Dynamic History-Length Fitting: A third level of adaptivity for branch prediction

Toni Juan Sanji Sanjeevan Juan J. Navarro

Depart. of Computer Architecture
Univ. Politècnica de Catalunya
08034 Barcelona (Spain)
antonioj@ac.upc.es

Abstract

Accurate branch prediction is essential for obtaining high performance in pipelined superscalar processors that execute instructions speculatively. Some of the best current predictors combine a part of the branch address with a fixed amount of global history of branch outcomes in order to make a prediction. These predictors cannot perform uniformly well across all workloads because the best amount of history to be used depends on the code, the input data and the frequency of context switches. Consequently, all predictors that use a fixed history length are therefore unable to perform up to their maximum potential.

*We introduce a method —called DHLF— that dynamically determines the optimum history length during execution, adapting to the specific requirements of any code, input data and system workload. Our proposal adds an extra level of adaptivity to two-level adaptive branch predictors. The DHLF method can be applied to any one of the predictors that combine global branch history with the branch address. We apply the DHLF method to gshare (**dhlf-gshare**) and obtain near-optimal results for all* `SPECint95` *benchmarks, with and without context switches. Some results are also presented for gskewed (**dhlf-gskewed**), confirming that other predictors can benefit from our proposal.*

1. Introduction

Branch prediction is a key performance component for wide-issue superscalar and deeply pipelined processors, where several wrong-path instructions can be in-flight before a branch is resolved. To reduce the number of lost cycles due to speculative execution of wrong-path instructions, branch prediction has evolved from static to more flexible predictors. These predictors try to adapt to dynamic program behavior in order to improve their performance. A first level of adaptivity has been the use of 2-bit saturating counters [12]. An additional level of adaptivity has been introduced using other sources of branch information such as the history of branch outcomes and the correlation between branches [16], [10], [17], [19], or even choosing among several predictors designed for different kinds of branch behavior [7], [3].

Some of the best current predictors are based on the two-level adaptive or correlated schemes proposed in [16] and [10]. They combine fixed amounts of global history and program counter (PC) bits to generate an index to one or several pattern history tables (PHT) of 2-bit saturating counters. Several parameters influence the performance of these predictors such as the size of the predictor tables, the way the branch history and PC bits are combined, the way the PHT is updated as well as the amount of history and PC information used.

Motivation

To illustrate the effect of history length on branch predictor performance, Figure 1a plots the misprediction rates for three two-level adaptive branch predictors: *gshare* [7] and the recently proposed *agree* [13] and *gskewed* [8]. The benchmarks used are `go` and `li` from `SPECint95`. The area occupied by each predictor is 8Kbits for *gshare*, and 12Kbits[1] for *agree* and *gskewed*. All three perform very much alike for `li` and even though their performance for `go` is different, all display the same kind of dependence on the history length. In Figure 1b we plot the same results as in Figure 1a, showing with shaded bars the range of misprediction rates when varying the history length for the three predictors. For all predictors, there is a significant range of variation in misprediction rate depending on

[1] *agree* has the same PHT as *gshare* (8Kbits) but has 4Kbits extra for the bias bit and *gskewed* has three banks of 2-bit saturating counters, each indexed with 11 bits. 12Kbits was the closest value greater or equal to 8Kbits that could be used.

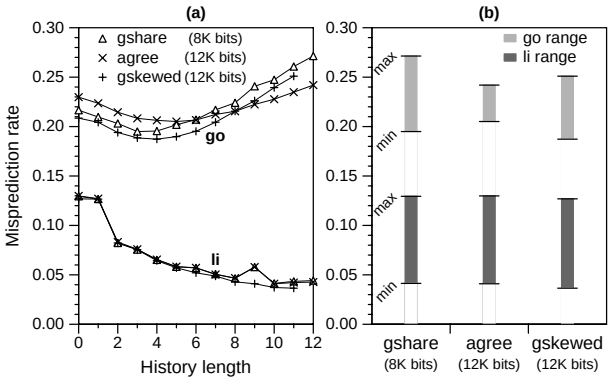

Figure 1. Misprediction rates of three different two-level predictors that use global history, using a 12-bit index (11-bit for gskewed), for go and li. a: Effect of the history length. b: Misprediction range for each benchmark and predictor.

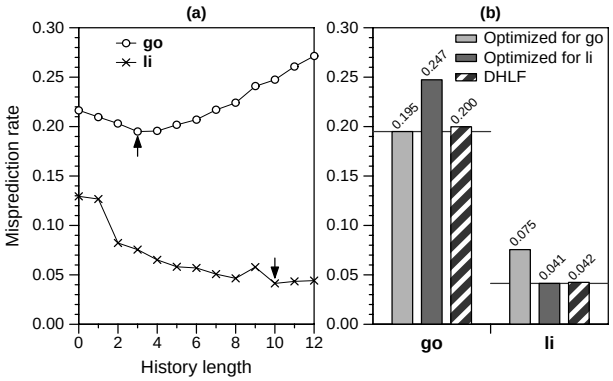

Figure 2. Misprediction rates of two SPECint95 benchmarks (go and li) using a gshare predictor with a 12-bit index. a: Effect of the history length. b: History length optimized for go, li and using our DHLF method (striped bar).

the history length (e.g *gshare* on `go` varies from 20% to 27% and from 4% to 13% for `li`). Two interesting observations can be made from the misprediction ranges of `go`: First, even though the *agree* predictor has lower variation as a function of the history length, when optimal history lengths are compared, *gshare* and *gskewed* perform better. Second, *gskewed* achieves the best misprediction rate with a history length of four bits. However, its performance for more than half of all possible history lengths is inferior to that of the best *gshare* configuration, despite the fact that it occupies 50% more area.

From Figure 1 we can conclude that

- the history length used for a particular code has a significant impact on predictor performance,
- determining the best predictor, for some codes, depends on the history length used,
- for a given predictor, achieving the best prediction accuracy requires the use of different history lengths for each code. All predictors that use a fixed history length are therefore unable to perform up to their maximum potential, and
- the performance of two-level branch predictors can be further optimized by adapting the history length to each code.

Dynamic selection of the history length

We present a new method that, when applied to existing two-level predictors, performs very close to the best case with fixed history lengths on all `SPECint95` benchmarks. Our proposal adds another level of adaptivity, trying to better fit the number of history bits needed by each benchmark and input data at execution time. This method can be applied to any member of the family of predictors that combine global branch history with PC bits to form an index to one or several PHTs such as *gshare* [7], *gselect* [10], *gskewed* [8], *agree* [13] and *bi-mode* [6]. We call this method Dynamic History-length Fitting or DHLF.

As an illustration, Figure 2a shows the same results as Figure 1a but only for *gshare*. The PHT is indexed using 12 bits of the branch address xor-ed with global history bits varied from 0 to 12. Increasing the history length improves the performance for `li`. Whereas for `go`, the misprediction rate reaches a minimum with a history length of 3 and as the number of history bits is increased, the performance rapidly degrades. The arrows indicate the optimum for each code.

Figure 2b shows what happens when *gshare* uses history lengths *optimal* for `go` (3 bits) or `li` (10 bits). In each case, optimizing for one code results in poor performance in the other. Finally, the striped bar of Figure 2b shows the performance achieved with the DHLF method applied to *gshare*, that we propose and evaluate in this paper. Note that nearly optimal results are obtained for both benchmarks with our method.

We will show that when context switches are considered, the history length becomes more critical for performance. The DHLF method is able to obtain the best results even in this environment.

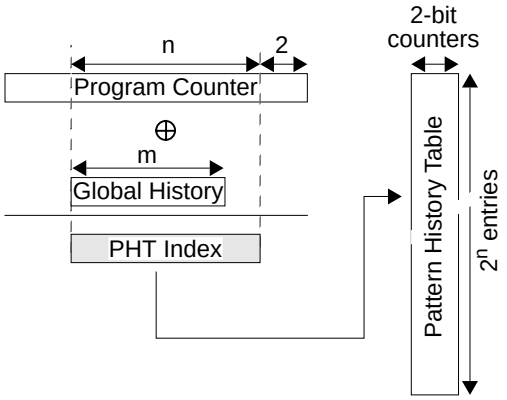

Figure 3. Detail of the gshare implementation evaluated

Simulation methodology

The simulations have been conducted using `ATOM` [2] to obtain a trace of conditional branches from all `SPECint95` benchmarks using reference inputs. The benchmarks were instrumented with `ATOM` and then executed on a DEC 21164 workstation running Digital UNIX V4.0A. We first looked at the results of simulating 10, 100, 200, 500 and 1000 million dynamic conditional branches for several benchmarks. Since the results were stable after 100 million, we carried out all our simulations up to 200 million conditional branches. Throughout the paper we will use the term 'branches' to refer to 'conditional branches'.

In all branch predictors simulated, the branch history register and the PHTs are immediately updated with the true outcome of the branch instead of using the predicted outcome or waiting for the outcome to be known —it has been shown in [18] that this has little overall effect on prediction accuracy.

The study is carried out with *gshare* as described in [7]. Figure 3 shows how the *m* bits of global history are xor-ed with the *m* higher-order bits of the *n* low-order bits of the PC (after discarding the 2 lowest bits) to generate the index into the PHT. The PHT consists of 2^n two-bit saturating counters initialized to saturated taken.

Due to lack of space, we exhaustively study the effect of different parameters on predictor performance using only two of the `SPECint95` benchmarks —`go` and `li`. These two benchmarks were chosen since they represent two distinct types of variation in performance with changing history lengths. After analizing these results, we fix some of the parameters and present the results for the rest of the benchmarks. More detailed results for all `SPECint95` codes are available as a technical report [5].

Paper organization

In section 2, we study the effect of the history length on misprediction rates for *gshare* as a function of predictor size in the absence of context switches. Section 3 describes our DHLF method and evaluates it's performance. The effect of context switches on predictor performance is studied and evaluated in section 4. In section 5 we apply our method to *gskewed* and present some results. Section 6 discusses related work and section 7 provides some concluding remarks.

2. Effect of history length on predictor performance

To understand the effect of history length on prediction accuracy, we simulated *gshare* on all `SPECint95` benchmarks for PHTs indexed with 10, 12, 14 and 16 bits and history lengths ranging from 0 to the number of index bits. The misprediction rate on each benchmark is presented in Figure 4. Each curve represents a particular predictor size. All graphs are plotted within a window ranging from 0% to 20% of misprediction rate, except for `go` that is plotted from 12% to 32%. Since for all plots the size of the misprediction range shown is the same, the differences in misprediction rates between any curve can be directly compared across the benchmarks.

From Figure 4 we can identify three different behaviors observing how the code predictability evolves when the history length is increased: the predictability for *compress* and *li* improves as more history is used. *ijpeg*, *m88ksim* and especially *perl* show some irregular behavior for different history lengths. Finally, prediction accuracy for *gcc*, *go* and *vortex* improves with more history bits but quickly starts to degrade. These different behaviors depend on the number of static branches that account for most of the dynamic branches, the degree of correlation between branches and the predictor size.

The range of predictor sizes studied —up to 128K bits— covers the predictor sizes for present and near future processors: from the MIPS R10000 that has 512 entries of 2-bit counters (1Kbit) indexed only with the PC [15] up to the recently announced DEC AXP 21264 that will have a hybrid predictor with an estimated area of 35Kbits [4]. To the best of our knowledge, except for the 21264, all processors use the equivalent of 16Kbits or less area for their predictors.

When the index size of the predictor reaches 16 bits (PHT of 128Kbits), the optimal results for all benchmarks are achieved with history lengths close to the maximum. However, this requires having very big predictors, four times larger than the biggest one announced at present (DEC AXP 21264). In section 4 we show that in a more realistic environment, where context switches happen quite

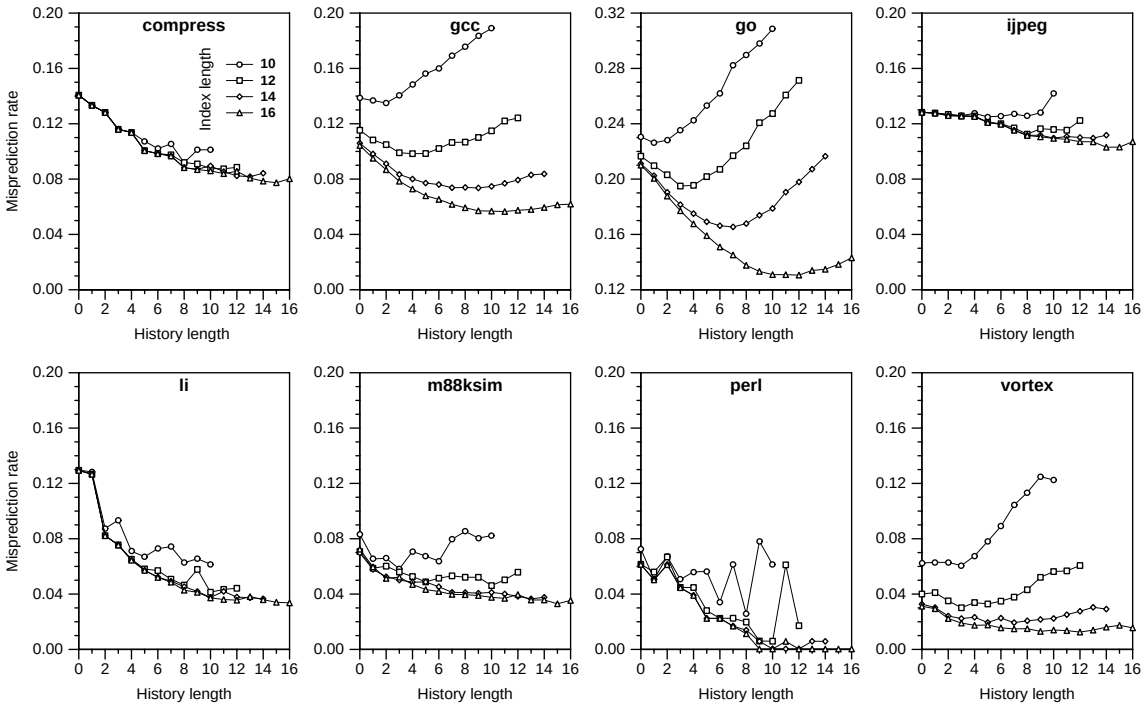

Figure 4. Effect of history length on the misprediction rate for all SPECint95 benchmarks using a gshare predictor indexed with 10, 12, 14 and 16 bits.

often and most of the PHT information is lost frequently, even the big predictors behave similar to the small and medium sized predictors presented in this section.

3. Dynamic history-length fitting

In the previous section it has been shown that each code requires a specific amount of history to give the best results. All known implementations of two-level dynamic predictors that combine global history and PC bits have a fixed amount of history length. Consequently, none of these predictors can give the best results across all benchmarks.

We propose and evaluate a new implementation of *gshare* called *dhlf-gshare* that, instead of always xor-ing a fixed number of history bits, is able to xor any number of history bits with the PC bits of the branch instruction. This predictor will try to dynamically find the amount of history that performs best for each code and input data at execution time. It will do this by using the best history length required for different phases of the code execution.

The extra hardware required to select the number of history bits to be xor-ed with the PC is very small. The Branch History Register (BHR) size has to be equal to the maximum history length envisaged. The number of entries of the PHT desired determines the number of PC bits that have to be used by the predictor ($\log_2[PHTentries]$). All bits of the BHR are xor-ed with the PC but a decoder and a set of two bit multiplexers select the desired number of bits of the BHR that hold the past history information. Figure 5a depicts a normal BHR at the bit level and Figure 5b shows the modifications required to select any number of consecutive BHR bits (from 0 to all of them). This additional logic is used in parallel with the xor and therefore does not introduce any extra delay in the index generation.

DHLF works on the basis of monitoring the mispredictions during program execution and changing the history length accordingly. We define an 'interval' to consist of a fixed number of consecutive dynamic branches. We call this number *step*. During the execution of the program the misprediction for each interval is computed using a fixed history length. At the end of each interval the history length to be used for the next interval is determined based on the current number of mispredictions and the minimum value encountered so far.

3.1. Structure and operation

The components of the DHLF control consist of:

- A misprediction table with as many entries as the number of bits of index to the PHT. Entry *n* holds the

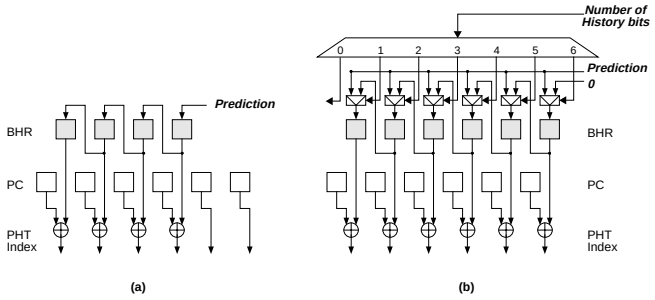

Figure 5. Hardware at the bit level to generate a 6-bit index to a PHT. a: for gshare with a fixed history length of 4 bits. b: for dhlf-gshare, 6 bits of PC can be xor-ed with any number of history bits between 0 and 6.

number of mispredictions that occured the last interval when a history length of *n* was used.

- A pointer to the table entry that corresponds to the number of history bits currently in use.
- A pointer to the table entry that contains the minimum misprediction count.
- A misprediction counter that counts the mispredictions for the current interval.
- A branch counter that counts the number of predicted branches in the current interval. When this counter reaches a value of *step* it indicates the end of an interval.

When the program starts execution, all entries of the misprediction table, the misprediction counter and the branch counter are initialized to zero. The number of history bits to be used is also set to zero.

During each interval of *step* branches, the history length remains fixed in order to determine the number of mispredictions for the current phase of execution with the current history length.

At the end of the interval the current number of mispredictions is stored in its associated entry in the misprediction table and compared with the minimum number recorded in there. If the current number of mispredictions is less than or equal to the minimum in the misprediction table, the history length is not changed for the next interval. If it is greater, then the history length is changed. Finally, the misprediction and branch counters are reset before starting a new interval.

Changing the history length could be done in different ways. One way would be to directly set the history length to the one corresponding to the minimum in the misprediction table. Another possibility would be to move towards it, increasing or decreasing by one the current history length. The latter option has been chosen because it enables the testing of history lengths in-between that may not have been tried for some time and might even yield lower mispredictions.

Updating the pointer to the entry of the misprediction table that contains the minimum number of mispredictions can be done easily. During each interval all entries of the misprediction table remain unchanged. At smaller periods (each *step/index-bits* for instance) the control can test one of the entries so that when *step* branches have completed, the entry of the misprediction table that has the minimum is already known.

Note that all possible history lengths will be tested at least once because all table entries are initialized to zero.

Each time the history length is changed, the index value generated for a given branch PC xor-ed with the same pattern of history bits changes. This means that a different entry of the PHT will be used to make the prediction, introducing aliases in the PHT. For this reason, when the history length is changed, most of the state in the PHT is lost and must be regenerated before reaching a stable state.

The increased amount of aliasing in the PHT immediately after a history length change introduces extra mispredictions that would corrupt the true performance of the current history length. The solution would be to allow some adequate warm-up time before starting to count the mispredictions for the current history length. After testing various values we chose this warm-up time to be equal to that of an interval, *step*. Consequently, the control treats the interval immediately after a change in history length in a special way. During this interval, the misprediction counter and table are not updated. At the end of this interval no comparison is made between the current number of mispredictions and the minimum value in the misprediction table. Then, a normal interval begins with the same history length.

Figure 6 shows how the history length evolves over time during the execution of `go` and `li`, using *dhlf-gshare*. The PHT is indexed with 12 bits and the *step* value is 16K. For both benchmarks, there are several history length changes during the initial phase of execution. This stabilizes around the optimal static history length during the latter phase of execution. On the right of this figure is a histogram that shows the percentage of branches predicted at each history length.

3.2. Tradeoffs

The *step* parameter controls the number of branches between possible history length changes. Using a small *step* value allows the DHLF to react to small changes in the pre-

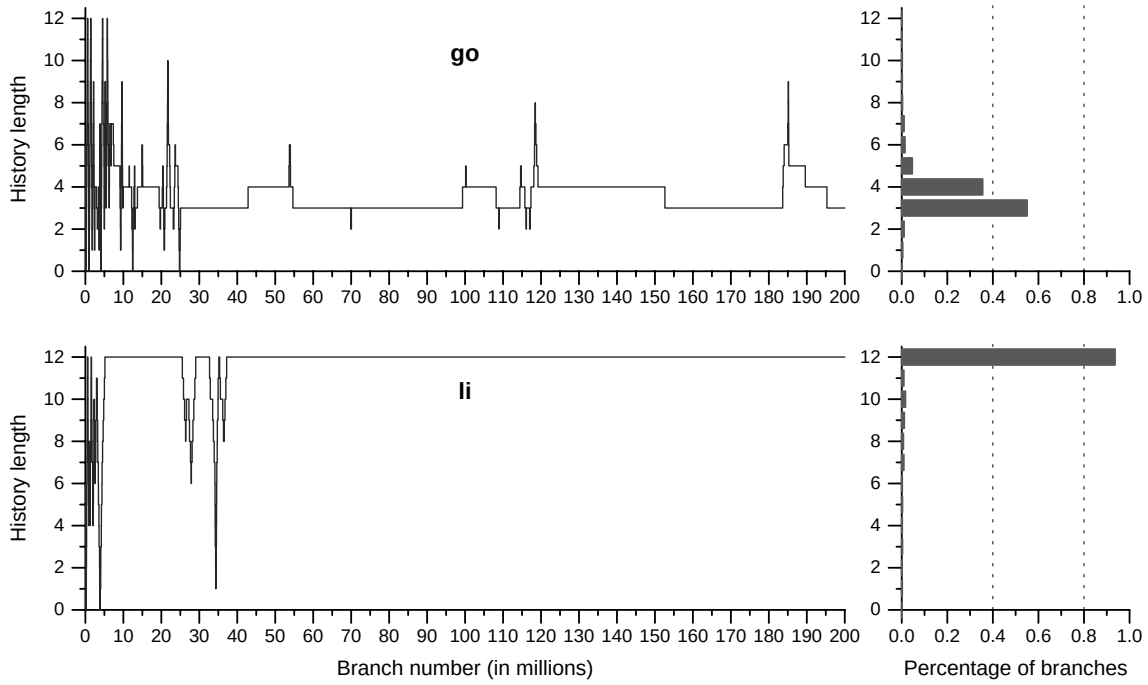

Figure 6. Evolution of the history length during the execution of go and li when DHLF is applied to gshare with an index of 12 bits and a step value of 16K. Also shown is the percentage of total time spent at each possible history length.

dictability of the code. In case the program has several phases that require the predictor to adapt to differing requirements, it could even behave better than *gshare* with the best fixed history length. Moreover, with a small *step* value the optimum history length can be determined faster. However, the *step* value has to be big enough to be able to count the mispredictions within a representative part of the code. Large PHTs require a large number of updates to reach a stable state. This implies that for larger predictor tables, larger *step* values will perform better.

The extra mispredictions due to a history-length change can reduce the benefits of DHLF. Therefore, the DHLF control has to test all history lengths as many times as possible to find the best one, but at the same time, as few times as possible so that the extra mispredictions are minimized.

It is possible that the control algorithm described in subsection 3.1 could lead to stagnation at a local optimum. In order to avoid this, we have added to the control the ability to move randomly to any history-length value when the history length hasn't changed for a large number of *steps*. However, in case it was at the global optimum, the end-result would be extra mispredictions due to the history length changes. For the simulated benchmarks this did not alter the results.

3.3. DHLF area requirements

The DHLF control mechanism increases the area required to implement *gshare*. For an index of 10 bits and a *step* value of 16K, it requires a misprediction table with 11 entries that are each 13 bits wide (assuming that the misprediction rate will be lower than 50%). This works out to $11 * 13 = 143$ bits extra, which is less than a 7% increase in area. For the case of an index of 16 bits, the extra area will be $17 * 13 = 221$ bits, an increase in predictor area of less than 0.02%.

For small PHTs (e.g 10 bits of index) the area required for the control can be minimized by storing the number of mispredictions divided by a power of two. We tested this by storing values divided by 2, 4, 8 up to 64, and there were no significant differences in the results.

Another way to reduce the area required to implement DHLF would be to reduce the number of history lengths allowed. For example, by using only even history lengths the number of entries in the misprediction table is halved with little effect on DHLF performance.

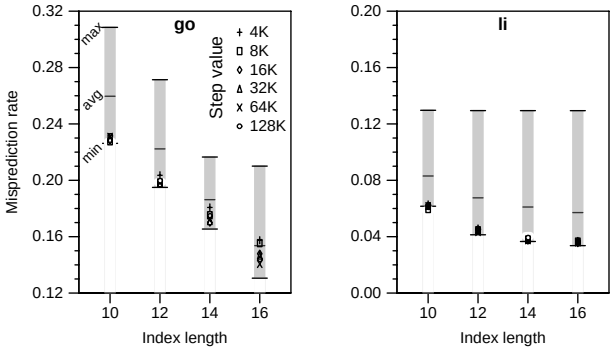

Figure 7. Misprediction rate of dhlf-gshare for several step values compared to the misprediction range of gshare with fixed history lengths. The results are presented for go and li with 10, 12, 14 and 16 index bits.

3.4. DHLF evaluation

The parameter that has to be determined for the DHLF control is the *step* value. In Figure 7 we have plotted the results of *dhlf-gshare* for *step* values starting at 4K up to 128K applied to `go` and `li`. The shaded bars represent the range of misprediction values obtained using only *gshare* with fixed history lengths. The maximum and minimum misprediction rates are obtained by running the simulation each time with a different history length, up to the number of index bits (see Figure 4). Also marked are the arithmetic mean values for the range of misprediction rates. On the shaded bars we have superimposed the misprediction rates obtained using DHLF for different *step* values. From Figure 7 it can be seen that the *step* value has no impact for all predictor sizes for `li`. In the case of `go`, as the predictor size increases, the effect of the *step* value on performance is more noticeable. In general, large predictors will need a longer warm-up time after a history length change and hence will benefit from larger *step* values. In order to simplify the presentation we selected the *step* value to be 16K for the rest of the simulations even though this is not the best choice for large predictors.

The DHLF method can effectively overcome the significant dependence on history length that prevents current predictors from achieving the best performance. DHLF introduces a new parameter, *step*. As shown, the chosen value for *step* —above a certain threshold— does not affect the overall performance.

In Figure 8, we plot the *dhlf-gshare* results for all `SPECint95` benchmarks, using a *step* value of 16K. As in Figure 7, we have superimposed the *dhlf-gshare* misprediction rate over the range computed for a *gshare* predictor. In almost all benchmarks, *dhlf-gshare* obtains near-optimal results in comparison to using *gshare* with the best fixed history length for each benchmark (labeled min). The heuristic search for the best history length and the extra mispredictions due to history-length changes prevents achieving the optimal performance in some cases[2]. In two particular cases (`compress` with a 16-bit index and `m88ksim` with a 12-bit index) the performance is even better than for any fixed length *gshare*. This confirms our intuition that in some cases, the DHLF method can respond better to the history length requirements of different phases of the execution of the same benchmark. Only `perl` exhibits irregular behavior and optimal results are obtained for just one predictor size (14-bit index). The reason for this could be the non-uniform variance of history length requirements of this benchmark as shown by the spikes in Figure 4.

4. Considering context switches

In real-world computing environments, context switches occur due to end of quantum, I/O, etc. It has been shown that the performance of even very accurate branch predictors degrades considerably when context switches are considered [3], [9]. The main reason for this is that the information maintained by the PHT is lost periodically with every context switch. Large PHTs and long history lengths require longer warm-up times because more PHT entries are indexed by each static branch. Predictors with shorter warm-up times will have a higher prediction accuracy immediately after a context switch.

Context switches introduce an additional restriction to these predictors that use fixed amounts of history. The best history length for a given code can also change depending on the frequency of context switches. Since the frequency of context switches depends on many unknown factors, such as the load of the system at a given time, it would be impossible to design a single predictor that uses a fixed history length and that always achieves optimum performance for even one benchmark. This highlights the importance of having the flexibility to dynamically change the history length of predictors. The results presented in the next section show that the DHLF method is able to find the best history length independent of the code, input data and context switch frequency for all predictor sizes.

4.1. Simulation methodology

To simulate the effect of context switches we flush the contents of the PHT each time a context switch occurs as in [3], reinitializing the PHT entries to saturated taken. We

[2] In [5] we present one way to reduce the mispredictions due to history length changes that we call *reverse-gshare*. It is also possible to improve the performance by utilizing better search algorithms.

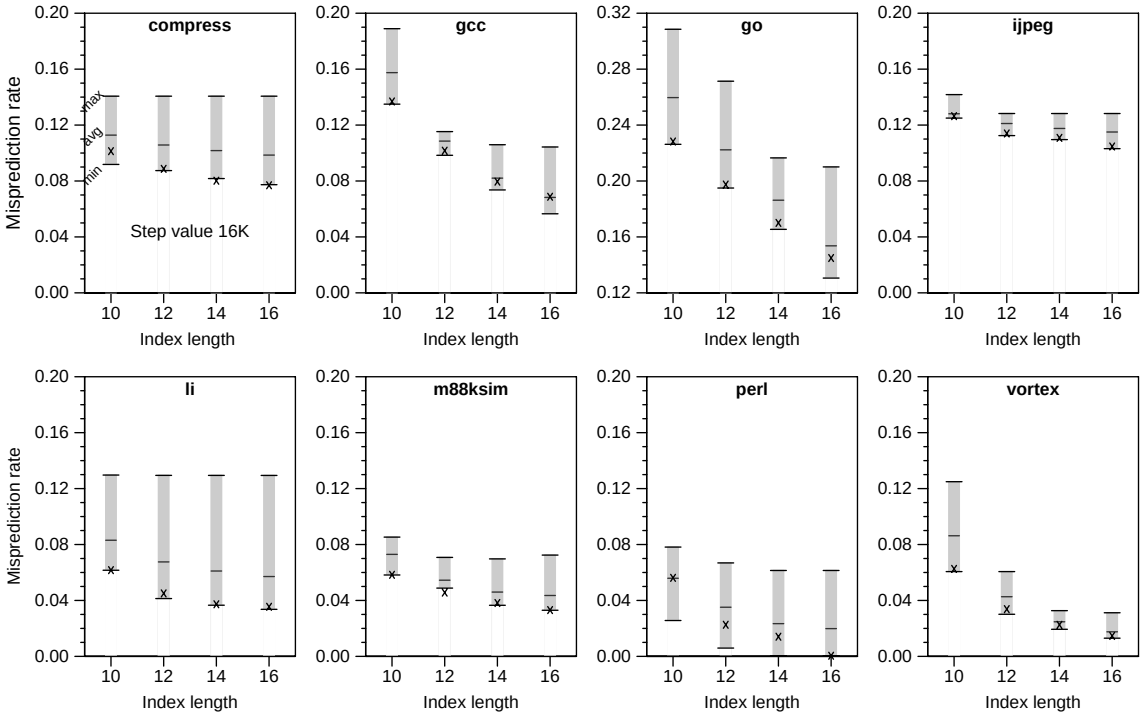

Figure 8. Misprediction rate of all SPECint95 codes using dhlf-gshare with a step value of 16K compared to the misprediction range of gshare with fixed history lengths (no context switches).

also tested reinitializing the PHT entries with random values and the results were very similar.

We define *context-switch distance* as the number of branches executed between context switches. The context-switch distance depends on the percentage of branches in the code, the system configuration and its load at execution time. It could even change during the execution of a code under real conditions. However, to simplify the evaluation we study the effect of fixed context-switch distances, from 8K up to 256K. For `SPECint95` codes this translates to between 40K and more than 3000K instructions between context switches (we have found that from 5% up to 12% of the instructions in the dynamic instruction stream are conditional branches). The context-switch distances selected are similar to those used in [3].

4.2. Effect of history length on predictor performance

Figure 9 shows the effect of context-switch distance on the misprediction rate for `go` and `li`, using the *gshare* predictor. The PHT is indexed with 16 bits while the history length varies statically from 0 up to 16 bits. Each curve corresponds to a different context-switch distance, ranging from 8K up to 256K. The gray curve shows the performance for the same predictor when context switches are not considered. The curves show the same dependence on the context-switch distance for all history lengths. As expected, the worst performance is obtained when the context-switch distance is the lowest.

We have plotted Figure 9 for the case of a PHT indexed with 16 bits to highlight the effect of context switches for big predictors. The arrows indicate the best history length for each context-switch distance. The best history length for `go` with the same input data varies from 2 up to 12 bits depending on the context-switch distance. Moreover, there is a large variation in performance depending on the context-switch distance and the history length (14% misprediction rate for `go` with no context switches compared to 36% for the same history length and a context switch each 8K branches —the latter data point is off the scale).

In Figure 10 we present the effect of context switches on misprediction rates for all `SPECint95` benchmarks using *gshare*. As before, the index lengths studied are 10, 12, 14 and 16 bits and the history lengths are varied from 0 to the number of index bits for each curve. We fix the context-switch distance at 64K in order to simplify our presentation since it is an intermediate value. Unlike in Figure 4 (the non context-switch case) `gcc`, `go` and `vortex` now show the same behavior for all predictor sizes. As before, all codes

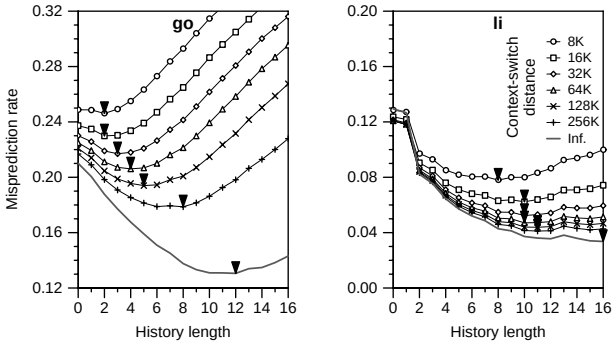

Figure 9. Effect of history length on the misprediction rate of go and li using a gshare predictor indexed with 16 bits. The number of conditional branches between context switches varies from 8K up to 256K. Also shown is the misprediction rate when there are no context switches (labeled inf.).

have differing history length requirements but now this aspect holds for all predictor sizes studied. It can also be seen that for some codes, increasing the predictor size brings no improvement because of the overhead introduced by context switches.

4.3. DHLF operation under context switching

The DHLF method applied to the case where there are context switches retains the same predictor control described in section 3. There are a few extra items to consider in order to allow *dhlf-gshare* to find the best history length across context switches

- the current value in the misprediction counter has to be discarded when a context switch occurs in the middle of an interval and is not stored in the misprediction table.
- The misprediction table and the current history length must be saved each time a context switch occurs. This means saving from 143 bits for a 10-bit index up to 221 bits for a 16-bit index, assuming a *step* value of 16K that requires 13 bits[3] for the misprediction counter (i.e, the equivalent of saving four 64-bit registers).
- Additionally, the first *step* branches after a context switch will not be considered to avoid the effect of the PHT reconstruction, as is done immediately after a history length change.

[3]assuming that the misprediction rate will always be lower than 50%

4.4. Evaluation

Figure 11 shows the performance of *dhlf-gshare* for all `SPECint95` codes using 10, 12, 14 and 16 index bits. The history length was dynamically adjusted every 16K branches as in previous figures. Once more, we have superimposed the *dhlf-gshare* results over the range of values obtained for *gshare*. The context-switch distance used was 70K because the behavior is almost the same as for 64K but is not a multiple of 16K, the *step* value used by *dhlf-gshare*. This allows us to test *dhlf-gshare* under negative conditions and shows that having a context switch within an interval does not reduce the performance of the method.

From Figure 11 we see that near-optimal results are obtained for all benchmarks except for `perl`. It should be noted that *dhlf-gshare* achieves the best performance in most codes for all predictor sizes considered with the same *step* value.

5. Applicability to other predictors

The DHLF method can be applied to any predictor that combines history and PC bits, even to hybrid predictors [7],[3]. As an example we present some results for one of the latest predictors, *gskewed*. The skewed branch predictor uses an odd number of PHTs and indexes each PHT using a different and independent hash function. All hash functions are computed from the same vector of PC and global history information. Predictions are read from each PHT and a majority vote decides the final outcome. Details about the predictor and the hash functions can be found in [8].

We evaluated *gskewed* with three PHTs and a partial update policy on `go` and `li`. Since *gskewed* requires three tables we have studied the performance for indices of 9, 11, 13 and 15 bits. This represents predictor sizes from 3K bits up to 192K bits, similar to the sizes used for *gshare* in previous sections.

The shaded bars in Figure 12 show the range of misprediction rates for *gskewed* depending on the history length used to calculate the indices into the PHTs in the absence of context switches. As we saw in previous sections for *gshare*, *gskewed* also exhibits a significant range of misprediction values for a given predictor size due to the effect of history length. We have superimposed the *dhlf-gskewed* results with a *step* value of 16K branches over the *gskewed* range. For `li` *dhlf-gskewed* always achieves the optimal performance for all predictor sizes. Applied to `go`, *dhlf-gskewed* achieves near-optimal performance with small and medium sized predictors. For bigger predictors the performance obtained with *dhlf-gskewed* is quite good but not optimal. This is mainly because we use the same *step* value for all predictor sizes. Better accuracy is obtained for big pre-

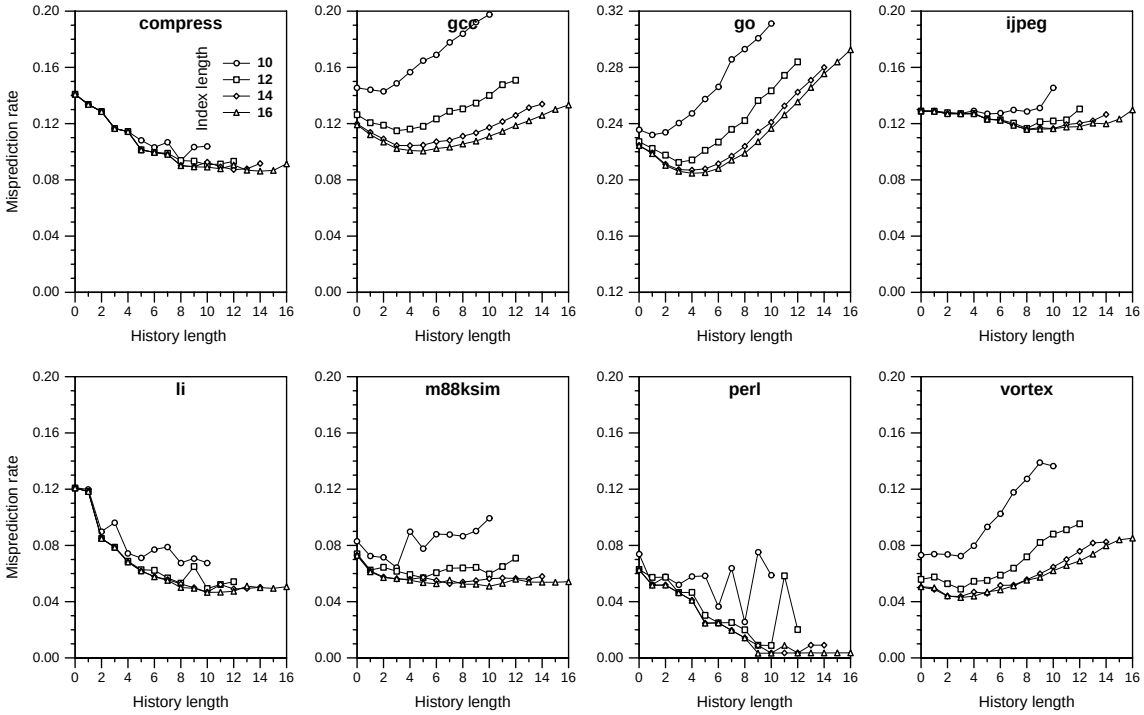

Figure 10. Effect of history length on the misprediction rates of all SPECint95 benchmarks using a gshare predictor with 10, 12, 14 and 16 bits of index and context switches occurring every 64K conditional branches.

dictors by using a larger *step* value —such as 32K or even 64K. In [5] the results for all `SPECint95` benchmarks for different *step* values are presented.

Figure 13 is similar to Figure 12 with the exception that we simulate context switches every 70K branches. As we pointed out before, context switching has a bigger negative impact on large predictors. Note that when context switches are simulated, *dhlf-gskewed* performs near-optimally for all predictor sizes in both benchmarks.

6. Related work

Several studies have looked at the effect of using different history lengths. Our DHLF proposal tries to find the best history length to be used dynamically at execution time. To the best of our knowledge, only two proposals [1] and [14] have tried to adjust the amount of history used.

- In [1] the static branches are classified depending on the bias of their behavior. The highly biased branches require a few bits of history whereas the less biased branches require a large history length. They propose using a hybrid predictor with two components, one with a few history bits and another with a large number. Only two possible values for history length are considered and further, these values are fixed. Their alternate proposal consists of profile-guided static prediction for highly-biased branches and dynamic prediction for the rest.

- The study in [14] extends the work of [1] and tries to determine the exact history length for each static branch at compile time.

These proposals are different from our proposal in significant ways: Since they focus on each specific branch instruction both studies require a complex profiling step to determine the amount of history to be used for each static branch. Further, both proposals would require modifications to the instruction set to be able to use the information gathered in the profiling phase at execution time. Finally, neither one of these proposals will be able to react to different input data or system workloads.

Other articles such as [7], [11], [8], [6], have studied the effect of history length to find the best static combination of PC and history or to better understand the behavior of their predictors.

In general, almost all studies based on two-level adaptive branch predictors assume that the final implementation

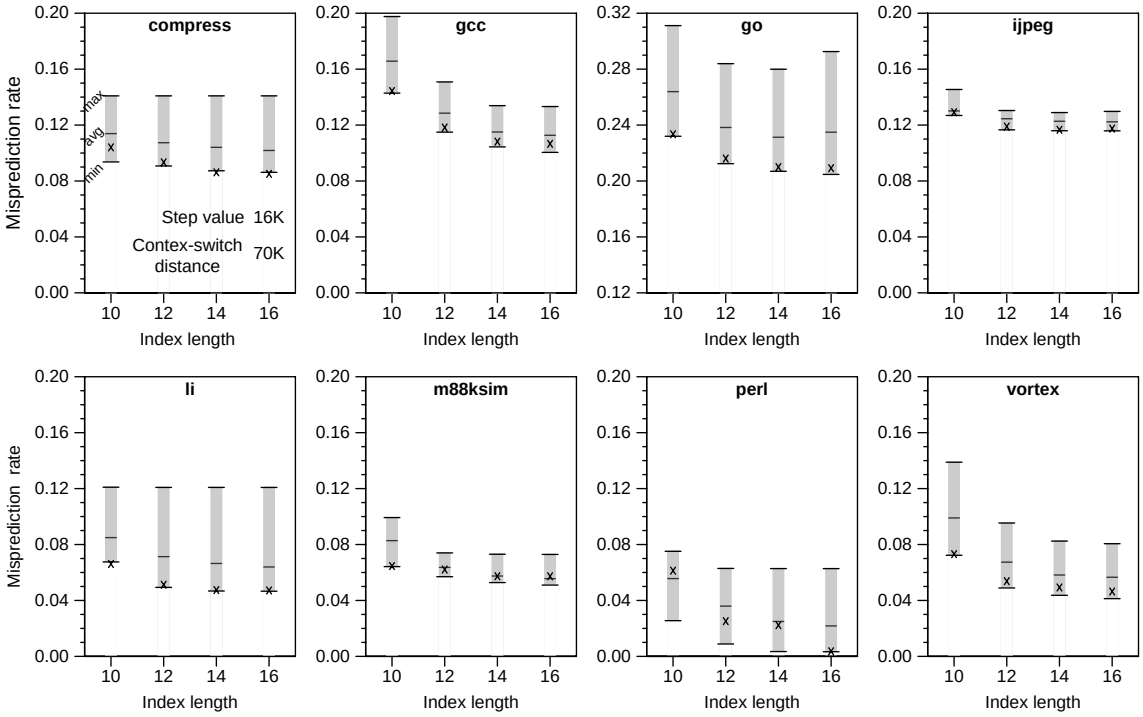

Figure 11. Misprediction rate of all SPECint95 codes using dhlf-gshare with a step of 16K compared to the misprediction range of gshare with all possible fixed history lengths. The PHT is flushed every 70K conditional branches to account for the effect of context switches.

will have a fixed number of history bits and usually select as many history bits as index bits. As we have seen in Figures 4 and 10 this can be the worst case for benchmarks such as `gcc`, `go` or `vortex`.

7. Summary

Almost all recently proposed predictors combine, in a fixed way, information from the branch address with the history of the branch outcomes to predict the direction of conditional branches. We have shown that the performance of this type of predictors, for different codes, displays significant variations depending on the history length used. These predictors that combine PC and history in a fixed way are losing a large part of their potential performance because the best history length depends on several factors that are only known at execution time. Some of these factors are the code to be executed, the input data and the number of conditional branches that can be executed between context switches in time-shared environments.

Before choosing one or another predictor it is more important to use the best history length for each code for any given predictor. We have presented DHLF, a method that finds the best history length for a given code at execution time, with and without context switches. The evaluation of DHLF has been carried out by applying it to *gshare* (*dhlf-gshare*). All `SPECint95` codes were run with this new predictor and the results confirm that it is able to achieve performances very close to optimum, compared to the best fixed history *gshare* configuration, for each code. DHLF can be applied to any predictor that combines global history with PC bits. As an example we show a few results with *dhlf-gskewed* where DHLF also obtains near-optimal performance for each specific benchmark.

DHLF has low area cost, does not affect the predictor critical path and does not require profiling nor instruction set modification.

We believe that using DHLF on any one of the two-level branch predictors will yield better prediction accuracies across a variety of codes, input data and context switch frequencies.

Acknowledgements

We would like to thank Jose Gonzalez and Roger Espasa who gave us insightful comments on drafts of this paper. We wish to also thank the anonymous referees for their valuable comments on the paper.

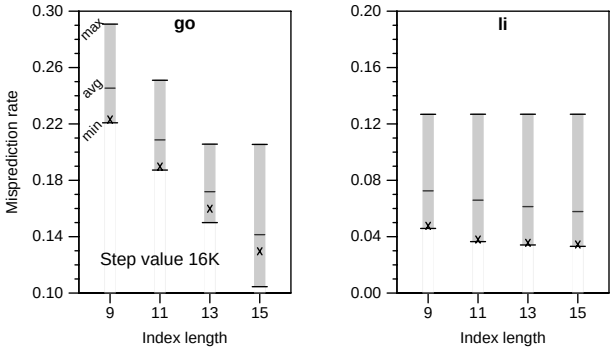

Figure 12. Misprediction rate of go and li using dhlf-gskewed with a step value of 16K compared to the misprediction range of gskewed with all possible fixed history lengths.

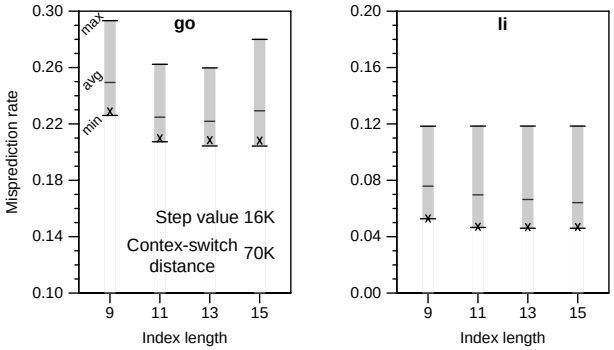

Figure 13. Misprediction rate of go and li using dhlf-gskewed with a step value of 16K compared to the misprediction range of gskewed with all possible fixed history lengths. The PHT is flushed each 70K conditional branches to account for the effect of context switches.

This work was supported by the Ministry of Education of Spain under contract CICYT TIC-0429/95 and by the CEPBA.

References

[1] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *27th Int. Symp. on Microarchitecture*, pages 22–31, Nov. 1994.

[2] A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the Winter 1995 USENIX Conference*, pages 303–314, Jan. 1995.

[3] M. Evers, P.-Y. Chang, and Y. N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *23d Annual Int. Symp. on Computer Architecture*, pages 3–11, May 1996.

[4] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14), Oct. 1996.

[5] T. Juan, S. Sanjeevan, and J. J. Navarro. A third level of adaptivity for branch prediction. Technical Report UPC-DAC-1998-4, Computer Architecture Department, UPC, Barcelona, March 1998.

[6] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The Bi-Mode branch predictor. In *30th Annual Int. Symp. on Microarchitecture*, Dec. 1997.

[7] S. McFarling. Combining branch predictors. Technical Note TN-36, Western Research Laboratory, DEC, June 1993.

[8] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *24th Annual Int. Symp. on Computer Architecture*, pages 292–303, June 1997.

[9] R. Nair. Dynamic path-based branch correlation. In *28th Int. Symp. on Microarchitecture*, pages 15–23, Nov. 1995.

[10] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Oct. 1992.

[11] S. Sechrest, C.-C. Lee, and T. Mudge. Correlation and aliasing in dynamic branch predictors. In *23d Annual Int. Symp. on Computer Architecture*, pages 22–32, May 1996.

[12] J. E. Smith. A study of branch prediction strategies. In *8th Annual Int. Symp. on Computer Architecture*, pages 135–148, May 1981.

[13] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *24th Annual Int. Symp. on Computer Architecture*, June 1997.

[14] M.-D. Tarlescu, K. B. Theobald, and G. R. Gao. Elastic history buffer: A low cost method to improve branch prediction accuracy. In *Proceedings of the 1997 IEEE International Conference on Computer Design*, pages 82–87, Oct. 1997.

[15] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[16] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *24th Annual Int. Symp. on Microarchitecture*, pages 51–61, Nov. 1991.

[17] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual Int. Symp. on Computer Architecture*, pages 124–134, May 1992.

[18] T.-Y. Yeh and Y. N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *25th Annual Int. Symp. on Microarchitecture*, pages 129–139, Nov. 1992.

[19] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual Int. Symp. on Computer Architecture*, pages 257–266, May 1993.