Assignment 4: Branch Predictor Implementations

CPEN 411 - Computer Architecture

Isabelle André – 12521589 *6 December 2022*

1.0 Project Description

This assignment consisted in understanding the implementation of branch prediction policies using ChampSim. Two branch predictors were implemented. A 2-bit correlated branch predictor with a single bit of history was first created, followed by a hashed-gselect predictor with 5 bits of branch history. The hashing employs a XOR function to index the table. The performance of both branch predictors were compared against the hashed perceptron predictor across 5 different spec2006 workloads using geomean.

2.0 Branch Predictor Implementations

In this assignment, two simple branch predictors were implemented using the ChampSim simulator. Each branch predictor offered a total budget of 16384 entries in a table with each entry being 2 bits in size. Bit manipulation was used to slice and dice this table as needed to be able to index the table rows and entries using the hash and history bits.

2.1 2-Bit Correlated Branch Predictor With 1-Bit History

2.1.1 Description

The 2-Bit Correlated Branch Predictor with 1 bit of history involves the use of 2-bit saturating counters with a minimum value of 0 and a maximum value of 3. For 1 bit of history, there are 2^1 = 2 entries per row, each containing a 2-bit saturating counter. Therefore to index the table rows, there must be 16384 entries/2 entries per row for a total of 8192 rows.
As the hash is a 32 bit integer, we take the 13 least significant bits for addressing 8192 rows, as they contain the most entropy.

This can be represented in a table format such as in Figure 2.1.

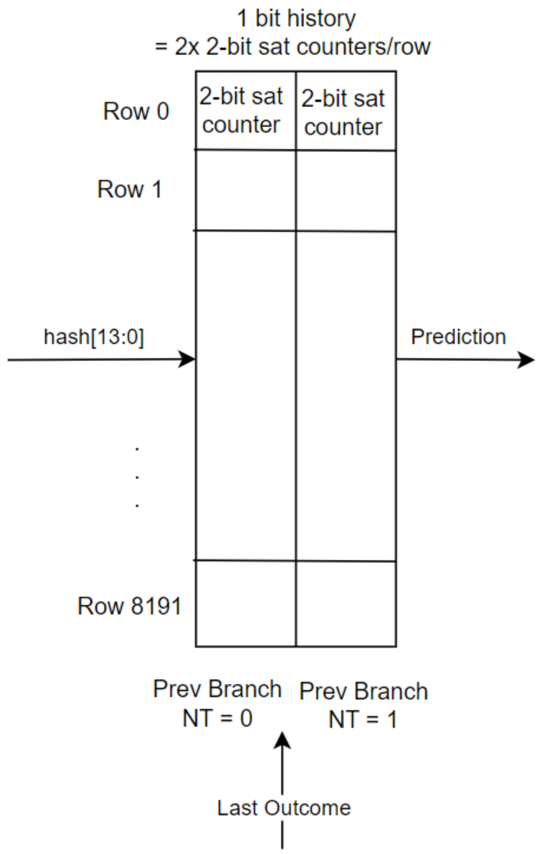

Figure 2.1: 2-Bit Correlated Branch Predictor and 1 Bit of History as a Table

2.1.2 Implementation

The majority of the implementation challenge related to bit manipulation for slicing the table of 16384 entries into the required table length and width as shown above.

First, the last 13 bits of the hash are extracted to use for row addressing by applying a mask to the address, obtaining a number in the range of [0, 8191]. Then, the index is multiplied by the number of entries per row known to be 2, to map the index to the corresponding entry in the array. Depending on the status of the history bit, the first or second entry of the row will be selected to predict the next branch outcome.

```
int mask = (1 << ADDRESS_BITS) - 1;
uint32_t index = (hash & mask)*ROW_SIZE;
```

The indexing of the branch history table can be seen in its unsliced state in Figure 2.2.

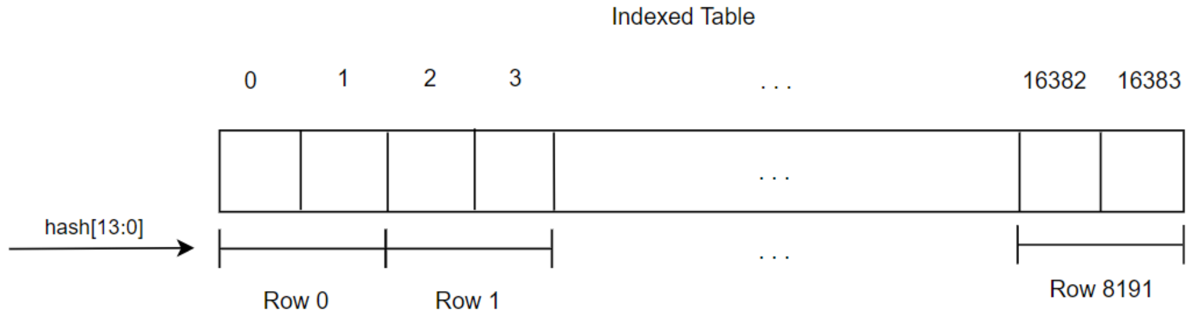

Figure 2.2: Unsliced 1 Bit History Branch History Table

Once the outcome of the prediction is received, the saturating counter is accessed once more to update its state. If the branch was taken, we increment the counter, otherwise, we decrement it.

2.1 Hashed GSelect Branch Predictor

2.1.1 Description

The hashed gselect branch predictor works similarly to the gshare and gselect predictor. This implementation uses a XOR function to hash the address to index the entries. As 5 bits of history are used, there are 2^5 = 32 entries per row, each containing a 2-bit saturating counter. Therefore to index the table rows, there must be 16384 entries/32 entries per row for a total of 512 rows.
As the hash is a 32 bit integer, we take the 9 least significant bits for addressing 512 rows, as they contain the most entropy.

The hashing process and table indexing is shown in Figure 2.3.

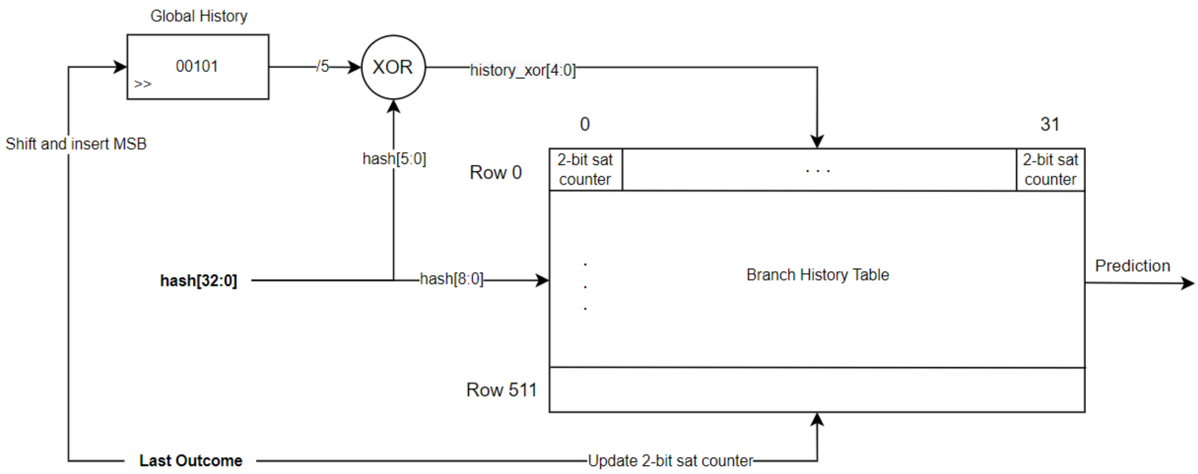

Figure 2.3: Hashed GSelect Branch Predictor and 5 Bit History

2.1.2 Implementation

First, the last 9 bits of the hash are extracted to use for row addressing by applying a mask to the address, obtaining a number in the range of [0, 511]. Then, the least significant 5 bits of the hash are extracted to XOR with the 5 bits of history to address each 2-bit saturating counter in a row. Depending on the output value of the XOR function, one of the 2-bit saturating counters of the 32 entry row will be selected to predict the next branch outcome.

```
int mask_hash = (1 << ADDRESS_BITS) - 1;
uint32_t index = (hash & mask_hash) * ROW_SIZE;

uint32_t hash_xor = index ^ history[cpu];

int mask_history = (1 << HISTORY_BITS) - 1;
int history_xor = (history[cpu] & mask_history);
```

The indexing of the branch history table can be seen in its unsliced state in Figure 2.4.

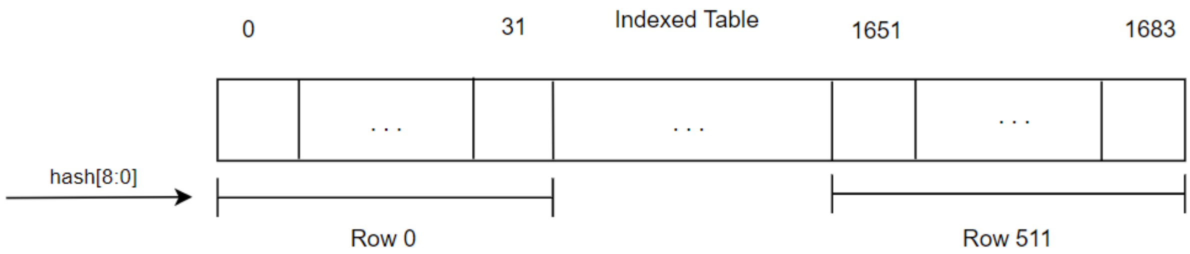

Figure 2.4: Unsliced 5 Bit History Branch History Table

3.0 Performance Comparison

3.1 Performance Per Benchmark

The performance of the 2-bit correlation with 1 bit history and the hashed gselect with 5 bit of history are analyzed by comparing their IPC, Branch Prediction Accuracy, MPKI Percentage, and ROB Occupancy at a Miss to the hashed perceptron branch predictor. As the hashed perceptron predictor is very good, we expect our results to be lower on average than its performance. The spec2006 benchmarks used include 400.perlbench, 401.bzip2, 403.gcc, 429.mcf, and 462.libquantum. The full results for each benchmark per branch predictor and performance metric including Geomean calculations are included in Appendix A.

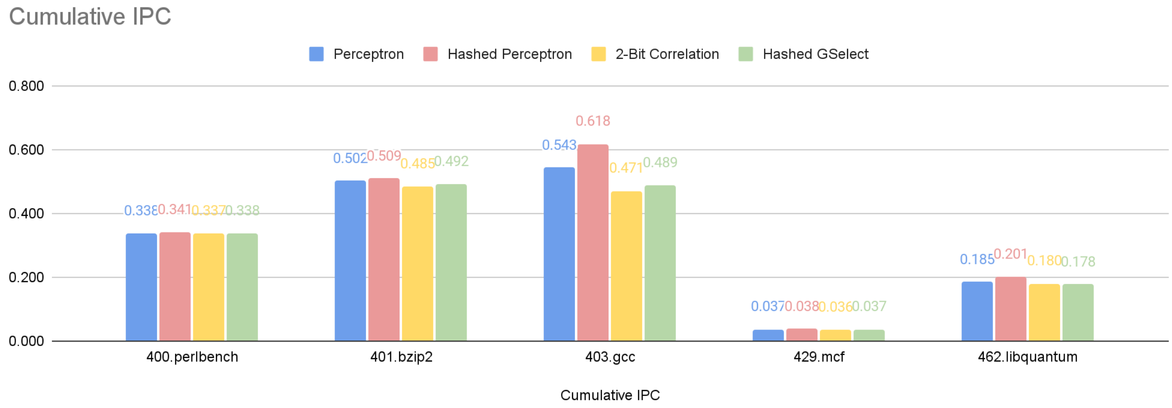

Figure 3.1: ROI_Cumulative_IPC Comparison

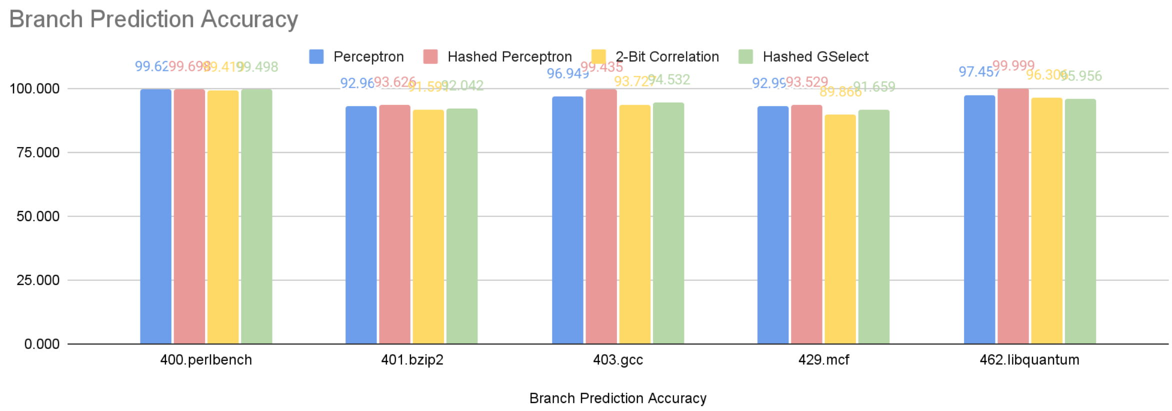

Figure 3.2: CPU_0_Branch Prediction Accuracy Comparison

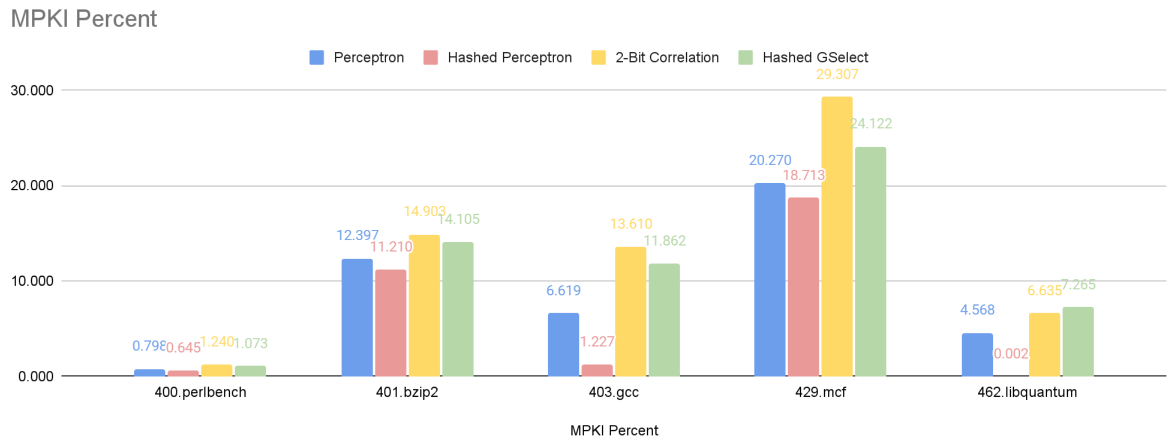

Figure 3.3: CPU_0_MPKIPERCENT Comparison

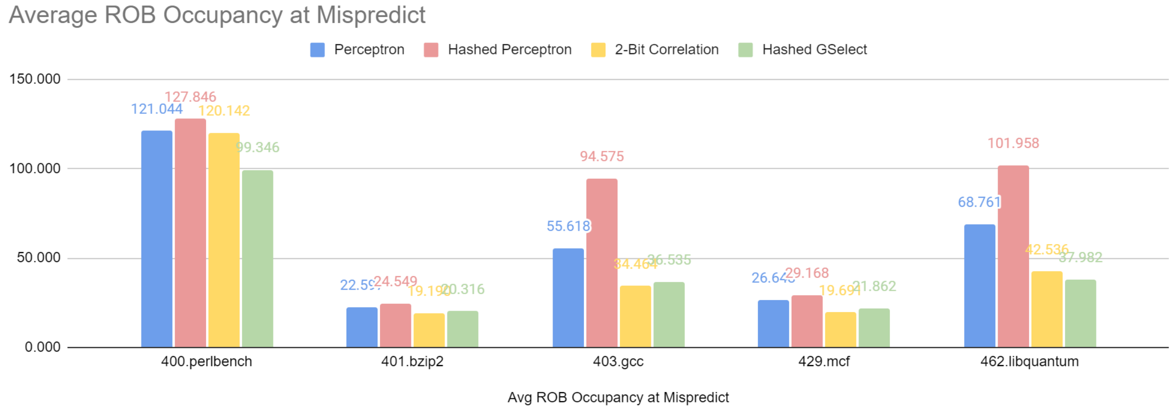

Figure 3.4: CPU_0_Average_ROB_Occupancy_at_Mispredict Comparison

3.2 Geomean

To calculate the geomean of each branch predictor IPC compared to Hashed perceptron, we first normalize the data to calculate individual speedups for each workload, then take the quintic root of all speedups multiplied.

$$Predictor\ Speedup\ over\ H.P.\ per\ Workload\ = \frac{Predictor\ IPC}{Hashed\ Perceptron\ IPC}$$

$$GEOMEAN(H.P,\ Predictor)\ = \sqrt[5]{WL1\ *\ WL2\ *\ WL\,3\ *\ WL\,4\ *\ WL\,5}$$

The resulting Geomean of each predictor compared to Hashed Perceptron IPC is shown in Table 3.1.

Speedup vs Hashed Perceptron	400.perlbench	401.bzip2	403.gcc	429.mcf	462.libquantum	**Speedup**
2-Bit Correlation	0.98665	0.95311	0.76236	0.96482	0.89657	0.90887
Hashed GSelect	0.98983	0.96703	0.79102	0.97705	0.88397	0.91856

Table 3.1: Normalized Workloads and Speedup

Appendix A: IPC, Branch Prediction Accuracy, MPKI, ROB Occupation

Cumulative IPC	Perceptron	Hashed Perceptron	2-Bit Correlation	Hashed GSelect
400.perlbench	0.338	0.341	0.337	0.338
401.bzip2	0.502	0.509	0.485	0.492
403.gcc	0.543	0.618	0.471	0.489
429.mcf	0.037	0.038	0.036	0.037
462.libquantum	0.185	0.201	0.180	0.178
Branch Prediction Accuracy	**Perceptron**	**Hashed Perceptron**	**2-Bit Correlation**	**Hashed GSelect**
400.perlbench	99.627	99.698	99.419	99.498
401.bzip2	92.969	93.626	91.591	92.042
403.gcc	96.949	99.435	93.727	94.532
429.mcf	92.991	93.529	89.866	91.659
462.libquantum	97.457	99.999	96.306	95.956
MPKI Percent	**Perceptron**	**Hashed Perceptron**	**2-Bit Correlation**	**Hashed GSelect**
400.perlbench	0.798	0.645	1.240	1.073
401.bzip2	12.397	11.210	14.903	14.105
403.gcc	6.619	1.227	13.610	11.862
429.mcf	20.270	18.713	29.307	24.122
462.libquantum	4.568	0.002	6.635	7.265
Avg ROB Occupancy at Mispredict	**Perceptron**	**Hashed Perceptron**	**2-Bit Correlation**	**Hashed GSelect**
400.perlbench	121.044	127.846	120.142	125.665
401.bzip2	22.597	24.549	19.190	20.535
403.gcc	55.618	94.575	34.464	39.788
429.mcf	26.648	29.168	19.691	22.143
462.libquantum	68.761	101.958	42.536	37.624