CSCE 692 – Design Principles of Computer Architecture

Lab 2 – Processor Simulations Using SimpleScalar

Second Lieutenant David Crow – ENG/20M

**Summary**

In this lab, I test different configurations of on-chip resources, processor sizes, and performance optimization techniques. I do so within a Linux distribution capable of running SimpleScalar, a program that simulates the performance of microprocessors. This program allows me to vary the width of a processor, the branch prediction strategy, the order of instruction execution, and the number of arithmetic logic units, multipliers, and memory ports. To efficiently run multiple tests, I employ a few shell scripts to automate consecutive SimpleScalar simulations; each script then outputs results to several files.

In my analysis of the results, I evaluate the relative speed of each simulated processor. As expected, I find that processors with more available resources and better optimization techniques (e.g. a more sophisticated branch prediction strategy) perform at a higher level.

**Methodology**

Results were obtained using the SimpleScalar simulation tool. I downloaded SimpleScalar 3.0 onto a virtual Ubuntu 16.04 installation running within Parallels 13 on OSX Mojave and then built a target for a portable instruction set architecture. After verifying the correctness of the installation (by comparing a set of test output data to a set of expected output data), I created a few shell scripts to automate testing. All shell scripts can be viewed in Appendix.zip.

In processor\_width.sh, I pass the file’s input – namely, the width of a processor, the order of instruction execution, and the number of arithmetic logic units (AMUs), mulitpliers, and memory ports – to SimpleScalar. After redirecting both stdout and stderr to their own files, I compare the simulation output to the expected output and store the results of the diff in a third output file.

The file processor\_width\_repeater.sh allows me to run processor\_width.sh multiple times with different input values. When I execute the script in processor\_width\_repeater.sh, twenty-four stderr, stdout, and diff files are generated – one of each for each of the eight different simulation configurations.

The files branch\_predictor.sh and branch\_predictor\_repeater.sh behave similarly to processor\_width.sh and processor\_width\_repeater.sh, respectively. However, these files allow me to test different configurations of branch prediction strategy and order of execution instruction (as opposed to testing the processor width, number of ALUs, etc.). They, too, generate several output files for later analysis.

**Results**

Results are summarized in the tables below. Table 1 shows the results of experiment 1, in which I evaluated four different configurations of available resources for both in-order and out-of-order instruction execution. For each of the eight trials, I listed the cycles per instruction (CPI) as given in the SimpleScalar output. I also computed the relative speedup, where *relative* means *relative to a processor of width 1 with in-order execution*, for each of the eight trials. Relative speedup was calculated using Amdahl’s Law – as given on page 49 of Hennessy and Patterson (2019) – which says

Of course, SimpleScalar does not give a simulation’s execution time as output; instead, I compared CPI values. For example, I calculated the relative speedup for the second row of Table 1 in this way:

Note, however, that CPIs are not the only way to approximate execution time. SimpleScalar, among other values, outputs the total simulation time in cycles for a given simulation. We can also use this value to approximate the execution time. For the second row of Table 1,

We should expect either CPI or total simulation time in cycles (or both) to effectively approximate execution time, and the fact that the relative speedup for the two metrics is equivalent over all configurations is reassuring.

Table 2 shows the result of a similar experiment. However, instead of varying the width of the processor, the number of ALUs, the number of multipliers, and the number of memory ports, I varied the branch prediction strategy (to maintain consistency, I used a processor of width four with three ALUs, two multipliers, and two memory ports). Again, I tested each configuration once with in-order execution and once with out-of-order execution. CPIs were given by SimpleScalar, and relative speedups were calculated exactly as in Table 1.

**Table 1: CPI and relative speedup when varying the width of the processor, the order of execution, and the number of ALUs, multipliers, and memory ports**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Machine Width | Integer ALUs | Integer Multipliers | Memory Ports | Execution Order | Cycles Per Instruction | Relative Speedup |
| 1 | 1 | 1 | 1 | In order | 1.4873 | 1.0000 |
| Out of order | 1.4663 | 1.0143 |
| 2 | 2 | 1 | 1 | In order | 1.3308 | 1.1176 |
| Out of order | 0.8340 | 1.7833 |
| 4 | 3 | 2 | 2 | In order | 1.2890 | 1.1538 |
| Out of order | 0.5941 | 2.5035 |
| 8 | 6 | 3 | 2 | In order | 1.2829 | 1.1593 |
| Out of order | 0.5426 | 2.7411 |

**Table 2: CPI and relative speedup when varying the branch prediction strategy and order of execution**

|  |  |  |  |
| --- | --- | --- | --- |
| Branch Predictor | Execution Order | Cycles Per Instruction | Relative Speedup |
| Taken | In order | 1.8943 | 1.0000 |
| Out of order | 1.1694 | 1.6199 |
| Perfect | In order | 1.2591 | 1.5045 |
| Out of order | 0.4608 | 4.1109 |
| Not taken | In order | 1.8971 | 0.9985 |
| Out of order | 1.1770 | 1.6094 |
| 2 level | In order | 1.2909 | 1.4674 |
| Out of order | 0.6459 | 2.9328 |

**Discussion**

It’s clear that, in all cases, out-of-order execution is significantly faster than in-order execution for an otherwise identical processor. Over these two experiments, out-of-order processors are, on average, faster than their equivalent in-order processors. This should not be a surprise. Out-of-order execution seeks to limit wasted instruction cycles by executing instructions when data and execution resources become available. In contrast, in-order execution executes instructions sequentially, so the processor may sit idly while waiting for data and/or resources. By removing a significant amount of this idle time, we allow for a faster processor.

Furthermore, Table 1 shows us that more resources allow for a faster processor. If we maintain the order of instruction execution, increasing the number of available resources from one row in the table to the next (that is, comparing a width 1 processor configuration to a width 2 processor configuration, a width 2 processor configuration to a width 4 processor configuration, etc.) improves the processor’s speed by on average. Although this is not as significant as switching from in-order to out-of-order instruction execution, it’s still a marked improvement; in my opinion, the first experiment shows that it is always worth considering the more resources/higher cost tradeoff.

Finally, Table 2 illustrates that some strategies for branch prediction are faster than others – at least for the simulated configurations. Specifically, *two-level* prediction is faster than *taken* branch prediction, which is itself ever-so-slightly faster than *not taken* branch prediction. As expected, *perfect* branch prediction is the fastest strategy of the four tested. However, it’s notable that two-level prediction is as fast as perfect prediction, so, clearly, branch prediction strategies are nearing their maximum possible performance.

**References**

Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach* (6th ed.). Cambridge: Morgan Kaufmann.