Profile-Supported Confidence Estimation for Load-Value-Prediction

Martin Burtscher and Benjamin G. Zorn
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430
{burtsche, zorn}@cs.colorado.edu

Abstract

Due to their occasional very long latency, load instructions are among the slowest instructions of current high-performance microprocessors. Unfortunately, the long latency of one instruction also delays the execution of its dependent instructions, which can significantly affect system performance. Load value prediction alleviates this problem by allowing the CPU to speculatively continue processing without having to wait for the slow memory access to complete.

The potential of today's load value predictors for making correct predictions is only about 40 to 70 percent. Confidence estimators are employed to estimate how likely a prediction is to be correct and to keep the predictor from making a (probably incorrect) prediction if the confidence is below a preset threshold. Setting this threshold to a higher level increases the probability that the attempted predictions will be correct (higher accuracy) but results also in more missed opportunities for making correct predictions (lower coverage).

The profile-supported confidence estimator we present in this paper is able to trade off coverage for accuracy and vice-versa with previously unseen flexibility and reaches an average prediction accuracy over SPECint95 of as high as 99.4%.

A detailed pipeline-level simulation shows that our profile-supported load value predictor not only outperforms other predictors, but is also the only predictor that yields a speedup at all when a re-fetch misprediction recovery policy is used.

1. Introduction

Due to their occasional long latency, load instructions have a significant impact on system performance. If the gap between CPU and memory speed keeps widening, the load latency will become even longer. Since loads are also among the most frequently executed instructions [LCB+98], improving their execution speed should significantly improve the overall performance of the CPU.

Most programs tend to load the same values repeatedly. This behavior is commonly referred to as *value locality* [LWS96, Gab96]. For example, about half of all the load instructions of SPECint95 retrieve the same value as they did the previous time they were executed.

Context-based load value predictors try to exploit the existing value locality. For instance, a very simple predictor could always predict the previously loaded value. We named this scheme *Basic LVP* (last value predictor). To reduce the number of mispredictions, load value predictors normally consist of two main parts: a *value predictor* that predicts a value based on previously loaded values and a *confidence estimator* (CE), which decides whether or not to make a prediction using the predicted value. All previously proposed predictors and our own contain these two parts in some form. Nevertheless, to our knowledge, we are the first to use this nomenclature. The CE only allows predictions to take place if the confidence that the prediction will be correct is high. This is important because sometimes the value predictor does not contain the right information to make a correct prediction. In such a case it is better not to make a prediction because incorrect predictions slow the processor more than making no prediction at all.

Since CEs have to make binary decisions (predict or don't-predict), they are similar to branch predictors, which also make binary decisions (branch taken or not-taken). One very successful idea in branch prediction, which is also applicable to load value prediction, is keeping a small history recording the most recent prediction outcome (success or failure) [SCAP97]. The intuition is that the past prediction behavior tends to be very indicative of what will happen next. For example, if a prediction was successful the last few times, there is a good chance that it will be successful again. Hence, the prediction-outcome history, as we termed it, represents a measure of confidence.

If load values are predicted quickly and correctly, the CPU can process 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 memory, which can take many cycles. *Speculative execution* allows the CPU to continue execution with a predicted value before the prediction outcome is known [SmSo95]. Because branch prediction requires a similar mechanism, most modern CPUs already contain the necessary hardware to perform this kind of speculation [Gab96].

However, using branch misprediction recovery hardware for load value mispredictions causes all the instructions that follow a misspeculated instruction to be purged and *re-fetched*. This is a very costly operation and makes a high prediction accuracy paramount.

Unlike branches, which invalidate the whole execution path if 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 have to be *re-executed* with the correct input value(s). Consequently, a significantly better recovery mechanism for load misspeculation would only have to re-execute the instructions that are dependent on the mispredicted load. Such a recovery policy is much less susceptible to mispredictions and favors a higher coverage, but may be prohibitively hard to implement [GrPa98].

We devised a load value predictor with a prediction outcome history-based confidence estimator that performs very well, in particular with the simpler re-fetch policy with which no other predictor is able to attain a speedup. Our predictor also outperforms all the other predictors we looked at with a re-execution policy, as preliminary pipeline-level simulations show. Section 5.4 provides more detailed results.

2. Predictor architecture

Figure 2.1 shows the elements of a confidence-based load value predictor. The largest component is an array (cache) of 2^n lines for retaining confidence information and previously fetched values. The hashing hardware is used to generate an *n*-bit index out of the load instruction's address (and possibly other processor state information). Finally, the decision logic computes whether a prediction should be made based on the confidence information.

All the predictors in this paper use *PC div 4 mod* 2^n as a hash-function[1]. Better hash-functions probably exist. However, an investigation thereof is beyond the scope of this paper.

When a prediction needs to be made, the hash-function computes an index to select a cache line. The value stored in the selected line (or one of the stored values if there are multiple) becomes the predicted value, and the decision logic decides whether a prediction should be attempted with this value.

Once it is known whether a prediction was correct, the value in the cache line that was used for making the prediction is replaced by the true load value and the corresponding confidence information is updated to reflect the prediction outcome.

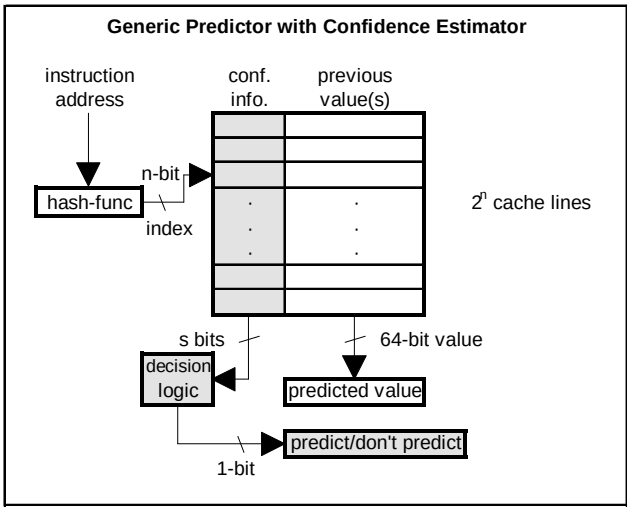

Figure 2.1: The components of a load value predictor with a confidence estimator (shaded).

3. Related work

Lipasti et al. [LWS96] describe an untagged last value predictor (predicts the previously fetched load value) to exploit the existing load value locality. Their predictor utilizes saturating up/down counters as confidence estimators. In Section 5.4, we compare our predictor to theirs.

Gabbay's dissertation proposal [Gab96] introduces a tagged last value predictor and a tagged stride predictor (predicts the previously fetched load value plus an offset). Both predictors use the tags as confidence estimators. Load instructions, as opposed to other types of instructions, exhibit virtually no stride behavior. Therefore, we exclude the stride predictor from our comparison since we feel that the extra hardware to store the strides is not cost effective.

Wang and Franklin [WaFr97] are the first to propose a multi-value predictor. It keeps the last four distinct values per line and uses the pattern of the last six accesses as index into an array of saturating up/down counters for confidence estimation. It has the highest prediction accuracy of all the predictors currently in the literature. Nevertheless, our predictor considerably outperforms theirs as the comparison in Section 5.4 shows.

In the area of branch prediction, a significant amount

[1] The *div 4* eliminates the two least significant bits which are always zero since the processor we use requires instructions to be aligned.

of related work exists. Yeh and Patt [YePa92, YePa93] describe sets of two-level branch predictors and invent a taxonomy to distinguish between them. We adopt one of their designs for use as a confidence estimator in our load value predictor.

Sechrest, Lee, and Mudge [SLM95] refine some of Yeh and Patt's two-level predictors by studying and describing how to program the static predictor components. They distinguish between profile-based and algorithmic approaches. We discuss both schemes in Section 5.1.

4. Methodology

All our measurements are performed on the DEC Alpha AXP architecture [DEC92].

To obtain the actual load values, we instrumented the binaries using the ATOM tool-kit [EuSr94, SrEu94], which allowed us to elegantly simulate the proposed predictor in software and to easily change its parameters.

For our detailed pipeline-level simulations we use the Superscalar back-end of AINT [Pai96]. It is configured to emulate a processor similar to the DEC Alpha 21264 [KMW98]. The final paper will contain a description of the parameters like the instruction window size etc.

4.1 Benchmarks

We use the eight integer programs of the SPEC95 benchmark suite [SPEC95] for our measurements. These programs are well understood, non-synthetic, and compute-intensive, which is ideal for processor performance measurements. They are also quite representative of desktop application code, as Lee et al. found [LCB+98]. Table 4.1 gives relevant information about the SPECint95 programs.

We use the larger ref-inputs and the more optimized peak-versions of the programs (compiled using DEC GEM-CC with *-migrate -std1 -O5 -ifo -g3 -non_shared*). The binaries are statically linked, which enables the linker to unify the different global object tables (GOTs). This significantly reduces the number of run-time constants that are loaded during execution [SrWa94]. All programs are run to completion. The result is approximately 87.8 billion executed load instructions per simulation. Note that the few floating point load instructions contained in the binaries are also measured and that load immediate instructions are not taken into account since they do not access the memory and therefore do not need to be predicted.

An interesting point to note is the uniformly high percentage of load instructions executed by the programs. About every fifth instruction is a load. This is in spite of good register allocation and GOT removal.

Another interesting point is the relatively small number of load sites that contribute most of the executed load instructions. For example, less than 5% of the load sites make for 90% of the executed loads. Only 43% of the load sites are executed at all.

In these benchmarks, an average of 52.3% of the load instructions fetch the same value as they did the previous time they were executed and 69.5% fetch a value that is identical to one of the last four distinct values fetched.

Information about the SPECint95 Benchmark Suite						
	total executed	load	load sites that account for			
program	load instructions	sites	Q50	Q90	Q99	Q100
compress	10,537 M (17.5%)	3,961	17	58	81	690
gcc	80 M (23.9%)	72,941	870	5,380	14,135	34,345
go	8,764 M (24.4%)	16,239	204	1,708	4,221	12,334
ijpeg	7,141 M (17.2%)	13,886	42	187	423	3,456
li	17,792 M (26.7%)	6,694	42	138	312	1,932
m88ksim	14,849 M (17.9%)	8,800	52	216	456	2,677
perl	6,207 M (31.1%)	21,342	44	169	227	3,586
vortex	22,471 M (23.5%)	32,194	57	585	3,305	16,651
average	10,980 M (21.8%)	22,007	166	1,055	2,895	9,459

Table 4.1: The number of load instructions contained in the binaries (load sites) and executed by the individual programs (in millions 'M') of the SPECint95 benchmark suite. The numbers in parentheses denote the percentage of all executed instructions that are loads. The quantile columns show the number of load sites that contribute the given percentage (e.g., Q50 = 50%) of executed loads.

4.2 Metrics for load value predictors

The ultimate metric for comparing load value predictors is the speedup attained by incorporating a given predictor into a CPU. Unfortunately, speedups are dependent on the architectural features of the underlying CPU. Therefore, non-implementation specific metrics are also valuable.

A value predictor with a *confidence estimator* can produce four prediction outcomes: correct prediction, incorrect prediction, correct non-prediction, and incorrect non-prediction. We denote the number of times each of the four cases is encountered by PCORR, PINCORR, NPCORR, and NPINCORR, respectively. To make the numbers independent of the number of executed load instructions, they are normalized such that their values sum to one.

Normalization: PCORR + PINCORR + NPCORR + NPINCORR = 1

Unfortunately, these four numbers do not represent adequate metrics for comparing predictors. For example, it is not clear if predictor A is superior to predictor B if predictor A has both a higher PCORR and a higher PINCORR than predictor B. Instead, we use standard metrics for confidence estimation, which have recently been adapted to and used in the domain of branch prediction and multi-path execution [JRS96, GKMP98]. To our knowledge, we are the first to use these standard metrics in the domain of load value prediction.

- Accuracy: $\text{ACC} = \frac{\text{PCORR}}{\text{PCORR} + \text{PINCORR}}$
- Coverage: $\text{COV} = \frac{\text{PCORR}}{\text{PCORR} + \text{NPINCORR}}$
- Potential: $\text{POT} = \text{PCORR} + \text{NPINCORR}$

The POT represents the fraction of predictable values, which is a property of the value predictor alone and is independent of the confidence estimator. However, if the potential is low, even a perfect confidence estimator is unable to make many correct predictions.

The ACC represents the probability that a prediction is correct and the COV represents the fraction of predictable values identified as such. Together they describe the quality of the confidence estimator. The accuracy is the more important metric, though, since a high accuracy translates into many correct predictions (which save cycles) and few incorrect predictions (which cost cycles), whereas a high coverage only translates into better utilization of the existing potential.

Note that ACC, COV, and POT fully determine PCORR, PINCORR, NPCORR, and NPINCORR given that they are normalized.

4.3 Cross-validation

Cross-validation is a technique incorporated to exclude self-prediction. It is used throughout this paper (where applicable) and works as follows: one program is removed from the set of benchmark programs, the behavior of the remaining programs is measured to tune the prediction hardware, and then the program that was removed is run on this hardware. This process is repeated for every program in the set. Thus, the performance of all the programs is evaluated using only knowledge about other programs.

5. Results

5.1 Prediction outcome histories

In addition to tags and saturating counters, the branch prediction literature also describes histories that record the recent prediction successes and failures as a very successful idea [SCAP97]. We found this to be true in the domain of load value prediction as well. In fact, prediction outcome histories (as we call them) are much better suited for load value prediction than tags and saturating counters because they scale better, they allow accuracy-coverage pairs to be chosen at a much finer granularity, and they yield much higher accuracies.

If such histories are to be used as a measure of confidence, it is necessary to know which ones are (normally) followed by a correct prediction and which ones are not.

Heuristics and algorithms to do this exist. For example, Sechrest et al. [SLM95] describe a scheme that first looks for repeating patterns and then falls back to population counting if none can be detected. They named this scheme *(algo)*. As an alternative, they suggest running a set of programs and recording their behavior. This profile-based approach is called *(comp)*. We use the (comp) scheme for our predictor since it performs considerably better and is much more flexible than (algo).

To better explain how (comp) works, we present Table 5.2, which shows the output of a 4-bit history run based on SPECint95 behavior. The second row of the table, for instance, states that a *failure, failure, failure, success* history (denoted by *0001*) is followed by a successful last value prediction 26.9% of the time. Of all the encountered histories, 2.7% were *0001*.

The table shows the sum over all the benchmarks and is for illustration purposes only. The results presented in the subsequent sections were generated using cross-validation (Section 4.3).

SPECint95 Last Value Predictability		
history	predictability	occurance
0000	6.9	32.2
0001	26.9	2.7
0010	19.1	2.9
0011	49.9	1.6
0100	34.3	2.9
0101	33.6	1.9
0110	44.9	1.3
0111	59.4	2.2
1000	24.2	2.7
1001	46.3	1.8
1010	66.8	1.9
1011	66.1	1.9
1100	53.1	1.6
1101	57.2	1.9
1110	52.3	2.2
1111	96.6	38.3

Table 5.2: Predictability and occurrence split up by history pattern. The predictability signifies the percentage of last value predictable loads following the given histories. The occurrence denotes the percentage of the time the respective history was encountered.

Note that it is not necessary to make a prediction following every history with a greater than 50% probability of resulting in a correct prediction. Rather, the predict/don't-predict threshold can be set anywhere. The optimal setting depends on the characteristics of the CPU the prediction is going to be made on.

If only a small cost is associated with making a misprediction (e.g., re-execute), it is probably wiser to predict a larger number of load values, albeit also a larger number of incorrect ones. If, on the other hand, undoing speculative operations takes a long time and should therefore be avoided (e.g., re-fetch), it makes more sense not to predict quite as many loads but to be confident that the ones that

are predicted will most likely be correct.

If we want to be highly confident that a prediction is correct, say 96.6% confident, the decision logic would only allow predictions for those histories that have a predictability of greater than or equal to 96.6%, i.e., only for history *1111* in our example. This threshold would result in 38.3% of all loads being predicted (of which 96.6% would be correct for our benchmark suite).

As the example illustrates, four history bits are enough to reach an average accuracy in the high nineties (for our benchmarks), which other proposed predictors cannot reach. With longer histories, our approach yields even higher accuracies and more correct predictions.

5.2 The SSg(comp) Last Value Predictor

Based on these encouraging results, we designed a load value predictor that consists of a *last value predictor* (LVP) and a *prediction outcome history-based confidence estimator*. The histories are stored in the confidence information field of the cache-lines (see Figure 2.1). The confidence estimator is similar to Yeh and Patt's SSg branch predictor [YePa93], programmed with Sechrest et al.'s (comp) approach. Hence we call our predictor *SSg(comp) LVP*.

Predictions are performed as described in Section 2. Once the prediction outcome is known, a new bit is shifted into the history of the selected cache line and the oldest bit is shifted out (lost). If the true value is equal to the value in the cache, a one is shifted in, otherwise a zero is shifted in. Then the value in the cache is replaced by the true value.

We decided to use a direct-mapped cache since we empirically observed few conflicts with moderate cache sizes. While this might be an artifact of our benchmarks, even much larger programs will not create significantly more conflicts as long as their active working set of load instructions does not exceed the capacity of the cache.

Note that, unlike instruction and data caches, predictor caches do not have to be right all the time. Hence, neither tag nor valid bits are a requirement. We decided to omit both since having them would only result in a virtually immeasurable increase in accuracy, which we believe does not justify the extra hardware.

5.3 SSg(comp) LVP results

All the results for the SSg(comp) last value predictor are generated using cross-validation (Section 4.3). Before every run all cache entries are initialized with zeros.

Figure 5.1 shows the attainable accuracy-coverage pairs (by varying the threshold) for different cache sizes when ten-bit histories are used. The numbers are averages over the eight SPECint95 programs. Values closer to the upper right corner are better.

The broad range in both dimensions is quite apparent and, hardly surprising, the larger the predictor the better its performance. However, it is also visible that both the performance and the delivered potential reach the saturation point at about 4096 entries.

Figure 5.2 is similar to the previous figure except now the cache size is held constant at 1024 entries and the length of the histories is varied. Again, longer histories perform better. Saturation sets in at about ten bits.

Note that we performed a much broader investigation of the parameter space but cannot include all the results. We picked Figure 5.1 and Figure 5.2 because they are quite representative of the generally observed behavior.

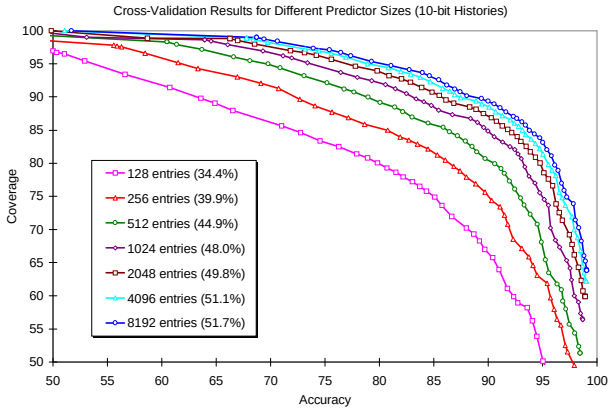

Figure 5.1: Accuracy-coverage pairs for different cache sizes and 10-bit histories. Each dot corresponds to a threshold (in 2% increments). The numbers in parentheses denote the potential of the respective predictor configuration.

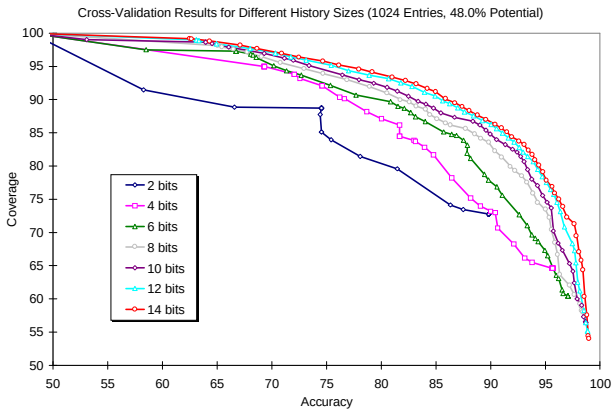

Figure 5.2: Accuracy-coverage pairs for different history sizes and 1024-entry caches. Each dot corresponds to a threshold (in 2% increments).

5.4 Predictor comparison

This section compares several load value predictors: a *Basic LVP* (without confidence estimator), a *Tagged LVP*

[Gab96], a *Bimodal LVP* [LWS96], our *SSg(comp) LVP*, an *SSg(algo) LVP*, and a *Last Distinct 4 Values* predictor [WaFr97]. We also look at increasing the data cache size as an alternative to adding a load value predictor.

To make the comparison between the predictors as fair as possible, all are allowed space to hold 2048 values plus whatever else they require to support that size. This results in approximately 19 kilobytes of state, which we find reasonable given that the DEC Alpha 21264 processor incorporates two 64 kilobytes L1 caches on chip [KMW98].

Table 5.3 shows the hardware cost of the five predictors in number of state bits. The table also lists the potential of the individual predictors.

Hardware Cost and Potential of several 2048-entry Predictors			
	state bits	rel. cost	potential
Basic LVP	131072	100.0 %	49.85 %
Tagged LVP (19-bit tags)	169984	129.7 %	49.85 %
Bimodal LVP (3-bit counters)	137216	104.7 %	49.85 %
SSg LVP (8-bit histories)	147456	112.5 %	49.85 %
SSg LVP (14-bit histories)	159744	121.9 %	49.85 %
Last Distinct 4 Values	217600	166.0 %	48.41 %

Table 5.3: Hardware cost in number of state-bits and the potential of various load value predictors.

The *Basic LVP* requires the smallest number of state bits (sum of counter, cache, history, tag, and valid bits). Since Alphas are 64-bit machines, every value in the cache counts as 64 bits. Consequently, the *Basic LVP* requires 131,072 bits of storage. This is our base case.

The *Tagged LVP* augments the *Basic LVP* with a tag per cache line. If we assume a 4GB address space, the tags have to be 19 bits long for a 2048-entry cache. This scheme requires 29.7% more storage than the base case. Predictions are only made if the tag matches and after each prediction the value and the tag is updated.

The *Bimodal LVP* incorporates a 3-bit saturating up/down counter per line. McFarling named this scheme *Bimodal* [McF93]. We found 3-bit counters and always updating the values to work the best. Predictions are only made if the counter value is greater or equal to a preset threshold, which can be varied between one and seven. This scheme only requires 4.7% additional hardware. In spite of this marginal increase, it performs a great deal better than the first two schemes, including the more hardware intensive one.

Our *SSg(comp) LVP* is 12.5% above the base case when 8-bit histories are used and 21.9% with 14-bit histories. The *(algo)* predictors require the same amount of state.

The *Last Distinct 4 Values* predictor is rather complex and stores four distinct values per line, so the cache only has 512 lines. The bit count for this scheme is 66.0% over the base case. It incorporates 4-bit saturating up/down counters as part of the confidence estimator. However, the counters are only allowed to count up to twelve [WaFr97], which limits the possible threshold values to one through twelve.

Figure 5.3 and Figure 5.4 show how the confidence estimators of different predictors perform with a small (1024 entries) and a large (8192 entries) configuration, respectively.

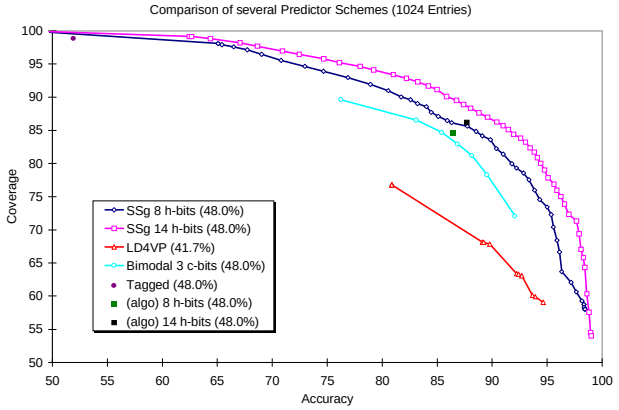

Figure 5.3: Accuracy-coverage pairs of different predictors with 1024-entry caches. The dots correspond to various thresholds. The numbers in parentheses denote the potential of the respective predictor.

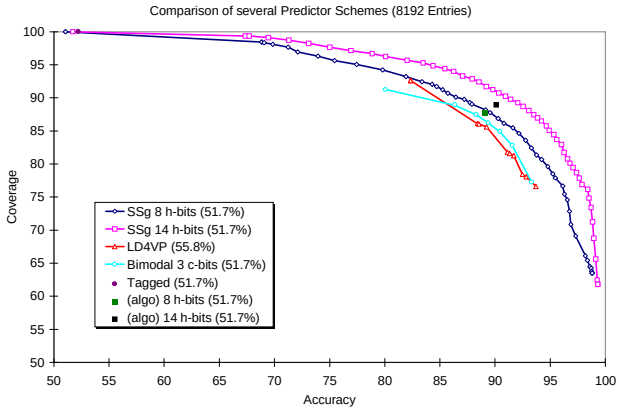

Figure 5.4: Accuracy-coverage pairs of different predictors with 8192-entry caches. The dots correspond to various thresholds. The numbers in parentheses denote the potential of the respective predictor.

Note that the *Basic LVP* is not visible. Its coverage is 100% but its accuracy is only about 50% in both cases. The *Tagged LVP* and the two *(algo)* schemes allow no variability and are therefore each represented by a single point.

With eight history bits, our predictor outperforms all other predictors except the 14-bit *SSg(algo) LVP*. We take this as evidence that prediction outcome histories are indeed better suited for load value prediction than other

approaches. Our 14-bit *SSg(comp) LVP* outperforms all other predictors. Note how much larger its range of accuracy-coverage pairs is in both figures and how much higher an accuracy it can reach in comparison to the other predictors.

All the predictors benefit from an increase in size. However, our measurements with infinite caches show that the 8192-entry results are already close to the limit for all predictors and that our predictor maintains its superiority. LD4VP profits the most from going from 1024 entries to 8192 entries. That is because LD4VP stores four values per cache line, which results in four times fewer cache lines and consequently more aliasing.

Figure 5.5 shows the speedups measured using a detailed pipeline simulation of a microprocessor similar to the DEC Alpha 21264. The displayed results are the speedups of gcc (one of the SPECint95 programs). It is the shortest running program but at the same time one of the hardest to predict and has by far the largest number of load sites (Table 4.1).

The results are given for both a re-fetch and a re-execute misprediction recovery policy. For predictors that allow multiple threshold values, the best result is listed. The thresholds that yield the highest speedup are, seven (out of seven) for the *Bimodal LVP* both for re-fetch and re-execute, 97% for *SSg(comp)* with re-fetch, 73% for *SSg(comp)* with re-execute, twelve (out of twelve) for *LD4VP* using re-fetch and eleven using re-execute.

Bimodal LVP, *SSg(algo) LVP*, and *LD4VP* perform quite well with a re-execute policy. Nevertheless, *SSg(comp) LVP* outperforms them. However, the simple and low cost *Bimodal* scheme gets very close to the performance of *SSg(comp)* and might therefore be the predictor of choice when re-execution is employed.

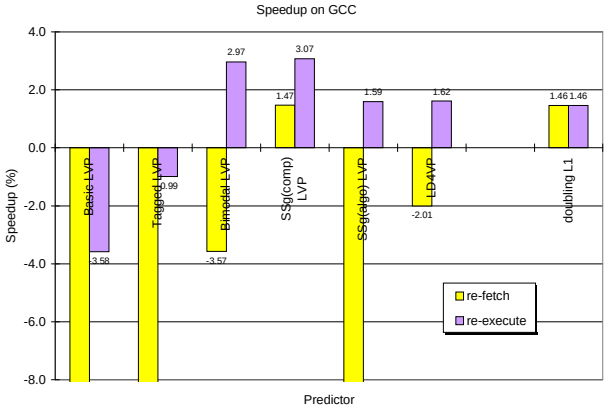

Figure 5.5: Attainable speedups of gcc on a DEC Alpha 21264-like processor (the cut-off negative speedup percentages for re-fetch are -46.6%, -40.3%, and -15.1% for *Basic*, *Tagged*, and *SSg(algo)*, respectively).

With the much simpler re-fetch mechanism [GrPa98], which most of today's CPUs already contain and therefore the more likely recovery mechanism in the near future, our *SSg(comp) LVP* is the only predictor to yield a performance improvement. Furthermore, when using re-fetch, our confidence estimator only makes a prediction if all history bits are ones. This means that the decision logic becomes trivial; it only has to look for this one pattern, which will make it very fast.

In the re-execute case, the optimal threshold for our predictor is 73%, which amounts to about 1100 history patterns causing a prediction (out of 16384). It is not feasible to look for this large number of patterns using comparators. Rather, one would probably use the pattern as an index into a preprogrammed 1 bit by 16384 ROM that returns a one for those histories that should trigger a prediction and a zero otherwise. The ROM effectively represents a second level of indirection. Performing two table lookups per cycle should be feasible since current branch predictors also comprise two levels.

The rightmost column in Figure 5.5 denotes the speedup resulting from doubling the simulated processor's L1 data-cache. Despite this hardware increase of 564,224 state-bits, the resulting speedup does not even reach the speedup of our *SSg(comp) LVP*, which requires four times less hardware.

Doubling the L1 data-cache reduces its load miss-rate from 2.5% to 1.2%. Obviously, there is not much potential for improvement left. We can only conclude that above a certain cache size, it makes more sense to add a load value predictor than to increase the cache size.

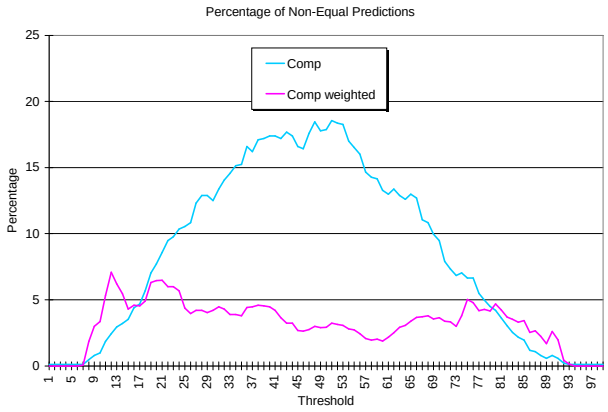

Figure 5.6: Average percentage of dissimilar prediction outcomes when using profile-based prediction *(comp)*. The weighted percentages are weighted by the relative occurrence of the non-equal history patterns.

One last point we would like to address is the fact that our predictor is profile-based and needs to be "preprogrammed", which might seem inconvenient. As we already mentioned, this is not a problem in case of re-fetch

since the profiles uniformly revealed that only histories of all ones need to be considered. This insight not only makes the decision logic of our predictor trivial and fast, but also eliminates the necessity of further profile runs.

With re-execute this is different. Nevertheless, as Figure 5.6 shows, the profile results exhibit a high degree of invariability. In other words, all the cross-validations yield the about same result (within a few percents, especially in the interesting range above 73%), meaning that those prediction-causing history patterns are most likely quite universal and not very dependent on the programs. This in turn means that for any given CPU, the optimal threshold value has to be evaluated only once. Then the predictor's decision logic can be programmed once and for all, again rendering further profile runs superfluous.

6. Summary and conclusions

In this paper we describe a novel confidence estimator for load value predictors. It uses recent prediction outcome histories to decide whether or not to attempt a prediction. Profile information is utilized to determine which history patterns should cause a prediction. Because of its high invariability, this can be done once and does not need to be repeated every time the predictor is to be used.

Our confidence estimator (CE) reaches much higher accuracies than tag and saturating-counter-based CEs, and combined with a simple last value predictor it significantly outperforms previously proposed predictors.

We conclude that prediction outcome histories are better suited for the domain of load value prediction than other approaches, including considerably more complex ones. When a re-fetch misprediction recovery mechanism is used, the decision logic of our predictor becomes trivial (it only has to look for a pattern or all ones), which should make it very fast. We believe that the simplicity and the relative low hardware cost combined with its superior performance make our predictor a prime candidate for integration into next generation CPUs.

We are currently working on a prediction outcome-based CE that dynamically decides which history patterns should cause a prediction, i.e., it does not need to be pre-programmed. Furthermore, we are adapting our CE to work with a predictor that stores more than one value per line.

Acknowledgments

This work was supported in part by the Hewlett Packard University Grants Program (including Gift No. 31041.1) and the Colorado Advanced Software Institute. We would like to especially thank Tom Christian for his support of this project and Dirk Grunwald and Abhijit Paithankar for providing and helping with the pipeline-level simulator.

References

[DEC92] Digital Equipment Corporation. *Alpha Architecture Handbook.* 1992.

[EuSr94] A. Eustace, A. Srivastava. *ATOM: A Flexible Interface for Building High Performance Program Analysis Tools.* WRL Technical Note TN-44, Digital Western Research Laboratory, Palo Alto. July 1994.

[Gab96] F. Gabbay. *Speculative Execution Based on Value Prediction.* EE Department Technical Report #1080, Technion - Israel Institute of Technology. November 1996.

[GKMP98] D. Grunwald, A. Klauser, S. Manne, A. Pleszkun. "Confidence Estimation for Speculation Control". *25th International Symposium on Computer Architecture*. June 1998.

[GrPa98] D. Grunwald, A. Paithankar. *A Comparison of the Benefits of Dependence Prediction and Value Prediction.* Submitted to the Fifth International Symposium On High Performance Computer Architecture. 1998.

[JRS96] E. Jacobsen, E. Rotenberg, J. Smith. "Assigning Confidence to Conditional Branch Predictions". *29th International Symposium on Microarchitecture.* December 1996.

[KMW98] R. E. Kessler, E. J. McLellan, D. A. Webb. "The Alpha 21264 Microprocessor Architecture". To appear in *1998 International Conference on Computer Design.* October 1998.

[LCB+98] 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.

[LWS96] 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*, 138-147. October 1996.

[McF93] S. McFarling. *Combining Branch Predictors.* TN 36, DEC-WRL. June 1993.

[Pai96] A. Paithankar. *AINT: A Tool for Simulation of Shared-Memory Multiprocessors.* Master's Thesis, University of Colorado at Boulder. 1996.

[SCAP97] 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*, 284-291. 1997.

[SLM95] S. Sechrest, C. C. Lee, T. Mudge. "The Role of Adaptivity in Two-level Adaptive Branch Prediction". *28th International Symposium on Microarchitecture.* 1995.

[SPEC95] *SPEC CPU'95*. August 1995.

[SmSo95] J. E. Smith, G. S. Sohi. "The Microarchitecture of Superscalar Processors". *Proceedings of the IEEE.* 1995.

[SrEu94] A. Srivastava, A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools". *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* ACM SIGPLAN 29(6):196-205. June 1994.

[SrWa94] A. Srivastava, D. W. Wall. "Link-time Optimization of Address Calculation on a 64-bit Architecture". *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* ACM SIGPLAN 29(6):49-60. June 1994.

[WaFr97] K. Wang, M. Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors". *30th Annual ACM/IEEE International Symposium on Microarchitecture.* December 1997.

[YePa92] T. Y. Yeh, Y. N. Patt. "Alternative Implementations of Two-level Adaptive Branch Prediction". *19th Annual International Symposium of Computer Architecture*, 124-134. May 1992.

[YePa93] 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*, 257-266. May 1993.