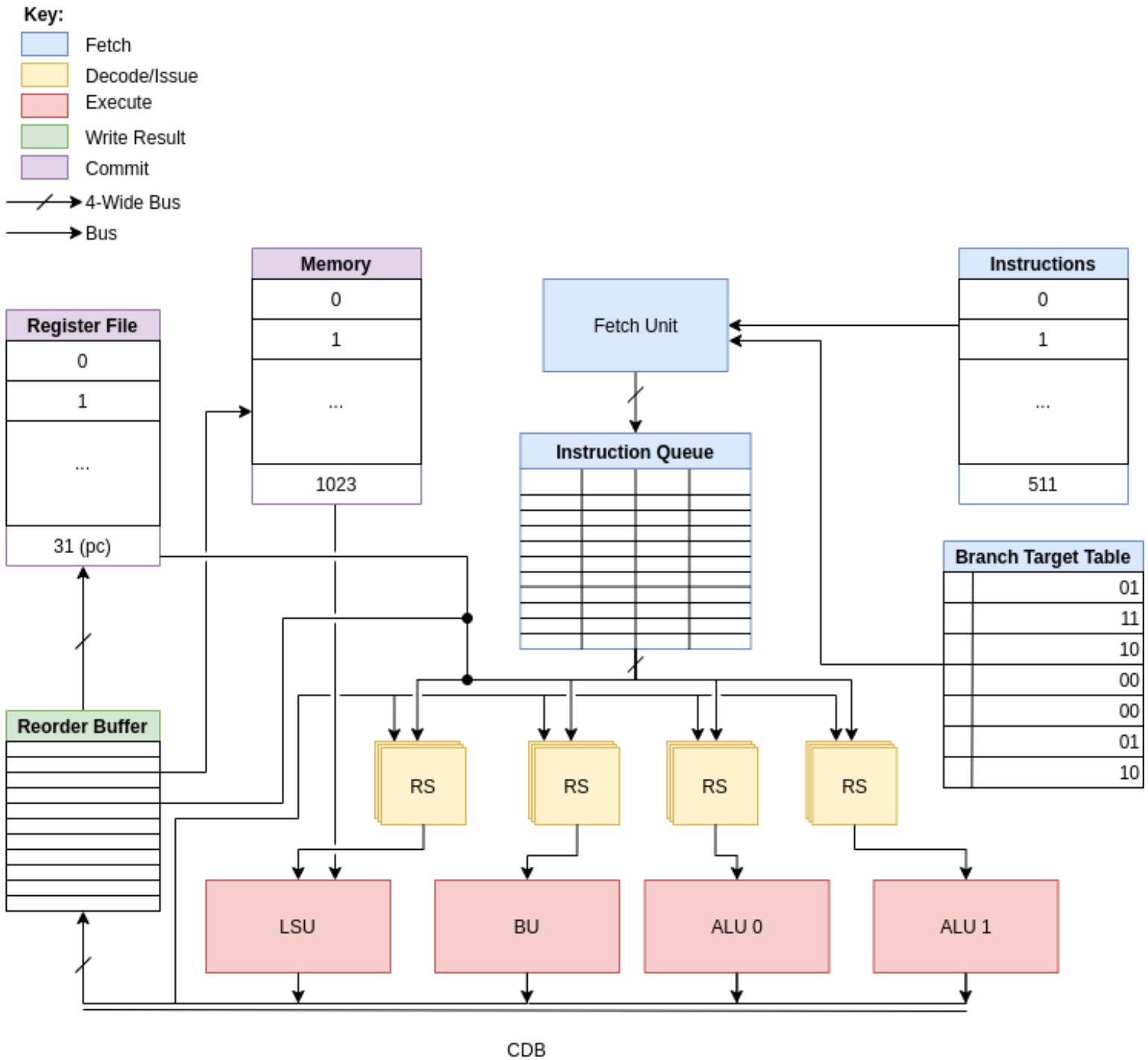
Advanced Computer Architecture 2021

Superscalar Processor Simulator

Tymoteusz Suszczyński

Processor Architecture

The figure depicts a brief overview of the architecture of the processor simulated with the default configuration.



Simulator Features

- **Fully pipelined**
  - Each instruction passes through **5 stages** before reaching completion: **Fetch, Decode/Issue, Execute, Write Result** and **Commit**.

- **n-Way superscalar**
  - Up to n (4 by default) instructions can be fetched and committed simultaneously, provided there is sufficient ILP to allow it.
  - There are **2 ALU's, 1 Load/Store and 1 Branch Unit** (by default) to allow simultaneous execution of instructions.

- **Out of Order dynamic scheduling**
  - An implementation of **Tomasulo's algorithm** is used to allow for OoO execution.
  - Instructions take **multiple cycles** depending on their complexity, so that shorter instructions can overtake longer instructions.
  - **Register renaming** to prevent WAW and WAR hazards.

- **Speculative execution**
  - **Reorder buffer** up to 128 entries.

- **Non-blocking issue**
  - Distributed **reservation station** layout (one per EU), each holding at most 8 entries for instructions allocated to that unit. These help prevent RAW hazards.

- **Dynamic Branch prediction**
  - 2-bit prediction scheme.
  - Static and Fixed branch prediction also available.

- **Configurable!**
- - Static, dynamic and fixed branch prediction options.
  - Scalar width can be changed.
  - Number of ALUs (and their corresponding RSs) can be changed.
  - Configurable via command line arguments.

Benchmarks

The Simulator comes with four benchmarking programs:

- 'gcd' implements Euclid's algorithm to calculate the greatest common divisor between two numbers.
- 'bubble' implements a bubble sort for a list of 100 numbers.
- 'peak' implements a divide-and-conquer peak finding algorithm for a list of 50 numbers.
- 'vector' carries out the sum of 2 vectors, each consisting of 100 numbers.

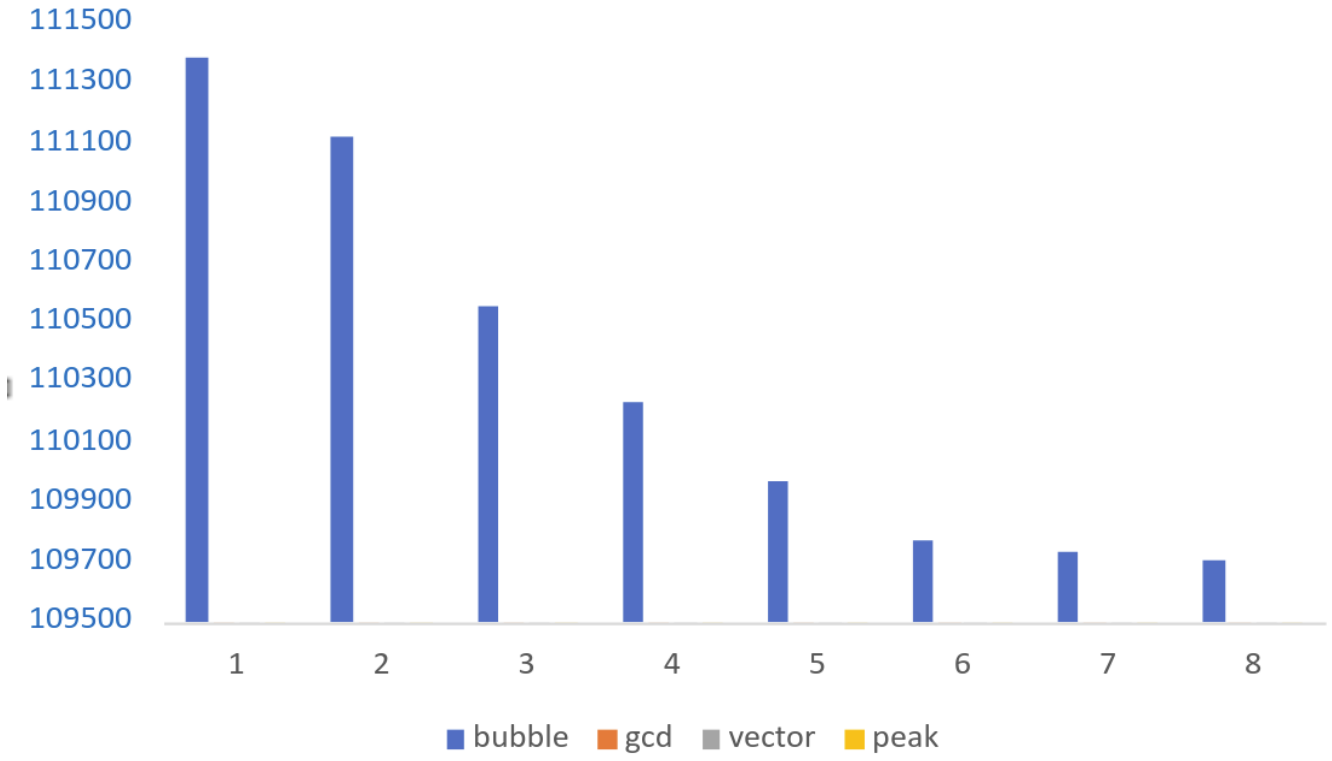
Experiment 1

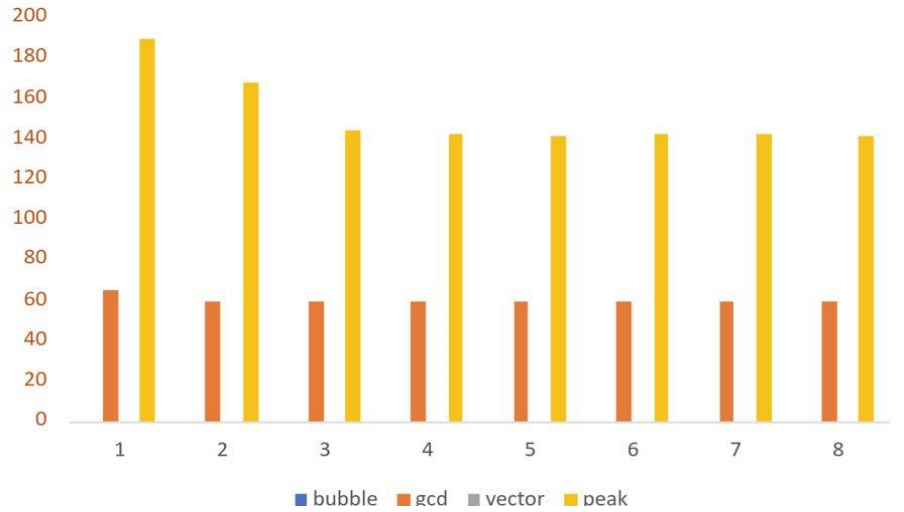
Hypothesis: The greater the scalar width the fewer the number of cycles required to complete the benchmark.

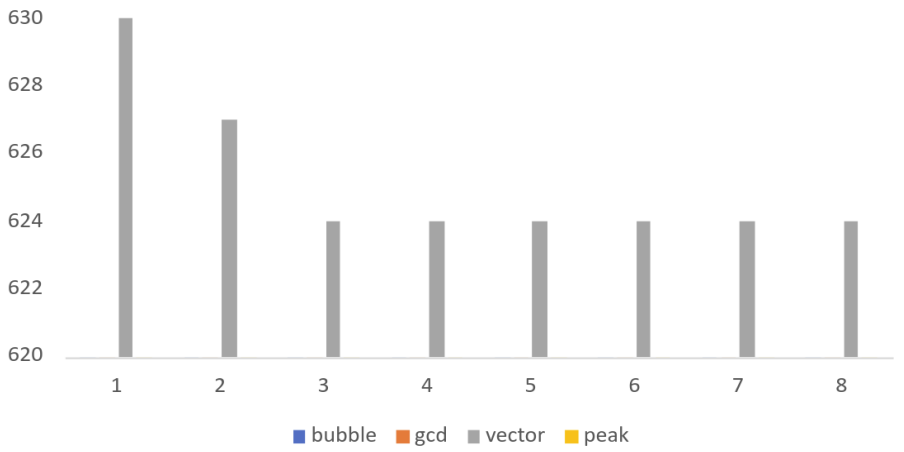
Independent variable: Scalar width
Dependent variable: Total cycles
Control variables:

- All benchmarks using dynamic branch prediction.
- All benchmarks using 4 EUs.

Experiment 1: Results (lower is better)







Experiment 1: Conclusion

All benchmarks, except gcd received a benefit from superscalar execution. This is likely due to gcd not having enough potential for ILP. bubble benefited the most, still receiving improvements with a scalar width of 7!

In conclusion, increasing scalar width may increase performance, proviging the ILP of the program allows for it. However, diminishing returns are seen after a scalar width of 4.

Experiment 2

Hypothesis: Dynamic branch prediction will allow the benchmark to complete in less cycles than Static or Fixed branch prediction methods.

Independent variable: Branch prediction method
Dependent variable: Total cycles
Control variables:

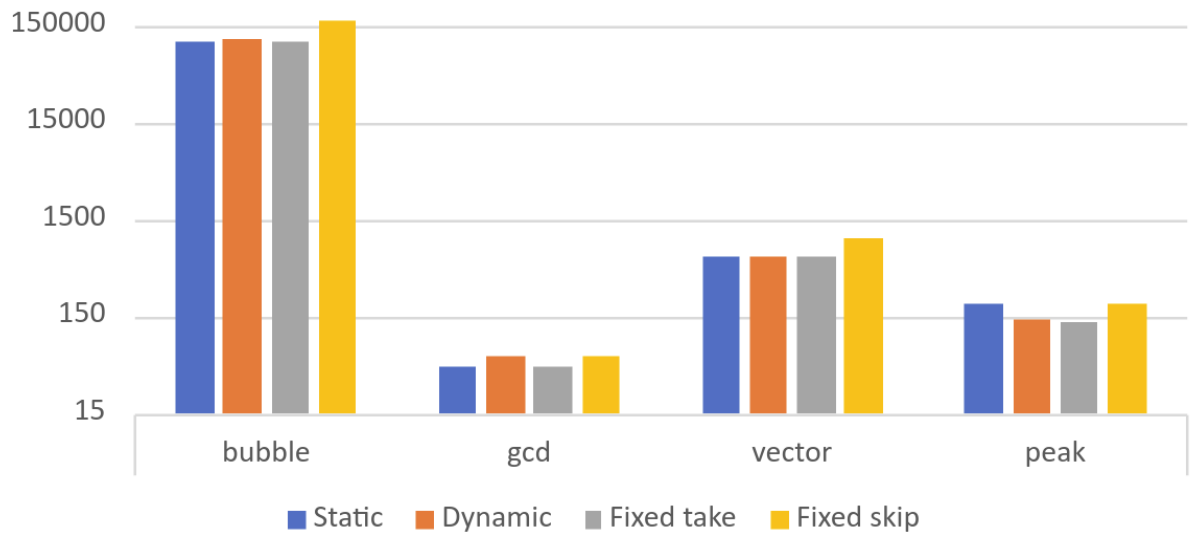
- All benchmarks are running 4-way superscalar.
- All benchmarks using 4 EUs.

Experiment 2: Results (lower is better)

From these results, we can see that both static and dynamic branch prediction methods offer an improvement over predicting the branches as not taken every time.

Surprisingly, predicting the branches are taken every time has received good performance throughout. This is likely due to the nature of these benchmarks, as they include plenty of loops where branches are taken far more often than not.

For all benchmarks except 'peak', static prediction has performed better than dynamic. Again, this is likely due to the nature in which these benchmarks were written.



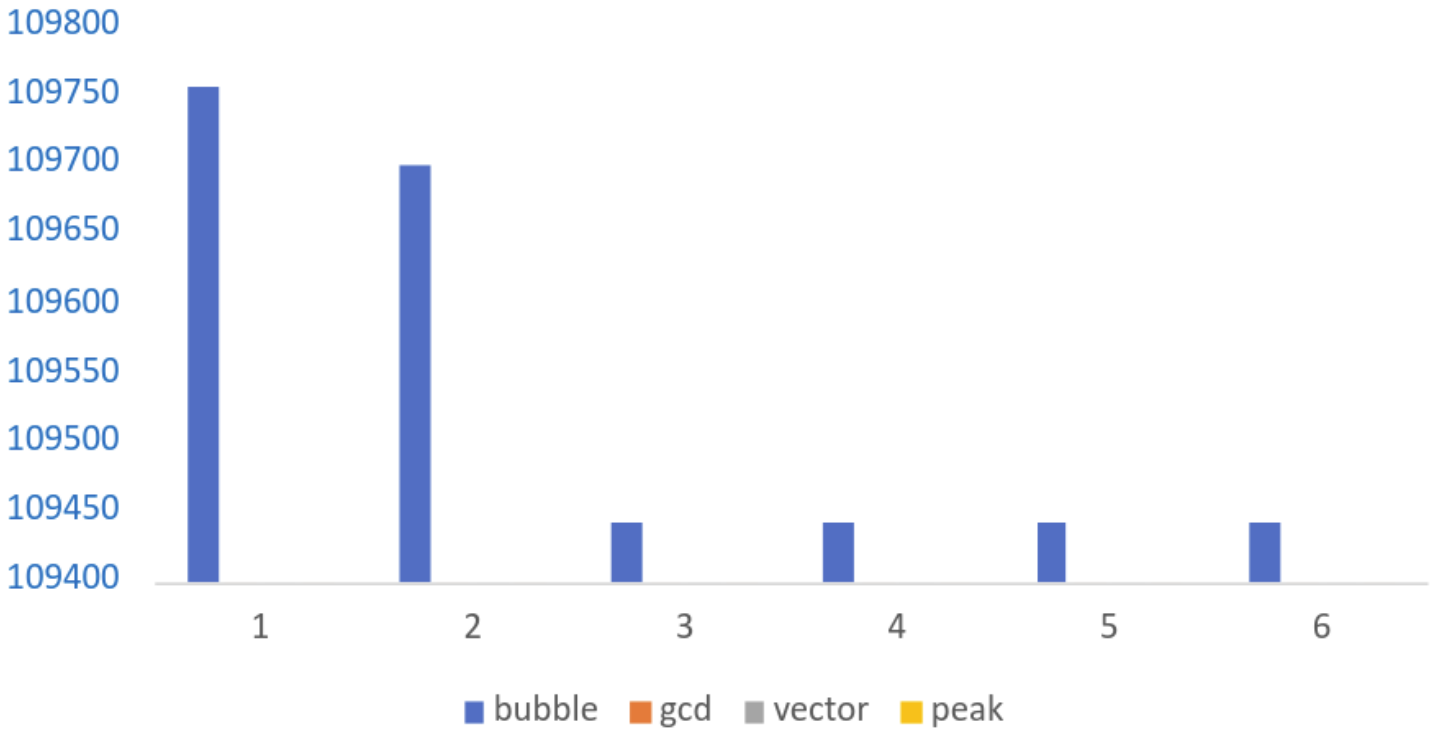
Experiment 3

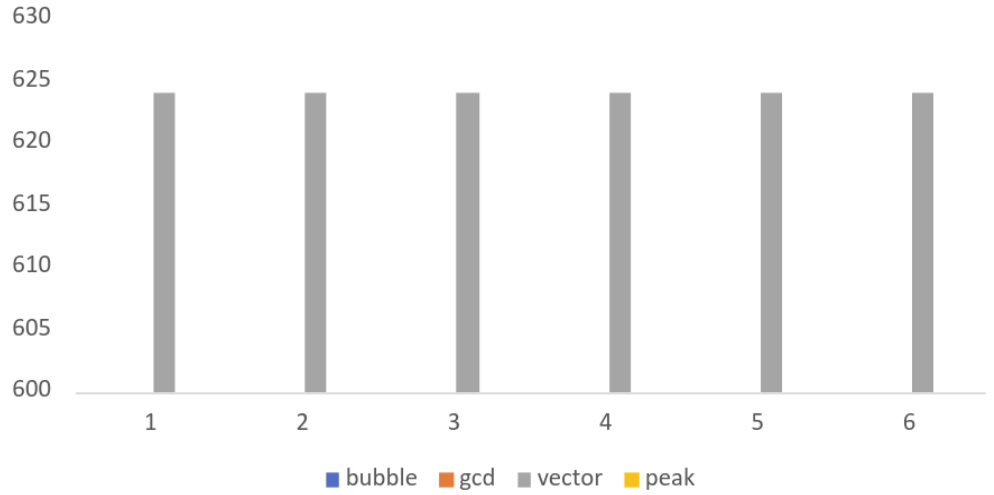
Hypothesis: The greater the number of ALUs, the fewer the number of cycles required to complete the benchmark.

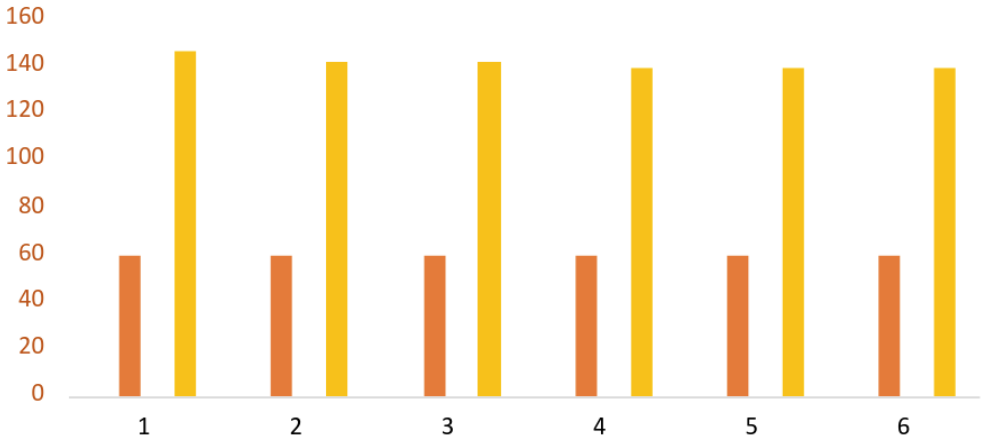
Independent variable: No. EUs
Dependent variable: Total cycles
Control variables:

- All benchmarks using dynamic branch prediction.
- All benchmarks are running 8-way superscalar.

Experiment 3: Results (lower is better)







Experiment 3: Conclusion

The purpose of this experiment was to follow-up from Experiment 1, which showed diminishing returns after a 4-way scalar width. I wished to determine if performance could be further improved with a wide superscalar architecture, if the number of execution units also increased, as I hypothesized that this might have been a bottleneck.

Experiment 1 having only 2 ALUs, these results show that a further performance increase can be gained by increasing the number of ALUs while keeping the same 8-way scalar width. The 'bubble' benchmark shows this especially when going from 2 to 3 ALUs.

Unfortunately, the other benchmarks showed either minimal, or, in the case of 'gcd', minimal gain. This could be due to not enough ILP in arithmetic instructions for those benchmarks. With only 1 LSU and 1 BU, there is no way those types on instructions can execute at the same time. To conclude, additional performance can be gained from a superscalar architecture when more ALUs are available, provided that there is enough ILP in the program.

Thank you