CSCE 689 Project Report

Tianfang Guo
Department of Electrical and Computer Engineering
Texas A&M University
College Station, United States
tiaguo@tamu.edu

Sahil Kaushal
Department of Computer Science and Engineering
Texas A&M University
College Station, United States
sahil_kaushal@tamu.edu

***Abstract*—In this project, we implement and evaluate a Weightless Neural Network based branch predictor in ChampSim. We compare the accuracy and storage efficiency of our predictor against the Hashed Perceptron predictor (already present in ChampSim) across a standardized sweep of hardware budgets ranging from 24KB to 1536KB. Source code for the project can be found at https://github.com/Sahilk5/689_BP_Champsim.**

I. Introduction

Branch prediction is a fundamental part of modern superscalar processors. It is essential to mitigate control hazards and maintain high instruction throughput. By speculatively executing instructions before branch outcomes are known, processors can avoid costly pipeline bubbles and stalls.

Although simple counters and correlation-based predictors were sufficient for early architectures, modern workloads require increasingly sophisticated conditional branch predictors. Today, state-of-the-art examples include the TAGE (Tagged Geometric) predictor [1] and perceptron-based neural predictors [2], [5], [7]. These designs have proven highly effective at extracting information and correlation from various microarchitectural features and utilizing this information to make accurate branch predictions.

In this project, we explore another promising alternative known as "weightless" neural networks (WNNs); specifically, the Wilkes, Stonham and Aleksander Recognition Device (WiSARD) [3]. Unlike standard neural predictors that rely on arithmetic operations (dot products) and weight updates, WNNs operate using lookup tables and memory addressing. For this project, we will implement a WiSARD based branch predictor in ChampSim, as recently proposed by Villon et al. [4]. We compare its performance against the Hashed Perceptron predictor [5] to determine if the WNN paradigm offers a viable alternative to traditional arithmetic-heavy neural predictors.

II. Background and Related Work

A. ChampSim simulation environment

ChampSim is a trace-based simulator that is widely used in the computer architecture community for academic research. For our project, it provides a robust environment for evaluating the impact of predictor accuracy on system-level metrics like Instructions Per Cycle (IPC) and Misses Per Kilo-Instructions (MPKI).

The original WNN study [4] utilized a python based implementation. We will implement our predictor in ChampSim (C++). This will help us gather and analyze more realistic results with respect to latency and pipeline integration.

B. Perceptron Neural Networks

The perceptron-based branch predictor was first introduced by Jiménez and Lin [2] as an alternative to two-level adaptive predictors. A perceptron is a linear classifier that maintains a vector of signed weights $w_0, w_1, \ldots, w_n$, where each weight is associated with one bit of the global history register. To make a prediction, the predictor computes a weighted sum

$$y = w_0 + \sum_{i=1}^{n} w_i \cdot x_i \tag{1}$$

where $x_i \in \{-1, +1\}$ encodes the i-th history bit (taken or not taken) and w_0 is a bias term. The branch is predicted taken if $y \geq 0$ and not taken otherwise. Training is performed only when the prediction is incorrect or when $|y|$ falls below a confidence threshold θ; in either case, every weight is incremented or decremented by one in the direction of the actual outcome.

The principal advantage of perceptrons over fixed-history predictors is that they can scale to very long history lengths with linear (rather than exponential) storage cost. However, the original formulation suffers from two well-known limitations: (i) it can only learn linearly separable functions of its inputs, and (ii) the latency of the dot-product computation grows with history length. Subsequent work has addressed these limitations through hashing [5] and through the use of multiple feature perspectives on branch history, as in the Multiperspective Perceptron predictor [7].

The Hashed Perceptron [5] is the variant most relevant to this work and is the baseline we compare against. It replaces the one-to-one correspondence between weights and history bits with a set of independently indexed weight tables, where each table is addressed by a hash of a different segment of branch history (typically a folded shift register XORed with the program counter). Geometric history lengths are used so that each table captures correlations at a different time scale. The prediction is computed as the sum of the selected weights from all tables, and training updates only those selected entries. This reduces storage and latency relative to the original formulation while retaining the perceptron learning rule. The

use of hashing also gives Hashed Perceptron access to non-linear features over its raw inputs, since the hash functions themselves are free to mix bits non-linearly; the per-table summation is then linear over those constructed features. We discuss the implications of this design choice and contrast it with WiSARD's approach in Section II-D.

C. Weightless Neural Networks

Weightless Neural Networks (WNNs) are a family of neural models that, unlike traditional artificial neural networks, do not store weights along their synaptic connections. Instead, the synaptic connectivity is fixed at design time, and learning is realized by updating the contents of Random Access Memory (RAM) units that act as the network's neurons [3]. Because computation reduces to memory addressing and counter updates rather than arithmetic on weights, WNNs are particularly attractive for hardware implementation.

The WiSARD model [3] is the most well-known weightless neural network. A WiSARD classifier consists of one *discriminator* per output class, where each discriminator is a bank of N RAM nodes. Every RAM node has n address lines, giving it 2^n entries. Inputs to the network are binary vectors; for each RAM node, a fixed pseudo-random mapping selects n bits from the input vector to form an address. During training, the counter at the addressed entry of every RAM node in the discriminator corresponding to the input's class is incremented. During inference, all RAM nodes in every discriminator are read at the addresses computed from the new input, and the counters that exceed a threshold, return one, which are summed into a per-discriminator response. The class associated with the discriminator producing the highest response is returned as the prediction.

Two practical mechanisms make this scheme robust. First, *bleaching* [6] resolves ties or low-confidence predictions by raising the threshold above which a counter contributes to the response, suppressing weakly-learned patterns until a clear winner emerges. Second, periodic *counter decay* prevents counters from saturating and allows the predictor to adapt to changes in workload behavior. With these additions, each RAM node functions as an empirical pattern memorizer over its n-bit input slice, and the discriminator votes on the class whose training set most closely matches the current input.

Recently, Villon et al. [4] proposed using the WiSARD model as a conditional branch predictor, demonstrating that it can achieve accuracy comparable to state-of-the-art neural predictors. Our work builds directly on this proposal, providing the first ChampSim-based implementation and a systematic comparison against the Hashed Perceptron predictor across a range of hardware budgets.

D. Hashed Perceptron vs. Weightless Neural Networks

Although both Hashed Perceptron and WiSARD ultimately make a prediction by summing values fetched from a set of indexed tables, the two models differ fundamentally in what is stored and how it is interpreted.

What is stored. A Hashed Perceptron table entry is a single *signed* counter representing aggregate evidence: "this hash bucket leans $+w$ toward taken." The bit pattern that produced the hash is discarded by the hash function; only the running tally remains. WiSARD, by contrast, stores an *unsigned* counter at every n-bit address of each RAM node. Because the address is the raw bit pattern itself, the counter directly records how often that exact pattern was observed during training of one specific class. A WiSARD RAM thus stores a frequency table over bit patterns rather than a single scalar of net sentiment.

Model class. Both Hashed Perceptron and WiSARD compute a thresholded sum of values fetched from a set of indexed tables, but the tables are populated and indexed differently. Hashed Perceptron stores one signed weight per hash bucket, where each hash is a designed combination of folded history and PC. The non-linearity in this model comes from the hash functions themselves: any branch-prediction signal must be linearly separable *after* hashing, but the hash is free to be highly non-linear over raw bits. WiSARD, by contrast, indexes each RAM with raw bits and stores an entire 2^n-entry truth table per RAM, so a single RAM can directly represent any Boolean function over its n input bits, including XOR. The two predictors thus realize non-linearity in different places: Hashed Perceptron embeds it in the hash construction, while WiSARD embeds it in the lookup table itself. Neither is universally more expressive — Hashed Perceptron benefits from feature engineering of its hash functions, while WiSARD benefits from full pattern memorization within each RAM but is limited by which bits the static mapping happens to group together.

Arithmetic. Hashed Perceptron requires signed arithmetic: signed counters, signed adder trees, and signed comparators against a threshold. WiSARD requires only unsigned counter increments during training and unsigned comparisons during inference. Negative evidence is encoded structurally—through the not-taken discriminator—rather than numerically. This eliminates the need for signed arithmetic units in the predictor's datapath, which is one of the historical motivations for weightless models in hardware [3].

Trade-offs. The expressive advantage of WiSARD relies on its random input mapping happening to group informative bits together within the same RAM. Since this mapping is fixed at design time, the predictor cannot adapt its feature selection to the workload. Hashed Perceptron, in contrast, sidesteps this lottery by relying on aggressively folded history that mixes bits across long temporal windows, but at the cost of model expressiveness. The two predictors thus represent different bets about where the predictive signal lives: in the joint structure of raw input bits (WiSARD) versus in temporally-aggregated linear summaries (Hashed Perceptron).

III. Predictor Architecture

A. Input composition

Following Villon et al. [4], the input to the WiSARD predictor is a 2158-bit binary vector constructed by concatenating

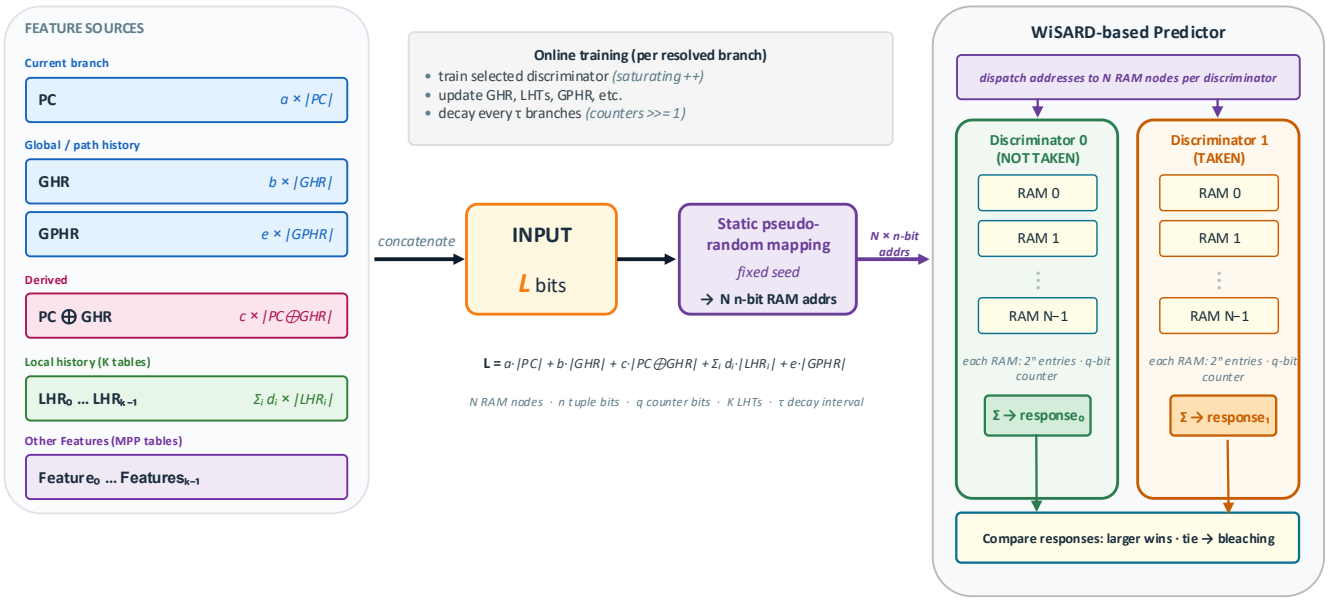

Fig. 1. WiSARD predictor architecture

multiple replicated copies of the following microarchitectural features:

- the lower 24 bits of the program counter (*PC*), replicated 24 times;
- the 24-bit speculative global history register (*GHR*), replicated 12 times;
- the bitwise XOR of *PC* and *GHR*, replicated 12 times;
- five local history registers (LHR_0 through LHR_4), of widths 24, 16, 9, 7, and 5 bits respectively, replicated 8, 8, 8, 6, and 12 times;
- the 64-bit global path history register (*GPHR*), replicated 8 times.

The replication factors act as importance weights: a feature copied more times has proportionally more chances of contributing bits to the address of any given RAM node. Each local history register is indexed from a 1024-entry local history table by the lower bits of the program counter. The GPHR records the lower 8 bits of the last 8 branch addresses and is updated by left-shifting and XOR-ing the new PC into the register.

The total replicated length sums to $24 \cdot 24 + 12 \cdot 24 + 12 \cdot 24 + (8 \cdot 24 + 8 \cdot 16 + 8 \cdot 9 + 6 \cdot 7 + 12 \cdot 5) + 8 \cdot 64 = 2158$ bits, matching the configuration found by Villon et al. [4] to provide the best accuracy on the CBP-3 dataset.

B. Architecture

Our WiSARD predictor consists of two discriminators – one for taken branches (D_T) and one for not-taken branches (D_{NT}) – each containing N RAM nodes with n-bit address widths and q-bit saturating counters. Together, the two discriminators implement the binary classifier whose output is the predicted branch direction. Figure 1 shows the architecture of our predictor.

Address generation. A static pseudo-random mapping, fixed at construction time using a seeded Mersenne Twister, partitions the 2158-bit input vector into N overlapping n-bit tuples. Each tuple becomes the address into one RAM node in each discriminator. Because the mapping is generated by shuffling the input bit indices and rolling the shuffle when exhausted, every RAM node receives a different random subset of the input bits. This decorrelates the RAM nodes, so each one observes a different "perspective" on the input.

Prediction. On a prediction request, the predictor first builds the input vector from the current architectural state, computes the N RAM addresses, and reads the corresponding counters from both discriminators. Each discriminator's response is the count of its RAM nodes whose addressed counter exceeds a bleaching threshold. If the two responses differ, the discriminator with the higher response wins. If the responses tie, the bleaching threshold is iteratively raised from 1 until one discriminator's response becomes strictly larger; if no separation is found at any threshold, the predictor defaults to not-taken. Algorithm 1 in the appendix summarizes this procedure.

Training. On branch resolution, only the discriminator corresponding to the actual outcome is trained: the addressed

counter in each of its N RAM nodes is incremented (saturating at $2^q - 1$). The opposite discriminator is left unchanged. The speculative GHR, real GHR, all five LHTs, and the GPHR are then updated. To prevent the system from drifting on misprediction, the speculative GHR is restored from the architectural GHR whenever the prediction was incorrect. Algorithm 2 in the appendix summarizes the training procedure.

Counter decay. To prevent the counters from saturating and to allow adaptation to phase changes in the workload, every counter in both discriminators is right-shifted by one bit every 2^{18} resolved branches. This halves the values of all stored counters simultaneously and is the only "aging" mechanism in the predictor.

Hardware budget accounting. The total storage of the predictor is dominated by the two discriminators, contributing $2 \cdot N \cdot 2^n \cdot q$ bits. Auxiliary state – the five LHTs (1024 entries each at widths 24, 16, 9, 7, and 5), the speculative and architectural GHRs, the GPHR, and the static input mapping – contributes a small, fixed overhead independent of N, n, and q. We use this accounting to match WiSARD configurations to the hardware budgets used in our evaluation.

C. Sensitivity analysis

One main goal of our project is to identify how well the WiSARD predictor scales across various hardware budgets. Thus, it is important to identify the importance of each of its main hardware parameters: tuple length, the number of RAM nodes, and counter width. To achieve this, we conducted a sensitivity analysis for each parameter.

We established a baseline WiSARD configuration consisting of an 8-bit tuple length, 94 RAM nodes, and 6-bit saturating counters. We then systematically isolated and swept each parameter independently while holding the other two constant. The evaluation was run against a representative subset of eight SPEC traces exhibiting diverse control flow behaviors, tracking the resulting IPC, Accuracy, and MPKI. The results revealed that each parameter scales differently, likely due to the fact that they target fundamentally different microarchitectural behavior.

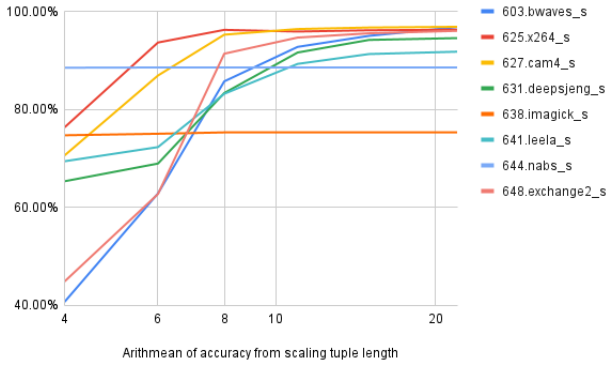

Fig. 2.

Increasing the tuple length, as shown in Figure 2, yielded the most drastic accuracy improvements. Because RAM nodes function as exact pattern memorizers, short tuples cause severe destructive aliasing by forcing mathematically distinct historical branch sequences to map to the same physical index. Once the tuple length reaches 15 bits, it seems that the physical tables are deep enough (32,768 entries) that aliasing is largely eliminated, and performance plateaus.

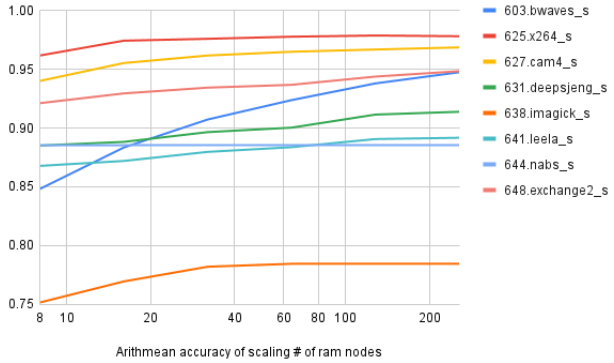

Fig. 3.

Scaling the number of RAM nodes, as shown in Figure 3, provided steady, marginal accuracy gains. Increasing the node count increases the physical sampling rate of the 2158-bit expanded input vector. and it seems that the performance provided by this wider context of architectural state is highly workload dependent.

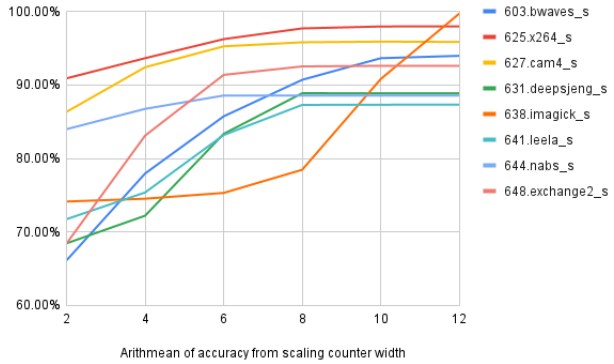

Fig. 4.

Expanding the saturating counters from 2 bits to 12 bits, as shown in Figure 4, exposed a critical dependency within the WiSARD bleaching algorithm. The benchmark 638.imagick remained stagnant at roughly 75% accuracy across smaller counter widths but rapidly scaled to 99.70% accuracy when provisioned with 10-bit and 12-bit counters. High-frequency branch patterns quickly saturate small counters in both the Taken and Not-Taken discriminators. When both discriminators uniformly reach the maximum representable value (e.g., 255 for an 8-bit counter), the bleaching threshold increments

to its maximum without isolating a clear winner, resulting in a prediction failure. Wider counters provide the necessary numerical range to separate the dominant state frequency from interference.

Ultimately, it seems that the performance of a particular predictor configuration is highly dependent on the workload, and how it interacts with the parameters of that specific configuration. This means that at limited hardware budgets, where we must make a decision between provisioning more or less hardware for a particular parameter, that that decision must be driven by the behavior of the target workload.

IV. Methodology

For our simulations we used the latest ChampSim version with a configuration emulating a single Intel Golden Cove core. All simulations were run on HPRC Grace. ChampSim output logs were parsed with a python script to extract IPC, Accuracy, and MPKI values of each simulation; the script outputs these values into a CSV which is then pasted into a spreadsheet for manipulation, analysis and graph generation.

We used a set of 100 ChampSim traces as the benchmark for our experiments. 29 traces from SPEC2006, 20 traces from SPEC2017, 26 traces from GAPS, 12 traces from XSbench, and 13 traces from DPC4. We ran 100 million instructions from each trace, along with a 50 million instruction warm-up period beforehand.

TABLE I
WiSARD Hardware Budget Configurations

Target Budget	Total Size	Tuple Bits	Counter Bits	RAM Nodes
24 KB	**24.2 KB**	10	6	11
48 KB	**47.7 KB**	11	8	10
96 KB	**95.7 KB**	12	8	11
192 KB	**183.7 KB**	13	8	11
384 KB	**391.7 KB**	13	12	16
768 KB	**775.7 KB**	14	12	16
1536 KB	**1543.7 KB**	15	12	16

We compared our WiSARD predictor against a storage-scaled variant of ChampSim's Hashed Perceptron (HP) predictor across 7 hardware budgets from 24KB to 1536KB. To accommodate this range, we modified the HP predictor's code by lowering the number of tables from 16 to 12, keeping the min and max history lengths the same while recalculating the terms in between to fit a 12 term geometric series.

For the WiSARD predictor, the results from the sensitivity analysis made it abundantly clear that we need to prioritize a high enough tuple length and counter width, before adding more RAM nodes. However, at larger hardware budgets, it is unclear whether we should prioritize increasing tuple length or adding more RAM nodes; nonetheless, we kept the general heuristic of prioritizing tuple length throughout the entire hardware range. Table 1 shows a detailed description of the configuration for our WiSARD predictor at each hardware budget.

V. Evaluation

In this section, we analyze the performance of our WiSARD predictor and compare it against the Hashed Perceptron predictor. Figures 5 through 14 show the geometric mean MPKI and arithmetic mean accuracy across each of the 5 benchmark suites (SPEC06, SPEC17, GAPS, XSBench, and DPC4).

Fig. 5.

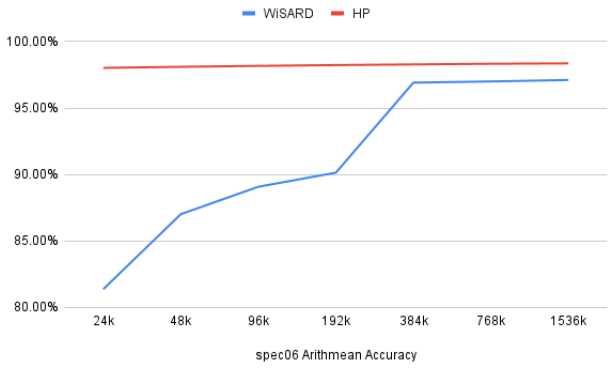

Fig. 6.

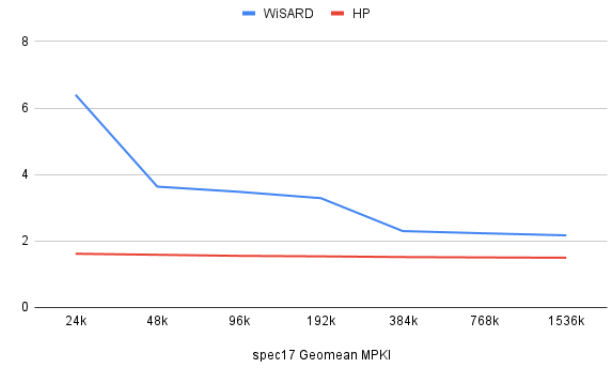

Fig. 7.

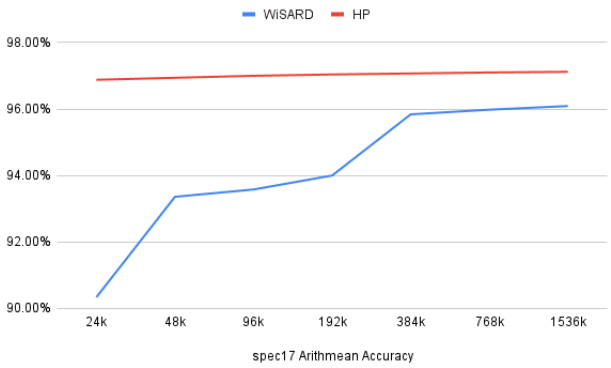

Fig. 8.

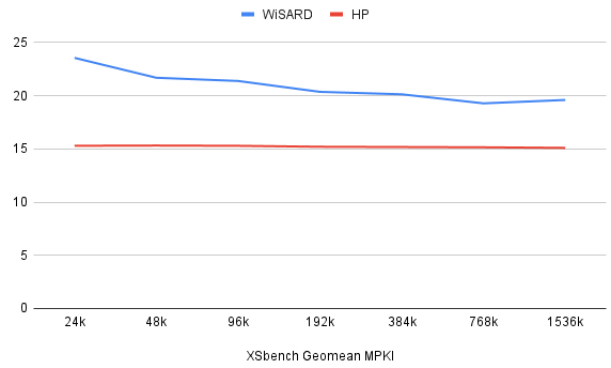

Fig. 11.

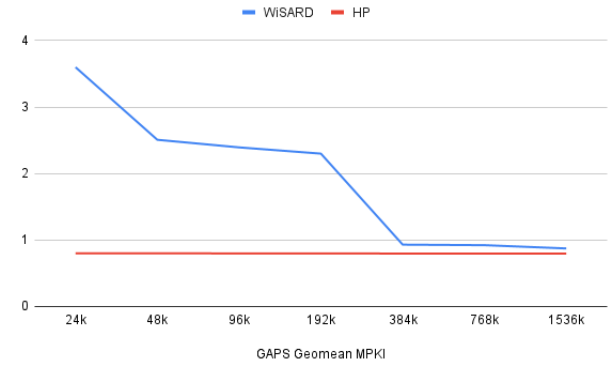

Fig. 9.

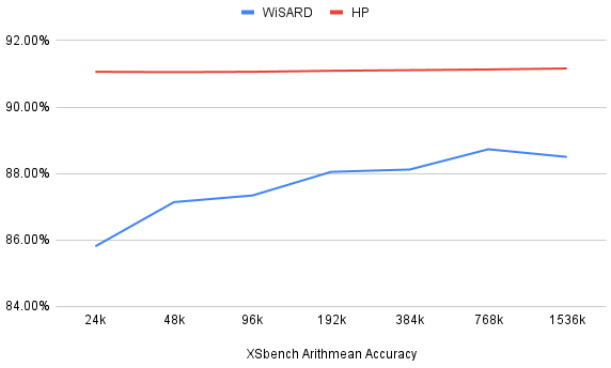

Fig. 12.

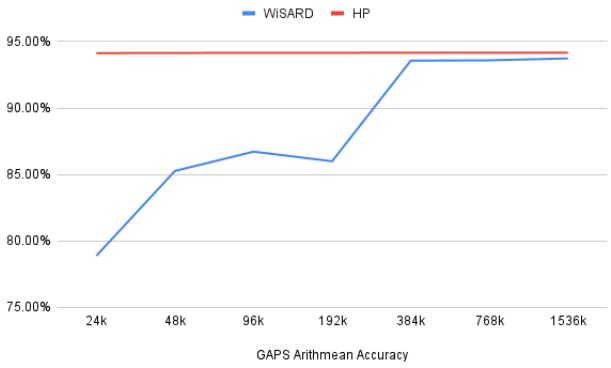

Fig. 10.

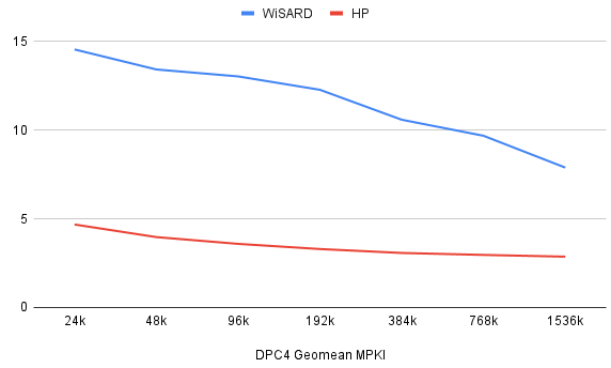

Fig. 13.

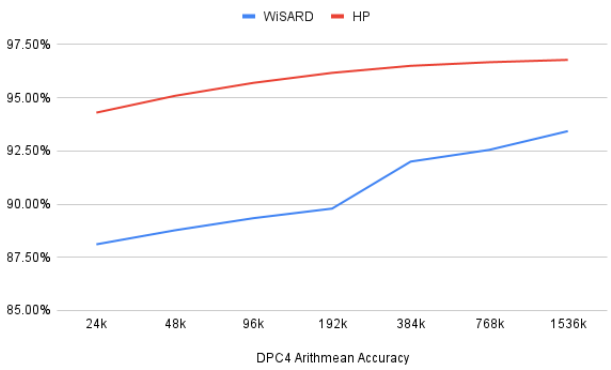

Fig. 14.

A. Aggregate performance

Across the five benchmark suites, the Hashed Perceptron consistently achieves lower Geometric Mean MPKI and higher Arithmetic Mean Accuracy than our WiSARD predictor. However, the performance scaling curves reveal a critical microarchitectural threshold.

For our WiSARD predictor, the vast majority of accuracy gains are realized by the 384KB hardware budget. Scaling the predictor up to 768KB and 1536KB yields heavily diminishing returns across all suites. This demonstrates that the WiSARD architecture is not unboundedly hardware hungry; the evaluated WiSARD model's feature representation fully saturates its informational capacity at moderate storage capacities, making it a realistic candidate for physical implementation.

B. Optimal configurations

The hardware configurations evaluated in this study (ranging from 24KB to 1536KB) were derived using generalized heuristics from the preliminary sensitivity analysis detailed in section III. Parameters such as tuple length, counter width, and the number of RAM nodes were scaled using broad calculations rather than exhaustive design space exploration.

Because WiSARD is highly sensitive to the exact balance of pattern resolution (tuple length) and input coverage (RAM nodes), these generalized configurations are almost certainly suboptimal for individual workloads. A more rigorous parameter search at each specific hardware budget would likely yield configurations that can garner better performance. Thus, our observed trends reflect a representative functional trajectory rather than a fully exhaustive search.

C. Limitations

Our current WiSARD implementation lacks three major domain specific optimizations that modern neural predictors rely on:

1) Feature engineering: The 2158-bit input vector was constructed using the static feature weights from the original WiSARD branch prediction paper, and we did not confirm whether these weights were truly optimal. Additionally, it does not incorporate any advanced feature extraction or additional architectural state, which are standard in modern predictors like the Multiperspective Perceptron predictor.

2) Optimal random mapping: The routing of the 2158 input bits to specific RAM nodes was handled by a generic C library random number generator. In a Weightless Neural Network, performance is strictly bound by the specific input mapping; informative bits must be routed to the same RAM node to create meaningful memorization states. Replacing the generic random distribution with a structurally optimized mapping function would directly improve prediction accuracy.

3) Additional features: We also experimented with adding feature families inspired by the Multiperspective Perceptron (MPP) predictor [7], including:

- modulo-history features
- global-history/path interactions
- coarse or "blurry" path-history summaries
- branch-recency position information
- lightweight loop-iteration counter

These features were evaluated in a preliminary ablation study on eleven SPEC2017 traces. Starting from our current WiSARD variant, whose conditional MPKI on this subset was 6.715, the best MPP-inspired variants reduced conditional MPKI to roughly 5.58, an improvement of about 16.8%. This indicates that WiSARD can benefit from richer architectural context and that the baseline 2158-bit input vector is not fully feature optimal.

However, these features were not included in the final hardware-budget sweep. First, the MPP-style feature additions were tested only as a pilot study and were not tuned independently across every storage budget. Second, their benefit did not scale cleanly with larger WiSARD configurations. At large budgets, WiSARD performance was already dominated by tuple length, RAM-node count, and counter width; adding more input features increased the candidate feature pool but did not proportionally improve the probability that useful feature combinations were routed into the same RAM node. In contrast, perceptron-style predictors are explicitly designed to combine many weak feature perspectives through signed accumulation, while WiSARD still depends on discrete tuple formation and exact-pattern memorization. Thus, although MPP-inspired features are promising future work, including them in the main study would have mixed an incompletely tuned feature-engineering experiment with the storage-scaling comparison that this report focuses on.

VI. Conclusion

In this project, we implemented a cycle-accurate WiSARD Weightless Neural Network branch predictor within the ChampSim framework and evaluated it across five diverse benchmark suites.

While the baseline Hashed Perceptron outperformed our WiSARD implementation across the evaluated hardware budgets, the results highly validate the Weightless Neural Network paradigm. The Hashed Perceptron is a highly refined architecture that has benefited from extensive tuning. In contrast, our WiSARD predictor was deployed as a raw baseline utilizing

unoptimized heuristic scaling, static legacy feature weights, and a generic random mapping function, serving as a pessimistic lower bound.

Despite these severe structural disadvantages, the WiSARD predictor achieved accuracy and MPKI metrics that closely trailed the Hashed Perceptron, capturing the majority of its predictive capability at a moderate hardware budget. By eliminating signed arithmetic and operating entirely on memory addressing and boolean thresholding, WNNs offer a fast, storage-efficient classification mechanism. The fact that an untuned WNN can approach HP's performance proves it has massive architectural potential, making it a prime candidate for future research focusing on optimized feature mapping and dynamic hardware allocation.

REFERENCES

[1] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction-Level Parallelism*, vol. 8, 2006.

[2] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001, pp. 197–206.

[3] I. Aleksander, W. Thomas, and P. Bowden, "WISARD—a radical step forward in image recognition," *Sensor Review*, vol. 4, no. 3, pp. 120–124, 1984.

[4] L. A. Q. Villon *et al.*, "A conditional branch predictor based on weightless neural networks," *Neurocomputing*, vol. 555, p. 126637, 2023.

[5] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 3, pp. 280–300, 2005.

[6] B. P. Grieco, P. M. Lima, M. De Gregorio, and F. M. França, "Producing pattern examples from 'mental' images," *Neurocomputing*, vol. 73, no. 7–9, pp. 1057–1064, 2010.

[7] D. A. Jiménez, "Multiperspective perceptron predictor," in *The Fifth Championship Branch Prediction Competition (CBP-5)*, June 2016.

APPENDIX A

Algorithm 1 WiSARD Branch Predictor: Prediction Phase

Constants: $N_{RAM} = 64$, $T_{bits} = 10$
State: D_T (taken disc.), D_{NT} (not taken disc.) GHR_{spec}, $LHT_{0..4}$, $GPHR$

function PREDICTBRANCH(PC)
  $features \leftarrow$ Extract features (PC, GHR_{spec}, $PC \oplus GHR_{spec}$, $LHR_{0..4}$, $GPHR$)
  $input\_vec \leftarrow$ Construct 2158-bit vector by replicating $features$ by weights
  for $i = 0$ **to** $N_{RAM} - 1$ **do**
    $addrs[i] \leftarrow$ Extract T_{bits} from $input\_vec$ using $static\_mapping[i]$
  end for
  $S_T \leftarrow D_T$.GETRESPONSE($addrs$, 0)
  $S_{NT} \leftarrow D_{NT}$.GETRESPONSE($addrs$, 0)
  if $S_T > S_{NT}$ **then**
    $pred \leftarrow$ True
  else if $S_{NT} > S_T$ **then**
    $pred \leftarrow$ False
  else
    ▷ Inc. threshold iteratively until tie breaks
    $pred \leftarrow$ APPLYBLEACHING($addrs$)
  end if
  Shift $pred$ into GHR_{spec} (Speculative)
  return $pred$
end function

APPENDIX B

Algorithm 2 WiSARD Branch Predictor: Training Phase

Constants: $T_{decay} = 2^{18}$
State: D_T, D_{NT}, GHR_{spec}, GHR_{real}, $LHT_{0..4}$, $GPHR$

function LASTBRANCHRESULT($PC, outcome$)
  $decay\_ctr \leftarrow decay\_ctr + 1$
  if $decay\_ctr \geq T_{decay}$ **then**
    ▷ Right-shift counts
    D_T.DECAY( )
    D_{NT}.DECAY( )
    $decay\_ctr \leftarrow 0$
  end if
  if $outcome ==$ True **then**
    ▷ Saturating inc.
    D_T.TRAIN($addrs$)
  else
    D_{NT}.TRAIN($addrs$)
  end if
  Shift $outcome$ into GHR_{real}
  if $pred \neq outcome$ **then**
    ▷ Recover state
    $GHR_{spec} \leftarrow GHR_{real}$
  end if
  Update $LHT_{0..4}$ using PC and $outcome$
  Update $GPHR$ using PC
end function