Designing Dynamic Two-Level Branch Predictors Based on Pattern Locality*

Chien-Ming Chen and Chung-Ta King

Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

Abstract. To design a good two-level predictor, we found that a low interference among branches and an even utilization of the entries in the pattern table are two key factors. In order to arrive at a balanced design and achieve the above two goals simultaneously, we first introduce the concept of branch pattern locality. Then, a new predictor design, called the *Global PAttern Locality predictor* (G-PAL), is introduced. The predictor is developed based on pattern locality and employs a cache-like pattern table to keep only those patterns that are referenced most frequently and recently. In this way, not only the interference among branches can be reduced, but the entries in the pattern table can be fully utilized.

1 Introduction

Dynamic predictors based on the *Two-level Branch Prediction* scheme [5] have been shown to achieve a substantially higher accuracy than previous predictors [3]. Two tables are used to maintain history information: *branch history table* and *branch pattern table*. With enough hardware support, the prediction accuracy can be as high as 97%. In [5], nine variations of the Two-level Adaptive Branch Prediction were identified. According to the simulation results, *GAs* (Global Adaptive Branch Prediction using per-set pattern history tables) is a good choice when 1K bytes are available to implement the predictor.

A good branch predictor should keep a long branch history, have low interference among branches, and achieve full utilization of the entries in the history and pattern tables. We say that a *branch interference* occurs when the counter that keeps track of the branch history of a particular branch is altered by the history of another branch. With a fixed hardware budget, the three goals listed above can seldom be reached simultaneously. In [1] a clever scheme, called *Gshare*, was proposed to maintain a longer branch history with limited hardware. The idea is to index into the pattern table with a certain combination of the history and the branch address. In this way, not only a longer history can be used but the table entries can be utilized better. It has been shown that Gshare has a higher prediction accuracy than previously proposed schemes.

Unfortunately although Gshare can better utilize the table entries, it spreads out the accesses to the pattern table with a pure mechanical scheme. It follows

* This work was supported in part by the National Science Council under grants NSC-85-2221-E-007-031 and NSC-85-2213-E-007-049.

that Gshare has the problem of high interference among branches, as will be shown shortly. To achieve an even better performance for a given amount of prediction hardware, we need to go back to the essential and examine branch behaviors more closely. One factor which has been overlooked in most previous works is the temporal locality of branch patterns. By *pattern locality* we mean that, if a history pattern is referenced by a branch, it is likely to be referenced again soon by this branch. As a result, accesses to the pattern table tend to concentrate on a few entries. This leads to an uneven utilization of the table and a large portion of the pattern table is seldom used for branch prediction.

In this paper a new mechanism for dynamic Two-level Branch Prediction is proposed, which is developed from the perspective of pattern locality. The key is to focus on those patterns which are referenced frequently and recently. A very nature design strategy is thus to configure the pattern table as a set-associative cache. We will describe the design of such a predictor, called the *pattern locality predictor*. Experimental results show that given a fixed hardware budget, our scheme can better reduce the interference among branches and fully utilize hardware resources. As a result, a prediction accuracy higher than previously proposed schemes can be achieved.

2 Branch Interference and Table Utilization

Table utilization: the number of accesses to each entry in the pattern table

Fig. 1 shows the distribution of the accesses to the pattern table entries for four different predictor configurations. These configurations use roughly the same amount of hardware. The SPECint92 benchmark 008.espresso was chosen for our study. The experimental environment will be described in detail in Section 4. In this experiment, we assumed that there were a total of 8192 entries in the pattern table. During execution, the number of accesses to each entry in the table was recorded. The entries were then sorted in a decreasing order according to the recorded numbers and plotted in the figure.

From the figure we can see that a large portion of pattern table entries in GAs(m, n) are seldom used for branch prediction, especially when n is large. For example, near half of entries are accessed fewer than 256 times in GAs(11,4) and GAs(13,1), while that number drops to 100 in GAs(9,16). For GAs(9,16), about 1350 entries are never even referenced at all, which is a waste of precious silicon resources. Gshare, on the other hand, has more even accesses to the table entries. For example, exceeding 6500 entries are accessed more than 100 times for Gshare, compared with 4000 entries for GAs(9,16). This is one reason why Gshare outperforms GAs.

Branch interference: the number of branches which share an entry in the pattern table

Fig. 2 shows the number of branches that share individual pattern table entries.

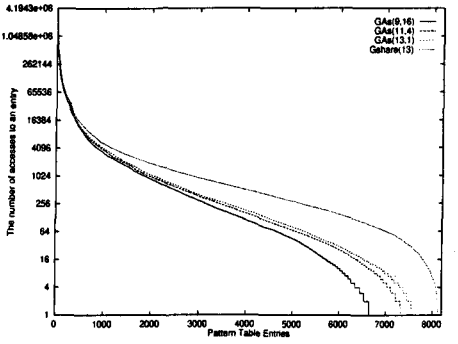

Fig. 1. The number of accesses to the entries in the pattern table for 008.espresso

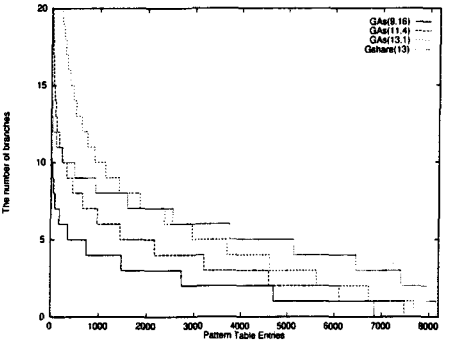

Fig. 2. The number of branches which share an entry of the pattern table for 008.espresso

Again, 008.espresso was used and the entries were sorted in a decreasing order according to the number of sharing branches.

From the figure we can see that table entries in Gshare are shared by more branches than in other predictors. This is due to the exclusive-OR operation between branch history and branch address in Gshare. As a result the interference among branches increases, which in turn influences the correctness of branch prediction [6].

From the above discussions we can see that Gshare suffers from the problem of branch interference while GAs cannot fully utilize hardware resources. In the next section, we will present a new predictor, called the *pattern locality predictor*, which utilizes available hardware better and reduces branch interference. The key to such a predictor is pattern locality.

3 The Pattern Locality Predictor

3.1 The Concept of Pattern Locality

The concept of branch pattern locality is similar to that of locality of memory references: if a history pattern is referenced by a branch, it is likely to be referenced again soon by this branch. For example, a pattern with a "taken" in every bit is likely to happen frequently, especially in loops. Note that branch patterns are defined by the length of the history register in the predictor. Thus, pattern locality is a function of both the application program and the predictor architecture. A high pattern locality means that accesses to the pattern table tend to concentrate on a few entries. This leads to an uneven utilization of the table and it is not desirable.

To observe pattern locality of programs quantitatively, we define the concept of *coverage percentage*. Let the length of the history register be m. Then, there are 2^m different history patterns. Let $B_1, B_2, ..., B_t$ be the static branches in the

given program. For each branch $B_i, 1 \leq i \leq t$, the number of times that a history pattern is referenced during the execution of the program can be recorded. Let it be $c_j(B_i)$ for $1 \leq j \leq 2^m$. Without loss of generality, assume that the reference counts of the history patterns are sorted in such a way that $c_j(B_i) \geq c_k(B_i)$ if $j < k$. Define the *coverage percentage for "n-pattern"* to be the percentage of the top n reference counts in the overall counts, averaged across all static branches, i.e.,

$$\text{coverage percentage for "}n\text{-pattern"} = \frac{\sum_{i=1}^{t} \left(\frac{\sum_{j=1}^{n} c_j(B_i)}{\sum_{k=1}^{2^m} c_k(B_i)} \right)}{t}$$

The coverage percentage serves as a measure of pattern locality. For a given n, a high coverage percentage indicates that a large portion of pattern references "hit" one of those n patterns. Thus, the pattern locality is high.

Fig. 3 shows the coverage percentage of several SPEC92 benchmarks. In this experiment, the history register was assumed to have a length of 12 bits. In other words, there were 4096 different patterns. It can be observed that the coverage percentage for 8-pattern is over 92% and that for 16-pattern over 97%. When 64-pattern is considered, the coverage percentage exceeds 99.8%. This implies that entries in the pattern table which do not correspond to these 64 patterns are hardly used.

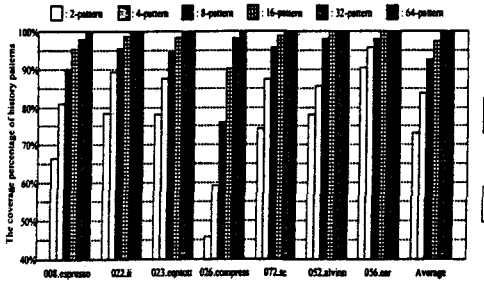

Fig. 3. The locality of branch history patterns

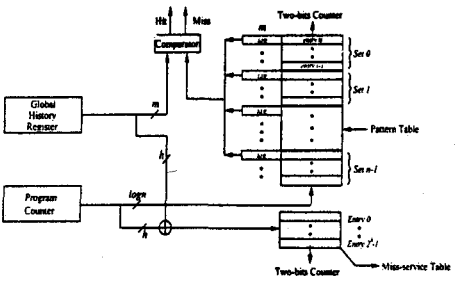

Fig. 4. The organization of a G-PAL predictor

From this experiment, we can conclude that branch patterns in typical programs exhibit high locality. Eliminating those entries in the pattern table that are seldom referenced should have a negligible effect on the prediction accuracy. A predictor designed based on pattern locality should have an even utilization of the pattern table.

We call the predictor that uses the pattern locality of the execution history as the *PAttern Locality Predictor* (PAL predictor). We will only consider the PAL predictor that uses a global history scheme. Such a predictor is called the G-PAL predictor and will be introduced in the next subsection.

3.2 The G-PAL Predictor

Our predictor design takes into account of both branch interference and table utilization. Fig. 4 shows our design of a G-PAL(m, n, i, h) predictor. In the predictor the pattern table is configured similar to an i-way set-associative cache. There are n sets in the table. Each set has i entries, each of which contains a 2-bit saturating counter and an m-bit tag, where m is the length of the history register. There is also a *miss-service* table, in which there are 2^h entries, each contains a 2-bit saturating counter.

Operations of the predictor are as follows. When a branch is encountered during program execution, the $\log n$ lower-order bits of the program counter is used to select one set from the n sets in the pattern table. The content in the global history register is compared simultaneously with the tags of all the entries in the selected set. If there is a match, then the branch is predicted according to the corresponding 2-bit counter. When the direction of the branch is finally resolved, the counter is updated accordingly.

If the global history cannot find a match with the tags of the selected set, a miss occurs. As cache misses, the fetch unit fetches the actual status bits of the branch from a mapping table maintained in the main memory. The mapping table consists of n sets, each of which has 2^m entries. Each entry is a 2-bit field. Again, the $\log n$ lower-order bits of the program counter selects the target set in the mapping table and the global history register indexes into an entry of the set.

One problem with the above design is that the memory access during a miss will take a very long time. It is unrealistic to have CPU wait for the completion of the access. The predictor in Fig. 4 uses the miss-service table to help the prediction in the case of misses. If a miss occurs, the h lower-order bits of the global history will be combined with the h lower-order bits of the program counter with an exclusive-OR. The resultant bits are indexed into one entry of the miss-service table, and the branch is predicted according to the selected entry.

When the requested data is finally returned from the main memory, the 2-bit counter in the entry which is least recently used in the corresponding set will be replaced by the data. Furthermore the replaced counter is written back to the main memory. When the branch is resolved, the 2-bit counters in the corresponding entries in the pattern table and in the miss-service table should be updated accordingly.

3.3 Characteristics of the G-PAL Predictor

In this subsection, we examine the G-PAL predictor in terms of branch interference and table utilization. Fig. 5 and 6 show the distribution of the accesses to the pattern table entries and the number of branches that share an entry, respectively. In the experiments, there were a total of 8192 entries in the pattern table. Three configurations, G-PAL(12,512,16,10), G-PAL(12,128,64,10), and G-PAL(12,32,256,10) were evaluated.

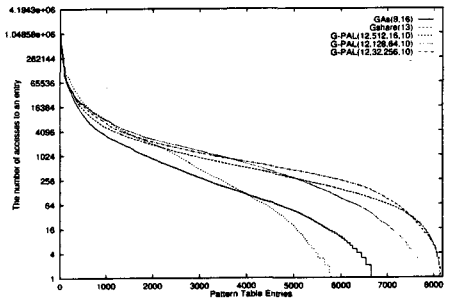

Fig. 5. The number of accesses to the entries in the pattern table in 008.espresso

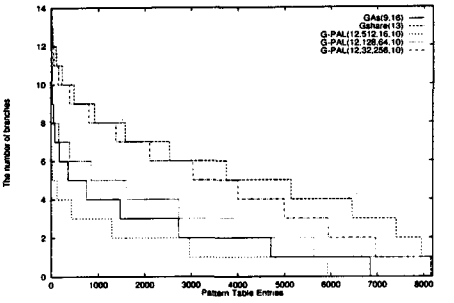

Fig. 6. The number of branches which share an entry of the pattern table in 008.espresso

From Fig. 5, it can be observed that G-PAL(12,512,16,10) and G-PAL(12,128, 64,10) have a utilization distribution close to that in GAs(9,16) and Gshare(13), respectively. However, G-PAL(12,512,16,10) and G-PAL(12,128,64,10) have much lower branch interference than GAs(9,16) and Gshare(13), as shown in Fig. 6. For G-PAL(12,32,256,10), the interference is slightly lower than Gshare(13), but it has more even table utilization.

The above observations can be explained as follows. For the G-PAL predictors, entries in set i are only used by those branches which have their addresses modulo by n being i. However, for Gshare each entry of the pattern table may be used by all the branches. Therefore, G-PAL potentially has a lower branch interference than Gshare. On the other hand, by taking the advantage of pattern locality, the G-PAL predictor can get rid of those seldom referenced entries in the pattern table. Given a fixed history register, G-PAL can use a smaller pattern table than GAs to achieve the same performance. These factors translate into performance, as is shown in Fig. 7.

4 Performance Evaluation

In this section, we compare the G-PAL predictor with previously proposed predictors, including Profiling [2], 2bC [3], GAs, and Gshare. We evaluated the performance of predictors using the benchmarks in the SPEC92. In the experiments, each benchmark was executed for 30 million branches or until completion.

4.1 The Experimental Environments and Assumptions

The experiments were conducted on a DEC 3000 workstation running OSF/1 version 3.2. The object code generated by the compiler was instrumented by ATOM [4]. We implemented the G-PAL predictor as an analysis procedure which was inserted before the conditional branch. During program execution, statistics information about branch prediction were collected and reported.

Table 1. The estimated costs for different predictors

Predictor	History Register Length	# of Pattern Tables	# of Entries in Each Pattern Table	Estimated Cost (bits)
Profiling(n)	N/A	N/A	N/A	n
2bC(n)	N/A	1	n	$2 \times n$
GAs(m, n)	m	n	2^m	$m + 2 \times n \times 2^m$
Gshare(n)	n	1	2^n	$n + 2 \times 2^n$
G-PAL(m, n, i, h)	m	n	i	$m + n \times (2i + m \times i + i \log_2 i) + 2^{h+1}$

4.2 Experimental Results

We compare the performance of various branch predictors for a fixed hardware budget. We use a simple cost model to estimate the hardware costs for various branch predictors, which is shown in Table 1. For the Profiling(n) predictor, only one prediction bit is needed, which is set by the compiler. Thus, the cost of the predictor is n bits, where n is the number of static branches. The costs for 2bC(n), GAs(m, n) and Gshare(n) predictors can be computed easily from their organization. For the G-PAL(m, n, i, h) predictor, each entry in the miss-service table is a 2-bit counter, while each entry in the pattern table contains a 2-bit counter, an m-bit tag, and a $\log_2 i$ counter for the LRU replacement algorithm. It follows that the predictor requires $m + n \times (2i + m \times i + i \log_2 i) + 2^{h+1}$ bits.

For predictors such as GAs and G-PAL, more than one configuration can be chosen under a specific budget. For example, if 8K bytes are available, then GAs(15,1), GAs(14,2), GAs(13,4), ..., are all viable configurations. In such a case, we select the one with the highest prediction accuracy for comparison.

Fig. 8 shows the experimental results of the comparison. The prediction accuracies of Profiling and 2bC are bounded above by 93%. They cannot compete with GAs, Gshare, or G-PAL. Gshare is the best choice for predictor designs when the hardware budget is limited to 256 bytes. As more hardware is available for predictors, its prediction accuracy can reach 96.5%. For GAs, the prediction accuracy is slightly lower than that of Gshare. As the hardware budget increases, the difference between GAs and Gshare decreases. Thus Gshare is a good design choice, especially when the allocated hardware budget is small.

For G-PAL, we use the configurations G-PAL(7,16,8,8), G-PAL(10,16,16,11), G-PAL(15,16,64,12) and G-PAL(21,16,256,11), if the allocated hardware is 256 bytes, 1K bytes, 4K bytes and 16K bytes, respectively. In G-PAL the tags consume the most hardware, and only a small fraction is needed for the 2-bit counters. For example, in G-PAL(7,16,8,8) the tags require 112 bytes, the 2-bit counters 32 bytes, the LRU counters 48 bytes, and the miss-service table 64 bytes.

Compared with Gshare, G-PAL surpasses Gshare in terms of prediction accuracy when more than 1K bytes are available for predictors. As the hardware budget increases, the difference between the prediction accuracies of G-PAL and Gshare increases. The prediction accuracy of G-PAL(21,16,256,11) reaches 96.9%

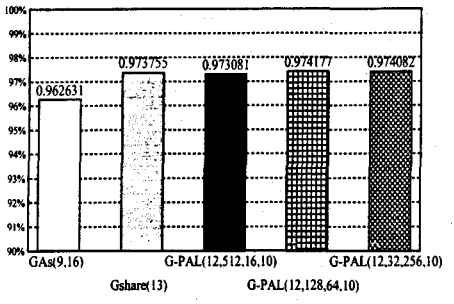

Fig. 7. The prediction accuracy of various predictors on 008.espresso

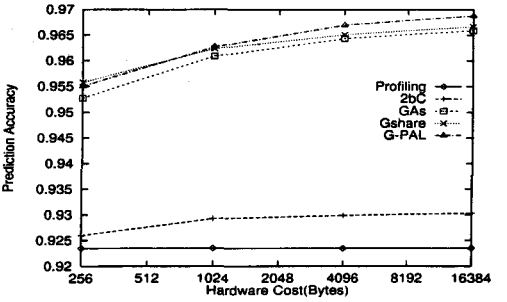

Fig. 8. Comparisons of various predictors

for the tested SPEC92 benchmarks. It should be noted that G-PAL has fewer 2-bit counters than GAs and Gshare — under the same amount of hardware. However, G-PAL still outperforms them.

5 Concluding Remarks

This paper presents a new approach to designing dynamic two-level branch predictors. The technique is based on the temporal locality of branch patterns. A cache-like pattern table is used to keep only those patterns that are referenced most frequently and recently. Experimental results show that the resultant G-PAL predictor achieves a higher prediction accuracy than previously proposed two-level branch predictors, when the hardware budget is more than 1K bytes.

References

1. Mcfarling, S.: Combining Branch Predictors. WRL Technical Note TN-36. Digital Equipment Corp. (1993)
2. McFarling, S., Hennessy, J.: Reducing the cost of branches. Proc. of the 13th Annual International Symposium on Computer Architecture. (1986)
3. Smith, J.: A Study of Branch Prediction Strategies. Proc. of the 8th Annual International Symposium on Computer Architecture. (1981)
4. Srivastava, A., Eustace, A.: ATOM: A System for Building Customized Program Analysis Tools. Proc. of the SIGPLAN'94 Conference on Programming Languages Design and Implementation. (1994)
5. Yeh T., Patt Y.: A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. Proc. of the 20th Annual International Symposium on Computer Architecture. (1993)
6. Young, C., Gloy, N., Smith, M.: A Comparative Analysis of Schemes for Correlated Branch Prediction. Proc. of the 22th Annual International Symposium on Computer Architecture. (1995)