Merging Path and Gshare Indexing in Perceptron Branch Prediction

DAVID TARJAN and KEVIN SKADRON
University of Virginia

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 the SPECint2000 set of benchmarks, the hashed perceptron predictor improves accuracy by up to 15.6% over a MAC-RHSP and 27.2% over a path-based neural predictor.

Categories and Subject Descriptors: C1.1 [**Computer Systems Organization**]: Procecessor Architectures—*Single data stream architectures*

General Terms: Performance

Additional Key Words and Phrases: Branch prediction, neural networks, two-level predictors

1. INTRODUCTION

The trend in recent high-performance commercial microprocessors has been toward ever deeper pipelines to enable ever higher clock speeds [Boggs et al. 2004; Hinton et al. 2001], with the issue width staying about the same as 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 terms of power and energy efficiency [Parikh et al. 2004].

Authors' address: D. Tarjan, Computer Science Dept., University of Virginia, 151 Engineer's Way PO Box 400740, Charlottesville, VA 22904-4740; email: {dtarjan,skadron}@cs.virginia.edu.

Most branch predictors explored in the last 10 years have been based on tables of two-bit saturating counters. The perceptron predictor is a new kind of predictor that 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 multicycle 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 [Seznec et al. 1996; Ipek et al. 2005; Jiménez 2003; Seznec and Fraboulet 2003] 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 [Jiménez 2003] 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 neural predictor 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.

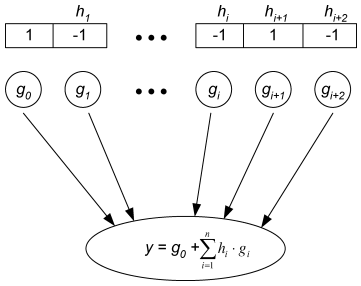

Fig. 1. The global perceptron assigns weights to each element of the branch history and makes its prediction based on the dot-product of the weights and the branch history plus a bias weight to represent the overall tendency of the branch. Note that the branch history can be global, local or something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

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 shows several small improvements to the hashed perceptron which help improve accuracy. Section 7 describes our simulation infrastructure, Section 8 compares the accuracy and performance of the different predictors. Section 9 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 [Jiménez and Lin 2001] 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 [Cormen et al. 1990], with no multiplication circuits needed.

2.2 Related Work

The idea of the neural branch prediction was originally introduced by Vintan and Iridon [1999] and Jiménez and Lin [2002] showed 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 four cycles of latency.

The path-based neural predictor [Jiménez 2003] 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 Seznec [2003, 2004a]. He introduced the concepts of redundantly representing the global history and skewing the index functions for the weights to reduce aliasing in [2003]. He introduced the MAC-RHSP (multiply-add contribution redundant-history skewed perceptron) predictor in [2004a]. He also reduces the number of adders needed by a factor of 4 (16 when using redundant history) over the normal global perceptron predictor.

In our terminology, the MAC-RHSP is similar to a global perceptron predictor that uses a concatenation of address and history information ({mi,AcG,∅}) 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 Seznec [2004a] and the MAC representation is one specific instance of an idea, which we generalize in the hashed perceptron.

The latency of the MAC-RHSP can be hidden from the rest of the pipeline by starting the prediction early and computing all possible combinations of the last four 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 two to three times fewer adders because it packs more branch history bits into fewer weights than the MAC-RHSP.

Ipek et al. [2005] 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, pseudotagging the perceptrons, and adding associativity to the weight tables.

The recently proposed piecewise linear-branch predictor [Jiménez 2004] improves on the path-based neural predictor by changing the mapping of each weight from the address of a previous branch to a hash of a previous and the current branch address. In our notation (see below), this would be a {mi, AhP, G} instead of a {mi, P, G}. In addition, the piecewise linear-branch predictor uses local and global history to make each prediction and dynamically adjusts the history length used. The predictor also uses a bias table, which is larger than the other weight tables and some other features to reduce aliasing. The {mi, AhP, G} mapping used is very similar to the {mi, PxP, ∅} introduced below, the only

Table I. Criteria in Our Taxonomy

ind: Index Type	Type of Information Used for Weight Selection **W** or Input Vector **IV**	Information Combination Method
`si` : single index `mi` : multiple indexes	**A**: branch address **G**: global branch history **P**: branch path history **L**: local branch history ∅: not used (set to constant 1)	**x**: exclusive or'ed **c**: concatenated **h**: hashed (any non-trivial hash function)

difference being that the current branch address is used and a more elaborate hash function. However, the {mi,AhP,G} mapping maintains the one-to-one mapping between branches and weights, while {mi, P, G} maps two branches to one weight.

The O-GEHL predictor [Seznec 2004b] was introduced at the same time as the piecewise linear predictor. It consists of the geometric history length (GEHL) branch predictor augmented with dynamic history-length fitting and dynamic threshold fitting. The last two features are orthogonal to the techniques presented in this work and could be applied to the hashed perceptron. The GEHL predictor is similar to the hashed perceptron in that it can use a very long history with relatively few weights. Instead of segmenting the global history into multiple parts like the hashed perceptron, it uses different hash functions to assign varying number of history bits (the history-lengths form a geometric series) to each weight.

3. THE HASHED PERCEPTRON

3.1 A Taxonomy for Perceptron Predictors

A perceptron can written as: $out = \sum_{i=0}^{n} weights[index[i], i] \cdot input[i]$

For any perceptron we can ask several questions:

1. Is the same index used for all the weights or not, i.e., does it have a single index or multiple indexes?
2. What kind of information is used as input to calculate the index/indexes? How about for the input vector? Are several kinds used?
3. If more than one kind of information is used in either index or input vector, how are they combined?

Table I lists possible answers to these questions. We can now write any perceptron indexing scheme in the form of a tuple:

{type of indexing used(**ind**),
type(s) of information used for **W**eight selection,
type(s) of information used for **I**nput **V**ector}

To clarify this taxonomy, let us look at a few examples (illustrated in Figure 2 and listed in Table II) by filling in an empty tuple {_,_,_}:

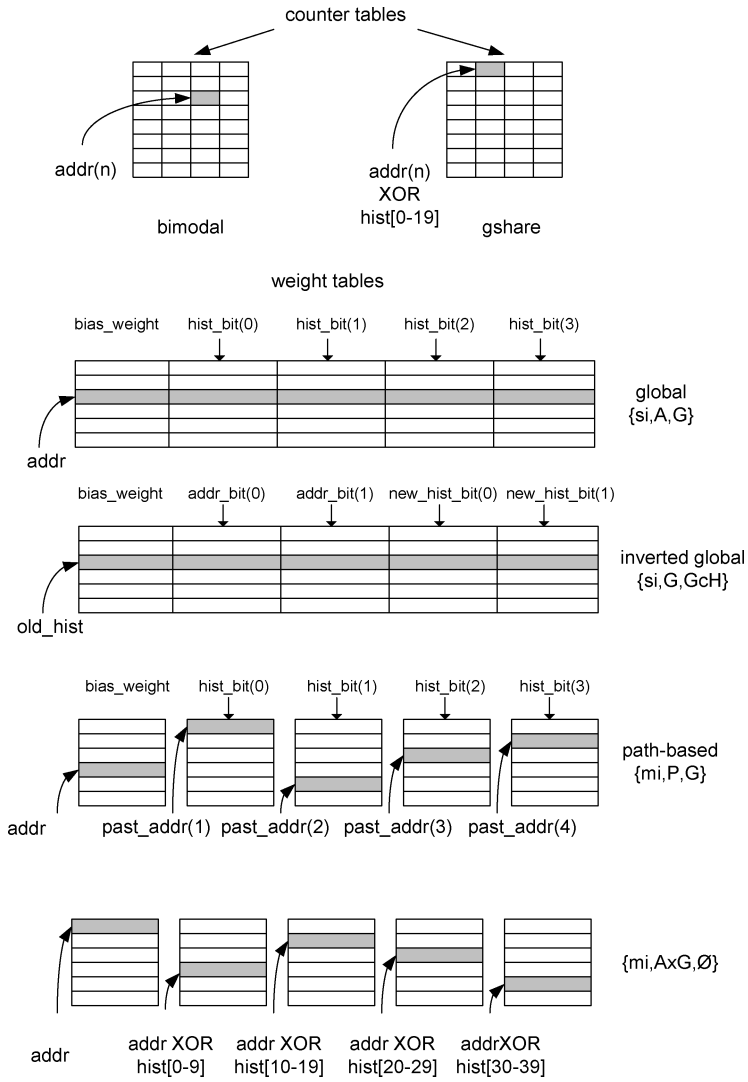

Fig. 2. When using gshare-style hashed indexing, weights are fetched by indexing into multiple tables with the exclusive OR of a branch address and a subsection of the speculative global branch history.

The global perceptron [Jiménez and Lin 2002] uses the branch address (→ {_, **A**, _}) as index for all the weights (→ { **si**, **A**, _}) and uses the global branch history as input vector (→ { **si**, **A**, **G**}).

The local/global perceptron differs from the global perceptron in that it concatenates global and local branch history (→ { **si**, **A**, **GcL**}).

The path-based neural predictor uses branch path history as index for its weights (→ {_, **P**, _}). Each weights uses the address of a different branch (→ { **mi**, **P**, _}) and the input vector again consists of the global branch history (→ { **mi**, **P**, **G**}).

Table II. Examples of Our Taxonomy

Original Mapping Name	Name in Our Taxonomy
global perceptron	{si, A, G}
local-global perceptron	{si, A, GcL}
inverted global perceptron	{si, G, GcA}
path-based neural	{mi, P, G}

3.2 Hashed Indexing

Previous perceptron predictors used each weight to measure the correlation between a single branch and the current prediction. This means that the number of tables and adders grows linearly with the amount of history used to make a prediction. It also means that the counters for each weight have to be large enough so that a single weight is able to override many other weights.

There are two main insights behind hashed indexing:

- Multiple branches can be assigned to a single weight.
- The global branch/path history can be subdivided into multiple segments to perform a series of partial pattern matches instead of a single one as in a traditional predictor.

The indexing scheme for the path-based neural predictor [Jiménez 2003] is that the index for the ith weight is the pc of the ith prior branch modulo the number of weights per table.

$$(past\_branch\_pc[0] = cur\_branch\_pc)$$
$$\{mi, P, G\}\text{: } index[i] = past\_branch\_pc[i] \bmod \#weights$$

Assume we want to assign two branches to each weight, while keeping the number of weights constant. One way is an analog of the gselect [McFarling 1993] approach, concatenating the pc of the $2i$th prior branch with the outcome of $(2i + 1)$th prior branch to form the index.

$$\{mi, PcG, \emptyset\} : index[i] = ((past\_branch\_pc[2i] \ll 1) \oplus history\_bit[2i + 1]) \bmod \#weights$$

Another approach uses the idea from Stark et al. [1998] to encode two branches in a single index by shifting the second branch by one and XORing the two pc's.

$$\{mi, PxP, \emptyset\} : index[i] = (past\_branch\_pc[2i] \oplus (past\_branch\_pc[2i + 1] \ll 1)) \bmod \#weights$$

Instead of concatenating path information with global information, we can use the idea behind the gshare predictor [McFarling 1993]. XORing a segment of the global branch history with the pc allows us to assign log(#weights) branches to a single weight. [For a given #weights per table each history_segment contains log(#weights) bits of global history.]

$$\{mi, PxG, \emptyset\} : index[i] = (history\_segment[i] \oplus past\_branch\_pc[i]) \bmod \#weights$$

Instead of using path history, we can just use the branch pc.

$$\{mi, AxG, \emptyset\} : index[i] = (history\_segment[i] \oplus cur\_branch\_pc) \bmod \#weights$$

3.3 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 the input vector is not used. This means that all the elements of the input vector are set equal to a constant 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 [Jiménez and Lin 2002]. 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.

3.3.1 *Computing Predictions and Training the Predictor.* Below we show pseudocode for predicting a branch and updating the predictor for a multiple indiexes perceptrons. Note that the mi-perceptron functions assume no input vectors.
Let

- h be the #tables
- *history_segment*[] be an array of unsigned integers representing the global branch history
- n be the number of weights per table

We show the algorithm for {mi,AxG,∅} as example of mi indexing:

```
function mi_perceptron_prediction (branch_pc, history_segment[]:custom):
   {taken, not_taken}
begin
      index[0] := branch_pc mod n
      out := W[0,index[0]]
      for j in 1 .. h do
         index[j] := (history_segment[j−1] ⊕ branch_pc) mod n
         out := out + W[j,index[j]]
      end for
      if out ≥ 0 then
         prediction := taken
      else
         prediction := not_taken
      end if
end
```

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 $\theta = \lfloor 1.93 * h + h/2 \rfloor$. If no input vector is used (such is in {mi,AxG,∅}), 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.

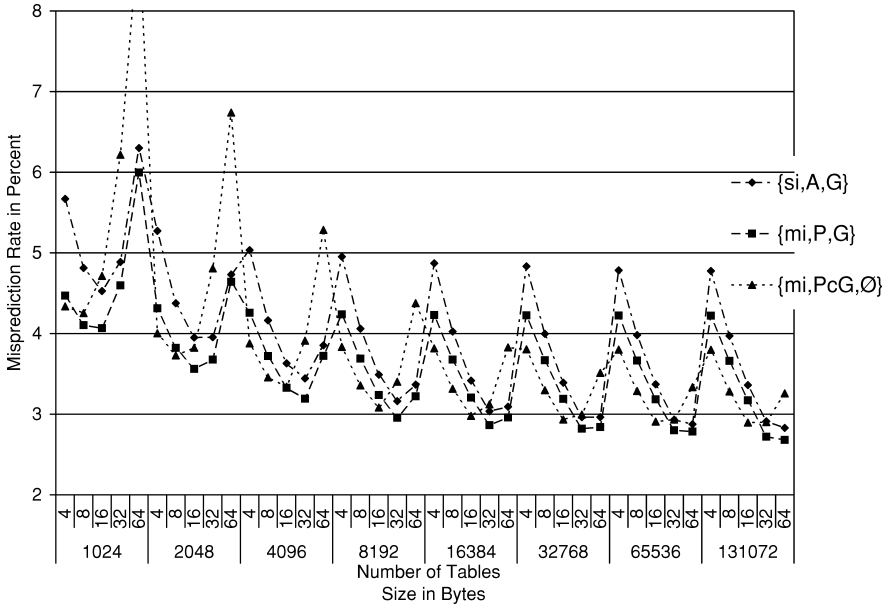

Fig. 3. The number of tables and weights is varied from 4 to 64 at each size for all predictors. All predictors improve with more weights/longer histories.

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

3.4 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,Ø} 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,Ø} needs fewer separate tables to reach a given accuracy than the other two predictors. This shows that the {mi,PcG,Ø} 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,Ø} it outperforms it at 1 and 2KB.

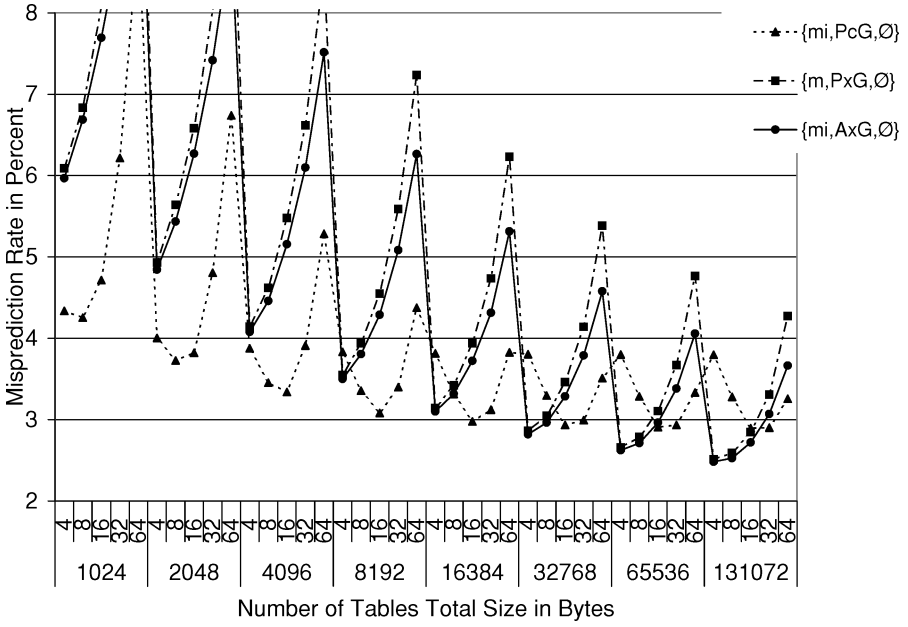

Fig. 4. The number of tables and weights is varied from 4 to 64 at each size for all predictors. {mi,AxG,Ø} and {mi,PxG,Ø} show the effects of aliasing and destructive interference at small sizes.

In Figure 4 it can be seen that at larger sizes and fewer tables the {mi,PxG,Ø} and {mi,AxG,Ø} dominate. We include the {mi,PcG,Ø} as a stand in for the {si,A,G} and {mi,P,G} predictors, since it is the best predictor of all three when using fewer tables. Interestingly, the {mi,PxG,Ø} performs worse than the {mi,AxG,Ø}. It seems that the extra path information does not give better correlation with the branch to be predicted than simply using the branch address.

Both {mi,PxG,Ø} and {mi,AxG,Ø} 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,PxG,Ø} are best at small sizes, while the {mi,AxG,Ø} is the best at larger sizes.

3.5 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,PxP,Ø} perceptron (which both belong to the class of path-based neural predictors) with

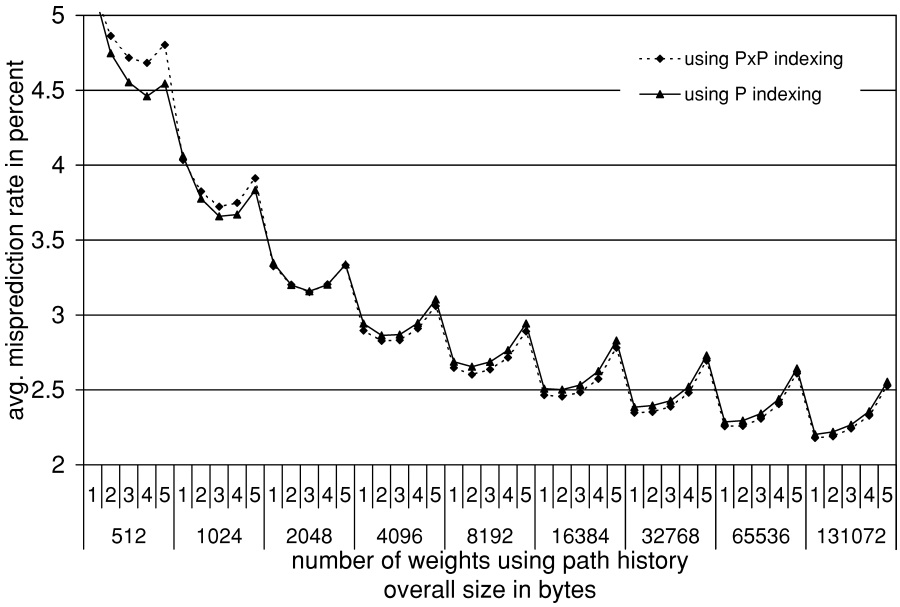

Fig. 5. Design space for trading of path history vs. branch history.

an {mi,AxG,∅} 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,AxG,∅} can use the most global history with the fewest weights and is more accurate than {mi,PxG,∅}.
- To obtain a uniform design (no input vector) for the whole predictor, we use {mi,P,∅} instead of {mi,P,G}.

3.6 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 and size of each table. A second tradeoff is between the two different mappings, that is to say, how 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,∅} or {mi,PxP,∅}. In Figure 5 we show some of the design space for all these tradeoffs.

Keeping the number of tables constant at 8, we want to find out what number of weights should use path history as opposed to global branch history at any given size. We also want to evaluate whether {mi,PxP,∅} or {mi,P,∅} is preferable.

In Figure 5 we show the results of this design space exploration.

We can see an overall improvement in accuracy with increasing hardware budgets. At small overall sizes using {mi,P,∅} gives a clear advantage, while at larger sizes {mi,PxP,∅} gives a slight advantage. The amount of path history needed decreases with increasing size. While at 512 Bytes using 4 weights with path history is best, using only 1 weight with path history is best from 32KB on upward.

Since the number of weights is kept constant at 8 in this experiment, the size of all the tables doubles as the size of the predictor doubles. Larger tables help reduce aliasing, which makes it easier for the predictor to identify different branches. Less aliasing seems to reduce the need for path history to separate different branches.

We can draw several conclusions from Figures 3, 4, and 5:

- The hashed perceptron outperforms both its component mappings at all sizes.
- The optimal number of weights with path history varies.
- Using as much global history as possible improves accuracy at larger sizes.

3.7 Advantages of a Hashed Perceptron

In total, a hashed perceptron with both {mi,AxG,∅} and {mi,P,∅} or {mi,PxP,∅} 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.
- The chance that many weights with no correlation overpower a single weight with good correlation is lessened, because the hashed perceptron predictor has fewer weights for the same history-length than a path-based({mi,P,G}) neural predictor.
- 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 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 to 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 [Jiménez et al. 2000] 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 [Calder and Grunwald 1995].

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.

If the overriding disagrees with the preliminary predictor, the front of the pipeline is usually flushed. Energy is wasted both by accessing and updating two different predictors as well as the extra work done in the front of the pipeline.

4.2 Ahead-Pipelined Predictors

A solution to this problem was introduced in Seznec et al. [1996] and expanded upon in Jiménez [2003], by "ahead" pipelining the branch prediction process.

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 that 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.

This problem was briefly mentioned in Seznec and Fraboulet [2003] 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 a pipelined perceptron, all partial sums in flight in the pipeline need to be checkpointed. Each partial sum is the sum of k w-bit weights and is thus $w + log(n)$ bits wide. (See Table III for the formulas used to determine the total amount of state to be checkpointed and Table IV for examples for deep pipelines.) 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.

Table III. Amount of State to be Checkpointed for Each Type of Predictor[a]

Predictor Type	Amount of State to be Checkpointed in Bits
path-based perceptron	$\sum_{i=2}^{x} w + \lceil \lg(i) \rceil$ bits
ahead pipelined perceptron	$(w \cdot x) + \sum_{i=2}^{x} w + \lceil \lg(i) \rceil$ bits
table-based	$2^{x-1} - 1$ bits for most significant bits

[a] x is the pipeline depth of each predictor and w is the number of bits for each weight in the perceptron predictor.

Table IV. Checkpointing Overhead as a Function of Pipeline Depth

Depth of Pipeline	Amount of State to be Checkpointed in Bits
13	133
18	195
20	221
32	377
34	405
37	447

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

5. AHEAD PIPELINING A PERCEPTRON PREDICTOR

To bring the latency of the pipelined path-based neural predictor down to a single cycle, it is necessary to decouple the table access for reading the weights from the adder. The idea, introduced in Tarjan et al. [2004], is 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,RAS 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 neural predictor. 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

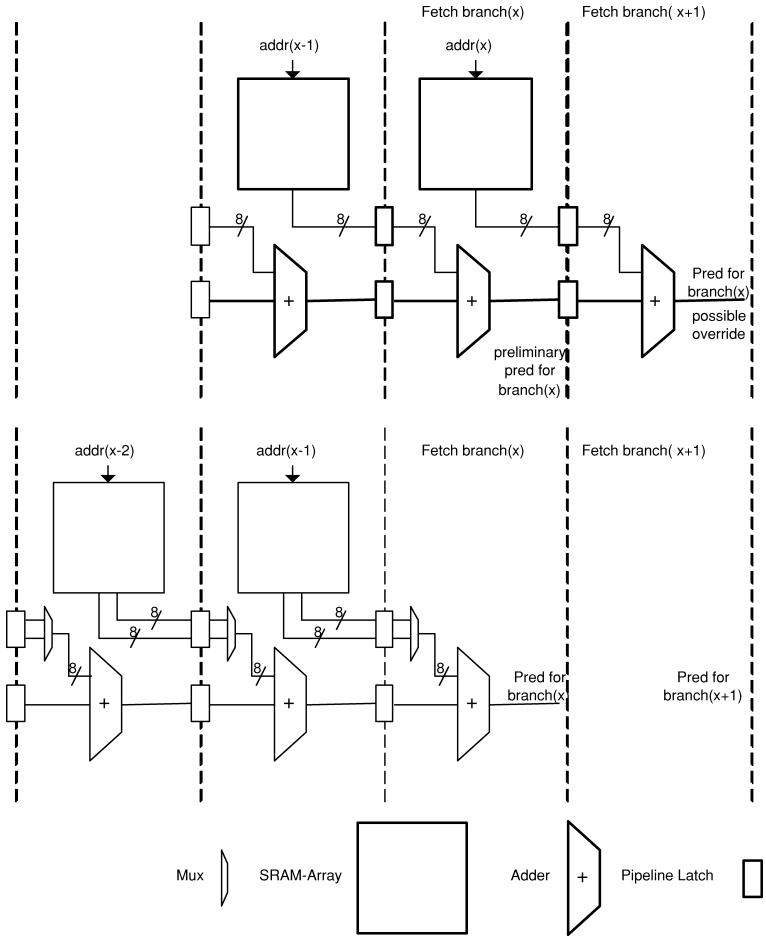

Fig. 6. (top) The original proposal for a pipelined perceptron uses the current address in each cycle to retrieve the weights for the perceptron. (bottom) Our proposed design uses addresses from the previous cycle to retrieve two weights and then chooses between the two at the beginning of the next cycle.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

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; 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 Perceptron and Path-Based Neural Predictor

To minimize the impact of ahead pipelining on the path-based neural predictor, 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,AxG,∅} have to be handled in two different ways. For weights using only older history, 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. For the weight using the most recent history, ahead pipelining similar to gshare.fast has to be used. All weights fetched using {mi,P,∅} or {mi,PxP,∅} can be fetch normally.

6. IMPROVEMENTS TO THE HASHED PERCEPTRON

The traces made available for the First Championship Branch Prediction Workshop differ markedly from those of SpecInt 2000 [Loh 2005]. To adapt the hashed perceptron to the CBP traces, we introduced several improvements:

- Sign-bit splitting: To reduce destructive aliasing between branches with opposite bias, the sign bits are stored in separate and larger tables from the rest of the weights. Each weight now has two or four sign bits assigned to it. This improvement is similar in spirit to the sharing of hysteresis bits [Seznec et al. 2003] or the agree predictor [Sprangle et al. 1997].
- Static weight boosting: Many benchmarks need only the weight from one table for an accurate prediction. The other weights have no good correlation with the branch. These tables are almost always either the bias table or the table with the most recent path or branch history. To increase the importance of these weights without increasing the size of their counters, the value of all weights from these tables is doubled when the output is computed.
- Head-splitting and tail-sharing: The basic insight, which has also been used in Loh [2004] and Jiménez [2004], is that older history is less important than more recent history or bias weights. This lead us to increase the size of the bias weight table by 1.75 and cutting in half the size of the three tables with the oldest history.
- Using {mi,AxPxPxP,∅} instead of {mi,PxP,∅}
- Smaller counters: We found that five bit counters (including the sign bit) were large enough and we did not have to use full 8-bit counters.
- Slightly biasing the predictor to not taken to reflect the distribution of branches in the traces.

6.1 Simulation Setup

We evaluate the hashed perceptron against a global perceptron, the path-based neural predictor, and the MAC-RHSP using all 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

Table V. Configuration Parameters of the Simulated Processor

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
Pipeline depth exposed by branch misprediction	33
ROB entries	512
IQ, FPQ entries LSQ entries	64 128
L2 miss latency	200 cycles

Table VI. Number of Tables for the Path-Based Neural, Hashed, and Global Perceptrons

	Number of Tables N and Latency (in Cycles) D			
Size (KB)	Hash	MAC-RHSP	Path	Global
1	N:8 D:2	N:5 D:3	N:14 D:2	N:14 D:4
2	N:8 D:2	N:7 D:3	N:18 D:2	N:18 D:5
4	N:8 D:2	N:7 D:3	N:24 D:2	N:30 D:5
8	N:8 D:2	N:8 D:3	N:30 D:2	N:48 D:5
16	N:16 D:2	N:8 D:3	N:38 D:2	N:47 D:5
32	N:16 D:2	N:8 D:3	N:44 D:2	N:47 D:6
64	N:16 D:3	N:19 D:4	N:46 D:3	N:59 D:7
128	N:16 D:3	N:24 D:4	N:48 D:3	N:59 D:8
256	N:16 D:4	N:24 D:4	N:48 D:4	N:60 D:9
512	N:16 D:5	N:28 D:6	N:48 D:5	N:67 D:10
1024	N:16 D:5	N:28 D:6	N:52 D:5	N:70 D:11

needed for the design space exploration, we used 500 million instruction traces to tune all predictors for optimal accuracy. These traces were chosen using data from the SimPoint [Sherwood et al. 2002] project. Simulations were conducted using EIO traces for the SimpleScalar simulation infrastructure [Burger et al. 1996].

For the main evaluation of all predictors and to collect performance numbers, 1 billion instruction traces chosen with SimPoint were used. We employed a greatly enhanced version of the sim-outorder simulator, called sim-modes, from the Simplescalar toolkit [Burger et al. 1996] to gather all performance data. Sim-modes models separate integer, floating point, and load/store queues, as well as a separate ROB. It also models the full depth of the pipeline, instead of simulating only a five-stage pipeline. We use the technique introduced in Skadron et al. [1998] and save the top of the return address stack with each prediction to avoid corruption of the RAS on a branch misprediction. 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 V.

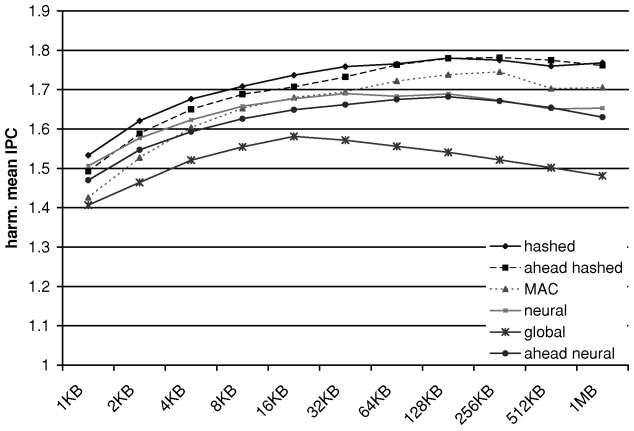

Fig. 7. Misprediction rates on the 12 SpecInt 2000 benchmarks.

The history lengths for the tuned predictors can be found in Table VI. To get the delay of each predictor, we used the delay numbers presented in Jiménez [2003] for predictors with equal sizes and comparable number of tables as those presented in Jiménez [2003]. We used CACTI 3.0 to extrapolate the delay numbers for predictors larger than 64KB. It should be noted, however, that at these sizes, factors such as transmitting the weights and partial sums between tables become very noticable. Since we took no account of these additional factors, the delay numbers used are only a first-order estimate. For all predictors with greater than one cycle latency, a 2K entry bimodal predictor is used as the initial predictor.

7. RESULTS

The average misprediction rates and the harmonic mean IPC for the path-based neural, MAC-RHSP, global, and hashed perceptrons are shown in Figures 7 and 8. Several interesting trends can be seen in Figure 7. The hashed perceptron is the most accurate predictor at all sizes. The MAC-RHSP starts out almost the worst but catches up and overtakes the path-based neural predictor at larger sizes. We theorize that because the MAC-RHSP is similar to a {mi,AcG,∅} perceptron, it needs a minimum table size so that a meaningful number of address and history bits can be used for each weight. Conversely, the path-based neural predictor is more comfortable with small hardware budgets, but tops out at a higher misprediction rate. This trends can be seen in the fact that the hashed perceptron is only 6.8% more accurate than the path-based neural predictor at 1KB, but has a 27.2% advantage at 1MB. The trend in comparison with the MAC-RHSP is the reverse, where the difference shrinks from 15.6% at 1KB to 9.5% at 1MB. The differences in accuracy are reflected in the IPC numbers in Figure 8. However, they are also attenuated by the increasing latency of the larger predictors. This leads to the fact that the best performance can found anywhere from 16 to 256KB for each predictor. At 32KB the hashed perceptron has a 3.7 and 4% advantage over the MAC-RHSP and the path-based neural predictor, respectively.

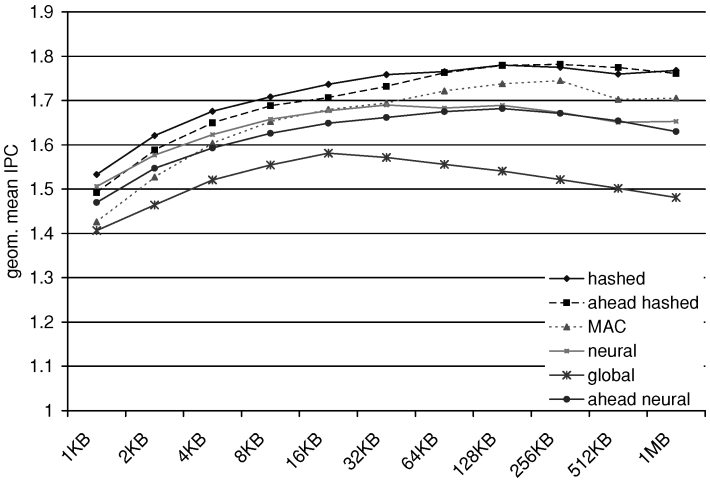

Fig. 8. Harmonic mean IPC on the 12 SpecInt 2000 benchmarks.

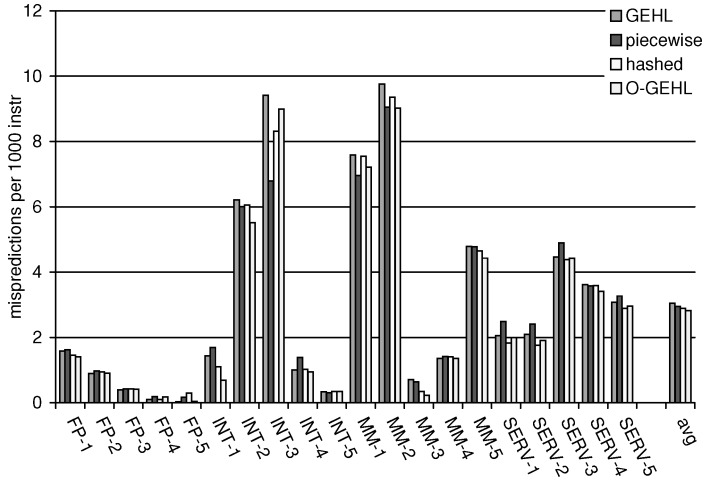

Fig. 9. Accuracy of the hashed perceptron, piecewise linear, GEHL, and O-GEHL branch predictors on the CBP traces.

7.1 Accuracy on the CBP Traces

On the CBP traces, we compare the improved hashed perceptron with the piecewise linear branch predictor and the O-GEHL predictor. All three predictors are configured to CBP specifications, using less than 64K + 256 bits. The O-GHEL and the piecewise linear predictor use the configurations which were submitted for the CBP contest. The results are shown in Figure 9. Since the O-GEHL uses dynamic history-length fitting and dynamic threshold fitting, which can alo be applied to the hashed perceptron, we also include the GEHL predictor as straight comparison with the hashed perceptron. The hashed perceptron achieves 2.89 mispredictions per 1000 instructions, which would have placed it one-third in the rankings when using the publicly distributed traces.

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 up to 15.6% over a MAC-RHSP and 27.2% over a path-based neural predictor on the SPEC2000 integer set of benchmarks, increasing IPC by up to 7.5%.
- The hashed perceptron can reduce the number of adders by almost a factor of four in comparison to the path-based neural predictor and up to a factor of six in comparison to the global perceptron.
- The amount of state that needs to be checkpointed and restored in case of a branch misprediction is reduced by almost a factor of four in comparison to the path-based neural predictor.
- 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, and offers superior accuracy, starting at low hardware budgets and scales, better than previous designs to larger configurations. It is 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.

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

BOGGS, D., BAKTHA, A., HAWKINS, J. M., MARR, D. T., MILLER, J. A., ROUSSEL, P., SINGHAL, R., TOLL, B., AND VENKATRAMAN, K. S. 2004. The Microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal 8*, 1 (Feb.).

BURGER, D., AUSTIN, T. M., AND BENNETT, S. 1996. Evaluating Future Microprocessors: The Simplescalar Tool Set. Tech. Rep. CS-TR-1996-1308, University of Wisconsin-Madison.

CALDER, B. AND GRUNWALD, D. 1995. Next cache line and set prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ACM Press, New York, 287–296.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1990. *Introduction to Algorithms*. McGraw-Hill, New York.

HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., AND ROUSSEL, P. 2001. The Microarchitecture of the Pentium 4 processor. *Intel Technology Journal 5*, 1 (Feb.), 13.

IPEK, E., MCKEE, S. A., SCHULZ, M., AND BEN-DAVID, S. 2005. Perceptron Based Branch Prediction: Performance of Some Design Options. Tech. Rep. CSL-TR-2005-1043, Cornell Computer Systems Lab. April.

JIMÉNEZ, D. A. 2003. Fast path-based neural branch prediction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 243.

JIMÉNEZ, D. 2004. Idealized piecewise linear branch prediction. In *The First Championship Branch Prediction Workshop, 2004*.

JIMÉNEZ, D. AND LIN, C. 2001. Dynamic branch prediction with perceptrons. In *Proceedings of The Seventh International Symposium on High-Performance Computer Architecture*. 197–206.

JIMÉNEZ, D. A. AND LIN, C. 2002. Neural Methods for Dynamic Branch Prediction. *ACM Trans. Comput. Syst. 20*, 4, 369–397.

JIMÉNEZ, D. A., KECKLER, S. W., AND LIN, C. 2000. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, New York, 67–76.

LOH, G. H. 2004. The Frankenpredictor: Stitiching together nasty bits of other predictors. In *The First Championship Branch Prediction Workshop, 2004*.

LOH, G. H. 2005. Simulation differences between academia and industry: A branch prediction case study. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*

MCFARLING, S. 1993. Combining Branch Predictors. Tech. Rep. TN-36, Digital Western Research Laboratory. June.

PARIKH, D., SKADRON, K., ZHANG, Y., AND STAN, M. 2004. Power-aware branch-prediction: Characterization and design. *IEEE Trans. Comput. 53*, 2, 168–186.

SEZNEC, A. 2003. Redundant history skewed perceptron predictors: Pushing limits on global history branch predictors. Tech. Rep. 1554, IRISA. Sept.

SEZNEC, A. 2004a. Revisiting the Perceptron Predictor. Tech. Rep. 1620, IRISA. May.

SEZNEC, A. 2004b. The O-GEHL Branch Predictor. In *The First Championship Branch Prediction Workshop, 2004*.

SEZNEC, A. AND FRABOULET, A. 2003. Effective Ahead Pipelining of Instruction Block Address Generation. In *Proceedings of the 30th Annual International Symposium on Computer architecture*. ACM Press, New York, 241–252.

SEZNEC, A., JOURDAN, S., SAINRAT, P., AND MICHAUD, P. 1996. Multiple-block ahead branch predictors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VII). Cambridge, Massachusetts, United States, ACM Press, New York, NY, 116–127. Available at http://doi.acm.org/10.1145/237090.237169.

SEZNEC, A., FELIX, S., KRISHNAN, V., AND SAZEIDES, Y. 2003. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. 295–306.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically Characterizing Large Scale Program Behavior. *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. http://www.cs.ucsd.edu/users/calder/simpoint/.

SKADRON, K., AHUJA, P. S., MARTONOSI, M., AND CLARK, D. W. 1998. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *International Symposium on Microarchitecture*. 259–271.

SPRANGLE, E., CHAPPELL, R. S., ALSUP, M., AND PATT, Y. N. 1997. The agree predictor: A mechanism for reducing negative branch history interference. In *ISCA*. 284–291.

STARK, J., EVERS, M., AND PATT, Y. N. 1998. Variable length path branch-prediction. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, 170–179.

TARJAN, D., SKADRON, K., AND STAN, M. R. 2004. An ahead pipelined alloyed perceptron with single cycle access time. In *The 5th Workshop on Complexity-Effective Design, 2004*.

VINTAN, L. AND IRIDON, M. 1999. Towards a High Performance Neural Branch Predictor. In *Proceedings of the 9th International Joint Conference on Neural Networks*. 868–873.

Received October 2004; revised April 2005 and July 2005; accepted July 2005