PIP: An Ensemble of Programming-Idiom Predictors

Karl H. Mose[†], Alexandra W. Chadwick[‡], Márton Erdős[‡], Jiayi Nie[†],
Rika Antonova[‡], Timothy M. Jones[‡], Robert D. Mullins[‡]
University of Cambridge
[†]{km781, jn517}@cam.ac.uk [‡]{firstname.lastname}@cl.cam.ac.uk

Abstract

We present PIP, an ensemble of branch predictors specialised for different patterns. We utilise a series of small but highly targeted predictors, each recognising a programming idiom with data-dependent structure. We call each of these an idiom tracker. When these idiom trackers are unable to make a prediction, we turn instead to a base predictor consisting of a tuned TAGE-SC-L and a specialised Multiperspective Perceptron predictor. The Perceptron predictor is used only for predicting branches that TAGE struggles with, and choosing between the two is handled by a novel arbiter algorithm.

Our predictor achieves an average of 3.47 mispredictions per kilo instructions (MPKI) and an average of 148 cycles on the wrong path per kilo instructions (cycWPKI) on the 6$^{\text{th}}$ Championship Branch Prediction (CBP2025) training traces, given a 192KiB budget. This is a decrease of 7.4% in MPKI and 3.0% in cycWPKI compared to the 2016 CBP winner, TAGE-SC-L 64KiB.

1 Introduction

Branch predictors traditionally leverage different kinds of branch and path history for prediction. However, they miss opportunities to improve accuracy by being unaware of the state of the register file, or the deeper semantics of code. Furthermore, there are many different ways of organising history, and managing all of these in a space-efficient manner can be challenging.

Towards solving these issues, we propose PIP. It uses an ensemble of idiom-tracking predictors that capture common programming idioms, combined with two history-based predictors for all other code, and a novel arbiter to choose between the two. The idiom trackers are a collection of specialised predictors that recognise and predict common programming idioms by observing the branch history, operations executed, and register-file state. These can capture branching behaviour that traditional history-based predictors are unable to detect.

When the idiom trackers are unable to make a prediction, we fall back to a combination of known-good designs: TAGE-SC-L, the winner of the 2016 Championship Branch Prediction Competition [15], and Multiperspective Perceptron, the runner-up [7]. Both achieve very high accuracy, but diverge somewhat in the features that they use. TAGE is highly space efficient, and while the GEHL predictors in TAGE's statistical corrector provide some exposure to additional features such as local history, Perceptron is able to balance the output of a wide range of features. As an example, the 64KiB configuration from the 2016 CBP competition using 37 different features. We use a novel strategy to decide between using TAGE or Perceptron for a given prediction, with an algorithm that dynamically learns when each one performs the best.

'CBP 2025', June 21, 2025, Tokyo, Japan

2 Related Work

Our history-based predictor uses a combination of TAGE-SC-L [15] and the Multiperspective Perceptron [7]. Previous works have combined TAGE and Perceptron [8], and we have continued down this track with a more advanced choosing mechanism.

Data-dependent branches have been commonly identified as a central problem to reducing MPKI beyond history-based predictors [5, 12]. Loops have been identified as particularly important for branch predictors, motivating designs such as dedicated loop predictors [14]. Runahead-style branch prediction [5, 12] is capable of using register data to predict loop outcomes, but often requires substantial resources.

Heil et al. [6] introduced the Branch Difference Predictor, which computes the difference between registers in comparison-based branches and feeds this, along with other runtime information, into the main branch predictor to identify correlations. Given that modern predictors are highly accurate for most branches but also sensitive to noise, we see value in using a separate and specialised set of register-based predictors, rather than inserting their data into the main branch predictor.

Otoom et al. [1] introduced EXACT predictors. EXACT separates branches by the addresses of their load dependencies to deal with load-dependent branches, a frequent source of misprediction. A chooser is employed since some branches benefit from this strategy while others do not. The idea is that discerning between branches based on their load addresses and values allows the main predictor to discover patterns that would otherwise be hidden. While this is potentially very general, we instead try to identify multiple common patterns and create specialised predictors for each.

The concept of classifying instructions and then applying an appropriate predictor from an ensemble of simple predictors has previously been used for data prefetching [11]. We borrow and extend this concept to branch prediction via our idiom trackers.

3 High-Level Design Overview

PIP incorporates four main components: (1) the ensemble of idiom trackers, (2) a TAGE-SC-L predictor, (3) a Multiperspective Perceptron Predictor (MPP), and (4) an Arbiter to choose select the latter two. The TAGE-SC-L, Perceptron and Arbiter together constitute the default predictor, which will provide predictions whenever the idiom trackers do not detect a pattern.

PIP includes two idiom trackers:

A `for`-loop tracker aims to recognise simple loops of the form
`for (int i = B; i != E; i += S)`
for any B, E, S.

A null-terminated string tracker (NTS) aims to recognise loops that iterate over a null-terminated string, i.e.
`for (char *p = S; *p != '\0'; ++p)`

It remembers the lengths of such strings to predict the bound in future loops that iterate over the same strings.

The predictions flow as follows:

(1) When a branch is encountered, the idiom tracker first decides if it wants to provide a prediction.
(2) If not, the combined prediction from TAGE and MPP is used.
(3) If they disagree, a final decision is made by the Arbiter. The Arbiter looks into a threshold table, and uses this along with the TAGE confidence to decide a winner.

Updating is handled as follows:

- Confidence counters track whether or not the idiom trackers are producing useful information.
- TAGE-SC-L is updated as usual.
- MPP is updated as usual, with the caveat that if it disagrees with TAGE, it uses a more aggressive update strategy for its weights.
- On a misprediction, the Arbiter either raises or lowers its threshold counters depending on the magnitude of the MPP and the confidence level of TAGE.

4 Predictor Operation

We describe our idiom trackers in detail, then the configurations of TAGE, MPP and the Arbiter.

4.1 An Ensemble of Idiom Trackers

Each idiom tracker observes decode and execute information from the core pipeline to detect programming idioms in an idiom-specific manner. If found, an entry is allocated in an idiom-tracking table. By default, tracked idioms in this table will be used to provide the prediction, falling back to the history-based predictors (TAGE + Perceptron) if the tracked idiom is unable to predict for any reason. Confidence counters track how well predictions from each entry are performing, and the entry is disabled if confidence drops too low. We also track whether or not this entry is beating the history-based predictors, and lock it to enabled or disabled if it is significantly over- or under-performing that predictor. If the table is full, a replacement policy is used to evict the 'least useful' entry, which is determined by a metric combining recency and usefulness.

4.1.1 The `for`*-loop tracker.* The `for`-loop tracker aims to recognise simple loops with an integer start, stride and end:

`for (int i =` B`; i !=` E`; i +=` S`)`

for any B, E, S. We observe that, in machine-code form, such a loop is typically translated to include some sort of two-argument arithmetic instruction that operates on `i` (the iteration variable) and E (the end bound of the loop), whose result is observed by a conditional branch. The `for`-loop tracker tracks any such arithmetic instructions to see if the difference between two operands is monotonically decreasing. If it observes that difference decreasing by a constant amount S in three successive executions (i.e. the stride), the tracker records this as a `for`-loop.

Once tracked, the `for`-loop tracker will predict the branch to be taken or untaken based on the *projected difference*. The execution units in a modern core run far behind the fetch stream, meaning that it is possible that instruction fetch is several loop iterations ahead in the `for` loop compared to instructions being executed. Hence, the tracker takes the latest-seen difference D of the arithmetic instruction, and subtracts S multiplied by the number of in-flight instances of that instruction between that point and the fetch address. If this projected difference value is zero (or negative), the tracker predicts the branch to go in the exit direction, otherwise it predicts the loop direction. The loop versus exit direction is learned by observing the first three executions of the branch.

Notably, this tracker has the remarkable property of being able to achieve 100% accuracy on some loop branches. It never needs to observe a change in direction of the branch before predicting the exit for the first time, and so in some traces we see loop branches with no mispredictions at all when this tracker is active. This tracker is particularly effective when a given loop is executed many times with different bounds E. Conventional history-based predictors may struggle to understand the pattern of the loop bounds, whereas the `for`-loop tracker simply observes the bound.

If the `for`-loop contains a `break` statement, it may exit early. This can be a problem for the `for`-loop tracker, as it cannot detect this and may mispredict the next invocation of the loop as being the same invocation. On the other hand, as soon as the arithmetic instruction's operands are available, the tracker can be updated to become aware of the new bound. Hence, `break` statements only typically cause mispredictions in very short loops.

One caveat with the experimental setup of the CBP2025 simulator is that the tracker is not able to see the opcodes of the branches or arithmetic operations. Consequently, the predictor has to assume that the loop uses a condition such as `i !=` E or `i <` E. This is the common case in practice, but some loops use `i <=` E, which results in the `for`-loop tracker mispredicting the final iteration. For this reason, our implementation of the tracker can switch to predicting the loop exit only if the projected difference is a negative number (i.e. instead of it being zero or less). This switch is made if the tracker is off-by-one in the first exit prediction. If this tracker were to be deployed in a real system, observing the opcode of the branch and comparison would obviate the need for such off-by-one errors.

In the CBP2025 sample traces, the `for`-loop tracker outperforms the TAGE+MPP+Arbiter predictor on more than one thousand `for`-loop branches.

4.1.2 The null-terminated string tracker. The null-terminated string (NTS) tracker aims to recognise loops that iterate over an NTS:

`for (char *p =` S`; *p != '\0'; ++p)`

It remembers the lengths of such strings to predict the bound in future loops. We observe that, in machine-code form, such a loop is typically translated to have a memory load operation that either feeds directly into a 'compare-and-branch' instruction, or first into a comparison against an immediate, and then a conditional branch. The NTS tracker tracks any such load operations to see if the source address is incrementing by a character width. It records the pattern as a null-terminated string loop if the load is observed as incrementing by a character width in three successive executions.

Once tracked, the NTS tracker records as a string any memory addresses that are accessed by an NTS-loop load operation. When it observes the load returning a value of zero, it can record the length of that string in a tracking table.

Whenever an NTS-loop is iterating over a string of known length, the NTS tracker will predict that the branch goes in the loop direction until the null-terminator, and then goes in the exit direction. The directions are learned by observing the first three executions of the branch.

Notably, any NTS-loop can benefit from a string length learned from any other NTS-loop, and so it is possible that the NTS tracker can predict a given loop branch 100% accurately if all the strings it operates on are known.

Unfortunately, some NTS manipulation code is vectorised by either the compiler or standard-library functions, meaning that the NTS tracker will not detect it as an NTS loop. As mentioned previously, the CBP2025 simulator does not provide access to instruction opcodes. With a greater visibility of the them it may nevertheless be possible to detect NTS manipulation code that is vectorised.

4.2 TAGE-SC-L

We employ a fine tuned version of the 2016 version of TAGE-SC-L, that was provided with the CBP2025 simulator. The final version included the following changes:

- We allocate a full 16KiB of space just for the bimodal predictor.
- We increase the counter width in the Statistical Corrector (SC) from 6 to 7.
- We increase the size of the bias tables in the SC from 256 entries to 2,048
- We increase the number of history-tables from 36 to 48.
- We track up to 4,900 bits of global history.
- Other, smaller changes were made to counter sizes, tag sizes, etc.

4.3 Multiperspective Hashed Perceptron

To complement TAGE-SC-L, we've built a custom predictor based on the Multiperspective Perceptron Predictor [7]. TAGE will provide most of the predictions, and this predictor will be used solely when TAGE is underperforming. Because of this, we use a different set of history functions than the original paper, which was intended to be a stand-alone predictor. Notably, we weigh all weight-tables equally.

The training algorithm has also been modified to increase learning pressure when the Perceptron disagrees with TAGE. Similar to O-GEHL predictors [13], we employ a dynamic threshold algorithm for training. We only train the predictor if it is correct and the total magnitude of its prediction is below this threshold, or if it is incorrect. We modify this strategy so that if TAGE is wrong, and the Perceptron is right, it is trained as long as its total magnitude is less than double the current threshold.

The standard MPP also has a strategy to update a few more weights randomly if, after an update, its prediction is still incorrect. We use this same strategy, but use a different number of updates depending on whether or not the MPP agreed with TAGE or not.

We employ a subset of the features from the original paper, and one new feature:

- MOD-LOCAL, a local-history table where branches congruent to modulo 0 with some constant are filtered. Since branches are also used to index into the local-history tables, we use the upper bits for indexing, and the lower bits for congruency testing.
- TAGE-HIST, a history vector of TAGE-SC-L's predictions.

4.4 TAGE-Perceptron Arbiter

When TAGE-SC-L and the MPP disagree, we choose between them using an Arbiter. The Arbiter consists of two tables of thresholds, med-conf and low-conf. If TAGE reports high-confidence (a saturated counter when hitting a tagged table), the Arbiter always selects TAGE. If not, the Arbiter looks up into the med-conf or low-conf table depending on TAGE's reported confidence level. The threshold in the table is compared with the magnitude of the MPP prediction—if the MPP prediction magnitude is greater than the threshold, the MPP prediction is chosen, otherwise TAGE is chosen.

For updating, the threshold entry that was used for prediction is decremented if the Perceptron was correct and TAGE was wrong. Alternatively, if the Perceptron predictor was wrong, and predicted with a magnitude M, the threshold in the entry is set to a value $M + \epsilon$, where ϵ is a small integer value that is dynamically updated depending how well the MPP is doing overall.

5 Parameter Tuning

We fine-tuned configuration parameters for TAGE-SC-L, the MPP and the Arbiter using Bayesian Optimisation (BO) through the SMAC3 software package [9]. BO is a family of global search methods for optimising non-differentiable functions (Shahriari et al. give an overview [16]). Formally, BO searches for a parameter vector $\mathbf{w}_*$ that optimises an objective function f, i.e. $f(\mathbf{w}_*) = \max_{\mathbf{w}} f(\mathbf{w})$. f is commonly modelled with a Gaussian Process that captures the predictive mean and uncertainty of f for any $\mathbf{w}$. BO chooses candidates to evaluate in a way that reduces uncertainty globally, while also coming back to sampling promising candidates with high predictive mean. BO does not make restrictive assumptions about f, supports optimisation with both discrete and continuous quantities [2–4], and can handle high-dimensional spaces [10, 18]. It has been widely used for neural-network architecture search, optimisation of robot controllers, and hyper parameter tuning for reinforcement learning methods (see the survey by Wang et al. [17] for example applications).

6 Evaluation

PIP achieves an average of 3.47 MPKI and 148 CycWpPKI, respectively a 7.4% and 3.0% decrease compared to the 64KiB configuration of TAGE-SC-L. Figure 1 shows the MPKI across various configurations: (1) 64KiB TAGE-SC-L (2) 64KiB TAGE-SC-L with Idiom Tracker (3) 128KiB TAGE-SC-L (4) 64KiB TAGE-SC-L with Perceptron and Idiom Tracker (PIP) (5) 128KiB TAGE-SC-L with Idiom Tracker (6) 160KiB TAGE-SC-L (7) 192KiB TAGE-SC-L (8) 160KiB TAGE-SC-L with Idiom Tracker.

The 160KiB and 128KiB configurations of TAGE were tuned using Bayesian optimisation. The 64KiB and 192KiB configurations were based on references in the CBP2025 framework.

We measured the effect of the idiom trackers in PIP. On average, they provided a prediction for 10.1% of all branches. For these predictions, the idiom trackers only disagreed with TAGE+MPP in around 1.1% of cases, leading to an average disagreement rate of

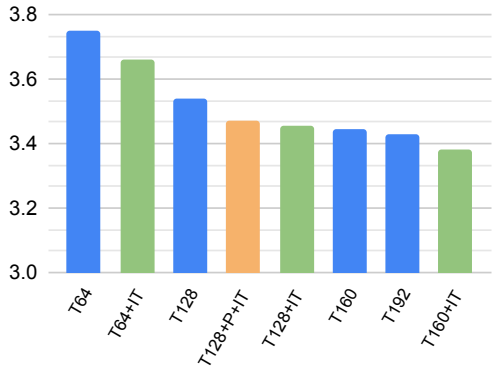

Figure 1: Comparison of average MPKI between different configurations (T=TAGE, P=Perceptron, IT=Idiom Tracker). Orange marks the submitted predictor (PIP), blue for TAGE-only configurations.

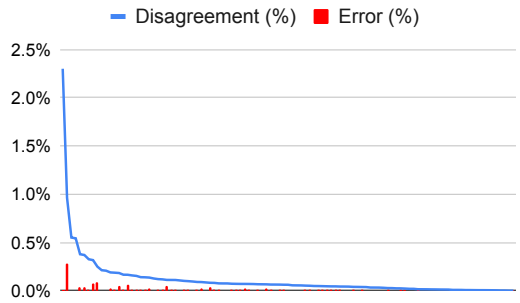

Figure 2: Percentage of predictions where the idiom trackers and the TAGE+MPP+Arbiter predictor disagree across all CBP2025 traces. 'Error' refers to disagreements where the idiom tracker's prediction is incorrect.

0.12% (see figure 2). On average, the idiom trackers were correct in 87.6% of disagreements.

Some specific benchmarks saw significant improvements, while others showed negligible impact. For example, in `fp_5`, `int_21`, and `int_0`, 2.3%, 0.68%, and 0.55% respectively of all predictions were inverted correctly due to the idiom trackers.

We found that a reasonably sized version of our Perceptron predictor often had negligible and occasionally small negative effects on performance when combined with TAGE and the idiom trackers. While PIP includes both components, in later experiments we found a 160KiB TAGE-SC-L predictor combined with the idiom trackers to be the configuration with the lowest MPKI.

After tuning the 160KiB TAGE-SC-L configuration using Bayesian optimisation we found that it exhibited negligible performance losses (approximately 0.5% MPKI) compared to the reference 192KiB configuration. Adding the idiom tracker provides a further 1.8% improvement, for a total of a 9.7% decrease in MPKI compared to the 64KiB configuration of TAGE-SC-L.

7 Discussion

The two idiom trackers employed in our design both demonstrate an ability to correctly predict loops the first time they are entered. This is commonly seen in the CBP2025 sample traces for the `for`-loop tracker. This is a novel property for a branch predictor: history-based predictors generally cannot handle the initial behaviour of a branch, and use this for training. The ability to predict loops correctly on the first invocation is particularly relevant for large code bases, where capacities of branch predictors often become problematic.

The `for`-loop idiom tracker detects and predicts branches in every single one of the CBP2025 sample traces. This suggests that the `for`-loop programming idiom is exceptionally common in practice. By contrast, the null-terminated string tracker makes little difference for many traces, suggesting it is a more niche programming idiom. Future work could extend the ensemble with other small predictors for niche idioms, which might collectively apply to a wide range of applications.

In a real design, the two idiom trackers would likely be somewhat easier to implement, as the CBP2025 infrastructure does not provide valuable information such as the opcode of decoded instructions. These opcodes would result in much better filtering resulting in fewer false positives and smaller tracking structures. Furthermore, a hardware-software co-design may be possible to benefit idiom-tracking prediction. Compilers could be tweaked to transform machine code into clearer idioms, for example by grouping the increment, compare and branch instructions to be adjacent in a `for`-loop, allowing much simpler pattern recognition.

8 Conclusion

We show that small, targeted data-dependent predictors can achieve significant MPKI reductions even when combined with very large statistical branch predictors. The combination of TAGE-SC-L and MPP provides accurate predictions for most branches. However, combining the two is not trivial and out of the configurations that we tried, a stand-alone TAGE-SC-L predictor achieved the best performance. Given that modern workloads are likely to see one or two orders of magnitude more branch-PCs compared to what previous predictors were designed for, and considering our observed MPKI improvements from scaling up existing designs, intelligently increasing predictor sizes will likely remain an important driver for future performance gains. Bayesian optimisation can effectively guide decisions about which predictor components to scale, helping make the most of constrained resources.

Acknowledgments

This work is supported in part by the Engineering and Physical Sciences Research Council (EPSRC) grant reference EP/W00576X/1 and the Advanced Research and Innovation Agency (ARIA). Additional data related to this publication is available in the repository at http://doi.org/10.17863/CAM.119036.

References

[1] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. 2010. EXACT: Explicit dynamic-branch prediction with active updates. In *Proceedings of the 7th ACM international conference on Computing frontiers*. 165–176.

[2] Maximilian Balandat, Brian Karrer, Daniel Jiang, Samuel Daulton, Ben Letham, Andrew G Wilson, and Eytan Bakshy. 2020. BoTorch: A framework for efficient Monte-Carlo Bayesian optimization. *Advances in neural information processing systems* 33 (2020), 21524–21538.
[3] Samuel Daulton, Xingchen Wan, David Eriksson, Maximilian Balandat, Michael A Osborne, and Eytan Bakshy. 2022. Bayesian optimization over discrete and mixed spaces via probabilistic reparameterization. *Advances in Neural Information Processing Systems* 35 (2022), 12760–12774.
[4] Aryan Deshwal, Syrine Belakaria, and Janardhan Rao Doppa. 2021. Bayesian optimization over hybrid spaces. In *International Conference on Machine Learning*. PMLR, 2632–2643.
[5] Saurabh Gupta, Niranjan Soundararajan, Ragavendra Natarajan, and Sreenivas Subramoney. 2020. Opportunistic early pipeline re-steering for data-dependent branches. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 305–316.
[6] Timothy H Heil, Zak Smith, and James E Smith. 1999. Improving branch predictors by correlating on data values. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 28–37.
[7] Daniel A. Jiménez. 2016. Multiperspective perceptron predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*. https://www.jilp.org/cbp2016/paper/DanielJimenez1.pdf
[8] Daniel A. Jiménez. 2016. Multiperspective Perceptron Predictor with TAGE. In *Online publication*. https://jilp.org/cbp2016/paper/DanielJimenez2.pdf
[9] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research* 23, 54 (2022), 1–9.
[10] Haitao Liu, Yew-Soon Ong, Xiaobo Shen, and Jianfei Cai. 2020. When Gaussian process meets big data: A review of scalable GPs. *IEEE transactions on neural networks and learning systems* 31, 11 (2020), 4405–4423.
[11] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 118–131. https://doi.org/10.1109/ISCA45697.2020.00021
[12] Stephen Pruett and Yale Patt. 2021. Branch runahead: An alternative to branch prediction for impossible to predict branches. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 804–815.
[13] André Seznec. 2005. Analysis of the o-geometric history length branch predictor. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 394–405.
[14] André Seznec. 2007. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)* 9 (2007), 1–6.
[15] André Seznec. 2016. Tage-sc-l branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*.
[16] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.
[17] Xilu Wang, Yaochu Jin, Sebastian Schmitt, and Markus Olhofer. 2023. Recent advances in Bayesian optimization. *Comput. Surveys* 55, 13s (2023), 1–36.
[18] Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando De Feitas. 2016. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research* 55 (2016), 361–387.

A Cost Analysis

We use a modified version of TAGE. It has the same configuration as the default, but it has been modified increasing total size of 128KiB

The Perceptron predictor has 10 weight tables, with 2 bits used for signs and 6 bits per weight.

We include in our costs a 'uop_tracker'. This structure provides information that is likely available 'for free' in a real pipeline; namely the operands to each executed instruction and the number of instances of a given instruction address in the pipeline. These are not provided by the CBP2025 infrastructure, so we have tracked them as part of our submission, but a real design could likely avoid this cost entirely.

Both the MPP and the Arbiter have structures to store prediction-time histories. These are not accounted for, and are highlighted in the code. For the MPP this could be a relatively expensive structure, since it will need to, among other things, store indices for each of the 10 tables.

Component	**Details of each field of each entry, # entries, *etc.***	**Cost (bits)**
Idiom tracker	uop info: 6 bits, 2,048 entries tracked idioms: 185 bits, 128 entries	35,968
`for`-loop tracker	candidates: 335 bits, 8 entries decode tracking: 79 bits	2,759
NTS tracker	candidates: 339 bits, 8 entries decode tracking: 5,135 bits known lengths: 91 bits, 128 entries load addresses: 75 bits, 64 entries	24,295
Perceptron	tables: 7 bits, 4,096 entries × 9 + 2 × 2,048 entries : 286,720 bits LOCAL HIST: 18,260 MODHIST: 3,490 bits GHIST: register: 336 bits PATHHIST: register: 1,152 bits TAGEHIST: register: 80 bits LFSR: 32 bits Addtional counters: 40 bits	310,110
TAGE-SC-L	TAGE: 943,007 bits LOOP-Predictor: 1,408 bits Statistical Corrector: 104,398 bits	1,048,813
Arbiter	med-conf table: 16 bits, 8 entries low-conf table: 16 bits, 8 entries counters: 16 bits	272
uop_tracker	per uop pc: 24 bits, 1,024 entries uop to instruction: 1 bit, 2,048 entries uop regs: 43 bits, 1,024 entries state: 11 bits phys reg file: 76 bits, 1,024 entries arch reg file: 11 bits, 128 entries	149,889
TOTAL		1,572,106 ≈ 191.9 KiB