

# Revisiting the Performance Impact of Branch Predictor Latencies

Gabriel H. Loh

Georgia Institute of Technology  
College of Computing  
loh@cc.gatech.edu

## Abstract

*Branch predictors play a critical role in the performance of modern processors, and the prediction accuracy is known to be the most important attribute of such predictors. However, the latency of the predictor can also have a profound impact on performance as well. In past studies that have considered branch prediction latency, most only consider the latency required to make a prediction. However, in deeply pipelined processors, the latency between prediction and update can also greatly affect performance. In this study, we revisit the performance impact of both of these latencies and demonstrate that update latency can also have a significant impact on performance. We then describe two techniques, multi-overriding and hierarchical updates, to address both latencies which provide 4.4% and 5.7% IPC improvements on moderately (20-stage) and deeply (40-stage) pipelined processors, respectively, for minimal hardware complexity.*

## 1. Introduction

The hardware branch predictor is a very important contributor in determining the overall performance of a processor. When considering the relationship between the predictor and performance, the branch prediction accuracy is usually the most important attribute of a given predictor [1]. However, the latency of the branch predictor can also have a profound effect on performance [8]. In this paper, we study the performance impact of two different latencies associated with branch predictors. The first is the latency required to make a prediction, and the second is the latency required to update the predictor.

A branch predictor with a long prediction latency either causes the overall processor clock frequency to decrease or it forces a pipelined implementation. Any impact to the clock speed is undesirable because it directly reduces performance and it is likely that a slightly less accurate, but faster predictor would be more beneficial. Pipelining the branch predictor is not as straightforward as pipelining other processor circuits because the predictor forms a critical loop [4]. For each cycle, the processor must make a branch prediction to determine the fetch location for the following cycle. A predictor with a two-cycle latency results in fetch bubbles every other cycle. Several techniques have

been proposed to address this issue [8, 10, 21].

The lookup and update events for a branch predictor are temporally separated by the processor pipeline. The branch prediction lookup occurs at the front of the pipeline during the fetch stage, while the update typically occurs at the end of the pipeline at commit. A branch may observe hundreds of cycles between fetch and commit if, for example, an earlier load instruction could not retire due to a cache miss that went to main memory. This could introduce a substantial delay between predictor lookup and update. A processor may have resolved a mispredicted branch within tens of cycles after the initial prediction, but it may be hundreds of cycles before the processor is able to feed back this information to train the predictor. In the mean time, the front-end will have been re-steered, and then the same branch may be re-encountered and then mispredicted again.

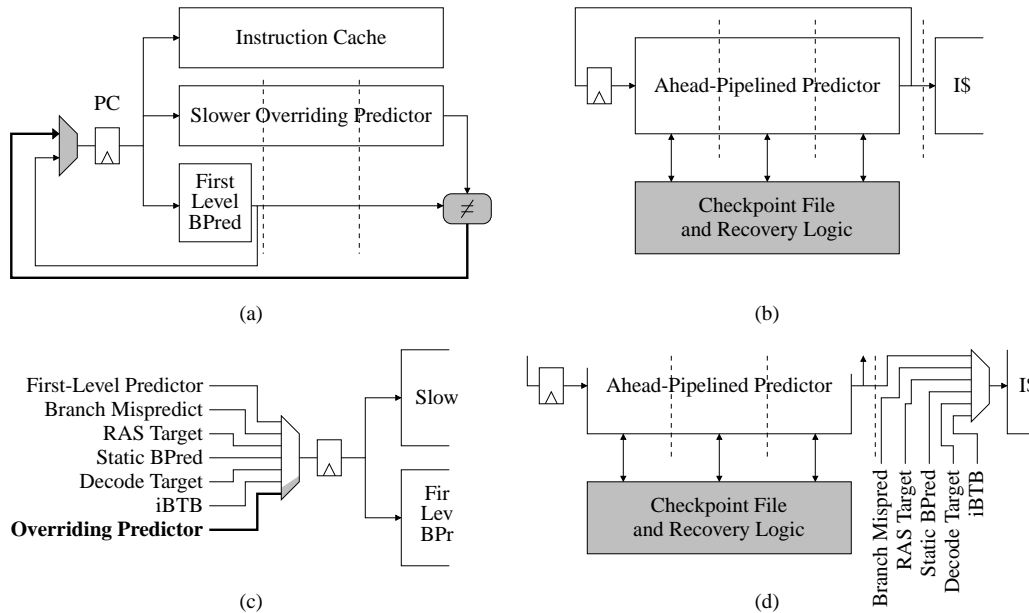
In this paper, we demonstrate the performance impact of these two types of branch predictor latencies. Our results re-validate previous observations that the lookup latency can greatly impact performance. We also show that the update latency can also have a significant impact on performance. We then present two techniques to address these two types of latency and demonstrate their benefits. The rest of the paper is organized as follows. Section 2 describes the various branch predictor latencies in more detail and quantifies their impact on processor performance. Section 3 describes the technique for hierarchical update to address the latency of updating predictors when an overriding predictor configuration is used. Section 4 explains how to adapt the perceptron predictor to provide a faster intermediate prediction to reduce the average prediction lookup latency. Section 5 presents the overall performance results from addressing the predictor latencies. Section 6 relates our work to previous research concerning branch predictor latency, and Section 7 concludes the paper with some final remarks.

## 2. Branch Predictor Latencies

The hardware branch predictor has multiple latencies associated with it. In this section, we review these latencies and their impact on processor performance.

### 2.1. Lookup Latency

The branch predictor forms a critical loop in the processor. A branch prediction must be made every cycle for the



**Figure 1. Additional hardware required for techniques to mitigate branch predictor lookup latency: (a) overriding branch predictor, (b) ahead-pipelined branch predictor, (c) overriding overhead compared to existing re-steer requirements, (d) ahead-pipelining overhead with existing re-steer requirements.**

processor to fetch instructions on every cycle as well. However, each branch prediction is dependent on the prediction from the previous cycle, which forms a fundamental recurrence. If looking up a prediction from the hardware branch predictor requires a long latency, then this may directly impact the overall clock frequency. If the predictor is pipelined across more than one cycle, then predictions will not be available on every cycle which introduces fetch bubbles.

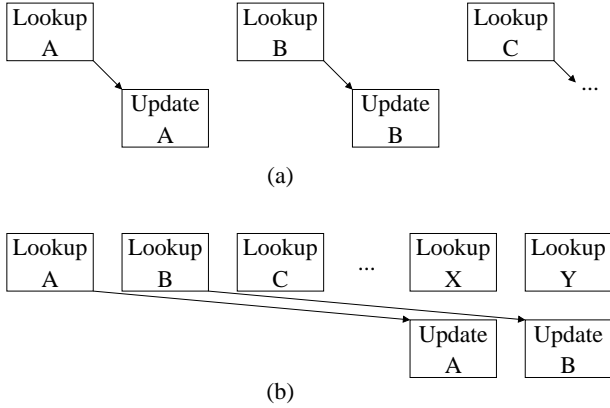
One technique for combating branch predictor lookup latency is to make a “quick-and-dirty” prediction that can be accomplished in a single cycle, and then make corrections later as better information becomes available. Jiménez et al. proposed *overriding* predictors where a fast, single-cycle predictor makes the initial branch prediction and then a slower but more accurate predictor provides a (hopefully) better prediction that may override the first [11]. If the predictors are in agreement, then there are no additional fetch bubbles injected into the pipeline. If the predictors disagree, then the front-end is re-steered based on the overriding predictor, which trades a full pipeline flush for just a few cycles of fetch bubbles.

The general concept of predictor overriding has been used in different forms even before the work by Jiménez et al. For example, a processor that does not use predicted decode information [21] will not know that a particular instruction is a subroutine return until the decode stage. In this case, the normal branch prediction logic provides the

predicted fetch target, but after decode the return address stack may override the earlier prediction with a better fetch target. The Alpha 21264 used a combination of line- and way-prediction to provide a continuous fetch stream without bubbles, and then backed this with a slower, but more accurate hybrid branch predictor [14].

Another approach to deal with branch predictor latency is to use *ahead-pipelining* [21]. The idea is that if a predictor requires  $N$  cycles to perform a lookup, then one could instead initiate the lookup  $N$  cycles ahead of time using whatever information is available *at that time*. Normally, the current branch address ( $PC$ ) is used to make the prediction about that branch’s direction. Since this address only becomes available at the start of the current cycle, the prediction could not normally be initiated any earlier. By using  $PC_{-N}$  (the  $N^{\text{th}}$  oldest  $PC$ ) and relying on the existence of correlation between  $PC_{-N}$  and  $PC$ ’s direction, the prediction could be ahead-pipelined.

Proponents of ahead-pipelining argue that overriding predictors introduce too much complexity to the fetch engine. An overriding predictor organization requires extra comparisons between the two levels of prediction and hardware support to re-steer the fetch engine to the “better” fetch target. Figure 1(a) illustrates the hardware for an overriding predictor organization, with the additional components shaded/bolded. With ahead-pipelining, there are no extra comparisons or fetch redirections, however the pro-



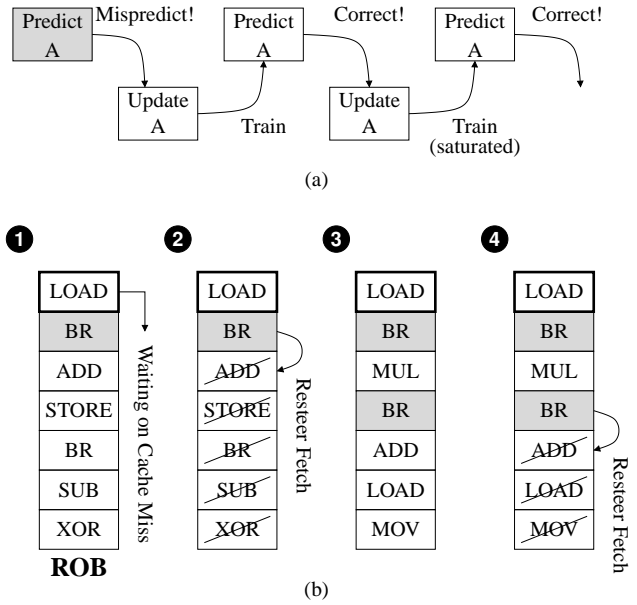
**Figure 2. (a) Ideal alternating branch predictor lookups and updates, and (b) realistic timing of branch predictor lookups and updates due to a pipelined processor organization.**

cessor must checkpoint the entire prediction pipeline for every branch to avoid exposing the predictor latency on a branch misprediction. This extra hardware is shown in Figure 1(b). The checkpoint enables the processor to revert the predictor pipeline to the exact state when the misprediction first occurred. While this provides a high-performance prediction architecture, the overhead of the checkpointing can be substantial [17].

We believe that the overriding predictor organization actually introduces far less complexity than the ahead-pipelined techniques. Processors already contain multiple “re-steer points” for the return address stack, the static branch predictor, decode-time branch target correction, normal branch misprediction correction, and possibly other reasons (indirect BTB predictions at decode, synchronous interrupts like page faults or divide-by-zero faults, load-store memory misspeculation). The overriding predictor organization only adds a single new re-steer condition as shown in Figure 1(c), whereas the ahead-pipelined predictor must include all of the infrastructure for predictor checkpointing as illustrated in Figure 1(d). Although an ahead-pipelined predictor was proposed for the canceled EV8 project [20], we are not aware of any processors that currently make use of ahead-pipelined predictors. For this reason and the complexity arguments already presented, we will only consider overriding predictor organization in the rest of this paper.

## 2.2. Update Latency

While the predictor lookup latency problem has been extensively studied [8], there has been far less attention paid to the predictor update latency. In the ideal branch prediction problem, predictor lookup and update would proceed in a



**Figure 3. Example predictor update behavior for (a) ideal alternating lookup and update, and (b) a realistic pipeline with a long commit stall.**

strictly alternating pattern as illustrated in Figure 2(a). In reality, the pipelined implementation of the processor causes a *slip* between when a branch is predicted and when the branch retires/commits and updates the predictor, as shown in Figure 2(b). This slip causes a divergence from the idealized alternating predict-update process.

The delay between predictor lookup and update can cause the same static branch to be dynamically mispredicted multiple times. Consider the branch in Figure 3(a) with ideal alternating lookup and update. After the first misprediction, the predictor update causes subsequent instances of the same static branch to be predicted correctly. However in a real pipeline as shown in Figure 3(b), the commit-time update may be blocked by an earlier load-miss that cannot retire (1). In the meantime (2), the execute stage detects the misprediction and re-steers the fetch engine. Later (3) the fetch engine encounters the same static branch again, but because the predictor update has not yet occurred, the branch predictor makes another misprediction. After some time, this second misprediction will be detected (4) causing another pipeline flush and re-steer, however the predictor still has not been updated. This may repeat many times before the original misprediction finally commits and trains the predictor.

Past studies have already identified the impact of not speculatively updating certain branch predictor state [7]. In particular, the branch history register needs to be updated immediately after prediction with the predicted branch out-

### Base Processor Configuration

Machine Width	4	IL1, DL1	16KB, 4-way, 2-cycle*
Instruction Fetch Queue	16 insts	Unified L2	512KB, 8-way, 6-cycle*
RS Size	64 insts	Unified L3	8MB, 16-way, 10-cycle*
LSQ Size	32 Load/32 Store	Main Memory	100-cycle*
ROB Size	256 insts	BTB	4K entries, 4-way

### Branch Predictor Configurations

Predictor	Description	Latency at different Pipeline Depths			
		10	20	30	40
First-Level	2K-entry <sup>o</sup> gshare	1	1	1 <sup>o</sup>	1 <sup>o</sup>
Hybrid	1K entry local history 1K entry local predictor 4K entry each, global and meta	2	3	4	5
Perceptron	47 history length/348 entries, 16KB	2	4	6	8

**Table 1. Processor and predictor configurations used in this study. \*The reported cache/memory latencies are for the 10-stage pipeline; the access latencies for the deeper pipelines are scaled appropriately. <sup>o</sup>The size of the first-level gshare predictor is reduced to 1K entries for the 30- and 40-stage pipelines.**

come. This actually leads to the correct branch history state if the predictions are correct, and the history register needs to be repaired after a misprediction. Speculative update of the branch history is a well understood problem and will not be further investigated in this study.

For highly accurate hardware branch predictors such as the perceptron [12] or a hybrid predictor [18], the update latency only poses a problem for the initial training phase of the predictor. After each static branch has been observed a few times, the predictor state stabilizes and most updates are ignored because the corresponding counter states are already saturated. However in an overriding predictor organization, the first-level single-cycle predictor is subject to a large number of capacity and conflict misses and may suffer from a much larger number of mispredictions due to predictor update delay. In the following section, we will quantify the performance impact of both the lookup and update latencies on overriding branch predictor organizations.

## 2.3. Performance Impact of Predictor Latencies

This section provides our initial performance characterizations illustrating the impact of branch predictor lookup and update latencies. We first provide the details of our experimental methodology, and then quantify the performance ramifications of non-ideal (real) branch predictor behavior.

### 2.3.1. Evaluation Methodology

In this study, we consider two branch prediction algorithms. The first is the global-local hybrid predictor [18] (denoted by an ‘h’ in the figures) used in the Alpha 21264 processor [14]. We speculatively update both the global and local histories, but only repair the global history on a branch

misprediction. The second is a 16KB perceptron predictor (denoted by a ‘p’) with a global history length of 47, which is the optimal length as reported by Jiménez and Lin [12]. In all cases, the small first-level, single-cycle predictor uses the gshare prediction algorithm [18]. We chose a gshare predictor because a bimodal (two-bit counter) predictor’s lower prediction rates would inflate the frequency and benefit of useful overrides; Jiménez et al.’s study on overriding predictors also used a gshare for the first-level predictor. For the more aggressively clocked pipelines (30 and 40 stages), we reduce the size of the first-level gshare predictor to account for the decrease in cycle time.

We evaluated the predictors on processors with pipeline depths varying from 10 stages to 40 stages, where the “depth” is the number of stages from the single-cycle branch predictor to fetch-restore on a misprediction. Any pipeline stages that come after execute are not included in this pipe-stage count. As the pipeline depth increases, the clock cycle time also decreases, which causes the branch predictor lookup latency (in cycles) to increase. The reduced clock cycle time also causes the first-level predictor to decrease in size to maintain cycle-time constraints. These details are listed in Table 1.

We used the SimpleScalar microarchitecture simulator [2, 15] for the Alpha instruction set architecture. We model the pipeline by explicitly simulating extra stages between the instruction fetch queue and the dispatch stage (this is different from and more accurate than the default SimpleScalar method of simply stalling fetch for some number of cycles). We simulated applications from the SPECint2000 benchmark suite, as well as programs from MediaBench [16], MiBench [6], X-graphics applications, and pointer-intensive benchmarks [3]. The applications are

SPECint2000	Other Benchmarks			
	Program Name	Suite	Program Name	Suite
bzip2 (3)	adpcm (2)	Media	ks (2)	Ptr
crafty	bc (3)	Ptr	mesa (2)	Media
eon (3)	crc32	MiB	mpeg2 (2)	Media
gap	dijkstra	MiB	povray (5)	Gfx
gcc (5)	doom (2)	Gfx	quake-GL (2)	Gfx
gzip (5)	epic (2)	Media	quake-X	Gfx
mcf	fft (2)	MiB	rsynth	MiB
parser	ft	Ptr	sha	MiB
perlbmk (4)	g721 (2)	Media	susan (2)	MiB
twolf	ghostscript	Media	tiff (4)	MiB
vortex (3)	ghostscript (2)	MiB	xanim	Gfx
vpr (2)	jpeg (2)	Media	yacr2	Ptr

Suite Name	Description
Media	MediaBench Multimedia Applications
MiB	MiBench Embedded Applications
Ptr	PtrDist Pointer-Intensive Programs
Gfx	X-Windows graphics applications

**Table 2. The benchmarks used in this study. The numbers in parentheses indicate the number of different input sets/modes (e.g., encode vs. decode).**

listed in Table 2. We also ran our experiments for the SPECfp2000 programs, but we do not report their results because the very low misprediction rates of these applications makes their performance largely insensitive to the branch predictor latencies. For all applications, we use the SimPoint toolset to find representative simulation sample points of 100M instructions each [19]. Figure 4 shows the baseline IPC rates for the different pipeline depths and the two branch predictors evaluated.

### 2.3.2. IPC Results

We simulated four predictor configurations representing different levels of “ideal” branch predictor latency. The baseline case (BASE) is a processor that uses a realistic overriding predictor organization and does not update<sup>1</sup> the predictor until the commit stage. The No-Lookup-Latency (NLL) configuration assumes an ideal situation where the slow overriding predictor only requires a single-cycle latency. The performance difference between BASE and NLL quantifies the performance impact of the predictor lookup latency. The Prediction-Time-Update (PTU) configuration simulates an ideal situation where the predictor update occurs immediately after lookup. We use oracle information to update the first-level and overriding predictors with the correct outcome, and we also make use of the oracle information to filter-out updates due to wrong-path branches. The oracle information is not used to re-steer the current branch, but only to influence the predictions of future in-

<sup>1</sup>By update, we mean the phase that actually changes the values of the predictor counters or weights. For all configurations that update at commit, the branch history register is still speculatively updated immediately after prediction and recovered on a misprediction.

stances of the same branch. Finally, the IDEAL configuration combines the ideal updates of NLL and PTU to provide an upper bound on performance when no predictor latencies are accounted for.

The impact of predictor lookup latency can have a profound effect on performance as evidenced by the difference between the BASE and NLL data in Figure 5. These results corroborate the previously reported impact of lookup latency [11]. Depending on the predictor latency and pipeline depth, idealizing the lookup latency improves performance by 2.4% (h40) to up to 9.4% (p40) for both SPECint and the other applications. Figure 5 shows that the performance impact of the update delay is surprisingly also quite substantial. While the ideal update latency does not improve performance by as much as an ideal lookup latency, the 1.5% (h40) to 3.3% (p40) performance is not negligible. While past research has focused on the lookup latency, the branch prediction community may have missed an opportunity to improve performance by addressing the update latency.

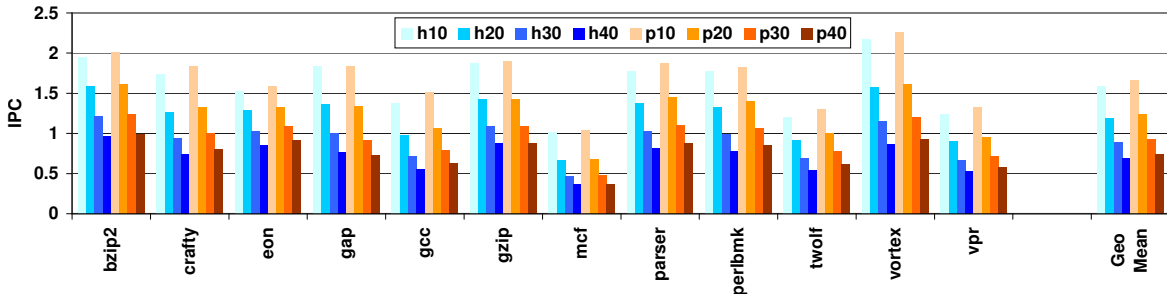
Finally, the IDEAL results show that the performance degradation due to lookup and update latency do exhibit some overlap. If an ideal lookup latency could be attained, then there is not much additional performance benefit to achieving an ideal update latency. However, this does not suggest that update latency is not interesting because we cannot actually build a high-performance branch predictor that meets the ideal lookup latency criteria!

## 3. Hierarchical Update

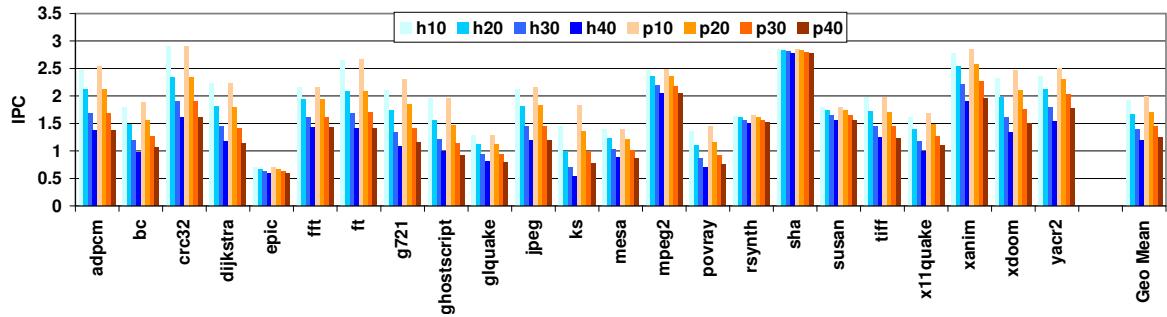
The primary problem with the update latency is that any sort of pipeline stall may result in delaying a branch from retiring and thereby updating the branch predictor. This latency introduces many mispredictions in the small and fast first-level predictor. Instead of waiting until the commit stage to update the first-level predictor, a processor can instead use the overriding predictor as a source of update information. The insight is that it is better to update the small predictor sooner with results that are 90% correct rather than wait for a long time to get 100% correct results. We call this update policy *hierarchical update* because there are now multiple levels of update.

The hardware modifications to support hierarchical update for an overriding predictor organization are minimal. For a conventional overriding predictor organization, there already exists an update path from the commit stage to the branch predictors. When using hierarchical update, a single extra multiplexer is needed for the first-level predictor’s update path to choose whether to update using the commit-time information or the overriding predictor’s information. Figure 6 shows the hardware modifications for hierarchical update, with the new components shaded.

The training algorithm for the first-level predictor needs to be modified to properly handle updates from multiple

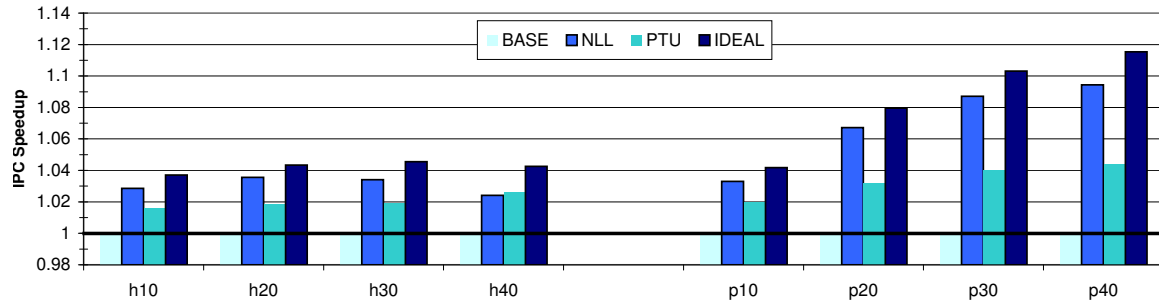


(a)

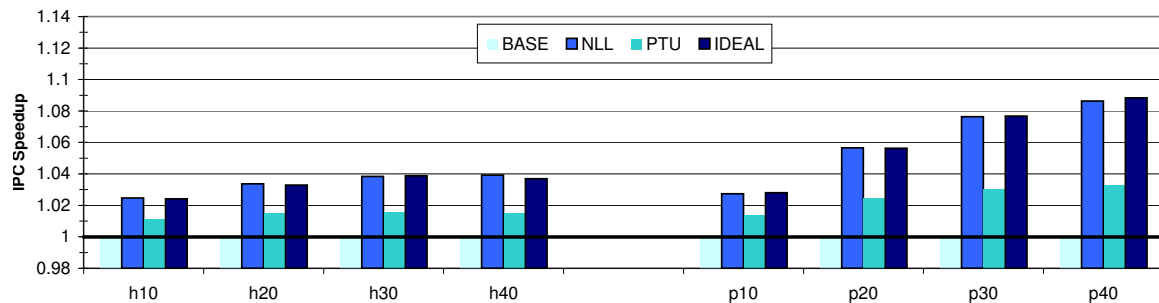


(b)

Figure 4. Base IPC rates (absolute) for the (a) SPECint benchmarks and (b) the other applications evaluated in this study for the two different branch predictors and four pipeline depths. “xNN” denotes the predictor ( $x \in \{h=\text{hybrid}, p=\text{perceptron}\}$ ) and pipeline depth ( $NN \in \{10, 20, 30, 40\}$ ).



(a)



(b)

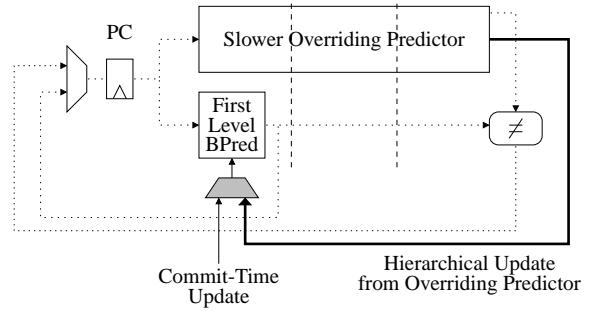
Figure 5. Relative IPC speedup when different aspects of branch prediction latency are idealized. NLL is no-lookup-latency, PTU is prediction-time-update, and IDEAL combines both.

sources. In particular, if the overriding predictor is incorrect, it will cause the first-level predictor’s counters to be trained in the incorrect direction. For example, an overriding predictor that predicts not-taken causes the corresponding gshare saturating counter to decrement from, say, weakly not-taken (01) to strongly not-taken (00). At commit time, the processor uses the correct outcome (taken branch) to update the predictors. In this scenario, the gshare counter is incremented back to weakly not-taken. It is possible that this update pattern repeats back and forth, which would cause the gshare counter to never leave the not-taken states.

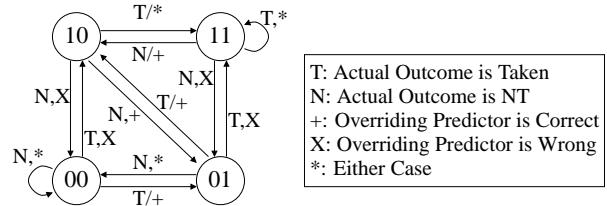
To address the problem of conflicting updates in the first-level predictor, the processor needs some additional compensation to deal with the training “pollution” introduced by the incorrect update from the overriding prediction. If the overriding predictor was incorrect, then, conceptually, the first-level predictor needs to be updated twice with the actual branch outcome. The first update “undoes” the overriding predictor’s bad update, and the second update performs the normal training. Referring back to the previous example, two updates would cause the gshare counter to increment twice, putting the counter in a weakly taken (10) state. Having to update the predictor twice in this fashion is not desirable because it would require two writes to the predictor per update. Instead, the saturating counter finite state machine logic could be modified such that the increment/decrement delta is two when the overriding predictor previously incorrectly updated the first-level predictor. Figure 7 illustrates this modified FSM state-transition diagram, where edges labeled with “+” indicate when the overriding predictor was correct, “X” indicated when the overriding predictor was wrong, and “\*” covers both cases. Note that if the early hierarchical update was correct, then the first-level predictor will not be redundantly updated again at commit.

On a branch misprediction, the first-level predictor may have already been incorrectly trained by wrong-path branches. Since the wrong-path branches are flushed on the misprediction, they will never reach the commit stage to apply the correction training as described above. In a processor architecture like the Intel Pentium-Pro where the ROB is drained at the commit rate (as opposed to “instantaneously” flushing all instructions as is typically modeled), the squashed branches could still be detected and some correction applied to the first-level branch predictor. In our model, we assume the default SimpleScalar behavior of flushing all instructions in a single cycle and simply do not make any attempt to repair the state of the first-level predictor.

Overall, hierarchical update provides a way to keep the contents of the first-level predictor “fresher” than waiting for commit-time updates. The technique of compensating updates prevents over-pollution due to incorrect information



**Figure 6. The hardware modifications to support hierarchical update of the first-level predictor.**



**Figure 7. Modification to the saturating two-bit counter algorithm for the first-level predictor to include compensating updates.**

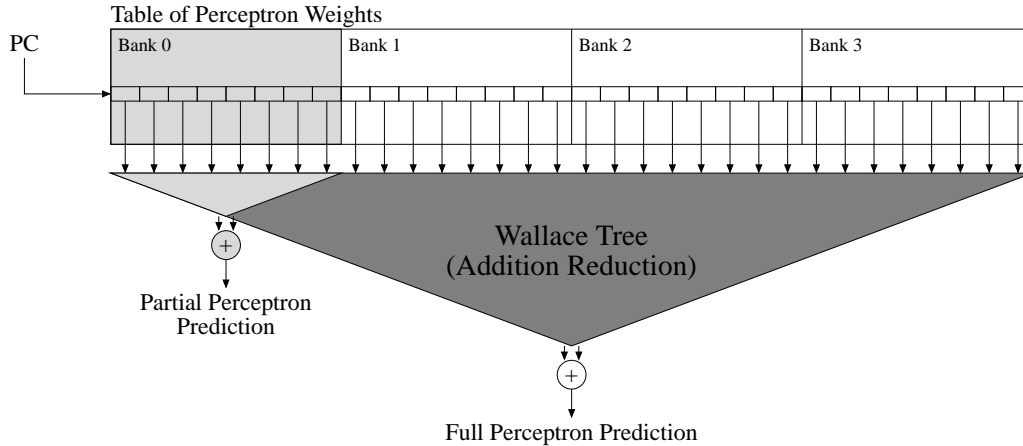
from the overriding predictor.

#### 4. Multi-Overriding Predictors

Jiménez also demonstrated that if a perceptron predictor’s lookup latency is allowed to become too great, it could also result in lower performance than a faster but less accurate gshare [10]. The total number of fetch cycles wasted due to an overriding predictor configuration can be approximated by the predictor latencies, the pipeline depths, and the predictor accuracies. Let  $L_{or}$  be the latency of the overriding predictor,  $L_{pipe}$  be the latency to resolve a branch misprediction,  $m_1$  and  $m_{or}$  be the misprediction rates of the first-level predictor and the overriding predictor, respectively. Then the number of fetch bubbles is approximately:

$$m_1 \times (1 - m_{or}) \times L_{or} + m_1 \times m_{or} \times L_{pipe} + (1 - m_1) \times m_{or} \times (L_{or} + L_{pipe})$$

The first term is the number of fetch bubbles due to the case where the first-level predictor is incorrect and the overriding predictor correctly re-steers the fetch engine. The penalty is only equal to the latency of the overriding predictor ( $L_{or}$ ) each time this occurs. The second term covers the cases where both predictors are incorrect and the full pipeline latency is exposed. The last term covers the rare cases where



**Figure 8. Modifying the original perceptron algorithm to provide a partial perceptron prediction based on only the more recent branch history. The only additional hardware is the single small adder (left). The banked table organization and the shallower addition tree provide a faster prediction latency.**

the first predictor is actually correct, the overriding predictor is wrong and misguidedly re-steers the fetch engine off the correct path ( $L_{or}$ ), and then later ( $L_{pipe}$ ) the processor back-end re-corrects the instruction flow to the original correct path. The main observation is that any increase in the branch predictor lookup latency must provide a large enough reduction in the misprediction rate just to offset the additional fetch bubbles (primarily) due to the second term of the above equation.

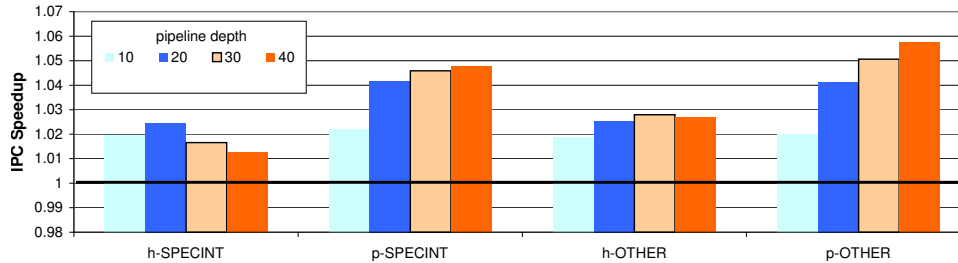
To combat the increasing latency of larger and more accurate predictors, we can take inspiration from the design of cache hierarchies. When the difference in latencies from one cache level to the next becomes too large, processor designers may choose to add an intermediate level of caching to reduce the *average* cache access latency. For example, the increasing difference in relative speeds of the processor and main memory have made several high-end processor designs to include level-three caches. As the latency of the overriding branch predictor increases, we can add another level of prediction that has latency and prediction rates that are somewhere between the first-level predictor and the overriding predictor.

The *multi-overriding* predictor configuration uses three (or more levels) of successively overriding predictors to provide a better *average* branch predictor lookup latency. The first-level predictor continues to provide fast single-cycle predictions, and then the second predictor provides a (usually) better prediction with moderate latency. Each successive predictor provides increasingly more accurate predictions with increasing latency. There is a practical limit to the latency of the branch predictors. Any override that occurs before the corresponding instructions reach the register rename stage can be easily handled by flushing the pipeline

stages up to the predictor and then resetting the fetch engine to the new fetch address. However, once instructions have been renamed, resteeering also involves recovering the rename table state. This represents a much greater level of complexity and therefore for all practical purposes the register rename stage represents a “point of no return” with respect to predictor overriding.

In this study, we only consider a three-level overriding predictor organization, partly due to the required additional hardware to implement a large number of predictors. However, adding even a third branch predictor to an overriding predictor organization requires extra hardware. The actual override data-path does not pose a significant hardware cost because the fetch resteer multiplexer (shown in Figure 1) already supports multiple resteer points. Adding one more resteer point certainly requires more logic, but not substantially more. On the other hand, implementing an extra branch predictor does potentially add a lot of extra hardware.

Ideally, adding an extra branch predictor would require little or no extra hardware. While this may seem unrealistic, we can potentially reuse the existing hardware to provide our intermediate branch prediction. In particular, past studies have shown that a branch outcome is typically correlated with only a few other branches in the branch history [25]. Furthermore, the strongest correlations tend to be with the most recent bits in the branch history [17]. Using these observations, we propose *partial perceptron prediction*. The perceptron’s addition reduction operation incurs a substantial amount of delay during lookup. However, for the cost of just a single extra adder, we can “tap” into the adder tree (Wallace Tree) and obtain a prediction based on a prefix of the branch history (i.e. a shorter branch history). Figure 8



**Figure 9. Relative IPC speedup when hierarchical update is applied to the different predictor and pipeline configurations.**

shows the one extra adder and how the partial prediction through this path has a lower latency. Furthermore, the table of perceptron weights could be banked such that the weights corresponding to the most recent branch history can be read even faster. Overall, this provides a short-history perceptron predictor in less latency than the full perceptron at the cost of a single narrow-width adder.

In the following section, we present performance results showing how Hierarchical Update and Multi-Overriding predictors can provide a substantial IPC speedup over a conventional overriding predictor configuration. We also evaluate using the two techniques in tandem to attack both types of predictor latency. When combining the techniques we only update the first-level predictor using the complete perceptron predictor. We do not perform a hierarchical update on the weights corresponding to the partial perceptron prediction because that creates the strange situation where part of the complete perceptron is being updated with its own predictions.

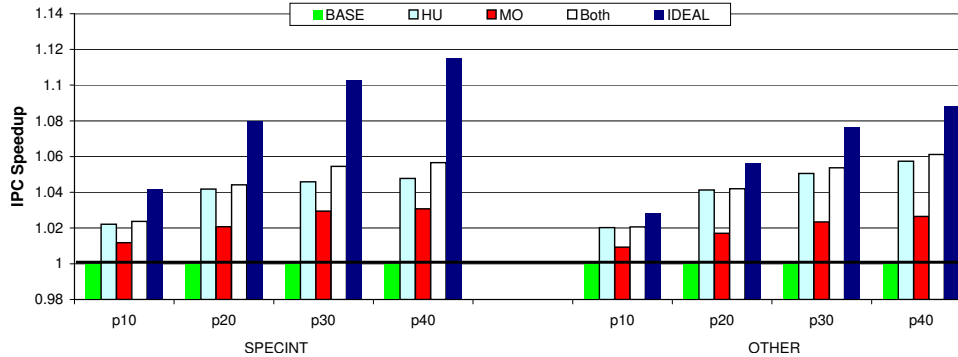
## 5. Results

Hierarchical update and multi-overriding attempt to improve processor performance by addressing two different aspects of branch predictor latency. We first study the performance impact of using hierarchical updates. Figure 9 shows the geometric mean IPC speedup attained when using hierarchical updates relative to the baseline configurations that always wait until commit to update the first-level predictor. As the pipeline depth increases, the delay to update the branch predictors also increases. This delay in turn tends to make the contents of the first-level predictor more stale, and explains why hierarchical update provides greater performance gains for the deeper pipelines. The difference in performance between the hybrid predictor and the perceptron stem from the fact that the perceptron is substantially more accurate. The benefit of the hybrid predictor for longer pipe-depths is not as great due to the lower accuracy of the overriding predictor, which feeds more incorrect early updates to the first-level predictor. For a moderately pipelined (p20) processor with the perceptron predic-

tor, simply changing the update policy to use hierarchical update can provide a 4.2% (4.1%) performance improvement for SPECint (other benchmarks). For a very deeply pipelined processor (p40), the speedup increases to 4.8% (5.7%). Between SPECint and the other benchmarks, the relative trends are similar.

For the multi-overriding predictors, we only consider the perceptron-based configurations because the partial-perceptron predictor obviously does not work with the hybrid branch predictor. While it might be possible to design a hybrid branch predictor where some components provide predictions faster than others, these additional designs are not explored in this paper. Figure 10 shows the geometric mean IPC speedups for the multi-overriding predictors (MO). In addition, Figure 10 also includes the results for hierarchical updates (HU), combining both together, and the IDEAL speedup when both lookup and update latencies are idealized. Multi-overriding provides less performance benefit than the hierarchical update. These results may seem surprising since Figure 4 showed that a perfect lookup latency has a greater impact than a perfect update latency. The reason why hierarchical update has a greater impact is that it indirectly reduces the average lookup latency as well. When a hierarchical update correctly trains the first-level predictor, the latency of the overriding predictor becomes inconsequential (assuming that the override is correct as well).

Combining both hierarchical update and multi-overriding should provide even greater performance improvements. While this is true, Figure 10 shows that the benefits are not strictly additive. The previous discussion about how hierarchical update indirectly affects the average lookup latency also provides insight into why the two techniques have overlapping contributions. At shallow pipeline depths (10-20 stages), adding multi-overriding does not provide any substantial benefit over the hierarchical update technique. For the deeper pipelines, it provides an additional 1% performance improvement for a very small cost in additional hardware and complexity. Figure 10 also shows the performance speedup for IDEAL predictor lookup and update latencies. Depending on the configuration, the combination of our techniques achieves



**Figure 10. Relative IPC speedup when hierarchical update (HU), multi-overriding (MO), and the combination of both for the perceptron-based configurations. IDEAL includes both ideal lookup and update latencies.**

49-74% of the potential performance benefit. For the low hardware overhead, this is a considerable IPC speedup. However, the IDEAL results show that some headroom remains.

## 6. Related Work

While there is a very large body of past research on branch prediction, we will limit our discussion to studies that have specifically targeted various aspects of branch predictor latency. Yeh and Patt studied different two-level branch predictor organizations and concluded that speculative updates of the branch predictor *history* (not the saturating counters) are required for maintaining a high prediction accuracy [26]. A subsequent study by Talcott et al. provided arguments that older branch histories can provide accurate predictions, thus contradicting the Yeh and Patt study [24]. These results would suggest that speculative branch history updates are not necessary, and it would further render hierarchical updates to be superfluous as well. However, Hao et al. countered with a more detailed cycle-level evaluation of the effects of speculative history update (the paper by Talcott et al. assumed a constant delay between prediction lookup and update) and demonstrated that the variability in lookup-to-update latency did in fact make speculative history updates a requirement [7]. Subsequent studies explored the impact of speculative updates on branch predictors using local branch history [23], the return address stack predictor [13, 22], as well as the interaction of speculative branch history updates with the corresponding history repair mechanisms after a misprediction has been detected [13]. Jourdan et al. also report that the updating of the PHT counters is insensitive to the latency of update. This at first may appear to be in contradiction with the results reported in this study. However, we only claim that the small, first-level predictor is sensitive to update latency; we conducted similar simulations and our results corroborate these earlier findings that

the large overriding predictor is largely unaffected by update latency.

Apart from the impact of speculative branch history update, the remaining latency-related branch prediction research has focused on the lookup latency of the branch predictor. Jiménez et al. examined the impact of increasing clock frequency on branch predictors and overall processor performance [11]. In particular, a higher clock frequency reduces the largest possible size of a predictor that can still be accessed in a single cycle. The combined effects of deeper pipelines (larger misprediction penalty) and smaller branch predictors (more frequent mispredictions) can cause a substantial decrease in overall performance. The study proposed a variety of mechanisms for dealing with the timing constraints of predictors; the overriding predictor organization is one of their proposed techniques, which also has similarities to the earlier proposed cascaded predictors [5]. Subsequent work has also proposed the idea of ahead-pipelining to hide the latency of large branch predictors. Seznec et al. proposed an ahead-pipelined hybrid predictor for the canceled Alpha EV8 processor [20], and then later extended the ahead-pipelining idea to other components in the fetch engine [21]. Jiménez adapted the gshare predictor in a technique similar to ahead-pipelining that initiates the prediction with an older branch address, and then makes use of a subset of the actual branch address at the last instance to compute the final prediction [10]. Jiménez also proposed ahead-pipelined versions of his neural predictors, but they still require multi-cycle lookups which incurs the overheads of both checkpointing and overriding [9].

## 7. Conclusions

In this study, we revisited the impact that branch predictor latency has on the performance of a processor. In particular, we examined both the well-known problem of predictor lookup latency as well as the less studied update

latency problem. We found that while the lookup latency does have a larger impact on overall performance, our performance characterizations indicate that the update latency should not be ignored. Inspired by our initial performance studies, we propose two techniques aimed at the two different predictor latencies. The combination of the two methods provides a substantial IPC improvement of 4.4% (5.7%) on a 20-stage (40-stage) processor pipeline, while introducing very little additional hardware.

One of the high-level lessons from this study is that it is very important to fully quantify the performance impact of all aspects of microarchitecture design decisions. The update portion of branch predictors has largely been ignored in the past, and as a result the branch prediction research community missed an opportunity to improve predictor behavior. There may be many other components in a modern microarchitecture that warrant a more detailed “second-look” to study how performance (or power, temperature, reliability, etc.) may be affected by design aspects that are usually taken for granted.

## Acknowledgments

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

## References

- [1] The 1st JILP Championship Branch Prediction Competition (CBP-1). <http://www.jilp.org/cbp>.
- [2] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.
- [3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, FL, USA, June 1994.
- [4] Eric Borch, Srilatha Manne, Joel Emer, and Eric Tune. Loose Loops Sink Chips. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pages 299–310, 2002.
- [5] Karel Driesen and Urs Hölzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 249–258, Dallas, TX, USA, November 1998.
- [6] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the 4th Workshop on Workload Characterization*, pages 83–94, Austin, TX, USA, December 2001.
- [7] Eric Hao, Po-Yung Chang, and Yale N. Patt. The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 228–232, San Jose, CA, USA, November 1994.
- [8] Daniel Jiménez. *Delay-Sensitive Branch Predictors for Future Technologies*. PhD thesis, University of Texas at Austin, January 2002.
- [9] Daniel A. Jiménez. Fast Path-Based Neural Branch Prediction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 243–252, San Diego, CA, USA, December 2003.
- [10] Daniel A. Jiménez. Reconsidering Complex Branch Predictors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 43–52, Anaheim, CA, USA, February 2003.
- [11] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The Impact of Delay on the Design of Branch Predictors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 4–13, Monterey, CA, USA, December 2000.
- [12] Daniel A. Jiménez and Calvin Lin. Neural Methods for Dynamic Branch Prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.
- [13] Stephan Jourdan, Jared Stark, Tse-Hao Hsing, , and Yale N. Patt. Recovery Requirements of Branch Prediction and Storage Structures in the Presence of Mispredicted-Path Execution. *International Journal of Parallel Programming*, 25(5):363–384, 1997.
- [14] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro Magazine*, 19(2):24–36, March–April 1999.
- [15] Eric Larson, Saugata Chatterjee, and Todd Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, pages 1–9, Tucson, AZ, USA, November 2001.
- [16] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, NC, USA, December 1997.
- [17] Gabriel H. Loh. A Simple Divide-and-Conquer Approach for Neural-Class Branch Prediction. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, USA, September 2005.
- [18] Scott McFarling. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.
- [19] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, New Orleans, LA, USA, September 2004.
- [20] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, Anchorage, AK, USA, May 2002.
- [21] André Seznec and Antony Fraboulet. Effective Ahead Pipelining of Instruction Block Address Generation. In *Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, CA, USA, May 2003.
- [22] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 70–79, Dallas, TX, USA, November 1998.
- [23] Kevin Skadron, Margaret Martonosi, and Douglas W. Clark. Speculative Updates of Local and Global Branch History: A Quantitative Analysis. *Journal of Instruction Level Parallelism*, 2:1–23, January 2000.
- [24] Adam R. Talcott, Wayne Yamamoto, Mauricio J. Serrano, Roger C. Wood, and Mario Nemirovsky. The Impact of Unresolved Branches on Branch Prediction Scheme Performance. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 12–21, 1994.
- [25] Renju Thomas, Manoj Franklin, Chris Wilkerson, and Jared Stark. Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 314–323, San Diego, CA, USA, May 2003.
- [26] Tse-Yu Yeh and Yale N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 124–134, Gold Coast, Australia, May 1992.