Exploring Last *n* Value Prediction

Martin Burtscher
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430
burtsche@cs.colorado.edu

Benjamin G. Zorn
Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052
zorn@microsoft.com

Abstract

Most load value predictors retain a large number of previously loaded values for making future predictions. In this paper we evaluate the trade-off between tall and slim versus short and wide predictors of the same total size, i.e., between retaining a few values for a large number of load instructions and many values for a proportionately smaller number of loads. Our results show, for example, that even modest predictors holding sixteen kilobytes of values benefit from retaining four values per load instruction when running SPECint95.

A detailed comparison of eight load value predictors on a cycle-accurate simulator of a superscalar out-of-order microprocessor shows that our implementation of a last four value predictor outperforms other predictors from the literature, often significantly. With 21kB of state, it yields a harmonic mean speedup of 12.5% with existing re-fetch misprediction recovery hardware and 13.7% with a not yet realized re-execution recovery mechanism.

1. Introduction

Due to their occasional long latency, load instructions can have a significant impact on system performance. If the gap between CPU and memory speed continues to widen, this latency will become even more detrimental. Since loads are not only among the slowest but also among the most frequently executed instructions in current high-performance microprocessors [8], improving their execution speed should significantly improve the overall performance of the processor.

Fortunately, load values are quite predictable. For instance, about half of the executed load instructions of the SPECint95 benchmark suite retrieve the same value that they did the previous time they were executed. Such behavior, which has been observed explicitly on a number of architectures, is referred to as *value locality* [3, 10].

Empirically, papers have shown that the results of most instructions are predictable [3, 9, 15]. However, of all the frequently occurring, result-generating instructions, load instructions are the most predictable [9] and incur the longest latencies. Since about every fifth executed instruction is a load, predicting only load values requires significantly fewer predictions and leaves more time to update the predictor. As a consequence, smaller and simpler predictors can be used. We therefore believe that predicting only load values may well be more cost effective than predicting the result of every instruction.

Load value predictors try to exploit the existing value locality. To decrease the number of mispredictions, the predictors usually consist not only of a *value predictor* but also of a *confidence estimator* (CE) that decides whether or not to make a prediction. The CE only allows predictions to take place if the confidence that the prediction will be correct is high. This is essential because sometimes the value predictor does not contain the right information for making a correct prediction. In such a case, it is better not to attempt a prediction because incorrect predictions incur a cycle penalty (for undoing the speculation and possibly blocking resources) whereas making no prediction does not.

CEs are similar to branch predictors in the sense that both make binary decisions (predictable or not-predictable and branch taken or not-taken, respectively). Hence, we adopt the nomenclature from the branch prediction literature to describe the CEs.

Most of the proposed load value predictors include *bimodal* [11] confidence estimators, i.e., they use saturating counters to “count” how frequently the predictor was correct in the recent past. The predictor is allowed to make predictions as long as this count is above a given threshold. Otherwise, predictions are inhibited.

In unpublished previous work [1] we adopted a different idea from the branch prediction literature to build a more accurate CE and hence a more accurate predictor. This CE is in essence a *SAg* predictor [23] that keeps a small history recording the most recent prediction outcomes (success or failure) [19]. The different history patterns each have a saturating counter associated with them to measure the confidence of the individual patterns. The confidence of the current pattern decides whether to make a value prediction or not. In this paper we explore possibilities to improve the coverage of a SAg-based load value predictor, that is, we try to increase the number of prediction attempts without decreasing the accuracy.

If load values are predicted quickly and correctly, the CPU can start processing the dependent instructions without having to wait for the memory access to complete, which potentially results in a significant performance increase. Of course, it is only known whether a prediction was correct once the true value has been retrieved from the memory, which can take many cycles. *Speculative execution* allows the CPU to continue execution with a predicted value before the prediction outcome is known [18]. Because branch prediction requires a similar mechanism, most modern microprocessors already contain the required hardware to perform this kind of speculation [3].

Unfortunately, branch misprediction recovery hardware causes all the instructions that follow a misspeculated instruction to be purged and *re-fetched*. This operation is costly and makes a high prediction accuracy paramount. Unlike branches, which invalidate the entire execution path when mispredicted, mispredicted loads only invalidate the instructions that depend on the loaded value. In fact, even the dependent instructions per se are correct, they just need to be *re-executed* with the correct input value(s) [9]. Consequently, a better recovery mechanism for load misspeculation only re-executes the instructions that depend on the mispredicted load value. Such a recovery policy is less susceptible to mispredictions and favors a higher coverage, but may be prohibitively hard to implement.

Our load value predictor's re-fetch performance is not only high but also close to its re-execute performance, making the added benefit of a re-execution core small in comparison. While other studies show larger performance differences, this is perhaps an indication that complex re-execution hardware may not be needed.

Load value predictors normally consist of an array of slots to store information about recently executed loads. It is this information that is used for making a prediction the next time a load is executed. Once a predictor contains enough slots to hold information about all the frequently executed loads, increasing the number of slots does not improve the predictor's performance much because the additional slots will at best be used for predicting infrequently executed and therefore unimportant load instructions. As an alternative, we suggest using the extra real-estate to increase the amount of information in each slot instead of the number of slots. This choice should enable the predictor to make better and/or more predictions and thus improve its performance.

Running SPECint95, quite small predictors already benefit more from storing additional information in the slots than from increasing the number of slots. For example, among 16kB predictors, the predictor with 512 slots each holding the last four loaded values performs better than the same size predictor with 1024 slots holding the last two values or with 256 slots holding the last eight values. Section 5.2.2 provides more detail.

Based on these results, we designed a last four value predictor that outperforms other predictors from the literature and reaches a harmonic mean speedup of 13.7% over SPECint95 with a re-execute misprediction recovery policy and 12.5% with a re-fetch recovery policy. Section 5.1.2 provides more results.

The remainder of this paper is organized as follows: Section 2 introduces the architecture of our last four value predictor. Section 3 presents related work. Section 4 explains the methods used. Section 5 presents the results. Section 6 concludes the paper with a summary.

2. The SAg Last *n* Value Predictor

Figure 2.1 shows the architecture of our last four value predictor. The predictor is composed of four identical components that each consist of an array of 512 slots for storing a 64-bit value and a ten-bit prediction outcome history. Furthermore, each component has an array of 1024 four or five-bit saturating counters associated with it. The prediction outcome histories together with the saturating counters represent the SAg confidence estimator. Each of the 512 lines of the predictor contains an eight-bit partial tag that is shared between the four components. In this configuration, our predictor requires 21kB of state.

Predictions are made in the following way: First, the nine least significant bits from the load instruction's program counter are used to index one of the predictor's lines (direct mapping). If the partial tag of that line does not match, no prediction is made. Otherwise, the four components each predict (in parallel) the value stored in their selected slot. At the same time, the components use their selected history to index a counter in their array of saturating counters. The four resulting counter values are then compared with each other and whichever component happens to report the highest counter value is selected to make the actual load value prediction if its counter value is also above a preset threshold.

If multiple components report the same maximum confidence, the component holding the youngest value is selected. Giving the component with the oldest value the highest priority results in slightly worse performance.

Once the outcome of a prediction is known, the predictor needs to be updated. This process is similar to making a prediction except that each component compares its predicted value with the true load value. If the two values are identical, the selected counter is incremented (unless it has already reached its maximum) and a one (indicating a success) is shifted into the selected prediction outcome history. If the two values differ, the corresponding counter is decremented by a preset penalty (but not below zero) and a zero (indicating a failure) is shifted into the prediction outcome history. Finally, the four values in the selected line are shifted over to the next component, i.e., the oldest value is lost, component four gets the value from component three etc. and component one receives the just loaded value. Thus, the first component always holds the most recently loaded value, the second component the second most recently loaded value and so on.

If accessing the two-level predictor in one cycle is not feasible, the accesses can readily be pipelined over two stages. During the first cycle, the values and histories are read and latched (and updated). In the second cycle, the counters are accessed (and updated) and the selection process takes place. Note that this works for updates because updates are performed independent of the confidence. The two-cycle access latency can be hidden if there are at least two stages between the decode and the execute stage in the processor's instruction pipeline (which is the case in most modern CPUs).

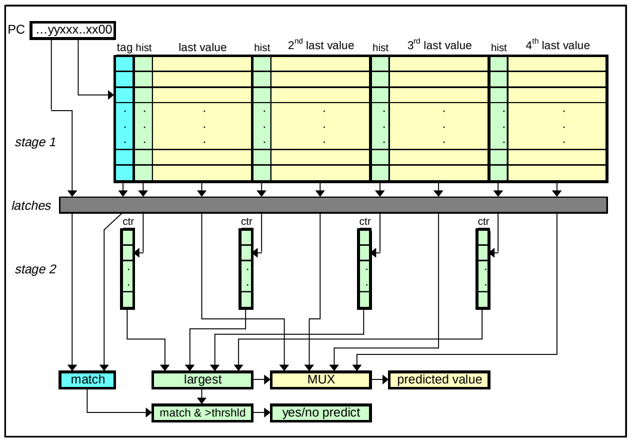

Figure 2.1: The architecture of our partially tagged SAg-based last four value predictor.

3. Related work

In this section we present related work and introduce the predictors that we later compare our predictor with. To make this comparison as fair as possible, we scale all predictors to a size of 16kB for retaining load values plus whatever else they need to support this size, since we believe 16kB to be a reasonable size for first generation load value predictors. We show the total predictor sizes (including the confidence estimators, selectors, etc.) to give an idea of the relative complexity. Note that the indicated size is in most cases not the size originally used by the authors of the individual predictors. We therefore show simulation results for predictor sizes raging from four to 64 kilobytes in Section 5.1.2.

Early Work: Two independent research efforts [3, 10] first recognized that load instructions exhibit *value locality* and concluded that there is potential for prediction.

In his dissertation proposal [3], Gabbay suggests several predictors and notes that there exists almost no (non-zero) stride predictability in load value sequences.

Lipasti et al. [10] investigate how predictable various kinds of load instructions are. In a follow-up paper, Lipasti and Shen [9] broaden their scope to predicting all result-generating instructions and show that a value predictor delivers three to four times more speedup than doubling the data-cache (same hardware increase). We found that our load value predictor outperforms doubling the L1 data-cache eightfold.

Techniques: Several research groups [9, 22] have investigated last *n* value predictability and found that there is potential for performance improvement. In this paper we present an implementable last four value predictor (Lipasti and Shen's predictor [9] is not) that outperforms Wang and Franklin's more complex predictor that retains the last four distinct values [22].

Rychlik et al. [14] and Reinman and Calder [13] propose reusing the confidence estimators in the components of a hybrid load value predictor as selector, thus eliminating the need for extra storage to guide the selection process. We utilize the same approach in our predictor.

Gabbay and Mendelson [4] use profiling to insert opcode directives that allow them to allocate only highly predictable values in their predictor, which improves the performance in most cases. We currently only investigate transparent prediction schemes that do not require any changes to the instruction set architecture.

Rychlik et al. [14] address the problem of useless predictions. They introduce a simple hardware mechanism that inhibits predictions that were never used from updating the predictor, which reduces predictor pollution and hence improves performance. Since the prediction outcome histories we use rely on seeing all updates, their scheme hurts performance when added to our predictor.

In their next paper [5], Gabbay and Mendelson show that general value prediction is more effective with high-bandwidth instruction fetch mechanisms. They argue that current processors can effectively exploit less than half of the correct value predictions since the average true data-dependence distance is greater than today's fetch-bandwidth (four). This is one of the reasons why we restrict ourselves to predicting only load values, which requires considerably smaller predictors while still reaping most of the potential.

Gonzalez and Gonzalez [6] found that the benefit of data value prediction increases as the size of the instruction window grows, indicating that value prediction will likely play an important role in future processors.

A detailed study by Sazeides and Smith [17] illustrates that most of the predictability originates in the program control structure and immediate values. Another interesting result of their work is that over half of the mispredicted branches actually have predictable input values, implying that a side effect of value prediction should be improved branch prediction accuracy. Gonzalez and Gonzalez [6] did indeed observe such an improvement.

Predictors: The predictor most closely related to our own is Wang and Franklin's last distinct four value predictor (*LD4V*) [22]. It retains only distinct values (ours does not) and uses a least recently used replacement policy. Instead of prediction outcome histories, their predictor uses the pattern of the last six accesses as an index into four arrays of saturating counters that correspond to the four components. The maximum counter value determines which component is selected to make a prediction. Predictions only take place if the counter value is above a preset threshold. A *LD4V* storing 16kB of load values requires about 26kB of state.

To improve the performance of the *LD4V* predictor, Wang and Franklin propose hybridizing their *LD4V* with a stride predictor [22]. We call the resulting predictor *LD4V+Stride*. A stride predictor predicts a value that is the sum of the last value plus an offset (stride). This offset is computed as the difference between the last value and the second to last value. Since the stride component only stores 8-bit partial strides that are added to the values from the *LD4V* component, the hybrid predictor is not significantly larger than *LD4V* and requires 27kB of state. Wang and Franklin did not perform cycle-accurate simulations of their predictors.

In previous work we have developed a tagged SAg-based last value predictor (*Tag SAg LV*) [1] with a size of 21kB. The last four value predictor presented in this paper essentially consists of four *Tag SAg LV* components one quarter as high that are operated in parallel. We found that replicating the components and making them shorter to maintain the overall predictor size results in improved performance for predictors with more than about eight kilobytes of state.

The predictor most closely related to our *Tag SAg LV* is Lipasti and Shen's bimodal last value predictor (*Bim LV*) [9]. It is untagged and uses two-bit saturating counters as a confidence estimator. The counters are located in the place where our predictor keeps the histories. A value prediction only takes place if the corresponding counter value is above a given threshold. *Bim LV* is the smallest predictor with a size of 17kB.

We expand on *Bim LV* by adding partial tags and performing a detailed parameter space analysis to find the optimal CE setting. As it turns out, three-bit counters with a penalty (decrement) above one perform the best. Because of the somewhat larger counters and the 8-bit partial tags, the size of *Tag Bim LV* amounts to 19kB.

We performed another detailed analysis for the *Tag Bim St2d* predictor, which is a tagged stride 2-delta [15] predictor with a bimodal CE. A stride 2-delta predictor retains two strides. The stride used to compute the next prediction is only updated if a new stride has been seen at least twice in a row. This results in significantly better performance than a conventional stride predictor. The size of the *Tag Bim St2d* predictor is 23kB due to the two partial strides.

The last predictor we compare against is *St2d+FCM*, which is similar to the one presented by Rychlik et al. [14] except it is not set-associative. The predictor is a hybrid between a stride 2-delta and a finite context method (FCM) predictor [16]. FCM predictors store entire sequences of load values. Upon prediction they try to identify the current location in the sequence and use the next value from the sequence to make a prediction. This is particularly useful for repeated traversals of dynamic data-structures. The configuration that yields the best result requires about 26kB of state with re-execute and 29kB with re-fetch, which makes it the largest predictor.

4. Methodology

All our measurements are performed on the DEC Alpha AXP architecture using the AINT simulator [12] with its cycle-accurate superscalar back-end. We configured the simulator to emulate a processor similar to the DEC Alpha 21264 [7]. In particular, the simulated 4-way superscalar out-of-order CPU has a 128-entry instruction window, a 32-entry load/store buffer, four integer and two floating point units, a 64kB 2-way set associative L1 instruction-cache, a 64kB 2-way set associative L1 data-cache, a 4MB unified direct-mapped L2 cache, a 4096-entry BTB, and a 2048-line gshare-bimodal hybrid branch predictor. The three caches have a block size of 32 bytes. The modeled latencies are shown in Table 4.1. The six functional units are fully pipelined. Operating system calls are executed but not simulated. Loads can only execute when all prior store addresses are known.

This configuration represents our baseline architecture. All the speedups reported in this paper are relative to this CPU, which does not contain a load value predictor.

We performed a detailed parameter space evaluation comprising hundreds of simulation runs to obtain the most effective configurations for the load value predictors presented in this paper.

The best performing last *n* value predictor under 30kB of state that we found is our *SAg Last 4 Value Predictor* with a height of 512, a history length of ten bits, four bit saturating counters for re-execute with a threshold of nine and a penalty (decrement) of three, and 5-bit counters for re-fetch with a threshold of sixteen and a penalty of sixteen. The predictor uses 16kB of state for storing values plus 5kB for the confidence estimator. Unless otherwise noted, these are the parameters used with our predictor.

Since we believe that next-generation CPUs will only contain moderately sized load value predictors, our study focuses on predictors under about thirty kilobytes of state. We do, however, also present results of larger and smaller predictors in Section 5.1.2.

Instruction Type	Latency
integer multiply	8-14
conditional move	2
other int and logical	1
floating point multiply	4
floating point divide	16
other floating point	4
L1 load-to-use	1
L2 load-to-use	12
Memory load-to-use	80

Table 4.1: The functional unit and memory latencies (in cycles) used for our simulations. The load-to-use latencies do not include the effective address calculation.

Note that the predictors used in this paper are optimized for speedup, which implies optimizing the predictor performance at instruction commit. The interaction between the CPU and the predictor, however, takes place during prediction and then again during update, possibly long before the time of commit. This discrepancy may be an issue because, for example, the accuracy with which wrong path instructions are predicted is most likely less important than the accuracy of correct path instructions. Consequently, a high overall accuracy measured at predict or update may not be representative of the predictor's performance since it makes no statement about the prediction accuracy of the instructions that are actually retired. We found the ratio of total predicted loads over committed value-predicted loads to be just under 1.5, indicating that there is a significant number of predictions that most likely have little impact on the overall performance. To account for any effects this might have, we model out-of-order and wrong-path updates of the predictor accurately in our simulator. Note that all the predictors used in this paper are updated as soon as the true load value becomes available, that there are no speculative updates, and that an out-of-date prediction will be made as long as there are pending updates (that are going to the same predictor line).

4.1 Benchmarks

We use the eight integer programs of the SPEC95 benchmark suite [20] for our measurements. These programs are well understood, non-synthetic, and compute-intensive, which is ideal for processor performance measurements. Despite the lack of desktop application code, the suite is nevertheless quite representative thereof [8].

We use the reference input set and the more optimized peak-versions of the programs (compiled on an Alpha 21164 using DEC GEM-CC with full optimization *-migrate -O5 -ifo*). The binaries are statically linked, which enables the linker to perform additional optimizations to further reduce the number of run-time constants that are loaded during execution. These optimizations include most of the optimizations that OM [21] performs. In spite of this high optimization level and good register allocation, 22.9% of the instructions executed by the programs are loads.

Note that the few floating point load instructions contained in the binaries are also predicted and that loads to the zero-registers as well as load immediate instructions are ignored.

We execute each of the benchmark programs for 300 million instructions on our simulator after having skipped over the initialization code in "fast-execution" mode. This fast-forwarding is very important if only part of the execution is simulated because the initialization part of programs is not usually representative of the general program behavior [13]. Table 4.2 shows the number of instructions that were skipped (in billions) and gives other relevant information about the simulated segment of each of the eight SPECint95 programs. GCC is executed for 334 million instructions and no instructions are skipped since this amounts to the complete compilation of the first reference input file.

Information about the Simulated Segments of the SPECint95 Benchmark Suite										
	exec	percent	skipped	base	L1 load	L2 load	load sites that account for			
program	instrs	loads	instrs	IPC	missrate	missrate	Q100	Q99	Q90	Q50
compress	300 M	17.9%	6.0 G	1.35	24.4%	2.8%	62	56.5%	45.2%	14.5%
gcc	334 M	23.9%	0.0 G	1.51	2.4%	6.4%	34345	41.2%	15.7%	2.5%
go	300 M	24.1%	12.0 G	1.44	1.4%	15.3%	9619	40.2%	17.9%	2.7%
ijpeg	300 M	16.8%	1.0 G	1.44	1.4%	51.3%	2757	13.7%	6.7%	1.9%
li	300 M	25.5%	4.0 G	1.99	5.4%	0.6%	419	56.6%	28.6%	10.3%
m88ksim	300 M	20.7%	1.0 G	1.25	0.1%	11.2%	747	71.9%	26.6%	3.3%
perl	300 M	31.2%	1.0 G	1.57	0.0%	46.9%	1437	15.7%	11.6%	3.1%
vortex	300 M	23.6%	5.0 G	2.89	2.2%	10.2%	1973	48.6%	18.0%	2.8%
average		22.9%		1.68	4.7%	18.1%	6420	43.0%	21.3%	5.1%

Table 4.2: This table shows, from left to right, the number of simulated instructions (in millions 'M'), the percentage of instructions that are loads, the number of skipped instructions (in billions 'G'), the instructions per cycle of the baseline CPU, the L1 data-cache load miss-rate, the L2 load miss-rate, and some quantile information. The quantile columns show the number of load sites that contribute the given percentage (e.g., Q50 = 50%) of executed loads in absolute terms for Q100 and percentages thereof for the remaining quantiles.

The results shown in Table 4.2 only take into account load instructions within the simulated segments of the benchmark programs. However, we found the eight segments to be very representative of the complete programs, as full program executions revealed. For example, the predictability over the entire programs is within five percent of the numbers measured for the simulated segments.

Except for compress, all the programs have a quite low L1 data-cache load miss-rate. However, some of the L2 load miss-rates are quite large. Since the corresponding number of accesses is very small, the large L2 miss-rates do not have a significant impact on the performance.

An interesting point is the relatively small number of load sites that contribute most of the executed load instructions. For example, 5% of the load sites that are executed at least once account for 50% of the dynamically executed loads and only 43% of the executed load sites account for 99% of the executed loads.

5. Results

The following subsections describe the results. In Section 5.1 we evaluate the performance of our *Tag SAg L4V* predictor by comparing it to oracles (Section 5.1.1) and other predictors from the literature (Section 5.1.2). Section 5.2 presents a sensitivity analysis of the predictor parameters. Section 5.2.1 investigates the prediction potential, Section 5.2.2 examines the predictor height versus width trade-off for different sizes, Section 5.2.3 studies the prediction outcome history length, and Section 5.2.4 explores the parameters of the saturating counters.

To better analyze the parameter space we only show averages over the eight benchmarks and not the individual programs. Note that all the averaged speedups presented in this paper are harmonic mean speedups. Furthermore, except for Sections 5.1.2 and 5.2.2, we restrict ourselves to predictor sizes between 16kB and 29kB since predictors of that size already perform well and are most likely not too large to be included in next generation microprocessors.

5.1 Predictor Performance

In brief, our *Tag SAg L4V* predictor's accuracy at commit is 98.1% using a re-fetch misprediction recovery policy. On average, 32.8% of the load instructions are predicted with the correct value, 0.6% with an incorrect value. This results in a harmonic mean speedup over SPECint95 of 12.5% relative to an otherwise identical CPU that does not include a load value predictor. With a re-execution architecture, the accuracy of the predicted load instructions that are committed/retired is 92.9%. 36.9% of the load instructions are correctly predicted and 0.8% are incorrectly predicted, resulting in a harmonic mean speedup of 13.7%.

5.1.1 Comparison with oracles

To get a better understanding of the performance of our predictor, we modified the simulator to provide various degrees of perfect knowledge.

The first oracle (called *perf-inh*) inhibits all the incorrect predictions that the oracle-less predictor (*normal*) would make (i.e., no incorrect predictions take place). The next oracle (*perf-ce*) incorporates a perfect confidence estimator. In addition to inhibiting all incorrect predictions, the predictor is now forced to make a prediction whenever the selected component contains the correct value. The last oracle (*perf-ce/sel*) includes both a perfect confidence estimator and a perfect selector. Hence, the oracle not only always makes a prediction if the correct value is available and never makes a prediction otherwise, but it also chooses the component that will make a correct prediction if there is such a component. In other words, if any component in the predictor can make a correct prediction, it is selected and a prediction takes place, otherwise no prediction is attempted.

Figure 5.1 shows the speedups of the oracle-less predictor and the three oracles for re-fetch and re-execute.

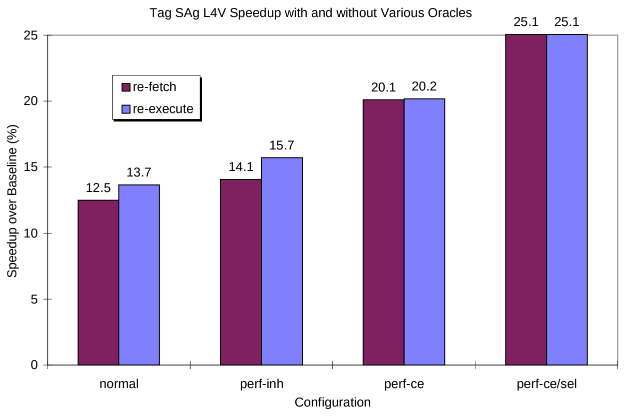

Figure 5.1: Re-fetch and re-execute speedups of various predictors with different degrees of perfect knowledge (oracles).

Inhibiting incorrect predictions (*perf-inh*) improves the speedup somewhat relative to the ordinary predictor (*normal*). The improvement is not very large, though, indicating that incorrect predictions either do not diminish the performance much or that there is only a small number of incorrect predictions to begin with. Since the predictor's accuracy is over ninety percent, the latter is the more probable explanation.

Note that the re-fetch configuration we use attempts fewer predictions than the re-execute configuration, which is why the re-fetch performance of the oracle is lower than its re-execution counterpart even though there are no incorrect predictions.

Adding perfect confidence estimation (*perf-ce*) results in a significant increase in speedup, suggesting that our imperfect CE is rather conservative. Since our CE setting is the result of a global optimization and hence yields one of the highest speedups possible, we conclude that trading off missing potentially correct predictions for reducing the number of incorrect predictions is beneficial with the CPU we are simulating. Apparently, incorrect predictions are indeed very harmful and should therefore be avoided, making a high prediction accuracy paramount.

Note that, because there is no difference in the CE setting, the *perf-ce* speedups for re-fetch and re-execute should be the same. The minor discrepancy in the two speedups stems from different timing behavior within the modeled CPU after a load value misprediction that affects the predictor updates, which are non-speculative. The reason for this is that *perf-ce* does not necessarily "correctly" predict wrong-path load instructions (that are executed due to branch mispredictions) since a correct load value is not always defined in such a case.

Adding a perfect confidence estimator and a perfect selector (*perf-ce/sel*) results in yet another big boost in speedup, implying that our selection mechanism could be improved. Overall, the confidence estimator and selector in our predictor are able to reap about half of the speedup that is theoretically achievable with this predictor.

5.1.2 Comparison with other predictors

In this section we compare several predictors from the literature with our own. The sole metric for this comparison is the harmonic mean speedup over SPECint95. To make the comparisons as fair as possible, every predictor is scaled to 16kB of state for storing values. Note that, due to the different confidence estimators, the overall predictor sizes vary between about 17kB and 29kB of state. Our predictor with 21kB is among the smaller ones.

We performed a detailed parameter space evaluation for our predictor to determine the setting that yields the highest speedup. For the predictors from the literature we use the best parameter setting indicated by the authors. Observing that the threshold has a significant effect on performance, we varied the threshold setting of all the predictors to find an optimal value. Figure 5.2 and Figure 5.3 present the resulting best mean speedup of the predictors with a re-execute and a re-fetch misprediction recovery mechanism, respectively. The predictors are sorted by size, with the smallest being on the left side. On the far right we show the speedup achieved by doubling the size of the 64kB L1 data-cache instead of adding a load value predictor.

Our predictor (*Tag SAg L4V*) outperforms all the other predictors, including larger and more complex ones. *LD4V+Stride* comes close to the speedup of our predictor when re-execution is utilized, but not with re-fetch.

While all the predictors perform quite well with the more complex but more forgiving re-execution policy, their performance suffers significantly when the currently available re-fetch misprediction recovery hardware is used. Our predictor sustains the least performance decrease. Surprisingly, its re-fetch speedup is higher than any other predictor's re-execute speedup with only one exception; *LD4V+Stride* performs somewhat better with re-execution than our *Tag SAg L4V* predictor using re-fetch. However, *LD4V+Stride* actually slows programs down with re-fetch.

Note how much better *Tag Bim LV* performs than *Bim LV*. Most of the difference in speedup does not come from the partial tags but from the more adequate choice of CE parameters. In particular, changing the counter penalty from one to seven for re-fetch made the biggest difference. Reinman and Calder performed a similar search to find the best CE parameters [13] and also determined that a large penalty is needed for a re-fetch architecture.

We suspect that other proposed predictors can also be improved upon by imposing a heavier penalty on their counters. However, we do not believe that they will reach the performance of our predictor unless they switch to a SAg-based CE, since in all our measurements with otherwise identical components, bimodal predictors always turn out to be inferior.

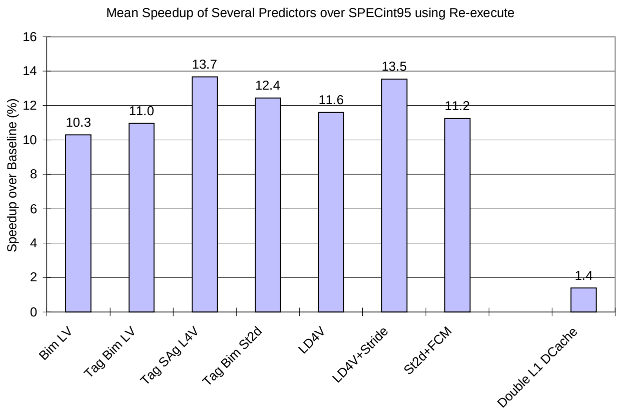

Figure 5.2: The best harmonic mean speedup of several predictors with sizes between 17kB and 27kB using a re-execution misprediction recovery mechanism.

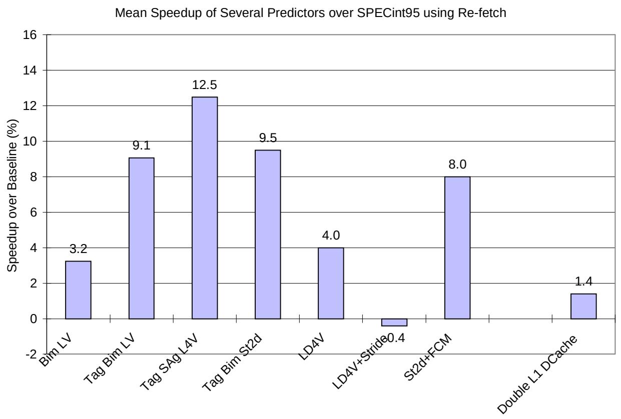

Figure 5.3: The best harmonic mean speedup of several predictors with sizes between 17kB and 29kB using a re-fetch misprediction recovery mechanism.

Since our predictor performs well with re-fetch, it is possible that no re-execution core is necessary. This result is encouraging, in particular for the near future because it means that microprocessor designers can simply use the already existing branch misprediction hardware to recover from value mispredictions.

All the predictors (except *LD4V+Stride* using re-fetch) outperform the doubled L1 data-cache. This is surprising because doubling the cache requires over 64kB of additional state, which is almost four times as much as the predictors require when all the cache hardware is accounted for. Clearly, there is a point beyond which adding a load value predictor is likely to yield more benefit than using the same number of transistors to increase the cache size.

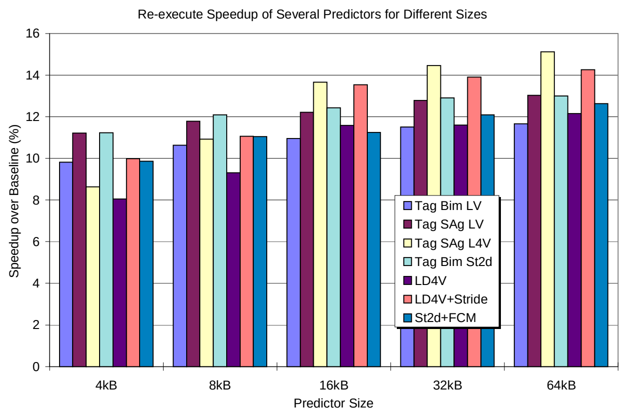

Figure 5.4: The re-execute speedup of several predictors for different sizes. The sizes refer to the amount of state used to store values and do not include the state required by the confidence estimators.

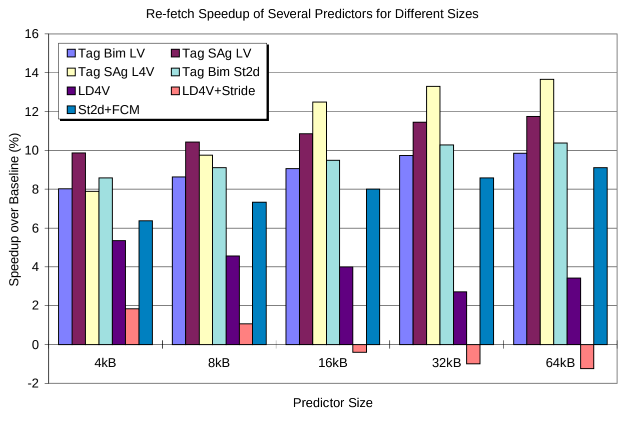

Figure 5.5: The re-fetch speedup of several predictors for different sizes. The sizes refer to the amount of state used to store values and do not include the state required by the confidence estimators.

To get a broader perspective on the performance of the various predictors, we present Figure 5.4 and Figure 5.5, which show the speedups of the predictors for different sizes. The figures no longer include *Bim LV* since *Tag Bim LV* outperforms it. We added a *Tag SAg LV* predictor in its place, which is identical to *Tag Bim LV* except it uses our SAg-based CE instead of the bimodal one.

As expected, with re-execution all the predictors perform quite well across the entire range of sizes. With 16kB and above, our last four value predictor outperforms the other predictors and the speedup gap to the second best predictor (*LD4V+Stride*) increases as the predictors become larger. Note how *Tag SAg LV* consistently delivers an additional percent of speedup over *Tag Bim LV*.

For small predictor sizes, the last four value predictor is no longer tall enough to hold all the frequently executed load instructions. As a result, its performance suffers significantly. However, with 4kB and 8kB, *Tag SAg LV* performs about as well as the best predictors for these two sizes.

With re-fetch, the performance of most predictors is significantly lower. Again, our last four value predictor outperforms the other predictors starting at 16kB.

Tag SAg LV is superior to *Tag Bim LV* also with re-fetch. We take this as strong evidence that SAg CEs are better suited for load value prediction than bimodal CEs. Note, however, that SAg CEs require more state and are more complex than their bimodal counterparts.

For the smallest two predictor sizes (4kB and 8kB), the last four value predictor is again too short and its performance is accordingly low. However, *Tag SAg LV* outperforms all the other predictors for these sizes. Apparently, a SAg-based CE with a last *n* value predictor (*LV* is a last *one* value predictor) represents a strong combination for both re-fetch and re-execute.

Note that the performance of some of the predictors from the literature actually decreases with re-fetch when increasing the predictor size due to a high percentage of mispredictions.

5.2 Sensitivity analysis

5.2.1 Using distinct last values

Load value predictors to exploit last value [3, 10], stride [3, 15], and finite context predictability [15] have been studied at length in the current literature. Last *n* value predictability, on the other hand, has been less explored despite its simplicity and considerable potential. The only proposed predictor to take advantage of last *n* value locality is Wang and Franklin's last distinct four value predictor [22]. Lipasti and Shen [9] study last *n* value predictability in combination with an oracle that always selects the correct value (if possible) but they do not propose an implementable predictor.

Retaining the last *n* fetched values of a given load instruction is straight forward. To make the most of the retained values, Wang and Franklin [22] suggest storing only values that are not already stored (i.e., only distinct values). Unfortunately, this approach requires content addressable memory. Storing the last *n* values regardless of whether any of them are identical is much simpler. Our results from Section 5.1.2 suggest that this lower complexity approach is not only more cost effective but also yields superior performance because it makes the selection process more accurate.

Figure 5.6 shows the prediction potential for different *n* when storing every loaded value versus only storing distinct values. The potential is given as the percentage of the fetched load values that are identical to at least one of the last *n* (distinct) fetched values. The results only take into account load instructions within the approximately 300 million simulated instructions of each of the eight benchmarks. However, the numbers in Figure 5.6 are very representative of the generally observed predictability. Complete executions of the programs revealed prediction potentials within five percent of the values shown in the figure.

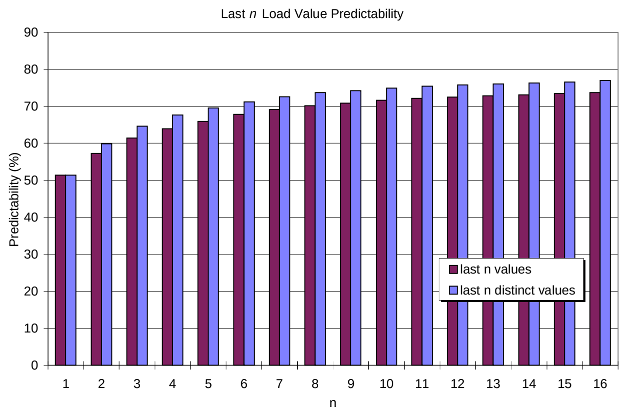

Figure 5.6: The average last *n* value predictability (duplicate values are allowed) and the average last *n* distinct value predictability (no duplicates) of the load values within the simulated segment of each of the eight SPECint95 programs.

Figure 5.6 shows that larger *n* result in higher predictability potential. This result is intuitive since the chance of finding the correct value increases as the number of stored values becomes larger. The increase is considerable for small *n* up to about four. Then the "curve" starts flattening out and reaches saturation at approximately *n* = 11, at which point almost no additional potential is gained by further increasing *n*.

One very interesting observation is that for *n* larger than four, the potential difference between distinct and non-distinct is virtually constant (3.3%). This means that the relative advantage of storing distinct values becomes smaller as *n* gets larger.

For *n* = 4, which is the predictor width Wang and Franklin chose [22], the difference of 3.6% represents one eighteenth of the total potential of 64%, indicating that the simpler approach of retaining not necessarily distinct values is in theory able to perform almost as well as its more complex counterpart.

5.2.2 Predictor size and width

It is important that a load value predictor's height be large enough to accommodate the (load instruction) working set size (Section 4.1). If the predictor is too short, multiple frequently executed load instructions will have to share a predictor slot, which almost always results in detrimental aliasing. Predictors that are too tall, on the other hand, underutilize many of their slots. Consequently, once a predictor is large enough to accommodate the working set size, further increases in the predictor height will not increase the performance because the additional slots will not be utilized effectively. Instead, additional real-estate could be used to increase the amount of information stored in each slot, which should enable the predictor to make better and/or more predictions and thus improve its performance. Hence, the optimal predictor width depends on the working set size of the programs and the available real-estate for the predictor.

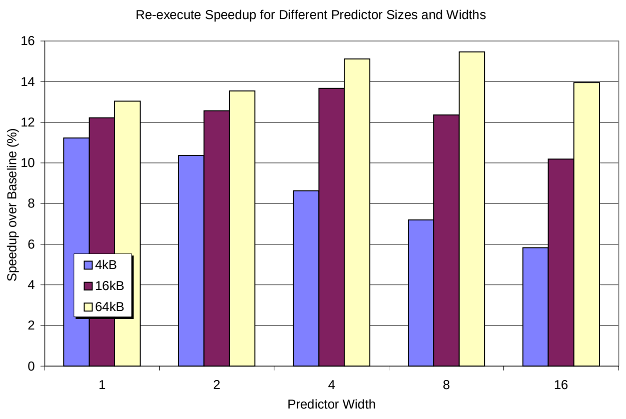

Figure 5.7: Maximum mean speedup for three predictor sizes and five predictor widths with a re-execution recovery policy.

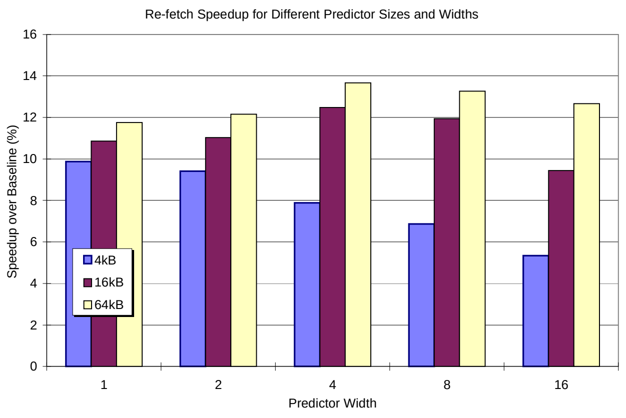

Figure 5.8: Maximum mean speedup for three predictor sizes and five predictor widths with a re-fetch recovery policy.

To better evaluate the tradeoff between predictor width and size, we present Figure 5.7 and Figure 5.8. They show the best mean speedup we were able to obtain for various predictor sizes and widths.

Figure 5.7 shows that for very small predictors (4kB of state for storing values), a width of one results in the highest speedup (see also Section 5.1.2). Storing two values per slot and halving the number of slots yields less speedup because there are not enough slots to hold the SPECint95 working set, which results in more aliasing and lower performance. This effect is even more pronounced for larger widths, hence the continuous decrease in speedup as the predictor becomes wider and shorter.

With 16kB of state a width of four yields the best speedup. Detrimental aliasing only sets in above four entries per slot. When we increase the predictor size to 64kB, the best width turns out to be eight. Only at a width of 16 does the performance start to decrease again.

Figure 5.8 is identical to Figure 5.7 except that the misprediction recovery mechanism used is re-fetch instead of re-execution. The resulting optimal predictor widths are the same with the exception that a width of four (instead of eight) now yields the best speedup in the 64kB case. This change is due to the high misprediction sensitivity of the re-fetch mechanism. It appears that in the 64kB case, $n = 8$ results in both more correct predictions and more incorrect predictions, which is advantageous with re-execution but harmful with a re-fetch architecture.

5.2.3 History length

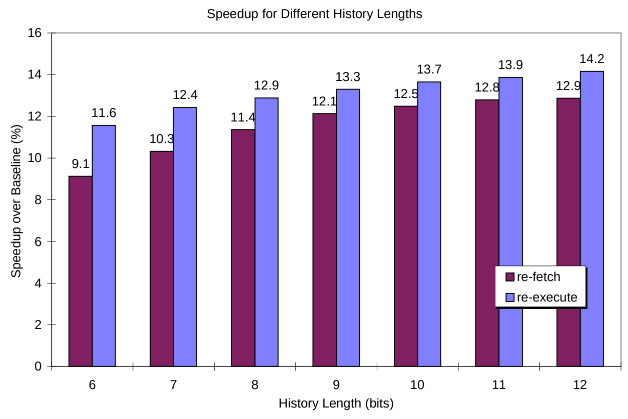

Figure 5.9: Mean speedup with varying history lengths.

We already found history lengths of ten bits to work well for a predictor width of one [1, 2]. Figure 5.9 shows that ten bits also mark the beginning of the saturation point for the last four value predictor, both for re-fetch and re-execute. Note that it is important not to choose the history length too large since every additional bit doubles the number of required saturating counters.

5.2.4 Counter parameters

For space reasons, we cannot present all the results pertaining to counter parameters and will only give a summary in this subsection.

With a re-execute misprediction recovery mechanism, we found four-bit saturating up/down counters to work best with our last four value predictor. Counters both smaller and larger than four bits yield less performance. Hence, we use a counter top of sixteen. We found a threshold of nine with a penalty (counter decrement) of three to work quite well for this counter size, but the exact values are not crucial. Only penalties of one or two result in significantly lower performance.

For re-fetch, thresholds and penalties of about half the counter top value work very well with the last four value predictor. Five, six, and seven bit counters yield the best speedup. We use the smallest of the three, which has a counter top of 32, with a threshold and a penalty of 16. Again, numbers near the ones we picked all result in approximately the same speedup.

6. Summary and conclusions

Once a load value predictor is tall enough to hold all the frequently executed load instructions, increasing its height further does not result in much better performance because the additional slots cannot be used effectively. As an alternative, we propose making predictors wider instead of taller, i.e., using the extra real-estate to increase the amount of information stored in each slot rather than to increase the number of slots. Doing so should enable the predictor to make better and/or more predictions and thus improve its performance.

One way to make a last value predictor "wider" is by having it retain the last *n* and not just the load value in each of its slots. In this paper we study a load value predictor that benefits from such an increase in width and present an effective implementation thereof.

Our measurements show that very small predictors (retaining 4kB or 8kB of values) perform the best in the conventional last value configuration. However, 16kB or larger predictors benefit from having a width of more than one. For instance, the best 16kB configuration for running SPECint95 has a width of four. In other words, a last four value predictor outperforms a last two value predictor with twice as many lines as well as a last eight value predictor with half as many lines (same total size).

Interestingly, our last four value predictor also outperforms its more complex counterpart from the literature that retains the last four distinct load values, even though there is slightly more potential for predictability in last *n* distinct value locality than there is in last *n* value locality.

We performed hundreds of cycle-accurate, pipeline-level simulations of a superscalar high-performance microprocessor to evaluate the performance of various predictors, including several from the literature. The results show that our last four value predictor outperforms the other predictors (that are all scaled to about the same size), including ones that are more complex. More importantly, our predictor performs well enough with the existing re-fetch misprediction recovery mechanism that the added benefit of a more complex and not yet realized re-execution core is small in comparison.

In spite of its good performance, a comparison of our predictor with some oracles revealed that there is still significant opportunity for improvement left.

We are currently trying to improve our predictor's performance by hybridizing it with other predictors, and we are investigating ways to shrink the predictor size. In future work we intend to study whether profiling can be used to further improve the performance, how prefetching affects the predictor, and whether other predictors can benefit from the confidence estimator we use in our predictors.

Acknowledgments

This work was supported in part by the Hewlett Packard University Grants Program (including Gift No. 31041.1). We would like to especially thank Tom Christian for his support of this project, the anonymous reviewers for their helpful comments, and Dirk Grunwald and Abhijit Paithankar for providing and helping with the simulator.

References

[1] M. Burtscher, B. G. Zorn. *Load Value Prediction Using Prediction Outcome Histories.* Unpublished Technical Report CU-CS-873-98, University of Colorado at Boulder. October 1998.

[2] M. Burtscher, B. G. Zorn. "Profile-Supported Confidence Estimation for Load Value Prediction". Submitted to the *Journal of Instruction Level Parallelism (JILP).* 1999.

[3] F. Gabbay. *Speculative Execution Based on Value Prediction.* EE Department Technical Report #1080, Technion - Israel Institute of Technology. November 1996.

[4] F. Gabbay, A. Mendelson. "Can Program Profiling Support Value Prediction?" *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[5] F. Gabbay, A. Mendelson. "The Effect of Instruction Fetch Bandwidth on Value Prediction". *25th International Symposium on Computer Architecture*. June 1998.

[6] J. Gonzalez, A. Gonzalez. "The Potential of Data Value Speculation to Boost ILP". *12th International Conference on Supercomputing*. July 1998.

[7] R. E. Kessler, E. J. McLellan, D. A. Webb. "The Alpha 21264 Microprocessor Architecture". *1998 International Conference on Computer Design*. October 1998.

[8] D. C. Lee, P. J. Crowley, J. J. Baer, T. E. Anderson, B. N. Bershad. "Execution Characteristics of Desktop Applications on Windows NT". *25th International Symposium on Computer Architecture*. June 1998.

[9] M. H. Lipasti, J. P. Shen. "Exceeding the Dataflow Limit via Value Prediction". *29th International Symposium on Microarchitecture*. December 1996.

[10] M. H. Lipasti, C. B. Wilkerson, J. P. Shen. "Value Locality and Load Value Prediction". *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. October 1996.

[11] S. McFarling. *Combining Branch Predictors.* TN 36, DEC-WRL. June 1993.

[12] A. Paithankar. *AINT: A Tool for Simulation of Shared-Memory Multiprocessors*. Master's Thesis, University of Colorado at Boulder. 1996.

[13] G. Reinman, B. Calder. "Predictive Techniques for Aggressive Load Speculation". *31st Annual ACM/IEEE International Symposium on Microarchitecture*. December 1998.

[14] B. Rychlik, J. Faistl, B. Krug, J. P. Shen. "Efficacy and Performance Impact of Value Prediction". *1998 International Conference on Parallel Architectures and Compiler Technology*. October 1998.

[15] Y. Sazeides, J. E. Smith. "The Predictability of Data Values". *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[16] Y. Sazeides, J. E. Smith. *Implementations of Context Based Value Predictors.* Technical Report ECE-97-8, University of Wisconsin-Madison. December 1997.

[17] Y. Sazeides, J. E. Smith. "Modeling Program Predictability". *25th International Symposium on Computer Architecture.* June 1998.

[18] J. E. Smith, G. S. Sohi. "The Microarchitecture of Superscalar Processors". *Proceedings of the IEEE*. 1995.

[19] E. Sprangle, R. Chappell, M. Alsup, Y. Patt. "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference". *24th Annual International Symposium of Computer Architecture*. June 1997.

[20] *SPEC CPU'95*. August 1995.

[21] A. Srivastava, D. Wall. "A Practical System for Intermodule Code Optimization at Linktime". *Journal of Programming Languages* 1(1). March 1993.

[22] K. Wang, M. Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors". *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[23] T. Y. Yeh, Y. N. Patt. "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History". *20th Annual International Symposium of Computer Architecture.* May 1993.