An Analysis of Branch Predictors for the

Photo Editor Program

Ashish Sabharwal and Mike Swift
January 29, 1999

Abstract

In this paper wedescribe an experiment that measures effectiveness of a correlating branch predictor for the Microsoft Photo Editor program running on a Pentium II Xeon . The experiments include a variety of cache layouts, with different numbers of entries, numbers of prediction bits, and lengths of correlation histories. Overall, these experiments demonstrated that even a small dynamic branch predictor was much better than the simple “predict backward branch taken” predictor. In addition, they also demonstrated that for this program a very small predictor causes almost the same improvement as a large predictor.

Introduction

For this experiment, we used the Visual Etch tool to instrument the Microsoft Photo Editor program. This program was chosen because of its modest size (the core program is 814 k) as well as it function: modifying images. It was hoped that this program would demonstrate a different mix of instructions than non-graphical code and that it might use recent additions to the Intel x86 instruction set, such as the MMX instructions. The branch predictor implemented used a subset of the address bits and the recent global history of branches taken to index into a table of predictors, could be either one or two bits.

Methodology

The Microsoft Photo Editor program was used to load a 400x591 bitmap, which was then resized to 800 x 1182. The bitmap was then sharpened with the sharpening tool, and then converted to a “Chalk and Charcoal” drawing with another tool. The program was then terminated without saving. The program was driven by hand, using keyboard shortcuts.

The program was studied with the Etch tool. This tool runs in two passes: the first one discovers and instruments the code, and the second runs the program and collects data. There were 18 modules used in Photo Editor, and in most of them 97-99% of the code was discovered. In **mside.dll** , a module used for keyboard input, only 76% of the code could be discovered.

Branch Predictor

The branch predictor was parameterized by several values:

- Number of branch addresses
- Ignored bits of branch address
- Number of bits of global history
- Number of bits of prediction

The table was implemented as a linear array of bytes of size (Number of branch addresses) x ($2^{\text{Number of bits of global history}}$). Initially, all entries in the table were initialized to a special value indicating that they did not contain a valid predictor.

During execution, at each branch instruction the instrumentation code recorded the next sequential address and whether or not the target branch address was ahead or behind the current program counter. To implement the predictor, it indexed into the predictor array by masking off part of the program counter and shifting it to add the global history. That byte contained the prediction bits for this jump, and the address of that byte was remembered so that after the jump the bits could be updated.

If the value of those bits was the initial value, than a static predictor was used. While this was not strictly necessary, it did help gauge what the impact of making a wrong prediction the first time through was. The static predictor simply predicted taken for all backwards branches and not taken for forwards branches.

If the predictor bits were valid, then they were compared with a threshold value, which was equal to $2^{(\text{Number of prediction bits} - 1)}$. If the value was greater than or equal to this threshold, then the branch was predicted to be taken. Otherwise, the branch was predicted to be not taken.

On the next instruction after a branch, the effectiveness of the predictor was judged. Counters of correct and mis-predictions were kept. In addition, the prediction bits were updated on every branch. The bits were incremented for every branch taken and decremented for every branch not taken, up to the maximum or down to the minimum value.

When the Photo Editor program terminated, the instrumentation code printed out its parameters as well as the total number of backwards and forwards branches taken or not taken, the number of correct and mis-predictions, and the number of branches statically and dynamically predicted.

Results

The branch predictor was run in nine configurations to determine the effects of the different parameters. Table 1 lists the parameters for each run.

Run Label	Table Size	History Size	Prediction Bits	Ignored Address bits
4096-4-2-0	4096	4	2	0
4096-4-2-3	4096	4	2	3
4096-4-1-3	4096	4	1	3
4096-0-2-3	4096	0	2	3
1024-4-2-3	1024	4	2	3
512-4-2-3	512	4	2	3
64-4-2-3	64	4	2	3
64-1-2-3	64	1	2	3
64-1-1-3	64	1	1	3

Table 1

The performance of the static predictor was similar for each run (within 0.01%), so the chart below shows the performance of the static predictor compared to all the dynamic predictors. The parameter values for each run were chosen to demonstrate independently the effect of each parameter as well as to demonstrate the effect of a far simpler predictor. Table 2 contains the actual number of predictions made in a sample run. Static predictions are the predictions made using a static predictor because the branch predictor table did not have a value, whereas dynamic predictions are those made based on previous experience with the same branch.

Property	Value
Run Label	4096-4-2-0
Table Size	4096
History Size	4
Prediction Bits	2
Ignored Address bits	0
Forwards Taken	22729312
Forwards Not Taken	28086432
Backwards Taken	55063206
Backwards Not Taken	5066935
Correct Predictions	105092818
Mis Predictions	5853067
Static Predictions	15387
Dynamic Predictions	110939521

Table 2

The prediction rates for the experiment are shown in Figure 1. Several things can be learned from this chart. First, the dynamic branch predictors performed much better then the simple static scheme, especially on forward branches. Second, some of the cache parameters had no influence. For example, varying which bits of the branch instruction address were used for lookup had no impact, and this is probably due to the variable-length instructions of the x86 architecture. As result, instructions in general have no special alignment on word boundaries. Second, for this particular program, varying the size of the predictor table had very little impact. This is probably due to a large number of loops with fairly few conditional branches. This program layout would lead to only a small number of branches having a large impact on overall predictor performance.

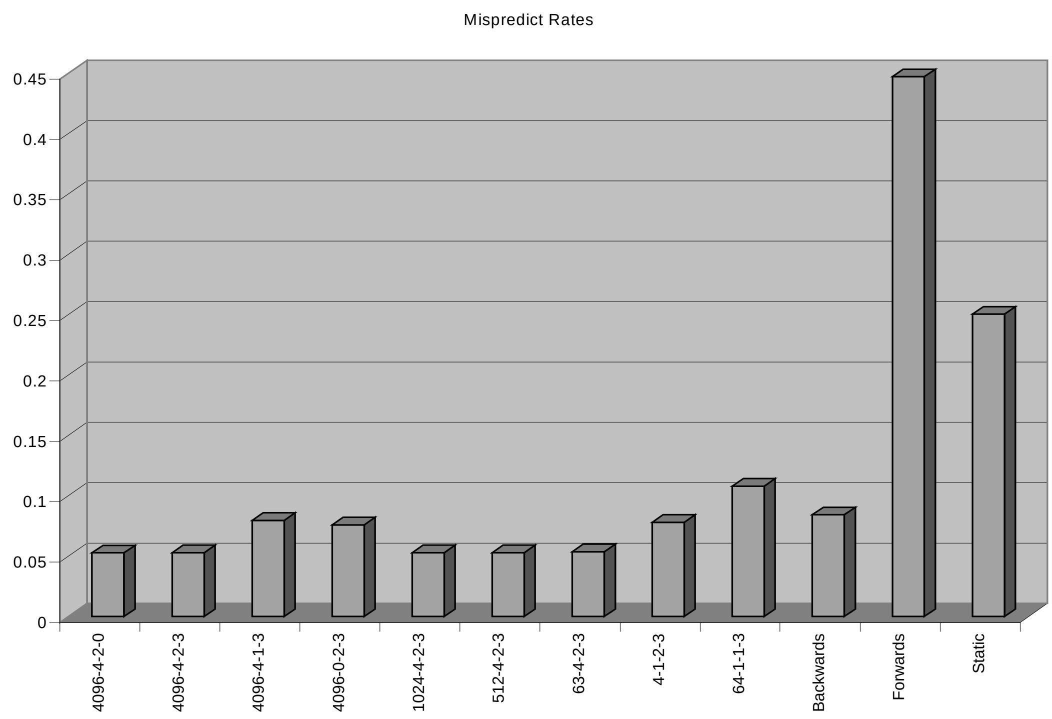

Figure 1

The factors that most impacted predictor performance were the size of the global history and the number of bits of prediction. Changing from two bits to one bit of prediction and from four bits to zero bits of global history worsened the mis-prediction rate from 5% to 8%. This same effect can be seen when the size of the table is shrunk – performance did not degrade until the number of prediction bits or history bits were also degraded. Both of these predictor elements aid in correctly predicting the first pass through a loop. Hence, for loops taken a small number of times, a sizeable improvement would be expected.

These results are probably not indicative of overall performance of these predictors – rather, they are highly dependent on the nature of the Photo Editor program, which performs a large number of similar operations when manipulating bitmapped images. The mis-prediction rates for a 4096 table, 2 prediction bit predictor on the Spec89 benchmarks are between one percent and 18 percent, which were widely different then those seen here.