An empirical study on performance of branch predictors with varying storage budgets

Moumita Das† Ansuman Banerjee† Bhaskar Sardar ‡
† Indian Statistical Institute, India ‡ Jadavpur University, India

***Abstract*—The objective of this paper is to examine the common branch predictor designs available in literature, and characterize their accuracy versus storage performance. As expected, many of the predictors which are known to have high accuracy in general, lose out on performance when exercised in low storage scenarios. This paper presents an empirical evaluation of different branch predictors at various storage points and the resulting effect on processor performance in terms of prediction accuracy and latency. We present our findings using the branch predictors and the traces of the Championship Branch Predictor-2 benchmarks. We believe that our study will be extremely beneficial for choosing a branch predictor design for embedded processors working in resource constrained environments.**

***Index Terms*—Branch Prediction, SRAM, Prediction Accuracy, Energy Efficiency**

I. Introduction

In modern pipelined processors, branch predictors are typically appointed to predict the direction of a branch at the fetch cycle before the condition actually gets resolved. This helps the processor continue the normal flow of a pipeline. However, it causes a considerable penalty in terms of computing cycles as well as extra instructions fetched, when a misprediction occurs, since the entire pipeline has to be flushed and new instructions need to be brought in. Embedded processors with deep pipelines face a severe difficulty because of the mis-predictions since mis-predictions have a toll on energy consumption, due to the extra overhead needed for restoring the execution to the correct path by flushing the pipeline and fetching new instructions from the alternate path when a misprediction is detected. Designing efficient branch predictors has therefore always been one of the top priority research tasks in computer architecture.

There are two broad classifications of predictors in literature, based on the stage they operate at, namely *static* predictors [9] [18] and *dynamic* predictors [5] [7]. Static predictors use program analysis [9] for creating prediction hints to be used at run time [1] [15], or learn branch directions from execution profiles [10]. However, dynamic predictors work when the program is in actual execution. Today's modern pipelined processors include efficient implementations of multiple dynamic predictors, with a reasonably high level of prediction accuracy. Another classification of predictors (specifically, the dynamic schemes) is based on the information that they use about other branches for predicting a specific branch at run-time. A *local* predictor uses history information only about the branch under consideration for its current prediction, and keeps per-branch history tables, while a *global* history-based predictor stores the direction histories of the preceding branches in a single predictor history structure, while making a prediction for a specific branch. Local predictors are difficult to implement in resource constrained embedded environments since they require more storage for maintaining individual prediction tables for each branch. Global history-based predictors, on the other hand, are not only less resource intensive, but also usually more accurate than their local counterparts, since they can exploit the correlation among branches, which the local schemes are unable to use. The challenge in these predictor designs is to come up with energy and space efficient designs for the predictor table structures, and more so, for a resource constrained environment. The necessity of striking the balance between accuracy and energy has inspired predictor designs with static, dynamic, local, global schemes and their combinations [4].

Branch predictors typically function based on the history information stored in the corresponding prediction tables. Evidently, accuracy of branch predictors is sensitive to the storage they are allowed to use, more history information usually has been seen to generate more accuracy of prediction. For a program with even a moderate number of branches, it is not possible to store the history information of all branch outcomes in the SRAM table individually, due to constraints on cost and resource size. As a result, each cell of the SRAM table has to be used for more than one branch to store history information. This phenomenon, known as interference, may manifest in either a constructive (positive) or a destructive (negative) way. When the information stored by one branch is incorrectly accessed and modified by some other branch that shares the same SRAM cell for prediction, it is called as negative interference. However, in the case of positive interference, the information stored by one branch has often been seen to help the other branch for correct prediction. Prediction accuracy is degraded largely due to the negative interference since it has been observed that negative interferences occur much more frequently than their positive counterparts [19]. Evidently, the amount of negative interference is expected to increase if the available predictor table storage is less, and more number of branch instructions access the same address for prediction, leading to performance and accuracy degradation.

The issue of storage for predictor tables is even more important for a resource constrained embedded environment with low storage budgets, since there is a more acute trade-off that needs to be worked out. On one hand, increased storage budget

978-1-5386-3032-7/17/$31.00 © 2017 IEEE

for the predictor tables, can lead to better accuracy of prediction. However, the cost and other overheads associated with SRAM structures increase. On the other hand, having a lower SRAM table for predictors reduces cost, however, increases misprediction and manifests in terms of wasted instructions, latency and energy. Selecting an appropriate branch predictor structure for an embedded environment is therefore, quite a crucial task. A branch prediction unit charges a significant amount of power consumption in modern processor designs, and consumes a significant amount of storage and becomes a major issue for relatively small embedded processors.

In this work, we empirically study the prediction accuracy of different branch predictors for different storage budgets. We also study the latency of these predictors for all the storage budgets. The motivation behind this empirical exploration is to highlight the storage sensitivity of branch predictors that are available in contemporary literature. The objective of our study is to examine the common predictor designs available in literature, and characterize their accuracy versus storage performance. As expected, many of the predictors which are known to have high accuracy in general, lose out on performance when exercised in low storage scenarios. This paper presents an empirical evaluation of different branch predictors at various storage points and the resulting effect on processor performance in terms of prediction accuracy and latency. We believe our study can help a designer select the appropriate predictor for a particular storage budget. It also helps to identify the best predictor for embedded processors for resource constrained environments. We present results of our experiments using the branch predictors and the traces of the Championship Branch Predictor-2 benchmarks [2].

II. Storage rquirement analysis of predictors

In this section, we describe the internal architecture of some popular branch predictors that are used in modern pipeline processors. As discussed earlier, the objective of a branch predictor is to predict the direction of a branch, whether it will be *taken* or *not taken* for a given input. We intend to highlight the elements of storage that are intrinsically used by these prediction units, with a discussion on why the predictor design may lose on accuracy if subjected to lower resources.

Gshare Predictor

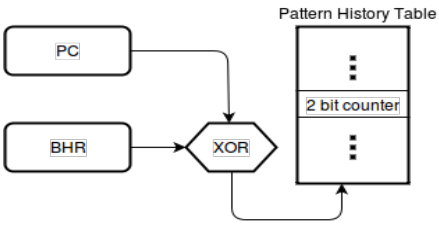


Fig. 1: GShare Branch Predictor

The GShare predictor [7] uses two main data structures - a Pattern History Table (PHT) and Branch History Register (BHR) as shown in Figure 1. BHR is a *n* bit shift register that contains the branch outcomes of the most recent *n* branches. In BHR, for a taken branch, a 1 is recorded, otherwise a 0 is recorded. For each of these BHR patterns, a corresponding PHT entry is achieved by XOR-ing the Program Counter (PC) value and the BHR content. Each PHT entry contains a two bit saturating counter and the MSB of that counter gives the final prediction. After resolving the branch condition, the states of the two bit counter and the BHR content are updated according to the actual branch outcome. For this predictor, storage size actually depends on the size of the PHT. A small PHT is able to store less number of prediction related information and uses less number of PC and BHR bits. On one hand, lesser PC bits and small sized PHT increases the negative interferences. On the other hand, BHR gives the flavor of branch correlation, lesser number of BHR bits looses the correlation to some extent. Hence, a smaller storage size increases the chances of negative interferences and the number of mis-predictions. GShare is thus quite sensitive to storage size, as evident from our experiments as well.

Local history based Two-level predictor

In dynamic branch prediction, two types of histories are mainly used - global branch history and local branch history. Global branch history contains the branch outcomes of previous branches, whereas local branch history contains the previous branch outcomes of the same branch. A local history based predictor is popularly used in many processors since it can detect the control structures like the loop structure more effectively than the global one. However, these predictors are more resource intensive. In this discussion, we highlight the main features of a popular local history based branch predictor, namely PAp [16]. It is a combination of multiple per-address BHRs with multiple per-address PHT as shown in Figure 2. In PAp, each branch has its own BHR as well as its own PHT [16]. The BHR content is used to select the index in a PHT whereas a PHT is selected by the branch instruction address. For this predictor, storage size depends on BHR size and per address PHT size. A small BHR size contains less number of entries that implies lesser number of per address PHT. This increases the chances of inter-branch interference since more number of branches now access the same PHT table. Similarly, smaller sized per address PHT can increase the chances of intra-branch interference since the same PHT entry will be searched for more number of different BHR patterns. It is quite expected that the number of mis-predictions due to negative interferences will increase with smaller storage size.

Perceptron Predictor

The perceptron predictor [6] uses a simple neural network, the perceptron instead of the two-bit counters. Figure 3 shows a diagram of a perceptron predictor internals. It learns a target Boolean function $f(x_1, x_2, ..., x_n)$ inputs that predicts whether a particular branch will be *taken* or not. Here $x_i s$ are the bits of a global branch history shift register. $x_i s$ are bipolar, -1 represents the *not taken* and 1 represents the *taken* outcome. A

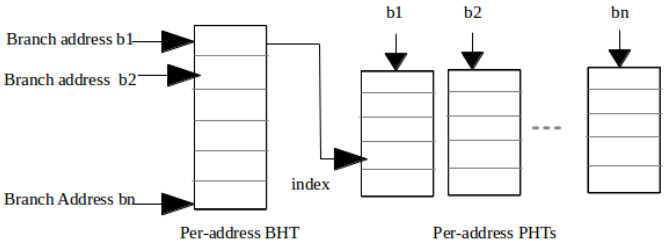


Fig. 2: PAp Branch Predictor

perceptron is represented as a vector of weights $w_0, w_1, ..., w_n$. Here, the weights are signed integers. The output is calculated as the dot product of the weight vector $w_0, w_1, ..., w_n$ and the input vector $x_0, x_1, ..., x_n$ (x_0 provides the bias input and is always set to 1). The output y of a perceptron is as below:

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

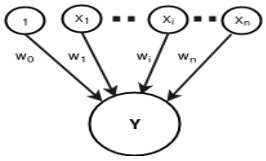


Fig. 3: Perceptron Model

A major weakness of perceptrons is their increased computational complexity when compared with two-bit counters. However, it provides better prediction accuracy compared to other popular predictors even at lower resource budget. In a perceptron predictor, the best performance can be achieved by tuning the history length, the number of bits used to represent the weights, and the threshold. A smaller predictor size affects the number of bits of these three parameters, thereby degrading the prediction accuracy in turn.

LTAGE Predictor

LTAGE [12] is widely recognized as the most popular state-of-the-art branch predictor that exploits several different history lengths to capture correlation between very remote branch outcomes as well as very recent branch history. This predictor combines a TAGE [14] predictor and a loop predictor. The default prediction is obtained from the TAGE component except when the loop predictor has a higher confidence than TAGE. The TAGE predictor is derived from the PPM like tag-based predictor [8] and uses geometric history lengths as shown in Figure 4. It has a base predictor P0, which is a simple PC-indexed 2-bit counter that provides the default prediction. The tagged predictor components Pi, $1 \leq i \leq M$ are indexed using different history lengths that form a geometric series [11]. Each entry of a tagged component contains a partial tag, an unsigned counter *u* and a signed counter *ctr* with the sign providing the prediction. At prediction time, the base predictor and the tagged components are accessed simultaneously. If no matching in the tagged component is found, the base predictor is used to provide the default prediction. The loop predictor identifies the regular loops with constant number of iterations. It provides the global prediction when the loop has successively executed three times with the same number of iterations. For this predictor, storage size depends on the size of storage for the loop predictor as well as the TAGE predictor. The size of the TAGE predictor is calculated using the size of the base predictor, the number of components and the size of each tagged component. A lesser number of tag components loses the correlation from very remote branch outcomes. A smaller size tagged component and the base predictor also increases the chances of negative interferences. Similarly, a smaller sized loop predictor may fail to identify the loop behavior of all branches correctly.

ISL-TAGE Predictor

ISL-TAGE [13] is also considered as a state-of-the-art branch predictor and is extensively used in modern pipelined processors. This predictor combines a TAGE predictor, a loop predictor, a Statistical Corrector predictor and an Immediate Update Mimicker (IUM). However, TAGE is the core of this ISL-TAGE predictor. TAGE captures most of the correlation on the branch outcomes for very long histories as discussed before. However, it fails to predict loops with constant number of iterations sometimes and so the loop predictor is included to predict these loops. Another drawback of TAGE is that it can not predict the branches that are not strongly correlated, but only statistically biased. This can be solved by including the Statistical Corrector predictor. The Immediate Update Mimicker is is also included in this predictor to avoid extra mis-predictions that occur due to delayed update at retire time by TAGE. TAGE updates at retire time to avoid pollution by the wrong path but this delayed update induces extra mispredictions compared with an optimistic fetch time update. In this case, storage size depends on all the components associated. Similar to the LTAGE predictor discussed above, smaller sized components as well as less number of components causes increased negative interference and increases misprediction.

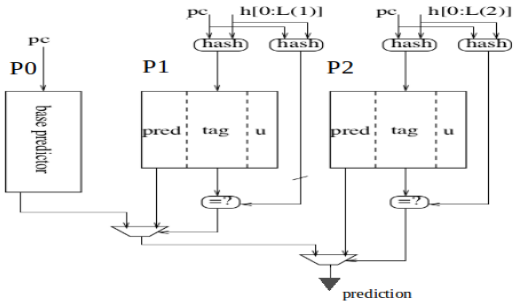


Fig. 4: Two component TAGE predictor

III. Implementation and Results

To support our study presented in the previous section, we carried out experiments with the different predictor designs available in literature, and recorded their accuracy and latency

performance at different storage points. The objective of this study was to show that indeed many of these predictors are quite sensitive to storage, and often fail to perform as per expectations when subjected to low resources. Our experiments indeed support our intuition, as detailed below.

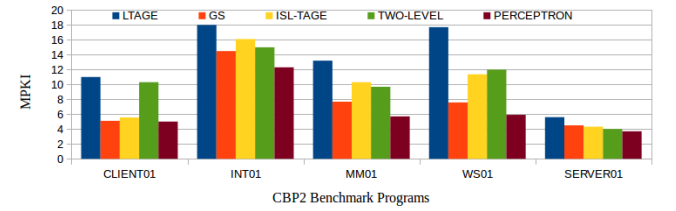
A. Experimental Setup

In this work, we use the five different predictor implementations, namely, GShare, a local history based Two-Level predictor, ISL-TAGE, LTAGE and Perceptron from the Championship Branch Prediction-2 (CBP-2) [2] benchmarks. In CBP-2, all predictors are designed for a fixed storage budget. For our experiments, we modified the predictor codes to work with 6 different storage budgets, namely, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB and 64 KB. The storage value mentioned here is the total storage allocated to a predictor for managing all data structures that it needs internally to store and manipulate history information. We performed our experiments on the CBP-2 traces and recorded the Mis-prediction Per Kilo Instructions (MPKI) and latency.

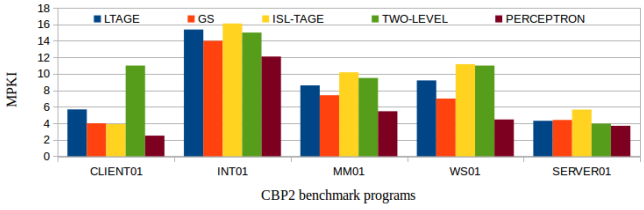
B. Results

Figure 5 presents the MPKI for the five predictors at the different storage points. Results show that the mis-prediction rates for different predictors vary with different storage budget. It is interesting to note that a predictor that has the lowest MPKI at a particular storage point can have the highest MPKI for some other storage point. For example, LTAGE has the highest MPKI for all the cbp traces with 2 KB storage size as shown in Figure 5, where as, it does not have the same for all the programs for other storage sizes presented here. Indeed, LTAGE has the lowest MPKI for almost all the cbp traces with 64 KB storage size. Additionally, it can be seen that the two state of art predictors - LTAGE and ISL-TAGE, perform well for storage sizes more than 16 KB. For a low storage budget, performances of the Two level predictor and Perceptron are better, as compared to LTAGE and ISL-TAGE. Predictors included in modern processors may not always be the best choice for the embedded processors with less storage.

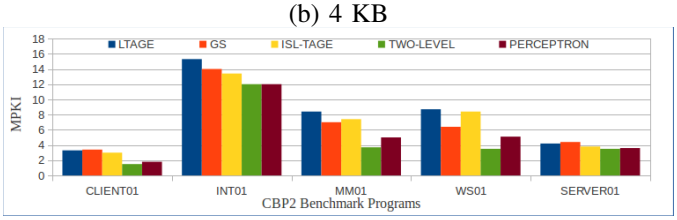
Figure 6 presents the latency for a program using different predictors for the different storage points. It is interesting to observe that the latency does not vary largely across the points. The Two-level predictor has the highest latency compared to the other predictors for all the predictor sizes considered here. From Figures 5 and 6, it is observed, though Two-level predictor gives better prediction accuracy than LTAGE and ISL-TAGE in a resource constrained environment, it is not better in terms of latency there. In this case, the perceptron predictor has the lowest MPKI as well as its latency is also comparable with the other predictors.



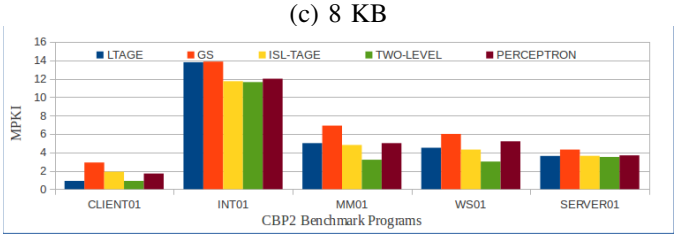
(a) 2 KB



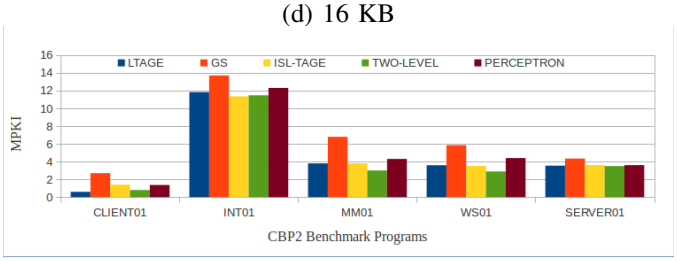
(b) 4 KB



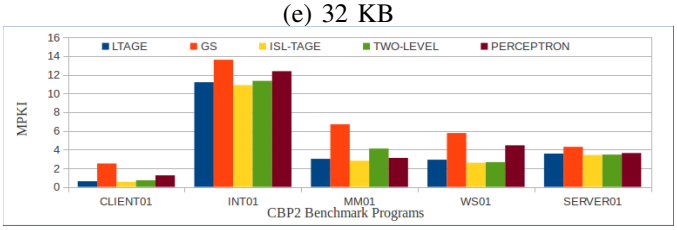
(c) 8 KB



(d) 16 KB



(e) 32 KB

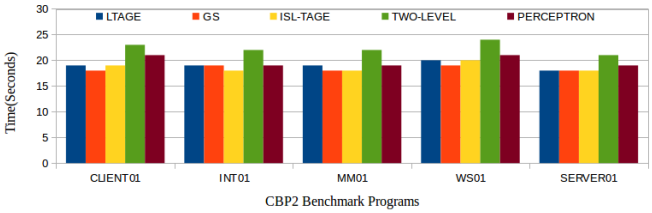


(f) 64 KB

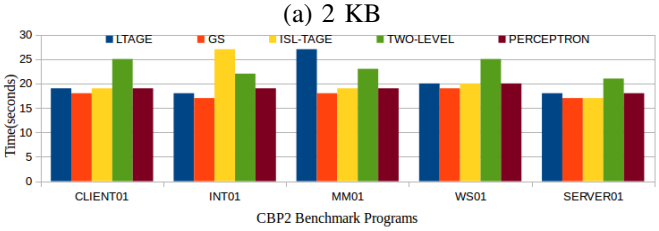
Fig. 5: MPKI for CBP2 Benchmark Programs

IV. Related Work

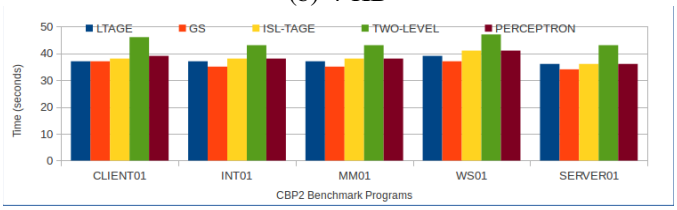
In this section, we present an overview of related research in the area of branch prediction. In [17], a comparison of different two-level dynamic predictors was studied. The authors showed the trade-off between performance and storage size. In the Championship Branch prediction programs, designers implemented several branch predictors for a fixed storage size. They measured the performance of the branch predictors in terms of prediction accuracy and latency [2] [3]. Various



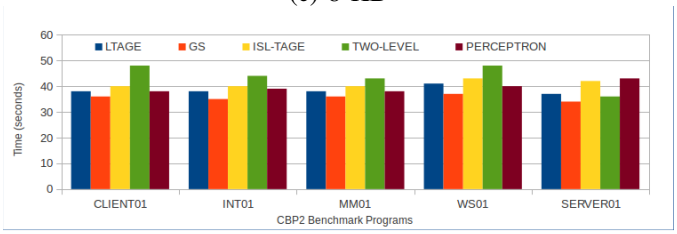
(a) 2 KB



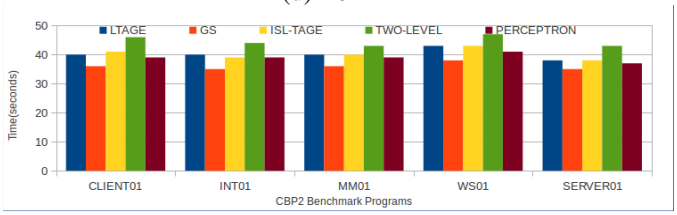
(b) 4 KB



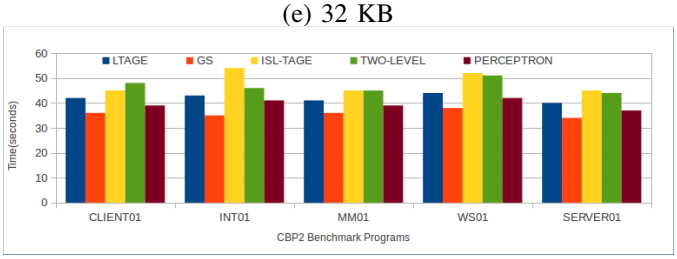
(c) 8 KB



(d) 16 KB



(e) 32 KB



(f) 64 KB

Fig. 6: Latency plot for CBP2 Benchmark Programs

predictors have been designed to improve the prediction accuracy. In these implementations, designers have predominantly attempted to minimize storage size as well as computational complexity [6] [11]. To the best of our knowledge, this is the first study that compares the performance of different branch predictors for different storage sizes.

V. Conclusion

In this paper, we empirically study the variation in prediction accuracy at different storage points. We analyze the designs of some popular state-of-the-art predictors that are used in modern pipeline processors and show that they are often not well suited in resource constrained environments. We also present our finding in term of latency for different predictors for different storage sizes. We believe that our study will help a designer select an appropriate predictor for a specific storage budget. We are currently working on extending the same analysis to multi-component predictor designs.

VI. Acknowledgement

This work was partially funded by a research grant from Defence Research and Development Organization, Government of India awarded to Indian Statistical Institute. The authors would like to thank Prof. Mainak Chaudhuri of IIT Kanpur for his suggestions on this work.

References

[1] M. Burrows. Dynamically determining instruction hint fields, Mar. 23 1999. US Patent 5,887,159.

[2] C. B. P. (CBP-2). 2nd jilp workshop on computer architecture competitions. https://www.jilp.org/jwac-2/.

[3] C. B. P. (CBP-4). 4th jilp workshop on computer architecture competitions. https://www.jilp.org/cbp2014/.

[4] Chang et al. Branch classification: a new mechanism for improving branch predictor performance. In *MICRO*, pages 22–31. ACM, 1994.

[5] Hicks et al. Towards an energy efficient branch prediction scheme using profiling, adaptive bias measurement and delay region scheduling. In *DTIS*, pages 19–24. IEEE, 2007.

[6] Jiménez et al. Dynamic branch prediction with perceptrons. In *HPCA*, pages 197–206. IEEE, 2001.

[7] S. McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[8] P. Michaud. A ppm-like, tag-based branch predictor. *Journal of Instruction Level Parallelism*, 7(1):1–10, 2005.

[9] J. R. Patterson. Accurate static branch prediction by value range propagation. 30(6):67–78, 1995.

[10] A. Ramirez et al. Branch prediction using profile data. In *Euro-Par 2001 Parallel Processing*, pages 386–394.

[11] A. Seznec. Analysis of the o-geometric history length branch predictor. In *ISCA*, pages 394–405. IEEE, 2005.

[12] A. Seznec. The l-tage branch predictor. *Journal of Instruction-Level Parallelism*, 9, 2007.

[13] A. Seznec. A 64 kbytes isl-tage branch predictor. *JWAC-2: Championship Branch Prediction*, 2011.

[14] A. Seznec. A new case for the tage branch predictor. In *MICRO*, pages 117–127. ACM, 2011.

[15] Shah et al. Method and apparatus for using static branch predictions hints with dynamically translated code traces to improve performance, Mar. 20 2001. US Patent 6,205,545.

[16] T.-Y. Yeh et al. Alternative implementations of two-level adaptive branch prediction. *ACM SIGARCH Computer Architecture News*, 20(2):124–134, 1992.

[17] T.-Y. Yeh et al. A comparison of dynamic branch predictors that use two levels of branch history. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 257–266. ACM, 1993.

[18] C. Young et al. Improving the accuracy of static branch prediction using branch correlation. In *ACM Sigplan Notices*, volume 29, pages 232–241. ACM, 1994.

[19] C. Young et al. *A comparative analysis of schemes for correlated branch prediction*, volume 23. ACM, 1995.