Assessed Exercise 1: Microarchitecture Optimisation in Splay Trees

Charlie Lidbury

November 10, 2023

1 Intro

This report aims to roughly follow the structure of the specification, starting with the fixed exploration and then finishing with the energy optimization challenge. Apologies for the US English throughout, I can't figure out how to change my spell-check.

2 Studying Microarchitectural Effects

At the start of the specification, there are a few direct questions, this section answers those.

2.1 Parameters Interact (RUU vs LSQ vs Energy/IPC)

The more RUU you give the CPU, the more instructions per clock you get, likely because it can better utilize functional units, as shown in Figure 1.

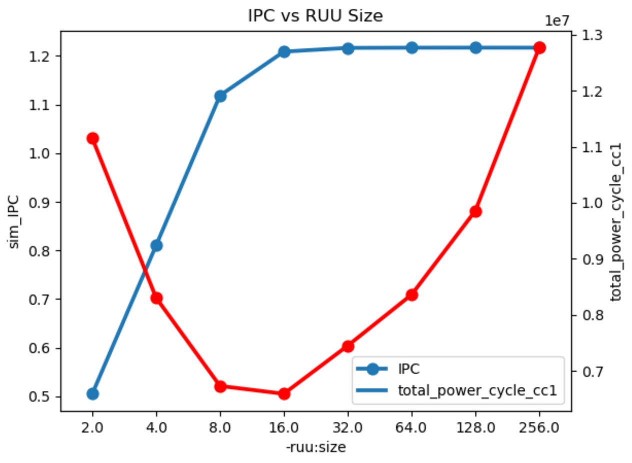

Figure 1: IPC/Energy vs RUU Size

However, as shown by the red line, this does not translate to better energy efficiency forever. This is likely because the extra instructions per clock are not worth the extra energy cost of the larger RUU.

If we vary both the RUU size and LSQ size at the same time, we can see that if either gets too small, the other becomes the bottleneck.

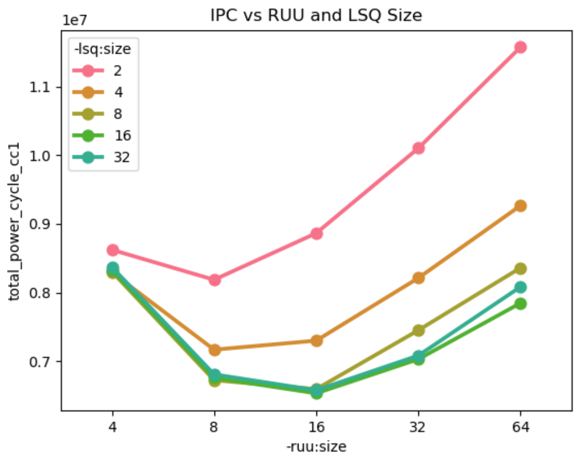

Figure 2: Energy vs RUU Size/LSQ Size

In Figure 2, when the LSQ size is small, say 2, the optimal RUU size is 8. This is because the LSQ is the bottleneck, so the extra RUU space can't be utilized and it's just wasted energy. When there is more LSQ space, higher RUU numbers provide benefits.

2.2 Bottlenecks

Something is a bottleneck if increasing its size would speed up overall performance, so to investigate which components are bottlenecks, we can vary their size and see if it affects performance.

We can alter the size/quantity of most components, but I'll assume for this question only the RUU and LSQ are under investigation.

The default parameters are RUU=16 and LSQ=8, so I'll start there and increase. SimpleScalar doesn't seem to record simulated time, so I will assume that microarchitectural changes don't affect clock speed, and therefore the number of cycles is a good proxy for time.

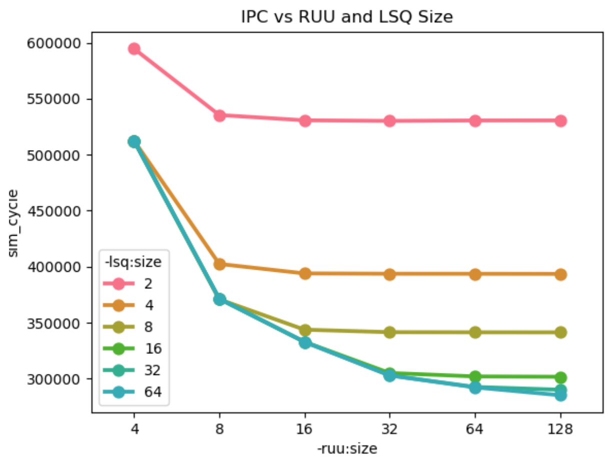

Figure 3: RUU and LSQ Increases vs performance

As shown in Figure 3, which one is the bottleneck depends on the size of the other. If the RUU is small, the LSQ is the bottleneck, and vice versa.

At the default parameters (RUU=16, LSQ=8) the LSQ seems to be the bottleneck because increasing the RUU past its default value of 16 doesn't increase performance. This is likely because the program has a lot of loads and stores, creating lots of entries in the LSQ.

2.3 Race-to-Finish Doesn't Always Win

As shown in Figure 2, the RUU/LSQ configuration with the lowest energy usage in RUU=16, LSQ=16, Figure 3 shows it is *at least* RUU=128, LSQ=64. Although there are diminishing returns, it looks like increasing the RUU and LSQ size always results in faster runtime.

3 Minimising Total Energy

3.1 Methodology

My search for the optimal parameters will occur in two main stages:

Informed Optimisation In the first section, I will reason about the program to find a set of reasonable variables; for instance, because I know there are very few floating point operations in the C source code, I can reduce the number of those to 1 without worrying too much about its implications on the other functional units.

Systematic Optimisation In this section, I will use pure statistical search to make sure I'm in a local minimum. After this, there could still be other places in the parameter space with better performance, but I will at least know that any valid change to any parameter will result in performance degradation.

3.2 Informed Optimisation

3.2.1 Program Analysis

Splaytest inserts many elements into a splay tree, then reads many elements, as a random walk. Because it's a random walk, elements that have previously been accessed are very likely to be accessed again. The design of splay trees means many of these operations will be very fast, as the accessed key will have recently been *splayed* over to the root.

Cache Looking at the splay tree code, it looks like each node is a struct with 4 8B fields, which probably means a cache block size of 32B is optimal. Because splay tree nodes don't get re-located upon splaying, there will probably be poor spatial locality, but due to the random walk, there will be good temporal locality. This will mean the overfetching we get from making the block size larger than the node size won't be useful, but making it smaller than 32B will be uneccesary and use more gates.

Memory access is free due to SimpleScalar not modelling it's energy usage, so small caches might be prefferable.

Branch Predictability This is an integer program with a high branching factor, which will probably make speculative execution and branch prediction less effective than they typically are. As a result, using a more complex branch predictor might help more than usual due to a large miss rate.

Functional Units Hardware dedicated to floating point operations and multiplies/divides will likely go almost entirely unused and do nothing but waste energy. The ALU will likely be used a lot in the random number generation, which looks expensive.

Speculative Execution Due to the high branching factor, I expect there will be a lot of stalls, resulting in speculating further and further into the future costing a lot of energy.

3.2.2 Where Is The Energy Being Used

Anaylsis of `power.c` shows us that `total_power_cycle_cc1` is calculated as a sum of `rename_power_cc1`, `bpred_power_cc1`, `lsq_power_cc1`, `window_power_cc1`, `regfile_power_cc1`, `icache_power_cc1`, `resultbus_power_cc1`, `clock_power_cc1`, `alu_power_cc1`, `dcache_power_cc1`, and `dcache2_power_cc1`; which does not include `fetch_stage_power_cc1`, `dispatch_stage_power_cc1`, or `issue_stage_power_cc1`. Figure ??ow each of those components contributes to the total:

3.2.3 Cache

3.2.4 L1 Block Size

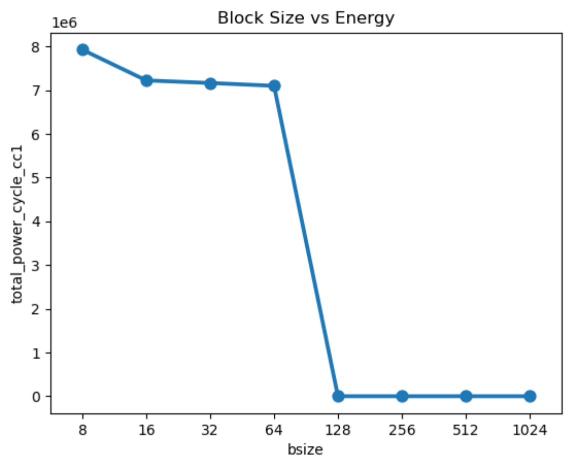

Figure 4: Energy vs Block Size

This analysis shows when block size reaches 128B, the energy usage falls off a cliff, I assume this is a bug so I will take 32B as my block size.

3.2.5 L1 Cache Size

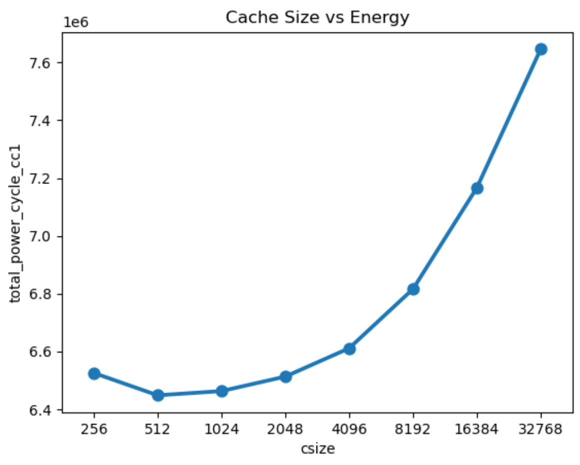

Figure 5: Energy vs L1 Size

With the block size minimized, we should be able to reduce the cache size until we start to see a performance hit. Figure 5 shows we can reduce the cache size from 16k to 1k while still reducing energy. This is likely because the random walk means we get a good temporal locality, so we don't need a large cache.

With this lower cache size, the optimal block size is still 32B. You might think that 64B would be counterproductive now that we have such a limited cache, but it only uses marginally more energy.

3.2.6 Functional Units

The only arithmetic-heavy thing in our program is the random number generation, so we should be able to reduce the number of functional units without affecting performance.

To test this, I will change the amount of each functional unit individually to see if it makes a difference.

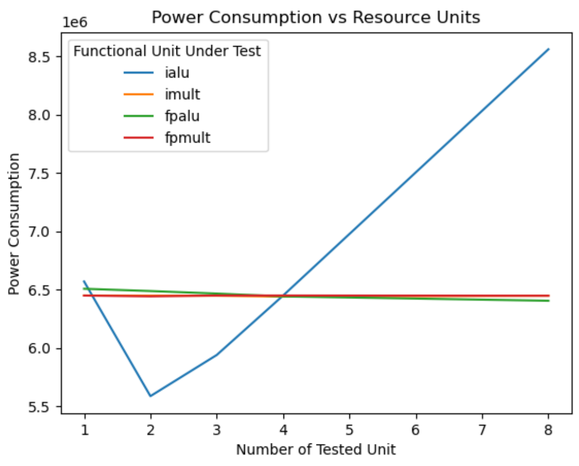

Figure 6: Functional Units vs Energy

Figure 6 shows the only functional unit that makes any difference is the integer ALU, which is best at 2. This makes sense as the other units are barely used, but the integer ALU is used for random number generation.