NC State University

Department of Electrical and Computer Engineering

ECE 463/563: Fall 2021 (Rotenberg)

Project #3: Dynamic Instruction Scheduling

by

STEVAN DUPOR

NCSU Honor Pledge: "I have neither given nor received unauthorized aid on this project."
Student's electronic signature: Stevan M Dupor
Course Number: 563

Introduction and Methods

This project involved implementing a valueless simulator for a superscalar out of order (OOO) processor. The simulator permits arbitrary-fixed-width execution throughout the pipeline, and simulates the movements of instructions from an instruction trace through said pipeline. As mentioned, the simulator is valueless, so no actual instructions are loaded – the intent is to study the effects of parameters like Reorder buffer (ROB) and issue queue (IQ) size on Instruction Per Cycle (IPC) as the permitted width is changed. Further simplifications include perfect branch prediction, and perfect cache performance such that no instructions need be squashed to handle a mispredicted branch, and all stalls result from head-of-line blocking in the pipeline, be this due to Read-after-write (RAW) hazards or true dependencies, limitations on the ROB/IQ size, or limitations on the pipeline width.

The simulator is implemented in C++ (2011 standard), and is implemented completely procedurally. Loading of traces into memory is abstracted from the pipeline simulation, and performed in a complete discrete step. Each phase of the pipeline is simulated by a section of a master function, and pipeline registers are simulated by STL vectors which hold pointers to instructions that are emplaced within these registers at any given time. The ROB and IQ are implemented as fixed-size circular arrays.

Numerous experiments across varying width, ROB size, and IQ size parameters were conducted across multiple traces, and plots of these results as well as discussion of the findings follow in proceeding sections. The experiment configuration/automation and data collection via comma-separated-value file were performed using C++ (code excluded from submission for convenience). The data analysis and plots were implemented in Python – code also excluded from submission for convenience.

Experiment A: Fixed, Large ROB, varying Width and IQ size

The first series of experiments were concerned with analysis of of a fixed-size ROB of 512 entries to eliminate the possiblity of full-ROB stalls. For each available trace (GCC and Perl), the IQ size was swept from 8 to 256 entries, across 1, 2, 4, and 8-wide superscalar pipelines. The results of these experiments, in the form of IPC curves, are displayed in plots below.

Figures of GCC and Perl Trace IPC w/ Varying IQ

The results of the GCC trace with varying IQ are displayed in Figure 1, and those for the Perl trace in Figure 2.

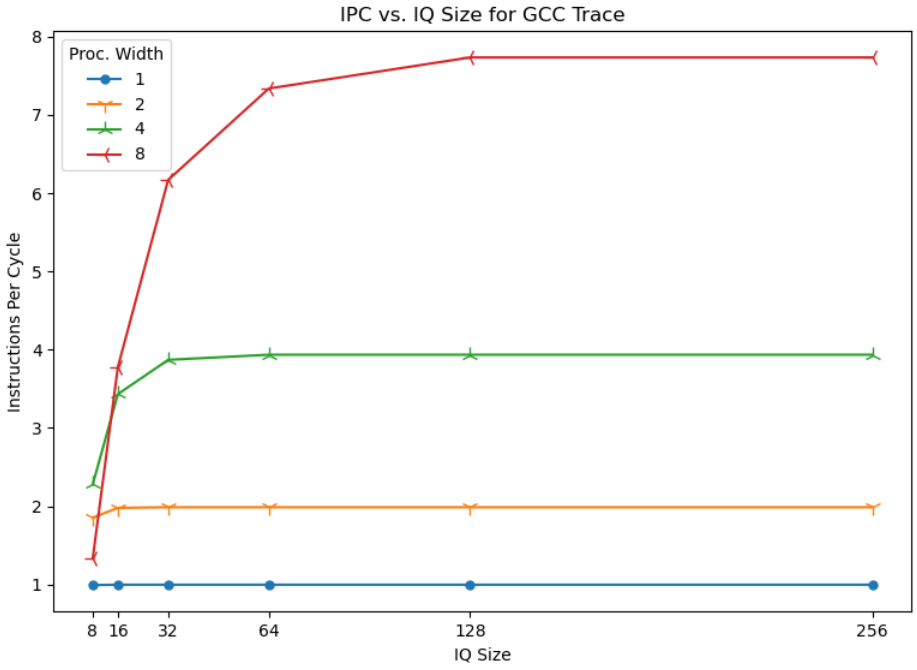

Figure 1: GCC trace, Varying IQ size

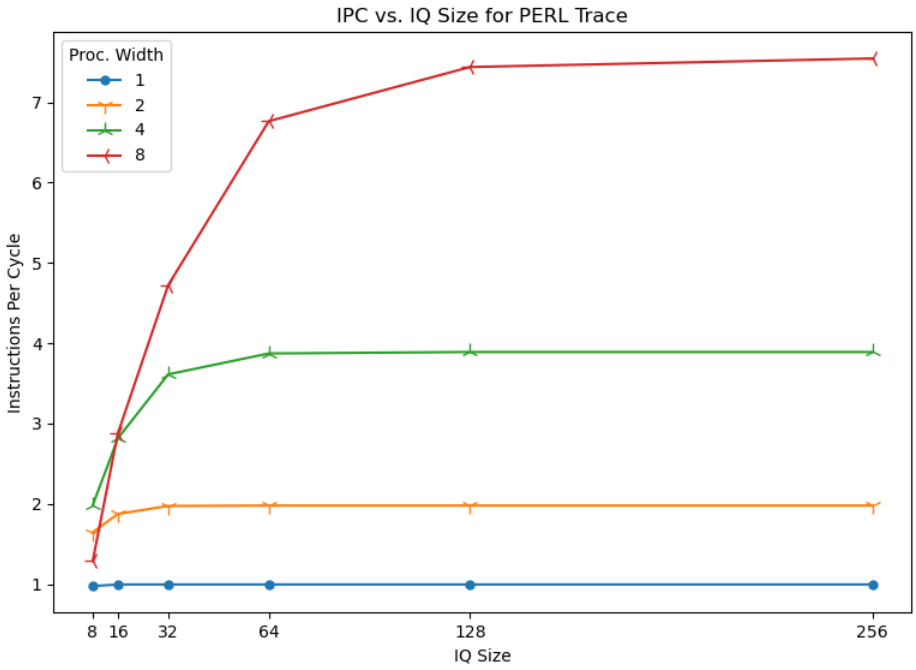

Figure 2: Perl trace, Varying IQ size

Discussion of GCC and Perl Trace IPC w/ Varying IQ

Notably, for both of these plots, all Instruction per Cycle curves approach an asymptote of approximately the **Width** of the curve in question. This is a logical limit on performance – with zero stalls in execution, we can expect to only ever execute WIDTH instructions simultaneously as there are no other processing units besides the maximum contained in WIDTH.

As a further point, I find that these curves quickly (exponentially) approach this asymptote, but the "room" for growth is much shorter as the width is smaller. In other words, a processor of WIDTH = 1 is already approximately at the asymptote at an IQ of 8, a processor of WIDTH = 2 at IQ=16, and so on. The logic that drives this trend is also fairly easy to derive – since the ROB will not bottleneck, and the only bottlenecks in the system are the WIDTH of the frontend stages and the size of the IQ: **when the width of the frontend stage is a more serious bottleneck than the size of the IQ, increasing the size of the IQ will not lead to higher performance**. In other words, instructions can only come into the IQ one per cycle at WIDTH=1, which leads to limited opportunities for bottlenecks to form in an IQ that is 8 or 16 times the size of the WIDTH. Further analysis of the benchmark traces is likely to reveal that the main drivers of this bottleneck are large "gluts" of multi-cycle instructions, which are significantly more rare (especially for 5 cycle/instruction length instructions) than single-cycle instructions. So, instructions which would be stuck in the IQ waiting for data from long multi-cycle instructions in the execute stage trends towards not bottlenecking as IQ grows beyond these critical points. The numeric results of these critical points are reviewed in Table 1.

A further point of discussion is the difference between these two plots for the same configuration (and differences between the critical points in the table). It is visually observable that the GCC trace benefits more from early growth in the IQ size, which means that at small sizes, for the GCC trace as compared to Perl, the IQ is the more serious bottleneck as compared to the Perl trace. Also, I note that the overall asymptotic performance of the GCC trace is slightly better than the Perl trace. I believe these points imply that the **GCC trace has more true dependencies between instructions, which means it benefits more from a larger IQ** (as the full pipeline need not stall while instructions wait for data to become ready in the ROB). However, regarding the second point about aysmptotic best-cases: I believe that the **Perl trace may contain statistically more multi-cycle instructions**, which cannot be forced to execute faster even with larger IQ/ROB buffer sizes.

In fact, upon running a simple instruction length count of the simulated traces, I am able to **confirm my prior claim that the Perl trace contains more slow 5-cycle instructions** than the GCC trace. The GCC trace contains 70.49 % single-cycle instructions, 10.2% 2-cycle instructions, and 19.29% 5-cycle instructions. Conversely, the Perl trace contains 65.86% single-cycle instructions, 9.21 % 2-cycle instructions, and 24.93%

5-cycle instructions. From this, it is clear that the tradeoff in the Perl trace is an increase of 5.64% in 5-cycle instructions as compared to GCC, with the shorter/faster instructions being inversely reduced.

Width	GCC Benchmark $\text{argmin}(IQ_{size}) = \{IPC_{width} \geq 0.95 \cdot IPC_{best}\}$	Perl Benchmark $\text{argmin}(IQ_{size}) = \{IPC_{width} \geq 0.95 \cdot IPC_{best}\}$
Width=1	8	8
Width=2	16	32
Width=4	32	64
Width=8	64[1]	128

Table 1: Optimized IQ_SIZE per WIDTH: The minimum IQ_Size that still achieves within 5% of the IPC of the largest IQ_SIZE

(1): The GCC Width=8 value of 64 was chosen because this value was within $\frac{2}{5}$s of one percent of the 5% cutoff, eg $p \leq 5.1357\%$, which is quite close to the whole-number threshold. This choice is especially attractive when considered in the light of die area and cost.

Experiment B: Fixed IQ, varying Width and ROB size

The second series of experiments were concerned with exploring the effects of the ROB size across different widths, for the same two traces. Notably, the "ideal" issue queue sizes from Table 1 were used for each Width, and the ROB sizes were swept from 32 to 512 entries in powers of two.

Plots of IPC for ROB Size Experiments

The plotted results from each of these traces are displayed in Figures 3 and 4, below.

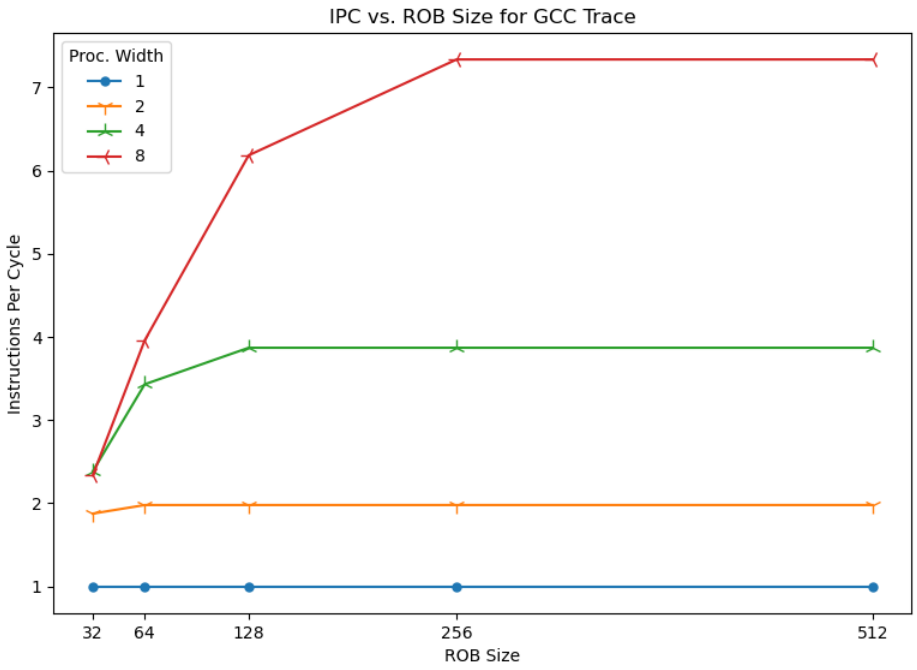

Figure 3: GCC trace, Varying ROB size

Discussion of ROB Size Experiments

In this series of experiments, I observe similar expected asymptotic trends; however, I note that the growth rate of IPC in the exponential sections is less than observed in the IQ experiments. In particular, the growth in the Perl trace is slower, which lends greater weight to the claim made in the prior discussion section that the Perl trace likely contains more multi-cycle instructions: These are more likely to stall at the ROB because this is the point in the pipeline where long-execute instructions will tend to "pile up."

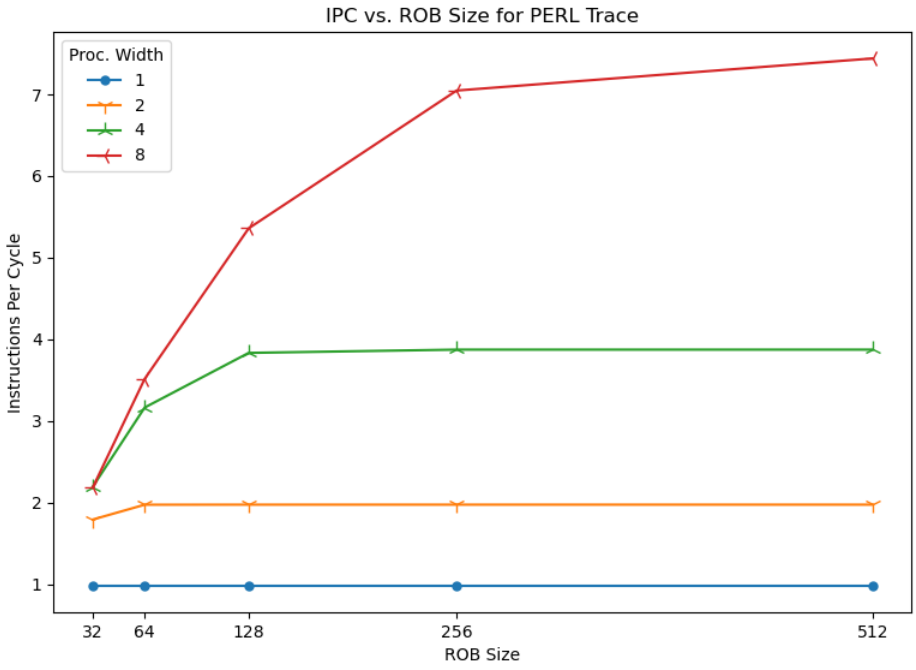

Figure 4: Perl trace, Varying IQ size

Conclusions and Roadmap

The implementation of the simulator and experiments were an excellent opportunity to reinforce the concepts of the scalar OOO pipeline, and the requirement of various bypasses to avoid deadlocks in the simulation. A natural roadmap progression for this project is to implement the support of several simplified or omitted features, namely: Imperfect caches with long cycle stalls, Imperfect branch prediction requiring in-pipeline instructions to be squashed, and fully-featured instructions with values. With these extensions, this simulator could perform more like an emulator, and many more parameters be studied by gathering execution statistics within the C++ code.