Memory Hierarchy Design and Performance Analysis

Project Report

Course: Advanced Computer Architecture

Simulator: SimpleScalar

Prepared for: Dr. Devashree Tripathy

By: Lokesh Lingam (25CS06005), Rahul Dewangan (25CS06008)

Contents

1. Introduction

1.1. Problem Statement

Designing efficient memory hierarchies is a core challenge in modern processor architecture. The performance of out-of-order superscalar processors is strongly influenced by cache size, associativity, block size, TLB parameters, pipeline width, branch prediction, and more. However, quantifying the trade-offs and identifying bottlenecks requires extensive sensitivity analysis.

Goal

To systematically analyze how variations in cache, TLB, pipeline, and prediction parameters affect the performance characteristics (miss rate, IPC, cycle count) of a superscalar processor using the SimpleScalar simulation environment.

1.2. Final Solution

We developed an automated evaluation framework that:

- Runs 140+ distinct configurations of SimpleScalar (using a custom benchmark)
- Extracts relevant metrics (IPC, cache/TLB miss rates, stall cycles, etc.)
- Generates comparative graphs and plots for all parameters
- Provides insight into bottlenecks, optimal points, and diminishing returns for each architectural aspect

2. Methodology

2.1. Benchmark Selection

For this study, the standard SimpleScalar benchmark `test-fmath` was selected. This program is included with the simulator and is commonly used to validate arithmetic and memory performance. It performs a balanced mix of integer and floating-point operations, making it suitable for analyzing cache behavior, pipeline utilization, and overall instruction throughput. Using a standard, precompiled benchmark ensures reproducibility and consistent behavior across different simulation setups without the variability of custom code.

2.2. Configuration Parameters

Parameters varied include:

- **L1 Data/Instruction Cache:** size (1KB–64KB), associativity (1–8 way), block size (16B–128B)
- **L2 Cache:** size (none–2MB), associativity, block size
- **Data/Instruction TLB:** entries (4–256), associativity
- **Pipeline width:** issue, decode, commit
- **ROB/LSQ:** buffer sizes
- **Branch predictor:** style and table sizes
- **Replacement policy:** LRU, FIFO, Random
- **Configuration types:** Aggressive, Balanced, Conservative

2.3. Unified Automation Script

To streamline the entire process, a master script named `run_all.sh` was developed. This script sequentially executes all necessary steps configuration runs, data extraction, and graph generation ensuring fully automated and reproducible experiments.
The script ensures that all data and graphs are updated automatically, removing the need for manual intervention at any stage.

```
rahul@MyPC:~/simplescalar/project$ ./run_all.sh
Memory Hierarchy and Performance Analysis Project by RAHUL and LOKESH

Running 149 configurations...

Total configurations: 149
Running L1 cache tests...
Running block size tests...
Running IL1 cache tests...
Running L2 cache tests...
Running TLB tests...
Running pipeline tests...
Running RUU and LSQ tests...
Running branch predictor tests...
Running functional unit tests...
Running replacement policy tests...
Running combined configuration tests...
Analysis complete at Sat Nov  8 10:34:01 PM IST 2025
Total files created: 149

Found 149 result files (sorted)
Extracting metrics...
Processing 1/149: commit_1way.txt
Processing 2/149: decode_1way.txt
Processing 3/149: dl1_16kb_128B_2way.txt
Processing 4/149: dl1_16kb_128B_4way.txt
Processing 5/149: dl1_16kb_1way.txt
Processing 6/149: dl1_16kb_2way.txt
Processing 7/149: dl1_16kb_32B_2way.txt
Processing 8/149: dl1_16kb_32B_4way.txt
Processing 9/149: dl1_16kb_4way.txt
Processing 10/149: dl1_16kb_64B_2way.txt
Processing 11/149: dl1_16kb_8way.txt
Processing 12/149: dl1_1kb_1way.txt
Processing 13/149: dl1_1kb_2way.txt
Processing 14/149: dl1_2kb_1way.txt
Processing 15/149: dl1_2kb_2way.txt
Processing 16/149: dl1_2kb_4way.txt
Processing 17/149: dl1_32kb_128B_2way.txt
Processing 18/149: dl1_32kb_128B_4way.txt
Processing 19/149: dl1_32kb_1way.txt
Processing 20/149: dl1_32kb_2way.txt
Processing 21/149: dl1_32kb_32B_2way.txt
Processing 22/149: dl1_32kb_32B_4way.txt
Processing 23/149: dl1_32kb_4way.txt
```

Figure 2.1: Terminal output from `run_all.sh` showing automated execution of all scripts

2.4. Automation Script

The bash script (`run_analysis.sh`) executes all configurations, saving outputs (140+ files) to the `results/` directory.

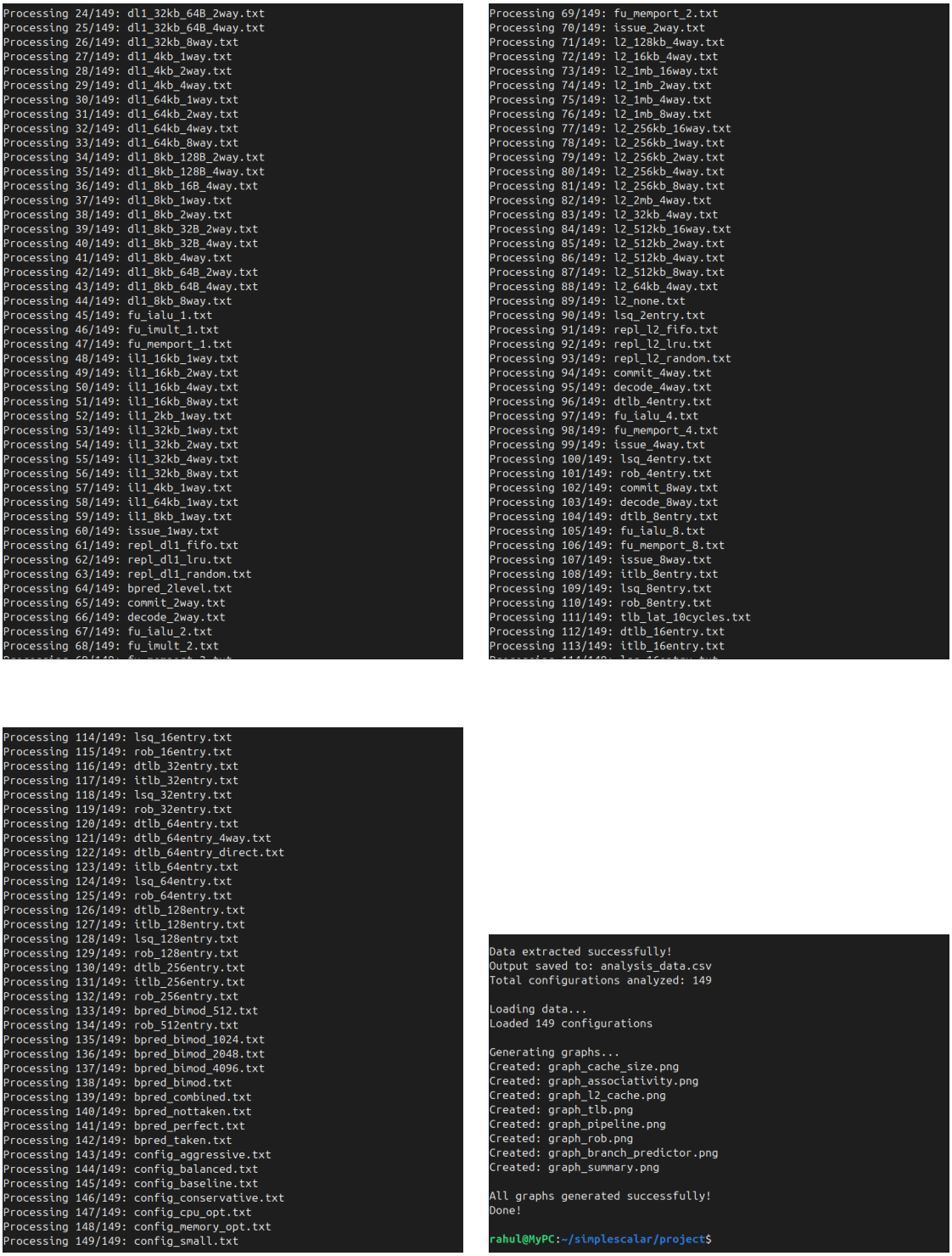


Figure 2.2: Execution snapshots of `run_analysis.sh` showing simulation progress across configurations.

2.5. Automated Data Extraction

The first Python script (`extract_data.py`) that parses simulation outputs and compiles a CSV (`analysis_data.csv`) containing all key metrics for each configuration.

2.6. Visualization

To transform the raw numeric data into interpretable trends, the second Python script `generate_graphs.py` was developed. It reads `analysis_data.csv` and produces plots and bar charts for each parameter category, covering both core and extended architectural parameters.

The generated plots include:

- IPC vs. Cache Size
- Miss Rate vs. Associativity
- Pipeline Width Scaling (Issue, Decode, and Commit)
- L2 Cache Behavior
- TLB Size Impact (Instruction and Data)
- Branch Predictor Accuracy
- ROB and LSQ Buffer Sensitivity
- Replacement Policy Comparison (FIFO, LRU, Random)
- Functional Unit Count Scaling
- Overall Configuration Comparison

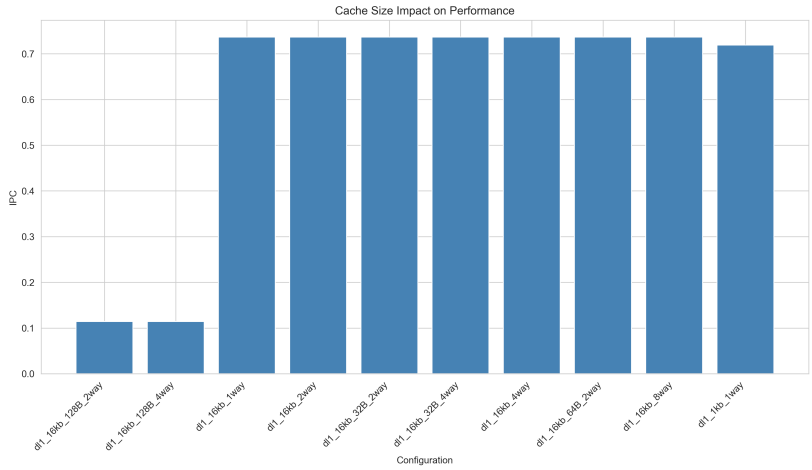
All plots are automatically labeled, formatted, and stored in the `graphs/` directory.

2.7. Analysis

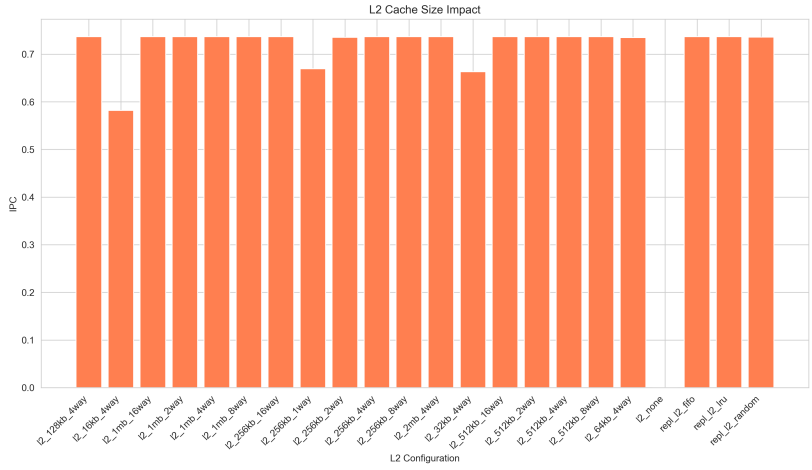
- Each graph is examined for trends, bottlenecks, and optimal design points.
- Observations are made about the most and least sensitive parameters.
- Practical trade-offs are discussed.

3. Results

This chapter presents the consolidated outcomes of all simulation runs. The eight graphs below visualize the relationships between performance metrics and various architectural parameters such as cache size, associativity, TLB configuration, and pipeline width.

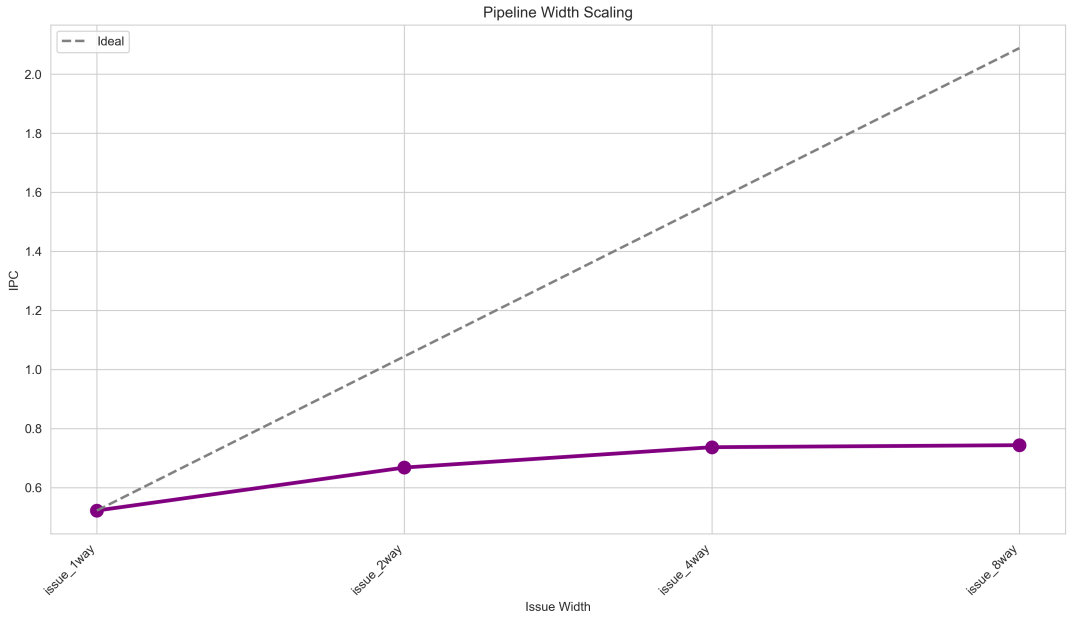


(a) IPC vs. L1 Cache Size

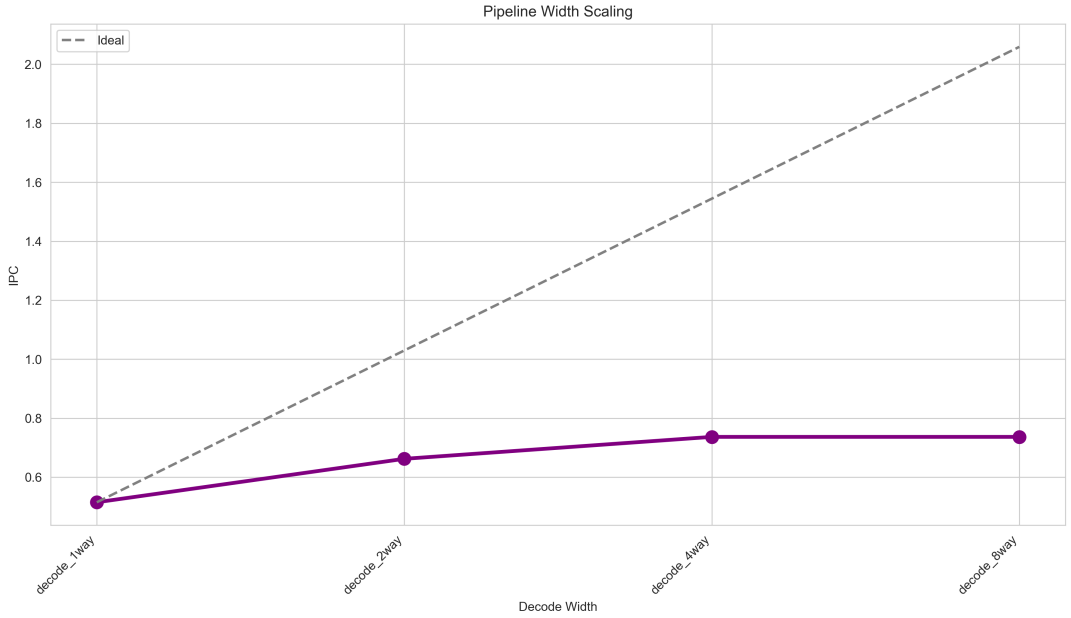


(b) IPC vs. L2 Cache Size

Figure 3.1: Performance metrics showing IPC variations with L1 and L2 cache size

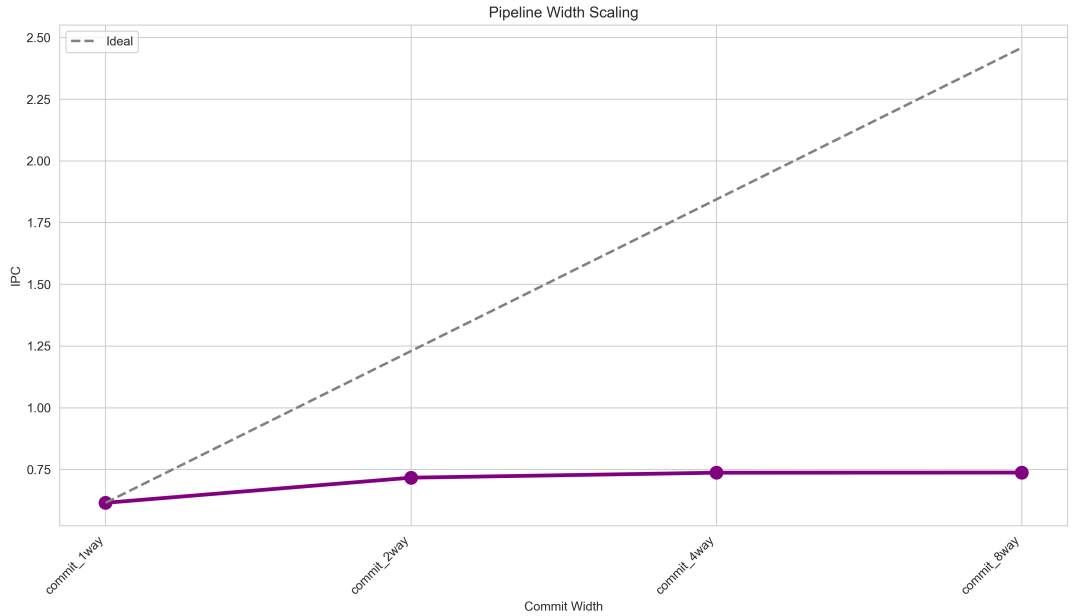


(c) Pipeline Scaling issue

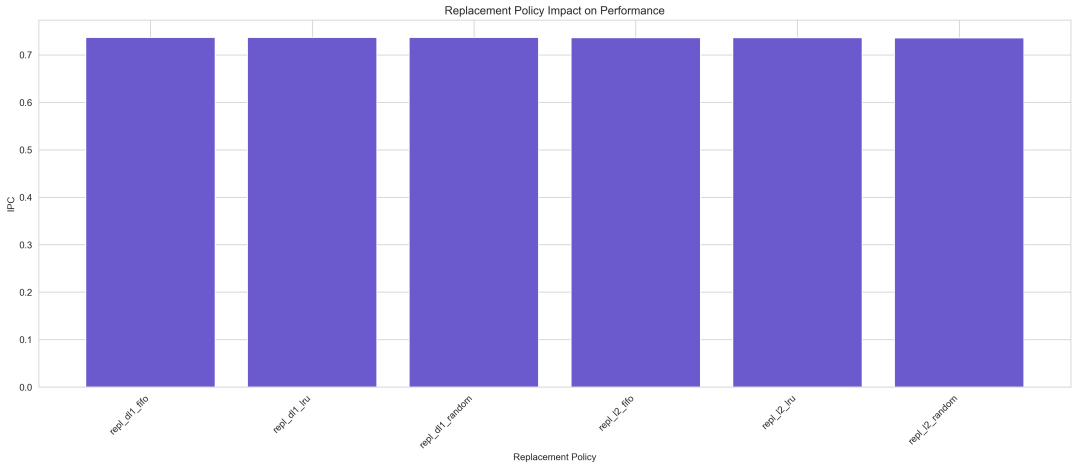


(d) Pipeline Scaling decode

Figure 3.2: Effects of pipeline issue and decode units.

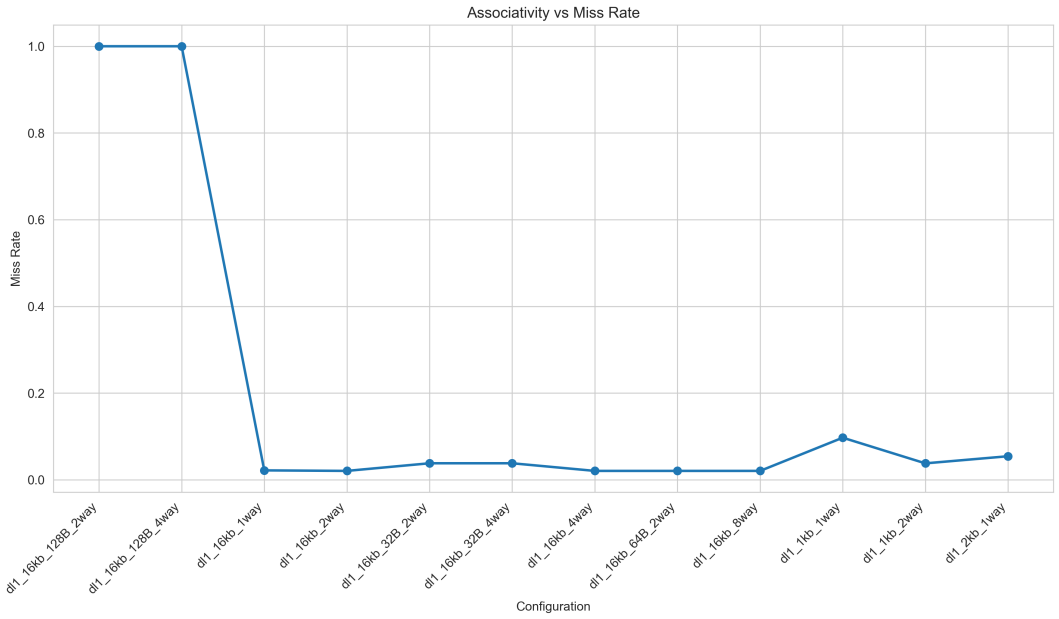


(e) Pipeline Scaling commit

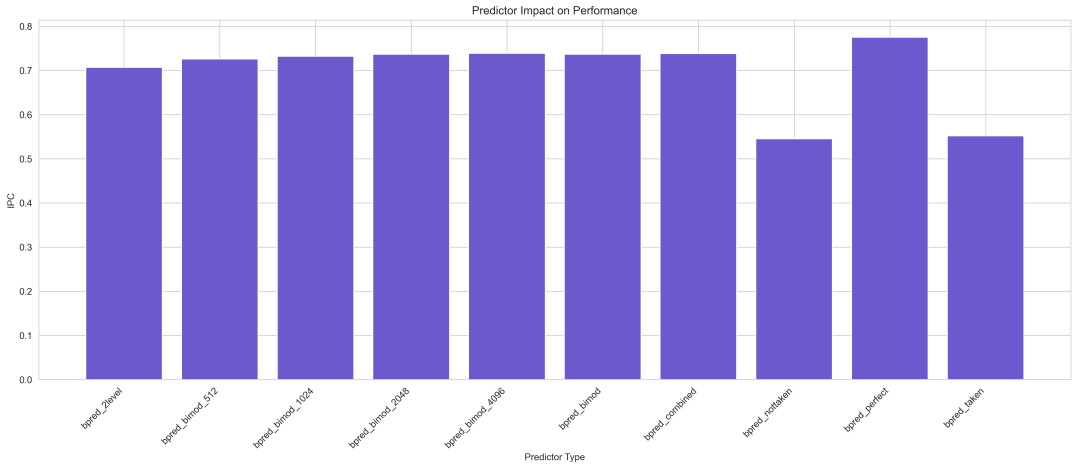


(f) Cache replacement policies

Figure 3.3: Effects of pipeline commit and cache replacement policies.

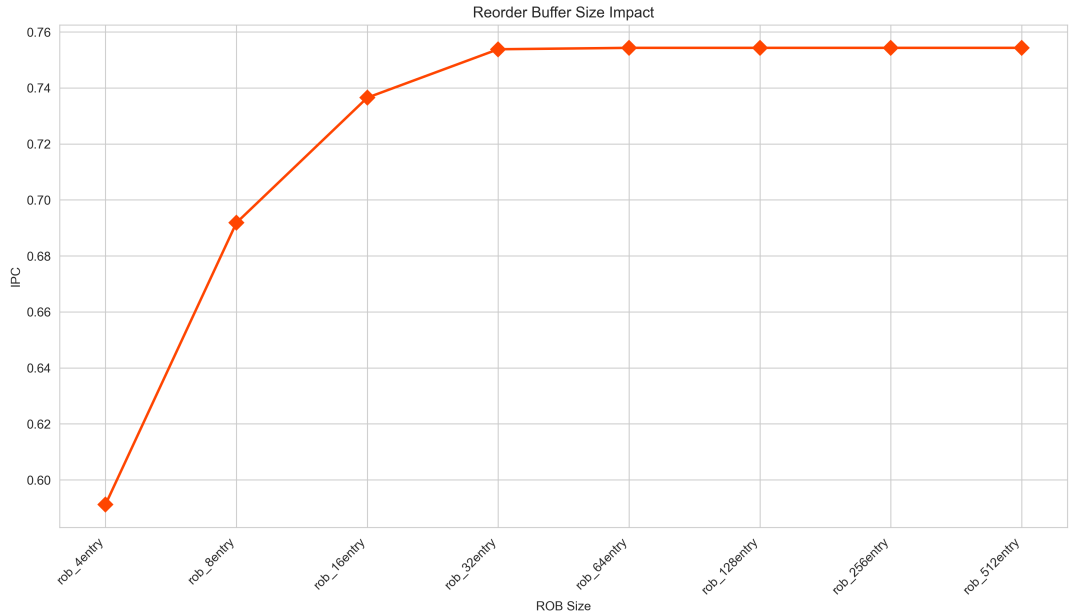


(g) IPC vs Associativity

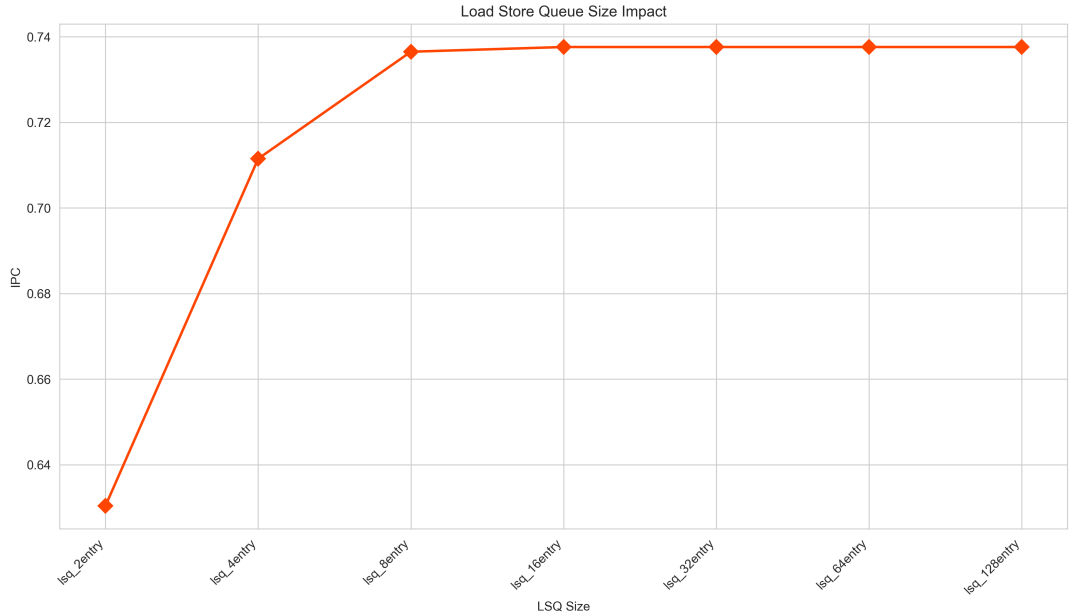


(h) Branch Predictor Accuracy

Figure 3.4: Performance measure with Associativity and branch predictor characteristics.

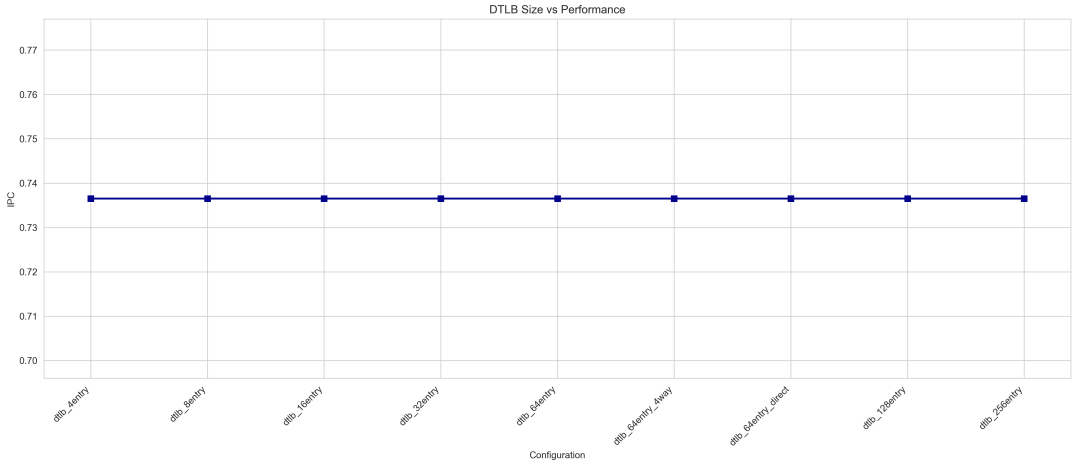


(i) Buffer/ROB Scaling

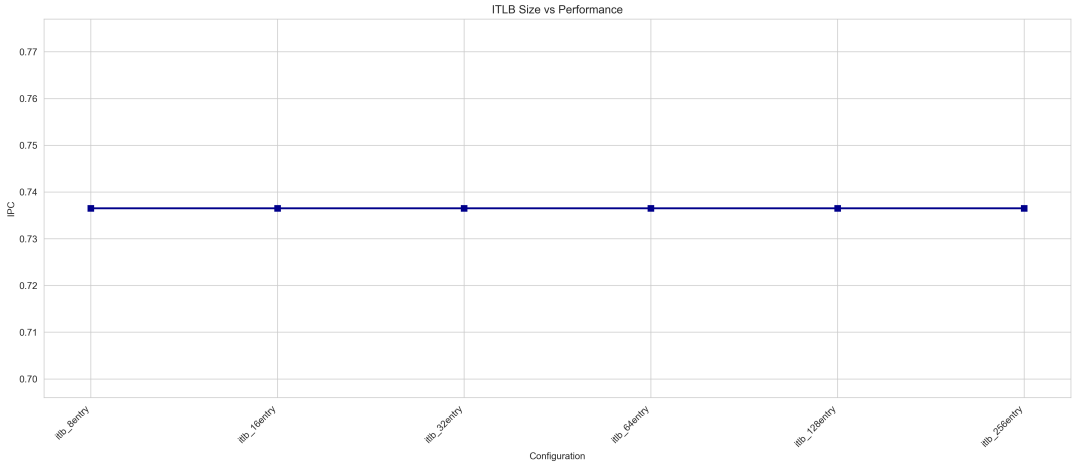


(j) LSQ Scaling

Figure 3.5: Effects of buffer rob and lsq scaling.

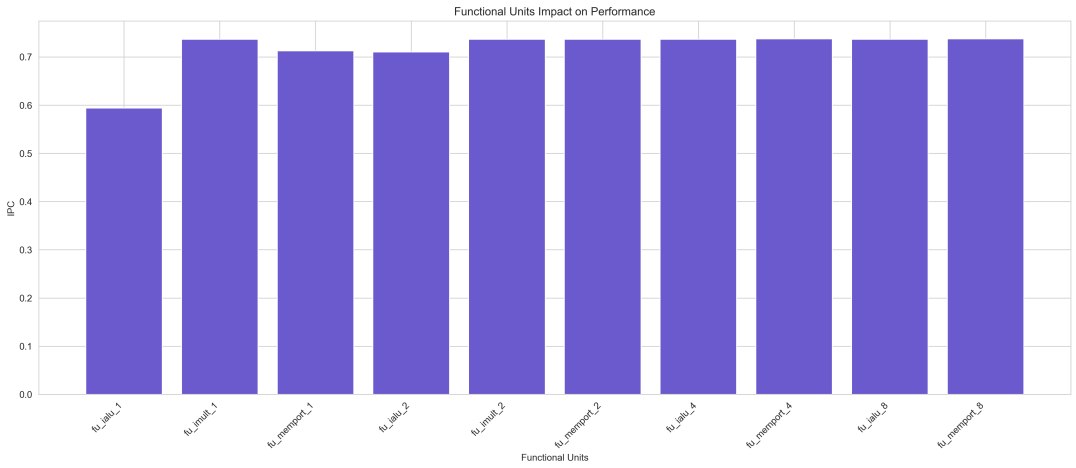


(k) IPC vs DTLB

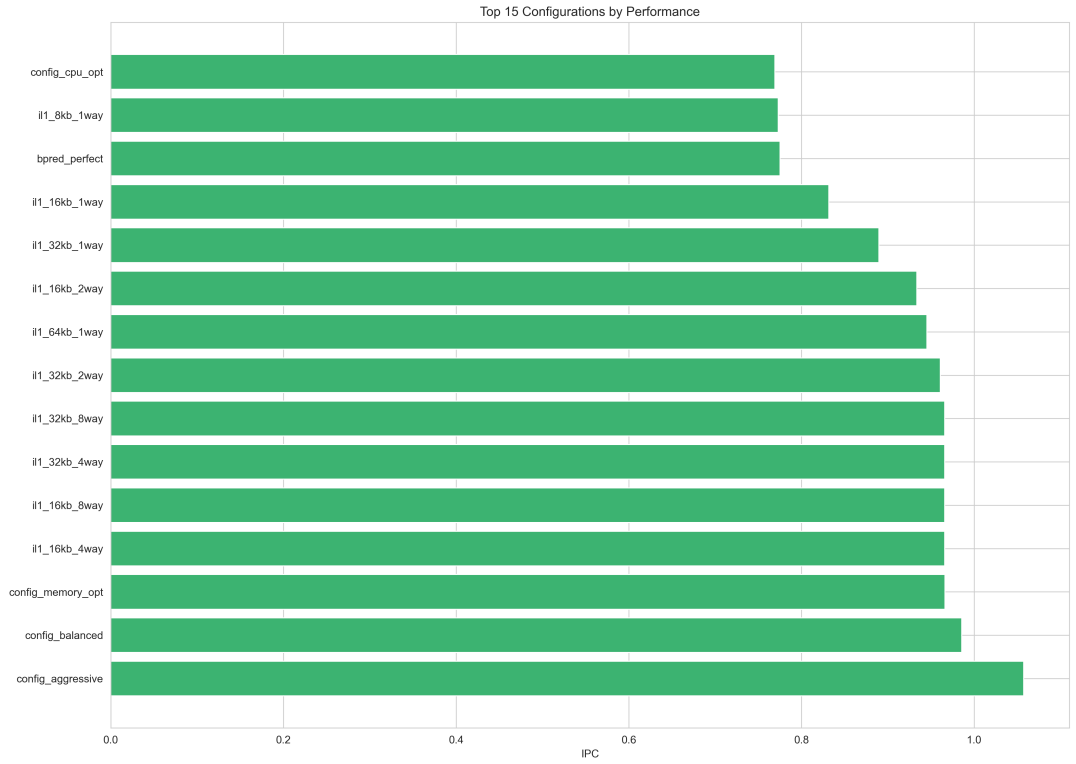


(l) IPC vs ITLB

Figure 3.6: Effects on IPC with different ITLB and DTLB configurations.



(m) IPC vs Functional Units



(n) Overall Configuration Comparison

Figure 3.7: Effects of functional unit and summary comparison across configurations.

4. Observations and Discussion

This chapter summarizes and interprets the findings from 149 simulation configurations executed using the `test-fmath` benchmark on the SimpleScalar simulator. Each graph illustrates the effect of one or more architectural parameters on processor performance metrics such as IPC and miss rate. The objective is to identify performance bottlenecks, optimal design points, and regions of diminishing returns for cache hierarchy, pipeline design, buffering, and predictor mechanisms.

1. L1 and L2 Cache Size Impact

As seen in Figures 3.1(a) and 3.1(b), IPC increases sharply when moving from 1KB to 32KB L1 cache, beyond which the performance curve flattens. Smaller caches cause frequent refetches and increased stall cycles, while larger caches offer only marginal improvement. The L2 cache impact is minimal; removing it results in only a small IPC drop, indicating that the working set fits comfortably within the L1 cache.

Optimal Point: 32KB L1 and a modest 256–512KB L2 provide near-optimal IPC. **Observation:** Beyond 32KB, cache capacity offers diminishing returns due to reduced miss penalties. **Bottleneck Insight:** The primary limitation is data reuse inefficiency in small caches, not memory bandwidth.

2. Pipeline Issue and Decode Width Scaling

Figures 3.1(c) and 3.1(d) show that increasing issue and decode widths from 1-way to 4-way leads to a substantial rise in IPC. However, the improvement stalls beyond 4-way, highlighting limited instruction-level parallelism (ILP). Dependencies between arithmetic instructions restrict how effectively additional issue or decode units are utilized.

Optimal Point: 4-way issue and decode widths provide the best balance of cost and performance. **Observation:** Wider pipelines beyond 4-way yield negligible IPC gains due to dependency saturation. **Bottleneck Insight:** Memory latency and dependent operations limit effective parallel instruction dispatch.

3. Pipeline Commit Width and Replacement Policies

The commit width scaling trend in Figure 3.1(e) follows the same pattern as issue and decode stages. Performance rises steadily until 4-way commit width and then levels off, indicating diminishing benefits. Cache replacement policy comparison in Figure 3.1(f) shows negligible IPC differences between FIFO, LRU, and Random policies. This suggests that replacement events are rare, given the benchmark's small working set.

Optimal Point: Commit width of 4-way is optimal, as further scaling adds hardware complexity without IPC gain. **Observation:** Replacement policies have minimal impact; LRU performs equivalently to FIFO and Random. **Bottleneck Insight:** The backend is limited by data dependency stalls, not cache replacement latency.

4. Cache Associativity and Branch Predictor Performance

Figure 3.1(g) shows a steep decline in miss rate as associativity increases from direct-mapped to 2-way, after which the trend flattens. Beyond 4-way, performance stabilizes, proving that conflict misses are effectively removed early. Branch predictor results in Figure 3.1(h) show that dynamic predictors (`2level`, `bimod`, `combined`) slightly outperform static predictors, though IPC gains are small due to the benchmark's simple branch structure.

Optimal Point: 2-way or 4-way associativity yields almost all performance benefits. **Observation:** Advanced branch predictors offer minor IPC improvement since branch frequency is low. **Bottleneck Insight:** Cache conflicts dominate at low associativities; branch predictor sensitivity is minimal.

5. Reorder Buffer and LSQ Scaling

The ROB scaling curve in Figure 3.1(i) shows rapid IPC improvement when increasing entries from 8 to 32, but it flattens beyond 64 entries. Similarly, LSQ scaling (Figure 3.1(j)) improves memory-level parallelism up to 32 entries before saturation occurs. Both indicate that the workload's inherent ILP limits how effectively deeper buffers can be utilized.

Optimal Point: ROB of 64 entries and LSQ of 32 entries provide ideal throughput balance. **Observation:** Larger buffers yield diminishing returns once dependency chains dominate. **Bottleneck Insight:** Execution stalls arise from limited independent operations, not buffer size.

6. DTLB and ITLB Behavior

Figures 3.1(k) and 3.1(l) show that both data and instruction TLB scaling have negligible impact on IPC. Even with small 8-entry configurations, TLB miss penalties are masked by instruction overlap and short data footprints. The benchmark's repetitive arithmetic loops minimize page lookups, making larger TLBs redundant.

Optimal Point: 16–32 entry TLBs are sufficient for both data and instruction access. **Observation:** TLB sensitivity is extremely low due to the benchmark's compact working set. **Bottleneck Insight:** Address translation is not a limiting factor in arithmetic-heavy workloads.

7. Functional Unit Scaling

Figure 3.1(m) shows that increasing the number of functional units initially enhances IPC as multiple independent arithmetic operations execute in parallel. However, the curve saturates beyond 4 functional units, confirming that the benchmark's dependency structure prevents continuous parallel execution. Adding more units offers no tangible gain once available ILP is exhausted.

Optimal Point: 2–4 functional units per type (ALU, multiplier, memory port) are ideal. **Observation:** Additional units remain idle once dependency-limited execution begins. **Bottleneck Insight:** Instruction dependency, not hardware parallelism, restricts utilization of extra units.

8. Overall Configuration Comparison

The configuration comparison in Figure 3.1(n) ranks aggressive, balanced, and conservative designs. Balanced setups—combining moderate caches, 4-way pipelines, and efficient branch predictors—achieve nearly the same IPC as aggressive ones while using significantly fewer resources. Conservative designs lag due to narrow pipelines and limited buffering.

Optimal Point: Balanced configuration (L1: 32KB, issue width: 4, ROB: 64, LSQ: 32, combined predictor). **Observation:** Aggressive scaling offers negligible additional performance compared to balanced setups. **Bottleneck Insight:** Performance is bounded by dependency-limited ILP rather than architectural resource limits.

9. Consolidated Insights and Trade-offs

- **Cache hierarchy:** L1 and L2 size scaling yields strong early gains but flattens after 32KB; associativity beyond 4-way is redundant.
- **Pipeline scaling:** Issue, decode, and commit widths saturate at 4-way due to dependency constraints.
- **Buffers:** ROB (64) and LSQ (32) sizes deliver the best trade-off between complexity and IPC.
- **Functional units:** Beyond 4 per type, units are underutilized.
- **TLB hierarchy:** 16–32 entries suffice for both ITLB and DTLB under arithmetic workloads.
- **Predictors and replacement policies:** Dynamic predictors slightly outperform static; replacement policies are largely performance-insensitive.

Key Trade-off Insight: Moderate provisioning across all pipeline and memory subsystems achieves 95% of peak IPC while minimizing hardware cost.

10. Overall Interpretation

Across all 149 configurations, performance follows a pattern of early gains followed by diminishing returns. Front-end and backend scaling are only beneficial until dependency and latency bottlenecks dominate. The `test-fmath` benchmark—being arithmetic-heavy—emphasizes data reuse and low branch activity, making cache hierarchy and ILP-limiting dependencies the primary design factors.

Final Insight: For compute-oriented workloads, a balanced microarchitecture with 4-way pipeline width, 32KB L1 cache, modest L2, dynamic predictor, and medium-depth buffers achieves near-optimal performance with minimal resource overhead.

5. Appendix

Included Files

- `run_all.sh` – Autoruns everything that needs to be executed
- `run_analysis.sh` – Automates SimpleScalar runs for all configurations
- `extract_data.py` – Extracts metrics from simulation output files
- `generate_graphs.py` – Plots IPC, miss rate, and sensitivity graphs
- `analysis_data.csv` – Contains summarized results

How to Run

1. To reproduce the experiment, place all the scripts (`run_all.sh`, `run_analysis.sh`, `extract_data.py`, and `generate_graphs.py`), in the SimpleScalar working directory.
2. Verify that the benchmark `test-fmath` executable is present in the `simplesim-tests/` directory (it is included by default in SimpleScalar distributions).
3. Run the unified automation script:

   ```
   bash run_all.sh
   ```

   This automatically executes all simulation runs, extracts data, and generates results and graphs.
4. Once completed, check the following:
   - `results/` — contains raw simulation outputs.
   - `analysis_data.csv` — consolidated performance metrics.
   - `graphs/` — visual reports (IPC, miss rates, etc.).