A Simple Divide-and-Conquer Approach for Neural-Class Branch Prediction

Gabriel H. Loh
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
loh@cc.gatech.edu

Abstract

The continual demand for greater performance and growing concerns about the power consumption in high-performance microprocessors make the branch predictor a critical component of modern microarchitectures. Recent research in applying machine learning techniques to the branch prediction problem has shown incredible improvements in branch prediction accuracy by exploiting correlations in very long branch histories. Nevertheless, these techniques have not been adopted by industry due to the high implementation complexity.

In this paper, we propose a global-history Divide-and-Conquer (gDAC) branch predictor that achieves IPC rates that are near that of the best neural predictors, but remains easy to implement because they only make use of simple tables of saturating counters. We show how to use ahead-pipelining to implement our gDAC predictor with a single-cycle effective latency. Our gDAC predictor achieves higher performance (IPC) than the original global history perceptron predictor across all predictor sizes evaluated, and outperforms the path-based neural predictor for predictors 16KB and larger. At 128KB, gDAC even achieves an IPC rate equal to the recently proposed piecewise-linear neural branch predictor.

1. Introduction

While hardware branch predictors have been extensively researched for decades [24], current and future design trends continue to place an increasing demand for more accurate algorithms. The drive for faster frequencies through deeper pipelines have increased the IPC penalty for mispredicted branches [25]. Even if clock frequencies do not continue to scale as rapidly as they have in the past decade, branch prediction will still be the performance bottleneck for future large effective window size processors [1, 3, 26]. Apart from the performance problems, the combination of deeper pipelines and wider machine widths have greatly increased the number of wasted wrong-path instructions flushed on a branch misprediction. This in turn presents a big problem for power/thermal-limited high-performance desktop and server processors as well as battery-life limited mobile processors [5]. The demand for better hardware branch prediction algorithms has not gone away.

Recent research in adapting machine learning algorithms to the branch prediction problem have been very successful in dramatically increasing branch prediction rates [7, 9, 11]. Despite the impressive simulation results of these neural prediction schemes, no industrial efforts have publicly announced the incorporation of such a branch predictor into a commercial processor design. The original perceptron proposal has the shortcoming of a long latency and a fairly complex implementation [11]. The path-based neural predictor (PBNP) uses a clever pipelining scheme to address the latency issues of the original perceptron predictor, and even manages to further increase the prediction accuracy [7]. Unfortunately, the implementation complexity of the PBNP increases substantially due to the large number of carry-completing adders required (as opposed to the more area- and power- efficient carry-save adders of the original perceptron). The recently proposed piecewise-linear neural predictor (PWL) further increases prediction accuracy, but also substantially increases both the number of adders and the amount of state that must be saved and restored for branch misprediction recovery.

Apart from being a performance-critical component, the hardware branch predictor is also one of the worst thermal hot-spots in the processor [23]. Except in the relatively rare instances where the processor is completely full, the branch predictor must be accessed every cycle, thus resulting in an activity factor of almost 100%. The branch predictor circuitry is also timing critical in that its latency must be minimized to avoid losing performance. This timing pressure results in very aggressive circuit implementations for the branch predictor critical path logic which in turn further aggravates the power and thermal problems. Despite the incredible accuracy of the neural prediction algorithms, the large number of adders required to implement them, especially the piecewise linear neural predictor which may require *hundreds* of adders, may render the predictors infeasible from a power, thermal and complexity perspective.

The principle contribution of this research is a new branch prediction technique that achieves neural-class performance while only employing simple gshare-like tables of counters for ease of implementation [16]. Restricting the implementation to simple tables of saturating counters (also called pattern history tables or PHTs) allows the latency problem to be addressed with ahead-pipelining or other similar techniques [8, 21, 22] and avoids the need for the large number of adders. While traditional PHTs scale very poorly with the branch history length, our proposed table-based scheme is able to deal with perceptron-sized branch histories by using a divide-and-conquer approach. The end result is the new gDAC branch predictor (**g**lobal-history **D**ivide-**A**nd-**C**onquer).

The rest of this paper provides a detailed analysis of our proposed branch predictor. Section 2 describes the previously proposed neural branch predictors and covers the sources of implementation complexity. Section 3 details our new gDAC prediction algorithm, which is followed by Section 4 which presents prediction accuracy and overall performance results. Section 5 introduces the concepts of correlation locality, redundancy and recovery to explain why gDAC is able to effectively make use of a long, but segmented branch history. Finally, Section 6 summarizes and closes the paper with some concluding remarks.

2. Background

This section only covers previous work that is most closely related to the primary focus of this paper: branch predictors that exploit very long branch histories and techniques to mitigate predictor latency and complexity.

2.1. Neural Predictors

Traditional PHT-based branch predictors 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 [11] 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, which 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). 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. The need to make the addition tree as fast as possible introduces a second problem, which is the power consumed by the adders. To meet cycle-time targets, the adders must be implemented with fast, leaky transistors which end up costing in both dynamic and static power consumption.

The second-generation path-based neural predictor (PBNP) largely solves the latency problems of the original perceptron, but it still suffers from significant implementation complexity [7]. As shown in Figure 1b, the clever pipelining of the PBNP provides a much faster effective predictor latency: the critical path is now the table lookup and a single addition. Unfortunately, this process still typically takes more than one cycle, which means the PBNP still must act as an overriding predictor on top of a first-level single-cycle predictor [10]. 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* neural (PWL) predictor is actually a generalization of the original perceptron and path-based neural predictors [9]. The PWL predictor achieves prediction accuracies that are even better than the PBNP by attacking the problem of linearly inseparable branches. Perceptron-based predictors are traditionally limited in that they can only learn boolean functions that are linearly separable, that

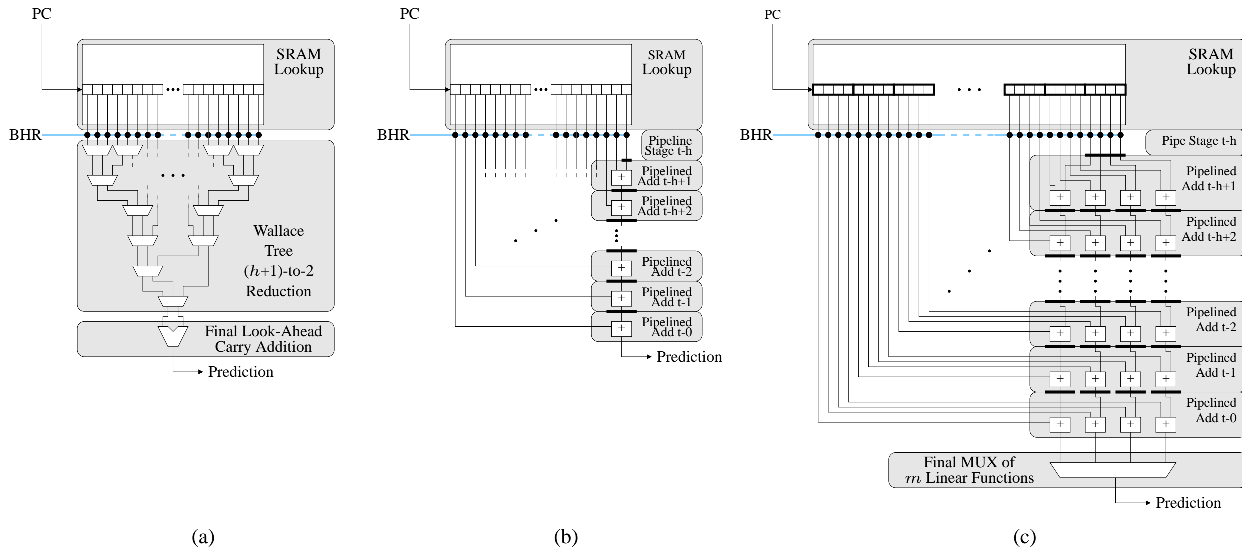


Figure 1. (a) The original perceptron predictor, (b) the path-based neural predictor, and (c) the piecewise-linear neural predictor.

is functions where the h-dimensional space of taken and not-taken outcomes can be separated by an (h-1)-dimensional hyperplane. Figure 1c shows a PWL predictor that deals with linearly inseparable functions by computing n different linear functions in parallel (in this example, $n = 4$ and is illustrated by the four columns of addition chains), and then using the current branch address to select from among these functions. In aggregate, these different linear functions form a piecewise-linear decision surface. The overall lookup delay is equal to the latency of the table of weights, the addition, and then a final n-to-1 multiplexer.

While the PWL predictor can achieve *very* impressive prediction rates, this performance comes at a cost of implementation complexity and power. As illustrated in Figure 1c, the PWL predictor performs n path-based neural predictions in parallel and then throws away all but one of these predictions. Although the predictor only makes use of a single outcome, the PWL still must have enough adders to compute all n predictions in parallel. While a PBNP with a history length of h requires h adders, the PWL predictor multiplies this hardware cost by n for a total of nh adders. This represents a significant cost in area, dynamic power, leakage power, and overall complexity.

2.2. Predictor Partitioning

Some of the previously proposed prediction schemes have proposed organizations that decompose or partition a predictor into smaller components. Skewed predictors [18] are partitioned to deal with aliasing effects, hybrid predictors [16] are functionally partitioned to target different types of correlation, 2bcgskew [21], the hashed-perceptron [28, 29], O-GEHL [20], and PPM [17] all propose dividing a predictor into individual components for targeting different history lengths. Our history segmentation approach was independently proposed by Tarjan and Skadron in the context of neural branch predictors [29]. Many of these proposed techniques are largely orthogonal. For example, one could consider a segmented, geometric history-length PPM predictor.

2.3. Other Predictors

Other research efforts have studied the effects of branch predictor latency on performance. Jiménez and Lin evaluated several designs for hierarchical branch predictors for tolerating long predictor access latencies [10]. Seznec and Fraboulet studied the overall problem of instruction address generation in a fetch architecture that supports multiple instructions per cycle [22]. In particular, they propose a general technique called *ahead-pipelining* that initiates predictor lookups multiple cycles before the prediction is needed using whatever information is currently available. In a similar vein, Jiménez proposed gshare.fast that uses pipelining techniques to provide effective single-cycle predictions from a gshare predictor [8].

Similar to the neural predictors, the dataflow branch predictor also uses very long branch histories, but the hardware support required is fairly complex [30]. The update stage of the predictor processes branches *and* all register writing instructions to determine which branches affect the dataflow leading to other branches. Using this information, the lookup stage is able filter out bits from the branch history that are not relevant, and it compacts the history register into a much smaller size that a conventional PHT-like table can handle.

Our gDAC predictor has similarities to other past research. The per-segment predictors make use of the Bi-Mode prediction algorithm [13]. The root predictor of the gDAC behaves like the fusion hybrid predictor [14]. We compare our predictor to a conventional gshare [16] and the neural predictors. We also make use of tournament hybrids in the analysis section [16].

2.4. Mispeculation Recovery

The pipelined organizations of the PBNP and the PWL predictors allow for very accurate branch prediction while maintaining reasonable access latencies. At first glance, it may appear that a branch misprediction which causes a pipeline flush would expose the latency of the more than h stages of the predictor's pipeline. To avoid this, such *ahead-pipelined* predictors must checkpoint the pipeline state for each branch. After the processor detects a branch misprediction, the predictor pipeline's checkpointed state is flash-copied back into all of the pipeline latches which allows the predictor to quickly resume making predictions down the new branch path.

As shown in Figure 1b, the PBNP's pipeline state consists of the weights and partial sums at each of the pipeline latches (solid black lines). For k-bit weights, this adds up to about kh bits of data.[1] For the PWL predictor, the amount of state that must be checkpointed increases by a factor of n to about khn. Using the 32KB configuration reported in the PWL prediction study, $k = 8$, $h = 26$ and $n = 8$ yields 1664 bits or 208 bytes of data that must be checkpointed for *every* branch. When accounting for the fact that the widths of the partial sums gradually increase, the total state is actually 282 bytes per checkpoint. A 256-entry reorder buffer would require at least 51 checkpoints just to satisfy the average number of branches in the instruction window (assuming one branch per five instructions). These 51 checkpoints, at 282 bytes each, would add over 14KB of extra state to the 32KB PWL branch predictor. The 32KB PBNP with a history length of 31 would require 354 bits per checkpoint, for a total of 2.2KB for the 51 checkpoints.

The implementation challenge for the ahead-pipelined neural predictors is a direct result of the large number of pipeline stages (proportional to the branch history length). This in turn raises the hardware cost in terms of the number of adders and the amount of state that must be checkpointed. The goal of our research is the development of a branch prediction algorithm that can exploit long branch histories and achieve neural-class performance while employing only simple components in an ahead-pipelined organization.

3. gDAC: An Ahead-Pipelined Neural-Class Branch Predictor

To maintain simplicity, we limit our predictor to only use conventional tables of saturating counters (PHTs). Unfortunately, PHTs do not scale well with increasing branch history lengths. A history length of h requires 2^h entries in a conventional PHT. Contrast this to a neural predictor whose size requirements only increase linearly with the history length. To deal with very long history lengths, we propose a Divide-and-Conquer approach where we partition the branch history register into smaller segments, each of which can be handled by a small, implementable PHT. A final table-based predictor combines all of these per-segment predictions to provide the overall decision.

3.1. Organization

Our global-history Divide-and-Conquer (gDAC) branch predictor combines the ideas of ahead-pipelining [21], history segmentation, and prediction fusion [14]. Figure 2a shows a generic gDAC branch predictor. We divide the long global branch history register into non-overlapping segments.[2] Each segment provides a branch history input to a PHT-based predictor that provides a branch prediction for that segment. A final *root predictor* takes all of the per-segment predictions as part of its input vector and computes the final prediction.

For a given hardware budget, much of the state will be consumed by the first-level segment predictors, and some portion of the state must be dedicated for the root

[1] The actual number of bits is even larger because the bit-width of the adders increases as the partial sums accumulate down the pipeline. This results in a total of $O(hk + h \lg h)$ bits per checkpoint.

[2] We conducted some experiments with overlapping segments as well, but there was no net benefit. For some applications overlap improves prediction accuracy, in other applications it decreases accuracy.

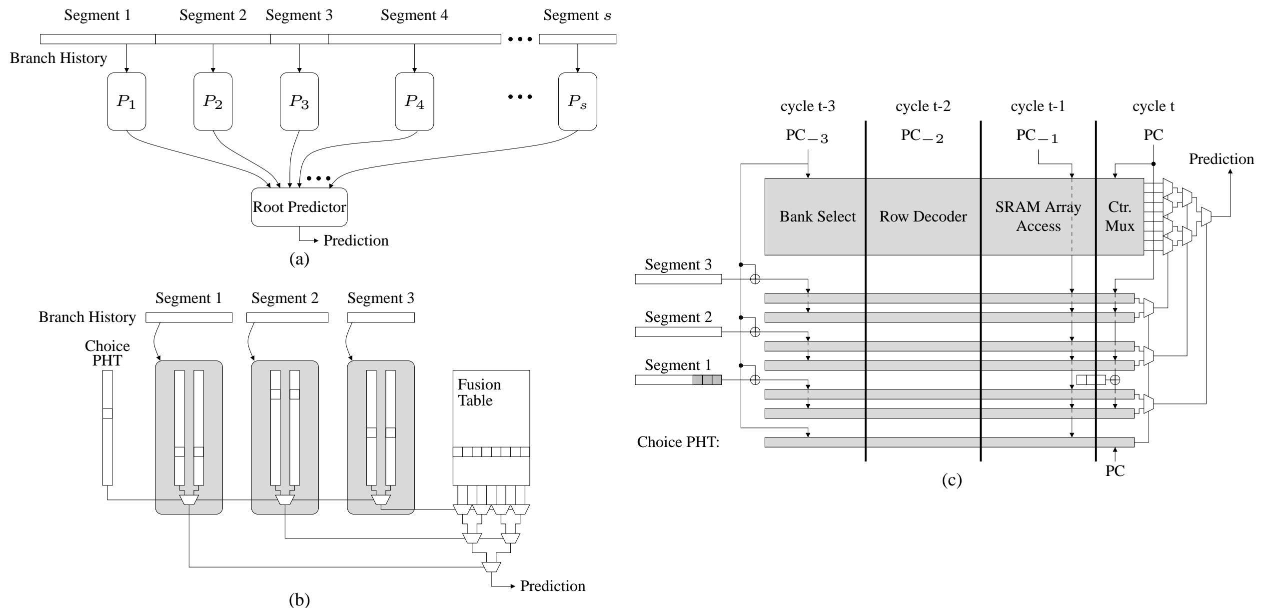


Figure 2. (a) Conceptual schematic of divide-and-conquer for long branch histories, (b) our gDAC predictor, and (c) the ahead-pipelined implementation of gDAC prediction.

predictor. For very long history lengths and large hardware budgets, it is conceivable to build multiple levels of predictors each covering unique segments and subsegments. For this study, we only considered two-level organizations as depicted in Figure 2a.

While conducting our design space exploration, we found that it was very important to make the per-segment predictors as accurate as possible. Unreliable inputs for the root predictor result in a "garbage-in garbage-out" scenario. To reduce the effects of destructive interference in the segment predictors, we used a Bi-Mode organization [13] instead of conventional gshare-style predictors. A Bi-Mode predictor sorts branches into two tables: one that tracks branches that are usually taken, and a second that tracks branches that are usually not-taken. The idea is that if two branches map to the same entry, but both are usually taken (or both not-taken), then the interference will be neutral. A third table called the choice-PHT tracks the bias of each branch and is used to select one of the two tables. Figure 2b shows the overall organization of our gDAC predictor. Note that the Bi-Mode segment predictors all share a single choice-PHT, which reduces the overhead of implementing interference reducing predictors. Similar to the shared hysteresis arrays of the 2bc-gskew predictor [21], we use only one hysteresis bit for every two direction bits in each PHT. The per-segment predictions form an index that selects a counter from a line of the prediction fusion table.

We also found that for some predictor sizes and configurations, it is beneficial to use a longer branch history segment than the number of bits required to index a PHT. For example, a 1024-entry PHT uses a 10-bit index, but a 17-bit segment may provide more correlation. This can be viewed as a segmented extension of how 2bcgskew and O-GEHL use long history lengths. For per-table history lengths of $h_0, h_1, ...$, 2bcgskew and O-GEHL use a total history length equal to the maximum of the h_i's, whereas gDAC uses the *sum* which provides a larger possible total history length.

Our gDAC root predictor is a simple fusion predictor [14]. We hash the most recent global history with the branch address, and then concatenate the per-segment predictions to form a final index. The index selects a 4-bit counter from a standard PHT structure where the most significant bit of the counter determines the final prediction. We experimented with Bi-Mode [13], skewed [18], and shared-hysteresis [21] fusion predictors and found that the simple PHT performed the best.

The update operation of the predictor is straightforward. Each segment predictor follows the update rules for a conventional Bi-Mode predictor, including the partial update policy. The root predictor performs a saturating increment or decrement of its counter depending

on whether the final branch outcome was taken or not-taken, respectively.

3.2. Ahead-Pipelining the gDAC Predictor

As described in their original studies, the Bi-Mode predictors and the prediction fusion mechanism are not ahead-pipelined. We propose to modify the Bi-Mode and fusion table to implement an ahead-pipelined gDAC predictor (Figure 2c). A conventional Bi-Mode predictor's choice-PHT uses the current branch address as an index. Instead, we initiate the choice-PHT lookup 3 cycles ahead of when the final prediction is needed by using whatever branch address was available at that time. In parallel, we also initiate the PHT lookups for the per-segment predictors and the fusion root predictor using the currently available PC and the corresponding branch history segment. The bank select and row decode for the SRAMs that implement the PHTs use the indexes generated in cycle t-3 to select a row of counters. In cycle t-1, bits from the then current PC are used in the column select mux of the SRAM array. We organize the SRAM such that a single word contains the predictions for 2^N counters. In cycle t, we use the p bits from the current PC combined with the s per-segment predictions to select the final prediction (therefore $N = p + s$).

Note that for segment 1, the three most recent (relative to cycle t) branch outcomes have not yet been produced. We instead incorporate these by XOR-ing them with the branch address in cycle t when they are all available.

The checkpointing mechanism behaves in a fashion analogous to the gshare.fast predictor [8]. A gshare.fast predictor that is ahead-pipelined by three cycles needs to checkpoint eight bits of information corresponding to the eight possible outcomes of the three branch history bits generated during each of ahead-pipelined cycles. For our gDAC predictor, we also need to keep eight versions of the predictor state for a three-cycle ahead-pipelined implementation. Assuming $p = 3$ and $s = 2$ (this is the case for our 32KB configuration detailed later), each "version" of the predictor state contains eight bits for each of the PHT structures, and 32 bits for the fusion table. This totals to 8 bits for the Bi-Mode choice PHT, 8 bits for each of the per-segment T-PHTs and NT-PHTs (in this example, 3 segments yields 6 PHTs), and the 32 bits for the fusion table, for a total of 88 bits or 11 bytes. Using the same assumptions about the ROB size and number of checkpoints needed for the PBNP and the PWL predictors, this adds up to 561 bytes of state.

The total amount of checkpoint state for the gDAC predictor is significantly less than what is needed for the PWL and PBNP predictors. For smaller hardware budgets, the number of cycles that the gDAC predictor must be ahead-pipelined decreases which further reduces the checkpointing overhead. If it is necessary to further reduce the checkpoint size, the number of PC bits p used for the final selection can be reduced.

3.3. Overall Configurations

The best gDAC configurations are different depending on the available hardware budget. The number of segments, the total amount of branch history used and the sizes of the different tables all vary. Table 1 lists the best gDAC configurations found for a range of hardware sizes. These are the best chosen from a large search space where we varied the number of segments, the sizes of individual segment predictors, the size of the Bi-Mode choice PHT, the size of the fusion table relative to the segment predictor sizes and the amount of branch history.

3.4. Power Consumption

In this section, we will provide a brief qualitative comparison of the power consumption of neural predictors and gDAC. A detailed power analysis is beyond the scope of this study as that would require circuit-level implementations of all of the predictors, as well as full-CPU power simulations to observe the impact of reducing wrong-path instructions on overall power consumption.

The power consumption of neural predictors come from several sources. In particular, we will discuss path-based neural predictors [7], which include the piecewise linear predictor [9]. To avoid long pipeline delays during the update of path-based neural predictors, the table of weights is divided into one table per history bit plus one table for the bias. The large number of tables makes each individual table smaller, which in turn reduces the effectiveness of power-saving techniques like banking and sub-banking [27]. Predictors based on a few large PHT structures, such as gshare, 2bcgskew, or gDAC, may be more amenable to banked SRAM organizations to reduce power consumption. Note that while the per-segment Bi-Mode predictors functionally have two PHTs each, the same is index is used for both tables, which allows them to be implemented as a single physical table where each entry contains one counter from each of the logical tables.

Hardware Budget	Number of Segments	1.5 bits per entry (shared hysteresis bit) Bias PHT Entries	PHT Entries (for each T-PHT and NT-PHT) Segment 1	Segment 2	Segment 3	4 bits per entry Root Predictor Entries	Per-Segment (Total) History Length
2KB	2	2048	2048	1024	—-	1024	7/14 (21)
4KB	2	4096	4096	2048	—-	2048	12/16 (28)
8KB	2	8192	8192	4096	—-	4096	12/16 (28)
16KB	3	8192	8192	8192	8192	8192	10/10/10 (30)
32KB	2	32768	32768	16384	—-	16384	17/25 (42)
64KB	3	65536	65536	32768	16384	16384	17/25/38 (80)
128KB	3	131072	131072	65536	32768	32768	18/27/41 (86)

Table 1. The gDAC configurations evaluated in this study.

The computational nature of the neural-inspired algorithms requires many adder circuits to compute the weighted sums. A conventional path-based neural predictor requires 30-50 adders; the piecewise-linear predictor requires 2 to 8 *times* more adders (one set per parallel linear function computation). While predictors like O-GEHL [20] and the modulo-path variant of the path-based neural predictor [15] reduce the number of adders down to about only 8, this still takes up an area that is similar to a full 64-bit adder.[3] The update logic, which is not frequently discussed, also requires a full complement of additional adders to increment and decrement the weights during the training process.

A PHT-based predictor allows for a partitioning of the prediction and hysteresis arrays [21]. Besides allowing different sized arrays to reduce area and power, the hysteresis array is not required during prediction lookup and therefore can be placed in a location in the processor that is physically separated from the prediction array. This in turn can reduce power density, which affects thermals, by moving the dynamic and leakage power associated with the hysteresis array away from the prediction arrays. Contrast this to the neural and O-GEHL predictors where the entire weight is required to compute the final prediction. For these predictors, there is no opportunity for partitioning the SRAM arrays to reduce power density.[4]

As compared to a gshare predictor, gDAC will consume more power because it contains extra logic for supporting prediction fusion and Bi-Mode selection within each component predictor. On the other hand, gDAC's superior prediction accuracy decreases overall power consumption by reducing the number of wrong-path instructions. Compared to a path-based neural predictor, gDAC has the power advantages of being implemented with only a few PHT-based structures, a PHT and multiplexor-based fusion mechanism as opposed to a network of adders, reduced checkpointing overhead, and a simpler update mechanism. We believe that the combination of these characteristics makes gDAC a more power- and complexity-effective way to achieve neural prediction rates.

[3]Each neural adder is typically 8 bits wide. Eight such adders has the area of about a full 64-bit adder, less the area required for the upper levels of the carry-propogate circuit.

[4]We evaluated neural predictors that use two sets of arrays: one for the most significant bits of the weights, and one for the least significant bits. We computed the prediction using only the array containing the most significant bits (thus allowing the second array to be placed elsewhere to manage power density), but this substantially reduces the prediction accuracy.

4. Evaluation and Results

In this section, we first explain our simulation methodology, and then we present the prediction accuracy and IPC results for the gDAC predictor as well as for several other recent predictors.

4.1. Methodology

We started with the SimpleScalar 4.0 infrastructure for the Alpha instruction set architecture [2]. For our initial design space exploration, we used an in-order simulator based off of the sim-safe functional simulator. Our timing simulator is a modified version of MASE [12]. In particular, we accurately simulate speculative branch history updates during fetch with non-speculative predictor updates deferred until the commit stage and branch history recovery on mispredictions. We also simulate a full front-end decode pipeline instead of merely stalling fetch for a number of cycles equal to the mispredict penalty. The overall configuration is similar to that used in the Piecewise Linear prediction paper [9], and the details are listed in Table 2. Functional unit latencies are set identical to the Pentium 4, except that the integer unit is not double-pumped [6].

We evaluated the branch predictors on the twelve SPEC integer benchmarks, using the reference inputs. To reduce the simulation time, we used 100 million instruction samples chosen with the single-early method of the SimPoint tool [19]. The SPEC binaries were com-

Machine Width	16-wide	IL1	16KB, 2-way, 3-cycle
IFQ Size	16 entries	DL1	16KB, 4-way, 3-cycle
Scheduler Size (RS)	128 entries	L2 (unified)	512KB, 8-way, 7-cycle
ROB Size	512 entries	L3 (unified)	8MB, 16-way, 20-cycle
Load Queue Size	64 entries	Main Memory	500 cycles, 64-bit bus
Store Queue Size	64 entries	ITLB	64-entry, fully associative, 4KB pages
Branch Penalty	40 cycles	DTLB	64-entry, fully associative
Scheduler Replay	3 cycles	L2-TLB	1024-entry, 8-way
Memory Dependency Prediction	Always no-alias	Functional Units	16 Int-ALU, 4 Int Mult, 4 cache ports 8 FP-ALU, 4 FP Mult/Div
BTB	4096 entry, 4-way	Hardware Prefetcher	16 Stream Buffers, 8 entries each

Table 2. The processor configuration used for our timing/IPC simulations.

piled on an Alpha 21264 using the Compaq cc compiler with -O4.

We compared our gDAC predictor against all three generations of neural branch prediction algorithms, as well as the gshare predictor for reference. We chose the best performing predictor configurations for different hardware budgets by sweeping all history lengths up to 64 bits and testing n={2,4,8} for the PWL predictor. The configurations are listed in Table 3, and for our applications the PWL predictor performed the best with $n = 8$ for all predictor sizes. While a real implementation would implement all of the neural predictor tables with a power-of-two number of entries, we do not impose this restriction which allows for better performing neural predictors. This places a slight handicap on the gDAC predictor since we do restrict its PHTs to always have a power-of-two number of entries.

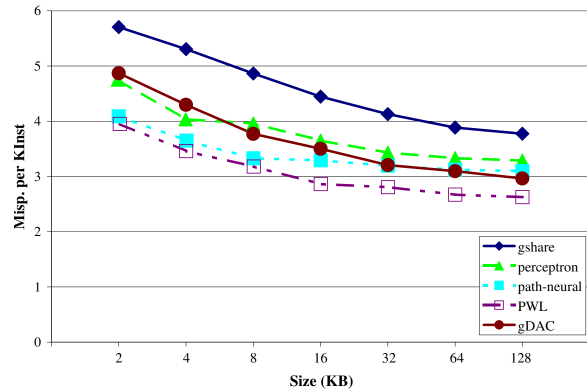


Figure 3. Average misprediction rate over a range of predictor sizes in misprediction-per-thousand instructions averaged across the SPECint benchmarks.

4.2. Accuracy Results

First we look at the branch prediction accuracy across the range of hardware budgets considered. Figure 3 shows that for small hardware budgets, the gDAC predictor provides prediction accuracies that are comparable to the original perceptron predictor. At moderate predictor sizes, gDAC catches up with the path-based neural predictor. Even at the largest sizes considered, the PWL predictor consistently provides the best prediction rates. The reasons why PWL provide a better overall accuracy is that PWL makes use of the full branch address, whereas gDAC does not due to its ahead-piplined organization, PWL makes use of a longer path history, and PWL's large number of separately indexed tables provide more interference tolerance. Overall, the gDAC predictor achieves raw prediction accuracy that is on par with the PBNP. The overall performance impact will also be affected by the latency characteristics of these predictors.

On a per-benchmark basis, the 128KB gDAC predictor consistently meets or beats the prediction accuracy of the PBNP. Figure 4 shows the misprediction rates for each of the twelve SPECint benchmarks for the 128KB versions of all of the predictors evaluated. For gcc and vortex, the gDAC predictor performs slightly worse than the PBNP, but only by a very small margin. For eon, gap and mcf, the gDAC predictor actually beats the highly accurate PWL predictor.

4.3. IPC Results

For our performance simulation, we assume an overriding branch predictor hierarchy for the neural predictors. The initial predictor is a 2048-entry table of saturating two-bit counters. For the perceptron and PBNP predictors, we use the latencies reported in earlier studies [7]. For the PWL predictor, we conservatively assume no additional latency overhead as compared to an equal-sized PBNP. While we assume ahead-pipelining of our gshare to enable a single-cycle latency, we conservatively allow the predictor to make use of the full branch address (normally gshare.fast only uses about three bits of the current PC which reduces accuracy).

Hardware Budget	gshare.fast Table Size (Entries)	gshare.fast History Length	perceptron Rows in Table of Weights	perceptron History Length	PBNP Rows in Table of Weights	PBNP History Length	PWL-neural Rows in Table of Weights	PWL-neural History Length
2KB	8192	12	75	26	81	24	73	27
4KB	16284	14	151	26	141	28	113	35
8KB	32768	15	182	44	282	28	167	48
16KB	65536	16	260	62	381	42	341	47
32KB	131072	16	520	62	697	46	528	61
64KB	262144	17	1057	61	1394	46	1057	61
128KB	524288	18	2114	61	2788	46	2114	61

Table 3. Configurations for the previously proposed predictors used in our simulations.

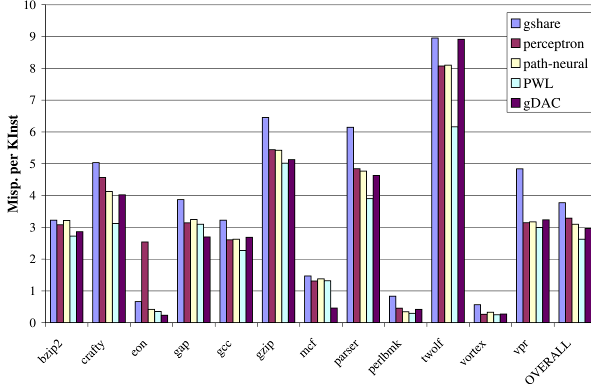


Figure 4. Per-benchmark misprediction rates at a 128KB budget.

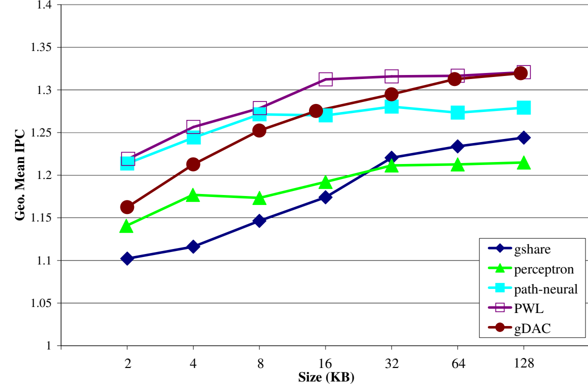


Figure 5. Geometric mean IPC over a range of predictor sizes for SPECint.

The gDAC predictors provide single-cycle prediction latencies due to their ahead-pipelined implementations.

Figure 5 shows the absolute geometric mean IPC for each of the predictors simulated across the range of hardware budgets. For all predictor sizes, gDAC achieves a higher IPC rate than a processor that uses a conventional perceptron predictor. This is due to a combination of the perceptron's long latency and, at larger hardware budgets, gDAC's higher prediction rate. From 16KB and larger, gDAC delivers more performance than the PBNP. At the largest size considered, gDAC manages to break even with the PWL predictor. The PWL predictor attains a higher prediction accuracy, but the difference in predictor latencies allows gDAC to catch up with the PWL predictor.

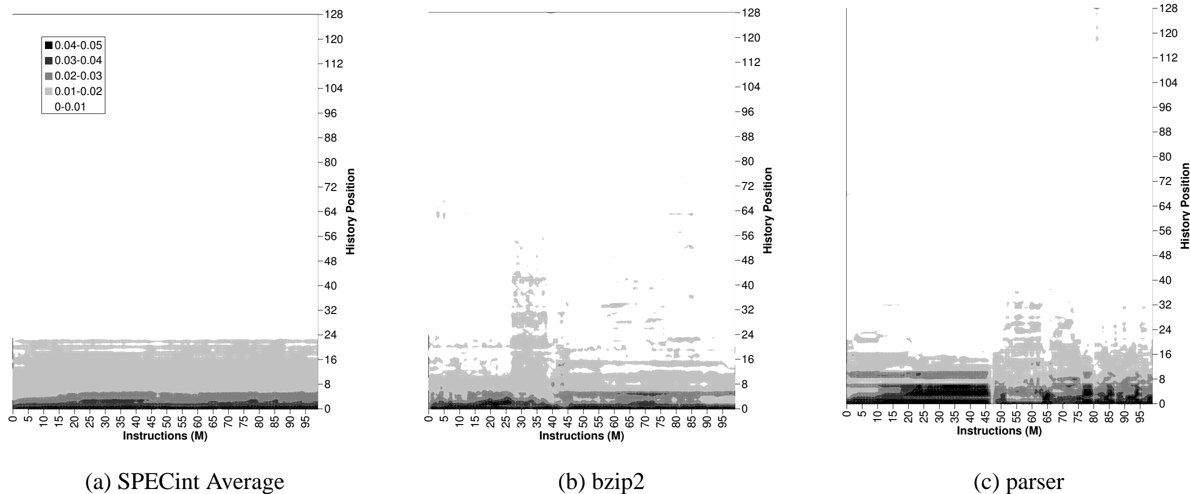
5. Analysis

History segmentation divides the global branch history into multiple smaller intervals. The benefit is that each of these individual segments are more easily handled by conventional table-based predictors. One possible disadvantage is that branch outcomes that depend on correlations with history from multiple segments may be difficult to predict. The gDAC predictor is able to cope with this due to the phenomenas of *correlation locality* and *correlation redundancy*, and through the fusion mechanism's ability to perform *correlation recovery*.

5.1. Correlation Locality

The perceptron is very useful to analyze branch correlation because there is a weight associated with each bit of history, and the magnitude of the weight indicates the degree of correlation. Using an interference-free perceptron predictor, we tracked the relative importance of each bit of history by monitoring the weights for each static branch.

By sampling the distribution of weights and plotting the results, we are able to visually locate the strongest branch correlations, and observe how they change with time. For each sample point, we compute the weighted correlation sum for each bit of branch history by adding the absolute value of the weights corresponding to that bit of history. These values are then normalized by the sum of the absolute value of all weights, weighted again by frequency. This is repeated every one million instructions for a total of one hundred sample points.



(a) SPECint Average (b) bzip2 (c) parser

Figure 6. The distribution of global branch correlation as determined by the weighted average of an interference-free perceptron's weights over 100 million instructions.

Strong branch correlations tend to cluster together. Figure 6a shows the average correlation distribution for the SPECint benchmarks. Darker areas represent stronger correlation. There are distinct bands of strongly correlated branches, and the more recent branch outcomes tend to be more important. On a benchmark-by-benchmark basis, the distributions are more varied and the correlation locality is much more striking. Figure 6b shows the correlation profile for bzip2. A few different phases of correlation behavior are visible in this simulation point. In the first phase (samples 0-25) there is one dominant cluster consisting of the most recent branch to about the 12th most recent. Then there are two secondary clusters consisting of branches 13-17 and 20-24. The second phase (samples 26-40) shows several larger clusters. Figure 6c shows that parser also exhibits similar correlation locality. The remaining applications are not shown for brevity, but also demonstrate similar locality. The existence and prevalence of correlation locality suggests that the potential loss of correlation due to history segmentation should be limited.

5.2. Correlation Redundancy

Past studies have shown that only a few branches are needed from the global history register to provide most of the correlation benefit. Evers showed that using the three most important bits (determined with hindsight) provides performance that is very close to that obtained by using the entire branch history register [4]. Thomas et al. show that only those branches that guard dataflow ancestors of the current branch, called affectors, are really important for branch correlation [30].

We hypothesize that one of the reasons the gDAC predictor can tolerate a segmented history is that correlations conveyed by branch history bits not included in a segment have a good chance of being conveyed by other bits that are included in the segment due to *correlation redundancy*. That is, several history bits may be heavily cross-correlated, and it is sufficient for a predictor to extract correlations from a subset of these bits. If some of these bits fall outside of the current segment, it may not seriously affect the predictability of the branch because other correlated branches still exist within the segment.

To provide evidence to support our hypothesis, we simulated the 16KB gDAC predictor, but we replaced the per-segment Bi-Mode predictors with per-segment perceptrons. Using perceptrons, we are able to pick and choose which branches we wish to contribute to the final prediction. In particular, we modify each perceptron to only use its k largest weights when computing its prediction. Figure 7 shows normalized prediction rates as the number of weights used is varied. Zero percent on the y-axis represents the prediction accuracy when no weights are used (all per-segment predictors always predict taken, and the root predictor makes the final prediction with no outside help), and 100% corresponds to when all weights are used. Superimposed on Figure 7 is the average percentage of the total "weight-mass" used. For example, if there were only three weights of values -5, -30 and +15, then choosing the single largest weight would cover 60% of the sum of the absolute weights;

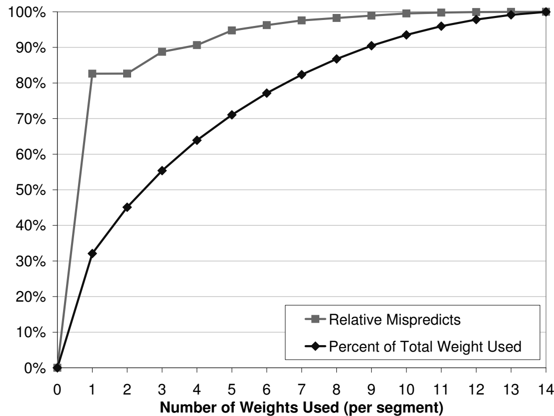


Figure 7. The normalized prediction accuracy of a divide-and-conquer predictor with perceptron segment predictors when only the k largest weights are used to make predictions. The graph also shows the average percentage of the total weight used.

choosing the two largest weights (-30 and +15) covers 90% of the total available weight-mass. When we consider only the largest weight per segment, we make use of only about 32% of the available weight-mass while being able to achieve 83% of the benefit of using all weights. Using a small portion of the available correlation to obtain most of the benefit indicates that the excluded weights do not contribute much additional useful information, or that their correlations are redundant.

5.3. Correlation Recovery

Another reason why the gDAC predictor is able to tolerate segmentation of the global history is that the fusion-based root predictor is able to reconstruct some of the cross-segment correlations that the individual segment predictors cannot see. To verify this, we again used the 16KB gDAC predictor (with the original Bi-Mode segment predictors), but we replaced the fusion-based root predictor with a selection-based mechanism. The 16KB gDAC uses three segment predictors. A tournament meta-predictor $M_{2,3}$ [16] chooses between the second and third segment predictions, and then another tournament meta-predictor M_1 chooses between the first segment and the prediction chosen by $M_{2,3}$. Because the selection process can only choose a single prediction, only information from a single segment can be utilized and cross-segment correlations cannot be handled. Contrast this with a fusion-based root predictor that can attempt to combine information from all segments. The fusion approach achieves 3.6% fewer mispredictions than using selection, which provides evidence that our gDAC root predictor is indeed able to recover some cross-segment correlation.

6. Conclusions

Despite the neural branch predictor's ability to achieve phenomenal prediction rates, the associated complexity due to latency, large quantity of adder circuits, area and power are still obstacles to the industrial adoption of this technique. Instead of trying to find a new formulation of the perceptron for easy implementation, we chose to design a new predictor that is quite different from the previous neural approaches but still takes advantage of the same phenomena of deep-history branch correlations. We have demonstrated that it is possible to achieve neural-class prediction rates and IPC performance using only simple PHT structures. The PHT-only approach enables a straight-forward ahead-pipelined implementation, and it also reduces the corresponding checkpointing overhead. As a result, we believe that the gDAC predictor is a practically implementable means of achieving the prediction accuracy of a neural branch predictor.

Acknowledgments

Funding and equipment were provided by a grant from Intel Corporation.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 423–434, San Diego, CA, USA, May 2003.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.

[3] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward Kilo-Instruction Processors. *Transactions on Architecture and Code Optimization*, 1(4):389–417, December 2004.

[4] M. Evers. *Improving Branch Prediction by Understanding Branch Behavior*. PhD thesis, University of Michigan, 2000.

[5] S. Gochman, R. Ronen, I. Anati, A. Berkovitz, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal*, 7(2), May 2003.

[6] Intel Corporation. IA-32 Intel Architecture Optimization Reference Manual. Order Number: 248966-011, 2004.

[7] D. 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] D. A. Jiménez. Reconsidering Complex Branch Predictors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 43–52, Anaheim, CA, USA, February 2003.

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

[10] D. A. Jiménez, S. W. Keckler, and C. Lin. The Impact of Delay on the Design of Branch Predictors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 4–13, Monterey, CA, USA, December 2000.

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

[12] E. Larson, S. Chatterjee, and T. Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, pages 1–9, Tucson, AZ, USA, November 2001.

[13] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The Bi-Mode Branch Predictor. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 4–13, Research Triangle Park, NC, USA, December 1997.

[14] G. H. Loh and D. S. Henry. Predicting Conditional Branches with Fusion-Based Hybrid Predictors. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, USA, September 2002.

[15] G. H. Loh and D. A. Jiménez. Reducing the Power and Complexity of Path-Based Neural Branch Prediction. In *Proceedings of the 5th Workshop on Complexity-Effective Design*, pages 1–8, Madison, WI, USA, June 2005.

[16] S. McFarling. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.

[17] P. Michaud. A PPM-like, Tag-Based Predictor. *Journal of Instruction Level Parallelism*, (7):1–10, 2005.

[18] P. Michaud, A. Seznec, and R. 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.

[19] E. Perelman, G. Hamerly, and B. 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.

[20] A. Seznec. Analysis of the O-GEometric History Length Branch Predictor. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

[21] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, Anchorage, AK, USA, May 2002.

[22] A. Seznec and A. Fraboulet. Effective Ahead Pipelining of Instruction Block Address Generation. In *Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, CA, USA, May 2003.

[23] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-Theoretic Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pages 17–28, 2002.

[24] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, Minneapolis, MN, USA, May 1981.

[25] E. Sprangle and D. Carmean. Increasing Processor Performance by Implementing Deeper Pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 25–34, Anchorage, AK, USA, May 2002.

[26] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. 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.

[27] C.-L. Su and A. M. Despain. Cache Design Tradeoffs for Power and Performance Optimization: A Case Study. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 63–68, Dana Point, CA, USA, April 1995.

[28] D. Tarjan and K. Skadron. Merging Path and Gshare Indexing in Perceptron Branch Prediction. CS 2004-38, University of Virginia, December 2004.

[29] D. Tarjan and K. Skadron. Revisiting the Perceptron Predictor Again. CS 2004-28, University of Virginia, December 2004.

[30] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark. Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 314–323, San Diego, CA, USA, May 2003.