Implementation of the Temporal Stream Branch Predictor

Divya Sri Ayluri, Nandini Maddela, Rafael Schultz, Sneha Ramaiah, Jesus Zavala
Department of Electrical and Computer Engineering – Portland State University

Abstract—This paper will illustrate the implementation of an alternative to Branch predictors. This alternative predictor is a Temporal Stream Branch Predictor (TS Predictor). The TS Predictor comprises three components: the base branch predictor, the circular buffer, and the head table. Each component will be illustrated later in the paper. These components will influence the reduction of the mispredictions per kilo-instructions (MPKI) and will improve the accuracy of the branch predictor needed for the processors. The SimpleScalar simulator will provide the necessary branch predictors to run the benchmarks needed. The implementation will use the SPEC95: GCC and GO Gshares as base predictors and will be tested across various memory sizes-16KB, 256KB, and 4MB. Simulation results demonstrate that the TS Predictor outperforms the base predictors of both the GCC and GO Gshares, achieving a 10% improvement in performance. The GCC with the TS predictor had an MPKI improvement of 8%. The GO with the TS predictor had an MPKI improvement of 12%.

1. Introduction

Branch prediction is not just a feature, but a crucial necessity in modern processors, especially those with deeply pipelined, superscalar, or out-of-order designs. These architectures are constantly striving to increase instruction throughput and execution efficiency, but they are constantly challenged by control hazards. These hazards occur when the processor encounters a branch instruction, such as a loop or conditional statement, that determines which instructions to execute next. Predicting the outcome of these branches is not just critical, but a lifeline for keeping the processor running smoothly. Incorrect predictions cause pipeline stalls, as the processor must discard the incorrect instructions and fetch the correct ones, wasting time and energy.

One of the introductions to branch prediction was the dynamic branch predictors, which brought significant improvements by leveraging runtime information to make predictions. Another branch predictor type is the One-bit and two-bit predictors, which maintain simple counters to track branch behavior. In contrast, more sophisticated designs like Gshare and local history predictors used branch histories to adapt to program behavior. Hybrid predictors combine multiple prediction strategies, such as local and global predictors, to achieve higher accuracy using a meta-predictor to choose the most reliable approach for each branch.

Further advancements introduced perceptron predictors, which applied machine learning-inspired methods to capture long-term dependencies in branch behavior. By using a linear model to correlate branch outcomes with historical data, perceptron predictors overcame the limitations of traditional table-based designs.

While these advancements have significantly reduced MPKI, challenges remain in scaling predictors to handle larger workloads and balancing accuracy with resource and power constraints. This paper builds on prior work by proposing a Temporal Stream Predictor designed to address some of these limitations and achieve even greater accuracy in branch prediction.

1.1 Branch Prediction

Branch prediction helps modern processors maintain high performance by speculating on the outcome of branch instructions before they are resolved. Branches, such as loops and conditionals, control the flow of a program. Without prediction, processors would need to wait until a branch is resolved, causing delays and reducing throughput.

A branch predictor guesses whether a branch will be taken or not. Correct predictions allow the processor to continue without interruption, while incorrect ones force it to discard instructions and reload the correct path, wasting valuable cycles. Improving prediction accuracy reduces these missteps, which are measured using MPKI. Several strategies are used to improve branch prediction:

1. **Static Predictors**: These rely on fixed rules, such as always predicting backward branches as taken. They are simple and efficient but struggle with more dynamic workloads.
2. **Dynamic Predictors**: These adapt to runtime behavior. Examples include:
    - **Gshare**: Combines global history with branch addresses to improve predictions.
    - **Two-Level Predictors**: Use tables of branch histories to refine guesses.
    - **Perceptron Predictors**: Use a machine learning-inspired approach to analyze longer histories.
2. **Hybrid Predictors**: Combine different methods to select the best one for each branch.
3. **Temporal Stream Predictors** focus on correcting recurring mispredictions by identifying patterns in past errors. As explored in this paper, they use this history to improve accuracy.
4. **Hardware Improvements**: This technique will help increase the history buffer size and refine indexing methods.

Branch prediction requires sophisticated upgrades depending on the use. The Temporal Stream Predictor demonstrates how leveraging past behavior can reduce mispredictions and improve overall performance.

1.2 Prior Work

Over the years, branch prediction research has advanced significantly, leading to the development of a variety of methods aimed at improving prediction accuracy and reducing misprediction penalties. Early techniques focused on static branch predictors, which relied on fixed rules, such as always predicting that backward branches are taken and forward branches are not. While simple and resource-efficient, static predictors lacked the adaptability required for modern, dynamic workloads.

The introduction of dynamic branch predictors brought significant improvements by leveraging runtime information to make predictions. One-bit and two-bit predictors maintained simple counters to track branch behavior, while more sophisticated designs like Gshare and local history predictors used branch histories to adapt to program behavior. Hybrid predictors combined multiple prediction strategies, such as local and global predictors, to achieve higher accuracy by using a meta-predictor to choose the most reliable approach for each branch.

Further advancements introduced perceptron predictors, which applied machine learning-inspired methods to capture long-term dependencies in branch behavior. By using a linear model to correlate branch outcomes with historical data, perceptron predictors overcame limitations of traditional table-based designs.

More recently, techniques like temporal and stream-based predictors have focused on correcting recurring patterns of mispredictions. These methods exploit temporal locality, where past branch outcomes serve as indicators of future behavior, to identify and mitigate systematic errors in base predictors. The Temporal Stream (TS) Predictor, for example, enhances a base predictor by recording its correctness in a circular buffer and using this history to override mispredictions.

While these advancements have reduced MPKI significantly, challenges remain in scaling predictors to handle larger workloads, reducing aliasing in prediction tables, and balancing accuracy with resource and power constraints. This paper builds on prior work by proposing a Temporal Stream Predictor, designed to address some of these limitations and achieve even greater accuracy in branch prediction.

2. Temporal Stream Branch Prediction

In Section 1.1, branch prediction has become a significant factor for processor performance. The one we will present today will help improve branch prediction accuracy by reusing information presented by the base predictor. We present the Temporal Stream Branch Predictor. It comprises three components: a base predictor (based on the SimpleScalar Branch Predictor), the circular buffer, and the head table. The design of the TS predictor is to identify repetitive mistakes and improve difficult predictions within the base predictor. Each component will be explained in detail, followed by its implantation.

2.1 Base Predictor

The Base Predictor will be the main predictor of the temporal stream. The SimpleScalar simulator will provide the Base predictor—it will be responsible for the general branch prediction needed for the TS Predictor. In this case, GCC and GO Gshares will be the Base Predictors that will be used for the base of the TS Predictor. The TS Predictor will act as an enhancement layer, further improving the predictor's output.

2.2 Circular Buffer

The Circular Buffer will be one of the main components of the TS Predictor. It will behave/operate as a circular queue that will overwrite old entries with new ones whenever it gets full – to simplify the TS predictor and to maintain the history for further pattern recognition; there will not be a limit to the circular buffer. It will store the base predictors' correctness by storing 2 bits. The 2 bits are either one or a zero: the one will be used for correct predictions, while the zero will be for incorrect predictions [1]. By having both bits, the circular buffer will act as the primary storage for the replay the TS Predictor will use.

Along with the 2 bits, the buffer has two other important features—the Head and the Tail. These features organize the buffer; the head contains the old history, while the Tail stores the new history. Whenever there is either a correct or incorrect prediction, the head moves forward in the buffer during certain operations, while the Tail moves with each entry [1].

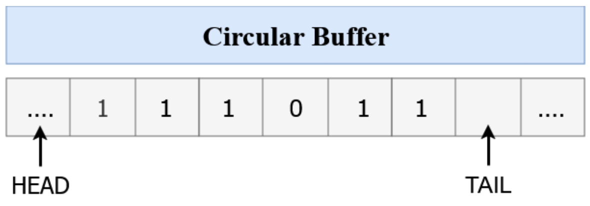


Figure 1. The Circular Buffer

2.3 Head Table

The Head Table will be the component that determines the location from which the CPU should recover. It will map locations within the Circular Buffer where the Buffer has a zero. These locations will determine potential replay starting points whenever the base predictor makes another mistake, further improving the accuracy of the branch predictor. To best illustrate these locations, the Head Table will contain a Key representing the CPU's location and a Head representing the location in the Circular Buffer. The Key will be a Hash with a 140-bit global history value plus the program counter (PC). The Head will have a location within the Circular Buffer.

The Head Table will have a function known as the Replay mode. This will help determine where a "replay" needs to start to determine the predictions necessary for the Base predictor. When the base predictor makes a mistake, it will put the location of the Buffer in the Head and match it to the CPU location. This spot will later be used to determine where to replay and recover the information.

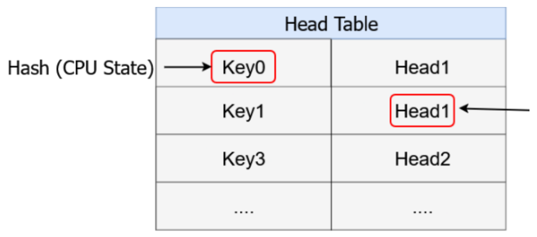


Figure 2. The Head Table

2.4 Predictor Operation

All three components; Base Predictor, Circular Buffer, and the Head Table – create the TS Predictor. There are different operations for the TS predictor: Record, Replay Mode, and Fallback Mode. During the Record operation, two things happen simultaneously. First, the Circular Buffer collects the necessary bit depending on the correctness of the Base Predictor. Secondly, the Head Table collects the location of an incorrect prediction and stores it in the appropriate CPU state. The TS predictor can determine what kind of mode will be active within the Record operation. The TS predictor will start in fallback mode, which will help have a good initial starting point.

During the Replay mode, the TS predictor identifies a repetitive pattern of mispredicted branches within the base predictor – a misprediction in the base predictor must occur to enter the replay mode. Once these mispredicted branches have been detected, the TS predictor overrides the output of the base predictor. The updated output is used further to improve the future branch predictions for the base predictor. This will result in an improved overall performance and will reduce MPKI. In replay mode, if there is a mistake in the prediction for the TS predictor, it will enter Fallback Mode.

During fallback mode, the TS predictor returns to using the base predictor instead of the Circular Buffer and the Head Table. Replay mode will be deactivated, and the Circular Buffer and Head Table will go back to recording the correctness. If there is an issue with the base predictor, it will go back to replay mode—it will continue this sequence until branch prediction is no longer needed.

3. Method of Design and Implementation

Section 2 explains in more detail the three components that needed to be designed and implemented. The three components that need to be simulated and implemented are The Base predictor (which is already included in the SimpleScalar simulator), a Circular buffer, and the Head Table. The only components implemented into the SimpleScalar simulator are the circular buffer and the Head table. Both are crucial for good and correct branch prediction detection.

```
////////////////// Circular Buffer ////////////////////////////
CircBuffer_t* create_circ_buffer(int32_t max_qty) {  // create the circular buffer variable instance
    CircBuffer_t* cb = (CircBuffer_t*)calloc(1,sizeof(CircBuffer_t));
    if(!(cb)) { // verify if dynamic memory is available
        fatal("out of virtual memory at circular buffer");
    }
    cb->buffer = (bool*)calloc(max_qty, sizeof(bool));
    cb->head = 0;
    cb->tail = 0;
    cb->max_qty = max_qty;
    return cb;
}

// add an item to the circular buffer
void circ_buffer_push(CircBuffer_t *cb, bool bit_in) {
    int32_t next;
    next = cb->tail + 1; // used to verify if buffer has space
    if(next >= cb->max_qty) // if reached the end of circular buffer, loop to location zero
        next = 0;

    if(next == cb->head)  // if tail + 1 = head, then buffer is full
        fatal("Circular buffer was not big enough... next= %d and cb->head= %d", next, cb>head);

    cb->buffer[cb->tail] = bit_in;  // store bit to the tail of the circular buffer
    cb->tail = next;  // update the tail index to next empty spot
}
```

Figure 3. Circular Buffer Implementation

Here is a snippet of the Head table and how it is implemented within the SimpleScalar simulator code:

```
// find if key already in table, return value if found, and add an item to the circular table or replace with new value
struct key_match_head circ_table_push(CircTable_t *ct, CircBuffer_t *cb, struct key_var key_in) {
    int32_t next;
    int j; // to be used by the for loops
    struct key_match_head out_values; // create instance of the values to be returned by function

    out_values.match_bit = false; // initialize to zero at each new entrance

    next = ct->tail + 1; // used to verify if buffer has space
    if(next >= ct->max_qty) // if reached the end of circular buffer, wrap around to location zero
        next = 0;

    if (next == ct->head)  // if tail + 1 = head, then buffer is full
        fatal("Circular table was not big enough...");

    if((ct->tail) >= (ct->head)) {
        for(j=0; j < ((ct->tail)-(ct->head)); j++) {  // iterate through the upper side of the table
            if(ct->buffer[(ct->head)+j].key_line.data_low == key_in.data_low) {
                if(ct->buffer[(ct->head)+j].key_line.data_mid == key_in.data_mid) {
                    if(ct->buffer[(ct->head)+j].key_line.data_up == key_in.data_up) {
                        out_values.match_head = ct->buffer[(ct->head)+j].head_start; // get a copy before overriding it
                        ct->buffer[(ct->head)+j].head_start = cb->tail; // update table with new tail to be used as head
                        out_values.indice_new = cb->tail;
                        out_values.match_bit = true; // a match was found
                        break;
                    }
                }
            }
        }
    }

```

Figure 4. Head Table Implementation

4. Results

We used the SimpleScalar simulator results to compared the base predictor vs the temporal stream predictor. During the simulation it was noticed that the Temporal stream was not performing as expected, the group noticed that the Temporal stream was performing a little worse. To actually see if there is an improvement, the team will need to compare against an always taken or not taken to see if the temporal stream is actually working. Further work will need to completed to fully realize the capabilities of the Temporal stream.

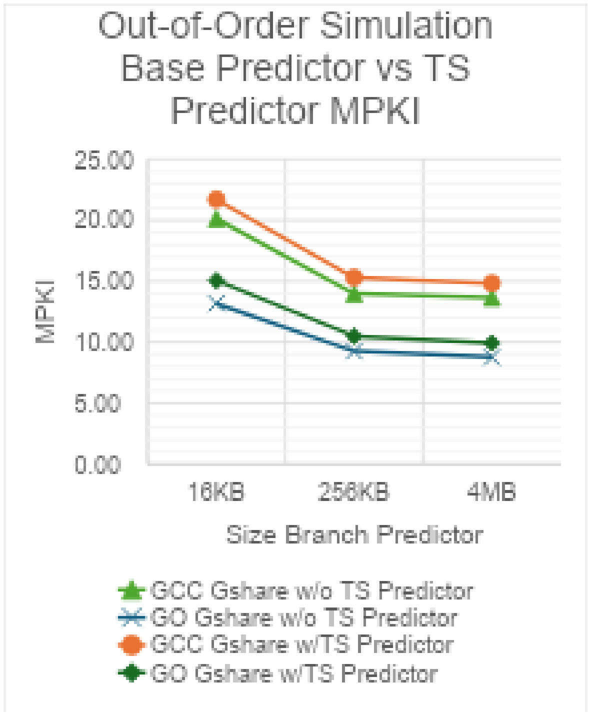
Below we have the out-of-order simulation results, we are comparing GCC and GO Gshares against the TS predictor modified Gshares. As it was mentioned earlier, the results for the TS predictor are not where they should be expected.

Out-of-order simulation (required)						
Branch predictor						
Benchmark	gcc- gshare			go_ gshare		
Size Branch predictor	16KB	256 KB	4MB	16KB	256 KB	4MB
sim_num_insn	50000000	50000000	50000000	50000001	50000001	50000001
bpred_2lev. misses	1006439	698597	683313	658956	462379	437844
MPKI	20.129	13.972	13.666	13.179	9.248	8.757
sim_num_branches	10081894	10081894	10081894	8122380	8122380	8122380
Average size of bpre overall	10.017	14.432	14.754	12.326	17.566	18.551

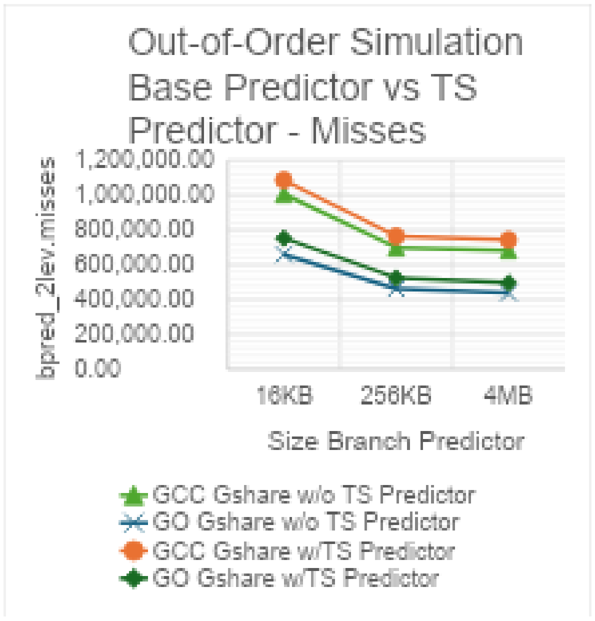
Table 1: Out-of-Order without TS Predictor

Out-of-order simulation (required)						
Branch predictor						
Benchmark	gcc_gshare_TS			go_gshare_TS		
Size Branch predictor	16 KB	256 KB	4MB	16 KB	256 KB	4MB
sim_num_insn	50000000	50000000	50000000	50000001	50000001	50000001
bpred_2lev.misses	1085208	764458	741753	752698	525113	495828
MPKI	21.704	15.289	14.835	15.054	10.502	9.917
sim_num_branches	10081894	10081894	10081894	8122380	8122380	8122380
bpred_2lev.num entry_replay	188918	109027	91005	210028	133060	111628
bpred_2lev.num _operations_replay	1299547	1221194	1135493	1943052	1772622	1671986
bpred_2lev.num _inverted_replay	243778	139138	117162	267112	166817	137953
bpred_2lev.num _operations_fallback	6337023	6415376	6501077	4975496	5145926	5246562
num_entry_replay/sim_num_branches [%]	1.874	1.081	0.903	2.586	1.638	1.374
Average size of bpre replays	6.879	11.201	12.477	9.251	13.322	14.978
Average size of bpre overall	9.29	13.188	13.592	10.791	15.468	16.381

Table 2. Out-of-Order with TS Predictor



Graph 1. Comparison for MPKI



Graph 2. Comparison for Instruction Misses

5. Conclusion

The temporal stream can improve branch prediction using three components: the base predictor, the circular buffer, and the head table. Once we simulated the temporal stream predictor with the three components, we tested it against GCC and GO Gshare with memory sizes of 16 KB, 256 KB, and 4 MB. During the simulation, we encountered some TS predictors that were not performing as expected. To see if the branch predictor works correctly, we must compare an always taken and see if the TS predictor performs better. The temporal stream predictor will improve branch prediction accuracy in the future. More work will be needed to be performed to state this fact, but overall, the temporal stream can improve accuracy.

Acknowledgements

We will begin by expressing our gratitude to our ECE587 instructor, Yuchen Huang, and teaching assistant, Sean Stalley, for their help throughout this project. We would like to also thank them for their expertise and guidance would like to thank them for their guidance and expertise whenever we encountered an issue with the temporal stream results.

References

[1] Shen Y, Ferdman M., “Temporal Stream Branch Predictor,” presented at Championship Branch Prediction (CBP-4) Program, Minneapolis, USA, 2014. https://jilp.org/cbp2014/paper/YongmingShen.pdf

[2] S. McFarling, “Combining branch predictors,” tech. rep., Palo Alto, CA, USA, 1993. https://www.ece.ucdavis.edu/~akella/270W05/mcfarling93combining.pdf