Advanced Computer Architecture

LAB1 – Get familiar with Simplescalar and Wattch

Deadline: 2024/12/20

Student : M11215075, 胡劭

1 Introduction

You need to have access to a Unix/Linux computer to be able to perform the laboratory exercises. All installation instructions refer to Linux distributions, but they may work on Sun/Solaris or PC/Solaris installations as well.

※ Note. Server IP: 140.118.115.163

The account and password to access the server is your student ID number. You may change the password using the following command: passwd <yourID i.e Old password><New password>

2 Objective of the laboratory exercise

After this laboratory exercise, you should:

- Understand how program behavior (instruction class profiles) relates to branch prediction efficiency.
- Be able to make tradeoffs related to branch-predictors' implementation.
- Understand the role of compiler optimizations for performance.

3 Branch-prediction assignments

3-1 Program behavior (Instruction profiling)

Run the benchmarks from the Mibench website on the profiling simulator with the test input data set to find out the distribution of instruction classes. Note that even with the test input data set the execution times are quite long and you can limit the instructions executed using - max:inst flag to SimpleScalar. A suitable limit is 200 million instructions. For some experiments below (e.g. studying optimizations) it is not meaningful to limit the number of simulated instructions so for these experiments you need to execute the whole program with the test input data set.

Table 1: Benchmark programs versus instruction class profiles.

Benchmark	Opt level	Loads	Stores	Uncond. branch	Cond. branch	Int	fp
susan	O4	4314(15.78%)	5004(18.30%)	753(2.75%)	4444(16.25%)	12804(46.83%)	0(0%)
qsort(large)	O3	583(7.93%)	3507(47.65%)	166(2.26%)	818(11.12%)	2274(30.92%)	0(0%)
qsort(small)	O3	583(7.94%)	3497(47.60%)	166(2.26%)	818(11.13%)	2274(30.95%)	0(0%)
Basicmath(large)	O3	1183593990(16.48%)	679404205(9.46%)	306373369(4.27%)	993474742(13.83%)	3986383489(55.5%)	33205902(0.46%)
Basicmath(small)	O3	31658398(17.56%)	18450634(10.23%)	8030802(4.45%)	24801193(13.76%)	94354792(52.33%)	2987292(1.66%)
CRC32	O3	363(5.73%)	3377(53.35%)	120(1.9%)	632(9.98%)	1831(28.93%)	0(0%)
rijndael	O3	645(8.2%)	3503(44.55%)	172(2.19%)	904(11.5%)	2631(33.46%)	0(0%)

Fill Table 1 with all available benchmark programs versus instruction class profiles. Choose three of the benchmarks for your further assignments based on the following considerations:

Q3-1.1 Is your benchmark memory intensive or computation intensive?

Ans: Memory intensive.

Q3-1.2 Is your benchmark mainly using integer or floating point?

Ans: Integer.

Q3-1.3 We will now use the branch prediction simulator (sim- bpred) to investigate the effects of branch predictors on the execution of the benchmark you chose. This simulator allows you to simulate 6 different types of branch predictors. You can see the list of them by looking at the menu 'branch predictor type' for the branch prediction simulator sim- bpred. For three of the possible branch prediction schemes 'nottaken, taken, bimod ', run the simulation for your benchmark as you did above and note the branch prediction statistics for each.

※ **Note.** The simulator statistics are for all branches both conditional and unconditional (which are regarded as predicted correctly). For this reason the reported prediction rates for taken and nottaken do not add to 1. Use the branch- direction measurements and number of updates, both corrected for unconditional branches to calculate accuracy (hit rate for conditional branches). Fill in the information in Table 2.

Table 2: Prediction accuracy and CPI

Benchmark	nottaken		taken		bimod		perfect	
	Pred. rate	CPI	Pred. rate	CPI	Pred. rate	CPI	Pred. rate	CPI
susan	0.3587	1.6959	0.3587	1.6770	0.9265	1.0853	1	1.0102
qsort(large)	0.2165	2.1459	0.2165	2.1323	0.8415	1.8464	1	1.7413
qsort(small)	0.2165	2.1683	0.2165	2.1562	0.8425	1.8928	1	1.7726

Q3-1.4 For the four branch prediction schemes ' nottaken, taken, perfect, bimod ', describe the predictor (bimod is the 2- bit prediction scheme). Your description should include what information the predictor stores (if any), how the prediction is made, and what the relative accuracy of the predictor is compared to the others. Fill in the values for other benchmarks with the help from other groups.

Ans: nottaken:

- No stored information needed
- Always predicts "no branch"

taken:

- No stored information needed
- Always predicts "branch"

bimod (2-bit predictor):

- Uses 2-bit counter for each branch state
- Predicts based on previous branch behavior
- Better accuracy than taken/nottaken due to dynamic adjustment

perfect:

- Always predicts correctly

- Theoretical best case
- Used as performance baseline

Relative accuracy: perfect > bimod > taken/nottaken

4 In-order versus out-of-order

Now you will conduct experiments to find out how the increase in the parallelism in processing instructions affects the CPI of your processor, and how you can improve the performance of memory reference instructions. In all experiments you will use the default cache and branch predictor configurations.

4-1 In-order and out-of-order issue.

Experiment with the width of the pipeline by running the simulation with the following combinations of parameters. Measure the CPI and total the number of cycles. Pipeline stage width (fetch, decode, issue).

- Pipeline width 1, in-order and out-of-order execution (out-of order execution is default, in-order must be selected explicitly, see the SimpleScalar manual for instructions on how to do this).
- Pipeline width 4, in-order and out-of-order execution.
- Pipeline width 8, in order and out of order execution.

Fill in Table 3 for your chosen benchmark, with cycle count (sim_cycle) and CPI from different pipeline width and issue strategy.

Benchmark：____susan_____.

Table 3: In- order versus out- of- order.

	Pipeline width 1		Pipeline width 4		Pipeline width 8	
	in-order	out-order	in-order	out-order	in-order	out-order
sim_cycle	56162	52838	47221	29674	45746	27840
CPI	2.0541	1.9325	1.7270	1.0853	1.6731	1.0182

Q4.1 What is the influence on CPI on the increase in available of pipeline width and with different execution strategy?

Ans: Increasing pipeline width and using different execution strategies have significant impacts on CPI:

Pipeline Width Effect:

- Wider pipelines (4,8) achieve lower CPI compared to narrow pipelines (1) due to increased instruction-level parallelism (ILP)
- However, wider pipelines come with higher hardware complexity and power consumption costs

Execution Strategy Impact:

- Out-of-order execution consistently achieves lower CPI than in-order execution
- This is because out-of-order execution can:
  - Better handle data dependencies
  - Utilize instruction parallelism more effectively
  - Hide memory latency by executing independent instructions during stalls

Combined Effects:

- The best CPI performance is achieved with wide pipelines and out-of-order execution
- However, this configuration also represents the highest implementation complexity and cost

5 Branch-prediction assignments using Wattch

Use the wattch module to estimate the power consumption of four branch prediction schemes ' nottaken, taken, perfect, and bimod '. Please use the same benchmarks as you used in assignment 4, and fill in the blank in Table 4.

Table 4: Estimate power consumption.

Benchmark	nottaken		taken		bimod		perfect	
	Total power	Branch power	Total power	Branch power	Total power	Branch power	Total power	Branch power
susan	3328675.1538	209728.3247	3291632.4438	207394.3915	2130242.9804	134219.2527	1982933.5986	124937.7973
qsort(large)	1133033.1253	71388.5039	1125854.3055	70936.1913	974883.7256	61424.0565	919391.4487	57927.6798
qsort(small)	1143801.3549	72066.9729	1137412.2053	71664.4146	998430.2545	62907.6419	935041.2759	58913.7213

Q5.1 In your opinion, please describe the influence on branch power for different branch prediction strategies.

Ans: The accuracy of prediction higher is, the branch power lower is.

Accuracy of prediction: nottaken < taken < bimod < perfect

branch power: nottaken > taken > bimod > perfect

The reason is:

1. Higher prediction accuracy means fewer mispredictions
2. Fewer mispredictions means less pipeline flushes and re-execution
3. Less unnecessary operations leads to lower power consumption

6 Compiler optimizations

6-1 Effects of compiler optimizations

Investigate how various compiler optimizations affect cache performance for data and instruction accesses. Use simulation to measure effects of optimization and fill in Table 5 for two programs, one integer and one with floating point calculations.

Q6-1 What are the differences between different compiler optimizations? Why?

Ans: 1. Differences between compiler optimizations:

Based on experimental results:

- For susan (integer program):
  * O2 achieves best CPI (1.0801)
  * O1 and O4 show higher CPI (1.0859 and 1.0853)
  * Instruction count slightly increases with higher optimization
- For basic_small (floating point program):
  * O2 and O4 show identical results
  * O1 has slightly better CPI than O2/O4

* Very small variations between optimization levels

2. Why:

- Different optimization levels find different balance points between:

  * Instruction count

  * Execution efficiency

  * Code organization

- Performance depends on:

  * Program characteristics (integer vs floating point)

  * Memory access patterns

  * Code structure

- O2 sometimes performs best because:

  * Better balance of optimizations

  * Avoids potential overhead from aggressive O4 optimizations

  * More suitable for programs like susan with mixed computations

Table 5: Effect of compiler optimizations.

Compiler optimization	**Program: susan**			**Program: basic_small**		
	#Instructions	#cycles	CPI	#Instructions	#cycles	CPI
-O1	27325	29672	1.0859	180306824	184215215	1.0217
-O2	27328	29518	1.0801	180302852	184466574	1.0231
-O4	27342	29674	1.0853	180302852	184466574	1.0231

7 Conclusions

At the end of this laboratory exercise, you should be able to answer following questions:

Q7.1 Why is the bimodal branch prediction more expensive to implement than predict nottaken?

Ans: Because:

1. It requires additional hardware components including:

- 2-bit saturating counters for each branch.
- Branch prediction table to store prediction history.

2. It needs more complex control logic for updating prediction states.

3. It consumes more chip area for storage elements. While nottaken only needs simple static prediction logic with minimal hardware requirements.

Q7.2 Why is the bimodal better than nottaken?

Ans: Because:

1. It adapts to program behavior by learning branch patterns:

- For loops, it can learn to predict taken for loop branches.
- For conditional code, it can learn typical paths.

2. Its 2-bit counter provides hysteresis, preventing prediction changes due to occasional mispredictions.

3. Nottaken fails particularly in loops, always predicting exit when the loop typically continues.

4. Dynamic prediction (bimodal) outperforms static prediction (nottaken) in most real programs.

Q7.3 Which branch prediction has more expensive- cost of energy to implement than others? List your reasons for detail.

Ans: A: Perfect branch prediction has the highest energy cost because:

1. It requires the most complex hardware to achieve 100% accuracy:

Must analyze future instruction flow

Needs extensive look-ahead logic

Requires largest prediction storage

2. Compared to other predictors:

Nottaken: Lowest energy (just static decision logic)

Bimodal: Moderate (only needs 2-bit counters and prediction table)

Perfect: Highest (complex analysis circuits and storage)

Q7.4 What, if any, is the influence on CPI by allowing more instructions to be processed in one cycle?

Ans: Allowing more instructions to be processed in one cycle generally reduces CPI, as more instructions are completed in fewer cycles. However, the extent of the reduction depends on several factors:

1. Instruction-Level Parallelism (ILP):

If the program has high ILP, more instructions can be executed concurrently, significantly reducing CPI. However, limited ILP in many real-world applications diminishes the impact.

2. Pipeline Stalls and Hazards:

Data dependencies (RAW, WAR, WAW), branch mispredictions, and cache misses can cause pipeline stalls, reducing the effectiveness of parallel instruction processing.

3. Hardware Resource Limitations:

If there are insufficient functional units or memory bandwidth to handle the increased demand, resource contention can negate some CPI improvements.

4. Diminishing Returns:

Increasing parallel instruction processing yields diminishing CPI benefits as certain portions of a program are inherently sequential (Amdahl's Law).

Q7.5 Do compiler optimizations affect CPI and if so why?

Ans: Yes, compiler optimizations can affect CPI because they modify the instruction mix and execution patterns. For example:

1. Instruction-Level Parallelism (ILP):

Optimizations like loop unrolling or instruction reordering increase parallelism, allowing more instructions to execute simultaneously. This can reduce CPI by better utilizing hardware resources.

2. Instruction Complexity:

Optimizations such as vectorization replace multiple scalar instructions with fewer but more complex vector instructions. While this reduces the total number of instructions, it may increase CPI as vector instructions typically have longer execution latencies.

3. Cache Behavior:

Optimizations like data locality improvements reduce memory access delays (e.g., by improving cache hit rates). This minimizes stalls, potentially lowering

CPI.

4. Branch Prediction Impact:

Optimizations that reduce the number of branches (e.g., by eliminating conditional branches or converting them to arithmetic operations) can lower branch misprediction rates, reducing pipeline flushes and thus CPI.

If you have any problems, please contact with the TA.
E-mail: D10602805@mail.ntust.edu.tw (Walle Haileeyesus Engdaw)