Anirban Khara
G40191570

A STUDY OF BRANCH PREDICTION STRATEGIES

Summary

In parallel computer systems, branch instructions can cause delays by interrupting instruction processing. Predicting branch directions can minimize delays, but incorrect predictions may increase them, underlining the importance of accurate prediction strategies. This paper reviews existing branch prediction techniques, then introduces new, more accurate and efficient methods. The study uses instruction trace data from six FORTRAN programs, emphasizing the variance in branching behavior across applications. The results are context-specific, but the core concepts have broader applicability. Earlier works in this field are acknowledged. The paper details two primary branch prediction strategies, laying the foundation for deeper discussions on enhancing prediction accuracy.

Two preliminary prediction:

Branch instructions evaluate a condition to determine execution flow. If true, execution starts at the target address; if false, it continues sequentially. Unconditional branches always have consistent outcomes, either always true or false, but were not differentiated in our statistics. Therefore, unconditional branches were included. A simple branch prediction method assumes branches are always taken or never taken. Given that most unconditional branches are always true and loops end with branches taken to their start, predicting all branches as taken often achieves over a 50% success rate.

Programs like ADVAN (99.4%) and SCI2 (96.2%) show high accuracy with this strategy, indicating most branches in these programs are indeed taken. For SINCOS (80.2%) and TBLINK (61.5%), the accuracy is moderate, suggesting a mix of taken and not-taken branches. GIBSON (65.4%) and SORSTT (57.4%) have lower accuracies, revealing that this strategy isn't always optimal.

In summary, the data from Figure 1 reinforces the concept that branching behavior can vary significantly between different programs. Strategy 1, which assumes all branches will be taken, can be incredibly effective for some programs but less so for others. The variation in prediction accuracy across different programs underscores the importance of having adaptable and diverse branch prediction strategies to cater to the unique behavior of different programs.

Strategy 2, a branch prediction mechanism, operates on the premise of predicting a branch's decision based on its last execution. If previously unexecuted, the branch is predicted to be taken. Analyzing the accuracy data from Figure 2, Strategy 2 yields notable accuracy across programs: ADVAN (98.9%), GIBSON (97.9%), SCI2 (96.0%), SINCOS (76.2%), SORTST (81.7%), and TBLINK (91.7%). Compared to Strategy 1, Strategy 2 typically offers superior prediction accuracy. However, it's not without challenges. Theoretically, its feasibility is contested due to the potential infinity of individual branch instructions a program might contain. Strategy 2 also demonstrates an inherent second-order program sensitivity, as it predicts unexecuted branches as taken. A distinguishing observation is that Strategy 1 occasionally trumps Strategy 2 in scenarios where frequently taken branches aren't. This occurs because Strategy 2, in such cases, makes two successive incorrect predictions. Despite its high accuracy, these nuances indicate that other strategies might achieve even higher success rates than Strategy 2.

Strategy 1a involves predicting that certain operation codes will always result in a branch being taken, while others will not. This strategy was developed based on analyzing six CYBER 170 FORTRAN programs. It found that operations like "branch if negative", "branch if equal", and "branch if greater than or equal" are commonly taken. Applying this strategy improved prediction accuracy in most programs, with the GIBSON program seeing the biggest increase from 65.4% to 98.5%. However, some inaccuracies were observed, particularly when predicting the "branch if plus" operation.

Strategy 3 predicts that all backward branches will be taken and all forward branches will not be taken, based on the idea that loops typically end with backward branches. While this strategy often performed well, sometimes even outdoing Strategy 2, its accuracy was significantly low at about 35% for the SINCOS program. This highlights that program-specific nuances can heavily influence prediction accuracy. A downside of Strategy 3 is the

potential delay in prediction since the target address might need computation or comparison with the program counter, making it slower than other strategies.

Dynamic prediction strategies in computer architecture focus on branch prediction, which seeks to improve the flow in instruction pipelines by predicting future branching behavior based on past activities. These strategies base their predictions on past branch history. An exemplar, Strategy 2, predicts using the outcome of the most recent branch instruction. However, some strategies, like Strategy 7, consider a majority of recent executions for predictions.

These prediction techniques often use bounded tables to record a finite amount of past branch data. If a branch instruction is not found in this table, default strategies come into play. A straightforward default might predict a branch is always taken, as seen in Strategy 2. Although this method simplifies predictions and can save on memory, it may not be the most accurate in diverse scenarios.

To address the issue of maintaining the most relevant branch history in bounded tables, replacement strategies are employed. Two common strategies are First-In-First-Out (FIFO) and Least Recently Used (LRU). While both have their merits, LRU often proves more effective in scenarios where branch instructions display periodic or iterative behavior. This adaptability of LRU makes it particularly suitable for dynamic branch prediction, offering a balance between prediction accuracy and computational efficiency.

Strategy 4 in computer architecture revolves around dynamic branch prediction, aiming to improve the flow of instruction pipelines by predicting the outcome of branch instructions based on recent history. The strategy maintains a table recording the most recently used branch instructions that were not taken. If a branch instruction is present in this table, the prediction is that it will not be taken; otherwise, the opposite is predicted. To ensure the table's efficiency, entries taken are purged, and the Least Recently Used (LRU) replacement strategy is used for new entries.

The provided figure, Figure 5, showcases the accuracy of Strategy 4 with tables of various sizes: 1, 2, 4, and 8 entries. As the table size increases, the strategy's prediction accuracy approaches that of Strategy 1 and 2, especially when the table can contain all active branch instructions. This convergence occurs because when the table holds all active branches, they are all predicted like in Strategy 2.

The table's data further corroborates this, showing higher prediction accuracies for larger table sizes across different programs. For instance, the 'GIBSON' program experienced a drastic accuracy improvement from 65.4% with a single entry to 97.9% with 2, 4, or 8 entries. This strategy's dynamic nature, paired with adaptive memory management, offers a balanced approach to branch prediction in computing.

Strategy 5 pertains to branch prediction in computer architecture. It involves maintaining a bit for each instruction in the cache. If an instruction is of the branch type, this bit records if it was taken during its last execution. For predictions, branches are assumed to follow their most recent behavior. If a branch has never been executed, it's predicted as "taken". An alternative approach, similar to Strategy 5, uses a combination of static prediction based on operation code and a cache prediction bit. The prediction gets adjusted only when there's a misprediction, providing a potential advantage in terms of memory update times.

Further refining dynamic strategies suggests replacing associative memory with random access memory. Instead of using a branch instruction's full address for indexing, it's hashed to a smaller number of bits. This method might lead to multiple branch instructions hashing to the same index, but erroneous predictions could be rectified. The hashing can employ various methods, ensuring branch instructions in proximity tend to hash differently.

Figure 6 showcases the prediction accuracy of Strategy 5 for various programs. For instance, the program "ADVAN" has an accuracy of 98.9%, while "SINCOS" is at 76.1%. Overall, Strategy 5 seems to offer commendable prediction accuracy, as it typically hovers around or exceeds 90% for most programs.

Strategy 6 for branch prediction employs a technique where the branch instruction address is hashed to "m" bits. This hashed value is used to access a random access memory (RAM) which contains the outcome of the most recent branch instruction indexing that location. The prediction assumes that the branch outcome will remain consistent. The memory can be initialized with all 0's or all 1's, although this has minimal impact on results.

Anirban Khara
G40191570

Further enhancement of Strategy 6 accounts for anomalous branch decisions. Instead of storing just a single bit, the memory now holds a count. If a branch instruction is executed, the corresponding memory index is incremented, but if the branch is not taken, it's decremented. This approach essentially captures the history of multiple branch executions for a more accurate prediction. For instance, when an address is hashed to predict a branch instruction, the sign bit of the count in RAM decides the prediction. A 0 predicts the branch will be executed and a 1 predicts the opposite.

Figure 7 showcases the efficiency of Strategy 6 by providing prediction accuracy percentages for various programs. For instance, the "ADVAN" program has a 98.9% prediction accuracy using this strategy, while "SINCOS" stands at 76.2%. This table indicates that Strategy 6, while efficient, varies in accuracy depending on the program.

Strategy 7 refines Strategy 6 by using two's complement counts in place of a single bit for branch prediction. This means that the prediction is based on the sign bit of the accessed count. A positive count (sign bit 0) indicates a prediction that the branch will be taken, while a negative count (sign bit 1) suggests the branch won't be taken. Every time a branch is taken, the counter increments, and when not taken, it decrements. Strategy 6 can be seen as a subset of Strategy 7 with a one-bit count.

A notable aspect of using count-based prediction is the introduction of a "vote" mechanism. This is evident when multiple branch instructions hash to an identical count, affecting the prediction.

Figure 8 provides insights into the prediction accuracy of Strategy 7 using 2 and 3 bit counters. The data reveals that the accuracy is commendable and often on par or better than many previous strategies. Notably, a 2-bit counter frequently outperforms a 1-bit counter. However, increasing to a 3-bit counter doesn't always enhance results. This limitation is partially due to the "inertia" from relying on distant history or from earlier branch instructions influencing predictions of subsequent branches.

Hierarchical Branch Prediction in CPU Performance

In computer architecture, hierarchical branch prediction is pivotal for optimizing the efficiency of CPUs. This method assesses the penalties associated with incorrect instruction predictions post-branching, emphasizing that penalties escalate the further an instruction progresses after a mispredicted branch.

For instance, consider a CPU where, based on a branch prediction, a wrong guess results in a 6-clock period delay to retrieve the right instructions. If the prediction is accurate but the instructions aren't promptly issued, a 3-clock period delay is incurred. Should instructions be pre issued regardless of prediction accuracy, the delay is eliminated for correct predictions, but shoots to 12-clock periods for incorrect ones.

Assuming an overall 70% correct branch prediction, the branches can be divided into two sets. Set A has 50% prediction accuracy, while Set B boasts 90% accuracy. Notably, it's identifiable in real-time to which set a branch instruction belongs.

Three strategies arise:

1. Prefetching all branches yields an average delay of 3.9 clock periods.
2. Prefetching and pre issuing for all branches result in an average 3.6 clock periods delay.
3. Differentially applying prefetch to Set A and both prefetch & preissue to Set B minimizes the delay to 2.85 clock periods, thus emerging as the optimal strategy.

To further refine this approach, Strategy 7 employs a counter mechanism. If a counter reaches its maximum, the previous prediction was likely accurate, implying the current prediction may also be correct. Predictions based on extreme counter values (maximum or minimum) are deemed high-confidence, while others are considered lower-confidence.

This counter method has been tested on various programs. A key observation from Figure 9, summarizing the accuracy of hierarchical predictions, reveals predictions made at counter extremes are consistently more accurate. For instance, in the 'SORTST' program, 78.5% of predictions at the extremes were correct, while the overall accuracy was 84.7%. This counter-based differentiation can serve as a heuristic in refining CPU performance, thereby streamlining operations and minimizing delays.

This research delves into the accuracy of various branch prediction methods, both previously

established and newly proposed. Branch prediction is a technique used in computer architecture to enhance the instruction throughput of a pipeline. The study presents seven strategies for branch prediction, each having its own unique approach.

Strategy 1 assumes that all branches will be taken, while Strategy 1a narrows this down by predicting that only specific branch operation codes will be executed.
Strategy 2 predicts the outcome of a branch based on its last execution.
Strategy 3 anticipates only the backward branches.
Strategy 4 employs a table containing the m most recent branches that were not taken.
Strategy 5 uses a history bit stored in cache to guide its predictions.
Strategy 6 hashes the branch address to m bits, accesses a 2^m word RAM with history bits, and then predicts using these history bits.
Strategy 7, similar to the sixth, utilizes counters instead of a single history bit.
The results table highlights the accuracy percentages of each strategy across various kernels. For the 'ADVAN' kernel, almost all strategies hover around 98-99% accuracy. However, for the 'GIBSON' kernel, there's a significant variance, with Strategy 4 having the lowest at 65.4% and Strategy 7 peaking at 97.9%. Similarly, for the 'SINCOS' kernel, Strategy 3 yields the lowest accuracy at 35.2%, whereas other strategies maintain a higher percentage.

Dynamic methods, especially Strategy 7, prove to be more precise. This strategy, utilizing random access memory (RAM) over associative memory, was adeptly adaptable and consistently showed good results across applications. The flexibility, cost, and accuracy make Strategy 7 a standout in branch prediction methods.