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.

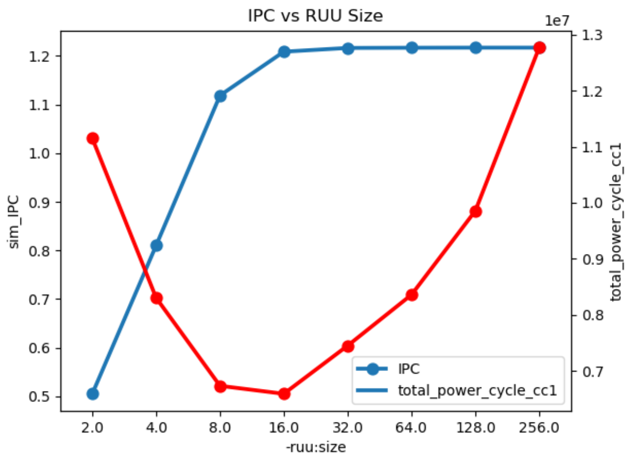

Figure 1: IPC/Energy vs RUU Size

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.

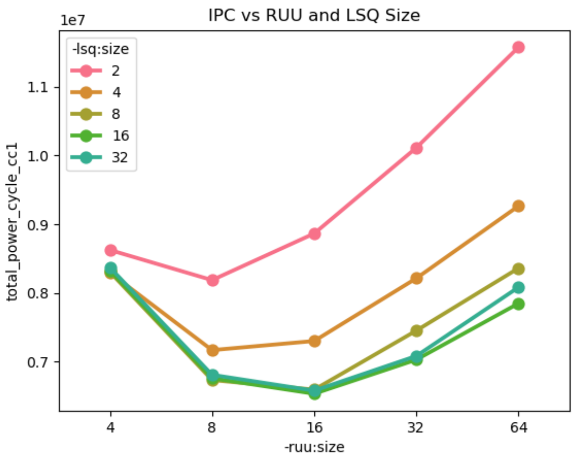

Figure 2: Energy vs RUU Size/LSQ Size

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.

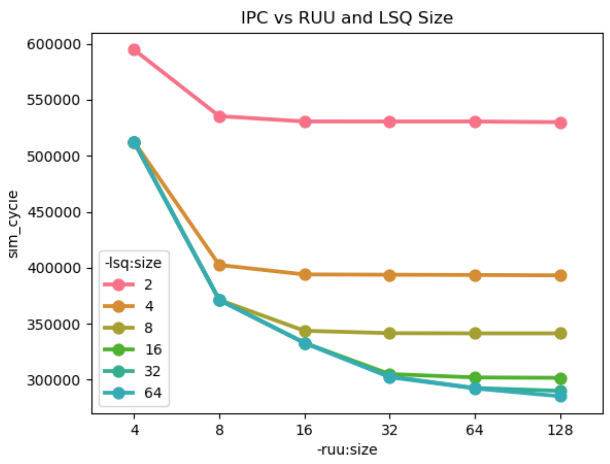

Figure 3: RUU and LSQ Increases vs performance

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 the optimal configuration for instructions per clock it *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 unnecessary and use more gates.

Memory access is free due to SimpleScalar not modeling its energy usage, so small caches might be preferable.

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

Analysis 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 4 shows how each of those components contributes to the total energy usage:

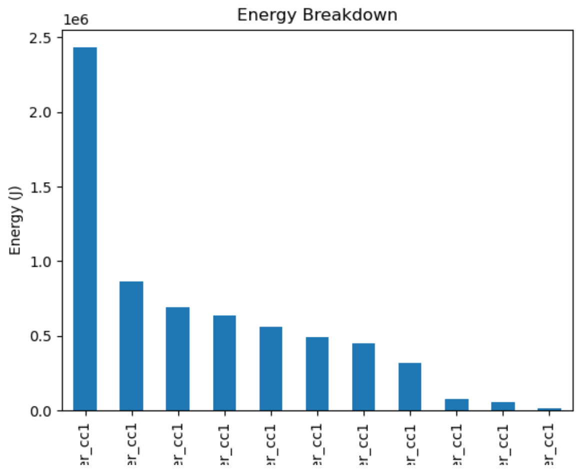

Figure 4: Energy Usage By Component

The following sections will go through the various components of the energy consumption, and attempt to optimize them.

3.2.3 Clock Power

I'm not aware of any parameters we have which directly reduce the overall clock power. Other changes may increase our IPC and therefore reduce the number of cycles, which will be caught by optimising individual components like the functional units or caches.

3.2.4 ALU Power

The second largest component of energy usage is the ALU, which might be quite a hard one to reduce because the instructions we need to execute are constant, however, we can try changing the number of ALUs we have and see if that makes a difference. While I'm here I will also try changing the number of multipliers and floating point units.

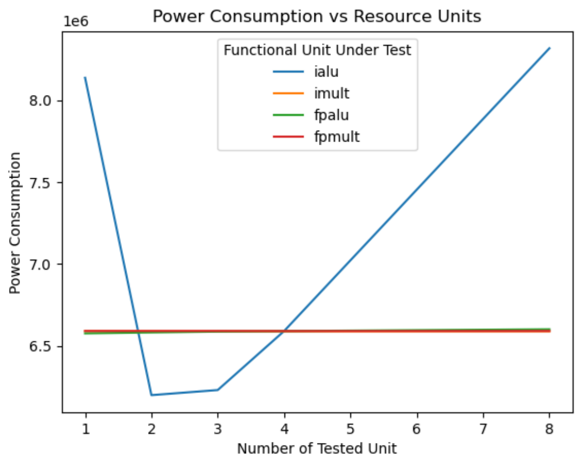

Figure 5: Functional Units vs Energy

Figure 5 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.

New Optimization

Params	`-res:ialu 2`
Energy	~~0.00659J~~ 0.00621J

I might have expected having unnecessary functional units to increase energy usage, but it seems like SimplerScalar has some mechanism to prevent unused functional units from using energy.

3.2.5 L1 Data Cache

To optimize L1 cache, I do a 2-way search between cache size and block size.

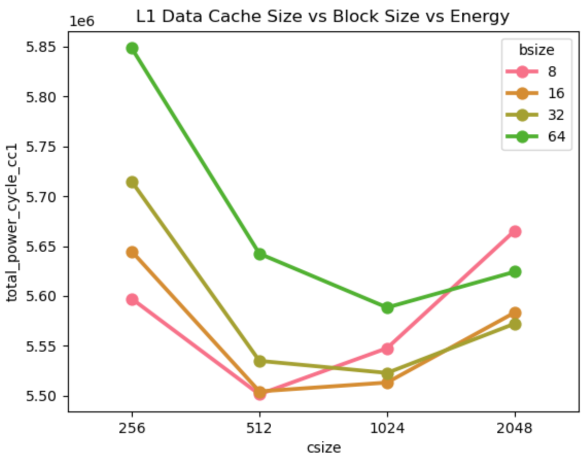

Figure 6: Block Size & Cache Size vs Energy

Figure 6 tells us the optimal cache is 512B large with 16B blocks. I suspected having a small cache would be better due to the temporal locality of splay trees (512B is a 32-fold reduction!) but I don't know why a 16B block performs better than a 32B one.

New Optimization

Params	`-cache:dl1 dl1:8:16:4:l`
Energy	~~0.00621J~~ 0.0055J

3.2.6 L2 (Unified) Cache

In SuperScalar, L2 cache seems extremely cheap, so we can fit an enormous amount of the splay tree in it, and prevent the CPU from ever having to go to main memory. To make it prefetch as much as possible, we use a huge block size.

Figure 7 shows the optimal for this is having 64KiB of L2 cache with 16KiB blocks.

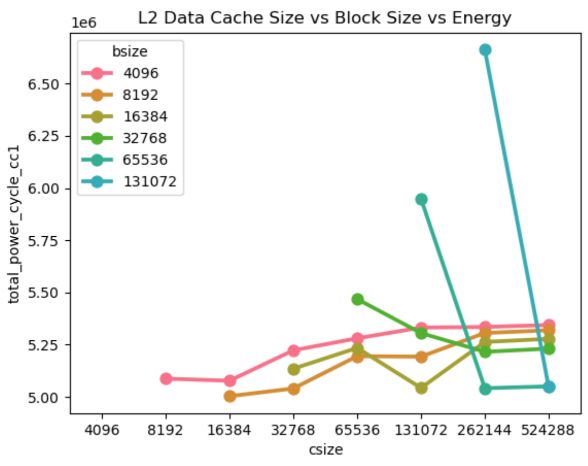

Figure 7: Energy vs L2 Cache

3.2.7 Branch Predictor

Branch prediction will be important due to the high branching factor of the splay tree program. Trying each predictor individually shows the combined predictor is the best, and Figure 8 shows the optimal meta table size is 16B.

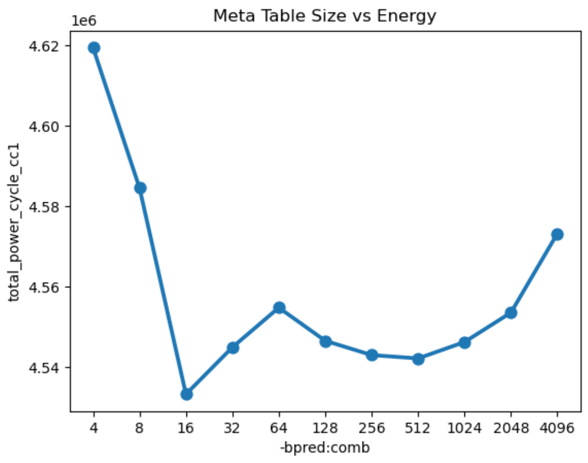

Figure 8: Meta Table Size vs Energy

New Optimization

Params	`-bpred comb -bpred:comb 16`
Energy	~~0.00468J~~ 0.00451J

The BTB size can be drastically reduced: as shown in Figure 9, the optimal size is 64B with an associativity of 4.

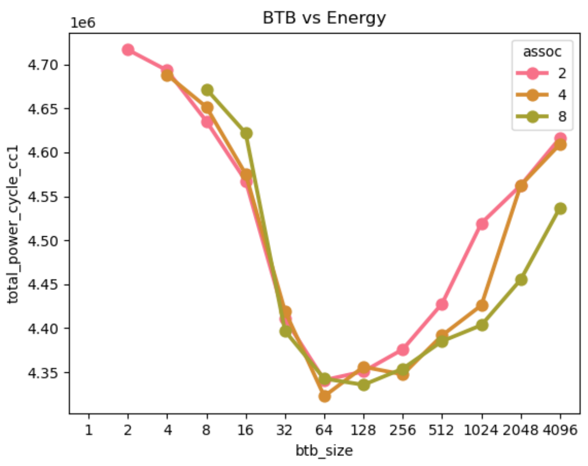

Figure 9: BTB vs Energy

New Optimization

Params	`-bpred:btb 16 4`
Energy	~~0.00451J~~ 0.00432J

3.2.8 Issue/Decode/Commit Width

While not explicitly an energy-consuming stage, everything is affected by the issue/decode/ commit width, so I will try to optimize it here. We alter the width of each of these stages individually and see how it affects energy usage.

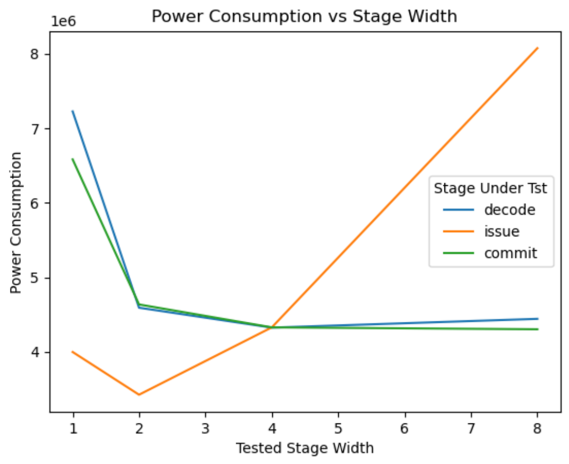

Figure 10: Stage Width vs Energy

Figure 10 shows the only improvement from the defaults are setting the issue width to 2. This might help because the high branching factor is causing a lot of pipeline stalls. This makes a huge difference.

New Optimization

Params	`-issue:width 2`
Energy	~~$0.00432J$~~ $0.00341J$