Received: 14 June 2017 | Revised: 23 April 2018 | Accepted: 24 April 2018
DOI: 10.1002/cpe.4666

WILEY

RESEARCH ARTICLE

A survey of techniques for dynamic branch prediction

Sparsh Mittal

Indian Institute of Technology, Hyderabad, India

Correspondence
Sparsh Mittal, IIT Hyderabad, Kandi, Sangareddy 502285, Telangana, India.
Email: sparsh0mittal@gmail.com

Funding information
Science and Engineering Research Board (SERB), India, Grant/Award Number: ECR/2017/000622; Intel Corporation

Summary

Branch predictor (BP) is an essential component in modern processors since high BP accuracy can improve performance and reduce energy by decreasing the number of instructions executed on wrong-path. However, reducing the latency and storage overhead of BP while maintaining high accuracy presents significant challenges. In this paper, we present a survey of dynamic branch prediction techniques. We classify the works based on key features to underscore their differences and similarities. We believe this paper will spark further research in this area and will be useful for computer architects, processor designers, and researchers.

KEYWORDS

dynamic branch predictor, hybrid BP, neural BP, perceptron predictorside BP, pipelining, predictor accuracy, review, side BP, speculative execution, two-level BP

1 | INTRODUCTION

Control-changing instructions such as branches add uncertainty in the execution of dependent instructions and thus lead to large performance loss in pipelined processors. To offset their overhead, accurate prediction of branch outcomes is vital. Since branch misprediction incurs high latency (eg, 14 to 25 cycles[1]) and wastes energy due to execution of instructions on wrong-path, an improvement in BP[*] accuracy can boost performance and energy efficiency significantly. For example, experiments on real processors showed that reducing the branch mispredictions by half improved the processor performance by 13%.[2]

Effective design and management of BPs, however, presents several challenges. Design of BP involves a strict tradeoff between area, energy, accuracy, and latency. An increase in BP complexity for improving accuracy may make it infeasible for implementation or offset its performance benefit. For example, a 512 KB perceptron predictor may provide lower instruction-per-cycle than its 32 KB version,[3] and a 2-cycle fully accurate BP may provide lower instruction-per-cycle than a 1-cycle partially inaccurate BP.[4] Further, due to its high access frequency, BP becomes a thermal hot-spot,[5] and hence, reducing its dynamic and leakage energy is important. Clearly, intelligent techniques are required for resolving these issues.

Further, recently, researchers have demonstrated two security vulnerabilities viz., "Meltdown" and "Spectre," which exploit speculative execution feature of modern CPUs.[6-8] For example, Spectre vulnerability works on the observation that the speculative execution due to a branch misprediction may leave side effects, which can reveal sensitive data to attackers, eg, if the memory accesses performed during speculative execution depends on sensitive data, the resulting state of the cache forms a side-channel through which the sensitive data can be leaked. These vulnerabilities affect nearly every modern CPU. While avoiding speculative execution (eg, disabling branch prediction) can mitigate these vulnerabilities, it leads to significant performance loss. In light of these vulnerabilities and associated tradeoffs, a deeper study of branch predictors is definitely required.

Contributions: In this paper, we present a survey of dynamic branch prediction techniques. Figure 1 shows the organization of this paper. Section 2 presents a brief background on BPs, discusses the challenges in managing BPs, and offers an overview of research in this area. Sections 3 and 5 discuss several BP designs for common and specific branches, and Section 6 discusses hybrid BP designs. Section 7 discusses techniques for improving BP accuracy. Section 4 discusses the neural BPs and techniques to reduce their implementation overheads. Section 8 discuses techniques for reducing latency and energy overheads of BPs. We conclude the paper in Section 9 with a mention of future challenges.

[*] Following acronyms are frequently used in this paper: basic block (BB), branch history register (BHR), branch predictor (BP), branch target buffer (BTB), global history register (GHR), (inverse) fast Fourier transform (IFFT/FFT), least/most significant bit (LSB/MSB), loop exit buffer (LEB), pattern history table (PHT), program counter (PC), return address stack (RAS), TAGE (tagged geometric history length BP).

FIGURE 1 Paper organization

Scope: To effectively summarize decades of BP research within the page limits, we had to restrict the scope of this paper in the following way. We focus on branch direction (outcome) prediction, which guesses *whether* a conditional branch will be taken or not. We do not include branch *target* prediction or the techniques for indirect or unconditional branches. Static branch prediction uses only source-code knowledge or compiler analysis to predict a branch,[9] whereas dynamic prediction accounts for time-varying and input-dependent execution pattern of a branch. Since techniques for static prediction and those involving compiler hints to assist dynamic BPs merit a separate survey, we do not include them here and focus on dynamic prediction only. We generally focus on the key idea of each paper and only include selected quantitative results. This paper is expected to be useful for computer architects, chip-designers, and researchers interested in performance optimization.

2 | BACKGROUND AND MOTIVATION

Here, we present a quick background on BP and refer the reader to prior works for more details on taxonomy,[10] value locality,[11] static BPs,[9] comparative evaluation of multiple BPs,[2,3,5,12,13] and discussion of commercial BPs.[3,14-18] We discuss the BP naming scheme[10,19] in Section 3.1.

2.1 | Useful terms and concepts

Types of branches: Branches can be classified into conditional or unconditional and direct or indirect. A direct branch specifies the target address in the instruction itself, whereas an indirect branch mentions where the target address is to be found (eg, a memory or register location).

Biased branches: Biased branches are those that resolve as either taken or not-taken nearly every time. An example of this is checking for a rare error condition. Biased branches form a large fraction of the total branches seen in real programs.[20] Biased branches are detected as those showing no change in the direction[20] or as those whose perceptron weights have reached a maximum or minimum value. [21]

Aliasing or interference: When multiple unrelated branches use the same predictor entry, they create destructive interference, which is termed as aliasing. Figure 2 presents an example of interference between branches. The interference is neutral (harmless) if it does not change the prediction, positive (beneficial) if it corrects the prediction, and negative (harmful) if it causes a misprediction. Generally, both the frequency and the magnitude of negative aliasing are higher than that of positive aliasing,[12] especially for the workloads with large working set.

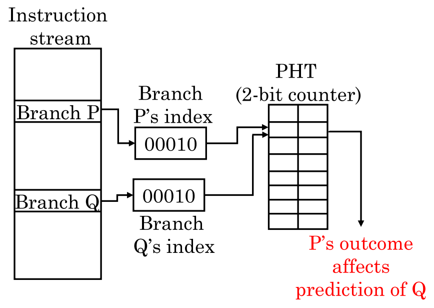


FIGURE 2 An example of interference between branches[22]

```
if(condition1) …                    //B1
if(condition2) …                    //B2
if(condition1 OR condition2) …      //B3
```

(A)

```
if(condition1) var=4;   //B4
if(var==0) …            //B5
```

(B)

```
if (condition1) …                 //B6
else if (condition2) …            //B7
else if (condition3) …            //B8
…
if (condition1 OR condition2) //B9
```

(C)

FIGURE 3 Example source-codes which lead to branch-correlation.[24] A, Directional correlation; B, Directional correlation; C, Path-based correlation

Side predictor: Commonly used BPs (eg, TAGE[23]) can accurately predict most of the branches; however, they fail to predict other relatively-infrequent classes of branches. To predict such branches, the main BP can be augmented with side (also called auxiliary or complementary or corrector) BPs for improving the overall accuracy.

Local and global history: Some branches can be predicted by their own execution pattern only and thus are said to work on "local" (or self or per-branch) history. For example, consider the loop-controlling branch in `for(i=1; i<=4; i++)`. If the loop test is performed at the end of the loop body, it shows the pattern $(1110)^n$, and thus, by knowing the outcome of the branch for last 3 instances, the next outcome can be predicted.

By comparison, if execution of previous branches can provide a hint about the outcome of a candidate branch, such branches are said to be correlated, and the BP exploiting this information is said to work on "global history." Branch correlations may arise due to multiple reasons[12,24]:

1. Two branches may be decided by related information, eg, in Figure 3A, if branch B1 is taken, branch B3 will also be taken.
2. The outcome of one branch changes the condition affecting the outcome of another branch, eg, in Figure 3B, if branch B4 is taken, then B5 will not be taken. This case and the previous case are examples of "direction correlation."
3. The information about the path through which we arrived at the current branch gives us hint about the branches before the correlated branch. For instance, in Figure 3C, although the direction of branch B8 is not correlated with that of branch B9, if branch B8 was observed on the path (ie, was one of the previous *K* branches), then outcome of branch B9 can be predicted. Such correlation is referred to as "in-path correlation." *Path-history* consists of branches seen on the path to the current branch whereas *pattern history* shows the outcome (direction) patterns of branches that led to the current branch. Compared to pattern history, use of path history allows better exploitation of correlation.[12]

In general, considering higher number of branches provides stronger correlation, eg, in Figure 3A, knowing of the outcome of B1 alone is not sufficient to predict B3, but knowing the outcome of both B1 and B2 is sufficient to predict B3. Note that several works on neural BP use perceptron weights as a measure of strength of correlation between branches.[2,25-27]

Basic block: A basic block is a maximal length sequence of branch-free code.[28] Unless an exception happens, the instructions in a BB always execute together. The control enters a BB at the first instruction and exits at the last instruction.

Metrics: As for the metrics for evaluating BPs, research works have used metrics such as performance, number of mispredicted branches (per kilo-instruction), micro-ops fetched and number of flushes per micro-ops, misprediction penalty,[29] etc.

2.2 | Challenges in managing BPs

Effective design and operation of BPs presents several challenges as we show below.

Latency constraints: The dependences that constrain the speed of a processor constitute the critical path of execution. Since BP lies on critical path, it needs to provide prediction within one cycle from the time the branch address is known. However, due to the complex designs of BPs, use of slow wires, and high clock frequency, BP access latency can exceed one cycle.[30,31] Apart from BP access latency, BP update latency also has large impact on the performance.[32]

Area and implementation overheads: Area and power considerations limit storage budget of BPs to tens of kilobytes (eg, 32-64 KB), which prohibits storing and benefiting from correlations with branches in distant past.[33] Even worse, blindly increasing BP size may provide marginal benefits, eg, increasing the size of 2bc+gskew BP[34] from 64 KB to 1 MB provides little improvement in accuracy.[1] This happens because a linear rise in global history length increases the BP size exponentially, and yet, the extra branches included may not be correlated with the branches being predicted. In such cases, increasing the BP size increases aliasing and BP training time. Furthermore, some BP designs may be strongly coupled with the architecture, eg, data value–based BPs may necessitate additional data paths. Such dependencies reduce portability and ease of implementation.

Energy overheads: Since the BP needs to be accessed in nearly every cycle and its circuitry is aggressively optimized for latency, BP has high energy consumption and becomes a thermal hot-spot.[5] As power budget becomes the primary design constraint in all ranges of computing systems,[35] improvement in energy efficiency of BP has become vital to justify its use in modern processors.

BP warm-up requirements: Large BPs may take long time to warm up, and this problem may be especially severe in the presence of frequent context-switching. The slow warm-up may even bring their accuracy lower compared to the otherwise less-accurate simple BPs.[31]

Challenges in use of tags: Unlike for cache, aliasing is acceptable in BP since a misprediction only impacts performance and not correctness. Hence, a direct-mapped BP can be tag-less. However, tags are required for implementing associativity or removing aliasing. Since each BP table entry is much smaller compared to a cache block (eg, 2 bits vs 64 bytes), for the same total capacity, the number of entries in BP becomes much larger than

TABLE 1 A classification based objective, design and optimization features

Category	References
Overall Goal	
Performance	Nearly-all
Energy	[21,38-49]
BP Designs and Related Features	
Neural-network BPs	[21,25-27,31,39,45,46,50-56]
Hybrid BPs	[1,13,18,47,51,57-61]
Side-predictor	[2,44,49,59,62-65]
Use of branch cache	[2,66,67]
Use of pipeline gating	[27,68]
Use of static BP	[13,59,67]
Use of profiling for	identifying biased branches,[69] choosing suitable hash function,[70] or input vector[25]
Use of genetic algorithm	[71]
Modeling context switches	[13,16,34,36]
Modeling impact of OS execution	[72-74]
Reducing Aliasing and Improving Accuracy	
Optimizing biased branches	using static BP[12,75] or side predictor[34,62] for them, not storing them in BH,[2,20] not updating all tables[1,75] and skipping dot-product[21] or BP access[68] for them
Predicting loop branches	[49,76]
Correlating on data values	[37,44,65,77,78]
Modifying or filtering global history	[40,41,59,75,79]
Adapting history length	[16,30]

the blocks in cache. Hence, the tag-size becomes disproportionately larger than the entry-size, and the decoding complexity also becomes higher than that in the cache.[3] Further, increasing the tag size beyond a threshold does not improve accuracy.[36]

Challenges of hybrid BPs: Since no solo BP can predict all branch types, hybrid BPs have been proposed, which use multiple component predictors along with a meta-predictor (also called selector or chooser) to choose the most-suitable component for each branch. The limitation of hybrid BPs is that they need to use large meta-predictors for achieving high accuracy. This, however, further reduces the hardware budget and the effectiveness of component predictors compared to an equal-budget solo BP.

Processor design factors: Compared to single-issue in-order processors, wide-issue out-of-order processors show lower BP accuracy[4] since some predictions may be required even before recent branch results can update the GHR. Also, increasing pipeline depth aggravates branch misprediction cost due to increased branch resolution time. This delays entry of correct path instructions in the pipeline and leads to filling of pipeline with instructions from wrong path,[37] which aggravates the performance loss due to BP.

2.3 | Classification and overview

Table 1 classifies the research works on several key parameters, eg, objective of the technique, BP design features, and the techniques for improving accuracy.

3 | BRANCH PREDICTOR DESIGNS

The simplest 2bc (2-bit counter) BP uses some of the branch address bits to select a 2-bit counter from a one-dimensional branch history table. The value of this counter decides the prediction. More complex BPs use both branch history and current branch address in a two-level organization (Section 3.1) or combine them in different ways to obtain index of a single-level predictor table (Section 3.2). Other predictors seek to improve accuracy by using multiple predictor tables accessed with different indexing functions (Section 3.3), exploiting biased branches (Section 3.4), and using geometric history length BPs to track very large history lengths (Section 3.5). We now discuss several BP designs.

3.1 | Two-level predictors

Yeh et al[80,81] propose a two-level BP which uses a branch history register (BHR) table and a "branch pattern history table" (PHT). Most recent branch results are shifted into BHR. History pattern bits show the most recent branch outcomes for a specific content in the BHR. For a branch, its instruction address is used to index BHR table, and the corresponding BHR contents are used to index PHT for making predictions. All history registers reference the same PHT, and hence, it is termed a global PHT. History pattern of last *P* results of a branch decide its prediction; hence, BHR has *P* bits, which can store 2^P different patterns. PHT has 2^P entries, each of which can be indexed by a different history pattern.

In PHT, branch behavior for the recent *S* occurrences of a given pattern of these *P* branches is stored. The branch is predicted by seeing the branch response for recent *S* instances of the pattern. For instance, let $P = 7$, and the outcome of last *P* branches was 1010011 (1=taken, 0=not-taken). If $S = 5$ and in each of the last 5 times, the past seven branches showed the pattern 1010011, and the branch outcome switched between taken and not-taken, then the level-2 history will be 10101. Then, the BP predicts the current branch as "not-taken."

They propose three implementations of their BP, viz., GAg, PAg, and PAp (see below for the meaning of the terminology). GAg scheme uses only one global BHR and one global PHT for all branches; however, this leads to aliasing in both the levels. PAg scheme uses one BHR for each static branch to record their branch history individually. This reduces aliasing in the first-level table; hence, this scheme is termed as "per-address" branch prediction with global PHT. To remove aliasing in the second-level table also, their PAp scheme uses a PHT for each branch along with one BHR for each static branch. In PAg and PAp schemes, the BHR table can be implemented as a set-associative or a direct-mapped structure. *Results:* Due to the interference effect, the decreasing order of performance of the designs is PAp, PAg, and GAg. For achieving same level of accuracy, PAg and PAp schemes incur the least and the highest hardware cost, respectively. This is because the GAg scheme requires long history register whereas the PAp scheme requires multiple PHTs.

Yeh et al[19] study and compare nine variants of two-level BP designs, which are shown as [G/P/S]A[g/p/s], depending on how branch history (level-1) and pattern history (level-2) are maintained and mutually associated. These designs are shown in Figure 4, and the meanings of the symbols G/P/S and g/p/s are shown in Table 2. The second letter in the name indicates whether the PHT is adaptive (ie, dynamic) or static.

A global history BP (also referred to as "correlation BP")[82] needs only one BHR. A per-address history BP uses one BHR for each static conditional branch, and thus, prediction of a branch is made based on its own execution history only. "S" refers to partitioning of branch addresses into sets, eg, the branches in a 1 KB block (256 instructions) may be members of the same set. The "SetP(B)" indexing function (middle 3 figures in Figure 4) is formed by combining set-index and low-order bits of branch address. In the per-set history BP, all branches in a set update the per-set BHR, and thus, they influence each others' prediction outcome.

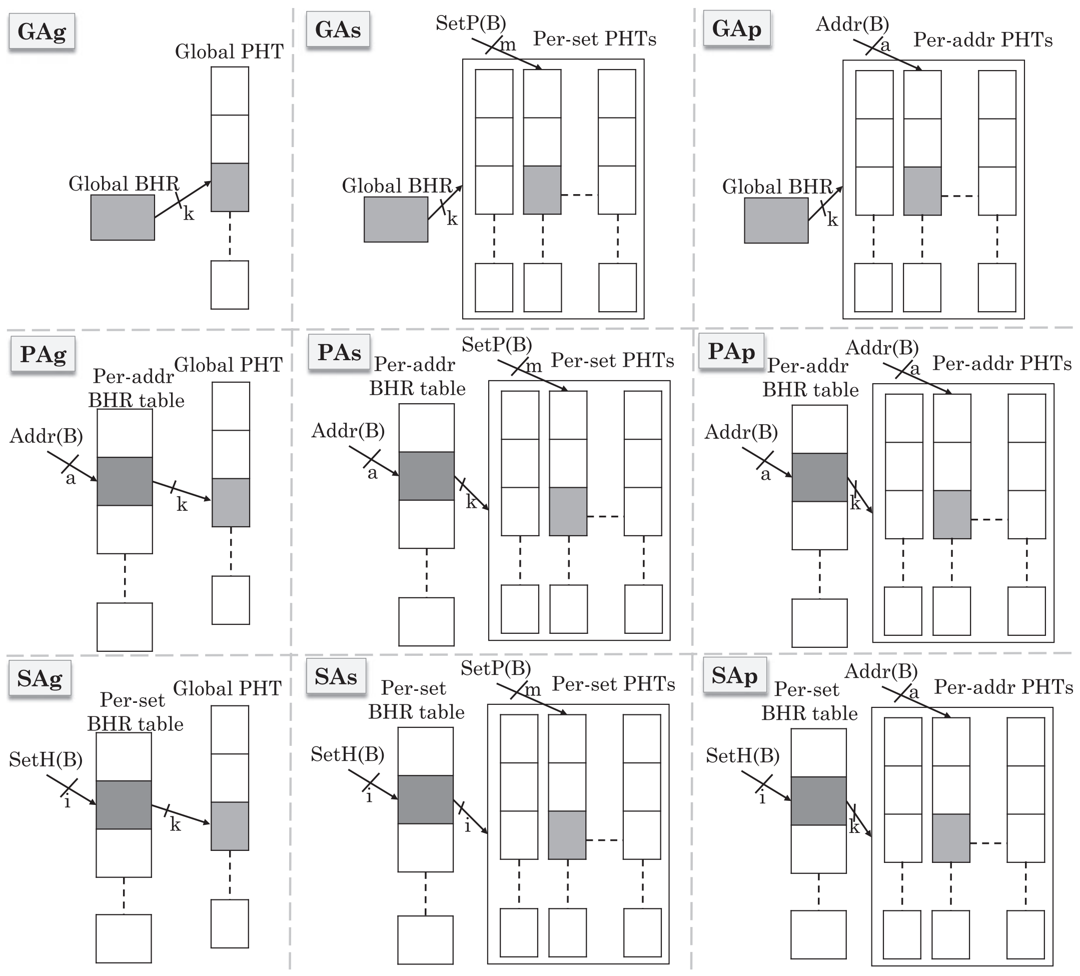


FIGURE 4 Nine variants ([G/P/S]A[g/p/s]) of two-level BPs[19]

TABLE 2 Distinguishing characteristics of each BP in level-1 and 2 tables[19]

Level-1	**G**	**P**	**S**
k branches stored are:	last *k* branches actually seen	last *k* instances of the same branch	last *k* instances of branches from the same set
Level-2	**g**	**p**	**s**
One PHT for:	all branches	each branch	a set of branches

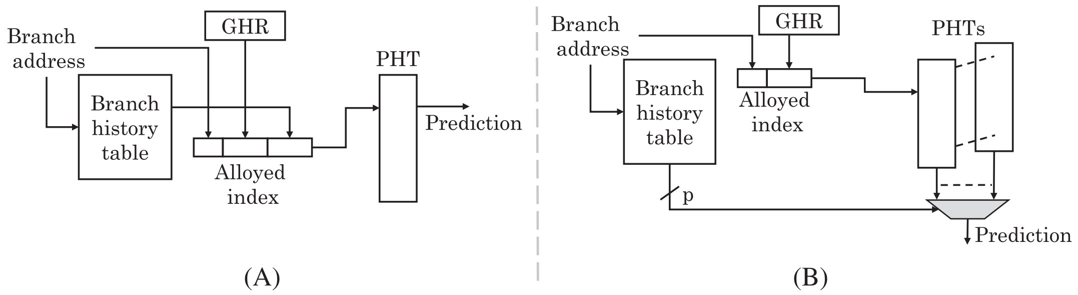


FIGURE 5 (A) Alloyed-index BP,[10] where both global and local history bits are "alloyed" in one PHT index. This allows using both local and global history without requiring a meta-predictor or use of hybrid BPs. (B) To avoid the need of two serial accesses, the PHT is splitted in multiple tables which can be accessed in parallel. A, Two-level BP with alloyed index; B, Reorganized BP of (A) to enable simultaneous access to branch-history table and PHT

Results: In applications with several `if/else` constructs, the branches correlate strongly with the earlier branches, and hence, global history BPs are effective for such applications. However, global history BPs incur large hardware cost since they need longer branch history or multiple PHTs to mitigate aliasing. Per-address history BPs provide high accuracy for applications containing recurring loop-control branches since their periodic branch characteristics are accurately captured by the per-address BHR tables. Further, due to reduced aliasing, they need smaller tables, which reduces their storage cost. Per-set history BPs provide high accuracy for both the above types of applications; however, their hardware cost is higher than even global history BPs since they need separate PHTs for every set. Further, in general, among low- and high-cost BPs, PAs and GAs BPs are the most cost effective BPs.

Skadron et al[10] note that other than aliasing, several mispredictions are caused due to tracking incorrect-type of history for a branch (eg, global history in place of local history and vice versa). They propose a BP which is especially effective in reducing such mispredictions. Their BP adds a GHR to the two-level BP, as shown in Figure 5A. Then, both global and local history bits are "alloyed" in one PHT index. This allows using both local and global history without requiring a meta-predictor or division of a BP in multiple predictor components as in a hybrid BP. Extending the terminology of Yeh et al,[19] their BP is termed as a MAs BP, where "M" refers to merging of local and global histories. Compared to PAs and GAs BPs, their MAs BP requires fewer bits to achieve the same level of anti-aliasing since the merging of histories already reduces aliasing. Bimodal BP[83] is a special case of MAs BP that uses just one bit of local history.

A limitation of their BP is the requirement of two serial accesses, which increases the overall latency. To resolve this issue, they split PHT in multiple tables, which can be accessed concurrently, as shown in Figure 5B. This is feasible in case of small number of local-history bits (eg, $p \leq 4$). This design allows accessing PHT and "branch history table" in parallel. The limitation of this design, however, is the overhead of accessing a large number of tables and a multiplexer. Their alloyed BP provides higher accuracy than the hybrid BPs at small area budget and comparable accuracy at large area budget. Further, it provides better accuracy than the two-level designs.

Sechrest et al[72] note that for small-budget BPs, aliasing can degrade accuracy significantly and even nullify the advantage of modeling inter-branch correlation in global history based BPs, eg, gshare[84] and GAs BP.[19] Thus, the need to avoid aliasing necessitates increasing the size of global history based BP. For local-history based BPs such as PAs, interference is more prononunced in the buffer that stores branch history (first-level) than in the predictor table (second-level). Overall, achieving high BP accuracy with large programs requires sufficient storage resources either in the first or the second-level table.

Pan et al[82] note that due to correlation between branches, the outcome of a branch can be predicted by not only its own history but that of other branches also. For example, in Figure 6A, branch B3 is correlated with B1 and B2. After B1 and B2 have executed, some information for resolving B3 is already known. As shown in Figure 6B, if conditions at B1 and B2 were true, then condition at B3 can be accurately predicted as false. However, BPs based only on self-history, shown in Figure 6C, are unable to use this information. By dividing history of B3 into four subhistories and selectively using proper subhistory, randomness of B3 can be reduced. In general, their technique examines past *K*-branches to split the history of a branch in 2^K subhistories and then predicts independently within each subhistory using any history-based BP, eg, *N*-bit counter-based BP. They use a *K*-bit shift register for storing the result of past *K* branches. The register identifies 2^K subhistories of a branch, and for any subhistory, its *N*-bit counter is used for predicting the outcome which provides *N* prediction bits. BP is updated as in the *N*-bit counter-based BP. Figure 6D shows their BP design assuming $K = 2$ and $N = 2$. Their technique achieves high accuracy with only small hardware overhead.

Chen et al[85] present a conceptual model of branch prediction based on data compression to allow BP research to benefit from the research on data compression. They show that the "two-level" or correlation based BPs are simple versions of "prediction by partial matching" (PPM),[86]

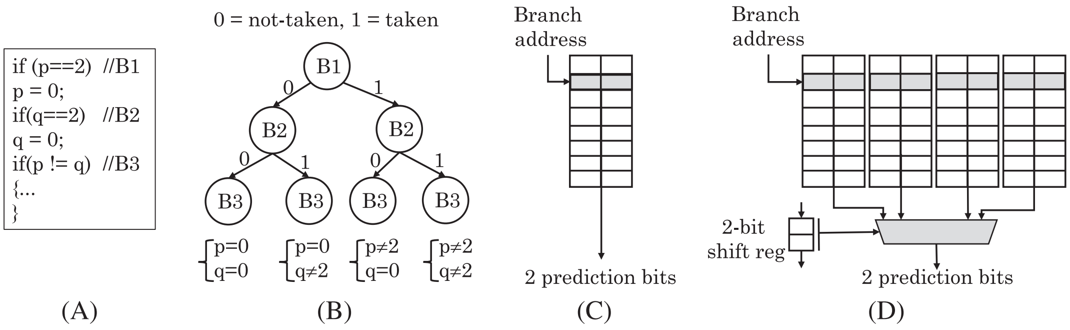


FIGURE 6 (A) A code segment with correlated branches (B) information provided by branch correlation (C) a conventional 2-bit counter BP (D) The correlation BP proposed by Pan et al[82] (assuming $K = 2$ and $N = 2$). A, Example code; B, Info available about p & q after B1 and B2 have executed; C, 2-bit counter BP; D, (2,2)-correlation BP

TABLE 3 Index with gselect and gshare schemes for different addresses and histories. Only gshare is able to distinguish all four cases

Branch Address	Global History	gselect Index (4b+4b)	gshare Index (8b XOR 8b)
00000000	00000001	00000001	00000001
00000000	00000000	00000000	00000000
11111111	00000000	11110000	11111111
11111111	10000000	11110000	01111111

an optimal predictor in data compression. They show that PPM predictor provides small improvement over the two-level predictor for small sized BTBs. As PPM is optimal, substantial asymptotic improvements in two-level BPs are unlikely as long as information presented to the BPs is not modified. Still, by using the predictors proposed in compression field, small improvements can be obtained in low-budget predictor designs. Some researchers have proposed PPM-like BPs[23,87,88] (see Section 3.5 for an example).

3.2 | Using both global history and branch address

Mcfarling[84] notes that for a given BP size, global history BP has lower accuracy than the local history BP. Also, it provides higher accuracy than bimodal BP only when the BP size exceeds 1 KB. This is because for small BP size, the branch address bits employed in bimodal BP can effectively distinguish between different branches. However, with increasing number of counters, every frequent branch is mapped to a different counter, and hence, for very large tables, the information value of each extra address bit approaches zero.

For global history BP, the information value of the counters increases with increasing table size. This allows the global BP to distinguish different branches, although less effectively than the branch address. However, the global BP can store more information than merely identifying the current branch, and as such, with increasing BP size, it outperforms the bimodal BP.

To reduce aliasing by bringing the best of global history and branch history information together, Mcfarling[84] presents "gselect" and "gshare" BPs. In the gselect BP, the (low-order) address and global-history bits are concatenated, whereas in the gshare BP, these bits are XORed to get the table index. Table 3 shows an example of two branches, each having only two common global histories. Clearly, while gshare BP is able to separate the 4 cases, gselect BP is unable to do so. This is because by virtue of using XOR function, gshare BP forms a more randomized index. For the same hardware cost (8b index in the example of Table 3), the gselect BP cannot use higher-order bits to distinguish different branches, and hence, gshare BP provides higher accuracy for large BP sizes. For small sizes, however, gshare BP provides lower accuracy since addition of global history worsens the already-high contention for counters. They also propose two-component hybrid BPs (eg, bimodal[83]+gshare) along with a meta-predictor, which selects the best BP out of the two for each branch. The hybrid BP provides higher accuracy than the component BPs.

3.3 | Using multiple indexing functions

Michaud et al[89] note that aliasing in BP tables is similar to cache misses, and thus, aliasing can be classified as conflict, capacity, and compulsory, similar to that in caches. Since use of tags for removing aliasing introduces large overhead, they propose a "skewed" BP design, similar to the skewed-associative cache design.[90] Since exact occurrence of conflicts depends on the mapping function, their BP uses multiple odd (eg, 3) numbers of BP banks, each of which uses different hash functions obtained from the same information (eg, global history and branch address). Each bank acts as a tagless predictor and is accessed in parallel. Every bank provides a prediction, and the final prediction is chosen using majority voting based on the idea that branches that interfere in one bank are unlikely to do so in other banks. Notice that for BPs divided in multiple banks or tables, different banks can differ in the indexing functions used[89,91] or the length of history used by them.[23,30]

For updating their BP, they consider a "total update" policy, where all banks are updated, and a "partial update" policy, where a bank providing wrong prediction is not updated when the final prediction is correct; However, when the final prediction is incorrect, all the banks are updated. Their BP with 3 banks achieves same accuracy as a large 1-bank BP while requiring nearly half storage resources. This is because their skewed BP increases redundancy by using multiple tables. This increases capacity aliasing but reduces conflict aliasing. Further, the partial update policy performs better than the total update policy since not updating the bank providing incorrect prediction allows it to contribute to the correct prediction for some other stream, which increases the overall effectiveness of the BP.

3.4 | Leveraging biased branches

Chang et al[67] propose classifying branches in different streams and predicting a branch using a BP which is most suited for it. They classify the branches in different classes based on their taken-rate. They experiment with multiple BPs, viz., profile-driven, 2-bit counter, and three two-level BPs, viz., PAs, GAs, and a modified GAg (gshare) BP. They note that biased branches are better predicted by BPs with short BHRs due to their quick warming-up, and the fact that for a fixed hardware cost, smaller history register implies larger number of PHTs which reduces aliasing. In contrast, mixed-direction (ie, unbiased) branches are better predicted by BPs with long BHRs due to their ability to distinguish between larger number of execution states and to effectively hold histories of correlated branches.

Lee et al[83] present the "bimodal" BP, which is shown in Figure 7. Their BP divides the level-2 counter table in two parts, viz. taken and not-taken "direction predictors." For any history pattern, one counter is chosen from each part. From these, one counter is finally chosen to make the prediction based on the input of another "choice predictor." Choice predictor is indexed by branch address only. The classification of global-history patterns in two groups is done based on per-address bias of the choice predictor. They use partial-update policy, where only the selected counter is updated based on the branch result. Their technique always updates the choice predictor, except when the choice is opposite of the branch result but the final prediction of the selected counter is right. Overall, bimodal BP separates branches showing negative aliasing and keeps-together those showing neutral or beneficial aliasing. By virtue of removing harmful aliasing, their BP achieves high prediction accuracy.

Sprangle et al[22] propose a technique which converts harmful interference to beneficial or harmless cases by changing the interpretation of PHT entry. Their BP is termed as "agree predictor," and it is shown in Figure 8. For each branch, they add a biasing bit, which predicts the most likely outcome of the branch. Instead of predicting the direction of a branch (as done in the traditional schemes), the PHT counters in their technique predict whether the branch outcome will agree with the biasing bit. When the branch is resolved, PHT counters are increased if the branch direction matched the biasing bit and vice versa. They set biasing bit to the direction of the branch in its first execution. With proper choice of biasing bit, the counters of two branches mapping to the same PHT entry are more likely to be updated in the same direction viz. agree state.

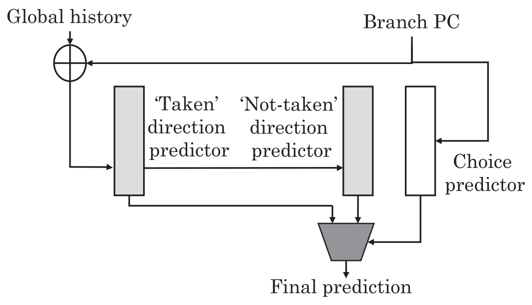


FIGURE 7 Bimodal BP.[83] The level-2 counter table is divided in two parts, namely taken and not-taken direction predictors. Based on the input of a "choice predictor," a counter is chosen from one of the direction predictors

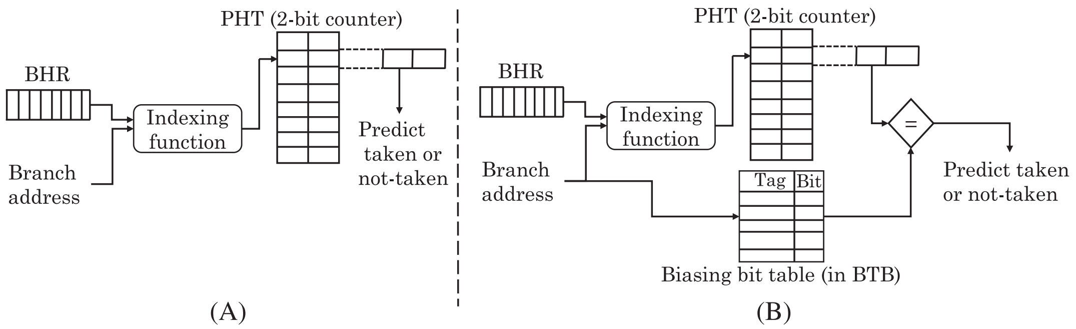


FIGURE 8 Agree predictor.[22] The biasing bit predicts the most likely outcome of the branch, and the PHT counters predict whether the branch outcome will agree with the biasing bit. A, Conventional BP; B, Agree BP

To see an example, assume two branches, B1 and B2, whose taken-rates are 85% and 15%, respectively. In a conventional BP, the probability of harmful aliasing between them is: B1Taken * B2Nottaken + B1Nottaken * B2Taken = (85% * 85% + 15% * 15%) = 74.5%. In the agree predictor, biasing bit is set to the direction the branch takes in its first execution. This biasing bit is expected to represent the correct direction of the branch nearly 85% of the time. Assuming that the biasing bits are correctly set for the branches (taken for B1 and not-taken for B2), the probability that any combination of B1 and B2 will have opposite direction is: (B1Agrees * B2Disagress) + (B1Disagrees * B2Agrees) = (85% * 15% + 15% * 85%) = 25.5%. Thus, the PHT entries in the agree BP are less likely to differ for two different branches, which reduces harmful aliasing. Another benefit of their technique is that PHT entries of a new branch are expected to be in warmed-up state (viz. agree state) already, which improves accuracy.

3.5 | Geometric history length BPs

Seznec[30] presents GEHL (geometric history length) BP, which allows capturing both recent and old correlations. GEHL BP uses *M* predictor tables T*i* ($0 \leq i < M$) whose indices are obtained by hashing global path/branch history with the branch address. Typical values of *M* are between 4 and 12. Each table stores predictions as signed counters. Prediction is performed as shown in Figure 4. Different tables use different history lengths, as shown in Table 4. Using history lengths in geometric series permits using long history lengths for some tables while still using most of the storage for tables with short history lengths. For example, assuming an 8-component BP, if $\alpha = 2$ and $L(1) = 2$, then $L(i)$ form the series $\{0, 2, 4, 8, 16, 32, 64, 128\}$ for $0 \leq i < M$. Thus, 5 tables are indexed using at most 16 history bits while still capturing correlation with 128-bit history.

The predictor is updated only on a misprediction or when the absolute value of *Sum* is below a threshold. Their BP allows adapting history lengths such that in case of high aliasing on updates, short history length is used and vice versa. Also, the update threshold is adapted with a view to keep the number of updates on mispredictions and those on correct predictions in the same range. Using ahead pipelining (discussed in Section 4.1), the latency of their BP can be kept low. The limitation of their BP is its complexity and the need of checkpointing a large amount of state that must be restored on a misprediction. Overall, GEHL BP provides high accuracy.

Seznec et al[23] present TAGE (tagged geometric history length) BP, which is an improved version of the tagged PPM-like BP.[88] The TAGE BP uses a base predictor (T0), which provides basic prediction and multiple partially-tagged components T*i* ($1 \leq i \leq M$). T0 can be a 2-bit counter bimodal table. T*i* predictor is indexed with a global history length L(*i*) = $\lfloor(\alpha^{i-1} \times L(1) + 0.5)\rfloor$. The T*i* predictors have a signed “prediction” counter whose sign gives the prediction a tag and a “useful” counter (U). Figure 9 shows the TAGE BP with 5 components. For making a prediction, both the base BP and the tagged BPs are simultaneously accessed. If multiple tagged BPs hit, the one with the longest history length (say Tc) provides the prediction; otherwise, the prediction of the base predictor is used. For example, if T2 and T4 hit but T1 and T3 miss, then T4's prediction is used ($c = 4$), and prediction of T2 is called “alternate prediction.” If no component hits, the default prediction acts as the alternate prediction.

As for updating, when the alternate prediction is different from the final prediction, the “U” counter is increased or decreased by one if the final prediction is correct or incorrect, respectively. Periodically, ‘U’ counter is reset so that the entries are not marked useful forever. The “pred” counter of Tc gets updated on correct and incorrect predictions. Additionally, on an incorrect prediction, if $c < M$, an entry is allocated in a predictor Tk ($c < k < M$).[23] If the prediction of a Tc component is provided by such “newly allocated” entries, alternate prediction is used as final prediction since

TABLE 4 Prediction and indexing process in GEHL BP[30]

Prediction	One counter C(*i*) is read from each table T*i* and they are added as follows: $Sum = M/2 + \sum_{i=0}^{M-1} C(i)$. If $Sum \geq 0$, prediction is taken and vice-versa
Table indexing	Table T0 is indexed using branch address. Tables $1 \leq i < M$ are indexed using history length of: L(*i*) = $\lfloor(\alpha^{i-1} \times L(1) + 0.5)\rfloor$

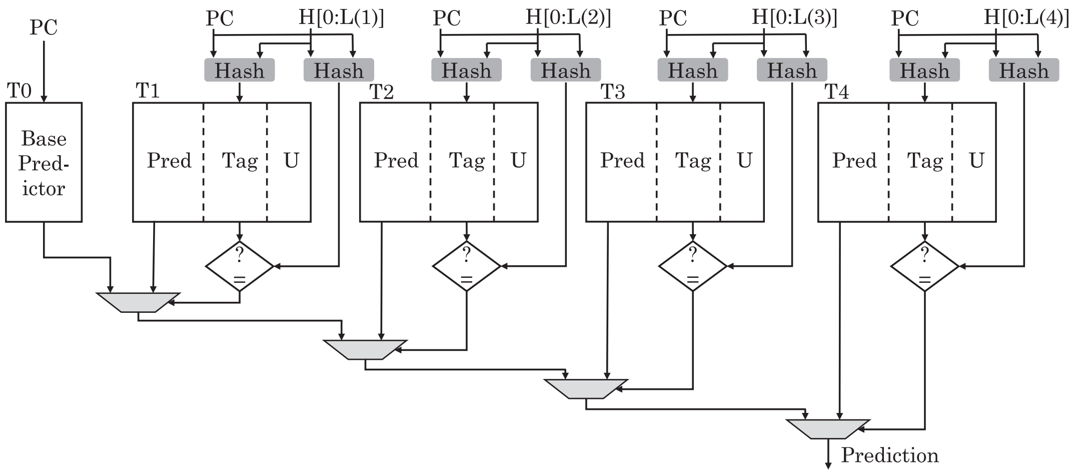


FIGURE 9 TAGE BP,[23] which uses a base predictor (T0) and multiple partially-tagged components (T1 to T4 in the above example)

newly allocated entries tend to provide wrong prediction for some time due to lack of training. Their BP outperforms previous predictors,[30,88] and thus, for BPs using geometric history lengths, partial-tagging[23] is more cost-effective in selecting the final prediction compared to the adder tree.[30] Also, using more than 8 tagged predictors does not provide any benefit. They also show that observing the Tc component and prediction counter value allows obtaining an estimate of misprediction probability.[92]

Comparison: For comparable hardware budget, gshare[84] and perceptron BPs[51] track correlations with nearly 20 and 60 branches, respectively,[42] whereas GEHL and TAGE track correlations with nearly 200 and 2000 branches, respectively. Among the solo-BPs, the TAGE predictor is considered the most accurate BP,[62] and by combining it with side predictors, even more accurate hybrid BPs have been obtained.[62,63,93,33] Also note that while TAGE BP[30] uses multiple tagged predictors, other BPs[36] use only one tagged predictor.

4 | DESIGNS AND IMPLEMENTATIONS OF NEURAL BPS

As for neural BPs, researchers have proposed BPs using perceptron,[51] back-propagation,[31,55,56] learning vector quantization,[55] and other neural BPs.[31] We now discuss several neural BP designs (Section 4.1), techniques for optimizing them (Section 4.2), and their implementation in analog domain (Section 4.3).

4.1 | Designs of neural BPs

Jimenez et al[51] propose a perceptron-based BP which uses one-layer perceptron, a simple NN, to represent each branch. A perceptron is a vector that stores multiple weights (w) showing the degree of correlation between different branches. The outcomes of previous branches (1 = taken, −1 = not-taken) form the input to perceptrons. Input x_0, being always 1, provides the bias input. The dot-product of weights and inputs provides the output Z based on the following formula: $Z = w_0 + \Sigma w_i x_i$. A positive or negative value of Z means taken or non-taken prediction, respectively. When the actual branch outcome is available, the weights of elements agreeing or disagreeing with the outcome are incremented or decremented, respectively. Figure 10 explains the working of perceptron BP with an example.

The size of the two-level BPs grows exponentially with history length since they index PHT with branch history. By comparison, the size of perceptron BP increases only linearly with history lengths.[31] Hence, their BP can account for much longer histories, which is useful since highly correlated branches may be separated by long distances. The weights remain close to 1, −1, or 0 in case of positive, negative, and no correlation, respectively. Thus, the weights give insight into degree of correlation between branches. Also, the output of their BP shows confidence level of predictions since the difference between the output and zero shows the certainty of taking a branch.

Their BP works well for applications with many linearly separable branches. A Boolean function over variables $x_{1..n}$ is linearly separable if values of $n + 1$ weights can be found such that all the false instances can be separated from all the true instances using a hyperplane. For example, AND function is linearly separable, whereas XOR function is linearly inseparable. This is illustrated in Figure 11A,B. The perceptron BP works well for

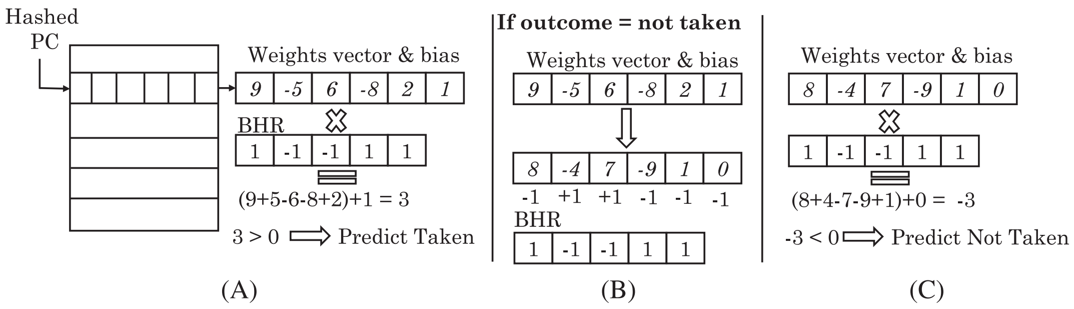


FIGURE 10 Prediction and training process of the perceptron predictor.[51] A, First time prediction; B, Retraining; C, Next time prediction

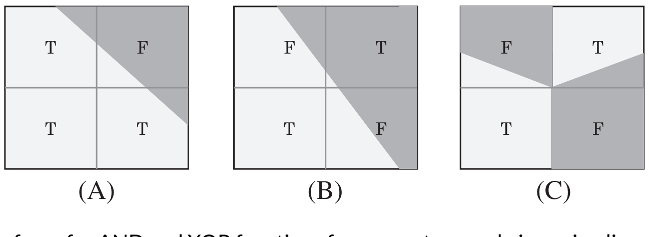


FIGURE 11 An example of decision surfaces for AND and XOR functions for perceptron and piecewise-linear BPs.[54] X and Y axes represent two input variables. Negative and positive sides show false and true, respectively. The output is shown as T or F (true or false). A, Perceptron decision surface for AND function (linearly separable): classifies all inputs correctly; B, Perceptron decision surface for XOR function (linearly inseparable): does NOT classify all inputs correctly; C, Piecewise linear decision surface for XOR function (linearly inseparable): classifies all inputs correctly

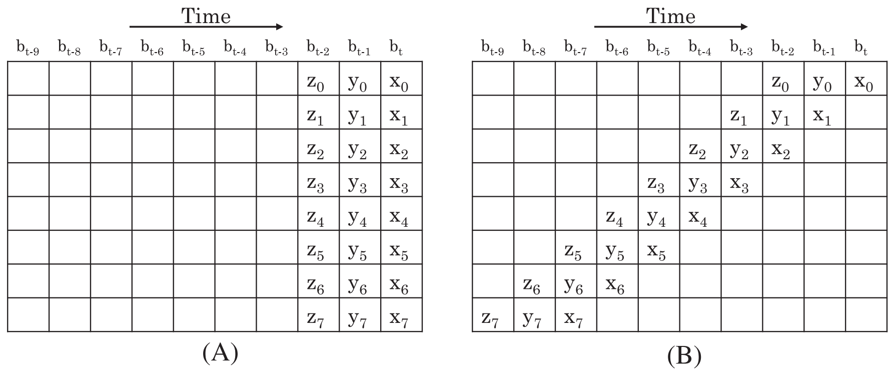

Time
b_{t-9} b_{t-8} b_{t-7} b_{t-6} b_{t-5} b_{t-4} b_{t-3} b_{t-2} b_{t-1} b_t
z_0 y_0 x_0 … z_7 y_7 x_7
(A)
(B)

FIGURE 12 Weights used for predicting branch b_t with (A) the perceptron BP[51] and (B) the path-based neural BP.[52] The history length is 7 and the vertical columns show the weights vectors. The approach of staggering the computation over time, shown in (B), is termed as "ahead pipelining"

AND function but not for XOR function. Thus, for linearly inseparable function, perceptron BP has lower accuracy, whereas previous predictors can learn any Boolean function if sufficient time is given to them. Hence, for general applications, their BP is more useful as a component of a hybrid predictor instead of as a solo predictor. Another limitation of their BP is its high latency.

Jimenez[52] presents a path-based neural BP that uses "ahead pipelining" approach to reduce latency. Their BP chooses its weight vector depending on the path that leads to a branch instead of only the branch address, and this helps in achieving higher accuracy. Similar to perceptron BP, their path-based BP maintains a matrix of weight vectors. For predicting a branch, a weight vector is read and only its bias weight needs to be added to a running sum for making a prediction, as shown in Figure 12. This running sum is updated for multiple previous branches. Thus, their ahead pipelining approach staggers the computation over time, and hence, the computation can start even before a prediction is made. The prediction can complete in multiple cycles, and this removes the requirement of BP overriding. A limitation of their approach is that it does not use PC of a branch to choose its weights, leading to reduced accuracy. Also, ahead pipelining works by starting the prediction process with stale or partial information and continually mixing latest information in the prediction process; however, only limited new information can be used in the ahead pipelining approach.[26] Further, branch misprediction requires rolling back a large amount of processor state to a checkpoint.

Jiménez[54] presents a piecewise-linear BP, which is a generalization of perceptron and path-based BPs.[51,52] For every branch B, their BP tracks components of all paths leading to B. It tracks likelihood of a branch at a certain position in the history to agree with the result of B; thus, their BP correlates each element of each path with the result of B. A prediction is made by aggregating correlations of every component of the current path. Overall, their BP generates multiple linear functions, one for each path to the current branch for separating predicted not-taken from predicted taken branches. Hence, their BP is effective for linearly inseparable branches, as shown in Figure 11C. Assuming no constraints of storage budget or latency, their BP achieves lower misprediction rate than other BPs. They further propose a practical implementation of their BP, which uses ahead-pipelining and small values of history length. Further, branch address and address of branch in the path history are stored as modulo N and M, respectively. Perceptron-based BP[51] utilizes a single linear function for a branch ($M = 1$) and path-based neural BP[52] utilizes a single global piecewise-linear function for predicting all the branches ($N = 1$), and thus, they are special cases of the piecewise-linear BP.[54] They show that the practical version of their BP also outperforms other predictors.

Jimenez et al[31] note that unlike (single-layer) perceptron based BP, multi-layer perceptron with back-propagation–based BP can learn linearly inseparable functions, and hence, it is expected to provide higher "asymptotic" accuracy than perceptron BP. However, in practice, back-propagation neural BP provides lower accuracy and performance since it takes much longer time for both training and prediction. These factors make back-propagation based BP infeasible and ineffective.[56]

Tarjan et al[26] propose a hashed perceptron BP, which uses the ideas from gshare, path-based and perceptron BPs. They note that perceptron BPs use each weight to measure the branch correlation. This, however, leads to linear increase in the number of tables and adders with the amount of history used and requires the counters for each weight to be large enough so that a weight can override several other weights. They propose a BP which assigns multiple branches to the same weight by XORing a segment of the global branch history with the PC. Instead of doing a single match with the global branch or path history, they divide it into multiple segments for performing multiple partial matches. Further, instead of using path history, they use only the branch PC. With hashed indexing, every table works as a small gshare BP, and thus, their BP can also predict linearly inseparable branches mapped to the same weight. Also, their BP uses smaller number of weights for the same history length compared to a path-based BP, and this reduces the chances that multiple weights with no correlation can overturn a single weight with strong correlation. Compared to the global or path-based BP, their BP has a shorter pipeline, which reduces the amount of state to be checkpointed. To reduce BP latency, they use ahead pipelining. Compared to path-based and global neural BP, their BP improves accuracy significantly while reducing the number of adders.

Gao et al[25] note that in perceptron-based BPs, the value of perceptron weights shows the correlation strength. For example, Figure 13 shows four branches, where Branch1 is decided by two random variables, Branch2 is correlated with Branch1 since they share the same random variable (`k1`), Branch3 is merely inverse of Branch1, and Branch4 relates to both Branch2 and Branch3 since it uses an XOR function. A perceptron BP is used where one perceptron with 8-bit GHR is used for each branch. The perceptron weights (w1-w8) and misprediction rates for 100M instruction

```
while (1) {
k1 =rand(0,1000); k2 = rand(0,1000);
D1 = k1>k2; D2 = k1>500; D3 = k1 ≤ k2;
if (D1)
    count1++
if (D2)
    count2++;
if (D3)
    count3++;
if (D2 ^ D3)
    count4++;
}
```

Branch1: misprediction rate (MR) = 50.01%

w0-w7:	-1	-1	-5	7	-3	-5	-1	-1

Branch2: MR = 25.41%

w0-w7:	26	0	0	2	2	2	0	0

Branch3: MR = 0%

w0-w7:	-1	-31	1	1	-1	-1	-3	-1

Branch4: MR = 19.09%

w0-w7:	12	22	-12	10	10	-2	0	-2

FIGURE 13 A program to show that perceptron predictor weights provide quantitative measure of branch correlation[25]

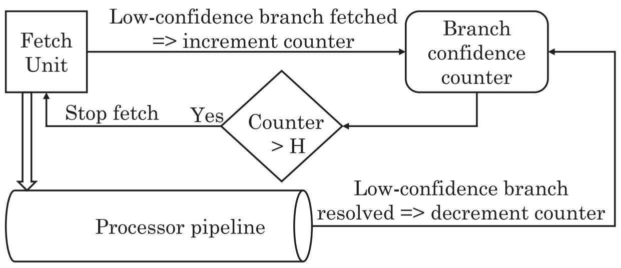

Fetch Unit
Low-confidence branch fetched => increment counter
Branch confidence counter
Stop fetch
Yes
Counter > H
Processor pipeline
Low-confidence branch resolved => decrement counter

FIGURE 14 Pipeline gating approach stops fetching instructions when more than a threshold number (H) of low-confidence branches are fetched. This prevents misspeculated instructions from entering the pipeline

simulation is shown on the right side of Figure 13. Evidently, since Branch1 has no correlation with previous branches, its weights have small random values. Branch2 has large w1 (strong correlation with GHR[0]) due to its correlation with Branch1 and small random w2-w8 values. Branch3 has a single large weight w2 (strong correlation with GHR[1]) since the outcome of Branch3 can be accurately found by knowing that of Branch1. Branch4 has larger weights since it correlates in nonlinear manner with the previous branches.

Further, by reassembling the input to BP, its accuracy can be improved, eg, if only two recent branches show the most correlation out of N, perceptron size can be reduced from N to 2, which improves accuracy due to avoidance of noise. Also, weak correlation can be replaced by stronger ones, eg, redundant history can be used, which helps in handling linearly inseparable branches. They use static profiling for finding a suitable input vector for each workload type, and this is selected at runtime based on the application type. Further, to account for change in inputs and phase behavior, the correlation strength is periodically tested at runtime, and the weakly correlated inputs are substituted by the strongly correlated inputs. Their technique improves BP accuracy significantly.

Akkary et al[27] propose a perceptron-based branch confidence estimator. The input to a perceptron is global branch history where taken branches are recorded as 1 and not-taken branches are recorded as −1. Training of confidence estimator happens at retirement (ie, non-speculatively). If p (=1 for incorrectly predicted and −1 for correctly predicted branch) and c (=1 and −1 for branches assigned low and high confidence, respectively) have different signs, then weights are updated as $w[i] += p * x[i]$ for all i. Training their BP using right or wrong prediction provides higher accuracy than using taken or not-taken outcome.[51] Also, their confidence estimator provides reasonable coverage of mispredicted branches. Further, they divide the non-binary output of their confidence estimator in two regions: strongly and weakly low confident and then apply branch reversal and pipeline gating (respectively) to them. This allows improving BP and pipeline gating using a single hardware. Note that pipeline gating approach, shown in Figure 14, stops fetching instructions when multiple low-confidence branches are fetched. This prevents misspeculated instructions from entering the pipeline and thus saves energy.

4.2 | Optimizing neural BPs

Despite their higher accuracy, neural BPs also have high overheads, as shown in Table 5. We now discuss works that seek to reduce the implementation overhead of neural BPs.

Jimenez et al[21,94] note that branch history length needs to be high for achieving high accuracy; however, keeping the path history length of path-based neural BP same as the history length increases its storage overhead significantly (refer Figure 15A). To resolve this tradeoff, they propose decoupling path and branch history lengths by using "modulo path-history." They use only most recent P ($< h$) branch addresses in path history, and this reduces the number of pipeline stages and tables to P and also reduces the amount of state to be checkpointed. For finding the index for w_i, they use $PC_i \ mod \ P$ in place of PC_i. Figure 15B shows this design assuming $P = 3$. Every Pth weight is indexed using the same branch address, and this allows interleaving the weights to reduce the number of tables required to P, as shown in Figure 15C.

TABLE 5 Features and challenges of neural BPs

Neural BPs	Features and Challenges
Perceptron BP[51]	Requires computing a dot product with tens or hundreds of values,[31,46] which increases its latency, energy, and area and renders it infeasible for implementation in real processors
Path-based neural BP[52]	Uses ahead pipelining to offset latency issue; however, it replaces more power and area efficient carry-save adders of original perceptron BP with several carry-completing adders, leading to higher complexity
Piecewise-linear neural BP[54]	Provides even higher accuracy but significantly increases the checkpointing and recovery overhead and the number of adders

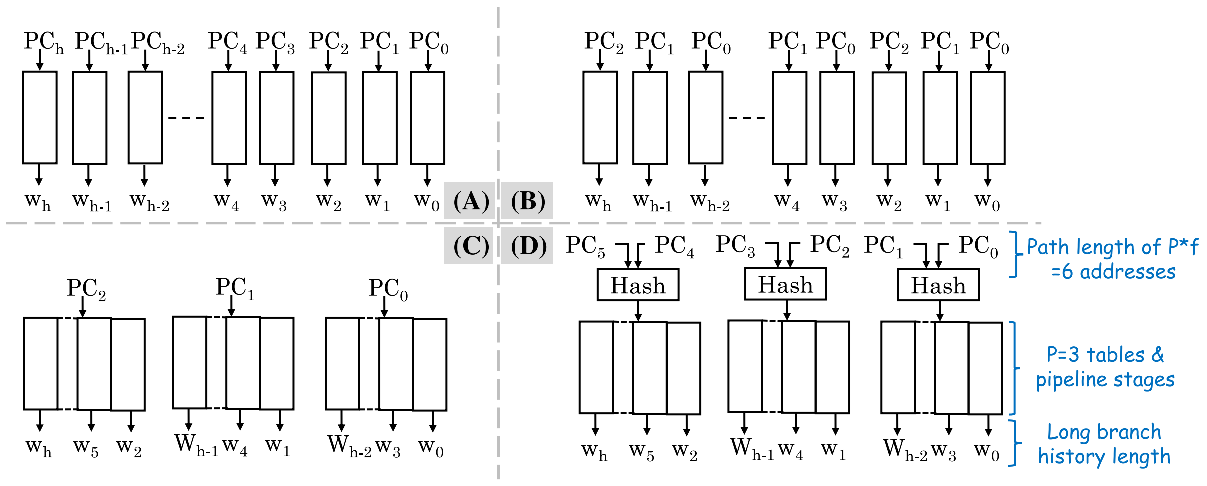

(A)
(B)
(C)
(D)
Hash
Path length of P*f =6 addresses
P=3 tables & pipeline stages
Long branch history length

FIGURE 15 (A) Use of h (= P) tables in original path-based neural BP (B) logical design and (C) physical design of path-based neural BP with modulo path-history (P = 3) [94] (D) design of path-based neural BP with folded modulo path-history (path-length, history-length and table-counts are all different)[94]

Since reduction in path-history length prohibits exploiting correlation with branches more than P address away, they propose a "folded modulo path" design for decoupling path-history length from the predictor pipeline depth. This design uses path history length of $P \times f$ while still using P tables (f = folding factor). While traditional path-based neural BP indexes each table with one branch address, their design hashes f addresses to obtain the index. Figure 15(D) illustrates this design with $P = 3$ and $f = 2$ for keeping a path-length of 6 with only 3 tables. Their history-folding approach can also be used with other BPs.

Loh et al[21] note that for biased branches, tracking h weights and doing their dot-product is unnecessary. Their technique considers a branch whose bias weight (w_0) has reached the minimum or maximum value as a biased branch. For such branches, only bias weight is used for making prediction, and no dot-product is performed. In case of correct prediction, no update is performed, which saves energy and reduces aliasing for remaining branches. Their proposed BP reduces energy consumption and area with negligible reduction in accuracy.

4.3 | Analog implementations of neural BPs

To lower the implementation overhead of neural BPs, some works use the properties of analog domain[45,46,53] or memristors.[45,53] Although these implementations reduce energy consumption, they also require conversion between digital and analog domains. We now discuss these techniques.

Amant et al[46] present a neural BP which employs analog circuitry for implementing power-hungry parts of BP. Digital to analog converters change digital weights into analog currents and current summation is used to combine them. By steering positive and negative weights to different wires and finding the wire with higher current provides a dot-product result as a single-bit, naturally working as an analog-to-digital converter. Their BP uses two features for improving accuracy. First, given the fast computation speed of analog circuitry, their BP obviates the need of using ahead pipelining and thus can use up-to-date PCs to generate the weights. Second, since recent weights generally correlate more strongly with the branch result than older weights, their BP scales the weights based on this correlation to improve accuracy. In analog domain, this only requires changing the transistor sizes, whereas in digital domain, this would have required performing several energy-hungry multiply operations. Their BP is fast, highly energy efficient, and nearly as accurate as L-TAGE[93] predictor. Also, its accuracy is only slightly lower compared to an infeasible digital design.

Jimenez[2] proposes several optimizations to analog neural predictor[46] to exercise tradeoff between accuracy and implementation overhead. First, instead of using only global history, they use both global and per-branch history for prediction and training, which improves accuracy. Second, since recent branches show higher correlation than older branches, their weights are also larger, and hence, they represent weight matrix such that row-size changes with the column-index. Third, depending on correctness of a partial prediction, the corresponding coefficient is tuned. Fourth, the minimum value of perceptron outputs below, which perceptron learning is issued, is changed dynamically. Next, to reduce aliasing between bias weights, a cache is maintained for storing partially tagged branch addresses and bias weight for conditional branches. Finally, since the accuracy of their BP becomes low when its output is close to zero, they also use two additional predictors, viz., gshare and PAg (if confidence of gshare is low). The proposed optimizations improve the accuracy significantly.

Wang et al[45] present an analog neural BP designed with memristor for saving BP energy. They store perceptron weights in a two-dimensional table of "multi-level cell" of memristor device. Each multi-level cell has two analog outputs, and their relative difference shows the weight stored in the multi-level cell. To negate the weight, only the roles of two weights need to be exchanged using a history bit. This allows multiplying with 1 or −1 in much less time than in digital domain. For making a prediction, a row of weights is indexed using PC. Also, the corresponding global history bit is read, and if it is 1, one analog current signal of the cell is put on positive line and another on negative line. Converse action is performed if the history bit is 0. All the current signals on the negative (positive) line are automatically added using Kirchoff's current law. This process is much faster than addition in digital domain. If the total current on positive line is greater than that on the negative line, the prediction outcome is "taken" and vice versa. Compared to a digital implementation, their implementation improves accuracy due to both characteristics of analog computation and memristor. This further helps in resolving some irregular dependent branches. A limitation of using memristor is relative lack of commercial maturity of memristor devices, which may reflect in high noise, latency, process variation, etc.

5 | BPS FOR HARD-TO-PREDICT BRANCHES

Several branches cannot be predicted by regular BPs such as TAGE BP. These branches may be loop-exit branches (Section 5.1), those inside nested loops (Section 5.2), those having a large period (Section 5.3), etc. Similarly, some branches can be better predicted based on data correlation or address correlation (Sections 5.4-5.5). We now discuss BPs that especially target such hard-to-predict branches, and hence, they are suitable to be used as side predictors.

5.1 | Predicting loop-exit branches

Sherwood et al[76] note that loop exits cannot be predicted by local history BP if the loop-count (*N*) is higher than the local history size. They can be predicted using global history only if the global history size is higher than *N* or there is a distinct branching sequence before the loop exit. They present a loop exit buffer (LEB)–based BP for predicting loop branches. Figure 16A shows an example of a loop branch. LEB detects a backward branch that is mispredicted by the primary BP, as a loop branch, and inserts it in the LEB. This is shown in Figure 16B. As for the overall BP design, the primary BP provides a prediction, which is overridden by their LEB-based BP in case its prediction has high confidence.

In LEB, the TripCount field stores the number of successive times the loop-branch was taken before the last not-taken;, and the confidence bit shows that the same loop TripCount has been observed at least twice consecutively. For any branch hitting in LEB, if IterCount is not equal to TripCount, IterCount is incremented by one, and thus, IterCount tracks the number of times the branch has been consecutively taken. When IterCount equals TripCount and the confidence bit equals one, a loop-exit is predicted. When a branch is resolved, TripCount and confidence bit of a not-taken loop-branch are updated. They also use extra counters (not shown in Figure 16B) to recover from branch mispredictions. Their loop predictor can predict with up to 100% accuracy after a brief warmup period. However, in case of frequently changing loop-count, their technique shows poor coverage and/or accuracy.

Sendag et al[49] propose a design which uses a complementary BP along with the conventional BP. The complementary BP focuses only on commonly mispredicted branches, whereas the conventional BP speculates on the predictable branches. They use a "branch misprediction predictor," which uses the number of committed branches between consecutive branch mispredictions for any index, to predict the time of next branch misprediction. Based on this, the direction of this expected misprediction is changed, allowing progress on the correct-path. Most mispredictions removed by "branch misprediction predictor" are due to (1) loop branches with changing loop-counts that are too long for a BP and (2) early

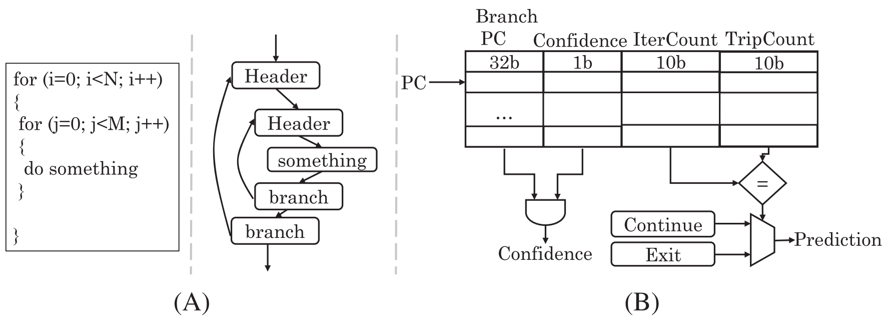


FIGURE 16 (A) Illustration of loop branches and (B) working and operation of loop exit buffer (LEB).[76] A, Source-code and flow-graph of nested loop; B, LEB design and operation

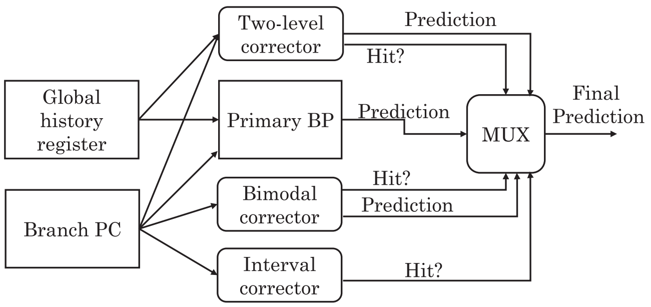


FIGURE 17 The overall BP design proposed by Lai et al.[95] It has a primary BP and three correctors: an interval corrector, a bimodal corrector, and a two-level corrector

loop-exit, eg, due to the `break` instruction. The "branch misprediction predictor" can work with any BP. Their technique improves overall prediction accuracy and saves energy.

Lai et al[95] propose a design where multiple correctors are used to correct the prediction of branches that are mispredicted by the primary BP. Each corrector focuses on a distinct pattern and thus can correct only the branches it remembers using the tags. In each corrector, when required, the entry with least confidence is replaced. They use gskew BP[89] with partial update policy as the primary BP. All correctors keep a confidence value, and the correction is made only if the confidence of their prediction is higher than a threshold. As shown in Figure 17, they use three correctors, (1) an interval corrector, which corrects loop-exit branches. This corrector assumes that after a fixed interval, the primary BP will mispredict, and hence, it corrects primary BP after a fixed interval. They use two interval values to handle early loop-exits. Depending on whether the primary prediction is correct or incorrect, the confidence is decreased or increased. If the primary BP makes a misprediction and the current counter does not match any of the two interval values, the current counter replaces the interval value with lower confidence. Also, the confidence of interval value with higher confidence is decreased by one. Notice that both their interval corrector and the "branch misprediction predictor" of Sendag et al[49] work by predicting when a misprediction will happen and then avoiding that misprediction.

(2) The bimodal corrector tracks local history or bimodal patterns. (3) Two-level corrector tracks two-level patterns[81] and its reversal and update schemes are similar to those of other correctors. Due to using global histories, it achieves high accuracy, and hence, its confidence threshold is set to be lower than that of other correctors. In terms of decreasing accuracy, the correctors are: two-level, interval, and bimodal. Their overall BP design achieves high accuracy. The limitation of their design is that primary BP and correctors operate in parallel, and they are found to have same predictions for more than half the times, which leads to redundancy and energy wastage.

5.2 | Predicting branches inside nested loops

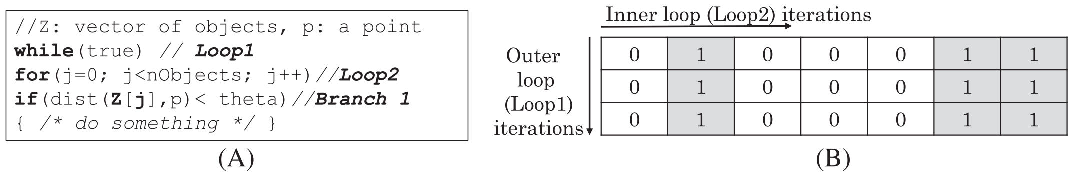
Albericio et al[64] note that results of many branches of nested loops are correlated with those of prior iterations of the outer loop and not recent results of the inner loop. Mathematically, for a branch in the inner loop, its `Outcome[i][j]` is correlated with `Outcome[i-1][j+D]`, where `D` is a small number such as 0, −1, or +1. Figure 18 shows examples of such branches.

On viewing as a linear history, the pattern present in the outcome stream of such branches is not clear, and hence, these branches appear hard-to-predict. However, on storing the history as multidimensional matrix instead of one-dimensional array, strong correlation can be observed. Figure 19B shows an example of storing the correlation in a two-dimensional matrix. They propose a BP which can identify patterns in branches showing multidimensional history correlations. They use their BP as a side predictor to ISL-TAGE BP.[33] Using ISL-TAGE BP, they identify candidate branches in the inner loop along with their current and total iteration counts. For branches that behave almost same in each iteration, their behavior in the first iteration is noted and using them, prediction is made for the subsequent iterations. Figure 19A shows a sample program code, where branch 1 depends only on `j` value, and hence, for the same value of `j` in different iterations of outer loop, the outcome of the branch remains same. This is evident from Figure 19B which shows the outcome of the branch for different iterations of outer and inner loops.

For other branches, eg, those showing diagonal pattern, their BP identifies correlations based on a part of history consisting of bits from the previous iteration and the present iteration streams. As an example, for branch 2 in Figure 20A, the branch outcome depends on both `i` and `j`. Hence,

```
for ( i = 0; i < iMax; i++ )
  for ( j =0; j < jMax; j++) {
     if ( arr1[i+j] > 0 ) {...}                  //Branch1
     if ( arr2[i][j]-arr2[i-1][j] > 0) {...}    //Branch2
     if ( arr3[j] > 0 )                          //Branch3
        if ( arr4[j] > 0 ) {...}                //Branch4
```

FIGURE 18 An illustration of branches whose results correlate with previous iterations of the outer loop.[63,64] Notice that for Branch1, `Outcome[i][j]` equals `Outcome[i-1][j+1]`, and for Branch2, `Outcome[i][j]` is weakly correlated with `Outcome[i-1][j]`. For both Branch3 and Branch4, `Outcome[i][j]` equals `Outcome[i-1][j]`. It is assumed that the values of arrays `arr1`, `arr2`, `arr3`, and `arr4` are not changed inside the loops



```
//Z: vector of objects, p: a point
while(true) // Loop1
for(j=0; j<nObjects; j++)//Loop2
if(dist(Z[j],p)< theta)//Branch 1
{ /* do something */ }
```

(A)

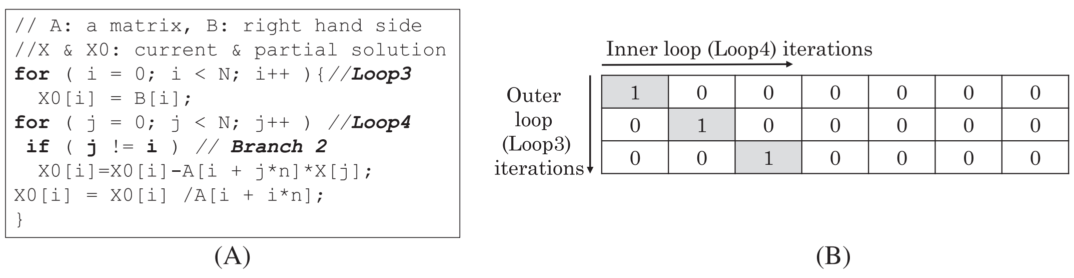
Inner loop (Loop2) iterations

Outer loop (Loop1) iterations

0	1	0	0	0	1	1
0	1	0	0	0	1	1
0	1	0	0	0	1	1

(B)

FIGURE 19 (A) Program 1 and (B) the outcome of its branch 1.[64] The outcome of branch remains same in different iterations of outer loop. A, Program 1; B, Branch 1 iteration space



```
// A: a matrix, B: right hand side
//X & X0: current & partial solution
for ( i = 0; i < N; i++ ){//Loop3
  X0[i] = B[i];
for ( j = 0; j < N; j++ ) //Loop4
 if ( j != i ) // Branch 2
  X0[i]=X0[i]-A[i + j*n]*X[j];
X0[i] = X0[i] /A[i + i*n];
}
```

(A)

Inner loop (Loop4) iterations

Outer loop (Loop3) iterations

1	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0

(B)

FIGURE 20 (A) Program 2 and (B) the outcome of its branch 2.[64] The branch outcome shows diagonal pattern. A, Program 2: Jacobi1 algorithm; B, Branch 2 iteration space

its outcome shows diagonal pattern, which is shown in Figure 20B. For such a branch, the outcome in first iteration is noted. Then, for the second iteration, "100" is taken from the first iteration along with "0" in the current iteration to form the history "0100." Based on this, next outcome of the branch is predicted as "1," as shown in Figure 20B.

The outcome of their "wormhole" BP overrides that of ISL-TAGE if the confidence in the prediction is high. Their BP is useful for applications that spend large fraction of time in nested loops. As a side predictor, their BP reduces the mispredictions significantly.

Seznec et al[63] present a technique to track branch correlations present in nested loops using "inner-most loop iteration" (IMLI) counter–based BPs. IMLI counter is the iteration-number of the loop that encapsulates the branch. IMLI-count is the number of times a backward conditional branch (ie, a loop-exit branch) has been successively taken. Using this, IMLI-count can be found at fetch time. They propose two components, which use this information. For some hard-to-predict branches, the inner-most iteration count is tested in the inner loop, and hence, their behavior depends only on the inner-most loop counter, ie, `Outcome[i][j] = Outcome[i-1][j]`. To capture such "same iteration correlation," their first component uses a table which is indexed with IMLI counter and PC. While wormhole predictor[64] tracks only those correlations existing in regular loops with fixed iteration counts and executed on every iteration of inner loop, their BP does not have this limitation.

However, wormhole BP can predict correlation between `Outcome[i][j]` and `Outcome[i-1][j-1]` (ie, diagonal pattern), which cannot be captured by this first component. To address this issue, their second component uses a table to store branch outcomes such that while predicting for `Outcome[i][j]`, both `Outcome[i-1][j]` and `Outcome[i-1][j-1]` can be retrieved. Their IMLI-based BP components can be used with a global history-based BP, and on using them, further benefit from using local history becomes small. Further, the speculative state of their proposed BP is easily manageable, and using their BP as a side predictor with any TAGE or neural based BP provides high accuracy.

5.3 | Predicting branches with long period

Kampe et al[59] note that several branches occur with periods much higher than the number of history bits generally used in BPs (eg, 32 bits), and to record the entire period of all branch execution patterns for making a prediction, 2^{13} bits per branch are required. By translating the history from time-domain to frequency domain, the size of history register can be significantly reduced, eg, only 52 bits are required for representing a history pattern of 2^{13} bits. Based on this, they propose a discrete Fourier transform (DFT) based side-BP for predicting branches with long periods that do not correlate with other branches. Their BP is used in a hybrid BP with 4 other BPs, viz., a static BP, the 2-bit dynamic BP,[9] and the PAp and GAp dynamic BPs,[19] Due to the latency-overhead of DFT, they apply it to a branch only if its accuracy with any of the four above-mentioned BPs is less than 99.99%.

Using 1 and 0 to represent taken and not-taken branches (respectively), they first transform the branch history pattern into frequency domain using DFT (Figure 21A →21B). Then, only frequency components with largest peaks are kept due to their high contribution to the total probability. Remaining components are removed as shown in Figure 21C. This filters noise (non-periodic events) from the history, and this is another advantage of their technique. Then, the time domain equivalent of unfiltered high peaks is utilized for predicting the original branch pattern. Then, all the frequency components ($A_n sin(\omega_n t)$) are added, and if the sum is higher than a threshold (Figure 21D), the branch at time *t* is predicted to be taken (Figure 21E). IFFT refers to adding the frequency components. The limitation of their BP is that it does not show high accuracy for branches showing large difference from a 50%-50% not-taken/taken rate since such patterns require many frequency signals for accurate reconstruction. On using only one frequency signal, the taken rate becomes much higher than that of the actual branch due to the width of sine wave for the threshold used.

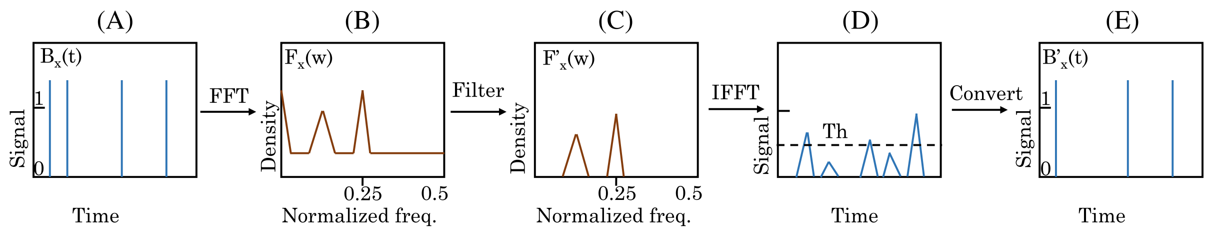


FIGURE 21 Fourier transform based BP (trans. = transformed, Th = threshold, freq. = frequency).[59] A, Original signal; B, Fourier trans. signal; C, Filtered signal; D, Distorted signal; E, Predicted signal

In practice, this limitation does not have large impact since such branches are already well-predicted by other BPs. When used as a component of hybrid BP, their BP reduces misprediction rate significantly.

Note that both storing the history in frequency-domain[59] and using *geometric* history length tables[23,30] are solutions for branches with large period. The former method focuses on branches requiring very long history, whereas the latter method focuses on branches requiring short- and medium-long histories.

5.4 | Predicting based on data correlation

Heil et al[65] present a BP which predicts based on the correlation of data values just as traditional BPs correlate on GBH. For example, consider the loop in Figure 22A, which executes for 21 iterations on average for a workload. This is much higher than the length of local or global BHR. Further, the loop-count varies depending on the value of `len`, and hence, the loop predictors based on the assumption of a fixed loop-bound (eg, see the work of Sherwood et al[76]) generally mispredict the loop-exit branch. However, since the loop-counter (`xlen`) reaches a specific value (0), using loop counter as an input to BP allows accurate prediction of this branch. Since several branch instructions compare two register values and check the sign or detect equal value, delta values correlate strongly with the branch result.[96] Hence, they store the delta between the source register values instead of the values themselves, which also saves space. The deltas for each static branch are stored in value history table, and for this reason, their BP is termed as "branch difference predictor." Figure 22B shows the design of their BP.

However, even with the deltas, the number of patterns becomes large. In fact, one static branch instruction may generate several values, of which only few branch deltas are prominent. Hence, they use a backing predictor which speculates most cases without using delta values. For predicting the remaining hard-to-predict rare branches, they a use "rare event predictor" which has tagged cache-like design. The "rare event predictor" preferentially replaces patterns leading to correct predictions that differ from the outcome of backing predictor. The "rare event predictor" is updated only on a misprediction in the backing predictor. The backing predictor is updated only when it makes a prediction, and this reduces aliasing and improves correlations.

Chen et al[78] present a value-based BP design that works by tracking dependent-chain of instructions. They note that if every register value associated with resolution of a branch in the data dependence chain has same value as in the previous instance, the branch-result will be the same. However, since all the branch register values required for making such value-based prediction are generally not available in-time, use of register values in the dependence chain is highly useful. The branch behavior on a specific path usually remains consistent, and this can be easily learnt by a 2-bit saturating counter. They use both PC and register ID to get the index into the table. Further, to distinguish instances of same path having different register values, they use hashed value of registers as the tag. Thus, their technique uses both value and path-based information for classifying branches. In deeply-pipelined superscalar processors, dependence-chains may span multiple loop iterations, and if dependent-registers remain same in every

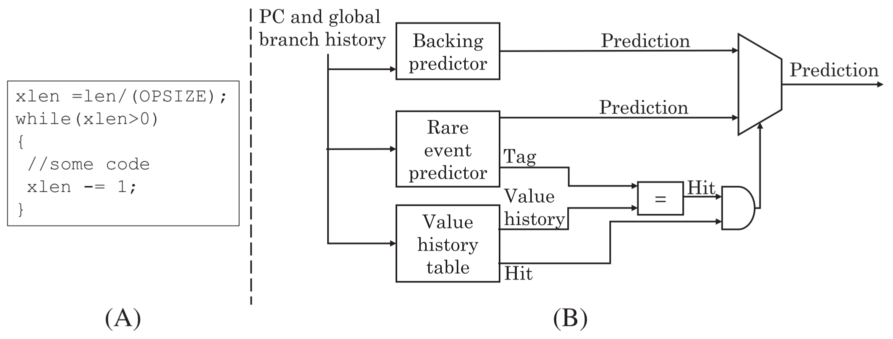

```
xlen =len/(OPSIZE);
while(xlen>0)
{
 //some code
 xlen -= 1;
}
```

FIGURE 22 (A) A loop that runs variable number of times. (B) Branch difference predictor,[65] which stores delta between the source register values for each branch in value history table. For predicting remaining values, rare-event predictor is used. A, Code showing a loop with variable loop-bound (depending on `len`); B, Branch difference predictor iteration, the path information becomes ambiguous. To avoid ambiguity between iterations, their BP uses maximum number of instructions spanned by dependence chain in the tag. Their BP provides higher accuracy than an iso-sized hybrid BP.

Thomas et al[97] present a technique which uses runtime dataflow information to ascertain the "influencer branches" for improving BP accuracy. If a branch B_i decides whether certain operations directly affecting the source operands of an upcoming dynamic branch B_0 get executed, then B_i becomes influencer branch for B_0. For instance, in Figure 23A, assume the control is flowing through the shaded path, and we want to predict branch B8. B8 depends on registers R2 and R3. R2 value is generated in BB3, and it takes R1 value generated in BB2. The R3 value consumed by B8 is generated in BB7. Thus, BB2, BB3, and BB7 are influencer BBs for B8. Also, the branches that determined the flow through these BBs are B0, B2, B5, and hence, these branches are called influencer branches for B8.

Influencer branches show stronger correlation with their influenced branch than rest of the branches in the history, and this is confirmed by comparison with the weights of a perceptron BP.[51] They track influencer branches for the latest instructions writing into each architectural register and store this information in an "influencer register file." Figure 23B shows the design of "influencer register file," which has one entry for every architectural register. For a conditional branch, the influencer branch details are inherited from producers of its source operand. For a conditional branch, the "influencer register file" entries of its source registers are read and OR'ed to obtain "influencer branch bitmap" (IBB).

They propose two techniques for using this information for making prediction. In the "Zeroing scheme," shown in Figure 24A, all non-influencer bits in the GHR are masked by ANDing them with IBB. Thus, influencer branches retain their positions in the global history, whereas non-influencer branches are shown as zero. Then, by using fold and XOR operations, masked GHR value is hashed down to the number of bits required for the predictor index. In the "Packing scheme," the only difference with "Zeroing scheme" is that after masking the non-influencer bits, they are removed completely, which compacts the subsequent influencers. This scheme in shown in Figure 24B.

In the overall design, the line BP provides a 1-cycle prediction for fetching the next instruction. The predictions of primary BP and corrector BP are obtained later one-after-another, and they override respective previous predictors in case of disagreement; since by virtue of using long global history, corrector BP can make more accurate predictions. Corrector BP is modified form of the "rare event predictor," and it predicts only for

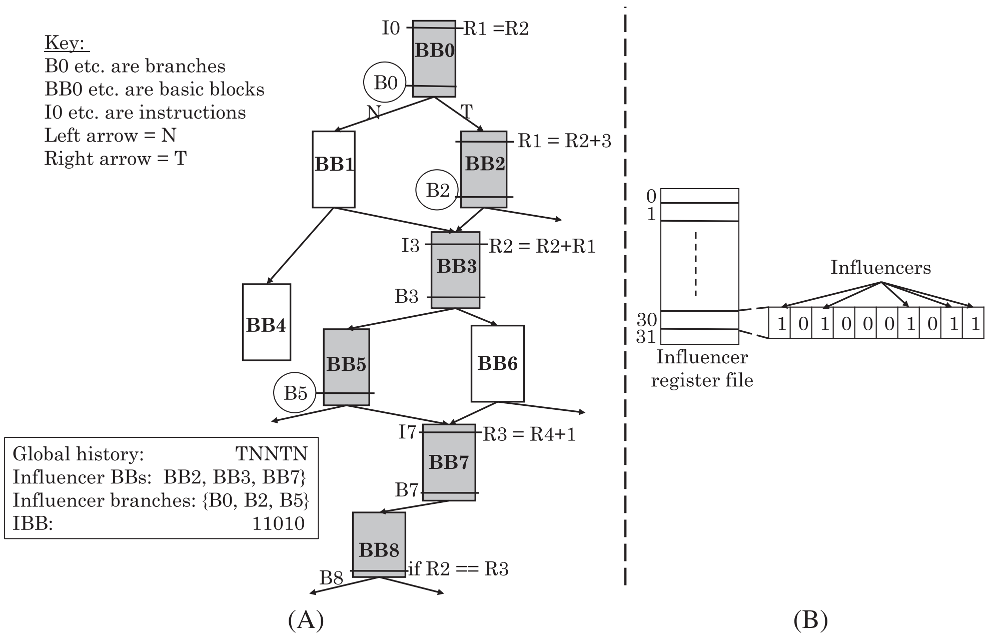


FIGURE 23 (A) An example control-flow graph (B) design of "influencer register file," which stores influencer branches for the latest instructions writing into each architectural register.[97] A, An example control-flow graph to show influencer branches; B, Design of "influence register file"

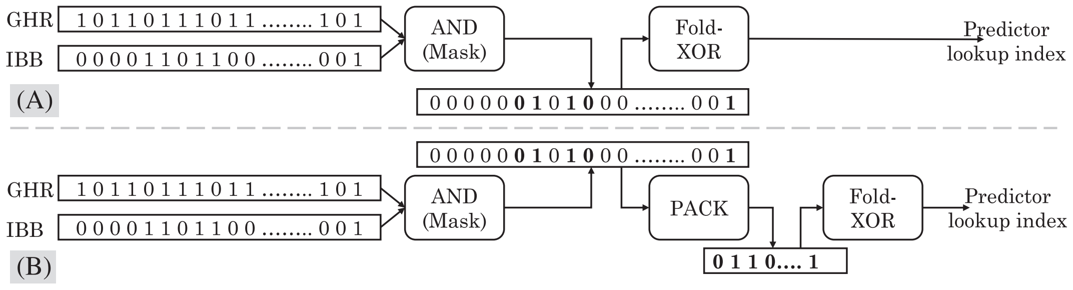
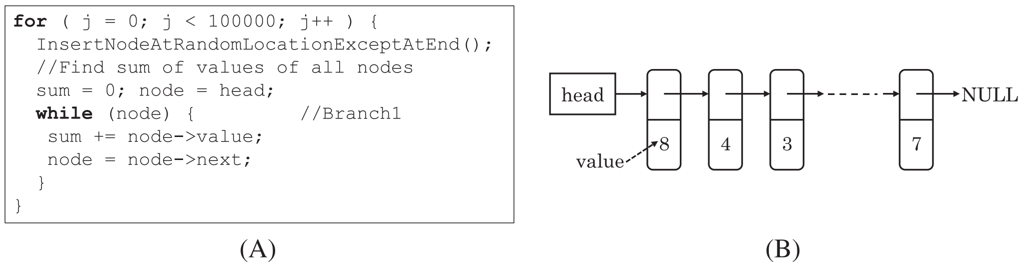


FIGURE 24 Use of influencing branch information for prediction in (A) Zeroing and (B) Packing schemes.[97] A, Zeroing scheme; B, Packing scheme



```
for ( j = 0; j < 100000; j++ ) {
  InsertNodeAtRandomLocationExceptAtEnd();
  //Find sum of values of all nodes
  sum = 0; node = head;
  while (node) {         //Branch1
   sum += node->value;
   node = node->next;
  }
}
```

FIGURE 25 For the case of traversing a linked-list, as long as the last node does not change, the outcome of branch 1 can be predicted simply based on the load address, without requiring the actual value. This is an example of address branch correlation.[44] A, Code showing address-branch correlation; B, Illustration of a linked list

influencer histories that can consistently correct the mispredictions of primary BP.[65] Using their technique with a perceptron BP improves accuracy with much less hardware cost than increasing the size of perceptron BP.

5.5 | Predicting based on address correlation

Gao et al[44] note that for branches depending on long-latency cache misses, if loaded values show irregular pattern, BPs based on leveraging branch history correlation show poor accuracy. They propose leveraging address-branch correlation to improve accuracy of such branches. For several programs, especially memory-intensive programs with significant pointer-chasing, addresses of key data structures do not change for a long duration. For example, for the code shown in Figure 25, the address of last node in a linked list is likely to remain same until another node is appended to it. Thus, the outcome of a branch depending on the end of the list can be determined based on the load address itself without requiring the actual value. This allows much faster resolution of the branch since the load address can be ascertained much before the value. By focusing on only few (eg, 4) branches showing address-branch correlation, most of the branch misprediction latency can be alleviated. Using their 9 KB BP as a side predictor with a 16 KB TAGE predictor provides better performance and energy efficiency than using a 64 KB TAGE predictor. The limitation of their BP is that it shows poor accuracy for codes that frequently append to the linked list.

6 | HYBRID BPS

In this section, we discuss hybrid BP designs. Table 6 highlights essential ideas of several hybrid BPs.

6.1 | Tagged hybrid BPs

Eden et al[36] propose a BP which is hybrid of gshare and bimodal BPs. Their technique divides the PHT into two branch streams corresponding to taken and not-taken bias and stores them in choice PHTs, as shown in Figure 26. Further, in direction PHTs, it stores the occurrences which do not agree with the bias. This lowers the information content in direction PHTs and allows reducing its size below that of choice PHT. To identify

TABLE 6 Key ideas of hybrid BPs

Number of Predictors or Tables that the Hybrid BP Allows to Choose From	
Two	[34,58,83,84]
Three	[34]
Arbitrary number	[13,23,30,92]
Accessing Choice Predictor using	
Branch address only	[83]
Both branch address and global history	[34]
From the Predictions of Multiple Components, the Single Final Prediction can be Chosen Based on	
Bias of each branch	[1,67,83]
Using choice predictors	[18,34]
Majority voting	[57,71,89]
Giving preference to	tagged predictor with longest history,[23] to predictor having highest confidence,[13] to the critic BP in prophet-critic hybrid BP design[58] and to the larger BP in overriding BP designs
Combining strategies	Adding[30] or fusing[71] the predictions of the components to obtain final prediction

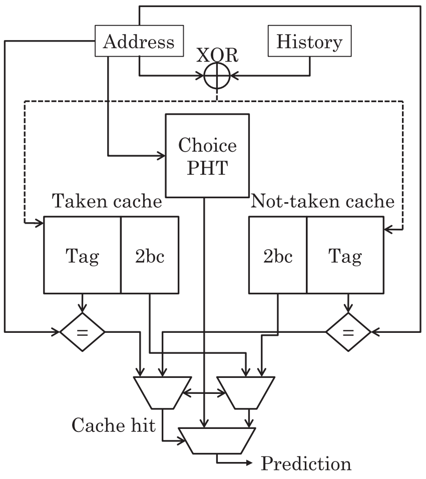


FIGURE 26 The tagged BP proposed by Eden et al,[36] which is hybrid of gshare and bimodal BPs

such occurrences in direction PHTs, they store 6-8 least significant bits of the branch address as the tags in every entry. These narrow tags almost completely remove aliasing, especially after context-switching.

For any branch, first, the choice PHT is consulted, and if it indicates “not-taken,” then the “taken” direction PHT (also called a cache due to use of the tags) is referenced to see whether this is a special case where the bias and the prediction disagree. On a hit or miss in the taken cache, its own outcome or that of choice PHT (respectively) is used as the prediction. Converse action is taken if the PHT access for a branch indicates “taken.” The addressing and updating of choice PHT happens as in the bimodal choice PHT.[83] The “taken” cache is updated if its prediction was used or if the choice PHT indicates “not-taken,” but the branch result was taken. Converse is true for updating the “not-taken” cache.

To remove aliasing for branch occurrences that do not agree with the bias of the branch, they design direction caches as two-way set-associative cache. They use LRU replacement policy, except that an entry in “not-taken” cache indicating “taken” is preferentially replaced for removing redundant information since this information is also available in the choice PHT. Their technique achieves high accuracy for the same reason as the bimodal BP and due to the use of tags. The limitation of their technique is that it provides only small improvement over bimodal predictor.[83] The reason for this is that, although use of tags reduces branch mispredictions by virtue of reducing aliasing, not all mispredictions are due to aliasing. Hence, introduction of tags provides only limited benefits. For the same reason, increasing the tag size beyond a threshold (eg, 6 or 8 bits) does not improve accuracy.

6.2 | Selection-based hybrid BPs

Chang et al[67] note that since both biased and non-biased branches appear frequently, a single BP cannot provide high overall accuracy. They propose and evaluate multiple hybrid BP designs (1) a GAs BP where short and long BHR is used for biased and non-biased branches, respectively, (2) using profile-guided BP for biased branches and gshare BP[84] for non-biased branches (3) 2bc+gshare (4) PAs+gshare (5) profile-guided BP for biased branches and PAs+gshare hybrid for non-biased branches. *Results:* These hybrid BPs provide higher accuracy than solo-BPs. Also, using static BPs for biased branches allows dedicating higher storage budget to dynamic BPs.

Seznec et al[34] design two hybrid BPs, namely 2bc+gskew and 2bc+gskew+pskew. 2bc+gskew has 4 predictor banks: 3 banks of e-gskew[89] and a meta predictor (refer Figure 27A). Bimodal predictor is already part of e-gskew. Meta predictor is accessed using a combination of branch address and global history. They use partial update policy whereby the three banks of e-gskew are updated on an incorrect prediction. Also, for a correct prediction, if the prediction was provided by the bimodal BP, only this is updated, but if it was provided by the e-gskew, only the banks that provided the prediction are updated. The meta predictor is updated only in case of disagreement between the two predictors. *Results:* 2bc+gskew provides higher accuracy than e-gskew since bimodal BP provides large fraction of predictions due to which other two banks are not updated and thus not polluted by branches not requiring the use of GHR for accurate prediction. Also, in most cases, bimodal and e-gskew BPs agree, and hence, meta predictor is neither required nor updated, and hence, aliasing in meta predictor does not harm accuracy.

In 2bc+gskew+pskew BP shown in Figure 27B, e-pskew component is the per-address history BP for accurately predicting branches that benefit from per-address history instead of global history; thus, 2bc+gskew+pskew has three constituents: bimodal, per-address history, and global history. Due to this, 2bc+gskew+pskew BP provides higher accuracy than 2bc+gskew BP, although it also incurs higher latency.

Seznec et al[1] present design of BP in Alpha EV8 processor, which takes 352 Kb space. Alpha EV8 fetches up to two, 8-instruction blocks in each cycle; thus, up to 16 branches may need to be predicted in each cycle. Hence, they use a global-history BP since a local history BP would require high resources (eg, 16-ported predictor table) and have high complexity (eg, competition for predictor entries from different threads). Their BP is based on 2bc+gskew hybrid BP.[34] In this hybrid BP, biased branches are precisely predicted by the bimodal BP, and hence, for such branches, other tables are not updated. This partial update scheme avoids aliasing due to biased branches and also simplifies implementation. This also allows separating

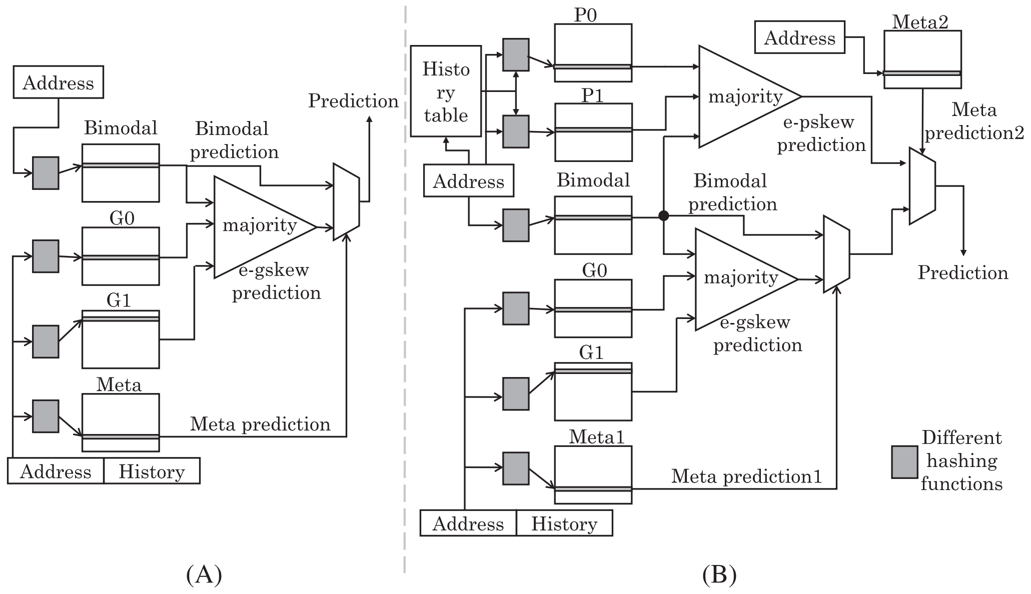


FIGURE 27 (A) 2bc+gskew and (B) 2bc+gskew+pskew hybrid BPs[34]

predictor and hysteresis arrays, and to meet area budget, hysteresis array for meta and G1 predictors (refer Figure 27A) are reduced to half the size. Use of partial update reduces the writes to hysteresis array, which lowers the aliasing effect. Also, they use larger history length for G1 than G0 and smaller predictor table for bimodal than for G0, G1, and meta predictors.

The BP is implemented as a 4-way bank-interleaved with single-ported memory cells, and bank conflicts are completely avoided by computing bank numbers in a way to ensure that two consecutive fetch blocks access different banks. They show that despite implementation constraints, the accuracy of their BP is comparable to other iso-size global history-based BPs.

6.3 | Fusion-based hybrid BPs

Loh et al[71] note that hybrid BPs that select one constituent BP ignore the information from unselected BPs. They propose a BP fusion approach that makes final prediction based on information from all constituents. These approaches are shown in Figures 28A and 28B, respectively.

Their proposed predictor has *N* constituent BPs along with a "vector of mapping tables" (VMT) for mapping their predictions to the final prediction. Figure 29 shows their proposed BP. Each entry of VMT is a 2^N entry mapping table, whose entries are saturating counters. Their BP first looks-up constituent predictors and, in parallel, selects one mapping table from VMT based on branch history and address bits (step 1). Then, based on individual predictions, one of the 2^N counters of selected mapping table is chosen (step 2). The MSB of this counter provides the final prediction. These counters are incremented or decremented based on whether the actual outcome was taken or not-taken, respectively. The constituent predictors of their BP are chosen using a genetic search approach with the constraint of a fixed area budget. Their BP provides high accuracy, although it also has a large lookup latency (4 cycles).

6.4 | Multi-hybrid BP designs

Evers et al[13] note that by combining more than two BPs, even higher number of branches can be accurately predicted. For each solo-BP, they use a 2-bit counter. The counter is initialized to the value three. The outcome of BP, whose corresponding counter has a value of three, is finally chosen, and ties are broken using a fixed precedence order. As for update, if one of the BPs that had the value three in its counter is found to be correct, the counters for remaining incorrect BPs are decreased by one. Otherwise, the counters for all the correct BPs are increased by one. This ensures that at least one of the counters will have a value of three. Compared to saturating counters, their update strategy can more accurately identify which solo-BPs are presently more accurate for different branches. They use variations of both per-address and global two-level solo-BPs as constituents of their hybrid BP. These solo-BPs have high accuracy by virtue of using a long history. However, due to this, they also have large warm-up time. Since static BPs and small-history dynamic BPs require no and short warm-up periods, respectively, they are also included in the hybrid BP. Their inclusion allows the hybrid BP to show high accuracy during warm-up period. Overall, their hybrid BP provides high accuracy.

6.5 | Prophet-critic hybrid BP

Falcon et al[58] propose a technique which uses two BPs called prophet and critic, as shown in Figure 30A. The prophet makes prediction based on current branch history and goes on the predicted path, making further predictions which form "branch future" for the original branch. Based on this

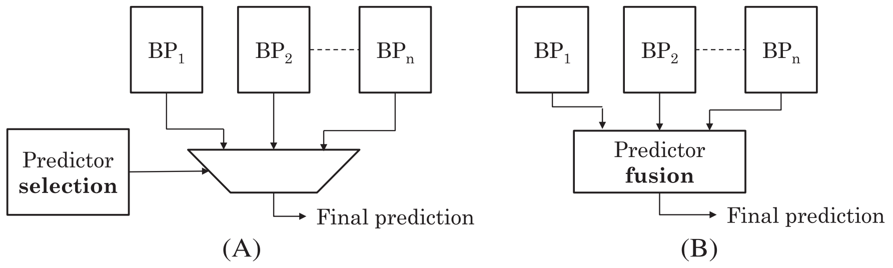


FIGURE 28 (A) Selection of one BP from *N* BPs. (B) Combining information from all *N* BPs.[71] A, Predictor selection; B, Predictor fusion

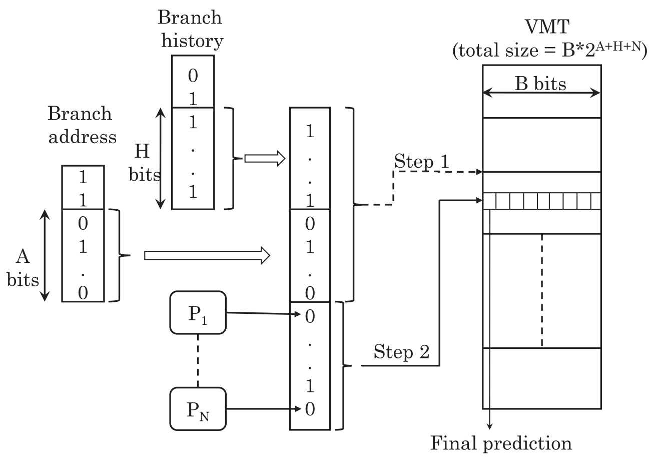


FIGURE 29 Combining predictions of all constituent BPs to arrive at the final prediction.[71] Each entry of mapping table is a *B*-bit counter

information, the critic correlates with both past and future and generates an agree/disagree critique of each prediction of the prophet. The critic's prediction is the ultimate prediction for the branch, since by using future code behavior, it makes more accurate predictions.

For example, in Figure 30B, the correct path is shown by the shaded blocks (A, C, G, I). The history of A is formed by the results of its previous branches: V, W, X, Y, and Z, and this history is used by the prophet to predict A. If all the branches are correctly predicted, branches A, C, G, and I (ie, outcome NNTT where N = not-taken and T = taken) form the future of A. However, if A is mispredicted, the execution proceeds on wrong path shown by red dashed line. Here, the prophet predicts branches A, B, D, and H, and their predictions are stored in branch result register of the critic. Thus, the critic has both branch history (result of V, W, X, Y, Z) and branch future (result of A, B, D, and H).

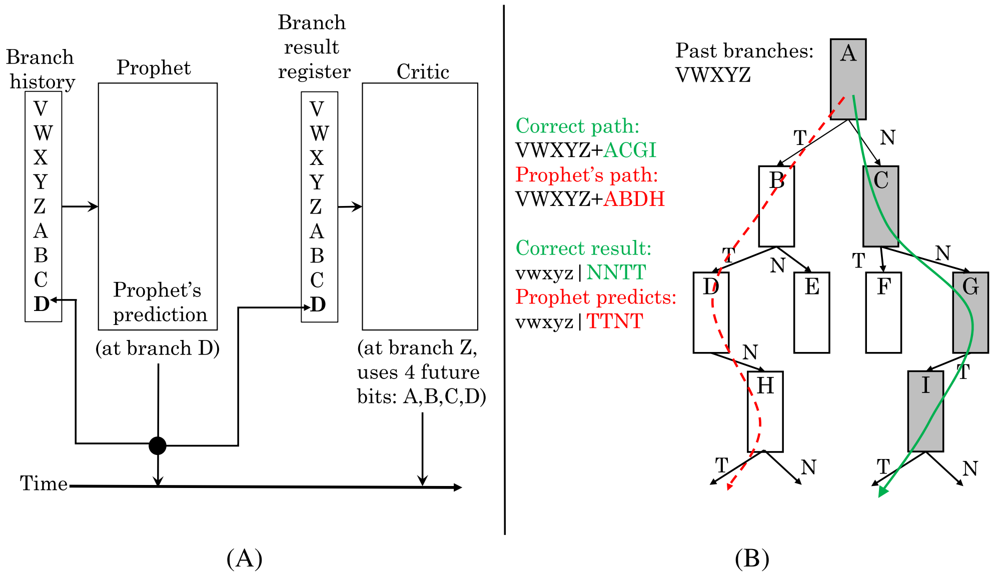


FIGURE 30 (A) Design of and (B) rationale behind prophet-critic predictor.[58] The prophet predicts based on current branch history and goes on the predicted path, making future predictions for the original branch. The critic generates an agree/disagree critique of each prediction of the prophet by correlating with both past and future of a branch. Due to this, it makes more accurate predictions than the prophet. A, Prophet/critic hybrid predictor; B, An example of why critic works

TABLE 7 Strategies for reducing aliasing

Use of Tag	2,30,36,37,65,76,78,91,95
Intelligent indexing functions	91
Using multiple indexing functions	50,89
XORing branch address with branch history	84
Converting harmful interference to beneficial or harmless ones	22
Storing only most recent instance of a branch in BP	20
Storing branches with different biases separately	36,83
Separate BPs for user and kernel branches	74

If for a branch *B*, the prophet makes prediction after observing *P* extra branches, the number of future bits used by prophet are said to be *P*, eg, in Figure 30, $P = 4$. With increasing *P*, prophet's accuracy increases since it can detect more progress along the wrong path. When prophet mispredicts a branch for the first time, critic learns this event, and when prophet mispredicts the branch again under same situation, critic disagrees with the prediction of the prophet. Prophet and critic BPs work independently, and any existing BP can be used for them. To improve performance, prophet should correctly predict more than 90% of branches, and critic is responsible for predicting only the remaining branches. In both traditional hybrid and overriding BPs, component BPs predict the same branch based on the same information; however, in their BP, two predictions happen at different times. Their technique provides large improvement in BP accuracy and performance.

7 | TECHNIQUES FOR IMPROVING BP ACCURACY

In this section, we discuss several approaches for improving BP accuracy such as dynamically adapting history or path-length (Section 7.1), reducing aliasing (Sections 7.2-7.4), improving accuracy in presence of multithreading and short-lived threads (Section 7.5), and learning from wrong-path execution (Section 7.6). Table 7 summarizes several strategies for reducing aliasing.

7.1 | Adapting history-length and path-length

Juan et al[16] note that using constant history length for all the branches does not perform well since the optimum history length depends on input data, code features, and context-switch frequency. They propose dynamically varying the number of history bits used for each application and input data in two-level BPs. For gshare BP,[84] in different execution intervals, they XOR different number of history bits with PC of branch instruction. The interval is determined by the fixed number of dynamic branches. The number of mispredictions with each history length is recorded in a table. After each interval, if the number of mispredictions in existing interval is higher than the smallest number in the table, history length is altered (increased or decreased) by one to move towards the optimum history length; otherwise, it is kept the same. The limitation of their scheme is that after a change in history length, a large portion of PHT state is lost and needs to be generated again. This leads to aliasing and increases mispredictions. To offset its impact, after a history length change, misprediction counters are not updated for one interval. Their technique approaches the accuracy obtained using fixed optimum history length in both absence and presence of context-switching. Their technique can be used with any BP that combines global branch history with PC to find the PHT-index, eg, agree,[22] bimodal,[83] gselect,[84] gshare[84] and e-gskew[89] BPs.

Stark et al[70] note that in path-based BPs, *K* most recent target addresses are hashed to obtain the table index for making a prediction. They propose a 2-level BP, which allows using different path length (*K*) for each branch to achieve high accuracy. The destination addresses of most recent *K* (eg, $K = 32$) branches form level-1 history, as shown in Figure 31. The lower *L* bits of these addresses are stored in the "target history buffer." Let A_j refer to j^{th} most-recent address stored in the "target history buffer." Only those destination addresses that provide useful information about the path leading to the candidate branch are stored in the "target history buffer." The *L*-bit index obtained from level-1 table is used for accessing level-2

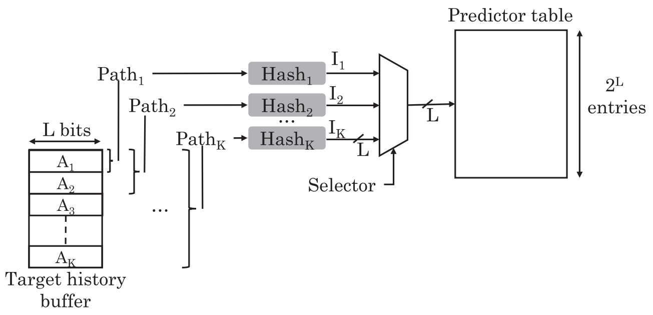


FIGURE 31 The BP proposed by Stark et al,[70] which allows using different path length (*K*) for each branch

history stored in a "predictor table." Each predictor table entry uses a 2-bit saturating counter. A path of length j ($PATH_j$) has latest j addresses in the "target history buffer"; thus, $PATH_j$ has A_1 to A_j addresses. K predictor table indices are obtained using K distinct hash functions ($HASH_j$). For example, $HASH_3$ uses A_1, A_2, and A_3 to provide I_3. The hash function rotates jth destination address $j-1$ times and then XORs all the addresses in the path. Rotating the address allows remembering the order in which the addresses were seen. For every branch, based on profiling information, a hash function is chosen that provides highest accuracy for that branch. The hash function can also be chosen dynamically. Their BP reduces training time and aliasing and hence improves accuracy.

7.2 | Filtering branch history

While increasing the history length generally improves accuracy by allowing exploitation of correlation with the branches in distant past, inclusion of uncorrelated branches can introduce noise in the history. Several techniques seek to reduce this noise by filtering the branch history.

Porter et al[79] note that for code zones with limited branch correlations, additional information stored in GHR harm accuracy of this and other branches by increasing the predictor training time and aliasing. Figure 32 shows part of the code from a function in `gcc` benchmark. Branch1 is the first branch in a function and it checks for null pointer. In the execution of 100M instruction, the branch is executed 2455 times but is never taken. The branch sees 208 different 16-bit histories and hence is mipredicted 9% of times. To improve accuracy in such scenarios, they propose temporarily removing unbeneficial correlations from GHR to insulate poorly correlated branches from the useless correlations while still allowing highly correlated branches to benefit from large histories.

They identify code zones with limited branch correlations using control flow instructions, eg, backward branches and function calls and returns. For example, the not-taken path of a backward branch generally shows a loop exit, and the branches after a loop are expected to have limited correlation with those in the loop. Similarly, branches in a function have limited correlation with those of the preceding and following code zone. On detecting code zones with limited branch correlations, their technique resets GHR similar to the approach of Choi et al.[98] For instance, on function calls, the PC of the calling instruction is stored in GHR, and on function return, the PC of return instruction is stored in GHR. For the example shown in Figure 32, using this approach reduces the misprediction rate to just 1%.[79] Their technique can be used with many BPs and especially benefits simple BPs allowing them to become competitive with complex BPs.

Xie et al[41] present a technique to reduce the impact of useless branch correlations (ie, noise) in BPs. They divide the original BP into two sub-predictors: a conventional BP (CBP) and a history-variation BP (VBP). Both BPs work the same, except that their history update mechanism is different: CBP stores branch histories until a branch is committed, whereas VBP uses a history stack to remove branch histories in loops and functions. While entering a loop or a function, the current history is pushed to the history stack, and on exiting the loop, the history is popped back. RAS and loop predictors present in the processors are augmented with history stack, and using them, functions and loops (respectively) are detected. Based on the execution histories of earlier branches, either VBP or CBP is chosen using a selector table.

Figure 33 summarizes the working of their technique. Figure 33A shows a typical source-code. Before entering the function, the GHR in both CBP and VBP are 1110. The VBP pushes or pops the history in the stack while entering or exiting the function, respectively. During code-execution, CBP works same as a traditional BP. Hence, while predicting the branch, CBP and VBP have different histories, and hence, they produce different predictions. The selector decides the final prediction, which, in this case, is "not-taken." Since actual result happens to be "taken," both CBP and VBP are trained towards "taken," and the selector is trained to choose VBP since it provided the correct prediction. They illustrate the use of their approach with several BPs such as gshare, perceptron, and TAGE. Their approach improves BP accuracy significantly.

Huang et al[42] note that aliasing reduces the effectiveness of BPs significantly, eg, hot branches in a loop can flush history of outside-loop branches, and this negates the benefit of maintaining long global history. Different from other BPs that use single global history, they divide global history into multiple groups based on lower-order bits of branch address to restrict aliasing due to hot branches. A branch (instance) can replace another branch from its own group only. By concatenating all history bits in the entries, final history is formed. Their approach allows tracking long correlation history without increasing the hardware cost.

As an example, in Figure 34B, a conventional BP uses 4-bit GHR, and *Ai* to *Fi* refer to the history bits of corresponding branches in the code shown in Figure 34A. Here, once D4 is committed, it is shifted in GHR, and A0 is evicted. By comparison, their technique divides the branches in 4 groups based on the two low-order bits. Only 1-bit history is stored in each group, and hence, the total storage cost is kept same as that in the previous GHR. Here, when D4 is committed, it evicts D3 and not A0, as shown in Figure 34C. Use of their approach with perceptron predictor reduces the misprediction rate significantly. The limitation of their technique is that it limits the local information tracked, and hence, in some cases, it may perform worse than using local and global history.

```
static void sched_analyze_2 (rtx x, rtx insn) {
  register int i, j; register char *fmt;
  if(x ==0)                         //Branch1
     return;
  //more code
}
```

FIGURE 32 An example of code-zone with limited branch correlation[79]

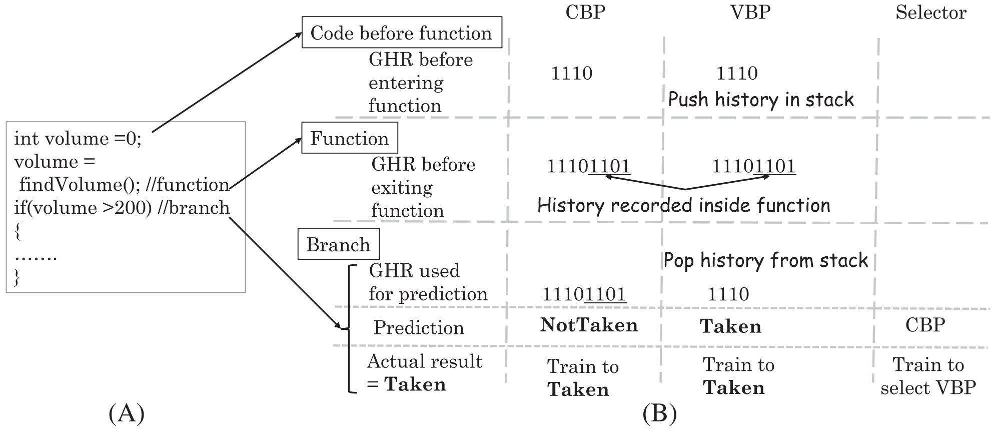


FIGURE 33 Example of working of technique of Xie et al.[41] CBP stores branch histories until a branch is committed, whereas VBP removes branch histories in loops and functions. Based on execution histories of earlier branches, either VBP or CBP is chosen using a selector table. This allows using full or filtered branch history. A, Source-code; B, Reducing noise in global history (Xie et al)

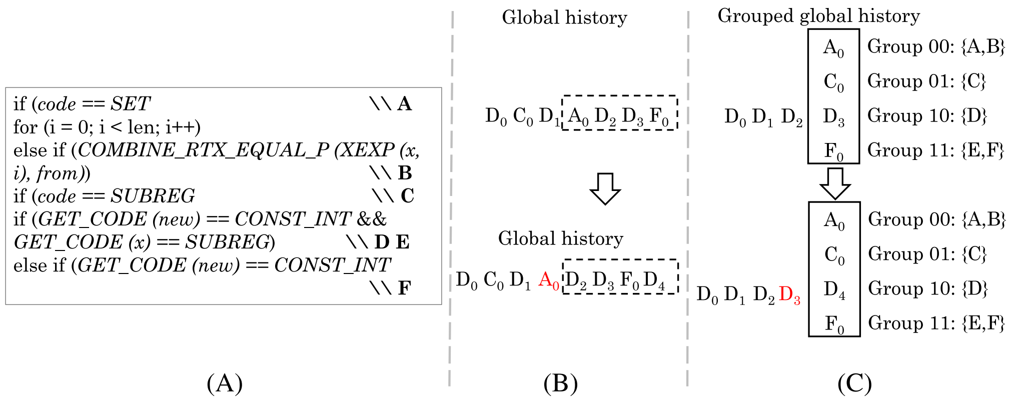


FIGURE 34 (A) selected branches in gcc code. (B) In conventional BPs, a branch replaces another potentially useful branch. (C) The technique of Huang et al[42] reduces aliasing by dividing global history into groups, and thus, a branch replaces another branch from its own group only. A, Selected branches in gcc (SPEC06) code; B, D4 replaces A0; C, D4 replaces history bit in its own group: D3

Gope et al[20] propose storing only non-biased branches in branch correlation history since due to (almost) always having a fixed outcome, biased branches do not impact the prediction of a subsequent non-biased branch. Their technique detects biased nature of a branch dynamically using a finite state machine. A conditional branch is initially in `Not-Present` state. When it is committed for the first time, it moves to `Taken` or `Not-Taken` state. If a branch in one of these states executes in opposite direction, it moves to `Not-biased` state. Unless confirmed to be not-biased, a branch is considered as biased and is not included in the history of upcoming branches.

They further note that multiple instances of repeating branches hardly provide additional value. Hence, their technique uses a recency-stack to record only the most-recent instance of a non-biased branch and tries to find correlation with that instance. This reduces the space taken by a single branch in path history. However, in some cases, different instances of a branch may have different correlations with the latest instance of a branch present in the recency-stack, and to take this into account, they also record the position information, which shows the distance of the branch from the present branch in the global history.

They further apply their bias-removal approach to the perceptron predictor. During training phase, their bias-free preceptron predictor does not work well for strongly biased branches that do not correlate with remote histories. To resolve this issue, they add a traditional perceptron predictor component, which records correlations for few unfiltered history bits to alleviate mispredictions during the training phase. Further, they use their approach to reduce storage overhead of TAGE predictor. Storing *only one instance of non-biased* branches (instead of all instances of all branches) allows finding correlated branches from very remote past (eg, 2000 branches) within a small hardware cost which improves performance.

Comparison: While Gope et al[20] organize the history by the branch instruction address, Huang et al[42] form groups by organizing the history by lower-order bits of the branch PC. Hence, Gope et al use associative search for the branch PC, while Huang et al use table lookup which incurs less overhead. Also, Huang et al can adapt the number of history bits in each group, whereas Gope et al use a fixed number of history bits for each static branch.

7.3 | Reducing aliasing with kernel-mode instructions

Most research works evaluate BPs for user-mode instructions only. Chen et al[73] note that when user-mode instructions account for more than 95% of total instructions, user-only and full-system misprediction rates match closely. However, when user-mode instructions account for less than 90% of total instructions, user-only studies do not reflect full-system results. Hence, the best BP for the user-only branch traces may not be the same as that for the full-system traces. Further, inclusion of kernel branches aggravates aliasing by increasing the number of static branches predicted, which reduces the effective size of BP. The impact of aliasing is higher in two-level BPs with long histories than in BPs using short depth of local history. Furthermore, flushing the branch history at regular intervals, as suggested by Nair et al,[29] does not accurately model the effects of user-kernel interactions since kernel or other processes may not always flush the branch history state. Also, the flushing is especially harmful for BPs with large table sizes.

Li et al[74] note that branch aliasing between user and kernel codes increases mispredictions in both their executions as their branches show different biases. Also, many branches in kernel mode are weakly biased, and hence, they are difficult to predict by most BPs. Further, increasing the predictor size does not alleviate this problem. They propose two techniques to reduce the aliasing. First, they use two branch history registers to separately capture branch correlation information for the user and the kernel code. However, with this technique, aliasing between user and kernel codes can still happen in PHT. To address this, their second technique uses different branch history registers and PHTs for user and kernel mode. Since the number of active branch sites in the kernel code are lower than that in the user code, the size of kernel-PHT is kept smaller than that of user-PHT. The limitation of second design is that for the same hardware budget, it can have lower-sized PHTs than that in the baseline or the first design. Their technique can be used with several BPs and provides large improvement in performance while incurring only small implementation overhead.

7.4 | Using conflict-reducing indexing functions

Ma et al[91] evaluate several indexing functions for reducing aliasing in BP tables, which are shown in Table 8. The symbols used in the table are defined in Figure 35. The BP table has 2^k entries. Note that the prime-modulo mapping does not utilize all the entries. The a-prime-modulo mapping uses all the table entries, although some of the initial entries are used more frequently than the remaining entries. To offset the latency of this mapping, prime-modulo results can be stored in the BTB by extending it. Further, they evaluate use of different indexing function in different BP banks, similar to skewed-BP design.[89] They evaluate these indexing functions with gshare and perceptron BPs and specify exact instantiations of these mapping functions for these BPs.

Results: The modulo mapping function is the baseline function. For gshare BP, XOR indexing generally improves accuracy; however, irreducible polynomial function increases mispredictions, and the prime-modulo function does not consistently provide higher accuracy than simpler mapping functions. The prime-displacement mapping performs better than XOR-mapping for large BP tables but shows lower effectiveness for smaller tables. The workloads having the largest branch working set benefit the most from these indexing functions.

For perceptron BP, all indexing functions reduce aliasing significantly, and their improvement is higher for small tables than for large tables due to higher aliasing in small tables. Different mapping functions perform the best at different table sizes. Specifically, using XOR mapping for large tables and prime-displacement mapping for small tables keeps the complexity low while achieving high accuracy. Even when intelligent indexing functions

TABLE 8 Mappings and their indexing functions.[91] The symbols used are defined in Figure 35

Mapping Name	Indexing Function
Modulo	$index = x = A \, mod \, 2^k$
Bitwise-XOR	$index = x \oplus y$
Irreducible polynomial	Binary representation of index is $R(x) = A(x) \, mod \, P(x)$ and is computed as $R(x) = a_{n-1}R_{n-1}(x) + \dots + a_1R_1(x) + a_0R_0(x)$. Here $R_i(x) = x^i \, mod \, P(x)$ can be precomputed after selection of the irreducible polynomial P(x).
Prime-modulo	$index = A \, mod \, p$. p is a prime closest but smaller than 2^k
A-prime-modulo	$index = A \, mod \, p \, mod \, 2^k$, where p is a prime number which is nearest to but bigger than the table entry count.
Prime-displacement	$index = (T * p + x) \, mod \, 2^k$, where p is a prime number (e.g., 17)

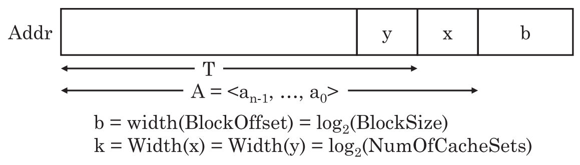

Addr
y
x
b
T
A = <a_{n-1}, …, a_0>
b = width(BlockOffset) = log_2(BlockSize)
k = Width(x) = Width(y) = log_2(NumOfCacheSets)

FIGURE 35 Address subdivision[91] and meaning of symbols used in Table 8

are used, significant amount of aliasing still remains, which can be removed by use of tags. Finally, use of multiple indexing functions is also found to be effective in reducing aliasing.

7.5 | Handling multithreading and short threads

With increasing number of threads, the BP size requirements rise further. Hily et al[99] find that in multithreaded workloads, using a 12-entry RAS with each hardware thread context increases the accuracy of branch prediction significantly and further increasing the number of entries does not provide additional gain. With multiprogrammed workloads (ie, different applications in a workload have no data-sharing), scaling the size of PHT and BTB tables with number of threads removes all aliasing. For parallel workloads with data-sharing, some BPs (viz., gshare and gselect) benefit slightly due to sharing effect, whereas other BPs (eg, 2bc) lose accuracy with increasing number of threads. For both types of workloads, on reducing the BTB sizes, conflicts in BTB increase the number of mispredictions.

Choi et al[98] note that BPs working on large control flow history fail to work well for short threads since GHR does not store thread-specific history information. They propose techniques for predicting or re-creating the expected GHR, using present or historical information. For example, the child thread inherits the GHR of the parent thread. Also, other techniques proposed by them only provide a consistent starting point for the BP whenever the same thread begins. For example, a unique initial GHR state can be provided to each thread by creating the GHR from the PC of the first instruction of the speculative thread. Their technique improves performance by reducing mispredictions for the short threads.

7.6 | Utilizing information from wrong-path execution

Akkary et al[100] note that in case of control independence, branch results on the wrong path match those on the correct path. However, current BPs do not exploit this information to improve accuracy since they study branch correlation on correct-path only. Their technique finds presence of correlation between the execution of a branch on correct and wrong paths using an algorithm similar to gshare BP.[84] In level-1, outcomes of last M branches seen on correct path are stored in a BHR. The address of mispredicted branch is shifted left by $M/2$ bits, as shown in Figure 36. Then, BHR, M bits from the branch on correct-path to be predicted and M bits from shifted mispredicted branch address are XORed to obtain the index into a table of 2-bit saturating counters. If the counter-value exceeds one, correlation of a branch with its wrong-path instance is assumed to exist, and hence, the wrong-path result is finally selected instead of the prediction of the BP.

As for updating the counter, when results on correct and wrong paths match and BP mispredicts, the counter is increased; however, if they mismatch and BP predicts correctly, the counter is decreased; otherwise, no change is made to the counters. Branch results on wrong path are stored in a tagged-buffer. The branch outcome in this buffer is used at most once and then evicted; thus, it needs to accommodate only the branches in the instruction window. Only outcomes of most-recent wrong-path are maintained since remembering outcomes of multiple wrong-paths provides only marginal improvement. Their technique provides large reduction in mispredictions and improvement in performance. The limitation of their technique is that it cannot improve accuracy for branches that have different outcomes on correct and wrong paths.

8 | TECHNIQUES FOR REDUCING BP LATENCY AND ENERGY

We now discuss techniques for reducing BP latency (Sections 8.1-8.4) and storage or energy (Sections 8.5-8.7). Since a decrease in BP accuracy only affects performance and not application correctness, BP energy can be aggressively optimized. Table 9 summarizes the techniques for reducing latency, energy, and storage overhead of BPs. Notice that some works perform partial update of BP to improve accuracy and/or save energy. Some works update BP tables speculatively (and not at retirement stage) for improving accuracy.

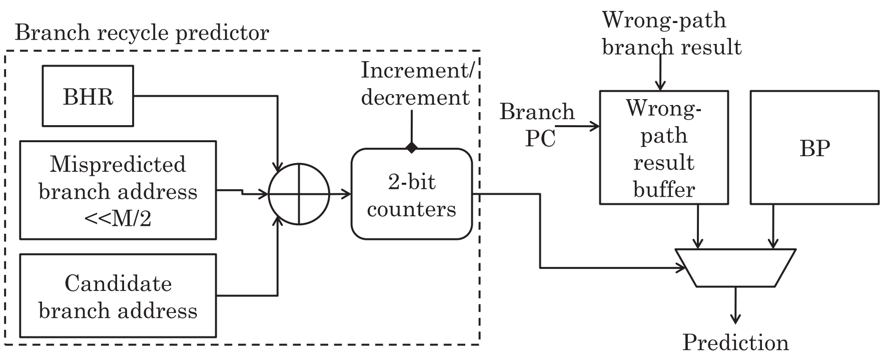


FIGURE 36 The technique for leveraging information on wrong-path.[100] If the executions of branch on correct and wrong paths show correlation, the counter-value exceeds one, and hence, the wrong-path result is finally selected. Otherwise, the result of BP is selected

8.1 BP pipelining

Jimenez[3] proposes pipelining the BP to bring its effective latency to one cycle and demonstrates his approach by pipelining gshare BP. The proposed design uses a 4-stage pipelined BP, 3 stages for reading PHT, and 1 stage for making a prediction. To record recent speculative global history from the beginning of PHT access, the first three stages use a "Branch present" latch and a "New History Bit." The "branch present" latch shows whether fetching and prediction of the branch was done in that cycle, and the "new history bit" shows the corresponding speculative global history bit shifted in from the subsequent stage of pipeline, as shown in Figure 37.

A branch fetched in cycle *T* is predicted as follows: (a) In cycle T-3 at stage 1, GHR is used to start fetching eight two-bit counters into eight-entry PHT buffer. If a branch was fetched in stage 2, its "new history" bit is shifted from stage 2, and "branch present" bit is set; else, "branch present" bit is reset. (b) In cycle T-2 at stage 2 and in cycle T-3 at stage 3, history and branch present bits are sent to previous stage in first half-cycle, and these bits of next stage are shifted in the second half-cycle. (c) Additionally, data read from PHT is placed in PHT buffer at the end of cycle T-1. (d) For a branch fetched in cycle T, lower bits of its address are XORed with lower bits of GHR, shifted left, and merged with at most 3 "new history" bits from earlier stages. From this, the PHT buffer is indexed, and the entry obtained gives the final prediction. By using a larger buffer size, their BP can make

TABLE 9 Techniques for reducing latency and storage overhead

Techniques for Reducing Latency	
Ahead pipelining	5,23,26,30,32,52,54,87
Overriding BP	4,32,101
Caching	4
Cascading lookahead	4
Making multiple predictions in single cycle	3,66,102
Dividing the large table into sub-tables which can be accessed concurrently	10
Accessing slow BPs only for hard-to-predict branches	59
Techniques for Reducing Storage or Energy Overhead	
BP virtualization	77,103
Storing history in modulo-*N* manner	21,54,94
Power-gating based on temporal or spatial locality	48,57,60,104
Avoiding access to BP for	biased branches,[68] branches that have reached their steady-state phase[47] and cache blocks with no control-flow instructions[68]
Partial update policies. Updating only	Selected sub-banks or predictor components,[1,5,34,36,83,89,95] if BP does not have sufficient information for accurate prediction,[47,62] on a misprediction,[65] on a misprediction or when confidence in prediction is low[30]
Speculative update of BP	3,81
Reducing BP size by intelligently managing global history	40-43
Banked-design of BP	68,104
Reducing BP complexity	21
Analog designs	45,46,53
Use of memristor	45,53

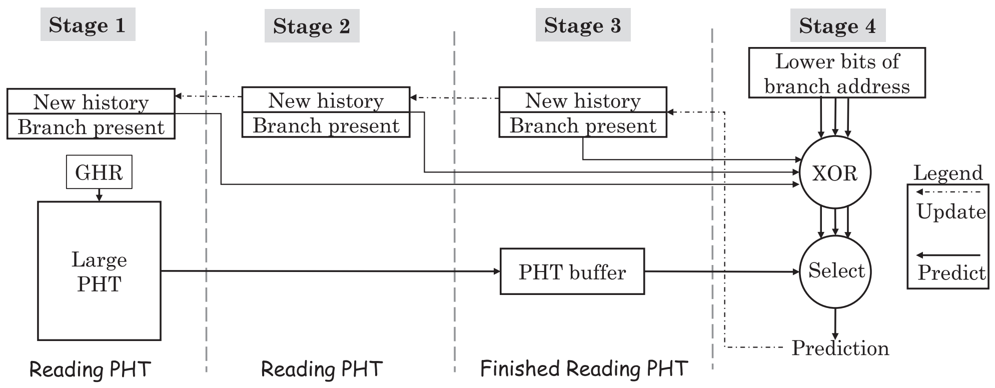


FIGURE 37 Pipelined gshare implementation.[3] Pipelining brings the effective latency of BP to one cycle

multiple predictions in every cycle. Despite having lower accuracy compared to perceptron, 2bc+gskew, and hybrid BPs, their BP provides slightly better performance due to its 1-cycle latency.

8.2 | BP caching and overriding

Jimenez et al[4] present three strategies for improving BP accuracy without increasing its latency. The common key idea of these strategies is to use a small predictor for making fast prediction and a large predictor for boosting accuracy. They illustrate their strategies for gshare BP. In the first strategy shown in Figure 38A, the BP table entries are cached in a small table that can be accessed in one cycle. The output of XOR gate is sent to both PHT-cache and a side predictor. The number of entries in PHT is the number of possible combinations of addresses produced by XOR function. PHT-cache stores a subset of these entries. A hit in PHT-cache provides its prediction, whereas on a miss, the prediction from the side predictor is used. The actual branch outcome updates both PHT and PHT-cache. The effectiveness of this strategy depends on side BP and the ability of PHT-cache to capture branch locality.

The second "cascading lookahead prediction" strategy works on the idea that if the distance between two branches is more than a cycle, then the multiple-cycle latency of BP can be hidden by predicting for the expected future branch. The subsequent prediction is determined by the predicted history and the last predicted branch target. The last prediction is appended to BHR, and the predicted branch target is taken from BTB. This strategy uses tables with increasing size and latency, as shown in Figure 38B. Both tables (PHT1 and PHT2) are simultaneously accessed. If PHT2 provides prediction before arrival of next-branch, its prediction is used; otherwise, that of PHT1 is used. This design can be extended to more than two levels of tables. The effectiveness of this design depends on inter-branch distance, latency of PHT2 and BTB accuracy.

In the third "overriding BP" strategy, two predictions are provided by a fast table (PHT1) and a slow and more-accurate table (PHT2), respectively (refer Figure 38C). Execution proceeds based on the prediction from PHT1; however, if the prediction from PHT2 obtained later differs from that of PHT1, then the prediction of PHT2 overrides that of PHT1 and execution on wrong path is discarded. Pipelining of BP ensures that a branch need not wait until PHT2 access is completed for the earlier branch. *Results:* The overriding BP performs the best, whereas cascading BP performs reasonably well. The cached BP provides no improvement over the original BP since the tags required for the caching strategy incur more storage cost than the cache itself and hence limit effectiveness and capacity of the cache.

Challenges of BP overriding: A limitation of BP overriding is that the speedup from it may not be commensurate with the implementation overheads since in case of disagreement between the fast and the slow BPs, high latency penalty is incurred.[31] Further, BP overriding increases complexity of fetch and recovery engine significantly. The prediction of fast BP needs to be stored and later compared with that of slow BP and in case of disagreement between them, either all or selected (ie, only those that depend on the prediction) instructions on wrong path need to be discarded. The former approach degrades performance and wastes energy due to squashing of large number of instructions, whereas the latter approach requires tracking instructions dependent on the prediction.[105]

One strategy to avoid the need of overriding is to increase the length of prediction unit, eg, predicting instruction stream and instruction traces. This helps in keeping execution engine busy for multiple cycles with initial prediction, and during this time, the second prediction can be generated, which allows hiding the latency of second prediction.[105]

8.3 | BP virtualization

Predictor virtualization approach seeks to increase the effective capacity of predictors without using a single large and slow monolithic design. It records the full BP metadata in a virtual table stored in memory hierarchy (eg, L2 cache) and brings only active entries in a smaller on-chip table. When the working set exceeds the on-chip table capacity, the data is spilled to and brought from the virtual table.

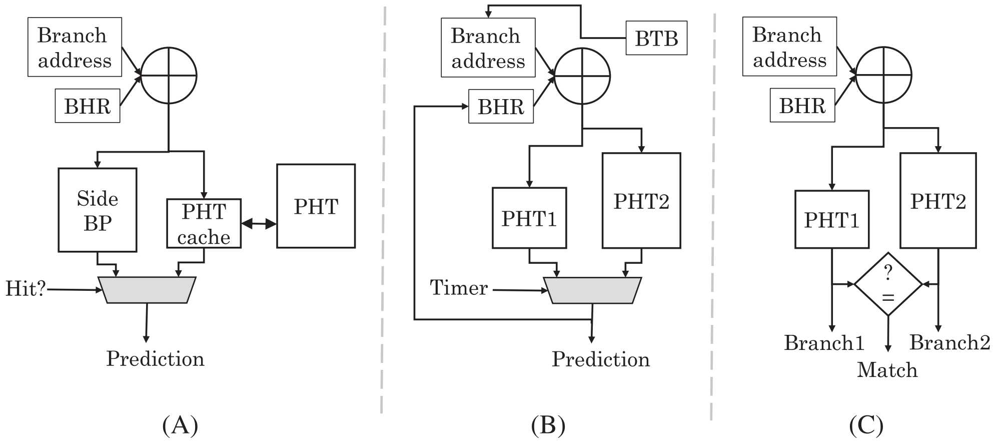


FIGURE 38 Three strategies for mitigating BP latency.[4] A, Caching branch predictor; B, Cascading branch predictor; C, Overriding branch predictor

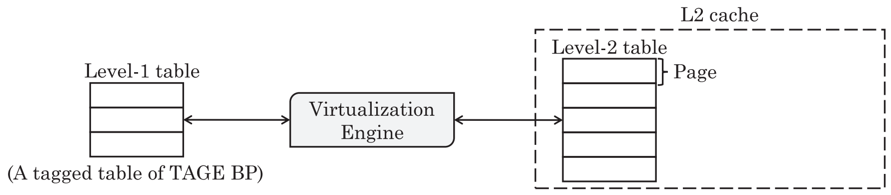


FIGURE 39 BP virtualization approach[103] (figure not drawn to scale). Here, one of the tagged predictor table is augmented with a second-level table, which is stored in L2 cache

Sadooghi et al[103] propose a technique for virtualizing BP. They apply their technique to TAGE multi-level BP. They augment one of the tagged predictor table with a second-level table stored in L2 cache. This is illustrated in Figure 39. To hide the latency of accessing second-level table, multiple correlated entries need to be fetched on each miss. Since higher effective capacity of 2-level table design already reduces aliasing, they lower the randomness of predictor access stream to increase its locality. For this, branch PC bits are directly used as higher portion of the index, and only the lower portion of the index is randomized. Thus, predictor table is partitioned into multiple sub-tables (called 'pages') such that each branch accesses a unique page, as shown in Figure 39. Thus, the stream of pages, which is BP access stream, show temporal and spatial locality already present in instructions.

The level-1 table caches recently-referenced pages of level-2 table. Based on their PCs, branches are assigned to unique pages, which has the disadvantage of increasing the contention. Their technique requires changing only the index hash function. The prediction mechanism needs to access only the first-level table. Swapping of pages between two levels is done by a separate module. Compared to the non-virtualized design, their virtualized BP design reduces storage requirement while maintaining accuracy. The limitation of virtualization is that for applications with poor locality, a high number of swapping operations are introduced.

8.4 | Predicting multiple branches in each cycle

Yeh et al[66] note that being able to predict only one branch and fetch only one set of successive instructions from I-cache in each cycle limits fetch capability to one basic block per cycle. To address this challenge, they propose a BP that can predict multiple branches and fetch multiple non-consecutive BBs in every cycle. For example, if every BB has 5 instructions and two branch paths can be correctly predicted in each cycle, then 10 instructions can be fetched in each cycle. For two branch predictions, they use the terms primary and secondary branches and BBs, as shown in Figure 40A. Their BP is extended from two-level GAg BP[19] such that the prediction is extrapolated to subsequent branches also.

For predicting the primary branch, the index for PHT is obtained from all k bits in the BHR table. For the second branch, right-most $k-1$ bits are used to index two consecutive entries in PHT from which one is selected based on the primary branch prediction (refer Figure 40B). Similarly, for third prediction, $k-2$ bits are used, and four consecutive entries are obtained, of which one is finally selected using primary and secondary predictions. Thus, k-bit branch history information is used by all predictions.

To record the branch addresses, they use a branch address cache, which is accessed using fetch address in parallel with I-cache. The number of fetch addresses used to access this cache is same as the number of BBs fetched. A hit in this cache implies that recently fetched instructions have a branch. The "branch-address cache" provides starting addresses of BBs following the multiple predicted branches. For primary, secondary and

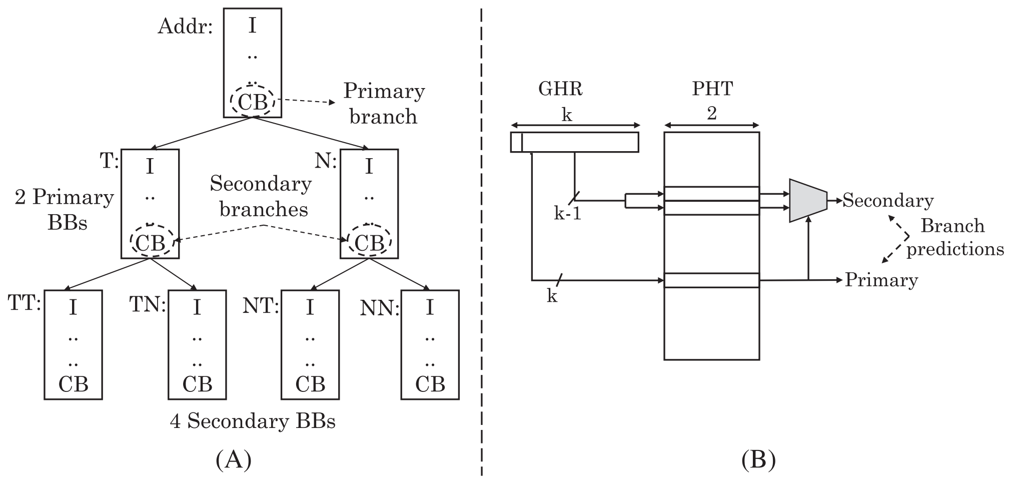


FIGURE 40 (A) Illustration of primary and secondary branch (B) making multiple predictions each cycle (CB = conditional branch).[66] For the second branch, two consecutive entries in PHT are accessed. From these, one is selected based on the primary branch prediction. A, Primary and secondary branches and BBs; B, Making two predictions from a single GHR

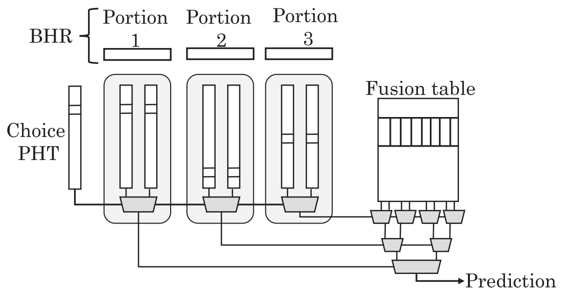


FIGURE 41 Fusion-based hybrid BP.[5] BHR is divided into smaller portions and a separate PHT is used to handle each portion (left side of the Figure). The predictions of each portion are fused using another table-based predictor[71] to arrive at the final prediction (right side of the Figure)

third BB, there are 2, 4, and 8 fetch addresses, respectively. A miss in the "branch-address cache" implies that there is a large BB, and hence, I-cache bandwidth is used entirely for fetching sequential instructions. To allow fetching multiple BBs, I-cache needs to provide low miss-rate and high bandwidth. Hence, they use an I-cache with multiple interleaved banks, which has lower overhead than using multiple ports. Their technique provides large performance improvement, although with increasing number of BBs fetched, the hardware cost of their technique increases exponentially.

8.5 | Reducing PHT size by BHR partitioning

Loh et al[5] note that the number of entries required by PHT-based predictor increases exponentially with history length compared to only linear increase in the neural predictor.[51] They propose dividing the branch history register (BHR) into smaller portions and using a separate PHT to handle each portion as shown in the left side of Figure 41. Each PHT uses bi-modal design to reduce harmful aliasing. Another table-based predictor[71] fuses per-portion predictions to make the final prediction as shown in the right side of Figure 41. To achieve low latency, they use ahead-pipelining[52] for bimodal predictors and prediction fusion scheme, which allows achieving an effective latency of one-cycle. Although dividing the branch history loses correlation across different segments, its impact on accuracy remains small since strong correlations tend to remain clustered, and many correlations are redundant due to which including only few correlations is sufficient. Their BP provides better performance than global-history perceptron BP[51] for all sizes and path-based neural BP[52] at 16 KB or higher sizes.

8.6 | Leveraging branch locality

Hu et al[104] note that the spatial and temporal locality present in applications provides opportunities for saving BP leakage energy by deactivating (decaying) unused entries. They note that a BP with K 2-bit entries is implemented as a square structure of $\sqrt{K}$ rows, each of which connects to $2 \times \sqrt{K}$ bits. Hence, they perform deactivation at the granularity of each row in the array instead of individual entries since deactivating at granularity of 2-bit entries would incur prohibitively high metadata overhead. Periodically, the predictor rows that were not accessed for the duration of one "decay interval" are deactivated since they are unlikely to be reused soon. On access to a deactivated row, the row is activated, and the default state of weakly "not-taken" is used.

As for the reason for presence of spatial locality, since only few neighboring regions of a program code are expected to be active for a short time-duration, the PC of branch access is also expected to vary in a small range. Further, several taken branches of "if-else" statements perform short-jumps only. Code locality leads to spatial locality in BP rows, especially for the BPs indexed only by PC, such as bimodal BP.[83] Hence, consecutive conditional branches are highly likely to fall in the same row. Also, for applications with few static branches, accesses to bimodal BP show high temporal locality since each static branch in the bimodal BP accesses a single BP entry only. In BPs, where PC is XORed with global branch history (eg, gshare), one branch can access multiple BP table entries. Still, accesses to these BPs shows sufficient temporal locality, albeit lower than that in bimodal BP.

They further study a global+local hybrid BP similar to that in Alpha 21264 processor and note that its organization provides additional opportunities for energy saving. Here, the indices of selector and global-history predictor are always the same due to their similar designs. Hence, a BP row has the same state (active or inactive) in both these components. When only one of local or global component row is active, its prediction is used. Further, if this active row is in the global component, then the selector row will also be active, and hence, if the selector's choice is the local component, then the local component is activated. However, if the global (and hence, selector) row is inactive, neither of global and selector rows are activated. In essence, by utilizing the decision of the selector, unnecessary activations are avoided, which increases leakage energy saving compared to a naive policy that always activates an inactive row. Decay intervals larger than 64K cycles are found to provide large leakage energy saving without substantially increasing mispredictions or performance loss. Their technique brings large reduction in BP leakage energy.

Banisadi et al[60] propose a technique to save energy in hybrid (eg, gshare+bimodal) BPs. They note that branch instructions show temporal locality, such that more than 50% branches appear within 8 branches and more than 80% branches appear within 64 branches. Also, a branch generally uses the same sub-predictor for prediction, still hybrid BPs access both sub-predictors and the meta-predictor for each branch which wastes energy. Based on these observations, they use a FIFO (first-in first-out) buffer, where each buffer entry uses PC of a recently used branch as the tag and sub-predictors used by the two subsequent branches in the dynamic execution stream. Assigning sub-predictor hints to a preceding branch helps in avoiding increase in prediction latency as the hints are available at least one cycle before the actual prediction.

The processor fetches at most two branches in each cycle for a total of 8 instructions. Then, it compares the last fetched branch with any buffer entry. On a match, sub-predictor hints of the next two buffer entries are retrieved and assumed to be same for the next two branches. Based on this, the unused sub-predictor and meta-predictor are both power-gated, and the used predictor is directly accessed. No power-gating is performed (and hence, all sub-predictors are accessed) if (1) no match was not found in the buffer or (2) the buffer indicates that the branch was previously mispredicted, and hence, confidence in the previously used sub-predictor is low. Their technique saves power for both small and large hybrid BPs. Also, using their technique leads to lower power consumption than using a single sub-predictor. However, in absence of temporal locality of branches, their technique can increase power consumption by choosing wrong sub-predictors.

Parikh et al[68] present three techniques to save BP energy. First, they use banking to reduce access latency and dynamic energy since only a portion of BP needs to be kept active at a time. Second, they use a structure called "prediction probe detector," which has same number of entries as the I-cache. This detector stores pre-decode bits to detect whether the cache block has conditional branches which avoids access to BP. If cache block has no control-flow instructions, then access to BTB itself is avoided. Thus, instead of BTB, only the detector needs to be accessed in each cycle, which is much smaller in size. However, since the detector entry corresponds to I-cache block, their design increases complexity, particularly with set-associative I-cache. Third, they use profiling to identify highly biased branches, and such branches do not require static prediction. They also note that even on using an adaptive control strategy, pipeline gating does not provide energy saving, which is due to the inaccuracy of pipeline gating itself and wrong-path instructions waste only small energy before being squashed on a misprediction.

8.7 | Avoiding redundant reads and updates

Baniasadi et al[47] note that for several branches, after completion of the learning phase, the collected information remains same for a long time (eg, a loop with large number of iterations). Thus, in their steady-state phase, these branches are easy to predict. Their technique identifies branches that were taken for a threshold number of times and were predicted accurately. For such branches, BP is not accessed. Also, if BP already has sufficient information for accurate prediction, BP is not updated. They further note that several other branches can be accurately predicted with only one of the component predictors in a hybrid predictor, and these branches generally use the same predictor as they used the last time. For such branches, only one component-predictor is accessed. Their technique reduces BP accesses and energy consumption with minor impact on performance.

Seznec et al[62] propose strategies to reduce hardware overhead of TAGE and improve its accuracy by enhancing it with side predictors. They note that updating predictor tables at retire time avoids pollution by wrong path; however, this late update leads to extra mispredictions over the ideal policy of fetch-time update. Any branch on correct path requires three accesses to predictor: reads at prediction and retire stages and write at retire stage. This necessitates complex bank-interleaved or multiported designs. They show that unlike in other predictors (eg, gshare,[84] GEHL[30]), in TAGE predictor, avoiding retire-time read leads to only minor loss in accuracy. Combining this with the redundant-update avoidance scheme[47] leads to significant reduction in BP accesses, eg, only 1.13 accesses for each retired branch on average. This allows designing TAGE predictor as a 4-way bank-interleaved design with single-ported memory banks, which provides large saving in area and energy. The bank-interleaving approach can be used with most global history BPs with only small accuracy loss, although using it with local history BPs leads to large losses.

To further reduce mispredictions due to late updates, they propose integrating side predictors with TAGE. The "immediate update mimicker" reduces such mispredictions by predicting branches that have in-flight non-retired instances. If two branches B1 and B2 are predicted by the same table and the same table entry and B1 has already executed but has not retired, then outcome of branch B1 is used as the prediction for branch B2 instead of the output of the predictor. Further, TAGE cannot predict loop exits for loops with irregular control flow. To predict such loop exits, they use a loop predictor.[25] Since TAGE fails to accurately predict biased branches that are uncorrelated with branch path or history, they also use a statistical-corrector predictor.[33] Statistical-corrector predictor can also utilize local history to improve effectiveness of TAGE. Overall, their enhanced TAGE predictor achieves high accuracy.

9 | CONCLUSION AND FUTURE OUTLOOK

In this paper, we presented a survey of dynamic branch prediction techniques. We classified the research works to present a bird's-eye view of the field. We reviewed several research proposals to bring forth their key insights and relative merits. We hope that real contribution of this survey will be able to stimulate further research in this area and make definite impact on the BPs used in the next-generation processors. We close the paper with a brief mention of future challenges in this area.

Since BP is a speculative technique, it may waste energy. Hence, its use in highly energy-constrained systems such as battery-operated mobile devices[106] is challenging. To reduce BP energy consumption, it can be designed with non-volatile memory, or aggressive energy management techniques such as power-gating[107] can be used. Also, developing techniques for improving BP accuracy and reducing energy wastage on wrong-path execution will be a major research challenge for computer architects.

BPs are useful not only for predicting program control flow to improve performance but also to predict occurrence of interesting event to guide power-management of processors. Recently, Bhattacharjee[38] has proposed using BP for predicting brain activity in brain-machine implants. Since these implants need to achieve extremely high energy efficiency, they propose running the embedded processor at normal frequency in case of an interesting activity and transitioning it to low-power mode otherwise. They use multiple BPs, eg, gshare,[84] two-level BPs,[80] Smith BP[9] and perceptron BP[31] for predicting neuronal activity in the cerebellum. They observe that perceptron BP can predict cerebellar activity with high accuracy (up to 85%). Based on this, the processor can be intelligently transitioned to low-power modes, which saves power. The reason for high accuracy of perceptron BP is its ability to track correlations with much longer history than other BPs. These and similar techniques are also effective in mitigating energy-overhead of BP. Evidently, BPs are versatile and powerful tools and can find use in processors used for medical, space (eg, saving energy in robots operated in remote environments), and military applications.

The lack of detailed information about BPs in real processors and subsequent differences in the simulation frameworks used in academia and industry[108] may threaten the validity of BP studies. To address this, a closer collaboration between academia and industry is required. Further, BPs evaluated with old workloads with small footprints (eg, SPEC92) may not be optimized for recent workloads[72] since in SPEC92 workloads, a few branches are executed very frequently and handling them improves overall accuracy, whereas this strategy may not reduce overall accuracy in recent workloads with large number of static branches. Clearly, evaluation of BPs with up-to-date workloads is important to obtain meaningful conclusions.

Due to aggressive performance optimization features present in modern processors, different branches have different misprediction overheads.[71] Hence, an improvement in BP accuracy may not translate into corresponding performance improvement. Yet, several works only report BP accuracy [1,2,16,19,21,23,25,30,34,43,50,51,55,59,62-64,70,72,75,79,80,82,83,87,91,95,103] without reporting system-performance, and hence, they may provide misleading impression of the potential of a BP management technique. To address this issue, researchers need to adopt comprehensive evaluation methodology where both accuracy and performance impact of BPs are evaluated.

ORCID

Sparsh Mittal http://orcid.org/0000-0002-2908-993X

REFERENCES

1. Seznec A, Felix S, Krishnan V, Sazeides Y. Design tradeoffs for the Alpha EV8 conditional branch predictor. In: Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02); 2002; Anchorage, Alaska.
2. Jiménez DA. An optimized scaled neural branch predictor. Paper presented at: 2011 IEEE 29th International Conference on Computer Design (ICCD); 2011; Amherst, MA.
3. Jiménez DA. Reconsidering complex branch predictors. In: Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA); 2003; Anaheim, CA.
4. Jiménez DA, Keckler SW, Lin C. The impact of delay on the design of branch predictors. In: Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000; 2000; Monterey, CA.
5. Loh GH. A simple divide-and-conquer approach for neural-class branch prediction. Paper presented at: 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05); 2005; St. Louis, MO.
6. Horn J. Reading Privileged Memory with a Side-Channel. https://bit.ly/2CtE9Jx
7. Kocher P, Genkin D, Gruss D, et al. Spectre attacks: Exploiting speculative execution. 2018. arXiv preprint arXiv:1801.01203.
8. Lipp M, Schwarz M, Gruss D, et al. Meltdown. https://meltdownattack.com/meltdown.pdf
9. Smith JE. A study of branch prediction strategies. In: Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA '81); 1981; Minneapolis, MN.
10. Skadron K, Martonosi M, Clark DW. A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions. In: Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques; 2000; Philadelphia, PA.
11. Mittal S. A survey of value prediction techniques for leveraging value locality. *Concurrency Computat Pract Exper*. 2017;29(21):e4250.
12. Young C, Gloy N, Smith MD. A comparative analysis of schemes for correlated branch prediction. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95); 1995; Ligure, Italy.
13. Evers M, Chang P-Y, Patt YN. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In: Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96); 1996; Philadelphia, PA.
14. Bonanno J, Collura A, Lipetz D, Mayer U, Prasky B, Saporito A. Two level bulk preload branch prediction. Paper presented at: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA); 2013; Shenzhen, China.
15. Fog A. The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers. Lyngby, Denmark: Technical University of Denmark; 2011.
16. Juan T, Sanjeevan S, Navarro JJ. Dynamic history-length fitting: a third level of adaptivity for branch prediction. In: Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98); 1998; Barcelona, Spain.
17. Milenkovic M, Milenkovic A, Kulick J. Demystifying Intel branch predictors. Paper presented at: Workshop on Duplicating, Deconstructing, and Debunking; 2002; Anchorage, Alaska.
18. Gwennap L. Digital 21264 sets new standard. *Microprocessor Report*. 1996;10(14):11-16.
19. Yeh T-Y, Patt YN. A comparison of dynamic branch predictors that use two levels of branch history. In: Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93); 1993; San Diego, CA.
20. Gope D, Lipasti MH. Bias-free branch predictor. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47); 2014; Cambridge, UK.
21. Loh GH, Jiménez DA. Reducing the power and complexity of path-based neural branch prediction. In: Proceedings of the 5th Workshop on Complexity Effective Design (WCED5); 2005; Madison, WI.

22. Sprangle E, Chappell RS, Alsup M, Patt YN. The agree predictor: A mechanism for reducing negative branch history interference. In: Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97); 1997; Denver, CO.
23. Seznec A, Michaud P. A case for (partially)-tagged geometric history length predictors. *J Instr Level Parallelism*. 2006;8:1-23 .
24. Evers M, Patel SJ, Chappell RS, Patt YN. An analysis of correlation and predictability: what makes two-level branch predictors work. *ACM SIGARCH Comput Archit News*. 1998;26(3):52-61.
25. Gao H, Zhou H. Adaptive information processing: an effective way to improve perceptron branch predictors. *J Instr-Level Parallelism*. 2005;7:1-10.
26. Tarjan D, Skadron K. Merging path and gshare indexing in perceptron branch prediction. *ACM Trans Archit Code Optim*. 2005;2(3):280-300.
27. Akkary H, Srinivasan ST, Koltur R, Patil Y, Refaai W. Perceptron-based branch confidence estimation. In: Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA '04); 2004; Madrid, Spain.
28. Cooper K, Torczon L. *Engineering a Compiler*. Burlington, MA: Elsevier; 2011.
29. Nair R. Dynamic path-based branch correlation. In: Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO 28); 1995; Ann Arbor, MI.
30. Seznec A. Analysis of the O-GEometric history length branch predictor. In: Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA '05); 2005; Madison, WI.
31. Jiménez DA, Lin C. Neural methods for dynamic branch prediction. *ACM Trans Comput Syst*. 2002;20(4):369-397.
32. Loh GH. Revisiting the performance impact of branch predictor latencies. Paper presented at: 2006 IEEE International Symposium on Performance Analysis of Systems and Software; 2006; Austin, TX.
33. Seznec A. A 64-Kbytes ISL-TAGE branch predictor. Paper presented at: JILP Workshop on Computer Architecture Competitions; 2011; San Jose, CA.
34. Seznec A, Michaud P. De-Aliased Hybrid Branch Predictors [PhD thesis]. INRIA; 1999.
35. Mittal S. Power management techniques for data centers: a survey. Technical Report. ORNL/TM-2014/381. Oak Ridge, TN: Oak Ridge National Laboratory. 2014.
36. Eden AN, Mudge T. The YAGS branch prediction scheme. In: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture; 1998; Dallas, TX.
37. Aragón JL, González J, García JM, González A. Selective branch prediction reversal by correlating with data values and control flow. In: Proceedings 2001 IEEE International Conference on Computer Design (ICCD); 2001; Austin, TX.
38. Bhattacharjee A. Using branch predictors to predict brain activity in brain-machine implants. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17); 2017; Cambridge, MA.
39. Sethuram R, Khan OI, Venkatanarayanan HV, Bushnell ML. A neural net branch predictor to reduce power. In: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07); 2007; Bangalore, India.
40. Ayoub R, Orailoglu A. Filtering global history: power and performance efficient branch predictor. In: Proceedings of the 2009 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors; 2009; Boston, MA.
41. Xie Z, Tong D, Cheng X. An energy-efficient branch prediction technique via global-history noise reduction. Paper presented at: International Symposium on Low Power Electronics and Design (ISLPED); 2013; Beijing, China.
42. Huang M, He D, Liu X, Tan M, Cheng X. An energy-efficient branch prediction with grouped global history. Paper presented at: 2015 44th International Conference on Parallel Processing (ICPP); 2015; Beijing, China.
43. Schlais DJ, Lipasti MH. BADGR: a practical GHR implementation for TAGE branch predictors. Paper presented at: 2016 IEEE 34th International Conference on Computer Design (ICCD); 2016; Scottsdale, AZ.
44. Gao H, Ma Y, Dimitrov M, Zhou H. Address-branch correlation: a novel locality for long-latency hard-to-predict branches. Paper presented at: 2008 IEEE 14th International Symposium on High Performance Computer Architecture (HPCA); 2008; Salt Lake City, UT.
45. Wang J, Tim Y, Wong W-F, Li HH. A practical low-power memristor-based analog neural branch predictor. Paper presented at: International Symposium on Low Power Electronics and Design (ISLPED); 2013; Beijing, China.
46. Amant RS, Jiménez DA, Burger D. Low-power, high-performance analog neural branch prediction. In: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41); 2008; Lake Como, Italy.
47. Baniasadi A, Moshovos A. SEPAS: a highly accurate energy-efficient branch predictor. In: Proceedings of the 2004 International Symposium on Low Power Electronics and Design (ISLPED '04); 2004; Newport Beach, CA.
48. Yang C, Orailoglu A. Power efficient branch prediction through early identification of branch addresses. In: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06); 2006; Seoul, Korea.
49. Sendag R, Joshua JY, Chuang P-f, Lilja DJ. Low power/area branch prediction using complementary branch predictors. Paper presented at: 2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS); 2008; Miami, FL.
50. Monchiero M, Palermo G. The combined perceptron branch predictor. In: *Euro-Par 2005 Parallel Processing*. Berlin, Germany: Springer; 2005:487-496.
51. Jiménez DA, Lin C. Dynamic branch prediction with perceptrons. In: Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture; 2001; Monterrey, Mexico.
52. Jiménez DA. Fast path-based neural branch prediction. In: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36); 2003; San Diego, CA.
53. Saadeldeen H, Franklin D, Long G, et al. Memristors for neural branch prediction: a case study in strict latency and write endurance challenges. In: Proceedings of the ACM International Conference on Computing Frontiers (CF '13); 2013; Ischia, Italy.
54. Jiménez DA. Piecewise linear branch prediction. In: Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05); 2005; Madison, WI.
55. Egan C, Steven G, Quick P, Anguera R, Steven F, Vintan L. Two-level branch prediction using neural networks. *J Syst Archit*. 2003;49(12):557-570.
56. Jiménez DA, Lin C. Perceptron learning for predicting the behavior of conditional branches. In: Proceedings of the International Joint Conference on Neural Networks (IJCNN '01); 2001; Washington, DC.

57. Chaver D, Piñuel L, Prieto M, Tirado F, Huang MC. Branch prediction on demand: an energy-efficient solution. In: Proceedings of the 2003 International Symposium on Low Power Electronics and Design (ISLPED '03); 2003; Seoul, Korea.
58. Falcon A, Stark J, Ramirez A, Lai K, Valero M. Prophet/critic hybrid branch prediction. In: Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04); 2004; München, Germany.
59. Kampe M, Stenstrom P, Dubois M. The FAB predictor: using Fourier analysis to predict the outcome of conditional branches. In: Proceedings Eighth International Symposium on High Performance Computer Architecture (HPCA); 2002; Cambridge, MA.
60. Baniasadi A, Moshovos A. Branch predictor prediction: a power-aware branch predictor for high-performance processors. In: Proceedings of the IEEE International Conference on Computer Design (ICCD); 2002; Freiberg, Germany.
61. Evers M. Improving Branch Prediction by Understanding Branch Behavior [PhD thesis]. Ann Arbor, MI: The University of Michigan; 2000.
62. Seznec A. A new case for the tage branch predictor. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44); 2011; Porto Alegre, Brazil.
63. Seznec A, Miguel JS, Albericio J. The inner most loop iteration counter: a new dimension in branch history. Paper presented at: 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO); 2015; Waikiki, HI.
64. Albericio J, Miguel JS, Jerger NE, Moshovos A. Wormhole: wisely predicting multidimensional branches. Paper presented at: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO); 2014; Cambridge, UK.
65. Heil TH, Smith Z, Smith JE. Improving branch predictors by correlating on data values. In: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 32); 1999; Haifa, Israel.
66. Yeh T-Y, Marr DT, Patt YN. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In: Proceedings of the 7th International Conference on Supercomputing (ICS '93); 1993; Tokyo, Japan.
67. Chang P-Y, Hao E, Yeh T-Y, Patt Y. Branch classification: a new mechanism for improving branch predictor performance. In: Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture; 1994; San Jose, CA.
68. Parikh D, Skadron K, Zhang Y, Stan M. Power-aware branch prediction: characterization and design. *IEEE Trans Comput*. 2004;53(2):168-186.
69. Parikh D, Skadron K, Zhang Y, Barcella M, Stan MR. Power issues related to branch prediction. Paper presented at: Proceedings Eighth International Symposium on High Performance Computer Architecture; 2002; Cambridge, MA.
70. Stark J, Evers M, Patt YN. Variable length path branch prediction. In: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII); 1998; San Jose, CA.
71. Loh GH, Henry DS. Predicting conditional branches with fusion-based hybrid predictors. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques; 2002; Charlottesville, VA.
72. Sechrest S, Lee C-C, Mudge T. Correlation and aliasing in dynamic branch predictors. In: Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96); 1996; Philadelphia, PA.
73. Gloy N, Young C, Chen JB, Smith MD. An analysis of dynamic branch prediction schemes on system workloads. In: Proceedings of the 1996 23rd Annual International Symposium on Computer Architecture (ISCA '96); 1996; Philadelphia, PA.
74. Li T, John LK, Sivasubramaniam A, Vijaykrishnan N, Rubio J. OS-aware branch prediction: improving microprocessor control flow prediction for operating systems. *IEEE Trans Comput*. 2007;56(1):2-17.
75. Chang P-Y, Evers M, Patt YN. Improving branch prediction accuracy by reducing pattern history table interference. In: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique; 1996; Boston, MA.
76. Sherwood T, Calder B. Loop termination prediction. Paper presented at: International Symposium on High Performance Computing; 2000; Tokyo, Japan.
77. Al-Otoom M, Forbes E, Rotenberg E. EXACT: explicit dynamic-branch prediction with active updates. In: Proceedings of the 7th ACM International Conference on Computing Frontiers (CF '10); 2010; Bertinoro, Italy.
78. Chen L, Dropsho S, Albonesi DH. Dynamic data dependence tracking and its application to branch prediction. In: Proceedings of the Ninth International Symposium on High-Performance Computer Architecture; 2003; Anaheim, CA.
79. Porter L, Tullsen DM. Creating artificial global history to improve branch prediction accuracy. In: Proceedings of the 23rd international conference on Supercomputing (ICS '09); 2009; Yorktown Heights, NY.
80. Yeh T-Y, Patt YN. Two-level adaptive training branch prediction. In: Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO 24); 1991; Albuquerque, NM.
81. Yeh T-Y, Patt YN. Alternative implementations of two-level adaptive branch prediction. In: Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92); 1992; Queensland, Australia.
82. Pan S-T, So K, Rahmeh JT. Improving the accuracy of dynamic branch prediction using branch correlation. In: Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V); 1992; Boston, MA.
83. Lee C-C, Chen I-CK, Mudge TN. The bi-mode branch predictor. In: Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture (MICRO 30); 1997; Research Triangle Park, NC.
84. McFarling S. Combining branch predictors. Technical Report. Digital Western Research Laboratory: Palo Alto, CA; 1993.
85. Chen I-CK, Coffey JT, Mudge TN. Analysis of branch prediction via data compression. *ACM SIGPLAN Not*. 1996;31(9):128-137.
86. Cleary J, Witten I. Data compression using adaptive coding and partial string matching. *IEEE Trans Commun*. 1984;32(4):396-402.
87. Gao H, Zhou H. PMPM: prediction by combining multiple partial matches. *J Instr-Level Parallelism*. 2007;9:1-18.
88. Michaud P. A PPM-like, tag-based branch predictor. *J Instr Level Parallelism*. 2005;7(1):1-10.
89. Michaud P, Seznec A, Uhlig R. Trading conflict and capacity aliasing in conditional branch predictors. In: Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97); 1997; Denver, CO.
90. Seznec A. A case for two-way skewed-associative caches. *ACM SIGARCH Comput Archit News*. 1993;21(2):169-178.
91. Ma Y, Gao H, Zhou H. Using indexing functions to reduce conflict aliasing in branch prediction tables. *IEEE Trans Comput*. 2006;55(8):1057-1061.

92. Seznec A. Storage free confidence estimation for the TAGE branch predictor. In: Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11); 2011; San Antonio, TX.
93. Seznec A. A 256 kbits L-TAGE branch predictor. *J Instr Level Parallelism*. 2007;9:1-6.
94. Jimenez DA, Loh GH. Controlling the power and area of neural branch predictors for practical implementation in high-performance processors. Paper presented at: 2006 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'06); 2006; Ouro Preto, Brazil.
95. Lai C, Lu S-L, Chen Y, Chen T. Improving branch prediction accuracy with parallel conservative correctors. In: Proceedings of the 2nd Conference on Computing Frontiers (CF '05); 2005; Ischia, Italy.
96. Mittal S. A survey of techniques for designing and managing CPU register file. *Concurrency Computat Pract Exper*. 2016;29(4).
97. Thomas R, Franklin M, Wilkerson C, Stark J. Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In: Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03); 2003; San Diego, CA.
98. Choi B, Porter L, Tullsen DM. Accurate branch prediction for short threads. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII); 2008; Seattle, WA.
99. Hily S, Seznec A. Branch prediction and simultaneous multithreading. In: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique; 1996; Boston, MA.
100. Akkary H, Srinivasan ST, Lai K. Recycling waste: exploiting wrong-path execution to improve branch prediction. In: Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03); 2003; San Francisco, CA.
101. Manne S, Klauser A, Grunwald D. Branch prediction using selective branch inversion. In: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques; 1999; Newport Beach, CA.
102. Seznec A, Jourdan S, Sainrat P, Michaud P. Multiple-block ahead branch predictors. In: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII); 1996; Cambridge, MA.
103. Sadooghi-Alvandi M, Aasaraai K, Moshovos A. Toward virtualizing branch direction prediction. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE); 2012; Dresden, Germany.
104. Hu Z, Juang P, Skadron K, Clark D, Martonosi M. Applying decay strategies to branch predictors for leakage energy savings. In: Proceedings of the IEEE International Conference on Computer Design (ICCD); 2002; Freiberg, Germany.
105. Santana OJ, Ramirez A, Valero M. Latency tolerant branch predictors. In: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems; 2003; Kauai, HI.
106. Mittal S. A survey of techniques for improving energy efficiency in embedded computing systems. *Int J Comput Aided Eng Technol*. 2014;6(4):440-459.
107. Mittal S. A survey of architectural techniques for improving cache power efficiency. *Sustain Comput Inform Syst*. 2014;4(1):33-43.
108. Loh GH. Simulation differences between academia and industry: a branch prediction case study. Paper presented at: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS); 2005; Austin, TX.

How to cite this article: Mittal S. A survey of techniques for dynamic branch prediction. *Concurrency Computat Pract Exper*. 2019;31:e4666. https://doi.org/10.1002/cpe.4666