Reducing the Power and Complexity of Path-Based Neural Branch Prediction

Gabriel H. Loh
College of Computing
Georgia Institute of Technology
loh@cc.gatech.edu

Daniel A. Jiménez
Department of Computer Science
Rutgers University
djimenez@cs.rutgers.edu

Abstract

A conventional path-based neural predictor (PBNP) achieves very high prediction accuracy, but its very deeply pipelined implementation makes it both a complex and power-intensive component. One of the major reasons for the large complexity and power is that for a history length of h, the PBNP must use h separately indexed SRAM arrays (or suffer from a very long update latency) organized in an h-stage predictor pipeline. Each pipeline stage requires a separate row-decoder for the corresponding SRAM array, inter-stage latches, control logic, and checkpointing support. All of these add power and complexity to the predictor.

We propose two techniques to address this problem. The first is modulo path-history *which decouples the branch outcome history length from the path history length allowing for a shorter path history (and therefore fewer predictor pipeline stages) while simultaneously making use of a traditional long branch outcome history. The pipeline length reduction results in decreased power and implementation complexity. The second technique is* bias-based filtering *(BBF) which takes advantage of the fact that neural predictors already have a way to track strongly biased branches. BBF uses the bias weights to filter out mostly always taken or mostly always not-taken branches and avoids consuming update power for such branches.*

Our proposal is complexity effective because it decreases the power and complexity of the PBNP without negatively impacting performance. The combination of modulo path-history and BBF results in a slight improvement in predictor accuracy of 1% for 32KB and 64KB predictors, but more importantly the techniques reduce power and complexity by reducing the number of SRAM arrays from 30+ down to only 4-6 tables, and reducing predictor update activity by 4-5%.

1. Introduction

After decades of academic and industrial research efforts focused on the branch prediction problem, pipeline flushes due to control flow mispredictions remain one of the primary bottlenecks in the performance of modern processors. A large amount of recent branch prediction research has centered around techniques inspired and derived from machine learning theory, with a particular emphasis on the *perceptron* algorithm [3, 4, 7–10, 14, 18]. These neural-based algorithms have been very successful in pushing the envelope of branch predictor accuracy.

Researchers have made a conscious effort to propose branch predictors that are highly amenable to pipelined and ahead-pipelined organizations to minimize the impact of predictor latency on performance. There has been considerably less effort on addressing power consumption and implementation complexity of the neural predictors. Reducing branch predictor power is not an easy problem because any reduction in the branch prediction accuracy can result in an overall increase in the *system* power consumption due to a corresponding increase in wrong-path instructions. On the other hand, peak power consumption, which limits the processor performance, and average power consumption, which impacts battery lifetime for mobile processors, are important design concerns for future processors [5]. Furthermore, it has been shown that the branch predictor, and the fetch engine in general, is a thermal hot-spot that can potentially limit the maximum clock frequency and operating voltage of the CPU, which in turn limits performance [16].

This paper focuses on the *path-based neural predictor* which is one of the proposed implementations of neural branch prediction [7]. In particular, this algorithm is highly accurate and pipelined for low effective access latency. We explain the organization of the predictor and the major sources of power consumption and implementation complexity. We propose a new technique for managing branch path-history information that greatly reduces the

number of tables, the pipeline depth, and the checkpointing overhead required for path-based neural prediction. We also propose a simple bias-based filtering mechanism to further reduce branch prediction power. While this paper specifically discusses the original path-based neural predictor [7], the techniques are general and can be easily applied to other neural predictors that use path history.

The rest of this paper is organized as follows. Section 2 provides an overview of the path-based neural predictor and discusses its power and complexity. Section 3 explains our proposed techniques for reducing the power consumption and implementation complexity. Section 4 presents the simulation-based results of our optimized path-based neural predictor in terms of the impact on prediction accuracy and power reduction. Section 5 concludes the paper.

2. Path-Based Neural Prediction

This section describes the original path-based neural predictor (PBNP), and then details the power and complexity issues associated with the PBNP.

2.1. Predictor Organization

The path-based neural predictor (PBNP) derives from the original *perceptron* branch predictor [9]. We define a vector $\overrightarrow{x} = \langle 1, x_1, x_2, ..., x_h \rangle$ where x_i is the ith most recent branch history outcome represented as -1 for a not taken branch and 1 for a taken branch. The branch history is the collection of taken/not-taken results for the h most recent conditional branches. The perceptron uses the branch address to select a set of weights $\overrightarrow{w} = \langle w_0, w_1, ..., w_h \rangle$ that represent the observed correlation between branch history bits and past branch outcomes. The sign of the dot-product of $\overrightarrow{w} \cdot \overrightarrow{x}$ provides the final prediction where a positive value indicates a taken-branch prediction. Figure 1a shows a block diagram of the lookup logic for the perceptron predictor.

At a high-level, the PBNP is very similar to the perceptron in that it computes a dot-product between a vector of weights and the branch history. The primary difference is that the PBNP uses a different branch address for each of the weights of $\overrightarrow{w}$. Let PC_0 be the current branch address, and PC_i be the ith most recent branch address in the *path history*. For the perceptron, each weight is chosen with the same index based on PC_0. For the PBNP, each weight w_i is chosen based on an index derived from PC_i. This provides path history information that can improve prediction accuracy, and spreading out the weights in different entries also helps to reduce the impact of inter-branch aliasing.

To implement the PBNP, the lookup phase is actually pipelined over many stages based on the overall path-/branch-history length. Figure 1b illustrates the hardware organization of the PBNP. For a branch at cycle t, the PBNP starts the prediction at cycle $t - h$ using PC_h. For each cycle after $t - h$, the PBNP computes partial sums of the dot-product of $\overrightarrow{w} \cdot \overrightarrow{x}$. Pipeline stage i contains the partial sum for the branch prediction that will be needed in i cycles. At the very end of the pipeline, the critical lookup latency consists of looking up the final weight and performing the final addition.

2.2. Power and Complexity

During the lookup phase of the PBNP, each pipeline stage reads a weight corresponding to the exact same PC. This is due to the fact that the current PC_0 will be next cycle's PC_1 and next-next cycle's PC_2 and so on. This allows an implementation where the weights are read in a single access using a single large SRAM row that contains all of the weights. During the update phase however, a single large access would force the update process to use a pipelined implementation as well. While at first glance this may seem desirable, this introduces considerable delay between update and lookup. For example a 30-stage update pipeline implies that even after a branch outcome has been determined, another 30 cycles must elapse before the PBNP has been fully updated to reflect this new information. This update delay can create a decrease in predictor accuracy. There are also some timing effects due to the fact that some weights of a branch will be updated before others.

An alternative organization uses h tables in parallel, one for each pipeline stage/history-bit position [7], as illustrated in Figure 1b. This organization allows for a much faster update and better resulting accuracy and performance. The disadvantage of this organization is that there is now a considerable amount of area and power overhead to implement the row decoders for the h separate SRAM arrays. Furthermore, to support concurrent lookup and update of the predictor, each of these SRAM arrays needs to be dual-ported (one read port/one write port) which further increases the area and power overhead of the SRAM row decoders. To use the PBNP, the branch predictor designer must choose between an increase in power and area or a decrease in prediction accuracy.

On a branch misprediction, the PBNP pipeline must be reset to the state that corresponded to the mispredicting branch being the most recent branch in the branch

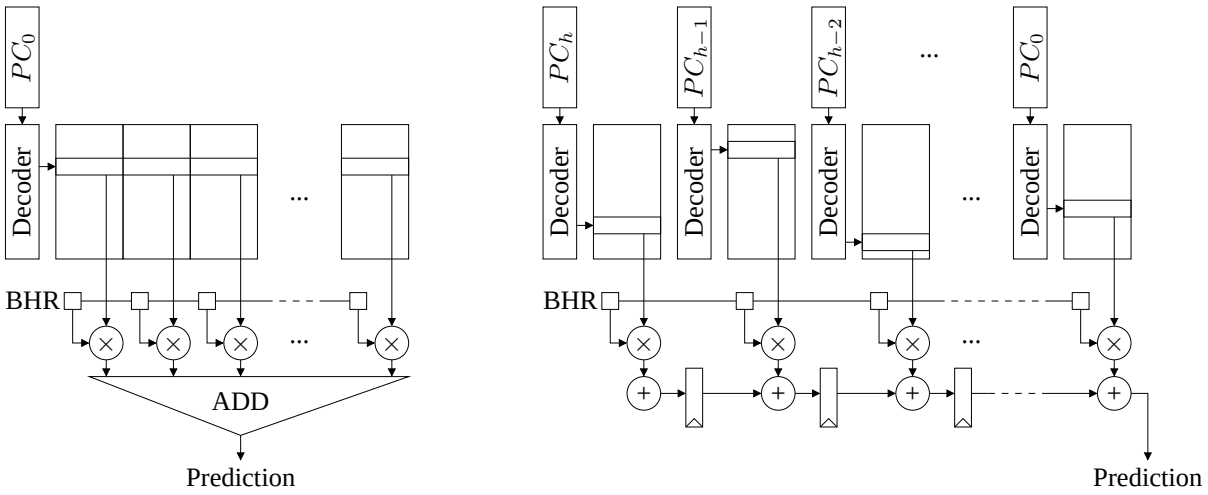

Figure 1: (a) Organization of the lookup logic for the perceptron branch predictor. (b) Lookup logic for the pipelined path-based neural branch predictor.

and path history. To support this predictor state recovery, each branch must checkpoint all of the partial sums in the PBNP pipeline. On a branch misprediction, the PBNP restores all of the partial sums in the pipeline using this checkpointed state. For b-bit weights and a history length of h, a PBNP checkpoint requires approximately bh bits of storage. The total number of bits is slightly greater because the number of bits required to store a partial sum increases as the sum accumulates more weights. The total storage for all checkpoints corresponds to the maximum number of in-flight branches permitted in the processor. For example, assuming one branch occurs every five instructions, then a 128-entry ROB would on average have 25 branches in flight. This means the PBNP checkpoint table must have about 25 entries to support the average number of branches. To avoid stalls due to a burst of branch instructions, the checkpoint table may need to be substantially larger. For proposals of very-large effective instruction window processors such as CFP [17], the checkpointing overhead further increases.

The checkpointing overhead represents additional area, power, and state that is often unaccounted for in neural predictor studies. This overhead increases with the history/path-length of the predictor since the PBNP must store one partial sum per predictor stage. A further source of complexity is the additional control logic required to manage the deeply pipelined predictor.

3. Reducing Perceptron Power and Complexity

In this section, we propose two techniques for reducing the power and complexity of the path-based neural predictor. Modulo-Path History is a new way to manage path-history information which also provides a new degree of freedom in the design of neural predictors. Bias-Based Filtering is a technique similar to previously proposed filtering mechanisms that takes advantage of the information encoded in the neural weights to detect highly biased branches.

3.1. Modulo-Path History

In the original PBNP, the path history length is always equal to the branch history length. This is a result of using PC_i to compute the index for the weight of x_i. As described in the previous section, the pipeline depth directly increases the number of tables and the checkpointing overhead required. On the other hand, supporting a long history length requires the PBNP to be deeply pipelined.

We propose *modulo path-history* where we decouple the branch history length from the path history length. We limit the path history to only the $P < h$ most recent branch addresses. Instead of using PC_i to compute the index for w_i, we use $PC_{i \mod P}$. In this fashion, we can reduce the degree of pipelining down to only P stages. Figure 2a shows the logical predictor organiza-

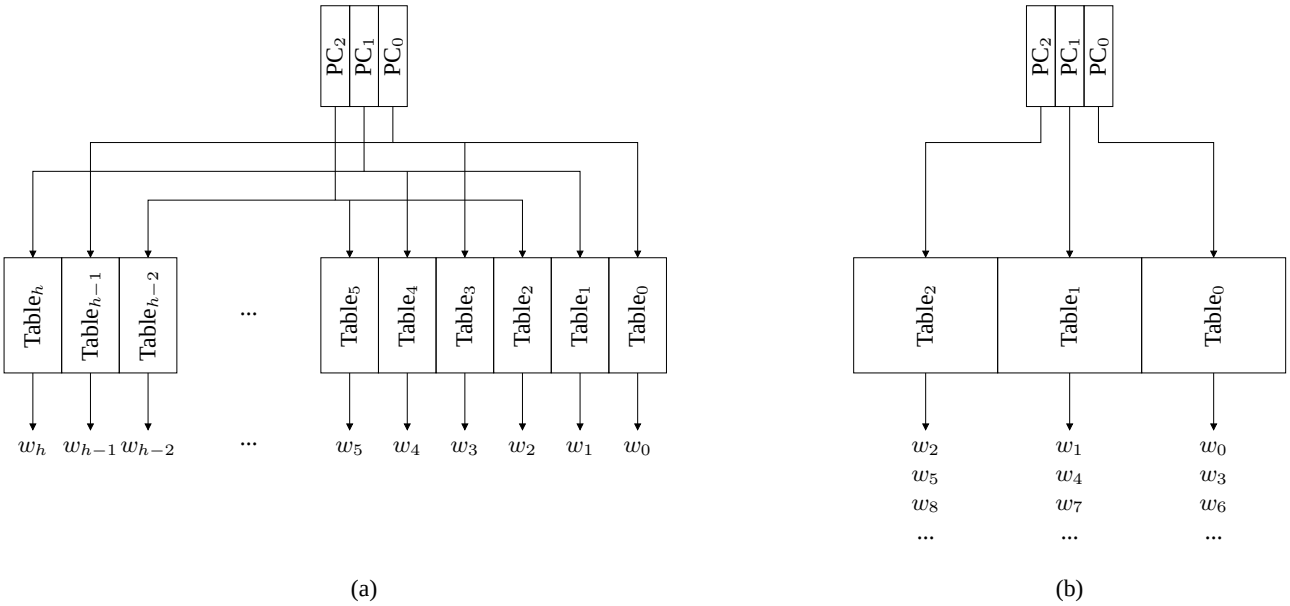

Figure 2: (a) Logical organization of a PBNP using modulo path-history. (b) Physical organization of a PBNP using modulo path-history for $P = 3$.

tion of a PBNP using modulo path-history (for $P = 3$). Since every Pth weight is indexed with the same branch address, we can interleave the order of the weights in the table such that only P tables are necessary. Figure 2b shows how each table provides weights that correspond to h/P branch history outcomes, where each branch history outcome is separated by P bit positions.

By reducing the PBNP implementation to only use P distinct tables, we address several of the main sources of power and complexity as described in Section 2. Using only P tables reduces the duplicated row-decoder overhead. The reduction in the number of tables reduces the overall pipeline depth of the predictor which reduces the amount of state that must be checkpointed (i.e. there are only P partial sums). With fewer stages, the control logic for the predictor pipeline can also be reduced. The number of inter-stage latches and associated clocking overhead is also correspondingly reduced.

Modulo path-history may also make the single-table implementation feasible. The pipelined update still adds latency to the update phase of the branch predictor, but the update latency has been reduced from $O(h)$ cycles down to $O(P)$ cycles. For sufficiently small values of P, the substantial reduction in complexity and associated power may justify the small increase in the update latency.

Modulo path-history is a unique way to manage the branch path history information. A PBNP can now choose between different lengths of branch and path history. Tarjan and Skadron proposed a comprehensive taxonomy of neural branch predictor organizations that can describe a very wide variety of neural predictor variations [18]. Nevertheless, modulo path-history is a new addition that does not fall into any of their categories.

3.2. Bias-Based Filtering

Earlier branch prediction studies have made the observation that there are a large number of branch instructions whose outcomes are almost always in the same direction [2, 6]. Some of this research has proposed various ways for detecting these strongly biased branches and removing or *filtering* them out to reduce the amount of interference in the branch prediction tables. We make the observation that from an energy and power perspective, keeping track of h distinct weights and performing an expensive dot-product operation is an overkill for these easy-to-predict branches. We also observe that the family of neural predictors have built-in mechanisms for detecting highly-biased branches. Combining these observations, we proposed *Bias-Based Filtering* (BBF).

The BBF technique is simple in principal. We consider a branch whose bias weight (w_0) has saturated (equal to maximum or minimum value) as a highly-biased branch. When the predictor detects such a branch, the prediction is determined only by the bias weight as opposed to the entire dot-product. If this prediction turns out to

be correct, the predictor skips the update phase which saves the associated power and energy. BBF does not reduce the lookup power because the pipelined organization must start the dot-product computation before the predictor knows whether the branch is highly biased. Besides the power reduction, BBF has a slight accuracy benefit because the act of filtering the strongly biased branches reduces the interference among the remaining branches.

The relatively long history lengths of neural predictors combined with the usage of multi-bit weights results in a table that has relatively few entries or rows. This greatly increases the amount of inter-branch aliasing in the tables which potentially reduces the effectiveness of BBF. To address this, we propose that the bias table uses a larger number of entries than any of the other tables. This makes sense since the bias table now has to keep track of all of the strongly biased branches as well as provide the bias weights for the regular branches.

To increase the number of strongly biased branches covered by BBF, we modify the neural prediction lookup slightly such that the bias weight (and only the bias weight) is indexed in a gshare fashion (xor of branch address and branch history). This improves the filtering mechanism by allowing the bias table to detect branches that are strongly biased but only under certain global history contexts. We also reduce the width of the bias weights to 5 bits which allows the bias weights to saturate more quickly and start filtering strongly biased branches sooner.

4. Performance and Power Results

In this section, we present the simulation results for an optimized PBNP predictor that uses modulo path-history and bias-based filtering.

4.1. Simulation Methodology

For our prediction accuracy results, we used the in-order branch predictor simulator sim-bpred from the SimpleScalar toolset [1]. We simulated all twelve SPECint applications using the reference sets and single 100M instruction simulation points chosen by SimPoint 2.0 [13]. Our applications were compiled on an Alpha 21264 with Compaq `cc` with full optimizations.

4.2. Impact of Modulo-Path History

To measure the impact of modulo-path history, we started with the original PBNP where there are $P = h$ tables that each provide a single weight corresponding to a single branch history position. We then increased the number of weights provided by each table by setting $P = h/2, h/3$, and so on. This reduces the length of the path-history while maintaining a constant branch history length. Figure 3 shows the impact on prediction accuracy as we vary the number of weights per lookup table over a range of predictor sizes. For the smaller predictors (2KB and 4KB), there is an initial increase in the misprediction rate when we add modulo path-history. For predictors 16KB and larger, the increase in the misprediction rate is less than 0.5% (8KB) and in some cases even *improve* prediction accuracy by a small amount (0.1% for 64KB).

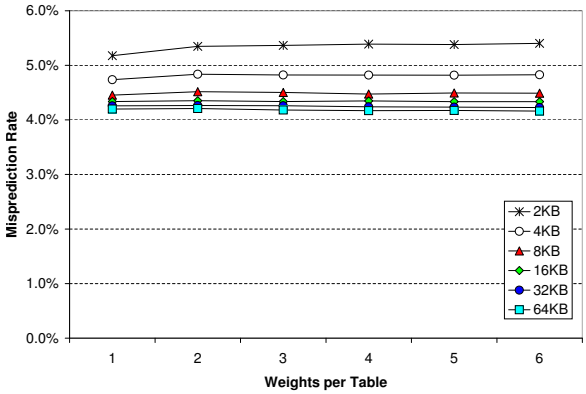

Figure 3: Average misprediction rates on SPECint when using modulo path-history.

As we increase the number of weights per table, the total number of tables decreases. This reduces the power and energy cost per access due to a reduction in the number of row decoders in the entire predictor. The power cost per table lookup increases with the number of weights per table, but the number of tables decreases. The modulo path-history approach for managing path history in the PBNP is overall performance-neutral while providing a power benefit by reducing the power consumed per lookup. We have not quantified the exact power benefit in Watts due to limitations of CACTI-like tools. We also have not quantified in detail the reduction of the checkpointing overhead or the impact of simplifying the control logic for the reduced pipeline depth, but simply observe that there will be some power and complexity benefit.

4.3. Impact of Bias-Based Filtering

For a fixed hardware budget, increasing the number of entries in the bias table forces the remaining tables to be decreased in size. We evaluated a range of table sizes. Figure 4 shows the prediction accuracy impact of dedicating some more weights to the bias table while reduc-

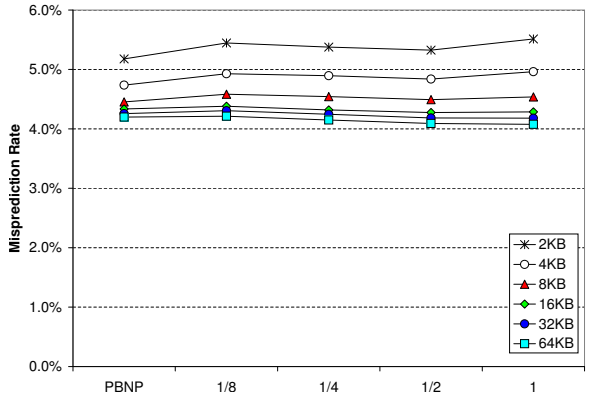

Figure 4: Average misprediction rates on SPECint when using bias-based filtering.

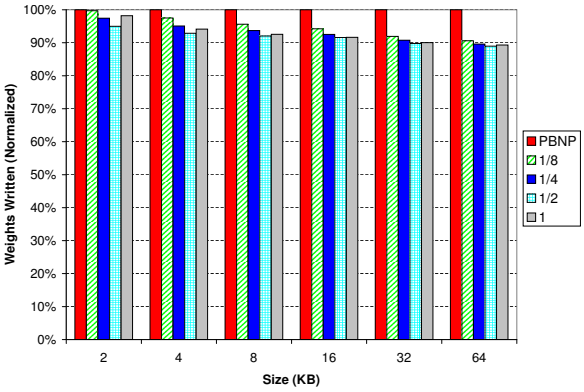

Figure 5: Average reduction in update activity for a PBNP using bias-based filtering.

ing the size of the other tables. These results include the Bias-Based Filtering effects. The left-most set of points in Figure 4 correspond to the baseline PBNP. The remaining configurations use BBF where "$1/n$" indicates that the bias table has X/n entries, where X is the number of bytes in the table. For example, the 1/4 configuration for an 8KB budget has a bias table with 8K/4 = 2K entries (*not* 2KB worth of entries). Similar to the modulo path-history results, BBF is less effective at the smallest predictor sizes, and is relatively performance-neutral at larger sizes. For predictors sized 16KB and greater, BBF actually results in a slight (1-3%) decrease in mispredictions. The reason for this slight accuracy benefit is that gating updates for highly-biased branches creates an interference-reducing effect similar to a partial update policy [12].

The primary purpose of BBF is to reduce the number of weights written to the tables during the update phase. Figure 5 shows the reduction in the number of weights written as compared to the baseline PBNP. Overall, the mid-sized to larger sized predictors achieve the greatest benefit, with about a 10% reduction in the update activity.

4.4. Impact of Power-Reduced Path-Based Neural Predictor

In the previous subsections, we have shown how the techniques of modulo path-history and bias-based filtering are relatively performance-neutral for mid-sized predictors and performance-beneficial for larger predictors. Simultaneously, these techniques provide a reduction in the predictor's power consumption by reducing the power per access and reducing the total number of accesses, and a reduction in the implementation complexity by reducing

	Baseline PBNP	With Mod. Path and BBF		
Predictor Size	History Length	History Length	Path Length	Bias Weights
2KB	17	17	4	1K
4KB	25	24	4	2K
8KB	31	29	4	4K
16KB	32	33	5	8K
32KB	42	42	3	16K
64KB	47	42	3	32K

Table 1: Parameters for the baseline PBNP and a PBNP using both modulo path-history and bias-based filtering.

the number of predictor pipeline stages. We now observe the effects of combining the two techniques. Table 1 lists the final configurations used for the PBNP with modulo path-history and bias-based filtering, as well as the baseline PBNP configurations.

Figure 6 shows the average misprediction rate for a conventional PBNP and a PBNP augmented with modulo path-history and bias-based filtering. Similar to the individual results, our techniques are not recommended for small predictor sizes. At 16KB the techniques do not help or hurt accuracy, and at 32KB and 64KB they provide a small accuracy benefit (about 1%).

As discussed earlier, the modulo path-history reduces power by reducing the number of tables and the reducing the pipeline depth of the predictor. BBF reduces the number of table updates. Figure 7 shows the relative decrease in update activity when compared to a conventional PBNP. Note that for the 2KB predictor size, the activity actually *increases*. This is due to the fact that for the smaller predictor, the slight decrease in prediction accuracy causes the neural prediction algorithm to train more frequently which causes more overall activity in the table

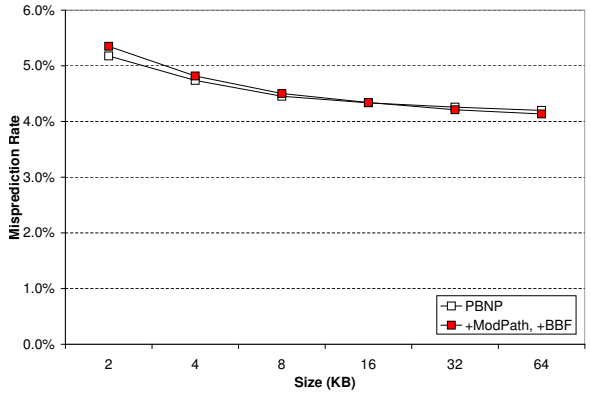

Figure 6: Average SPECint misprediction rate for the baseline PBNP and a PBNP using both modulo path-history and bias-based filtering.

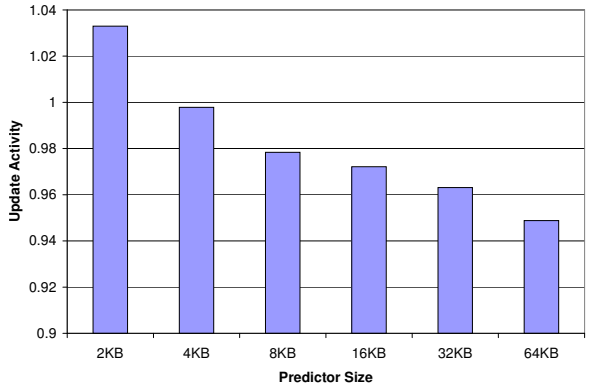

Figure 7: Change in the update activity of a PBNP using modulo path-history and bias-based filtering as compared to a conventional PBNP.

of weights. At the larger sizes, BBF reduces the frequency of updates by 4-5%.

4.5. Overall Impact on Implementation Complexity

The original goal of this research was to redesign a path-based neural predictor to be less complex and hence easier to implement. Table 2 summarizes the overall benefits in terms of key sources of implementation complexity. Modulo-path history reduces the number of separate SRAM arrays from 18-48 down to only 4-6. The reduction in the table count is directly correlated to the depth of pipelining needed to implement a PBNP. This 72-92% reduction of the predictor pipeline length greatly simplifies the control logic needed to control stalling, checkpointing and recovering the predictor pipeline.

The extra storage needed for checkpointing the predictor at every branch impacts the physical design of the predictor. Table 2 lists the number of bits required per checkpoint for copying the partial sums. The bit counts are slightly underestimated because we assumed that every stage only needs an 8-bit value to simplify the arithmetic. In practice the bit-width increases with the number of accumulated weights. The original PBNP needs 144-384 bits of information checkpointed on every branch, while using modulo-path history reduces this to only 32-48 bits per branch. The checkpointing overhead reduction not only reduces the size of the SRAM needed to store the checkpoints, but it also reduces the number of wires entering and leaving the predictor for the checkpoints. To reduce the impact on the physical layout and latency of the predictor, the checkpoint SRAM may be placed slightly further away from the main predictor. This physical separation requires longer wires (more capacitance) which results in increased power consumption for the communication between the predictor and the checkpoint SRAM.

Modulo path-history and BBF impact the power consumed by the predictor itself and also the overall system-wide power. In this study, we did not quantify the exact power benefits because SRAM latency/power estimation tools such as CACTI [15] and eCACTI [11] do not handle the non-power-of-two SRAM sizes used in this study. For the non-SRAM portions of the predictors such as the adders, pipeline latches, control logic, and communication between the main predictor and the checkpoint SRAM, we would need a detailed physical design and layout to even begin to accurately estimate the overall power impact. This level of analysis is beyond the scope of this paper, but will be examined in future research.

5. Conclusions

We have introduced two new techniques for reducing the complexity and power of path-based neural branch predictors. While this study has focused on the original path-based neural predictor, our proposal can apply to any of the similar neural prediction algorithms such as the hashed perceptron [18] or the piecewise-linear branch predictor [8]. We have shown that the combination of modulo path-history and bias-based filtering can reduce power by decreasing the total number of tables used by the predictor as well as reducing the activity factor of the update phase of prediction. The modulo path-history technique also reduces the implementation complexity of the path-based neural predictor by reducing the predictor pipeline depth to only 4-6 stages, as opposed to 18-48 stages for the original predictor.

While this study has focused on a conventional path-

Predictor Size	PBNP		With Mod. Path and BBF		% Reduction
	Num SRAM Arrays	Bits per Checkpoint	Num SRAM Arrays	Bits per Checkpoint	
2KB	18	144	5	40	72.2
4KB	26	208	5	40	80.8
8KB	32	256	5	40	84.4
16KB	33	264	6	48	81.8
32KB	43	344	4	32	90.7
64KB	48	384	4	32	91.7

Table 2: Impact on predictor pipeline depth and checkpointing overhead. The number of SRAM arrays is equal to the path length, plus one for the bias table. Checkpoint overhead estimates assume one 8-bit value per predictor pipeline stage.

based neural predictor, other similar predictors could also benefit from either or both modulo path-history and BBF. The piecewise-linear neural branch predictor is a generalization of the PBNP that computes m different PBNP summations in parallel [8]. These m parallel computations increase the complexity of a deeper pipelined predictor, and modulo-path history may be very useful in this context to keep that complexity under control. The m computations also require m times as many weights to be updated, and bias-based filtering may also be very useful to reduce the activity of the piecewise-linear predictor. There are likely other predictors designs that can make use of the ideas presented in this study.

Acknowledgements

Gabriel Loh is supported by funding and equipment from Intel Corporation. Daniel Jiménez is supported by a grant from NSF (CCR-0311091) as well as a grant from the Spanish Ministry of Education and Science (SB2003-0357).

References

[1] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.

[2] Po-Yung Chang, Marius Evers, and Yale N. Patt. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 48–57, 1996.

[3] Veerle Desmet, Hans Vandierendonck, and Koen De Bosschere. A 2bcgskew Predictor Fused by a Redundant History Skewed Perceptron Predictor. In *Proceedings of the 1st Championship Branch Prediction Competition*, pages 1–4, Portland, OR, USA, December 2004.

[4] Hongliang Gao and Huiyang Zhou. Adaptive Information Processing: An Effective Way to Improve Perceptron Predictors. In *Proceedings of the 1st Championship Branch Prediction Competition*, pages 1–4, Portland, OR, USA, December 2004.

[5] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovitz, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C. Valentine. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal*, 7(2), May 2003.

[6] Dirk Grunwald, Donald Lindsay, and Benjamin Zorn. Static Methods in Hybrid Branch Prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 222–229, Paris, France, October 1998.

[7] Daniel A. Jiménez. Fast Path-Based Neural Branch Prediction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 243–252, San Diego, CA, USA, December 2003.

[8] Daniel A. Jiménez. Piecewise Linear Branch Prediction. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

[9] Daniel A. Jiménez and Calvin Lin. Neural Methods for Dynamic Branch Prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.

[10] Gabriel H. Loh. The Frankenpredictor. In *Proceedings of the 1st Championship Branch Prediction Competition*, pages 1–4, Portland, OR, USA, December 2004.

[11] Manhesh Mamidipaka and Nikil Dutt. eCACTI: An Enhanced Power Estimation Model for On-Chip Caches. TR 04-28, University of California, Irvine, Center for Embedded Computer Systems, September 2004.

[12] Pierre Michaud, Andre Seznec, and Richard Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, Boulder, CO, USA, June 1997.

[13] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, New Orleans, LA, USA, September 2004.

[14] Andrè Seznec. Revisiting the Perceptron Predictor. PI 1620, Institut de Recherche en Informatique et Systèmes Aléatoires, May 2004.

[15] Premkishore Shivakumar and Norman P. Jouppi. CACTI 3.0: An Integrated Timing, Power, and Area Model. TR 2001/2, Compaq Computer Corporation Western Research Laboratory, August 2001.

[16] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-Aware Microarchitecture: Modeling and Implementation. *Transactions on Architecture and Code Optimization*, 1(1):94–125, March 2004.

[17] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual Flow Pipelines. In *Proceedings of the 11th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Boston, MA, USA, October 2004.

[18] David Tarjan and Kevin Skadron. Merging Path and Gshare Indexing in Perceptron Branch Prediction. CS 2004-38, University of Virginia, December 2004.