Global Context-Based Value Prediction

Tarun Nakra, Rajiv Gupta and Mary Lou Soffa
Department of Computer Science
University of Pittsburgh

Abstract

Various methods for value prediction have been proposed to overcome the limits imposed by data dependencies within programs. Using a value prediction scheme, an instruction's computed value is predicted during its fetch stage and forwarded to all dependent instructions to exploit parallelism. Value prediction schemes have focused on predicting the values generated by an instruction based on a **local context** *related to the* **same** *instruction. These schemes predict instruction values by analyzing values generated by the instruction during prior executions. This paper proposes techniques that predict values of an instruction based on a* **global context** *where the behavior of* **other** *instructions is also used in prediction. The global context considers the path along which an instruction is executed and the values computed by other previously completed instructions. We first present techniques that augment conventional last value and stride predictors with path information to improve prediction accuracy. We then propose a technique that predicts instructions by capturing the correlations between different instructions. Experiments performed using path-based techniques and for realistic table sizes resulted in an increase in prediction of 7-8.9% over the current prediction schemes. Prediction using correlations of instructions not predictable using path-based techniques suggests a further improvement of 10.6% accuracy over the best path-based predictor.*

Keywords: last value, stride, speculative execution, value prediction, branch history.

1 Introduction

The recent advancements in processor technology and microarchitecture research has led to the design of modern architectures capable of executing instructions in parallel. The efficiency of these architectures is highly dependent on the amount of parallelism extracted both by the compiler and the hardware. Besides the hardware limitations, the barriers to parallelism are the dependencies existing between instructions of a program. These dependencies are classified as 1) *control dependencies*, arising from the control flow of the program, and 2) *data dependencies*, which are constraints from the flow of data between statements.

One of the techniques to achieve high parallelism is to avoid the limitations imposed by control dependencies by predicting the control flow and executing statements speculatively. By predicting the outcome of branches, the instruction stream pipeline can be prevented from stalling. Branch prediction has been shown to be highly successful in predicting the outcome of branches and avoiding instruction pipeline stalls. Current branch predictors have been shown to accurately predict more than 98% of the branches [10] and even approach similar numbers for branch intensive benchmarks [17].

The data dependencies existing between the instructions are more prevalent than control dependencies. An obvious way to overcome these dependencies is to perform speculation on data as well. The data speculation could be performed based on the storage addresses or data values. The prediction of storage addresses has been applied to perform memory disambiguation [13], as well as speculative use of load values [6]. The other method of predicting data values has only recently been addressed. Recent studies have shown that the benefits of value prediction have significant potential for performance improvement [1, 2, 4, 16].

Data value prediction is not as straightforward as branch prediction. In branch prediction, we need to choose one out of two possible outcomes of the branch. On the other hand, data values can encompass the entire range of values a register can hold. However, studies performed to observe the patterns of values of instructions have found that instructions do demonstrate locality of data values [3]. Based on this observation, interest to determine ways of predicting instruction sequences of values is growing. The sequences that occur in certain patterns are easier to predict than others. Typically, the patterns of values include:

- Constant value - these are the easiest to predict by always predicting the same constant value. Constant values have been shown to occur often in benchmark programs [3].

- Values differing by a stride - these usually occur in case of loop induction variables or variables that are dependent on induction variables. Constant values are a special case of this class since they can be viewed as variables with strides of zero.

- Other sequences - these include sequence of values that differ by non-constant strides. This category is the hardest to predict.

The techniques that perform prediction based on a local context of an instruction include **last value prediction** in which the value predicted is the one that was computed in the last execution of that instruction. Previous work using last value prediction indicates successful prediction for 40-50% of the dynamic instruction instances [4]. Another technique is **stride prediction**, that predicts values to be the sum of the last computed value of the same instruction and a stride. Stride prediction has been compared with last value prediction and shown to improve the prediction performance significantly [15]. In other work, **finite context method** (FCM) predictors are proposed that predict values based on previously observed patterns computed by the same instruction [1]. Recent values generated by the same instruction are used to form the patterns for prediction and these predictors have been shown to improve the accuracy of last value and stride predictors by as much as 20% . However, the study used upper bound conditions, that is, unbounded history table sizes. Also, **hybrid predictors** [11] have been proposed that attempt to detect sequences that are composition of stride and non-stride sequences. Due to their complexity, the FCM and hybrid predictors are difficult to implement efficiently.

This paper proposes techniques that associate a more global context with an instruction to help predict its value. This global context association is analogous to branch prediction performed by correlating with other branches [7] and has not been considered for value prediction before. Two types of global contexts are considered.

- Path information (branch history): Different values for an instruction are predicted along different paths within the program. Branch history is used to separate the different paths.

In the work of Lipasti[5], branch history is used to either select instructions for prediction, or predict input operands for estimating data dependencies However, branch history is not involved in the actual prediction of computed values.

- Values produced by recently completed instructions: Values are predicted by using the correlation between values produced by instructions close to each other within the dynamic instruction sequence.

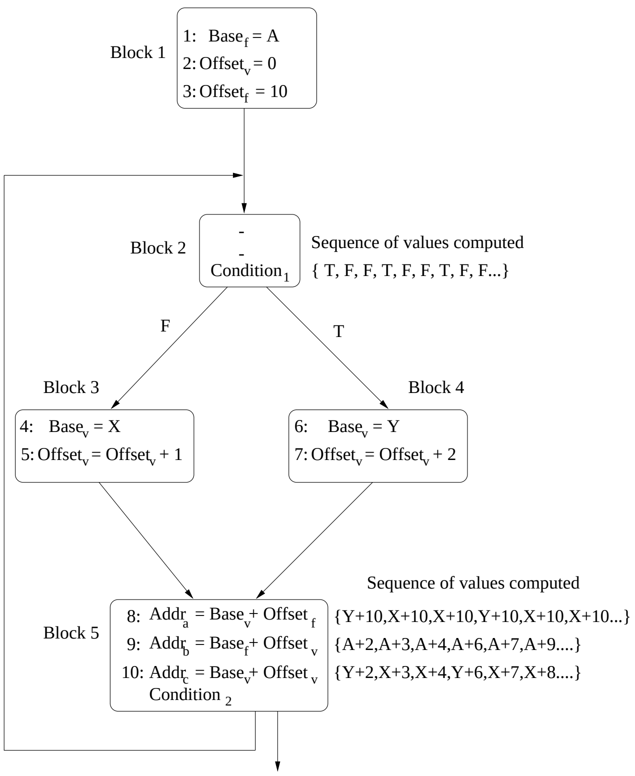


Figure 1: Example

Consider the code sequence shown in Figure 1. Instructions 8, 9 and 10 in block 5 compute three address values from previously defined base and offset values. $Addr_a$ depends on a base value that has either the value Y or X, depending on whether the condition $Condition_1$ evaluates to true or false, while the offset value is fixed at 10 for all loop iterations. $Addr_b$ uses a base value that is fixed for all loop iterations and an offset that is incremented by either 1 or 2, depending on the outcome of $Condition_1$. Finally, $Addr_c$ depends on base and offset values, both of which are different for the two paths within the loop.

Let us now consider how to predict these computed values assuming the *true, false, false* sequence of outcomes for $Condition_1$ repeats itself. The conventional last value and stride predictors would not work in this case. It is possible to perform prediction using the FCM or the hybrid predictor by maintaining the patterns of values. However, if the branch outcomes are arbitrary, the number of patterns of values generated would be high. This would result in frequent mispredictions because of new patterns being generated often.

It is possible to predict these values using control flow information. Consider instruction 8 first. The sequence of its computed values is as shown in the figure. We can predict the next value by looking at the most recent outcome of $Condition_1$. If $Condition_1$ evaluates to false/true, the next value will be the last value of instruction 8 along the false/true branch (i.e. *Y+10/X+10*).

Now consider the instruction 9 for prediction. From the sequence of values shown, we notice that this sequence is similar to a stride sequence. In case the true branch is taken, the stride is 2; otherwise it is 1. In this case, the next value can be predicted from the stride pattern of the offset occurring along a path.

Finally, the computed value of instruction 10 depends on base and offset values, both of which depend on the control path taken from block 2 to block 5. In this case, it is possible to perform prediction by remembering both the last value and the stride values for each incoming path.

In all the above cases, we can perform the prediction for any arbitrary sequence of branch outcomes. Values of an instruction along different paths are stored separately and branch history is used to predict the value on the current path. Also, in this example, a branch history comprising of the last two branches is sufficient for correct prediction. In general, outcomes from several branches may be required to perform correct prediction.

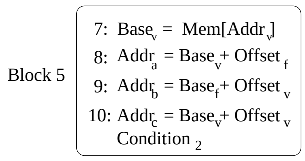


Figure 2: Modified block 5 of example

Let us modify basic block 5 in the example as shown in Figure 2 . The assignment to base value $Base_v$ is moved from blocks 3 and 4 to block 5. This value is now loaded from a memory location whose address, $Addr_v$, may be different for different iterations of the loop. In this case, the value of $Addr_a$ may not be predictable using path information since a different value can be produced for separate iterations of the loop along the same path. However, this value can be predicted as the sum of the loaded value of $Base_v$ and $Offset_f$. This implies that we can predict the value of $Addr_a$ by correlating with the value loaded by a previous instruction. Essentially, we need to capture the dependence occurring between the instruction computing $Base_v$ and the one using this value. Notice that the values that were predicted using path information could also be predicted using correlation with previous instructions. For example, values of $Addr_b$ and $Addr_c$ can be predicted by correlating with the instruction computing $Offset_v$.

In this paper, we perform value prediction for an instruction using the context derived from execution of other instructions. We present prediction techniques that combine the use of branch history with last value and stride predictors. Also, a technique that predicts values based on the correlations between instructions is developed. The predictors resulting from these techniques were experimentally evaluated and compared with conventional predictors.

The rest of the paper is organized as follows. Sections 2, 3 and 4 present the new prediction schemes and evaluations of their performances. The conclusions are presented in Section 5.

2 Path-based Last Value Prediction

The first scheme proposed, extends the last value predictor by storing the most recent values of an instruction for different branch histories. The history of recent branch outcomes is maintained, and used along with the instruction address to predict the next value of the instruction. For the example of last section, two different values of $Addr_a$ are stored and then, based on the path followed, the appropriate value computed on that path is predicted.

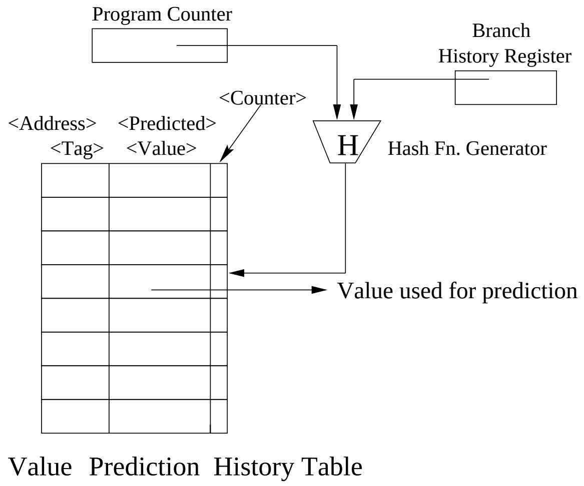


Figure 3: Per path Last value Predictor Microarchitecture

This implementation of this predictor is depicted in Figure 3. The components of the predictor are as follows:

- **Branch History Register (BHR)**: This register is similar to the history registers used in branch prediction techniques [10, 7, 17]. The register stores the most recent branch outcomes, initialized to zero. After each branch is evaluated, the entire history contents are shifted left by one bit and the least significant bit is set/reset depending on whether the branch was taken/not taken. The number of paths that can be represented is limited by the size of the register.

- **Value Prediction History Table (VPHT)**: This table stores the values of instructions for different branch histories. It differs from the value prediction table proposed in [4] in that this table may store several values computed by an instruction for different branch histories. The table is set-associative and accessed using the instruction address and BHR bits. Each entry stores the predicted value and a saturating counter for performing replacement. The counter is incremented after a correct prediction and decremented upon a misprediction. Whenever an entry needs to be replaced, the one with the lowest counter value is selected for replacement.

- **Hash Function Generator** : The VPHT is accessed using both the address of the instruction and the history register. This unit performs the hashing of the two to use for indexing the VPHT. For instruction i and BHR's value b, the function used is

  $$f_1(i,b) = ((Addr(i) << Size(BHR)) + b) \bmod Size(VPHT)$$

  where *Addr(i)* denotes the address value of instruction i and *Size(BHR)/Size(VPHT)* gives the total number of distinct entries in buffer BHR/VPHT. This function maps an instruction within a segment of the VPHT of size $2^{Size(BHR)}$. Within each segment, different branch histories for the same instruction get mapped to different entries of the table. This function is easy to implement in hardware and the experimental results presented throughout this paper are based upon this function.

For each instruction, the predictor computes the index in the VPHT by hashing the instruction's address bits with the current branch history contents. If the index maps to a valid entry, the corresponding value is used for prediction. After execution, the computed value is compared with the predicted one. Upon a misprediction, the instruction using the predicted value is squashed and re-issued with the correct value. At the instruction's completion, the information about its last value for the current branch history is updated in the VPHT.

The performance of this scheme was analyzed using trace-driven simulations and instrumenting the code to perform the prediction. The experimental framework consisted of an instruction-set simulator SHADE [9], a code instrumentation tool from SUN Microsystems, which simulates the SPARC (Versions 8 and 9) instruction sets. The benchmarks were first instrumented with SHADE and then executed over a simulated microarchitecture on a 64-bit SPARC Ultra-2 processor running SunOS 5.5.1. They were compiled using the SUN Workshop C Compiler, Version 4.2 with the -O option, and run until completion with different input parameters and files. The data input to the programs is described in Table 1. Only the integer instructions were analyzed in the benchmarks. Since previous work [4] shows that floating point instructions exhibit value prediction patterns similar to those of integer instructions, similar performance is expected for predicting floating point instructions.

We implemented the PLV predictor and compared its performance with the performance of the last value predictor and the FCM predictor. The FCM predictor was implemented as a two-level table. The first level table entries consisted of an address tag and partial values from the last three instances of the instruction. The address tag along with the stored values are hashed to a second table storing predicted values for the instruction. The hashing function used performs exclusive-OR on the most significant bits of the stored values.

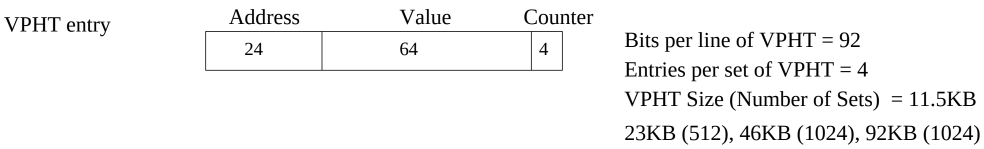


Figure 4: Table entry for PLV predictor

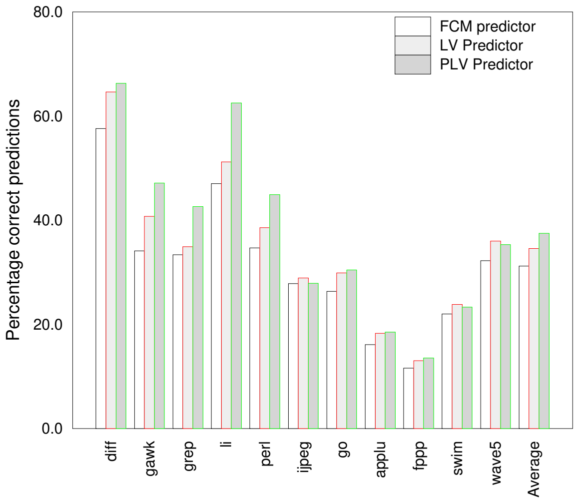
For each scheme, we performed experiments for different table sizes. The different sizes of

Benchmark	Input set	Instructions analyzed
Unix Utilities		
diff	4K C files	5.4M
gawk	parse 250K input	6.9M
grep	search 1.28M file	20.9M
SPEC INT 95 programs		
li	train.lsp	125.3M
perl	primes.in	10.3M
ijpeg	vigo.ppm	1282.4M
go	2stone9.in	397.1M
SPEC FP 95 programs		
applu	applu.in	147.1M
fpppp	natoms.in	112.5M
swim	swim.in	176.8M
wave5	wave5.in	934.5M

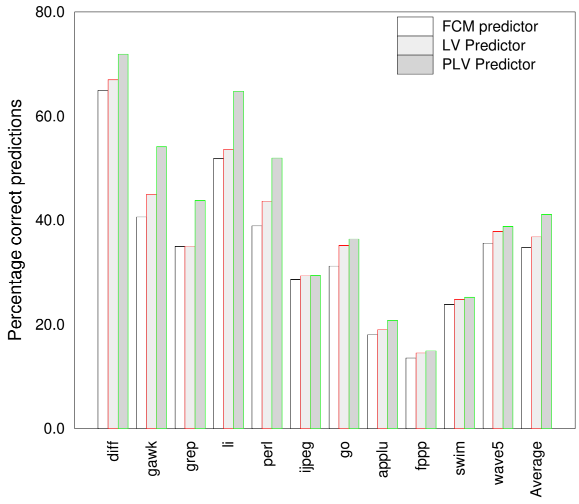
Table 1: Description of benchmarks

the VHPT in PLV predictor are shown in Figure 4. Each set consisted of four entries. In our experiments, a number of hashing functions were used for the PLV predictor, and the function f_1 described above, was found to be the most effective one. It is possible that some of the programs require only a small branch history depth and others require greater path information to be stored. We ran our experiments for three different branch history depths, last 2, 4 and last 8 branches. It was observed that for smaller table sizes, a small branch history of the last two branches usually performed best. For the largest table sizes of 4K entries, a larger branch history performed better since the effects of aliasing were lower. Overall the branch history size of two produced the best results as compared with the other history sizes , and to allow a fair comparison, we selected this number for all our experiments.

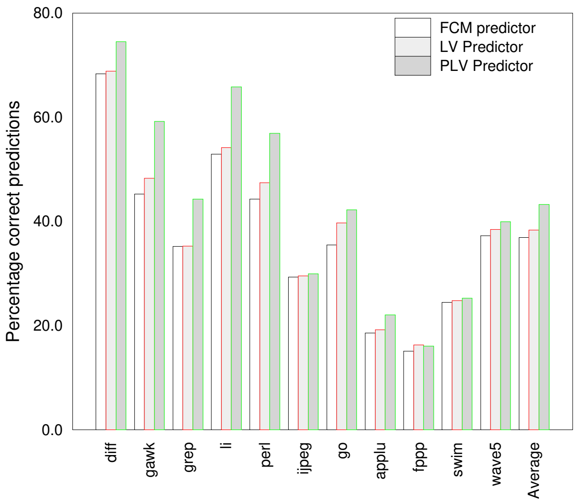
Performance comparison of the PLV predictor with the last value and the FCM predictors, for same number of entries per table, is shown in Figure 5. The figures show the percentage of analyzed instructions that were successfully predicted. From the comparisons depicted in the figures, it can be seen that the FCM method did not match up to the other techniques for realistic table sizes. This is due to the large number of patterns that need to be stored, resulting in frequent collisions within the tables. Also, the PLV predictor outperforms the conventional predictors for most of the cases. For VHPT set size of 256, there is slight degradation in performance in cases of *ijpeg*, *swim* and *wave5*, as compared to last value prediction. This can be attributed to the aliasing problem apparent for smaller table sizes, since we need to store several values for different branch histories for an instruction. With an increase in set size, the aliasing problem reduces and this results in a considerable performance improvement. The average performance improvement is 7% for the table size of 4K entries. For larger table sizes, this improvement is higher for larger branch history but this is not apparent from the results shown which use a fixed history size throughout. The predictor performs better for integer benchmarks as compared to floating point benchmarks.



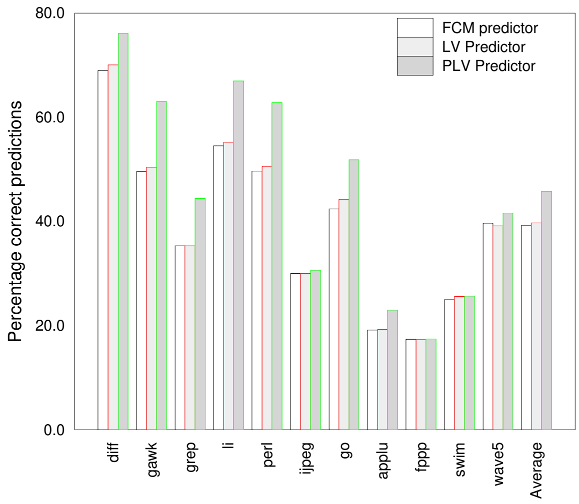
(a) 256 sets



(b) 512 sets



(c) 1024 sets



(d) 4096 sets

Figure 5: Performance of FCM vs. Last value vs. PLV Predictors for different sizes

In the above experiments, the branch history and predictor table state are updated immediately after the values are available. In a real machine, this update may take a few cycles, in which case the instruction is not allowed to execute immediately. In order to take these extra cycles into account, we ran the benchmarks using a delayed update of 2 cycles as an estimate of the extra cycles needed for an update. This resulted in reduction in prediction accuracy in the range of 2-7% in case of both conventional and the new prediction schemes. However, the average performance improvement decreased only by a fraction of a percentage indicating the improvement for the proposed predictors was not impacted severely by updating tables immediately.

3 Per-Path Stride Prediction

This section presents two prediction schemes that improve stride prediction by incorporating path information. The first of these two schemes, Per-Path Stride (PS) prediction stores different stride values for an instruction along different paths. The extension of stride prediction is analogous to extending last value prediction within the PLV predictor. Path information can be useful for stride prediction when an induction variable is updated by different amounts along different paths within a loop. In the example of Figure 1, the variable $Offset_v$ demonstrates this characteristic. While the strides are stored separately for different branch histories, the last value is stored globally for all the strides. The table storing the strides is accessed using the instruction address and branch history bits.

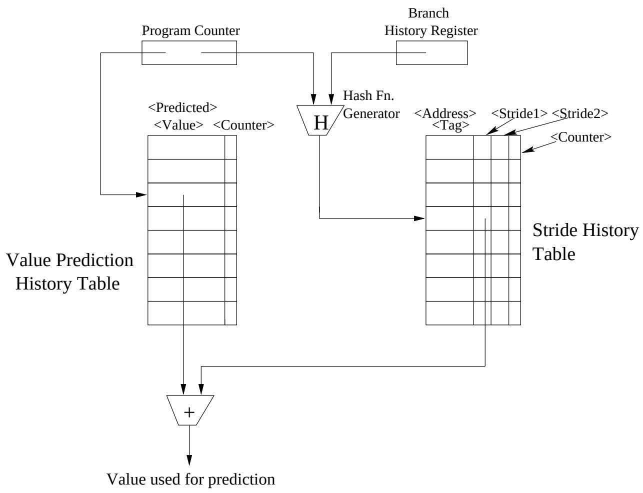


Figure 6: Per-path Stride Predictor Microarchitecture

The components of the predictor are shown in Figure 6. Besides using the components of the PLV predictor this scheme makes use of an additional table.

- **Stride History Table**: This table stores the stride values for different execution histories of an instruction. It is accessed by hashing the address bits and the branch history register value. The stride update policy used here is the **two-delta policy** [14]. In this policy, the stride gets updated only if the difference between the two most recent values of an instruction occurs twice in a row. To implement this policy, two strides are stored for an instruction. The first one stores the difference between the last two values computed by the instruction while the other stride stores the value being used for prediction. The latter value is updated when the first stride has the same value twice in a row.

For each analyzed instruction, the predictor uses its lower (k bits) to map to the VPHT. The last value of the instruction is picked from this table. Simultaneously, the address bits are hashed with the branch history to map to the SHT. This mapping accesses the stride used for the instruction and current branch history. The sum of the two values is used to predict the next value. Similar to the PLV scheme, the correctness of the prediction is checked after execution of the instruction is completed, and the update of the last value and stride is performed during completion.

The second stride path-based predictor scheme combines the last two proposed schemes by maintaining a last value for each path, as well as a stride value for each path. This scheme, Per-Path Stride Per-Path Last Value (PS-PLV) prediction, is used to handle conditions such as that of $Addr_c$ in the example of Figure 1, whose value depends on operands evaluated differently along different paths. The implementation is shown in Figure 7. It is similar to the PLV predictor with the exception that each entry in the VPHT consists of a last value and a stride.

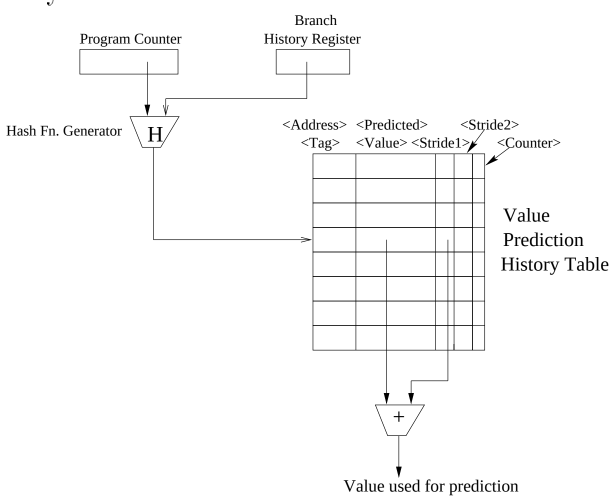


Figure 7: Per-path Stride Per-path Value Predictor Microarchitecture

Each instruction is hashed to index the VPHT based on its address and branch history. This table stores the last value as well as the strides for each branch history. A single access results in accessing both the values to be used for prediction. These values get updated during the completion stage of the instruction, as in the previous methods.

The configurations of PS and PS-PLV predictors that were used are shown in Figure 8. They were implemented and compared with conventional stride predictors. The performances of the PS and PS-PLV predictors are shown in Figure 9. Similar to the study for the PLV predictor, we chose a branch history size of two for our analysis. From the figure, we observe that the PS predictor almost always outperforms the conventional stride predictor (except for the benchmark *fpppp* which has a marginal degradation in performance). The PS-PLV predictor also gives a better performance when compared to the conventional predictor for most of the cases. However, the

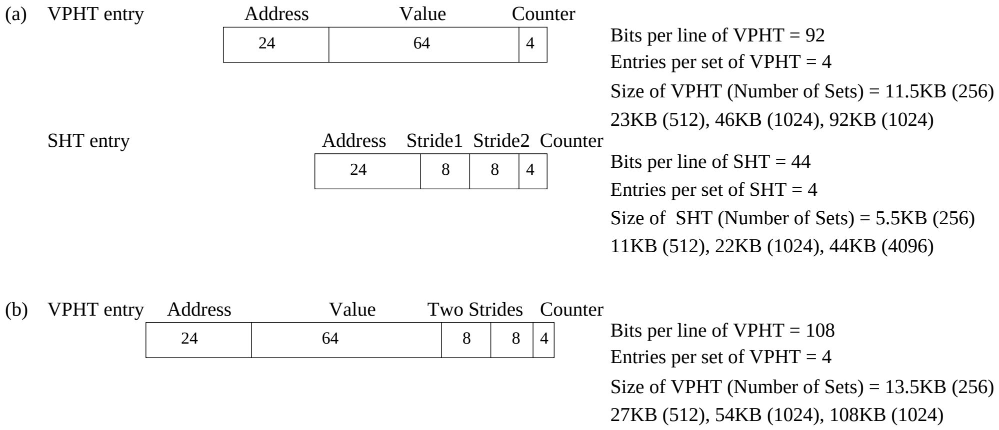


Figure 8: Table entries for (a) PS predictor (b) PS-PLV Predictor

performance improvement is less than with PS predictor. Overall, the PS predictor shows an average improvement of 8.9% for 4K entries while the PS-PLV predictor improves performance by an average of 7.4%.

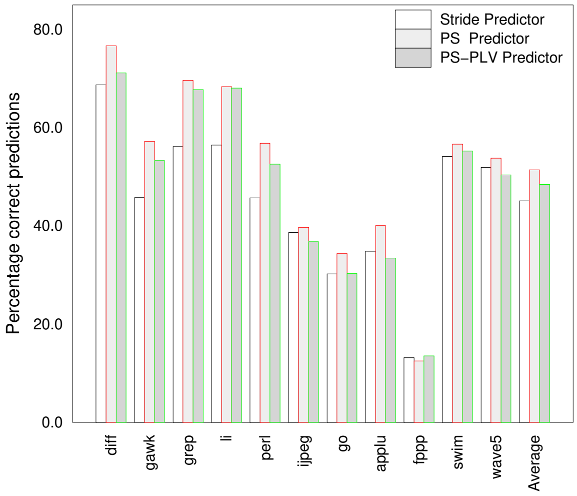
From the above results, we infer that the PS predictor improves conventional prediction most significantly, Notice that both PS and PS-PLV prediction schemes would be able to capture different prediction cases (as shown in the example of Section 1). Hence a hybrid predictor that uses both these prediction schemes would potentially give an additive improvement in performance. The main problem to be explored would be to select which predictor to use every time a prediction is to be made.

The performance of the new and conventional predictors reduced by 3-8% when the the predictors were implemented with delayed update as described in the previous section. But the overall reductions in performance improvements of the new predictors were marginal.

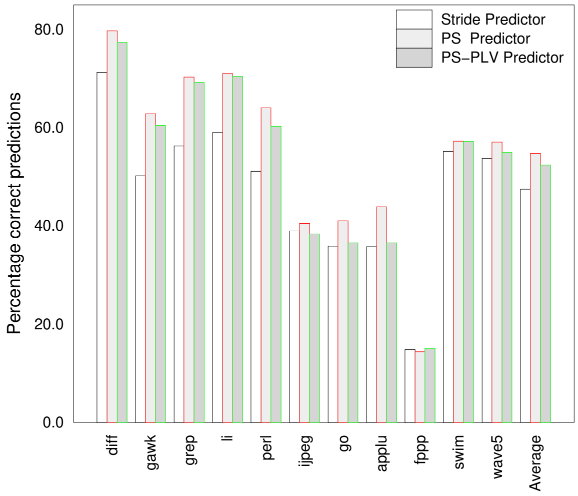
4 Path and Instruction-based Value Prediction

It is possible that path information may not be useful in prediction of an instruction value. This was demonstrated in the example when $Base_v$ was moved to block 5 in Figure 2, and $Addr_a$ could no longer be predicted using path history. A scheme that performs prediction of the value of an instruction by correlating with the previous instruction in the instruction sequence was shown to predict $Addr_a$ in the modified example. However, using simply the last instruction will not allow us to predict $Addr_b$ and $Addr_c$ in this case. In order to correlate with a previous instruction value, we need to store values generated by a number of previously completed instructions. The correlation is detected by selecting an instruction from these instructions and using its value to predict the current instruction's value.

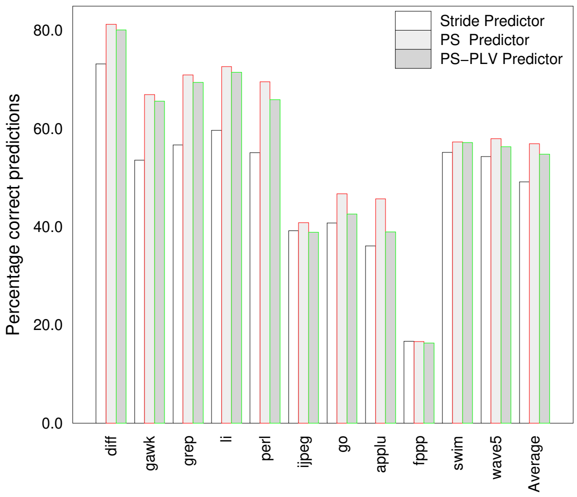
One of the issues here is which instruction to select out of the recently completed instructions. We performed experiments to observe the detectable correlation with recently completed instructions. From Figure 10, we notice that 37% of the instructions are correctly predictable due to their correlation with the immediately preceding instruction appearing in the dynamic sequence. There-



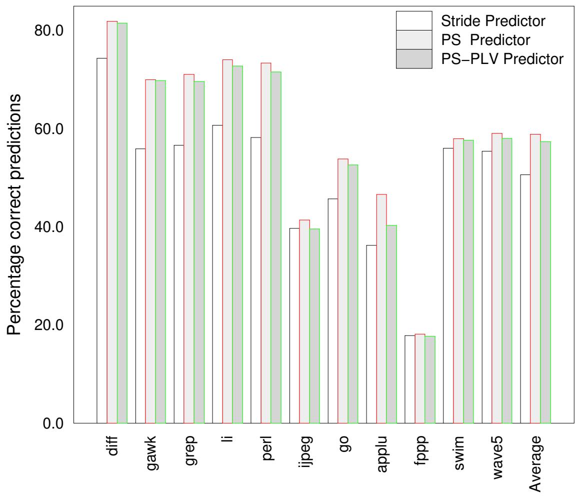
(a) 256 sets



(b) 512 sets



(c) 1024 sets



(d) 4096 sets

Figure 9: Performance of Stride vs. PS vs. PS-PLV Predictors for different sizes

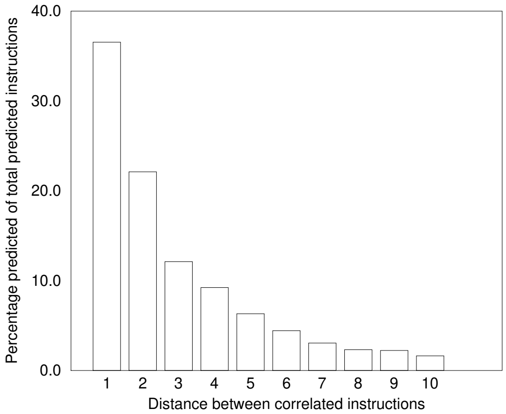


Figure 10: Percentage predicted vs. Distance between correlation

fore, to maintain simplicity of the predictor we choose to correlate the instruction to be predicted with the immediately preceding instruction

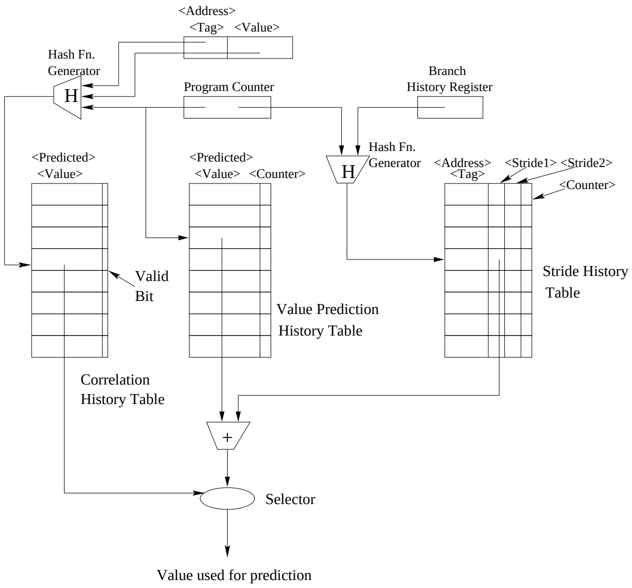


Figure 11: Per-Stride Previous Instruction-based Predictor Microarchitecture

The previous instruction-based predictor was implemented as a hybrid predictor in conjunction with the PS predictor. We implemented this predictor on top of the PS predictor to observe the additional improvement by prediction based on correlation. The PS predictor was chosen as part of the hybrid predictor since it produced the best performance out of the different prediction schemes that are presented. The new predictor, Per-Path Stride Per-Instruction (PS-PI), attempts to predict the instruction value using the PS predictor component first. If the instruction cannot be predicted, the instruction-based component of the PS-PI predictor would make a prediction on the instruction. This order of applying the two predictor components was chosen to reduce the number of entries stored in the predictor table of the instruction-based predictor. In this way, we can study the performance of the previous instruction-based predictor by reducing the effects of aliasing.

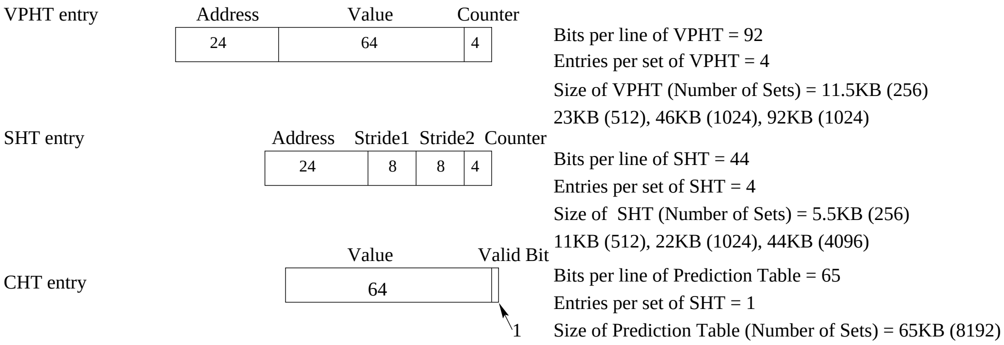


Figure 12: Table entries for PS-PI Predictor

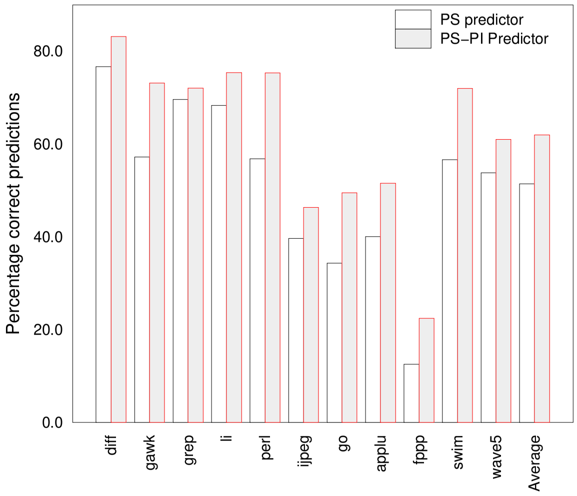
The implementation of the PS-PI predictor is shown in Figure 11. The left half of the figure depicts the previous instruction-based component of the predictor. It uses a table, **Correlation History Table (CHT)** which stores values correlated with other instruction values. Accessing this table is based on the values of the Program Counter and previous instruction address and data. The right half of the figure shows the PS component described in the last section. The selector chooses one of the two values predicted. As mentioned before, the previous instruction-based prediction is chosen if the PS component is unable to predict the value.

The size of the CHT was fixed to 8K entries. We analyze the performance of previous instruction-based correlation by comparing the PS-PI predictor with the PS predictor using the experimental design described in Section 2.1. The results are depicted in Figure 13. The graphs give the potential prediction available that is not captured by path information but may be exploited using correlation. From the graphs, we observe the improvement is significant for all benchmarks. The average improvement is 10.6% for table sizes of 256 entries and similar performance for other table sizes. The improvement percentages are similar in case of both integer and floating point benchmarks. This was not the case for path-based predictors where integer benchmarks had a higher improvement.

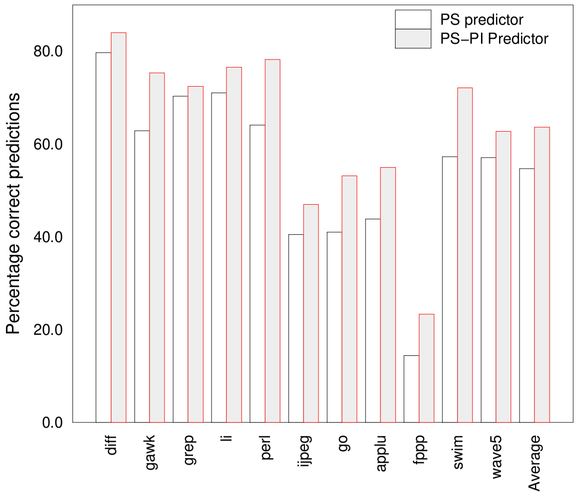
When the predictor was modified to incorporate delayed update of branch history and tables as discussed in the last two sections, the prediction accuracy reduced upto 10% for some benchmarks. However, the degradation in performance improvement over PS predictor was less than 2%.

5 Conclusions

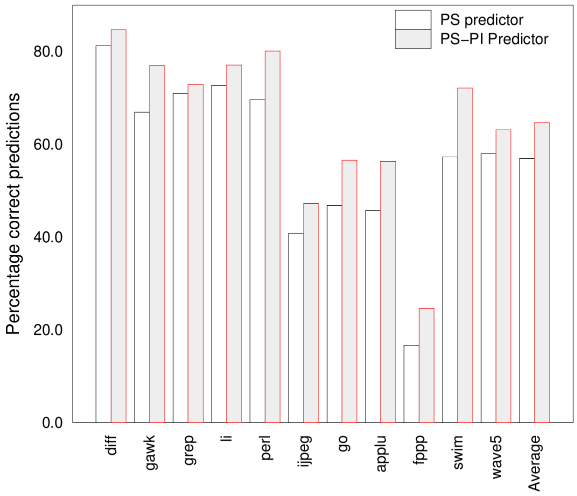
From our experiments, a number of conclusions can be drawn.



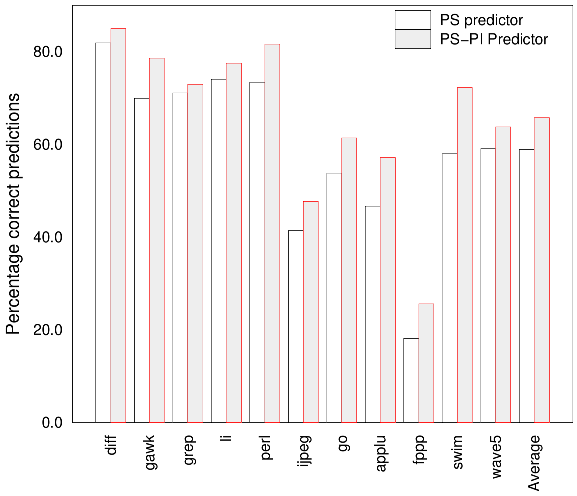
(a) 256 sets



(b) 512 sets



(c) 1024 sets



(d) 4096 sets

Figure 13: Performance of PS vs. PS-PI Predictors for different sizes

- *Path information is helpful in performing prediction of data values of instructions.* Three novel schemes have been proposed for performing value prediction sensitive to path information. The first scheme, PLV predictor, improves the last value prediction. The prediction improvement compared to last value prediction was found to be as high as 13% for some benchmarks and 7% on average. The other two proposed predictors, PS and PS-PLV, extend the predictors that predict values differing by strides. The PS prediction scheme shows most improvement in accurate predictions over conventional prediction schemes. This improvement exceeds 15% for some benchmarks and 8.9% on average; for PS-PLV prediction, the improvement exceeds 14% for some benchmarks and 7.4% on average. The PS and PS-PLV schemes are able to capture different patterns of values predictable using path information. Incorporating both of these in a hybrid predictor is suggested.

- *Prediction based on correlating instruction values can produce additional benefits over path-based predictors.* The proposed previous instruction-based prediction scheme is incorporated within a hybrid predictor, PS-PI. Analysis of the PS-PI predictor indicates that this method can capture several cases of instruction values that are not predictable by path-based predictors. The percentage instructions of this kind exceeds 19% for some benchmarks and 10.8% on average. The instruction-based predictor used in this paper considers only the last instruction in the dynamic sequence for prediction. Incorporating several instructions as part of the history is expected to produce further improvement.

- *Significant performance improvement is attainable with a simple hardware scheme and realistic table sizes.* The proposed prediction schemes provide improvement over the conventional last value and stride predictors for limited number of entries per table. The FCM predictor does not match up to these predictors for the table sizes under consideration. We have shown that each of the proposed schemes would require very simple hardware for implementation. Analysis of these schemes on these hardware indicates that aliasing is an important parameter for performance. The branch history information, used to differentiate values computed by the same instruction along different paths, also affects the performance significantly.

- *Delaying update of information as is done in real microarchitecture does not affect the performance significantly.* We perform experiments by updating branch history a few cycles after the branch outcome is available. This attempts to give a more realistic view of the performance and results indicate that the degradation in performance is minimal.

References

[1] Y. Sazeides and J.E. Smith, "The Predictabilty of Data Values," *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248-258, Dec. 1997.

[2] Y. Sazeides, S. Vassiliadis and J.E. Smith, "The performance potential of data dependence speculation and collapsing ," *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 238-247, Dec. 1996.

[3] M.H. Lipasti, C.B. Wilkerson and J.P. Shen, "Value Locality and Load Value Prediction," *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138-147, Oct. 1996.

[4] M.H. Lipasti and J.P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*", pages 226-237, Dec. 1996.

[5] M.H. Lipasti, "Value Locality and Speculative Execution," *Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University*", May 1997.

[6] A. Moshovos and G. Sohi, "Streamlining Inter-Operation Memory Communication via Data Dependence Prediction," *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 235-245, Dec. 1997.

[7] T.Y. Yeh and Y.N. Patt, "Alternate Implementation of Two-level Adaptive branch Prediction," *Proceedings of the 19th International Symposium on Computer Architecture*", pages 124-134, May 1992.

[8] S-T. Pan, K. So and J.T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76-84, Oct. 1992.

[9] R. F. Cmelik and D. Keppel, "Shade: A Fast Instruction Set Simulator for Execution Profiling," Tech. Rep. TR-93-12, Sun Microsystems Laboratories, July 1993.

[10] S. McFarling, "Combining Branch Predictors," Tech. Rep. DEC WRL TN-36, Digital Western Research Laboratory, June 1993.

[11] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 281-290, Dec. 1997.

[12] T-F Chen and J-L Baer "A Performance Study of Software and Hardware Data Prefetching schemes," *Proceedings of the 21st International Symposium on Computer Architecture*," 223-232, April 1994.

[13] D.M.Gallagher, W.Y. Chen, S.A. Mahalke, J.C. Gyllenhaal and W.W. Hwu "Dynamic memory disambiguation using the memory conflict buffer," *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183-193, Oct. 1994.

[14] R.J.Eickemeyer and S.Vassiliadis "A load instruction unit for pipelined processors," *IBM Journal of Research and Development* , pages 547-564, July 1993.

[15] F.Gabbay and A.Mendelson "Speculative execution based on value prediction," *EE Department TR#1080, Technion - Israel Institute of Technology* , November 1996.

[16] F.Gabbay and A.Mendelson "Can Program Profiling Support Value Prediction?," *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 270-280, Dec. 1997.

[17] Po-Yung Chang, Eric Hao, Tse-Yu Yeh and Yale Patt "Branch Classification: A New Mechanism for Improving Branch Predictor Performance," *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 22-31, November 1994.