Journal of Instruction-Level Parallelism 1 (1999) Submitted 12/98; published 5/99

Prediction Outcome History-based Confidence Estimation for Load Value Prediction

Martin Burtscher BURTSCHE@CS.COLORADO.EDU
Benjamin G. Zorn ZORN@CS.COLORADO.EDU
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430

Abstract

Load instructions occasionally incur very long latencies that 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.

Current load value predictors can only correctly predict about forty to seventy percent of the fetched load values. To avoid the cycle-penalty for mispredictions in the remaining cases, confidence estimators are employed. They inhibit all predictions that are not likely to be correct.

In this paper we present a novel confidence estimator that is based on prediction outcome histories. Profiles are used to identify the high-confidence history patterns. Our confidence estimator is able to trade off coverage for accuracy and vice-versa with great flexibility and reaches an average prediction accuracy over SPECint95 of as high as 99.3%. Cycle-accurate pipeline-level simulations show that a simple last value predictor combined with our confidence estimator outperforms other predictors, sometimes by over 100%. Furthermore, this predictor is one of two predictors that yield a genuine speedup for all eight SPECint95 programs.

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 continues to widen, this latency will become even longer. Since loads are not only among the slowest but also among the most frequently executed instructions of current high-performance microprocessors [13], improving their execution speed should significantly improve the overall performance of the processor.

Fortunately, loads often do not fetch random sequences of values. Rather, many load instructions fetch the same values repeatedly, which makes them predictable [17]. About half of all the load instructions of the SPECint95 benchmark suite retrieve the same value that they did the previous time they were executed. This behavior, which has been demonstrated explicitly on a number of architectures, is referred to as *value locality* [6, 17].

A load instruction's memory-access can take many cycles to complete. As a consequence, all the instructions that directly or indirectly consume the load value are delayed, which decreases the performance. A load value predictor quickly provides a predicted value to those consumer instructions so that they can start executing without having to wait for the memory to return the requested value. If it later turns out that the predicted value was correct, some cycles are saved. If the prediction was incorrect, all the instructions that were given an incorrect value have to be executed again with the correct value. Restarting those instructions requires time. Hence, load value prediction only makes sense if it is often correct so that the saved cycles outweigh the extra cycles incurred by incorrect predictions.

For example, if a load instruction that takes ten cycles to access the memory is followed by ten consumer instructions that each take one cycle to execute, then it takes a total of twenty cycles to execute the eleven instructions on a single-issue processor. If a load value predictor that takes one cycle to make a prediction is added to this processor, that same sequence of eleven instructions can be executed in eleven cycles (almost twice as fast), given that the predicted value is correct. However, if the value is mispredicted, it might take 22 cycles to execute the eleven instructions, ten cycles for the memory access to complete, one cycle to detect the misprediction, one cycle to reset the consumer-instructions, and finally ten more cycles to execute the ten instructions with the correct value.

Empirically, papers have shown that the results of most instructions are predictable [6, 15, 24]. However, of all the frequently occurring, result-generating instructions, loads are the most predictable [15] and incur the longest latencies. Since only about every fifth executed instruction is a load, predicting only load values requires significantly fewer predictions than predicting every instruction and leaves more time to update the predictor. As a consequence, smaller and simpler predictors can be used, which is why we believe that predicting only load values may well be more cost effective than predicting the result of every instruction.

Context-based load value predictors try to exploit the existing value locality by retaining previously seen load values. The simplest such predictor always makes a prediction using the previously fetched value of that same load instruction. We call this scheme *Basic LVP* (last value predictor).

To reduce the number of mispredictions and to avoid the cost associated with them, context-based predictors normally contain both a *value predictor* and a *confidence estimator* (CE) to decide whether or not to make a prediction. All previously proposed predictors and our own contain these two parts in some form. 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 necessary information to make a correct prediction. In such a case, it is better not to make a prediction because incorrect predictions incur a cycle penalty (for undoing the speculation) whereas making no prediction does not.

CEs are similar to branch predictors in the sense that both have to make binary decisions (predictable or not-predictable and branch taken or not-taken, respectively). Therefore, we chose to investigate whether some of the ideas that work well for branch predictors could be used as CEs for load value predictors.

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 outcomes (success or failure) [27]. The intuition is that the past prediction behavior tends to be very indicative of what will happen in the near future. For example, if an instruction was successfully predicted the last few times, there is a good chance that the next prediction will be successful, too. Hence, the *prediction outcome history*, as we call it, represents a measure of confidence.

Such histories consist of a short bit-pattern in which every bit indicates whether the corresponding prediction was correct. For instance, the left-most bit represents the most recent prediction outcome, the next bit to the right the second most recent outcome, etc. A correct prediction is recorded by a one and an incorrect prediction by a zero.

Saturating counters are also used as confidence estimators. Such counters can count up and down within two boundaries, say zero and 15. If the counter has reached 15, counting up will not change its value. Likewise, counting down from zero leaves the value at zero. Similar to prediction outcome histories, saturating counters can be used to record how many correct predictions were made in the recent past. The higher the value of the counter, the higher the confidence that the next prediction will be correct since a high value means that many of the most recent predic-

tions were correct.

The difference between counters and histories is that counters do not retain any sequence information and they suffer from saturation effects. For example, it is hard to distinguish two correct predictions that are followed by two incorrect predictions from two incorrect predictions followed by two correct predictions using counters because the total number of correct and incorrect predictions is the same in both cases. Furthermore, an incorrect prediction followed by a correct prediction will have no overall effect on a counter that was already at its top for it will be decremented and then incremented back to the top value. Since the counter would also be at its top after a long series of correct predictions, the information that the second to last prediction was incorrect is lost after only one correct prediction due to saturation of the counter.

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. Because branch prediction requires a similar mechanism, most modern microprocessors already contain the necessary hardware to perform this kind of speculation [6].

Unfortunately, branch misprediction recovery hardware 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 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). Consequently, a better recovery mechanism for load misspeculation would only re-execute 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.

We devised a load value predictor with a prediction outcome history-based confidence estimator that performs better than predictors with counter-based CEs. Profiles are used to program the predictor with the history patterns that should trigger a prediction. The predicted value is always the value that was previously loaded by the same load instruction. Our predictor reaches a harmonic mean speedup over SPECint95 of 12.0% with a re-execute policy and 9.3% with a re-fetch recovery policy. It is able to attain a positive speedup for all the SPECint95 programs, even with the simpler re-fetch mechanism. Of all the other predictors we looked at, only one does not slow down at least three of the eight benchmark programs when re-fetch is employed, as our detailed pipeline-level simulations revealed. Section 6.2 provides more results.

We also investigated using saturating counters instead of profiling to identify the history patterns that should trigger a prediction [1], hoping that the adaptivity of the counters would result in better performance. Surprisingly, the profile-based approach presented in this paper outperforms the dynamic, counter-based approach for more than half the benchmark programs and yields higher accuracies and a much broader range of possible accuracy-coverage pairs. Furthermore, the profile-based predictor requires less hardware, is simpler in its design (and therefore potentially faster), and requires only one-level predictor updates.

The remainder of this paper is organized as follows: Section 2 introduces the predictor architecture and nomenclature. Section 3 illustrates the use of prediction outcome histories and introduces our predictor. Section 4 presents related work. Section 5 explains the methods used. Section 6 presents the results. Section 7 concludes the paper with a summary.

2. Predictor Architecture

Figure 1 shows the components of a context-based load value predictor with a confidence estimator. The largest element is an array of 2^n lines for storing the confidence information and the previously fetched values. Clearly, this "cache" has to be very fast or there would be no performance advantage over accessing the conventional memory. The hashing hardware generates 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.

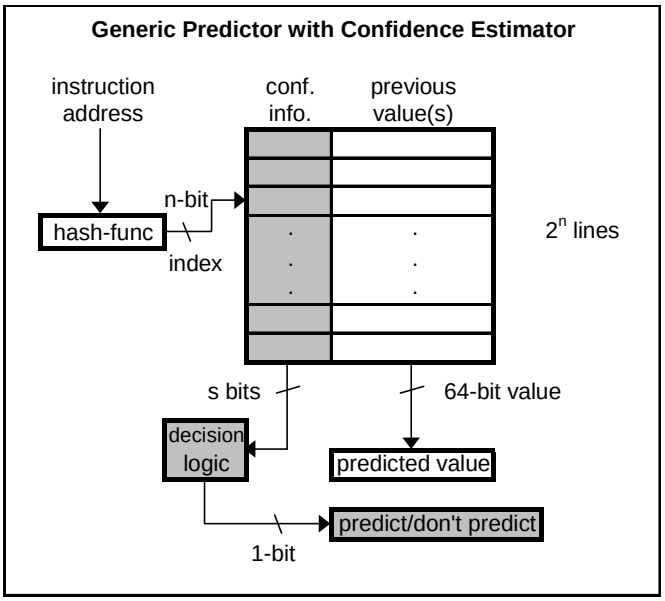

Figure 1: The components of a load value predictor with a confidence estimator (shaded).

All the predictors in this paper use *PC div 4 mod* 2^n as a hash-function. The *div 4* eliminates the two least significant bits that are always zero since the processor we use requires instructions to be word aligned. Hence, the hash-function extracts the *n* least significant bits from the *PC* excluding bits zero and one. Adding a more complex hash-function may result in somewhat less aliasing but will most likely increase the length of the critical path, which is why this very simple but quite effective approach is normally used.

When a prediction needs to be made, the hash-function computes an index to select one of the predictor lines. The value stored in the selected line becomes the predicted value. If there are multiple values, a selector first has to determine which value to use. Finally, the decision logic decides whether a prediction should be attempted with this value.

Once the outcome of a prediction is known, the corresponding confidence information field is updated to reflect the new outcome, and the true load value is stored in the line that was used for making the prediction.

While most other predictors use saturating counters [17, 25, 31], we propose keeping prediction outcome histories as confidence information. Which history patterns should trigger a prediction and which ones should not is determined by profile runs (see Section 3.1) and the decision logic is preprogrammed accordingly.

3. Using Prediction Outcome Histories

Histories that record the recent prediction successes and failures are a very successful idea for branch prediction [27]. We found the same to be true in the domain of load value prediction. In fact, prediction outcome histories seem to be better suited for load value prediction than other approaches. For example, they yield higher accuracies, allow accuracy-coverage pairs to be chosen at a finer granularity, and result in better speedup than saturating counters. We believe the reason is that histories, as opposed to counters, retain sequence information and do not suffer from saturation effects.

If such histories are to be used as a measure of confidence, it is necessary to know which ones are (normally) followed by a successful prediction and which ones are not. Heuristics and algorithms to do this exist in the branch prediction literature. For instance, Sechrest et al. [28] describe a scheme that tries to identify repeating patterns of branch outcomes. If no repeating pattern can be detected, a simple population count is used. They call this scheme *algo*. As an alternative, Sechrest et al. suggest running a set of programs and recording the behavior. They call this profile-based approach *comp*. We use *comp* for our predictor since it performs considerably better and is much more flexible than *algo*.

To better explain how the two approaches work, we present Table 1. It shows the output of a 4-bit history run based on SPECint95 behavior and the resulting configurations of two *comp* schemes and one *algo* scheme. The second row of the table, for example, states that a *failure, failure, failure, success* history (denoted by *0001*) is followed by a successful last value prediction 26.9% of the time. In this history, *success* denotes the outcome of the most recent prediction. Of all the encountered histories, 2.7% were *0001*. All three CE schemes do not allow a last value prediction to take place following this particular history pattern—the two *comp* schemes because the predictability is both under 60% and under 90% and the *algo* scheme because there is no discernable repeating bit-pattern in the history and the population count is in favor of a zero.

SPECint95 Last Value Predictability					
history	predictability	occurrence	*comp 60%*	*comp 90%*	*algo*
0000	6.9%	32.2%	no	no	no
0001	26.9%	2.7%	no	no	no
0010	19.1%	2.9%	no	no	no
0011	49.9%	1.6%	no	no	yes
0100	34.3%	2.9%	no	no	no
0101	33.6%	1.9%	no	no	no
0110	44.9%	1.3%	no	no	yes
0111	59.4%	2.2%	no	no	yes
1000	24.2%	2.7%	no	no	no
1001	46.3%	1.8%	no	no	yes
1010	66.8%	1.9%	yes	no	yes
1011	66.1%	1.9%	yes	no	yes
1100	53.1%	1.6%	no	no	no
1101	57.2%	1.9%	no	no	yes
1110	52.3%	2.2%	no	no	yes
1111	96.6%	38.3%	yes	yes	yes

Table 1: Predictability and occurrence split up by history pattern. The predictability signifies the percentage of last value predictable loads following the given 4-bit prediction outcome histories. The occurrence denotes the percentage of the time the respective history was encountered. The three rightmost columns show how three confidence estimators would be configured based on the information on the left.

The table shows the average over all eight programs and is for illustration purposes only. The results presented in the subsequent sections were generated using cross-validation (Section 5.4).

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 predictable/not-predictable threshold can be set anywhere. The optimal setting strongly 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., as is the case with a re-execute architecture), it is most likely wiser to predict a larger number of load values, albeit also a somewhat larger number of incorrect ones. If, on the other hand, the misprediction penalty is high and should therefore be avoided (e.g., as is the case with a re-fetch architecture), it makes more sense not to predict quite as many loads but to be confident that the ones that are predicted will be correct.

If we want to be highly confident that a prediction is correct, say at least 90% confident, the decision logic would only allow predictions for histories whose predictability is greater than 90%, i.e., only for history *1111* based on the data in Table 1. This threshold would result in 38.3% of all loads being predicted (of which 96.6% would be correct). In other words, an ideal *comp* confidence estimator with a threshold of 90% yields a 96.6% prediction accuracy and a 38.3% coverage for our benchmark suite.

As the example illustrates, four history bits are enough to reach an average accuracy in the high nineties. With longer histories, our approach yields even higher accuracies and better coverages. In the result section we show that ten-bit histories with thresholds of 67% and 86% work well for our processor.

A closer look at Table 1 reveals that *algo* corresponds most closely to *comp* with about a 44% threshold. This indicates that *algo* will probably not perform well because its "threshold" is too low. Furthermore, defaulting to "predict" (denoted by "yes") in case the population count results in a tie is obviously a poor choice and should be changed if *algo* is to be used.

3.1 The SSg(comp) Last Value Predictor

Our load value predictor 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 predictor-lines (see Figure 1). Since the resulting confidence estimator is similar to Yeh and Patt's *SSg* branch predictor [33], programmed with Sechrest et al.'s *comp* approach, we call our predictor *SSg(comp) LVP*.

Predictions are performed as described in Section 2, i.e., predictions are made if the prediction outcome history of the current load instruction is one of the histories the profile information indicated should be followed by a prediction. Once the outcome of a prediction is known, a new bit is shifted into the history of the corresponding line in the predictor and the oldest bit is shifted out (lost). If the true load value is equal to the value in the predictor, a one is shifted in, otherwise a zero is shifted in. Then the value in the line is replaced by the true value.

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

Note that, unlike instruction and data caches, load value predictors do not have to be correct all the time. Hence, neither tag nor valid bits are a requirement. We decided to omit both since having them only results in a small increase in accuracy, which we believe does not justify the extra hardware and complexity.

Note that our predictor requires no content addressable memory and that all the operations can be performed using at most two table lookups. This is similar to current branch predictors [32] and should therefore not affect the cycle time.

The only external input to our predictor is the PC of the load instruction. Since the PC of every instruction is available from the first pipeline stage on, predictions can be made in any stage. Also, the prediction mechanism works autonomously and can therefore be run in parallel with any other activity that might be going on in the CPU. Since the predicted values are not needed before the execute stage, the predictions could even be pipelined (take more than one cycle) over the fetch and decode stages.

With such an architecture, only one prediction or update can be made per cycle. If more than one access per cycle is needed, the prediction hardware needs to be split into several independent banks, which would allow multiple predictions or updates to be performed in parallel.

It is possible that a next prediction needs to be made before the previous one has updated the predictor. Of course, this only poses a problem if both predictions go to the same line and is only likely to happen in tight loops where the same load instruction is executed repeatedly with few intervening instructions.

We can think of two possible remedies. Either the predictor lines could be marked as "in use" and further predictions will stall until the lines have been updated, or further predictions could be made using the old information. We leave the investigation of the performance impact of these two schemes to future work. Nevertheless, we believe that the latter will perform much better since we found that loads only infrequently change behavior from being predictable to not being predictable or vice-versa. In fact, about half of all executed loads belong to load instructions that are ≥95% last-value predictable or ≥95% not predictable. Furthermore, we measured a geometric mean of about 65 load instructions between any two loads that go to the same line in the 4096-line predictor, indicating that in most cases the required line is up-to-date.

We believe that the proposed predictor is likely to be easily integrated into any CPU that already supports speculative execution. The predictor works with any instruction set and requires no changes to the instruction set architecture, such as adding bits to the op-code.

4. Related Work

In this section we try to give a detailed overview over the current load value prediction literature.

Early Work: Two independent research efforts [6, 17] first recognized that load instructions exhibit *value locality* and concluded that there is potential for prediction.

Lipasti et al. [17] investigated why load values are often predictable and how predictable the different kinds of load instructions are. While all types of loads exhibit significant value predictability, it turns out that address loads have slightly better value locality than data loads, instruction address loads hold an edge over data address loads, and integer data values are more predictable than floating-point data values.

In a follow-up paper, Lipasti and Shen [15] broaden their scope to predicting all result-generating instructions and show how value prediction can be used to exceed the existing instruction level parallelism (ILP). They found that using a value predictor delivers three to four times more speedup than doubling the data cache (same hardware increase) and they argue that a value predictor is unlikely to have an adverse effect on processor cycle time, whereas doubling the data-cache size probably would. Furthermore, they note that loads are the most predictable frequently executed instructions.

Gabbay's dissertation proposal [6] mostly discusses general value prediction and how to boost the ILP beyond the data-flow limit, but he also studies load value prediction by itself.

Load Value Predictors: Lipasti et al. [17] describe a last value predictor (predicts the last seen load value) that uses 2-bit saturating up/down counters to classify loads as unpredictable, predictable, or constant. In Section 6.2, we compare our predictor with a predictor similar to theirs.

Gabbay [6] proposes four predictor schemes: a tagged last value predictor, a tagged stride predictor, a register-file predictor, and a sign-exponent-fraction (SEF) predictor. We compare our predictor to the tagged last value predictor in Section 6.2. However, we refrain from comparing against the register-file and the SEF predictor because the former performs very poorly and the latter can only be used for floating-point loads. We include a stride predictor in our comparison as part of a hybrid predictor.

Wang and Franklin [31] are the first to propose a two-level prediction scheme and the first to make predictions based on the last four distinct values rather than based on just the last value. They further propose a hybrid predictor that combines their last four distinct value predictor with a stride predictor, which has the highest prediction accuracy of all the predictors in the literature. However, our predictor significantly outperforms theirs when a re-fetch misprediction policy is used (see Section 6.2).

Sazeides and Smith [24] perform a theoretical limit study of the predictability of data values. They investigate the performance of three models: last value, stride, and finite context. Their finite context predictor predicts the next value based on a finite number of preceding values by counting the occurrences of a particular value immediately following a certain sequence of values. In a follow-up paper [25], Sazeides and Smith design an implementable two-level value predictor based on the finite context method. They found that their predictor outperforms other, simpler predictors, but only when large tables are used.

We include a finite context method predictor in our comparison as part of the hybrid predictor proposed by Rychlik et al. [23]. It combines a stride 2-delta [24] with a finite context method-based (FCM) predictor. The FCM predictor stores entire sequences of fetched load values. Upon prediction it tries to identify the current position in the sequence and uses the next value from the sequence to make a prediction. The stride 2-delta predictor is an improved version of the conventional stride predictor. It maintains two separate strides. The stride used for making predictions is only updated if a new stride has been seen twice in a row, which significantly reduces the number of mispredictions.

Profiling: Gabbay and Mendelson [7] explore the possibility of using program profiling to enhance the efficiency of value prediction. They use their profiling results to insert opcode directives that allow them to allocate only highly predictable values, which reduces the amount of aliasing. However, even manual fine-tuning of the user supplied threshold value does not allow them to outperform their hardware-only predictor in all cases. They found that different input sets result in very similar behavior, which makes profiling appropriate to use in this context.

Calder et al. [2] examine the invariance found from profiling instruction values and propose a new type of profiling called *convergent profiling*, which is much faster than conventional profiling. Their measurements reveal that a significant number of instructions (including loads) generate only one value with high probability. They found that the invariance of load values is crucial for the prediction of other types of instructions (by propagation). They further found that the invariance is quite predictable, even across different sets of inputs.

We also use profiling. However, the novelty of our approach is that we do not profile actual load values but the success-rate of a last value predictor with respect to its recent prediction behavior (see Section 3). The result is then used to configure the confidence estimator rather than to modify executables. Note that once our confidence estimator has been configured, no further profiling is required.

As mentioned in the introduction, we also investigated using saturating counters instead of profiling [1]. While the average speedup of this dynamic predictor is slightly higher than the speedup delivered by the profile-based predictor, the profile-based predictor is simpler in design and yields higher speedups for more than half of the benchmark programs.

Dependence Prediction: In their next paper [16], Lipasti and Shen add dependence prediction to their predictor and switch to predicting source operand values rather than instruction results. The latter decouples dependence detection from value-speculative instruction dispatch. They found their approach to be particularly effective in wide and deeply pipelined machines.

Reinman and Calder [22] also examine dependence prediction and conclude that, due to its small hardware requirement, dependence prediction should be added to new processors first even though value prediction provides the larger performance improvement. Furthermore, they found that both address prediction and memory renaming are inferior to dependence and value prediction.

In another paper [21], Reinman et al. propose a software-guided approach for identifying dependencies between store and load instructions and devise an architecture to communicate these dependencies to the hardware. Their approach requires changes to the instruction set architecture.

Other Related Work: Rychlink et al. [23] address the problem of useless predictions. They introduce a simple hardware mechanism that inhibits predictions that were never used (because the true value became available before the predicted value was needed) from updating the predictor, which results in improved performance due to reduced predictor pollution. Unfortunately, the prediction outcome histories we use rely on seeing all updates.

In their next paper [8], Gabbay and Mendelson show that the instruction fetch bandwidth has a significant impact on the efficiency of value prediction. They found that value prediction (of one-cycle latency instructions) only makes sense if producer and consumer instructions are fetched during the same cycle. Hence, 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 and simpler predictors while still reaping most of the potential.

Gonzalez and Gonzalez [10] found that the benefit of data value prediction increases significantly as the instruction window grows, indicating that value prediction will most likely play an important role in future processors. Furthermore, they observed an almost linear correlation between the predictor's effective accuracy (the percentage of all loads that are correctly predicted) and the resulting performance improvement. Our results in Section 6.2.1 show that our predictor yields much higher accuracies (with the same coverage) than other predictors from the literature do, which indeed results in higher speedups, in particular with a re-fetch architecture.

Fu et al. [5] propose a hardware and software-based approach to value speculation that leverages advantages of both hardware schemes for value prediction and compiler schemes for exposing instruction level parallelism. They propose adding new instructions to load values from the predictor and to update the predictor. We currently only look at transparent prediction schemes, that is, predictors that do not require changes to the instruction set architecture.

A more detailed study about predictability by Sazeides and Smith [26] illustrates that most of the predictability originates in the program control structure and immediate values, which explains the often observed independence of program input. 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 [10] did indeed observe such an improvement in their study.

Confidence Estimation: Jacobsen et al. [11] and Grunwald et al. [9] introduce confidence estimation to the domain of branch prediction and multi-path execution to decide whether or not to make a prediction. We adopt their metrics for load value prediction (Section 5.2). While their goals are similar to ours, the approaches for branch confidence estimation and load value prediction differ. In particular, their confidence estimator (a two-bit saturating up/down counter, which is what Lipasti et al. [17] use) does not yield very good results when applied to load value prediction (Section 6.2).

Branch Prediction: In the area of branch prediction, a significant amount of related work exists. Lee and Smith [14] keep a *history* of recent branch directions for every conditional branch and systematically analyze every possible pattern.

Yeh and Patt [32, 33] and Pan, So, and Rahmeh [20] describe sets of two-level branch predictors and introduce a taxonomy to distinguish between them. Their predictors consist of a branch history register (BHR) and a pattern history table (PHT). Three-letter combinations are used to describe the two components. By convention, the first two letters are uppercase and the third letter is lowercase.

The first letter characterizes the BHR. If all branches share a common BHR, a *G* is used for global. If every branch has its own BHR, a *P* is used for per-address. If sets of branches are mapped to individual BHRs, an *S* is used for set. Note that *G* and *P* represent the two extremes of the set case. *P* means all sets have size one and *G* corresponds to one set containing all the branches.

The second letter specifies whether the PHT is adaptive. *S* stands for static, indicating that the PHT entries are fixed. *A* stands for adaptive, meaning that the PHT entries can be modified on-the-fly.

The third (lowercase) letter is identical to the first letter, except it describes the PHT instead of the BHR. Hence, *g* means one global PHT, *s* means one PHT per set, and *p* means one PHT per address.

Sechrest, Lee, and Mudge extend this nomenclature [28]. They distinguish between two ways of programming the fixed entries of the non-adaptive PHT schemes. Profile-based is denoted by appending *comp* to the three letters and algorithm-based is denoted by appending *algo*.

We adopt the *SSg(comp)* design for use as a confidence estimator in our load value predictor. This means that sets of load instructions (i.e., loads that have the same PC modulo the predictor height) are mapped to the same line in the predictor (the lines correspond to the BHRs). Furthermore, the decision logic (which corresponds to the PHT) is static and cannot be changed during program execution. Finally, there is only one global decision logic, which is programmed using the *comp* approach.

Sprangle et al. [27] describe a technique called *agree prediction*, which reduces the chance that items mapped to the same predictor slot will interfere negatively. They achieve this by recording whether the previous predictions were a success or failure instead of whether the branches were taken or not. We use the same technique.

Summary: The novelty of our predictor is that it uses prediction outcome histories, an idea taken from the branch prediction literature, for confidence estimation instead of saturating counters. While others have used profiling [2, 7], our use of profiling is different in so far that we do not profile actual load values and do not need to modify the binaries in any way.

5. Methodology

All our measurements are performed on the DEC Alpha AXP architecture [3]. To perform a thorough design-space evaluation, we instrumented our benchmark suite using the ATOM tool-kit

[4, 30]. This allowed us to efficiently simulate the proposed predictor in software and to identify good configurations. The most promising configurations were then fed to our cycle-accurate pipeline-level simulator for detailed evaluation.

To obtain actual speedup results, we use the AINT simulator [19] with its out-of-order back-end, which is configured to emulate a high performance microprocessor similar to the DEC Alpha 21264 [12]. In particular, the simulated 4-way superscalar 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 modeled latencies are given in Table 2. Operating system calls are executed but not simulated. Loads are only allowed to issue when all prior store addresses are known. The six functional units are fully pipelined and each unit can execute all operations in its class. Furthermore, up to four load instructions can issue per cycle. As a consequence, all the load value predictors used in this study are split into four banks that can operate in parallel. Since the modeled CPU fetches naturally aligned four-tuples of instructions, it is not possible to fetch or issue two load instructions during the same cycle that go to the same predictor bank. Load value predictions take place during the rename-stage in the instruction pipeline.

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 2: The functional unit and memory latencies (in cycles) of our simulated processor.

5.1 Benchmarks

We use the eight integer programs of the SPEC95 benchmark suite [29] 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 [13]. Table 3 gives relevant information about the SPECint95 programs.

The suite includes two sets of inputs for every program and allows two levels of optimization. To acquire as many load value samples as possible we use the larger reference inputs. However, due to a restriction in our simulation infrastructure, only the first of the multiple input-files from the reference set was used with gcc. We ran the more optimized peak-versions of the programs that were compiled with DEC GEM-CC using the highest optimization level "*-migrate -O5 -ifo*". The optimizations include common sub-expression elimination, split lifetime analysis, code scheduling, nop insertion, code motion and replication, loop unrolling, software pipelining, local and global inlining, inter-file optimizations, and many more. Furthermore, the binaries were statically linked which allows the linker to perform additional optimizations that considerably reduce the number of run-time constants that are loaded during execution. For ATOM simulations, all programs are run to completion. The result is approximately 87.8 billion executed load in-

structions. Note that the few floating point load instructions contained in the binaries are also measured, that loads to the zero-registers are ignored, 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.

For the speedup measurements, we executed each of the eight 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 representative of the general program behavior [22]. The rightmost column of Table 3 shows the number of instructions that were skipped. gcc is completely executed (334 million instructions) since this amounts to the full compilation of the first reference input-file.

Information about the SPECint95 Benchmark Suite											
	total executed		load	load sites that account for				predictability		million instrs	
program	load instructions		sites	Q50	Q90	Q99	Q100	last val	l4d vals	skipped	simul.
compress	10,537 M	(17.5%)	3,961	0.4%	1.5%	2.0%	17.4%	40.4%	41.3%	6,000	300
gcc	80 M	(23.9%)	72,941	1.2%	7.4%	19.4%	47.1%	48.5%	69.1%	0	334
go	8,764 M	(24.4%)	16,239	1.3%	10.5%	26.0%	76.0%	45.9%	68.1%	12,000	300
ijpeg	7,141 M	(17.2%)	13,886	0.3%	1.3%	3.0%	24.9%	47.5%	56.2%	1,000	300
li	17,792 M	(26.7%)	6,694	0.6%	2.1%	4.7%	28.9%	43.4%	66.6%	4,000	300
m88ksim	14,849 M	(17.9%)	8,800	0.6%	2.5%	5.2%	30.4%	76.1%	85.9%	1,000	300
perl	6,207 M	(31.1%)	21,342	0.2%	0.8%	1.1%	16.8%	50.7%	86.8%	1,000	300
vortex	22,471 M	(23.5%)	32,194	0.2%	1.8%	10.3%	51.7%	65.6%	82.3%	5,000	300
average		(22.8%)		0.6%	3.5%	9.0%	36.6%	52.3%	69.5%		

Table 3: The number of load instructions contained in the binaries (load sites) and executed by the individual programs (in millions) of the SPECint95 benchmark suite. The numbers in parentheses denote the percentage of all executed instructions that are loads. The quantile columns show the percentage of load sites that contribute the given fraction (e.g., Q50 = 50%) of executed loads. The two predictability columns show the percentage of loads that fetch a value that is identical to the last fetched value or identical to one of the last four distinct fetched values. The two rightmost columns show the number of instructions (in millions) that are skipped before starting the pipeline-level simulations and the number of simulated instructions.

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 the high optimization level and good register allocation.

Another interesting point is the relatively small number of load sites that contribute most of the executed load instructions. For example, only 3.5% of the load sites contribute 90% of the executed loads. Less than 37% of the load sites are visited at all during execution.

In these benchmark programs, an average of 52.3% of the load instructions fetch the same value that 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.

5.2 Averaging Speedups

In this paper, the term *speedup* denotes how much faster a processor becomes when a load value predictor is added to it.

To obtain the speedup delivered by a load value predictor for a given program, the program is executed on both a baseline CPU (CPU_{Base}) and a CPU with a load value predictor (CPU_{LVP}). The speedup is the runtime on CPU_{Base} divided by the runtime on CPU_{LVP}. To be independent of the clock speed the runtime is normally measured in cycles rather than seconds.

$$speedup = \frac{runtime_{without\ predictor}}{runtime_{with\ predictor}} = \frac{cycles_{without\ predictor}}{cycles_{with\ predictor}}$$

Since a speedup of one indicates no improvement in performance, the *speedup over baseline* is often easier to understand. It is defined as the regular speedup minus one. Hence, the speedup over baseline is positive if the load value predictor improves the execution speed and negative if it slows the execution down. Note that the regular speedup is always positive.

$$speedup\ over\ baseline = speedup - 100\%$$

To better estimate the expected performance improvement that a load value predictor will deliver, the speedup over more than one program is usually measured. This is done because a suite of programs is assumed to exhibit a more "average" program-behavior than an individual program.

Once the individual speedups have been obtained, they need to be combined into a single speedup. Several approaches to combining (or averaging) speedups can be found in the literature, the most prominent of which are the *harmonic mean*, the *geometric mean*, and the *arithmetic mean*. The harmonic mean always yields the lowest and therefore the most conservative result. Since the arithmetic mean always produces the highest result, the geometric mean is sometimes used as a compromise.

Intuitively, the combined speedup should be equal to the speedup over the single program *P* that does nothing but run the benchmark programs one after the other (in any order). However, to avoid over-representing longer running programs, *P* must execute all the programs for the same amount of time. This corresponds to weighing (i.e., normalizing) the individual benchmark programs with the inverse of their runtimes.

The runtimes can be normalized for either CPU_{Base} or CPU_{LVP}. If the normalization is done for CPU_{Base}, the combined speedup evaluates to the *harmonic mean* of the individual speedups. If, on the other hand, the normalization is done for CPU_{LVP}, the combined speedup turns out to be the *arithmetic mean* of the individual speedups.

For example, let us assume a benchmark suite consisting of the two programs *A* and *B* that require c_a and c_b cycles, respectively, to execute on the baseline CPU. Let us further assume a load value predictor *L* that speeds up program *A* by a factor of 10 and program *B* by a factor of one (i.e., *B*'s runtime remains the same). The runtimes on CPU_{LVP} are consequently $.1*c_a$ and c_b.

When normalizing for CPU_{Base}, the combined speedup should be equal to the speedup of the program *P* that executes *A* $\boldsymbol{c_b}$ times and *B* $\boldsymbol{c_a}$ times. Doing so takes $c_a*\boldsymbol{c_b}+c_b*\boldsymbol{c_a} = 2*c_a*c_b$ cycles on the baseline CPU (both programs are executed for c_a*c_b cycles). When predictor *L* is added, the total runtime becomes $.1*c_a*\boldsymbol{c_b}+c_b*\boldsymbol{c_a} = 1.1*c_a*c_b$ cycles. The resulting speedup is therefore $2/1.1 \approx 1.818$, which is equal to the harmonic mean of the two individual speedups.

When normalizing for CPU_{LVP}, program *P* needs to execute *A* $\boldsymbol{c_b}$ times and *B* $\boldsymbol{.1*c_a}$ times. This takes $.1*c_a*\boldsymbol{c_b}+c_b*\boldsymbol{.1*c_a} = .2*c_a*c_b$ cycles on CPU_{LVP} and $c_a*\boldsymbol{c_b}+c_b*\boldsymbol{.1*c_a} = 1.1*c_a*c_b$ cycles on the baseline processor. The speedup now evaluates to $1.1/.2 = 5.5$, which is the arithmetic mean of the individual speedups. For reference, the geometric mean speedup is $10^{(1/2)} \approx 3.162$.

As the example illustrates, normalizing for CPU_{LVP} weighs program *B*, which cannot be sped

up by the LVP, ten times less heavily than when normalizing for CPU_{Base} (the weights are shown in bold face). In general, the more a program can be sped up the relatively more weight it is given when using the arithmetic mean to compute the combined speedup. Hence, the arithmetic mean speedup assumes the "average" program to contain proportionately more code that benefits from a load value predictor than code that does not. We do not believe this to be a valid assumption, which is why all the averaged speedups presented in this study are harmonic mean speedups.

5.3 Metrics for Load Value Predictors

The ultimate metric for comparing load value predictors is of course the speedup attained by incorporating them into a CPU. Unfortunately, the speedups are quite dependent on the architectural features of the underlying CPU. This is why non-implementation specific metrics are also important.

A value predictor with a *confidence estimator* can produce four prediction outcomes: correct prediction PCORR, incorrect prediction PINCORR, correct non-prediction NPCORR (no prediction was made, and the guessed value would not have been correct), and incorrect non-prediction NPINCORR (no prediction was attempted even though the guessed value would have been correct). PCORR, PINCORR, NPCORR, and NPINCORR denote the number of times each of the four cases is encountered. To make the four numbers independent of the total number of executed load instructions, they are normalized such that their values sum to one.

Normalization: Pcorr + Pincorr + NPcorr + NPincorr = 1

Unfortunately, the four numbers by themselves 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 does, i.e., predictor A makes both more correct and more incorrect predictions than predictor B. This is why 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 [9, 11]. These metrics are all higher-is-better metrics.

- $Potential: P_{OT} = P_{CORR} + NP_{INCORR}$
- $Accuracy: A_{CC} = \frac{P_{CORR}}{P_{CORR} + P_{INCORR}}$
- $Coverage: C_{OV} = \frac{P_{CORR}}{P_{CORR} + NP_{INCORR}} = \frac{P_{CORR}}{P_{OT}}$

The potential represents the fraction of all load values that are predictable, which is a property of the value predictor alone since predictor updates are not controlled by the confidence estimator. However, if the potential is low, even a perfect confidence estimator is unable to make many correct predictions.

The accuracy represents the probability that an attempted prediction is correct, and the coverage 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 more correct predictions (which save cycles) than incorrect predictions (which cost cycles), whereas a high coverage merely means better utilization of the existing potential. Nevertheless, a high coverage is still desirable.

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

5.4 Cross-Validation

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

6. Results

The following subsections list the results. Section 6.1 evaluates the performance of our *SSg(comp)* confidence estimator. In Section 6.2 we compare our predictor with a number of predictors from the literature. To better explore the parameter space, we only show averages over the eight benchmark programs and not the individual programs. For improved readability, several figures in the following subsections are not zero-based.

6.1 *SSg(comp)* Confidence Estimator Results

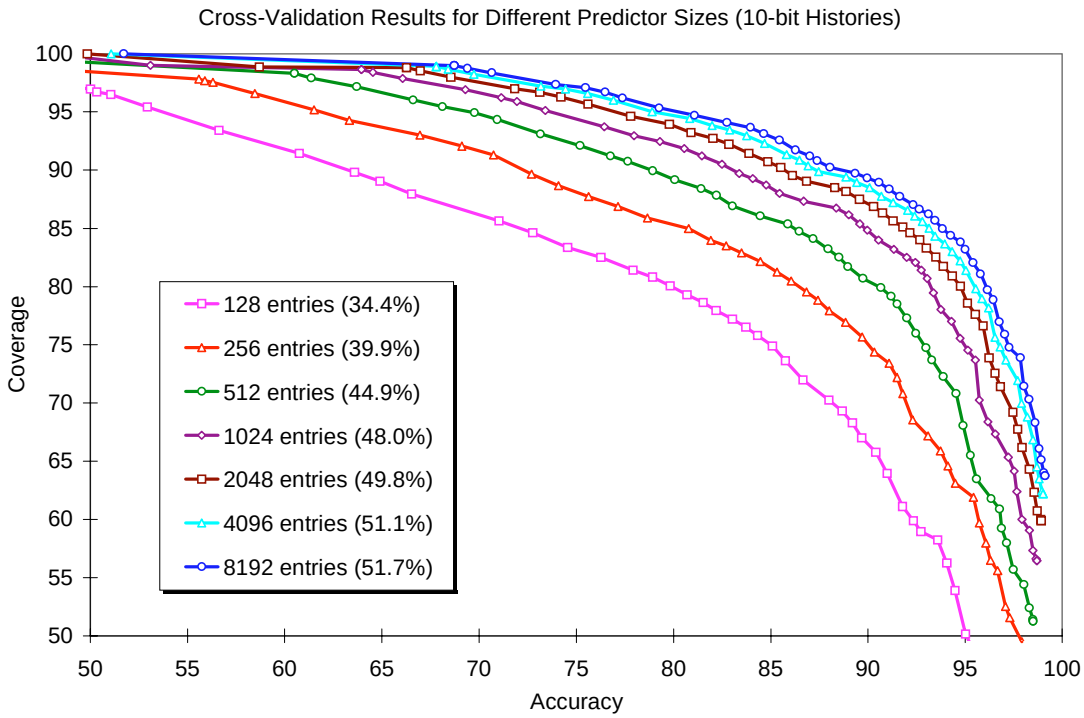

Figure 2: Accuracy-coverage pairs for different predictor sizes and 10-bit histories. Each dot corresponds to a threshold (in 2% increments). The numbers in parentheses denote the potential delivered by the respective load value predictor.

The results for the *SSg(comp)* confidence estimator are generated using cross-validation (Section 5.4). Furthermore, all predictor entries are set to zero before every run.

Figure 2 shows the attainable accuracy-coverage pairs for different predictor 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.

Each curve was generated by varying the prediction threshold. Each point in the lines corresponds to a threshold setting, starting at 98% (from the right) and decreasing in 2% steps.

The broad range in both dimensions is quite apparent and, hardly surprising, the larger the predictor the better its performance. As expected, there is a trade-off between the accuracy and the coverage. Nonetheless, both the performance of the CE and the delivered potential saturate at about 4096 entries. Apparently, a 4096-entry predictor is big enough for our benchmark suite and has a performance that is close to the performance of an infinitely large predictor. This was to be expected based on the quantile numbers from Table 3.

Figure 3 is similar to Figure 2, except the predictor size is held constant (1024 entries) and the length of the histories is varied.

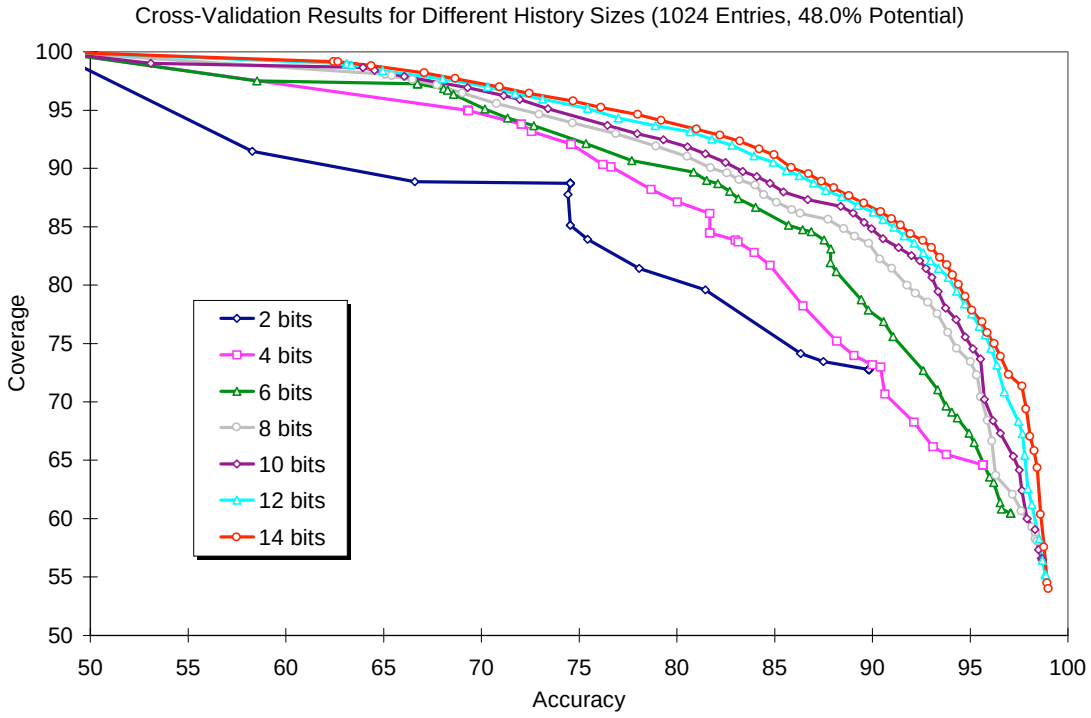

Figure 3: Accuracy-coverage pairs for different history sizes and 1024-entry predictors. Each dot corresponds to a threshold (in 2% increments).

The figure shows that longer histories perform better, but saturation sets in at about ten bits. These results indicate that ten history bits provide good prediction potential over a range of thresholds while keeping the number of bits low.

Note that we performed a much broader investigation of the parameter space, i.e., we studied

varying history sizes for predictors with fewer and more than 1024 lines as well as varying predictor sizes for predictors with fewer and more than ten-bit histories. We show Figure 2 and Figure 3 because they are representative of the generally observed behavior.

6.2 Predictor Comparison

In this section we compare several confidence estimators and load value predictors: a *Basic LVP* (without a confidence estimator), a *Tagged LVP* [6], a *Bimodal LVP* [17], an optimized *Tagged Bimodal LVP*, our *SSg(comp) LVP*, an *SSg(algo) LVP*, a *Last Distinct 4 Values* predictor [31], a *Last Distinct 4 Values + Stride* predictor [31], and a *Stride 2-Delta + Finite Context Method* predictor [23]. 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 of them are allowed to hold 2048 values plus whatever else they require to support that size. This results in roughly 20 kilobytes of state, which we find reasonable given that the DEC Alpha 21264 microprocessor incorporates two 64-kilobyte L1 caches on chip [12]. Table 4 shows the hardware requirement of the nine predictors in number of state bits. Note that, in order to support up to four predictions or updates per cycle, the predictors are split into four banks that can operate independently. However, banking the three predictors marked with a star would result in a significant increase in their sizes (they were not designed with banking in mind) and would most likely be detrimental for the FCM predictor since it relies on communicating information between loads. Hence, we did not bank those three predictors, but allowed them to make an unlimited number of predictions/updates per cycle.

Hardware Cost of Several 2048-Entry Predictors		
	state info	CE cost
Basic LVP	16.0 kB	0.0 %
Tagged LVP (19-bit tags)	20.8 kB	29.7 %
Bimodal LVP (3-bit counters)	16.8 kB	4.7 %
Tag Bim LVP (8 tbits, 3 cbits)	18.8 kB	17.2 %
SSg LVP (8-bit histories)	18.0 kB	12.7 %
SSg LVP (14-bit histories)	21.5 kB	34.4 %
Last Distinct 4 Value Pred*	26.6 kB	66.0 %
Last Distinct 4 Value + Stride*	27.2 kB	69.9 %
Stride 2-Delta + FCM Pred*	26.3 kB	64.1 %

Table 4: Hardware cost in kilobytes of state and the percentage of state used for the confidence estimator (CE) of various load value predictors.

The *Basic LVP* requires the least amount of state information (i.e., counter, cache, history, tag and valid bits, etc.) because it only retains previously loaded values and no other information. Since we model a 64-bit machine, the *Basic LVP* needs 16kB of storage for the 2048 values. This is our base case.

The *Tagged LVP* augments the *Basic LVP* with one tag per predictor line. If we assume a 4GB address space, the tags have to be 19 bits long for a 2048-entry predictor. This scheme requires 29.7% more state than the base case. Predictions only take place if the tag matches. After each

prediction the value and the tag are updated. Partial tags would considerably reduce the hardware cost of this scheme, but not even full tags result in good performance.

The *Bimodal LVP* incorporates 3-bit saturating up/down counters as a CE. (McFarling termed the corresponding branch predictor *Bimodal* [18], hence the name.) Predictions are only made if the counter value is greater or equal to a preset threshold, which can be varied between one and seven. If a prediction turns out to be correct, the corresponding counter is incremented by one, otherwise it is decremented by one. The values are always updated, independent of the current state of the corresponding counter. This scheme requires only 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.

The *Tag Bim LVP* is a combination of the previous two predictors. It includes partial 8-bit tags and 3-bit saturating counters. Predictions are made if the tag matches and the counter value is greater or equal to the threshold. We performed a detailed parameter space evaluation for this predictor (see Section 6.2.2). This scheme requires 17.2% additional hardware and performs very well.

Our *SSg(comp) LVP* is 12.7% larger than the baseline predictor when 8-bit histories are used and 34.4% larger with 14-bit histories. The corresponding *SSg(algo)* predictors require the same amount of state.

The *Last Distinct 4 Values* predictor retains four values per line, so the predictor has only 512 lines. In addition, every line in the predictor includes a 21-bit tag, eight bits of least-recently-used information, and twelve bits to store which one of the four values was used during the last six accesses. The twelve bits form an index into a second-level table of 4096 lines containing four four-bit counters each. The highest of the four selected counters determines which one of the four values to use for the next prediction. However, no prediction is made if the highest counter is below a preset threshold. The counters whose corresponding value ends up not matching the true load value are decremented by three and the remaining counters are incremented by one. The least recently used value is updated with the new load value if the new value is not currently among the four stored values. The necessary amount of state for this scheme is 66.0% over the base case. The counters saturate at twelve [31], which limits the possible threshold values to one through twelve.

The *Last Distinct 4 Values + Stride* predictor is identical to the previous predictor except it also stores an eight-bit partial stride per predictor line and a two-bit state field. The stride is added to the selected value and used for making a prediction if the four counters are below the threshold and the same stride has been seen at least twice in a row (the two state-bits record this information). The partial stride is always updated. This predictor requires 70.3% more state than the base case.

Finally, the *St2d + FCM* predictor combines a stride 2-delta predictor with a finite context method predictor. Whichever predictor reports the higher confidence gets to make the prediction if its confidence is above the preset threshold. Both components are updated in parallel. The stride component includes an 8-bit partial tag, a 3-bit saturating counter, a 64-bit last value field, and two 8-bit partial stride values in each of its 512 lines. The FCM component is a two-level predictor. The first level comprises 1024 lines, each storing an 8-bit partial tag, a 3-bit saturating counter, and the least significant ten bits of the last three fetched load values. The three ten-bit values are shifted and combined via the logical exclusive-or function to form an index into the second level, which comprises 2048 lines storing a 64-bit value each. The 64-bit value is the value that followed those same three load values the last time they were seen in sequence. This predictor requires 66.4% more state than the baseline predictor.

6.2.1 Confidence Estimator Comparison

Figure 4 and Figure 5 show how the confidence estimators of several selected predictors perform with a small (1024 entries) and a large (8192 entries) configuration, respectively.

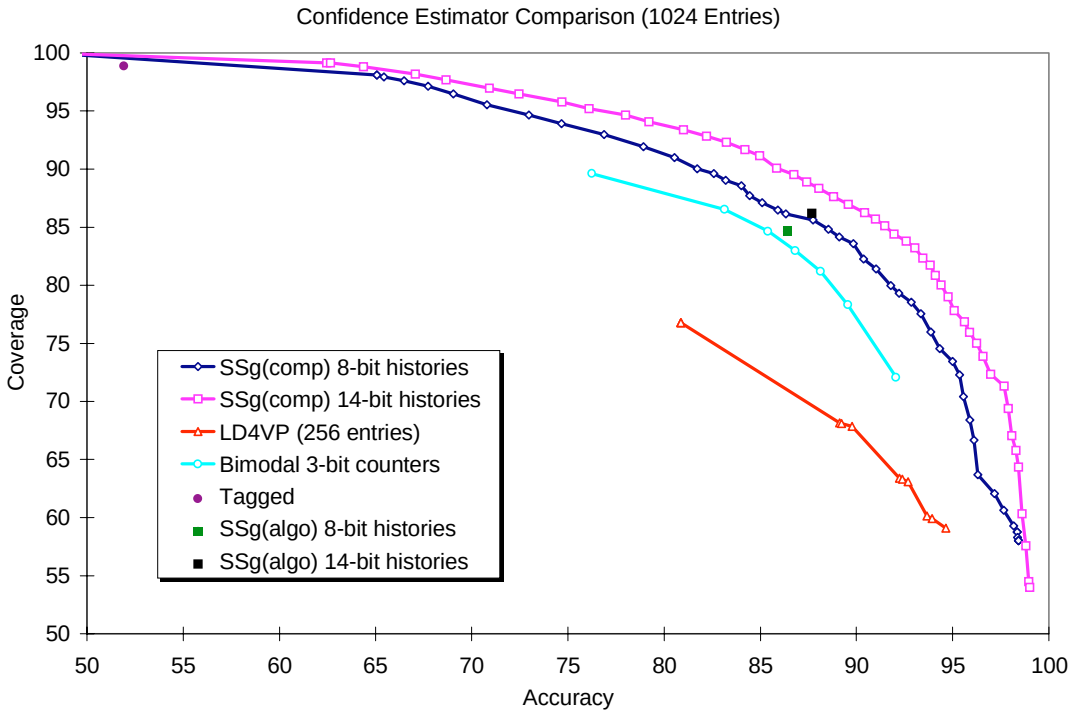

Figure 4: Accuracy-coverage pairs of several confidence estimators with 1024-entry predictors. The dots correspond to various thresholds.

Note that the *Basic* predictor is not visible in either figure. Its coverage is 100% but its accuracy is only about 50% for both configurations. The *Tagged* and the two *SSg(algo)* predictors allow no variability and are therefore each represented by a single point.

With eight history bits, our CE outperforms all other CEs except the 14-bit *SSg(algo)* CE. 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)* CE outperforms all other CEs. 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 CEs.

All the CEs benefit from an increase in size. However, our measurements with infinite CE-sizes show that the 8192-entry results are close to the limit for all predictors and that our predictor maintains its superiority with one exception. For accuracies under 92%, *LD4VP* slightly surpasses *SSg(comp)* in the infinite case.

LD4VP benefits the most from going from 1024 entries to 8192 entries. That is because *LD4VP* stores four values per predictor line, which results in four times fewer lines and consequently more aliasing, particularly with the smaller configuration.

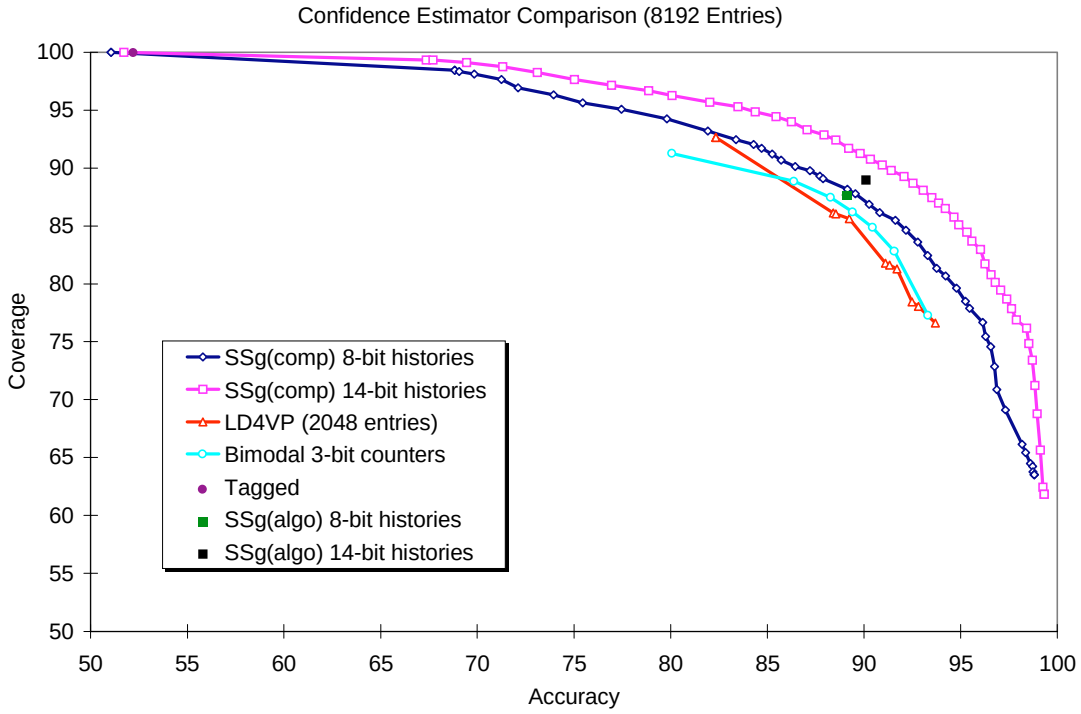

Figure 5: Accuracy-coverage pairs of several confidence estimators with 8192-entry predictors. The dots correspond to different thresholds.

6.2.2 Speedup Results

Figure 6 shows the speedups we measured using a detailed cycle-accurate pipeline-level simulation of a microprocessor similar to the DEC Alpha 21264 (see Section 5). The displayed results are harmonic mean speedups over the eight SPECint95 programs.

The results are given for both a re-fetch and a re-execute misprediction recovery policy. For predictors that allow multiple threshold values, the result of the configuration with the best average speedup is listed. The thresholds that yield the highest average speedup are seven (out of seven) for the *Bimodal LVP* using re-fetch and five using re-execute, 86% for *SSg(comp)* with re-fetch, 67% for *SSg(comp)* with re-execute, twelve (out of twelve) for *LD4VP* and *LD4V+Stride* both for re-fetch and re-execute, and seven (out of seven) for the *St2d+FCM* using re-fetch and five using re-execute.

For *Tag Bim LVP* we performed a thorough parameter space evaluation and found that it performs best on the modeled CPU with three-bit counters, a threshold of three and a decrement of two upon a misprediction when re-execute is used. For re-fetch, the best configuration is four-bit counters with a threshold of nine and a decrement of seven.

Most predictors perform quite well with a re-execution policy. Nevertheless, *SSg(comp) LVP* is only outperformed by the significantly more complex and hardware intensive *LD4V+Stride* hybrid predictor.

With the much simpler re-fetch mechanism, which most of today's CPUs already incorporate and therefore is the more likely recovery mechanism in the near future, our *SSg(comp) LVP* out-

performs all the other predictors. Surprisingly, *Tag Bim LVP* is the only predictor that comes close to the speedup delivered by *SSg(comp)*. In looking at more detailed results, we note that those are the only two predictors that are capable of delivering a genuine speedup for all the eight benchmark programs. All the other predictors actually slow down at least three of the eight programs, often significantly, when re-fetch is used.

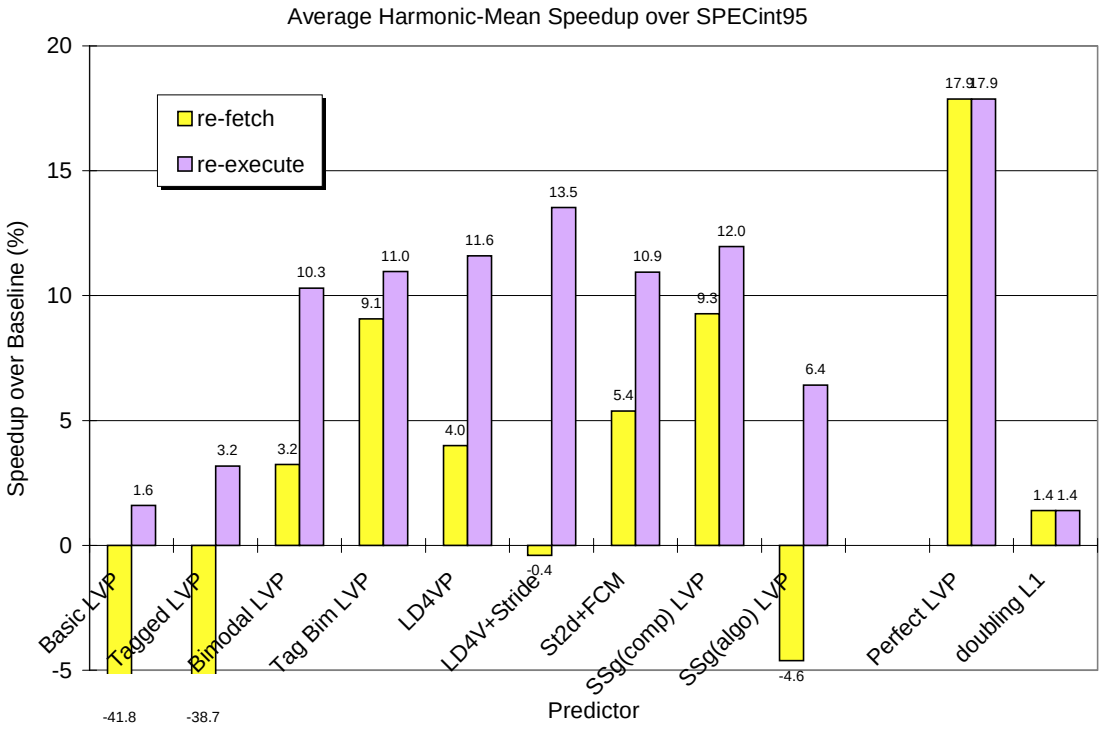

Figure 6: Average speedups of the eight SPECint95 programs on a DEC Alpha 21264-like processor. All the predictors are sized to hold an equivalent of 2048 load values.

The *Perfect LVP* in Figure 6 contains an oracle that always makes a prediction if the prediction will be correct and no prediction otherwise. In spite of the good performance of *SSg(comp)*, the comparison with the *Perfect LVP* shows that *SSg(comp)* is only able to reach about half of the theoretically possible speedup using re-fetch and two thirds using re-execute. Clearly, there is considerable room for improvement left.

Using re-fetch, the *SSg(comp)* confidence estimator causes a prediction following about 150 history patterns. In the re-execute case, the number of prediction causing history patterns increases to about 2500 (out of 16384). It is not feasible to look for this large a number of patterns using comparators. Rather, one would probably use the history pattern as an index into a preprogrammed $1 \times 2^{\#\text{history-bits}}$ bit read only memory (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 [12].

Of course the extra hardware that a load value predictor requires could also be used to improve

other parts of the processor. For example, the rightmost column in Figure 6 denotes the speedup resulting from doubling the simulated processor's L1 data-cache from 64 kilobytes to 128 kilobytes. Despite this large hardware increase, the resulting speedup is very small. Some of the predictors outperform the doubled cache by more than a factor of seven while requiring four times less hardware.

Doubling the L1 data-cache reduces its average load miss-rate from 4.7% to 3.6%. For decent refill penalties, this small miss-rate reduction does not result in a significant performance improvement. Only for very large refill penalties would a 1% miss-rate difference yield an improvement comparable to the one we get with our load value predictor. Hence we conclude that above a certain cache size, it makes more sense to add a load value predictor than to further increase the size of the cache.

Our measurements of most of these predictors for smaller and larger sizes show the same relative performance that they do for the 2048-entry size. The FCM is the only predictor that may be able to take advantage of much larger predictor sizes. The other predictors do not benefit from larger sizes and their performance stays the same.

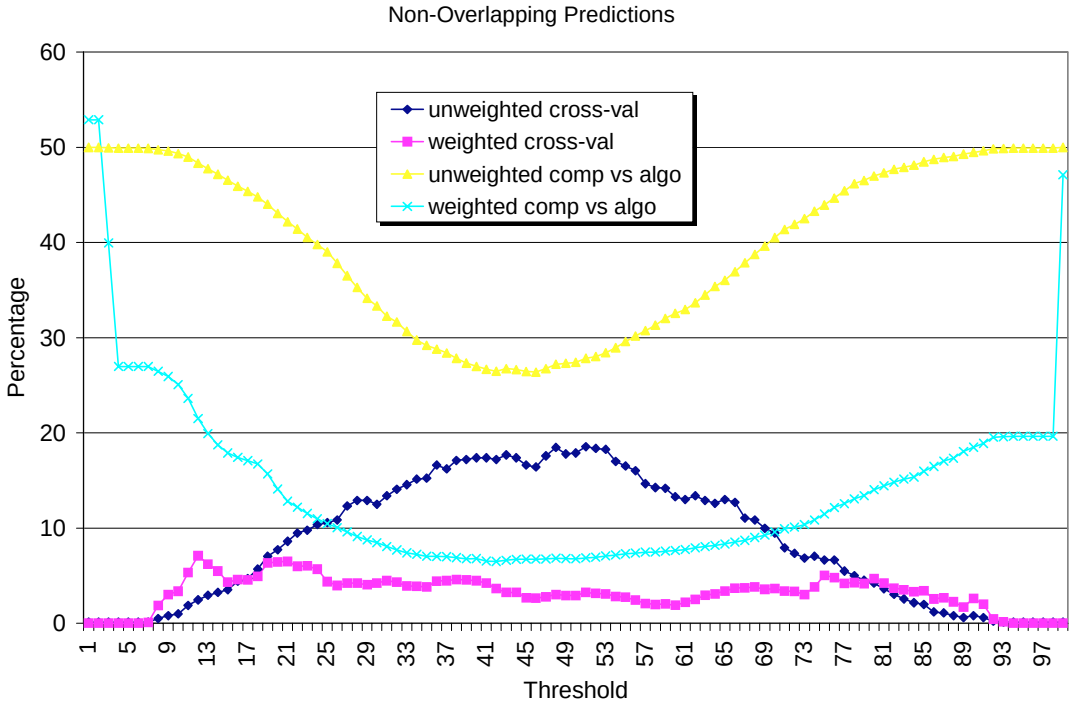

Figure 7: Percentage of dissimilar cross-validation outcomes and the percentage of history patterns that are classified differently by *algo* than by *comp*. The weighted percentages are weighted by the frequency of occurrence of the respective history patterns.

Figure 7 depicts the percentage of history patterns that do not yield overlapping cross-validation results and the percentage of history patterns that *algo* classifies differently than *comp*. The results are shown over the full range of *comp* thresholds. Furthermore, there are two curves

each, one showing the non-weighted result and one showing the same result when the patterns are weighted by their frequency of occurrence. For instance, the eight cross-validations classify 11.0% of the history patterns not uniformly when a threshold of 67% is used. However, those patterns only represent 3.7% of all the occurring patterns during program execution. Apparently, seldom occurring patterns are harder to classify than frequently occurring ones, which may simply be a consequence of a small sample set. The figure further shows that *algo* classifies 37.9% of the history patterns differently than *comp* with a threshold of 67%. The non-overlapping classifications affect 8.7% of the patterns encountered during execution.

Figure 7 shows that *comp* is closest to *algo* at a 39% threshold in the weighted case and at 48% in the non-weighted case. These thresholds are close to the 44% we derived from Table 1 in Section 3 for the 4-bit case. The implicit threshold of the *algo* scheme is significantly lower than the thresholds that work well for our CPU, which explains the poor performance of *algo*.

An interesting point to note is the uniformity of the cross-validation results. According to Figure 7, all eight cross-validations yield approximately the same classification result (within a few percent), in particular for the "interesting" thresholds above 66%. The prediction-causing history patterns hence seem to be quite universal and not very dependent on the programs.

The optimal threshold value and consequently the prediction causing histories are, however, rather dependent on the characteristics of the underlying CPU. For example, when changing the misprediction recovery mechanism from re-fetch to re-execute, the optimal threshold drops from 86% to 67%.

These results, in combination with the relatively poor performance of *SSg(algo)*, suggest that profiling is very effective at finding a good threshold value, i.e., at identifying which history patterns should be followed by a prediction. Fortunately, this has to be done only once for a given type of CPU and compiler infrastructure.

7. Summary and Conclusions

In this study we describe a novel confidence estimator for load value predictors. It uses histories of the recent prediction outcomes to decide whether or not to attempt a prediction. Profile information is utilized to determine which history patterns should be followed by a prediction. In our measurements of SPECint95 we observe low variability of high-confidence history patterns, suggesting that the prediction causing history patterns would not have to be changed on a per application basis.

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

When a re-fetch misprediction recovery mechanism is used, which all processors that support branch prediction already incorporate, our predictor outperforms other predictors from the literature by over 40% and yields an average speedup of 9.3% on SPECint95. 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 microprocessors.

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

[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] B. Calder, P. Feller, A. Eustace. "Value Profiling". *30th International Symposium on Microarchitecture*. December 1997.

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

[4] 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.

[5] C. Y. Fu, M. D. Jennings, S. Y. Larin, T. M. Conte. "Value Speculation Scheduling for High Performance Processors". *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. October 1998.

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

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

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

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

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

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

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

[13] 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.

[14] J. K. F. Lee, A. J. Smith. "Branch Prediction Strategies and Branch Target Buffer Design". *IEEE Computer* 17(1). January 1984.

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

[16] M. H. Lipasti, J. P. Shen. "The Performance Potential of Value and Dependence Prediction". *EUROPAR-97*. August 1997.

[17] 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.

[18] S. McFarling. *Combining Branch Predictors*. WRL Technical Note TN-36, Digital Western Research Laboratory, Palo Alto. June 1993.

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

[20] S. T. Pan, K. So, J. T. Rahmeh. "Improving the Accuracy of Dynamic Branch Prediction using Branch Correlation". *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. October 1992.

[21] G. Reinman, B. Calder, D. Tullsen, G. Tyson, T. Austin. *Profile Guided Load Marking for Memory Renaming*. Technical Report CS-98-593, University of California San Diego. 1998.

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

[23] 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.

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

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

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

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

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

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

[30] A. Srivastava, A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools". *1994 Conference on Programming Language Design and Implementation.* June 1994.

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

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

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