Limits to Branch Prediction

Trevor N. Mudge*, I-Cheng K. Chen, and John T. Coffey
Electrical Engineering and Computer Science Department
The University of Michigan
Ann Arbor, Michigan, 48109-2122

Abstract

Branch prediction is an important mechanism in modern microprocessor design. The focus of research in this area has been on designing new branch prediction schemes. In contrast, very few studies address the inherent limit of predictability of program themselves. Programs have an inherent limit of predictability due to the randomness of input data. Knowing the limit helps us to evaluate how good a prediction scheme is and how much we can expect to improve its accuracy.

In this paper we propose two complementary approaches to estimating the limits of predictability: exact analysis of the program and the use of a universal compression/prediction algorithm, prediction by partial matching (PPM), that has been very successful in the field of data and image compression. We review the algorithmic basis for both some common branch predictors and PPM and show that two-level branch prediction, the best method currently in use, is a simplified version of PPM. To illustrate exact analysis, we use Quicksort to calibrate the performance of various branch predictors. With other programs, too complicated to analyze exactly, we use PPM, as a measure of inherent predictability.

Our initial results show that PPM can approach the theoretical limit in an analyzable program and perform just as well as the best existing branch predictors for SPECInt92. This suggests that universal compression/prediction algorithms, such as PPM, can be used to estimate the limits to branch prediction for a particular workload of programs.

Keywords:

branch prediction, limit of branch predictability, prediction by partial matching, text compression.

1. Introduction

As the design trends of modern superscalar microprocessors move toward wider issue and deeper super-pipelines, effective branch prediction becomes essential to exploring the full performance of microprocessors. A good branch prediction scheme can increase the performance of a microprocessor by eliminating the instruction fetch stalls in the pipelines. As a result, various branch prediction schemes have been proposed and implemented on new microprocessors. While many researchers focus their attention on designing new branch prediction schemes, very few studies address the inherent limits of predictability. Programs have an inherent limit of predictability due to the randomness of input data. This predictability varies from data set to data set; therefore, results should not be based upon benchmarks with fixed data sets, as they usually have been in the past. This especially applies when profiling is used to "learn" the best prediction. There is a potential danger that patterns in a fixed data set may produce catastrophically bad results from the derived predictor. We will illustrate this problem with a simple example from Quicksort.

Knowing the limit of predictability helps us to calibrate how good a prediction scheme is for a given set of programs. Besides, a limit also indicates how much more we can improve the existing predictors. Unless we have some idea about what the limit should be, we cannot tell whether 90% prediction accuracy is good or 75% accuracy is bad for a particular program.

We present two approaches to measuring the limits of branch prediction: exact analysis of the program and the use of a universal compression/prediction algorithm. For simple programs, such as sorting programs, the theoretical limit of predictability can be exactly analyzed. For more complicated programs, exact analysis is impractical, so we propose that a universal compression/prediction algorithm be used to measure inherent predictability. Universal compression/prediction algorithms are well-understood and have been applied with great success in the fields of text and image compression. More recently, two examples, Lempel-Ziv and prediction by partial matching (PPM) [Cleary84, Moffat90] have been successfully applied to prefetching data from disk memories [Vitter91, Curewitz93]. In this paper we used PPM and show that, although designed for text compression, it performs just as well in this new domain of branch prediction.

This paper is organized into five sections. In section 2 we provide background for the remainder of the paper by summarize the operation of the PPM algorithm. We also summarize several branch prediction methods used in current high-performance computers. We conclude the section by showing that the best of these, two-level adaptive branch prediction is in fact a simplified version of PPM.

In section 3, we consider the prediction of the comparison branches in Quicksort, a relatively simple problem that can be analyzed exactly. In particular, we develop a theoretical limit for the predictability of Quicksort and show that PPM can closely approach this limit as its order increases, while the branch predictors of section 2.2 cannot. Section 4 further shows that PPM can perform slightly better than the best existing predictors, such as the two-level predictor, on programs from the SPECInt92 benchmark suite. This is significant in that PPM is an off the shelf algorithm that has not been tuned to provide good performance for the branch prediction problem. It is also consistent with our observation in section 2 that two-level adaptive branch prediction is just a simplified version of PPM. Finally, we present conclusions and suggestions for further work in section 5.

From these initial results, we believe that a universal compression/prediction algorithm, like PPM, can serve as a diagnostic tool to measure the inherent limit of predictability. Branch predictor designers can therefore use this powerful diagnostic tool to estimate the limit of predictability for programs. Once this limit is obtained, designers can readily evaluate the performance of their new prediction schemes.

2. Branch Prediction Algorithms

2.1 Prediction by Partial Matching

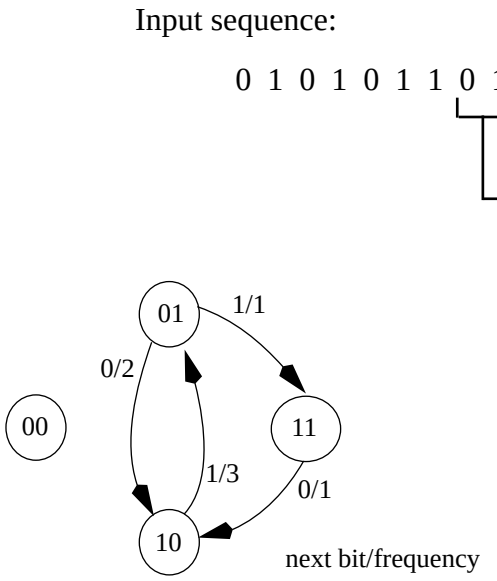
In order to compress data effectively, a compression algorithm has to predict future data accurately to build a good probabilistic model for compression [Bell90]. Many compression algorithms form a model of the probability distribution for the next symbol, and then encode the next symbol with a compressor tuned to that probability distribution. Universal compression/prediction algorithms make essentially no a priori assumptions about the source, and build the probability model recursively by adapting to the characteristics of previously examined symbols. The problem of designing efficient and general universal compressors/predictors has been extensively examined. In our experiments we draw on these techniques, adapting them to the new context of branch prediction.

Prediction by partial matching is a universal compression/prediction algorithm that has been theoretically proven optimal in data compression and prefetching [Cleary84, Krishnan94, Moffat90, Vitter91]. Indeed, it usually outperforms the Lempel-Ziv algorithm (found in Unix *compress*) due to implementation considerations and faster convergence rate [Curewitz93, Bell90, Witten94]. The PPM algorithm for text compression consists of a predictor to estimate probabilities for characters and an arithmetic encoder. We only make use of the predictor. We encode the outcomes of a branch, taken or not taken, as a 1 or a 0. Then the PPM predictor is used to predict the value of the next bit given the prior sequence of bits that have already been observed.

2.1.1 Markov predictors

The bases of the PPM algorithm of order m are a set of $(m + 1)$ Markov predictors. A Markov predictor of order j predicts the next bit based upon the j immediately preceding bits—it is a simple Markov chain. The states are the 2^j possible patterns of j bits. The transition probabilities are proportional to the observed frequencies of a 1 or a 0 that occur given that the predictor is in a particular state (has seen the bit pattern associated with that state). The predictor builds the transition frequency by recording the number of times a 1 or a 0 occurs in the $(j + 1)$-th bit that follows the j-bit pattern. The chain is built at the same time that it is used for prediction and thus parts of the chain are often incomplete. To predict a branch outcome the predictor simply uses the j immediately preceding bits (outcomes of branches) to index a state and predicts the next bit to correspond to the most frequent transition out of that state.

Figure 1 illustrates how a Markov predictor works. Let the input sequence seen so far be 010101101, and the order of Markov predictor be 2. The next bit is predicted based on the two immediately preceding bits, that is, 01. The pattern 01 occurs 3 times in the input sequence. The frequency counts of the bit following 01 are: 0 follows 01 twice, and 1 follows 01 once. Therefore, the predictor predicts the next bit to be 0 with a probability of 2/3. The (incomplete) 4-state Markov



pattern	next bit	frequency count
00	0	0
	1	0
01	0	2
	1	1
10	0	0
	1	3
11	0	1
	1	0

Figure 1: Example of a Markov predictor of order 2

The Markov chain at left corresponds to the information collected from the input sequence in the table at right. Note that the chain is incomplete, because of 0 frequency count transitions.

chain is shown at the left of the figure. Note that a 0-th order Markov predictor simply predicts the next bit based on the relative frequency in the input sequence.

2.1.2 Combining Markov predictors to perform PPM

We noted earlier that the bases of a PPM algorithm of order *m* are a set of (*m* + 1) Markov predictors. The algorithm is illustrated in Figure 2. PPM uses the *m* immediately preceding bits to search a pattern in the highest order Markov model, in this case *m*. If the search succeeds, which means the pattern appears in the input sequence seen so far (the pattern has a non-zero frequency count), PPM predicts the next bit using this *m*th-order Markov predictor as described in the previous subsection. However, if the pattern is not found, PPM uses the (*m* – 1) immediately preceding bits to search the next lower order (*m* – 1)-th order Markov predictor. Whenever a search misses, PPM reduces the pattern by one bit and uses it to search in the next lower order Markov predictor. This process continues until a match is found and the corresponding prediction can be made.

There are a number of variations on how the frequency information in the individual Markov predictors can be updated as the PPM process proceeds. In our experiments we use *update exclusion*. This means that we only update the frequency counters for the predictor that makes the prediction and the predictors with higher order. Lower order predictors are not updated.

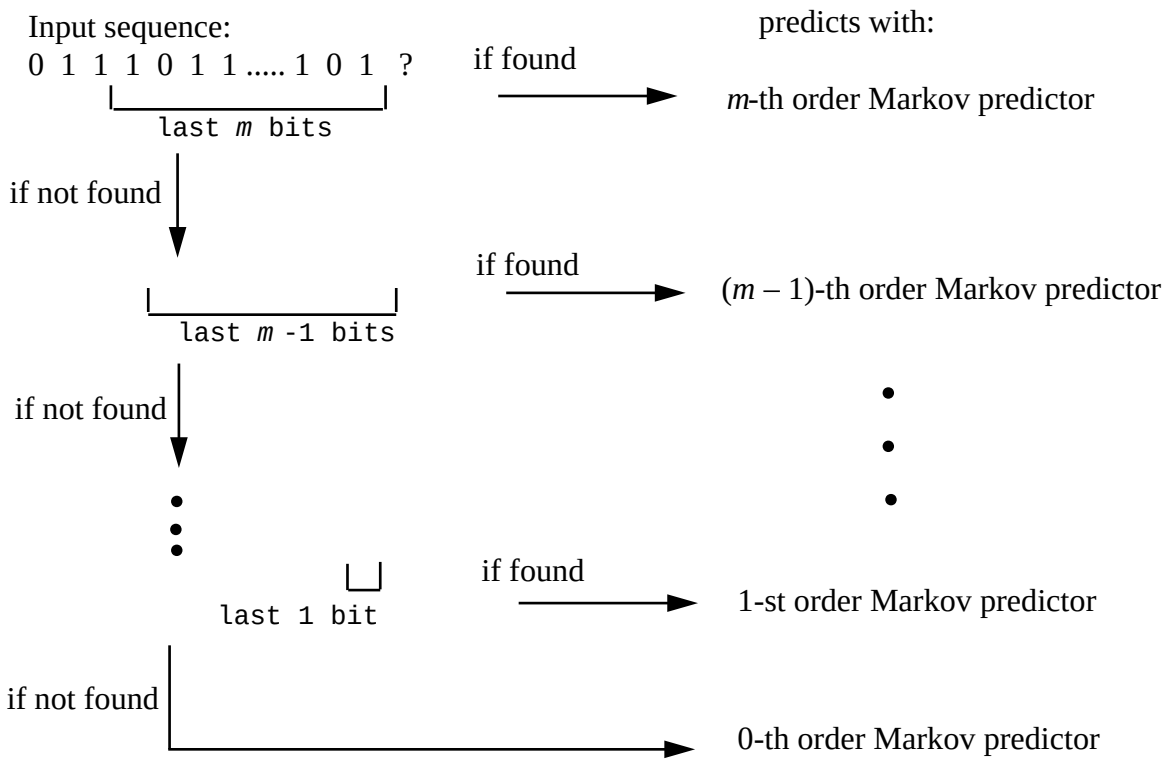


Figure 2: Prediction flowchart of a PPM predictor of order *m*

2.2 Hardware Branch Predictors

Most of today's high performance microprocessors support some form of branch prediction [MReport95]. In the next two subsections we will describe some simple methods that are widely used and one advanced method, two-level adaptive prediction, that so far has only been used on the Intel Pentium-Pro (formerly the P6).

2.2.1 Simple 1- and 2-bit Predictors

A 1-bit branch prediction scheme simply records for each branch the most recent outcome. The next time the branch is encountered during instruction execution its outcome is predicted to be in the same direction as before. This strategy says that branches tend to repeat themselves. In the case of loops for example this is usually the case: the branch controlling the iterations of a loop branches backwards for as many times as the loop is repeated. However, as has been frequently observed, this scheme results in two mispredictions when the loop condition eventually fails [Smith81].

A simple improvement that avoids this problem is to use a 2-bit saturating up-down counter. It records for each branch a count of the recent outcomes of the branch. The counter increments

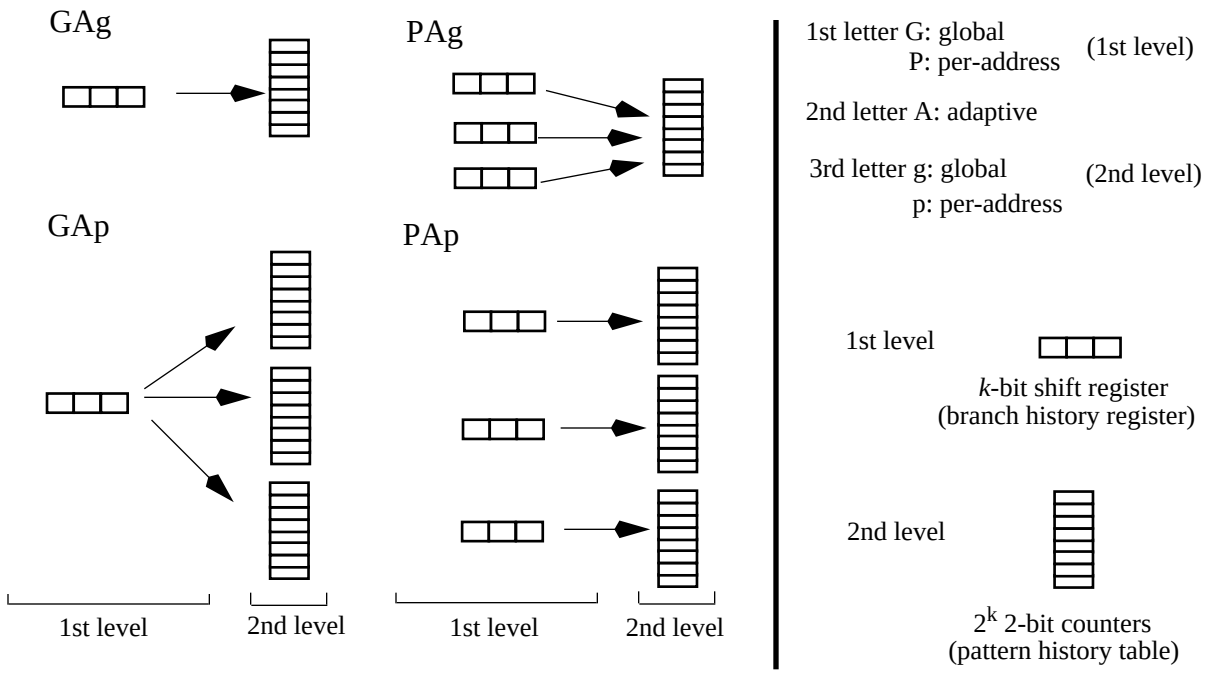


Figure 3: Four schemes for two-level predictors

its count by one when a branch is taken and decrements its count when the branch is not taken. The saturating property means that further increments (decrements) have no effect when the counter reaches its maximum(minimum) value—the counter saturates. Predictions are made based on the value of the counter selected. Typically, the predictor predicts taken when the value of the counter is a 0 or 1 and not taken when the value is -1 or -2. Work by Lee et al. and recently by Nair has shown this to be the best of the possible 2-bit schemes [Lee84, Nair95].

The 1- and 2-bit schemes as outlined above are examples of per address schemes because a separate 1- or 2- bit counter is associated with each branch in the program.

2.2.2 Two-level adaptive branch predictors

Two-level adaptive predictors are among the best predictors currently in use [Yeh92, Yeh93]. In the first level, one or more shift registers (branch history registers) are used to record past branch outcomes as a string of 1s and 0s. The branch outcome patterns recorded in the first level are then used to index into a table of counters at the second level (the pattern history table) that determines the prediction. The best performance has been observed when the second level consists of a set of 2-bit saturating up-down counter.

In this paper we consider the four schemes shown in Figure 3. We refer to them using the taxonomy proposed by Yeh and Patt. Different schemes are described by three letters. The first letter indicates, in the first level, whether the past branch outcomes are collected globally from all branches (G), or on a per address basis from each branch (P). The second letter indicates whether

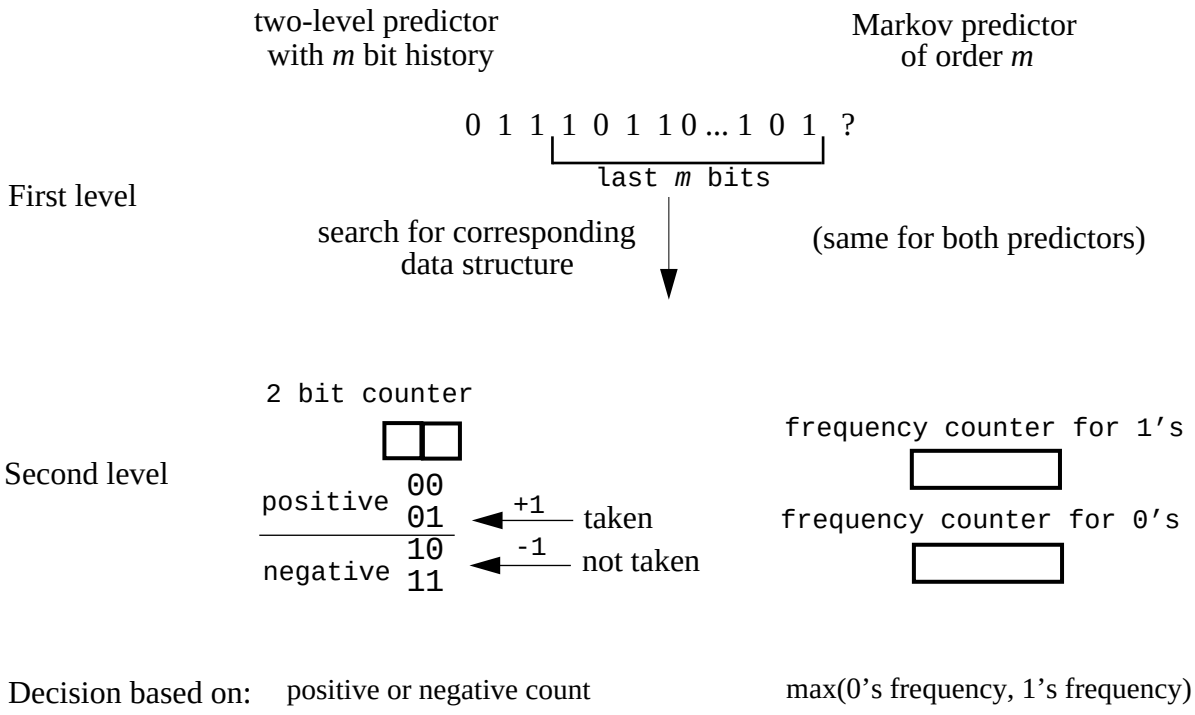


Figure 4: A two-level branch predictor vs. a Markov predictor

---

it is adaptive (A) (e.g., a counter of some kind). The third letter indicates whether the counters (or constant values if the algorithm is not adaptive) in the second level are indexed by all shift registers globally (g), or indexed by one per-address shift register (p). The four schemes shown in Figure 3 are: global branch history register with global pattern history table (GAg), global branch history register with per-address pattern history tables (GAp), per-address branch history register with global pattern history table (PAg), and per-address branch history register with per-address pattern history tables (PAp).

2.3 Two-level adaptive branch predictors as Markov predictors

From the above discussion on two-level adaptive branch predictors in section 2.2.2 and the one on Markov predictors in 2.1.1 it can be seen that there are strong similarities. Though different schemes of two-level branch predictors exist, they differ just in what subsets of branch outcomes are used to index and update the counters, and there exists a corresponding Markov predictor for each scheme. We can, without loss of generality, take a global history register and a global pattern history table scheme (GAg) to illustrate the similarity.

Figure 4 shows the similarity between a two-level and a Markov predictor. Both predictors behave exactly the same in the first level. They both use the last *m* bits of branch outcome to search the corresponding data structure. In the second level, the Markov predictor uses frequency counter

for each outcome, while two-level predictor uses a saturating up-down 2-bit counter. Whenever a branch is taken/not taken, the 2-bit counter increments/decrements. The decision for a two-level predictor depends on whether the value falls in the positive half or the negative half. Similarly, a Markov predictor simply predicts the next bit to be the most frequent outcome. The saturating counter is an approximation to this that can be realized in hardware efficiently.

The complete PPM predictor can be viewed as a set of two-level predictors, having not one size of branch history register (*m*) but a set that spans *m* down to 0 (a simple two-bit counter—equivalent to a per-address predictor with zero history length). To implementing this in hardware would require a doubling of the hardware, straining the limits of practicality. The extra hardware required for predictors of order (*m*-1) to 0 (PPM predictor) is roughly the same as that of a single predictor of order *m* (two-level predictor).

3. Exact Analysis of Predictability

The predictability of branches in some programs can be analyzed exactly, providing a provable limit to branch predictability. The analysis follows the approach common found in the concrete analysis of algorithms with the conditional branches being the object of interest. We have chosen Quicksort algorithm [Sedgewick92] to illustrate this point.

3.1 Description of Quicksort

Quicksort is a divide-and-conquer algorithm. It selects an element from the array being sorted as pivot. Then the array is partitioned into left and right subarrays such that all elements in the left subarray are less than or equal to the pivot and all elements in the right subarray are greater than or equal to the pivot. Quicksort recursively partitions each subarray until the entire array is sorted. Different variations of Quicksort exist, and we have chosen one described in [Sedgewick92]. This implementation, shown in Figure 1, first picks the right end element to be the pivot. It also keeps two scan pointers that initially point to the left end element and the next-to-rightmost element respectively. The left pointer scans to the right until an element greater than the pivot is found. Similarly, the right pointer scans to the left until an element less than the pivot is found. Then the two elements that stopped the pointers are swapped. When two scan pointers cross, the pivot and the element pointed by the left pointer are swapped. Now the pivot is in its final sorted position and it partitions the original array into right and left subarrays. The same process repeats with the right and left subarrays recursively.

Here we only consider the two branches that compare elements pointed to by pointers with the pivot value (the two while-statements printed in bold face in Figure 5). These two branches form the kernel of the algorithm and are hard to predict: their outcomes depend heavily on the distribution of the input data set. The other branches in the program are essentially 100% predictable if enough past branch outcomes and computation time are provided.

3.2 Predictability of branches in Quicksort

We assume that the *n* numbers to be sorted are distinct, and that each possible initial ordering is equally likely.

```
/** function to swap two elements **/
void swap(itemType array[], int i, int j)
{
  itemType t = array[i];
  array[i] = array[j];
  array[j] = t;
}


void quicksort(itemType array[], int left, int right)
{
   int left_pointer, right_pointer; itemType pivot;

   if(right > left)
   {
       /**assign the right end element as pivot**/
      pivot = array[right];

       /**set the initial positions of two pointers**/
      left_pointer = left - 1; right_pointer = right;

       /**infinite loop to partition the array**/
      for(;;)
      {
         /**left_pointer scans for element greater or equal to pivot**/
        while ((array[++left_pointer]<pivot)&&(left_pointer<=right));

         /**right_pointer scans for element less or equal to pivot**/
        while ((array[--right_pointer]>pivot)&&(right_pointer>=0));

         /**stop if two pointers cross**/
        if (left_pointer > right_pointer) break;

        swap(array, left_pointer, right_pointer);
      }
      swap(array, left_pointer, right);
      quicksort(array, left, left_pointer-1);
      quicksort(array, left_pointer+1, right);
   }
}
```

Figure 5: A Quicksort program and its two comparison branches

---

It is well known that each subarray of each iteration is in random order, i.e., each possible ordering is equally likely [Sedgewick92]. The expected predictability for a subarray varies according to the number of elements. It has also been shown that the overall performance of Quicksort coincides, for large enough arrays, with the performance in one iteration of the algorithm on a sufficiently large array.

At any branch, our prediction of whether the program will branch or not depends on whether the new element being examined is more likely to be greater than or less than the pivot. Suppose we have compared j elements to the current pivot, of which i have been greater. Since we have assumed that all orderings are equally likely a priori, the conditional probability that the next element is greater than the pivot is simply $(i+1)/(j+2)$ as the following conditional probability argument

shows: Compare the new element to the j numbers examined so far plus the pivot. It can be greater than from 0 to j+1 of these numbers, with each possibility being equally likely. Of these j+2 possibilities, i+1 mean the new element is greater than the pivot.

So the optimal prediction algorithm maintains a running count of the proportion of elements examined so far that are greater than the pivot, and compares this quantity to 1/2 to decide which way to predict the next branch. Equivalently, if a majority of the elements so far have been greater than the pivot, we predict that the new element will also be greater, and vice versa. (We guess randomly in the case of a tie.)

This scheme is optimal, and its prediction success rate must approach 75% from below as n becomes large. For we need only estimate whether the pivot is above or below the median, and we can do this with arbitrarily high accuracy from the first $\sqrt{n}$ computations for $\sqrt{n}$ large enough. The predictions made while this estimate is being formed make up a negligible fraction $(1/\sqrt{n})$ of the total number of predictions. Thus the scheme's performance approaches that of the situation where the rank of the pivot is known a priori. If we let p = (rank of pivot)/n, then we have p uniformly distributed over (0,1) as n becomes large, and our success rate is max*(p,* 1-*p)*. Since the pivot is equally likely to have any rank, our expected success rate is $\int_0^{1/2} (1-p)\,dp + \int_{1/2}^1 p\,dp = 0.75$.

As an aside, note that if we were attempting to compress the branch history of the Quicksort program, we would feed each symbol into an arithmetic coder that encoded according to the best estimate of the probability of the next symbol, and compress to *H(p)* bits per decision, where $H(p) = p\log_2(1/p) + (1-p)\log_2(1/(1-p))$ is the binary entropy function. Over the whole Quicksort program (involving many pivots) we would compress to $\int_0^1 H(p)\,dp$ bits per decision almost surely. The integral is $1/(2ln2)$. However, we know that the program trace can be compressed to $\log_2(n!) \cong n\log_2 n$ bits, and no further, since each of the $n!$ orderings is equally likely a priori. Thus we conclude that Quicksort has almost certainly $(2ln2)\,n\log_2 n$ decisions on average. This matches the well-known estimate of the performance of Quicksort [Sedgewick92].

Quicksort also provides a simple example of the potential dangers of extrapolating prediction performance from one program run. Suppose the program we run consists of one iteration of the Quicksort algorithm (so that one pivot is chosen). Ten million numbers are to be sorted, and the right end element (the pivot) is higher than nine million of these. Experimentation would demonstrate that an optimal predictor would be to predict branch back always in the loop for `left_pointer`, as shown in Figure 5, and do not branch back always in the loop for `right_pointer`. This would achieve the limit of 90% accuracy for this choice of pivot. However, there are two major problems. First, this figure gives a very misleading impression of the overall program predictability, since no algorithm can do better than 75% on the average. What has happened is that a pattern particular to this input data has been picked up by the predictor. Secondly, note that the predictor thus developed will perform very poorly on average, achieving only 50% accuracy (and would achieve arbitrarily close to 0% if a sufficient low pivot is chosen), and in particular the predictor developed is very poor compared to the optimal one. These problems exist even though the raw data set used is large (ten million operations).

4. Predicting branches in Quicksort

In this section we compare PPM to a one-bit saturating counter, a two-bit saturating counter,

Figure 6: Comparison of prediction accuracy for Quicksort

This graph compares the prediction accuracy of different predictors for Quicksort. Note that we only consider the two comparison branches illustrated in Figure 1. The dashed line indicates the 75% asymptotic limit.

and a two-level adaptive branch predictor. Again, we only consider the two comparison branches in Quicksort, as shown in Figure 5. The results are shown in Figure 6: PPM shows the best performance, followed by the two-bit counter (which also happens to be the best of the two-level predictors in this example), and finally the one-bit counter. Note that PPM can most closely approach the optimal predictor. Here the optimal predictor described in section 3.2 is designed specifically for Quicksort and would approach the 75% asymptotic limit if the given data sets are truly random and uniformly distributed.

For the two-level predictor, we examine all four of the schemes shown in Figure 3: a global history register with global pattern history table (GAg), global history register with per-address pattern history table (GAp), per-address history register with global pattern history table (PAg), and per-address history register with per-address pattern history table (PAp). We also use 2-bit saturating up-down counters for the pattern history table as suggested in [Yeh92]. The best results were obtained with a PAp scheme. It is plotted in Figure 6 as the line of closed circular bullets.

In practice only GAg or PAg schemes are implemented in hardware. GAp and PAp can quickly become unwieldy. In particular, a PAp scheme is usually too large to be practical but in the case of Quicksort where only two branches are under consideration, it is reasonable to consider for the purposes of a simulation. PPM still improves on the PAp scheme even though it is the most com-

# of branches	compress	espresso	eqntott	sc	xlisp
static count	90	1300	154	600	333
dynamic count	11,399,364	73,516,892	179,498,636	38,153,945	152,923,286

Table 1: Static and dynamic branch counts in our SPECInt92 programs

plex of the prediction schemes, that has the potential to capture the most information.

5. Limits of Predictability of the SPECInt92 benchmark suite

In this section, we compare the empirical results of two-level and PPM predictors on programs from the SPECInt92 integer benchmark suite. The SPECInt92 programs are not easily analyzed and therefore make ideal candidates for our study to see if a universal predictor like PPM can be used as a tool for characterizing the limits to predictability of programs.

We used ATOM [Eustace95], a code instrumentation interface from Digital Equipment Corporation, to conduct our experiments. The benchmarks are first instrumented with ATOM, then executed on a DEC 21064-based workstation running the OSF/1 operating system. The benchmarks used for the comparison are programs from the SPEC CPU Integer Benchmark Suite, Release 2/1992 (see Table 1). The branches are predicted on the fly as the execution of the program proceeds. Their agreement with the actual branch outcomes is recorded to determine the prediction success rate.

5.1 Single global branch stream and single global predictor scheme (GAg)

We start our comparisons of different predictors with the simplest scheme: using a global branch stream consisting of outcomes from all the branches and one global predictor, a set of 2-bit counters. PPM predictors of various orders are compared with equivalent two-level predictors having the same length of global history register and a global pattern table. The results are shown in Table 2.

5.2 Multiple per-address branch streams and single global predictor scheme (PAg)

We next compare the PAg two-level predictor scheme and PPM. PAg means that the inputs are divided into per-address branch outcome streams, then they are fed into one global predictor. The advantage of this, and other per-address schemes, scheme is that aliasing that may arise by mixing stream from different branch histories are reduced. However, cold starts become a more significant problem, because there may not be enough branch outcomes in the first level to train the predictor (a set of 2-bit counters) adequately. The cold start problem gets worse as the number

GAg two-level predictor

history length	compress	espresso	eqntott	sc	xlisp	Average	Weighted average
9	87.17	94.90	98.55	95.32	88.32	92.852	93.970
15	89.43	96.39	98.65	96.96	96.17	95.522	97.082
20	89.98	96.65	98.69	97.35	96.87	95.907	97.417

Global-PPM

order of PPM	compress	espresso	eqntott	sc	xlisp	Average	Weighted average
9	87.47	94.84	98.55	95.17	88.62	92.930	94.056
15	89.69	96.43	98.65	96.84	96.27	95.577	97.118
20	90.23	96.74	98.69	97.28	96.83	95.952	97.418

Table 2: Prediction accuracy of GAg style two-level predictor and PPM
The weighted average is weighted by the number of branches executed in the six benchmarks.

of distinct branches increases, since there are more individual branch outcome streams that need to be exercised.

The results are shown in Table 3. PPM performs better than the corresponding two-level predictor of the same length of history. The improvement comes from a better mechanism for dealing cold starts in which a set of Markov predictors rather than just the largest one are employed, as PAg does. If PPM can not find a complete match of the branch history, it reduces the length of the pattern and search in the lower model. On the other hand, two-level predictors filled the history registers with default values initially, which can lead them to index to the wrong counters. The accuracy decreases because not only is the wrong counter selected for prediction, but it is also incorrectly trained. The situation gets worse in the real hardware implementation. Only finite records of distinct branches can be maintained due to limited buffer size, consequently, the history will be flushed from time to time. This causes further cold start effects. PPM solves the cold start problem more gracefully than two-level predictors.

The universal algorithm, PPM, was originally designed for compressing English text and has not been applied to the field of branch prediction before. However, from the results in Figure 6, Table 2, and Table 3, we see that PPM can perform as well as, or even better on the average than, the two-level predictor that was designed specifically for branch prediction. The difference is most pronounced in the case of Quicksort where there is variance in the input data set. With regard to variance in the input data set, SPECInt92 is less satisfactory because it has a fixed input and thus

PAg two-level predictor

history length	compress	espresso	eqntott	sc	xlisp	Average	Weighted average
9	87.91	95.34	98.38	97.33	94.60	94.711	96.272
15	88.41	95.88	98.46	98.09	96.60	95.690	97.474
20	88.03	96.31	98.47	98.53	98.71	96.010	97.946

Per-address-PPM

order of PPM	compress	espresso	eqntott	sc	xlisp	Average	Weighted average
9	88.29	95.51	98.39	97.38	94.77	94.868	96.360
15	88.76	96.02	98.46	98.11	97.72	95.814	97.542
20	88.60	96.40	98.48	98.56	98.80	96.170	98.010

Table 3: Prediction accuracy of PAg style of two-level predictor and PPM

Different orders of PPM are compared with equivalent two-level predictors of corresponding history length. PPM performs better than the corresponding two-level predictor in all comparisons, since the cold-start effect in the PAg scheme gets worse as the number of distinct branches increases.

no variance. For calibrating prediction algorithms, a set of inputs would be more desirable [Sechrest95].

6. Conclusions and further work

In this paper, we assert that there is a fundamental limit to branch predictability, due to the randomness of the input data. Knowing or estimating this limit before attempting to design branch predictors is valuable. If there is a large gap between the performance of simple predictors and the limit, we can be assured that better performance can be obtained with more complex predictors. Conversely, when simple branch predictors achieve results very close to the estimated limit, we know that further effort to develop ever more complicated predictors will probably be futile.

We have applied two approaches to measure the inherent limits of branch prediction. For simple programs, such as Quicksort, we analyze asymptotic predictability exactly. In dealing with more complicated programs, we use well-understood universal compression/prediction algo-

rithms, such as PPM, as a measure of inherent predictability. Despite the fact that PPM was originally designed with text compression in mind, it is a universal algorithm, and can be expected to perform well asymptotically for almost any source. Our results indicate that it performs very well in practice in this new field, branch prediction. We have shown that, in the Quicksort program, PPM can increasingly approach the theoretical limit, while many other existing simple predictors cannot. Additionally, the simulation results on the SPECInt92 benchmark suite have further confirmed that PPM can perform as well as the best known predictors, such as two-level adaptive branch predictors. This is significant in that it establishes the usefulness of PPM as a universal predictor.

We have used PPM to establish an estimate of the ultimate limit of predictability. We do not claim that any number thus derived is provably an absolute limit. However, although it is possible that a simple, implementable predictor will substantially outperform a complex universal algorithm, this seems a remote possibility at best. Thus a universal algorithm, such as PPM, can serve as a diagnostic tool to measure the inherent limit of predictability. Branch predictor designers would therefore benefit by applying such universal algorithms to estimate the inherent limit of predictability and the performance of their new prediction schemes.

Finally, we have proposed PPM purely as a measure of predictability. Its complexity, which is roughly twice that of a two-level predictor using same history, suggests that it is not suitable for realization in hardware. Nevertheless, it may suggest some improvements to existing predictors.

References

[Bell90] Bell, T.C., Cleary, J.G. and Witten I.H. *Text Compression*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[Cleary84] Cleary, J.G. and Witten, I.H. *Data compression using adaptive coding and partial string matching*. IEEE Transactions on Communications, Vol. 32, No. 4, 396-402, April 1984.

[Curewitz93] Curewitz K. M., Krishnan, P. and Vitter, J.S. *Practical prefetching via data compression*. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D. C., 257-266, May 1993.

[Eustace95] Eustace, A. and Srivastava, A. *ATOM: A flexible interface for building high performance program analysis tools*. Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems, 303-314, January 1995.

[Krishnan94] Krishnan, P. and Vitter, J.S. *Optimal prediction for prefetching in the worst case*. Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, January 1994.

[Lee84] Lee, J.K.F. and Smith, A.J. *Branch prediction strategies and branch target buffer design*. IEEE Computer, Vol. 21, No. 7, 6-22, January 1984.

[Moffat90] Moffat, A. *Implementing the PPM data compression scheme*. IEEE Transactions on Communications, Vol. 38, No. 11, 1917-1921, November 1990.

[MReport95] Microprocessor Report, Sebastopol, CA: MicroDesign Resources, March 1995.

[Nair95] Nair, R. *Optimal 2-bit branch predictors*. IEEE Transactions on Computers, Vol. 44, No. 5, 698-702, May 1995

[Sechrest95] Sechrest, S., Lee, C. and Mudge, T. *The role of adaptivity in two-level adaptive branch prediction*. Proceedings of the 26th Annual International Symposium on Microarchitecture, Ann Arbor, December 1995.

[Sedgewick92] Sedgewick, R. *Algorithms in C++*. Reading, Massachusetts: Addison-Wesley, 1992.

[Smith81] Smith, J.E. *A study of branch prediction strategies*. Proceedings of the 8th International Symposium on Computer Architecture, Minneapolis, 135-148, May 1981.

[Vitter91] Vitter, J.S. and Krishnan, P. *Optimal prefetching via data compression*. Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 121-130, October 1991.

[Witten94] Witten I.H., Moffat, A. and Bell T.C. *Managing Gigabytes*. New York, NY: Van Nostrand Reinhold, 1994.

[Yeh92] Yeh, T-Y. and Patt, Y. *Alternative implementation of Two-Level Adaptive Branch Prediction*. Proceedings of the 19th International Symposium on Computer Architecture, Gold Coast, Australia, 124-134, May 1992.

[Yeh93] Yeh, T-Y. and Patt, Y. *A comparison of dynamic branch predictors that use two levels of branch history*. Proceedings of the 20th International Symposium on Computer Architecture, San Diego, 257-266, May 1993.