CS 259 Report: Neural Branch Predictors

Cyrus Tabatabai-Yazdi, Akshay Shetty, and Tyler Davis

Abstract— **Branch predictors are essential for achieving high performance via instruction-level parallelism by alleviating the need to wait for branch instructions to move down the pipeline before being able to fetch the next instruction. With the rising prevalence of machine learning techniques, branch prediction seemed an apt application for such a system that could automatically learn patterns without outside assistance. In this project, we explored the use of perceptron-based branch predictors in order to explore their performance and latency compared to traditional branch predictors. We implemented and evaluated a simple perceptron that takes a linear combination of weights with the global history as well as a path-based perceptron that incorporates the path taken to a branch in order to stagger computation for greater efficiency.**

I. INTRODUCTION

In order to achieve high throughput, modern CPUs are often pipelined, allowing for several instructions to be processed at once, with each instruction undergoing a different stage of execution at any particular time. While pipelining allows for higher throughput due to increased instruction level parallelism, it also introduces a number of difficulties. One of these difficulties is the introduction of control hazards. When a pipelined processor processes a branching instruction, the processor will not know the address of the next instruction to fetch until the branching instruction has progressed sufficiently far down the pipeline. In a naive implementation without a branch predictor, this means that the pipeline must remain stalled until the new value for the program counter is known, resulting in dramatically reduced performance.

To alleviate this issue, CPUs may make use of speculative execution, wherein they make a guess as to which instructions will be executed after the branch instruction. If the CPU guesses correctly, then the computation was done at full speed without any drawbacks. However, if the CPU guesses incorrectly, it must then flush the pipeline and reset to a previous state before beginning computation from the proper instruction. Branch predictors are a component of CPUs that aim to enhance the accuracy with which the next instruction is predicted. Branch predictors make predictions as to whether or not a branch will be taken, thus allowing a CPU to more rapidly fetch either the next sequential instruction, or the instruction that the branch will lead to.

With today's superscalar processors and deep pipelines, typically with ten or more stages, achieving high accuracy is crucial as the penalty of a misprediction could be severe. Since conditional branches occur around every 7-8 instructions, it is imperative that branch predictors perform well as the penalty as a failed prediction leads to a flushed pipeline. Today, there are two main types of branch predictors: static and dynamic predictors. Static predictors make all decisions at compile time and do not react to a program's behavior at runtime. One example of static branch prediction can be found in some implementations of MIPS, where all branches were always assumed to never be taken. A dynamic branch predictor uses information gathered at runtime about whether or not past branches have been taken in order to predict the outcome of the current branch. Neural network based branch predictors have been proposed as an addition to dynamic branch predictors that can automatically learn a mapping between the global history and whether a branch will be taken. However, the main drawback with neural network predictors has historically been their latency, which is imposed by the requirement for the predictor to run prediction and update algorithms. Since many modern methods already achieve high levels of accuracy, an increase in accuracy at the expense of prediction latency did not seem like an acceptable trade-off. At the same time, research by Daniel Jimenez et al. has shown that perceptron based predictors are comparable to more typical dynamic predictors in terms of latency and accuracy. Our goal in this project is to implement two perceptron based predictors in order to validate the claims through evaluation and comparison to traditional branch predictors.

II. RELATED WORK

Branch Predictors, including neural based predictors, have been the subject of a great deal of research due to their importance.

At their simplest, traditional dynamic branch predictors may make use of a variety of prediction methods. One of the simplest methods for branch prediction is to keep track of whether or not the last branch was taken, and then assume that the current branch will have the same behavior as the last branch. While this strategy works, it is also susceptible to a single branch result out of the ordinary pattern resulting in the next branch being mispredicted as well. To remedy this, a branch predictor may make use of a saturating counter. A saturating counter is essentially a state machine with 2^n states, where n is the number of bits being used to store state. An example of such a state machine is shown in 1. In this system, when a branch is taken, it moves to a more strongly taken state, and when a branch is not taken, it moves to a more weakly taken state. The introduction of additional states provides resilience against branches outside the typical pattern affecting the ability to correctly predict a subsequent branch that falls within the typical pattern. One limit to these simplistic approaches is that they are not capable of

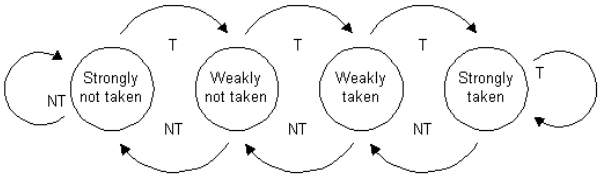

Fig. 1. A two bit saturating counter branch predictor [22]

learning complex patterns and instead simply predict the most frequently taken direction.

One improvement to these prior designs, which are known as one-level predictors, is the introduction of a table containing multiple saturating counters from which a branch predictor is able to select the most relevant one. Such systems, known as two-level branch predictors, were first proposed in 1991 by Yeh and Patt. [23]. The methods with which an entry in the table is selected can vary, but they can include indexing with the program counter or the branch history. The history buffer that is kept in these situations can be either a global buffer, where a single buffer is maintained which records the outcome of all branches, or local, where each conditional branch instruction has its own history buffer. Predictors that make use of local history can often make better predictions than those that make use of a global history buffer, but implementing one is often impractical due to the increased requirements to keep significantly greater amounts of history.

Systems in which the entry is selected by XORing a global history buffer with the program counter to index into a table of two bit saturating counter are termed $gshare$ predictors.

A further advancement in branch predictor designs was the introduction of hybrid predictors, first proposed by Scott McFarling in [21]. Hybrid predictors incorporate multiple methods for predicting branch outcomes, and then incorporate a system to combine the results obtained for a final prediction. This mechanism may include some form of system that keeps track of which mechanism has historically provided the best prediction for a given branch.

One of the best performing modern branch predictors, TAGE, an acronym for (partially) TAgged GEometric history branch prediction, was first introduced in 2006 by André Seznec and Pierre Machaud [11]. The innovation behind TAGE is that it incorporates multiple history buffers, each of a different length. The branch predictor then keeps track of the usefulness of the predictions obtained with each of these buffers and uses this information to inform future predictions, similar to the approach taken by hybrid predictors. TAGE also makes use of tagging in order to determine whether a particular table entry actually corresponds to the current program counter. The use of a range of history lengths enables TAGE not only to warm up quickly, which is typically a weakness of long history lengths, while also enabling it to learn longer and more complex patterns that are simply not possible to learn with shorter history lengths. The TAGE family of predictors has been repeatedly revisited in an effort to improve its accuracy as well as the feasibility of implementing it in the real world [18] [17] [16] [12].

In an effort to attain higher accuracy, some hybrid predictors include prediction mechanisms designed especially for specific cases. One such example would be the use of loop predictors. Loop predictors aim to recognize a situation in which a branch is taken many times in a row, after which it is not taken once. These systems have received attention from academia, and have been incorporated into some modern predictors and microprocessors [15] [17] [20].

Compile time neural prediction was studied in detail by Calder and team [9]. Vintan et al. proposed the dynamic branch prediction using neural networks [8]. While static predictors provide worse predictions than dynamic predictors, information gleaned at compile time is crucial for performing compiler optimizations, and it may also provide additional useful information to dynamic predictors which might not have been available otherwise.

Neural networks learned through evolutionary computation have been proposed as a method for managing on chip resources for chip multiprocessors [2]. Daniel A. Jiménez provided the basis for the first perceptron based neural predictor in 2001 [3]. The predictor in this paper makes its predictions through the use of a perceptron, which is a simple single layer and single neuron prediction system. The paper discussed how the perceptron takes advantage of long history lengths better than gshare and bimodal predictors. Schemes that use the history register as an index into a table, such as gshare, require space exponential in the history length, whereas the perceptron predictor requires space linear in the history length. The perceptron based neural predictor performed better than gshare when the length of the history was increased. In 2003, Jiménez followed up on his original paper with an improved design, developing a path based neural predictor [1]. The new approach overcome the high latency of perceptron predictor by parallelizing computation and performing predictions in a staggered fashion, thus distributing the work and reducing the amount of computation required when a prediction is requested. The latency is mitigated by making y_{out} computations begin in advance. Additionally, this predictor incorporates the path taken in its calculations, which Jiménez found to increase accuracy over his original design.

One technique that has been employed to help reduce the latency of perceptron based predictors is to implement them as a part of an overriding predictor. An overriding predictor in which a fast but inaccurate predictor makes an initial prediction, but a slower and more accurate predictor may then retroactively recognize an error and correct the prediction, thus providing predictions just as accurate as those of the normal perceptron predictor while sometimes avoiding the latency penalty. This setup was employed in [3] and in [1].

More recent branch predictor designs, namely those based on TAGE, seem to outperform neural predictors [27]. Nevertheless, there have been continued efforts to improve accuracy of branch predictors through the use of neural

based techniques. One example of such as system may be found in [18], where Jiménez worked to incorporate a neural predictor into TAGE. Other efforts have included adding additional features to neural predictors, further optimizing their design [19], or even investigating implementing neural based methods on analog hardware [14].

While a great deal of information is available about different predictors that have been proposed by academia, it is difficult to know what type of predictors are being implemented in actual processors to the degree to which chip manufacturers guard the details of their designs. However, there is still some information available. Namely, AMD and Samsung have both publicly disclosed that they are making use of some sort of neural prediction in their chips [24] [25], though no information is available about how they have implemented these systems.

Overall, while neural network based predictors sound highly promising, issues with latency have hampered their use in some scenarios, and they are often outperformed in simulated systems by other predictors. As such, neural network based predictors are likely not the be-all and end-all for branch predictor designs. Nevertheless, these hurdles and drawbacks are reasonable enough that they are still seeing use in a variety of real world situations.

III. PROPOSAL

We propose to implement both simple perceptron and fast-path based neural predictors in order to to understand the workings and performance of neural networks for making branch predictions. We aim to understand and compare these predictors with current standards such as gshare and bimodal. We plan to simulate the branch systems using gem5. The idea is to try out software and hardware configurations to analyze the performance for different variety of systems. While the original papers detailing perceptron predictors suffered from high latency, they also claimed that they had the highest accuracy. However, more recent studies have shown that it is possible to significantly reduce the latency of a neural predictor, thus making their use more attractive. At the same time however, modern branch predictors such as TAGE are known to perform very well, and it would be beneficial to compare the performance of neural predictors to that of the state of the art.

We have limited our implementations to a single perceptron layer. Multilayer neural networks have been shown to have higher latency than expected of an industrial branch predictor and they are quite complicated to implement in hardware [4]. One other difference between our predictors and those that we are basing our work off of is that the perceptron predictors in literature made use of parallel execution in hardware in order to minimize the latency of the predictor. In our situation, there is no need for such parallelism as in gem5, the complexity of a branch predictor does not effect the simulated runtime of a program. As such, we implement our branch predictors with serial execution instead.

function *prediction* (pc: **integer**): { *taken* , *not_taken* };
begin
$i := pc \bmod n$
$y_{out} := W[i,0] + \sum_{j=1}^{h} \begin{cases} W[i,j] & \text{if } G[j] = taken \\ -W[i,j] & \text{if } G[j] = not\_taken \end{cases}$
if $y_{out} \geq 0$ **then**
prediction := *taken*
else
prediction := *not_taken*
end if
end

procedure *train* (i, y_{out}: **integer**; *prediction* , *outcome* : { *taken* , *not_taken* });
if *prediction* $\neq$ *outcome* **or** $|y_{out}| \leq \theta$ **then**
$W[i,0] := W[i,0] + \begin{cases} 1 & \text{if } outcome = taken \\ -1 & \text{if } outcome = not\_taken \end{cases}$
for j **in** $1..h$ **in parallel do**
$W[i,j] := W[i,j] + \begin{cases} 1 & \text{if } outcome = G[j] \\ -1 & \text{if } outcome \neq G[j] \end{cases}$
end for
end if
$G := (G << 1)$ **or** *outcome*
end

Fig. 2. Perceptron training and update [3]

A. PERCEPTRON PREDICTOR

We based the perceptron predictor off of the work of Daniel Jiménez et. al. in [3] Our perceptron predictor is similar to other predictors in the sense that it keeps a global history shift register. The perceptron keeps a n x (h-1) matrix W[0...n-1,h] of integer weights. When a branch outcome is known, the train algorithm updates the predictor. There is a trade off between long term accuracy and ability to adapt to phase behavior. The sign of y_{out} determines taken and not taken decisions. If our prediction is incorrect, we update the the corresponding weights in W. We decrease W[i] by one if outcome is not taken and increase by one when prediction is taken. After updating the weights, we update the global history shift register with the outcome of the branch. The prediction and update algorithms are shared in 2.

B. FAST-PATH PREDICTOR

We based the fast-path predictor off of the work of Daniel Jiménez in [1]. The fast path predictor is very similar to the perceptron predictor in that it keeps a weight matrix W of weight vectors. However only the 0^{th} weight i.e the bias weight is used to predict the current branch. y_{out} is the partial sum from the last prediction including the bias weight. We keep two more vectors, SR[0..h] and R[0..h]. The first column of W form the bias weights. SR contains the partial computed sums. However ,unlike the perceptron algorithm, each row of the Weight vector corresponds to h branches rather than just one. The prediction algorithm is shown in 3.

IV. METHODOLOGY

In this section we describe how we evaluated our work, including the frameworks we used, the metrics we selected, and the applications we evaluated these metrics on.

A. FRAMEWORK

To implement and evaluate our branch predictor we used the gem5 simulator to simulate an X86 architecture for

```
function prediction (pc: integer) : { taken , not_taken }
begin
    i := pc mod n
    y := SR[h] + W[i, 0]
    if y ≥ 0 then
        prediction := taken
    else
        prediction := not_taken
    end if
    for j in 1..h in parallel do
        k_j = h − j
        if prediction = taken then
            SR'[k_j + 1] := SR[k_j] + W[i, j]
        else
            SR'[k_j + 1] := SR[k_j] − W[i, j]
        end if
    end for
    SR := SR'
    SR[0] := 0
    SG := (SG << 1) or prediction
end
```

Fig. 3. Path based neural prediction algorithm to predict branch at address [1]

running our branch predictors [7]. We evaluate the branch predictors on Gem5 as it is the most common architecture for desktop and server usage. Gem5 is a free and widely used simulator that supports multiple architectures like X86, ARM, and SPARC and it provides a useful interface for testing system components like branch predictors without getting mired in the complexity of the system. In other words, it is relatively simple to create and test custom predictors, making the cycle of tinkering and testing more efficient.

We evaluated our branch predictors using gem5's DerivO3CPU, a pipelined CPU with support for out of order execution. This processor was set to run at 1 GHz, with 64 kB L1 cache, 256 kB L2 cache, and 512 MB memory. Memory accesses were made using gem5's timing memory model, which works to accurately reflect the performance impact of memory accesses. All other processor parameters are the defaults provided in gem5's DerivO3CPU class.

B. PREDICTORS

For our evaluation, evaluated a set of seven branch predictors. Four of these simulators are built-in to the gem5 simulator, one is a static branch predictor of our design, and the other two are perceptron based branch predictors that we implemented based on the work of [4] and [1].

Gem5 includes a number built-in branch predictors including LTAGE and a tournament branch predictor, which we used to compare the performance of these designs to that of the perceptron-based predictors. In addition, we added a static branch predictor that always predicts a branch to be taken in order to provide a solid baseline against which to evaluate the dynamic branch predictors. The dynamic branch predictors include a local branch predictor, a tournament branch predictor, a bimodal branch predictor, and LTAGE.

For our evaluation, we used the default parameters for each of the branch predictors that we employed. This means that in most cases, the branchpredictors which use a global history buffer default to the default buffer length in gem5.

1) TournamentBP: The tournament branch predictor makes predictions based on the last n branches. It is similar to the local branch predictor in that it keeps track of which of the past n branches are most correlated with outcomes. However, it is capable of making predictors that depend on results further back. The TournamentBP in gem5 is based on that used in the Alpha 21264 microprocessor [26], and it includes both a local predictor and a global predictor. Each of these predictors uses a history buffer to index into a table of counters. A mechanism then chooses between the results returned by the two different systems.

2) BiModeBP: The bimodal branch predictor maintains both a global and local history of conditional branches and keeps track of which history has been the most useful when making predictions for the current branch. Using a bimodal predictor can increase the flexibility of a branch predictor by allowing it to detect whether a global or local pattern is present [13] [21]. Bimodal branch predictors do very well and have become standard models of branch predictors.

3) LTAGE: LTAGE was born of out of online branch predictor competitions where the goal is to accurately predict conditional outcomes through CPU dumps, being penalized for latency [10]. LTAGE maintains a variable length local history for different branches to see which one correlates most with the branches that are taken. This essentially allows it to avoid having to preset a size for history and be stuck with it which may or may not be beneficial. The TAGE predictor is often considered one of the most accurate predictors available, though a hardware implementation is not always practical [12].

4) LocalBP: This branch predictor is one of the standard predictors included with gem5. It works by using the program counter and local history to index into a table of two bit saturating counters, which are then used to make predictions. By using the program counter to index into the table, this predictor aims to ensure that the value of each counter is set only by a specific program counter, thus meaning that the counters track local history.

5) AlwaysBP: This branch predictor is a static branch predictor of our own design. The predictor will always predict that a branch is taken, serving as a baseline for a low performing branch predictor.

6) Perceptron: This predictor is our implementation of the basic perceptron predictor as described by Jiménez et. al in [1]. Unlike the predictor that Jiménez uses in testing and that is proposed in [4], our predictor does not make use of local history to any extent. Our Perceptron predictor keeps a global history buffer of length $globalPredictorSize$, which is a default value set by gem5 and that is shared across predictors.

7) FastPath: This predictor is our implementation of the path based perceptron predictor as described by Jiménez in [1]. Our FastPath predictor keeps a global history buffer of length $globalPredictorSize$, which is a default value set by

gem5.

C. METRICS

We will be using three main metrics to evaluate the performance of our branch predictors. The first of these metrics, mispredictions per one thousand instructions, abbreviated MPKI, is a measure that is commonly used to measure how frequently a branch predictor will make a misprediction in a given workload. We obtain this value by dividing the total number of mispredictions by the number of instructions executed, after which we multiply by one thousand to get the correct units. The benefit of this measure is that it is a measure solely of how frequently mispredictions are made and does not incorporate latency or any other factors. This makes it useful for comparing the predictive power of branch predictors, even across different CPU architectures. However, the singular focus of this metric is also a drawback. As this metric does not take into account latency, it does not fully show the effects of all the design decisions present in the predictor.

The next metric that we are investigating is overall prediction accuracy. This metric is similar to MPKI in that it is not affected by changes in CPU architecture, and instead focuses solely on performance. While we expect results to be similar between MPKI and predictive accuracy, it may be possible that the prediction patterns of a branch predictor could somehow cause these metrics to diverge. In the tables, this metric represents the ratio of correct predictions made to the overall number of predictions made.

Lastly, we will be evaluating the branch predictors using their simulated execution time on a number of benchmark applications. This metric provides the best insight as to how the differences in misprediction rate can affect real world performance while running an application. As such, it is the most helpful for seeing how much of a speedup a new branch predictor may provide on a particular workload. However, this metric is also highly dependent on the hardware being simulated, and as such, branch predictors must be reevaluated on each potential platform on which they could be used. Additionally, this metric is highly affected by any assumptions made regarding latency of predictors. In fact, in gem5, we have been unable to find a way to impose a latency on the branch predictors, and as such this metric will not accurately reflect how a hardware implementation of these branch predictors would work in the real world. In our tables, we represent this metric as latency, representing the time required to complete the program.

We do not use instructions per cycle (IPC) as a metric because each of our predictors is assumed to have the same latency, and as such, the only differences in IPC would be due to pipeline stalls caused due to incorrect predictions. The effect of these stalls is also effectively captured by the measurement of simulated time to completion, and as such we omit this metric.

D. BENCHMARKS

We are using the LLVM test suite provided by Stanford [5].The LLVM Project is a collection of modular and reusable compiler and toolchain technologies[6]. The suite includes a number of test programs, which we make use of for benchmarking purposes. Though these programs are not necessarily made for benchmarking, many of these programs are lightweight, thus allowing the simulations to be run in a reasonable amount of time. These programs include Bubblesort, Oscar, Quicksort, Towers, Treesort, Perm, IntMM, RealMM, Puzzle, and Queens. The programs in the test suite vary in the the intensity with which they make use of branching. Some of the programs are heavily reliant on branching. The program Perm contains a lot of permutations while IntMM and RealMM deal with matrix multiplications. Both IntMM and RealMM involve loops which makes them suitable candidates for branch predictions. Bubblesort, while not typically used in programs due to its poor performance, offers a great number of comparisons and loops. Quicksort and Treesort all rely on loops and conditionals to interchange elements and compare values while sorting arrays. Towers solves the towers of Hanoi puzzle. Queens solves the eight queens problem repeatedly, using the same path to derive a solution each time.

Some of the benchmark programs, such as Bubblesort, execute their task only once before terminating. As a result, branch predictors are unable to see a complete execution of the problem and apply what they learn to future repetitions. However, some other programs, such as Queens and Tower, execute their task given task repeatedly and from the same starting position. As such, in these situations a branch predictor has many more opportunities to learn the pattern due to the consistent branching patterns, potentially allowing for perfect accuracy on later repetitions of the task.

These programs seem to not have been originally designed as benchmark programs, and as such each program is generally quick to run, with little variation in the workload over the course of the program's execution. One consequence of the short length of these programs is that some branch predictors may not have sufficient time to learn the patterns of the different programs, thus making their accuracy lower than may otherwise be expected. However, this shortcoming also provides an interesting new dimension to the evaluation of the branch predictors as the highest performing branch predictors will be those that not only are capable of learning patterns well, but also capable of learning them quickly.

The programs are compiled for the X86 architecture using GCC with the $-o3$ optimization flag set and the executables are then run on gem5 using each of the different branch predictors. Each program is run once, with the results of that single run being used for the performance evaluation. We are able to use the results from a single simulation as the simulations are deterministic, and thus the results will remain consistent across experiments.

V. EVALUATION

Before investigating performance using the metrics that we derived, it should first be noted that the metrics do not track one another in the way that we had expected. Before performing our evaluation, we believed that as branch mispredictions were the only factor that would differ among the different branch predictors, that execution time, accuracy, and MPKI would all scale with one another. However, even after checking our methods for calculating these metrics multiple times, we found that they still did not track one another. One example of this can be seen in the BubbleSort benchmark, where AlwaysBP records an accuracy of .91, despite having a higher MPKI than the perceptron predictor, which has an accuracy of .90. Additionally, despite having a higher accuracy than the Perceptron predictor, the AlwaysBP is 1.28 times slower to execute. We are not sure exactly what the source of this inconsistency is, but we believe that it could either be an unexpected side effect of the out of order execution of the CPU model that we used or a result of the number of branch instructions not scaling directly with the number of total instruction as a side effect of some aspect of the branch prediction patterns. As such, while latency remains the best metric for measuring the overall performance impact of each branch predictor in each benchmark, MPKI and accuracy must be regarded separately. Throughout the results it seems that MPKI is the metric that correlates most strongly to execution time, and as such it is likely a more reliable metric than accuracy.

Overall, we found that as measured by execution time, the LTAGE predictor built into gem5 outperforms all of the other branch predictors in every benchmark. In some cases, LTAGE outperforms the other predictors by a wide margin, with its MPKI being up to four times lower than the second best predictor. This result is not especially surprising as the LTAGE predictor won the championship branch prediction competition in the year that it was introduced [10], and the predictors that we compared it against here are all older designs.

As the LTAGE predictor easily outperforms the other predictors, we may instead turn our attention to analyzing how the older predictors fare against one another, as these are the predictors that were in literature at the time that perceptron based predictors were originally introduced.

Due to our inability to impose any sort of latency penalty on the slower perceptron based predictors in gem5, the execution time of each of the benchmarks is a function solely of the number of mispredictions. As such, the results that we gain here will not necessarily align with those of past studies. However, our results are unique as each branch predictor is implemented in a ”pure” fashion, without the additional overhead imposed through the use of overriding predictors.

When comparing the FastPath and Perceptron predictors, we see that there is not a consistent winner in every benchmark. However, in most situations we see that the FastPath predictor either performs on par with or better than the perceptron predictor, as measured by MPKI. However, the perceptron predictor still outperforms the FastPath predictor in the BubbleSort benchmark. This is in line with Jiménez's results in [1], indicating that the addition of the path taken to a particular branch is useful in predicting the outcome of that branch.

TABLE I
BUBBLESORT PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	14.7	24.77	0.91
BiModeBP	9.6	12.28	0.93
LocalBP	10.7	15.11	0.92
LTAGE	6.5	2.87	0.97
Perceptron	11.4	21.61	0.90
FastPath	14.5	28.89	0.90
TournamentBP	7.3	5.57	0.96

While the perceptron predictors were billed at their time of introduction as outperforming all other branch predictors, we find that they are often outperformed by other branch predictors in our benchmarks. One reason for this may be that our implementations of the perceptron predictors only made use of global history, while some of the ones that they are competing against, such as TournamentBP and LocalBP, are making use local history in order to support their decisions. This is in line with the general result that local history increases prediction accuracy, although it comes at the cost of more difficult hardware implementations. In fact, in the presentations given by Daniel Jiménez detailing his entries into the most recent branch prediction competition, he explicitly states that he does not advocate for using local history in a real world implementation and that he only included local history in his designs in order to gain an edge in the competition[27].

The performance of the Perceptron predictor, in the average case, was far worse than we expected, especially in terms of MPKI. This stems from the poor performance of the perceptron predictor in a number of the benchmarks, including Perm and Treesort. Overall however, the perceptron predictor is often outperformed by all other predictors in the majority of cases studied. We are not sure what the cause of this is. It is possible that the perceptron predictor did not have enough time to converge on these benchmarks.

One surprising result was that AlwaysBP actually attained decent overall accuracy, with its lowest prediction accuracy being 90%. In fact, as measured by accuracy, AlwaysBP outperformed other branch predictors on multiple occasions, including in Quicksort and Bubblesort. This is likely a result of these programs containing branching patterns where the branches are overwhelmingly taken, perhaps a result of the heavy reliance of some of the programs on for loops rather than other more complex branching patterns. At the same time, it is interesting that the more complex branch predictors were outperformed in these situations. Perhaps it is possible that the more complex predictors were thrown off from the trend of branches tending to be taken by the occasional branch which was not taken, or perhaps the more complex

branch predictors, especially the perceptron predictor, did not have enough time to learn the pattern.

TABLE II
QUICKSORT PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	11.5	35.55	0.89
BiModeBP	10.3	41.16	0.87
LocalBP	9.6	28.20	0.90
LTAGE	9.3	28.07	0.90
Perceptron	11.1	48.77	0.87
FastPath	11.0	44.16	0.86
TournamentBP	10.6	43.68	0.87

TABLE III
TREESORT PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	16.6	36.23	0.90
BiModeBP	16.1	33.96	0.91
LocalBP	16.2	27.76	0.92
LTAGE	15.2	25.01	0.93
Perceptron	24.1	97.63	0.85
FastPath	18.1	44.09	0.90
TournamentBP	18.7	45.39	0.90

TABLE IV
PERM PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	7.4	15.02	0.92
BiModeBP	6.4	5.42	0.96
LocalBP	7.9	19.88	0.90
LTAGE	6.2	0.96	0.99
Perceptron	10.1	42.67	0.83
FastPath	7.4	13.10	0.92
TournamentBP	6.3	2.07	0.99

TABLE V
OSCAR PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	3.9	3.07	0.91
BiModeBP	3.7	2.42	0.93
LocalBP	3.7	2.29	0.93
LTAGE	3.6	1.59	0.95
Perceptron	3.8	3.33	0.91
FastPath	3.8	2.81	0.92
TournamentBP	3.7	1.94	0.94

TABLE VI
QUEENS PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	8.3	38.39	0.90
BiModeBP	7.0	32.15	0.89
LocalBP	7.7	33.22	0.90
LTAGE	3.1	2.10	0.99
Perceptron	8.1	41.14	0.88
FastPath	7.7	32.6536	0.91
TournamentBP	6.7	31.29	0.89

TABLE VII
INTMM PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	4.6	2.83	0.94
BiModeBP	4.6	3.05	0.94
LocalBP	4.6	2.16	0.95
LTAGE	4.4	1.53	0.97
Perceptron	4.6	3.32	0.94
FastPath	4.6	2.71	0.95
TournamentBP	4.6	2.48	0.95

TABLE VIII
PUZZLE PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	41.6	10.35	0.95
BiModeBP	41.2	15.58	0.92
LocalBP	41.7	11.76	0.94
LTAGE	30.9	2.37	0.98
Perceptron	49.9	29.30	0.89
FastPath	43.89	17.20	0.92
TournamentBP	41.3	15.06	0.93

TABLE IX
TOWERS PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	8.5	11.45	0.95
BiModeBP	6.3	6.33	0.96
LocalBP	6.0	4.61	0.97
LTAGE	5.9	0.89	0.99
Perceptron	6.8	11.46	0.94
FastPath	8.4	11.42	0.95
TournamentBP	6.2	3.44	0.98

TABLE X
ARITHMETIC MEAN PERFORMANCE

Predictor	Latency (ms)	MPKI	Accuracy
AlwaysBP	13.03	19.74	.92
BiModeBP	11.69	16.93	.92
LocalBP	12.01	17.32	.93
LTAGE	8.90	7.82	.96
Perceptron	14.95	34.70	.89
FastPath	13.60	21.02	.92
TournamentBP	14.40	18.17	.93

VI. CONCLUSIONS

The focus of our work was to implement branch predictors in a way such that the latency incurred by predicting on real hardware was not a factor, thus allowing for each branch predictor to be implemented without consideration for latency and maximizing each algorithm's potential to its fullest. Our work also compared state of the art modern predictors against predictors that were the state of the art in the past, thus enabling us to see just how far performance has come.

Overall, we found that the TAGE predictor consistently provide the highest prediction accuracy as well as the shortest execution times, thus showing that it is the most capable of learning the patterns present in the workloads provided. However, this does not necessarily mean that TAGE is the universally best branch predictor that should always be used. Namely, there are a number of weaknesses caused by the use of the simulator, as well as the nature of the benchmarks. First, gem5 makes it difficult to assign latency to a branch predictor, and thus there is no penalty incurred when predictors perform complex operations that would not be feasible if implemented in hardware, where they would slow the system down to an unacceptable degree. Another issue with simulating in this system is that it is difficult to define a metric by which it should be determined that branch predictors are on a level playing field in terms of the amount of state which they can hold. Gem5 also makes it difficult to implement the sort of overriding predictor which was used heavily in [1], and as such it is not possible to compare our results with those obtained by Jiménez Additionally, the nature of the benchmarks used gives an advantage to branch predictors that are capable of learning a pattern rapidly, and then remembering that pattern across a largely static workload. While the theory behind perceptron based predictors sounds very promising, the performance of these classical neural predictors cannot compete with the current state of the art.

In the future, it would be interesting to use gem5 as a platform to rapidly prototype and experiment with different branch predictor designs. While it by no means provides a perfect indication of suitability for real world use, it can provide insight as to accurately a new design may be able to make predictions. Some additional aspects to explore would be the addition of new features to the perceptron based predictors such as innermost loop counters [15] and how impactful those new features are. It would also be interesting to explore the use of local history in perceptron predictors, as local history had a more significant effect than we had anticipated in the overall performance of the different predictors. Another interesting factor to explore would be the impact of changing the history lengths in the perceptron based predictors, allowing us to see if a point of diminishing returns exists.

Lastly, it would be good to investigate why our accuracy metric provided results inconsistent with our other two metrics. While we suspect that it may be due to different branch predictors causing different distributions of branches to be encountered, we need to investigate this more thoroughly before any firm conclusions can be drawn using the accuracy metric.

VII. STATEMENT OF WORK

All members of the group contributed to creating the presentation.

A. Cyrus

Cyrus worked on implementing the simple perceptron and path-based perceptron with Akshay as well as using Tyler's test infrastructure to extract the relevant results using the LLVM test suite.

B. Akshay

Akshay worked on simple perceptron and path-based percepton along and their testing and benchmarking on LLVM test suite along with Cyrus and Tyler. He contributed to the writing of related works, proposal and results for report.

C. Tyler

Tyler helped to build testing infrastructure for the project, as well as a tool to extract information from log files. Additionally, he built the static branch predictor that was used in evaluations. He contributed to writing the conclusion, methodology, evaluation, related work, and introduction.

REFERENCES

[1] Daniel A. Jimenez. Fast Path-Based Neural Branch Prediction, at the 36th International Symposium on Microarchitecture (MICRO-36), San Diego, California, December 2003.

[2] Faustino Gomez, Doug Burger, and Risto Miikkulainen. A neuroevolution method for dynamic resource allocation on a chip multiprocessor. In Proceedings of the International Joint Conference on Neural Networks (IJCNN-01), July 2001.

[3] Daniel A. Jimnez and Calvin Lin. 2001. Dynamic Branch Prediction with Perceptrons. In Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA '01). IEEE Computer Society, Washington, DC, USA, 197-.

[4] Daniel A. Jimenez, Calvin Lin. Neural Methods for Dynamic Prediction.

[5] LLVM Single source benchmarks from Stanford. https://github.com/llvm-mirror/test-suite/tree/master/SingleSource/Benchmarks/Stanford

[6] LLVM Compiler Infrastructure. https://llvm.org/ProjectsWithLLVM/

[7] The gem5 Simulator. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. May 2011, ACM SIGARCH Computer Architecture News.

[8] Lucian N. Vintan and M. Iridon. Towards a high performance neural branch predictor. In Proceedings of the International Joint Conference on Neural Networks, volume 2, pages 868 873, July 1999.

[9] Brad Calder, Dirk Grunwald, Donald Lindsay, James Mar- tin, Michael Mozer, and Benjamin Zorn. Corpus-based static branch prediction. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pages 7992, June 1995.

[10] A.Seznec. The L-TAGE predictor. Journal of Instruction Level Parallelism, 2007.

[11] Andre Seznec and Pierre Michaud. A case for (partially)- tagged geometric history length predictors. Journal of Instruction Level Parallelism (http://www.jilp.org/vol7), April 2006.

[12] Andre Seznec. 2011. A new case for the TAGE branch predictor. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44). ACM, New York, NY, USA, 117-127. DOI: http://dx.doi.org/10.1145/2155620.

[13] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. 1997. The bi-mode branch predictor. In Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30). IEEE Computer Society, Washington, DC, USA, 4-13.

[14] Renee St. Amant, Daniel A. Jimenez, and Doug Burger. 2008. Low-power, high-performance analog neural branch prediction. In Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41). IEEE Computer Society, Washington, DC, USA, 447-458. DOI: https://doi.org/10.1109/MICRO.2008.4771812

[15] André Seznec, Joshua San Miguel, and Jorge Albericio. 2015. The inner most loop iteration counter. Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48 (2015). DOI:http://dx.doi.org/10.1145/2830772.2830831

[16] André Seznec. TAGE-SC-L Branch Predictors Again. 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5), Jun 2016, Seoul, South Korea. 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5) 2016, http://www.jilp.org/cbp2016/. hal-01354253

[17] André Seznec. TAGE-SC-L Branch Predictors. JILP - Championship Branch Prediction, Jun 2014, Minneapolis, United States. Proceedings of the 4th Championship Branch Prediction, http://www.jilp.org/cbp2014/. hal-01086920

[18] Jimnez, D. "Multiperspective perceptron predictor with TAGE." Championship Branch Prediction (CBP-5) (2016).

[19] Daniel A. Jiménez. An optimized scaled neural branch predictor. In In Proceedings of the 29th IEEE International Conference on Computer Design (ICCD-2011), October 2011.

[20] Jorge Albericio, Joshua San Miguel, Natalie Enright Jerger, and Andreas Moshovos. 2014. Wormhole: Wisely Predicting Multidimensional Branches. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47). IEEE Computer Society, Washington, DC, USA, 509-520. DOI: http://dx.doi.org/10.1109/MICRO.2014.40

[21] S. McFarling, Combining Branch Predictors, June 1993.

[22] https://commons.wikimedia.org/wiki/File:Branch_prediction_2bit_saturating_counter-dia.svg

[23] Tse-Yu Yeh and Yale N. Patt. 1991. Two-level adaptive training branch prediction. In Proceedings of the 24th annual international symposium on Microarchitecture (MICRO 24). ACM, New York, NY, USA, 51-61. DOI=http://dx.doi.org/10.1145/123465.123475

[24] https://www.amd.com/en/technologies/sense-mi

[25] https://www.androidauthority.com/closer-look-samsung-mongoose-cpu-712587/

[26] R. E. Kessler, E. J. McLellan and D. A. Webb, "The Alpha 21264 microprocessor architecture," Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273), Austin, TX, 1998, pp. 90-95. doi: 10.1109/ICCD.1998.727028

[27] https://www.jilp.org/cbp2016/program.html