Prophet/Critic Hybrid Branch Prediction

Ayose Falcón § Jared Stark ‡ Alex Ramirez § Konrad Lai ‡ Mateo Valero §

§ Computer Architecture Department
Universitat Politècnica de Catalunya
{afalcon, aramirez, mateo}@ac.upc.es

‡ Microarchitecture Research Lab
Intel Corporation
{jared.w.stark, konrad.lai}@intel.com

Abstract

This paper introduces the prophet/critic *hybrid conditional branch predictor, which has two component predictors that play the role of either prophet or critic. The prophet is a conventional predictor that uses branch history to predict the direction of the current branch. Further accesses of the prophet yield predictions for the branches following the current one. Predictions for the current branch and the ones that follow are collectively known as the branch's future. They are actually a prophecy, or predicted branch future. The critic uses both the branch's history and future to give a critique of the prophet's prediction for the current branch. The critique, either agree or disagree, is used to generate the final prediction for the branch.*

Our results show an 8K+8K byte prophet/critic hybrid has 39% fewer mispredicts than a 16K byte 2Bc-gskew predictor—a predictor similar to that of the proposed Compaq Alpha* EV8 processor—across a wide range of applications. The distance between pipeline flushes due to mispredicts increases from one flush per 418 micro-operations (uops) to one per 680 uops. For* `gcc`*, the percentage of mispredicted branches drops from 3.11% to 1.23%. On a machine based on the Intel® Pentium® 4 processor, this improves uPC (Uops Per Cycle) by 7.8% (18% for* `gcc`*) and reduces the number of uops fetched (along both correct and incorrect paths) by 8.6%.*

1. Introduction

Processor design is an exercise in trading off performance, power, and energy. Techniques that don't require making this tradeoff, that is, that provide a win for all three metrics, are highly desirable because they can give your design an advantage over competing designs. Better branch prediction is such a technique. It increases performance by reducing the time spent speculating on mispredicted paths, reduces power by allowing the processor to run at a lower frequency (and hence voltage) and still meet its performance target, and reduces energy consumption by reducing the work wasted on misspeculation.

In addition, the branch predictor is not tightly coupled with the microarchitecture, making it relatively simple to replace with a better one, so that an improved version of the processor can be made available to customers. Despite abundant research on branch prediction, however, the branch prediction problem has not been solved. Leading microarchitects and researchers have said branch prediction will be even more important as pipelines deepen and issue widths increase [30, 29].

This paper introduces a technique for better branch prediction, which we call *prophet/critic* hybrid branch prediction. We will initially describe this technique by drawing an analogy between running a program and taking a ride in a taxicab. The taxi is the processor, the driver is the branch predictor, and the passenger is the pipeline. The system of roads represents the possible control flow paths through the program. The intersections are branches; that is, points where the driver must decide a particular path to follow. It is the driver's job to navigate the taxi through the system of roads, making the correct turns at intersections, to get to the destination; i.e., the end of the program. Wrong turns waste the passenger's time.

Conventional predictors are analogous to a taxi with just one driver. He gets the passenger to the destination using knowledge of the roads acquired from previous trips; i. e., using history information stored in the predictor's memory structures. When he reaches an intersection, he uses this knowledge to decide which way to turn. The driver accesses this knowledge in the context of his current location. Modern branch predictors access it in the context of the current location (the program counter) plus a history of the most recent decisions that led to the current location.

Prophet/critic hybrids are analogous to a taxi with two drivers: the front-seat and the back-seat. The front-seat driver has the same role as the driver in the single-driver taxi. This role is called the *prophet*. The back-seat driver has the role of *critic*. She watches the turns the prophet makes at intersections. She doesn't say anything unless she thinks he's made a wrong turn. When she thinks he's made

Intel® and Pentium® are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
*Other names and brands may be claimed as the property of others.

Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)
1063-6897/04 $ 20.00 © 2004 IEEE

a wrong turn, she waits until he's made a few more turns to be certain they are lost. (Sometimes the prophet makes turns that initially look questionable, but, after he makes a few more turns, in hindsight appear to be correct.) Only when she's certain does she point out the mistake. To recover, they backtrack to the intersection where she believes the wrong-turn was made and try a different direction.

Using prophet/critic hybrids greatly reduces the number of mispredicts. And, as the critic uses more future bits (i. e., waits for the prophet to make more turns before reporting they are lost), the reduction in mispredicts grows. Our experiments use some of the best predictors proposed in literature to play the roles of prophet and critic. Depending on the prophet and critic, a critic using 1 future bit reduces mispredicts by 10–20% over a prophet scaled up to the same size as the prophet/critic hybrid. For 12 future bits, this reduction grows to 15–30%.

2. Related work

Abundant research has been done in the field of dynamic branch prediction. See Evers and Yeh [8] for a good introduction. We will only review the work most relevant to prophet/critic hybrid branch prediction.

McFarling [20] first proposed hybrid branch predictors. His hybrid is composed of two component predictors and a selector that decides which one is used to predict each branch. Each component can exploit a different kind of predictability, so a particular branch can be predicted by the component that best captures its behavior. McFarling also introduced a simple mechanism for selecting the component to use for a particular branch, in which a two-bit counter read from a table indicates which component is more accurate for the branch.

Chang et al. [3] proposed a mechanism to classify branches. Using profile information, the hybrid component is selected that best predicts each branch's behavior. Evers et al. [6] added flexibility by increasing the number of component predictors that can be combined in a hybrid. Their *Multi-Hybrid* predictor also incorporates a new selection mechanism that permits fast predictor training under context switches.

Loh and Henry [19] improved prediction accuracy by replacing the selection mechanism traditionally used in hybrids with a *fusion* mechanism. Typical selection mechanisms pick one of the component predictors to provide a branch's prediction. The unpicked components don't contribute to this prediction. Their fusion mechanism, rather than picking a component, combines all component predictions to provide the branch's prediction.

Jiménez et al. [15] explain how predictor overriding provides fast and accurate predictions. Two predictors are used: a small low-latency predictor with poor accuracy and a large high-latency predictor with good accuracy. Two predictions are initiated in parallel, one from each predictor. Each predictor has the same pool of branch history information from which its prediction can be computed. The prediction from the small predictor completes first, and can be used while the prediction from the large predictor is still being computed. Once that prediction has been computed, it overrides the earlier prediction if they differ, forcing all work done based on the earlier prediction to be discarded.

Grunwald et al. [9] show that using the prediction for the current branch in the history of the JRS confidence estimator [12] improves speculation control. In the terminology of this paper, they use one future bit to get a more accurate confidence estimation.

The prophet/critic hybrid combines and builds upon this related work—it is a hybrid, and it does use overriding, and it does use future bits. Its key distinction is that, rather than initiating all component predictions at the same time, they are, at least logically, initiated at different times. This allows the output of the prophet to be used as the input to the critic, eliminating the need for a selection or fusion mechanism. It also allows the critic to gather multiple (i. e., more than one) future bits for a branch. When the branch is on the correct path but mispredicted by the prophet, the critic uses these future bits—of which at least one is wrong due to the mispredict—to train its prediction structures. When the branch is encountered again, the critic uses the future bits as context to identify if the prophet is likely to be wrong and should be overridden. This greatly increases the prediction accuracy for the branch.

3. Prophet/critic hybrid branch prediction

3.1. Overview

Current branch predictors make predictions using history information. Once a branch has been predicted, the predictor can not use information from subsequent predictions to re-predict the branch. Our prophet/critic hybrid can effectively use these subsequent predictions. Figure 1 shows the structure of a prophet/critic hybrid. The hybrid is composed of two conventional predictors that play the role of either prophet or critic.

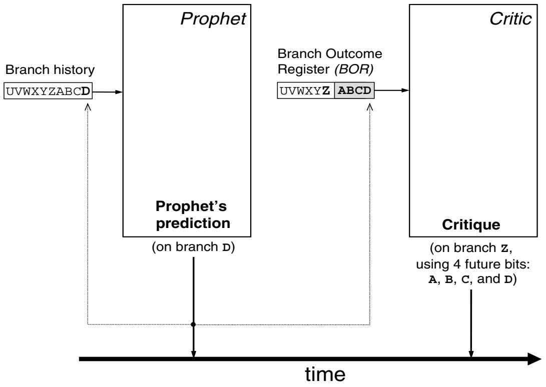


Figure 1. Structure of a prophet/critic hybrid.

As in a typical hybrid, the components of the prophet/critic hybrid can be any existing predictors. The prophet provides predictions based on the past behavior of previous dynamic branches. Once a prediction for a branch is made, the prophet goes on along the predicted path generating new predictions. This prediction and the new predictions form the *branch future*. As in current history-based predictors, the prediction of the current branch is determined by the history, and that prediction determines the branch's future.

While the prophet generates the branch's future, the critic collects this future information, inserting the prophet predictions into its branch outcome register (BOR). Therefore, when a critic prediction is made, its BOR contains two kinds of branch outcomes: (a) outcomes of branches before the one being predicted, which are branch history, and allow the predictor to correlate on the past, and (b) outcomes of the branch being predicted and those after it, which are branch future, and allow the predictor to correlate on the future. Using a combination of past and future correlation, the critic provides a critique of each prophet prediction. This critique will either agree or disagree with the prophet prediction, and will determine the final prediction for the branch. We use the term *critique* as a synonym for critic prediction, but we tend to use it when we want to stress agreement or disagreement with the prophet. For the remainder of this paper, the critic's prediction is the final prediction for the branch.

The prophet/critic hybrid takes advantage of the fact that the prophet and critic operate autonomously, predicting the same branch at different times. In a conventional hybrid predictor, both components are accessed in parallel, making predictions for the same branch. A selection mechanism then picks the prediction that is most suitable for the branch. The same situation occurs for overriding predictors. Two predictions begin in parallel on two different predictors. A first prediction is generated in an early pipeline stage, while a second prediction (for the same branch) is provided some cycles later. In both a conventional hybrid and an overriding predictor, the predictors predict the same branch with the same available information.

However, in the prophet/critic hybrid, although both prophet and critic predict the same branch, the predictions are not initiated at the same time. Since the critic initiates its prediction some cycles later, it can incorporate information about future code behavior (i. e., the branch future provided by the prophet) into its prediction. [1] We will show this future information increases prediction accuracy, especially when branch history is not enough to guess the outcome of the branch.

Example. Consider the control flow graph shown in Figure 2. Assume the correct path in this portion of the code is shown by the shaded blocks (basic blocks 0, 1, 4, and 7). Initially, when branch A is predicted by the prophet, the history is formed by the outcomes of the previous branches: W, X, Y, and Z (not shown in the figure). The prophet uses this history to predict A.

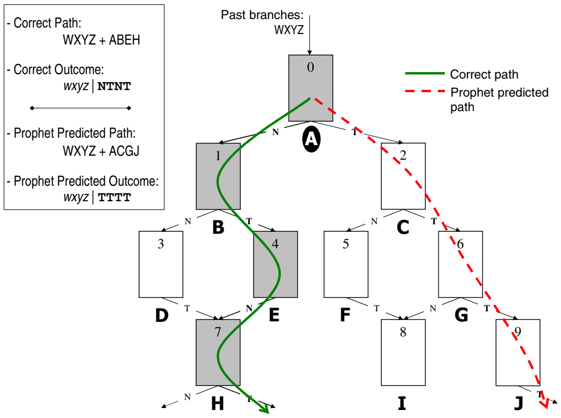


Figure 2. Example of why the critic works.

If the prophet correctly predicts all branches along the correct path, the branch future of A would be formed by the real outcomes of branches A, B, E, and H; i. e., NTNT.

Suppose the prophet mispredicts A and the execution follows the wrong path marked with the dotted line (blocks 0, 2, 6, and 9). The prophet will predict branches A, C, G, and J, and the results of these predictions will be inserted into the critic's BOR. Therefore, when the critic predicts A, its BOR contains two kinds of information:

- **Branch History**: Outcomes of branches before A (i. e., W, X, Y, and Z), which allow the predictor to correlate on the past.
- **Branch Future**: Predictions for A and the branches after it (i. e., A, C, G, and J), which allow the predictor to correlate on the future.

Inserting just a single future bit into the critic's BOR is enough for the critic to know that the prophet mispredicted the branch; that is, **T** was inserted instead of **N**. But, as in the example in the introduction, the more future bits that are inserted (i. e., the more turns a taxi makes after a wrong turn), the higher the amount of information available to confirm that there was a mispredict (i. e., a wrong turn) at branch A.

The first time the prophet generates the wrong prediction for branch A and that mispredict is detected, the critic is trained to recognize that situation. Consequently, the next time the prophet mispredicts the branch under the same conditions, the critic remembers the previous behavior, and disagrees with the prophet's prediction. Using more future bits allows the critic to identify longer paths down the wrong path (i. e., more precise branch future contexts).

[1] Actually, to reduce the latency of critic predictions, the prophet and critic predictions can begin in parallel using the branch history, and the branch future bits can later be factored into the critic prediction as the prophet produces them.

However, for a fixed size BOR, doing so comes at the expense of losing branch history information. In the next sections we will show how the number of future bits influences the performance of the prophet/critic hybrid.

3.2. Updating the predictors

Updating branch history registers (BHRs) and BORs. The prophet BHR is updated speculatively with the prophet's predicted branch outcome at the prediction stage. Past studies have shown that the outcomes of the most recent branches are crucial for accurate branch prediction, and that BHRs should be speculatively updated instead of waiting for the branches to resolve [33, 11].

The critic BOR is filled with the predicted outcomes from the prophet, as explained earlier. The critic predictions *are not* used to fill the critic BOR.

Updating pattern tables. The prophet and critic pattern tables are updated non-speculatively, at commit time, when the real outcomes of the branches are known. For example, if the prophet (or the critic) is a two-level adaptive predictor [33], the two-bit counter that provided the prediction is only incremented if the branch was actually taken, and only decremented if the branch was actually not-taken.

3.3. Recovering from a mispredict

On a branch mispredict, the prophet BHR and the critic BOR are repaired via checkpointing. When the prophet predicts a branch, a copy of the current BHR and the current BOR are assigned to the branch. If a mispredict is detected for the branch, the BHR and BOR are restored from the values assigned to the branch, the mispredicted branch's correct outcome is inserted into the BHR and BOR, and fetch is directed to the correct path. Other repair mechanisms are also possible [17]. The pattern tables don't require repair since they are updated non-speculatively.

We would like to point out that a branch can be mispredicted and still be on the correct path. It is the instructions that follow it that are on the wrong path. Such a branch is not flushed from the pipeline when the mispredict is detected. It continues down the pipeline, and when it commits, it trains the critic using the same BOR value that was used to generate the critic prediction. This BOR value has a single (mispredicted) future bit that corresponds to the branch, plus future bits for the branches on the wrong path that were flushed. If the BOR value did not contain the future bits for the wrong path, the critic would never be trained to recognize when the prophet has mispredicted a branch and gone down the wrong path.

4. Filtering the critic

The critic's accuracy can be limited by multiple branches contending for the same prediction resources; that is, by conflicts. Chang et al. [2] realized that conflicts can be reduced by filtering easy-to-predict branches from a predictor. They (dynamically) identified the easy-to-predict branches and predicted them with a simple predictor. Only the difficult branches were predicted with the original predictor, for which conflicts were a problem. By allowing fewer branches to access the original predictor's tables, they were able to reduce the number of conflicts among branches, and increase overall prediction accuracy.

In this paper we explore filtering the critic. The prophet provides a prediction for every branch, so the processor always has an available prediction regardless of whether the critic provides a prediction. For good performance, the critic should only provide a prediction in the cases where the prophet is likely to be wrong. We use prophets that correctly predict 90–95% of all branches. The critics should then only be responsible for predicting the 5–10% of branches that the prophets mispredict.

Figure 3 shows a filtered critic. The critic has a table of tags it uses to filter branches. When a branch needs a critique, two actions are performed in parallel: first, the critic is queried for its prediction; and second, the tag table is accessed to determine if there is a tag hit. If there is a hit, the critic's prediction is used as the critique. If there is a miss, the critic implicitly agrees with the prophet's prediction, and the critic's prediction is simply ignored. The critic is only trained for branches that have hits.

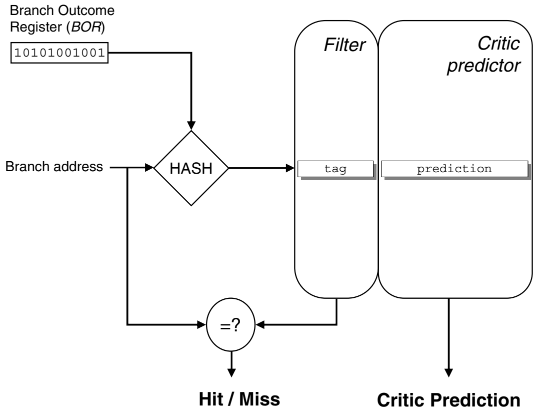


Figure 3. Filtered critic.

New entries are inserted into the table when a branch has a tag miss and it is mispredicted. The tag for the particular branch address and BOR value combination is inserted so that the next time that context is encountered, the critic's prediction will be used for the branch. The critic's prediction structures are also initialized according to the branch's outcome. The tags are managed using a least-recently-used (LRU) replacement algorithm.

The index into the table and the tags are computed with two different hash functions. The hash functions are selected to minimize the probability that a particular branch address and BOR value combination will use the same table entry and have the same tag as another combination.

In our experiments, the hash functions are different XOR functions of the branch address and BOR value. Our experiments have shown that only 8–10 bit tags are needed to clearly identify the different branch contexts.

5. A prophet/critic hybrid implementation

Countless front-end architectures have been proposed, and many of them can benefit from a prophet/critic hybrid. For example, the front-end pipeline of the Compaq Alpha 21264 processor [10] contains a hybrid consisting of a line predictor and a tournament local-global branch predictor. This hybrid could be replaced with a prophet/critic hybrid. The prophet would provide an initial prediction at the beginning of the pipeline. This prediction would drive the fetch unit along the predicted path, and also provide a future bit for the critic. Several stages later, the critic would provide a second prediction based on the future bits, which would potentially override the prophet's prediction. The number of stages between the prophet and critic predictors determines the number of future bits used to generate the critique. Other front-end architectures, like the trace cache [22] or the stream fetch architecture [23] can also benefit from a prophet/critic hybrid.

For our uPC (Uops [i. e., micro-operations] Per Cycle) performance results, which are presented in Section 7.4, we modeled a decoupled front-end architecture [24] that separates branch prediction generation from branch prediction consumption. We also use an overriding scheme: the critic's prediction is always trusted to be correct, and will override the prophet's prediction for a branch.

Figure 4 shows this architecture. A queue (*fetch target queue*, or FTQ) decouples the hybrid from the instruction cache. The hybrid produces predictions and inserts them in the FTQ, and the cache later consumes them. Our prophet/critic hybrid requires that the FTQ be full (or mostly full) most of the time. To accomplish this, the hybrid is designed to produce predictions faster than the cache consumes them, so that the FTQ fills to capacity. For IA32, conditional branches—averaged over all benchmarks, not just integer—occur every 13 uops. A machine twice as aggressive as the Intel Pentium 4 processor, fetching 6 uops per cycle, encounters a conditional branch about every other cycle. A hybrid producing one prediction per cycle would therefore meet the requirement.

The hybrid uses a *branch target buffer* (BTB) to identify branches. When a conditional branch is identified, the hybrid predicts its direction. When a branch misses the BTB, a BTB entry is allocated for the branch when it commits. Other allocation policies are also possibly, such as only allocating entries for taken branches.

When a branch is identified by the BTB, the prophet provides its initial prediction and inserts the prediction in the FTQ. This prediction *can be* immediately consumed by the cache, but, since insertions occur at the end of the FTQ and the FTQ is typically full, the prediction usually spends many cycles in the FTQ before it is consumed. When the prediction is inserted in the FTQ, it is also inserted in the critic's BOR as a future bit for branches previously predicted by the prophet. As subsequent predictions are inserted in the FTQ, the critic gathers them as future bits for the branch. When it has gathered the required number of future bits (this is a fixed number) for the branch, it provides a critique of the prophet's prediction. The critique is typically generated well before the prediction is consumed by the cache.

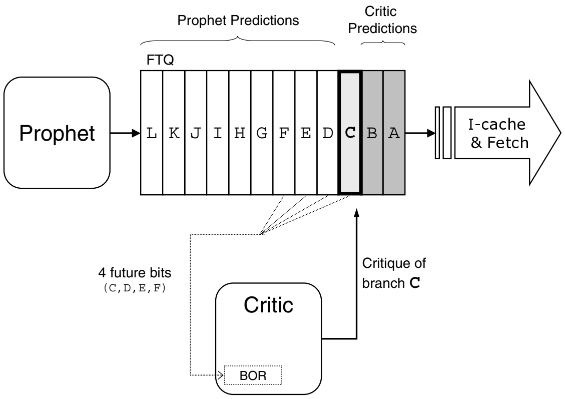


Figure 4. Prophet/critic hybrid implementation on a decoupled front-end architecture.

If the critic agrees with the prophet's prediction, the prediction is marked as having been criticized, and then the critic moves on to the next uncriticized prediction. In the figure, unshaded FTQ entries hold uncriticized predictions, and shaded FTQ entries hold predictions that are being or have been criticized. On the other hand, if the critic disagrees with the prophet, several actions are taken: (a) the critic's prediction overrides the prophet's prediction, (b) FTQ entries holding uncriticized predictions are flushed, and (c) the prophet is redirected to the path predicted by the critic. The flush is confined to the FTQ, since the cache and the rest of the machine haven't received any of the flushed predictions. The criticized predictions are left alone, so if the FTQ is sufficiently full, the flush causes no performance penalty. (We've measured the percentage of times the FTQ is empty when the cache requires a prediction for both a prophet/critic hybrid and a conventional predictor, and they are very nearly equal.)

There are cases where the cache requires a prediction but the critic has not gathered enough future bits to provide a critique of the prediction. When this happens, either the prediction can be passed to the cache without providing a critique, or the critic can generate a critique using the future bits available at this point. In our experiments, the prophet produces 2 predictions per cycle and the critic produces 1 prediction per cycle. For 8 future bits, these cases account for less than 0.1% of the times the cache requires a prediction. (For fewer future bits, this percentage will be smaller; and for more, it will be larger.) Although these cases are rare, we obtained the best results by generating a critique using the future bits that were available.

6. Simulation methodology

We use a cycle-accurate IA32, uop based, execution driven simulator that executes Long Instruction Traces (LITs). A LIT is not actually a trace, but a snapshot of the processor state, including memory, that can be used to initialize an execution-based performance simulator. It also contains a list of system interrupts needed to simulate system events such as DMA. Since the LIT includes an entire snapshot of memory, we can simulate both user and kernel instructions, as well as model wrong-path effects.

Unlike conventional branch predictors, prophet/critic hybrids *must* be evaluated on simulators that model going down wrong paths. In a prophet/critic hybrid, the branch future bits generated along a wrong path are essential to the critic, because they provide the information necessary to detect mispredicts. In addition, these future bits must be generated by actually going down the wrong path. Generating these bits while traversing a (correct-path only) instruction trace provides the critic with oracle information, which it does not actually have.

Table 1 lists the benchmark suites we use. In total, we simulate 108 benchmarks, some of them with different inputs, which comprise 341 LITs. The LITs are traces of 30 million IA32 instructions extracted from the corresponding benchmark after the initial startup phase and are carefully chosen to be representative of the overall characteristics of the benchmark.

Suite	No. of Bench.	Description or Sample Benchmarks
SPECint*2K (INT00)	12	http://www.spec.org
SPECfp*2K (FP00)	14	http://www.spec.org
Internet (WEB)	28	SPECjbb*, WebMark*
Multimedia (MM)	15	MPEG, speech recognition, Quake*
Productivity (PROD)	27	SYSmark*2K, Winstone*
Server (SERV)	2	TPC-C*, TimesTen*
Workstation (WS)	12	CAD, Verilog*

Table 1. Simulated benchmark suites.

Table 2 gives our simulation parameters. We evaluate our prophet/critic hybrid on a superscalar out-of-order microarchitecture derived from the Intel Pentium 4 processor. This microarchitecture runs at the same frequency as that processor, but is twice as wide and has caches that are (roughly) twice as big and associative. We scaled up the instruction window size (and associated buffers, the scheduling window and load/store buffers) by a factor of 16 to reflect where we believe future microarchitectures are headed [1]. In addition, we evaluate our hybrid on a decoupled front-end architecture [24], and so have replaced the original trace-cache-based front-end with the decoupled front-end. The decoupled front-end uses a decoded instruction cache (storing uops) instead of a trace cache.

Processor Frequency	3.8 GHz
Fetch/Issue/Retire Width	6 uops
Branch Mispredict Penalty	30 cycles
BTB	4096 entries, 4-way
FTQ Size	32 entries
Instruction Window Size	2048 uops
Scheduling Window Size	256 int, 128 mem, 384 fp (sizes in uops)
Load/Store Buffer Sizes	768 load, 512 store (sizes in uops)
Functional Units	6 int, 4 mem, 2 fp
Hardware Data Prefetcher	Stream-based (16 streams)
Instruction Cache	64 KB, 8-way, 64-byte line
L1 Data Cache	32 KB, 16-way, 64-byte line, 3 cycle hit
L2 Unified Cache	2 MB, 16-way, 64-byte line, 16 cycle hit
Memory Latency	100 ns

Table 2. Simulation parameters.

Predictors simulated

Any predictor can play the role of prophet or critic. The only restriction is the critic must be able to use the predictions generated by the prophet. We have implemented some of the best predictors proposed in literature, and used them as prophet and critic. Other predictors were also tested, but their results are not shown due to space constraints. A brief description of the predictors we used follows:

- **Gshare** [20]: McFarling found that using global history causes interference in the pattern tables of two-level predictors, because branches tend to use a limited number of the possible table entries. His solution is to increase the usefulness of branch history, by XORing it together with the branch address. The new indexing mechanism allows branches to share the pattern table in a more efficient way, reducing the aliasing among them.

- **2Bc-gskew** [28]: A derivation of this predictor is implemented in the Compaq Alpha EV8 processor [26]. The original 2Bc-gskew is composed of four tables accessed using global history information: a bimodal table (BIM), two gshare-like tables (G0 and G1), and a metapredictor table (META). Depending on the output of the META table, the final prediction is given either by the BIM table or by the majority vote of the predictions from the BIM, G0, and G1 tables.

  De-aliased branch predictors, such as 2Bc-gskew and YAGS [5], have been shown to achieve higher prediction accuracy at equivalent hardware budgets than larger aliased global history branch predictors such as gshare and GAs [33].

- **Perceptron** [32, 16]: Perceptron prediction is a two-level scheme using perceptrons instead of two-bit counters as the prediction unit. A perceptron is a simple implementation of a neural network that provides predictive capabilities. Perceptrons are represented by vectors of weights, which are implemented as signed integers. The branch address selects a perceptron from

			Total hardware budget				
			2KB	**4KB**	**8KB**	**16KB**	**32KB**
gshare		# entries	8K	16K	32K	64K	128K
		history length	13	14	15	16	17
perceptron		# perceptrons	113	163	282	348	565
		history length	17	24	28	47	57
2Bc-gskew		# entries (per table)	2K	4K	8K	16K	32K
		history length	11	12	13	14	15
tagged gshare		# entries	256*6-way	512*6-way	1024*6-way	2048*6-way	4096*6-way
		BOR size	18	18	18	18	18
filtered perceptron	perceptron	# perceptrons	73	113	163	282	348
		history length	13	17	24	28	47
	filter	# entries	128*3-way	256*3-way	512*3-way	1024*3-way	2048*3-way
		history length	18	18	18	18	18
	BOR size		18	18	24	28	47

Table 3. Prophet and critic configurations.

a pool of perceptrons that is used to compute the branch's prediction. Global history bits are used as inputs of the chosen perceptron to compute the output. The output of the perceptron computation determines the final branch prediction: if the output is negative, the branch is predicted not taken and if it is positive, the branch is predicted taken.

A key advantage of the perceptron predictor is its ability to consider much longer histories than schemes that use tables with saturating counters. This is also an advantage for our mechanism. Note that as more future bits are added to a critic's BOR due to new prophet predictions, the older history bits are lost. By using a perceptron predictor, we can incorporate many more future bits without losing history.

Table 3 shows the parameters used for simulating the selected predictors. We considered hardware budgets ranging from 2–32 kilobytes. History lengths for gshare and 2Bc-gskew are the maximum that can be used for their table sizes. For the perceptron predictor, we chose the combination of perceptron table size and history length that gives the highest prediction accuracy [16].

Table 3 also shows configuration parameters for the filtered versions of gshare and perceptron that will be used as the critic. The *tagged gshare* is a variant of the gshare predictor, in which a tag is assigned to each two-bit counter. Its structure is similar to a N-way associative cache, with each data item being a two-bit counter. The *filtered perceptron* consists on an ordinary perceptron predictor plus an N-way associative table of tags. The perceptron prediction and the tag table lookup are done in parallel, as shown in Figure 3. The critic's prediction is given only when there is a tag hit. A tag miss (i. e., filter miss) implies implicit agreement with the prophet's prediction.

In the filtered perceptron critic, the *history length* values are the number of bits from the BOR that are used by each structure. The bits used are the bits that have been most recently inserted into the BOR. When the (unfiltered) perceptron is used as a critic, its BOR size is the history length of the unfiltered perceptron given in Table 3.

For all critics, the number of BOR, history, and future bits were tuned to give the highest accuracy for the given hardware budget.

With the advance of technology, the number of cycles required to perform a prediction increases. Since this study is primarily concerned with the potential accuracy of the proposed predictor, we will not consider prediction latency in our simulations. Our assumption is that any branch predictor can be pipelined to provide the required predictor bandwidth for the processor [14, 27].

7. Evaluation

We measure mispredict rate in *mispredicts per one thousand uops* (misp/Kuops), and processor performance in *uops per cycle* (uPC). Unless specially mentioned, all results are averaged over all benchmarks.

Section 7.1 analyzes the importance of future bits for accuracy, and the impact that the number of future bits has on the critic's performance. Section 7.2 compares the accuracy of different prophet/critic combinations. Section 7.3 describes the types of critiques and their distribution. Finally, Section 7.4 presents processor performance results.

7.1. The importance of future bits

Two questions naturally arise regarding future bits: *How important are they? And, what is the optimal number needed to provide the best critiques?*

Figure 5 shows simulation results for an 8KB perceptron prophet with an 8KB tagged gshare critic. Mispredict rate is shown as the number of future bits is varied from 0 to 12. We selected 6 benchmarks showing the different behaviors we saw when varying the number of future bits.

For all six benchmarks, adding just one future bit decreases the mispredict rate, averaged over all six benchmarks ("AVG" line in Figure 5) by 15% for this prophet/critic configuration. Note that 0 future bits means that no future information is used, which is the same way a conventional hybrid or overriding predictor operates, where all components have the same available history information from which they can generate predictions. The

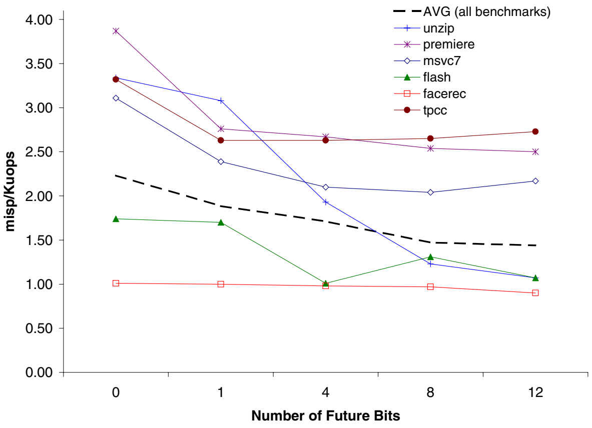


Figure 5. Effect of varying the number of future bits used by the critic on prediction accuracy for selected benchmarks. (prophet: 8KB perceptron; critic: 8KB tagged gshare)

first future bit is the prophet's prediction for the branch. Knowing whether this prediction is taken or not taken is highly valuable information to the critic for determining whether the prophet mispredicted the branch. Our results with other combinations of prophets and critics also show that adding just 1 future bit decreases the mispredict rate over conventional hybrids.

Increasing the number of future bits beyond 1 has different effects depending on the benchmark. For `premiere` and `unzip` the mispredict rate continues to decrease as the number of bits increases. However, `premiere` sees most of its decrease from 0 to 1 (a 29% decrease from 0 to 1, versus 9% from 1 to 12), whereas `unzip` sees most of its decrease from 1 to 12 (8% versus 65%).

For `msvc7` and `flash`, adding future bits decreases the mispredict rate up to a point, after which, the rate increases. For `msvc7`, this point is 8 future bits, with a 34% decrease in mispredict rate from 0 to 8, but a 6% increase from 8 to 12. For `flash`, this point is 4 future bits, with a 42% decrease from 0 to 4. From 4 to 8, the rate increases by 30%, and from 4 to 12, by 6%.

Both `facerec` and `tpcc` show little improvement by going beyond 1 future bit, and for `tpcc`, a small degradation. `facerec` is insensitive to the number of future bits, showing a 2% *increase* in mispredict rate from 0 to 4 bits, but a 10% decrease from 0 to 12. For `tpcc`, the future bits beyond 1 never help, increasing the mispredict rate by 4% from 1 to 12.

In summary, adding some future bits always helps, but more is not always better. Research has shown that the optimal number of history bits depends on the static branch [31] and the program's execution phase [18]. Research has also shown that for a given number of history bits, it is possible to select only those bits that contribute to the predictability of the branch [7]. This same research can be applied to determine the optimal number of future bits and to select the best future bits for a prediction.

7.2. Prediction accuracy

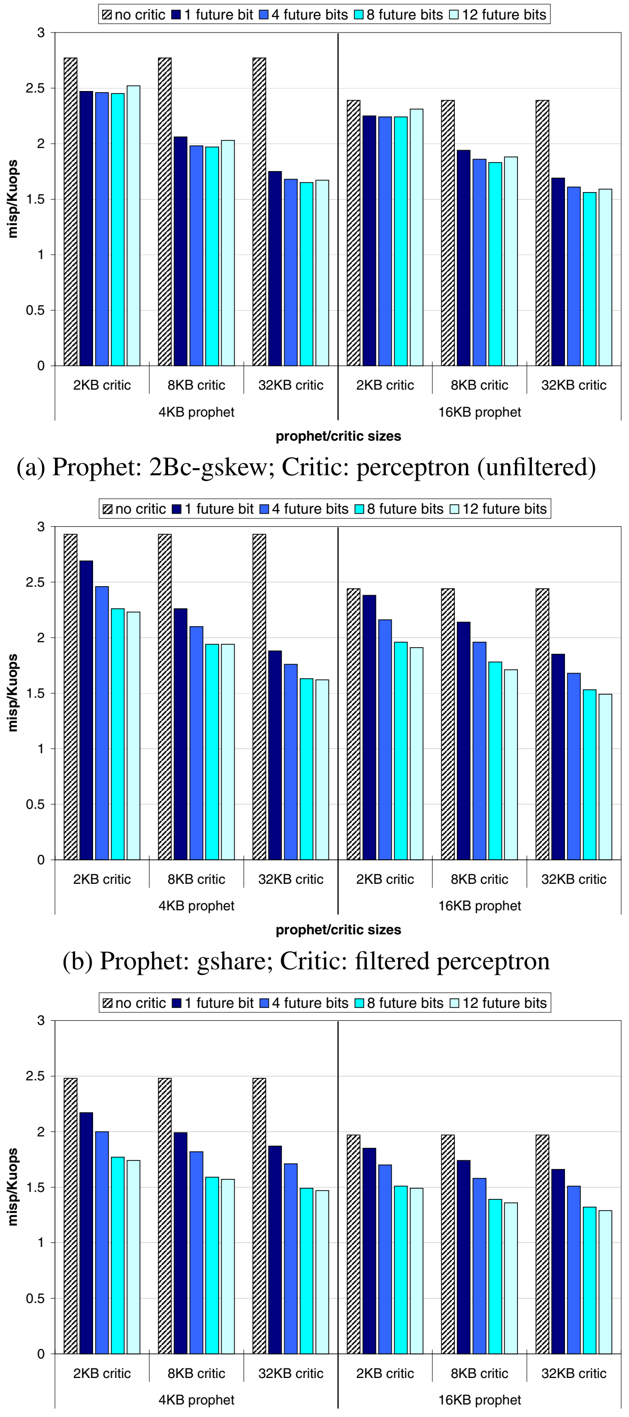
Figure 6 shows the average mispredict rates for different prophet/critic combinations of different sizes. We experimented with many different prophet/critic combinations, including combinations without perceptron predictors, but only show three representative combinations. For each configuration, we show the mispredict rate of using the prophet alone (prophet with no critic), and the mispredict rates of using the prophet/critic hybrid for critics using various numbers (i. e., 1, 4, 8, and 12) of future bits.

Figure 6 shows that adding a critic to a single predictor (i. e., a prophet alone) decreases the mispredict rate. The larger the critic size, the lower the mispredict rate. Adding future bits decreases the mispredict rate up to a point, after which, there are no further improvements. Although adding future bits generally improves accuracy, since the hardware budget is fixed, it typically comes at the expense of having to remove older history bits. At a certain point, the future bit being added provides less information about the branches being predicted than the history bit that must be removed, and consequently, the mispredict rate increases.

This is especially true when the critic provides critiques for branches easily predicted by the prophet, because the information given by future bits for correctly predicted branches is less important than the information given by future bits for mispredicted branches. Figure 6(a) shows that with an unfiltered perceptron critic, the mispredict rate increases when more than 8 future bits are used. Because the critic is unfiltered, it provides a critique for all branches, even the 90–95% of branches that the prophet correctly predicts. The critiques for these correctly predicted branches are of lower quality than they could be, since history bits were removed to make room for relatively useless future bits, and perceptron predictors achieve high accuracy through long histories [16]. In addition, they compete against critiques for mispredicted branches for space in the critic's prediction structures, increasing the mispredict rate.

Our solution to this problem is to filter the critic. Figures 6(b) and 6(c) show the results. With a filter, the critic only provides critiques for branches that the prophet previously mispredicted. These critiques are more likely to use future bits for mispredicted branches than correctly predicted branches, and these future bits provide a more important information. Also, there are fewer critiques for correctly predicted branches, reducing competition for space (i. e., aliasing) in the critic's prediction structures. The overall result is that the mispredict rate continues to decrease as future bits are added.

Figure 7 compares the mispredict rate of the prophet/critic hybrid to some of the best predictors proposed in literature: gshare, 2Bc-gskew, and perceptron. The prophet was made the same as one of those predictors, but



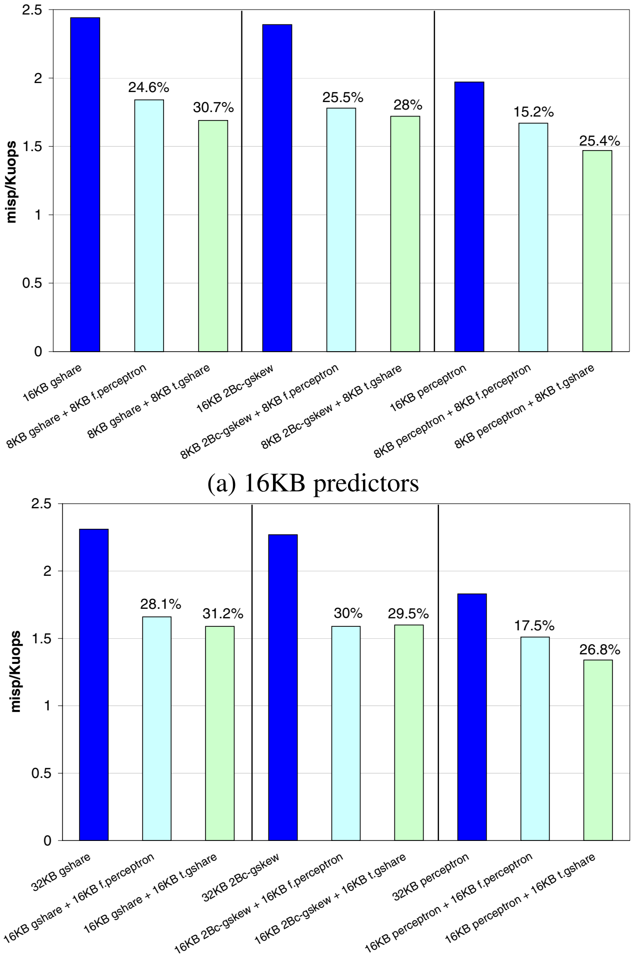
(a) Prophet: 2Bc-gskew; Critic: perceptron (unfiltered)

(b) Prophet: gshare; Critic: filtered perceptron

(c) Prophet: perceptron; Critic: tagged gshare

Figure 6. Prediction accuracies of different prophet/critic combinations and sizes.

given only half the hardware budget. The other half was used for the critic, which was either a tagged gshare or a filtered perceptron. The results show that a prophet/critic hybrid with a tagged gshare critic can reduce the mispredict rate by 25–31%.



(a) 16KB predictors

(b) 32KB predictors

Figure 7. Mispredict rates of conventional predictors compared to prophet/critic hybrids using 8 future bits. Numbers indicate percent reduction in mispredict rate.

7.3. Distribution of critiques

The critic's critiques of the prophet's predictions determine the effectiveness of the prophet/critic algorithm. We classify the final critique of a branch according to the prophet's prediction (i. e., correct or incorrect) and the critic's critique (i. e., agree or disagree). Ideally, the critic disagrees when the prophet mispredicts (*incorrect_disagree*). When the critic agrees with the prophet (*correct_agree* if the prophet didn't mispredict; *incorrect_agree* if it did), the final prediction does not change, so there is neither gain nor harm. However in both these cases, the use of the critic was needless since there was no change in the final prediction; and for incorrect_agree, an opportunity was lost to correct a mispredict. The worst case—and the case that should be minimized—is when the critic disagrees with a prophet's correct prediction (*correct_disagree*).

Figure 8 shows the distribution of critiques for a filtered critic as the number of future bits is varied. The distribu-

	4K perceptron +			*4K perceptron +*			*4K perceptron +*		
	2K tagged gshare			*8K tagged gshare*			*32K tagged gshare*		
	1 fb	4 fb	12 fb	1 fb	4 fb	12 fb	1 fb	4 fb	12 fb
% correct_none	67.6	70.0	76.3	65.9	73.5	75.1	65.3	73.0	75.2
% incorrect_none	1.1	1.1	1.3	0.7	0.9	1.0	0.4	0.7	0.8
% none (Total)	68.7	71.2	77.7	66.6	74.3	76.1	65.7	73.7	76.0

Table 4. Percentage of prophet predictions filtered by the critic, for varying critic sizes and numbers of future bits. Note the size of the filter is proportional to the size of the critic.

tion is only over critiques for which there was a tag hit in the filter; i. e., the implicit agree critiques that result from tag misses in the filter are not shown in this graph.

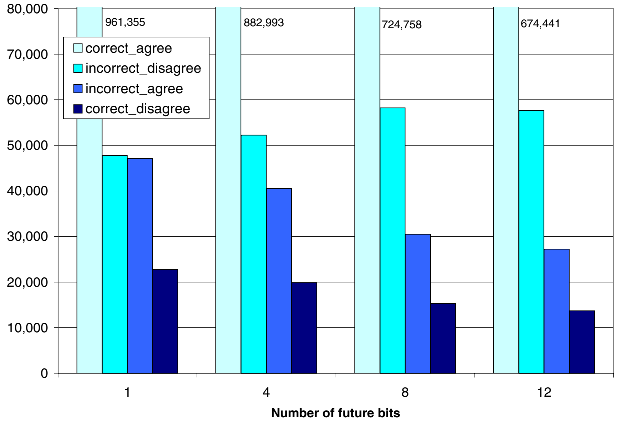


Figure 8. Distribution of critiques (prophet: 4KB perceptron; critic: 8KB tagged gshare).

The number of incorrect_disagree is larger than the number of correct_disagree, confirming the results of Section 7.2, in particular, those in Figure 6(c). As the number of future bits increases from 1 to 12, the number of incorrect_disagree increases by 20%, while the number of correct_disagree decreases by 40%. Also, the number of times the critic fails to override a prophet's mispredict (incorrect_agree) decreases by 43%. Note that the majority of critiques are correct_agree, which don't affect prediction accuracy.

The critic may also provide two implicit critiques when there is a miss in the filter: *correct_none* and *incorrect_none*, depending on the prophet's prediction of this branch. A miss in the filter occurs either the first time a new mispredicted branch context (branch address and BOR value combination) is found in the execution flow (no previous instance of the context was introduced in the critic before), or when a context that was mispredicted in the past, is mispredicted again after a long time (its entry in the critic was replaced by another context). The best case is that the critic filters those branches that the prophet correctly predicts, and that gives a critique for branches that the prophet misses. Table 4 shows the percentage of predictions that were filtered and the prophet's prediction (and, therefore, the final prediction) was correct or incorrect. The last row shows the sum of both correct_none and incorrect_none.

Increasing the filter size allows more contexts to be stored in its tag table. The results show that the percentage of total predictions without critiques, for either incorrect_none, correct_none, or the total, generally decreases slightly as the filter size increases. That is, there are more tag hits as the filter size increases.

Increasing the number of future bits is also beneficial to the filter, as it is better able to identify the contexts where the prophet mispredicted a branch. On average, the critic gives a critique to 1 out of every 3 branches using 1 future bit, and to 1 out of every 4 branches using 12 future bits. This is why, in Figure 8, the total number of critiques decreases as the number of future bits increases.

7.4. Processor performance

Figure 9 shows our performance results. To reduce simulation time, we only simulated one LIT per benchmark (the one that obtained the intermediate mispredict reduction of all LITs for that benchmark) and we simulated only the first 15 million IA32 instructions of each LIT.

We simulated three prophets (i. e., gshare, 2Bc-gskew, and perceptron) combined with a tagged gshare critic using 4, 8, and 12 future bits. The first bar in each group is a prophet alone with the same hardware budget as the prophet/critic hybrids.

The results show the same trend shown in previous sections for prediction accuracy: as the number of future bits increases, performance increases as well. With 4 future bits, speedups of 4.7% for gshare, 3.4% for 2Bc-gskew, and 2.7% for perceptron are obtained over the 16KB prophet alone. If we use 12 future bits in the critic, speedups grow to 8%, 7%, and 5.2%, respectively.

Figure 10 shows performance results classified per benchmark suite. These results correspond to the "2Bc-gskew + tagged gshare" bars shown in Figure 9. The prophet/critic hybrid always outperforms the 2Bc-gskew alone for all suites. Using a prophet/critic hybrid with 4 future bits, speedups over a 16KB 2Bc-gskew predictor range from 0.6% for FP00, to 3% for WEB, to 4.2% for INT00. Using 12 future bits, speedups grow to 1.7% for FP00, 6% for WEB, and 10.7% for INT00.

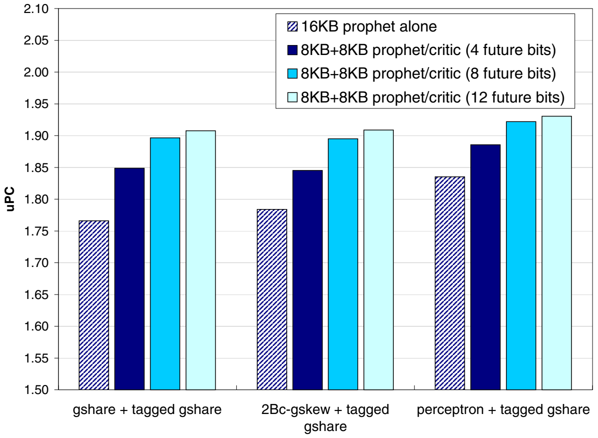


Figure 9. Average uPC of 16KB conventional predictors compared to 8KB+8KB prophet/critic hybrids using 4, 8, and 12 future bits.

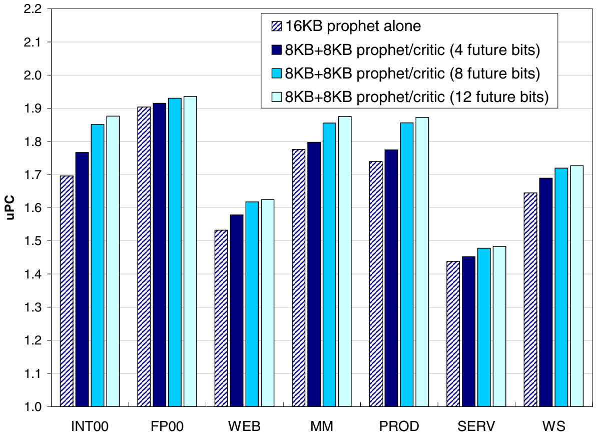


Figure 10. Average uPC for different benchmark suites using 4, 8, and 12 future bits (prophet: 8KB 2Bc-gskew; critic: 8KB tagged gshare).

8. Why it works—the theory

In 1996, Chen, Coffey, and Mudge [4] linked data compression to branch prediction.

Compressors typically operate in stream-mode, compressing each symbol as it is received. A predictor is used to generate a probability distribution for the next symbol. When the symbol is received, it is encoded according to the distribution. The predictor is then updated to generate the distribution for the next symbol. The better the predictor, the better the compression rate.

Conventional branch predictors and prophets in prophet/critic hybrids are similar to a stream-mode compressor's predictor. The branches are the symbols, and have two possible values: taken and not-taken. Each branch is predicted as it is encountered. The prediction is the symbol (i. e., branch direction) with the highest probability. After the prediction is made, the predictor is updated. Since the predictor operates in stream-mode, it is limited to using previously encountered branches (i. e., branch history) to generate the probability distribution. This limits its accuracy.

Critics, on the other hand, are different. They do not operate in stream-mode. When a branch is encountered, they wait until they have a few following branches before they provide its prediction. Because they wait, they can use a probability model (e. g., a Markov model) that generates a probability distribution for the branch using the outcomes of the branches both before and after it; that is, using both branch history and future. Since they are accessed with predicted branch future instead of actual future, they actually maintain a probability model for whether the prophet's prediction is wrong, rather than a model for the branch's outcome. However, the predicted outcome can be easily generated from the prophet's prediction plus the critic's prediction of whether it is wrong. Because critics do not operate in stream-mode, they can delay making their predictions, reducing the prophet's mispredicts by up to 45%.

9. Conclusion

Prophet/critic hybrid branch prediction is a general technique for reducing mispredicts. This paper shows the technique works and provides some insights as to why it works. The paper's intent is to provide information that can be used to build better predictors. However, the ultimate goal—at least when designing a processor—it to build the best predictor. To reach this goal, microarchitects should experiment with using different predictors as prophets and critics; for example, those proposed by Jiménez [13], Seznec [25], and Michaud and Seznec [21].

Acknowledgments

We thank the members of the Microarchitecture Research Lab for their help and comments on this work; especially Tom Holman for his help with the simulator used in this research. We would also like to express our deep gratitude to the folks of Intel Labs Barcelona for providing us with the help and infrastructure needed to finish this work.

This work was supported by the Ministry of Science and Technology of Spain under contract TIC–2001–0995–C02–01, the HiPEAC European Network of Excellence, and by an internship provided by Intel.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 35th Annual Intl. Symposium on Microarchitecture*, pages 423–434, Dec. 2003.

[2] P.-Y. Chang, M. Evers, and Y. N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *Proceedings of the Intl. Conference on Paral-*

lel Architectures and Compilation Techniques*, pages 48–57, Oct. 1996.

[3] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th Annual Intl. Symposium on Microarchitecture*, pages 252–257, Nov. 1995.

[4] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, Oct. 1996.

[5] A. N. Eden and T. N. Mudge. The YAGS branch prediction scheme. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 69–77, Oct. 1998.

[6] M. Evers, P.-Y. Chang, and Y. N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *Proceedings of the 23rd Annual Intl. Symposium on Computer Architecture*, pages 3–11, May 1996.

[7] M. Evers, S. J.Patel, R. S. Chapell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 52–61, June 1998.

[8] M. Evers and T.-Y. Yeh. Understanding branches and designing branch predictors for high-performance microprocessors. *Proceedings of the IEEE*, 89(11):1610–1620, Nov. 2001.

[9] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 122–131, June 1998.

[10] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):24–36, Oct. 1996.

[11] E. Hao, P.-Y. Chang, and Y. N. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proceedings of the 27th Annual Intl. Symposium on Microarchitecture*, pages 228–232, Dec. 1994.

[12] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual Intl. Symposium on Microarchitecture*, pages 142–152, Dec. 1996.

[13] D. A. Jiménez. Fast path-based neural branch prediction. In *Proceedings of the 35th Annual Intl. Symposium on Microarchitecture*, pages 243–252, Dec. 2003.

[14] D. A. Jiménez. Reconsidering complex branch predictors. In *Proceedings of the 9th Intl. Conference on High Performance Computer Architecture*, pages 43–52, Feb. 2003.

[15] D. A. Jiménez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual Intl. Symposium on Microarchitecture*, pages 67–76, Dec. 2000.

[16] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, Nov. 2002.

[17] S. Jourdan, T.-H. Hsing, J. Stark, and Y. N. Patt. The effects of mispredicted-path execution on branch prediction structures. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 58–67, Oct. 1996.

[18] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 155–166, June 1998.

[19] G. H. Loh and D. S. Henry. Predicting conditional branches with fusion-based hybrid predictors. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 165–176, Sept. 2002.

[20] S. McFarling. Combining branch predictors. Technical Report TN-36, Compaq Western Research Lab., June 1993.

[21] P. Michaud and A. Seznec. A comprehensive study of dynamic global history branch prediction. Technical Report 1406, IRISA, June 2001.

[22] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. *U.S. Patent Number 5.381.533*, Jan. 1995.

[23] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero. Fetching instruction streams. In *Proceedings of the 36th Annual Intl. Symposium on Microarchitecture*, pages 371–382, Nov. 2002.

[24] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th Annual Intl. Symposium on Computer Architecture*, pages 234–245, May 1999.

[25] A. Seznec. Redundant history skewed perceptron predictors: Pushing limits on global history branch predictors. Technical Report 1554, IRISA, Sept. 2003.

[26] A. Seznec, S. Felix, V. Hrishnan, and Y. Sazeides. Design tradeoffs for the EV8 conditional branch predictor. In *Proceedings of the 29th Annual Intl. Symposium on Computer Architecture*, pages 295–306, May 2002.

[27] A. Seznec and A. Fraboulet. Effective ahead pipelining of instruction block address generation. In *Proceedings of the 30th Annual Intl. Symposium on Computer Architecture*, pages 241–252, June 2003.

[28] A. Seznec and P. Michaud. Dealiased hybrid branch predictors. Technical Report PI-1229, IRISA, Feb. 1999.

[29] J. E. Smith. Is there anything more to learn about high performance processors? Keynote Address at the 2003 International Conference on Supercomputing, June 2003.

[30] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual Intl. Symposium on Computer Architecture*, pages 25–34, May 2002.

[31] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch prediction. In *Proceedings of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 170–179, Oct. 1998.

[32] L. N. Vintan and M. Iridon. Towards a high performance neural branch predictor. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 868–873, July 1999.

[33] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual Intl. Symposium on Computer Architecture*, pages 124–134, 1992.