Lab 2: Exploring the Impact of Branch Prediction and Instruction-Level Parallelism (ILP) on Processor Performance

Task 1: Evaluating the base configuration

*Goal.* In this report the goal is to understand the branch prediction in order to overcome major obstacles and achieve higher performance for a high-performance CPU. An accurate branch prediction is required in order to achieve this performance but is limited in the different branch predictors. We will also see that it is the key to many techniques for enhancing and exploiting ILP. This lab is a comparison on different dynamic branch predictions and a static branch prediction schema.

Finding a base configuration based on the Lab 1 was difficult however after some time we decided to do testing with both the optimal solution we found for qsort but also a different approach with the 512 cache size, 8 block size and 8 associativity. Here are the results:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Application | dij |  | 2-level | comb | “not-taken” |  | Bimodal/double | Comb/double | 2-level/double |
| Dijkstra | 5.4351 | 6.0082 | 5.4345 | 5.4320 | 6.7618 |  | 5.4352 | 5.4320 | 5.4352 |
| Qsort | 10.3433 | 12.0878 | 10.0443 | 10.0402 | 13.1292 |  | 10.3433 | 10.0402 | 10.0397 |
| Stringsreach | 16.0798 | 17.5117 | 16.1518 | 16.0959 | 5.9811 |  | 16.0811 | 16.0959 | 16.2073 |
| Gsm-untoast | 2.6478 | 2.6474 | 2.6531 | 2.6495 | 3.2940 |  | 2.6477 | 2.6494 | 2.6500 |
| Jpeg-cjpeg | 3.7628 | 4.2048 | 3.7786 | 3.7681 | 4.5827 |  | 3.7627 | 3.7680 | 3.7809 |

Table 1: Base

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Application | bimodal | perfect | 2-level | comb | “not-taken” |  | Bimodal/double | Comb/double | 2-level/double |
| Dijkstra | 6.3433 | 6.9157 | 6.3427 | 6.3402 | 7.6688 |  | 6.3433 | 6.3433 | 6.3433 |
| Qsort | 19.1803 | 20.6540 | 18.8266 | 18.8245 | 21.6767 |  | 19.1802 | 19.1803 | 19.1803 |
| Stringsreach | 16.2369 | 17.6611 | 16.3026 | 16.2537 | 17.722 |  | 16.2382 | 16.2369 | 16.2369 |
| Gsm-untoast | 2.6622 | 2.6616 | 2.6675 | 2.6638 | 3.3080 |  | 2.6620 | 2.6622 | 2.6622 |
| Jpeg-cjpeg | 4.6628 | 5.0928 | 4.6748 | 4.6657 | 5.4810 |  | 4.6627 | 4.6628 | 4.6628 |

Table 2: Base configuration for base3.txt for <512><8><8>

The results are somewhat surprising since both of them show a better performance in combined and bimodal branch prediction models. The perfect one does not however show any signs on improvement. With that being said, once we double the size of the best performing model (bimodal) we realised that it yet again gained in performance in CPI. This should have to do with the fact that for some of the programs, the branch prediction is so accurate that it does not stall the process and has to wait due to high missing prediction rate/direction prediction rate. Of course, this is dependent on the programs again we see that the string search and quicksort is the most affected since they have memory operations affecting the runtime and misses. Missing the prediction means flushing the memory and changing the prediction. Referring to the reference paper.

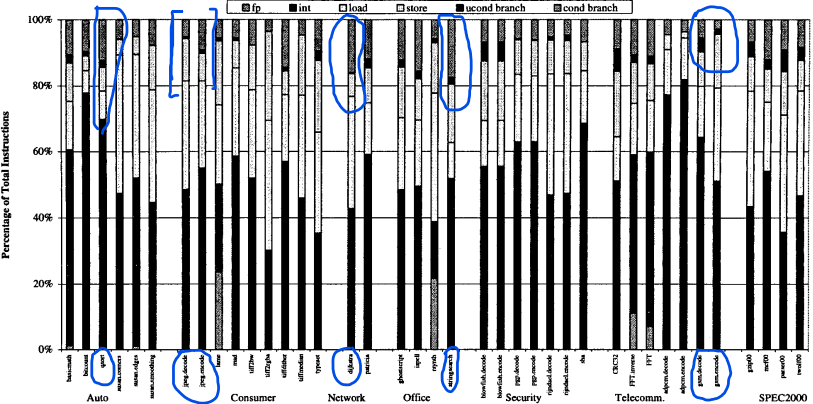
Every one of them acts different in regard to IC (highest for jpeg encode/decode), floating-point and memory (load and store), control etc. 

Figure 1. Dynamic Instruction Distribution for large data set.

Here we see that branch prediction can have a large impact in the application is forced to constantly workloads and stores. The difference between the static and dynamic vary in performance and programs gain a boost but it’s not that evident in all cases. The program that benefits from it mostly are also the two most affected. Bimodal (doubled L1 and L2) as well as combined for the programs that do indeed have more unconditional and conditional branches. Usually, on modern CPUs the branch prediction cycle is between 1 and 20 cycles. There are four control flow instructions as mentioned unconditional branch, call/return, conditional branch taken, and conditional branch not taken.

The branch variation in the different branch predictors was not at all varied, for cases such as Dijkstra’s algorithm the optimization was already achieved with just implementing the bimodal, one needn’t go further.

The branch prediction in this lab is very important, it has been an important role in the single pass process of data. Branch prediction enables the processor to execute instructions long before they are established to be true. Not only have we looked at the branch prediction but also jump prediction.

A modern processor may predict that, say an actual function, is the same as in the previous call and starts executing before pointer is pointing to the actual function. If the jump prediction was wrong, then the CPI is increased even further.

This concludes, bimodal branch prediction can be optimised by utilizing more branch history. However, the main advantage we seen that static branch prediction has been the simplified complexity of the architecture. Lastly, we have seen that static branch prediction is worse in performance since the dynamic branch prediction has analysed the history of access and made realistic prediction in the history table. Also, we see that the optimization with implementing the branch prediction is a bit better but needs some time to “warm up”.

Task 2: Optimization of ALU, Mult/Dividers and conclusions

Conditional and unconditional branch instructions remain as a critical issue of scalar processing. About 20% of the dynamic instructions are branches. By taking advantage of the profile information, the IMPACT-I instruction placement has reduced the percentage of the dynamic taken branches from 67% to 35%. This improves the efficiency of the instruction pipeline and the instruction cache/buffer. Meanwhile, we showed that 92% of the dynamic branches can be predicted correctly at the compile time. Therefore, supercomputer architectures can significantly improve their scalar performance by taking advantage of the compile-time branch prediction.