Microarchitecture Issues in Three-Dimensional Modeling of RNA

Register Update Unit and Load/Store Queue

A first look at the performance by varying the size of the Register Update Unit (RUU) yielded the following results:

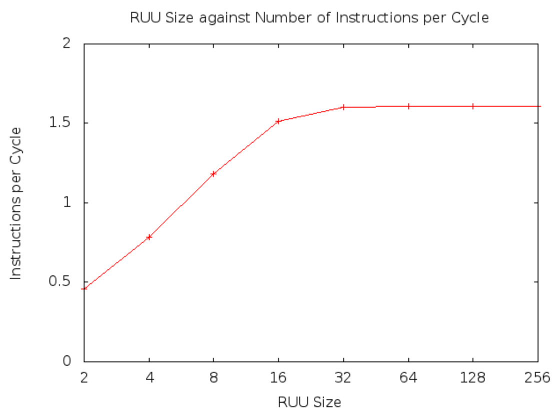

Figure 1. RUU Size against Instructions per Cycle

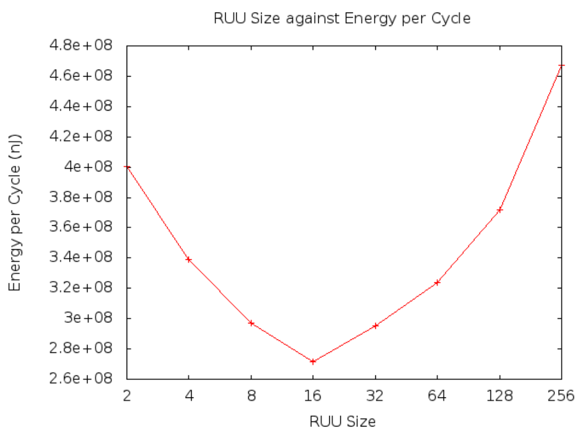

Figure 2. RUU Size against Energy (for entire execution not per cycle)

Figure 1 shows that as the RUU size increases, the number of instructions per cycle also increases. However the RUU size ceases to become the limiting factor for performance at a size of 64. I believe that as the number of instructions loaded into the RUU at any one time increases, there are a greater number of instructions that can be dynamically scheduled upon a stall, but all instructions beyond a certain distance (around 64) have dependencies on instructions that have not yet been scheduled, and so must wait to be executed. It appears that very few instructions can be scheduled out of order past a distance of 16 instructions.
Figure 2 shows that an RUU size of 16 uses the least amount of energy. As the RUU size increases, it would use more power as more transistors are required as the number of memory cells increase. However as the program is run faster, the cache uses power for less time. I think that the decreasing half of the graph is where the speed increase from an increased RUU is enough to reduce the power consumption as the RUU is used for less time, but after a size of 16 (as can be observed in figure 1) the speed increase is not significant enough to counter the increased energy consumption of a larger RUU.

The next parameter I varied was the size of the load/store queue (LSQ). This is a queue of pending memory operations, and works in a similar way to the RUU, allowing out of order (or in some cases even parallel) load/store instruction execution. I varied the LSQ and RUU together to find the best combination of the two. An RUU size of 16 was still found to be best. I plotted the performance for an RUU size of 16 and of 256, varying the LSQ size:

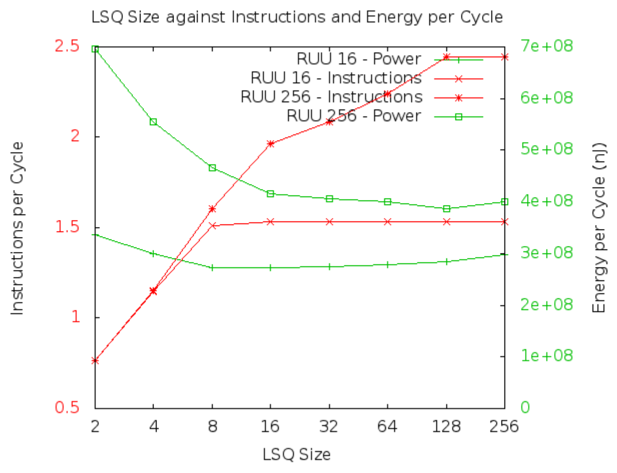

Figure 3. LSQ Size against Instructions and Energy (for entire execution not per cycle) for RUU Size of 16 and 256

I noticed that the performance in terms of instructions per cycle increased when the RUU size was set to 256. The optimal LSQ size in terms of instructions per cycle for an RUU of 256 seems to be 128, and for an RUU of 16 there is no significant increase in instructions per cycle above an LSQ size of 8. This indicates that an LSQ size of about half the RUU size is best. From this perhaps we can conclude that roughly half of the instructions in the compiled RNA modelling program are load/store operations. In terms of energy consumption, best combination was RUU of size 16 and LSQ of size 8.

This gives us a current best score in energy consumption of 2.718x10^8nJ (to 4 significant figures) which is 0.2718J.

Fetch, Decode, Issue, Commit

Fixing the RUU at 16 and the LSQ at 8, I examined the effects of the instruction fetch queue size and the instruction decode, issue and commit bandwidth on the performance of the program.

The instruction fetch queue size dictates the number of instructions that are held waiting to be decoded at a given time. A larger queue will reduce the latency of the fetch stage (some fetches may miss the instruction cache), but as with RUU and LSQ will increase in energy consumption as they use more transistors.

The instruction decode bandwidth is the number of instructions that are decoded and then passed to the RUU at any one time. A greater bandwidth would fill the RUU faster, giving it a wider choice of instructions for out-of-order execution.

The instruction issue bandwidth is the number of concurrent instructions that are allowed to execute at the same stage in the pipeline, following a superscalar architecture. The greater the number of parallel execution, the more wires and components in the processor and so the higher the energy consumption, but the greater the number of instructions able to run at any one time.

The instruction commit bandwidth dictates the number of instructions that can be retired in a given cycle. This follows the same principle as above, that a greater bandwidth will consume more energy and run faster, and a smaller bandwidth will consume less energy but be slower.

The configuration of these four parameters that achieved the lowest energy consumption was:

Fetch Queue Size	8
Decode Bandwidth	8
Issue Bandwidth	4
Commit Bandwidth	8

This provided a total energy usage of 0.2608 J (2.608x10^8 nJ).

I was surprised to see that they were not all equal, as I would expect the entire pipeline being of the same bandwidth to be most efficient. From the configuration above I would expect the issue bandwidth to be the bottleneck, executing instructions at a slower rate than they are fetched, decoded and finally committed. The issue bandwidth of 4 seems to imply that no more than 4 instructions can be executed at the same time on average due to dependencies between them.

Branch Prediction

A branch predictor attempts to guess which instruction will be executed next in the pipeline without knowledge of the result of the branch condition. Incorrectly predicting a branch means that the processor must stall and not write-back or commit the result of the instruction(s) on the incorrectly predicted branch currently in the pipeline.

The strategies “Not Taken” and “Taken” are easy to implement and require very little hardware, but “Bimodal” and “Two Layer” are more complicated. The “Bimodal” strategy has a 2 bit counter, used to determine which state it is in, requiring two consecutive mis-predictions before to switch between “Taken” and “Not Taken”. “Two Layer” requires the most hardware and therefore the most energy to implement. It remembers the history of taking or not taking previous branches, and requires a pattern table to store this information.

Using different branch prediction strategies gave the following results:

Branch Prediction Strategy	**Instructions per Cycle**	**Energy Consumption (J)**
Not Taken	0.9554	0.2990
Taken	0.9571	0.3000
Bimodal	1.5600	0.2608
Two Level	1.5434	0.2624
Combination of Two Level and Bimodal	1.5690	0.2604
Perfect	1.5148	0.2519

Of course the perfect branch prediction strategy yielded the best results, however this is not realistic and cannot be done in practice (or at least has not yet been achieved!). Excluding the perfect strategy, the best was a combination of the two level and bimodal branch prediction strategies. It would seem that even the speedup of 0.009 instructions per cycle is enough to warrant the use of both the two level predictor and bimodal predictor in combination, and to make using up more space on the chip and consuming more energy worthwhile in the total energy consumption of the program.

Memory Access Width

The memory access bus width was the next parameter I modified. This dictates the amount of data that can be retrieved from memory at any one time. The results from varying this were as follows:

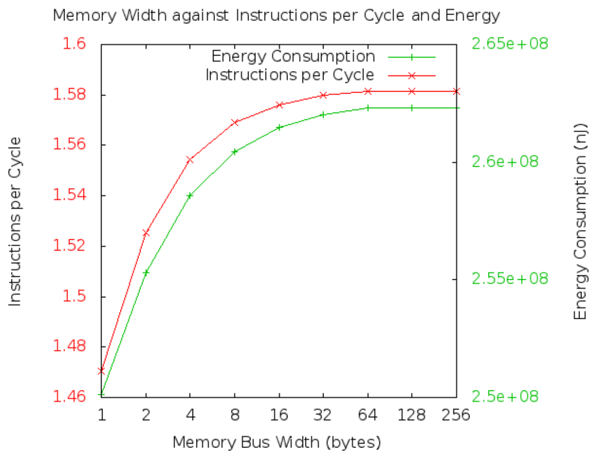

Figure 4. Memory Bus Width against Instructions per Cycle and Energy Consumption

As the memory access bus width increased so did the speed of execution. The energy consumption also increased however as a wider memory bus requires more energy as it is constructed of more wire. It was interesting to notice that the speedup did not reduce the energy consumption much (as the program would complete faster). Investigating this further, I discovered that this is due to the level 1 data cache miss rate being very low (0.5%) and so the memory bus is rarely utilised and should be made as small as possible.

Arithmetic Parameters

Finally, I varied the arithmetic parameters. The four parameters were the number of integer and floating point ALUs and multipliers. Unused ALUs or multipliers will not slow the CPU, but will use unnecessary energy. Too few of these may mean that they become the bottleneck and slow the processor enough to increase energy consumption. The best combination of these were as follows:

Integer ALUs	2
Integer Multipliers	8
Floating Point ALUs	1
Floating Point Multipliers	8

This gave a final energy consumption of 0.1829J. It appears that the RNA modelling program has many multiplication operations and so fully utilises 8 integer and floating point multipliers (the maximum simplescalar will allow). There are less additions, subtractions and logical operators in the program, and so there is no need for extra ALUs of both kinds.

Sweet Spot

The best energy consumption of 0.1829J was achieved with the following microarchitecture parameters:

Parameter	**Value**		**Parameter**	**Value**
RUU Size	16 instructions		Branch Prediction Strategy	Combination
LSQ Size	8 instructions		Memory Bandwidth	1 byte
Fetch Queue Size	8 instructions		Integer ALUs	2 ALUs
Decode Bandwidth	8 instructions/cycle		Integer Multipliers	8 multipliers
Issue Bandwidth	4 instructions/cycle		Floating Point ALUs	1 ALU
Commit Bandwidth	8 instructions/cycle		Floating Point Multipliers	8 multipliers

Bottleneck

With the default architecture, the energy consumption is 0.2718 J and the instructions per cycle is 1.5118.

Having found the “sweet spot” with the lowest energy consumption, I tested each of the parameters individually in turn, changing the value from the default to the optimal value above. The parameter change that gave the greatest reduction in energy (a difference of 0.0556 J) was reducing the number of floating point ALUs to 1 and so it would appear that this was the individual energy bottleneck. It is a reduction of 3 from the default value of 4, and the RNA program seems to do very little floating point computation (besides multiplication) and so 3 of these ALUs were almost redundant.

To find the bottleneck in terms of speed (not caring about energy), I varied each parameter individually setting it to it’s maximum (or a very large) value. The microarchitecture parameter that gave the largest increase in instructions per cycle was the RUU Size (increasing to 256) which gave an increase of 0.0936 instructions per cycle. This seems to be because it makes better use of instructions that can be used to fill bubbles in the pipeline. However I do not think that there is a strict bottleneck in a single parameter, as all components seem to be quite dependent upon each other.

Evaluation

In the above, I varied groups of microarchitecture parameters in combination, but these groups were varied separately to other groups. Therefore I do not think that I found the perfect “sweet spot”, as all parameters seem to depend upon each other in different ways. I would have liked to have tried every combination of every parameter, but unfortunately the number of combinations would have been nearly a billion, which would take a while to run (even with parallelism). However I think that the groups I chose were logical and so believe my energy consumption to be close to the minimum.

Appendix

To vary multiple parameters together, I created a parallelised version of `varyarch` which gave me results faster. An example of this script would be for the fetch, decode, issue and commit parameters:

```
#!/bin/bash

i=0
for ifqsize in 2 4 8 16 32 64
do
  for decodewidth in 2 4 8 16 32 64
  do
    for issuewidth in 2 4 8 16 32 64
    do
      for commitwidth in 2 4 8 16 32 64
      do
        ((i++))
        # Echo the process number that we have started to stderr to report progress to user
        echo "$i start" 1>&2
        # Write the parameter values to the individual process output file
        echo -n "ifq:$ifqsize iss:$issuewidth dec:$decodewidth com:$commitwidth "
          > "temp/"$i"sim.out"
        # Run the simulation with the desired parameters
        ./run-wattch-arg "-fetch:ifqsize $ifqsize -issue:width $issuewidth -decode:width
$decodewidth -commit:width $commitwidth" 2>&1 |
          # Extract the results we are interested in (IPC and Energy)
          grep -E 'sim_IPC|# total power per cycle_cc1' |
          # Extract from this only the numeric data (easier for gnuplot)
          # Store output in temporary file
          grep -E -o '[0-9]+.?[0-9]*' >> "temp/"$i"sim.out" &&
          # Echo that process is done to stderr.
          # ‘&’ used to execute simulation in new thread
          echo "$i done" 1>&2 &
      done
    done
  done
done

# Wait for all threads started to complete
wait

for ((j=1;j<=$i;j++))
do
  # Collate all of the temporary files and output to stdout
  # (this script’s stdout would usually be redirected to a file)
  cat "temp/"$j"sim.out" | echo `perl -ne 'chomp and print'`
done
```

Where `run-wattch-arg` is a modified version of `run-wattch` which uses a command line argument rather than a global environment variable.