Merging Path and Gshare Indexing in Perceptron Branch Prediction

UNIV. OF VIRGINIA DEPT. OF COMPUTER SCIENCE TECH. REPORT CS-2004-38
DEC. 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 the *hashed* perceptron predictor, which merges the concepts behind the gshare, path-based and perceptron branch predictors. This predictor can achieve superior accuracy to a path-based and a global perceptron predictor, previously the most accurate dynamic branch predictors known in the literature. We also show how such a predictor can be ahead pipelined to yield one cycle effective latency. On 11 programs from the SPECint2000 set of benchmarks, the hashed perceptron predictor improves accuracy by up to 22% over a path-based perceptron and improves IPC by up to 6.5%.

1 Introduction

The trend in recent high-performance commercial microprocessors has been towards ever deeper pipelines to enable ever higher clockspeeds [2, 7], with the issue width staying about the same from earlier designs. This trend increases pressure on the branch predictor from two sides. First, the increasing branch misprediction penalty increases emphasis on the accuracy of the branch predictor. Second, the decreasing cycle time makes it difficult to use large tables or complicated logic to perform a branch prediction in one cycle.

A second major trend has been the emergence of power as a fundamental design constraint on microprocessor design. The increasing use of computers in a mobile setting has also put a premium on energy efficiency. Branch predictors have a large degree of leverage on both factors. Improvements in the branch predictor, a relatively small subunit of the whole processor, can lead to disproportionate improvements for the whole processor in power and energy efficiency [14].

Most branch predictors explored in the last ten years have been based on tables of two-bit saturating counters. The perceptron predictor is a new kind of predictor which is based 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 operate every cycle they make a prediction, increasing both the area of the predictor and the energy per prediction.

Previous perceptron predictors assign one weight per local, global or path branch history bit. This means that the amount of storage and the number of adders increases linearly with the number of history bits used to make a prediction. One of the key insights of this paper is that the one-to-one 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 (a process we call *hashed* indexing), a perceptron can work on multiple partial patterns making up the overall history.

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.

Most large table-based and perceptron predictors cannot make a prediction in a single 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 [8, 10, 17] 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 [10] was also proposed. 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.

The main contributions of this paper are:

- We show that the one-to-one correlation of weights to number of history bits in a perceptron is not necessary.
- A perceptron predictor using hashed indexing can perform equally well or better than a global or path-based perceptron while having an order of magnitude fewer adders.
- Combining multiple ways to index weights in a single perceptron improves accuracy over using only a single way.
- A perceptron can be ahead pipelined to reduce its effective latency to one cycle, obviating the need for a 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 introduces hashed indexing and explores its benefits. Section 4 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 5 shows how a perceptron predictor can be ahead pipelined to yield a one cycle effective latency. Section 6 describes our simulation infrastructre, Section 7 compares the accuracy and performance of the different predictors and Section 8 finally concludes.

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 the dot-product is then used as the prediction. In the context of a global perceptron [9] branch predictor, 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 stored in an SRAM array, where n is typically 8 in the literature. The input vector consists of 1's for taken and -1's for not taken branches. The dot-product can then be calculated using a Wallace-tree adder [5], with no multiplication circuits needed.

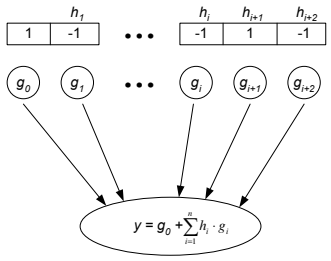


Figure 1: The global perceptron assigns weights to each element of the branch history and makes its prediction based on the dot-product of the weights and the branch history plus a bias weight to represent the overall tendency of the branch. Note that the branch history can be global, local or something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

2.2 Related Work

The idea of the perceptron predictor was originally introduced by Vintan and Iridon [19] and Jiménez and Lin showed in [12] 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 [10] 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.

Seznec proposed several improvements to the original global perceptron in [15, 16]. In [16] 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. Our work was partly inspired by [16] 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 needs to calculate 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.

cornell-perc 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 The Hashed Perceptron

There are two main insights behind the hashed perceptron predictor:

1. The gshare indexing approach can be applied to the perceptron predictor.

2. The global branch/path history can be subdivided into multiple sections to perform a series of partial pattern matches instead of a single one as in a traditional predictor.

To better distinguish between the different perceptron predictors which have been proposed, we introduce a new taxonomy. We classify perceptron according to whether they use one or multiple indices to fetch all their weights, what type of information they use to fetch their weights and what type of information they use for their input vector. A perceptron predictor can thus be described by a string

A_B_C :
A = `si` : single index or `mi` : multiple index
B and C = `A` : address, `G` : global history, `P` : path history, `L` : local history, `X` : nothing
B and C = YsZ : Y $\oplus$ Z (Y *shared with* Z)
B and C = YcZ : Y *concatenated with* Z

3.1 The Idea of Hashed Indexing

In the global perceptron [12], each history bit 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

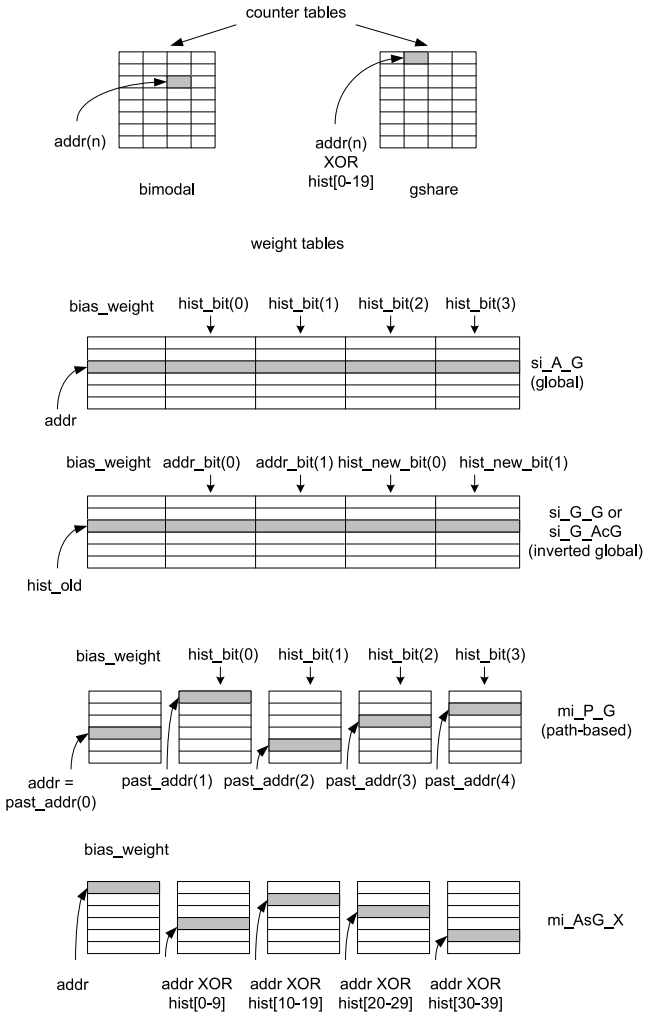


Figure 2: When using gshare-style hashed indexing, weights are fetched by indexing into multiple tables with the exclusive OR of a branch address and a subsection of the speculative global history.

dot product. The mapping

$$\texttt{si\_A\_G} : index = address \bmod nr\_perc$$

is used to fetch *all* the weights. The `si_A_G` mapping (second from the top in Figure 2) 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 a large number of 8-bit weights. (In fact the 2-bit counter can be reinterpreted as a degenerate weight.) We look at the conventional global perceptron predictor as an evolution of the bimodal predictor. All the weights are read from the tables using the same piece of information and each weight is trained on the correlation between the history bit assigned to it and the outcome of the branch which is being predicted. The history used can be be global history, local history or a combination of the two as in the alloyed perceptron [12].

But this mapping is not the only one possible. The path-based perceptron [10] (second from the bottom in Figure 2) fetches each weight based upon a different piece of information, namely the PC of a particular past branch. The mapping for the i^{th} weight becomes

$$\texttt{mi\_P\_G}\text{: } index_i = past\_branch\_address_i \bmod nr\_weights\_per\_table$$

$$(past\_branch\_address_0 = address)$$

This means that the path-based perceptron can distinguish between different branches which are in the same position in the branch history. This extra information helps the path-based perceptron to potentially be more accurate than the global perceptron. However, the same weight can now be used to predict different branches, introducing a source of aliasing or noise into the prediction process.

Up to this point the global branch history was only used as the input vector. It could also be used for indexing the weights.

Changing the mapping from only address to only history results in

$$\texttt{si\_G\_G}\text{: } index = history \bmod nr\_perc$$

or the perceptron equivalent of a GAg predictor. In this case all the weights are fetched based on a part of the global branch history. Each weight is then trained on the correlation with one of the history bits not used for the indexing. This approach was tried by cornell-perc, but they also included some address bits in the input vector to better distinguish between different branches. Note that address bits could also be used as part of the input vector of the global perceptron.

Merging the two mappings leads back to the concept of the gshare predictor as introduced by scott93combining.

$$\texttt{si\_AsG\_AsG} : index = (history \oplus address) \bmod nr\_perc$$

This mapping still leaves the fact that an increase in the amount of history and/or address used leads to a one-to-one increase in number of weights and adders. Using the idea from the mi_P_G of using different pieces of information to fetch each weight, we can break the global history into multiple pieces. The resulting mapping then is

$$\texttt{mi\_AsG\_X} : index_i = ((history \gg i * hist\_per\_weight) \oplus address) \bmod nr\_weights\_per\_table$$

For a given amount of history the number of weights is now reduced from ***history - log(nr_perc)*** to ***history/hist_per_weight***.

Starting from the path-based perceptron we can arrive at a very similar mapping. Again, the goal is reducing the number of weights while keeping the amount of history constant. Two bits of history can be assigned to one weight by changing the mapping to

$$\texttt{mi\_PcG\_X} : index_i = ((past\_branch\_address_{2i} \ll 1) \oplus history\_bit_{2i+1}) \bmod nr\_weights\_per\_table$$

The above approach can be continued, but the extra history bits come at the expense of address bits from past branches. Also, using only a sparse path history will lessen the positive effect of using path history. The gshare approach once again seems a natural solution, with the mapping ((at the bottom in Figure 2)) being

$$\texttt{mi\_PsG\_X} : index_i = ((history \gg i * hist\_per\_weight) \oplus past\_branch\_address_i) \bmod nr\_weights\_per\_table$$

If path information is critical, another evolution of the path-based perceptron is also possible:

$$\texttt{si\_PsP\_G} : index_i = (past\_branch\_address_{2i} \oplus (past\_branch\_address_{2i+1} \ll 1)) \bmod nr\_weights\_per\_table$$

3.2 Implications of Using Hashed Indexing

In all of the proposed mappings, multiple past branches are assigned to a single weight. This breaks the clean symmetry with the input vector. For these mappings we thus set all the elements of the input vector equal to one.

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[12]. The most common example of linearly inseparable branches are branches which are dependent on the exclusive OR of two previous branches.

Using hashed indexing, linearly inseparable branches which are mapped to the same weight can be accurately predicted, because each table acts like a small gshare predictor.

3.2.1 Computing Predictions

Below we show pseudocode for predicting a branch and updating the predictor for both single index and multiple index perceptrons. Note that the mi-perceptron functions assume no input vectors.
Let

- *h* be the number of pipeline stages/weight tables
- *hist_per_table* be the number of history bits used to compute the hashed index for each weight
- *n* be the number of weights per table

Note that in this work for all perceptrons using hashed indexing we assume that *n* is always a power of two and that $hist\_per\_table$ is always $log_2(n)$.

function *si_perc_pred* (*pc, history* : long): {1,-1}
begin
index:= **hash_index**(*pc,history*) **mod** *n*
input_vector:= **hash_input**(*pc,history*)
IV:= **conv_bipolar**(*input_vector*)
out := W[0,*index*] + $\sum_{j=1}^{h}$ IV[*j*]*W[*j*,*index*]
if out $\geq$ 0 **then**
prediction := 1
else
prediction := -1
end if
end

As an example of hashed indexing we show the algorithm for `mi_AsG_X` :

function *mi_perc_pred* (*past_pc*[], *history* : long): {1,-1}
begin
index[0] := *past_pc*[0]) **mod** *n*
out := W[0,*index*[0]]
for *j* **in** 1 .. *h* **do**
index[*j*] := ((*history* $\gg$ *j*hist_per_table*) $\oplus$ *past_pc*[0]) **mod** *n*
out := *out* + W[*j*,*index*[*j*]]
end for
if out $\geq$ 0 **then**
prediction := *taken*
else
prediction := *not_taken*
end if
end

3.2.2 Updating the Predictor

The predictor is trained if the prediction was wrong or if the absolute value of out was below the training threshold θ. The formula for θ is the same as for previous perceptrons, with the number of pipeline stages/weights replacing the number of history bits. If no input vector is used (such is in `mi_AsG_X`), all 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.

function *si_perc_train* (*index, out* : integer,*prediction, outcome*: {1,-1})
begin
if *prediction* $\neq$ *outcome* **or** $|out| \leq \theta$ **then**
W[0,*index*] := W[0,*index*] + *outcome*
for *j* **in** 1 .. *h* **do**

```
            W[j,index] := W[j,index] + IV[j]*outcome
        end for
    end if
end
```

```
function mi_perc_train (index[], out : integer,prediction, outcome: {1,-1})
begin
    if prediction ≠ outcome or |out| ≤ θ then
        for j in 0 .. h do
            W[j,index[j]] := W[j,index[j]] + outcome
        end for
    end if
end
```

3.3 Evaluating Hashed Indexing

The different mappings were evaluated for accuracy across a range of sizes and number of tables by varying the number of tables at a given size. Note that for clarity only the best mappings are shown.

The si_A_G, mi_P_G and mi_PcG_X were the best at smaller sizes, as can be seen in Figure 3. All of them improve with more tables and thus longer histories. Starting at 2KB the mi_PcG_X needs fewer separate tables to reach a given accuracy than the other two predictors. This shows that the mi_PcG_X can take advantage of the longer history over the mi_P_G at a given number of tables. Because the mi_P_G suffers from less aliasing than the mi_PcG_X it outperforms it at 1 and 2KB.

In Figure 4 it can be seen that at larger sizes the mi_PsG_X and mi_AsG_X dominate. We include the mi_PcG_X as a stand in for the si_A_G and mi_P_G predictors. Interestingly the mi_PsG_X performs worse than the mi_AsG_X. It seems that the extra path information doesn't give better correlation with the branch to be predicted than simply using the branch address.

Both mi_PsG_X and mi_AsG_X suffer more from aliasing than the other predictors, as can be seen from their performance at small sizes. This makes intuitive sense, since they are similar to gshare in how they access their tables. The mi_P_G is similar to a bimodal predictor in indexing, and it is well known, that bimodal predictors suffer less from aliasing than a simple gshare predictor, especially at small sizes.

We can conclude that the mi_P_G and mi_PsG_X are best at small sizes, while the mi_AsG_X is the best at larger sizes.

3.4 Combining Multiple Mappings in One Predictor

Ideally we would like to combine the advantages of the different mappings in one predictor. The solution to this problem is to combine multiple mappings in one predictor. Different weights of the same perceptron are fetched using different mappings, but are still trained as one perceptron.

The approach used in this work is to combine a short mi_P_G or mi_PsP_X perceptron (which both belong to the class of path-based perceptrons) with an mi_AsG_X perceptron. The reasoning behind this choice is as follows:

- Most branches are either highly biased or show very good correlation with a branch in the very recent past. A short si_A_G or mi_P_G could predict these kind of branches well.

- Some branches need path information to be accurately predicted. This fact, as well as its superior accuracy, leads to the choice of mi_P_G over si_A_G.

- Some branches need as much global branch history as possible to be accurately predicted. The mi_AsG_X can use the most global history with the fewest weights and is more accurate than mi_PsG_X.

3.5 Accuracy of a Multiple Mapping Perceptron

When using the multiple mappings as described above, there is a tradeoff at any given size between the total number of tables and the size of each table. A second tradeoff is between the two different mappings, that is to say how

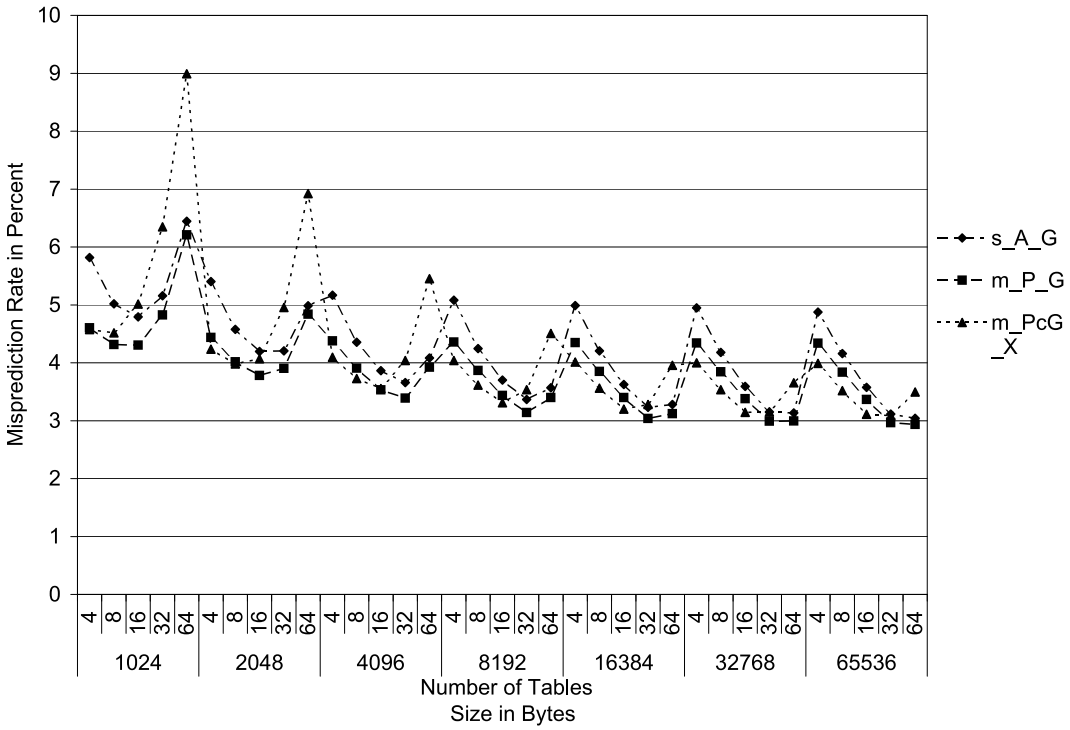


Figure 3: The number of tables and weights is varied from 4 to 64 at each size for all predictors. All predictors improve with more weights/longer histories.

many weights should be fetched using one mapping and how many using the other mapping. A third tradeoff is whether/when to use `mi_P_G` or `mi_PsP_X`. In Figure 5 we show some of the design space for all these tradeoffs.

The number of weights fetched using `mi_P_G` or `mi_PsP_X` is varied from 2 to 4 and the number of weights fetched using `mi_AsG_X` is varied from 3 to 7. One weight is always used as the bias weight. The size of the tables is varied from 128 Bytes to 4KB. For clarity `mi_P_G` is shown from 640 Bytes to 5KB, where it is the more accurate option, `mi_PsP_X` is shown from 3.5 to 96KB.

We can draw several conclusions from Figure 5:

- The hashed perceptron outperforms both its component mappings at all sizes.
- Using only 2 weights with path information seems to be enough.
- Using as much global history as possible improves accuracy.
- Using a greater number of smaller tables instead of fewer larger tables helps accuracy.

3.6 Advantages of a Hashed Perceptron

In total, a hashed perceptron with both `mi_AsG_X` and `mi_P_G` mappings has several advantages, some new and some incorporated from previous perceptrons:

- The hashed perceptron predictor can accurately 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(`mi_P_G`) perceptron, correlation between the weights and the outcome of the branch which is to be predicted is easier to establish.

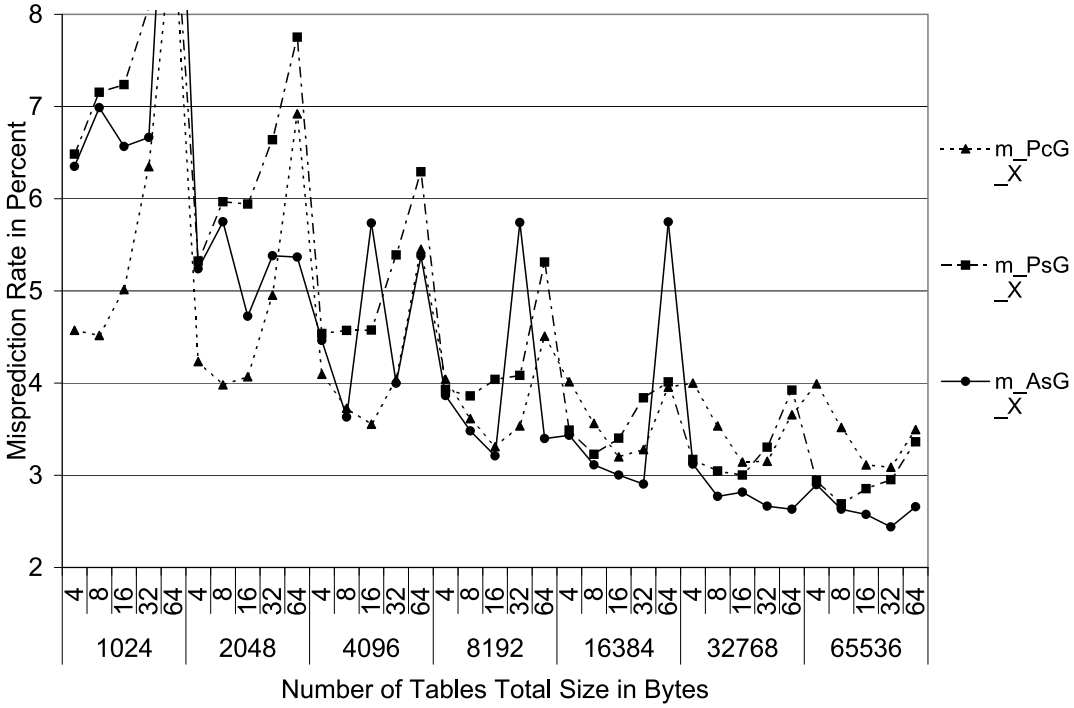


Figure 4: The number of tables and weights is varied from 4 to 64 at each size for all predictors. `mi_AsG_X` and `mi_PsG_X` show the effects of aliasing and destructive interference at small sizes.

- A shorter pipeline means less noise due to aliasing 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.

4 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.

4.1 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 [11] 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.

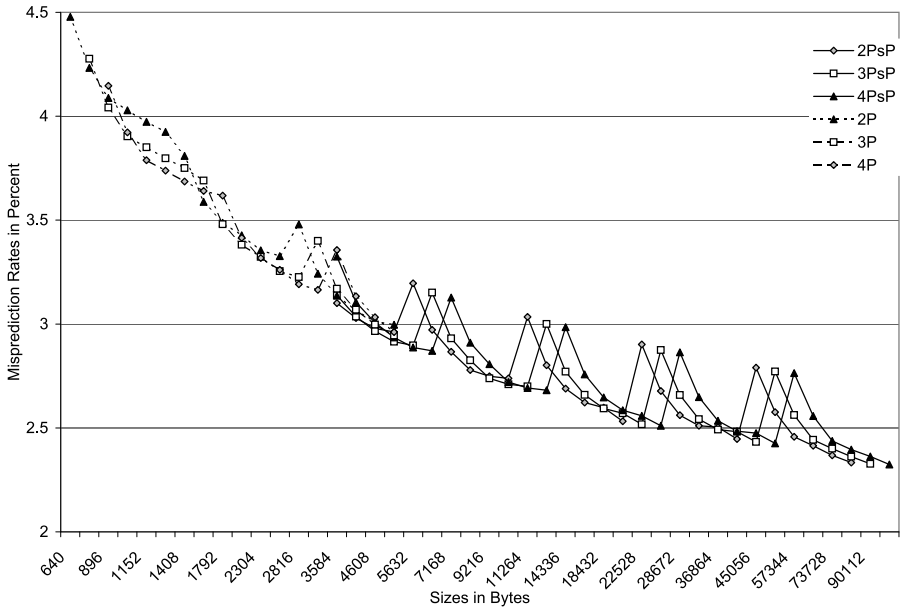


Figure 5: The hashed perceptron shows good scaling with increasing size. Each time the maximum number of tables is reached the table size is doubled. Predictors with larger but fewer tables underperform predictors with more but smaller tables. This leads to the clearly visible sawtooth pattern.

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, 6] 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.

4.2 Ahead-Pipelined Predictors

A solution to this problem, which was introduced in [10], 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.

4.3 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.

predictor type	Amount of state to be checkpointed in bits
path-based 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:

This problem was briefly mentioned in [17] in the context of 2BCgskew 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 pipelined 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 across 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.

It is obvious that a hashed perceptron, which has a much shorter pipeline than a tuned global or path-based perceptron, is easier to design under such constraints.

5 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 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.

Note that there is also the possibility that no branch needs to be predicted in a certain cycle (We do not show the associated circuitry in Figure 6 for reasons of clarity). In this case the pipeline does not advance. For this case the old partial sums need to be kept in an additional shadow latch, which loads its contents into the normal pipeline latch if no signal from the BTB or from predecode bits is received.

In the case of a branch misprediction, the pipeline has to be restored the same as a normal path-based perceptron.

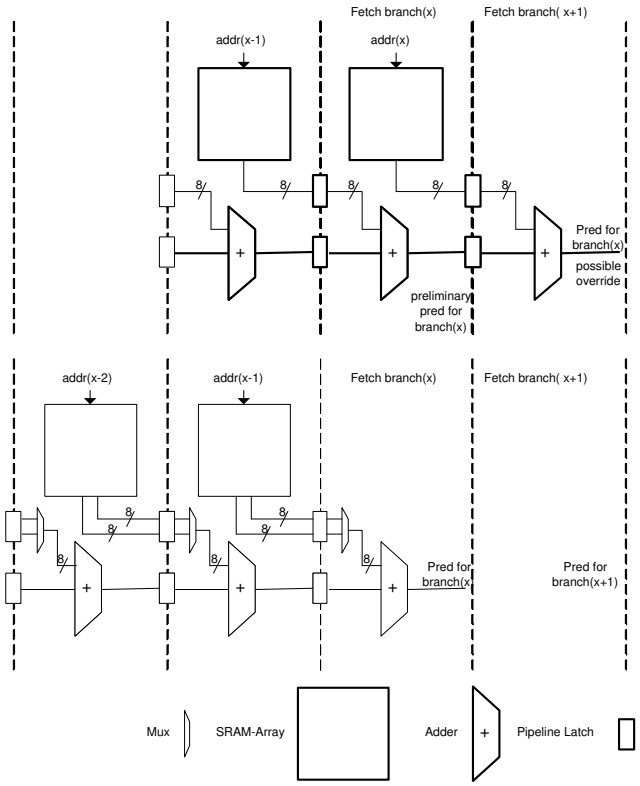


Figure 6: (top)The original proposal for a pipelined perceptron uses the current address in each cycle to retrieve the weights for the perceptron. (bottom) Our proposed design uses addresses from the previous cycle to retrieve two weights and then chooses between the two at the beginning of the next cycle. Note that the mux could be moved ahead of the pipeline latch if the prediction is available early enough in the cycle.

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.

5.1 Ahead Pipelining the Hashed and Path-Based Perceptrons

To minimize the impact of ahead pipelining on the path-based perceptron, we ahead pipeline only the bias weight. This means that the sum of three small integers needs to be calculated in the final cycle.

The hashed perceptron is a more complex case. The weights fetched using `mi_AsG_X` do not have to be ahead pipelined. We simply move fetching them a couple of cycles ahead of the actual branch. This means that a previous branch address has to be used for indexing the weights, and that the most recent history bits cannot be used, because they are not yet available. All weights fetched using `mi_P_G` or `mi_PsP_X` are ahead pipelining as described in the above Section.

An ahead pipelined global perceptron was explored in [16], but we do not include it here, since an ahead pipelined global perceptron without all the additional improvements presented in [16] would be clearly inferior to the other predictors.

6 Simulation Setup

We evaluate the different branch predictors using all SPEC2000 integer benchmarks except mcf. The performance of mcf is totally dominated by L2 cache misses and its performance shows no sensitivity to branch prediction. 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 [18] project. Simulations were conducted using EIO traces for the SimpleScalar simulation infrastructure [3]. All predictor were tuned for optimal accuracy at all sizes.

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

We first compare the different perceptrons without ahead pipelining, to show the upper bound on a number of configurations. The history lengths for the tuned predictors can be found in Table 4.

The harmonic mean misprediction rates and IPC for the global, path-based and hashed perceptrons are shown in Figures 7 and 7.

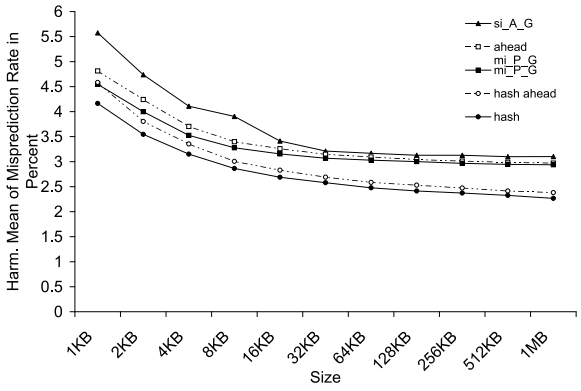
The hashed perceptron is superior at all sizes, showing an improvement of 18.25 % in misprediction rate over the path-based perceptron at 64KB. It also scales better than the other two predictors, increasing its lead in misprediction rate from 8.45 % at 1KB to 22.8% at 1MB. The individual misprediction rates shown in Figure 7 show that the hashed perceptron is the most accurate predictor in all 11 benchmarks. The biggest improvements can be seen in twolf, eon and gcc. The more accurate branch prediction leads to better overall IPC, with an improvement of 5% at 64KB for an

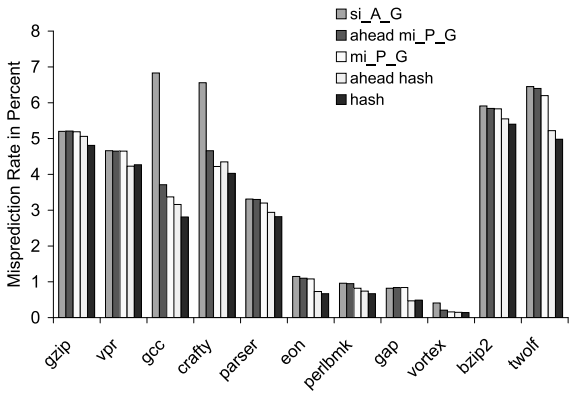
Parameter	Configuration
L1-Icache	64KB, 32B, 2-way, 3 cycle latency
L1-Dcache	64KB, 32B, 4-way, 3 cycle latency
L2 unified cache	4MB, 128B, 8-way, 15 cycle latency
BTB	4096 entry, 4-way
Indirect Branch Predictor	512 entry, 8-way
Processor width	6
Branch Penalty	33
ROB entries	512
IQ, FPQ entries LSQ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

Table 3: Configuration parameters of the processor simulated

Size (KB)	history length: hash	history length: path	history length: global
1	28	10	12
2	32	14	23
4	45	24	26
8	50	27	33
16	55	41	35
32	60	55	56
64	64	61	57
128	64	64	64
256	64	64	64
512	64	64	64
1024	64	64	64

Table 4: Global history lengths for the hashed, path-based and global perceptrons





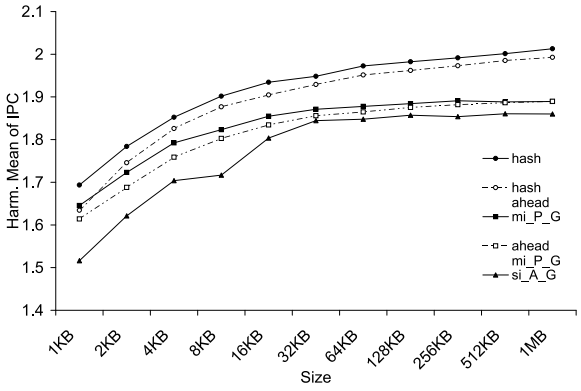
ideal hashed perceptron over an ideal path-based perceptron and 6.5% at 1MB.

The ahead pipelined hashed perceptron also outperforms the ahead pipelined path-based perceptron, with an 16.3% lower misprediction rate and a 4.6% higher IPC at 64KB.

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 22% over a path-based perceptron on 11 programs out of the SPEC2000 integer set of benchmarks, increasing IPC by over 6.5%.
- The hashed perceptron reduces the number of adders by up to a factor of eight 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 up to eight.
- The update logic is greatly simplified by only having to keep track of 8 weights instead of 40 or more 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. 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.

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] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[6] 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.

[7] 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.

[8] E. Ipek, S. A. McKee, M. Schulz, and S. Ben-David. On Accurate and Efficient Perceptron-Based Branch Prediction, 2003. Unpublished Work.

[9] D. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of The Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, 2001.

[10] 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.

[11] 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.

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

[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] D. Parikh, K. Skadron, Y. Zhang, and M. Stan. Power-aware branch prediction: Characterization and design. *IEEE Trans. Comput.*, 53(2):168–186, 2004.

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

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

[17] 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.

[18] 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/.

[19] 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.