Imperial College London

COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Splay Trees Exploration

Authors:
Jacob Peake

Date: December 22, 2023

1 Studying Microarchitecture Effects

1.1 Varying RUU Size

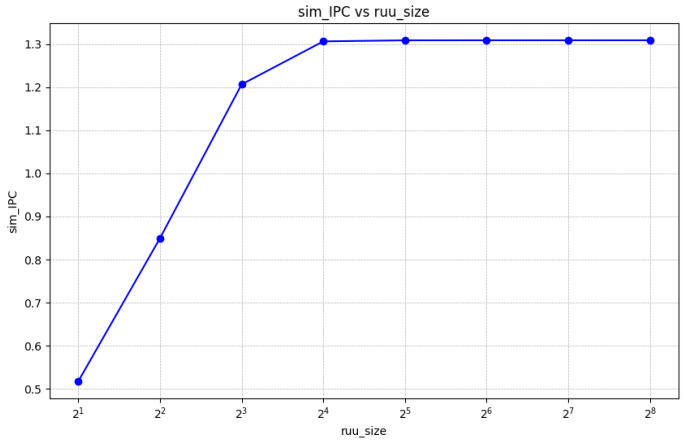

Figure 1: Instructions per Clock Cycle vs RUU Size

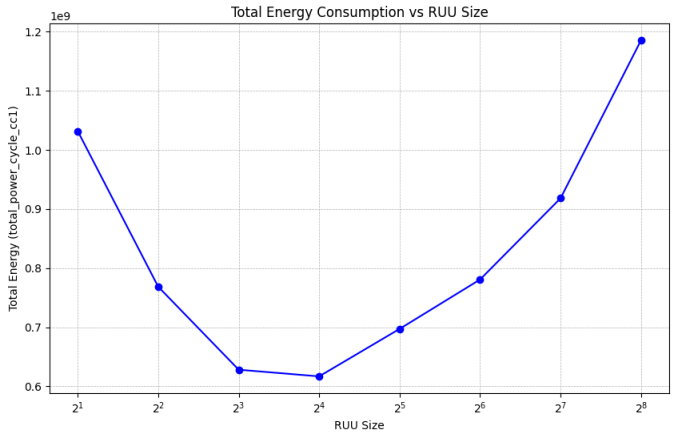

Figure 2: Total Energy Consumption vs RUU Size

For smaller RUU sizes (less than 16), there is a clear trend of increasing IPC with a larger RUU size. With a smaller RUU, the CPU is likely significantly constrained by the RUU Size (bottleneck), as the RUU gets full & cannot hold all of the issued instructions, leading to stalls in instruction execution. As RUU size increases, the RUU can hold more in-flight instructions, & the CPU can exploit more instruction level parallelism - leading to a higher IPC. As the Splay Tree has a skewed access distribution - an RUU that can handle more in-flight instructions can better handle the series of dependent instructions that arise & better exploit the ILP provided by the Splay Tree's Access Patterns.

The increase in IPC tapers of significantly past a RUU size of 16, from 8 to 16 there is a smaller relative increase, and after 16 the curve plateaus. As RUU size increases past this point, there is enough space in the RUU to handle all in-flight instructions, and the RUU Size does not contribute to additional ILP with other factors becoming bottlenecks, such as fetching instructions, or LSQ size. This also suggests that the Splay Tree operations may not have enough inherent parallelism to benefit from larger RUU sizes - the processor's ability to execute parallel instructions may become limited by the sequential nature of the workload itself. This is shown in the decrease in RUU Full Time. The IPC plateaus at around 1.3 instructions per clock cycle.

In exploring the effect of RUU Size on Energy Consumption, both the dynamic power (energy per cycle due to switching activity) & static power (energy per cycle due to leakage current in transistors) should be considered. Initially, there is an energy saving in increasing the RUU size, up to 16, suggesting greater energy-efficiency. As the RUU size increases, the time the RUU is full decreases significantly, leading to fewer stalls, higher throughput, & decreasing execution time, hence decreasing energy-consumption (despite the higher energy per cycle consumed by the larger RUU). This is shown in the IPC & execution BW metrics, which increase (more work done per energy consumed). The optimal RUU size of 16 suggests there is a balance between the increased energy per cycle (due to larger structure) & decreased number of cycles (due to higher throughput).

After an RUU size of 16, the energy consumption increases sharply as RUU size increases. As seen previously, past RUU size 16, there is a diminishing return in performance, therefore the larger RUU consumes more static power without providing any further increase in the IPC (hence, paying heavily in energy use for little performance improvement). After RUU of 16, the time RUU is full decreases significantly, and the RUU is no longer a bottleneck. After an RUU size of 32, LSQ full time increases, suggesting the LSQ becomes a bottleneck, causing increased consumption due to the increased waiting times & pipeline stalls due to memory accesses. For this Splay Trees benchmark, increasing RUU past 16 gives no performance benefit for a much higher energy cost.

1.2 Interaction between LSQ & RUU Size

To investigate the interaction between LSQ & RUU size, & how this impacts energy consumption - this can be visualised using a contour plot. RUU Size starts at 16 - which was seen previously to be the point at which increasing RUU Size further, increases energy consumption. In can be seen that, in general, energy consumption increases as both RUU size increases & LSQ size increases - due to the larger structures consuming more static power.

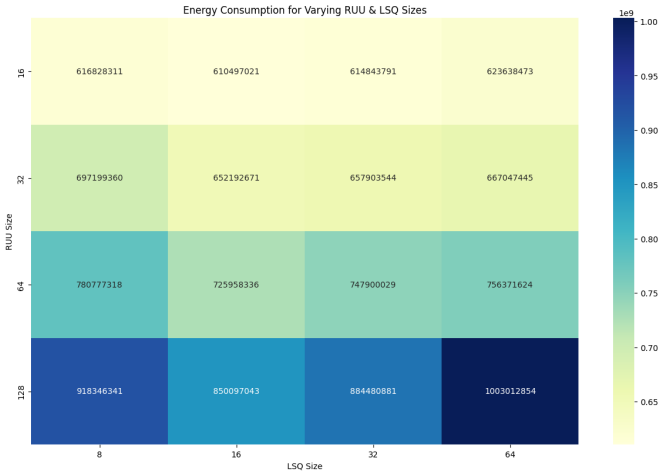

Figure 3: Total Energy Consumption for Varying RUU & LSQ Sizes

For smaller RUU sizes (such as 16) - increasing the LSQ size does not have a significant impact on the energy consumption. This means that even though increasing the LSQ increases the capacity for many memory operations - there is not enough instruction-level parallelism for it - so the RUU remains as the bottleneck. As the LSQ is not being filled, increasing the size will not increase the dynamic power consumption significantly - despite the static power increasing (small contribution).

As RUU size increases past 16, total energy consumption increases, as seen previously. Moreover - with greater RUU size - increasing LSQ size has a more significant impact on the increase of energy consumption. There is a clear trend between the increase of RUU size & the relative impact on energy consumption when increasing the LSQ size. This may be due to, if the RUU is too large relative to the LSQ, the potential parallelism from having many instructions in-flight could be bottle-necked by a lack of available slots for memory operations. A larger RUU will push more instructions to the LSQ, potentially causing the LSQ to become full, causing stalls & increasing execution time, & therefore energy consumption. If the LSQ does not become full (such as in the case of RUU 128, & LSQ 64), energy consumption still increases due to the increase in dynamic power. The dynamic power increases due to larger LSQs consuming more power per access, & having more accesses.

Despite the energy consumption increasing with LSQ size, it can be seen that the energy consumption actually decreases when increasing the LSQ size from 8 to 16. This suggests that the initial performance benefit from increasing LSQ size, therefore reducing LSQ stalls, leads to faster execution time & thus lower power consumption, before the dynamic power consumption of a large LSQ becomes too great.

This provides insight into a 'sweet spot' between RUU & LSQ sizes the provides the highest energy efficiency. As the Splay Trees benchmark involves a mix of computational (which heavily utilise the RUU for ILP) & memory operations (which utilise the LSQ for loads & stores due to splaying operations & tree manipulations) - the optimal configuration needs to both accommodate for the mix of ILP & Memory-Level Parallelism required for Splay Tree Operations. If these operations can be performed efficiently by adequate RUU & LSQ sizes - then there will be no bottlenecks in either structure & the processor will spend less cycles stalled, therefore being more energy efficient.

1.2.1 Bottleneck

As shown previously, depending on the respective sizes of the RUU & LSQ, either can become a bottleneck that limits the simulated execution speed depending on the circumstances. For smaller RUU sizes (2,4,8) - the RUU full time is significantly high - suggesting that the RUU is often full & is a bottleneck. This is due to the fact that their is a high degree of ILP available in the Splay Tree operations, but the RUU is too small to fully exploit it. This may become more pronounced in the computationally expensive parts of Splay Tree operations, such as rotations during splaying. As RUU size increases past 8, the RUU full time drops significantly, indicating that the RUU is less of a bottleneck. This is shown in the IPC improving, alleviating this bottleneck.

After an RUU size of 16, the LSQ full time begins to increase, and the LSQ starts to become the bottleneck. For RUU sizes above 32, the LSQ full time becomes significant, while the RUU full time still remains low - suggesting that the LSQ is the bottleneck in this circumstance. This is due to the significant amount of memory manipulation present in the node movements & rotations of Splay Tree operations - which give a high amount of load & store operations. If the RUU is sufficiently large to fully exploit ILP, but the LSQ is too small to handle all of the memory operations efficiently - the LSQ limits the execution speed.

1.2.2 Race-to-the-Finish

The combination of RUU Size 16 & LSQ Size 16 gives the lowest total energy consumption, as shown in the contour plot. It is observed that some alternative configurations (with larger RUU & LSQ sizes) provide faster execution time, but give greater total energy consumption. This is because, although reducing the execution time can often lead to reduced energy consumption as components are active for less time, the architectural features designed to increase performance may come with an increased energy cost. Such as increased RUU & LSQ sizes, giving larger structures which consume more static power, without providing significant performance gains to the Splay Trees Benchmark. Thus, although active for less

time, this configuration may consume more power per unit of time, that does not lead to a lower energy consumption. This is the trade-off between performance & energy efficiency.

The optimal sizes for RUU & LSQ mean that both the instruction scheduling, exploiting maximum ILP, and memory operation handling, exploiting maximum memory-level parallelism, are balanced. Neither RUU or LSQ becomes a bottleneck - leading to efficient execution with minimal stalls. The RUU & LSQ maximise performance & IPC, without becoming larger than required & consuming more power. Instructions can be processed as fast as data dependencies & memory latency allow, minimising execution time, & energy consumption. The Skewed Access Pattern of Splay Trees creates significant reliance on temporal locality - which can be exploited by the optimum configuration, keeping relevant instructions in-flight, and ensuring all memory operations can be handled.

1.3 Minimising Total Energy

To select an architecture configuration to run this program with optimum energy efficiency - a systematic strategy was implemented for varying each parameter. Initially, it was considered to simply implement an Exhaustive Search or Design of Experiments (DoE) approach - however, exhaustively searching microarchitecture configurations would lead to 27,648 possible configurations of SimpleScalar parameters - taking too long to search the entire parameter space. A DoE approach would allow me to explore more closely the interaction between parameters, but would require a more complex algorithm, & may not yield an optimal configuration. Therefore a modified One Factor at a Time (OFAT) approach was implemented, in which scripts would vary the microarchitectural features one at a time (i.e RUU & LSQ, Branch Predictor, Number of Functional Units, etc.) & observed which configuration gave the lowest energy consumption. To explore interaction between parameters, the parameters likely to interact with each other (e.g. RUU & LSQ Size) are included in the same script/factor - while using different scripts to vary other parameters, once optimal values had been found. This approach may miss a potential optimal configuration (as OFAT finds local optimum rather than global optimum across multi-dimensional configuration space) - but, exploring the interaction of parameters likely to affect one another meant this could be mitigated. This would also allow the observation of how varying each individual parameter affected the total energy consumption. Parameters are varied within a reasonable, realistic range of possible values that a configuration could take.

From previous results, it could be reasoned that an RUU Size & LSQ Size of 16 was optimum for energy efficiency. Varying the Type of Branch Predictor with an optimum RUU & LSQ size - led to a predictor that combined a bimodal and a 2-level predictor being optimum (comb). The Combined Branch Predictor displayed a higher IPC value & execution time - indicating more accurate branch prediction, with fewer cycle spent on branch misprediction paths - leading to greater energy-efficiency. This is shown by the improved branch address prediction rate. When a branch is mispredicted, the speculative instructions must be discarded, & pipeline flushed - wasting energy on such speculative instructions & stalling the pipeline which takes more clock cycles (& more energy per work done). More accurate predictions are due to the combined predictor leveraging the strength of both bimodal & 2-level predictors to use both global & local branch history to make predictions. The Bimodal Predictor will capture patterns in branches that are consistent - while the 2-level predictor will capture more complex, historical patterns. This also leads to better utilisation of the RUU & LSQ - as it prevents the RUU & LSQ becoming full with operations that will not be committed, shown by the decreased RUU & LSQ full times. Thus - the conditional branches in the Splay Tree benchmark are executed more efficiently, & with lower energy consumption.

Further, a script varied the number of functional units in the machine, giving an optimum of 2 Integer ALUs, 1 Integer Multiplier, 4 FP ALUs, & 1 FP Multiplier. This configuration is significantly dependent on the workload. This configuration suggests that the Splay Trees operations require more floating point ALUs that other functional units, to match the inherent parallelism of the workload & maximise utilisation of the functional units (maximise energy spent on useful computations). This may be due to the floating-point arithmetic involved in computation of rotations & splaying - which are utilised more often than Integer operations. This configuration suggests that the optimum is providing enough functional units to handle the parallel computations in the program, without adding more functional units that necessary that consume more power. This is shown in the IPC & execution time metrics - suggesting an efficient use of resources, & also giving a reasonable average slip between issue & retirement of instructions. This configuration must also not create bottlenecks in other parts of the microarchitecture - which could lead to stalls & increase IPC & therefore energy consumption.

Next varying the Branch Target Buffer configuration & Return Address Stack Size - gave an optimum value of a 256-set BTB with an associativity of 2, & a RAS size of 8. The optimal RAS size likely matches the typical call depth of the Splay Tree operations - not too large to waste energy, & not too small to underflow & mispredict returns. The optimal BTB configuration provides a balance between enough entries to effectively cache branch targets for the program's working set, while not having too many entries to become a significant energy drain, due to the static power draw. With a sufficiently sized BTB & RAS - the processor can more accurately make predict branch

targets & return addresses - reducing energy consumption (higher throughput, fewer stalls). This is shown in the higher Branch Address & RAS Prediction Rates given by the simulation of this configuration.

The final script varied the Instruction Fetch Queue Size & Decode, Issue, & Commit Bandwidth - giving an optimal Fetch Queue Size of 8, & Decode, Issue, & Commit Bandwidths of 2, 2, & 4 respectively. The IFQ size of 8 is the maximum configuration - this allows the processor to fetch the maximum amount of instructions per cycle - providing the decode stage with a steady stream of instructions, & ensuring cycle time & energy is not wasted waiting for fetching instructions. The decode width of 2 matches the issue width - meaning the processor decodes instructions at the same rate issued - ensuring that the decode stage is cannot become a bottleneck (leading to stalls & more energy), but also does not consume unnecessary power. The commit width of 4 means the processor can retire up to 4 instructions per cycle (more than issued). This is the maximum commit width possible & allows reduced stalls in execution due to waiting for instructions to commit. This configuration can fetch & commit instructions as the maximum rate (minimising stalls) while conversing the amount of instructions it can issue & decode to avoid additional power consumption. Although it does not give the lowest IPC, the IPC is still reasonable enough to lead to lowest consumption.

After finding the optimum microarchitectural configuration to minimise total energy consumption - the optimum memory system configuration must be found. To begin, the L1 Data Cache parameters can be varied - giving a L1D Cache of 64 sets of 32-byte blocks, & associativity 1 (direct-mapped). The L1 cache set & block configuration are likely the optimum cache size that can accommodate the locality of the benchmark - avoiding being too large and incurring higher energy costs. Larger caches consume more energy due to their size (static power) & also the increased complexity of tag checking & data retrieval. The L1 D-Cache miss rate is greater than other configurations with more associativity - leading to more accesses to main memory, however, clearly this cost is outweighed in the lower energy consumption of a direct-mapped design. Configurating a direct-mapped cache reduces energy consumption as it simplifies the hardware significantly as each memory address maps to only one cache line, hence only one tag needs to be checked (single comparator). The Splay Tree operations do exploit temporal locality - but it is observed that this is likely not enough to require a highly associative cache, & direct-mapped is sufficient to capture all of the locality present.

This cache configuration significantly decreases the total energy consumption, compared to previous microarchitectural changes. This is likely due to the L1 D-cache being frequently accessed for almost every load/store instruction - making its power usage a substantial contribution to the total energy consumption (as caches consume energy on each access - dynamic power).

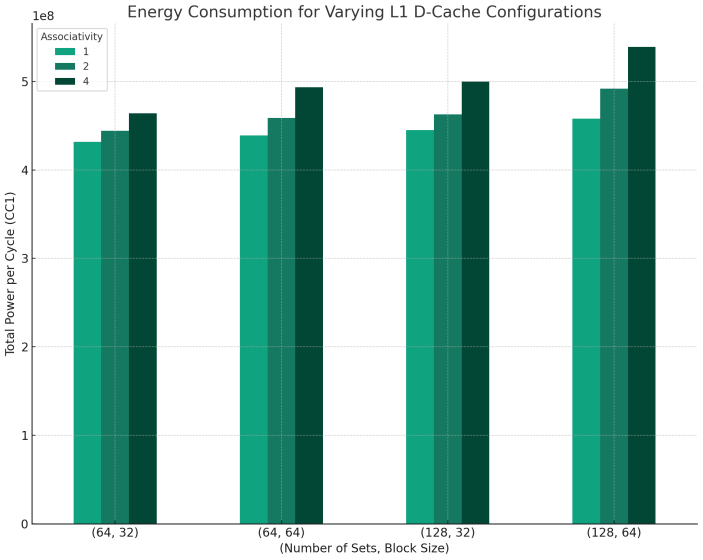

Figure 4: Total Energy Consumption for Varying D-1 Cache Configurations

Next, the optimum L1 Instruction Cache parameters are found - a 64-set I-Cache with 32B blocks & associativity of 2. The configuration likely balances efficiency & cache size - with 32B blocks typically providing a compromise that is sufficient of the smallest loop bodies & code sequences. Unlike D-Cache, the optimum I-cache has an associativity of 2, this trades off some energy consumption for an improved hit rate - which may prove beneficial for total energy consumption when fetching instructions rather than data.

For L2-Cache, the optimum configuration is a 256-set D-Cache of 64B Blocks, with associativity of 4, & a 256-set I-Cache of 64B Blocks, with associativity of 1. This optimum is a balance between accommodating the entire working set of the program & not being too large that it becomes inefficient due to slow access times or higher energy per access. The variation in energy consumption between L2 configurations is significantly less than L1 cache - suggesting the L2 structure has much less impact on total consumption. L1 Cache is accessed much more frequently than L2, making their energy consumption a significant part of the total, whereas L2 is only accessed on a miss of the L1 cache. Moreso, the L1 cache is physically included in the CPU core, meaning it typically operates at the same voltage & frequency of the CPU & is designed with power-hungry components, consuming more energy, whereas L2 caches often operate at lower frequencies & voltages & are not required to be as fast, consuming less energy. The final optimum energy consumption was $3.65x10^8$J, a 40% reduction from the default configuration.