An Intelligent Wrong Path Issue Scheme
A method for improving performance

Meghana Sardesai Chidiogo Madubike Nmita Sarna

Department of Electrical Engineering
University of Houston, Houston, Texas
{msardesai, cmadubik, nsarna}@mail.uh.edu

Abstract

Predicting branches has increasingly become an area of interest due to its effect on the performance of processors. Various methods have been proposed to speculate the path of an instruction stream after a branch. One such scheme is issuing down the other path (the non-predicted path or the wrong path) as well as the predicted path when a branch is encountered. This paper questions this technique and proposes to eliminate issuing the wrong path at all times. In fact, issue the wrong path, but do so" intelligently". This technique may give birth to an approach that maximizes performance and is optimal. We use selected benchmark programs from SPEC2000.

1. Introduction

Branch prediction is used to solve the problems of limited fetches that are imposed by control hazards in order to exploit instruction level parallelism (ILP). Clearly, ILP is the key ingredient for processors to mask execution latency. The goal is to have efficient branch prediction techniques that result in high prediction rates and in turn have better performance and thus resulting in more instructions being committed per cycle (IPC). This paper focuses on implementing a scheme that issues the wrong path of a branch "intelligently" based on parameters that represent the prediction of that particular branch.

1.1 Paper Overview

Section 2 covers our motivation and hypothesis of this paper. The detail of the architecture is presented in section 3. Section 4 describes the experimental methodology employed in our proposed scheme. We analyze our experimental results in section 5 and draw our conclusions in section 6.

2. Motivation and Hypothesis

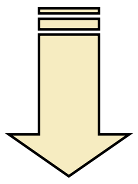
When executing the wrong path in conjunction with the predicted path, the execution units execute instructions that may or may not be used. For example, in the case of correct prediction, the wrong path that has also been issued is simply not used. On the other hand, in the case of incorrect prediction, the predicted path is invalidated and the wrong path that has already been issued is used. This is done every time a branch is encountered and it is clearly beneficial in the event of a wrong prediction. The question that arises at this point is how good are branch predictors? Today, branches are predicted with an accuracy that reaches the high 90 percentile [1]. And the motivation behind the question leads to another question; when braches are predicted with such good accuracy, then why execute the wrong path simultaneously with the predicted path for *every* branch prediction? In fact, we hypothesize that the execution of the wrong path should only be done when the branch is mispredicted often and not when the branch is regularly predicted correctly. This means that the functional units are not executing useless instructions that will invariably be flushed from the pipeline.

On one hand, this scheme will work at its best when enough independent instructions are found that can be executed in parallel. On the other hand, this scheme will also perform well when the behavior of the branch is highly predictable, despite the lack of parallelism. In this latter case, knowing the behavior of the branch is beneficial in deciding whether or not to issue the wrong path. Figure 1 below shows a trace of the branch instruction BLEZ within a particular execution window. This instruction has been extracted from other branch instructions to observe its behavior within the program. A 1 denotes correct prediction while a 0 stands for an incorrect prediction.

…………1 0 1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 0 1 0 1 1

0 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1…………….



BLEZ instruction.

…………01101110 11111111111111111111111……….

Figure 1: Program trace of BLEZ

3. Architecture Details

A bimodal branch predictor is employed to use various past instances of a branch behavior to predict the next instance. The fields of the branch target buffer (BTB) in the baseline architecture include the branch address, branch opcode and the branch target address. Each branch address indexes the BTB for its corresponding target address. An additional field in the BTB is now added that represents the branch threshold value of a particular branch (see figure 2 below). This value is initialized to be 20 for each branch. In addition, we included confidence values in the predictor which contain the following information (the importance of each field is explained in the next paragraph):

- Increment Value for each Correct Prediction (IVCP): this value is set to be 1
- Maximum Value of Branch Threshold (MVBT): 30
- System Threshold Value: 28 is the optimum value determined via simulations
- Prediction Penalty for a Wrong Prediction (PPWP): this value, 4, is chosen to be greater than the difference between the maximum value and the system threshold

When a branch address indexes the BTB, two pieces of information are output. The first information is the branch target address and the second is the branch threshold value for that particular branch instance. This branch threshold is compared against the system threshold value. If the branch threshold value is greater than the system threshold value of 28, this means that the branch is predicted correctly most of the time. In this case, the wrong path is not issued. Conversely, if the branch threshold value is less than or equal to the system threshold value of 28, this means that the branch prediction is not very accurate and the wrong path is also fetched and issued. We have tried using lower system threshold values but it was found from our simulations that these lower values lead to worse performance. This is because we are now setting an easier constraint to be reached by the branch and in some cases the wrong path is not issued when it should have been. Recall, in the baseline architecture, the wrong path is issued with every prediction despite the accuracy of the prediction.

The branch threshold value is updated on the resolution of the branch instruction. If the prediction was found to be correct then the branch threshold is incremented by IVCP. On a wrong prediction, the branch threshold value is decremented by PPWP.

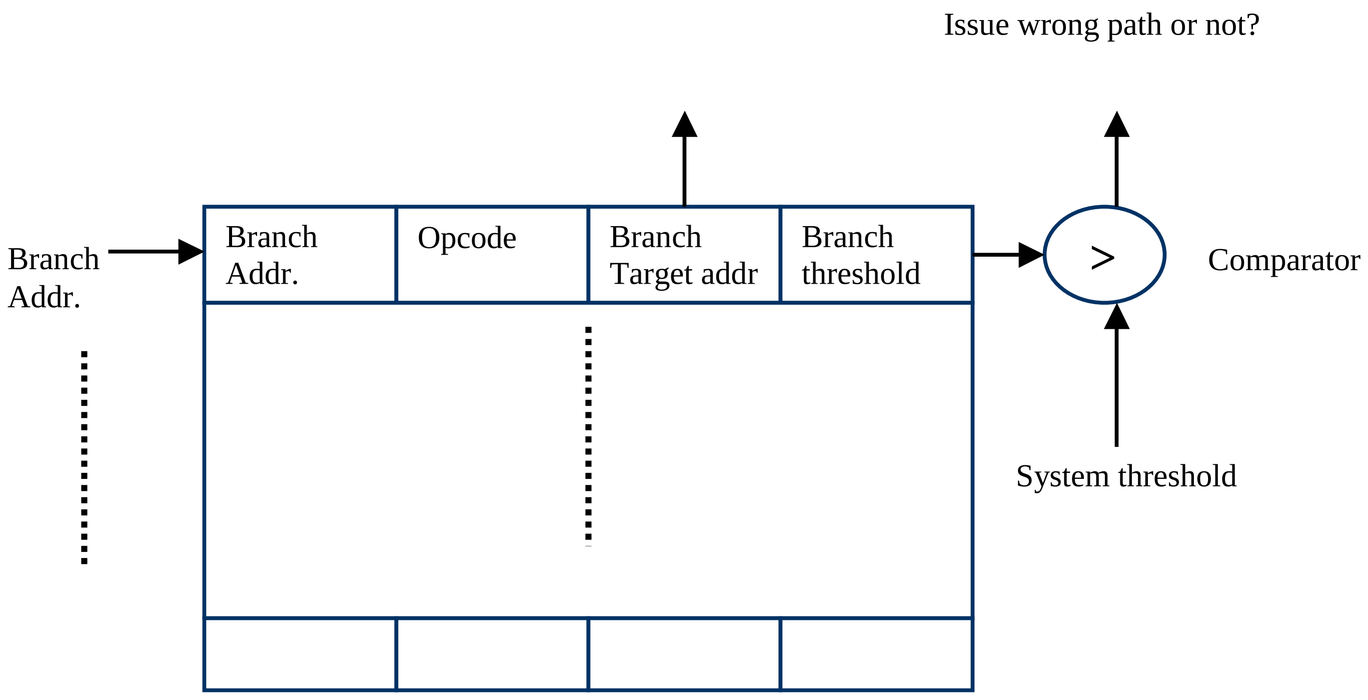


Figure 2: BTB Entries

4. Experimental Methodology

Simulations were run on three SPEC2000 reduced benchmarks namely, Equake, mcf and Parser. Equake is a floating point program that does simulation of seismic wave propagation in large basins. Mcf is an integer program that performs combinatorial optimization and single-depot vehicle scheduling while Parser, an integer program, does word processing and is a bit more parallel than Equake and Mcf.

The baseline architecture was simulated with these three programs and the clock cycles per instruction on average (CPI), IPC and execution bandwidth was recorded. The results are summarized in table 1 below.   The programs were then run on our modified architecture using two different sets of parameters. The first set included 4 FPU adders, 4 integer ALUs, 1 integer MULT/DIV unit and 1 FPU MULT/DIV unit. The results obtained are summarized in table 2.

Benchmark	**CPI**	**IPC**	**Exec BW**
Equake	0.7175	1.3938	1.5320
Mcf	1.4201	0.7042	0.7672
Parser	0.6121	1.6338	1.9959

Table 1: Baseline architecture with first set of parameters

Benchmark	**CPI**	**IPC**	**Exec BW**
Equake	0.7181	1.3927	1.4303
Mcf	1.4250	0.7018	0.7466
Parser	0.6036	1.6567	1.7259

Table 2: Modified architecture with first set of parameters

Equake suffered 0.11% degradation in IPC from the baseline architecture while Mcf degraded 0.24%. Parser, in contrast, showed a 2.29% improvement in the instructions commited per cycle on average.

The second set of parameters included 3 FPU adders, 3 integer ALUs, 1 integer MULT/DIV unit and 1 FPU MULT/DIV unit. These results are shown in table 4 while the baseline architecture results with the same set of parameters are shown in table 3.

Benchmark	CPI	IPC	Exec BW
Equake	0.7225	1.3840	1.5256
Mcf	1.4174	0.7055	0.7700
Parser	0.6228	1.6057	1.9657

Table 3: Baseline architecture with second set of parameters

Benchmark	CPI	IPC	Exec BW
Equake	0.7231	1.3829	1.4204
Mcf	1.4271	0.7007	0.7450
Parser	0.6146	1.6272	1.6877

Table 4: Modified architecture with second set of parameters

With the second set of parameters, Equake's degradation remained the same while Mcf's degraded to 0.48% for IPC. Parser performed slightly worse than before but still showed a 2.15% improvement.

5. Experimental Analysis

Our results from both sets of parameters show that for programs having a little parallelism like Parser, improvements in IPC can be seen despite the number of functional units available. At the same time the execution bandwidth decreased. For Parser we observed up to 28% reduction in execution bandwidth, 10% for Equake and 2% for Mcf. This means that useless instructions from the wrong path are not executed and are replaced by useful instructions.

We feel our algorithm will work best with programs that are highly parallel because more instructions can be issued and executed in parallel. Therefore, to some degree, our results prove our hypothesis; moreover, using our algorithm on highly parallel programs may additionally support our hypothesis. We will run our algorithm using such programs and show results in future papers.

6. Conclusions

Issuing down the wrong path “intelligently”, on parallel programs, shows performance gain. This can be seen with Parser despite it being not too highly parallel. Further more, the complexity involved in always issuing and executing the wrong path, especially in the case of a correct prediction, is reduced.

Our simulation results show execution bandwidth reduction for the programs. This implies that usage of resources for useless instructions was reduced with a minute penalty of less than 0.5% degradation in IPC.

The effect of the number of execution resources on one hand implies that the more resources available, the less the IPC degradation. On the other hand, fewer resources available do not show dramatic performance degradations. Therefore, this proves that useful instructions can be allocated with our scheme when the number of resources is increased. However, we feel that the effect of fewer functional resources on IPC require the need for extra simulations. Unfortunately, due to time constraints, these simulations could not be run.

In conclusion, the degree of success of this scheme is highly depended on the amount of parallelism available in an application. Even though Parser is not highly parallel, it still showed improvements, we can imagine how our algorithm will perform for programs that are even slightly more parallel which is the incentive for further future work on this topic.

Finally, our algorithm can be modified to keep up with the changes in the increasing lengths of pipelines in today’s modern processors.

7. References

[1]: Swanson, McDowell, Swift, Eggers, Levy. "An evaluation of Speculative Instruction Execution on Simultaneous Multithreaded processors", submitted for publication.

[2]: Johnny K.F. Lee, Alan Jay Smith, "Branch Prediction Strategies and Branch Target Buffer Design", Computer, January 1984.

[3]: Joel Emer, Nikolas Gloy. "A Language for Describing Predictors and its Application to Automatic Synthesis", Proceedings of the 24th International Symposium on Computer Architecture, June 1997.

[4]: Eric Rotenberg, Steve Bennet, James E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", Proceedings of the 29th International Symposium on Microarchitecture, December 1996.

[5]: Daniel Holmes Friendly, Sanjay Jeram Patel, Yale N. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism", Proceedings of the 30th International Symposium on Microarchitecture, November 1997.