Revisiting the Perceptron Predictor Again

UNIV. OF VIRGINIA DEPT. OF COMPUTER SCIENCE TECH. REPORT CS-2004-28
SEPT. 2004

David Tarjan Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904
dtarjan@cs.virginia.edu

Kevin Skadron Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904
skadron@cs.virginia.edu

Abstract

We introduce a new kind of branch predictor, the *hashed* perceptron predictor, which merges the concepts behind the gshare and perceptron branch predictors. This is done by fetching the perceptron weights using the exclusive-or of branch addresses and branch history. This predictor can achieve superior accuracy to a path-based and a global perceptron predictor, previously the most accurate fully dynamic branch predictors known in the literature, at the same storage budgets. Additionally, it reduces the number of adders by a factor of four compared to a path-based perceptron. We also show how such a predictor can be ahead pipelined to yield one cycle effective latency, making it the first standalone perceptron predictor. On the SPEC integer set of benchmarks, the hashed ahead-pipelined path-based perceptron predictor (hashed perceptron for short) improves accuracy by 20% over a path-based perceptron and improves IPC by 5.8%. We believe these improvements make the perceptron predictor a promising choice as a branch predictor for a future high-performance microprocessor.

1 Introduction

The trend in recent high-performance commercial microprocessors has been towards ever deeper pipelines to enable ever higher clockspeeds [2, 6], with the width staying about the same from earlier designs. This trend has put increased pressure on the branch predictor from two sides. First, the increasing branch misprediction penalty puts increased emphasis on the accuracy of the branch predictor. Second, the sharply decreasing cycle time makes it difficult to use large tables or complicated logic to perform a branch prediction in one cycle. The consequence has been that recent designs often use a small one cycle predictor backed up by a larger and more accurate multi-cycle predictor. This increases the complexity in the front end of the pipeline, without giving all the benefits of the more accurate predictor.

Recently, it was proposed [7, 9, 16] that a branch predictor could be *ahead pipelined*, using older history or path information to start the branch prediction, with newer information being injected as it became available. While there is a small decrease in accuracy compared to the unpipelined version of the same predictor, the fact that a large and accurate predictor can make a prediction with one or two cycles latency more than compensates for this.

Using a different approach to reducing the effective latency of a branch predictor, a pipelined implementation for the perceptron predictor [9] was also proposed. The perceptron predictor is a new predictor which is based not on two bit saturating counters like almost all previous designs, but on a simple neural network.

Perceptrons have been shown to have superior accuracy at a given storage budget in comparison to the best table based predictors. Yet they need a large number of small adders to switch every cycle they make a prediction, increasing both the area of the predictor and the energy per prediction. Finally, hiding the latency of a perceptron predictor requires that such a predictor be heavily pipelined, leading to problems similar as those encountered when designing modern hyperpipelined execution cores.

Previous perceptron predictors assigned one weight per local or global branch history bit. This meant that the amount of storage and the number of adders increased linearly with the number of history bits used to make a prediction. The key insight of this paper is that the 1 to 1 ratio between weights and number of history bits is not necessary. By assigning a weight not to a single branch but a sequence of branches we can transform the mapping of weights to history bits from a location based mapping (as used in a bimodal predictor) to a pattern or history based mapping (as used in a global history predictor). Moreover, combining the location and history pattern based mappings leads very

naturally to the rediscovery of a joint or shared mapping using both sources of information (as used in GAs or gshare predictors).

Decoupling the number of weights from the number of history bits used to make a prediction allows us to reduce the number of adders and tables almost arbitrarily.

The main contributions of this paper are:

- We show that a hashed perceptron has equal or better prediction accuracy than a traditional path-based perceptron for a given storage budget, while reducing the number of adders and separate tables by factor of 4 at 40kb.
- We show how such a perceptron can be ahead pipelined to reduce its effective latency to one cycle, obviating the need for complex overriding scheme.

This paper is organized as follows: Section 2 gives a short introduction to the perceptron predictor and gives an overview of related work, Section 3 talks about the impact of delay on branch prediction and how it has been dealt with up to now, as well as the complexity involved in such approaches, Section 4 shows how a perceptron can be ahead pipelined to yield one cycle effective latency, Section 5 then explains the concept of a hashed perceptron, Section 6 describes the simulation infrastructure used for this paper, Section 7 shows results both for table-based predictors as well as comparing the hashed perceptron with prior proposals. Finally, Section 8 concludes the paper.

2 The Perceptron Predictor and Related Work

2.1 The Idea of the Perceptron

The perceptron is a very simple neural network. Each perceptron is a set of weights which are trained to recognize patterns or correlations between their inputs and the event to be predicted. A prediction is made by calculating the dot-product of the weights and an input vector (see Figure 1). The sign of dot-product is then used as the prediction. In the context of branch prediction, each weight represents the correlation of one bit of history (global, path or local) with the branch to be predicted. In hardware, each weight is implemented as an n-bit signed integer, where n is typically 8 in the literature, stored in an SRAM array. The input vector consists of 1's for taken and -1's for not taken branches. The dot-product can then be calculated as a sum with no multiplication circuits needed.

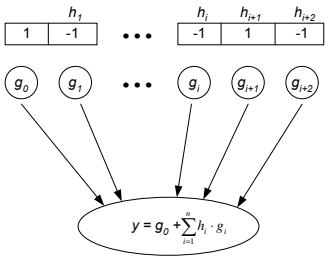


Figure 1: The perceptron assigns weights to each element of the branch history and makes its prediction based on the dot-product of the weights and the branch history plus a bias weight to represent the overall tendency of the branch. Note that the branch history can be global, local or something more complex.

2.2 Related Work

The idea of that perceptron predictor was originally introduced by Vintan [18] and Jiménez showed in [11] that the global perceptron could be more accurate than any other then known global branch predictor. The original Jiménez perceptron used a Wallace tree adder to compute the output of the perceptron, but still incurred more than 4 cycles of latency.

The recently introduced path-based perceptron [9] hides most of the delay by fetching weights and computing a running sum along the path leading up to each branch. The critical delay of this predictor is thus the sum of the delay of a small SRAM array, a mux and one small adder. It is estimated that a prediction would be available in the second

cycle after the address became available. For simplicity we will call this organization the *overriding perceptron*, since it can only act as a second-level overriding predictor, and not as a standalone predictor. A single cycle predictor is needed to make a preliminary prediction, which is potentially overriden by the perceptron predictor.

Seznec proposed several improvments to the original global perceptron in [14, 15]. In [15] he introduced the MAC-RHSP (multiply-add contribution redundant history skewed perceptron) predictor. He reduces the number of adders needed by a factor of four (16 when using redundant history) over the normal global perceptron predictor, by storing all the 16 possible combinations of four weights in separate table entries and selecting from them with a 16-to-1 mux after they have been fetched from the weight tables.

In our terminology, the MAC-RHSP is similar to a global perceptron predictor that uses a concatenation of address and history information (GAs or gselect) to fetch its weights. However the MAC-RHSP fetches all weights which share the same address bits from the tables, and then uses a 16-to-1 mux to select among them. This work was partly inspired by [15] and the MAC representation is one specific instance of an idea, which we generalize in the hashed perceptron.

The latency of the MAC-RHSP is hidden from the rest of the pipeline by starting the prediction early and computing all possible combinations of the last 4 branches in parallel. This requires 15 individual adders in addition to the 15-entry adder tree which is required to calculate the rest of the dot-product. The hashed perceptron only calculates the two possible outcomes of the last branch in parallel because of its lower latency and in general requires 2 to 3 times fewer adders because it packs more branch history bits into fewer weights than the MAC-RHSP.

Ipek et al. [7] investigated inverting the global perceptron. Theirs is not a pipelined organization per se, but rather uses older history to allow prefetching the weights from the SRAM arrays, hiding the associated latency. During fetch, these prefetched weights are combined with an input consisting of newer history and address bits, but this still incurs the delay of the Wallace-tree adder. There is no need for this kind of inversion for a pipelined perceptron, since the critical path is already reduced to a small SRAM array and a single adder. They also looked at incorporating concepts from traditional caches, i.e. two-level caching of the weights, pseudo-tagging the perceptrons and adding associativity to the weight tables.

3 Delay in Branch Prediction

An ideal branch predictor uses all the information which is available at the end of the previous cycle to make a prediction in the current cycle. In a table-based branch predictor this would mean using a certain mix of address, path and history bits to index into a table and retrieve the state of a two-bit saturating counter (a very simple finite state machine), from which the prediction is made.

3.1 Overriding Prediction Schemes

Because of the delay in accessing the SRAM arrays and going through whatever logic is necessary, larger predictors often cannot produce a prediction in a single cycle in order to direct fetch for the next cycle. This necessitates the use of a small but fast single cycle predictor to make a preliminary prediction, which can be overridden [10] several cycles later by the main predictor. Typically this is either a simple bimodal predictor or, for architectures which do not use a BTB, a next line predictor as is used by the Alpha EV6 and EV7 [4].

This arrangement complicates the design of the front of the pipeline in several ways. Most obviously, it introduces a new kind of branch misprediction and necessitates additional circuitry to signal an overriding prediction to the rest of the pipeline.

While traditionally processors checkpointed the state of all critical structures at every branch prediction, this method does not scale for processors with a very large number of instructions in flight. Moshovos proposed the use of selective checkpointing at low confidence branches [13]. Since the number of low confidence branches is much higher for the first level predictor than for the overriding predictor, this negates much of the benefit of selective checkpointing. Other proposals [1, 5] for processors with a very large number of instructions in flight similarly rely on some kind of confidence mechanism to select whether to checkpoint critical structures or not. As mentioned above, the overriding scheme introduces a new kind of branch misprediction. In a normal pipeline even without overriding, all structures which are checkpointed because of branch predictions must be able to recover from a BTB misprediction, signaled from the front of the pipeline, or a full direction misprediction, which is signaled from the end of the pipeline. The

predictor type	Amount of state to be checkpointed in bits
overriding perceptron	$\sum_{i=2}^{x-1} 1 + \lceil \lg(i-1) \rceil$ bits
ahead pipelined perceptron	$(w \cdot x) + \sum_{i=2}^{x-1} 1 + \lceil \lg(i-1) \rceil$ bits
table-based	$2^{x-1} - 1$ bits for most significant bits

Table 1: Amount of state to be checkpointed for each type of predictor. x is the pipeline depth of each predictor and w is the number of bits for each weight in the perceptron predictor.

depth of pipeline	Amount of state to be checkpointed in bits
13	133
18	195
20	221
32	377
34	405
37	447

Table 2: Example of amount of state to be checkpointed (in bits) for an overriding perceptron with 8-bit weights. We use the pipeline depth determined to be optimal in [9] as examples.

case of the slower predictor overriding the faster one introduces a new possible recovery point, somewhere between the first two.

3.2 Ahead-Pipelined Predictors

A solution to this problem, which was introduced in [9], was to "ahead pipeline" a large gshare predictor. The access to the SRAM array is begun several cycles before the prediction is needed with the then current history bits. Instead of retrieving one two-bit counter, 2^m two-bit counters are read from the table, where m is the number of cycles it takes to read the SRAM array. While the array is being read m new predictions are made. These bits are used to choose the correct counter from the 2^m counters retrieved from the array.

In an abstract sense, the prediction is begun with incomplete or old information and newer information is injected into the ongoing process. This means that the prediction can stretch over several cycles, with the only negative aspect being that only a very limited amount of new information can be used for the prediction.

An ahead pipelined predictor obviates the need for a separate small and fast predictor, yet it introduces other complications. In the case of a branch misprediction, the state of the processor has to be rolled back to a checkpoint. Because traditional predictors only needed one cycle, no information except for the PC (which was stored anyway) and the history register(s) were needed.

3.3 Checkpointing Ahead-Pipelined Predictors

For an ahead pipelined predictor, all the information which is in flight has to be checkpointed or the branch prediction pipeline would incur several cycles without a prediction being made in the case of a misprediction being detected. This would effectively lengthen the pipeline of the processor, increasing the branch misprediction penalty. The reason that an ahead pipelined predictor can be restored from a checkpoint on a branch misprediction and an overriding predictor cannot, is that the ahead pipelined predictor only uses old information to retrieve all the state which is in flight, while the overriding predictor would use new information, which would be invalid in case of a branch misprediction.

This problem was briefly mentioned in [16] in the context of 2BC-gskew predictor and it was noted that the need to recover in one cycle could limit the pipeline length of the predictor. In a simple gshare the amount of state grows

exponentially with the depth of the branch predictor pipeline, if all the bits of new history are used. Hashing the bits of new history down in some fashion of course reduces the amount of state in flight.

For an overriding perceptron, all partial sums in flight in the pipeline need to be checkpointed. See Table 1 for the formulas used to determine the amount of state to be checkpointed. Since the partial sums are distributed accross the whole predictor in pipeline latches, the checkpointing tables and associated circuitry must also be distributed. The amount of state that needs to be checkpointed/restored and the pipeline length determine the complexity and delay of the recovery mechanism. Shortening the pipeline and/or reducing the amount of state to be checkpointed per pipeline stage will reduce the complexity of the recovery mechanism.

A final factor to consider is the loss of accuracy in an ahead pipelined predictor with increasing delay. Since this was not explicitly investigated in [8], we investigate the response of some basic predictors and the pipelined perceptron predictor to delay. The first basic predictor is the gshare predictor, since it serves as the reference predictor in so many academic studies of branch predictors. Unlike the gshare.fast introduced in [8], we vary the delay from zero to four cycles. Our ahead pipelined version of the gshare predictor also differs from the gshare.fast presented in [8], as can be seen in Figure 2.

Our ahead pipelined gshare predictor uses a previous branch address along with the then current speculative global branch history to index into a table and retrieve 2^x counters, where x is the number of cycles the predictor is ahead pipelined. The gshare.fast only uses global history to retrieve the 2^x counters. Our ahead pipelined gshare uses the new global branch history which becomes available in the next x cycles to select from all the counters retrieved from the table, while the gshare.fast uses a hash of the actual branch address and the new global branch history to do the same.

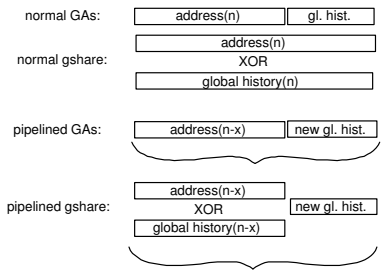


Figure 2: The ahead pipelined versions of each predictor uses the address information from several cycles before the prediction to initiate the fetch of a set of counters. In the case of the gshare predictor these are XOR'ed with the then-current global history. The new bits of global history which become available between beginning to access the pipelined table and when the counters become available from the sense amps are used to select one counter from the 2^x retrieved to make the actual prediction.

In general, when we say a predictor has delay x, we mean that only address bits from cycle (n - x), where cycle n is the cycle in which the prediction is needed, are used. In the case of the gshare predictor, we XOR the address(n - x) during cycle (n - x) with the speculative global history shift register and start to fetch a group of 2^x 2-bit counters from the prediction table. We then use the newest x bits of global history, which become available while the table lookup is still in progress, to select one counter from this group.

A bimodal predictor can similarly be pipelined, by using only the address bits from address(n - x) to initiate the table read. The bimodal predictor in this case becomes similar to a GAs [19] (also known as *gselect* [12]), but it uses a past address as opposed to the present address used by a GAs.

4 Ahead Pipelining a Perceptron Predictor

To bring the latency of the pipelined path-based perceptron down to a single cycle, it is necessary to decouple the table access for reading the weights from the adder. We note that using the address from the cycle $n - 1$ to initiate the reading of weights for the branch prediction in cycle n would allow a whole cycle for the table access, leaving the whole cycle when the prediction is needed for the adder logic. We can use the same idea as was used for the ahead pipelined table based predictors to inject one more bit of information (whether the previous branch was predicted taken or not taken) at the beginning of cycle n. We thus read two weights, select one based on the prediction which becomes

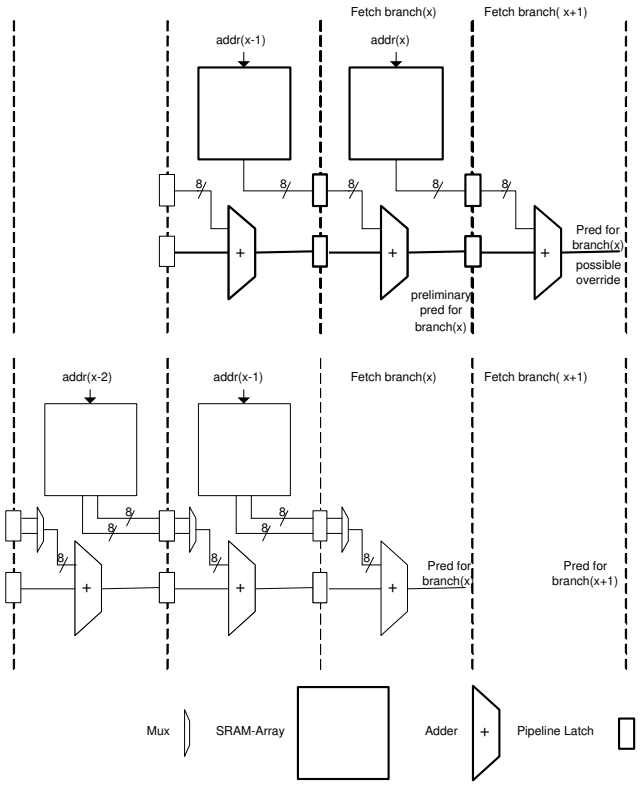


Figure 3: (top)The original proposal for a pipelined perceptron uses the current address in each cycle to retrieve the weights for the perceptron. (bottom) Our proposed design uses addresses from the previous cycle to retrieve two weights and then chooses between the two at the beginning of the next cycle. Note that the mux could be moved ahead of the pipeline latch if the prediction is available early enough in the cycle.

available at the end of cycle n-1, and use this weight to calculate the result for cycle n. While this means that one less bit of address information is used to retrieve the weights, perceptrons are much less prone to the negative effects of aliasing than table based predictors.

In the case of a branch misprediction, the pipeline has to be restored the same as an overriding perceptron. Because the predictor has to work at a one cycle effective latency, additional measures have to be taken. One possibility is to checkpoint not just the partial sums but also one of the two weights coming out of the SRAM arrays on each prediction. Only the weights which were not selected need be stored, because by definition, when a branch misprediction occurred, the wrong direction was chosen initially. A second possibility is to also calculate the partial sums along the not chosen path. This reduces the amount of state that needs to be checkpointed to only the partial sums, but necessitates additional adders. A third possibility is to only calculate the next prediction, for which no new information is needed, and advance all partial sums by one stage. This would lead to one less weight being added to the partial sums in the pipeline and a small loss in accuracy. The difference between options two and three is fluid and the number of extra adders, extra state to be checkpointed and any loss in accuracy can be weighed on a case by case basis.

For our simulations we assumed the first option, and leave evaluation of the second and third option for future work.

5 The Hashed Perceptron

The main idea behind the hashed perceptron predictor is that the gshare indexing approach can be applied within a perceptron predictor by using the exclusive OR of branch address and branch history to index into the perceptron weight tables.

However, we also found that combining hashed indexing with path-based indexing increased accuracy, i.e. hashed indexing only helps with some branches, not all.

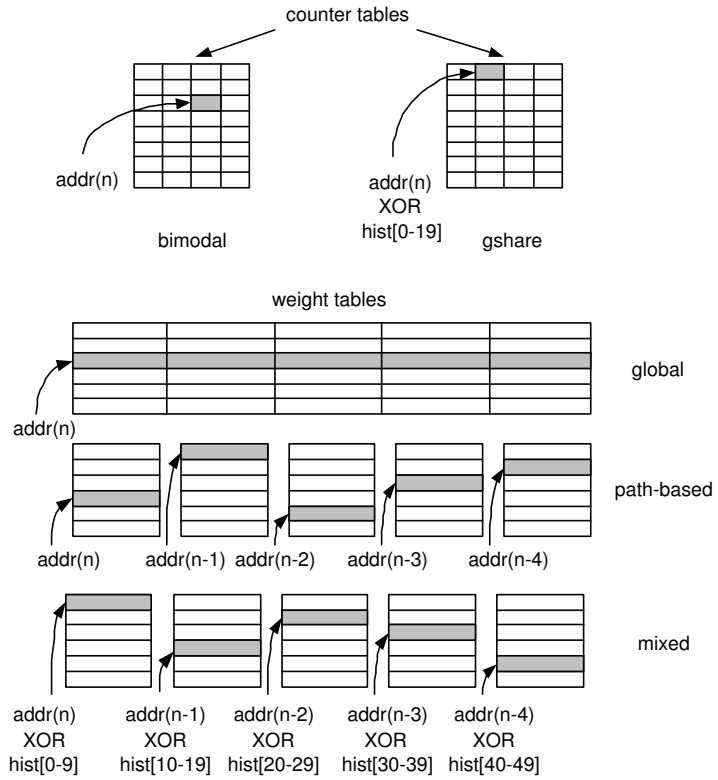


Figure 4: The hashed perceptron fetches weights by indexing into multiple tables with the exclusive OR of a branch address and the speculative global history.

5.1 The Idea of Hashed Indexing

In a traditional perceptron predictor, each history bit (whether global, local or path history) is assigned one weight. This means that the number of weights in each perceptron increases linearly with the number of history bits and so does the number of adders needed to compute the dot product. But this mapping from input (history bits and path information) to predictor state (perceptron weights) is not the only one possible. The mapping

$$index = address \bmod nr\_weights$$

is the same as used in a bimodal predictor, the only difference being that in a bimodal predictor the predictor state is a 2-bit counter and not an 8-bit weight. (In fact the 2-bit counter can be reinterpreted as a degenerate weight.) We look at the conventional global perceptron predictor as an accumulation of bimodal predictors, with each predictor being trained on the correlation between the history bit assigned to it and the outcome of the branch which is being predicted. The perceptron is trained by training all the predictors based upon the result of their cumulative prediction. Because each weight is 8-bits and not only 2-bits, weights assigned to branches with higher correlation can implicitly override the prediction of weights assigned to branches which do not correlate clearly with the outcome of the branch under consideration, if trained in such a fashion. Changing the mapping from only address to only history results in

$$index = history \bmod nr\_weights$$

or the perceptron equivalent of a GAg predictor. Ipek et al.[7] used such a mapping for their inverted perceptron, but because they inverted a global perceptron all weights in a perceptron still had the same mapping.

Merging these two mappings leads back to the concept of the gshare predictor as introduced by McFarling[12].

$$index = (address \oplus history) \bmod nr\_weights$$

The great advantage of applying the gshare concept to the path-based perceptron is that each weight is fetched with a different mapping, meaning it is not necessary to use the whole of the branch history we want to use to fetch each weight. Instead we can use different parts of the history for each weight. By using different parts of the history for fetching different weights we perform a *series* of partial pattern matches and use the sum of these individual predictions to make the overall prediction. By using a series of small gshare-like predictors instead of a single large predictor we can avoid some of the problems associated with large global predictors, such as excessive training time.

A fundamental problem of the previous perceptron predictors in comparison to two-level correlating predictors such as the gshare predictor, was that they could not reliably predict linearly inseparable branches[11]. The most common example of linearly inseparable branches are branches which are dependent on the exclusive OR of two previous branches.

A hashed perceptron predictor can predict these branches just like the gshare predictor, if they fall within one of the segments of global history (such as 0-9 or 20-29 in Figure 4) which are used to index into the different weight tables.

5.2 Combining Hashed and Path-Based Indexing

We have found that using path-based indexing for the most recent branches in combination with using hashed indexing for the rest of the branch history offers superior accuracy to pure hashed indexing.

The intuition behind this combination of hashed and path-based indexing is that the most recent branches are the most important for the outcome of the branch which is to be predicted. Assigning separate weights to them reduces aliasing and improves the chances that multiple paths can be distinguished. Techniques improving either one of these factors have enhanced the accuracy of many previous predictors.

5.2.1 Predicting a Branch

The pseudocode for the a pipelined version of the hashed perceptron is provided in Figure **??**. The hashed perceptron predictor consists of a number of weight tables organized into a pipeline, with each table and associated circuitry having its own pipeline stage. Let

- p be the number of pipeline stages/weight tables which use path-based indexing,
- h be the number of pipeline stages/weight tables which use hashed indexing,
- t be the number of history bits used to compute the hashed index in each pipeline stage
- n be the number of weights per table

Note that in this work for simplicity we assume that n is always a power of two and that t is always $log_2(n)$. $SR[0..h + p + 1]$ is an array which contains the partial sums for all the predictions in the pipeline at any given point in time. When predicting a branch, a final weight is added to $SR[0]$ to compute y, the total output of the perceptron predictor for that branch. If y is greater or equal to zero, the branch is predicted taken, and it is predicted not_taken otherwise.

5.2.2 Computing Predictions

The first p tables are accessed in parallel using the concatenation of the previous branch PC and the of that branch modulo n. The next h tables are accessed in parallel using the XOR of the previous branch PC and their respective t history bits.

For example, assume that $t = 10$, than the $(p + 1)^{th}$ table uses history bits 1-10 (history bit 0 being used for path-based indexing), the $(p + 2)^{th}$ table uses history bits 11-20 and so on.

Each of the fetched weights is added to the corresponding partial sum $SR[j]$ and stored in the next entry $SR[j - 1]$ in the array.

5.2.3 Updating the Predictor

Once the actual outcome of a branch is known, the addresses enter delay queues, which ensure that all weights which contributed to one prediction can be updated at the same time. The predictor is trained if the prediction was wrong

```
history: long                                          Speculative global branch history shift register
v[h + p] : integer                                     Array of indices of weights used

function prediction (history , last_pc : long) : { taken, not_taken}
y: integer                                             perceptron output
begin
        path_index := ((last_pc << 1) + (history & 1)) mod n    Hash last branch PC with last branch outcome
        y := SR[0] + W[path_index,0]                   Calculate perceptron output
        if y >= 0 then                                 Determine branch prediction
                prediction := taken
        else
                prediction := not_taken
        end_if
        v[0]  := path_index
        for j in 1..p - 1 in parallel do               Update the next p partial sums in flight
                SR'[j - 1] := SR[j] + W[path_index , j ]
                v[j]  := path_index                    Store index used for easy updating
        end for

        for j in 0..h - 1 in parallel do               Compute hashed indices for the next h weights
                hash_index_j := ((history >> (j*t + 1)) xor last_pc) mod n
                SR'[j + p - 1] := SR[j + p] + W[hash_index_j , j + p]
                v[j + p]  := hash_index_j              Store index used for easy updating
        end for
        SR := SR'
        SR[h + p] := 0                                 Initialize the partial sum for the (p + h) th branch
        history := (history << 1) or prediction        Update the speculative global history register
end
```

Figure 5: Pseudocode for the prediction function of a hashed percpetron

or if the absolute value of y was below the training threshold `theta`. The formula for `theta` is the same as for previous perceptrons, with the number of pipeline stages/weights replacing the number of history bits. All the weights are incremented if the outcome was taken and decremented otherwise. Note that saturating arithmetic has to be used because of the limited number of bits with which each weight is represented.

5.2.4 Checkpointing and Restoring the Predictor

When a branch was incorrectly predicted or another type recovery event occurs in the pipeline, the contents of *SR* have to be restored to the state they had prior to the misprediction. This means that *SR* has to be checkpointed for every branch in flight.

In total, a hashed perceptron with both hashed and path-based indexing has several advantages, some new and some incorporated from previous perceptrons:

- The hashed perceptron predictor can predict some linearly inseparable branches, something which traditional perceptron predictors cannot, as long as they are mapped to the same weight.
- Because the hashed perceptron predictor has a shorter pipeline for the same history length than a path-based perceptron, correlation between the weights and the outcome of the branch which is to be predicted is easier to establish.
- A shorter pipeline means less noise (as a consequence of aliasing with other branches) is injected into the prediction process.
- Separate weights for the most recent branches allows the hashed perceptron to distinguish between multiple paths leading up to a branch.

```
theta : integer                                   training threshold for training weights
H[h + p − 1, h + p − 1] : integer                 Delay queues for updating all the weights of one
                                                  prediction at the same time
function train (prediction, outcome: {taken , not_taken }, y : integer)
begin
    for j in 0..h + p − 1 in parallel do
        H[j , j] := v[j]                          Insert addresses into delay queues
        k := H[0 , j]                             Retrieve the index for the j^th weight
        if prediction != outcome or |y| <= theta then   If prediction was incorrect or below threshold
            if outcome = true then                Increment or decrement weight
                W[k , j] := W[k , j] + 1          in saturating arithmetic
            else
                W[k , j] := W[k , j] − 1
            end if
        end if
    end for
    shift_down(H)                                 Shift all entries down by one in the first index
end
```

Figure 6: Pseudocode for the prediction function of a hashed percpetron

Parameter	Configuration
L1-Icache	64KB, 32B, 2-way, 3 cycle latency
L1-Dcache	64KB, 32B, 4-way, 3 cycle latency
L2 unified cache	4MB, 128B, 8-way, 15 cycle latency
BTB	4096 entry, 4-way
Processor width	6
Branch Penalty	33
ROB entries	512
IQ, FPQ entries LSQ entries	64 128
L2 miss latency	200 cycles

Table 3: Configuration parameters of the processor simulated

6 Simulation Setup

We evaluate the different branch predictors using the 12 SPEC2000 integer benchmarks. All benchmarks were compiled for the Alpha instruction set using the Compaq Alpha compiler with the SPEC *peak* settings and all included libraries. Exploring the design space for new branch predictors exhaustively is impossible in any reasonable timeframe. To shorten the time needed for the design space exploration, we used 1-billion-instruction traces which best represent the overall behavior of each program. These traces were chosen using data from the SimPoint [17] project. Simulations were conducted using EIO traces for the SimpleScalar simulation infrastructure [3]. For our studies of the effects of delay on branch predictor accuracy, we used the sim-bpred simulator because of its greater speed. For the main evaluation of all predictors and to collect performance numbers, we used the a greatly enhanced version of the sim-outorder simulator, called sim-modes, from the Simplescalar [3] suite for simulating the accuracy and performance of all branch predictors.

For all the main simulations, sim-modes was run for 100M instruction prior to the beginning of the selected traces to warm up all caches and other microarchitectural structures. All statistics were restarted after this warmup period. The details of the processor model used can be found in Table 3.

7 Results

In this section we first show several examples of how using older information to make a branch prediction hurts accuracy in different predictors. We then go on to discuss how we chose the final configuration of the hashed perceptron and show performance results comparing the hashed perceptron to other predictors.

7.1 The Impact of Delay

First, we will present results from ahead pipelined versions of basic predictors to show that the impact of delay is universal and has similar but not equal effects on all branch predictors. Prediction accuracy degrades in a non-linear fashion for most predictors with increasing delay, most probably due to increasing aliasing between branches. We will then go on to show the impact of delay on the overriding perceptron predictor, which behaves similarly to table based predictors with respect to increasing delay, despite its very different structure.

Figure 7 shows the accuracy of a pipelined GAs predictor. We use only address bits to start the prediction and use the new history bits to select from the fetched predictions. This means that each predictor uses as many bits of global history as its delay in cycles. This of course implies that the predictor with 0 cycles of delay is in fact a bimodal predictor. As can be seen in Figure 7, the addition of global history bits increases the accuracy of such a predictor. The predictor goes from being a bimodal predictor to being a pipelined GAs predictor. For comparison we show non-pipelined GAs predictors with 0 to 4 bits of global history in Figure 8. Such predictors are more accurate than the pipelined GAs predictors with an equivalent number of global history bits. It should be noted that these are not meant to be optimal configurations for GAs predictors, but are only meant to illustrate the impact of using older branch addresses in such a predictor. In contrast to GAs, the gshare predictor in Figure 9 shows a consistent loss

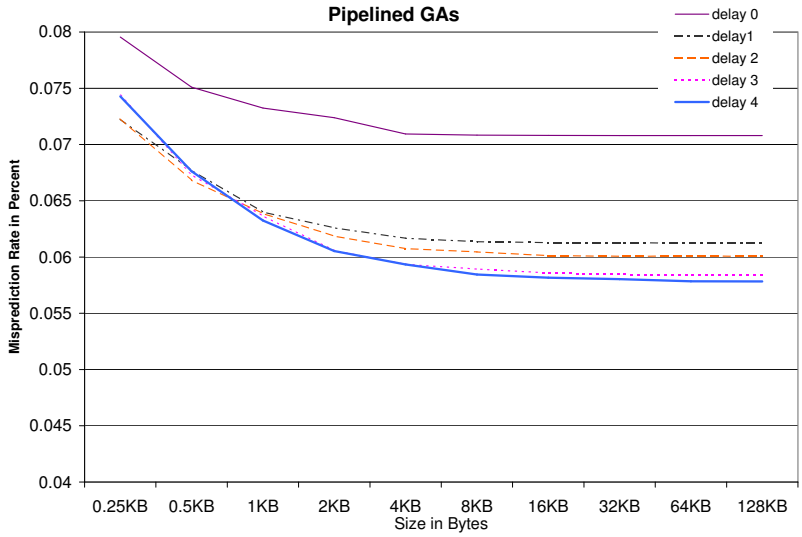


Figure 7: The impact on accuracy of using older addresses on a pipelined GAs predictor. The accuracy of the predictor actually improves with increasing delay, the inclusion of more bits of global history compensating for the effects of increasing delay.

of accuracy with increasing delay. However, the increase is not linear, with a predictor with one cycle delay showing only a very mild degradation in comparison to the normal gshare predictor. This can be attributed to the fact that the gshare predictor gains most of its accuracy from information in the global history and uses the address bits mainly to more evenly distribute usefull predictions across the whole table [12]. A GAs which uses little global history on the other hand is dependent upon good address information to make accurate predictions.

Figure 10 shows that the overriding perceptron predictor behaves similarly to the table based predictors; in that the loss of accuracy with increasing delay is not linear. However, it exhibits different behavior from the gshare predictor in that the impact of delay decreases much more quickly with increasing size. We attribute this to the very small number of perceptrons for the smaller hardware budgets, which necessarily means that fewer address bits are used. The loss of even one bit of address information seems to lead to greatly increased aliasing. At larger hardware budgets the

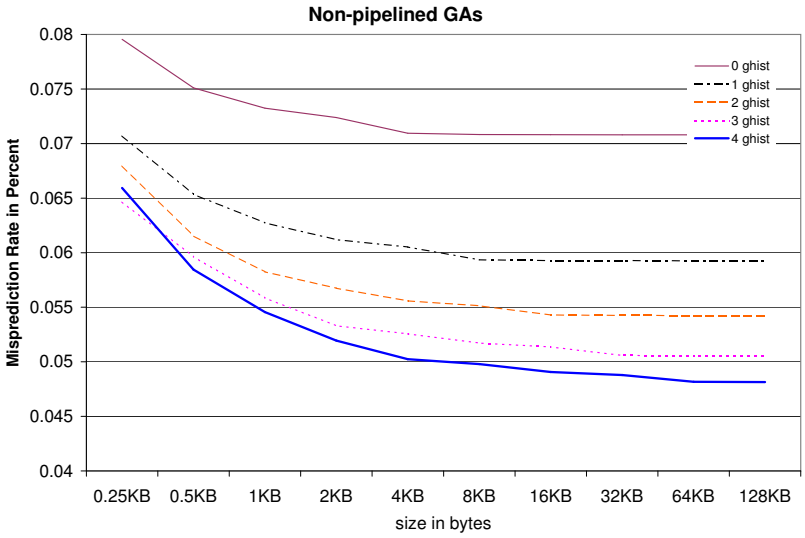


Figure 8: Accuracy of non-pipelined GAs predictors with zero (i.e., bimodal) to four bits of global history.

increasing number of perceptrons and the tendency of the perceptron not to suffer from destructive aliasing begin to dominate.

7.2 Pipelined Perceptron Predictors

In Figure 11 we compare the hashed perceptron to the path-based and global perceptron when varying the history length. Note that the hashed perceptron is always ahead-pipelined while the global and path-based perceptrons are not. The global and path-based perceptron predictors have access to h tables, where h is the history length, with each table containing 4K weights. The large number of weights per table should eliminate the effects of aliasing on this comparison.

The perceptron predictor using only hashed indexing (denoted as pure hash in Figure 11) has access to h/12 tables, because twelve bits of branch history are combined into one weight. The second predictor using hashed indexing (denoted as hash) has three tables which are accessed as in the path-based perceptron and the rest use hashed indexing.

The results show that the global perceptron predictor can make good use of long branch histories, while the path-based perceptron predictor outperforms at shorter history lengths. The pure hashed perceptron trails behind both the global and the path-based perceptron predictors, but adding separate tables for the three most recent branches improves the misprediction rate by 0.35 to 0.8 percent and outperforms the other predictors despite only having 1/8th of their storage budget.

We found that for optimal accuracy a hashed perceptron should separate out the last 5 bits of global history from the rest, meaning these are treated as in a regular ahead-pipelined path-based perceptron. As can be seen in Figure 11 the hashed perceptron can uses the maximum history length for optimal accuracy. Table 4 shows the configurations for hashed and overriding perceptrons used in all the following comparisons. Note that the pipeline depth for the hashed perceptron was 10 for all configurations, meaning 5 tables were indexed in a pipelined bimodal fashion (as in an ahead-pipelined perceptron) and 5 were indexed as in a gshare predictor.

It should be noted that the 20 and 40 kb hashed perceptrons access 2 and 4 kb tables respectively, which cannot be accessed within a single cycle. We thus also include a hashed perceptron (denoted as *fast hash* in Figure 12) which uses 1kb tables for the last 3 pipeline stages and larger tables for the hashed weights (we found that this yielded better performance than the usual 5 - 5 split in this case). The hashed weights can easily be fetched in two cycles, because they do not need the most recent address information to be effective. Note that the actual size of the fast hash is actually 17 and 31kb respectively.

We also include a gshare predictor as a baseline for comparisons. We do not pipeline the gshare predictor in any way, assuming that it always has one cycle latency. Note that all the following comparisons are with gshare predictors which are 60% larger than the perceptrons they are being compared to. We chose this comparison to illustrate the hardware saving made possible by using a perceptron predictor.

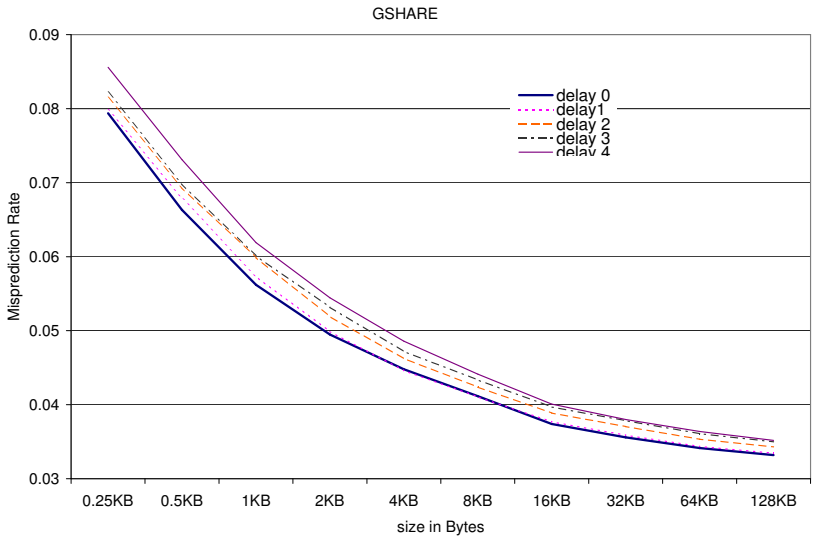


Figure 9: The impact on accuracy of using older addresses on a pipelined gshare predictor. Accuracy decreases with increasing delay. However a one cycle delay leads to only minimal loss of accuracy.

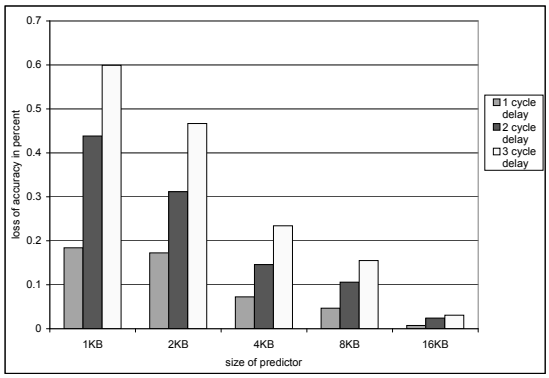


Figure 10: The impact on accuracy of using older addresses to fetch the perceptron weights.

Figure 12 shows the arithmetic mean IPC for the hashed perceptron, path-based perceptron and gshare predictors. It can be seen that the hashed perceptron offers superior performance at all hardware budgets. The hashed perceptron offers an improvement in IPC of about 4% at 1.25 to 20KB and increases its lead to 5.8% at 40KB. For very aggressive clockrates, a fast hashed perceptron of 31KB still offers a 4.8% improvement over a 40KB overriding perceptron.

The misprediction rates in Figure 12 show a similar picture, with the improvement in misprediction rate steadily improving from 11% at 1.25KB to almost 21% at 40KB. The fast hashed perceptron offer improvements of about 17% in misprediction rate compared to somewhat larger overriding perceptrons.

Gcc, crafty and parser show the largest improvement relative to a overriding perceptron. These are benchmarks with relatively large branch footprints. This indicates that the larger number of weights per table in a hashed perceptron can reduce the negative impact of aliasing relative to an overriding perceptron of the same size. We exclude gap from this group since the misprediction rates for all predictors are so low that no clear conclusion can be drawn.

The results repeat themselves at 40KB, with major improvements again seen in gcc, crafty and parser.

8 Conclusion and Future Work

We have introduced the hashed perceptron predictor, which merges the concepts behind the gshare and path-based perceptron predictors. This predictor has several advantages over prior proposed branch predictors:

- The hashed perceptron improves branch misprediction rate by 20% over a path-based perceptron on the SPEC2000

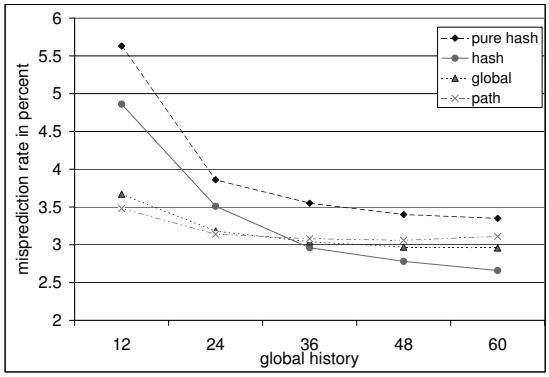


Figure 11: Misprediction rates for the global, path-based and hashed perceptron predictors when varying the history length from 12 to 60 bits.

Size (kb)	**history length: hash**	**history length: path**
1.25	40	20
2.5	45	20
5	50	20
10	55	40
20	60	40
40	64	40

Table 4: Configurations for the hashed and overriding perceptron

integer set of benchmarks, increasing IPC by over 5.8%.

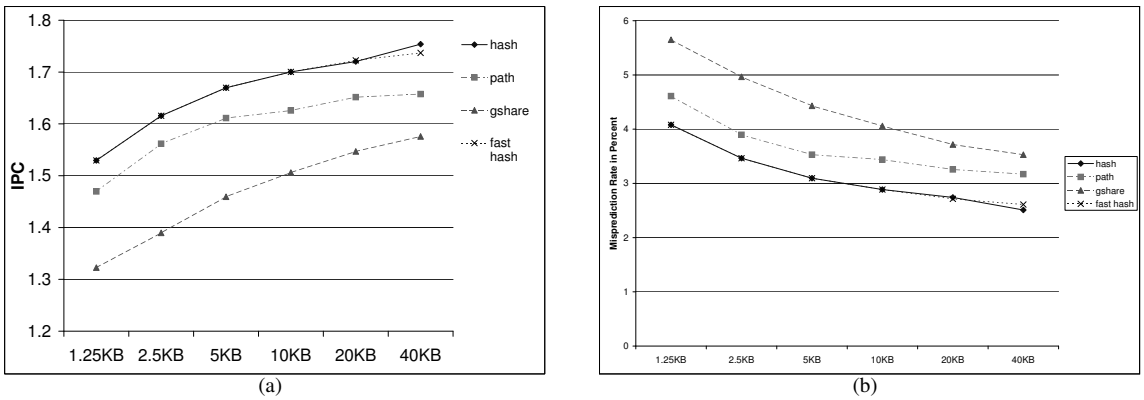
- The hashed perceptron reduces the number of adders by a factor of four and shortens the predictor pipeline by the same factor.
- The amount of state that needs to be checkpointed and restored in case of a branch misprediction is also reduced by a factor of four.
- The update logic is greatly simplified by only having to keep track of 10 weights instead of 40 for each branch.
- By ahead pipelining the hashed perceptron predictor the overhead and added complexity of associated with having a large predictor overriding a smaller predictor are eliminated.

The hashed perceptron eliminates the need for a preliminary predictor and overriding mechanism, it offers superior accuracy starting at low hardware budgets and scales better than previous designs to larger configurations. It is an small enough, fast enough and simple enough to be a promising choice as a branch predictor for a future high-performance processor.

We think the hashed perceptron offers a good base for further research: The introduction of gshare-style indexing to perceptron predictors should allow many of the techniques developed to reduce aliasing and increase accuracy in two-level correlating predictors to be applied to perceptron predictors. In the other direction, it might be possible to use the idea of matching multiple partial patterns to increase accuracy in two-level correlating predictors.

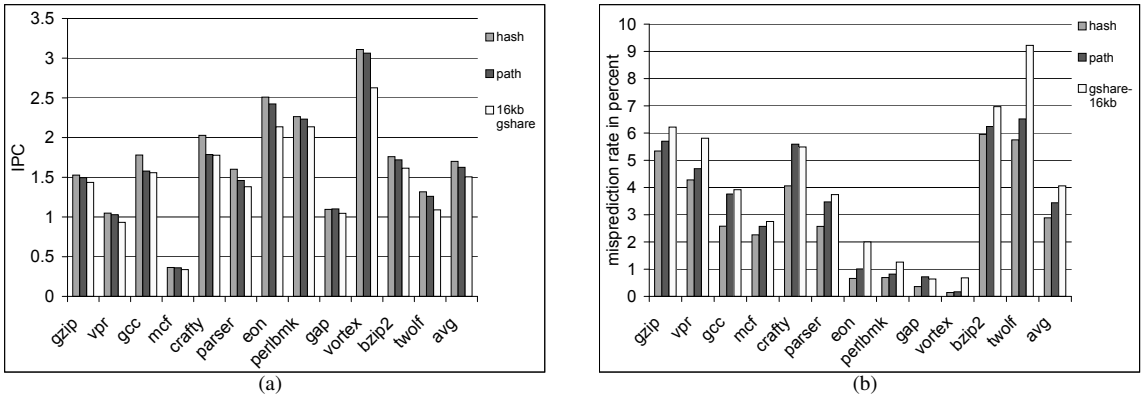
9 Acknowledgments

This work is supported in part by the National Science Foundation under grant nos. CCR-0105626, EIA-0224434, CCR-0133634, a grant from Intel MRL and an Excellence Award from the Univ. of Virginia Fund for Excellence in Science and Technology. The authors would also like to thank Mircea R. Stan for many fruitful discussions about neural networks.



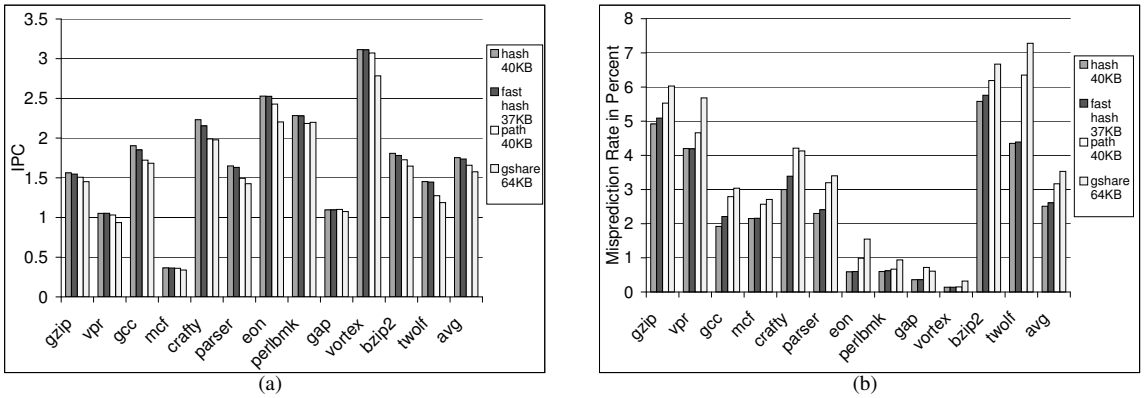
(a) (b)

Figure 12: IPC and misprediction rates (in percent) for the hashed and overriding perceptrons, as well as non-pipelined gshare predictors. Note that the gshare predictors are always 60% larger than the perceptron predictors, i.e. 2,4,8,16,32 and 64KB.



(a) (b)

Figure 13: IPC and misprediction rates in percent for the 12 SPEC2000 integer benchmarks for the hashed and overriding perceptrons, as well as agshare predictor. Note that the gshare predictor is 16kb while the hashed and overriding perceptron predictors only have 10kb storage budgets.



(a) (b)

Figure 14: IPC and misprediction rates in percent for the 12 SPEC2000 integer benchmarks for the hashed, fast hashed and overriding perceptrons, as well as a gshare predictor. Note that the gshare predictor is 64kb while the hashed and overriding perceptron predictors only have 40kb storage budgets. The fast hashed perceptron has a 31kb storage budget.

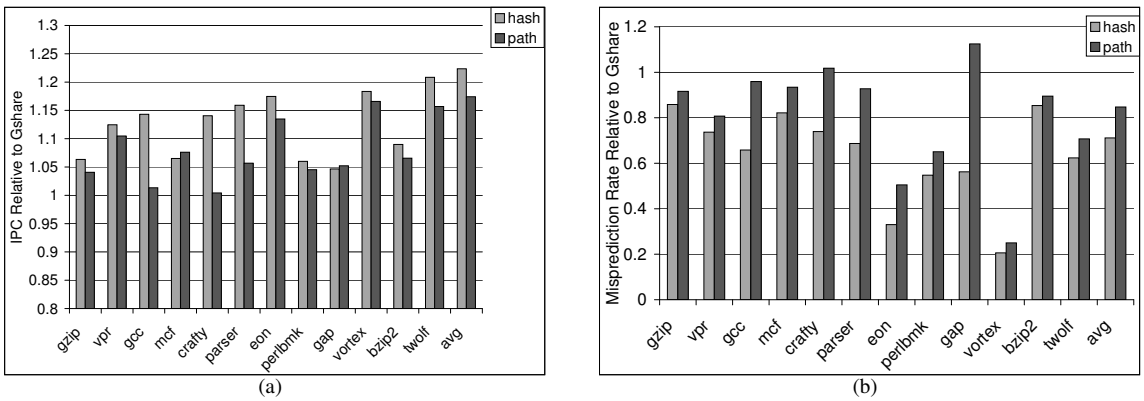


Figure 15: Relative IPC and misprediction rates for the 12 SPEC2000 integer benchmarks for the hashed, path-based and gshare predictors. Note that the gshare predictor is 16kb while the two perceptron predictors only have a 10kb storage budget.

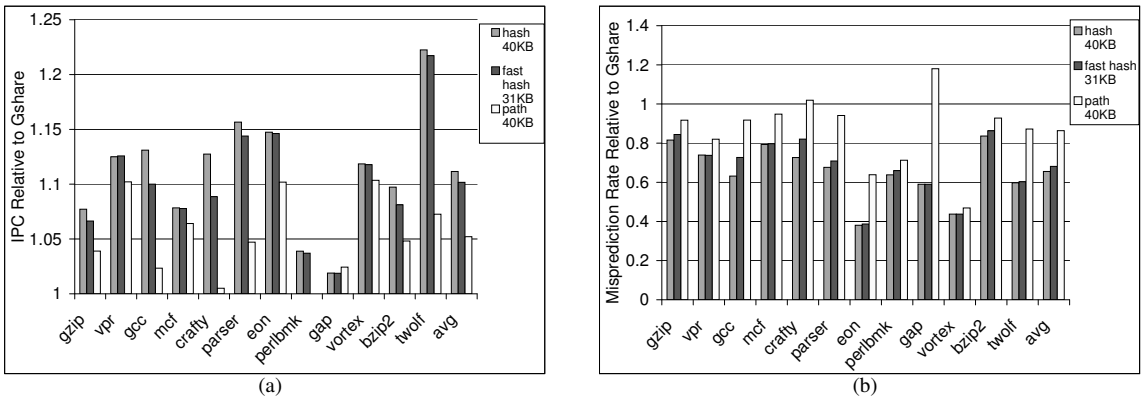


Figure 16: Relative IPC and misprediction rates for the 12 SPEC2000 integer benchmarks for the hashed, path-based and gshare predictors. Note that the gshare predictor is 16kb while the two perceptron predictors only have a 40kb storage budget.

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 Annual IEEE/ACM International Symposium on Microarchitecture*, page 423. IEEE Computer Society, 2003.

[2] D. Boggs, A. Baktha, J. M. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1), February 2004.

[3] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.

[4] B. Calder and D. Grunwald. Next Cache Line and Set Prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 287–296. ACM Press, 1995.

[5] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-Order Commit Processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 48. IEEE Computer Society, 2004.

[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 5(1):13, February 2001.

[7] E. Ipek, S. A. McKee, M. Schulz, and S. Ben-David. On Accurate and Efficient Perceptron-Based Branch Prediction. Unpublished Work.

[8] D. Jiménez. Reconsidering Complex Branch Predictors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 43–52, 2003.

[9] D. A. Jiménez. Fast Path-Based Neural Branch Prediction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 243. IEEE Computer Society, 2003.

[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 Annual ACM/IEEE International Symposium on Microarchitecture*, pages 67–76. ACM Press, 2000.

[11] D. A. Jiménez and C. Lin. Neural Methods for Dynamic Branch Prediction. *ACM Trans. Comput. Syst.*, 20(4):369–397, 2002.

[12] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[13] A. Moshovos. Checkpointing Alternatives for High Performance, Power-Aware Processors. In *Proceedings of the 2003 International Symposium on Low Power Wlectronics and Design*, pages 318–321. ACM Press, 2003.

[14] A. Seznec. Redundant History Skewed Perceptron Predictors: pushing limits on global history branch predictors. Technical Report 1554, IRISA, September 2003.

[15] A. Seznec. Revisiting the Perceptron Predictor. Technical Report 1620, IRISA, May 2004.

[16] A. Seznec and A. Fraboulet. Effective Ahead Pipelining of Instruction Block Address Generation. In *Proceedings of the 30th Annual International Symposium on Computer architecture*, pages 241–252. ACM Press, 2003.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior, 2002. Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[18] L. Vintan and M. Iridon. Towards a High Performance Neural Branch Predictor. In *Proceedings of the 9th International Joint Conference on Neural Networks*, pages 868–873, July 1999.

[19] T.-Y. Yeh and Y. N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266. ACM Press, 1993.