but is instead given a small hardware implementation to make predictions very quickly. As with invariant branch predictors, a modern processor often combines approaches in variant branch prediction to more accurately reflect a branch instruction's changing context within a program.

20. There is one occurrence of a 'D' type branch in *block_profile*, at address **0x00018CAC**, representing a branch which calculates a result and directly writes it to the PC. The source address appears nowhere else in *block_profile*, suggesting it is not part of the regular control flow and is unlikely to be a virtual function call or switch-case statement, as variant branches often are. Instead, it matches the functionality of returning from an exception/interrupt handler. In newer architectures, this branch type is handled explicitly such as ARM's ERET or RISC-V's MRET.