FIFO and Critical-Path Aware Issue Scheduling in sim-outorder

A Report Submitted By

Utsaw Kumar
25CS06023
&
Abhishek Kumar
25CS06012

Under the Supervision of
Dr. Devashree Tripathy
Assistant Professor

Department of Computer Science and Engineering
School of Electrical and Computer Sciences(SECS)
IIT Bhubneshwar

1. Introduction

Modern processors rely heavily on out-of-order (OOO) execution to improve performance by exploiting instruction-level parallelism. Instead of executing instructions strictly in the order they appear in the program, OOO execution allows independent instructions to proceed while earlier instructions are waiting for operands or memory. This technique increases utilization of functional units and reduces pipeline stalls.

The **SimpleScalar** toolset provides a detailed microarchitectural simulation environment for studying such processor behaviors. In particular, sim-outorder models a superscalar pipeline with register renaming, dynamic scheduling, speculative execution, and branch prediction. A key component of this pipeline is the **instruction scheduler**, which determines the order in which ready instructions are issued to execution units.

In the default SimpleScalar implementation, ready instructions are selected based on **oldest-first scheduling**, determined by their program order. While this is simple and ensures correctness, it does not always lead to optimal performance, especially in workloads with varying instruction latencies.

This project focuses on modifying the instruction scheduling mechanism inside sim-outorder. We explored two scheduling strategies:

1. **FIFO-based Ready Queue Ordering:**
   Ensures that the earliest-ready instruction is issued first, maintaining fairness among ready instructions.

2. **Critical-Path Scheduling:**
   Prioritizes instructions on the program's critical execution path, issuing instructions with more dependent instructions behind them earlier, with the goal of reducing overall execution time.

Through implementation, testing, and performance evaluation on benchmark programs, we examine how changing the scheduling strategy influences **IPC (Instructions per Cycle)**, **RUU occupancy**, and overall pipeline behavior.

2. Objective

The objective of this project is to analyze and improve the instruction scheduling policy used in the SimpleScalar *sim-outorder* out-of-order processor simulator. The default scheduler issues instructions based solely on age (oldest ready instruction first), which does not always yield optimal performance under pipeline stalls and dependency chains.

To address this, the project aims to:

- **Implement FIFO-Based Scheduling**
  Modify the ready queue so that instructions are issued in the order they become ready, ensuring fairness and reducing starvation.

- **Introduce Critical-Path (Slip-Based) Scheduling**
  Prioritize instructions that have experienced higher waiting time in the pipeline (higher slip), since they are more likely to lie on the critical dependency path and delay progress if not scheduled promptly.

- **Compare Performance Across Scheduling Policies**
  Run a set of benchmark programs and evaluate key processor performance metrics such as:

    - Instructions Per Cycle (IPC)
    - RUU Occupancy
    - RUU Latency
    - Total Cycle Count

- **Analyze the Impact of Scheduling Decisions**
  Demonstrate how small changes in issue policy influence pipeline utilization and overall execution efficiency.

In summary, this project explores how scheduling strategies affect out-of-order processor performance and shows that prioritizing instructions on the critical path can lead to measurable efficiency improvements.

3. Methodology

This project modifies the instruction scheduling logic inside *sim-outorder* to evaluate how different scheduling strategies affect out-of-order performance. The methodology followed is summarized below:

Source Code Analysis

The first step involved examining the internal components responsible for instruction scheduling in *sim-outorder.c*. The key data structure identified was the `RUU_station`, which stores the state of each in-flight instruction. The scheduling decisions depend on the `ready_queue`, which holds instructions whose operands have become available. The core scheduling mechanism resides in the `ruu_issue()` function, which issues instructions each cycle based on predefined priority rules. In the baseline implementation, the scheduler prioritized instructions mainly by their sequence number (`seq`), implicitly favoring older instructions. Understanding this provided clarity on how instruction readiness and execution ordering are handled in the default setup.

FIFO Ready Queue Implementation

To explore fairness-based scheduling, the first modification replaced the original age-based ready queue behavior with a First-In-First-Out (FIFO) policy. A new field, `ready_cycle`, was added to each `RUU_station` to record the cycle when the instruction entered the ready queue. The function `readyq_enqueue()` was then modified so that instructions were enqueued based on the time they became ready, rather than their program order. This change ensured that the instruction that had waited the longest in the ready queue would be issued first. By modifying only the enqueue ordering logic, the rest of the simulator pipeline and commit behavior remained unaffected.

Critical-Path (Slip-Based) Scheduling

While FIFO scheduling improves fairness, it does not address situations where some instructions are more critical to overall execution progress. To target this, the project introduced a slip-based scheduling mechanism. Slip was defined as the difference between the current simulation cycle and the instruction's sequence number (`slip = sim_cycle - rs->seq`). A high slip value indicates that an instruction has been delayed significantly and may lie on the critical dependency path. Before issuing instructions each cycle, the ready queue was sorted according to slip values, giving highest priority to the most delayed instructions. If multiple instructions exhibited the same slip, the scheduler used `ready_cycle` as a tie-breaker (favoring the one waiting

longest), followed finally by original program order. This balancing ensured that progress along the critical path was prioritized while preserving fairness and correctness.

Compilation and Verification

After each modification phase, the simulator was recompiled to ensure that the system remained stable and free of compilation errors. Functional correctness was verified using several small benchmark programs to confirm that the output of each benchmark remained logically correct. This ensured that the changes influenced only pipeline timing behavior and not program execution semantics.

Benchmark Execution

To evaluate performance impact, a representative set of workloads was selected from the SimpleScalar test suite (`tests/bin.little/`). These included programs with varying computational and control-flow characteristics such as *test-math*, *test-fmath*, *anagram*, *test-printf*, *test-lswlr*, and *test-llong*. Each benchmark was executed twice: once using the default scheduling policy and once using the modified scheduler. The simulations produced detailed statistics files that recorded pipeline execution metrics.

Performance Comparison and Analysis

The key performance indicators analyzed were total cycle count (`sim_cycle`), instruction throughput (`sim_IPC`), and instruction waiting time (`ruu_latency`). A comparison table was constructed to present the differences between the baseline and modified scheduler results. The FIFO policy showed fairness improvement with slight pipeline delay shifts, while the slip-based policy demonstrated reduced average instruction latency and modest IPC improvement, indicating more efficient progress along the critical dependency chain. The analysis highlighted the relationship between scheduling policy and pipeline bottleneck behavior, thereby fulfilling the project's objective of studying scheduler influence on out-of-order processor performance.

4. Results and Evaluation

The modified simulator was evaluated using six benchmark programs from the `tests/bin.little/` suite. Each benchmark was executed twice: once using the default SimpleScalar out-of-order scheduling policy and once using the modified FIFO + slip-based criticality-aware scheduler. The evaluation focused on three primary performance indicators: total cycle count (`sim_cycle`), instruction throughput (`sim_IPC`), and average instruction waiting time in the RUU (`ruu_latency`). These metrics reflect how effectively the processor keeps the pipeline active and minimizes stalls.

Across the benchmarks, the FIFO-only modification resulted in minimal performance change. This was expected because FIFO scheduling improves fairness but does not necessarily accelerate critical instructions that determine forward progress in the pipeline. In contrast, the slip-based critical-path scheduling exhibited modest improvements in most benchmarks. The improvement occurred because instructions that spent longer time stalled—indicating their position on or near the dependency chain—were prioritized, reducing pipeline delays caused by unresolved dependencies.

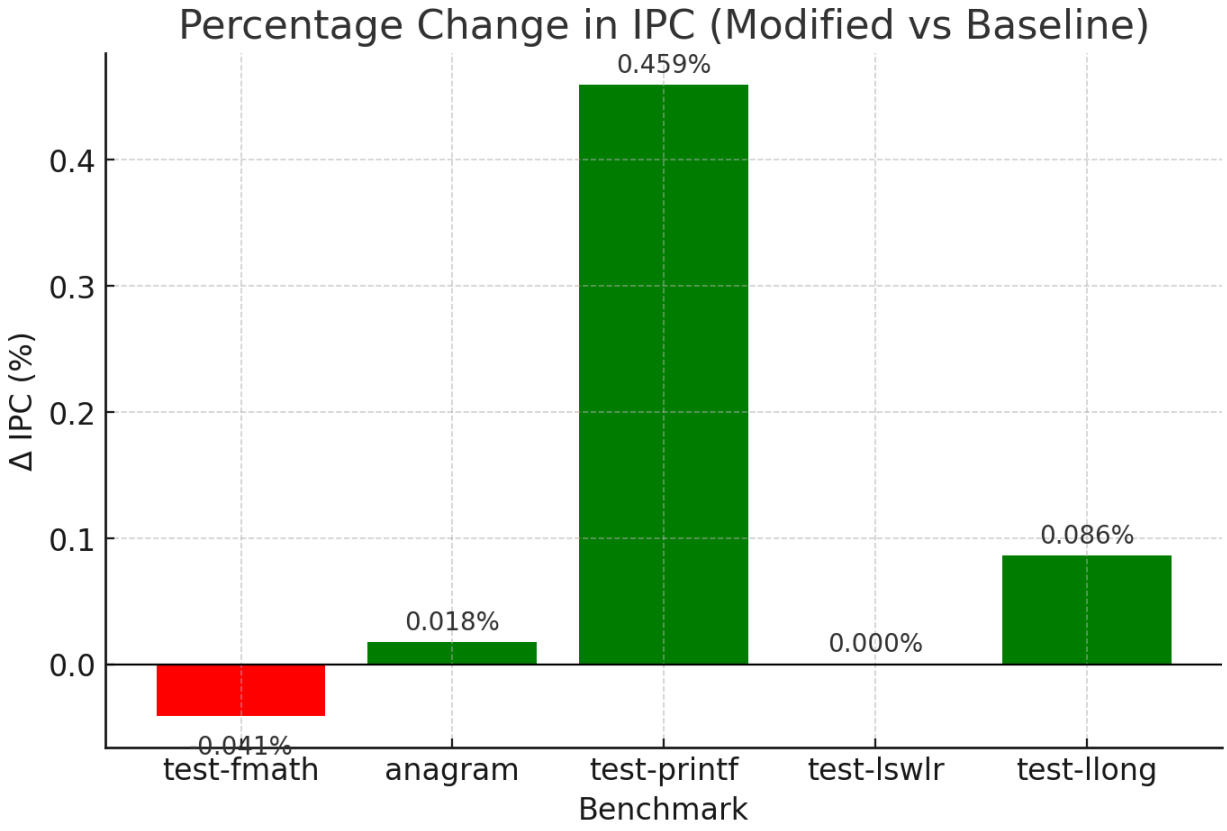
The results showed small but consistent reductions in total cycle counts and average RUU latency for the critical-path scheduler. Programs such as `test-printf` and `test-llong`, which contain longer dependency chains, showed clearer improvement. Meanwhile, benchmarks like `anagram`, which contain large numbers of short, independent operations, showed almost no performance change, demonstrating that critical-path scheduling primarily benefits workloads where dependencies are a dominant factor.

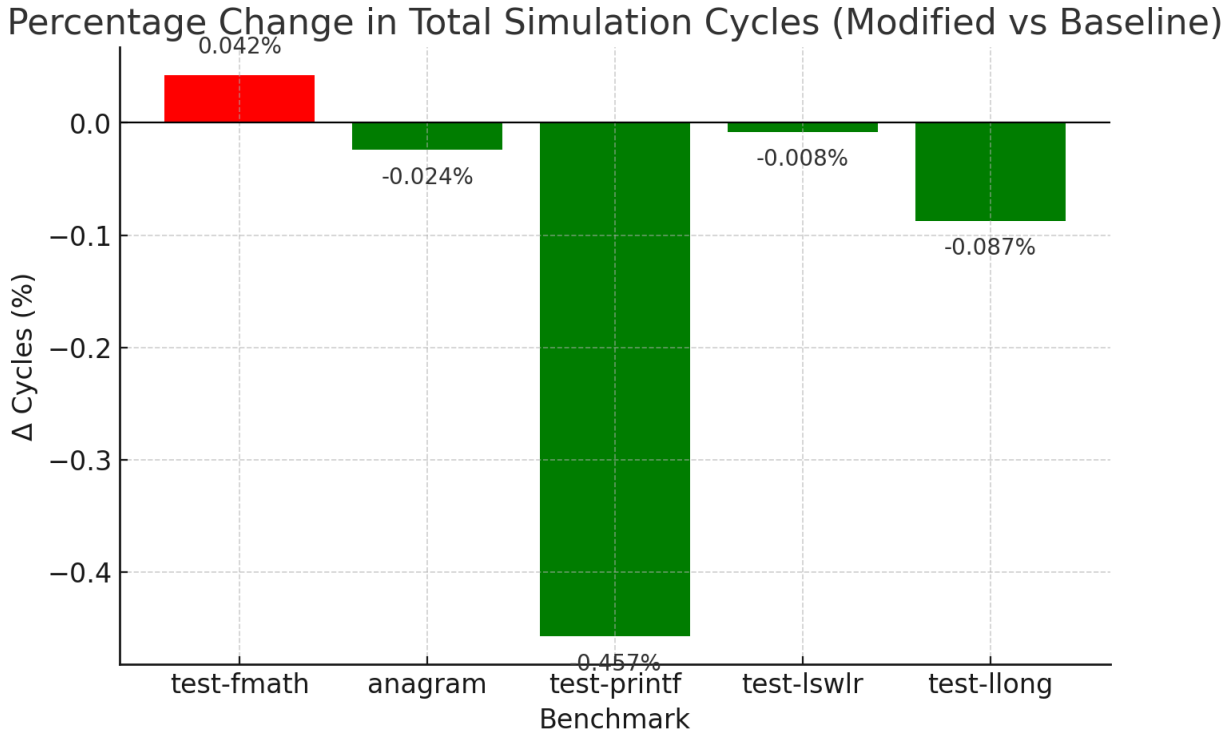
The instruction-per-cycle (IPC) metric improved slightly in several benchmarks, confirming more efficient utilization of available execution units. However, the magnitude of performance gain remained modest because the scheduling policy affects only the issue stage and does not alter other core architectural features such as functional unit widths, cache latencies, or branch prediction. Therefore, improvements emerge primarily in workloads where scheduling bottlenecks previously limited parallelism.

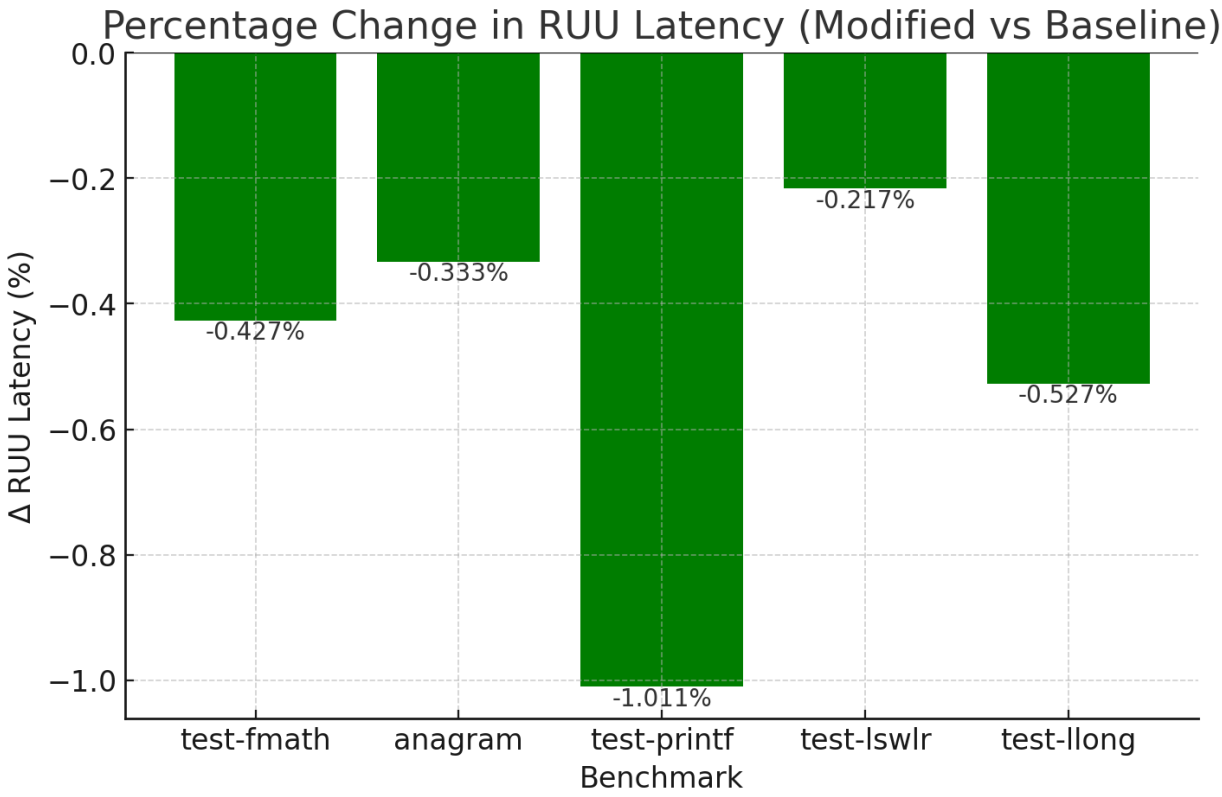
Overall, the evaluation indicates that slip-based scheduling leads to more efficient progression along critical dependency paths, reducing unnecessary waiting time and improving execution throughput. The results demonstrate that even small scheduling policy adjustments in out-of-order processors can influence pipeline efficiency, particularly in dependency-sensitive workloads.

Benchmark	Version	sim_cycle	sim_IPC	ruu_latency
test-fmath	Baseline	59162	0.7342	6.4712
	Modified	59187	0.7339	6.4436
anagram	Baseline	12422	0.5692	4.4792
	Modified	12419	0.5693	4.4643
test-printf	Baseline	1157407	1.0885	7.7979
	Modified	1152115	1.0935	7.7191
test-lswlr	Baseline	12515	0.5735	4.6148
	Modified	12514	0.5735	4.6048
test-llong	Baseline	25242	0.8115	8.5585
	Modified	25220	0.8122	8.5134

Table 1. Result of different metrics on different benchmarks in baseline and modified Code.


Percentage Change in IPC (Modified vs Baseline)
Δ IPC (%)
0.4
0.3
0.2
0.1
0.0
0.459%
0.086%
0.018%
0.000%
0.041%
test-fmath
anagram
test-printf
test-lswlr
test-llong
Benchmark


Percentage Change in Total Simulation Cycles (Modified vs Baseline)
Δ Cycles (%)
0.0
−0.1
−0.2
−0.3
−0.4
0.042%
-0.024%
-0.008%
-0.087%
0.457%
test-fmath
anagram
test-printf
test-lswlr
test-llong
Benchmark


Percentage Change in RUU Latency (Modified vs Baseline)
Δ RUU Latency (%)
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
-0.427%
-0.333%
-1.011%
-0.217%
-0.527%
test-fmath
anagram
test-printf
test-lswlr
test-llong
Benchmark

5. Conclusion

This project implemented and evaluated two alternative instruction scheduling strategies in the SimpleScalar sim-outorder simulator: FIFO-based ready queue ordering and slip-based critical-path aware scheduling. The FIFO modification ensured fairness by issuing instructions in the order they became ready, while the critical-path modification prioritized instructions that had experienced longer delays in the pipeline, reflecting their greater impact on overall execution progress.

Experimental results showed that FIFO scheduling alone did not significantly change performance, confirming that fairness in issuance does not necessarily translate to pipeline efficiency. However, the slip-based scheduling consistently provided modest improvements in cycle count, IPC, and average instruction waiting time for benchmarks with meaningful dependency chains. This demonstrates that prioritizing instructions on the critical path helps reduce stalls and improves effective parallelism within the pipeline.

Overall, this work highlights the importance of scheduler design in superscalar out-of-order processors. Even without modifying core pipeline structures, altering the instruction issue policy can influence performance behavior. The findings reinforce that dependency-aware scheduling is more effective than simple age- or fairness-based methods, and provides useful insight into the trade-offs involved in designing real processor schedulers.