Perceptron-Based Branch Confidence Estimation

Haitham Akkary[1,2], Srikanth T. Srinivasan[2], Rajendar Koltur[1], Yogesh Patil[1] and Wael Refaai[1]

[1]*Electrical and Computer Engineering*
Portland State University
{akkary,yogeshp,rkoltur,refaai@ece.pdx.edu}

[2]*Microarchitecture Research Lab*
Intel Corporation
srikanth.t.srinivasan@intel.com

Abstract

Pipeline gating has been proposed for reducing wasted speculative execution due to branch mispredictions. As processors become deeper or wider, pipeline gating becomes more important because the amount of wasted speculative execution increases. The quality of pipeline gating relies heavily on the branch confidence estimator used. Not much work has been done on branch confidence estimators since the initial work [6]. We show the accuracy and coverage characteristics of the initial proposals do not sufficiently reduce mis-speculative execution on future deep pipeline processors.

In this paper, we present a new, perceptron-based, branch confidence estimator, which is twice as accurate as the current best-known method and achieves reasonable mispredicted branch coverage. Further, the output of our predictor is multi-valued, which enables us to classify branches further as "strongly low confident" and "weakly low confident". We reverse the predictions of "strongly low confident" branches and apply pipeline gating to the "weakly low confident" branches. This combination of pipeline gating and branch reversal provides a spectrum of interesting design options ranging from significantly reducing total execution for only a small performance loss, to lower but still significant reductions in total execution, without any performance loss.

1. Introduction

Speculative execution past unresolved branches by means of branch prediction is key to high performance in modern processors. However, branch mispredictions adversely affect both processor performance and power. The negative impact on performance is because: 1) the mis-speculative execution consumes resources that could have been allocated to useful work, such as another thread on a multithreaded processor [9], and 2) execution stalls occur when the pipeline is re-filled with instructions on the correct path. The negative impact on power is due to the mis-speculative execution causing a lot more instructions to be executed than necessary. This paper aims to reduce the negative impact of branch mispredictions on both processor power and performance, by reducing the amount of wasted mis-speculative execution.

Pipeline gating [10] using confidence estimation of conditional branches has been proposed for reducing the wasted mis-speculative execution resulting from branch mispredictions. In this method, fetch is stalled when some selected number of low confidence branches (e.g. 2 branches) have been fetched but have not been yet resolved. When the confidence estimation is correct (i.e. a mispredicted branch is correctly identified as having low confidence), wastage due to the speculative execution is reduced. However, when the confidence estimation is incorrect (i.e. a correctly predicted branch is identified as having low confidence), the pipeline is stalled unnecessarily resulting in performance loss.

Current processor trends – deeper pipelines [14] and/or larger instruction windows [1] – result in a significant increase in wasted mis-speculative execution. We observe an increase in wasted speculative execution due to branch mispredictions, from 24% on a 20-cycle 4-wide pipeline to almost 50% on 20-cycle 8-wide and 40-cycle 4-wide pipelines. Further, deep pipelines make pipeline gating more challenging by increasing the number of cycles needed after the fetch pipeline stage, for mispredicted branches to be resolved, and the confidence estimation to be validated. Unless the confidence estimation method is highly accurate, significant performance loss could result from pipeline gating on deep pipelines. In this paper, we show the accuracy and mispredicted branch coverage characteristics of existing branch confidence estimators are ineffective in dealing with the increasingly challenging speculation waste problem on deeply pipelined processors.

We present a new perceptron-based branch confidence estimator, which is twice as accurate as the current best-known method and achieves reasonable mispredicted branch coverage. We train the perceptron, which is a model of an artificial neuron [3][12], using correct/incorrect branch prediction information. We show for effective branch confidence estimation, it is necessary to train the perceptron using correct/incorrect prediction

instead of taken/not-taken information as suggested by Jimenez and Lin [7].

The multi-valued (non-binary) output provided by our perceptron branch confidence estimator enables further classifying low-confidence branches into two categories: strongly low confident and weakly low confident. This sub-classification allows us to reverse the directions [2][8] of strongly low confident branches and apply pipeline gating for weakly low confident branches. Using this combination of branch reversal and pipeline gating, we show it is possible to improve branch prediction as well as achieve effective pipeline gating, with a single hardware structure.

The main contribution of this paper is our perceptron training scheme that allows a perceptron-based branch confidence estimator to be used both for pipeline gating and branch reversal. This provides a spectrum of interesting design options ranging from significantly reducing total execution for only a small performance loss, to lower but still significant reductions in total execution, without any performance loss.

The paper is organized as follows. Section 2 presents background and related work on pipeline gating and current branch confidence estimation schemes. Section 3 describes perceptron-based branch confidence estimation and our scheme for training the perceptrons. Section 4 outlines our simulation methodology. Section 5 provides experimental results, and Section 6 has concluding remarks.

2. Background and previous work

In this Section, we provide some background on pipeline gating and branch confidence estimation, and discuss prior work.

2.1. Pipeline gating

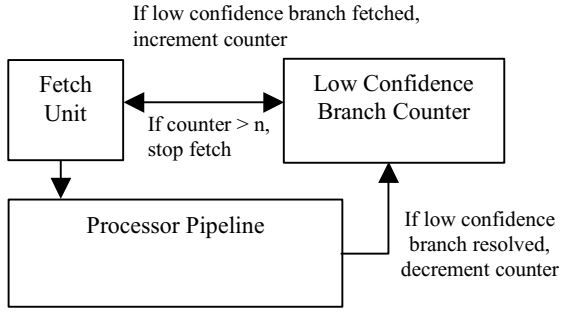

Figure 1. Pipeline gating mechanism

Pipeline gating refers to halting the fetch of new instructions, until a likely mispredicted branch is resolved. Figure 1 illustrates the method of pipeline gating using branch confidence estimation [10]. A branch confidence estimator inside the fetch unit determines whether a branch is likely to be correctly predicted (high confidence), or mispredicted (low confidence). A low confidence branch counter is used to count the number of unresolved low confidence branches in the pipeline. When the count is higher than a specific threshold, instructions fetched subsequently are considered to be most likely on the wrong-path. The fetch unit is gated/stopped at this point to prevent any further wasteful fetch and execution.

When a mispredicted branch is correctly identified as having low confidence, wastage due to the speculative execution is reduced. However, when a correctly predicted branch is identified as having low confidence, the pipeline is stalled unnecessarily, resulting in performance loss. Thus, the effectiveness of pipeline gating is highly dependent on the branch confidence estimator.

2.2. Branch confidence estimation metrics

We use two primary metrics, Specificity and Predictive value of a negative test [4], to understand the effectiveness of branch confidence estimators. They are defined as follows:

- *Specificity* (Spec) is the percentage of all mispredicted branches classified as having low confidence.
- *Predictive value of a negative test* (PVN) is the probability that a low confidence prediction is correct.

Spec is a measure of mispredicted branch coverage while PVN is a measure of accuracy. A higher value of Spec indicates more mispredicted branches are identified as low-confidence branches, while a higher value of PVN indicates the branch confidence estimator is more accurate in classifying mispredicted branches as having low confidence and correctly predicted branches as having high confidence. We refer to Spec and PVN as coverage and accuracy respectively in the paper.

In the context of pipeline gating, a higher Spec value means fetch and execution is stopped for more mispredicted branches, leading to a higher reduction in wasted speculative execution. A higher value of PVN means fetch and execution is gated less often for correctly predicted branches and hence, performance loss from unnecessary pipeline gating is less. Therefore, higher values for Spec and PVN are desirable for a branch confidence estimator to be used for pipeline gating.

2.3. Prior branch confidence estimation work

The first proposal for assigning confidence to conditional branch predictions [6], which we call the JRS estimator (after the authors' initials), uses a miss distance

saturating-counter (MDC) table in addition to the branch predictor. The table is indexed using an XOR of the branch address and the global history of previous branches, similar to the indexing method in gshare predictors [11]. The indexed counter is incremented when the branch is correctly predicted and is reset to 0 when the branch is mispredicted. A branch is estimated to have high confidence when its counter is at or above a specific threshold (λ), and is estimated to have low confidence when it is below the threshold. For example, with a 3-bit saturating counter and a threshold value of 7, a branch has to be predicted correctly seven consecutive times before it is identified as a high confidence branch.

Grunwald et al. propose an enhanced JRS confidence estimator [4]. In this method, rather than just using the previous branch history to index the MDC table, the branch is predicted first and the prediction is included along with the previous branch history in the index to the MDC table. The enhanced JRS confidence estimator is shown to perform better than the original JRS proposal.

Tyson et al. propose a pattern history based confidence estimator [15]. It uses a PAs predictor and assigns high confidence for a fixed set of selected patterns (such as, always taken, always not-taken or almost always taken), and low confidence for all other patterns. This method was compared to the enhanced JRS confidence estimator [4] and shown to be less accurate.

Smith proposed using the saturating counters of a branch predictor itself for confidence estimation [13]. When the state of the branch predictor's saturating counter is not close to or at the maximum or minimum values, the predicted branch is estimated to have low confidence. As an example, with a 2 bit saturating counter, states 1 and 2 could be classified low confidence while 0 and 3 could be classified high confidence. A comparison presented by Grunwald et al. [4] shows this method does not perform as well as enhanced JRS.

Thus, enhanced JRS is the best branch confidence estimator presented so far. In this paper we compare our perceptron confidence estimator with an enhanced JRS predictor.

3. Perceptron based branch confidence estimation

The perceptron is a model of an artificial neuron that learns how to map input vectors to different classes, using a linear function of the input vector [12]. Figure 2 shows a model of a single perceptron. A vector of weights (w[0], w[1],….,w[n]) represents the perceptron. The weights are small, signed integers. One of the inputs is always set to 1 and is called a bias input. The other inputs (x[1],x[2],…,x[n]) can be either 1 or –1. The output of the perceptron (y) is the dot product of the weights and the inputs. Sometimes a threshold is used to convert the output into a binary 0 or 1 value.

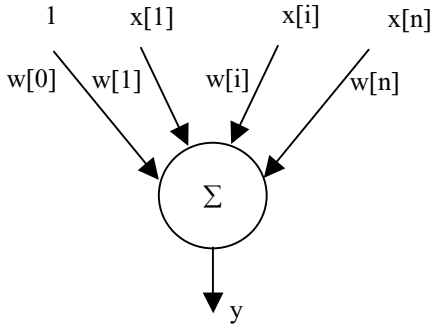

Figure 2. Perceptron model

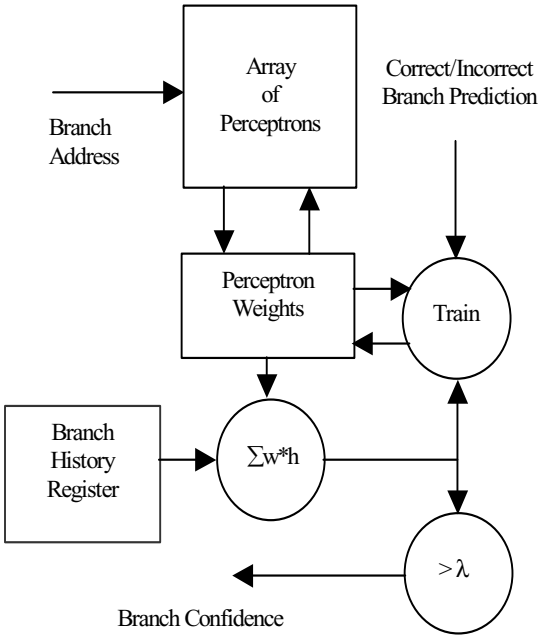

Figure 3. Perceptron confidence estimator

Figure 3 shows one implementation of a perceptron based confidence estimator. We use the simplest perceptron architecture, a single layer of perceptrons. There is an array of perceptrons indexed using a conditional branch address, just like in a regular branch predictor. The input vector to the selected perceptron in our method is the global branch history register in which a taken branch is represented by a 1, and a not-taken branch is represented by a -1. The output of the perceptron obtained by a dot product of the input vector and the weight vector stored in the perceptron, is multi-valued (not

just a 0 or 1). This perceptron output is used to classify the conditional branch corresponding to the perceptron, as having low or high confidence. When the output is larger than a specific threshold λ (typically set to 0), the perceptron predicts the execution is likely on the wrong path (low confidence). When the output is smaller than the threshold, the perceptron predicts the execution is likely on the correct path (high confidence).

Training the perceptrons. The earliest time a perceptron corresponding to a conditional branch can be trained is when it is determined with certainty the branch is on the correct path or on the wrong path. A branch is determined to be certainly on the correct path when the branch and all previous branches have been resolved. We therefore train the perceptron non-speculatively, at retirement, to allow all previous branches to complete and be resolved. Thus, low-confidence prediction happens in the front-end of the machine and updating/training the perceptrons happens in the back-end of the machine, just like for a traditional branch predictor.

Training a perceptron involves updating the weight vector stored in that perceptron. When a branch instruction gets retired, let p denote its outcome (p is 1 if the branch is incorrectly predicted and –1 if the branch is correctly predicted). Let c denote the confidence assigned to the branch in the front-end (c is 1 for branches predicted low-confidence and –1 for branches predicted high-confidence). For some training threshold T (a parameter used to determine how long a perceptron needs to be trained), the weights of the selected perceptron are adjusted as follows:

```
if (sign(c) ≠ sign(p) || abs(y) <= T) {
    for (i = 0; i <= n; i++) {
        w[i] += p*x[i];
    }
    y++;
}
```

Since p*x[i] can be either 1 or –1, the weight is incremented (when p and x[i] are the same) or decremented (when p and x[i] are different). Thus, the perceptron is trained using current branch's outcome (correct/incorrect information), similar to the JRS confidence estimator and unlike branch predictors. When there is a frequent correlation between a previous branch direction (taken/not-taken information) and the current branch outcome, the weight corresponding to the previous branch is trained to become either an increasingly large positive or an increasingly small negative number. When there is only a weak or no correlation, the weight stays close to zero.

Perceptrons have been proposed earlier for branch prediction [7][16], where they are trained using the current branch's direction (taken/not-taken information). In the same work, the authors suggest a perceptron branch predictor could assign a confidence level to its prediction since the perceptron output is multi-valued and not just a Boolean value. The distance of the perceptron output value from 0 gives an indication of the level of certainty that the branch is correctly predicted. In this paper, we evaluate this method and show it does not achieve the high classification accuracy that a perceptron predictor trained with correct/incorrect information achieves.

4. Simulation methodology

We use a cycle-accurate IA32, micro-operation (uop) based, execution driven simulator. The simulator executes "long instruction traces" (LIT). A LIT is a snap shot of the architectural state including memory. It contains system interrupt injections that are needed to simulate system events such as DMA. The LIT allows us to execute user and kernel instructions and model speculative effects.

Our simulations are based on out-of-order superscalar microarchitecture similar to that of the Intel Pentium® 4 processor [5]. The simulator includes a detailed memory subsystem that fully models buses and bus contention.

Table 1. Baseline model parameters

Fetch/Issue/Retire width	4
ROB size	128
Load and store buffers	48 loads, 32 stores
Memory disambiguation	Perfect
Scheduling window size	48 int, 24 mem and 56 fp
Execution units	3 int, 2 mem, 1 fp
Hardware data prefetch	Stream-based, 16 streams
Branch predictor	Combined: 16K bimodal, 64K gshare, 64K Meta
Trace cache	12K uops, 8-way
L1 DCache	32K, 8-way, 64-byte line
L2 unified cache	1M, 8-way, 64-byte line

Our baseline processor uses an aggressive 40-cycle pipeline. Table 1 gives the parameters for our baseline processor. The branch predictor we use in our baseline model is a big bimodal-gshare hybrid predictor. Most of our results are reported for the baseline configuration, but in Section 5.5, we evaluate the perceptron branch confidence estimator on another futuristic processor configuration, a 8-wide 20-cycle machine.

The JRS confidence estimator used in all experiments has a 8k-entry table of 4-bit resetting counters, and the perceptron estimator, unless stated otherwise, has 8-bit weights, 32-bit global branch history register, and 128 entries in the perceptron array. Thus, the two estimators have storage arrays of equal size, each totaling 4KB.

Table 2. Benchmarks and their speculative execution characteristics

	Branch mispredicts / 1000 uops	% increase in uops executed due to branch mispredictions		
		20-cycle 4-wide pipeline	20-cycle 8-wide pipeline	40-cycle 4-wide pipeline
gzip	5.2	30	66	61
vpr	6.6	35	75	78
gcc	2.3	11	19	24
mcf	16	110	225	226
crafty	3.4	13	38	31
link	4.6	28	60	65
eon	0.5	2	4	6
perlbmk	0.7	3	7	7
gap	1.7	9	16	19
vortex	0.2	1	2	2
bzip	1.1	5	14	13
twolf	6.3	30	49	64
average	4.1	24	48	50

In this paper we use the SpecInt 2000 benchmarks. The benchmarks are compiled using Intel IA32 compiler. We execute two traces from each benchmark, each trace containing 30 million IA32 instructions of which 10 million instructions are used to warm up the caches and predictors. Each 30 million-instruction trace is extracted from the corresponding benchmark after the initial startup phase and is carefully selected to well represent the overall characteristics of the benchmark.

Table 2 lists the benchmarks, their average number of branch mispredictions per thousand uops, and the increase in total uops executed due to branch mispredictions on a 20-stage 4-wide, 20-stage 8-wide and 40-stage 4-wide pipelines.

From Table 2 we see there is a significant increase in mis-speculative execution due to branch mispredictions on deep or wide pipelines. This mis-speculative execution gets mostly wasted[1] and can become a severe limitation on futuristic processors unless effective methods are identified to reduce it.

5. Experimental Results

In this Section, we present simulation results. In Section 5.1, we compare the perceptron and JRS confidence estimators in terms of their accuracy and coverage and discuss pipeline gating using both estimators. In Section 5.2, we evaluate pipeline gating using our perceptron-based confidence estimator on more aggressive baseline branch predictors. In Section 5.3, we compare the effectiveness of training the perceptron with branch directions (taken/not-taken information), the method proposed in [7], as opposed to our training method using branch outcomes (correct/incorrect information). In Section 5.4, we discuss important perceptron design parameters with respect to branch confidence estimation. Finally, in Section 5.5, we present results for applying branch reversal and pipeline gating simultaneously using our perceptron branch confidence estimator.

5.1. Enhanced JRS vs. Perceptron branch confidence estimators

Table 3 shows the PVN and Spec values for various thresholds of the JRS and perceptron branch confidence estimators. The JRS resetting counter thresholds used are 3, 7, 11 and 15 and the perceptron thresholds used are –50, -25, 0 and 25.

Table 3. Enhanced JRS vs. Perceptron (Confidence estimation metrics)

λ = Branch confidence estimator threshold

Enhanced JRS			Perceptron		
λ	PVN %	Spec %	λ	PVN %	Spec %
3	36	85	25	77	34
7	28	92	0	74	43
11	24	94	-25	69	54
15	22	96	-50	61	66

From Table 3, the different nature of the two estimators can be seen clearly. JRS has lower accuracy (PVN) and higher coverage (Spec) than the perceptron. When the JRS estimator is used for pipeline gating, its higher coverage is expected to stall fetch for a higher fraction of mispredicted branches, resulting in larger reductions in wasted speculative execution. However, the lower accuracy of the enhanced JRS estimator could lead to fetch stalls more frequently on correctly predicted branches that are classified as low confidence, which could result in a higher performance loss. The perceptron estimator has a significantly higher accuracy, more than twice that of JRS, which is a very useful property for pipeline gating on deep pipelines.

Table 4 shows the average reduction in the total number of micro-ops (uops) executed across all benchmarks and the performance loss from pipeline gating with JRS and perceptron estimators for various threshold values. The pipeline length used is 40 cycles. The low confidence branch counter threshold which is the number of low confidence branches needed to stall fetch (see Figure 1) is 1 in case of the perceptron, and 1, 2 or 3 for JRS. Since the JRS estimator has significantly lower PVN

[1] We say "mostly wasted" because there could be some prefetch benefits.

value, the higher branch counter values are needed to reduce the probability of false negatives and the resulting unnecessary pipeline stalls.

Table 4. Enhanced JRS vs. Perceptron (Pipeline gating metrics)

U = Reduction in total uops executed (in %)
P = Performance Loss (in %)
λ = Branch confidence estimator threshold
PLn (n = 1,2,3) = Branch counter threshold

JRS							Perceptron		
λ	PL1		PL2		PL3		λ	PL1	
	U	P	U	P	U	P		U	P
3	26	17	14	4	9	2	25	8	0
7	29	25	19	9	13	4	0	11	1
11	31	29	21	12	14	5	-25	14	2
15	31	32	22	14	15	7	-50	18	3

From Table 4 we see a significant reduction in total execution can be achieved with the perceptron estimator, with very little or no loss in performance. Using a perceptron threshold value of 25, it is possible to reduce the total number of uops executed by 8% with no loss in average performance. With the JRS estimator, we are unable to achieve any significant reduction in uops executed for no performance loss.

Several data points in Table 4 show the perceptron estimator is better suited for pipeline gating than the JRS estimator. For example, comparing a perceptron estimator with a threshold of –50 to a JRS estimator with a threshold of 7 and branch counter value of 2, the reduction in uops executed is almost equivalent, 18% and 19% respectively. However, the performance loss is 9% for JRS compared to only 3% for the perceptron. Alternatively, comparing the perceptron estimator with a threshold of –25 and the JRS estimator with a threshold of 3 and PL3, for a fixed amount of performance loss (2% in both cases), the reduction in uops executed is more for the perceptron estimator (14%) than the JRS estimator (9%).

Thus, using the perceptron branch confidence estimator, a spectrum of pipeline gating options are possible, ranging from modest reductions in wasted speculative execution for no performance loss, to significantly higher reductions in wasted speculative execution for a small increase in lost performance.

5.2. Effect of baseline branch predictor on perceptron estimator

To understand the usefulness of the perceptron estimator, as branch prediction gets better, we perform similar pipeline gating experiments using a gshare-perceptron hybrid predictor [7] as the baseline processor's branch predictor. For this baseline branch predictor, the perceptron component is trained using taken/not-taken information. The average number of branch mispredictions per 1000 uops decreases from 4.1 for the bimodal-gshare hybrid predictor to 3.6 when the gshare-perceptron hybrid predictor is used as the baseline branch predictor.

Table 5. Effect of better baseline branch predictor

bimodal-gshare hybrid			gshare-perceptron hybrid		
λ	U	P	λ	U	P
25	8	0	0	4	0
0	11	1	-25	8	1
-25	14	2	-50	12	2
-50	18	3	-60	14	3

Table 5 compares pipeline gating results using both the bimodal-gshare and gshare-perceptron baseline branch predictors. The perceptron thresholds for the branch confidence estimator in both sets of simulations are chosen to achieve 0% - 3% performance loss with pipeline gating. For the same performance loss, the percentage reduction in total execution decreases, as the baseline branch predictor gets better. With the better baseline branch predictor, the accuracy of the branch confidence estimator decreases. This suggests predicting low confidence branches gets tougher as the baseline branch predictor accuracy increases. However, our perceptron branch confidence estimator produces significant reductions in total execution for a small performance loss, even with a better baseline branch predictor.

5.3. Perceptron training comparison

An important component of building a perceptron branch confidence estimator is the training process. Jimenez et al. [7] suggest a perceptron branch predictor can be used for branch confidence estimation, based on the perceptron output's proximity to the value 0. Such a confidence estimator is trained using a branch's direction (whether a branch is taken/not-taken) and is labeled perceptron_tnt. The reasoning here is if the perceptron output is a large positive or a large negative number, the corresponding branch is predicted to be either "strongly taken" or "strongly not-taken", respectively. Hence, closer the perceptron output is to the value 0, lower the confidence on the branch.

In this paper, we use a branch's outcome (whether a branch is correctly/incorrectly predicted) for training its corresponding perceptron. This training scheme is labeled

perceptron_cic. Here, if the perceptron output for a branch is above the perceptron threshold, it is likely to be incorrectly predicted (low confidence) and if the perceptron output is below the threshold, the branch is likely to be correctly predicted (high confidence). Training the perceptrons using correct/incorrect information lets us classify branches further as "strongly low confident", if the perceptron output is well above the threshold, and "weakly low confident", if the perceptron output is closer to or just above the threshold.

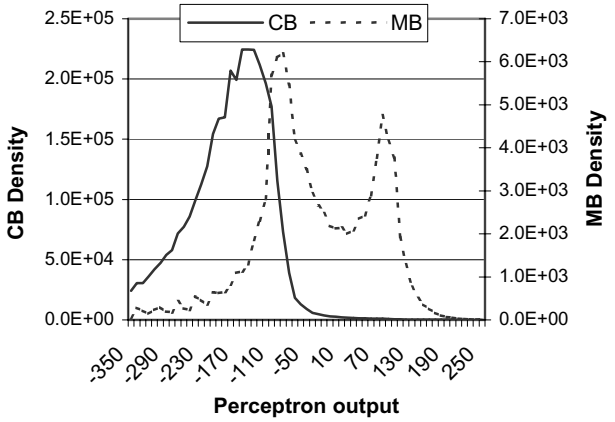

Figure 4. Perceptron output density function for perceptron_cic

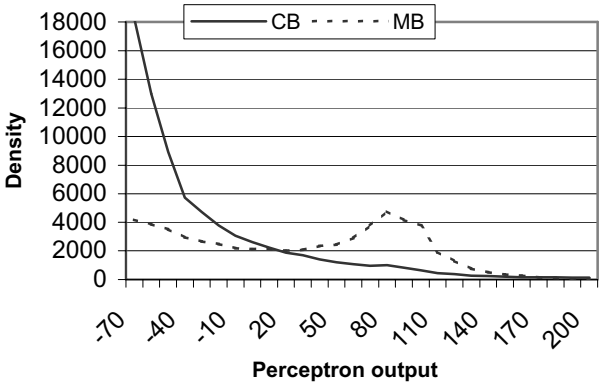

Figure 5. Zoom-in on range [-70, 200] of perceptron output density function for perceptron_cic

In this Section, we compare these two perceptron training schemes. Our experiments indicate perceptron_cic is more accurate than perceptron_tnt and hence is better suited for pipeline gating. Furthermore, sub-classifying low-confidence branches as "strongly low confident" and "weakly low confident" using perceptron_cic allows branch reversal to be applied to the strongly low confident branches and pipeline gating to be applied to the weakly low confident branches. This gives perceptron_cic a clear edge over other branch confidence estimation schemes.

To understand why perceptron_cic gives higher accuracy than perceptron_tnt, we examine the density function of the perceptron output for both training schemes. We use *gcc* benchmark from spec2000 as an example. The other benchmarks also show similar behavior. Figure 4 shows the density function of the perceptron output value for perceptron_cic for both correctly predicted branches (CB) and mispredicted branches (MB). Since the number of mispredicted branches is significantly less than the correctly predicted branches, different scales are used to display the two density functions.

The perceptron output for most correctly predicted branches is clustered around a negative number of about -130. The perceptron output for mispredicted branches is clustered along the x-axis further to the right, with a higher peak around a value of -90, and a lower peak around the value 100.

Figure 5 shows the CB and MB density functions using the same y-axis scale but only for perceptron output values in a narrower range between –70 and 200. Three distinct output regions can be identified:

1. When the perceptron output is larger than 30, more branches are mispredicted than correctly predicted (the MB curve is higher than the CB curve). This suggests that for positive perceptron outputs above a certain threshold, reversing the outcome of the branch [2][8] will correct the branch prediction most of the time, therefore gaining performance as well as reducing waste.
2. When the perceptron output is between 30 and –30, more branches are correctly predicted than mispredicted. However, for this output range, the ratio of mispredicted to correctly predicted branches is large enough to benefit from pipeline gating.
3. When the perceptron output is less than –30 the vast majority of branches are correctly predicted. This range may be unsuitable for both pipeline gating and branch reversal due to too many incorrect low-confidence predictions.

This example shows a perceptron trained with correct/incorrect branch information can be used both for pipeline gating and branch reversal.

Figure 6 shows similar output density functions for a perceptron trained with taken/not-taken branch outcomes (perceptron_tnt), for correctly predicted branches (CB) and mispredicted branches (MB). Figure 7 shows the same output density functions over the range –50 to 50. There are many more correctly predicted branches than mispredicted branches, for all output values, including those close to 0. Consequently, there is no output range that can be used to provide good coverage and a highly accurate classification of branches. For the same Spec

numbers (coverage), the PVN values of the perceptron_tnt estimator are significantly less than the PVN values of perceptron_cinc and the JRS estimators. Therefore, taken/not-taken training of the perceptron is not as effective as correct/in-correct branch outcomes training.

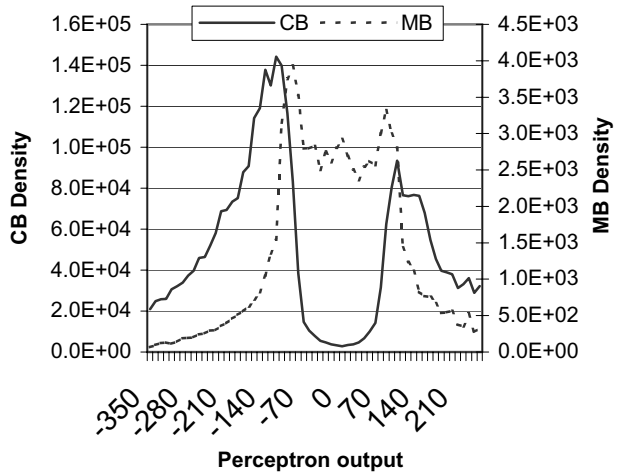

Figure 6. perceptron output density function for perceptron_tnt

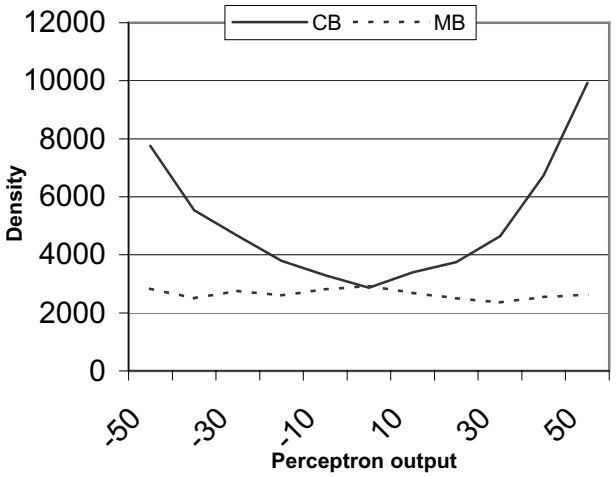

Figure 7. Zoom-in on range [–50, 50] of Perceptron output density function for perceptron_tnt

5.4. Perceptron design parameters

Next we study the sensitivity of the perceptron branch confidence estimator to the perceptron design parameters, array size and latency.

Table 6. Perceptron size sensitivity.

U = Reduction in total uops executed (in %)
P = Performance Loss (in %)
Configuration labeled PiWjHk, is interpreted as
i – number of perceptron entries,
j – the number of bits per weight and
k – the history length.

Size	Configuration	P	U
4 KB	P128W8H32	1	11
3 KB	P96W8H32	1	11
	P128W6H32	2	10
	P128W8H24	1	10
2 KB	P64W8H32	1	10
	P128W4H32	6	8
	P128W8H16	1	8

5.4.1. Effect of perceptron array size. The perceptron array size is the product of the number of entries, the number of bits per weight, and the number of weights (which is equal to the length of the branch history register or BHR). So far in our experiments we have used a perceptron estimator with 128 entries, 8-bit weights and 32-bit BHR resulting in 4KB perceptron array.

Table 6 shows the performance loss and the reduction in executed instructions for 4KB, 3KB and 2KB perceptron array configurations. The results shown are for pipeline gating with a low confidence branch counter value of 1 on our baseline 40-cycle pipeline. Depending on the configuration, the decrease in the size of the perceptron array from 4KB to 3KB to 2KB is obtained by different combinations of reducing the number of perceptrons, the number of bits per weight, and the length of the branch history register. The different combinations are shown as multiple entries for 3KB and 2KB perceptron sizes in Table 6.

The most negative impact on the performance of the perceptron confidence estimator comes from reducing the number of bits per weight. The performance loss is particularly high, 6%, when the array size is reduced from 4KB to 2KB by decreasing the number of bits per weight from 8 to 4 (compare P128W8H32 with P128W4H32).

Reducing the number of history bits has little impact on performance but has a large impact on the reduction in total uops executed. The average reduction in total uops executed drops from 11% to 8% when the perceptron size is reduced from 4KB to 2KB by decreasing the length of the branch history register from 32 to 16 (compare P128W8H32 with P128W8H16).

Reducing the number of entries in the perceptron array has the least impact on performance and the reduction in total uops executed. Decreasing the number of entries by half, from 128 to 64, to reduce the array size from 4KB to 2KB (compare P128W8H32 with P64W8H32) has very little impact on performance and only decreases the

average reduction in total uops executed from 11% to 10%.

5.4.2. Effect of perceptron latency. One disadvantage of the perceptron is the additional delay required for computing the output of the perceptron after reading the array. Since the inputs to the perceptron are either 1 or –1, a multiplication is not needed to compute the dot product [7]. Instead the weight is simply added to the total sum of the dot product when the input is 1, or subtracted when the input is -1. Still, there is significant delay to complete the total sum of the dot product.

For a history length of 32, we estimate 9 cycles are needed to complete the perceptron output computation on a 40-cycle pipeline implemented on a 0.09um process technology. Assuming the perceptron estimator is pipelined to allow a prediction throughput of one branch every cycle, we compare a 9-cycle latency perceptron estimator with an ideal, single cycle latency estimator. We find there is very little drop in the reduction in total uops executed for similar performance loss, even with the 9-cycle perceptron latency. On deep pipelines with large instruction buffers, slipping the start of pipeline gating by a few cycles to compute the confidence estimation does not allow too many additional instructions to enter the pipeline, relative to the total number of speculative instructions that could have been executed and squashed without pipeline gating.

The results reported in this Section thus show our perceptron-based branch confidence estimation mechanism is robust and works across a good range of perceptron array size and latency parameters.

5.5. Combining pipeline gating and branch reversal

We have seen the motivation behind using the perceptron branch confidence estimator for both pipeline gating and branch reversal simultaneously in Section 5.3. In this Section, we seek to minimize wasted speculative execution and maximize performance by applying both pipeline gating and branch reversal simultaneously.

Two thresholds are selected based on empirical data: 0 and -75. Branches are reversed when the perceptron output is larger than 0, and pipeline gating with a branch counter value of 2 is applied to branches with outputs between –75 and 0. All branches with output smaller than –75 are classified as high-confidence branches.

Figure 8 shows the reduction in total execution and the performance gain for each SpecInt 2000 benchmark when both branch reversal and pipeline gating are applied on our baseline 40-cycle pipeline. We see an average 10% reduction in total uops executed for no loss in overall average performance. Individual benchmarks gain/lose in performance due to correct/incorrect pipeline gating and/or branch reversals. The 10% reduction in total uops executed without any average performance loss obtained when both pipeline gating and branch reversal are applied is higher than the 8% reduction obtained using pipeline gating alone.

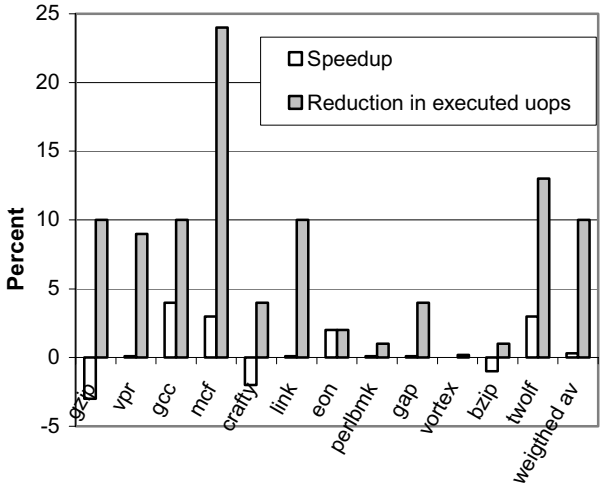

Figure 8. Combining pipeline gating and branch reversal on 40-cycle pipeline processor

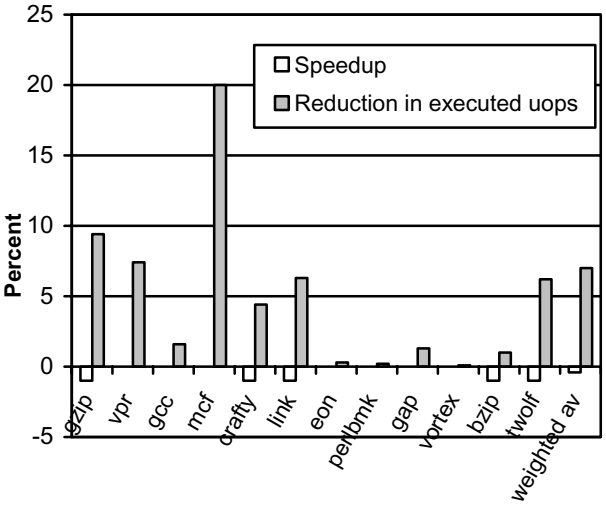

Figure 9. Combining pipeline gating and branch reversal on 8-wide 20-cycle pipeline processor

To understand the usefulness of our branch confidence estimator if machines got wider rather than deeper, we apply branch reversal and pipeline gating on 8-wide 20-cycle pipeline. Figure 9 shows the reduction in total execution and the performance for each SpecInt 2000 benchmark. Even though the 4-wide 40-cycle pipeline and the 8-wide 20-cycle pipeline have similar wasted mis-

speculative execution to begin with (Table 2), the wide machine achieves a lower reduction in uops executed for no performance loss than the deep machine. The reason is the reduced pipeline length for the wide machine configuration benefits less from branch reversal due to lower branch misprediction recovery time. Still, a significant reduction in mis-speculative execution, about 7% overall, is achieved for no performance loss using perceptron-based branch confidence estimation.

As with pipeline gating alone, other design options that achieve higher reductions in total uops executed at the cost of some performance loss are possible. In general, applying both pipeline gating and branch reversal reduces total execution more than just doing pipeline gating alone.

6. Concluding remarks

As pipelines get deeper or wider, wasted mis-speculative execution due to branch mispredictions increases. The percentage increase in total instructions executed is 24% on 4-wide 20-cycle pipeline and could be as high as 50% on 8-wide 20-cycle and 4-wide 40-cycle pipelines. Pipeline gating using a branch confidence estimator has been proposed as a means of reducing the amount of wasted speculative execution. The quality of pipeline gating relies heavily on the branch confidence estimator used. There has not been much work on branch confidence estimators since the initial work [6], and we show the current best-known method, enhanced JRS, does not work well on deep pipelines.

In this paper, we present a new, perceptron-based, branch confidence estimator, which is twice as accurate as the current best-known method and achieves reasonable mispredicted branch coverage. We train our perceptron-based branch confidence estimator using correct/incorrect branch information and show why it is better than training using taken/not-taken branch information. Furthermore, the output of our estimator is multi-valued, which enables us to classify branches further as "strongly low confident" and "weakly low confident". We reverse the predictions of "strongly low confident" branches and apply pipeline gating to the "weakly low confident" branches. This combination, gives a spectrum of interesting design options ranging from significantly reducing total execution for only a small performance loss, to lower but still significant reductions in total execution, without any performance loss.

7. References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors", *Proceedings of the 36th International Symposium on Microarchitecture,* San Diego, December 2003.

[2] J. L. Aragon, J. Gonzalez, J.M. Garcia, and A. Gonzalez, "Confidence Estimation for Branch Prediction Reversal", *Proceedings of the 8th International Conference on High Performance Computing,* December 2001.

[3] H. D. Block, "The Perceptron: A Model for Brain Functioning", *Reviews of Modern Physics,* 34:123-135, 1962.

[4] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. "Confidence Estimation for Speculation Control", *Proceedings of the 28th Annual International Symposium on Computer Architecture,* 2001.

[5] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium® 4 Processor", *Intel Technical Journal,* Q1 2001 Issue.

[6] E. Jacobson, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions", *Proceedings of the 31st International Symposium on Microarchitecture,* 1998.

[7] D. A. Jimenez, and C. Lin, "Dynamic Branch Prediction with Perceptrons", *Proceedings of the 7th International Symposium on High Performance Computer Architecture,* 2001.

[8] A. Klauser, S. Manne, and D. Grunwald, "Selective Branch Inversion: Confidence Estimation for Branch Predictors", *International Journal of Parallel Programming,* 29(1):81-110, February 2001.

[9] K. Luo, M. Franklin, S. S. Mukherjee, and A. Seznec, "Boosting SMT Performance by Speculation Control", *Proceedings of the International Parallel and Distributed Processing Symposium,* April 2001.

[10] S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction", *Proceedings of the 29th Annual International Symposium on Computer Architecture,* 2002.

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

[12] F. Rosenblatt, "The Perceptron, a Probabilistic Model for Information Storage and Organization in the Brain", *Psyhcology Review,* 62:386-408, 1958.

[13] J. E. Smith, "A Study of Branch Prediction Strategies", *Proceedings of the 8th Annual International Symposium on Computer Architecture,* May 198, pages 135 – 148.

[14] E. Sprangle, and D. Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines", *Proceedings of the 29th Annual International Symposium on Computer Architecture,* 2002.

[15] G. Tyson, K. Lick, and M. Farrens, "Limited Dual Path Execution", *CSE-TR 346-97*, University of Michigan, 1997.

[16] L. N. Vintan, and Mihaela Iridon, "Towards a High Performance Neural Branch Predictor", *Proceedings of the International Joint Conference on Neural Networks, Vol.2,* pp. 868 – 873, July 1999.