# Evaluation of branch-prediction methods on traces from commercial applications

by R. B. Hilgendorf
G. J. Heim
W. Rosenstiel

**For modern superscalar processors, branch prediction is a must, and there has been significant progress in this field during recent years. For the IBM System ESA/390™ environment, a set of traces exists which represent different kinds of commercial workloads, and they include operating-system interactions. We have used four of these traces to evaluate a large variety of branch-prediction algorithms in order to identify possible design tradeoffs. One property of ESA/390 architecture is that for most branches, target address calculation involves the use of values stored in general-purpose registers. Therefore, not only branch directions but target addresses must be predicted. When performing prefetch-time prediction, a branch target buffer (BTB) is used to provide/predict the target address. In this paper, all evaluated prediction methods are combined with such a BTB. The resulting size for the BTB is significantly larger than for designs evaluated with SPECmark™ traces. Algorithms for determining branch direction are examined and compared. These algorithms include *local branch history* methods as well as *global history* and *path history* procedures. Finally, combinations of some of these methods, known as hybrid predictors, are evaluated. The path history algorithm we use is an adaptation of a known algorithm, but including it in the hybrid predictor is new. For all of these methods, design parameters are varied to find the tradeoff between the hardware needed and the prediction quality achieved. Results, except for those for the path predictor, are comparable to SPECmark results, except that for most cases less history must be used. Another property of ESA/390 architecture, the absence of specific subroutine call and return instructions, led to the investigation of hardware for self-detecting call/return pairs. A new approach has been developed, and its prediction quality is demonstrated. All of the methods described above use a BTB. A BTB performs well if branches have fixed targets. However, about 5% of the branches we consider have changing target addresses. Very recently an algorithm was proposed for treating such branches using a modification to the BTB approach. We have implemented an enhancement to this method, and the prediction correctness achievable using the enhanced method is shown in the results presented in this paper. Finally, combining several of the investigated schemes increases**



0018-8646/99/$5.00 © 1999 IBM

**Table 1** Trace statistics.

| *Trace* | *T1* | *T2* | *T3* | *T4* |
|---|---|---|---|---|
| No. of dynamic instructions | 1,300,881 | 1,325,359 | 1,309,178 | 1,667,468 |
| No. of dynamic branches | 285,528 | 321,441 | 312,865 | 359,550 |
| No. of other instructions | 1,015,359 | 1,003,918 | 996,313 | 1,307,918 |

**Table 2** Dynamic vs. static branches.

| *Trace* | *T1* | *T2* | *T3* | *T4* |
|---|---|---|---|---|
| No. of dynamic branches | 285,528 | 321,441 | 312,865 | 359,550 |
| No. of static branches | 19,176 | 27,878 | 21,202 | 15,491 |

**branch-prediction correctness in commercial environments. However, it remains to be shown whether the tremendous increase in hardware required for their implementation can be justified.**

## 1. Introduction

Branch prediction is one of the key issues in modern superscalar processor design. Better branch prediction will improve the performance a processor can achieve. Even though there has been significant progress in this field during recent years [1–7], it is still not clear which if any of the currently advocated methods is superior.

The quality of a method is often measured by applying it to the traces of the SPECmark** test suite. These traces are taken from typical runs of single applications and with respect to branch prediction show quite different behaviors for different traces. Often the average of the prediction rate for each member of the suite is taken, while in other cases only selected programs are used to stress certain aspects. However, the *gcc* trace seems to be the most difficult for branch prediction [4].

There is much more activity in a processor if an operating system is running and/or multitasking is allowed than can be seen in a trace from a single task. Commonly, this is taken into account by frequently flushing the history of branch behavior that has accrued up to the flush time [8]. For the IBM ESA/390* there exists a completely different set of traces that specifically show the involvement of the operating system. In this paper we apply to this set of traces the known methods for branch prediction and show the influences of frequent task switching and interrupt handling as they are reflected in those traces.

The remainder of the paper is organized as follows. The next section provides details on the traces used for all of the experiments. Section 3 deals with some specifics of the ESA/390 instruction set, which in turn have influence on methods used and impose additional requirements. Results are presented starting with Section 4, which deals with the influence of size and organization of the branch target buffer. It is followed by direction-prediction results which can be achieved using local history information (detailed in Section 5) and using global history algorithms (in Section 6). Section 7 details results from combining several history algorithms for direction prediction into hybrid predictors. Section 8 describes the prediction of the branch target addresses, including both methods for subroutine returns and methods for indirect branches which frequently change their target address. Section 9 then summarizes the results and shows how to combine the partial results of the preceding sections to obtain an overall prediction result for each of the traces.

## 2. Trace description

The analysis in this work is based on four different traces. Each trace is a condensed version of several traces taken for a well-defined sequence of applications in a well-defined environment. The traces as such are consistent; i.e., there are no breaks due to the compression. As the application sequences are defined, the four traces represent completely different areas of commercial workloads. These areas are as follows:

T1: Transaction processing, such as database queries in warehouse management and stock keeping.
T2: Interactive usage in program development, such as searches, editing, compilation, and testing.
T3: Transaction processing from tasks, such as hotel reservation, banking, and order processing.
T4: Commercial batch jobs compiled from 130 single jobs, including compilation, link, and run steps as well as assist programs used for data manipulation.

To provide statistics on our traces, we must first make a distinction between static branches and dynamic branches.

*Static* branches are those branches found in the binary program. From traces these branches can be extracted by sorting a trace according to the instruction addresses and eliminating all entries having the same instruction address as the first occurrence.

*Dynamic* branches are all branches which occur as a result of running the program and thus are found in the trace. A static branch may have many occurrences as a dynamic branch; e.g., when the program performs a loop, the static branch at the end of the loop will occur as many times dynamically as the loop does.

**Table 3** Dynamic executions per static branch for Trace T4.

| No. of dynamic execs. | 1 | 2–3 | 4–7 | 8–15 | 16–31 | 32–63 | 64–127 | 128–255 | >256 |
|---|---|---|---|---|---|---|---|---|---|
| No. of static branches | 3,402 | 3,663 | 2,402 | 1,743 | 1,532 | 1,323 | 981 | 245 | 190 |

In analogy we use the term *dynamic* instructions to mean the total number of instructions in the trace. The traces are relatively short, with a rate of 4 to 1 between instructions and branches. **Table 1** gives the exact numbers. If we look now for the static branches in the trace (**Table 2**), it can be seen that on average each branch is used between 10 and 20 times. This suggests that history-based branch prediction could work well. However, the number of times the branches are used in the trace is not the average. As shown in **Table 3** for trace T4, about half of the branches occur only up to three times [9]. This may give an early hint as to the length of history that might be necessary.

The statistics above differ greatly from those of other traces. Yeh and Patt [10] used traces including only 200 to 7000 static branches, with a majority of the traces being well below 700 static branches. The average length of their trace is about 10 000 000 instructions.

Besides being different in length and number of static branches, our traces also include jumps to interrupt service routines (initiated without a branch instruction). See **Table 4**. Interrupts as such are not problematic for branch prediction, but due to the completely different address range, they may force replacement of a branch from a finite history table that may be needed again after the interrupt service routine is completed and the original task has resumed.

Because branch prediction normally works on effective (virtual) addresses, switching address spaces (the same virtual address points to a different absolute address) as usual on task switches brings some problems into branch prediction. A branch for which history has been gathered will no longer be present or will have been replaced by another branch. Each of our traces contains a number of switches of the address space.

## 3. Properties of the ESA/390 instruction set

The ESA/390 instruction set uses instructions of three different lengths: two bytes, four bytes, and six bytes. All instructions must be aligned on a two-byte boundary. For the remainder of this section, we concentrate on branch-related instructions only.

The ESA/390 instruction set contains about 17 different branch instructions. Even though this set contains branches which allow the branch target address to be calculated relative to the current instruction pointer, the majority of the branches in the traces under investigation receive their target address in other ways:

**Table 4** Number of interrupts.

| *Trace* | *T1* | *T2* | *T3* | *T4* |
|---|---|---|---|---|
| No. of interrupts | 35 | 65 | 97 | 115 |

- The target address is to be taken from a general-purpose register.
- The target address is to be calculated as base register plus index register plus displacement, a calculation involving two register values and one constant value.

Furthermore, coding the index register address as zero will force even unconditional branches never to branch. When using a certain mask for a conditional branch, the branch will always be taken. Thus, using only the opcode of a branch will not provide enough information.

A further obstacle presented by the ESA/390 instruction set is the fact that no distinct call nor return instruction exists. There are instructions that are commonly used for this purpose, but these are used in other contexts as well.

From the above we conclude that our approach should be as follows:

- We combine all methods evaluated with the use of a *branch target buffer* (BTB) when performing prefetch time branch prediction [11]. The basics on BTB size and its influence are detailed in the next chapter.
- We exclude the problem of the nonexisting call/return pair from the normal analysis of prediction methods for branch direction and deal with it in a separate chapter on moving targets. Here techniques such as those in [12] are examined.

## 4. BTB size and organization

The BTB is a storage structure that is intended to contain the target address for each branch in the program as well as prediction information [8]. It is normally addressed using the instruction address of the branch instruction. As it is not possible to have a BTB as large as the address space of a processor, techniques known from cache design are used to fold the complete address space into a usable BTB of affordable size. The basic question is that of the influence of a finite table size on branch-prediction rates. **Figure 1** shows simulation results for all of our traces, varying the BTB size.

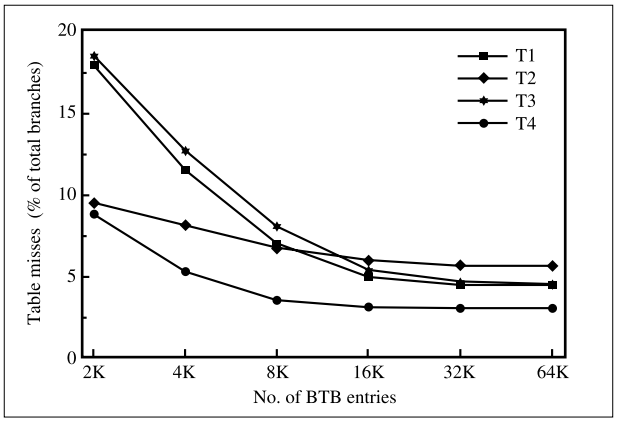


**Figure 1**

Influence of BTB size on table misses.

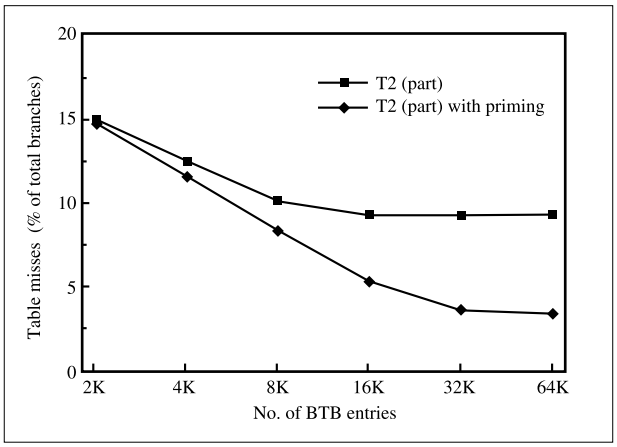


**Figure 2**

Influence of priming on table misses.

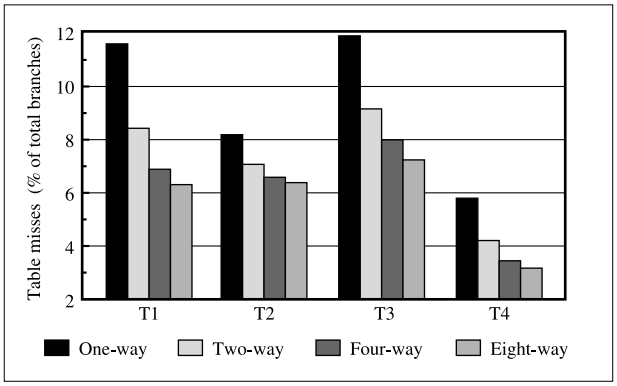


**Figure 3**

Influence of BTB associativity on table misses.

The number of times a taken branch is not found in the BTB is counted as an equal number of misses. Thus, this number consists of the *first* or *cold* misses occurring when a branch is detected the first time, as well as the misses due to replacement. For BTB sizes greater than 32K entries, the curve becomes flat for all traces. Here only the first misses appear to be left over. The values are close to the number of static branches for each of the traces in Table 2.

The BTB size and the number of first misses are obviously different from those deduced for SPEC traces, because those traces contain fewer static branches and each branch is executed many more times. Unless the BTB is preloaded (primed), the first misses are always some minimum number of mispredicted branches. To show the effect that priming may have, a trace of about one million instructions, taken immediately prior to a part of the trace which is used for measuring the miss rate, was executed on our simulator. After this, all of the measuring counters were reset and then the usual trace was executed. In **Figure 2** it can be seen that for small BTB sizes there is almost no difference in the miss rate between a primed and an unprimed environment. Here replacement misses are the dominant factor. However, for large BTB sizes the number of table misses falls below the number of static branches in the trace used to measure the misses.

Making the BTB as large as possible appears to be advantageous. A large BTB, however, has its own disadvantages. First, its physical size and its access time may be too large to fit the design point of the processor. Second, the BTB stores effective addresses, not absolute addresses. After an address space change, e.g., a task switch, part of the BTB content is useless, or worse, counterproductive. Instructions which in fact are not even branches may be predicted as branching instructions because of mapping to a BTB entry from the previous task. These are called false hits and could require partial purges to a large BTB. The number of replacement misses for a given BTB size is a function of the associativity. **Figure 3** shows the simulation results for all traces using different associativities for a BTB of 8K entries. We chose a size of 8K because this size appears to be implementable, and enough replacement takes place not to be affected by artifacts. It can be seen that using an eight-way-associative buffer increases the hit rate by 0.3% to 0.6% over the base four-way BTB used for creating the results shown in Figures 1 and 2.

Finally, modern processors generally fetch more than a single instruction per cycle. Therefore, an address range rather than a single instruction address must be tested for the occurrence of a branch. This is achieved by addressing the BTB not with instruction addresses but rather with block addresses. **Figure 4** shows the degradation resulting from such coarser access. Degradation is not high, but if possible, access using instruction addresses is preferable.

For all following discussion, unless stated otherwise, an 8K-entry, eight-way-associative BTB with instruction address access resolution is presumed.

## 5. Prediction of branch direction based on local history

In this section we summarize all methods that predict the outcome of a branch from the recorded previous outcomes of that branch.

• *History patterns with fixed assignments of branch direction*

The simplest way of predicting a branch is to assume that a new occurrence of the branch will behave the same way as the last prior occurrence. Such prediction uses a so-called "local 1"-bit history. If the outcomes of several consecutive occurrences of a branch are recorded, the prediction may become more accurate. Perleberg [8] performed some statistical analysis on the behavior of a branch according to the last two and/or last three outcomes of that branch. To form the history, a shift register which holds one bit per outcome is used. **Table 5** shows the results from such an analysis; the history pattern in the first column shows the leftmost being the oldest of the last three branch outcomes. The second column shows as a percentage the frequency with which a branch with such a history will be taken, and the last column shows the suggested prediction for that branch. Patterns with a frequency of more than 50% taken will be predicted as *taken*. We used these static assignments for simulation with our traces. The results are shown in **Figure 5**.

To build up its history, a branch must be executed several times without being fully predictable. If branch history begins to be recorded only after the branch has occurred the first time, the history register may be initialized using a predefined pattern. The best results are achieved if a pattern is chosen which will predict the branch the next time as *taken*. If this prediction turns out to be wrong, the new pattern that is created by augmenting the history pattern with an N should then predict *not taken* for the third occurrence of the branch. It can be seen that increasing the number of history bits increases the branch prediction quality as well. However, from a two-bit to a three-bit history the increase is very moderate, about 0.2%.

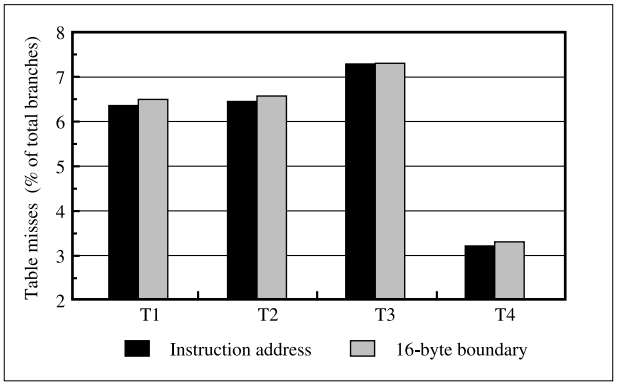


**Figure 4**

Influence of addressing resolution on table misses.

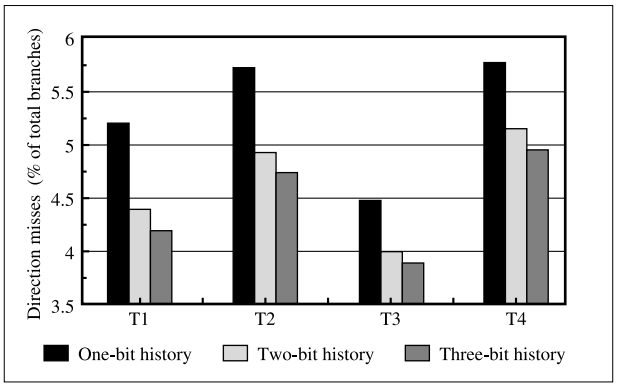


**Figure 5**

Local-history prediction using static patterns.

**Table 5** Static prediction for three-bit history pattern.

| *Pattern* | *Next is taken* (%) | *Prediction* |
|---|---|---|
| NNN | 7.8 | not taken |
| NNT | 34.1 | not taken |
| NTN | 51.9 | taken |
| NTT | 67.9 | taken |
| TNN | 32.6 | not taken |
| TNT | 64.4 | taken |
| TTN | 79.1 | taken |
| TTT | 97.7 | taken |

Analyzing the pattern and the action to follow shows that only when a branch is not taken twice in a row or more will it be predicted as not taken. Such a condition can also be satisfied using a counter or a finite-state machine.

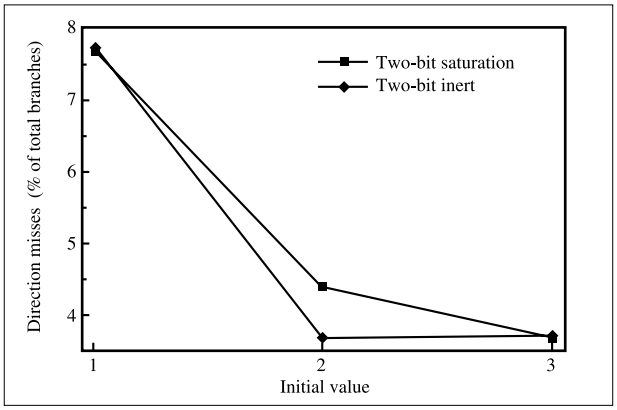


**Figure 6**

Influence of initial value of FSM.

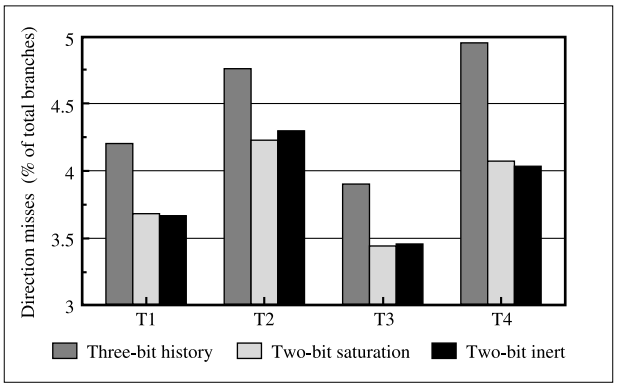


**Figure 7**

Local-history predictor using FSMs.

• *Local history recorded in finite-state machines*
In [13] and [14], two-bit counters are suggested for storage of history information for branches. For each branch in the BTB, a two-bit counter value is stored. On each execution of the branch, this value is loaded into a counter and updated according to the outcome of the branch. If the branch is taken, the value will be incremented or kept at its maximum value (3). If the branch is not taken, the value decrements or is kept at its minimum value (0). Such a counter is called a saturation counter, as there is no overflow or underflow.

Predicting the outcome of the next execution of a branch is done by inspecting the current value of the counter stored for that branch. Whenever the counter value is 0 or 1, *not taken* is predicted; when the value is 2 or 3, *taken* is predicted.

Such a counter can also be viewed as a four-state finite-state machine (FSM) with a specific set of transitions defined for each state for a taken or not taken outcome of a branch. Some investigation went into changing the transitions to find better predictors. Nair [15] made an exhaustive search by simulating 5248 possible FSMs. From an analysis of his results, the two-bit saturation counter turned out to give the best prediction. We simulated our traces using the average over several traces of the four top-ranked FSMs. For each of these machines we also investigated the influence of the initial value, the value written into the FSM when the branch is found the first time. **Figure 6** shows this influence for T1, while **Figure 7** shows the results of our top two FSMs for all traces using the best-fitting initial values. As a reference, the three-bit history prediction from Figure 5 is added. From Figure 6 it follows that there is a considerable improvement (0.7%) when initializing the two-bit saturation counter with "3" instead of the regular "2" initialization. Such a large influence could not be found for SPEC traces.

Whenever the local-history two-bit saturation counter is referred to hereafter, it is implicit that this counter has been initialized with "3."

In Figure 7 we can see that a two-bit FSM improves branch prediction significantly. The two machines themselves do not differ much. The two-bit saturation counter is better for T2 (0.07%), almost the same for T1 and T3, and little worse (0.04%) for T4. Results are comparable to the results in [13].

• *Two-level adaptive assignment of branch direction*
Yeh and Patt [16] introduced the two-level adaptive branch predictor. As for the static predictor, the history of a branch is recorded as a series of "0" (not taken) and "1" (taken). It is not used to address a lookup table in which the prediction direction is stored as derived from statistical analysis of the workload, but rather addresses an array of two-bit saturation counters, called a *pattern history table*, where the current value of the counter determines which direction should be predicted for the branch. The counters are updated according to the outcome of the branch; in this way, the pattern table adapts itself to the behavior of the workload. If there is only a single pattern table, called the global pattern history table *PAg*[1] [10], two branches may disturb each other, so the next idea is to provide a separate pattern table for each branch *PAp*. In our case this would require 8K tables, each with, e.g., 64 entries for six history bits, a vast amount of storage hardware. However, if several branches share one table *PAs* [7], hardware cost may be reduced to a justifiable amount.

[1] *PAg*: per-address history recording and lookup in global table.

**Figure 8** shows simulation results for several combinations of history bits and instruction-address bits for T1. Zero address bits used is equal to *PAg*, and, at the other extreme, 13 address bits used is equal to *PAp*. We see that for the same number of history bits, using more instruction-address bits results in a better prediction. Looking back to the statistics of our traces, where it was found that about half of the branches occur less than four times each, it becomes clear that using more history bits will not be effective. This is in contrast to the SPEC traces, where most of the branches have a high occurrence repetition and large histories still have advantages. For a specific branch, a pattern may have occurred once and never again, so learning does not occur and thus improvements will not develop.

On the other hand, the global pattern table needs five history bits to outperform the local two-bit saturation counter described in the previous section. This more than doubles the number of storage bits required. The storage required for the adaptive predictor is the sum of the number of history bits times 8K entries in the BTB and the number of bits in the pattern tables.

A local two-bit saturation counter requires $2 \times 8K$ storage bits. For a two-level adaptive predictor, only two combinations exist (one history and eleven address bits, or two history and zero address bits) that require almost the same amount of hardware. If we double the amount of hardware, we may choose among three combinations. Doubling again will provide still other possible combinations. We selected the best-performing predictors out of each of the combinations for the three different quantities of hardware described above. The results of simulations with these predictors for all traces are shown in **Figure 9**. For T1 and T2, using a local adaptive predictor having the same hardware cost as the two-bit saturation counter results in slightly improved prediction. For T3 and T4, such a predictor behaves worse. Increasing the amount of hardware increases the prediction correctness. The predictor with hardware cost twice as high as the two-bit saturation counter behaves better for all traces, but on T3 is only 0.1% and on T1 0.5% better. Doubling the hardware once more does not produce much further improvement.

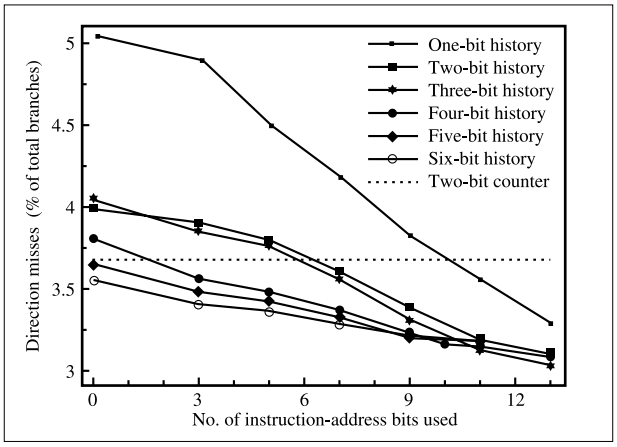


**Figure 8**

Local two-level adaptive predictor.

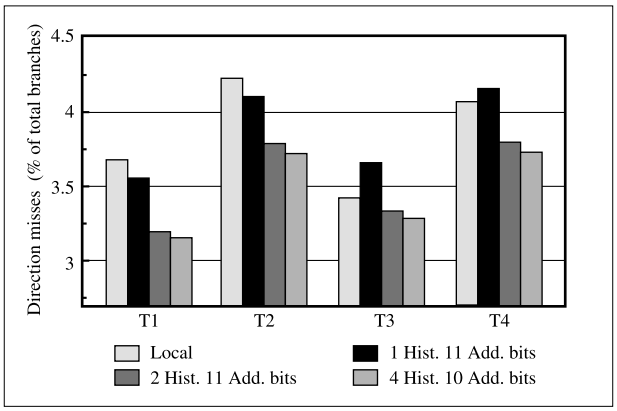


**Figure 9**

Local-history predictor using adaptive pattern table.

## 6. Prediction of branch direction based on global history

Here methods are summarized which use information about the last several branches executed to predict the direction of the next upcoming branch.

- *Branch correlation using branch direction history*

For a fixed number of BTB entries, two-level prediction may increase prediction correctness by adding more history bits and thus more storage hardware to each entry in the BTB.

Prediction schemes using global history information were proposed in [6] and [10]. Instead of recording the history for each branch separately, only one large history register exists. For each branch executed, the branch direction is recorded into this register to form a global history pattern. In other words, the path through the control-flow graph by which a certain branch is reached is used to predict the outcome of that branch. In a way similar to two-level prediction, the global history pattern is used to address a pattern table that contains two-bit saturation counters. If only one such table exists, the method is called *GAg* [10]. If several tables are used and

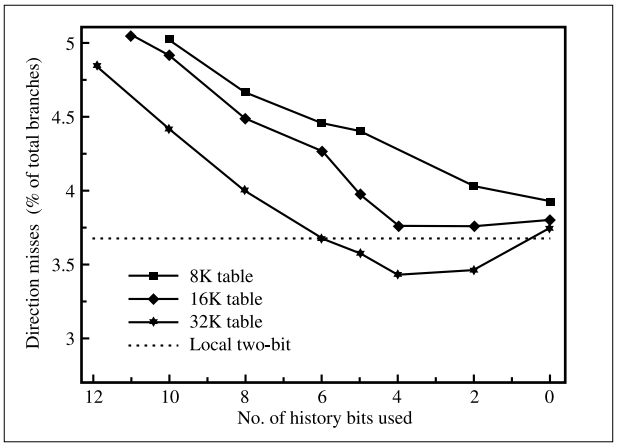


**Figure 10**

Global-history predictor using *gselect*.

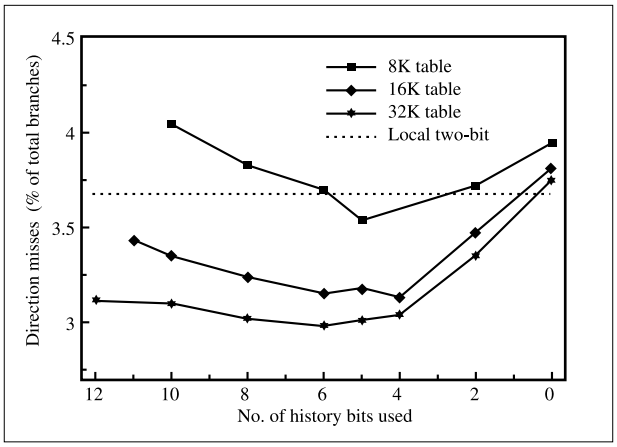


**Figure 11**

Global-history predictor using *gshare* varying history length.

the required table is selected by parts of the branch instruction address, it is called *gselect*. This collection of tables may be viewed as one big table with history bits and address bits being concatenated. **Figure 10** shows simulation results for T1 and three table sizes (the number of history bits used in the address is the independent variable).

The theoretical processor with which this prediction method should work fetches several bytes from instruction storage containing up to eight instructions per cycle. Thus, there can be several branches contained in one access, and the prediction should not be the same for all branches. (Predicting several branches in parallel using the same

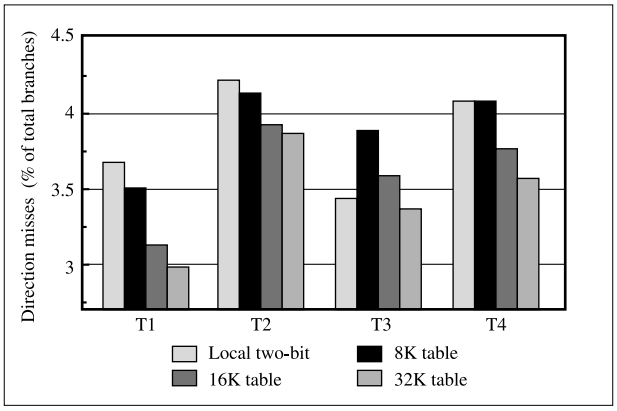


**Figure 12**

Global-history predictor using *gshare*.

history is not widely discussed in the literature.) We recommend always using *gselect* with a minimum of three address bits to permit a selection among eight entries for eight branches in parallel. From this it follows that for an 8K pattern table, a maximum of ten history bits may be used. The hardware needed to support a *gselect* with an 8K pattern table is roughly equivalent to that needed for an 8K local two-bit saturation counter. It can be seen that the prediction quality achieved with this table is always worse. A table four times larger is needed to get a better prediction. Furthermore, it can be seen that adding instruction-address information increases the quality of the prediction, provided there is enough room left in the table to allow the history to influence the results at all. This is similar to SPECmark behavior. McFarling [4] examined these phenomena and found that the pattern table is used only very sparsely. He therefore recommends that the address and the history bits be *shared* by XORing them. This method is called *gshare*.

*gshare*

We have analyzed *gshare* prediction for the three tables described in the previous section. The results for T1 are summarized in **Figure 11**. As can be seen, prediction quality is much better. The 8K pattern table already surpasses the 8K local two-bit counter. However, we still have the effect that adding history will not necessarily add prediction precision. For the 8K table, the best prediction is achieved using five bits of history, for a 16K table, four or five bits, and for a 32K table, six bits. These "best" prediction setups have been used to examine the other traces as well. The results are shown in **Figure 12**. For T1 and T2, *gshare* always outperforms the local two-bit predictor. However, for T3 only the 32K table performs

slightly better, and for T4 the 8K table performs exactly as well as the local two-bit predictor. Obviously, traces or parts of traces exist which are better predicted by local history, while others are better predicted by global history. Generally this statement also holds for branches themselves. Therefore, a combined prediction seems advantageous.

- *Branch correlation using branch path history*

In [5], Nair proposed a scheme in which the *taken*/*not taken* type of branch history from the previous chapter is replaced by a history representing the path leading to the branch to be predicted. This path history is made up of the instruction addresses of the branches in the path. Ideally, the complete address of all preceding branches should be stored. However, given a fixed size for the pattern table, a tradeoff must be made between the number of branches and the number of unique address bits per branch to be used. Nair suggested two methods:

- Concatenating some of the address bits from each branch target.
- Shifting the current history by some number of bits and then XORing the new branch target address with this shifted history to form the new history.

We have found that using the branch instruction address rather than the branch target address and using the concatenating mode did not reduce the achievable prediction correctness but was easier to implement. In using partial branch instruction address information to build the history, only low-order address bits are used. **Figure 13** shows the simulation results for trace T1 with one, two, three, and four address bits per branch being stored, respectively.

The 16K pattern table itself is accessed in a *gselect* fashion called *gpselect*[2] for global path history. As can be seen, the prediction correctness is much better than for a predictor of comparable size using direction history (dotted line). Again, the best prediction results are achieved using four to five history bits. The degradation from using more history bits is not as dramatic as it is with *gselect*, especially if three to four address bits per branch are added to the history. It appears that the pattern table is addressed much more evenly and that using a strategy like *gshare* would not achieve much additional prediction correctness. By selecting a path history formed by adding two address bits per branch, a *gshare*-like predictor called *gpshare* has been simulated.

[2] *gpselect* is a new term not found in the literature; there are very few published results that are based on path prediction.

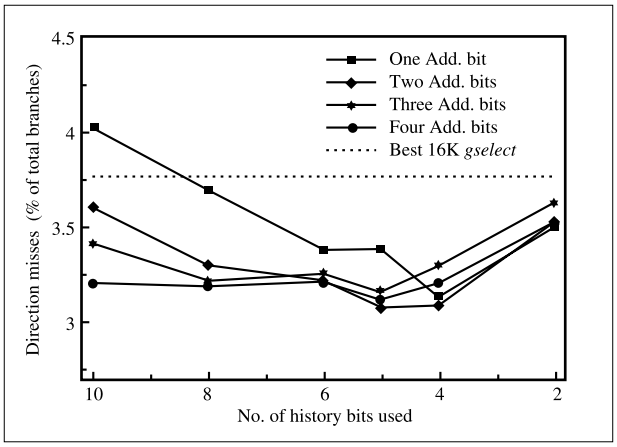


**Figure 13**

Global-path-history predictor using *gpselect*.

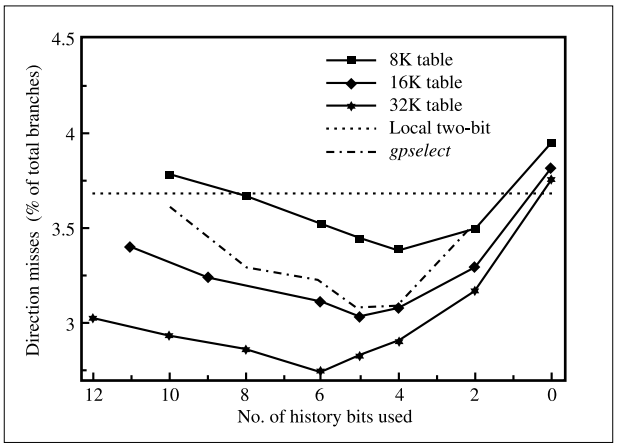


**Figure 14**

Global-path-history predictor using *gpshare* varying history length.

*gpshare*

We simulated the *gpshare* predictor for the three pattern table sizes. The result for T1 is shown in **Figure 14**. The plot for the 16K table around four to five bits of history shows almost no difference from *gpselect*, as expected. With more or fewer history bits, *gpshare* becomes better. Overall, *gpshare* outperforms *gshare* for an equivalent amount of hardware.

Taking again the best predictor configuration for each of the three table sizes and applying them to the other traces for simulation gives the results shown in **Figure 15**. It is seen that *gpshare* is the better predictor for those traces as well. However, the problem still exists that T3 is

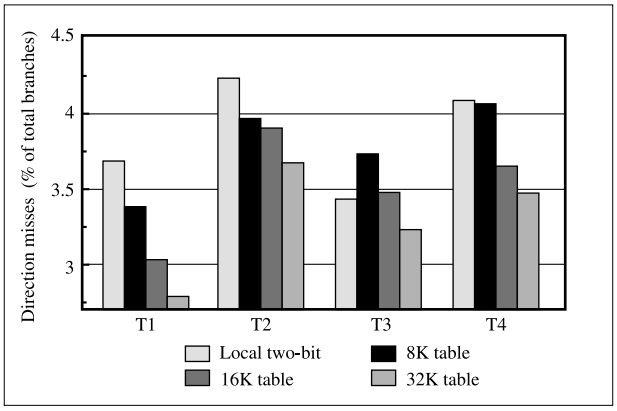


**Figure 15**

Global-path-history predictor using *gpshare*.

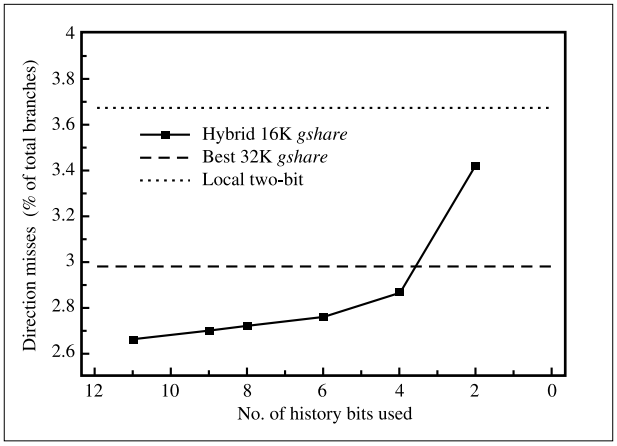


**Figure 16**

Hybrid predictor using local two-bit counter and *gshare*.

better predicted by local history. The techniques presented in the next section will address this problem.

## 7. Hybrid methods

If local and global history methods are compared, it can be seen that both result in about the same amount of misprediction. However, analysis on a branch-by-branch basis shows that some of the branches predicted badly by one method are predicted far better by the other one, and vice versa [4].

A hybrid method consists of two or more prediction methods for branch direction and a selection mechanism that chooses the probable better method for the branch to be currently predicted. Basically the selector records for each branch separately how well each method works for that branch. On the basis of this information, the decision is made as to which method to use.

The total amount of hardware needed for a hybrid method is greater than for certain local methods, because each prediction method needs its own hardware and the selector requires hardware as well. In the simulation results below, we show whether (and when) such hardware increases can be justified.

- *Hybrid predictor combining one local and one global predictor*

If the hybrid predictor must select between only two predictors, the selection mechanism can be a two-bit saturation counter that is incremented if method A has predicted correctly and decremented if method B was correct. Thus, if both are correct or both are incorrect, the value is not changed. Whenever the value of this selection counter is above 1, method A is chosen; otherwise method B is used. There are other possible selection methods, but we have simulated only this one.

To make the results comparable with respect to the amount of hardware involved, we choose as method A the local two-bit counter, and as B a global predictor using a 16K pattern table and a two-bit counter as the selector. This is equivalent to the hardware needed for a global predictor with a 32K pattern table. **Figure 16** shows the simulation results for T1 when *gshare* is used as the global predictor.

Using a four-bit history (the value for the best *gshare* prediction), only a slight improvement over the best 32K *gshare* predictor is achieved. When analyzing the *gshare* behavior, we saw that increasing the number of history bits decreased the prediction quality because several branches were predicted less well with the longer history, while only a few had better predictions. However, if with hybrid prediction those branches which behaved badly with longer history are predicted well enough with the local predictor, only those branches which do require a long history to be predicted well are left for the *gshare* predictor. This is shown in Figure 16, where prediction correctness improves with increasing number of history bits used.

**Figure 17** shows similar simulation results for T1 when *gpshare* is used. Here also, the hybrid predictor using a 16K *gpshare* surpasses the prediction quality of the best 32K predictor. It shows the same tendency as in Figure 16: using a longer history results in better prediction. Figure 13 shows that for long histories it is advantageous to store more history bits per branch, but this cannot be verified for the hybrid predictor. Storing two address bits per branch outperforms the four-address-bit case at all points.

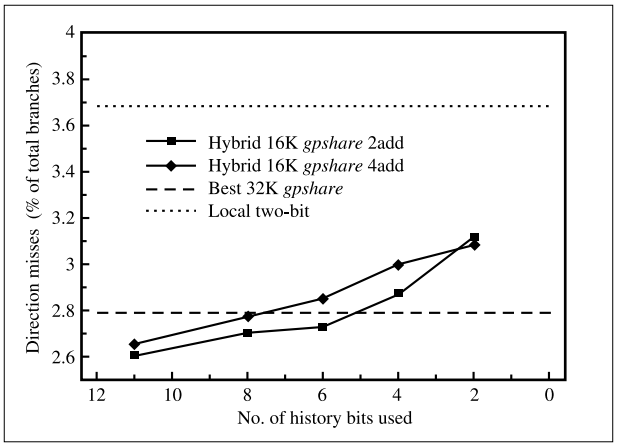


**Figure 17**

Hybrid predictor using local two-bit counter and *gpshare*.

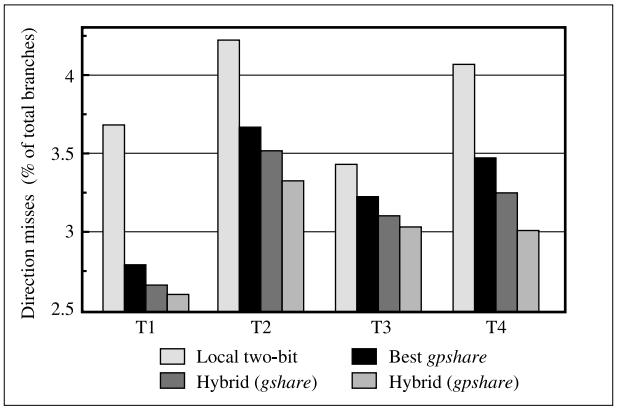


**Figure 18**

Hybrid predictor comparison for all traces.

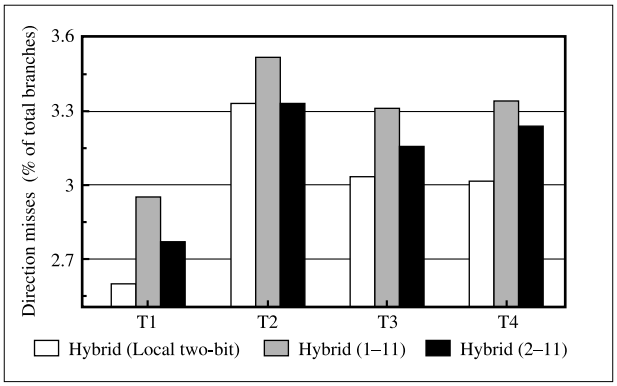


**Figure 19**

Hybrid predictor using local adaptive pattern predictor.

*gpshare* as global predictor in a two-predictor hybrid predictor outperforms *gshare* for all traces, as shown in **Figure 18**. The advantage, however, is not as great as for *gpshare* versus *gshare*. Since the method of using a path predictor as part of a hybrid predictor has not been published previously, no comparison to SPECmark results can be made here.

**Figure 19** uses the local adaptive predictor instead of the local two-bit saturation counter. Two versions were simulated, both having the same hardware cost as the other hybrid predictors. Thus, if the hardware for the local predictor increases, hardware for the global predictor is decreased. Under this boundary condition, a hybrid predictor using *gpshare* and the local two-bit saturation counter achieves the best results.

• *Hybrid predictor combining multiple prediction algorithms* Having already achieved better prediction by selecting between two predictors, in Evers et al. [3] it is claimed that choosing among multiple predictors would improve prediction quality even more. It is clear that here even more prediction hardware is required, because each predictor needs its dedicated and/or shared hardware resources *and* each predictor needs hardware to establish and store its prediction quality for each branch as well as some hardware that evaluates this information and selects the "best" predictor. While Evers [3] still uses two-bit saturation counters (one per predictor and branch), Rotenberg [17] claims that more elaborate methods for branch confidence assignment are needed.

We investigated a hybrid predictor by combining three predictors in each of four combinations: our two global predictors with either local predictor and vice versa; and the two local predictors with either global predictor. All four predictors are comparable in hardware resources but require *three* times as much hardware as the simple hybrid predictor. To select which predictor should be used, we extended the two-bit counter method from the simple hybrid predictor. In the extension, three counters are used to store the relative performance of one algorithm against those of the other two algorithms. Counter 1 is incremented if method A is better than method B and decremented for the opposite relationship. Counter 2 shows the relationship between A and C, and counter 3 that between B and C. To find which method should be used for the next prediction, two counters are examined; e.g., Counters 1 and 2 indicate whether A is the method of choice. Adding one more method will increase the

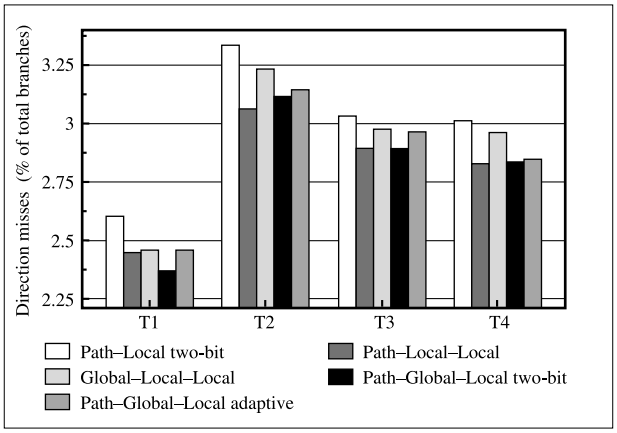


**Figure 20**

Comparing multiple hybrid predictors.

number of counters to six, as the LRU for four-way associativity requires six bits.

Besides choosing among the methods, it is likewise important to choose the method to be used after a branch is encountered the first time. This might even be crucial for the behavior of the multihybrid predictor. Whenever it was present, we selected the local two-bit counter as the starting method. **Figure 20** shows the results for the four traces. As can be seen, using more predictors does further improve the prediction. However, improvement is not consistent over the combinations chosen and the different traces. Overall, a maximum of 0.25% might be achieved, but the amount of hardware is increased dramatically for only a small reduction in missed predictions.

## 8. Moving targets

Until now we have concentrated on predicting the direction a branch should take. It was implied that the target address stored in the BTB was correct. However, this is true for only some of the branches. Some branches, such as returns from subroutines, may have different target addresses. If there is a special return instruction in the instruction set, a return stack creates the possibility of obtaining the correct return address.

There is a second group of branches, indirect branches, which commonly have changing target addresses. The target address of an indirect branch is created dynamically while the program is executed. It is used to implement case statements, jump tables, or computed go-tos. Little research has been done on correctly obtaining the target addresses of these branches. Only recently [2], a new approach was presented which uses global history patterns like those from Section 6. The number of misses resulting from changing target addresses is noticeable, and results for our implementation of this approach are detailed in this section.

- *Identified return cache*

Most of the instruction-set architectures which are used to publish results on branch prediction use specific call and return instructions, so it is not difficult to have a return stack and use it to achieve 100% correct return target addresses. However, as stated earlier, ESA/390 does not have such instructions, so we must first determine whether a branch is used as a call or as a return.

In [12] Kaeli and Emma proposed a two-stack approach to detect call/return pairs for instruction-set architectures that do not have explicit call or return instructions. A slightly different version was published in [18] for the ESA/390. Both methods require a two-stage lookup, which is not a problem as long as one is interested only in the number of misses, but it may be harmful to the overall performance of the processor. We have modified the approach by adding hardware such that a lookup for a return can be done in parallel with that in the BTB. The results presented in the following are for this new approach, which uses two independent memorizing elements:

- An *identification stack* into which each possible call is written with its target and return address.
- A *return cache* that receives new entries from the identification stack when the stored return address matches the target address of a possible return. The regular BTB entry of such a matching return gets a specific type indication which informs the structure that the target address for this entry is to be taken from the return cache.

Both storage structures may vary in size independently. **Figures 21** and **22** show the simulation results. As can be seen, prediction quality depends on stack size. Simulation was performed using a 256-entry return cache. With as few as four to eight entries, saturation is reached, with an exception for T2. For T2, saturation is achieved with 16 entries. Therefore, a 16-entry identification stack was chosen for varying the number of entries in the cache itself, as shown in Figure 22. The cache is organized as a four-way-interleaved structure. A steady improvement for all traces can be seen as the number of entries grows; the improvement rate flattens after 512 entries. Design tradeoffs may suggest a 256-entry cache, since except for T3 only a moderate 0.04% prediction correctness is lost compared to the double-sized 512-entry cache.

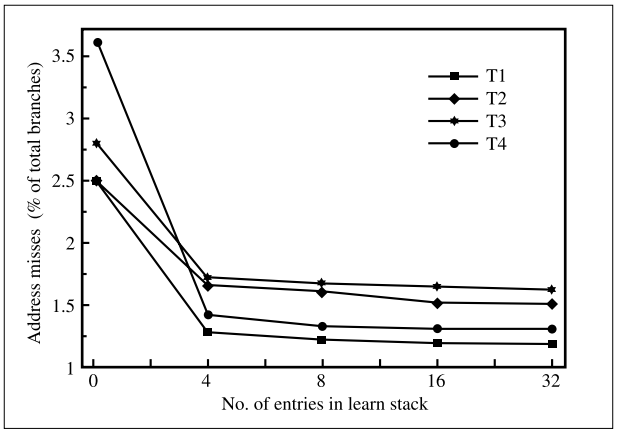


**Figure 21**

Return cache with different learn stack sizes.

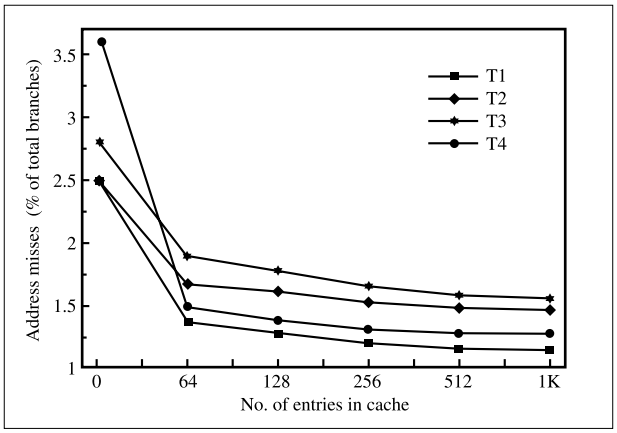


**Figure 22**

Return cache varying cache size.

- *Moving target cache*

Until the publication of Chang [2], it was common to exchange a wrong target address in the BTB for the newly calculated address. Exchanging the target had a slight advantage over leaving the BTB entry unchanged or removing that branch from the BTB.

As with adaptive prediction of branch directions, where a history pattern is used to define the entry in a pattern table which then predicts the branch outcome, Chang suggested the use of a history pattern to select the correct target address from among several target addresses stored for that branch. As in direction prediction, this method has the problem of aliasing; e.g., a given history will always point to the same entry, even for different branches. Since it is not reasonable that different branches will have the same target address just because they have the same history pattern, the target addresses must be identified more accurately. Using methods from cache design, associative sets with tagging are used to store several addresses for the same history and use the branch address as a tag to identify the correct entry. Thus, the structure is called a target cache. However, a specific branch may already have several entries in the target cache but none that matches the current history. Thus, no target can be selected.

We have changed the above approach by switching the addressing and tagging. We use the instruction address to index into the target cache in the same way as a BTB is accessed. In the target cache we use a history pattern as a tag to select one of the target addresses. If there is no perfect match for the tag, i.e., no target is stored for exactly that history, the entry whose history is closest to the current history is chosen. To enable many targets to be stored for one branch, the target cache should have a high associativity, as this will provide the maximum number of different available targets for a given branch at any time. We chose an associativity of 8, the same as for the BTB. In **Figures 23** and **24** we show the results of simulating this new approach, varying the size of the trace cache and the kind of history stored in the tag field. Except for T3, employing a target cache with 256 entries improves the address miss rate drastically. Increasing its size improves prediction for both T1 and T3. However, this increase is not dramatic, and design tradeoffs can be made, resulting in a target cache of 512 entries.

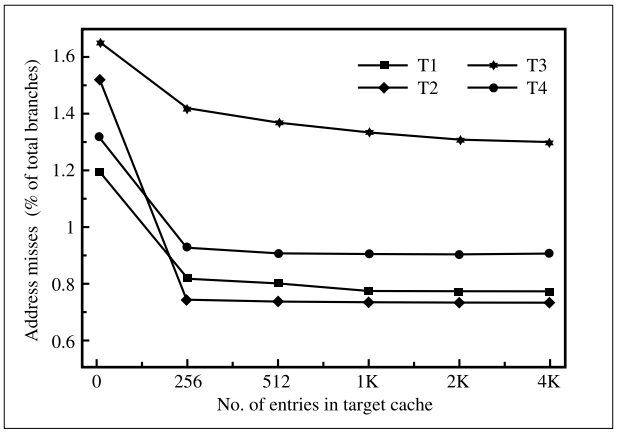


**Figure 23**

Moving target cache varying cache size.

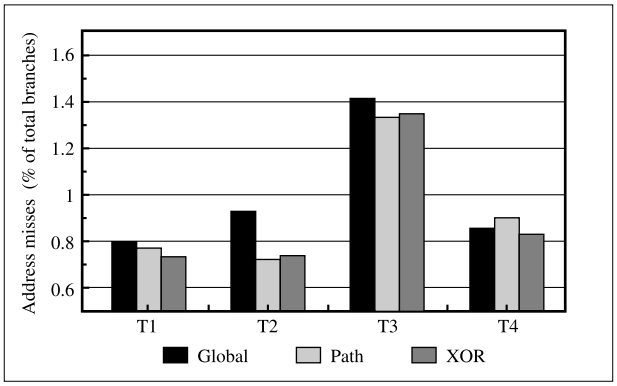


**Figure 24**

Moving target cache using different history pattern as tag.

Three different history patterns were analyzed: the global direction history, the global path history, and a combination denoted XOR. Except for T4, global branch-direction history behaves worse than global path history. Combination by simply XORing both history patterns is intended to distinguish different target addresses even when one of the two patterns stays the same. Using this combination, it can be seen that the prediction for T2 and T3 decreases only slightly (0.01%), while T1 and T4 are predicted best by the combined history pattern.

## 9. Summary

It has been shown that branch-prediction misses can be separated into three categories: table misses, direction misses, and address misses. Each category may be dealt with in another way, but reducing the number of misses always increases prediction hardware. It has also been shown that on the basis of SPECmark traces one would probably choose to invest hardware in a different way than is now suggested. Typically, BTB size falls between 1K and 4K entries. For our traces this would mean a miss rate of around 12% just due to missing branches in the table. To this, all of the other types of misses must be added. On the other hand, SPECmark traces also suggest fairly complex hybrid predictors for direction misses when long history information must be stored. Here we show that a modest two-way hybrid predictor performs almost as well as predictors involving several more algorithms. Finally, even if there is no distinct return instruction available in a specific instruction set, our approach is able to find a dominant majority of the occurring returns and predict their target addresses correctly.

The sensitivity of a processor to branch-prediction correctness depends on the microarchitecture of that processor. This sensitivity must be well understood in order to justify expending hardware for better branch prediction. However, on the basis of our simulation we suggest the following configuration for the branch-prediction unit of a processor:

- BTB with 8K entries, eight-way set-associative.
- Hybrid direction predictor with 8K two-bit local saturation counter and 16K entry-path history table.
- Return cache.

Such a configuration would yield compound miss rates for T1: 10.8%; T2: 11.4%; T3: 12.8%; and T4: 7.9%. Adding a target cache would reduce those numbers by about 0.6%, whereas doubling the BTB could reduce the numbers by more than 2% for at least two of the traces.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Standard Performance Evaluation Corporation.

## References

1. P.-Y. Chang, E. Hao, and Y. N. Patt, "Alternative Implementations of Hybrid Branch Predictors," *Proceedings of the 28th Annual International Symposium on Computer Microarchitecture*, 1995, pp. 252–257.
2. P.-Y. Chang, E. Hao, and Y. N. Patt, "Target Prediction for Indirect Jumps," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 274–283.
3. M. Evers, P.-Y. Chang, and Y. N. Patt, "Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996, pp. 3–11.
4. S. McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, Digital Equipment Corporation, Palo Alto, CA, 1993.
5. R. Nair, "Dynamic Path-Based Branch Correlation," *Proceedings of the 28th Annual International Symposium on Computer Microarchitecture*, 1995, pp. 15–23.
6. S. T. Pan, K. So, and J. T. Rameh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proceedings of the 5th International Conference on Architectural Support for Programming Language and Operating Systems*, 1992, pp. 76–84.
7. T.-Y. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors That Use Two Levels of Branch History," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 257–266.
8. C. Perleberg and A. J. Smith, "Branch Target Buffer Design and Optimization," *IEEE Trans. Computers* **42,** 396–412 (1993).
9. G. Heim, "Untersuchungen zur Vorhersage des Sprungverhaltens in modernen Prozessorarchitekturen," Diplomarbeit, Eberhard-Karls-Universitaet Tuebingen, 1996.
10. T.-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1991, pp. 51–61.
11. T.-Y. Yeh and Y. N. Patt, "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative

Execution," *Proceedings of the 25th Annual International Symposium on Computer Microarchitecture*, 1992, pp. 129–139.
12. D. R. Kaeli and P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 34–42.
13. J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Annual Symposium on Computer Architecture*, 1981, pp. 135–148.
14. J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer* **17,** 6–22 (1984).
15. R. Nair, "Optimal 2-Bit Branch Predictors," *IEEE Trans. Computers* **44,** 698–702 (1995).
16. T.-Y. Yeh and Y. N. Patt, "Two-Level Adaptive Branch Prediction," *Proceedings of the 24th Annual International Symposium on Computer Microarchitecture*, 1991, pp. 51–61.
17. E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Prediction," *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996, pp. 142–152.
18. C. F. Webb, "Subroutine Call/Return Stack," *IBM Tech. Disclosure Bull.,* pp. 221–225 (1988).

*Received July 24, 1998; accepted for publication July 1, 1999*

**Rolf B. Hilgendorf** *IBM Entwicklung GmbH, Boeblingen, Germany (hilgendo@de.ibm.com).* Dr. Hilgendorf received his diploma in electrical engineering from the University of Bremen, Germany, in 1977. In 1979 he joined IBM at the Scientific Center, Haifa, Israel, working on signal and image processing projects. Since 1986 he has been with the IBM Laboratory in Boeblingen, Germany, working in S/390 logic design. His current interests include computer microarchitecture, especially branch- and value-prediction algorithms, and performance modeling. He is a member of the IEEE.

**Gerald J. Heim** *Wilhelm Schickard-Institut fuer Informatik, Universitaet Tuebingen, Tuebingen, Germany (heim@informatik.uni-tuebingen.de).* Mr. Heim received his diploma in computer science from the University of Tuebingen, Germany, in 1996. Since then he has worked as a research scientist in the Department of Computer Engineering at the Wilhelm-Schickard-Institut. His research interests include CPU and computer architecture as well as computer and performance modeling.

**Wolfgang Rosenstiel** *Wilhelm Schickard-Institut fuer Informatik, Universitaet Tuebingen, Tuebingen, Germany.* Dr. Rosenstiel received his diploma and his Ph.D. in computer science from the University of Karlsruhe, Germany, in 1980 and 1984, respectively. Since 1990 he has been Professor for Informatics (Computer Engineering) at the University of Tuebingen and Director of the Department of System Design in Microelectronics at FZI. His research interests include high-level synthesis, hardware/software codesign, computer architecture, parallel computing, and neural nets.