COMP3710 ASSIGNMENT 2

Branch Prediction

u6961332, u6940136

Table of Contents

1. Benchmark Suite Description

Before exploring the characterisation of various branch predictors, we present an overview of the benchmark suites and their trace details. There are four integer-based traces and four floating point traces. In our tests, we evaluated eight suites;

- Hmmer: Searching a gene sequence database [Integer]
- MCF: Combinatorial optimisation / single-depot vehicle scheduling [Integer]
- AStar: Path finding with A-Start algorithm [Integer]
- GemsFDTD: Computational electromagnetics (CEM) [Float]
- POVRay: Ray tracing visualisation [Float]
- NAMD: Parallelised molecular dynamics simulation [Float]
- OMNeT++: Discrete event simulation over large ethernet network [Integer]
- Leslie3D: 3D computational fluid dynamics simulation [Float]

All traces and their respective descriptions can be found here at: https://www.spec.org/cpu2006/Docs/

1.1 Hmmer

This benchmark implements search over a gene sequence database based on computational modelling of relations between sequences. This lends to high mathematical operation presence in the trace due to search operations utilising mathematical relations more so than comparative operations.

1.2 MCF

High quantities of numerical operations are present in this trace, as an optimisation program. The dominant behaviour is that of numerical optimisation, non-linear at that. Given that the program is modelling vehicle flow over multiple, varying restriction parameters, its behaviour is highly bound to numerical relation as opposed to stateful context.

1.3 Star

AStar is a pathfinding algorithm that has been adapted in three variants for this suite. Given that the algorithm is entirely self-contained, the dominant behaviour here is memory IO and numerical operations. This is due to the evaluation of conditions about the context of a graph or game map with regards to conditions of position and structure. It is likely that conditional evaluation is also highly correlated with this and less that of stateful context.

1.4 GemsFDTD

In this trace, the simulation of Maxwell's equations for electromagnetism are simulated for various conductors. This employs fine grained operations on the state of a conductor as mostly floating point operations. From the description of the trace, there are very few jumps between subroutines, meaning the algorithms are mostly self-contained. However, the implementation would suggest there is branching on internally held states for the conductor models.

1.5 POVRay

Scene ray tracing is implemented to simulate accurate light interaction with scene objects with reflection and refraction. Ray tracing implementations utilise predominantly linear algebra definitions for scene interactions and geometric objects. The majority of operations are between transforms, vectors and matrices in floating point precision. Instructions are heavily biased in favour of mathematical operations with branches being hard to predict as they are based on the results of maths operations and not that of container or structure definitions.

1.6 NAMD

In NAMD complex biomolecular systems are simulated with high levels of parallelism. We can expect that depending on the architecture, the context switching behaviour can be problematic, especially in the scenario of a small physical thread count versus the required thread count. Branch prediction with this data set could become unpredictable as a result.

1.7 OMNeT++

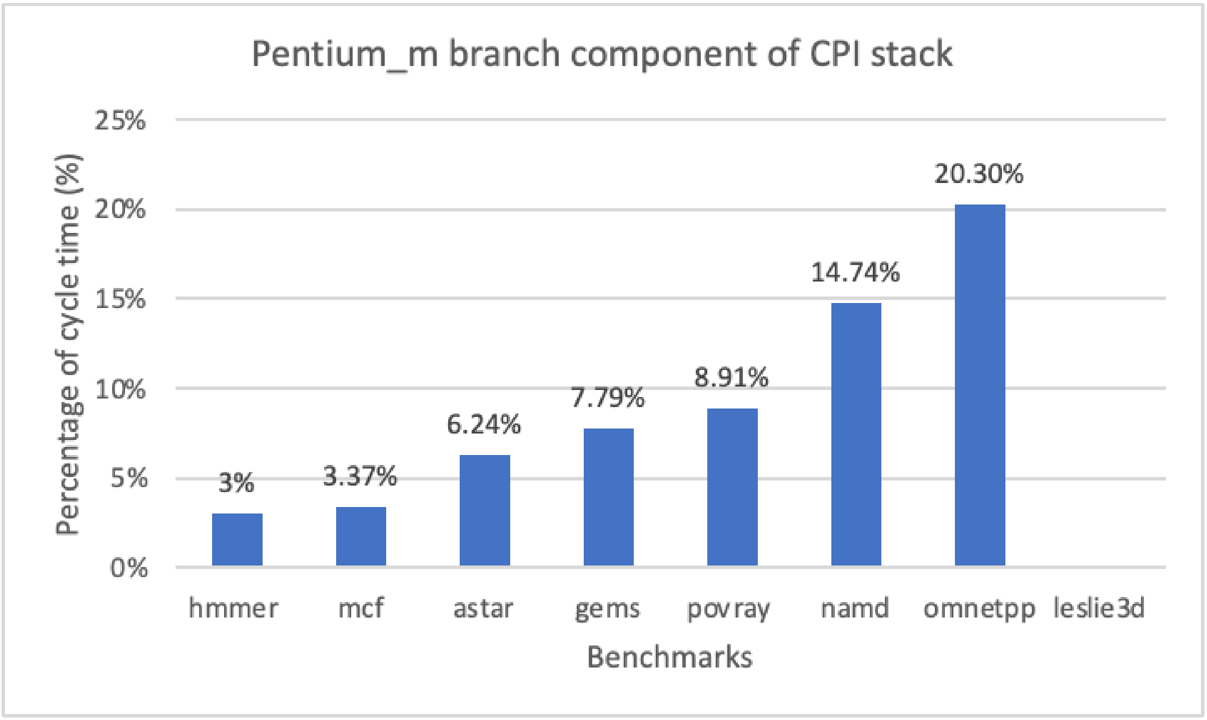
Simulation of large ethernet networks is complex and requires large state tracking. For this trace, the processes simulated are that of switches at nodes in a network. We can expect to see complex correlated behaviour in the logic of the program. This potentially does not scale well for predictors that lack an ability to analyse discrete behavioural changes as opposed to relational, localised changes. It should be noted that there are low mathematical operations to be performed here as the model is based around the request/response behaviour for network interconnections, lending well to stateful behaviour and predictions.

1.8 Leslie3D

Turbulence and fluid mixing modelling is a computationally heavy topic in physics. LESLie3D models these behaviours in the description of mixing between fluids in a 3D environment. The vast majority of calculations are calculus and linear algebra, lending toward a high dominant count of mathematical operations and memory operations for simulation state. It is likely that branching will be locally correlated due to large matrix and array operations with predictable iterative patterns.

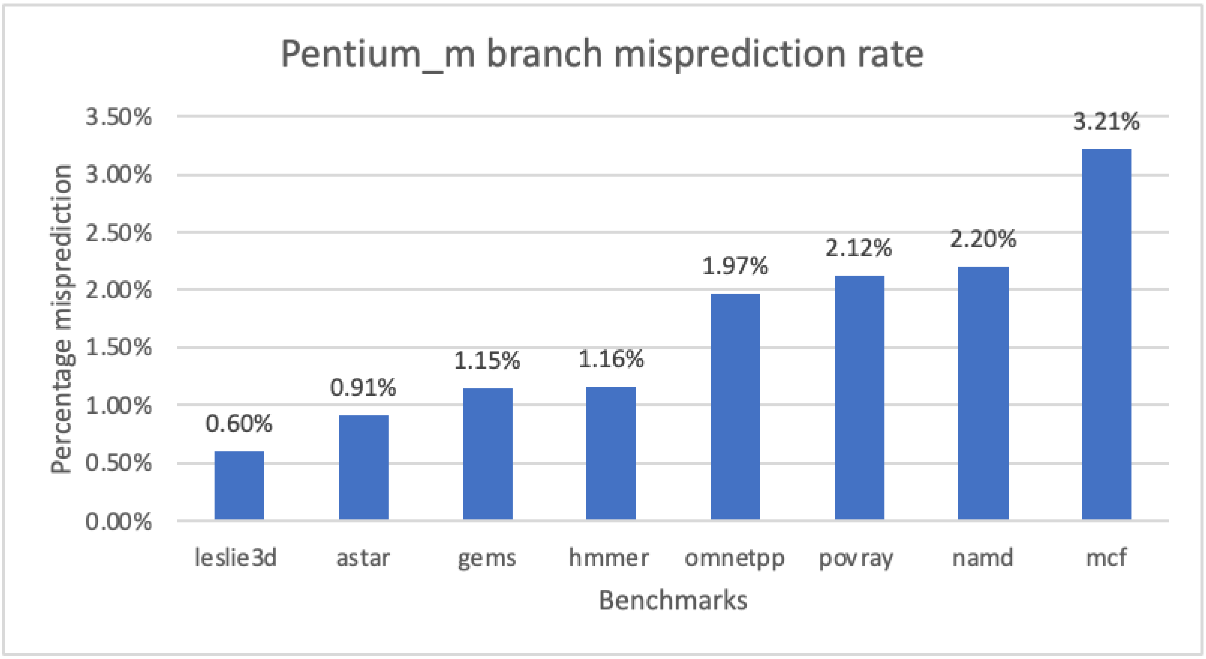
2. Pentium_m branch predictor

2.1 Part 1



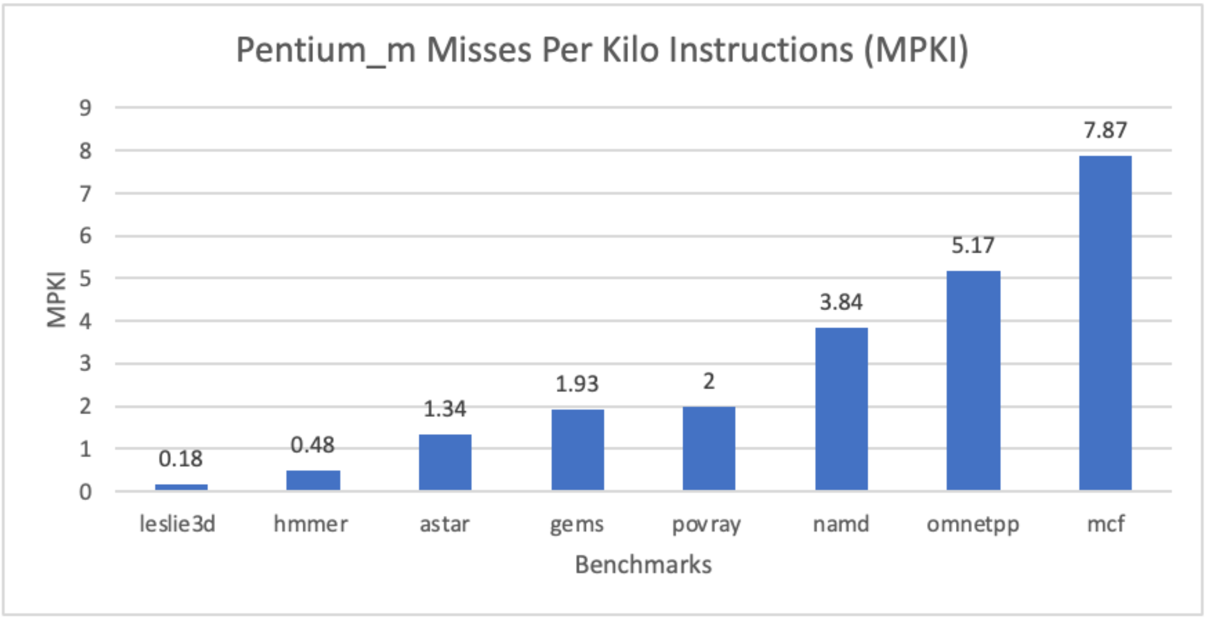
Graph 1: Pentium_m branch predictor's percentage of cycle time for each benchmark

2.2 Part 2



Graph 2: Pentium_m branch predictor's misprediction rates for each benchmark

2.3 Part 3



Graph 3: Pentium_m branch predictor's Misses Per Kilo Instructions (MPKI) for each benchmark

2.4 Part 4

Observations:

- A lower percentage of cycles per instruction spent on branching corresponds to a lower MPKI for all benchmarks. This is because a small number of branching operations would result in fewer misses across all instructions.

- For most benchmarks including Namd, OMNeT++, POVRay, AStar and GEMS, a higher branch time in the cycles per instructions corresponds to a higher branch misprediction rate and MPKI.
  - OMNeT++ models a request-response relationship between nodes in a graph to simulate traffic flow. This implicitly has a lot of conditional evaluation both for evaluating packet context but also for jumping to relevant processing handlers. We can see that this is reflected in the branch percentage being relatively high, of approximately 20%. Most of these branch predictions will be regular in their appearance given there are a very limited subset of possibilities that can be evaluated when considering network context, which the miss rate would agree with.
  - POVRay, AStar and GEMS are similar in an algorithmic context, in that they have localised branching context that is often repeated. The caveat here is with the POVRay trace, where the calculation handler varies within that localised context, such as whether to calculate a diffuse, specular or reflective property and whether to stop a re-trace ray.
  - AStar provides an interesting insight into the types of regular branching that can be easily predicted. Given that AStar, as an algorithm, interprets map topology as a graph, there is a very limited subset of evaluation that needs to be done in that context. In particular, the decisions around which path to take next reoccur often.

- The MCF benchmark has the second lowest percentage of cycles spent on branching per instruction. It does, however, have the highest rate of branch misprediction and Misses Per Kilo Instructions (MPKI) of any benchmark. This is due to irregular branching behaviour that is restricted to the mathematical context of a specific optimisation condition. Because the entire system is numerically optimised and branches are computed where necessary in irregular condition placements, the results are difficult to quantify.

- Hmmer has the lowest percentage of cycles per instructions spent on branching but produces a relatively high branch misprediction rate and low MPKI compared to other benchmarks. Since hmmer is a gene sequencing database, it would have less repeating branches which increases the likelihood of mispredictions. HMMer uses Hidden Markov Models to search through sequences, highlighting patterns that occur relative to some target. This is mathematically intensive as matrix reductions with parallelised Gauss-Jordan elimination take the precedent of these operations. Relative to branch predictors, there are substantially less branches to predict and much less those of the search content itself and affectuals therein. Given that search elements are scored and then fits it against an extremal value decomposition histogram for priority, we see little to no opportunity for prediction given that this is entirely defined within the bounds of mathematical operation. Relative to a perfect branch predictor, there would be little to no noticeable improvement in the bulk of the application's execution for all the reasons stated above.

All benchmarks produce a variety of results due to the nature of their data as mentioned in points 1-4. Pentium_m seems to perform very well for data which provides regular branching behaviour that can either be learned through localised context and consistency in periodicity between contexts. However, branches that depend on numerically generated conditions are complex to predict and lead to indeterminate behaviour that cannot be modelled with a limited set of variables in hardware.

Compared to a perfect branch predictor, the mcf benchmark suffers the most severe degradation in performance due to pentium_m. As mentioned previously, mcf is used for combinatorial optimisation / single-depot vehicle scheduling and has a high number of mathematical operations in this data. Comparisons of numerical results tend to have less repeating branches and are difficult to predict, which causes this irregularity in branch mispredictions and MPKI.

2.5 Part 5

Given that each of the metrics are interdependent it is hard to define an absolute metric of performance. However, we can expand on an existing performance metric of execution time. In doing so we can highlight which metrics are valuable and which do not contribute. The primary definition of execution time is as follows

ET = I * CPI * SPC

Where CPI is cycles-per-instruction and SPC is seconds-per-cycle. We have three metrics at our disposal, all of which are in terms of relative instruction time occupancy. Miss rate as a percentage of CPI stack (let this be MR), misses per kilo instructions (MPKI) and branches as a percentage of PCI (BCPI). In the execution time model, we can adjust the CPI term to remove the branch misses as a singular component. Let CPI_M denote the CPI for only branch misses

```
CPI_M = CPI − (CPI * BCPI * MR)
```

In doing so, we can compare the relative performance impact between inclusion and exclusion of branch misses. We define this as ET_BM (execution time of branch misses).

```
ET_BM = I * CPI_M * SPC
```

Utilising this, we can create a metric of predictors in terms of their performance impact on execution time with regards to misses as a percentage where higher is better.

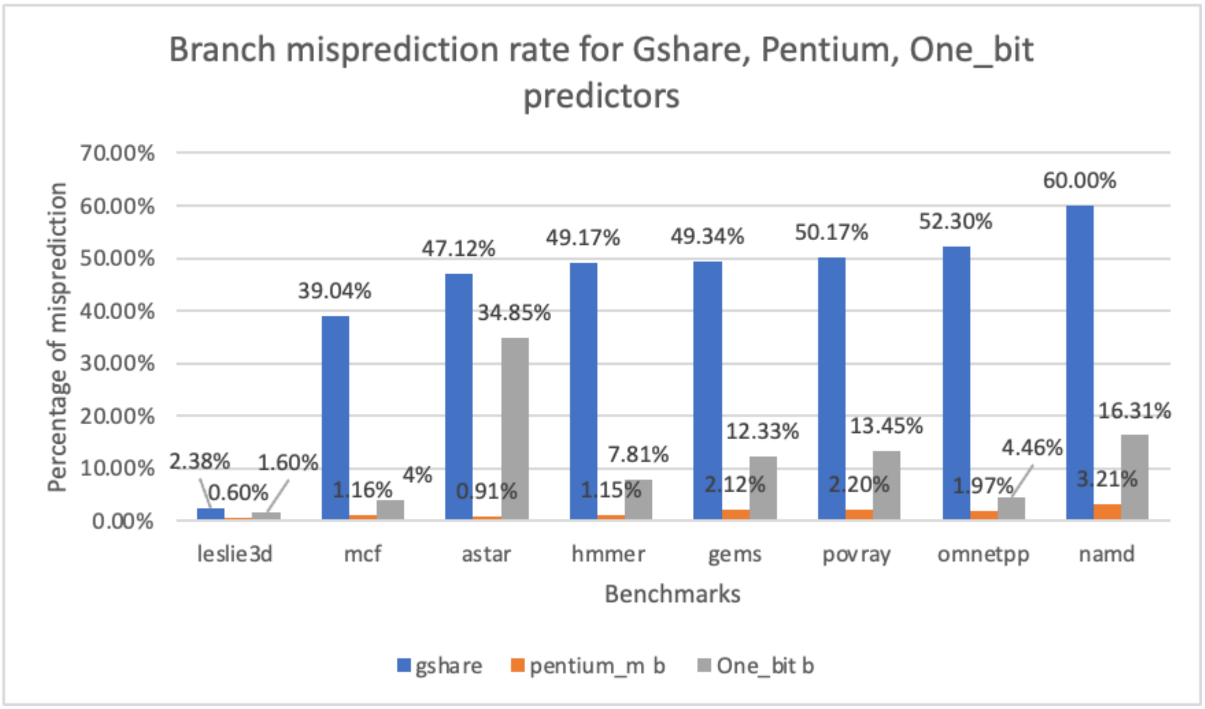
```
ET_Bperf = ET_BM / ET = 1 / (1 − (BCPI * MR))
```

Note that in our definition, MPKI was not included. This is because it is an approximative metric over an aggregation of a thousand instructions. Given that we want a metric of absolute performance, this aggregation does not provide the accuracy required. Using the more precise value per instruction provides a higher precision metric.

In order to show this as a graph, additional data needs to be collected from Sniper in the format of the instruction count, base CPI and SPC. With that we can generate values for ET_Bperf using the above formulation. We utilise the existing graph structure, where benchmarks are on the X-axis and the ET_Bperf value is the Y-axis, maintaining each column as a distinct predictor. In doing this we can evaluate the relative cost in performance of each predictor to verify which is more performant in general. The X-axis values should be sorted from high to low, given that a higher ET_Bperf value indicates a better predictor.

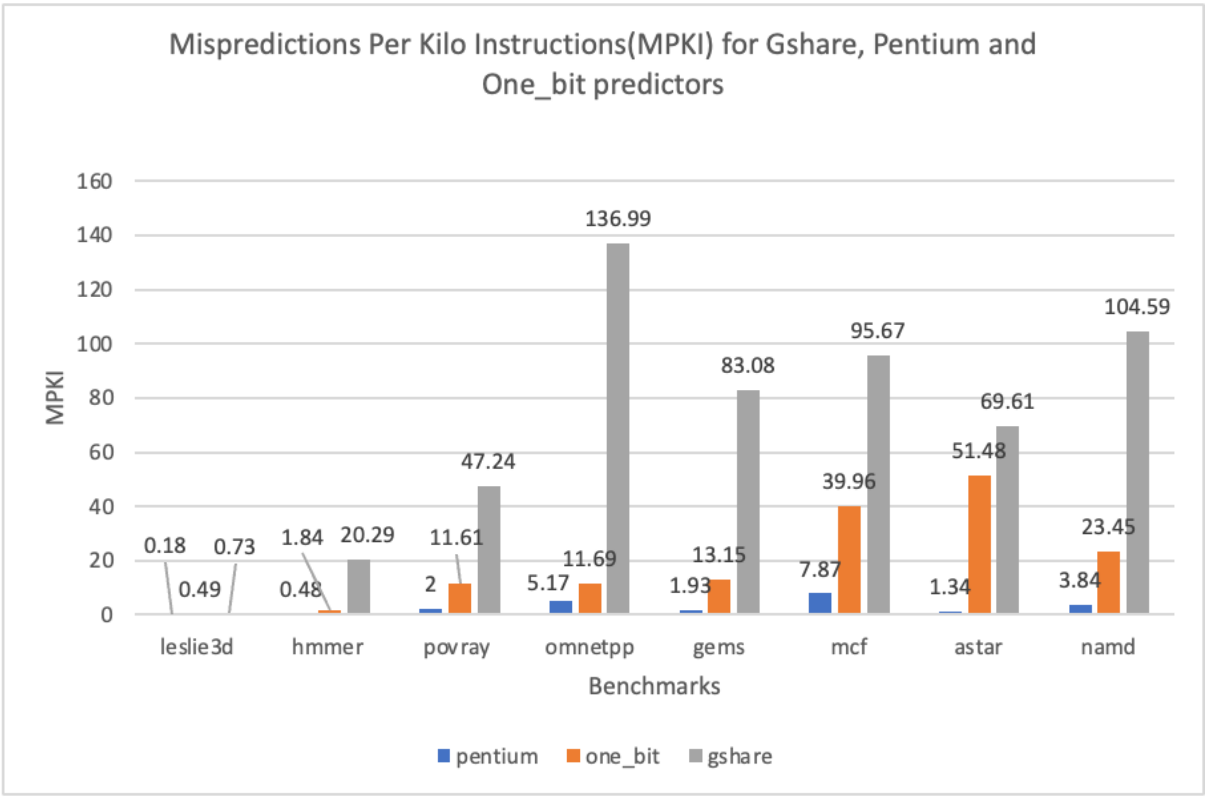
3. GShare predictor

3.1 Part 1



Graph 4: gshare branch predictor's branch misprediction rate for each benchmark

3.2 Part 2



Graph 5: gshare branch predictor's Misses Per Kilo Instructions (MPKI) for each benchmark

3.3 Part 3

Our implementation of ghsare has the following configurations:

```
PHT size = 4096 bits,
BHR size = 12,
Address = 12
```

The hardware budget of 1024 bytes is fully dedicated to the predictor's PHT, and thus gshare would perform best if the PHT has the maximum number of entries possible for the given budget. Each entry in the PHT table has two bits that indicate whether or not the branch is taken. As a result, the PHT size was calculated as follows:

```
Size = (1024* 8 bits) / 2 bits per entry
     = 4096 entries
```

The address must have enough bits to index 4096 entries as it is used to index the PHT table. Therefore, number of address bits was calculated as follows:

```
Address length = log2(number of entries)
               = log2(4096)
                = 12
```

Any length larger would be a waste of hardware space, and any smaller would not be sufficient to index all of the elements in the PHT table.

Before referencing the PHT, the BHR is XORed with the address and therefore any bits in the BHR that exceed the address length will be unused. As a result, the size of the BHR is limited to the address length of 12 in our design.

3.4 Part 4

Observations:

- Over all benchmarks, GShare has a higher branch misprediction rate and MPKI than pentium m and one bit. Pentium m and one bit outperform GShare because:
    - One_bit recovers faster from mispredictions than GShare which uses 2-bit saturated counters for counters
    - Unlike GShare, Pentium m has both local and global history, as well as the ability to learn from mispredictions.

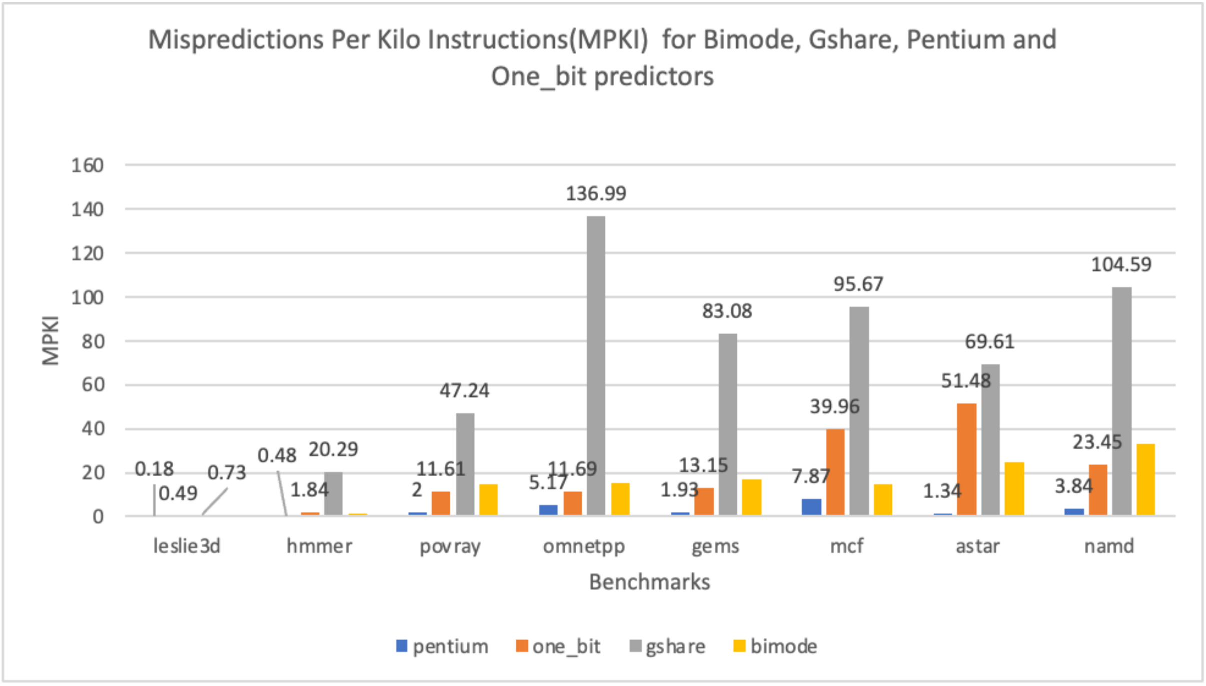
- When compared to other predictors, the mcf benchmark has a much higher misprediction rate for gshare. As previously stated, mcf exhibits irregular branching behaviour, with branches computed where needed in irregular condition placements, making branches difficult to predict. GShare's inability to recover well from mispredictions makes tracking accurate predictions over infrequent branches more difficult.

- The OMNet++ benchmark suffers the most degradation in performance with branch misprediction and MPKI due to the gshare predictor. OMnet++ requires large state tracking over networks and much of the branching operations are deterministic. Since gshare is only a two bit saturated counter, it does not recover well from mispredictions and is unable to achieve a high accuracy with repeating branches.

4. Bimode predictor (Full)

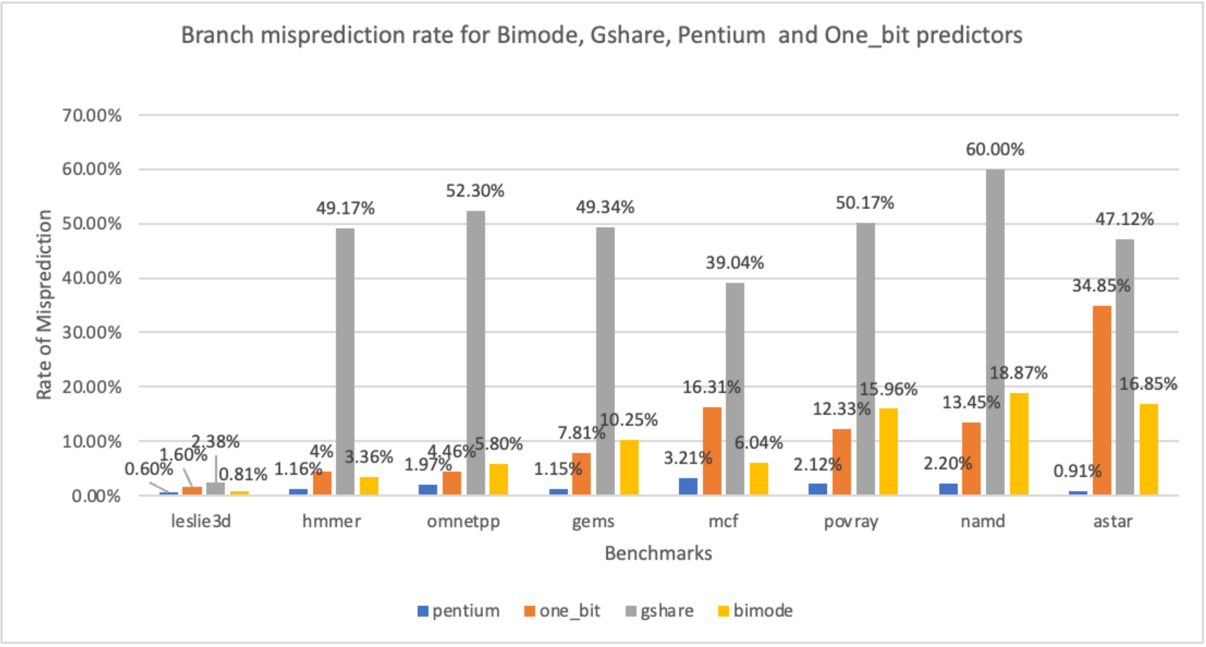
4.1 Part 1

We utilised full update for our implementation of the bimode predictor, given that most cases of branching behaviour should adjust to circumstance. Since we have a dual PHT system where one or the other is referenced based on the result of the choice value, these should reflect the actual branching behaviour. If both PHTs update with only correct values, then incorrect values are persistent in between rectifying the entries of the PHT. In theory this degrades the performance and accuracy of the predictor.



Graph 6: Bimode predictor's branch predictor rate for each benchmark

4.2 Part 2



Graph 7: Bimode predictor's Misses Per Kilo Instructions (MPKI) for each benchmark

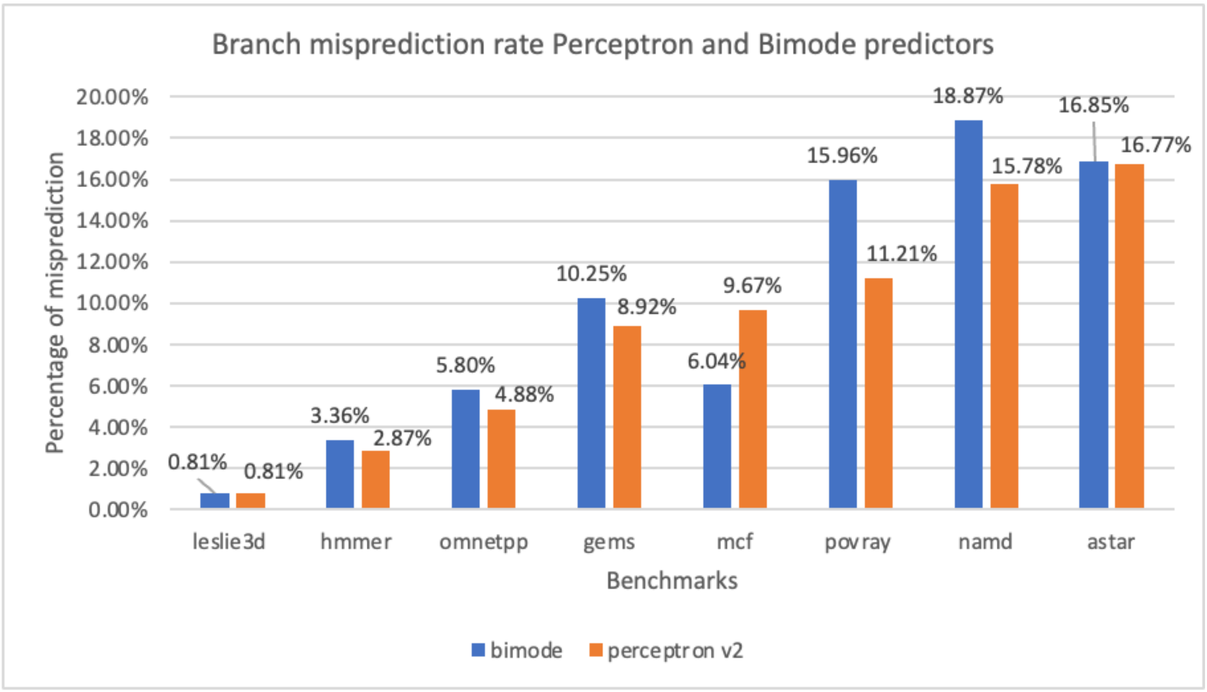
4.3 Part 3

Observations

- The AStar trace provides a roughly equivalent miss rate to that of the POVRay trace, despite there being substantially less MPKI for POVRay compared to AStar. This would suggest that there is more inaccuracy in a smaller amount of branches with AStar that don't lend well to bimode. Conceptually this could be attributable to the AStar algorithm using more branches than that of POVRay, mostly due to the predominantly mathematical computation required for ray tracing.
- We can see that bimode produces better hit rates than one bit in the integer based tests (excluding OMNeT++), and exceptionally does so in the floating point test for LESLie3D. We could conclude that floating point traces have less locally deterministic branching than that of the integer based tests. Which bimode lends well towards in comparison to one bit.
- Pentium M consistently outperforms bimode in all traces. Simply put, bimode's limitation on being able to learn irregular patterns restricts its accuracy and overall ability as a predictor.

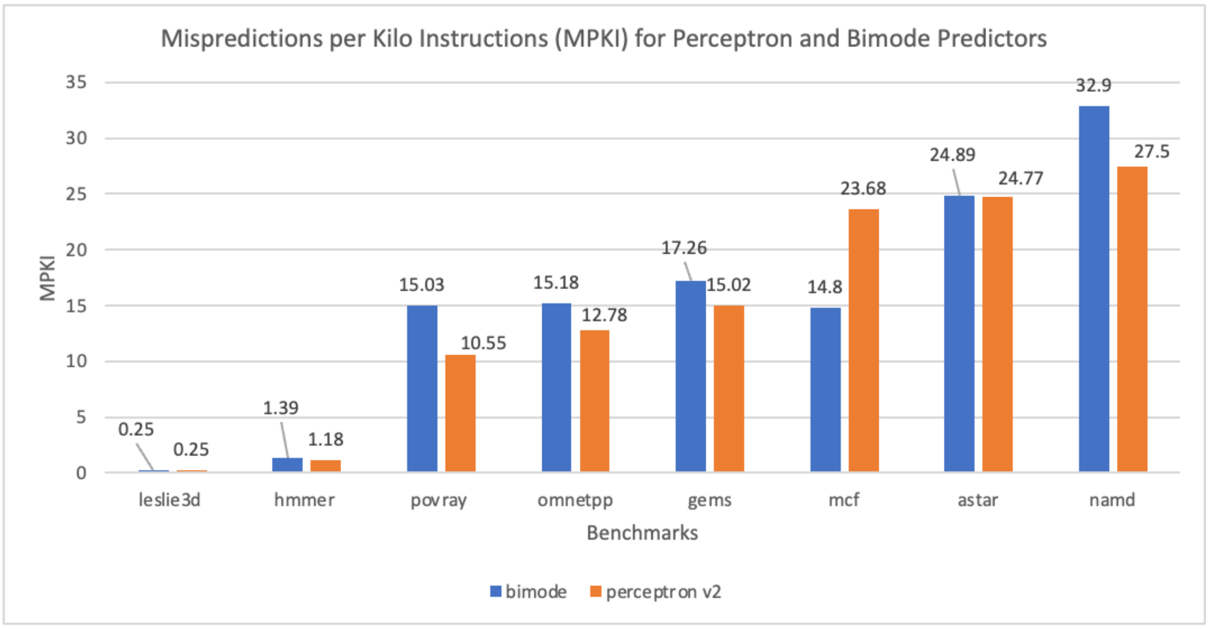
5. Perceptron predictor

5.1 Part 1



Graph 8: Perceptron predictor's misprediction rate for each benchmark

5.2 Part 2



Graph 9: Perceptron predictor's Misses Per Kilo Instructions (MPKI) for each benchmark

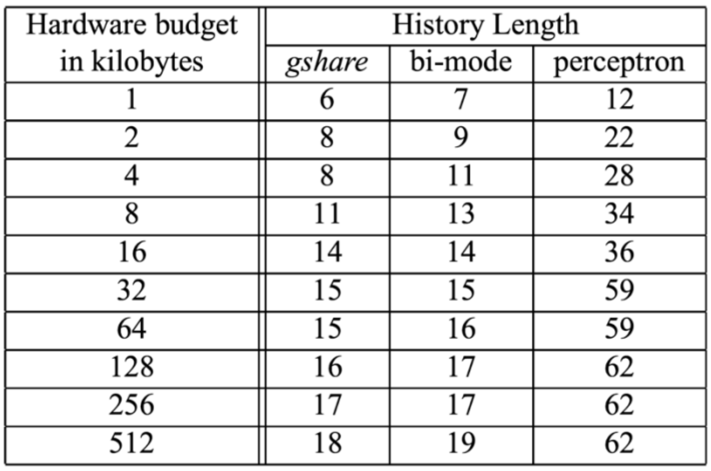
5.3 Part 3

Perceptron Version	No. of weights	No. of Perceptrons	Weight size (bits)	Threshold
1	4	4	48	1
2	12	97	7	37
3	13	90	7	39

Table 1: Different configurations for perceptrons tested against gshare and bimode predictors

Of all three versions of the perceptron predictor, version 2 was shown to have the most optimal configurations with a BHR length of 11, 4 perceptrons, weight size of 7 bits and threshold of 37. Prior work by Jim'enez et al. evaluated the parameters for the perceptron predictor compared to GShare, Bimode and others. Our configurations leverage their work to determine optimality around the performance of the perceptron predictor.

According to this paper, the optimal number of weights per perceptron for a 1k hardware budget is 12 because longer history lengths can increase the accuracy of predictions but too many can reduce the number of perceptrons.



Hardware budget in kilobytes	History Length		
	gshare	bi-mode	perceptron
1	6	7	12
2	8	9	22
4	8	11	28
8	11	13	34
16	14	14	36
32	15	15	59
64	15	16	59
128	16	17	62
256	17	17	62
512	18	19	62

Table 2: Best History Lengths. This table shows the best amount of global history to keep for each of the branch prediction schemes. (Jim´enez & Lin, Dynamic branch prediction with Perceptrons - University of Texas at Austin)

The paper also found a relationship between history length and threshold for training where the optimal threshold (ε) for a history length (h) is exactly ε = 1.93h + 14 (Jim´enez et al. 2002). Based on this calculation, the the best threshold for a history length of 12 for our perceptron is `ε= 1.3*12 + 14`, which gives a result of 39.

The bit length of each weight was chosen to be 7 because the paper found that this is the most optimal bit length for a history length of 12.

The number of perceptrons were calculated using the following equation:

```
no. of perceptrons * no. of entries * bits per entry <1024 bytes

no. of perceptrons = 1024 * 8/ (no. of entries * bits per entry)
                   = (1024 * 8)/ (12 * 7)
                   = 97.52
                   = 97 (rounded)
```

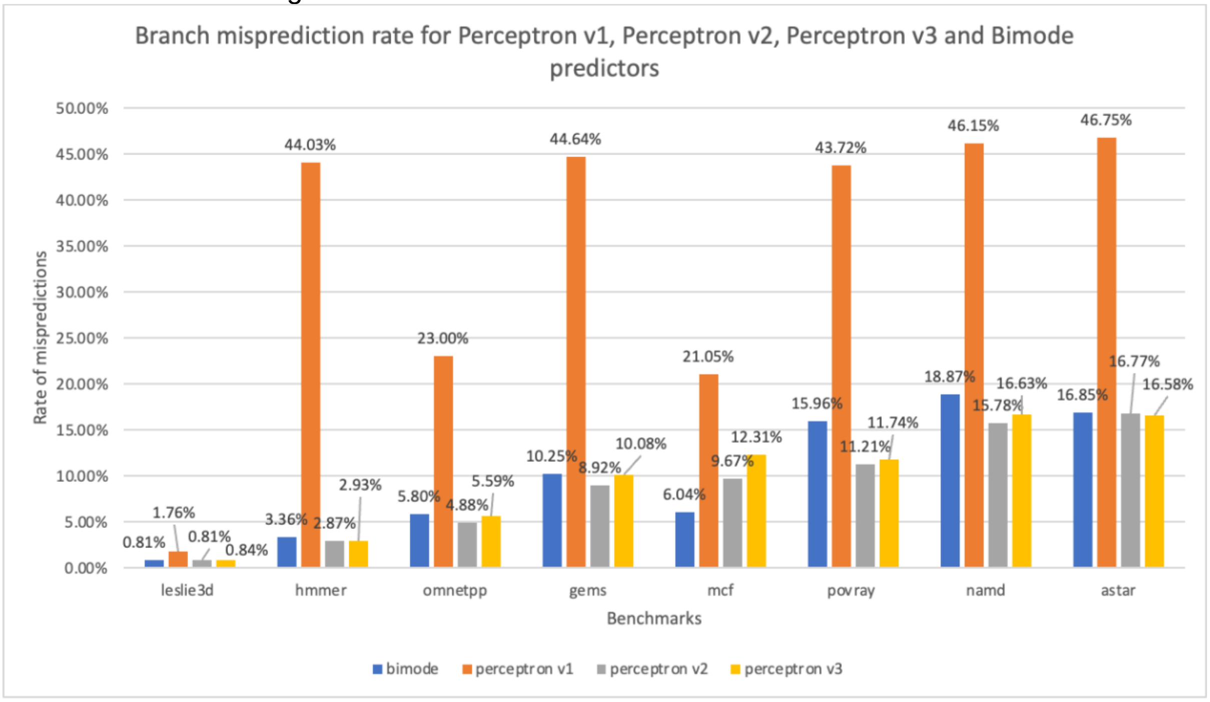
When these configurations are combined, the perceptron predictor has the lowest misprediction rates and MPKI when compared to bimode predictors on 7 of 8 benchmarks.

In graphs 10 and 11, the misprediction rates and MPKI for the final perceptron predictors (v2) are compared to the other versions. Version 2 of the predictor performs significantly better than the other versions across all benchmarks for the following reasons:

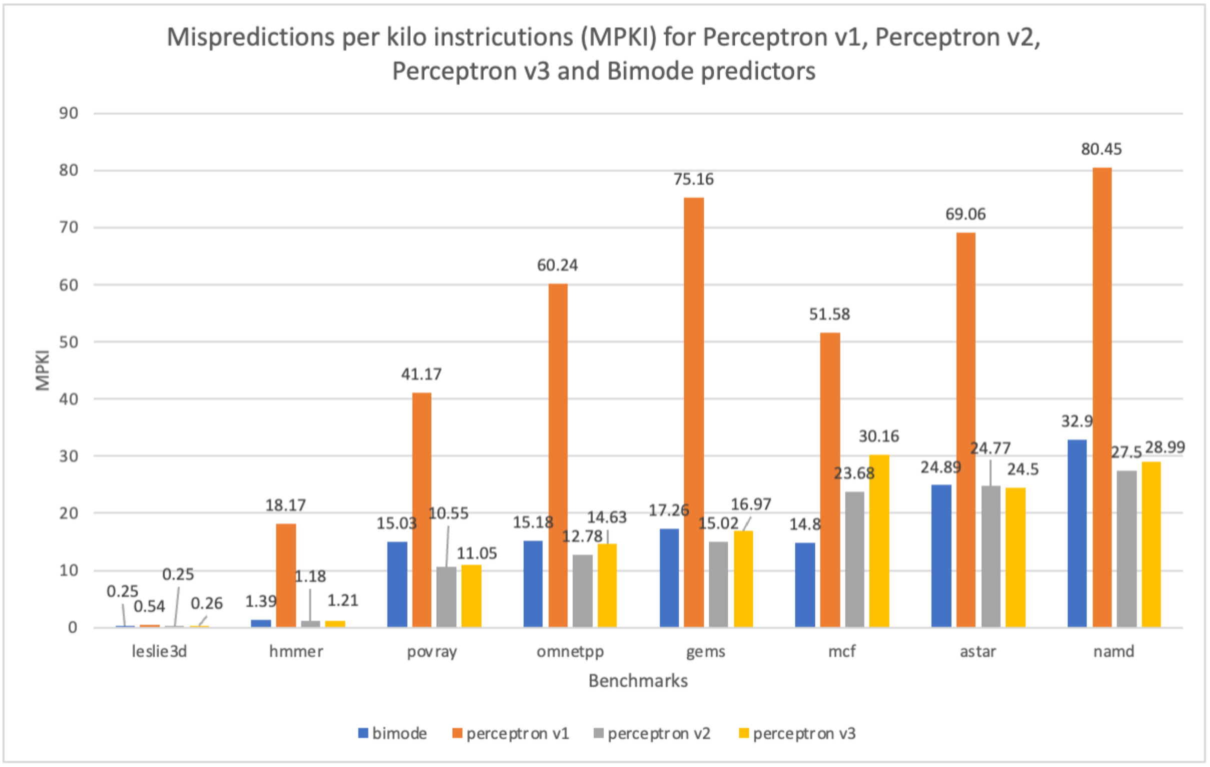
- Longer history lengths produce more accurate predictions but reduce the number of table entries, resulting in overwriting. Version 1 has a short history, a small number of perceptrons, and longer entry lengths (bits). Many of the bits in each entry are unused and the smaller number of perceptrons provide less data for training to improve accuracy.

  Using a larger history length of 13 for version 3 degraded the misprediction rate and MPKI. Performance is comparable worse than that of the Bimode predictor for benchmarks such as LESLie3D. Due to the larger history length, it reduces the number of perceptrons to 7 which again provides less data for training. Therefore, the history length of 12 in the final version of the perceptron provides the best balance of branch history and number of perceptrons.

- Version 1’s threshold of 1 does not sufficiently define a lower bound on inclusive weights for retraining. We see that weights are more often adjusted in favour of poorly predictive values, leading to further inaccuracy. Version 3 utilises a threshold of 39 which for most weights in perceptrons, exceeds their value at any given point in time. Granted this not universally true, however it does apply in more cases than that of version 1. Version 2 implements a threshold that achieves the highest accuracy because it is neither too restrictive of accurate data nor too permissive of inaccurate data for training.



Graph 10: Comparison of branch misprediction rates over all versions of the perceptron predictor



Graph 11: Comparison of MPKI over all versions of the perceptron predictor

5.4 Part 4

Observations:

- The misprediction rate and MPKI for the perceptron predictor are lower than these rates for the Bimode predictor for the NAMD, OMNet++, POVRay, GemFDTD, AStar, HMMer and LESLie3D benchmarks.The performance of the perceptron predictor is better than the Bimode predictor for the following reasons:

    - Perceptron predictors have a larger history length than the Bimode predictor which means it can look for correlated branches over much larger distances

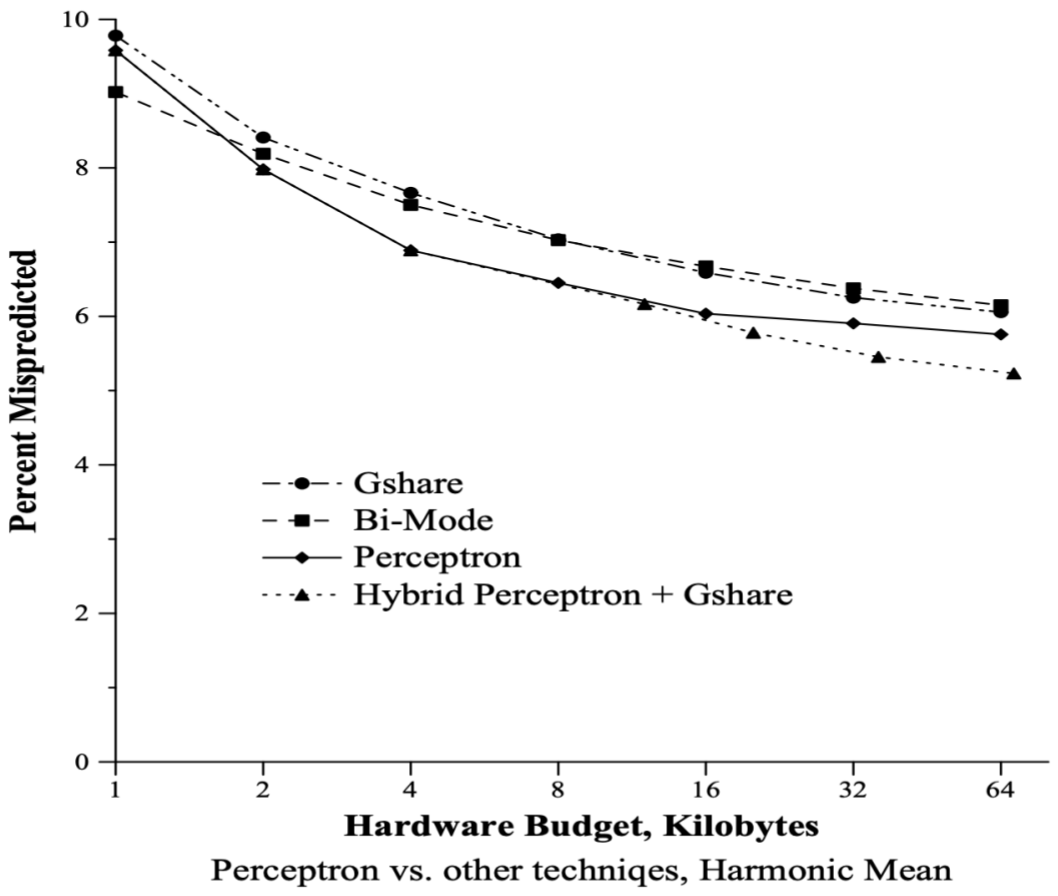
    - They provide a higher confidence over whether a branch is taken or not-taken compared to a Bimode predictor which only has the states taken or not-taken. The weights of the perceptron are non-binary which provides a more accurate prediction of which branch will be taken

    - Since the benchmarks listed above include operations with many of the same recurring branches, the training algorithm increases prediction accuracy while lowering the misprediction rate.

- The perceptron predictor's MPKI and misprediction rate are greater than the Bimode predictor's values for the MCF benchmark. As was discussed in the benchmark characterisation section, MCF uses a lot of mathematical operations on the data and is utilised for single-depot vehicle scheduling and combinatorial optimisation. This inconsistency in branch mispredictions and MPKI is caused because the numerical results comparisons in the data typically include fewer repeated branches and are difficult to predict. As a result, when it comes to non-repeating branches, the training algorithm used to increase perceptron predictor accuracy has no advantage over a Bimode predictor.

- Contrary to the other Perceptron predictor benchmarks, POVRay has a low MPKI but a relatively high misprediction rate. Instructions in the POVRay are mathematically heavy with branches being hard to predict as they are based on the results of maths operations and not that of container or structure definitions. They have less branching operations which is the cause for the smaller MPKI. The difficult to predict branching operations are the cause for high misprediction rates. These findings hold true for both bimode and perceptron predictors, indicating that neither has an edge over the other in terms of delivering superior performance on difficult-to-predict data when compared to other types of data.

5.5 Part 5



(Jim´enez & Lin, Dynamic branch prediction with Perceptrons - University of Texas at Austin)

With regards to the perceptron performance with a hardware budget of 1024 bytes, we saw our predictor make significant improvements over that of the Bimode predictor. It is interesting to note since this is in stark contrast to the 1K results of the original perceptron paper (Jim'enez et al. 2002). In the above graph we can see that their results showed perceptor being outperformed by bimode. However, it is important to note that this is over the harmonic mean trace, which we do not have in our test configurations.

Given that we utilised the same configuration for perceptron, in our work compared to the paper, this could suggest that complex branching in mathematically dense traces could be problematic for the perceptron. More specifically, that recency between predictable branching behaviour does not follow an easily learnt pattern. Bimode may be able to address this in regards to its focus on localised optimisation.

In general however, our observations would suggest that the perceptron predictor will consistently outperform bimode, gshare and one bit given its ability to learn more complex patterns. This is shown through our results as well.

6. References

1. Jim´enez, D.A. and Lin, C. 07 August 2002, *Dynamic branch prediction with Perceptrons - University of Texas at Austin*, *cs.utexas*. Available at: https://www.cs.utexas.edu/~lin/papers/hpca01.pdf (Accessed: October 29, 2022).
2. Henning , J. (no date) *SPEC CPU2006 Documentation*, *Spec CPU2006 documentation*. Available at: https://www.spec.org/cpu2006/Docs/ (Accessed: October 29, 2022).