The Predictability of Data Values

Yiannakis Sazeides and James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
1415 Engr. Dr.
Madison, WI 53706
{yanos,jes}@ece.wisc.edu

The Predictability of Data Values

Yiannakis Sazeides and James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
1415 Engr. Dr.
Madison, WI 53706
{yanos,jes}@ece.wisc.edu

Abstract

The predictability of data values is studied at a fundamental level. Two basic predictor models are defined: **Computational** *predictors perform an operation on previous values to yield predicted next values. Examples we study are stride value prediction (which adds a delta to a previous value) and last value prediction (which performs the trivial identity operation on the previous value);* **Context Based** *predictors match recent value history (context) with previous value history and predict values based entirely on previously observed patterns.*

To understand the potential of value prediction we perform simulations with unbounded prediction tables that are immediately updated using correct data values. Simulations of integer SPEC95 benchmarks show that data values can be highly predictable. Best performance is obtained with context based predictors; overall prediction accuracies are between 56% and 91%. The context based predictor typically has an accuracy about 20% better than the computational predictors (last value and stride). Comparison of context based prediction and stride prediction shows that the higher accuracy of context based prediction is due to relatively few static instructions giving large improvements; this suggests the usefulness of hybrid predictors. Among different instruction types, predictability varies significantly. In general, load and shift instructions are more difficult to predict correctly, whereas add instructions are more predictable.

1 Introduction

There is a clear trend in high performance processors toward performing operations speculatively, based on predictions. If predictions are correct, the speculatively executed instructions usually translate into improved performance.

Although program execution contains a variety of information that can be predicted, conditional branches have received the most attention. Predicting conditional branches provides a way of avoiding control dependences and offers a clear performance advantage. Even more prevalent than control dependences, however, are data dependences. Virtually every instruction depends on the result of some preceding instruction. As such, data dependences are often thought to present a fundamental performance barrier. However, data values may also be predicted, and operations can be performed speculatively based on these data predictions.

An important difference between conditional branch prediction and data value prediction is that data are taken from a much larger range of values. This would appear to severely limit the chances of successful prediction. However, it has been demonstrated recently [1] that data values exhibit "locality" where values computed by some instructions tend to repeat a large fraction of the time.

We argue that establishing predictability limits for program values is important for determining the performance potential of processors that use value prediction. We believe that doing so first requires understanding the design space of value predictors models. Consequently, the goals of this paper are twofold. Firstly, we discuss some of the major issues affecting data value prediction and lay down a framework for studying data value prediction. Secondly, for important classes of predictors, we use benchmark programs to establish levels of value predictability. This study is somewhat idealized: for example, predictor costs are ignored in order to more clearly understand limits of data predictability. Furthermore, the ways in which data prediction can be used in a processor microarchitecture are not within the scope of this paper, so that we can concentrate in greater depth on the prediction process, itself.

1.1 Classification of Value Sequences

The predictability of a sequence of values is a function of both the sequence itself and the predictor used to predict the sequence. Although it is beyond the scope of this paper to study the actual sources of predictability, it is useful for our discussion to provide an informal classification of data sequences. This classification is useful for understanding the behavior of predictors in later discussions. The following classification contains simple value sequences that can also be composed to form more complex sequences. They are best defined by giving examples:

Constant(C)	5 5 5 5 5 5 5...
Stride(S)	1 2 3 4 5 6 7 8...
Non-Stride(NS)	28 -13 -99 107 23 456...

Constant sequences are the simplest, and result from instructions that repeatedly produce the same result. Lipasti and Shen show that this occurs surprisingly often, and forms the basis for their work reported in [1]. A stride sequence has elements that differ by some constant delta. For the example above, the stride is one, which is probably the most common case in programs, but other strides are possible, including negative strides. Constant sequences can be considered stride sequences with a zero delta. A stride sequence might appear when a data structure such as an array is being accessed in a regular fashion; loop induction variables also have a stride characteristic.

The non-stride category is intended to include all other sequences that do not belong to the constant or stride category. This classification could be further divided, but we choose not to do so. Non-strides may occur when a sequence of numbers is being computed and the computation is more complex than simply adding a constant. Traversing a linked list would often produce address values that have a non-stride pattern.

Also very important are sequences formed by composing stride and non-stride sequences with themselves. Repeating sequences would typically occur in nested loops where the inner loop produces either a stride or a non-stride sequence, and the outer loop causes this sequence to be repeated.

Repeated Stride(RS)	1 2 3 1 2 3 1 2 3...
Repeated Non-Stride(RNS)	1 -13 -99 7 1 -13 -99 7...

Examination of the above sequences leads naturally to two types of prediction models that are the subject of discussion throughout the remainder of this paper:

Computational predictors that make a prediction by computing some function of previous values. An example of a computational predictor is a stride predictor. This predictor adds a stride to the previous value.

Context based predictors learn the value(s) that follow a particular context - a finite ordered sequence of values - and predict one of the values when the same context repeats. This enables the prediction of any repeated sequence, stride or non-stride.

1.2 Related Work

In [1], it was reported that data values produced by instructions exhibit "locality" and as a result can be predicted. The potential for value predictability was reported in terms of "history depth", that is, how many times a value produced by an instruction repeats when checked against the most recent n values. A pronounced difference is observed between the locality with history depth 1 and history depth 16. The mechanism proposed for prediction, however, exploits the locality of history depth 1 and is based on predicting that the most recent value will also be the next. In [1], last value prediction was used to predict load values and in a subsequent work to predict all values produced by instructions and written to registers [2].

Address prediction has been used mainly for data prefetching to tolerate long memory latency [3, 4, 5], and has been proposed for speculative execution of load and store instructions [6, 7]. Stride prediction for values was proposed in [8] and its prediction and performance potential was compared against last value prediction.

Value prediction can draw from a wealth of work on the prediction of control dependences [9, 10, 11]. The majority of improvements in the performance of control flow predictors has been obtained by using correlation. The correlation information that has been proposed includes local and global branch history [10], path address history [11, 12, 13], and path register contents [14]. An interesting theoretical observation is the resemblance of the predictors used for control dependence prediction to the prediction models for text compression [15]. This is an important observation because it re-enforces the approach used for control flow prediction and also suggests that compression-like methods can also be used for data value prediction.

A number of interesting studies report on the importance of predicting and eliminating data dependences. Moshovos [16] proposes mechanisms that reduce misspeculation by predicting when dependences exist between store and load instructions. The potential of data dependence elimination using prediction and speculation in combination with collapsing was examined in [17]. Elimination of redundant computation is the theme of a number of software/hardware proposals [18, 19, 20]. These schemes are similar in that they store in a cache the input and output parameters of a function and when the same inputs are detected the output is used without performing the function. Virtually all proposed schemes perform predictions based on previous architected state and values. Notable exceptions to this are the schemes proposed in [6], where it is predicted that a fetched load instruction has no dependence and the instruction is executed "early" without dependence checking, and in [21], where it is predicted that the operation required to calculate an effective address using two operands is a logical *or* instead of a binary addition.

In more theoretical work, Hammerstrom [22] used information theory to study the information content (entropy) of programs. His study of the information content of address and instruction streams revealed a high degree of redundancy. This high degree of redundancy immediately suggests predictability.

1.3 Paper Overview

The paper is organized as follows: in Section 2, different data value predictors are described. Section 3 discusses the methodology used for data prediction simulations. The results obtained are presented and analyzed in Section 4. We conclude with suggestions for future research in Section 5.

2 Data Value Prediction Models

A typical data value predictor takes microarchitecture state information as input, accesses a table, and produces a prediction. Subsequently, the table is updated with state information to help make future predictions. The state information could consist of register values, PC values, instruction fields, control bits in various pipeline stages, etc.

The variety and combinations of state information are almost limitless. Therefore, in this study, we restrict ourselves to predictors that use only the program counter value of the instruction being predicted to access the prediction table(s). The tables are updated using data values produced by the instruction – possibly modified or combined with other information already in the table. These restrictions define a relatively fundamental class of data value predictors. Nevertheless, predictors using other state information deserve study and could provide a higher level of predictability than is reported here.

For the remainder of this paper, we further classify data value predictors into two types – computational and context-based. We describe each in detail in the next two subsections.

2.1 Computational Predictors

Computational predictors make predictions by performing some operation on previous values that an instruction has generated. We focus on two important members of this class.

Last Value Predictors perform a trivial computational operation: the identity function. In its simplest form, if the most recent value produced by an instruction is $\mathbf{v}$ then the prediction for the next value will also be $\mathbf{v}$. However, there are a number of variants that modify replacement policies based on hysteresis. An example of a hysteresis mechanism is a saturating counter that is associated with each table entry. The counter is incremented/decremented on prediction success/failure with the value held in the table replaced only when the count is below some threshold. Another hysteresis mechanism does not change its prediction to a new value until the new value occurs a specific number of times in succession. A subtle difference between the two forms of hysteresis is that the former changes to a new prediction following incorrect behavior (even though that behavior may be inconsistent), whereas the latter changes to a new prediction only after it has been consistently observed.

Stride Predictors in their simplest form predict the next value by adding the sum of the most recent value to the difference of the two most recent values produced by an instruction. That is if $\mathbf{v}_{n-1}$ and $\mathbf{v}_{n-2}$ are the two most recent values, then the predictor computes $\mathbf{v}_{n-1} + (\mathbf{v}_{n-1} - \mathbf{v}_{n-2})$.

As with the last value predictors, there are important variations that use hysteresis. In [7] the stride is only changed if a saturating counter that is incremented/decremented on success/failure of the predictions is below a certain threshold. This reduces the number of mispredictions in repeated stride sequences from two per repeated sequence to one. Another policy, the **two-delta** method, was proposed in [6]. In the two-delta method, two strides are maintained. The one stride (*s1*) is always updated by the difference between the two most recent values, whereas the other (*s2*) is the stride used for computing the predictions. When stride s1 occurs twice in a row then it is used to update the prediction stride s2. The two-delta strategy also reduces mispredictions to one per iteration for repeated stride sequences and, in addition, only changes the stride when the same stride occurs twice - instead of changing the stride following mispredictions.

Other Computational Predictors using more complex organizations can be conceived. For example, one could use two different strides, an "inner" one and an "outer" one – typically corresponding to loop nests – to eliminate the mispredictions that occur at the beginning of repeating stride sequences. This thought process illustrates a significant limitation of computational prediction: the designer must anticipate the computation to be used. One could carry this to ridiculous extremes. For example, one could envision a Fibonacci series predictor, and given a program that happens to compute a Fibonacci series, the predictor would do very well.

Going down this path would lead to large hybrid predictors that combine many special-case computational predictors with a "chooser" - as has been proposed for conditional branches in [23, 24]. While hybrid prediction for data values is in general a good idea, a potential pitfall is that it may yield an ever-escalating collection of computational predictors, each of which predicts a diminishing number of additional values not caught by the others.

In this study, we focus on last value and stride methods as primary examples of computational predictors. We also consider hybrid predictors involving these predictors and the context based predictors to be discussed in the next section.

2.2 Context Based Predictors

Context based predictors attempt to "learn" values that follow a particular context – a finite ordered sequence of previous values – and predict one of the values when the same context repeats. An important type of context based predictors is derived from finite context methods used in text compression [25].

Finite Context Method Predictors (fcm) rely on mechanisms that predict the next value based on a finite number of preceding values. An **order k** fcm predictor uses k preceding values. Fcms are constructed with counters that count the occurrences of a particular value immediately following a certain context (pattern). Thus for each context there must be, in general, as many counters as values that are found to follow the context. The predicted value is the one with the maximum count. Figure 1 shows fcm models of different orders and predictions for an example sequence.

In an actual implementation where it may be infeasible to maintain exact value counts, smaller counters may be used. The use of small counters comes from the area of text compression. With small counters, when one counter reaches the maximum count, all counters for the same context are reset by half. Small counters provide an advantage if heavier weighting should be given to more recent history instead of the entire history.

In general, n different fcm predictors of orders 0 to n-1 can be used for predicting the next value of a sequence, with the highest order predictor that has a context match being used to make the prediction. The combination of more than one prediction model is known as *blending* [25]. There are a number of variations of blending algorithms, depending on the information that is updated. *Full blending* updates all contexts, and *lazy exclusion* selects the prediction with the longer context match and only updates the counts for the predictions with the longer match or higher.

Other variations of fcm predictors can be devised by reducing the number of values that are maintained for a given context. For example, only one value per context might be maintained along with some update policy. Such policies can be based on hysteresis-type update policies as discussed above for last value and stride prediction.

Correlation predictors used for control dependence prediction strongly resemble context based prediction. As far as we know, context based prediction has not been considered for value prediction, though the last value predictor can be viewed as a 0th order fcm with only one prediction maintained per context.

2.3 An Initial Analysis

At this point, we briefly analyze and compare the proposed predictors using the simple pattern sequences shown in Section 1.1. This analysis highlights important issues as

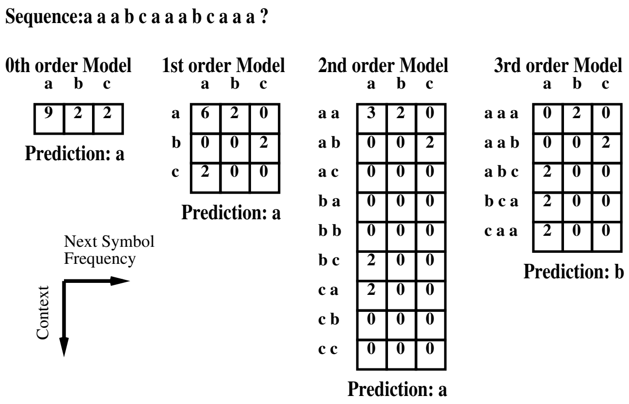

Figure 1: Finite Context Models

well as advantages and disadvantages of the predictors to be studied. As such, they can provide a basis for analyzing quantitative results given in the following sections.

We informally define two characteristics that are important for understanding prediction behavior. One is the Learning Time (**LT**) which is the number of values that have to be observed before the first correct prediction. The second is the Learning Degree (**LD**) which is the percentage of correct predictions following the first correct prediction.

We quantify these two characteristics for the classes of sequences given earlier in Section 1.1. For the repeating sequences, we associate a period (**p**), the number of values between repetitions, and frequency, the number of times a sequence is repeated. We assume repeating sequences where p is fixed. The frequency measure captures the finiteness of a repeating sequence. For context predictors, the order (**o**) of a predictor influences the learning time.

Table 1 summarizes how the different predictors perform for the basic value sequences. Note that the stride predictor uses hysteresis for updates, so it gets only one incorrect prediction per iteration through a sequence. A row of the table with a "-" indicates that the given predictor is not suitable for the given sequence, i.e., its performance is very low for that sequence.

As illustrated in the table, last value prediction is only useful for constant sequences – this is obvious. Stride prediction does as well as last value prediction for constant sequences because a constant sequence is essentially zero stride. The fcm predictors also do very well on constant sequences, but an order o predictor must see a length o sequence before it gets matches in the table (unless some form of blending is used).

For (non-repeating) stride sequences, only the stride

Sequence	Prediction Model					
	Last Value		Stride		FCM	
	LT	LD(%)	LT	LD(%)	LT	LD(%)
C	1	100	1	100	o	100
S	-	-	2	100	-	-
NS	-	-	-	-	-	-
RS	-	-	2	p-1/p	p+o	100
RNS	-	-	-	-	p+o	100

Table 1: Behavior of various Prediction Models for Different Value Sequences

predictor does well; it has a very short learning time and then achieves a 100% prediction rate. The fcm predictors cannot predict non-repeating sequences because they rely on repeating patterns.

For repeating stride sequences, both stride and fcm predictors do well. The stride predictor has a shorter learning time, and once it learns, it only gets a misprediction each time the sequence begins to repeat. On the other hand, the fcm predictor requires a longer learning time – it must see the entire sequence before it starts to predict correctly but once the sequence starts to repeat, it gets 100% accuracy (Figure 2). This example points out an important tradeoff between computational and context based predictors. The computational predictor often learns faster – but the context predictor tends to learn "better" when repeating sequences occur.

Finally, for repeating non-stride sequences, only the fcm predictor does well. And the flexibility this provides is clearly the strong point of fcm predictors. Returning to our Fibonacci series example – if there is a sequence containing a *repeating* portion of the Fibonacci series, then an fcm predictor will naturally begin predicting it correctly following the first pass through the sequence.

Of course, in reality, value sequences can be complex combinations of the simple sequences in Section 1.1, and a given program can produce about as many different sequences as instructions are being predicted. Consequently, in the remainder of the paper, we use simulations to get a more realistic idea of predictor performance for programs.

3 Simulation Methodology

We adopt an implementation-independent approach for studying predictability of data dependence values. The reason for this choice is to remove microarchitecture and other implementation idiosyncrasies in an effort to develop a basic understanding of predictability. Hence, these results can best be viewed as bounds on performance; it will take additional engineering research to develop realistic implementations.

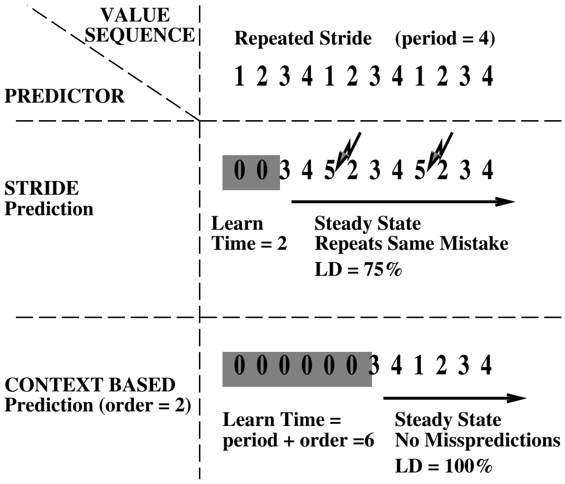

Figure 2: Computational vs Context Based Prediction

We study the predictability of instructions that write results into general purpose registers (i.e. memory addresses, stores, jumps and branches are not considered). Prediction was done with no table aliasing; each static instruction was given its own table entry. Hence, table sizes are effectively unbounded. Finally, prediction tables are updated immediately after a prediction is made, unlike the situation in practice where it may take many cycles for the actual data value to be known and available for prediction table updates.

We simulate three types of predictors: last value prediction (**l**) with an always-update policy (no hysteresis), stride prediction using the 2-delta method (**s2**), and a finite context method (**fcm**) that maintains exact counts for each value that follows a particular context and uses the blending algorithm with lazy exclusion, described in Section 2. Fcm predictors are studied for orders 1, 2 and 3. To form a context for the fcm predictor we use full concatenation of history values so there is no aliasing when matching contexts.

Trace driven simulation was conducted using the Simplescalar toolset [26] for the integer SPEC95 benchmarks shown in Table 2[1]. The benchmarks were compiled using the simplescalar compiler with -O3 optimization. Integer benchmarks were selected because they tend to have less data parallelism and may therefore benefit more from data predictions.

For collecting prediction results, instruction types were grouped into categories as shown in Table 3. The ab-

[1]For ijpeg the simulations used the reference flags with the following changes: compression.quality 45 and compression.smoothing_factor 45.

Benchmark	Input Flags	Dynamic Instr. (mil)	Instructions Predicted (mil)
compress	30000 e	8.2	5.8 (71%)
gcc	gcc.i	203	137 (68%)
go	9 9	132	105 (80%)
ijpeg	specmun.ppm	129	108 (84%)
m88k	ctl.raw	493	345 (70%)
perl	scrabbl.in	40	26 (65%)
xlisp	7 queens	202	125 (62%)

Table 2: Benchmarks Characteristics

Instruction Types	Code
Addition, Subtraction	AddSub
Loads	Loads
And, Or, Xor, Nor	Logic
Shifts	Shift
Compare and Set	Set
Multiply and Divide	MultDiv
Load immediate	Lui
Floating, Jump, Other	Other

Table 3: Instruction Categories

Type	com	gcc	go	ijpe	m88k	perl	xlis
AddSu	898	32770	18246	4434	3056	4099	1819
Loads	686	29138	9929	3645	2215	3855	1432
Logic	149	2600	215	278	674	460	157
Shift	146	5073	4500	712	581	467	154
Set	79	3099	1401	402	278	357	75
MultDi	19	313	196	222	77	26	25
Lui	116	3289	5679	343	564	284	146
Other	108	5848	1403	517	482	778	455

Table 4: Predicted Instructions - Static Count

Type	com	gcc	go	ijpe	m88k	perl	xlis
AddSu	42.6	38.9	42.1	52.4	42.6	34.1	36.1
Loads	20.5	38.6	26.2	21.4	24.8	43.1	48.6
Logic	3.1	3.1	0.5	1.9	5.0	3.1	3.4
Shift	17.4	7.7	13.3	16.4	3.2	8.2	3.2
Set	7.4	5.4	4.9	4.2	15.2	5.6	3.2
MultDi	0.01	0.4	0.3	3.2	0.1	0.2	0.01
Lui	3.3	3.7	11.4	0.2	6.9	2.4	0.8
Other	5.7	2.1	1.3	0.3	2.1	3.3	4.8

Table 5: Predicted Instructions - Dynamic(%)

breviations shown after each group will be used subsequently when results are presented. The percentage of predicted instructions in the different benchmarks ranged between 62%–84%. Recall that some instructions like stores, branches and jumps are not predicted. A breakdown of the static count and dynamic percentages of predicted instruction types is shown in Tables 4-5. The majority of predicted values are the results of addition and load instructions.

We collected results for each instruction type. However, we do not discuss results for the **other**, **multdiv** and **lui** instruction types due to space limitations. In the benchmarks we studied, the multdiv instructions are not a significant contributor to dynamic instruction count, and the lui and "other" instructions rarely generate more than one unique value and are over 95% predictable by all predictors. We note that the effect of these three types of instructions is included in the calculations for the **overall** results.

For averaging we used arithmetic mean, so each benchmark effectively contributes the same number of total predictions.

4 Simulation Results

4.1 Predictability

Figure 3 shows the overall predictability for the selected benchmarks, and Figures 4-7 show results for the important instruction types. From the figures we can draw a number of conclusions. Overall, last value prediction is less accurate than stride prediction, and stride prediction is less accurate than fcm prediction. Last value prediction varies in accuracy from about 23% to 61% with an average of about 40%. This is in agreement with the results obtained in [2]. Stride prediction provides accuracy of between 38% and 80% with an average of about 56%. Fcm predictors of orders 1, 2, and 3 all perform better than stride prediction; and the higher the order, the higher the accuracy. The order 3 predictor is best and gives accuracies of between 56% and over 90% with an average of 78%. For the three fcm predictors studied, improvements diminish as the order is increased. In particular, we observe that for every additional value in the context the performance gain is halved. The effect on predictability with increasing order is examined in more detail in Section 4.4. Performance of the stride and last value predictors varies significantly across different instruction types for the same benchmark. The performance of the fcm predictors varies less significantly across different instruction types for the same benchmark. This reflects the flexibility of the fcm predictors – they perform well for any repeating sequence, not just strides.

In general both stride and fcm prediction appear to have higher predictability for add/subtracts than loads. Logical instructions also appear to be very predictable especially by the fcm predictors. Shift instructions appear to be the most difficult to predict.

Stride prediction does particularly well for add/subtract

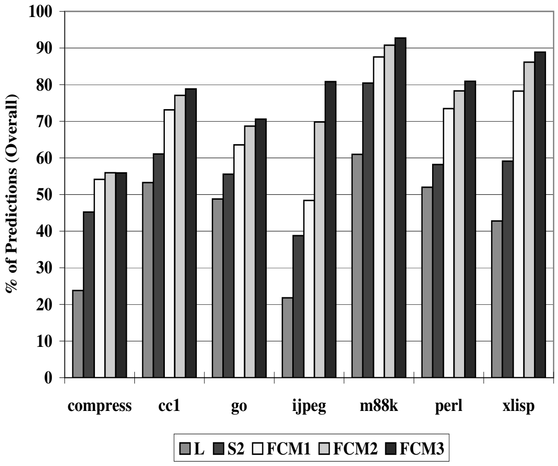

Figure 3: Prediction Success for All Instructions

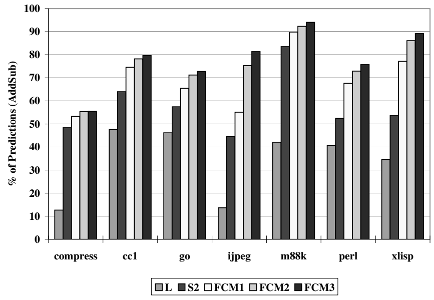

Figure 4: Prediction Success for Add/Subtract Instructions

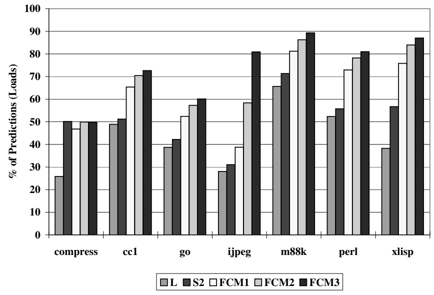

Figure 5: Prediction Success for Loads Instructions

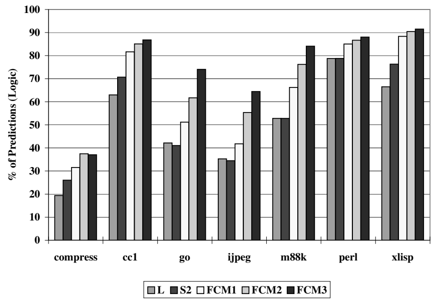

Figure 6: Prediction Success for Logic Instructions

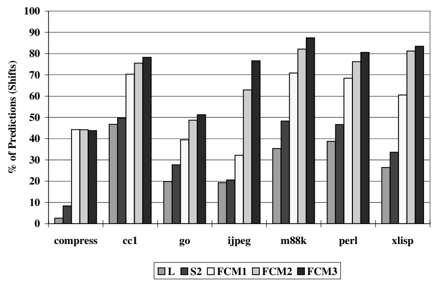

Figure 7: Prediction Success for Shift Instructions

instructions. But for non-add/subtract instructions the performance of the stride predictor is close to last value prediction. This indicates that when the operation of a computational predictor matches the operation of the instruction (e.g. addition), higher predictability can be expected. This suggests new computational predictors that better capture the functionality of non-add/subtract instructions could be useful. For example, for shifts a computational predictor might shift the last value according to the last shift distance to arrive at a prediction. This approach would tend to lead to hybrid predictors, however, with a separate component predictor for each instruction type.

4.2 Correlation of Correctly Predicted Sets

In effect, the results in the previous section essentially compare the sizes of the sets of correctly predicted values. It is also interesting to consider relationships among the specific sets of correctly predicted values. Primarily, these relationships suggest ways that hybrid predictors might be constructed – although the actual construction of hybrid predictors is beyond the scope of this paper.

The predicted set relationships are shown in Figure 8. Three predictors are used: last value, stride (delta-2), and fcm (order 3). All subsets of predictors are represented. Specifically: **l** is the fraction of predictions for which *only* the last value predictor is correct; **s** and **f** are similarly defined for the stride and fcm predictors respectively; **ls** is the fraction of predictions for which *both* the last value and the stride predictors are correct but the fcm predictor is not; **lf** and **sf** are similarly defined; **lsf** is the fraction of predictions for which all predictors are correct; and **np** is the fraction for which none of the predictors is correct.

In the figure results are averaged over all benchmarks, but the qualitative conclusions are similar for each of the

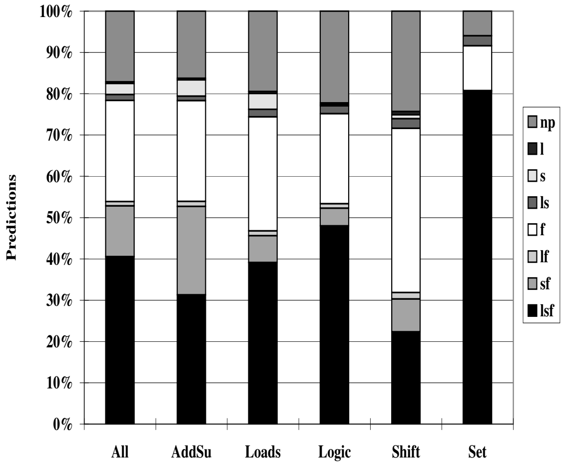

Figure 8: Contribution of different Predictors

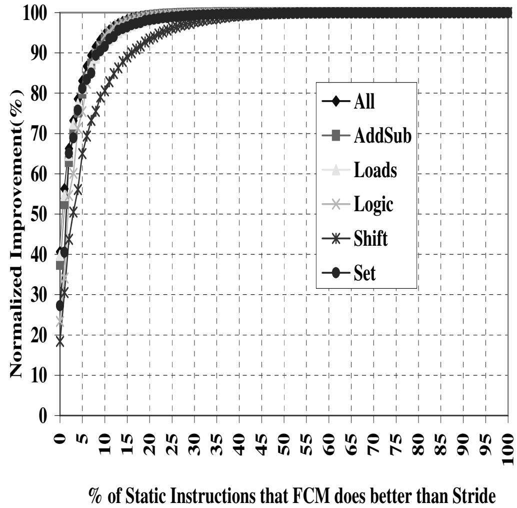

Figure 9: Cumulative Improvement of FCM over Stride

individual benchmarks. Overall, Figure 8 can be briefly summarized:

- A small number, close to 18%, of values are not predicted correctly by any model.
- A large portion, around 40%, of correct predictions is captured by all predictors.
- A significant fraction, over 20%, of correct predictions is only captured by fcm.
- Stride and last value prediction capture less than 5% of the correct predictions that fcm misses.

The above confirms that data values are very predictable. And it appears that context based prediction is necessary for achieving the highest levels of predictability. However, almost 60% of the correct predictions are also captured by the stride predictor. Assuming that context based prediction is the more expensive approach, this suggest that a **hybrid** scheme might be useful for enabling high prediction accuracies at lower cost. That is, one should try to use a stride predictor for most predictions, and use fcm prediction to get the remaining 20%.

Another conclusion is that last value prediction adds very little to what the other predictors achieve. So, if either stride or fcm prediction is implemented, there is no point in adding last value prediction to a hybrid predictor.

The important classes of load and add instructions yield results similar to the overall average. Finally, we note that for non-add/subtract instructions the contribution of stride prediction is smaller, this is likely due to the earlier observation that stride prediction does not match the functionality of other instruction types. This suggests a hybrid predictor based on instruction types.

Proceeding along the path of a hybrid fcm-stride predictor, one reasonable approach would be to choose among the two component predictors via the PC address of the instruction being predicted. This would appear to work well if the performance advantage of the fcm predictor is due to a relatively small number of static instructions.

To determine if this is true, we first constructed a list of static instructions for which the fcm predictor gives better performance. For each of these static instructions, we determined the difference in prediction accuracy between fcm and stride. We then sorted the static instructions in descending order of improvement. Then, in Figure 9 we graph the cumulative fraction of the total improvement versus the accumulated percentage of static instructions. The graph shows that overall, about 20% of the static instructions account for about 97% of the total improvement of fcm over stride prediction. For most of individual instruction types, the result is similar, with shifts showing slightly worse performance.

The results do suggest that improvements due to context based prediction are mainly due to a relatively small fraction of static instructions. Hence, a hybrid fcm-stride predictor with choosing seems to be a good approach.

4.3 Value Characteristics

At this point, it is clear that context based predictors perform well, but may require large tables that store history values. We assume unbounded tables in our study, but when real implementations are considered, of course this will not be possible. To get a handle on this issue, we study the value characteristics of instructions. In particu-

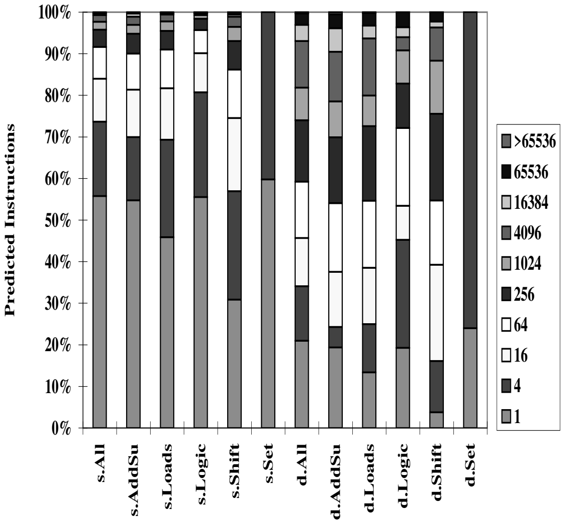

Figure 10: Values and Instruction Behavior

lar we report on the number of **unique** values generated by predicted instructions. The overall numbers of different values could give a rough indication of the numbers of values that might have to be stored in a table.

In the left half of Figure 10, we show the number different values produced by percentages of static instructions (an **s** prefix). In the right half, we determine the fractions of dynamic instructions (a **d** prefix) that correspond to each of the static categories. From the figure, we observe:

- A large number, $\geq$50%, of static instructions generate only one value.
- The majority of static instructions, $\geq$90%, generate fewer than 64 values.
- The majority, $\geq$50%, of dynamic instructions correspond to static instructions that generate fewer than 64 values.
- Over 90% of the dynamic instructions are due to static instructions that generate at most 4096 unique values.
- The number of values generated varies among instruction types. In general add/subtract and load instructions generate more values as compared with logic and shift operations.
- The more frequently an instruction executes the more values it generates.

The above suggest that a relatively small number of values would be required to predict correctly the majority of dynamic instructions using context based prediction – a positive result.

From looking at individual benchmark results (not shown) there appears to be a positive correlation between programs that are more difficult to predict and the programs that produce more values. For example, the highly predictable m88ksim has many more instructions that produce few values as compared with the less predictable gcc and go. This would appear to be an intuitive result, but there may be cases where it does not hold; for example if values are generated in a fashion that is predictable with computational predictors or if a small number of values occur in many different sequences.

4.4 Sensitivity Experiments for Context Based Prediction

In this section we discuss the results of experiments that illustrate the sensitivity of fcm predictors to input data and predictor order. For these experiments, we focus on the gcc benchmark and report average correct predictions among all instruction types.

Sensitivity to input data: We studied the effects of different input files and flags on correct prediction. The fcm predictor used in these experiments was order 2. The prediction accuracy and the number of predicted instructions for the different input files is shown in Table 6. The fraction of correct predictions shows only small variations across the different input files. We note that these results are for unbounded tables, so aliasing affects caused by different data set sizes will not appear. This may not be the case with fixed table sizes.

In Table 7 we show the predictability for gcc for the same input file, but with different compilation flags, again using an order 2 fcm predictor. The results again indicate that variations are very small.

Sensitivity to the order: experiments were performed for increasing order for the same input file (gcc.i) and flags. The results for the different orders are shown in Figure 11. The experiment suggests that higher order means better performance but returns are diminishing with increasing order. The above also indicate that few previous values are required to predict well.

5 Conclusions

We considered representatives from two classes of prediction models: (i) computational and (ii) context based. Simulations demonstrate that values are potentially highly predictable. Our results indicate that context based prediction outperforms previously proposed computational prediction (stride and last value) and that if high prediction correctness is desired context methods probably need to be used either alone or in a hybrid scheme. The obtained results also indicate that the performance of computational prediction varies between instruction types indicating that

File	Predictions (mil)	Correct (%)
jump.i	106	76.5
emit-rtl.i	114	76.0
gcc.i	137	77.1
recog.i	192	78.6
stmt.i	372	77.8

Table 6: Sensitivity of 126.gcc to Different Input Files

Flags	Predictions (mil)	Correct (%)
none	31	78.6
-O1	76	75.3
-O2	121	76.9
ref flags	137	77.1

Table 7: Sensitivity of 126.gcc to Input Flags with input file gcc.i

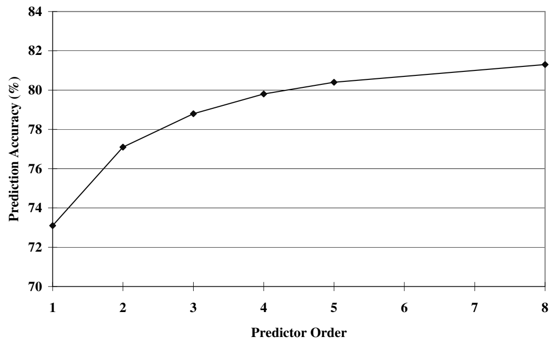

Figure 11: Sensitivity of 126.gcc to the Order with input file gcc.i

its performance can be further improved if the prediction function matches the functionality of the predicted instruction. Analysis of the improvements of context prediction over computational prediction suggest that about 20% of the instructions that generate relatively few values are responsible for the majority of the improvement. With respect to the value characteristics of instructions, we observe that the majority of instructions do not generate many unique values. The number of values generated by instructions varies among instructions types. This result suggests that different instruction types need to be studied separately due to the distinct predictability and value behavior.

We believe that value prediction has significant potential for performance improvement. However, a lot of innovative research is needed for value prediction to become an effective performance approach.

6 Acknowledgements

This work was supported in part by NSF Grants MIP-9505853 and MIP-9307830 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

The authors would like to thank Stamatis Vassiliadis for his helpful suggestions and constructive critique while this work was in progress.

References

[1] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and data speculation,” in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, October 1996.

[2] M. H. Lipasti and J. P. Shen, “Exceeding the dataflow limit via value prediction,” in *Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 226–237, December 1996.

[3] T. F. Chen and J. L. Baer, “Effective hardware-based data prefetching for high performance processors,” *IEEE Transactions on Computers*, vol. 44, pp. 609–623, May 1995.

[4] S. Mehrotra and L. Harrison, “Examination of a memory access classification scheme for pointer intensive and numeric programs,” in *Proceedings of the 10th International Conference on Supercomputing*, May 1996.

[5] D. Joseph and D. Grunwald, “Prefetching using markov predictors,” in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 252–263, June 1997.

[6] R. J. Eickemeyer and S. Vassiliadis, “A load instruction unit for pipelined processors,” *IBM Journal of Research and Development*, vol. 37, pp. 547–564, July 1993.

[7] J. Gonzalez and A. Gonzalez, “Speculative execution via address prediction and data prefetching,” in *Proceedings of the 11th International Conference on Supercomputing*, pp. 196–203, July 1997.

[8] A. Mendelson and F. Gabbay, “Speculative execution based on value prediction,” Tech. Rep. (Available from http://www-ee.technion.ac.il/fredg), Technion, 1997.

[9] J. E. Smith, “A study of branch prediction strategies,” in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[10] T.-Y. Yeh and Y. N. Patt, “Alternative implementations of two-level adaptive branch prediction,” in *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 124–134, May 1992.

[11] P.-Y. Chang, E. Hao, and Y. N. Patt, “Target prediction for indirect jumps,” in *Proceedings of the 24th International*

Symposium on Computer Architecture, pp. 274–283, June 1997.

[12] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 232–241, October 1994.

[13] R. Nair, "Dynamic path-based branch correlation," in *Proceedings of the 28th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 15–23, December 1995.

[14] S. Mahlke and B. Natarajan, "Compiler synthesized dynamic branch prediction," in *Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 153–164, December 1996.

[15] I.-C. K. Cheng, J. T. Coffey, and T. N. Mudge, "Analysis of branch prediction via data compression," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[16] A. Moshovos, S. E. Breach, T. J. Vijaykumar, and G. Sohi, "Dynamic speculation and synchronization of data dependences," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 181–193, June 1997.

[17] Y. Sazeides, S. Vassiliadis, and J. E. Smith, "The performance potential of data dependence speculation & collapsing," in *Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 238–247, December 1996.

[18] S. P. Harbison, "An architectural alternative to optimizing compilers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 57–65, March 1982.

[19] S. E. Richardson, "Caching function results: Faster arithmetic by avoiding unnecessary computation," Tech. Rep. SMLI TR-92-1, Sun Microsystems Laboratories, September 1992.

[20] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 194–205, June 1997.

[21] T. M. Austin and G. S. Sohi, "Zero-cycle loads: Microarchitecture support for reducing load latency," in *Proceedings of the 28th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 82–92, June 1995.

[22] D. Hammerstrom and E. Davidson, "Information content of cpu memory referencing behavior," in *Proceedings of the 4th International Symposium on Computer Architecture*, pp. 184–192, March 1977.

[23] S. McFarling, "Combining branch predictors," Tech. Rep. DEC WRL TN-36, Digital Western Research Laboratory, June 1993.

[24] M. Evers, P.-Y. Chang, and Y. N. Patt, "Using hybrid branch predictors to improve branch prediciton in the presence of context switches," in *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 3–11, May 1996.

[25] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Prentice-Hall Inc., New Jersey, 1990.

[26] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set," Tech. Rep. CS-TR-96-1308, University of Wisconsin-Madison, July 1996.