Lab Report - Lab 2

TDTS08 - Advanced Computer Architecture
HT17
Linköping University

December 3, 2017

Authors

Name	**SSN**	**Liu-ID**
Kim Larsson	940826-3615	kimla207
Niklas Nilsson	920727-0654	nikni459

Contents

1 Introduction

The purpose of this lab is to give us as students an insight on how pipelines work in terms of computer architecture and how different algorithms on branch prediction can improve the performance of these pipelines [1].

2 Part 1: Pipeline basics I

We consider a processor with the six pipeline stages: IF (instruction fetch), DA (instruction decode), CO (calculate operands addresses), FO (fetch operands), EI (execute instruction), WB (write results). We also consider two instructions going through the pipeline, the LB instruction with displacement addressing (i.e. offset) and the ADD instruction.

The two instructions would then go through the pipeline with the following results:

LB instruction

1. IF: Fetch the LB instruction from memory
2. DA: Determine the op-code and the specified operands. In this case the op-code is 0x20, meaning we are using displaced addressing.
3. CO: Calculate the effective address, meaning the address in memory where the operand is stored.
4. FO: Fetch the operand from memory
5. EX: Not used since this is a ”basic” instruction that does not execute any other instruction.
6. WB: Write byte to a general purpose register (GPR)

ADD instruction

1. IF: Fetch the ADD instruction from memory.
2. DA: Determine the op-code and the specified operands. In this case the op-code is 0x40, meaning we use addition with overflow check.
3. CO: Calculate the effective address, meaning the address in memory where the operand is stored.
4. FO: Fetch the operand from memory

5. EX: Add values and check for overflow.

6. WB: Store the sum in specified general purpose register (GPR).

3 Part 2: Pipeline basics II

Here we are given two problems to solve assuming a two-stage pipeline (fetch and execute). The first problem is given in [2] and tells us to redraw the diagram shown so that four instructions are executed using the two-stage pipeline for four instructions. The solution to this problem can be seen in table 1.

Instruction/Time	**1**	**2**	**3**	**4**	**5**
Instruction 1	FI	EI			
Instruction 2		FI	EI		
Instruction 3			FI	EI	
Instruction 4				FI	EI

Table 1: Two-stage pipeline with four instructions

The next problem is to use the original problem with one addition, the second instruction is a conditional jump and we assume that the taken predictor is implemented in the system. We also assume that the wrong prediction is made and the result of this action can be seen in table 2.

Instruction/Time	**1**	**2**	**3**	**4**	**5**
Instruction 1	FI	EI			
Instruction 2		FI	EI		
Instruction 42			FI		
Instruction 4				FI	EI

Table 2: Two-stage pipeline with four instructions and branch

The difference in the two diagrams is the branch instruction in table 2. Since the prediction turned out to be wrong and the branch was not taken the fetching of instruction 42, that would have been the instruction branched to, was therefore discarded. If the prediction had been correct, however, we would have had earned a time unit giving us a total of four time units instead of five.

4 Part 3: Branch prediction

When dealing with branching in a pipeline it is needed to use an algorithm in order to try to predict if a branch is taken or not. If a prediction is wrong it would end up as in the prior problem that can be seen in table 2, the potential of the pipeline would be wasted. In this part we will use sim-outorder to evaluate the effects of different branch predictors.

4.1 Branch predictors

The ”Always taken” and ”Perfect” predictors simple works as their name states. The former always predict a branch to be taken and the later always predict the correct branching behavior. The other more complex ones are described below with the

4.1.1 Bimodel

The ”Bimodel” predictor uses a two-bit counter to keep track of the latest branches taken, and not taken, by each branch instruction. Each branch instruction has its own counter that increments if a branch is taken and decrements if branch is not taken. These counters are saturated, which means that they wont overflow when reaching 3 or 0. The most significant bit is used to make the branch prediction based on the branching history of that specific instruction. In other words if a branch is often taken it is predicted to be taken again, and vise versa.

4.1.2 2level (local)

The ”2level” predictor uses several of the counters used by the bimodel predictor alongside with a history table. The history table keeps track of the recent n branches of the instructions mapped to a specific history entry. Each history entry is indexed by the low-order bits of the branch address, meaning that several branch instructions uses the same history entry. If we have a history entry consisting of the history for the last n branches, then we would also have an array consisting of 2^n counters, one for each possible history content. As an example: say that a history entry’s content is (0010), this means that the counter indexed as number 2 would be used when predicting the outcome of the next branch. One clear disadvantage here is that if several instructions are mapped to the same history entry this might cause problems due to the difficulty of pattern recognition, which is the core concept of the 2level predictor.

4.1.3 Combined

The ”Combined” predictor is, as one can tell from the name, a combination of predictors. The combined predictor uses a 2-bit saturated counter to keep track of which of the predictors that have the best accuracy for the branches that share that counter. This results in an accuracy that is at least as good as the best of the two predictors used.

4.2 Results

The metrics used for this part is the total simulation time in clock cycles alongside with the branch address-prediction rate, i.e. addr_hits/updates. The results of the tests can be seen in table 3.

Algorithm	**prediction ratio**	**# clock cycles**
Always taken	0.3164	82 335 864
Bimodel	0.8247	55 565 715
2level (local)	0.7294	59 930 331
Combined	0.8257	55 310 616
Perfect	1	46 345 442

Table 3: Simulation results for different prediction algorithms

The ”Perfect” predictor is obviously the best predictor which does not need to be commented. The ”Always taken” predictor shows that more than half of the branch instructions are not taken resulting in a low prediction of 0.3164.

The ”Bimodel” predictor shows that the branching behavior is the same for a large portion of the branch instructions with a few irregularities due to the high prediction ratio of 0.8247. The ”2level” predictor uses, as mentioned before, a history table in order to find patters of when branches are taken/not taken. Notice that the prediction ratio of this predictor were lower than for the ”Bimodel” predictor, it was 0.7294. This concludes that the irregularities that appeared in the branching behavior resulted in patterns that was hard to find. This can be because of several reasons. To many instructions might have used the same counter and disturbed each other or the design of the program might have been structured so that the branch instruction’s behavior were seemingly random. This gave the ”Bimodel” an advantage since it only traces the behavior of the most recent branches that seems to have been more or less the same.

As predicted it can also be seen that the ”Combined” predictor had a ratio of at least the best of the two predictors used. In this case the best predictor was the ”Bimodel” predictor. As the ”Combined” predictor had a prediction rate of 0.8257,

that is better than the ”Bimodel” on 0.8247, it can be concluded that at least some patterns were found by the ”2level” predictor which made it a better predictor in some cases. Even though those cases were not at all many.

4.3 Conclusion

By using the data from table 3 it can be concluded that the worst predictor was the ”Always taken” predictor and the, clearly, best was the ”Perfect” predictor. Table 4 shows the performance of all the predictors compared to the worst predictor, the ”Always taken” predictor, both in terms of prediction ratio and number of clock cycles. Notice that a higher prediction ratio is better and a lower number of clock cycles is better.

Algorithm	**prediction ratio**	**# clock cycles**
Always taken	100 %	100 %
Bimodel	260.7 %	67.5 %
2level (local)	230.5 %	72.8 %
Combined	261.0 %	67.2 %
Perfect	316.1 %	56.3 %

Table 4: Speed-up ratios compared to the worst branch predictor

References

[1] Lab Assignment 2: Instruction Pipelining,
`http://www.ida.liu.se/ TDTS08/labs/lab2.shtml`

[2] Lab Problem 2,
`http://www.ida.liu.se/ TDTS08/labs/labprob2.shtml`

[3] WRL Technical Note,
`http://www.ida.liu.se/ TDTS08/labs/pipeline/WRL-TN-36.pdf`