A STUDY OF BRANCH PREDICTION STRATEGIES

JAMES E. SMITH

Control Data Corporation
Arden Hills, Minnesota

ABSTRACT

In high-performance computer systems, performance losses due to conditional branch instructions can be minimized by predicting a branch outcome and fetching, decoding, and/or issuing subsequent instructions before the actual outcome is known. This paper discusses branch prediction strategies with the goal of maximizing prediction accuracy. First, currently used techniques are discussed and analyzed using instruction trace data. Then, new techniques are proposed and are shown to provide greater accuracy and more flexibility at low cost.

INTRODUCTION

It is well known[1-3,10] that in a highly parallel computer system, branch instructions can break the smooth flow of instruction fetching and execution. This results in delay, because a branch that is taken changes the location of instruction fetches and because the issuing of instructions must often wait until conditional branch decisions are made.

To reduce delay, one can attempt to predict the direction that a branch instruction will take and begin fetching, decoding, or even issuing instructions before the branch decision is made. Unfortunately, a wrong prediction may lead to more delay if, for example, instructions on the correct branch path need to be fetched or partially executed instructions on the wrong path need to be purged. The disparity between the delay for a correctly predicted branch and an incorrectly predicted branch points to the need for accurate branch prediction strategies.

This paper discusses branch prediction strategies with the goal of maximizing the likelihood of correctly predicting the outcome of a branch. First, previously suggested branch prediction techniques are discussed. Owing to the large number of variations and configurations, only a few representative strategies have been singled out for detailed study, although several are mentioned. Then, new techniques are proposed that provide more accuracy, less cost, and more flexibility than methods used currently.

Because of the wide variation in branching behavior between different applications, different programming languages, and even individual programs, there is no good analytic model for studying branch prediction. For this reason, we used instruction trace data to measure experimentally the accuracy of branch prediction strategies.

Originally, ten FORTRAN programs, primarily scientific, were chosen. It was found, however, that several were heavily dominated by inner loops, which made them very predictable by every strategy considered. Two programs, SCI2 and ADVAN, were chosen from this inner-loop-dominated class. Four other programs were not as heavily dominated by inner loops and were less predictable. All were chosen for the study.

The six FORTRAN programs used in this study were:

1. ADVAN: Calculates the solution of three simultaneous partial differential equations
2. SCI2: Performs matrix inversion
3. SINCOS: Converts a series of points from polar to Cartesian coordinates
4. SORTST: Sorts a list of 10,000 integers using the shell sort algorithm[9]
5. GIBSON: An artificial program that compiles to instructions that roughly satisfy the so called GIBSON mix[5]
6. TBLLNK: Processes a linked list and contains a variety of conditional branches

The programs were compiled for a CDC CYBER 170 architecture.

Note that other than ADVAN and SCI2, the test programs were chosen for their unpredictability, and that a more typical scientific mix would contain more programs like ADVAN and SCI2.

0149-7111/81/0000/0135$00.75 © 1981 IEEE

Because of the method used for evaluating prediction strategies, any conclusions regarding their relative performance must be considered in light of the application area and the language used here. Nevertheless, the basic concepts and the strategies are of broader interest, because it is relatively straightforward to generate instruction traces and to measure prediction accuracy for other applications or languages.

Results published previously in this area appear in Shustek[6], who used instruction trace data to evaluate strategies for the IBM System 360/370 architecture. Ibbett[7] described the instruction pipeline of the MU5 computer and gave experimental results for a particular branch prediction strategy. A rather sophisticated branch predictor has been described for the S1 processor,[8] but as of this writing, no information appears to have been published regarding its accuracy. Branch prediction strategies have also been used in other high performance processors, but, again, experimental results have not been published.

Our study begins in the next section, with two branch prediction strategies that are often suggested. These strategies indicate the success that can reasonably be expected. They also introduce concepts and terminology used in this paper. Strategies are divided into two basic categories, depending on whether or not past history was used for making a prediction. In subsequent sections, strategies belonging to each of the categories are discussed, and further refinements intended to reduce cost and increase accuracy are presented. Levels of confidence are attached to branch predictions to minimize delay when there are varying degrees to which branch outcomes can be anticipated (for example, prefetching instructions is one degree, preissuing them is another). Conclusions are given in the final section.

TWO PRELIMINARY PREDICTION STRATEGIES

Branch instructions test a condition specified by the instruction. If the condition is true, the branch is *taken*: instruction execution begins at the target address specified by the instruction. If the condition is false, the branch is *not taken*, and instruction execution continues with the instruction sequentially following the branch instruction. An unconditional branch has a condition that is always true (the usual case) or is always false (effectively, a pass). Because unconditional branches typically are special cases of conditional branches and use the same operation codes, we did not distinguish them when gathering statistics, and hence, unconditional branches were included.

A straightforward method for branch prediction is to predict that branches are either always taken or always not taken. Because most unconditional branches are always taken, and loops are terminated with branches that are taken to the top of the loop, predicting that all branches are taken results typically in a success rate of over 50%.

Strategy 1

- Predict that all branches will be taken.

Figure 1 summarizes the results of using strategy 1 on the six FORTRAN benchmarks.

From Figure 1, it is evident that the majority of branches are taken, although the success rates vary widely from program to program. This points to one factor that must be considered when evaluating prediction strategies: *program sensitivity*. The algorithm being programmed, as well as the programmer and the compiler, can influence the structure of the program and, consequently, the percentage of branches that are taken. High program sensitivity can lead to widely different prediction accuracies. This, in turn, can result in significant differences in program performance that may be difficult for the programmer of a high-level language to anticipate.

Strategy 1 always makes the same prediction every time a branch instruction is encountered. Because of this, strategy 1 is called *static*. It has been observed and documented,[6] however, that the likelihood of a conditional branch instruction at a particular location being taken is highly dependent on the way the same branch was decided previously. This leads to *dynamic* prediction strategies in which the prediction varies, based on branch history.

Strategy 2

- Predict that a branch will be decided the same way as it was on its last execution. If it has not been previously executed, predict that it will be taken.

The results (Figure 2) of using strategy 2, indicate that strategy 2 generally provides better accuracy than strategy 1. Unfortunately, strategy 2 is not physically realizable, because theoretically, there is no bound on

the number on individual branch instructions that a program may contain. (In practice, however, it may be possible to record the history of a limited number of past branches; such strategies are discussed in a subsequent section.)

Strategies 1 and 2 provide standards for judging other branch prediction strategies. Strategy 1 is simple and inexpensive to implement, and any strategy that is seriously being considered for use should perform at least at the same level as strategy 1. Strategy 2 is widely recognized as being accurate, and if a feasible strategy comes close to (or exceeds) the accuracy of strategy 2, the strategy is about as good as can reasonably be expected.

Strategy 1 is apparently more program sensitive than strategy 2. Evidence of this is the wide variation in accuracy for strategy 1 and the much narrower variation for strategy 2 (Figures 1 and 2). Strategy 2 has a kind of second-order program sensitivity, however, in that a branch that has not previously been executed is predicted to be taken. Lower program sensitivity for dynamic prediction strategies is typical, as results throughout this paper show.

Finally, it is interesting that one aspect of branch behavior leads occasionally to better accuracy with strategy 1 than strategy 2. Often, a particular branch instruction is predominately decided one way (for example, a conditional branch that terminates a loop is most often taken). Sometimes, however, it is decided the other way (when "falling out of the loop"). These *anomalous decisions*, are treated differently by strategies 1 and 2. Strategy 1, if it is being used on a branch that is most often taken, leads to one incorrect prediction for each anomalous not taken decision. Strategy 2 leads to two incorrect predictions; one for the anomalous decision and one for the subsequent branch decision. The handling of anomalous decisions explains those instances in which strategy 1 outperforms strategy 2 and indicates that there may exist some strategies that consistently exceed the success rate of strategy 2.

STATIC PREDICTION STRATEGIES

Strategy 1 (always predict that a branch is taken) and its converse (always predict that a branch is not taken) are two examples of static prediction strategies. A further refinement of strategy 1 is to make a prediction based on the type of branch, determined, for example, by examining the operation code. This is the strategy used in some of the IBM System 360/370 models[9] and attempts to exploit program sensitivities by observing, for example, that certain branch types are used to terminate loops, while others are used in IF-THEN-ELSE-type constructs.

Strategy 1a

- Predict that all branches with certain operation codes will be taken; predict that the others will not be taken.

The six CYBER 170 FORTRAN programs were examined, and it was found that "branch if negative", "branch if equal", and "branch if greater than or equal" are usually taken, so they are always predicted to be taken. Other operation codes are always predicted to be not taken. This strategy is somewhat tuned to the six benchmarks, because only the benchmarks were analyzed to determine which opcodes should be predicted to be taken. For this reason, the results for strategy 1a may be slightly optimistic.

Figure 3 shows the results for strategy 1a when it was applied to the CY170 programs. Generally, greater accuracy was achieved with strategy 1a than with strategy 1. The largest increase was in the GIBSON program in which the prediction accuracy was improved from 65.4% to 98.5%. The only program showing a decrease in accuracy was the SINCOS program in which there was a drop from 80.2% to 65.7%. The changes in both the GIBSON and SINCOS programs can be attributed to predicting that "branch if plus" was not taken. If it had been predicted as taken, the accuracy of the GIBSON program would have dropped nearly to its original value, and the accuracy of the SINCOS program would have risen nearly to its original value.

Other static strategies are possible. For example, predictions based on the direction of the potential branch or on the distance to the branch target can be made. Following is a detailed description of one of these strategies.

Strategy 3

- Predict that all backward branches (toward lower addresses) will be taken; predict that all forward branches will not be taken.

The thought behind strategy 3 is that loops are terminated with backward branches, and if all loop

branches are correctly predicted, the overall accuracy will be high.

Figure 4 indicates that strategy 3 often worked well, sometimes exceeding strategy 2 (probably because of the anomalous decision case). There is, however, one program in which its performance was poor: in the SINCOS program, the accuracy for strategy 3 was about 35%. This indicates that program sensitivity is significant and that performance can suffer considerably for some programs.

A disadvantage of strategy 3, and of other strategies using the target address, is that the target address may need to be computed or compared with the program counter before a prediction can be make. This tends to make the prediction process slower than for other strategies.

DYNAMIC PREDICTION STRATEGIES

Some strategies base predictions on past branch history. Strategy 2 is an idealized strategy of this type, because it assumes knowledge of the history of all branch instructions. The strategies discussed in this section are actually realizable, because they use bounded tables to record a limited amount of past branch history.

Branch history can be used in several ways to make a branch prediction. One possibility is to use the outcome of the most recent execution of the branch instruction; this is done by strategy 2. Another possibility is to use more than one of the more recent executions to predict according to the way a majority of them were decided; this is done by strategy 7. A third possibility is to use only the first execution of the branch instruction as a guide; a strategy of this type, although accurate, has been found to be slightly less accurate than other dynamic strategies.

First, strategies are considered that base their predictions on the most recent branch execution (strategy 2). The most straightforward strategy is to use an associative memory that contains the addresses of the *n* most-recent branch instructions and a bit indicating whether the branch was taken or not taken. The memory is accessed with the address of the branch instruction to be predicted, and the taken or not taken bit is used to make the prediction.

If a branch instruction is not found in the table, two issues must be considered: (1) the prediction that is to be made, and (2) the table entry that should be replaced to make room for the new branch instruction. First, if a branch instruction is not in the table, some static strategy must be reverted to for a default prediction. A good choice is to predict that the branch is taken as in strategy 2.

A more complex default strategy could be used (strategy 1a, for example), but using the simpler always predict taken strategy has a positive side effect. In particular, only branch instructions that are not taken need to be put into the table; then, the existence of a branch in the table implies it was previously not taken. Branches that were recently taken are given the proper prediction by default. One bit of memory is saved, but more importantly, histories of more branch instructions are effectively remembered. For example, if two out of the eight most-recent branch instructions executed are not taken, then all eight consume only two table entries, although all are predicted to have the same outcome as on their previous executions. A dual strategy is to use a default prediction of branch not taken and to maintain a table of branches most recently taken. Because most branch instructions are taken, however, this strategy is generally less accurate.

As far as replacement strategies, first-in first-out (FIFO) and least-recently used (LRU) seem to be two reasonable alternatives. For the application here, in which the sequence of branch instructions tends to be periodic because of the iterative structure of most programs, there is actually little difference between the FIFO and LRU strategies as far as prediction accuracy. The LRU strategy does appear to be more compatible with the scheme mentioned previously in which only branches that were not taken are recorded. Then, if a branch in the table is taken, it is purged from the table, and that table location is recorded as being least recently used. A branch that is taken subsequently fills the vacancy in the table rather than replacing a good table entry. Such a scheme for filling vacancies in the table fits naturally with the LRU replacement strategy.

Strategy 4

- Maintain a table of the most recently used branch instructions that are not taken. If a branch instruction is in the table, predict that it will not be taken; otherwise predict that it will be taken. Purge table entries if they are taken, and use LRU replacement to add new entries.

Figure 5 indicates the accuracy of strategy 4 for tables of 1, 2, 4, and 8 entries. In some cases, the accuracy

was close to strategy 1 for small table sizes and became close to strategy 2 as the table size grew. This is because small table sizes are not big enough to contain all active branch instructions, and they keep replacing each other. As a result, few branch instructions are ever found in the table, and most branches are predicted as taken. As the table size becomes large enough to hold all active branches, they are all predicted as in strategy 2.

A variation[7] allows earlier predictions than with the strategies discussed thus far. In this variation, instruction words being fetched are compared with an associative memory to see whether the following word was in sequence or out of sequence the last time the word was accessed. If it is out of sequence, a memory alongside the associative memory gives the address of the out-of-sequence word, and instruction fetching can begin at the out-of-sequence location. In this way, the prediction is, in effect, made before decoding an instruction as a branch and even before decomposing the instruction word into separate instructions. The accuracy of this strategy (75%)[7] is lower than many of the strategies given here, partly because the default prediction is effectively that the branch will not be taken. The prediction, however, can be made earlier in the instruction-fetching sequence, and can therefore lead to a smoother stream of prefetched instructions.

Another possibility for implementing a dynamic strategy when the system contains cache memory is to store previous branch outcomes in the cache.[6, 8]

Strategy 5

- Maintain a bit for each instruction in the cache. If an instruction is a branch instruction, the bit is used to record if it was taken on its last execution. Branches are predicted to be decided as on their last execution; if a branch has not been executed, it is predicted to be taken (implemented by initializing the bit cache to taken when an instruction is first placed in cache).

Figure 6 shows the result of using strategy 5 when there is a 64-word instruction cache with 4 blocks of 16 words each; replacement in the cache is the LRU strategy. The results are close to strategy 2, as expected, because an instruction cache hit ratio is usually at least 90%.

There is also a strategy that is similar to strategy 5 except in its implementation.[8] A bit is maintained for each instruction in the cache, but the bit is not directly used to make a branch prediction. First, a static prediction based on the operation code is made. Then, the prediction is exclusive ORed with the cache prediction bit. This changes the prediction if the bit is set. Whenever a wrong prediction is made, the cache prediction bit is complemented. In this way, branches are predicted as in strategy 5, but the prediction memory only needs to be updated when there is a wrong prediction. This is an advantage if there is a time penalty for updating the memory.

IMPROVED DYNAMIC STRATEGIES

Several dynamic strategies in the preceding section are quite accurate. In this section, they are refined to (1) use random access memory instead of associative memory and (2) deal with anomalous decisions more effectively.

In any of the strategies, there is always the possibility that a prediction may be incorrect, and there must be a mechanism for reversing a wrong prediction. This implies that there is room for error, and this fact can be used to replace associative memory with random access memory.

Instead of using the entire branch instruction's address for indexing into a table, it can be hashed down to a small number of bits. More than one branch instruction might hash to the same index, but at worst, an incorrect prediction would result, which could be compensated for.

The hashed address can be used to access a small random access memory that contains a bit indicating the outcome of the most recent branch instruction hashing to the same address. Hashing functions can be quite simple; for example, the low-order m bits of the address can be used, or the low-order m bits can be exclusive ORed with the next higher m bits. With these methods, branch instructions in the same vicinity will tend to hash to different indices.

Strategy 6

- Hash the branch instruction address to m bits and use this index to address a random access memory containing the outcome of the most recent branch instruction indexing the same location. Predict that the branch outcome will be the same.

Although it contributes negligibly to the results, the default prediction (when a location is accessed the first time) can be controlled by initializing the memory to all 0's or all 1's.

Figure 7 indicates the results of using strategy 6 for m equals 4 (a random access memory of 16 one-bit words). The exclusive OR hash was used. The results are similar to those of strategy 2.

A variation of strategy 6 can be used to deal with anomalous branch decisions more effectively. This variation uses random access memory words that contain a count rather than a single bit. Say the counts are initially 0 and the word length is n; the maximum count is $2^{n-1} - 1$, and the minimum count is -2^{n-1} (twos complement notation). When a branch instruction is taken, the memory word it indexes is incremented (up to the limit of $2^{n-1}) - 1$; when it is not taken, the memory word is decremented (down to the limit of -2^{n-1}).

When a branch instruction is to be predicted, its address is hashed, and the proper count is read out of random access memory. If the sign bit is 0 (a positive number or 0), the branch is predicted to be taken. If it is 1, the branch is predicted to be not taken. In this way, the histories of several of the more-recent branch executions determine a prediction rather than just that of the most recent branch execution. In the case of an anomalous branch decision, other preceding decisions tend to override the most recent anomalous decision, so only one incorrect prediction is made rather than two.

Strategy 7

> Use strategy 6 with twos complement counts instead of a single bit. Predict that the branch will be taken if the sign bit of the accessed count is 0; predict that it will not be taken if the sign bit is 1. Increment the count when a branch is taken; decrement it when a branch is not taken.

Note that strategy 6 is actually a special case of strategy 7 with a count of one bit. Also, using a count tends to cause a "vote" when more than one branch instruction hashes to the same count.

Figure 8 summarizes the results of using strategy 7 with counts of 2 and 3 bits and with a hash index of 4 bits. The accuracy is quite good; in fact, it is usually as good as or better than any strategies looked at thus far. Also, a count of 2 bits often gives better accuracy than a count of one bit, but going to larger counters than 2 bits does not necessarily give better results. This is partially attributed to the "inertia" that can be built up with a larger counter in which history in the too-distant past is used, or the history of an earlier branch instruction hashing to the same address influences predictions for a later branch instruction.

HIEARCHIAL PREDICTION

Generally, the farther an instruction is processed following a predicted branch, the greater the time penalty if the prediction is wrong. For example, if only instruction prefetches are based on a conditional branch prediction, the time penalty will probably be less than if instructions are not only prefetched but also preissued. That is, the time needed to redirect instruction prefetches is probably less than the "cleanup" time for instructions issued incorrectly.

Of course, the rewards are greater the farther an instruction is processed when the prediction turns out to be right. If a level of confidence can be attached to a branch prediction, then performance can be optimized by limiting the processing of an instruction based on the confidence that a branch prediction is correct.

Example

Assume that for CPU, if instruction prefetches are based on a branch prediction, an incorrect prediction leads to a 6 clock period (cp) delay to fetch the correct instructions. If the prediction is correct, but instructions are not issued before the outcome is known, there is a 3 cp delay to wait for the branch decision. If instructions are preissued anyway, and the prediction is correct, there is no delay at all, but if the prediction is incorrect, there is a total of a 12 cp delay.

Assume that overall, 70% of the branches can be predicted correctly. Half of the branches (Set A) can be predicted with 50% accuracy, and the other half (Set B) can be predicted with 90% accuracy. Further, assume that it is known at the time a prediction is made whether the branch instruction belongs to set A or set B.

The three possible strategies and their average delays are as follows:

1. Prefetch for all branches:
   (0.3 x 6 cp) + (0.7 x 3 cp) = 3.9 cp

2. Prefetch and preissue for all branches:

(0.3 x 12 cp) + (0.7 x 0 cp) = 3.6 cp

3. Prefetch for branches in set A; prefetch and preissue for branches in set B:
0.5 x [(0.5 x 3 cp) + (0.5 x 6 cp)] + 0.5 x [(0.1 x 12 cp) + (0.9 x 0 cp)] = 2.85 cp

The third strategy is best, because it risks the high 12 clock period penalty only when there is higher confidence of being correct.

Strategy 7 provides a natural way of implementing such a hiearchical prediction strategy. If a counter (of at least 2 bits) is at its maximum value when a prediction is to be made, the last prediction must have been that the branch would be taken, and it must have been correct. A following similar prediction would then seem likely to be correct. An analogous statement holds if a count is at its minimum value.

Consequently, a prediction based on an extremal counter value is a high-confidence prediction, and a prediction based on any other counter value is a lower-confidence prediction. Figure 9 summarizes the results of using such an approach. A 16-word RAM is used with a count of 3 bits.

In all cases studied, the predictions made at the counter extremes were more accurate. The greatest variation in accuracy was in the SORTST program in which 78.5% of the predictions were made at the counter extremes (-4, +3), and about 92% were correct. Of the 21.5% of the predictions made away from the counter extremes, only about 58% were accurate. The least variation was in the ADVAN program in which the counters were virtually always at their maximum values, and hiearchical prediction would have been of no real value.

The counter method usually achieved what was anticipated in making predictions with two confidence levels. This method could be generalized if counter ranges were broken into several intervals, a different confidence level being attached to each.

CONCLUSIONS

This paper studied the accuracy of branch prediction methods proposed elsewhere as well as new methods proposed here. A summary of the strategies follows. In the cases of strategies with several variations, only one or two representatives are indicated. Figure 10 gives a summary of the results.

Strategy 1: Predict that all branches will be taken.

Strategy 1a: Predict that only certain branch operation codes will be taken.

Strategy 2: Always predict that a branch will be decided as on its last execution.

Strategy 3: Predict that only backward branches will be taken.

Strategy 4: Maintain a table of the m most recent branches not taken. Predict that only branches found in table will be not taken.

Strategy 5: Maintain a history bit in cache and predict according to the history bit in cache and predict according to the history bit (a 64-word instruction cache was used).

Strategy 6: Hash the branch address to m bits and access a 2^m word RAM containing history bits, and predict according to the history bit.

Strategy 7: Like Strategy 6, but use counters instead of a single history bit.

The dynamic methods tended to be more accurate. Of the feasible strategies, strategy 7 was the most accurate. It also had the advantage of using random access memory rather than associative memory. For attaching levels of confidence to predictions, strategy 7 was easily adapted and gave good results. At least for the applications studied here, strategy 7 is probably the best choice based on accuracy, cost, and flexibility.

REFERENCES

[1]D. W. Anderson et al. The IBM System/360 model 91: Machine philosophy and instruction handling, *IBM Journal* (Jan. 1967), 8-24.

[2]M. J. Flynn. Some computer organizations and their effectiveness, *IEEE Transactions on Computers*, C-21 (Sept. 1972), 948-960.

[3]H. D. Shapiro. A comparison of various methods for detecting and utilizing parallelism in a single instruction stream, *Proceedings of the 1977 International Conference on Parallel Processing* (Aug. 1977), 67-76.

[4]D. E. Knuth. *The Art of Computer Programming*, vol. 3–*Sorting and Searching* (Reading, Mass.: Addison-Wesley 1973), 84-95.

[5]J. C. Gibson. The Gibson mix (Report TR 00.2043), IBM Systems Development Division, 1970.

[6]L. J. Shustek. Analysis and performance of computer instruction sets (Report 205), Stanford Linear Accelerator Center, 1978.

[7]R. N. Ibbett. The MU5 instruction pipeline, *The Computer Journal*, 15 (Feb. 1972), 42-50.

[8]S1 Project Staff. Advanced digital computing technology base development for Navy applications: The S-1 project, Lawrence Livermore Laboratories (Technical report UCID 18038), 1978.

[9]I. Flores. Lookahead control in the IBM System 370 Model 165, *Computer*, 7 (Nov. 1974), 24-38.

[10]E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps, *IEEE Transactions on Computers*, C-21 (Dec. 1972), 1405-1411.

Program	Prediction Accuracy
ADVAN	99.4
GIBSON	65.4
SCI2	96.2
SINCOS	80.2
SORTST	57.4
TBLLNK	61.5

Figure 1. Accuracy of Prediction (%) for Strategy 1

Program	Prediction Accuracy
ADVAN	98.9
GIBSON	97.9
SCI2	96.0
SINCOS	76.2
SORTST	81.7
TBLLNK	91.7

Figure 2. Accuracy of Prediction (%) for Strategy 2

PROGRAM	PREDICTED ACCURACY
ADVAN	99.4
GIBSON	98.5
SCI2	97.9
SINCOS	65.7
SORTST	82.5
TBLLNK	76.2

Figure 3. Accuracy of Prediction (%) for 1a on CY170 Kernels

PROGRAM	PREDICTION ACCURACY
GIBSON	81.9
SCI2	98.0
SINCOS	35.2
SORTST	82.5
TBLLNK	84.9

Figure 4. Accuracy of Prediction (%) for Strategy 3

	PREDICTION ACCURACY			
	Table Size			
PROGRAM	1	2	4	8
ADVAN	98.9	98.9	98.9	98.9
GIBSON	65.4	97.9	97.9	97.9
SCI2	96.1	96.1	96.0	96.0
SINCOS	76.2	76.2	76.2	76.2
SORTST	57.3	81.7	81.7	81.7
TBLLNK	61.5	61.5	91.7	91.7

Figure 5. Accuracy of Prediction (%) for Strategy 4

PROGRAM	PREDICTION ACCURACY CY17O
ADVAN	98.9
GIBSON	97.0
SCI2	96.0
SINCOS	76.1
SORTST	81.7
TBLLNK	91.7

Figure 6. Accuracy of Prediction (%) for Strategy 5

PROGRAM	PREDICTION ACCURACY
ADVAN	98.9
GIBSON	97.9
SCI2	96.0
SINCOS	76.2
SORTST	81.7
TBLLNK	91.8

Figure 7. Accuracy of Prediction (%) for Strategy 6

	PREDICTION ACCURACY	
PROGRAM	2 BIT COUNTER	3 BIT COUNTER
ADVAN	99.4	99.4
GIBSON	97.9	97.3
SCI2	98.0	98.0
SINCOS	80.1	83.4
SORTST	84.7	81.7
TBLLNK	95.2	94.6

Figure 8. Accuracy of Prediction (%) for Strategy 7 for Counters of 2 and 3 Bits

PROGRAM	% Prediction Made At Extremes	% Correct at Extremes	% Correct Not At Extremes	% Correct Overall
ADVAN	99.3	99.4	87.8	99.4
GIBSON	91.3	98.3	86.8	97.3
SCI2	98.0	99.8	99.4	98.0
SINCOS	78.4	85.7	75.1	83.4
SORTST	78.5	92.1	57.7	84.7
TBLLNK	96.8	95.2	76.2	94.6

Figure 9. Accuracy of Hierarchical Prediction for Strategy 7 with 16-Word Memory and Counters of 3 Bits

Figure 10. Summary of Results

	STRATEGY								
Kernel	1	1a	2	3	4		5	6	7
					1 entry	8 entries		16 wds	16 wds, 2 bits
ADVAN	99.4	99.4	98.9	99.2	98.9	98.9	98.9	98.9	99.4
GIBSON	65.4	98.5	97.9	81.9	65.4	97.9	97.0	97.9	97.9
SCI2	96.2	97.9	96.0	98.0	96.1	96.0	96.0	96.0	98.0
SINCOS	80.2	65.7	76.2	35.2	76.2	76.2	76.1	76.2	80.1
SORTST	57.4	82.5	81.7	82.5	57.3	81.7	81.7	81.7	84.7
TBLLNK	61.5	76.2	91.7	84.9	61.5	91.7	91.7	91.8	95.2