Conlusions for LAB1, LAB2, and LAB3

In the three labs, I got a taste of different levels of optimization: I worked on assembly level optimization in lab1, and microarchitecture level optimization (processor and cache) in lab2 and lab3.

Algorithm optimizations

The three labs didn't involve alogrithm level optimization. An efficient algorithm can reduce running time of a program in order of magnitudes, which neither code nor microarchitecture can achieve.

However, when we are stuck with a program's performance, it is always hard to come up with an algorithm with lower time complexity, and it turns out that a lot of problems are in NP (i.e., problems that have no efficient algorithms). That's why we need lower level optimizations to reduce the constant factors.

Code optimizations

In lab1 I dealt with a mtrix multiplication problem, and my task was to optimize the assembly code as mush as possible. The mostly-used optimization methods were to (1) use registers as much as possible to reduce memory access (2) use shift to replace multiplication (3) try to reduce branch/jump (e.g., by loop unrolling, using `swtich-case` instead of `if-else`), etc. For matrix multiplications, we could also apply blocking to fully utilize cache and registers. There are many other code-level optimizations, and it turns out that this level of optimization can improve performance greatly, even if the algorithm was unchanged. (For example, in lab1, with some techniques of code-level optimization, the cycle number of the program was reduced from 70909765 to 2315602, about 30 times faster).

However, there still exists drawbacks. For example, the long latencies of some instructions were unavoidable, and it is also tedious to optimize assembly code for every program. So we need a lower level of optimization for hardware, which can hide implementation details to software programmers while providing performance enhance.

Microarchitecture optimizations

In lab2 and lab3, I dealt with hardware-level optimization involving processors and caches.

Lab2 introduces many methods for processor optimization: pipelining, forwarding and hazard-detection, OoO, etc. For single processor, implementing a 5-stage one can greately reduces cycle numbers compared with a single-stage one. Adding forwarding units, hazard-detection units and branch-prediction units further optimize the performance, and also save software programmers from tedious code optimizaition. Meanwhile, the design complexity and cost of power and energy also increases. There also exists the risk of hardware attacks that take the advantage of these hardware units (e.g., BPU attacks utilize BPU to detect changes of branch patterns and hack secret bits).

Lab3 introduces different cache replacement policies and skewed-associative caches. Different cache replacement policies fit different program patterns, and during the design we also need to take into acount the hardware cost to implement a certain policy. For example, PLRU is a good approximation of LRU and the former saves bits overheads of the latter. Skewed-associative caches deal with conflicts by allowing two ways of a set to have two distinct hash functions, that is, the set an address can be mapped to is decided by both the address and the way it chooses. It turns out that cache can at least double the performance (according to the data I collected from lab3). However, cache implementation increase power and energy cost, and to find a good replacement policy increase design complexity.