Performance Impact of OoO Execution and Speculation in gem5

Sai Kaushik S (2025CSZ8470)
Yosep Ro (2025ANZ8223)
Indian Institute of Technology Delhi

October 6, 2025

1 Introduction

The objective of this project is to study the effect of out-of-order (OoO) execution, branch and memory speculation, and a multi-level cache hierarchy using the `gem5` simulator. The complete source code, simulation scripts, and data used for generating the plots in this report are publicly available on GitHub at
https://github.com/sai-kaushik-s/gem5-cache.

2 Experimental Setup

The host system specifications are as follows:

- **Processor:** 13th Gen Intel Core i7-13700 (16 Cores, 24 Threads, 3.2 GHz Base Frequency, up to 5.2 GHz Max Frequency)
- **Memory:** 16 GB DDR4 (Dual Channel, 3200 MHz)
- **Instruction Set Architecture:** x86 64-bit
- **Operating System:** Ubuntu 22.04.5 LTS (Kernel 6.10.0)

The analysis was conducted using the **gem5 simulator (v25.0)** in System Emulation (SE) mode. In SE mode, gem5 simulates user-level execution of programs without modeling the full operating system or peripheral devices. This mode provides faster simulation speeds compared to Full-System mode while still allowing detailed microarchitectural analysis of CPU and memory subsystem behavior.
The simulation system specifications are as follows:

- **Processor:** O3CPU (Out-of-Order, 1 Core, 3.2 GHz Base Frequency)
- **Memory:** 3 GB Single Channel DDR3 1600MHz
- **Instruction Set Architecture:** x86

The processor and cache configurations used in the experiments are as follows:

- **ROB Entries:** 64, 128, **192** and 256

- **IQ Size:** 32, **64**, and 128
- **Pipeline Widths:** 2, **4** and 8 (fetch, decode, rename, issue, writeback, commit)
- **Private L1 Cache Sizes:** 16 KiB, **32 KB** and 64 KiB (instruction and data)
- **Shared L2 Cache Sizes: 256 KB**, 1 MB and 4 MB
- **Branch Predictors: None**, Bimodal, Local, Tournament, TAGE, Perceptron
- **Associativity:**
  - L1: 2, **4**, 8
  - L2: **8**, 16, 32
- **Replacement Policies:**
  - L1: **None**, PLRU, LRU, Random
  - L2: **None**, Dueling, LRU, Random
- **Prefetchers:**
  - L1: **None**, Stride, Tagged
  - L2: **None**, Stride, Tagged, AMPM, Signature

The bolded values in the above configuration represent the baseline setup used in all experiments. For each parameter, the value is varied individually as indicated, while keeping all other parameters fixed at their baseline values, to analyze its impact on performance.

3 Architectural Components

3.1 Replacement Policies

3.1.1 None

"No replacement policy" means that when a cache line needs to be replaced, no sophisticated mechanism decides which line to evict — usually a default or static strategy is used. This can be as simple as evicting the first line in a set (FIFO) or leaving cache lines untouched until explicitly overwritten. It's rarely used in high-performance systems because it can cause inefficient cache utilization.

3.1.2 LRU (Least Recently Used)

LRU tracks the access order of cache lines and always evicts the line that has not been used for the longest time. This policy assumes that recently used lines are likely to be used again soon. LRU gives high cache hit rates for programs with temporal locality but requires more hardware or bookkeeping to maintain usage order.

3.1.3 PLRU (Pseudo-LRU)

Pseudo-Least Recently Used approximates the true LRU behavior but with much lower hardware cost. Instead of tracking the exact usage order of all lines in a set, it keeps a simple structure (like a binary tree) to track which lines are "less recently used." PLRU performs well in most workloads, balancing simplicity and performance.

3.1.4 Random

Random replacement chooses a cache line to evict uniformly at random. While simple and cheap to implement, its performance is unpredictable. Surprisingly, it can perform reasonably well in some workloads because it avoids the overhead of maintaining usage tracking, and over time, random replacement statistically approximates fairness among cache lines.

3.1.5 Dueling

Dueling is a dynamic policy where multiple replacement strategies (e.g., LRU vs. Random) compete ("duel") on a subset of cache sets. The policy that performs better in that subset is then applied to the rest of the cache. It's adaptive and can improve hit rates when no single policy works best for all workloads.

3.2 Prefetchers

3.2.1 None

"No prefetcher" means the cache only loads data on demand from memory, without trying to predict future accesses. This keeps the system simple and avoids extra memory traffic but misses opportunities to hide memory latency.

3.2.2 Stride

Stride prefetchers detect regular access patterns with a fixed stride (e.g., accessing every 8th element of an array). Once the pattern is detected, the prefetcher brings in future cache lines ahead of time. It's simple and effective for array-like or linear memory accesses.

3.2.3 Tagged

Tagged prefetchers track past accesses and attach "tags" to memory addresses to predict future accesses. They can handle more irregular patterns than stride prefetchers, using the history of accesses to guide prefetching. This makes them more flexible but slightly more complex.

3.2.4 AMPM (Access Map Pattern Matching)

AMPM prefetchers track memory access patterns across multiple cache lines using a map structure. They try to recognize and prefetch recurring access patterns, including non-contiguous ones, and can adapt to complex program behaviors. This often yields high prefetch accuracy for scientific or irregular workloads.

3.2.5 Signature

Signature prefetchers build a compressed "signature" of recent access history for each memory region. They use these signatures to predict future addresses even in irregular or sparse memory access patterns. Signature prefetchers are more sophisticated and can prefetch accurately for workloads with complicated or pointer-based accesses.

4 Expected behaviors of Microbenchmarks

Our experiments target three microbenchmarks:

- **Compute-bound kernel**: Performs iterative arithmetic operations involving integer additions and floating-point multiplications inside a tight loop, representing a simple arithmetic workload dominated by computational instructions.
- **Pointer-chasing kernel**: Traverses a dynamically allocated singly linked list multiple times, representing workloads with irregular memory access and pointer chasing behavior.
- **Streaming kernel**: Iterates over large arrays with a configurable stride, performing element-based simple arithmetic operations, representing sequential memory access and array streaming behavior.

4.1 Compute-bound Kernel

The compute-bound kernel consists of ALU-intensive loops that stress the OoO execution engine and branch speculation mechanisms. We expect this workload to be highly sensitive to out-of-order parameters such as ROB size and pipeline width. Increasing the ROB entries and widening the pipeline should expose more instruction-level parallelism (ILP) and improve IPC, although the gains are likely to diminish beyond 192 entries or 8-wide issue. Any branch predictor would work to improve the performance as it is a simple loop with a single exit. Also, cache size and prefetching should have a negligible impact, since the workload primarily resides in registers and the instruction cache.

4.2 Pointer-chasing Kernel

The pointer-chasing kernel involves linked-list traversals and stresses memory-dependence speculation and branch prediction. In this workload, memory-dependence speculation is expected to be the dominant factor: enabling it should reduce stalls, while disabling it will increase replay and stall cycles. Any branch predictor would work to improve the performance as it is a simple loop with a single exit. Prefetchers are generally ineffective due to the irregular access pattern and, in some cases, aggressive prefetching may cause cache pollution. Scaling the ROB and pipeline width has limited benefits, since memory latency dominates overall performance.

4.3 Streaming Kernel

The streaming kernel performs sequential array traversals, which primarily stress cache behavior and prefetching. Performance in this workload is strongly dependent on memory bandwidth and latency, making L1 and L2 cache sizes and effective prefetchers the most critical factors. Prefetchers such as AMPM, Tagged, and Signature are expected to deliver significant performance gains by reducing AMAT and hiding memory latency. Changes in ROB size and pipeline width should have minimal effect, as execution is limited by memory throughput rather than ILP. Similarly, branch predictor choice has negligible impact due to the lack of control flow, while cache replacement policies may matter only under smaller L2 configurations, with LRU expected to outperform random in sequential access patterns.