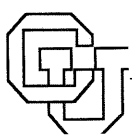
Profile-Supported Confidence Estimation for Load Value Prediction

Martin Burtscher
Benjamin G. Zorn

CU-CS-872-98



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Profile-Supported Confidence Estimation for Load Value Prediction

Martin Burtscher and Benjamin G. Zorn

Technical Report CU-CS-872-98
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430
{burtsche, zorn}@cs.colorado.edu

Abstract

Load instructions occasionally incur very long latencies. These long latencies also delay the execution of all the 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.

Current load value predictors can only correctly predict about 40 to 70 percent of the dynamically fetched load values. *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.

In this paper we present a novel confidence estimator that works based on *prediction outcome histories*. Profiles are used to identify high-confidence history patterns. Our confidence estimator 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 a simple last value predictor combined with our confidence estimator outperforms other predictors by more than 50% when a re-fetch misprediction recovery mechanism is used. Furthermore, our predictor is the only predictor that yields 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 [LCB+98], improving their execution speed should significantly improve the overall performance of the processor.

Fortunately, load values are not random but often follow certain patterns, which makes them predictable [LWS96]. For instance, many load instructions tend to fetch the same values repeatedly. 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* [LWS96, Gab96].

Empirically, papers have shown that the results of most instructions are predictable [Gab96, LiSh96, SaSm97a]. However, of all the frequently occurring, result-generating instructions, load instructions are the most predictable [LiSh96] and incur by far the longest latencies. Since only about every fifth executed instruction is a load, predicting only load values requires significantly fewer predictions and leaves more time to update the predictor. As a consequence, smaller and simpler predictors can be used. We therefore believe that predicting only load values may well be more cost effective than predicting the result of every instruction.

Context-based load value predictors try to exploit the value locality present in programs. We termed the simplest such predictor *Basic LVP* (last value predictor). It always predicts the previously loaded value.

To reduce the number of mispredictions, 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) which making no prediction does not.

CEs are similar to branch predictors because 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 improved branch predictors could also be used to improve 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) [SCAP97]. The intuition is that the past prediction behavior tends to be very indicative of what will happen next. 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.

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 [SmSo95]. Because branch prediction requires a similar mechanism, most modern microprocessors already contain the required hardware to perform this kind of speculation [Gab96].

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 very well. 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 an average speedup over SPECint95 of 15.8% with a re-execute policy and 13.1% with a re-fetch recovery policy. It is able to attain a speedup for all the SPECint95 programs, even with the much simpler re-fetch mechanism. All the other predictors we looked at actually slow down four or more of the eight benchmark programs when re-fetch is employed, as our pipeline-level simulations revealed. Section 6.2 provides more detailed results.

We also investigated using saturating counters instead of profiling to identify the history patterns that should trigger a prediction [BuZo98], 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 presents related work. Section 4 illustrates the use of prediction outcome histories and introduces our predictor. Section 5 explains the methods used. Section 6 presents the results. Section 7 concludes the paper with a summary.

2. Predictor Architecture

Figure 2.1 shows the components of a confidence-based load value predictor. The largest element is an array (cache) 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.

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 which are always zero since the processor we use requires instructions to be word aligned. 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 one of the cache 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 cache line that was used for making the prediction.

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 4.1), and the decision logic is preprogrammed accordingly.

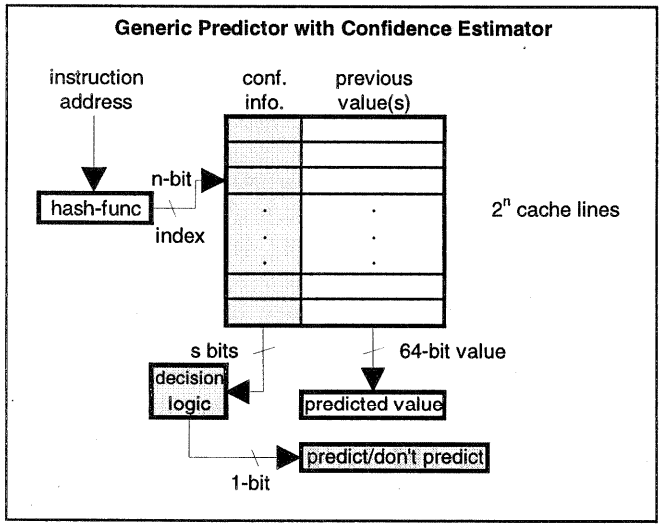


Figure 2.1: The components of a load value predictor with a confidence estimator (shaded).

3. Related Work

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

Early Work: Two independent research efforts [LWS96, Gab96] first recognized that load instructions exhibit *value locality* and concluded that there is potential for prediction.

Lipasti et al. [LWS96] 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 [LiSh96] broaden their scope to predicting all result generating instructions and show how value prediction can be used to exceed the existing 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 [Gab96] deals mostly with general value prediction and how to boost the instruction level parallelism (ILP) beyond the data-flow limit, but he also studies load value prediction by itself.

Load Value Predictors: Lipasti et al. [LWS96] describe a last value predictor (predicts the last seen load value) to exploit the existing load value locality. Their predictor utilizes 2-bit saturating up/down counters to classify loads as unpredictable, predictable, or constant. In Section 6.2, we will compare our predictor with a predictor similar to theirs.

Gabbay [Gab96] proposes four predictor schemes: a tagged last value predictor, a tagged stride predictor, a register-file predictor, and a sign-exponent-fraction predictor. We compare our predictor to the tagged last value predictor in Section 6.2. However, we refrain from comparing our predictor with the register-file and the SEF predictor because the former performs extremely poorly and the latter can only be used for floating-point loads. We also exclude the stride predictor from our comparison, since load values exhibit virtually no stride behavior (constant offset between the values), as Gabbay points out.

Wang and Franklin [WaFr97] 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. Their predictor has the highest prediction accuracy of all the predictors in the literature. Nevertheless, our predictor significantly outperforms theirs when a re-fetch misprediction policy is used (see Section 6.2). Wang and Franklin also proposed a hybrid predictor. We feel that it is premature at this point to hybridize predictors before the individual

components have been thoroughly studied and optimized.

Sazeides and Smith [SaSm97a] 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 pattern of values. In a follow-up paper [SaSm97b], 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 only when large tables are used. Since the table size required for good performance is considerably larger than the roughly 20 kilobytes we allow each predictor in this study, we do not include a finite context method predictor in our comparison.

Profiling: Gabbay and Mendelson [GaMe97a] explore the possibility of using program profiling to enhance the efficiency of value prediction. They use profiling to insert opcode directives, which allow them to allocate only highly predictable values. Manual fine-tuning of the user supplied threshold value allows them to outperform their hardware-only predictor in almost all cases. They found that different input sets make no significant difference, i.e., that train runs correlate to test runs.

Calder et al. [CFE97] 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 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 also 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 instead we profile the success-rate of a last value predictor with respect to its recent prediction behavior (see Section 4). The result is then used to configure our confidence estimator rather than to modify an executable. Once the confidence estimator is configured, no further profiling is required.

As mentioned in the introduction, we also investigated using saturating counters instead of profiling [BuZo98]. While the average speedup of our dynamic predictor is slightly higher than the speedup of the profile-based predictor, the profile-based predictor is simpler and yields better speedups for more than half of the individual benchmark programs.

Dependence Prediction: In their next paper [LiSh97], Lipasti and Shen add dependence prediction to their predictor and switch to predicting source operand values rather than instruction results, since this decouples dependence detection from value-speculative instruction dispatch.

Reinman and Calder [ReCa98] also examine dependence prediction and conclude that, due to its small hardware requirement, it 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 [RCT+98], 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. Like other profile based approaches, their approach requires changes to the ISA.

Other Related Work: Rychlink et al. [RFKS98] 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 pollution.

In their next paper [GaMe97b], 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 why we restrict ourselves to predicting only load values, which requires significantly smaller and simpler predictors while still reaping most of the potential.

Gonzalez and Gonzalez [GoGo98] 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 accuracy and the resulting performance improvement. Our results in Section 6.2.1 show that our predictor yields much higher

accuracies than other predictors from the literature. These higher accuracies do indeed result in significantly higher speedups with a re-fetch architecture.

Fu et al. [FJLC98] 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 [SaSm98] 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 [GoGo98] did indeed observe such an improvement in their study.

Confidence Estimation: Jacobsen et al. [JRS96] and Grunwald et al. [GKMP98] introduce confidence estimation to the domain of branch prediction and multi-path execution to decide whether 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. [LWS96] use) does not yield very good results when applied to load value prediction (as we show in Section 6.2).

Branch Prediction: In the area of branch prediction, a significant amount of related work exists. Lee and Smith [LeSm84] keep a *history* of recent branch directions for every conditional branch and systematically analyze every possible pattern.

Yeh and Patt [YePa92, YePa93] and Pan, So, and Rahmeh [PSR92] describe sets of two-level branch predictors and invent a taxonomy to distinguish between them. We adopt one of their designs for use as confidence estimators in our load value predictor.

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

Sprangle et al. [SCAP97] 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 branch predictions were a success or failure instead of whether the branches were taken or not.

Summary: In this study, we only consider load value predictors that require about 20 kilobytes of state information since we believe that larger predictors cannot be added to CPUs in the near future. Predictors that do not perform well with this table size are excluded.

We also do not believe that microprocessor manufacturers are willing to change their instruction set architecture just to accommodate load value prediction, which is why we currently only investigate transparent prediction schemes. This is also why we exclude profile-based approaches that require extra bits in the opcode for our study.

Furthermore, we feel that it is premature to think about hybrid predictors. Instead, we try to evaluate and improve the individual components first.

The novelty of our predictor is that it uses prediction outcome histories for confidence estimation, an idea from the branch prediction literature. Unlike all other profile-based approaches, we do not profile actual load values and do not need to modify binaries.

4. Using Prediction-Outcome Histories

Just like with branch prediction [SCAP97], we found histories that record the recent prediction successes and failures to be a very successful idea in the domain of load value prediction as well. In fact, we found prediction outcome histories to be better suited for load value prediction than other approaches like saturating counters because they scale better, they allow accuracy-coverage pairs to be chosen at a 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 successful prediction and which ones are not. Heuristics and algorithms to do this exist in the branch prediction literature. For example, Sechrest et al. [SLM95] describe a scheme that tries to identify repeating patterns of branch outcomes. If no 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 their 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 *(comp)* works, we present Table 4.1. It shows the output of a 4-bit history run based on SPECint95 behavior. 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*.

The table shows the average over all the benchmarks and is for illustration purposes only. The results presented in the subsequent sections were generated using cross-validation (see Section 5.3).

SPECint95 Last Value Predictability		
history	predictability	occurrence
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 4.1: 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.

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 on which the prediction is going to be made.

If only a small cost is associated with making a misprediction (e.g., as is the case with a re-execute architecture), it is probably 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 96.6% confident, the decision logic would only allow predictions for histories whose predictability is greater than or equal to 96.6%, i.e., only for history *1111* based on the data in Table 4.1. 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, which other proposed predictors cannot reach. With longer histories, our approach yields even higher accuracies and better coverage.

4.1 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). Since our confidence estimator is similar to Yeh and Patt's *SSg* branch predictor [YePa93], programmed with Sechrest et al.'s *(comp)* approach, we call our predictor *SSg(comp) LVP*.

Predictions are performed as described in Section 2, that is, 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 cache line and the oldest bit is shifted out (lost). If the true load 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 the size of 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 only results in an almost immeasurable increase in accuracy, which we believe does not justify the extra hardware.

Note that our predictor requires no content ad-

dressable memory and that all the operations can be performed using at most two table lookups. This is similar to current branch predictors and should therefore not affect the cycle time [YePa92].

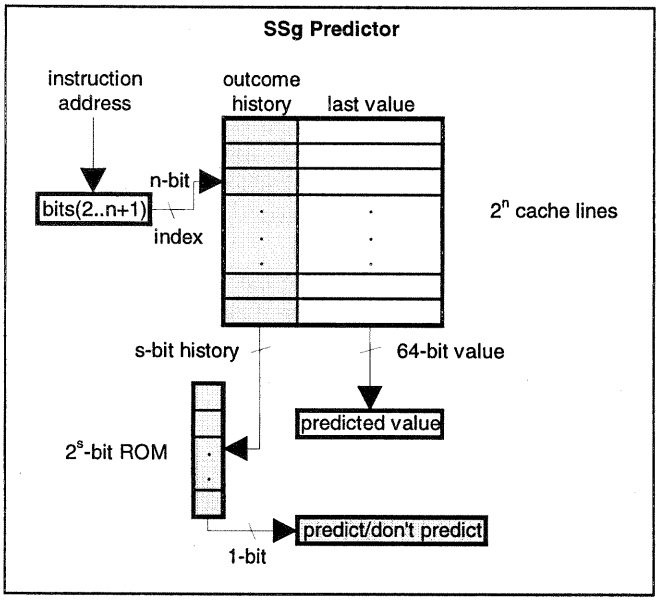


Figure 4.1: The components of the *SSg last value predictor*. The *SSg* confidence estimator is shaded.

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 stage on, predictions can be made in any pipeline 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 value is only needed at the beginning of the execute stage, the predictions could even be superpipelined (take more than one cycle) over the fetch and decode stages.

With this architecture, only one prediction can be made per cycle. If more than one prediction per cycle is necessary, the prediction hardware would have to be duplicated, which would allow multiple independent predictions to be performed in parallel. Naturally, all the predictors would have to be updated together.

Updating takes another cycle during which the predictor is busy and no prediction can be made. As Table 5.2 shows, only about every fifth instruction is a load so it is likely that there would be cycles available during which the predictor can be updated.

Another possible scenario, however, is that a next prediction needs to be made before the previous one has been updated. This outcome only poses a problem if both predictions go to the same cache line (or the same counter) 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 cache lines could be marked as "in use" and further predictions will stall until the lines have been updated, or further predictions could be made with 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 very 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 cache line, indicating that in almost all cases the required cache 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 without causing a bottleneck. The predictor works with any instruction set and requires no changes to the instruction set architecture, such as adding bits to the op-code [GaMe97a].

5. Methodology

All our measurements are performed on the DEC Alpha AXP architecture [DEC92]. To perform a thorough design-space evaluation, we instrumented the benchmarks using the ATOM tool-kit [EuSr94, SrEu94]. This allowed us to efficiently simulate the proposed predictor in software and to identify good configurations, the most promising of which were then fed to our pipeline-level simulator for more detailed evaluation.

To obtain actual speedup results, we use the AINT simulator [Pai96] with its out-of-order back-end, which is configured to emulate a processor similar to the DEC Alpha 21264 [KMW98]. 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 5.1. Operating system calls are executed but not simulated. Loads can only execute when all prior store addresses are known.

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 5.1: Functional unit and memory latencies (in cycles) of our simulator.

5.1 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 5.2 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 ref-inputs. Furthermore, we ran 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 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 instructions. 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 the benchmark programs for 300 million instructions on our simulator after having skipped over the initialization code in "fast execution" mode. This fast-forwarding is very important because the initialization part of programs is not representative of the general program behavior [ReCa98]. The rightmost column of Table 5.2 shows the number of instructions that were skipped. GCC is completely executed (334 million 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, 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 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.

Information about the SPECint95 Benchmark Suite							
program	total executed load instructions	load sites	load sites that account for Q50	Q90	Q99	Q100	skipped inst (M)
compress	10,537 M (17.5%)	3,961	17	58	81	690	6,000
gcc	80 M (23.9%)	72,941	870	5,380	14,135	34,345	0
go	8,764 M (24.4%)	16,239	204	1,708	4,221	12,334	12,000
ijpeg	7,141 M (17.2%)	13,886	42	187	423	3,456	1,000
li	17,792 M (26.7%)	6,694	42	138	312	1,932	4,000
m88ksim	14,849 M (17.9%)	8,800	52	216	456	2,677	1,000
perl	6,207 M (31.1%)	21,342	44	169	227	3,586	1,000
vortex	22,471 M (23.5%)	32,194	57	585	3,305	16,651	5,000
average	10,980 M (21.8%)	22,007	166	1,055	2,895	9,459	

Table 5.2: 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 when using the ref-inputs. 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. The rightmost column shows the number of instructions (in millions) that are skipped before starting the detailed pipeline-level simulation.

5.2 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, speedups are dependent on the architectural features of the underlying CPU. Consequently, non-implementation specific metrics are also important.

A value predictor with a *confidence estimator* can produce four prediction outcomes: correct prediction, incor-

rect prediction, correct non-prediction (no prediction was made, and the guessed value would not have been correct), and incorrect non-prediction (no prediction was attempted even though the guessed value would have been correct). We denote the number of times each of the four cases is encountered by PCORR, PINCORR, NPCORR, and NPINCORR, respectively. 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. 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. They are all higher-is-better metrics.

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

The POT represents the fraction of all load values that are predictable, 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 an attempted 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 merely translates into better utilization of the existing potential. Nevertheless, a high coverage is still desired.

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

5.3 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 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 to a number of predictors from the literature. To better explore the space of parameters we only show averages over the eight benchmarks and not the individual programs. Note that, for improved readability, several figures in these subsections are not zero based.

6.1 *SSg(comp)* Confidence Estimator Results

All the results for the *SSg(comp)* confidence estimator are generated using cross-validation (Section 5.3). Before every run, all cache entries are set to zero.

Figure 6.1 shows the attainable accuracy-coverage pairs 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.

Each curve was generated by varying the 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 cache is big enough for our benchmarks and has a performance that is close to the performance of an infinite cache. This was to be expected based on the quantile numbers from Table 5.2

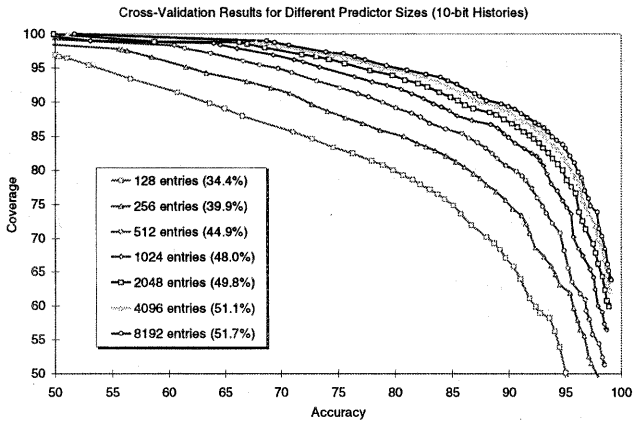


Figure 6.1: Accuracy-coverage pairs for different cache sizes and 10-bit histories. Each dot corresponds to a threshold (in 2% increments). The numbers in parentheses denote the potential delivered by the respective load value predictor.

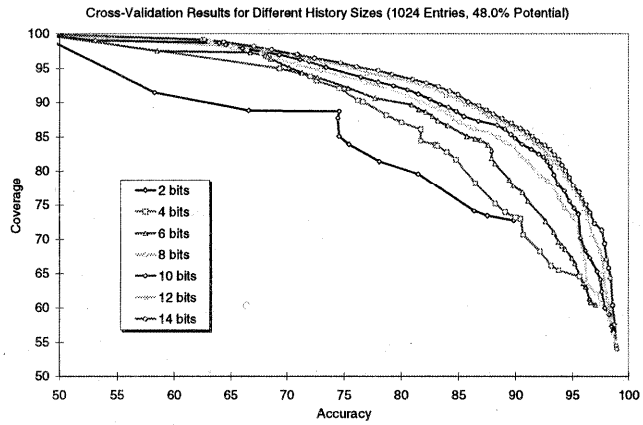


Figure 6.2: Accuracy-coverage pairs for different history sizes and 1024-entry caches. Each dot corresponds to a threshold (in 2% increments).

In Figure 6.2, which is similar to Figure 6.1, the cache size is held constant at 1024 entries and the length of the histories is varied.

The figure shows that longer histories perform better. Saturation sets in at about ten bits. These results indicate that ten history bits provide most of the prediction potential over a range of thresholds.

Note that we performed a much broader investigation of the parameter space but cannot include all the results due to space limitation. We picked Figure 6.1 and Figure 6.2 because they are quite representative of the generally observed behavior.

6.2 Predictor Comparison

In this section we compare 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.

Hardware Cost and Potential of several 2048-entry Predictors			
	state bits	rel. cost	potential
Basic LVP	131072	0.0 %	49.85 %
Tagged LVP (19-bit tags)	169984	29.7 %	49.85 %
Bimodal LVP (3-bit counters)	137216	4.7 %	49.85 %
SSg LVP (8-bit histories)	147456	12.5 %	49.85 %
SSg LVP (14-bit histories)	159744	21.9 %	49.85 %
Last Distinct 4 Values	217600	66.0 %	48.41 %

Table 6.1: Hardware cost in number of state-bits and the potential of various load value predictors.

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 approximately 19 kilobytes of state, which we find reasonable given that the DEC Alpha 21264 microprocessor incorporates two 64 kilobyte L1 caches on chip [KMW98]. Table 6.1 shows the hardware cost of the five predictors in number of state bits and lists their delivered potential.

The *Basic LVP* requires the least amount of state information (i.e., counter, cache, history, tag and valid bits). Since Alphas are 64-bit machines, every value in the cache requires 64 bits. Consequently, the *Basic LVP* needs 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. After each

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

The *Bimodal LVP* incorporates a 3-bit saturating up/down counter per line. (McFarling named the corresponding branch predictor *Bimodal* [McF93], hence the name.) We found 3-bit counters and always updating the values to work very well. 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 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.

Our *SSg(comp) LVP* is 12.5% larger than the base case when 8-bit histories are used and 21.9% larger with 14-bit histories. The corresponding *SSg(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 had only 512 lines. The bit count for this scheme is 66.0% over the base case. The pattern of the last six accesses is used to select a set of 4-bit saturating counters, of which the highest counter determines which of the four values to use for the prediction. No prediction is made if the selected counter is below a preset threshold. The counters saturate at twelve [WaFr97], which limits the possible threshold values to one through twelve.

6.2.1 Confidence Estimator Comparison

Figure 6.3 and Figure 6.4 show how the confidence estimators of different predictors perform with a small (1024 entries) and a large (8192 entries) configuration, respectively.

Note that *Basic* is not visible. Its coverage is 100% but its accuracy is only about 50% in both cases. The *Tagged* and the two *SSg(algo)* schemes 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 predictors benefit from an increase in size. However, our measurements with infinite cache-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 cache line, which results in four times fewer cache lines and consequently more aliasing, particularly with the smaller configuration.

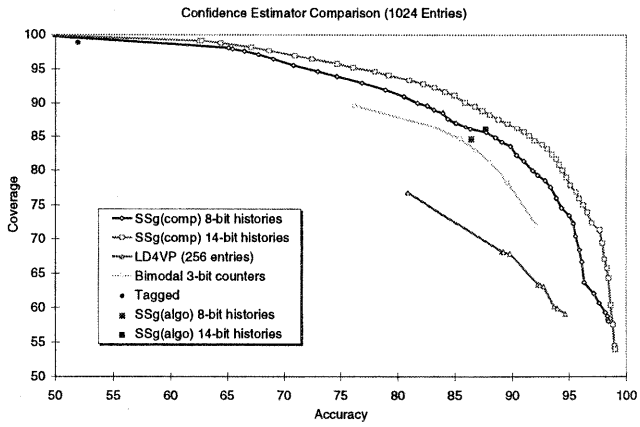


Figure 6.3: Accuracy-coverage pairs of several confidence estimators with 1024-entry caches. The dots correspond to various thresholds.

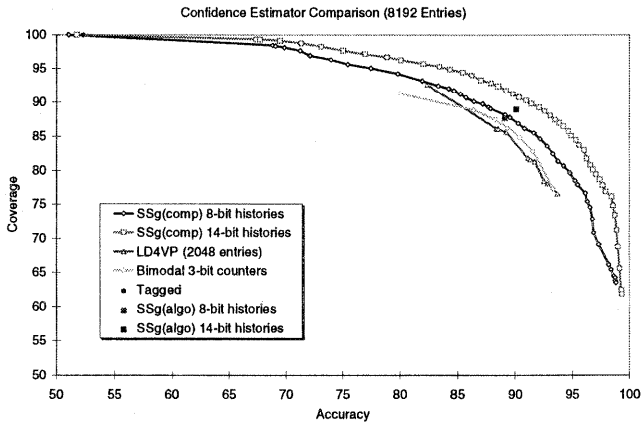


Figure 6.4: Accuracy-coverage pairs of several confidence estimators with 8192-entry caches. The dots correspond to various thresholds.

6.2.2 Speedup Results

Figure 6.5 shows the speedups we measured using a detailed pipeline-level simulation of a microprocessor similar to the DEC Alpha 21264 (see Section 4). The displayed results are average speedups over SPECint95.

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, 65% for *SSg(comp)* with re-execute, and twelve (out of twelve) for *LD4VP* both for re-fetch and re-execute.

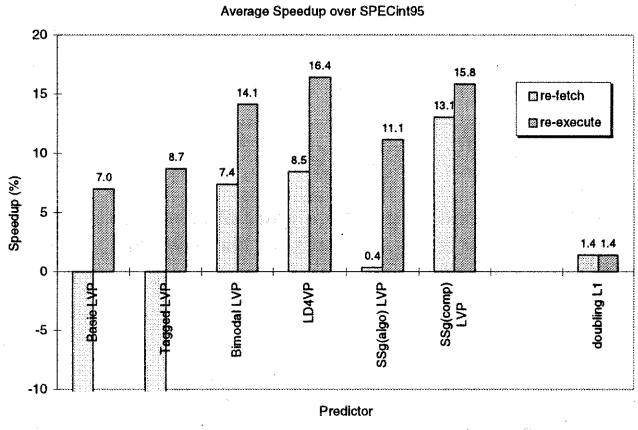


Figure 6.5: Average speedups of the eight SPECint95 programs on a DEC Alpha 21264-like processor (the cut-off negative re-fetch speedup percentages are -35.5% for *Basic* and -32.2% for *Tagged*).

Bimodal LVP, *SSg(algo) LVP*, and *LD4VP* perform quite well with a re-execute policy. *SSg(comp) LVP* outperforms the first two and almost reaches the performance of the significantly more complex and hardware intensive *LD4VP*. Given the CE-results from the previous section, we have to attribute *LD4VP*'s superior performance to the fact that it keeps four values per line rather than having four times as many lines holding just one value.

With the much simpler re-fetch mechanism, which most of today's CPUs already contain and therefore the more likely recovery mechanism in the near future, our *SSg(comp) LVP* outperforms all the other predictors by at least 50%. In looking at more detailed results, we note that all the predictors but ours actually slow down at least half the benchmark programs, often significantly. Only our predictor is capable of delivering a genuine speedup for all the programs.

Using re-fetch, our confidence estimator makes a prediction following about 150 history patterns. In the re-execute case, about 2500 history patterns cause a prediction (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 bit by $2^{\#\text{history-bits}}$ 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 [KMW98].

The rightmost column in Figure 6.5 denotes the speedup resulting from doubling the simulated processor's L1 data-cache from 64 kilobytes to 128 kilobytes. Despite this hardware increase of 564,224 state-bits, the resulting speedup is very small. Some of the predictors outperform the doubled cache tenfold while requiring only a fourth of the 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 further increase the cache size.

An interesting point to note is the uniformity of the cross-validation results, as is depicted in Figure 6.6. The figure shows that all the profiles yield about the same result (within a few percent), meaning that the prediction-causing history patterns are most likely quite universal and not very dependent on the programs.

The optimal threshold value seems to be rather independent on the programs as well, as the following experiment illustrates. Instead of maximizing the average speedup, we also tried maximizing every program individually, i.e., we identified the best threshold for every program and then averaged the speedups. For re-fetch, this improved the average speedup by 7.8% (from 13.1% to 14.1%), and for re-execute by 8.4% (from 15.8% to 17.2%). These small improvements indicate that the threshold that maximized the average speedup is close to the thresholds that maximize the individual programs.

The optimal threshold value is, on the other hand, quite 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 65%.

These results, in combination with the relatively poor performance of *SSg(algo)*, suggest that profiling is very important for finding the best threshold value, i.e., for identifying which history patterns should be followed by a prediction. However, once this has been done for a given type of CPU, the *SSg(comp)* predictor's decision logic can be programmed once and for all, rendering further profile runs superfluous.

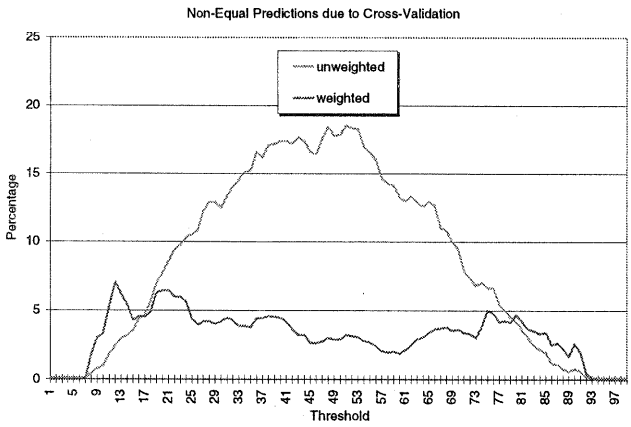


Figure 6.6: Average percentage of dissimilar prediction outcomes when using profile-based prediction. The weighted percentages are weighted by the relative occurrence of the non-equal history patterns.

7. Summary and Conclusions

In this paper 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 cause a prediction. Based on the high invariability of our measurements of SPECint95, finding high-confidence history patterns can be done once and need not 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 with a re-fetch and almost reaches the performance of the best predictor that we are aware of with a re-execute misprediction recovery policy.

The contributions of this paper include:

- We propose using prediction outcome histories as a measure of confidence for load value prediction.
- We propose making a strict distinction between the confidence estimator and the actual value predictor.
- We provide standard metrics to measure the performance of the confidence estimator and the value predictor separately as well as in combination.

We conclude that prediction outcome histories are very well suited for the domain of load value prediction and outperform other approaches, including considerably 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 a factor of over 1.5 and yields an average speedup of 13.1% 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 CPUs.

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

[BuZo98] M. Burtscher, B. G. Zorn. *Load Value Prediction Using Prediction Outcome Histories.* Technical Report CU-CS-873-98, University of Colorado at Boulder. October 1998.

[CFE97] B. Calder, P. Feller, A. Eustace. "Value Profiling". *30th Annual ACM/IEEE International Symposium on Microarchitecture.* December 1997.

[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.

[FJLC98] 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.

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

[GaMe97a] F. Gabbay, A. Mendelson. "Can Program Pro-filing Support Value Prediction?". *30th Annual ACM/IEEE International Symposium on Microarchitecture.* December 1997.

[GaMe97b] F. Gabbay, A. Mendelson. "The Effect of Instruction Fetch Bandwidth on Value Prediction". *25th International Symposium on Computer Architecture.* June 1998.

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

[GoGo98] J. Gonzalez, A. Gonzalez. "The Potential of Data Value Speculation to Boost ILP". In *12th International Conference on Supercomputing.* 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". *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.

[LeSm84] J. K. F. Lee, A. J. Smith. "Branch Prediction Strategies and Branch Target Buffer Design". *IEEE Computer* 17(1):6-22. January 1984.

[LiSh96] M. H. Lipasti, J. P. Shen. "Exceeding the Dataflow Limit via Value Prediction". *29th International Symposium on Microarchitecture.* December 1996.

[LiSh97] M. H. Lipasti, J. P. Shen. "The Performance Potential of Value and Dependence Prediction". In *EUROPAR-97.* August 1997.

[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.

[PSR92] S. T. Pan, K. So, J. T. Rahmeh. "Improving the Accuracy of Dynamic Branch Prediction using Branch Correlation". *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* 76-84. October 1992.

[RCT+98] 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. July 1998.

[ReCa98] G. Reinman, B. Calder. "Predictive Techniques for Aggressive Load Speculation". *31st Annual ACM/IEEE International Symposium on Microarchitecture.* December 1998.

[RFKS98] B. Rychlik, J. Faistl, B. Krug, J. P. Shen. "Efficacy and Performance Impact of Value Prediction". *Proceedings of the 1998 International Conference on Parallel Architectures and Compiler Technology (PACT '98).* October 1998.

[SaSm97a] Y. Sazeides, J. E. Smith. "The Predictability of Data Values". *30th Annual ACM/IEEE International Symposium on Microarchitecture.* December 1997.

[SaSm97b] Y. Sazeides, J. E. Smith. *Implementations of Context Based Value Predictors.* Technical Report ECE-97-8, University of Wisconsin-Madison. December 1997.

[SaSm98] Y. Sazeides, J. E. Smith. "Modeling Program Predictability". *25th International Symposium on Computer Architecture.* June 1998.

[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.

[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.