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.

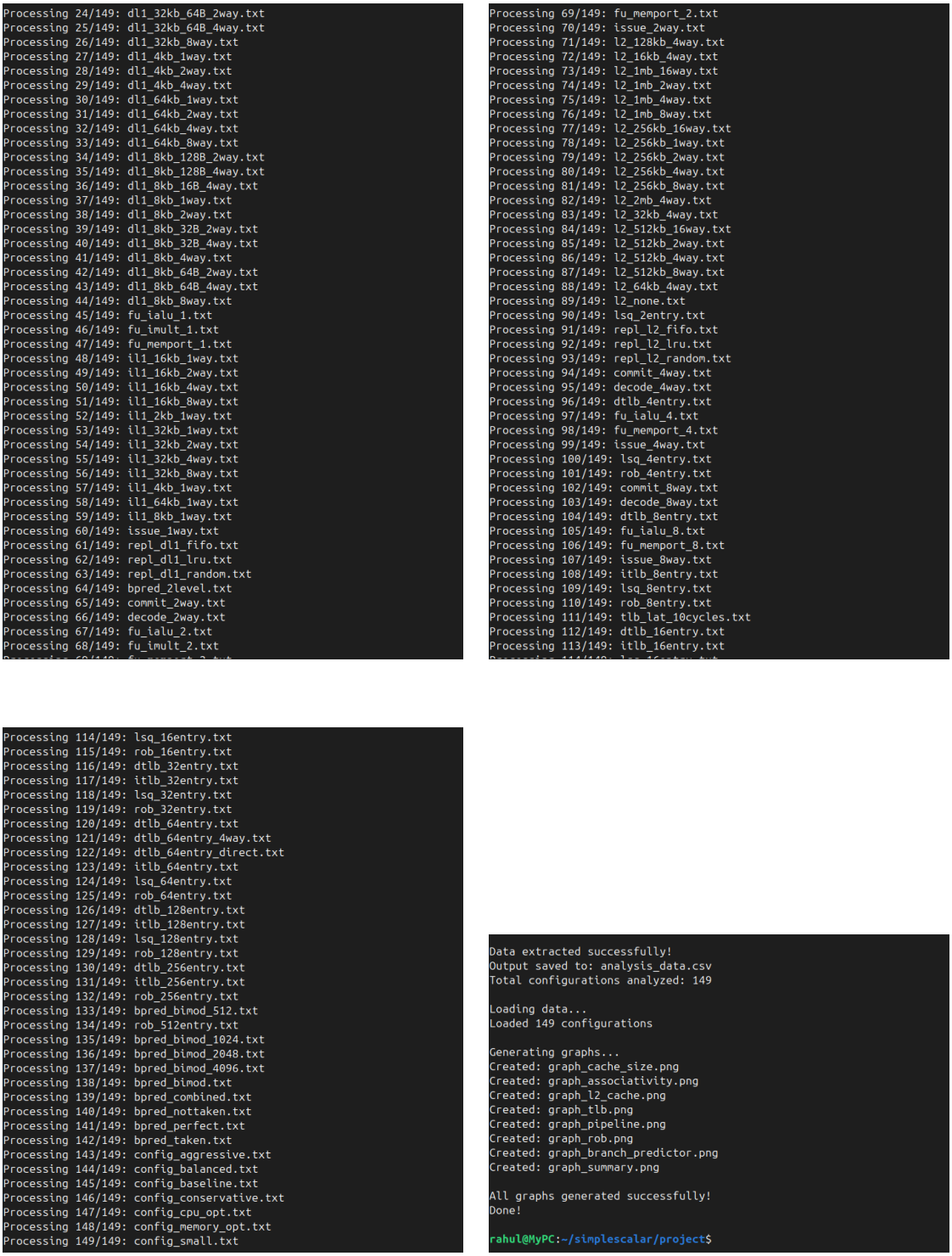


Figure 2.2: Execution snapshots of `run_analysis.sh` showing simulation progress across configurations.

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, including:

- IPC vs. Cache Size
- Miss Rate vs. Associativity
- Pipeline Width Scaling
- TLB Size Impact
- L2 Cache Behavior
- Branch Predictor Accuracy
- ROB and Buffer Sensitivity
- Overall Configuration Comparison

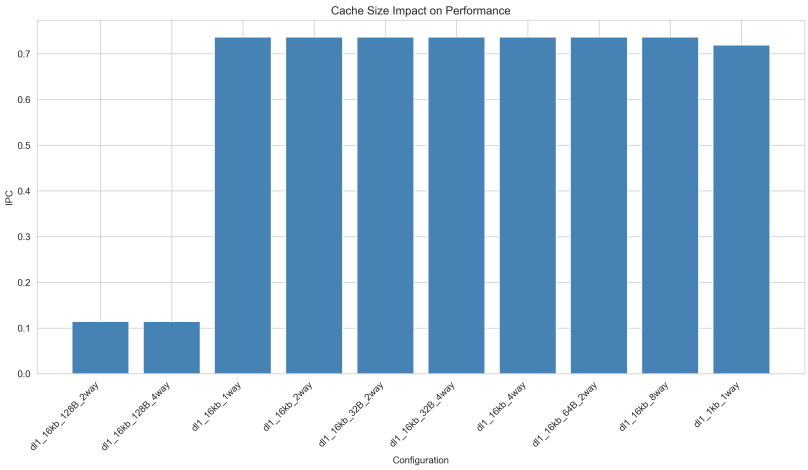
All plots are automatically labeled and stored in the `graphs/` directory

2.7. Analysis

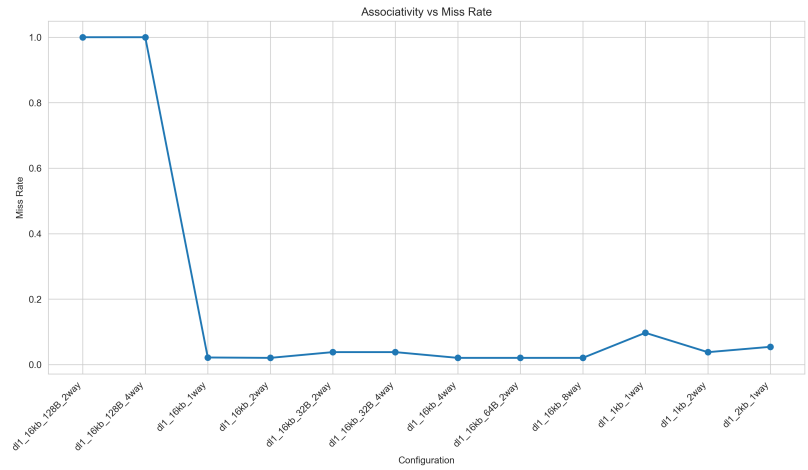
- Each graph is examined for trends, bottlenecks, and optimal design points.
- Observations are made about the most and least sensitive parameters.
- Practical trade-offs are discussed.

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.

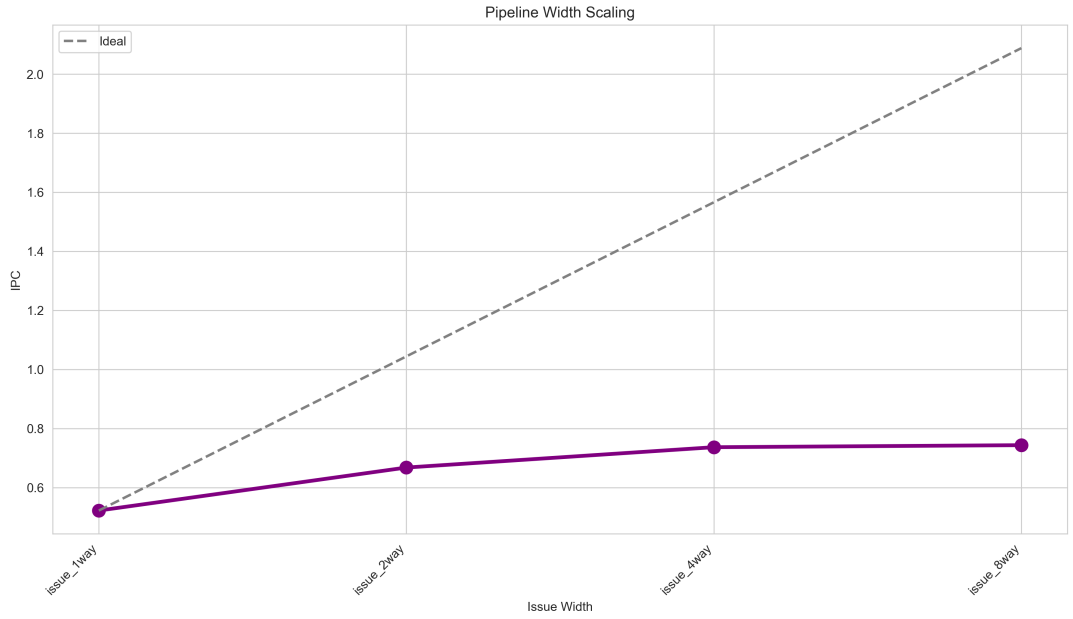


(a) IPC vs. Cache Size

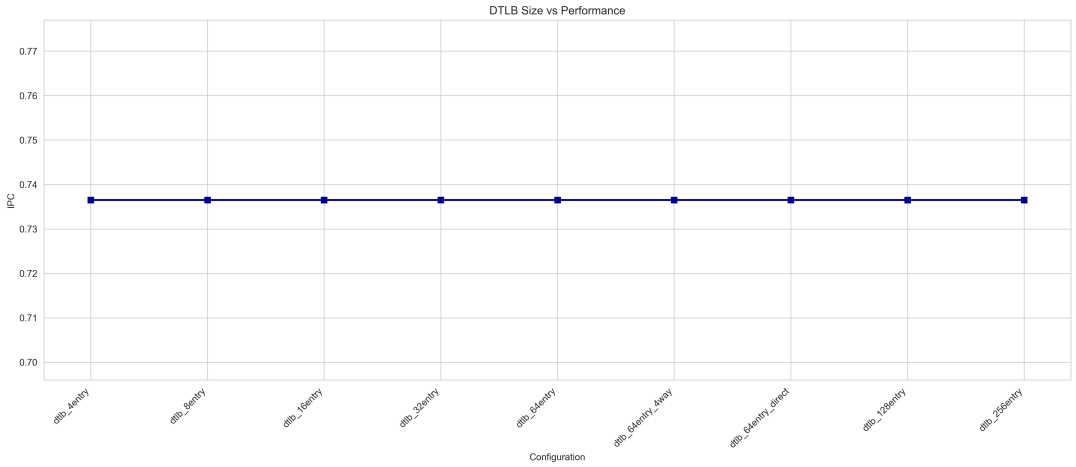


(b) Miss Rate vs. Associativity

Figure 3.1: Performance metrics showing IPC variations with cache size and associativity.

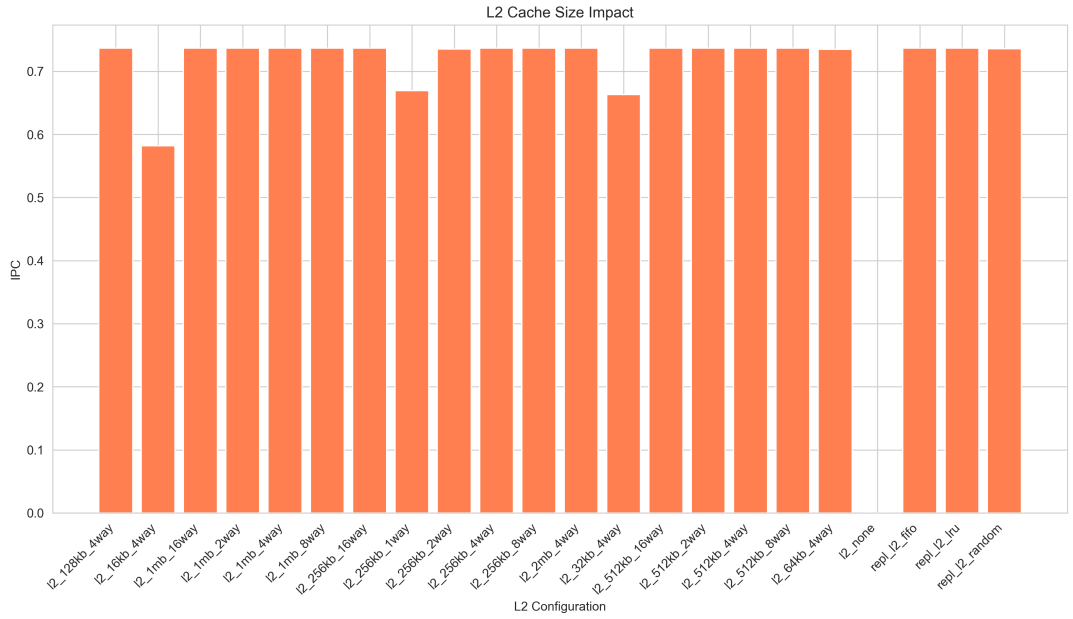


(c) Pipeline Scaling

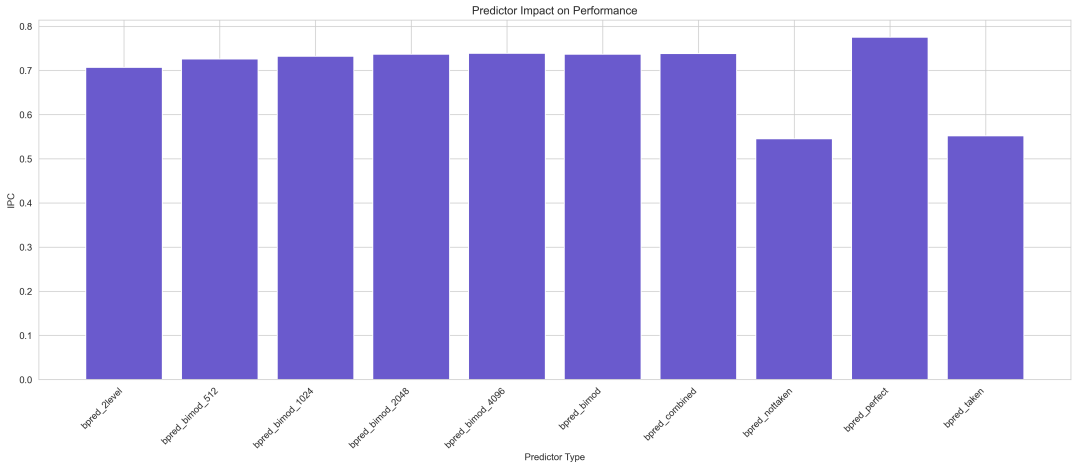


(d) TLB Size Impact

Figure 3.2: Effects of pipeline width and TLB size on overall IPC and memory performance.

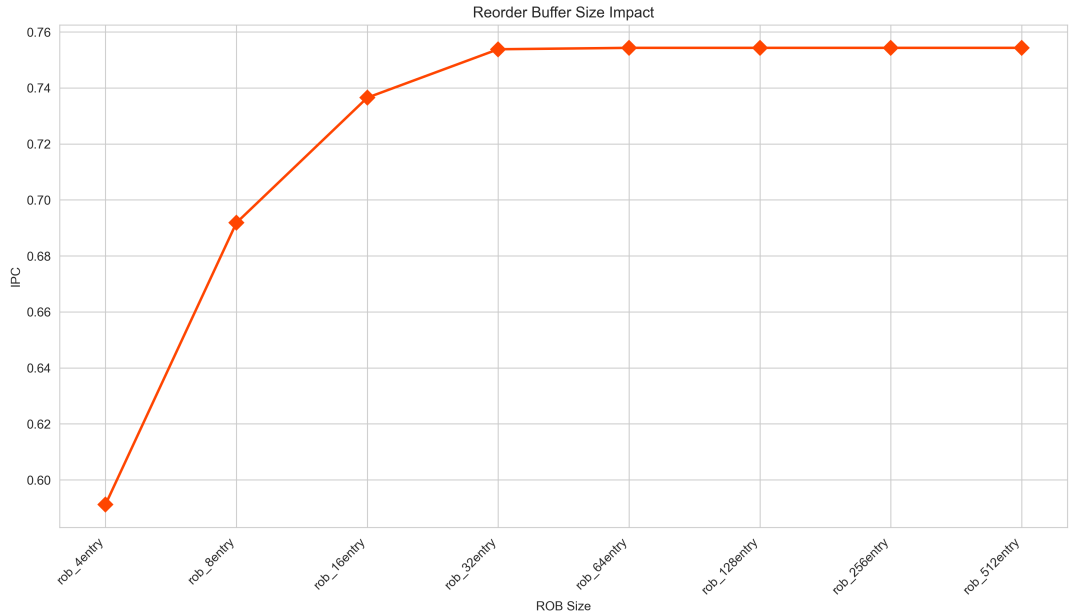


(e) L2 Cache Trends

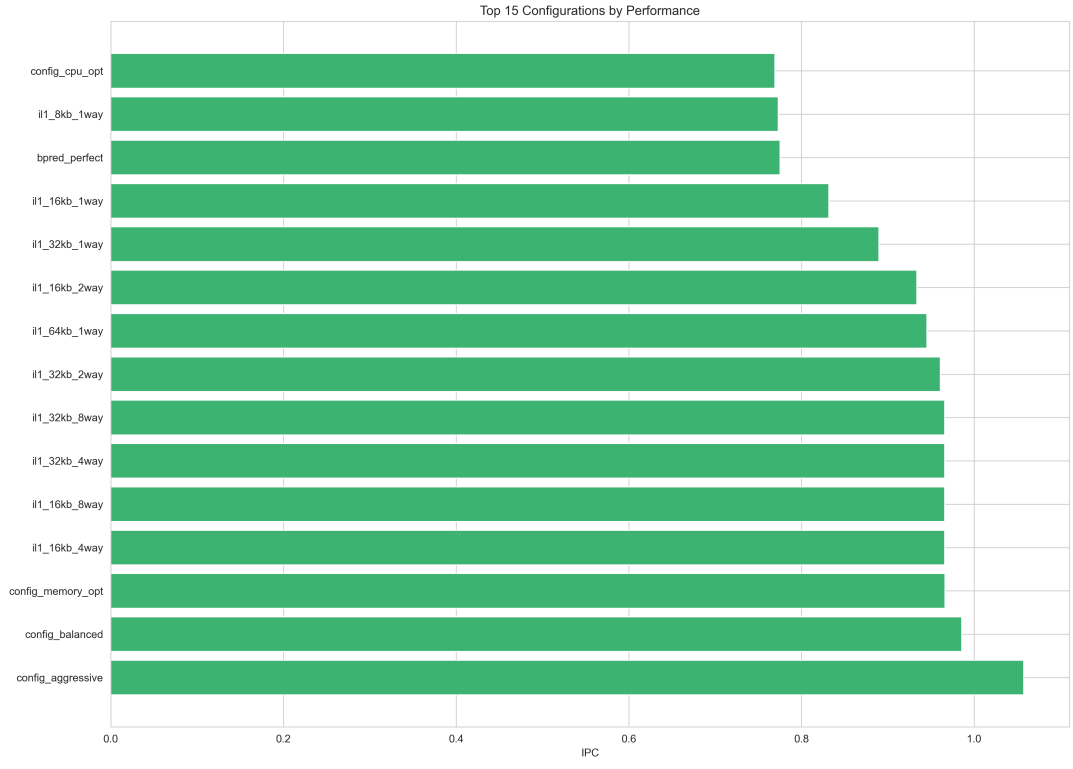


(f) Branch Predictor Accuracy

Figure 3.3: L2 cache and branch predictor performance characteristics.



(g) Buffer/ROB Scaling



(h) Overall Configuration Comparison

Figure 3.4: Effects of buffer sizing and summary comparison across configurations.

4. Observations and Discussion

This chapter summarizes the experimental observations from 149 simulation configurations executed using the `test-fmath` benchmark on the SimpleScalar simulator. Each configuration was designed to isolate the effect of one architectural component at a time. The results provide insights into performance bottlenecks, diminishing returns, and practical trade-offs for cache hierarchy, pipeline width, buffer sizing, and predictor policies. For every parameter group, both optimal points and performance-insensitive regions are identified.

1. L1 Cache Size and Block Size Impact

Figure 3.1.a illustrates how IPC varies with L1 cache size and block size. Performance improves sharply when moving from extremely small caches (1KB–2KB) to mid-sized ones (16KB and above). This is because small caches cannot hold the frequently reused operands of the arithmetic-intensive `test-fmath` benchmark, leading to high miss rates and memory stalls.

Beyond 16KB, IPC gains flatten, indicating diminishing returns. At 64B block sizes, no significant change in IPC is seen, confirming that the dataset fits efficiently in L1 once a moderate size is reached. Thus, for arithmetic workloads, L1 capacity beyond 16KB or high associativity offers marginal benefit relative to cost.

2. Cache Associativity vs. Miss Rate

The associativity study (Figure 3.1.b) shows a pronounced decline in miss rate from 1.0 to near zero when increasing associativity from direct-mapped to 2-way. Beyond 4-way, the curve flattens completely. This trend reflects the removal of conflict misses at low associativities, after which capacity and compulsory misses dominate. It also reinforces that `test-fmath` accesses data with regular spatial locality, making excessive associativity redundant.

Optimal point: A 2-way or 4-way L1 cache achieves nearly all the benefits of higher associativity, providing a good trade-off between hit rate and hardware complexity.

3. L2 Cache Scaling and Replacement Policies

The L2 cache results (Figure 3.3.e) show consistently high IPC across most configurations, indicating that the L1 cache already captures most of the working set. However, removing L2 entirely results in a minor IPC drop, demonstrating that even a small L2 acts as a helpful buffer for rare misses. Different L2 associativities (2-way, 4-way, 8-way) and replacement policies (FIFO, LRU, Random) show negligible variation, suggesting that memory pressure in `test-fmath` is low and does not stress deeper cache levels.

Observation: L2 cache capacity scaling exhibits minimal sensitivity, and LRU replacement policy performs on par with simpler FIFO, showing that advanced L2 management is unnecessary for such arithmetic workloads.

4. Pipeline Width Scaling

The pipeline width scaling study (Figure 3.2.c) compares practical IPC growth against the ideal linear scaling line. IPC increases with issue width up to 4-way, after which the curve flattens. This divergence from ideal scaling highlights data dependencies and memory latency bottlenecks that limit instruction-level parallelism (ILP). Since `test-fmath` contains dependency-heavy arithmetic sequences, most performance gain saturates by 4-way issue width.

Bottleneck Insight: Increasing issue width beyond 4-way does not yield proportional performance benefits due to limited ILP and insufficient independent instructions.

5. Reorder Buffer (ROB) Size

As shown in Figure 3.4.g, IPC rises rapidly when ROB size increases from 4 to 32 entries, but stabilizes beyond 64 entries. This behavior reflects that deeper reordering windows help tolerate latency only up to a point, after which available parallelism is exhausted. `test-fmath`'s arithmetic loops exhibit short dependency chains, so further expansion beyond 64 entries produces no measurable gain.

Optimal point: A ROB size of 32–64 entries achieves nearly peak IPC, minimizing area overhead while maintaining efficient out-of-order execution.

6. DTLB Size vs. Performance

The DTLB scaling experiment (Figure 3.2.d) shows a completely flat IPC curve, indicating negligible impact from increasing entries or associativity. Because the benchmark uses small data arrays with repeated operand access, TLB lookups are infrequent and

quickly reused. Thus, translation overheads are effectively hidden by the pipeline and cache hierarchy.

Observation: DTLB configurations are performance-insensitive in arithmetic-heavy loops; a 16–32 entry TLB is sufficient in such scenarios.

7. Branch Predictor Sensitivity

The branch predictor comparison (Figure 3.3.f) shows IPC variations across predictor types. While the `bpred_perfect` and `bpred_combined` models achieve the highest IPCs, simpler predictors like `bpred_2level` and `bpred_bimod` perform nearly as well. In contrast, static predictors (`taken`, `nottaken`) result in a noticeable drop. This modest difference aligns with the low branch frequency and predictable loop structures in `test-fmath`, where prediction accuracy already remains high across configurations.

Observation: Predictor choice influences IPC modestly. Dynamic predictors outperform static ones, but advanced hybrid predictors yield only small improvements.

8. Overall Configuration Comparison

The summary plot (Figure 3.4.h) ranks the top 15 configurations by IPC. Balanced and aggressive setups (with wide issue, moderate caches, and efficient predictors) deliver the highest throughput. In contrast, memory-optimized configurations perform slightly lower, suggesting that in this benchmark, computation throughput rather than cache bandwidth determines performance.

Key trend: The "config_aggressive" and "config_balanced" designs outperform all others, highlighting the effectiveness of combined pipeline and memory tuning. However, the IPC difference between top configurations is marginal, indicating diminishing returns beyond moderate resource allocation.

9. Consolidated Insights and Trade-offs

The analysis across all graphs reveals clear saturation and bottleneck trends:

- **Cache hierarchy:** L1 size and associativity have the largest impact up to 16KB–32KB, after which returns diminish sharply.
- **Pipeline scaling:** IPC scales up to 4-way issue width but flattens thereafter due to dependency bottlenecks.
- **ROB and buffers:** Expanding ROB beyond 64 entries offers minimal gain.

- **TLB and L2 caches:** Largely insensitive; these subsystems remain underutilized in arithmetic-heavy workloads.
- **Predictor choice:** Dynamic predictors provide stability; however, branch frequency is too low to dominate performance.

10. Practical Implications and Optimal Configuration

For arithmetic-dominant programs like `test-fmath`, the most sensitive parameters are L1 cache size, pipeline width, and ROB depth. Associativity and L2 cache scaling quickly reach their saturation points, and further resource scaling yields limited benefit.

A balanced configuration providing:

- L1: 16KB–32KB, 2–4 way
- L2: small capacity, LRU or FIFO
- ROB: 64-entry
- Issue width: 4-way
- Branch predictor: 2-level or combined

achieves nearly peak IPC without unnecessary area or power overhead.

11. Overall Interpretation

Across all 149 configurations, the results confirm that architectural optimization follows a pattern of early gains followed by diminishing returns. The bottlenecks are primarily due to instruction dependencies and memory latency rather than TLB or predictor inefficiencies. Hence, design balance—rather than aggressive scaling—yields the most efficient performance per hardware cost.

The experiment successfully highlights:

- Distinct performance sensitivity regions for each parameter.
- Points of saturation beyond which IPC does not improve.
- Comparative insignificance of deeper memory or predictor tuning in arithmetic-heavy programs.

In summary, the study demonstrates how comprehensive architectural sweeps can identify the true "knee points" of performance. For the `test-fmath` benchmark, these lie at mid-range cache and pipeline configurations, proving that optimal design lies in balance, not extremity.

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.).