Final Report on Instruction-level-parallelism in the Fast Fourier Transform Program

Xuan Zhao
Advanced Computer Architecture Coursework 1
Nov 4 2021

I. Preliminary Experiments

A. Vary the RUU size between 2 and 256, looking at the `sim_IPC` metric, the number of instructions completed per clock cycle, plot a graph showing the result. Explain what you see.

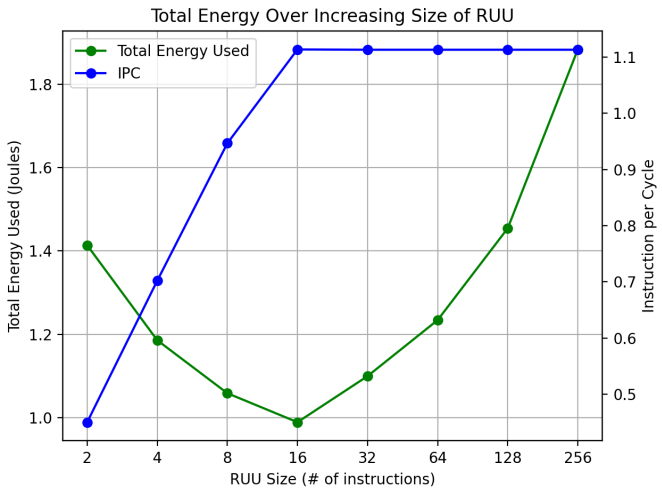

Fig. 1. Total Energy Consumption Estimate and IPC value Versus RUU Size.

The average instructions per cycle count increased from 0.4505 to 1.1133 when the RUU size increases from 2 instructions to 16 instructions. This is because a larger RUU will prevent more stalls. When pipeline hazard, particularly data hazard, is present, the subsequent instructions in the RUU need to wait to be issued until the operands in previous instructions have been resolved and instructions are executed and committed. If the RUU size is too small and the pipeline hazard occurs frequently, the RUU will be completely filled and new instructions will have to be stalled before entering the RUU. Hence, larger RUU will prevent such stalls and increase average instructions per cycle. This can be corroborated by the fact that the value of `ruu_full` is really high when RUU is small and decreases drastically when RUU size increases from 8 to 16 to 32 instructions. (See Fig. 2)

When the RUU size goes from 16 instructions to 32 instructions, `sim_IPC` slightly drops, then plateaus at 1.1128 from 64. The plateau is due to the fact that RUU becomes unnecessarily large while other configuration does not change. When the pipeline hazard is present, the subsequent instructions have to wait for the pipeline hazard to be resolved. However, the process of resolving operands of previous instruction will only

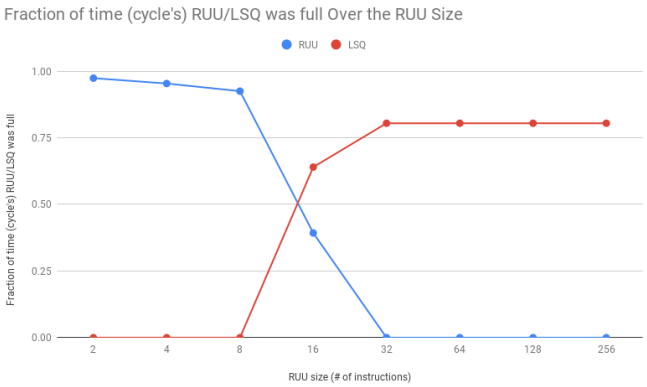

Fig. 2. Fraction of time (cycle's) RUU/LSQ was full Versus the RUU Size

take finite amount of cycles, thus previous instructions will not stay in the RUU indefinitely. In other words, each instruction will leave the RUU as soon as its operands are known, thus there is an optimal RUU size which accords with the pipeline hazard (e.g. a data dependency) that needs the longest cycles to resolve. In the case of the given fast Fourier transfer program, the RUU size of 16 instructions is the optimal size so that there is enough space in RUU to resolve the pipeline hazard with the longest time (or the most cycles) to be resolved.

In addition, the `sim_IPC` value becomes greater than 0 because the architecture supports the issue and commit multiple instructions within a cycle. (4 instructions per cycle by default)

B. Now do the same but look at the total energy consumed: plot the graph and explain what you see.

The total energy consumption goes down from RUU size of 2 instructions to RUU size of 16 instructions, then it goes up since then.

The decreasing phase actually matches the increasing phase of `sim_IPC` value (See Fig. 1): both steadily increase/decrease when RUU size changes from 2 instructions to 16 instructions. This is due to the reducing number of stalls described in the previous answer. Since the IPC value is high, it takes less time to run the entire program and thus consume less energy, despite the fact that larger RUU means more transistor and larger multiplexer. However, the increasing phase of energy consumption is precisely caused by the stagnation of the improvements of `sim_IPC`. Therefore,

larger RUU at this point will not lower the run-time but only add more energy consumption.

C. What micro-architectural structure is limiting the simulated execution speed when running this application in the default simulated architecture? Under what circumstances is the RUU size? Under what circumstances is the LSQ size?

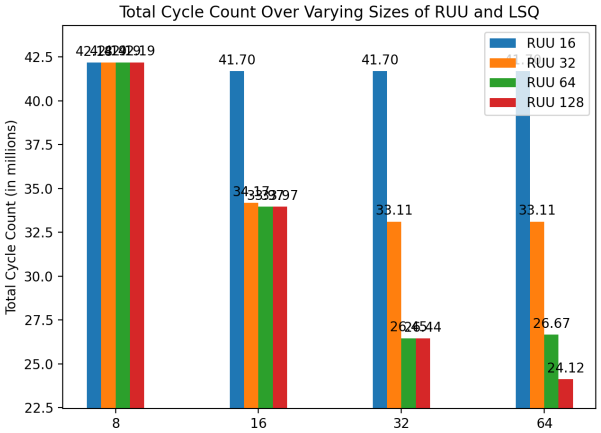

Fig. 3. Total Cycle Count Versus Various LSQ and RUU size (in number of instructions)

In terms of execution time, increasing the RUU alone will not reduce the total cycle count if the LSQ size is too small. The reason is that larger RUU will increase the IPC and effectively dispatch more instructions per cycle on average. If load and store frequently appear in the program, the amount of load and store instructions, or memory operations, needed to be processed and waits in the LSQ per cycle on average will increase. If the LSQ size is too small, the frequent load and store will eventually clog up the LSQ queue and create stalls, which will not improve execution time until the LSQ size gets reasonably large. The same holds true if the LSQ size is large but the RUU size is small. This phenomenon can be confirmed by data in Fig. 2, in which the size of RUU and LSQ constrains each other's performance by exhausting each other's space.

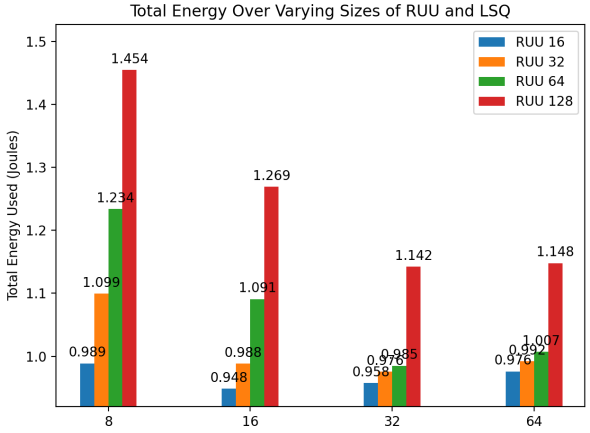

Fig. 4. Total Energy Consumption Estimate Versus Various RUU and LSQ Size (in number of instructions)

In terms of energy consumption, it is clear that the power (energy consumption in unit time) of a larger RUU and a larger LSQ will be higher than a smaller RUU and LSQ. However, since larger RUU and LSQ can increase average IPC, making both components reasonably larger will decrease execution time. In order to lower total energy consumption by only configuring RUU and LSQ size, we need to find the "sweet spot" where the higher power consumption brought by larger RUU and LSQ units does not override the benefit of decreasing execution time.

In the default configuration, both RUU and LSQ size limits the execution speed, as the total cycle count will decrease when increasing both RUU and LSQ size. However, when LSQ size equals 8 instructions, the limiting factor is RUU size; when LSQ size equals 16 instructions and RUU has size 32 instructions, the limiting factor is again RUU size, etc. To put it simply: when RUU size is small, it becomes the limiting factor; similarly, when LSQ size is small, it becomes the limiting factor.

D. What combination of RUU size and LSQ size leads to the lowest total energy consumption to complete the computation?

In Fig. 4, the total energy consumption is the lowest when both the RUU size and the LSQ size are equal to 16 instructions, resulting in 0.948 Joules in total.

II. Finding the Configuration with Lowest Total Energy Consumption

A. Altering Branch Prediction Scheme

Surprisingly, when changing the branch prediction scheme, the lowest total energy used is no longer produced at the configuration of RUU and LSQ both being 16 instructions. (See Fig. 5) According to the graph above (excluding the perfect branch predictor), the total energy used reaches the lowest when using the combined predictor (`comb`) with both RUU and LSQ size being 32 instructions. In all different RUU and LSQ size configurations above, the combined predictor always produces the lowest total energy used. The reason is that the combined predictor improves the execution time of the fast Fourier program much more than other branch schemes, resulting in a shorter total execution time. Increasing the size of RUU and LSQ will further increase IPC and reduce execution time. The benefit of a shorter execution time towards energy consumption brought by combined predictor out-weighs the energy cost that the larger RUU and LSQ have incurred. This can be confirmed by the fact that the combined predictor brings the highest average IPC value, which is equivalent to the lowest total cycle counts or lowest execution time. (See Fig. 6)

B. Adjusting Cache Configurations

The experiments below is carried out with the optimal RUU/LSQ size and branch prediction scheme determined in section A, specifically RUU and LSQ size of 32 instructions and the `comb` predictor.

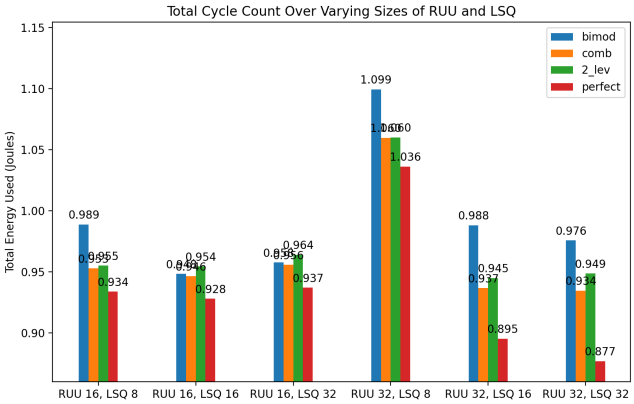

Fig. 5. Total Energy Consumption Estimate Versus Various Branch Prediction Scheme and RUU, LSQ size (in number of instructions)

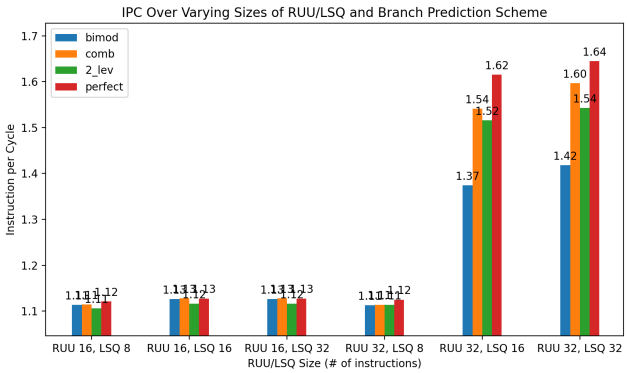

Fig. 6. IPC Value Versus Various Branch Prediction & RUU, LSQ size (in number of instructions)

The first experiment regarding cache is the adjustment of level 1 data cache size, specifically by varying the number of cache sets. The number of cache sets is adjusted while other properties, such as the cache line size and associativity, remain the same. From Fig. 7, the lowest energy used is produced at cache configuration with 32 sets. Increasing the cache size, or the number of cache set, will increase cache hit rate and reduce associativity conflicts since there are more spaces/sets to map a main memory address. (e.g. address 1 and 33 will both map to cache set 1 when there are 32 sets, but will map to cache set 1 and set 33 respectively when there are 64 sets) However, larger cache means more energy consumption. From Fig. 7, the benefit of increasing cache hit rate and thus reducing execution time is neutralized by the energy consumption of increasing cache size (or increasing number of cache sets), thus reaching a lowest point at cache set 32.

When changing the value of level 1 data cache line size together with its cache size (number of sets), the total energy consumption becomes even lower when the cache line size is 32 and the cache sets count is 32. This might be determined by the data structure and array size used in the fast Fourier transform program with default problem size: when cache line size is lower than 32, it's too small for holding array elements within the same spatial locality domain inside one cache line;

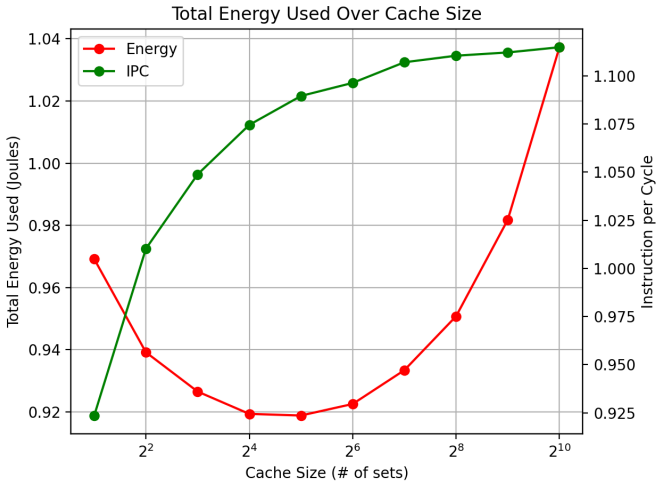

Fig. 7. Total Energy Use and IPC value Versus Cache Size

when cache size bigger than 32, it consumes too much energy.

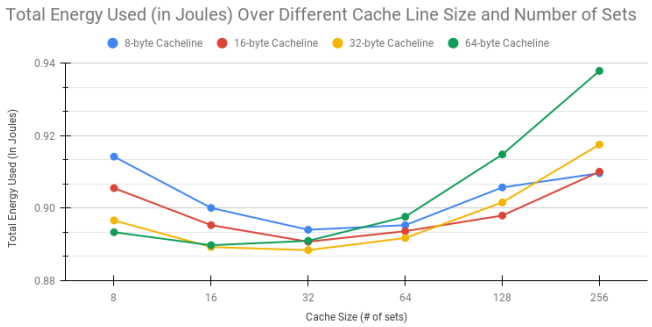

Fig. 8. Total Energy Used Versus Different Cache Line Size and Number of Sets of Level 1 Data Cache

On top of changing cache line size and number of cache sets of level 1 data cache, experiments have also been carried out on changing the configurations of level 2 data cache and level 1/2 instruction cache. Fig. 9-11 contains a series of experiment results. Each experiment is carried out with the optimal configuration found from the last experiment. The level 1 instruction cache brought the most significant improvement in total energy use, while level 2 instruction cache brings the least significant improvement.

Besides changing the cache line size and the number of cache sets, associativity and cache replacement scheme have also been altered. However, increasing associativity with a relatively large data or instruction cache will only increase energy consumption because the average memory access time will increase when the associativity increases. Changing the cache replacement policy does not significantly reduce the energy consumption either because benchmark studies show that LRU policy does not perform significantly better than random replacement in large caches on average.

C. *Changing the number of ALU*

Surprisingly, changing the number of ALU can significantly bring lower energy cost. Specifically, when reducing the

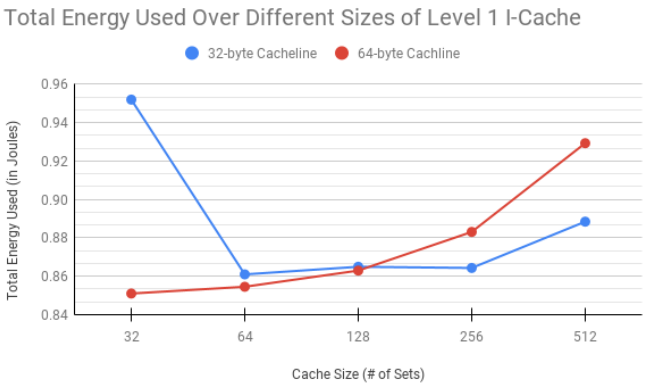

Fig. 9. Total Energy Used Versus Different Sizes of Level 1 Instruction Cache

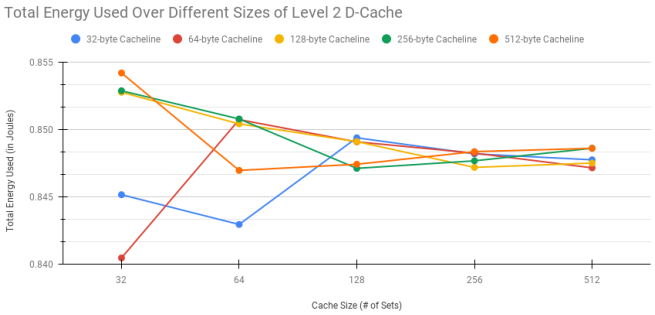

Fig. 10. Total Energy Used Versus Different Sizes of Level 2 Data Cache

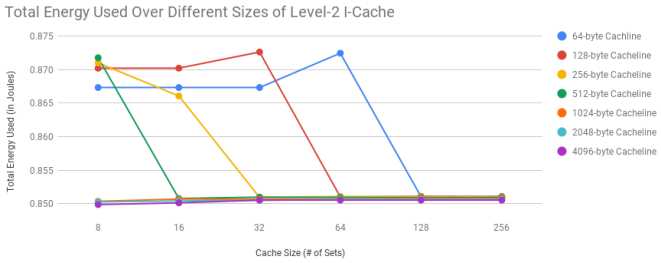

Fig. 11. Total Energy Used Versus Different Sizes of Level 2 Instruction Cache

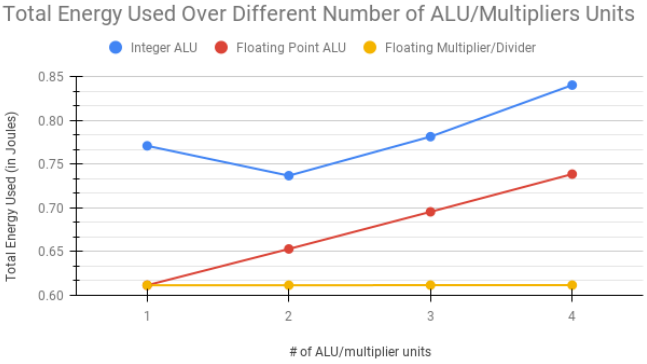

Fig. 12. Total Energy Used Versus Different Number of ALU/Multiplier Units. Notice that the experiment first finds an optimal number of integer ALU, then fixing this optimal number and find the optimal number of floating point ALU, then fixing this optimal number and find the optimal number of floating point multiplier.

number of integer ALU to 2 and floating point ALU/multiplier to 1, the total energy drops significantly. Since the program involves a lot of floating point multiplication and division (e.g. in the `fft()` and `ifft()` functions, the main operations are multiplication and division) rather than integer or floating point addition/subtraction, decreasing the number of integer ALU and floating point ALU will save more energy.

D. Changing the number of instruction fetch/issue/decode

The experiment below is carried out with the optional configuration found in previous experiments. The number of instructions fetched, decoded, and issued are changed simultaneously to the same value from 1 to 8.

Reducing the number of instruction fetched/issued/decoded per cycle can also significantly lower energy consumption. Higher rate of fetch/issue/decode instructions might cause pipeline congestion: the incoming instructions have to wait until previous instructions have been resolved, resulting in stalls and longer execution time. This can be confirmed by the fact that `ruu_full` and `lsq_full` values increase when the number of instruction fetched/issued/decoded per cycle increases from 1 to 4.

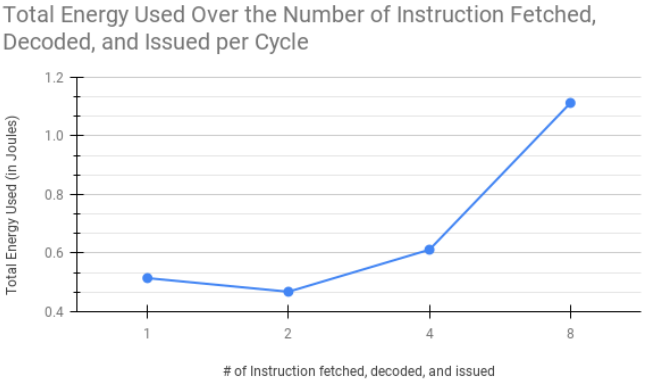

Fig. 13. Total Energy Used Versus the Number of Instruction fetched, decoded, and issued per cycle.

III. CONCLUSION

The final configuration that produces the lowest total energy used is: fetch/decode/issue with 2 instructions per cycle, 2 integer ALU, 1 integer multiplier, 1 floating point ALU and 1 floating point multiplier, with cache configuration of: `-cache:dl2 dl2:16:64:1:r,-cache:il2 il2:8:4096:1:r, -cache:il1 il1:32:64:1:r,-cache:dl1 dl1:32:32:1:r`, and branch predictor `comb`. This optimal configuration produces 0.468 J as the total energy used. The main principle used in finding those values is to balance the cost of energy and the increase in IPC. Some further explorations include: configuring branch prediction parameters, such as changing the branch predictor history size, changing the TLB size for both instruction and data cache, profiling the program to optimize the source code, and explore dynamics between different problem size and different cache configuration, etc.