Int J Parallel Prog (2008) 36:267–286
DOI 10.1007/s10766-007-0063-0

Modulo Path History for the Reduction of Pipeline Overheads in Path-based Neural Branch Predictors

Gabriel H. Loh · Daniel A. Jiménez

Received: 1 November 2006 / Accepted: 2 April 2007 / Published online: 24 January 2008

Abstract Neural-inspired branch predictors achieve very low branch misprediction rates. However, previously proposed implementations have a variety of characteristics that make them challenging to implement in future high-performance processors. In particular, the path-based neural predictor (PBNP) and the piecewise-linear (PWL) predictor require deep pipelining and additional area to support checkpointing for misprediction recovery. The complexity of the PBNP predictor stems from the fact that the path history length, which determines the number of tables and pipeline stages, is equal to the history length, which is typically very long for high accuracy. We propose to decouple the path-history length from the outcome-history length through a new technique called *modulo-path* history. By allowing a shorter path history, we can implement the PBNP and PWL predictors with significantly fewer tables and pipeline stages while still exploiting a traditional long branch outcome history.

Keywords Computer architecture · Branch prediction

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

G. H. Loh (✉)
College of Computing, Georgia Institute of Technology, 266 Ferst Drive, Atlanta,
GA 30332-0765, USA
e-mail: loh@cc.gatech.edu

D. A. Jiménez
Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway,
NJ 08854-8019, USA
e-mail: djimenez@cs.rutgers.edu

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 [1–7]. 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 [8] 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, both of which are now first-class design considerations in high-performance microprocessors. Reducing branch predictor power is not an easy problem because any resultant 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 [9].

The goal of this work is to demonstrate that the complexity of neural branch predictors can be substantially reduced without altering the fundamental behaviors and characteristics of the prediction algorithm. In this article, we will *not* re-argue the benefits of the conventional neural predictors as that has already been demonstrated in numerous previous works [1–4,6,7].

In the rest of this article, we will first review the design of neural-based branch predictors in Sect. 2. Section 3 details our new branch predictor organizations targeted at reducing complexity and overall energy consumption. Section 4 presents our experimental results demonstrating the impact of our proposal on processor performance and energy. Section 5 describes how to extend the proposed ideas to the more recent piecewise-linear neural predictor, and Sect. 6 concludes the article.

2 Neural Branch Predictors

In this section, we review neural-based branch predictors. We then qualitatively describe the sources of complexity and power consumption that make conventional neural predictors difficult to implement in high-performance processors.

2.1 Background

Traditional PHT-based (pattern history table) branch predictors such as GAs [10] and gshare [11] do not scale gracefully with longer branch history lengths. For h bits of branch history, a conventional PHT needs a table size exponential in h. The neural predictors are interesting because they can exploit deep correlations from very long history lengths with subexponential scaling.

The basic perceptron predictor [1] employs a vector of weights that learns correlations between the branch direction and the results of previous branches. Figure 1a shows how a table of weights is indexed by the program counter (PC) to choose a single vector of weights that is then combined with the branch history register in a dot-product operation (each • represents conditionally negating the weight depending on the direction of the corresponding branch history bit). Conceptually, a

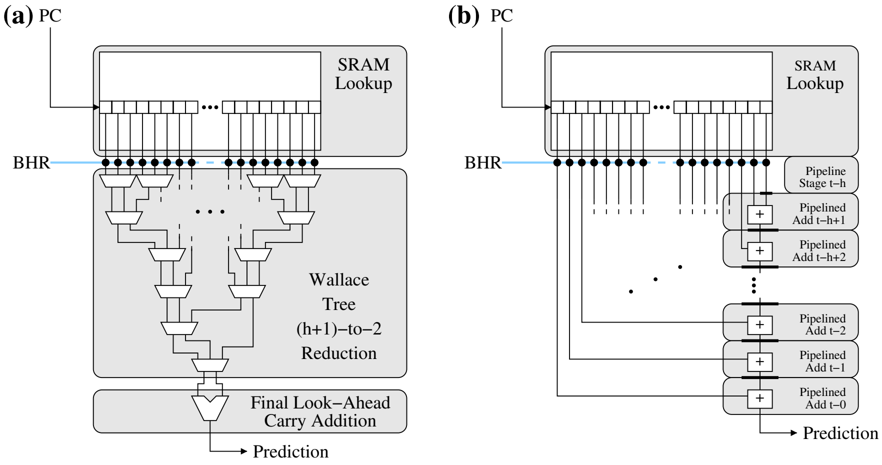

Fig. 1 (**a**) The perceptron branch predictor and (**b**) the path-based neural predictor or PBNP

past branch that is strongly correlated with the outcome of the current branch will have a corresponding weight with a large magnitude. The perceptron trains (increments/decrements) the weights to predict only according to those branches in the history that have exhibited strong correlations to the branch under consideration. A Wallace tree reduces the $h + 1$ weights down to only two weights in $O(\log_{3/2} n)$ carry-save adder gate delays. A final carry-completing adder such as a look-ahead carry adder computes the final sum. The sign of this resulting sum indicates the final prediction. The main obstacle to implementing a perceptron branch predictor is the long latency required to read the weights and then perform the large dot-product operation. To reduce the latency of the predictor, it may be necessary to implement the adders with fast, leaky transistors that end up consuming more dynamic and static power.

The second-generation path-based neural predictor (PBNP) largely solves the latency problems of the original perceptron [2]. The central idea is that the ith previous branch address (i.e., from the path history) can be used to look up the weight corresponding to the ith oldest branch i cycles ahead of time. While this largely addresses the latency issues of the perceptron predictor, the PBNP still suffers from significant implementation complexity. As shown in Fig. 1b, the pipelining of the PBNP provides a much faster effective predictor latency: the critical path is now the table lookup and a single addition. The PBNP also requires a number of adders equal to the depth of the branch history, further increasing the hardware cost.

The third-generation piecewise-linear (PWL) neural predictor is a generalization of the previous two neural predictors. That is the original perceptron and the PBNP are both special cases of the PWL predictor. The PWL predictor can learn more complex decision functions by aggregating multiple linear decision surfaces in a piecewise manner [3]. The PWL predictor effectively consists of k parallel PBNP pipelines each of which handles one of the linear decision surfaces. However, the k pipelines further increase the implementation overhead.

2.2 Power and Complexity

In this article, we will first focus on the path-based neural predictor (PBNP). The original perceptron predictor's long lookup latency makes it difficult to implement without making use of an overriding predictor organization [12] that just adds more power and complexity. The PBNP arguments presented below hold for the piecewise linear (PWL) predictor as well, except most overheads are further increased by a factor of k.

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 the next branch's PC_1 and next–next branch'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 odd 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 [2], as shown in Fig. 2a. 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. Using a large number of small SRAM arrays makes it more difficult to derive energy savings through SRAM banking and sub-banking techniques [13].

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 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. 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. The combination of these issues makes the conventional PBNP difficult to implement for high-performance microprocessors.

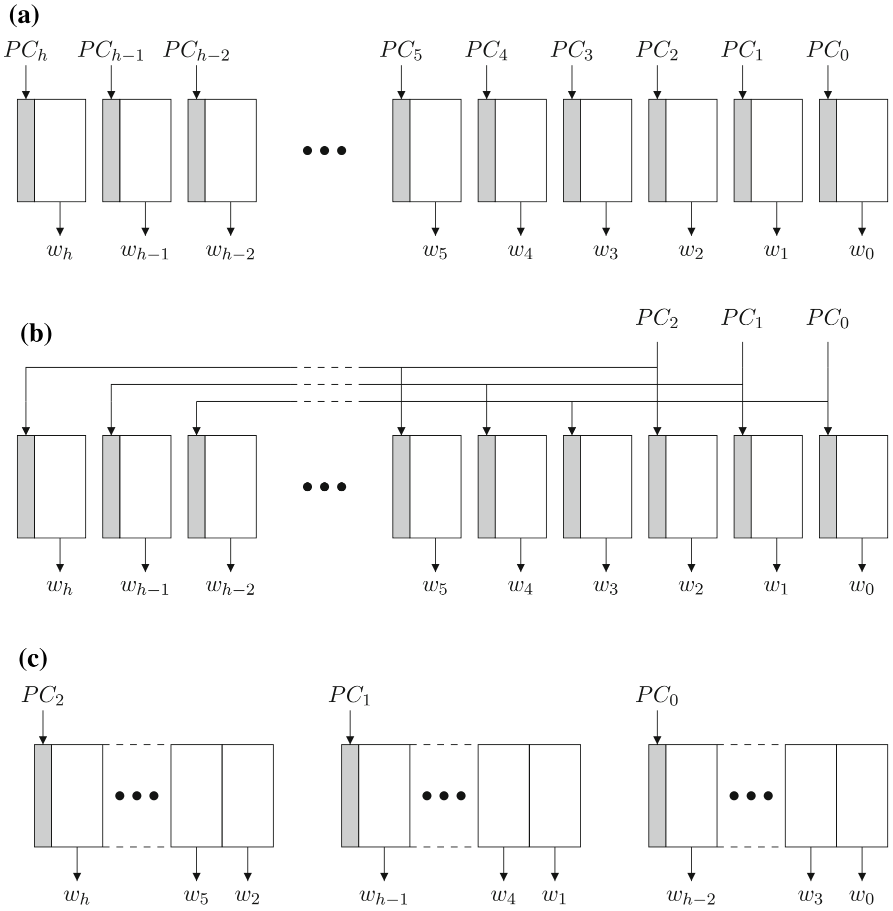

Fig. 2 (**a**) Organization of the h tables of the PBNP, (**b**) logical organization of a PBNP using modulo path-history for $P = 3$, and (**c**) the corresponding physical organization of the same. The shaded portion represents the SRAM row decoder and related access logic

3 Decoupling the Path and Branch History Lengths

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.

3.1 Modulo-path History

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 $PC_{i \bmod P}$. In this fashion, we can reduce the degree of pipelining down to only P stages. Figure 2b shows the logical organization of a PBNP using modulo path-history (for $P = 3$). In this example, we only use a path length of three, but Fig. 2b still appears to use $O(h)$ separate tables. Since every Pth weight is indexed with the same branch address, we can interleave the order of the weights in the tables such that only P tables are necessary. Figure 2c shows the physical organization where 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 Sect. 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 total bits of state that must be checkpointed (i.e., there are only P partial sums). The number of inter-stage latches and associated clocking overhead is also correspondingly reduced.

3.2 Folded Modulo-path History

Reducing the path-history length of a path-based neural predictor may reduce prediction accuracy because the long path history may provide additional context for detecting correlations. That is, a branch prediction may be highly correlated to the kth address in the branch history, but reducing the path length to $P < k$ by using modulo-path history eliminates this source of correlation. To recapture the correlation in the kth path address, we would need to increase the modulo-path history length to at least k (so indexes are computed using $PC_{i \bmod k}$ rather than $PC_{i \bmod P}$). This unfortunately increases the predictor pipeline depth and the associated power and complexity.

We propose to decouple the predictor pipeline depth from the path-history length by using *folded modulo-path* history. Similar to the normal modulo-path history, our PBNP uses only P tables, but now we employ a path-history length that is $P \cdot f$ addresses long, where f is the *folding factor*. In a conventional PBNP we only use one branch address to compute an index for each table. With folded modulo-path history, we use f addresses hashed together. Folded modulo-path history can be considered the path-history analogue of the folded long branch outcome histories used in other predictors such as 2bc-gskew [14]. Figure 3 shows an example PBNP with $P = 4$ and $f = 3$ for a total path length of 12 while only using four tables.

To combine the f path addresses into a single index, we used a simple XOR and shift-based hash function. For each of the path addresses $PC_i, i \in \{0 \dots f - 1\}$ used to index a table, we hash the addresses by taking the exclusive-OR of $PC_i \ll i$. This is similar to the hash function used by Stark et al. for their path hashing [15].

Note that the folded-modulo-path history predictor needs a shift register to track the path-history, and this shift register will need to be checkpointed for misprediction recoveries. However, the size of this shift register is relatively small due to the reduced number of predictor stages enabled by the modulo history and each entry only needs to store enough bits to index into the perceptron table (e.g., a 128-entry SRAM only requires seven bits from each branch address).

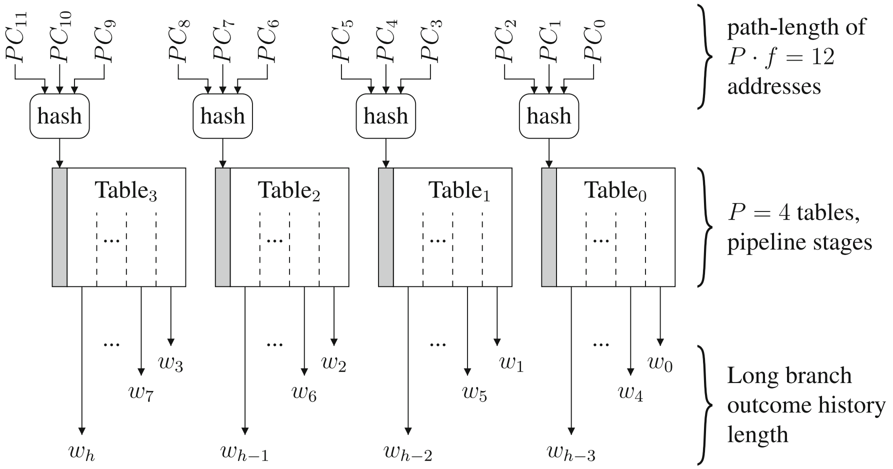

Fig. 3 A neural predictor employing folded modulo-path history, where the path-length ≠ history-length ≠ number of tables

Modulo path-history is a unique way to manage the path history information. A PBNP can now choose between different lengths of branch and path history. Tarjan and Skadron proposed a "hashed" perceptron indexing scheme that removed the rigid relationship between history length and the number of tables [6]. Seznec's GEHL predictors use a similar hashing approach to map multiple bits of branch history to a single correlation weight [16]. Our work provides a different way of separating history length from table count, and also makes the contribution of separating the path length from either of these parameters as well.

3.3 Generality of the Techniques

In this article, we focus on the path-based neural predictor. However, the proposed history-folding techniques can potentially be applied to other predictor organizations. Multi-table, ahead-pipelined predictors such as the Hashed-Perceptron [6], GEHL [16], PPM [17] or TAGE [18] could all include folded history to incorporate additional information and context in their indexing functions.

4 Results

In this section, we present our experimental results to demonstrate the merits of our proposed branch predictor organizations.

4.1 Experimental Methodology

For our initial design space exploration, we used the in-order branch predictor simulator sim-bpred from the SimpleScalar toolset [19]. We simulated applications from the

Table 1 The processor configuration used for our IPC simulations

Parameter	Value	Parameter	Value
Machine	4-wide	Integer units	ALU:2, Mult:2
IFQ size	8 entry	FP units	Add:1, Mult:1, Div:1
Scheduler	24 entry	Latencies	Same as Pentium-M [24]
LSQ size	24 entry	Memory ports	2
ROB size	64 entry	ITLB/DTLB	64 entry each
IL1, DL1	16 KB/4-way	Branch penalty	13 cycles
Unified L2	512 KB/8-way	DRAM latency	200 cycles

SPEC2000cpu integer benchmark suite with reference inputs, MiBench [20] with the large inputs and the MediaBench [21] multimedia benchmark suites with expanded inputs. Some applications (e.g., the `lame` MP3 encoder) are not included because we could not compile them in our Alpha environment due to unsupported libraries. We used 100 million instruction simulation points chosen by SimPoint 2.0 [22]. Our applications were compiled on an Alpha 21264 with Compaq `cc` with full optimizations. For our IPC simulations, we used the MASE simulator from SimpleScalar 4.0 [23]. We simulated a four-wide out-of-order processor; the details are listed in Table 1. The processor parameters were chosen to model a machine with a level of aggressiveness similar to an Intel Pentium-III/Pentium-M microarchitecture.

We used CACTI 3.2 [25] to estimate the energy consumption of 90 nm implementations of the branch predictors. Since CACTI does not simulate tables with non-power-of-two numbers of entries, we simply rounded-up the sizes of our structures to the next largest power of two. While this introduces some slight overestimation in power consumption, a realistic implementation of a neural branch predictor would likely use SRAMs with a power-of-two number of entries. We used CACTI to estimate the energy consumption of the predictor's tables of weights as well as the checkpoint tables. We also extrapolated the energy consumption of the predictors' adders based on the logic model of CACTI's row decoders. For our predictor die-area estimates, we use the register bit equivalent (rbe) methodology proposed by Mulder et al. [26].

We simulated a large number of PBNP configurations to find the best parameter settings. For the modulo-path and folded-modulo-path versions, we maintained the same branch history length and number of entries per table of weights (i.e., same total number of perceptrons) while allowing the predictor pipeline depth and path history length to vary. The final configurations are listed in Table 2. We could have potentially improved the performance of the modulo-path versions by allowing the history length and number of perceptrons to change. However, we decided to keep these parameters the same as the baseline PBNP to directly quantify the impact of our techniques.

Table 2 Parameters for the baseline PBNP and versions using modulo path-history and folded-modulo-path history

Size (KB)	Common parameters		Modulo-path	+Folded path	
	History length	Rows per SRAM (# Perceptrons)	Path length	Path length	f = Path folding
1	24	40	8	11	2
2	24	81	8	11	2
4	31	128	6	15	2
8	32	248	7	11	2

4.2 Predictor Accuracy

The usage of modulo-path history potentially compromises the prediction accuracy of the path-based neural predictor (PBNP) due to the reduction in the total amount of unique path information. While the modulo-path versions of the PBNP make for simpler and more practical implementations, a substantial reduction in accuracy and overall performance would simply make both conventional *and* modulo-path versions of the predictor undesirable.

Figure 4 shows the average prediction accuracy of the different versions of PBNPs across a range of predictor sizes for SPECint, MediaBench and MiBench, respectively. Overall, the modulo-path modifications only slightly increase the misprediction rates of the predictors, with a greater sensitivity at the smallest hardware budget. For future high-performance processors, the predictor sizes are more likely to be toward 8 KB or larger [27], making the sensitivity at the smaller sizes less of a problem. This is a very positive result as it means that we can employ the simpler modulo-path versions of the neural predictor without crippling performance.

It is important that the performance results apply reasonably uniformly across the individual benchmarks of each application suite. Figure 5 shows the per-benchmark branch misprediction rates for every application evaluated. The SPECint suite shows some variability between benchmarks, but in most situations the simplified versions of the predictor do not cause too many additional mispredictions. What is interesting to observe is that there is a distinct tradeoff between overhead reduction and the variability or sensitivity of the misprediction rates. In particular, the folded-modulo-path predictor provides a greater reduction in hardware (see Sect. 4.5), but the misprediction rate increase varies more than the simple non-folded modulo-path history predictor. The MiBench and MediaBench suites exhibit similar trends. It is also important to observe that the sensitivity of performance to misprediction rates also varies, and so we evaluate the IPC impact of our techniques in the following section.

4.3 Predictor Performance

Given that our modified versions of the PBNP do not affect prediction accuracy by much, we expect that the overall performance will also be similar to that of the original

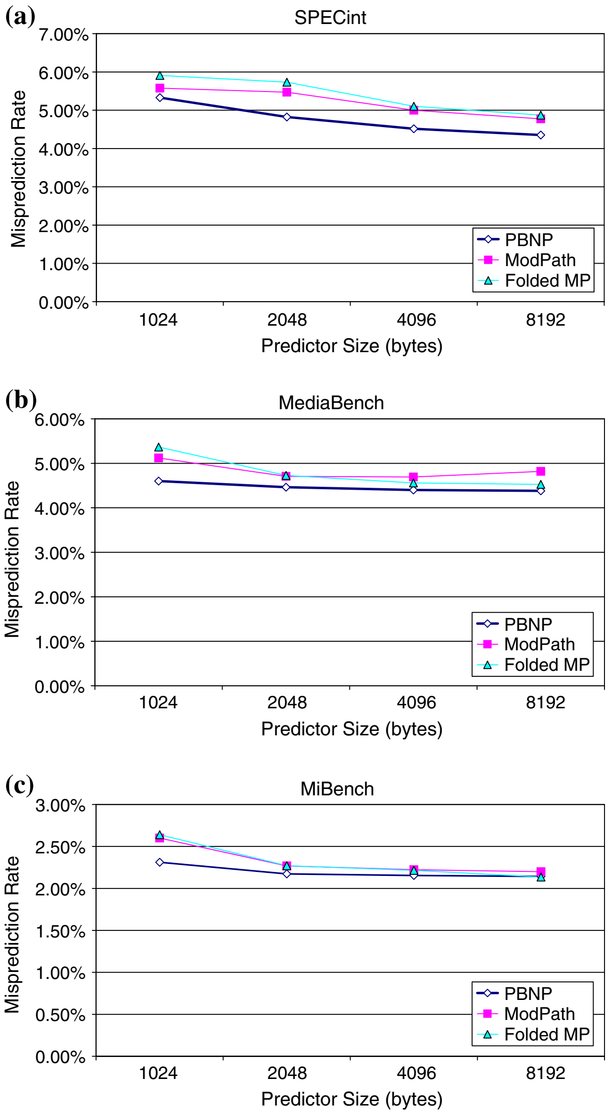

Fig. 4 Arithmetic mean misprediction rates across the (**a**) SPECint, (**b**) MediaBench and (**c**) MiBench applications for the original path-based neural predictor (PBNP) as well as versions using modulo- and folded-modulo-path history

PBNP. Figure 6 shows the geometric mean IPC rates across the SPECint, MediaBench and MiBench applications, respectively, for different predictor sizes. Overall, the IPC of our simplified PBNPs matches the performance of the original predictor very closely. At an 8 KB budget on the SPECint applications, the IPC degradation is only 0.8% for the ModPath predictor, and 1.3% for the Folded version. Across the range, the difference in overall performance is within the noise of the simulator.

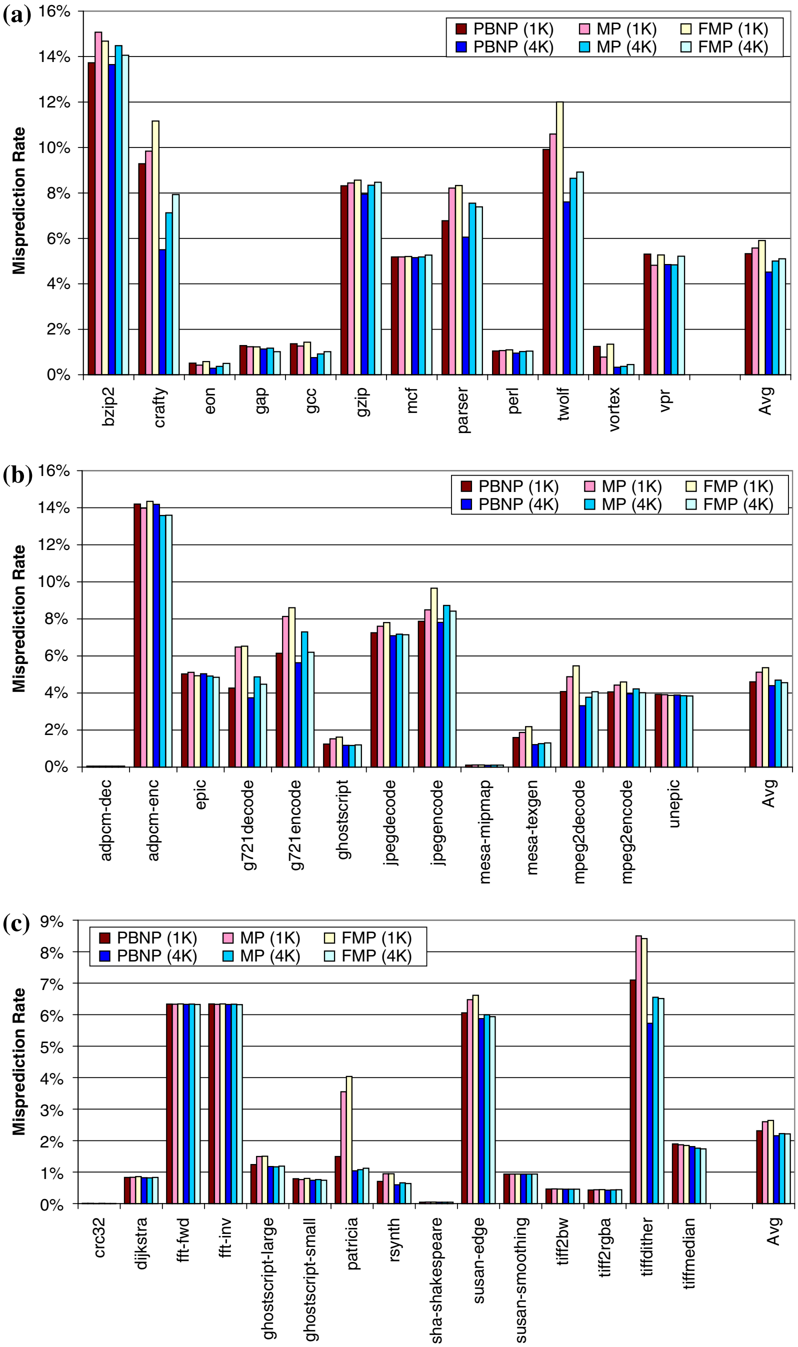

Fig. 5 Per-benchmark misprediction rates for 1 KB and 4 KB predictors on the (**a**) SPECint, (**b**) MediaBench and (**c**) MiBench workloads for the original path-based neural predictor (PBNP) as well as versions using modulo- and folded-modulo-path history

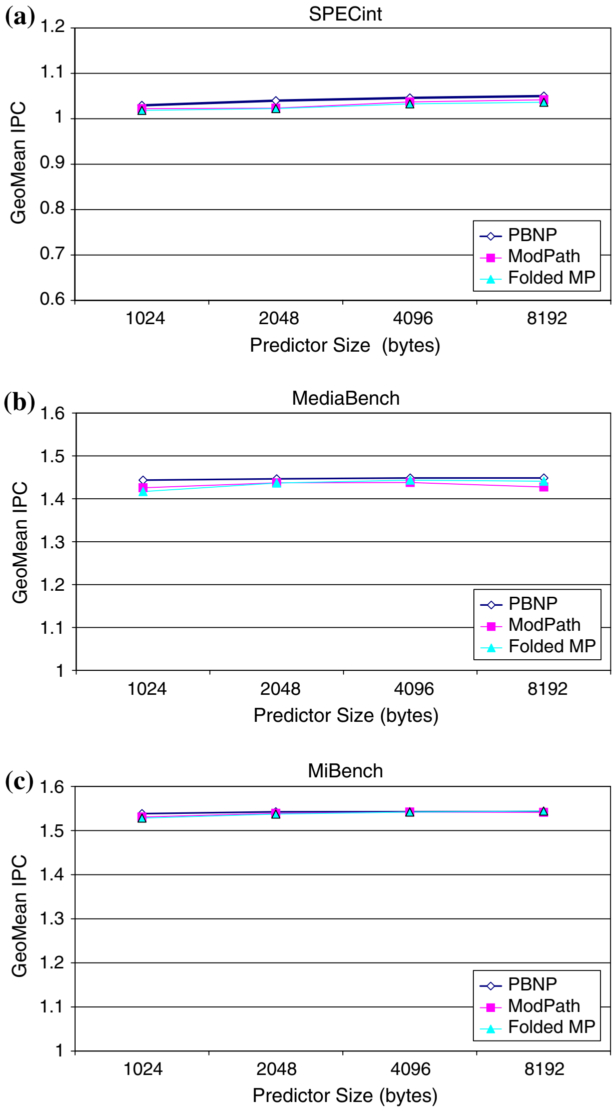

Fig. 6 Geometric mean IPC rates for the (**a**) SPECint, (**b**) MediaBench and (**c**) MiBench applications for the original path-based neural predictor (PBNP) as well as versions using modulo- and folded-modulo-path history

For the MediaBench applications, we observe 1.5% and 0.5% IPC degradations for the ModPath and Folded-ModPath 8 KB predictors, respectively. For MiBench, the performance penalties are 0.1% and 0.0% for the same predictors.

The choice of target applications affects which predictor organization is most appropriate. For example, ModPath performs better for SPECint, while the Folded ModPath is better for MediaBench. On the other hand, the MiBench applications are fairly insensitive to the choice of the neural predictor implementation, which means we can reap energy and area benefits (described in the next section) without *any* performance impact.

It is important to keep in perspective that even though the modulo-path history versions of the PBNP cause a slight performance drop, this is relative to a processor that uses a conventional PBNP. Without our proposed modifications, a processor would not even be able to use the PBNP in the first place. Our contribution is a new design for the PBNP that makes it much more practical while delivering nearly the same benefit as the more complex version. The alternative is a processor with much less performance due to a less sophisticated non-neural prediction algorithm [1].

Figure 7 illustrates the IPC impact of the proposed techniques on a per-application basis. On the SPECint suite, the crafty and twolf benchmarks exhibited the largest relative increases in misprediction rates. In the case of crafty, the increase in misprediction rate leads to a corresponding performance reduction. However, this performance degradation is only prominent in the folded-modulo-path history version of the predictor. For the non-folded path history predictor, the performance decrease is quite small. In the other case of a large misprediction rate increase for twolf, it turns out that the application is not overly sensitive to the increase in branch mispredictions. This may be due to other mitigating factors such as a larger number of stall cycles due to non-branch-related events such as cache misses. For the other MiBench and MediaBench suites, the sensitivity of the IPC performance to the misprediction rate is even less than SPECint, and so the overall IPC performance of our simplified PBNP predictors remains robust across the different applications. As discussed earlier, these results are very encouraging as it enables the implementation of a sophisticated prediction algorithm that delivers performance that is very close to the ideal PBNP technique. In the following sections, we explore the implementation benefits in terms of energy consumption and estimated area overhead.

4.4 Energy Impact

We anticipate that the modulo-path PBNPs will consume less energy per access for two reasons. The first is that the folded organization reduces the total number of SRAM arrays which reduces the per-table overhead such as the row decoders. However, packing the same number of bits into a smaller number of tables tends to increase the wordline lengths, which could increase power. The second reason for an energy reduction is in the decrease of the checkpointing overhead. By reducing the total number of SRAM tables, we reduce the length of the predictor pipeline, thereby requiring less state to be checkpointed for each branch. Figure 8 shows the overall energy consumption per predictor access, which includes the predictor portion (SRAMs and adders) as well as the checkpointing overhead. The adders' energy consumption does not vary because we maintain the same history length between the conventional PBNP and the modulo-path versions. The results clearly show a substantial reduction in the predictor energy consumption, ranging from 30% and 42% for 1 KB modulo-path and folded-modulo-path versions, respectively, to 29% and 37% for the 8 KB configurations.

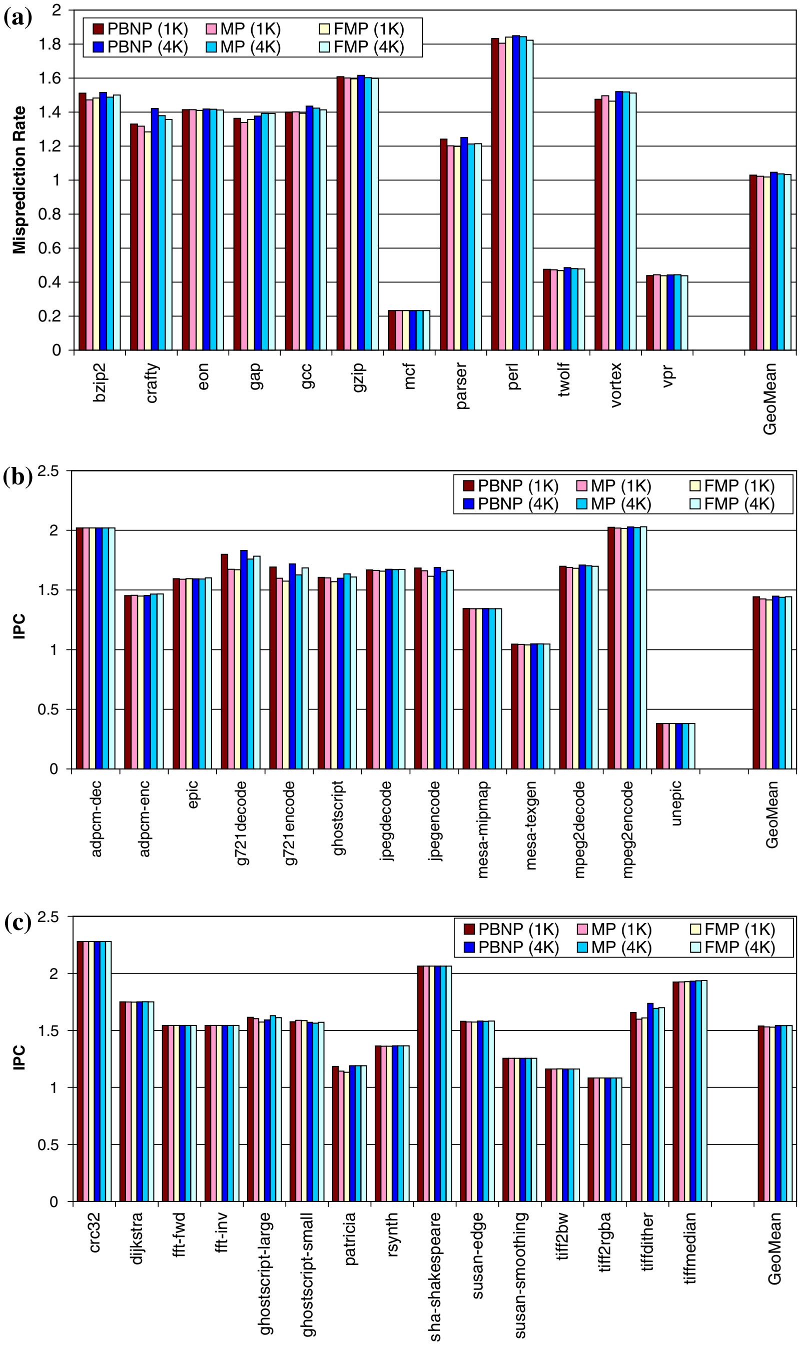

Fig. 7 Per-benchmark IPC rates for 1 KB and 4 KB predictors on the (**a**) SPECint, (**b**) MediaBench and (**c**) MiBench workloads for the original path-based neural predictor (PBNP) as well as versions using modulo- and folded-modulo-path history

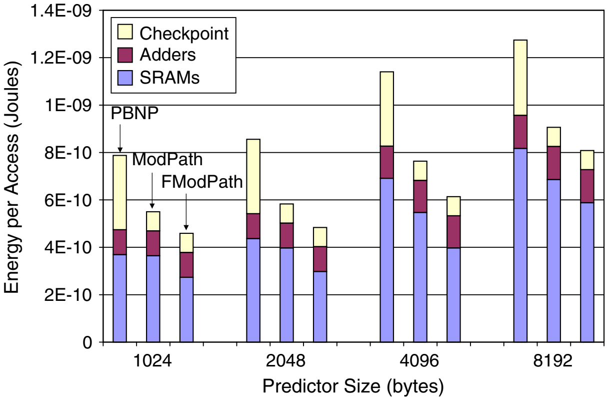

Fig. 8 Energy per access including the lookups in the SRAMs for the predictor weights, the adders for computing the predictor output, and one checkpoint access

4.5 Area

To estimate the die area of each predictor configuration, we use the process independent register bit equivalent (rbe) metric proposed by Mulder et al. [26]. The rbe methodology provides for a way to estimate the area overhead of the decoder/driver logic, sense amps, and other related circuitry. Figure 9 shows the estimated areas of the predictor configurations in rbe's. The majority of the area reduction comes from a reduction in the total number of SRAM tables which reduces the overhead of duplicated decode logic (the total number of predictor bits remains constant). This estimate may be slightly generous in that additional wordline repeaters/drivers may need to be inserted to avoid a substantial increase in the overall predictor access time. Any additional drivers would add to the area overhead; however, the drivers should still take up less area than a full decoder tree. The checkpoint overhead[1] (includes bitcells, decoders, sense amps) also contributes a small but non-negligible amount of area. Depending on the overall hardware budget for the PBNP, the modulo-path history can reduce area requirements by 18–37% without any substantial impact on performance.

Note that in our earlier discussions and comparisons, we kept the overall hardware budget the same between the original and optimized PBNP configurations. However, the hardware budget as measured by bits of storage is only a proxy for the actual die area required for the predictor. In a practical setting, our 8 KB optimized predictors would require substantially less area to implement (up to 37%). Instead of reducing the die footprint of the predictor, we could instead reclaim the area to add more perceptron entries to the tables of weights. This would help to relieve capacity conflicts in the predictor structures, but could still maintain the same hardware cost in die area as the original PBNP even though it contains a larger total number of bits of state.

[1] We assume that at most one out of four instructions will be a branch, and therefore for a ROB of 64 entries, we use 16 checkpoints.

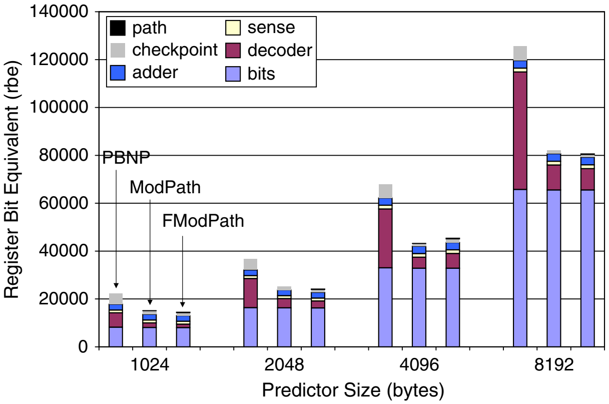

Fig. 9 Area requirements of the different versions of PBNP in register bit equivalents (rbe)

5 Generalization to Other Branch Predictors

The piecewise linear (PWL) neural predictor is a more sophisticated generalization of the previously proposed neural branch predictors [3]. While the PWL achieves significantly higher prediction accuracies than the previous neural predictors, the checkpointing overhead also increases substantially. In this section, we review the structure of the PWL predictor, explain the application of modulo-path and folded-modulo-path history to the PWL.

5.1 Piecewise Linear Neural Prediction

The piecewise linear neural predictor uses a more complex algorithm than previous neural predictors. In particular, its ability to compose multiple linear decision surfaces in a piecewise manner allows the predictor to learn a much larger and richer set of decision functions. Despite the increase in algorithmic sophistication, the PWL predictor's hardware structure can effectively be decomposed into k copies of the original PBNP pipeline. Figure 10 illustrates the hardware organization of the PWL predictor.

Each of the PBNP pipelines learns a single linear decision surface. At the end of the pipeline, the least significant bits from the current branch's PC select one of the k predictions. The update process only affects the weights belonging to the one PBNP pipeline selected.

For rapid recovery of the PWL pipeline, the partial sums from each of the k PBNP pipelines must be checkpointed. This directly increases the amount of state per branch predictor checkpoint by a factor of k, and can increase the corresponding area, latency and power consumption of the checkpoint support by even more than k (for example, unbuffered wire delay increases quadratically with wire length). In such a situation, techniques to reduce checkpoint overhead become even more critical.

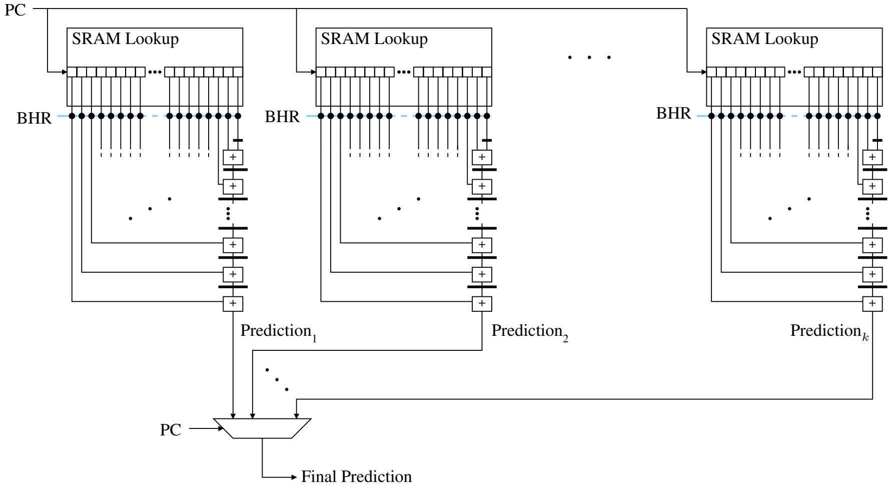

Fig. 10 The pipeline organization of the piecewise linear predictor consists of *k* parallel copies of the PBNP pipeline

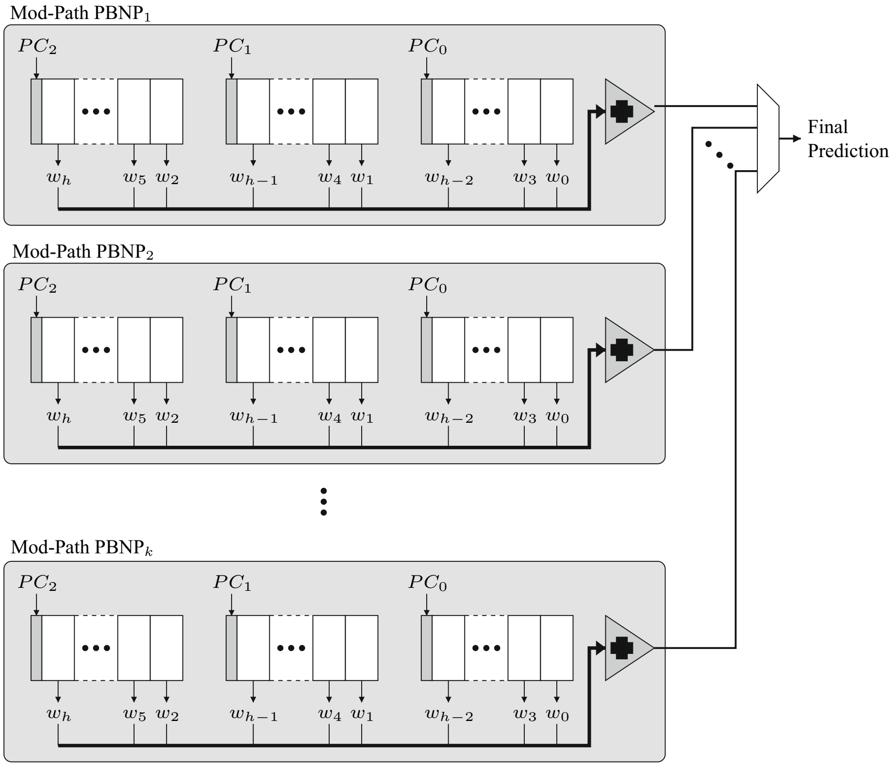

Fig. 11 The modulo-path PWL predictor consists of *k* parallel copies of the modulo-path PBNP pipeline

5.2 Modulo-path PWL Predictors

The organization of the PWL predictor as *k* parallel PBNP pipelines greatly assists in the application of the modulo-path history technique to this more sophisticated predictor. Specifically, a modulo-path PWL predictor is composed of *k* parallel modulo-path PBNP pipelines as illustrated in Fig. 11. In a similar fashion, a folded-modulo-path PWL predictor can be constructed from *k* instances of folded-modulo-path PBNP pipelines. Compared to the PBNP versions, the PWL predictors have *k* times more checkpoint overhead. However, when compared to the original PWL predictor, the modulo-path versions of PWL require substantially less overhead.

6 Conclusions

Despite the high accuracy of the neural-based branch predictors, none have yet been implemented in any commercial processors. We believe that the primary obstacles to the adoption of neural predictors is in the complexity of the previously proposed schemes. Our two new proposed schemes, *modulo-path history* and *folded modulo-path history*, provide different tradeoffs between the reduction in the predictor implementation overhead and how close the resultant IPC performance matches that of an ideal path-based neural predictor. Based on the experimental results, we conclude that the (non-folded) modulo-path history predictor provides the best design point. This version of the predictor provides better prediction rates and performance than the folded version, more robust performance on a per-application bases (i.e., the variance in performance degradation across the entire workload is lower), and also results in lower energy consumption. We conclude that the slight additional area benefit of the folded version does not justify the increase in misprediction rates, decrease in performance and increase in power.

The modulo-path history predictors proposed in this article provide substantial reductions in the hardware complexity as measured by the number of tables, the predictor pipeline depth, and the checkpointing overhead, while simultaneously reducing predictor energy consumption and die-area requirements. The reduction in power and area can potentially be used to reduce the cost of the processor, or they could also be traded to implement larger predictors than was previously possible with a conventional neural predictor organization. We believe that research aimed at providing practical implementations of sophisticated predictors is critical to successfully transferring this technology to industrial implementations.

Acknowledgements Gabriel Loh is supported by funding and equipment from Intel Corporation. Daniel Jiménez is supported by Grants from NSF (CCR-0311091 and CCF-0545898).

References

1. Jiménez, D.A., Lin, C.: Neural methods for dynamic branch prediction. ACM Trans. Comput. Syst. **20**(4), 369–397 (2002)
2. Jiménez, D.A.: Fast path-based neural branch prediction. In: Proceedings of the 36th International Symposium on Microarchitecture, pp. 243–252, San Diego, CA, USA, December 2003

3. Jiménez, D.A.: Piecewise linear branch prediction. In: Proceedings of the 32nd International Symposium on Computer Architecture, pp. 382–393, Madison, WI, USA, June 2005
4. Desmet, V.: Vandierendonck, H., De Bosschere, K.: A 2bcgskew predictor fused by a redundant history skewed perceptron predictor. In: Proceedings of the 1st Championship Branch Prediction Competition, pp. 1–4, Portland, OR, USA, December 2004
5. Loh, G.H.: The Frankenpredictor. In: Proceedings of the 1st Championship Branch Prediction Competition, pp. 1–4, Portland, OR, USA, December 2004
6. Tarjan, D., Skadron, K.: Merging path and gshare indexing in perceptron branch prediction. ACM Trans. Architect. Code Optimization **2**(3), 280–300 (2005)
7. Gao, H., Zhou, H.: Adaptive information processing: an effective way to improve perceptron predictors. In: Proceedings of the 1st Championship Branch Prediction Competition, pp. 1–4, Portland, OR, USA, December 2004
8. Seznec, A., Fraboulet, A.: Effective ahead pipelining of instruction block address generation. In: Proceedings of the 30th International Symposium on Computer Architecture, San Diego, CA, USA, May 2003
9. Co, M., Weikle, D.A.B., Skadron, K.: A break-even formulation for evaluating branch predictor energy efficiency. In: Proceedings of the Workshop on Complexity-Effective Design, Madison, WI, USA, June 2005
10. Yeh, T.-Y., Patt, Y.N.: Two-level adaptive branch prediction. In: Proceedings of the 24th International Symposium on Microarchitecture, pp. 51–61, Albuqueque, NM, USA, November 1991
11. McFarling, S.: Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory (1993)
12. Jiménez, D.A., Keckler, S.W., Lin, C.: The impact of delay on the design of branch predictors. In: Proceedings of the 33rd International Symposium on Microarchitecture, pp. 4–13, Monterey, CA, USA, December 2000
13. Ghose, K., Kamble, M.B.: Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In: Proceedings of the International Symposium on Low Power Electronics and Design, pp. 70–75, San Diego, CA, USA, August 1999
14. Seznec, A., Felix, S., Krishnan, V., Sazeides, Y.: Design tradeoffs for the alpha EV8 conditional branch predictor. In: Proceedings of the 29th International Symposium on Computer Architecture, Anchorage, AK, USA, May 2002
15. Stark, J., Evers, M., Patt, Y.N.: Variable length path branch prediction. ACM SIGPLAN Notices **33**(11), 170–179 (1998)
16. Seznec, A.: Analysis of the O-GEometric history length branch predictor. In: Proceedings of the 32nd International Symposium on Computer Architecture, Madison, WI, USA, June 2005
17. Michaud, P.: A PPM-like, tag-based predictor. J. Instr. Level Parallel. **7**, 1–10 (2005)
18. Seznec, A., Michaud, P.: A case for (partially) TAgges GEometric history length branch prediction. J. Instr. Level Parallel. **8**, 1–23 (2006)
19. Austin, T., Larson, E., Ernst, D.: SimpleScalar: an infrastructure for computer system modeling. IEEE Micro Magaz. pp. 59–67, February 2002
20. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: a free, commerically representative embedded benchmark suite. In: Proceedings of the 4th Workshop on Workload Characterization, pp. 83–94, Austin, TX, USA, December 2001
21. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. In: Proceedings of the 30th International Symposium on Microarchitecture, pp. 330–335, Research Triangle Park, NC, USA, December 1997
22. Perelman, E., Hamerly, G., Calder, B.: Picking statistically valid and early simulation points. In: Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques, pp. 244–255, New Orleans, LA, USA, September 2004
23. Larson, E., Chatterjee, S., Austin, T.: MASE: a novel infrastructure for detailed microarchitectural modeling. In: Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software, pp. 1–9, Tucson, AZ, USA, November 2001
24. Gochman, S., Ronen, R., Anati, I., Berkovitz, A., Kurts, T., Naveh, A., Saeed, A., Sperber, Z., Valentine, R.C.: The intel pentium M processor: microarchitecture and performance. Intel Technol. J. **7**(2) (2003)
25. Shivakumar, P., Jouppi, N.P.: CACTI 3.0: An Integrated Timing, Power, and Area Model. TR 2001/2, Compaq Computer Corporation Western Research Laboratory (2001)

26. Mulder, J.M., Quach, N.T., Flynn, M.J.: An area model for on-chip memories and its application. IEEE J. Solid-State Circ. **26**(2), 98–106 (1991)
27. The 1st JILP Championship Branch Prediction Competition (CBP-1). http://www.jilp.org/cbp. Accessed on 5 Dec 2004