Multiperspective Perceptron Branch Predictor

ISCA 2026 Submission **#70** – Confidential Draft – Do NOT Distribute!!

***Abstract*—The *multiperspective perceptron predictor* (MPP) is a conditional branch predictor based on the idea of viewing branch history from multiple perspectives. The predictor is a hashed perceptron predictor using previous outcomes and addresses of branches organized in ways that go beyond traditional global and local history. MPP focuses on finding useful perspectives of control-flow history rather than relying on a small, fixed set of features. This paper describes MPP and its novel features. It also shows that combining MPP with TAGE yields accuracy superior to either branch predictor by itself because of complementary properties of both. The paper presents a design methodology based on genetic algorithms that automatically finds good features for the predictor, freeing designers from hand-crafting specialized features. At a 192 KB budget on a set of evaluation workloads sourced from industry, our combined predictor yields an average MPKI of 2.386, 11% lower than an equivalent sized TAGE and competitive with the state of the art TAGE-SC.**

I. Introduction

Conditional branch predictors typically use global and/or local (*i.e.* per-branch) history to match or find correlations between previous branch outcomes and the current branch. However, there are many other ways to organize history beyond global and local. For example, inspired by the work of Albericio *et al.* [4], Seznec *et al.* propose using the innermost loop iteration counter (IMLI) [27] as a feature in a hashed perceptron predictor together with local and global history as an additional component of a TAGE-SC-L predictor.

This paper describes a hashed perceptron predictor [31] that incorporates many kinds of branch history to make a prediction. This *multiperspective perceptron predictor* (MPP) is combined with TAGE-SC-L to achieve good accuracy given a 192KB hardware budget. The main idea is that branch prediction is limited not by algorithms but by which perspectives of control-flow history we exploit. MPP provides a systematic, general framework for discovering these perspectives. We argue that the space of design space of useful branch-prediction features is far larger than the traditional features explored in previous work (*e.g.* local, local, path-based), and that exploring this space requires automated discovery rather than human intuition. Previous work focuses heavily on intricately detailed bespoke configurations. This sort of "mad scientist" approach is not scalable to future advances in reducing misprediction. We propose automated discovery of features with genetic algorithms.

In this paper, we describe the multiperspective approach to branch prediction. We show that its ability to exploit correlations between branch history and outcomes is complementary to TAGE. We show how to combine the two approaches to yield a predictor with accuracy competitive with the state of the art.

At a realistic 192 KB budget on the 673 CBP 2025 evaluation traces, our combined MPP+TAGE-SC-L predictor achieves an average MPKI of 2.385, essentially matching the state-of-the-art TAGE-SC (2.390 MPKI) while significantly outperforming standalone TAGE (2.680 MPKI) and standalone MPP (2.638 MPKI)."

This paper makes the following contributions: 1) We propose the concept of multiperspective branch histories as a design space. 2) We propose a genetic-algorithm-based methodology to automate the search of this design space. 3) We propose several novel control-flow history features beyond those described in previous work. 4) We demonstrate how to combine MPP with a TAGE-based predictor to leverage the strengths of both, resulting in good accuracy, and 5) We give a quantitative evaluation of the resulting predictors in the context of modern Arm-based workloads

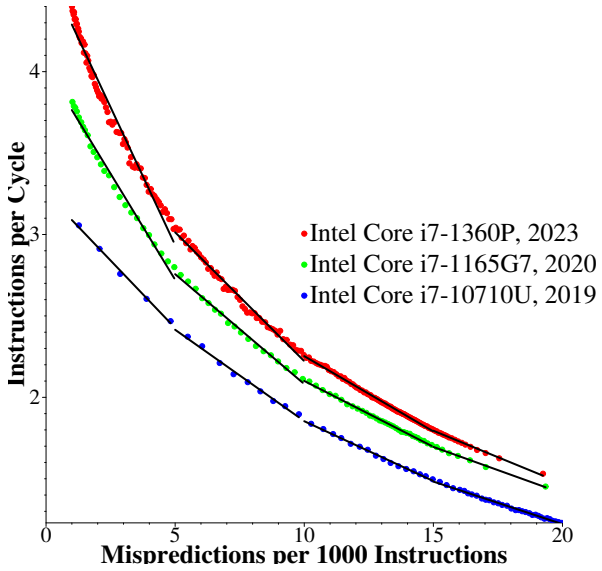

Fig. 1. Improving predictor accuracy pays off more in recent processors

II. Motivation: Branch Prediction Research is Increasingly Important

To motivate why improvements in branch accuracy still matter, we give experimental results from real systems that isolate the effect of mispredictions on performance. Over the years, branch predictor accuracy has become more important because of the increasing size of processor instruction windows. Figure I shows the relationship between branch accuracy (measured as mispredictions per 1000 instructions, MPKI) and processor performance (measured as instructions per cycle,

IPC) for recent Intel processors. The figure shows the results of an experiment using a microbenchmark designed to execute the same number of instructions with moderate cache activity while varying the branch predictor accuracy by increasing the complexity of branch patterns. There are two interesting observations we can make from this graph:

First, the more accurate the predictor is, the more gains can be made by making it even more accurate. Performance is not linear with respect to accuracy. As MPKI is reduced, performance increases superlinearly. The line segments in the graph represent piecewise linear regression of the relationship between IPC and MPKI. For example, for the Intel Core i7-1360P, in the range between 5 and 10 MPKI, reducing MPKI by 1 results in an increase in IPC of 0.16. However, in the range from 1 to 5 MPKI, reducing MPKI by the same amount results in more than double the performance gain, 0.34 IPC per MPKI. This non-linear improvement in performance is due to the fact that, as branch mispredictions become more rare, each one has more of an impact, necessitating a pipeline flush and redirection of the instruction fetch. When there are many mispredictions, each one can be discovered relatively quickly, allowing relatively few wrong-path instructions that contribute nothing to performance to enter the pipeline.

Second, the rate of improvement accelerates for newer processors. For the Intel Core i7-10710U released in 2019, reducing MPKI from 5 to 1 results in a 25% improvement in IPC. For the newer Intel Core i7-1360P released in 2023, the same MPKI reduction results in a speedup of 45%. More recent processors are built with deeper fetch queues and larger instruction windows. In the absence of branch mispredictions, these features would yield vast improvements in performance. However, as these structures grow, they allow branch mispredictions to take the processor further down the wrong path, resulting in larger performance penalties.

The trend of increasing execution resources continues. A given improvement in branch prediction accuracy in five years will yield more performance improvement than the same improvement now. The research proposed in this paper can have a real impact in that timeframe.

III. Background and Related Work

Today, most branch predictors are variations or combinations of two main algorithms: TAGE [26] and Perceptron [11].

TAGE hashes global branch and path histories of different lengths to index into various tables composed of tagged saturating counters. TAGE-SC-L [22], [23], which won the fifth championship branch prediction competition (CBP5), is a popular TAGE variant that uses additional loop predictor and perceptron components to improve accuracy. Intel does not publicly reveal its branch predictor designs, but reverse-engineering work shows that recent Intel processors likely use the TAGE algorithm [33]. BATAGE is a TAGE-based predictor that improves TAGE learning [18]. Public microarchitectural descriptions indicate that recent AMD cores (*e.g.* Zen 2) employ both TAGE-style and perceptron-based branch predictors in combination. TAGE-SC-L combines TAGE with a perceptron component called the "statistical corrector" (SC). The SC hashes several global and local history features to index tables of weights to form a sum, then adjust the weights with perceptron learning. These features are hand-tuned by Seznec to deliver minimal mispredictions. One can view the SC as a sort-of multiperspective perceptron predictor with bespoke features. Our work greatly increases the kind of features available, and presents a design methodology using genetic algorithms that automates the selection of features so that hand-tuning them is not necessary.

The original perceptron branch predictor [12] used a table of perceptron weights vectors to predict the outcome of conditional branches by selecting a vector of small integer weights using the branch address, then finding the dot product of this vector with the global history in a bipolar representation. The weights were trained with perceptron learning: when the outcome of the branch matches a branch in the history, the corresponding weight is incremented, otherwise it is decremented, saturating at the maximum and minimum values for the bit width of the weights. A bias weight is also trained with the outcome of the branch without respect to any previous branch in the global history.

A number of improvements were proposed to improve accuracy and latency: ahead pipelining with path information [10], piecewise linear decision surfaces to get around the linear separability problem of perceptrons [14], and hashing history and path features to reduce the number of tables needed [16], [21], [31]. Perceptron-based branch prediction has gained considerable adoption in industry by companies such as AMD, Samsung, Oracle, IBM, and others [2], [9], [28] and other areas of microarchitectural prediction [3], [5], [32].

The hashed perceptron predictor [31] is similar to an idea of Loh and Jiménez called *modulo path-history* [16], while O-GEHL [21] is a very specific instance of the general technique. The idea is to have several tables, each indexed by a different hash of branch history. The tables have saturating confidence weights. The selected weights are summed and the prediction is taken if the sum is at least zero, not taken otherwise. On a mispredict or low-confidence correct prediction, the corresponding weights are incremented if the branch is taken, decremented otherwise. The hashed perceptron predictor, like modulo path-history and O-GEHL, improves over the original perceptron predictor [12] by breaking the one-to-one correspondence between weights and history bits, allowing a more efficient representation.

Features for MPP are evolved using a genetic algorithm. The procedure and general approach to prediction is similar to that of the Multiperspective Reuse Predictor [13], a dead block predictor intended to improve last-level cache replacement. That work used a simpler stochastic search, with authors mentioning that a genetic algorithm would have been too costly computationally. With our simpler simulator and several subsequent years of improvements in supercomputing technology, we find that using a genetic algorithm works and

outperforms our initial attempts with simpler kinds of search. Branch predictors have been designed with genetic algorithms before. Emer and Gloy describe an encoding of contemporary branch predictor designs corresponding to combinations of two-level adaptive branch predictors to be used by a genetic algorithm to discover new, superior configurations [6].

IV. Design and Evaluation Assumptions

In this section, we describe the methodology used to design and evaluate MPP, as well as assumptions we make about branch predictor design for this project.

A. CBP 2025 Workloads

The 6th Championship Branch Prediction (CBP 2025) was an ISCA 2025 workshop inviting participants to submit simulator code for conditional branch predictors [1]. The organizers provided a detailed out-of-order simulation model as well as 105 training traces for workloads sourced from Arm, Inc. Participants developed their predictor code targeted to optimize simulated aggregate mispredictions per 1000 instructions (MPKI) and instructions per cycle (IPC) over these workloads. Organizers evaluated entries based on an undisclosed set of 673 testing traces also sourced from Arm. After the workshop, organizers released the full set of 673 evaluation traces. Both sets of traces include many workloads from the following categories: compress, fp, infra, int, media, and web. The evaluation traces each represent snippets of between 10 and 130 million instructions, with a median of 40 million instructions. The astute reader may ask, "Why not use SPEC or other disclosed benchmarks?" We do not know what actual applications were used to generate these traces, but we believe it is safe to assume they are relevant to Arm's corporate mission to provide high-performance general-purpose microprocessor IP core designs. Arm employees spent considerable effort curating these traces, getting their legal department to clear them, and developing the simulation infrastructure, all of which would have had to have been justified as helping Arm's bottom line as an investment in pushing the state of the art in branch predictor design. Thus, these traces represent the most appropriate set of workloads for branch predictor research in 2025. For this study, we use the same 105 training traces to develop our predictors, and the same 673 testing traces to present testing results.

B. Implementation Concerns

This paper proposes a highly accurate branch prediction algorithm. We do not propose a particular realizable implementation, but rather we explore the capabilities of this algorithm, leaving designers to glean realistic implementation ideas for future branch predictors.

In branch predictor design, there are two main sources of difficulty: complexity within the branch predictor itself (*predictor complexity*, and complexity at the interface between the branch predictor and other components (*interface complexity*). We propose an algorithm that may increase predictor complexity, but we stop short of allowing increased interface complexity. An example of predictor complexity would be the details of repairing speculative local histories [30].

Most conditional branch predictors use control-flow history as the only kind of input to find correlation with branch outcome. In this study, we also consider only control-flow history. Some previous work goes outside of this paradigm to explore new sources of correlation. For example, we note that the winner of CBP 2025, RUNLTS [15], uses register values combined with control-flow history. This kind of predictor can capture hard-to-predict data-dependent branches. However, it also introduces increased interface complexity, as the branch predictor in the fetch unit will have to communicate with disparate parts of the processor, and the result will be very dependent on microarchitectural details and decisions made outside of the instruction fetch team.

In general, we find that problems with predictor complexity can often be overcome by clever RTL engineers, but problems with interface complexity are more difficult due to the social cost of designing and implementing an interface, requiring coordination among unrelated groups of engineers. Thus, aligning ourselves with industry's goal of streamlining the design process, we choose to see how far we can push branch prediction algorithms based on control-flow history.

C. Hardware Budget

Participants were asked to limit their predictors to 192KB of total mutable state. This is a large increment over the previous championship from 2016 that was divided into tracks with 8KB and 64KB. Details of branch predictor design are closely guarded by industry. There is no paper we can cite that will definitively state the size (or any other implementation details) of, say, the latest Intel, Apple, AMD, or Arm branch predictor. The most recent documented industrial predictor that gives a size is the Samsung Exynos M5, a mobile processor that used 32KB in the branch direction predictor in 2020 [9]. However, we have very good reasons to believe that the 192KB budget given for CBP 2025 is quite realistic for modern and near-future high-performance processors. Thus, we believe our results at this hardware budget would be useful for designers of future branch predictors.

D. MPKI is the Figure of Merit

While the CBP 2025 simulator provides IPC estimates, we report MPKI as our figure of merit. IPC in a publicly released simulator depends on many microarchitectural details such as timing, BTB and instruction cache behavior, replay mechanisms, wrong-path side effects, etc. that can vary substantially from real designs and are often abstracted for tractability. In contrast, MPKI is directly determined by the conditional predictor itself and is the metric used in most industrial evaluations of branch prediction algorithms. Thus, we use MPKI to isolate predictor quality independent of simulator-specific microarchitectural assumptions.

E. Our Simulator

We developed a fast simulator compatible with predictor code written for CBP 2025. We instrument the CBP 2025

simulator to emit traces containing branch type (e.g. conditional, return, indirect, etc.), address, target, and taken/not-taken outcome. We collect traces for all the CBP 2025 traces and use them as input for our fast simulator that drives the conditional branch predictor code. By running only the branch predictor simulator code and not the detailed out-of-order simulation, our simulator is an order of magnitude faster than the CBP 2025 simulator, allowing it to be used for computationally intensive design space exploration we will describe in Section VI.

V. Multiperspective Perceptron Predictor

The multiperspective perceptron predictor (MPP) is a hashed perceptron predictor that uses several different kinds of control-flow history information to form hashes into tables, read out weights, sum them, and threshold the sum to make a prediction. The sum of weights is called y_{out} from the original perceptron paper. It is updated by incrementing or decrementing 6-bit saturating weights based on whether the branch is taken or not taken, respectively. The paper describes the features in detail. Training is done for mispredicted branches as well as for branches where the magnitude of the perceptron sum fails to exceed a training threshold, θ, that is set using a simplified version of Seznec's training algorithm from O-GEHL [21].

MPP uses a transfer function to improve accuracy by boosting values of more confident weights. The features and the transfer function were initially tuned using a genetic algorithm, then fine-tuned by making small random tweaks and hill-climbing.

To index the prediction tables, the hash value of a feature is computed using recent history information, hashed together with the address of the branch to be predicted, then taken modulo the size of the prediction table. The weights corresponding to the indices are read, summed, and thresholded to make a prediction of taken or not taken.

A. MPP Features

The novelty of MPP lies within its distinct set of features derived from control-flow history. We describe each kind of features along with the parameters it accepts. Each feature is used to generate a hash that is then XORed with the branch address to form an index into a table of weights.

After exploring many organizations, we found the following features useful for branch prediction:

B. Traditional Features

The following features from traditional branch predictors are used:

a) **GHIST:** Global history is the outcome of a branch shifted into a large register, with 0 meaning not taken and 1 meaning taken [17], [34]. It is hashed by bitwise XORing multi-bit blocks of histories. Parameters to this feature give the starting and ending indices of the history register to hash. This feature is not actually used in the final configuration, but rather only used in combination with other histories as described below.

b) **PATH:** Path history is the sequence recent branch addresses. Addresses truncated to 16 bits are shifted into an array. Parameters include a depth and a shift. The array is hashed by accumulating a hash value up to the given depth.

c) **LOCAL:** Local history is a first-level table of per-branch shift registers selected by a hash of the branch address, then hashed as an index into a second level table of perceptron weights. Thus, the table corresponds roughly to a PAs-style local history predictor in Yeh and Patt's taxonomy [35].

d) **GHISTPATH:** This is a combination of GHIST and PATH. The GHIST and PATH features are XORed together as they are computed.

e) **BIAS:** The value of this feature is 0. It will be XORed with the branch address to form an index into the table. This feature tracks the tendency of a branch to be taken or not, regardless of other branch history. It corresponds roughly to what is commonly known as a bimodal or "Smith" style branch predictor [29] that uses the branch address to index a table of counters.

C. Novel Features

The following novel features are used:

a) **Alternate IMLI:** Seznec introduced the innermost loop iteration counter (IMLI) [27], an elegant and simple form of local history predictor. It needs only a few bits of state to form an index into a perceptron table: a counter. In Seznec's formulation, when a backward branch is encountered, the IMLI counter is incremented if the branch is taken, otherwise it is reset. For some branches, this counter is highly correlated with branch outcome. Incrementing on a backward taken branch captures the behavior of loops with back-edges at the bottom of the sequence of instructions comprising the loop body. These kinds of loops are commonly produced by many compilers.

We explore an alternate IMLI: when a forward branch is encountered, the IMLI counter is incremented when the branch is not taken, and reset when it is taken, representing a loop exit. This represents the case where the loop exit is at the top of the loop body, followed by an unconditional jump at the bottom. We find that only the forward formulation IMLI turned out to be useful for MPP. We speculate that the modern ARM compilers that produced the code used to generate the CBP 2025 traces favor the sort of loop exits exploited by this version of IMLI. In practice, both could be combined depending on the nature of the expected workloads.

b) **MODHIST:** Some branches in the global history are not correlated to branch outcome, but a single bit different in two histories can lead to two different hash values, causing longer training times and increased aliasing pressure. Thus, both TAGE and hashed perceptron are susceptible to what we call *branch misalignment*. Suppose a branch branches over another branch. The second branch sometimes appears in the branch history and sometimes does not appear, leading to the same branch outcomes appearing in different locations in the global history. Neither traditional nor hashed perceptrons, nor other hashed-based predictors such as TAGE, can handle

this problem without expending additional table entries and increasing training time.

We propose "modulo history" where only branches with addresses congruent to 0 modulo some modulus are recorded. Incongruent branches causing misalignment are filtered out of the history and ignored. The problem is that those filtered branches may indeed have some correlation; that correlation is hopefully captured by the other features. Modulo history has two parameters: the modulus (a small integer), and the history length.

c) **MODPATH:** Modulo path history is the same as modulo history, but uses branch addresses instead of outcomes. The parameters are the same and the hash is computed similarly to the PATH feature.

d) **GHISTMODPATH:** This feature combines modulo history with modulo path history in a way analogous to GHISTPATH above.

e) **RECENCY:** MPP keeps a list of recently visited branch addresses in a structure called a recency stack with N elements. As part of updating branch histories, the branch address is inserted into the recency stack with least-recently-used (LRU) discipline, exactly as a set in an LRU-managed cache. That is, if the address is not in the stack, the least recently used element is evicted and the new address is inserted at position 0 (the most recently used (MRU) position). If the address is found in the stack, it is moved to the MRU position and all subsequent elements are moved back by one. The `RECENCY` feature hashes this stack as an index into a table, The parameters are the depth into the stack to hash, a shift by which to shift the accumulator after hashing.

f) **RECENCYPOS:** This feature also uses the recency stack. The position of the currently predicted address in the stack is sometimes highly correlated with the outcome of the branch. The `RECENCYPOS` features search the recency stack for the current branch address and used the position $(0..N)$, or N if the address is not found, as an index into a perceptron table.

Figure 2 illustrates when this feature is useful. For the CBP 2025 workloads, we profiled every static branch executed at least 30 times, collecting pairs of recency position and branch outcome. The figure shows the absolute value of the correlation coefficient of sequences of such pairs, sorted by coefficient. That is, the x axis denotes a given static branch that has executed at least 30 times, and the y axis shows the correlation of that branch's outcome with its recency position. About 1% of all static branches have a very strong correlation coefficient of at least magnitude 0.8, and 2.5% of all branches have a strong correlation of at least magnitude 0.5. About 5% of branches have a moderate correlation between magnitudes 0.3 and 0.5, and most other branches have weak or no correlation. RECENCYPOS is unique among the other features in that it has numerical semantics rather than being a hash of a string of bits. It also encoded only a small amount of information, a 5 or 6 bit distance, rather than a signature of a long history. Although RECENCYPOS is not the dominant feature in our evolved sets, this analysis shows that a small but non-negligible fraction of branches show strong recency-position correlation, justifying its inclusion as one of the perspectives explored by MPP.

g) **BLURRYPATH:** Traditional branch path history is a precise record of the sequence of recent branches. "Blurry" path history records larger-granularity regions where branches have been encountered, and only shifts a region into the history when a new region is entered. Regions are computed as branch addresses right shifted by a certain amount given as a parameter. When a branch from a new region is encountered, the previous region is shifted into the history. The feature's parameters are the amount to shift, the depth within the history to hash, and a parameter that controls how much to shift each region number while generating the hash.

h) **ACYCLIC:** This feature keeps a history register H of length n and records the outcome of a branch with address PC in $H[PC(\mathrm{mod}n)]$. The intuition is that we would like to remove the effect of loops (*i.e.* cycles) in the history and just keep the most recent outcome of any branch. The parameters is the size of H.

i) **BACKPATH:** This feature is like PATH, but only records the history of backward branches.

j) **TAGE:** We present a version of MPP combined with the version of TAGE-SC-L that won CBP 2016 [24]. This feature incorporates the TAGE-SC-L prediction and confidence shifted left by tuned parameters, allowing MPP to benefit from the cases where TAGE-SC-L provides more accuracy than MPP alone. This feature seems to introduce a sequential dependence on the TAGE-SC-L prediction that can be mitigated by computing both predictions in parallel and muxing off the correct output.

D. Discussion of Features

None of the novel features are particularly good predictors by themselves. However, together with each other and with the traditional features, they provide multiple perspectives on branch history and allow finding new correlations. We explored many other potential features, but the ones described above proved helpful on the CBP 2025 training workloads.

VI. Feature Selection

This section described the methodology for finding a set of features for MPP for the best accuracy.

Feature selection was done using a genetic algorithm followed by a hill-climbing phase where hundreds of thousands of combinations of features were evaluated on the CBP 2025 traces. Since the main work is done by the genetic algorithm, we say that a set of featured was "evolved" for MPP.

We developed two versions of MPP: a standalone version that uses only perceptron learning, and a version that combines TAGE-SC-L with MPP, using a choosing algorithm to select the prediction most likely to be correct. We evolved separate sets of features for the standalone and combined predictors. The two sets of features are described in Figures I and VI-B. Since TAGE-SC-L already incorporates global history in the TAGE component, the combined MPP's features have more of

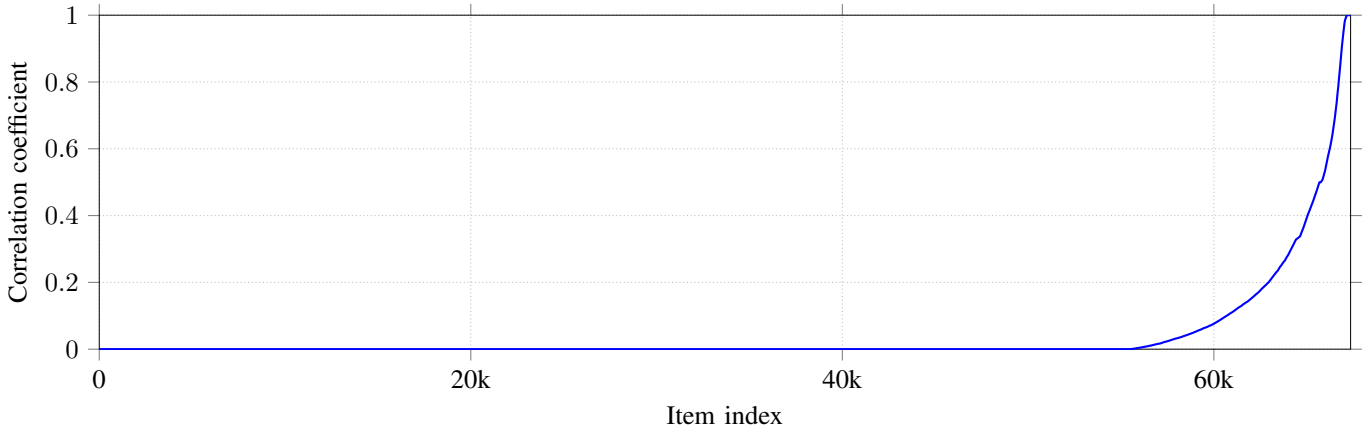

Fig. 2. Analysis of the RECENCYPOS feature. For each of the approx. 67,000 static branch addresses executing at least 30 times in the training traces, the graph shows the correlation coefficient between the recency position and the taken/not-taken branch outcome.

the modulo-history and other somewhat orthogonal features than the standalone features, which features global history much more heavily.

A. Details of the Genetic Algorithm

1) Genetic Algorithm: A genetic algorithm emulates Darwinian evolution to search a multidimensional space for a point yielding a desirable value for a fitness function. Points in the space are encoded as strings. A population of these strings is initialized to random values, and each string's fitness is evaluated. Then the algorithm proceeds to search for highly fit strings by producing subsequent generations of the population. New elements of the next generation are made by *crossover*, *i.e.* combining two highly fit individuals from the previous generation, and *mutation*, *i.e.* altering parts of the string with some low probability to provide diversity. After many generations, the fitness of the best individual of the population tends to converge. At that point, the algorithm stops and the most fit individual is chosen as the desirable point. There is no guarantee of optimality, but genetic algorithms have proven in practice to yield highly satisfactory results searching complex non-linear spaces where traditional optimization methods are inapplicable.

2) Adapting the Genetic Algorithm for MPP: We use a genetic algorithm to search for a set of good features for MPP. The fitness function is the average MPKI computed by our fast simulator over the 105 CBP 2025 training traces. On our compute cluster featuring Intel Xeon Gold 6248R 3.00GHz processors, one computation of the fitness function takes approximately 1500 seconds, depending on the computational nature of the particular features and contemporaneous load on the machines. This high latency presents a challenge for running the genetic algorithm, but fortunately the algorithm to evaluate the population is embarrassingly parallel.

We choose a population size of 1024 individuals. We encode MPP features as binary strings divided into portions representing feature type, four small integer parameters, and another parameter that selects one of three hash functions for the process of deriving a table index from the features and branch address. The strings are initialized to random binary values for the first generation. The fitness function for each individual is evaluated and recorded in a set of files maintained throughout the genetic algorithm process. Then, the genetic algorithm begins in earnest. We run 512 copies of a program that selects two individuals of the population at random for crossover and mutation. This process finishes producing a new generation after approximately one half hour. The resulting individuals are placed back into the population, ranked by fitness. Initially, any of the 1024 individuals may be selected for crossover, but over time the algorithm increases the probability that an individual is chosen proportionately to its fitness. Thus, the most fit individuals are mated. As the individuals become more fit, the program enters a hill-climbing phase where, with probability 1/2, the program will either continue with crossover and mutation, or will alternatively choose only mutation of the most fit individual.

Figure 3 shows the value of the fitness function for every individual evaluated for the standalone version of MPP features. The process was allowed to continue for 86 hours until the fitness seemed to have converged and further efforts would have been (more) wasteful of computational resources. From the graph, we can see that the initial generations have a wide range of MPKI values from 3.9 to 4.5, converging over time to 3.64. Somewhat surprisingly, the average random initial individual yields a reasonable MPKI of 4.1, showing that the MPP algorithm does reasonably well even with noisy random features. The evolution of the features for the combined MPP + TAGE-SC-L features (not illustrated) was similar.

B. Evolved Features

Tables I and VI-B show two sets of features evolved by the genetic algorithm: one for MPP only, and one for MPP+TAGE-SC-L. This discussion is intended to give an idea of which features seem to be more important, but we hesitate to dive too deeply into the meaning, since the genetic algorithm chose these features at random. Standalone MPP

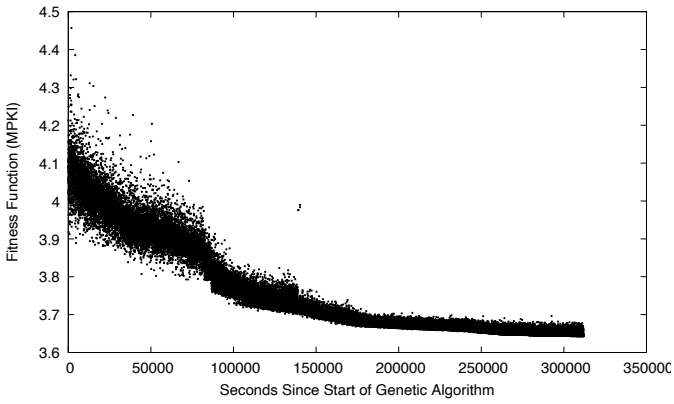

Fig. 3. Fitness Function Over Time

is dominated by 14 GHISTPATH features. Global history is has proved to be the most useful feature in previous branch predictor work, and is the only feature used by TAGE. It make sense that a standalone MPP would use this feature more than any other. It also makes significant use of several other features, especially LOCAL and GHISTMODPATH. By contrast, the features evolved for MPP+TAGE-SC-L have only 5 GHISTPATH features. These features were evolved using a combination that includes TAGE, which already incorporates many TAGE tables and a few SC tables dedicated to global history. These evolved features focus more heavily on LOCAL and GHISTMODPATH. Both sets of features use several instances of various forms of modulo history. The presence of these features suggests that these workloads benefit from mitigating the branch alignment problem.

We performed an ablation study using the training workloads to estimate the contribution of each feature. Replacing each feature with a simple BIAS feature (correlating only with the branch's own past outcomes) showed that no single MPP feature dominates prediction accuracy. In the combined MPP+TAGE-SC-L predictor, the TAGE component understandably has the largest effect, and a small group of LOCAL features contribute modestly more than the rest, but the remaining feature set shows very uniform impact. The standard deviation of MPKI increase across all 33 features is only 0.01, indicating that the GA-selected features provide balanced, complementary correlation rather than relying on any isolated trick. A similar pattern holds for standalone MPP, where LOCAL and IMLI are the strongest, but the rest of the features again show very even contribution.

VII. Other Predictor Details

This section describe other details of the MPP algorithm.

A. Filtering Trivial Branches

Many conditional branches have a strong bias of taken or not-taken [8]. Some branches are always or never taken; let us call these *trivial* branches. In modern microarchitectures, trivial branches are normally not predicted by the main conditional branch predictor, but are filtered by the branch target buffer (BTB). A branch is only inserted into the BTB if it is taken, thus necessitating a target. If the BTB is large enough, it may be used to filter trivial branches. If a branch is fetched that misses in the BTB, it may be safely predicted as not-taken, regardless of what the conditional branch predictor would have predicted. Each BTB entry contains an extra bit of metadata that is reset for a new entry and set to true when the branch is observed to not be taken. When a branch hits in the BTB, this "always taken" bit may be checked. If it is false, then the branch has only ever been taken and may be safely predicted as taken, again not consulting or updating the conditional branch predictor. Taken branches may still update branch predictor histories to provide path context, but trivial branches are not allowed to updated second level branch prediction tables (e.g. the perceptron weights or the TAGE entries). This BTB-based filtering is a standard trick in modern microarchitecture design but we are not aware of it having been documented in the peer-reviewed literature. We use this idea to filter trivial branches in our simulator, modeling 8,192 BTB entries in a 4-way set associative organization.

B. Transfer Function

Garza *et al.* explored the use of a non-linear transfer function for improving perceptron predictor accuracy [7]. The idea is to apply a function to each perceptron weight before summation, with the effect of amplifying stronger correlations. After much experimentation evolving a fitness function with a genetic algorithm, similarly to the way we found the MPP features, we found that a transfer function of the form $\frac{ax}{1-bx^2}$ gave a significant improvement over the identity function. In particular, the function $\frac{3.2x}{1-\frac{x^2}{1600}}$ worked well. The function is represented as a lookup table indexed by the weight. Figure 4 shows the transfer function. The function has been tweaked slightly to improve accuracy. We applied the same optimization to the perceptron predictor (SC) in TAGE-SC-L for the combined MPP + TAGE-SC-L predictor.

C. Making a Prediction

To make a prediction, the combiner (described later) or standalone MPP checks to see if the branch has trivial behavior and should be filtered. Branches observed to be only taken or only not taken are predicted in that direction. Otherwise, MPP is used. Each feature with its parameters are used to hash the various histories together with the branch address to yield an index into that feature's table to select a weight. The transfer function is applied to the weight and the results are summed. If the sum is at least zero, the branch is predicted taken, otherwise it is predicted not taken. Weights are 6 bits each.

D. Adaptive threshold training

MPP uses the perceptron training rule to update the weights tables in two instances: when the prediction is incorrect, or when the magnitude of the perceptron output falls below a certain threshold θ. Seznec found that good accuracy is achieved when the number of updates to due mispredictions

TABLE I
FEATURES AND PARAMETERS EVOLVED FOR STANDALONE MPP. P1..P4 ARE PARAMETERS SPECIALIZED TO EACH FEATURE. HASH SPECIFIES ONE OF 3 HASH FUNCTIONS TO COMBINE FEATURE BITS WITH THE BRANCH ADDRESS

Type	P1	P2	P3	P4	Hash	Notes
BACKPATH	2	1			0	hash the 2 recent backward branch addresses, shifting by 1
BACKPATH	49	1	0		0	hash the 49 recent backward branches, shift by 1
BLURRYPATH	11	10	1		0	hash the 10 recent 2^{11}-byte regions, shift by 1
GHISTMODPATH	2	16	5		16	hash 16 recent addresses equal to 0 mod 2 and their outcomes, shift by 5
GHISTMODPATH	5	10	6		8	hash 10 recent addresses equal to 0 modulo 5 and their outcomes, shift by 6
GHISTMODPATH	6	8	6		8	hash 8 recent addresses equal to 0 modulo 8 and their outcomes, shift by 6
GHISTMODPATH	7	15	1		8	hash 15 recent addresses equal to 0 modulo 7 and their outcomes, shift by 1
GHISTPATH	0	11	12	0	0	hash 12 recent branch addresses, not shifted, with global outcomes 0..11
GHISTPATH	0	130	2	0	0	hash 2 recent branch addresses, not shifted, with global outcomes 0..130
GHISTPATH	0	18	17	1	0	hash 16 recent branch addresses, shift by , with global outcomes 0..18
GHISTPATH	0	23	0	0	8	hash global outcomes 0..23
GHISTPATH	0	48	0	2	0	hash global outcomes 0..48
GHISTPATH	0	5	0	0	0	hash global outcomes 0..5
GHISTPATH	0	9	1	0	8	hash global outcomes 0..9
GHISTPATH	18	37	0	0	0	hash global outcomes 18..37
GHISTPATH	2	14	0	0	0	hash global outcomes 2..14
GHISTPATH	2	30	0	0	0	hash global outcomes 2..30
GHISTPATH	32	54	12	8	0	hash global outcomes 32..54 with 12 recent branch addresses, shift by 8
GHISTPATH	48	68	32	4	8	hash global outcomes 48..68 with 32 recent branch addresses shift by 4
GHISTPATH	53	169	30	0	8	hash global outcomes 53..169 with 30 recent branch addresses, no shift
GHISTPATH	64	101	0	0	0	hash global outcomes 64..101
IMLI					16	alternative innermost loop iteration counter
LOCAL	0	2			8	recent local outcomes 0..2
LOCAL	0	37			0	recent local outcomes 0.37
LOCAL	0	9			16	recent local outcomes 0..9
LOCAL	1	17			16	recent local outcomes 1..17
LOCAL	22	30			0	recent local outcomes 22..30
MODPATH	3	19	2		16	hash 19 recent branch addresses congruent to 0 modulo 3, shift by 2
MODPATH	3	29	2		8	hash 29 recent branch addresses congruent to 0 modulo 3, shift by 2
PATH	4	6			8	hash recent branch addresses 4..6
RECENCY	18	4			16	hash recency stack to a depth of 18, shift by 4
RECENCYPOS	61				16	position of this branch in a recency stack of depth 61

is roughly the same as the number due to low-confidence predictions [21]. We use his algorithm to adapt the threshold to maintain this property.

E. Updating the Predictor

When a nontrivial branch resolves, the algorithm decides whether to update the predictor. It uses the perceptron training rule as described above to decide when to train [11]. To train the predictor, each weight that was used to make the prediction is incremented if the branch was taken or decremented otherwise. Weights are incremented or decremented with saturating arithmetic. A learning rate is applied to the perceptron output for threshold setting.

F. Extra information.

Bits from the addresses and targets of other control-flow instructions (*e.g.* unconditional branches, calls, and returns) are also considered in the branch history.

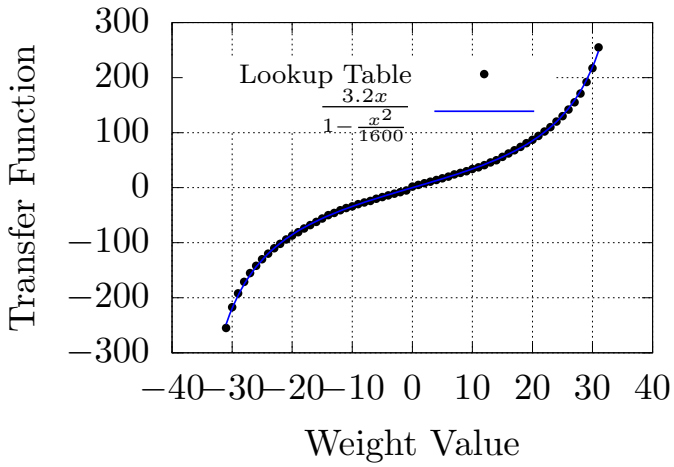

Fig. 4. Transfer Function

VIII. THE COMBINER

We combine MPP with the version of TAGE-SC-L that won CBP 2016 [23]. This predictor features the TAGE global branch predictor, a perceptron-based statistical corrector, and a loop predictor. TAGE-SC-L is finely tuned to provide very accurate predictions. We observe, as Seznec does, that the loop component contributes negligible improvement on the

TABLE II
EVOLVED FEATURES INTENDED TO COMBINE WITH TAGE-SC-L.

Type	P1	P2	P3	P4	Hash	Notes
ACYCLIC	10				0	hash an acyclic history array of size 10
ACYCLIC	9				0	hash an acyclic history array of size 9
BACKPATH	22	6	0	0	8	hash recent 22 backward branches, shift by 6
BLURRYPATH	11	9	2		8	hash the 9 recent 2^{11}-byte regions, shift by 2
GHISTMODPATH	2	19	5		16	hash 19 recent addresses equal to 0 modulo 2 and their outcomes, shift by 5
GHISTMODPATH	4	7	1		8	hash 7 recent addresses equal to 0 modulo 4 and their outcomes, shift by 1
GHISTMODPATH	4	16	6		0	hash 16 recent addresses equal to 0 modulo 4 and their outcomes, shift by 6
GHISTMODPATH	5	14	6		16	hash 14 recent addresses equal to 0 modulo 5 and their outcomes, shift by 6
GHISTMODPATH	7	8	5		8	hash 8 recent addresses equal to 0 modulo 7 and their outcomes, shift by 5
GHISTMODPATH	7	14	1		8	hash 14 recent addresses equal to 0 modulo 7 and their outcomes, shift by 1
GHISTMODPATH	7	5	2		16	hash 5 recent addresses equal to 0 modulo 7 and their outcomes, shift by 2
GHISTPATH	0	9	3	0	8	hash 3 recent branch addresses with global outcomes 0..9
GHISTPATH	1	16	0	0	16	hash global outcomes 1..16
GHISTPATH	1	22	6	8	16	hash 6 recent branch addresses, shift left by 8, with global outcomes 1..22
GHISTPATH	22	33	6	8	16	hash 6 recent branch addresses, shift left by 8, with global outcomes 22..33
GHISTPATH	29	41	8	6	0	hash 8 recent branch addresses, shift left by 6, with global outcomes 29..41
IMLI					16	alternative innermost loop iteration counter
LOCAL	0	1				hash local outcomes 0..1
LOCAL	0	13			0	hash local outcomes 0..13
LOCAL	0	20			8	hash local outcomes 0..20
LOCAL	0	5			8	hash local outcomes 0..5
LOCAL	0	9			0	hash local outcomes 0..9
LOCAL	10	32			0	hash local outcomes 10..32
LOCAL	23	27			16	hash local outcomes 23..27
LOCAL	3	34			8	hash local outcomes 3..34
MODHIST	5	22			0	hash 22 recent branch outcomes with addresses congruent to 0 modulo 5
MODHIST	7	17			8	hash 16 recent branch outcomes with addresses congruent to 0 modulo 7
MODPATH	3	20	1		8	hash 20 recent branch addresses congruent to 0 modulo 3, shift by 1
MODPATH	3	26	3		8	hash 26 recent branch addresses congruent to 0 modulo 3, shift by 3
MODPATH	5	9	4		8	hash 9 recent branch addresses congruent to 0 modulo 5, shift by 4
RECENCY	10	1			8	hash recency stack of recent branch address do a depth of 10 shift by 1
RECENCYPOS	56				0	position of this branch in a recency stack of depth 56
TAGE	11	9	0	0	8	use the TAGE-SC-L prediction at bit 11 and copy the 2-bit TAGE confidence to bits 9 and 10

TABLE III
FEATURES EVOLVED FOR COMBINED MPP + TAGE-SC-L

CBP 2025 workloads, but we retain it in our combined MPP+TAGE-SC-L to avoid changing the CBP 2016 TAGE-SC-L code.

A. Why Combine?

TAGE and perceptron-based predictors each have their strengths and weaknesses. When Seznec proposed TAGE-SC-L, he recognized that combining them resulted in a predictor much more accurate than either could be separately. Describing the similarities, strengths and weaknesses of each could fill an entire doctoral dissertation, but the discussion can be summarized as follows.

Both predictors hash control-flow features to give indices into several tables. In this way, both predictors can mitigate the impact of destructive aliasing that plagued previous predictor designs. Unfortunate collisions in hashing can lead to interference for a given table, but with multiple tables interference is less likely to affect a prediction. This is the intuition behind skewed-associative caches and the subsequent skewed-associative branch predictors [20], [25], that re-emerged with the design of hashed perceptron predictors and TAGE.

The predictors differ in the way they aggregate the information from the multiple tables. TAGE selects the table with the longest matching history, on the theory that a more specific history is more likely to be accurate. Thus, TAGE is very space-efficient: each prediction relies on a very small number (1 or 2) of TAGE table entries. The hashed perceptron predictor aggregates the weights read out from each table by summation. Perceptron is less space efficient, with each prediction requiring entries read from each of the multiple tables. However, TAGE is limited by the longest-matching criteria. The histories must be global and they must be arranged in a total ordering so that "longest match" makes sense. Perceptron predictors have no such limitation, and thus may consider and sort of feature. Each feature makes an independent contribution to the final prediction, allowing perceptron to potentially reach accuracy beyond that attainable by TAGE. It was this limitation that motivated Seznec to add

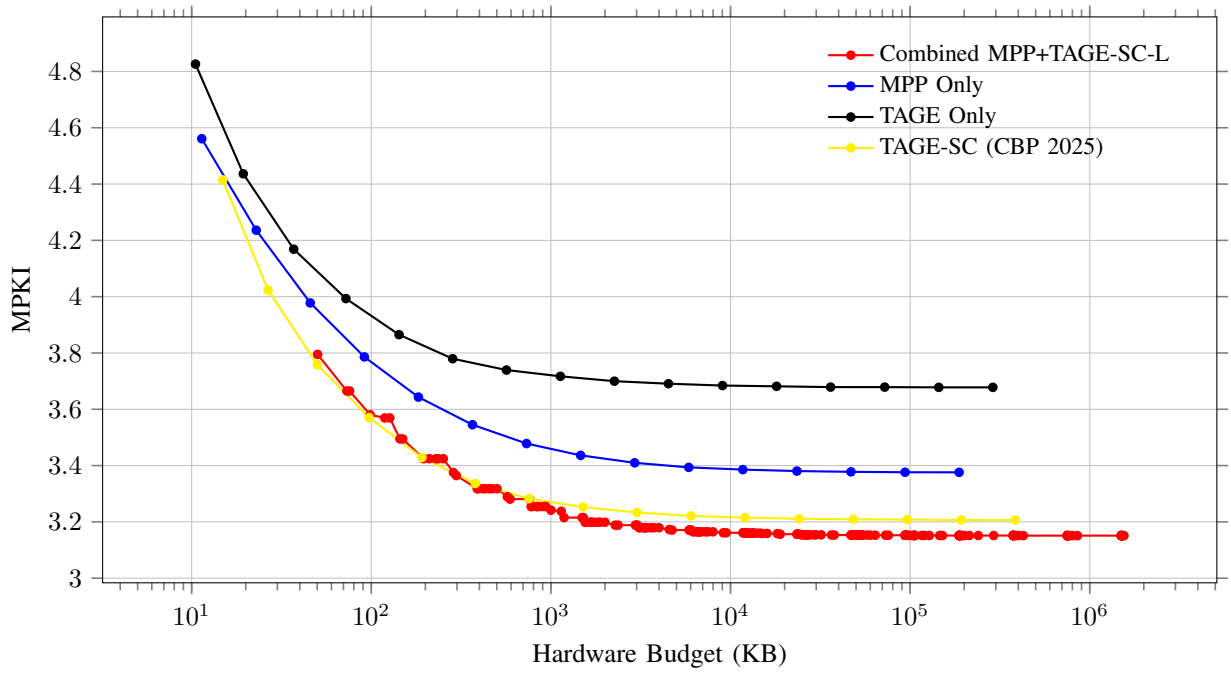

Fig. 5. Branch predictor MPKI vs. hardware budget on a logarithmic x-axis.

the perceptron-based statistical corrector to TAGE, resulting in TAGE-SC-L.

Figure 5 illustrates the limitations of TAGE and perceptron. Using the simulation infrastructure described earlier, we explore the accuracy of TAGE, a standalone MPP, TAGE-SC, and the combined MPP-TAGE-SC-L at different hardware budgets. The x-axis gives the total hardware budget in kilobytes on a log scale, and the y-axis shows the average MPKI on the 105 training traces. The TAGE and TAGE-SC simulations are taken from Seznec's code for the CBP 2025 workshop [19]. Seznec's code provided a parameter, `LOGSCALE`, that can be varied to result in predictors with different hardware budgets. We augment the MPP and MPP+TAGE-SC-L code with the same scaling idea. The predictors graphed are impossibly large, but illustrate the limits of the algorithms outside the influence of destructive aliasing.

The graph shows that each predictor becomes more accurate with a larger hardware budget. TAGE reaches a plateau of 3.67 MPKI around 10,000 KB. MPP continues to improve, leveling off at around 23,000 KB with 3.38 MPKI. TAGE-SC improves to 3.21 MPKI up to the maximum hardware budget test of 385,000 KB, beyond which limitations in the code caused it to crash. MPP+TAGE-SC-L reaches 3.15 MPKI around 100,000 KB.

These results show that, while their potential accuracies are limited when considered separately, TAGE and MPP can synergistically combine to produce very good accuracy.

B. Combining MPP with TAGE-SC-L

The main idea of the combiner is to compute a linear combination of the perceptron confidence and the SC confidence. The slope and bias for the linear combination is tuned individually for each combination of: the TAGE-SC-L prediction (0 or 1), the MPP prediction (0 or 1), the TAGE(only) prediction (0 or 1), and the TAGE confidence(0, 1, or 2). There are $2 \times 2 \times 2 \times 3 = 24$ possible combinations. We tuned most of them individually but some of them come up very infrequently and were not worth the computation so we tuned their slope and bias together.

Another bias is added to the sum, then thresholded to make a prediction. This bias is determined to minimize the recent number of misses. It is trained per combination so we keep track of 24 different biases ranging over 64 possible values of the bias. For each combination, the combiner keeps track of the number of misses each of the 64 possible bias values would have produced by counting up each time a value would have resulted in a miss and then decaying all the counters when one of them saturates at 7. Each counter requires 3 bits.

IX. Results

This section presents accurate results of 192KB versions of standalone MPP, combined MPP+TAGE-SC-L, standalone TAGE, and TAGE-SC over the 673 evaluation workloads provided for CBP 2025. Note that MPP+TAGE-SC-L and TAGE-SC were developed using the 105 training traces. Neither predictor was tuned for the evaluation traces, so, to the extent that the CBP 2025 organizers did a good job of selecting independent evaluation workloads, there should be no overfitting effect.

Figure VIII-B shows the arithmetic mean MPKI achieved for the 5 workload categories as well as the overall arithmetic mean for each of the four predictors. MPP+TAGE-SC-L

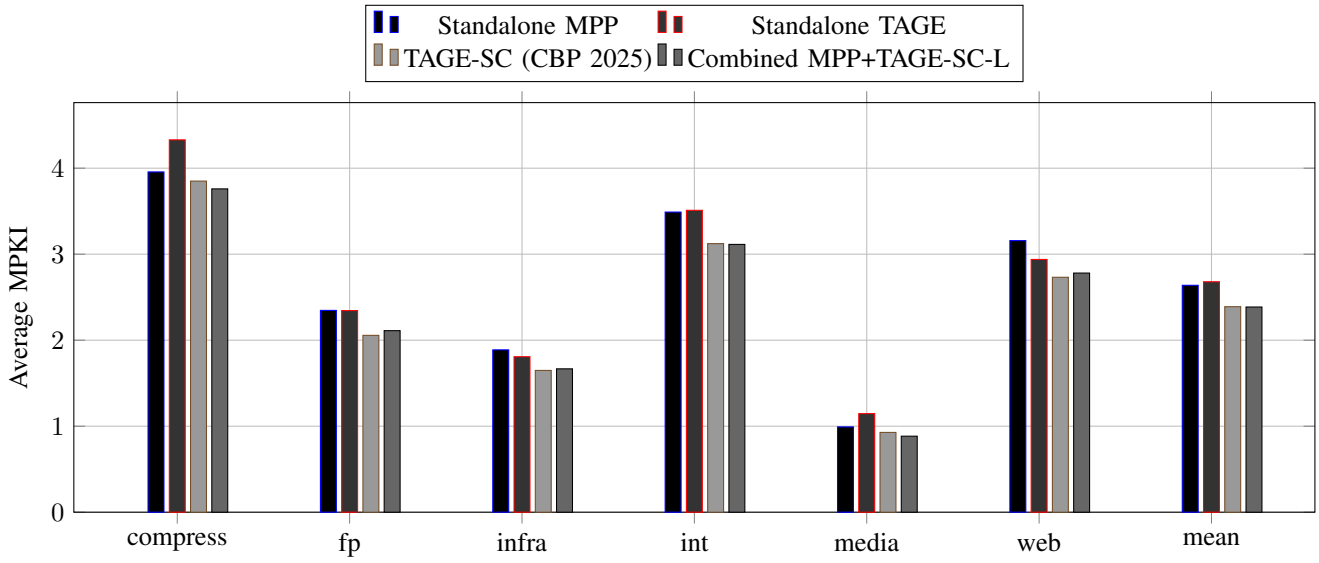

Fig. 6. MPKI for CBP2025 Evaluation Traces

performs competitively with TAGE-SC. It has slightly lower average MPKI for compress and int, and slightly higher MPKI for fp, infra, and web. For media, MPP+TAGE-SC-L achieves 0.884 MPKI, 4.5% lower than TAGE-SC at 0.927 MPKI. On average, MPP+TAGE-SC-L yields 2.385 MPKI, very similar to TAGE-SC with 2.390 MPKI.

Both predictors significantly outperform the standalone versions of MPP and TAGE. By itself, TAGE gets 2.680 MPKI while standalone MPP gets 2.638 MPKI. Clearly, perceptron and TAGE-based predictors should be used together to achieve lower misprediction rates.

X. Cost Analysis

The two predictors evaluated, MPP and MPP+TAGE-SC-L, use a hardware budget of 192KB of mutable state, allowing them to be compared with the version of TAGE-SC presented in CBP 2025.

Table IV details the contribution to the cost for each structure in the combined MPP+TAGE-SC-L predictor. The genetic algorithm evolved a predictor with 33 tables. To fit the 192KB hardware budget, we allocated 27 tables with 4,096 entries and 6 tables with 8,192 entries. Together with the other structures, this left about 2KB unused. There are a few small counters and other few-bit items unaccounted in the table that have negligible impact on the budget. The accounting for the standalone MPP predictor is similar, with the main differences that there are 32 tables, and there is no TAGE-SC-L or miss counters, so the additional storage is allocate to additional capacity for the weights tables.

TABLE IV
Hardware budget for MPP+TAGE-SC-L

Structure	**Cost (bits)**
TAGE-SC-L	524,288 (64KB)
Tables, first 27 features	$27 \times 4{,}096 \times 6 = 663{,}552$
Tables, last 6 features	$6 \times 8{,}192 \times 6 = 294{,}912$
"Always Taken" bit per BTB entry	$8,192$
Combiner miss counters	$24 \times 64 \times 3 = 4,608$
Local histories	$1280 \times 35 = 44,800$
Other histories (global, etc.)	$3,323$
Total	1,543,675 = 188.44 KB

XI. Conclusion

This paper introduces the multi-perspective perceptron predictor, and shows how it can be cominbed with a TAGE-base predictor to deliver state-of-the-art accuracy. The main contributions are the set of novel features and the genetic-algorithm-based design methodology that allows the designer to automate the feature selection process, rather than hand-tune each feature. This paper shows the potential of prediction algorithms, leaving to designers to decide how or which components of the algorithms to include in their designs. In follow-up work, we plan to investigate more thoroughly the value of each features, and find the particular code behaviors that lead to one or another feature leading to a more accurate prediction. We also plan to investigate practical implementation concerns such as timing and energy for MPP.

References

[1] *The 6th Championship Branch Prediction (CBP 2025), https://ericrotenberg.wordpress.ncsu.edu/cbp2025/.*

[2] N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky, and A. Saporito, "The ibm z15 high frequency mainframe branch predictor industrial product," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 27–39.

[3] H. Akkary, S. T. Srinivasan, R. Koltur, Y. Patil, and W. Refaai, "Perceptron-based branch confidence estimation," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04. USA: IEEE Computer Society, 2004, p. 265. [Online]. Available: https://doi.org/10.1109/HPCA.2004.10002

[4] J. Albericio, J. S. Miguel, N. E. Jerger, and A. Moshovos, "Wormhole: Wisely predicting multidimensional branches," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*,

ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 509–520. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.40

[5] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 1–13.

[6] J. Emer and N. Gloy, "A language for describing predictors and its application to automatic synthesis," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997, pp. 304–314.

[7] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jiménez, "Bit-level perceptron prediction for indirect branches," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 27–38.

[8] D. Gope and M. H. Lipasti, "Bias-free branch predictor," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 521–532. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.32

[9] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 40–51.

[10] D. A. Jiménez, "Fast path-based neural branch prediction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*. IEEE Computer Society, December 2003, pp. 243–252.

[11] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), Nuevo Leone, Mexico, January 20-24, 2001*. IEEE Computer Society, 2001, pp. 197–206. [Online]. Available: https://doi.org/10.1109/HPCA.2001.903263

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

[13] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 436–448. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123942

[14] D. A. Jiménez, "Piecewise linear branch prediction," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pp. 382–393.

[15] T. Koizumi, T. Maekawa, M. Mizuno, M. Kuroki, T. Tsumura, and R. Shioya.

[16] G. H. Loh and D. A. Jiménez, "Reducing the power and complexity of path-based neural branch prediction," in *Proceedings of the 2005 Workshop on Complexity-Effective Design (WCED'05)*, June 2005, pp. 28–35.

[17] S. McFarling, "Combining branch predictors," Digital Western Research Laboratory, Tech. Rep. TN-36m, June 1993.

[18] P. Michaud, "An alternative tage-like conditional branch predictor," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, August 2018. [Online]. Available: https://doi.org/10.1145/3226098

[19] A. Seznec.

[20] ——, "A case for two-way skewed-associative caches," in *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, New York, NY, USA, 1993, pp. 169–178.

[21] ——, "Genesis of the o-gehl branch predictor," *Journal of Instruction-Level Parallelism (JILP)*, vol. 7, April 2005.

[22] ——, "Tage-sc-l branch predictors," in *JILP-Championship Branch Prediction*, 2014.

[23] ——, "TAGE-SC-L Branch Predictors Again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Seoul, South Korea, June 2016.

[24] ——, "Tage-sc-l branch predictors again," in *Proceedings of the Fifth JILP Championship Branch Predictor Competition (CBP-5)*, June 2016.

[25] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the Alpha EV8 conditional branch predictor," in *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[26] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *J. Instr. Level Parallelism*, vol. 8, 2006.

[27] A. Seznec, J. S. Miguel, and J. Albericio, "The inner most loop iteration counter: A new dimension in branch history," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 347–357. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830831

[28] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja, "Sparc t4: A dynamically threaded server-on-a-chip," *IEEE Micro*, vol. 32, no. 2, pp. 8–19, 2012.

[29] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981, pp. 135–148.

[30] N. Soundararajan, S. Gupta, R. Natarajan, J. Stark, R. Pal, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney, "Towards the adoption of local branch predictors in modern out-of-order superscalar processors," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-52. New York, NY, USA: Association for Computing Machinery, 2019, p. 519–530. [Online]. Available: https://doi.org/10.1145/3352460.3358315

[31] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 3, pp. 280–300, September 2005. [Online]. Available: http://doi.acm.org/10.1145/1089008.1089011

[32] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 2:1–2:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195641

[33] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Half&half: Demystifying intel's directional branch predictors for fast, secure partitioned execution," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1220–1237.

[34] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture*, November 1991, pp. 51–61.

[35] ——, "Alternative implementations of two-level adaptive branch prediction," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, p. 124–134, April 1992. [Online]. Available: https://doi.org/10.1145/146628.139709