Efficient and Accurate Value Prediction Using Dynamic Classification

Bohuslav Rychlik, John W. Faistl, Bryon P. Krug,
Albert Y. Kurland, John J. Sung, Miroslav N. Velev, John P. Shen

Carnegie Mellon University
Microarchitecture Research Team (CMuART)
Department of Electrical and Computer Engineering
{bohuslav,shen}@ece.cmu.edu

In an effort to increase instruction level parallelism in superscalar microprocessors, several recent works have proposed using value prediction mechanisms to break the data dependency links between value-producing and value-consuming instructions. Some of these mechanisms achieve very high prediction accuracies, albeit at considerable hardware cost. Other mechanisms have low hardware cost but obtain mediocre prediction performance. In this paper, we propose combining three prediction mechanisms into a hybrid predictor. Each predictor has different hardware cost and prediction capability. The choice of a predictor for each instruction is guided by a dynamic classification mechanism. This mechanism partitions the instruction stream to maximize the prediction rate for each predictor. It also utilizes the predictors more efficiently by allocating an entry for each static instruction in at most one predictor. We achieve prediction rates of 72% and 43% with two possible realistic classification mechanisms for the SPECint95 benchmark set.

1. Introduction

If there is one clear trend in microprocessor development, it is the inevitably increasing density of transistors on a silicon die, informally known as Moore's Law. This trend allows chip designers to continually put more and more execution resources on a chip in an attempt to increase performance. These hardware resources can be utilized to support parallel instruction execution in the form of the superscalar model of processor design. The superscalar model attempts to increase execution performance by tapping instruction-level parallelism (ILP) and executing multiple instructions per cycle. However, as supported machine parallelism increases, actual instruction level parallelism is lagging behind. Specifically, IPC does not scale with available issue width because of two problems: insufficient instruction supply and insufficient instruction consumption. Instruction supply is limited by changing control flow directions and the nonconsecutive layout of the dynamic instruction stream in the instruction cache. The instruction supply problem can be dealt with through control flow speculation [McF93][YP91][Nai95], fetching from multiple cache lines [CM95][YM93] and, most recently, by using a trace cache [RB96][RJ97]. Once sufficient instruction supply is available, the instruction consumption problem must be addressed. The obstacle limiting instruction consumption is the serialization of instructions due to inter-instruction data dependencies. Because value consumers must execute serially with their value producers, the available machine parallelism is not exploited.

Recent studies have proposed prediction-based speculative techniques for overcoming these data flow dependencies [LWS96][LS96][WF97][SS97]. These techniques involve predicting the result values, thus breaking the serialization constraint between value producers and value consumers. However, to effectively break these constraints without incurring high misprediction penalties in real machines, value prediction techniques need to reach high accuracies. Some techniques such as context-based prediction can reach prediction accuracies as high as 80% [SS97], though at considerable hardware cost. Simpler predictors, such as last value and stride schemes have low hardware cost but also lower prediction accuracy. This paper continues the exploration of value prediction techniques by combining multiple predictor types into a hybrid predictor scheme [WF97]. The instruction stream is then partitioned through a classification mechanism that assigns each instructions to the lowest hardware cost predictor that can effectively predict its values. The goal is to leverage the performance of the low-cost predictor schemes to predict most of the static instructions while still achieving high prediction accuracy for the remaining instructions with small versions of the expensive prediction schemes.

The remainder of the paper is organized as follows: Section 2 discusses recent relevant work in value prediction. Section 3.1 introduces the three predictor types employed in the hybrid scheme and Section 3.2 describes the three classification schemes explored. Section 4 summarizes our experimental methodology, while Section 5 presents performance results for the various predictors and classification mechanisms. Section 6 discusses conclusions obtained from this work and Section 7 proposes future research.

2. Related Work

Value prediction is possible because of value locality [LWS96] - the property of recent values to recur in computer system storage locations. Value prediction can be viewed as an extension of branch prediction, where a single prediction bit is predicted based on its previous values, to multiple-bit register value prediction. The concept is based on the observation that programs and input data sets exhibit data redundancy - the presence of compile-time and dynamic constants, respectively. The results in [LWS96] show that a large fraction of values computed by the same static instruction are a repetition of a value within the 16 most recent values produced by the instruction. That work also proposes a Load Value Prediction Unit, assuming a hypothetical perfect mechanism for selecting the correct value to predict out of a history queue of recently produced values for each instruction.

A follow-up study [LS96] extends value prediction to all register writing instructions. The proposed Value Prediction Unit consists of a Value Prediction Table (VPT) and a Classification Table (CT), both PC-indexed. An entry in the VPT contains several previously seen values, whereas an entry in the CT contains a saturating counter, which is incremented on every successful prediction and decremented on every unsuccessful one. The purpose of the CT is to determine whether the prediction by the VPT is to be

considered. In that study, no method is introduced for selecting the correct value among the stored alternatives. Instead, the work focuses on the impact of an idealized value prediction on performance.

A study of realistic value prediction is presented in [WF97], where two predictors are analyzed separately and in combination. The first predictor, which is based on stride, targets values that differ by some constant amount. The second predictor, which is pattern based, uses a two-level scheme to determine a most likely value to be produced by an instruction, given a history pattern of recent values generated by that instruction and a set of its frequent values. The history pattern is used to index a pattern history table, which contains saturating counters reflecting the occurrences of each of the instruction's frequent values relative to a pattern, with the greatest-value counter indicating the value to be used for prediction. The combination of the stride and context based predictors has been shown to be more effective than any of them individually. This result calls for further investigation of hybrid prediction mechanisms.

A more theoretical analysis of value predictors appears in [SS97], where the characteristics of context based and computational predictors are analyzed. The results show that context based prediction is necessary for high accuracy and that stride and last value predictors are correct for only a minor fraction of the mispredictions made by a context based predictor. The conclusion made is that last value prediction is less accurate than stride prediction, which is less accurate than context based prediction.

3. Advanced Value Prediction and Classification

This section describes the predictor set and classification mechanisms used in our hybrid predictor. The predictor set determines the maximum predictability coverage of the instruction stream while the classification mechanism determines the degree of overlapping of predictions stored in the predictors.

3.1 Predictor Set

Our predictor set consists of three predictors, each with different hardware cost and complexity. The simplest predictor is the *Popular Last Value* predictor, intended to correctly predict a large number of instructions at low cost. The *Stride+* predictor was introduced to correctly and efficiently predict striding values, and the *Finite Context Method* (FCM) predictor predicts repeating sequences of values not predictable by the other two predictors.

3.1.1 Popular Last Value Predictor

The Popular Last Value (PLV) predictor is a space-efficient variation of the standard Last Value (LV) predictor. The LV predictor stores the last value generated by an instruction and predicts that the next instance of that instruction will produce the same value. It can accurately learn and predict sequences of the form: 1 1 1 1 1 1...

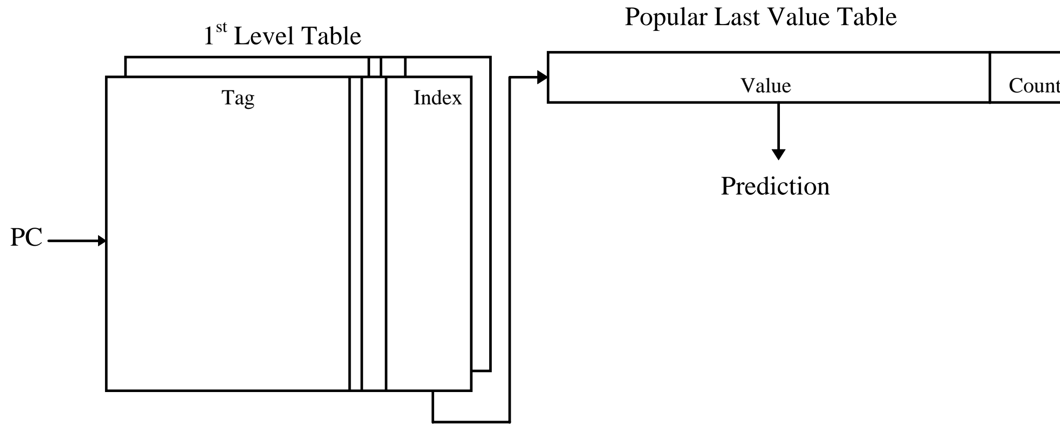


Figure 1

The design of the PLV is based on the observation that in a LV predictor, the number of unique values in the table at any given time is relatively small. The PLV was designed as a two-level table to take advantage of this observation (see Figure 1). The 2nd level table is the Popular Last Value Table (PLVT); it contains the most popular last values. The 1st level table is indexed by PC and contains indexes into the PLVT. The PLV predictor fields are described in Table 1.

1st Level Table	Bits	Description
Tag	X bits	This is the portion of PC not used as an index.
Most Recently Updated (MRU)	1 bit	Similar to a most recently used bit. Used during replacement.
Confidence	2-3 bits	Determines whether a prediction should be made. Also used during replacement.
PLVT-index	6-8 bits	This is the value used to index into the PLVT.

2nd Level Table – PLVT	Bits	Description
Value	32 bits	A popular last value. Several instructions may refer to this as their last value.
Count	5-7 bits	A saturating counter representing the popularity of Value. When the counter saturates, all Counts in the PLVT are halved. Used during replacement.

Table 1

There are three basic steps necessary to make a prediction.

1. PC indexes into the 1st level table and the tag is checked.
    1.1. If there is a match, then the confidence and the PLVT-index are read.
    1.2. Else, no prediction is made.
2. The confidence is checked to see if a prediction should be made:
    2.1. If a prediction should be made, the PLVT-index is used to index into the PLVT
    2.2. Else, no prediction is made.
3. The Value read from the PLVT is used as the prediction.

There are three basic steps necessary to update the PLV predictor.

1. The 1st level table and the PLVT are read in parallel:
    1.1. The 1st level table is checked to see if there is a tag match for the appropriate index.
    1.2. A content addressable lookup is done to see if the PLVT contains the correct value.
2. If there is a match in both tables, the index where the correct value was found in the PLVT is compared to the PLVT-index currently contained in the 1st level table.
    2.1. If they are the same, the confidence is increased (unless saturated).
    2.2. Else, confidence is decreased and if it falls below a certain threshold, the PLVT-index is updated.
3. If there is not a match in both tables, the tables are updated as appropriate:

3.1. The entry created in the 1st level table will replace the current entry with the least confidence that was not MRU.
3.2. The entry created in the PLVT will replace some entry with a Count less than a certain threshold.

For a size comparison between a LV predictor and a PLV predictor, consider a 4096-entry Last Value predictor and a PLV predictor with 4096 1st level table entries. We have observed that 128 entries in the PLVT are sufficient for the PLV predictor to achieve 98% of the accuracy of the LV predictor while taking up less than half the space (57% size reduction).

3.1.2 Stride+

Many instructions that do not produce constant values still behave in a predictable fashion. Many of these values can be predicted arithmetically. To predict values that are increasing or decreasing with regular intervals, or strides, a stride predictor was used. This predictor calculates the difference between consecutive values for a given instruction and then makes its prediction by adding the last seen value and this difference. The stride predictor can predict sequences of the form: 1 2 3 4 5 6...

We experimented with several variations of stride predictors for both performance and size before arriving at the Stride+ Predictor. The Stride+ Predictor that we used for these simulations was a modified version of the matched stride predictor [SS97][EV93]. The Stride+ predictor is basically a cache of value prediction information that is indexed by the instruction address. Each entry in the predictor contains the following information: the tag (based on instruction address), the last value produced by the instruction, the difference between the last two values produced by the instruction (i.e. the last stride), the value of the last stride to repeat between consecutive instances of the instruction (i.e. the matched stride), and a saturating counter. The last stride and last value are updated every time that the instruction is executed. However, the matched stride changes only when the difference between the current result of the instruction and the last value is equal to the last stride value stored in the entry. Storing both a last stride and a matched stride increases the number of bits per entry and, consequently, the size of the table. However, because the matched stride adds hysteresis to the table by not changing the stride and affecting the next prediction every time that an uncommon result occurs (e.g. the end of a loop), prediction accuracy is greatly increased. On average, the Stride+ predictor was 5% more accurate than a standard stride predictor that did not keep track of matched stride.

The Stride+ predictor is also much more efficient than previous implementations of stride predictors. Though previous stride value predictors used 32 bits for the last value and another 32 bits for the stride value(s), all of these bits are typically not necessary. In fact, analysis of the number of bits needed to represent the matched stride for correct predictions on SPECint95 benchmarks revealed that over 98% of these strides could be represented with only eight bits. The Stride+ predictor is able to use two 8-bit stride fields to achieve prediction accuracy approximately 5% better than the standard stride predictor with one 32-bit stride field. Thus, the Stride+ predictor is substantially more efficient than previous stride predictors.

3.1.3 FCM

While the Stride+ and Popular Last Value predictors are fairly simple and able to attain reasonable levels of accuracy, they are limited by the fact that they can only predict one type of pattern (i.e. global can catch patterns that are basically constant and stride can catch patterns that are changing by the exact same amount every time.) Many values do not follow these types of simple patterns. However, this does not mean they do not follow predictable patterns. For instance, strings of data will often tend to be found in regular patterns (e.g. if an instruction processed the char "t" followed by the string "h", it would not be uncommon for "e" to be the next value.) To predict these more complex patterns, Sazeides and Smith proposed a two-level finite context method (FCM) predictor [SS97]. The FCM predictor can accurately learn and predict repeating sequences of the form: 1 5 44 3 1 5 44 3 1 5 44 3...

To this point, the research involving two-level FCM predictors has involved either infinite-sized predictors or extremely large predictors. The goal of this research was to use the FCM predictor to improve the overall prediction accuracy in as efficient of a manner as possible.

Finite context method predictors are composed of two levels of tables. The first level is typically indexed by the instruction address. Each entry in the first-level table contains a lookup tag and the last n values produced by the instruction, with n being the depth of the history of values for the instruction. (This history depth typically varies between 2 and 4. [SS97] suggests that the optimal value for the history depth is three in terms of prediction accuracy.) Assuming that the history depth for each entry is three, each entry would need 96 bits plus tag bits. The second-level table is indexed by some hashing of the history values and the instruction address. It contains only a tag and the value that occurred the last time the hash value was produced.

Our FCM predictor has a similar basic structure with various modifications to improve the efficiency of the predictor so that a realistically-sized predictor could be implemented. Specifically, since only certain bits from the value history for each instruction are used in the hash function, it is unnecessary to store all 32 bits for each of the previous results. Only the necessary bits should be stored. Our hash function uses the bottom eight bits of each of the three previous values. Thus the number of bits for history values has been reduced from 96 bits to 24 bits – a decrease of approximately 75% with only a minor reduction in prediction accuracy. In addition, involving the instruction address in the hash function to the second-level table reduces the FCM's efficiency. Using the instruction address limits the amount of sharing that can occur between different instructions, and it makes a larger sized table necessary. Also, in some cases, this may limit the predictor's accuracy. For instance in the above example of the string "the", the string could be used in various functions throughout the program, and it might be helpful for instructions to "learn" from other instructions.

3.1.4 Eliminated Predictors

In this section, we briefly describe some of the other value predictors that we implemented but eliminated. These predictors were either covered by other predictors or did not perform well. These predictors were: Always Predict Zero, Last Value, Arithmetic Stride Variations, Shift Stride, and 2-Level History Predictors.

The Always Predict Zero predictor achieves approximately 17% accuracy, which is remarkably high considering its simplicity. However, the Popular Last Value predictor covers Always Predict Zero very well, so there is no added benefit in having this predictor. Since the PLV also covers the Last Value predictor, Last Value Predictor was also eliminated.

In order to accurately predict repeating arithmetic sequences without using the FCM, we experimented with additional variations of the Stride predictor. To learn and predict without mispredictions sequences of the form 1 2 3 4 1 2 3 4 1 2 3 4..., we added two additional fields to the Stride entry: the first and last values of a repeating sequence. When the previously produced value by a striding instruction was equal to the last value of a repeating sequence, the modified Stride predictor would predict the first value of the sequence instead of the sum of the previous value and the stride. This addition resulted in minimal performance gain and additional hardware cost so it was discarded.

Finally, we experimented with a Shift Stride predictor that would maintain the last value and a stride to shift the value by. Such a predictor could predict sequences of the form: 1 2 4 8 16 32... However, the Shift Stride predictor added negligible additional prediction accuracy and was likewise eliminated.

3.2 Classification

Given a predictor set, the goal of a classification scheme is to choose which predictor (if any) to use to make a prediction (the *predict* operation), and which predictor(s) to update with the correct value (the *update* operation). Unless multiple path speculative execution is allowed, at most one predictor may be chosen to make a prediction. However, multiple predictors can be updated with the correct value. Multiple updates allow more predictors to potentially make a prediction (overlap) but also make inefficient use of the predictor resources. Therefore, while multiple update schemes offer higher potential accuracy with unlimited predictor sizes, they may also suffer a greater performance drop as predictor size is decreased. We implemented three dynamic classification schemes to explore these tradeoffs.

3.2.1 Ideal Dynamic Scheme

The Ideal dynamic classification scheme updates all the predictors with correct values from all completing instructions for the *update* operation. For the *predict* operation, the Ideal scheme perfectly selects the correct prediction from the set of predictors that have sufficient confidence to make a prediction. This means that if *any* predictor is correct and confident, a correct prediction is made. The Ideal scheme can also mispredict, but only when all the confident predictors mispredict. Otherwise, the Ideal scheme does not predict. This scheme results in high prediction accuracy with large predictor tables, but suffers from significant overlap.

3.2.2 Simple Dynamic Scheme

The Simple dynamic classification scheme does not reduce the overlap in the predictors. Similar to the Ideal dynamic scheme, for the *update* operation all predictors are updated with completing instructions. However, unlike the Ideal scheme, the Simple scheme does not have perfect knowledge about which predictor produces the correct prediction. Instead, for the *predict* operation a predictor is chosen according to a simple rule. The predictors are accessed in the order 1) Popular Last Value, 2) Stride+, and 3) FCM, and the first predictor with sufficient confidence to predict is used. A misprediction occurs when the chosen predictor mispredicts. If none of the predictors have sufficient confidence to predict, no prediction is made.

3.2.3 Efficient Dynamic Scheme (Zero Overlap)

The goal of the Efficient dynamic classification scheme was to maintain high prediction accuracy while minimizing the overlap in the predictors. In the classification scheme, each static instruction is assigned to at most one predictor in the predictor set.

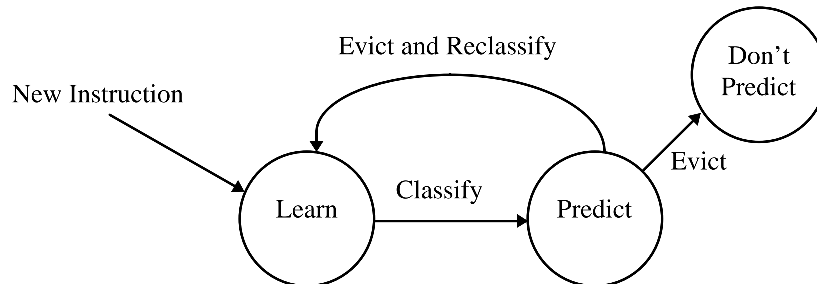


Figure 2

The operation of the classifier is conceptually described by Figure 2. When a new static instruction is first encountered, instead of immediately being assigned to a predictor and potentially replacing useful entries, it first goes through a short learning period. The learning stage allows the classifier to choose which predictor is best suited for that instruction. Once an instruction has been classified to a predictor, it allocates an entry in the predictor's table. The predictor then makes value predictions for the instruction and is updated with actual values, keeping track of prediction confidence internally. If prediction confidence drops sufficiently, the instruction is no longer deemed predictable by the predictor and is evicted from the predictor, releasing its allocated entry. It can then either be reclassified to another predictor by repeating the learning process or it can be classified as unpredictable by any predictor. Because the FCM predictor implements the most general form of value prediction, instructions evicted from the FCM are classified as unpredictable. Instructions evicted from the other predictors repeat the classification process.

The conceptual description of the classifier can be directly mapped to a hardware implementation using two tables, a Classified Instruction Table (CT) and an Unclassified Instruction Table (UT). The CT (Figure 3) is tagged by instruction address and stores a Predictor ID field for each classified instruction. The Predictor

ID field is one of: *Don't Predict*, *Use Popular Last Value*, *Use Stride+*, and *Use FCM*. This field is used to determine which predictor to use to make a prediction for an instruction and which predictor to update. The UT (Figure 4) is also tagged by instruction address but stores the *h*-value history for the instruction. For this paper, we used a history depth of 3 (h=3). The value history is filled during the learning period and subsequently used by the classifier mechanism to choose a predictor for each instruction.

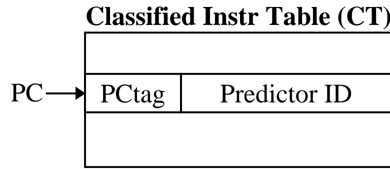


Figure 3

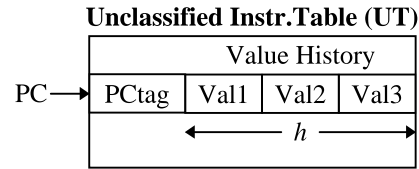


Figure 4

Both tables are 4-way associative. Use of these tables can be divided into four operations: *Predict*, *Update*, *Classify*, and *Evict*. These operations can be described as follows:

Predict: When an instruction is fetched by the machine, the CT is accessed by the PC to obtain a Predictor ID. If the field is set to *Don't Predict* or if there is a CT miss, no prediction is made. Otherwise, the field is used to select which predictor should be accessed for a prediction. The predictor returns a predicted value and a prediction confidence. If the prediction confidence is high, the predicted value is used. Otherwise, no prediction is made. Additionally, if the prediction confidence is zero, the *Evict* operation is triggered.

Update: When an instruction completes, the CT is again accessed by the PC to determine the Predictor ID for that instruction (in a real implementation this second access could be avoided by carrying the Predictor ID field along with the instruction). The field is used to select which predictor should be updated with the produced value. If there is a CT miss, then the instruction is unclassified and the UT is accessed with the instruction address. A UT entry is either created (in the case of a UT miss) or updated (in the case of a UT hit) with the produced value. Additionally, if all the value history entries for the instruction are filled, the *Classify* operation is triggered.

Classify: The classify operation examines the value history for an instruction in the UT. It computes the differences (deltas) between each pair of values and determines a Predictor ID for the instruction. If all the deltas are zero, the Predictor ID is set to *Popular Last Value*. Otherwise, if the deltas are equal, the Predictor ID is set to *Stride+*. Otherwise the Predictor ID is set to *FCM*. Finally, an entry for the instruction is written into the CT with the Predictor ID and the instructions UT entry is deallocated.

Evict: When a predictor returns a confidence of zero for a prediction, it means that it can no longer accurately predict the instruction and it is removed from the predictor. If the Predictor ID for the instruction is FCM, the instruction is deemed unpredictable by any predictor and the Predictor ID in the CT entry is set to *Don't Predict*. Otherwise, the CT entry for the instruction is cleared. Consequently, the next time the instruction enters the machine, it will be reclassified like a new instruction.

While the CT needs to be large to accommodate the current instruction working set of the program, the UT can have fewer entries because of the relatively small number of new instructions entering the machine each cycle. Additionally, the *Classify* and *Evict* operations can in theory be completely decoupled from the *Predict* and *Update* operations and do not need to support the same instruction bandwidth. While an 8-IPC machine could potentially need to do 8 *Predicts* and 8 *Updates* per cycle, it might perform well with only 1 or 2 *Classify* and *Evict* operations available per cycle because they are out of the critical path of the machine. *Classify* and *Evict* requests could be queued up and serviced later.

4. Experimental Methodology

We use the SPECint95 integer benchmark suite, cross-compiled on x86 Linux platforms using GNU's gcc 2.7.2 to produce PowerPC binaries. These are then executed on the PowerPC functional simulator PSIM [Cag96] and the predictor and classifier mechanisms are simulated simultaneously. To reduce simulation times, we use a subset of the SPECint95 reference input data sets shown in Table 2. All result averages are weighted by number of register-writing instructions.

Benchmark	Input Set	Register-writing Instructions
compress	10000 e 2231	22,504,999
gcc	-O regclass.i -s regclass.s	150,133,834
go	5 9	51,285,296
ijpeg	tinyrose.ppm	50,470,570
li	queen6.lsp	27,917,704
m88ksim	dhry.big.100iter, cacheoff	66,796,573
perl	trainscrabbl.in	26,594,210
vortex	tiny.in	74,586,730
	Total:	470,289,916

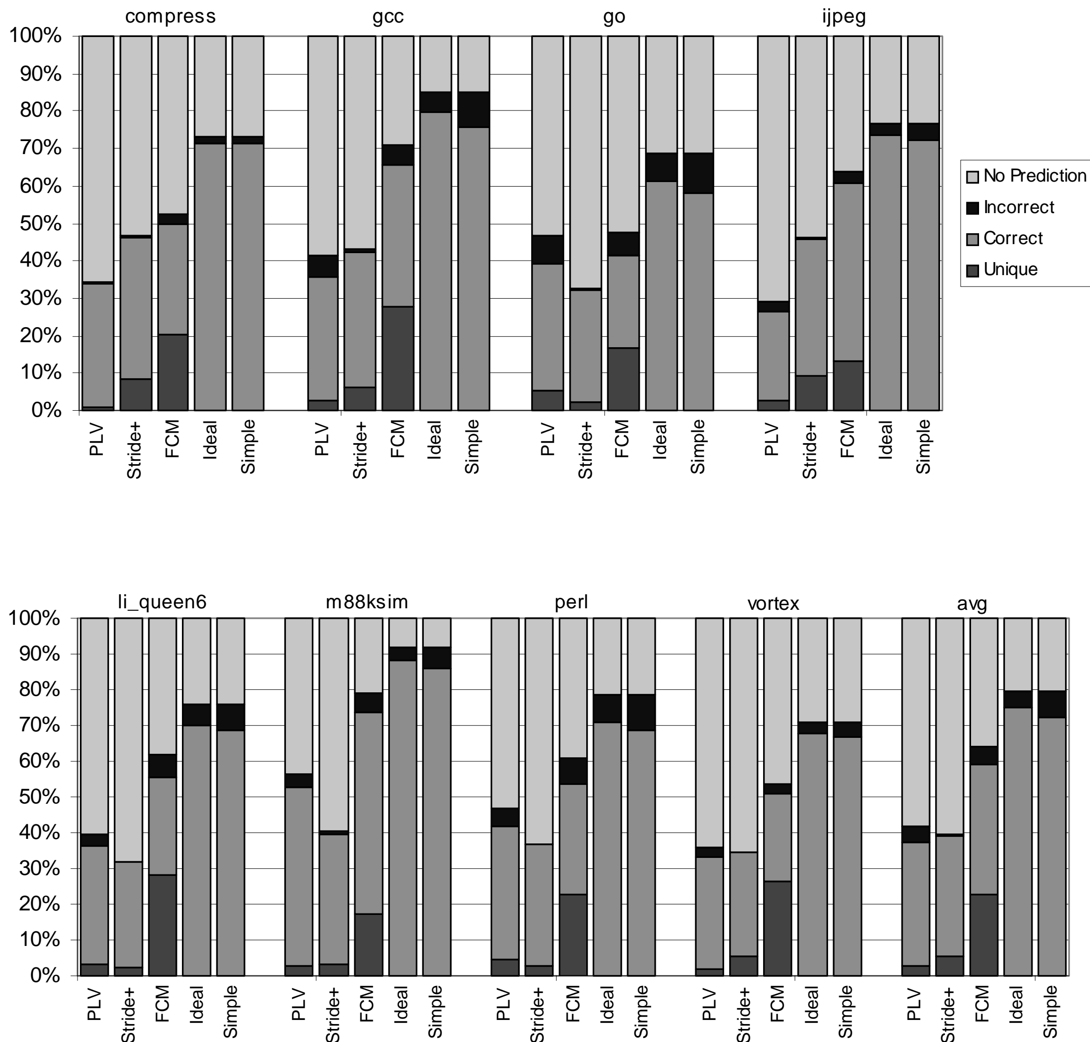
Table 2

5. Experimental Results

5.1 Performance with Large Predictor Tables

Initially simulations were run to determine the effectiveness of the chosen predictor set. All predictors were setup large enough to try and eliminate replacement of useful values in the table. All the predictors were updated with every instruction. The results of these initial runs are show in Figure 5. Each predictor's results are broken into the four categories listed. Of special interest is the Unique category. This category corresponds to the percentage of the time a predictor was the only predictor to make a correct prediction. For all the benchmarks, the PLV had a very low Unique percentage, as is expected. Stride+'s Unique percentage was highest in compress, gcc, and ijpeg. The FCM predictor was the only predictor that consistently had a significant portion of its correct predictions be unique, about 22% on average.

Figure 5 - Large Predictors with Overlap



The other two bar graphs show prediction performance results when these predictors are used using the Ideal Dynamic and Simple Dynamic classification schemes described in Sections 3.2.1 and 3.2.2, respectively. The Ideal classifier always chooses predicts correctly whenever one of the predictors predicts correctly, giving it a correct prediction rate of 75%. It only predicts incorrectly when all the individual predictors mispredict. This is a theoretical upper bound on prediction accuracy with the given predictor set. The Simple classifier chooses a prediction from the first predictor that is confident, looking at PLV first, then Stride+, then FCM. With 72% correct predictions, the Simple classifier is very close to Ideal.

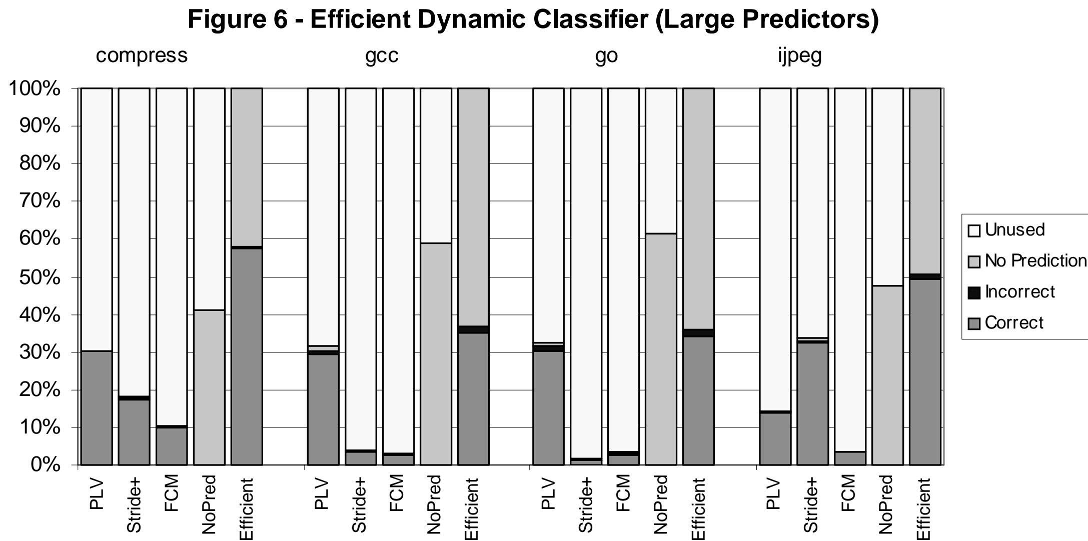
The Ideal and Simple dynamic classifiers are inefficient because of the overlap between predictors - all the predictors are always updated. As Figure 5 illustrated, most correct predictions by a predictor were not unique. An Efficient classifier was developed (described in Section 3.2.3) to try and eliminate overlap by

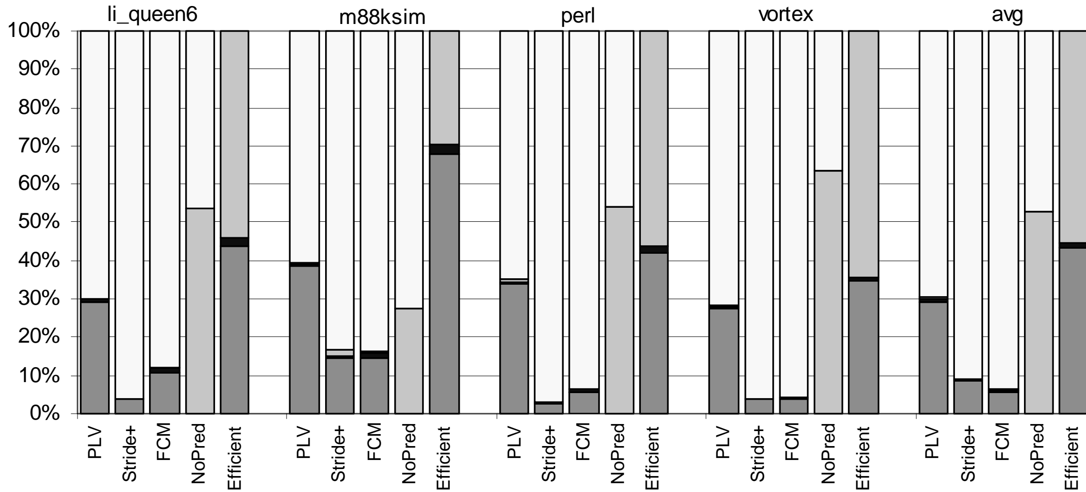
classifying an instruction to at most one predictor. The instruction only updates that predictor. The results for this classification scheme are in Figure 6.

The Efficient Dynamic classification scheme achieves an overall prediction rate of 43%, which is much smaller than the 75% and 72% prediction accuracies achieved by the Ideal and Simple Dynamic classification schemes. However, its overall misprediction rate is only 2%, which is better than the 5% and 7% misprediction rates of the Ideal and Simple schemes, respectively. It also never updates more than one predictor with the correct value and therefore uses the predictor tables much more efficiently. This efficiency benefit is not reflected in performance for the very large predictor sizes. The tradeoff between prediction accuracy and efficiency is explored further in Section 5.3.

The first three bars indicate the prediction, misprediction, and no prediction rates for the individual predictors. Note that one of the predictors is *Don't Predict* (NoPred). Also, note that unlike in Figure 5, the prediction rates of all the individual predictors add up to the prediction rate of the Efficient Dynamic classification scheme, because there is no prediction overlap. *All* of the predictions are unique.

Figure 6 - Efficient Dynamic Classifier (Large Predictors)





5.2 Analysis of the Efficient Dynamic Classifier

5.2.1 Performance Impact of Classification Table Size

Table 3 shows the correct prediction rate and the misprediction rate as a function of the CT and UT sizes averaged over three representative benchmarks (*compress*, *go*, and *m88ksim*). As can be seen, increasing the CT and UT sizes beyond 256 lines does not lead to a significant improvement in the performance of the hybrid predictor. (Many of the conflicts in these tables are avoided by using 4 way set associative lines.) Hence, it is possible to implement efficiently a dynamic classification scheme which has a high correct prediction rate in addition to a low misprediction rate.

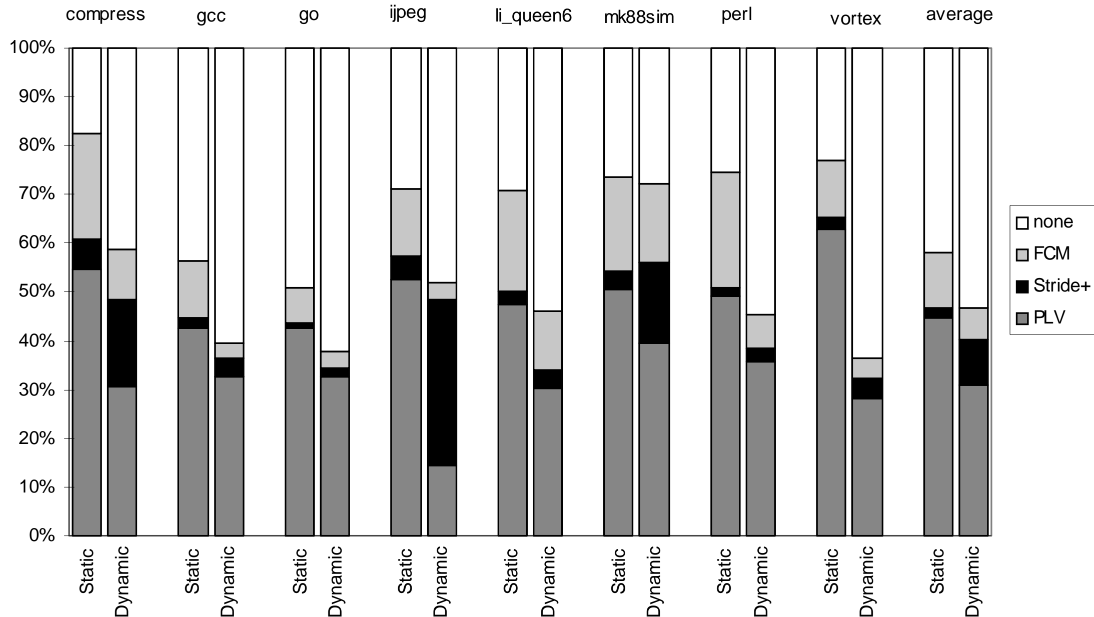
UT Size	Prediction Rate, %			Misprediction Rate, %		
	CT Size			CT Size		
	256	***512***	***1024***	***256***	***512***	***1024***
256	49.81	51.53	52.08	1.91	1.78	1.93
512	50.30	51.83	52.35	1.91	1.87	1.89
1024	49.23	51.66	52.67	1.91	1.86	1.95

Table 3

5.2.2 Partitioning of the Instruction Stream

In this section, we analyze how the Efficient Dynamic Classifier partitions the instruction stream. In Figure 7, the left bar graph shows the percentage of static instructions that are classified to each predictor (including *Don't Predict*) for each benchmark. The right bar shows how this partitioning maps to the dynamic instruction stream.

Figure 7 - Instruction Stream Partitioning



On average, 45% of the static instructions are classified to use the PLV predictor, while only 31% of the dynamic instructions are classified as *Use PLV*. One possibility for this reduction is that the PLV covers fewer instructions that are in loops. Except for *ijpeg*, more than half of the predicted static and dynamic instructions are assigned to the Popular Last Value predictor.

On average, only 2.1% of the static instructions are assigned to the Stride+ predictor, but these instructions account for 9.3% of the predictions made. This gives it a better cost/performance ratio than the other predictors. In particular, for the *ijpeg* benchmark, the set of static instructions classified to use the Stride+ predictor maps to a very significant percentage of the predictions made.

The FCM, the most complex predictor, is assigned to 11.5% and 6.4% of the static and dynamic instructions, respectively. This means that it offers a lower cost/performance benefit than the other predictors. However, as shown in section 5.1, it can predict instructions that the other predictors can't and therefore is necessary to achieve high prediction accuracy.

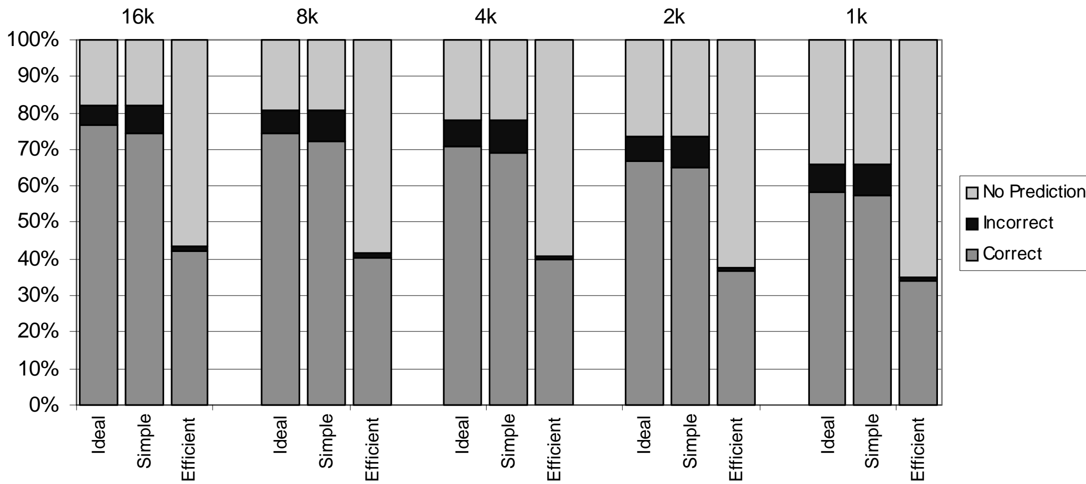
The percentage of static instructions that were classified as *Don't Predict* is 41.8%. These instructions account for 53.4% predictions *not* made. Reclassifying these instructions to one of the predictors could improve overall prediction rate, suggesting that a mechanism for moving instructions from *Don't Predict* back to one of the predictors would be useful. This indicates that the classifier should tolerate lower prediction confidence in a predictor before evicting an instruction or that instructions classified as *Don't Predict* should periodically be given the opportunity to be reclassified to one of the other predictors.

These results show that the Popular Last Value Predictor is assigned the highest percentage of static instructions. This suggests that the PLV Predictor should be the largest, followed by the FCM predictor. The Stride+ table can be relatively small. We used this information to select a ratio of relative predictor sizes. This ratio is 8:1:4 for PLV:Stride+:FCM. In the next section, we used this ratio to keep the relative predictor sizes constant while varying the total predictor set size.

5.3 Performance Impact of Predictor Table Size

In Section 5.1 we explored the performance of the three classification schemes while using very large predictor sizes. In this section, we explore the impact of reducing the predictor set size on the overall prediction performance of the three classification schemes. In Figure 11, the five groups of bar graphs correspond to the five total predictor set sizes we evaluated. The label is the number of entries in the 1st level table of the PLV predictor. The number of entries in the first level tables of the other predictors can be derived from the 8:1:4 ratio. Therefore, the predictor size label of "8k" indicates 8192 1st level table PLV entries, 1024 Stride+ entries, and 4096 1st level table FCM entries. For the two-level predictors (PLV and FCM), we maintained the number of 2nd level table entries constant (128 for PLV and 2^16 for the FCM). Since the focus of this investigation was on the classification mechanisms, we left the exploration of varying 2nd level table for future investigation.

Figure 8 - Classifier Accuracy vs. Predictor Size



As the total predictor set size is decreased, the Ideal and Simple mechanisms' prediction rates to decrease at a rate faster than for the Efficient scheme. This result is consistent with the observation that the Efficient scheme makes more efficient use of the predictors. However, for the predictor set sizes we explored, the prediction rates of the Ideal and Simple schemes never reached that of the Efficient scheme. On the other hand, the Efficient scheme does far better at avoiding mispredictions independent of the predictor set size, unlike the Ideal and Simple mechanisms whose misprediction rates increase to 7% and 9%, respectively.

6. Conclusion

In order to explore the prediction performance potential of a hybrid predictor scheme, we implemented three different predictor mechanisms and three different classification schemes. We chose three prediction mechanisms with different prediction capabilities to obtain good coverage of the predictable instruction stream. We paid particular attention to reducing the size of the predictor entries. We then combined these prediction mechanisms into a hybrid predictor with three different dynamic classification mechanisms: Ideal, Simple, and Efficient. The Ideal and Simple classification mechanisms allowed us to explore the performance potential of the hybrid predictor. Using the Simple classification mechanism, we obtained a prediction accuracy of 72% and a misprediction rate of 7%. We also proposed the Efficient classification mechanism in an attempt to reduce the inefficient use of the predictor sets by the other schemes. Unfortunately, while the Efficient mechanism eliminates the overlap in the predictor sets and achieves a misprediction rate of only 1-2%, its prediction rate is only 43%. One cause of this low prediction rate is that approximately 42% of the static instructions are classified as unpredictable, even though the other classification schemes can correctly predict some of these instructions. This suggests that the Efficient mechanism may be too pessimistic in assessing a predictor's performance. Allowing more mispredictions by a predictor for an instruction before evicting that instruction could potentially increase prediction rate at some cost to misprediction rate. Additionally, providing a path for instructions classified as unpredictable to be re-classified could also raise the prediction rate. Finally, it may be that some overlapping updates of multiple predictors may be beneficial if instructions often change predictability behavior.

7. Future Work

We plan to experiment with different replacement policies for the individual predictors, particularly with replacement policies based on the confidence of the entries within a given associative set, possibly combined with the LRU information of these entries. We will also study the effect of small fully associative victim caches for each table within the classifier and the individual predictors on the overall prediction accuracy. Furthermore, we plan to investigate other dynamic classifiers and to rank them based on a cost/performance analysis. We will engineer these classifiers in order to minimize their impact on the cycle time, e.g., by providing a way to store predictor classification information for instructions in the trace cache. Most importantly, we plan to incorporate the hybrid predictor into a machine model and measure its influence on the IPC. Finally, additional efficiency and performance gains are possible by only predicting instructions that will benefit from value prediction due to their position in the data flow graph, i.e., by only predicting instructions whose operands are not ready.

References

[Cag96] A. Cagney. PSIM User's Guide. Available as ftp://cambridge.cygnus.com/pub/psim/index.html, August 1996.

[CM95] T. Conte, K. Menezes, P. Mills and B. Patel, “Optimization of Instruction Fetch Mechanisms for High Issue Rates.” In Proceedings of the 22nd International Symposium on Computer Architecture, pp. 333-343, June 1995.

[DS94] K. Diefendorf, and E. Silha, “The PowerPC User Instruction Set Architecture.” In IEEE Micro, pp. 30-41, 1994

[EV93] 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.

[LS96] M.H. Lipasti and J.P. Shen, "Exceeding the Dataflow Limit via Value Prediction," Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29), pp. 226-237, December 1996.

[LWS96] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value locality and load value prediction," Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS- VII), October 1996.

[McF93] S. McFarling, “Combining Branch Predictors.” Technical Report TN-36, Digital Equipment Corp., June 1993.

[Nai95] R. Nair, “Dynamic Path-based Branch Correlation,” Proceedings of the 28th International Symposium on Microarchitecture, December 1995.

[PS92] S-T. Pan, K. So, and J. Rahmeh, “Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation.” In Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems, pp. 76-84, October 1992.

[RB96] E. Rotenberg, S. Bennett and J. E. Smith, “Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching,” In Proceedings of the 29th International Symposium on Microarchitecture, pp. 24-34, December 1996.

[RJ97] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, “Trace Processors,” Proceedings of the 30th International Symposium on Microarchitecture, December 1997.

[SS97] Y. Sazeides, J.E. Smith, "The Predictability of Data Values," Proceedings of 30th International Symposium on Microarchitecture (MICRO-30), December 1997.

[WF97] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," Proceedings of 30th International Symposium on Microarchitecture (MICRO-30), December 1997.

[YM93] T-Y. Yeh, D. Marr, and Y. Patt, “Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache.” In Proceedings of the 7th ACM International Conference on Supercomputing, pp. 67-76, July 1993.