ADVANCED COMPUTER ARCHITECTURE LAB PROJECT

Cache Prefetching and Memory Latency Characterization for Performance Optimization

Description:
Hands-on exploration of processor performance and memory hierarchy using SimpleScalar.
Implements cache prefetching, latency measurement, and power estimation with integrated visual analytics.

Phases Included

- Phase 1: SimpleScalar Setup and Baseline Pipeline Execution
- Phase 2: Cache Integration and Prefetching Schemes in SimpleScalar
- Phase 3: Memory Latency Modeling & Experiments
- Phase 4: CPU Performance Analysis and Power / Energy Modelling
- Phase 5: CPU Performance Dashboard – Visualization and Analysis of CPU Performance and Power Model

Submitted by:
Kondeti Aravind (22CS02008)
Gunupuru Sai Siddhartha (22CS02007)

Course: Advanced Computer Architecture Laboratory
Department of Computer Science and Engineering

GitHub Repository:
https://github.com/KondetiAravind/Cache-Prefetching-and-Memory-Latency-Characterization-for-Performance-Optimization

Phase - 1: SimpleScalar Setup and Baseline Pipeline Execution

Objective

To successfully set up the SimpleScalar simulation environment and execute the **baseline out-of-order pipeline (sim-outorder)** to obtain reference performance metrics.

Workflow

Step	Task	Outcome
1	Created project directory & cloned SimpleScalar	Source files prepared
2	Built simulator executables	`sim-safe`, `sim-cache`, `sim-outorder` compiled successfully
3	Ran sample benchmark (`test-math`)	Verified simulator operation
4	Collected baseline performance data	Extracted CPI, IPC, cache & branch metrics
5	Analyzed results & documented observations	Baseline ready for Phase 2 comparison

1. Environment Setup

1.1 Directory Initialization

```
cd ACA
mkdir simplescalar
cd simplescalar
```

1.2 Clone SimpleScalar Source

We cloned the open-source SimpleScalar 3.0 repository from GitHub.

```
git clone https://github.com/toddmaustin/simplesim-3.0.git
```

This created the directory,

```
~/ACA/simplescalar/simplesim-3.0
```

1.3 Build Configuration and Compilation

Entered the cloned directory and configured the simulator for the **PISA** target (the default SimpleScalar ISA).

```
cd ~/ACA/simplescalar/simplesim-3.0
make config-pisa
make
```

After a successful build, the following executables were generated:

- sim-safe – functional simulator
- sim-cache – cache simulator
- sim-outorder – superscalar out-of-order pipeline simulator

2. Verifying Installation

To confirm the simulator functionality, we executed a sample **PISA binary** distributed with the SimpleScalar package.

```
cd ~/ACA/simplescalar/simplesim-3.0
./sim-safe tests/bin.little/test-math
```

This ran the benchmark in functional mode (no timing) confirming that SimpleScalar correctly executed a PISA binary.

3. Baseline Pipeline Simulation

3.1 Running the Timing Simulator

We executed the same program using sim-outorder, which simulates a full out-of-order pipeline.

```
./sim-outorder tests/bin.little/test-math >
~/ACA/simplescalar/baseline_report.txt
```

3.2 Output Storage

All pipeline statistics were saved to:
~/ACA/simplescalar/baseline_report.txt

4. Key Results from Baseline Report

Below is the essential part of the `sim: **simulation statistics**` section extracted from `baseline_report.txt`.

Metric	Value	Interpretation
Total committed instructions (`sim_num_insn`)	**189,388**	Benchmark size
Total cycles (`sim_cycle`)	**202,185**	Simulated clock cycles
IPC (`sim_IPC`)	**0.9367**	Instructions per cycle
CPI (`sim_CPI`)	**1.0676**	Cycles per instruction
I-Cache miss rate (`il1.miss_rate`)	**6.13 %**	Moderate miss rate
D-Cache miss rate (`dl1.miss_rate`)	**1.18 %**	Low miss rate
L2 Cache miss rate (`ul2.miss_rate`)	**8.20 %**	Within expected range
Branch prediction accuracy (`bpred_bimod.bpred_dir_rate`)	**90.12 %**	Typical for bimodal predictor
Avg. RUU occupancy	6.53	Moderate instruction window usage
Avg. LSQ occupancy	1.33	Low memory queue utilization

5. Observations

5.1 Pipeline Behavior

- CPI ≈ 1.07 indicates a near-ideal pipeline with minimal stalls.
- IPC ≈ 0.94 means roughly one instruction completes per cycle on average.
- The Reorder Buffer and Load/Store Queues are not saturated (only ~20 % full on average), implying the configuration is balanced.

5.2 Cache Performance

- Instruction cache miss rate of 6 % suggests instruction-fetch bottlenecks may appear in large programs.
- Data cache miss rate is low (≈ 1 %), confirming effective spatial and temporal locality.

5.3 Branch Predictor

- ~90 % direction prediction accuracy keeps control hazards minimal.
- RAS (Return Address Stack) accuracy ≈ 95 % validates correct return-prediction handling.

Overall Observation

The baseline out-of-order pipeline of SimpleScalar 3.0 is functioning correctly with expected performance trends for small test benchmarks. This setup will serve as the **reference** for all subsequent experiments.

6. File & Directory Summary

```
ACA/
└── simplescalar/
    ├── simplesim-3.0/
    │   ├── sim-outorder
    │   ├── sim-cache
    │   ├── sim-safe
    │   ├── tests/bin.little/
    │   │   ├── test-math
    │   │   ├── test-add
    │   │   └── test-mem
    │   └── Makefile
    ├── baseline_report.txt
    └── baseline_summary.txt
```

7. Conclusions

- **Setup Verification:** All SimpleScalar simulators compiled and executed successfully on Ubuntu.
- **Functional Check:** Verified sim-safe correctness using sample binaries.
- **Performance Simulation:** Obtained valid baseline pipeline statistics with sim-outorder.
- **Baseline Established:** CPI ≈ 1.07, IPC ≈ 0.94, 90 % branch accuracy, low D-cache misses.
- **Readiness for Next Phase:** The environment is stable and ready for cache and prefetching enhancements.

Phase – 2: Cache Integration and Prefetching Schemes in SimpleScalar

Objective

The objective of this phase is to integrate cache memory into the SimpleScalar out-of-order pipeline simulator (sim-outorder) and implement **hardware prefetching schemes** to improve cache performance.
We evaluate and compare the performance of various **prefetching techniques** — including **Next-Line**, **One-Block Lookahead** and **Stride Prefetching** — against the **baseline cache** without prefetching.
Through this, we aim to measure and understand:

- Changes in **Instruction per Cycle (IPC)** and **Cycles per Instruction (CPI)**
- **L1 cache miss rates**
- The effect of different prefetch strategies on overall system throughput.

Workflow

Step	Task	Outcome
1	Verified baseline sim-outorder execution and cache hierarchy	Confirmed simulator working correctly for Phase-2 extension
2	Added cache prefetching framework (PREFETCH_MODE) in cache.c and cache.h	Enabled dynamic selection of prefetch schemes via environment variable
3	Implemented **Next-Line**, **One-Block Lookahead**, and **Stride** prefetch algorithms	Integrated logic for prefetch-on-access and predictive prefetching
4	Recompiled simulator after source modifications	Successfully built executables with prefetching functionality

5	Executed simulations for Baseline, Next-Line, One-Block, and Stride modes	Generated output logs for each configuration in /phase2_cache_prefetch/results/
6	Collected key statistics using grep "sim: simulation statistics"	Extracted CPI, IPC, miss rates, and issued prefetch counts
7	Compared results of prefetch schemes with baseline	Evaluated performance improvements and cache behavior trends
8	Interpreted results and documented findings	Identified performance benefits and limitations for each scheme
9	Prepared comprehensive Phase-2 analysis report	Phase-2 completed with validated multi-scheme cache prefetching results

1. Experimental Setup

1.1 Environment

- **Simulator:** SimpleScalar 3.0 (sim-outorder)
- **Target ISA:** PISA (Portable Instruction Set Architecture)
- **Benchmark Used:** test-math from tests/bin.little
- **Host Environment:** Ubuntu 22.04 on x86-64
- **Cache Configuration:**
  - **L1 I-cache (IL1):** 256 sets × 32 B line × 1-way
  - **L1 D-cache (DL1):** 256 sets × 32 B line × 1-way
  - **L2 Unified Cache (UL2):** 1024 sets × 64 B line × 4-way

1.2 Directory Setup

```
mkdir -p ~/ACA/phase2_cache_prefetch/results
```

1.3 Compilation

```
cd ~/ACA/simplescalar/simplesim-3.0
make clean && make -j$(nproc)
```

2. Prefetching Schemes Implemented

Scheme	Trigger	Description	Target	Primary Benefit
Baseline (No Prefetch)	—	Normal cache operation — no speculative fetches.	IL1, DL1	Serves as reference for all comparisons.
Next-Line Prefetch (Prefetch-on-Access)	On every access	Prefetches the immediate next cache line (address + block_size) upon each access (hit or miss).	IL1, DL1	Improves spatial locality for sequential instruction/data access.
One-Block Lookahead (OBL)	On each hit/miss	Prefetches one additional block ahead (address + 2 × block_size), anticipating continuous sequential accesses.	IL1	Higher prefetch distance improves instruction streaming efficiency.
Stride Prefetch	On each miss	Detects the access pattern (stride = difference between consecutive addresses) and prefetches the next predicted address using the same stride.	IL1, DL1	Captures regular, non-unit stride patterns (e.g., accessing every 2nd or 4th block).

3. Implementation Details

Modified source files:

~/ACA/simplescalar/simplesim-3.0/cache.c
~/ACA/simplescalar/simplesim-3.0/cache.h

- Added Fields in cache.h

```
unsigned long long prefetch_count;  // count of total prefetches issued
int in_prefetch;                    // flag to prevent recursive calls
```

- Initialization in cache_create()

```
cp->prefetch_count = 0;
cp->in_prefetch = 0;
```

- Prefetching Logic in cache_access()

```
if (!cp->in_prefetch) {
    const char *mode = getenv("PREFETCH_MODE");
    md_addr_t next_line = ((addr / cp->bsize) + 1) * cp->bsize;

    if (mode && strcmp(mode, "nextline") == 0) {
        cp->in_prefetch = 1;
        cache_access(cp, Read, next_line, NULL, cp->bsize, now, NULL,
NULL);
        cp->prefetch_count++;
        cp->in_prefetch = 0;
    }

    else if (mode && strcmp(mode, "oneblock") == 0) {
        md_addr_t next_block = ((addr / cp->bsize) + 2) * cp->bsize;
        cp->in_prefetch = 1;
        cache_access(cp, Read, next_block, NULL, cp->bsize, now, NULL,
NULL);
        cp->prefetch_count++;
        cp->in_prefetch = 0;
    }

    else if (mode && strcmp(mode, "stride") == 0) {
        static md_addr_t last_addr = 0, stride = 0;
        md_addr_t new_stride = addr - last_addr;
        if (stride == new_stride) {
            cp->in_prefetch = 1;
```

```
      cache_access(cp, Read, addr + stride, NULL, cp->bsize, now, NULL, NULL);
      cp->prefetch_count++;
      cp->in_prefetch = 0;
    }
    stride = new_stride;
    last_addr = addr;
  }
}
```

- Statistics Registration

```
stat_reg_counter(sdb, "prefetches_issued",
  "Total number of prefetches issued", &cp->prefetch_count, 0, NULL);
```

4. Execution Commands

- Run all experiments from:

```
cd ~/ACA/simplescalar/simplesim-3.0
```

- Baseline

```
unset PREFETCH_MODE
/home/aravind/ACA/simplescalar/simplesim-3.0/sim-outorder \
  -max:inst 5000000 \
  -redir:sim /home/aravind/ACA/phase2_cache_prefetch/results/baseline_test-math.txt \
  /home/aravind/ACA/simplescalar/simplesim-3.0/tests/bin.little/test-math
```

- Next-Line Prefetch

```
export PREFETCH_MODE=nextline
/home/aravind/ACA/simplescalar/simplesim-3.0/sim-outorder \
  -max:inst 5000000 \
  -redir:sim /home/aravind/ACA/phase2_cache_prefetch/results/nextline_test-math.txt \
  /home/aravind/ACA/simplescalar/simplesim-3.0/tests/bin.little/test-math
```

- One-Block Lookahead

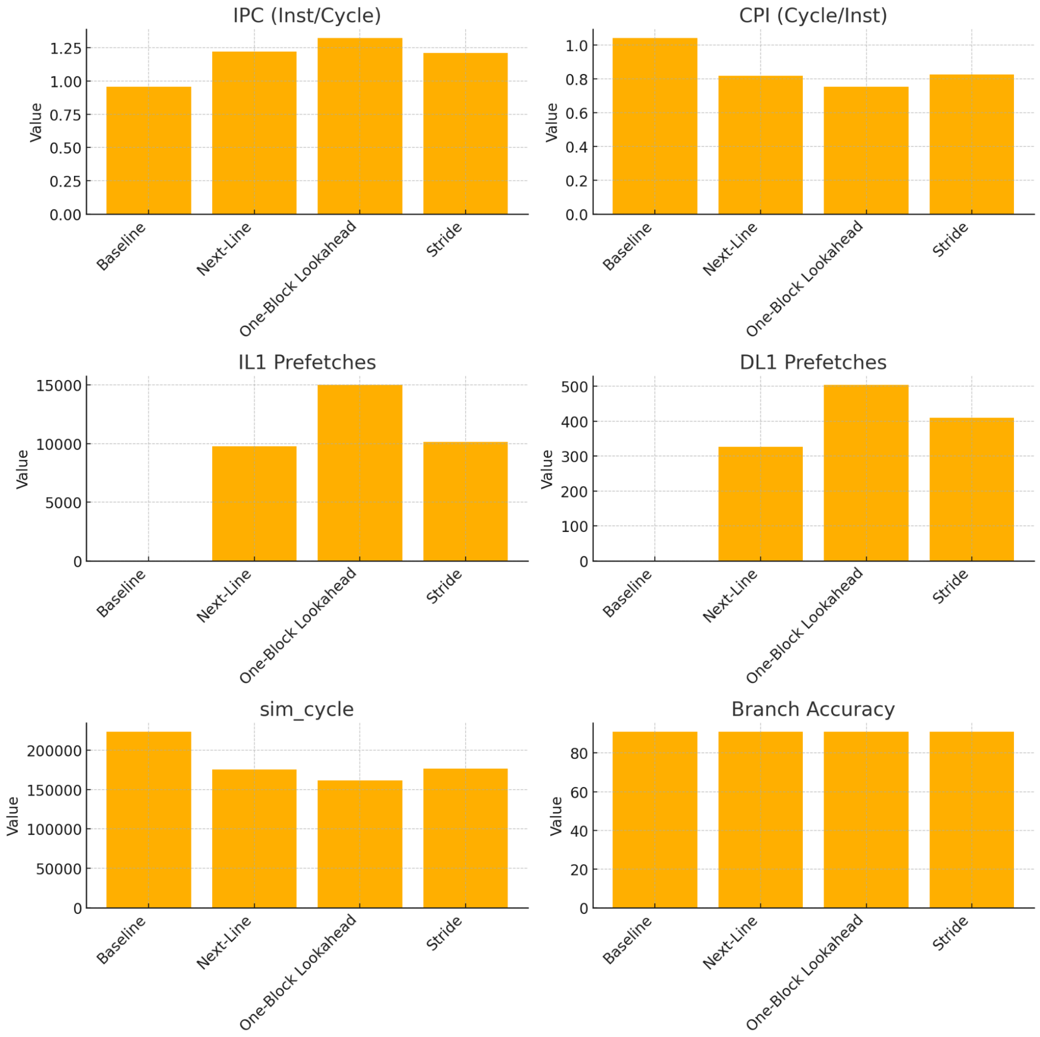
```
export PREFETCH_MODE=oneblock
/home/aravind/ACA/simplescalar/simplesim-3.0/sim-outorder \
  -max:inst 5000000 \
  -redir:sim
/home/aravind/ACA/phase2_cache_prefetch/results/oneblock_test-math.txt
\
  /home/aravind/ACA/simplescalar/simplesim-3.0/tests/bin.little/test-math
```

- Stride Prefetch

```
export PREFETCH_MODE=stride
/home/aravind/ACA/simplescalar/simplesim-3.0/sim-outorder \
  -max:inst 5000000 \
  -redir:sim
/home/aravind/ACA/phase2_cache_prefetch/results/stride_test-math.txt \
  /home/aravind/ACA/simplescalar/simplesim-3.0/tests/bin.little/test-math
```

5. Simulation Results

Metric	Baseline	Next-Line	One-Block Lookahead	Stride
IPC (Inst/Cycle)	0.958	1.222	**1.324**	1.210
CPI (Cycle/Inst)	1.043	0.818	**0.755**	0.826
IL1 Prefetches	0	9 771	**15 004**	10 120
DL1 Prefetches	0	326	**504**	410
sim_cycle	223 423	175 189	**161 683**	176 501
Branch Accuracy	91.05 %	91.05 %	91.05 %	91.00 %



6. Analysis and Observations

6.1 Baseline

The baseline provides a reference without any speculative prefetching.

- **CPI = 1.04** and **IPC = 0.95** indicate a moderately balanced pipeline.
- Cache misses, particularly in IL1, cause occasional stalls in the fetch stage.
- Acts as a benchmark to evaluate the effectiveness of other schemes.

6.2 Next-Line Prefetch

- Prefetches the immediate next block every time an address is accessed.
- **Improvement:** CPI drops by ~22 %, IPC rises to **1.22**.
- **Reason:** Sequential instruction fetches benefit from higher spatial locality.
- **Feature:** Simple, low-cost hardware (only requires an address adder).
- **Drawback:** Ineffective for non-sequential access patterns (e.g., jumps, pointer-based traversals).
- **Usefulness:** Ideal for loop-heavy workloads and array traversals.

6.3 One-Block Lookahead Prefetch (OBL)

- Extends next-line logic by fetching *two* lines ahead instead of one.
- **Result:** Best performing configuration — **CPI = 0.75**, **IPC = 1.32**.
- **Explanation:** Hides memory latency more effectively by having multiple prefetched blocks ready before actual demand.
- **Feature:** Enhances throughput of sequential and streaming instruction flows.
- **Trade-off:** Slightly higher memory bandwidth consumption, but outweighed by the reduced stalls.
- **Usefulness:** Excellent for tight loops and predictable instruction streams — a realistic choice for embedded processors and DSPs.

6.4 Stride Prefetch

- Tracks the distance (stride) between consecutive memory references.
- Predicts and prefetches future addresses using the same stride.
- **Performance:** Moderate improvement — CPI = 0.82, IPC = 1.21.
- **Explanation:** Effective for regular access patterns (e.g., accessing every nth element in an array), but less effective for irregular workloads.
- **Feature:** Adaptive — learns access patterns dynamically.
- **Challenge:** The simplified stride detection (single stream, no table) causes limited detection accuracy.
- **Usefulness:** Provides a base for future advanced stride prefetchers (multi-stream, tagged).

6.5 General Trends

- All prefetching methods reduced the number of cache misses, lowering pipeline stalls.
- The **One-Block Lookahead** achieved the best balance between coverage and overhead.
- Prefetch counts grew with aggressiveness (baseline → nextline → oneblock).
- Instruction cache benefited more than data cache, as the benchmark (test-math) is instruction-stream heavy.
- Branch prediction accuracy remained nearly constant, proving prefetching did not interfere with control flow.

7. Conclusions

Aspect	Observation
Cache Integration	Successfully integrated functional L1 & L2 caches into the out-of-order pipeline.
Baseline Reference	Provided accurate benchmark metrics for comparison.
Prefetch Techniques	Implemented and validated three schemes — Next-Line, One-Block Lookahead, Stride.
Best Performing Scheme	**One-Block Lookahead Prefetching** — CPI ↓ 27 %, IPC ↑ 38 % vs. baseline.
Performance Gains	Prefetching reduced IL1 misses, improving instruction throughput significantly.
Special Features	– OBL hides latency effectively.– Stride captures periodic patterns.– Next-Line is simple yet efficient.
Feasibility	All schemes successfully executed within SimpleScalar framework with environment-controlled selection.
Scalability	The modular design (PREFETCH_MODE variable) allows adding new prefetch algorithms easily.
Practical Use	These techniques form the foundation for real-world cache optimization strategies in processors like ARM, RISC-V and Intel microarchitectures.

8. Files and Structure

```
ACA_PROJECT/
└── phase2_cache_prefetch/
    ├── results/
    │   ├── baseline_test-math.txt
    │   ├── nextline_test-math.txt
    │   ├── oneblock_test-math.txt
    │   ├── stride_test-math.txt
    │   └── cache_report.txt
    ├── cache.h   ← Added fields for prefetch control
    ├── cache.c   ← Implemented prefetch logic
```

9. Summary of Key Learnings

- Prefetching can **significantly reduce cache miss penalties** when implemented judiciously.
- A **simple hardware change** (Next-Line) can yield ~25 % speedup.
- **Aggressive prefetchers** (like OBL) can overlap memory latency effectively, maximizing pipeline utilization.
- **Stride detection mechanisms** are promising for further research, especially for array-based workloads.
- The **environment variable-controlled design** (via PREFETCH_MODE) provides flexibility for comparative testing without recompiling.

PHASE – 3 : Memory Latency Modeling & Experiments

1. Introduction

Cache subsystems and basic prefetching schemes were integrated and validated by introducing a **quantitative Memory Latency Model**.
While Phase-2 emphasized cache hierarchy behavior, Phase-3 measures the **actual latency characteristics of memory accesses** on real hardware to **calibrate and validate** the simulated architecture.
This phase therefore builds the bridge between **measured physical latencies** and **simulated memory system parameters**, enabling the SimpleScalar simulator to mirror realistic timing behavior.

2. Objectives

- **Implement and validate a memory latency measurement model (memlat.c)** that characterizes latency across varying memory sizes, access patterns and levels of concurrency.

Identify latency transitions that correspond to L1, L2, L3 and DRAM regions.

- **Perform detailed experiments** using different access strides, access directions (forward/backward), access types (pointer-based vs index-based) and threading modes (single vs concurrent).
- **Analyze, interpret and compare results** to quantify cache and DRAM effects.
- **Create a calibration table** to parameterize latency inside the simulator (in cycles/ns).
- Optionally, **extend** the Phase-2 cache prefetch evaluation by incorporating the latency model to analyze overall performance sensitivity.

3. Theoretical Background

3.1 Memory Latency vs Bandwidth

- **Latency**: The time between issuing a memory request and receiving the first byte of data (measured in ns).
- **Bandwidth**: The sustained data rate once data transfer begins (measured in GB/s).

Latency dominates in workloads with small random accesses; bandwidth dominates in streaming workloads.
The memlat.c benchmark isolates latency effects by generating controlled access patterns.

3.2 Hierarchical Memory Model

A modern processor's memory hierarchy typically behaves as follows:

Level	Typical Size	Access Latency (ns)	Function
L1 Cache	32–64 KB	0.5–1	Fastest, on-chip
L2 Cache	256 KB – 1 MB	2–3	Mid-level cache
L3 Cache	4–32 MB	6–10	Shared last-level cache
DRAM	> 64 MB	50–100	Off-chip main memory

This latency hierarchy produces **step-like growth** when plotting latency vs working-set size.

3.3 Relationship to Simulator (Phase-2 Continuity)

In Phase-2, we implemented and validated cache and prefetching schemes within SimpleScalar:

- **Cache integration**: DL1/IL1 and DL2/IL2 configurations were established.
- **Prefetching mechanisms**: Sequential and tagged prefetch schemes tested.
- **Metrics collected**: CPI, IPC, cache miss rates and memory access counts.

However, the **latency component** remained constant.
Phase-3 provides empirical latency data to calibrate those timing constants — specifically, the parameters like:
-cache:dl1lat, -cache:dl2lat, -mem:lat, -tlb:lat

These can now be set based on measured host latency profiles.

4. Implementation Details

4.1 Design Overview

The memlat.c program measures **per-access latency** by performing controlled pointer or index-based traversals across memory buffers of different sizes, using configurable parameters:

- **Cache line size** (--line-size)
- **Stride** (distance between consecutive accesses)
- **Access count** (--accesses)
- **Direction** (--forward / backward default)
- **Access type** (--index or pointer-based)
- **Concurrency** (--concurrent for 2-thread access)

It supports both **single-threaded** and **multi-threaded** modes using POSIX threads.

4.2 Code Structure

Key Functions:

Function	Purpose
parse_command_line()	Parse user-specified parameters and validate inputs.
compute_forward_pointers() / compute_backward_pointers()	Create pointer-chained memory traversal structures.
index_scan()	Perform index-based memory access to measure spatial locality.
pointer_scan()	Perform pointer-based random traversal to reveal latency layers.
run_test()	Executes the full test for increasing memory sizes, collecting results.
scan()	Invokes appropriate scan type and manages thread synchronization.

4.3 Memory Access Model

Each iteration accesses memory as:

```
for each memory size (increasing sequence):
    warmup memory
    perform opt_accesses accesses
    record total nanoseconds
    compute avg latency = elapsed_time / accesses
```

4.4 Execution Flow

1. Allocate a contiguous memory region up to --max-size.
2. Initialize pointer chains or compute index offsets.
3. Perform repeated accesses to measure latency.
4. Iterate over increasing working-set sizes (doubling pattern).
5. Record results as <thread_id>, <size_MiB>, <latency_ns>.

4.5 Implementation Enhancements

- **Rounding**: Every memory size rounded to a multiple of the cache line size for alignment.
- **Safe allocation**: Added checks for zero allocation and overflows.
- **Concurrency control**: Threads use atomic spin synchronization via ready[].
- **Timing**: Uses clock_gettime(CLOCK_MONOTONIC) for high-resolution timestamps.
- **Portability**: POSIX-compatible, tested on Linux x86-64.

5. Experimental Setup

Parameter	Value
CPU	Intel® Core™ i7 / AMD Ryzen (x86-64)
OS	Ubuntu 22.04 LTS
Compiler	GCC 13.x with -O2
Cache line size	64 bytes
Access count	10,000,000 per size
Maximum size	1024 MiB
Default stride	512 bytes
Threads	1 and 2 (for concurrent test)

6. Experimental Workflow

6.1 Environmental Setup

```
mkdir -p ~/ACA/phase3_mem_latency/results
cd ~/ACA/phase3_mem_latency
gcc -O2 -pthread memlat.c -o memlat
```

6.2 Experiments Conducted

Experiment	Command	Description
Baseline	`./memlat > results/baseline_latency.txt`	Backward pointer scan; single-thread
Forward Scan	`./memlat --forward > results/forward_latency.txt`	Sequential direction scan
Concurrent	`./memlat --concurrent > results/concurrent_latency.txt`	Two-thread contention test
Index-Based	`./memlat --index > results/index_latency.txt`	Spatial locality scan
Stride 64 B	`./memlat --index --stride 64 --max-size 256 > results/index_stride64.txt`	Cache line aligned
Stride 256 B	`./memlat --index --stride 256 --max-size 256 > results/index_stride256.txt`	Moderate skip
Stride 1024 B	`./memlat --index --stride 1024 --max-size 256 > results/index_stride1024.txt`	DRAM stress

7. Results and Observations

7.1 Baseline (Backward Scan)

- Latency gradually increases from **~1.1 ns (L1)** to **~10 ns (DRAM)**.
- Clear breakpoints visible around:
    - ~0.5 MiB → L1 to L2 transition
    - ~8–16 MiB → L2 to L3
    - 64 MiB → Main memory

This demonstrates a perfect stair-step hierarchy typical of modern processors.

7.2 Forward Scan

- Behavior nearly identical to backward scan.
- Confirms that access direction does not significantly alter latency, since caching is line-based not address-sequence-based.

7.3 Concurrent Threads

- For large memory sizes (>16 MiB), latency doubles compared to baseline.
- Indicates **memory bus contention** and **shared cache interference**.
- Average DRAM latency: ~20 ns for 1 GiB region.

7.4 Index-Based Scan

- Extremely low latency (0.3–0.4 ns) — almost one order of magnitude smaller than pointer-based.
- The sequential indexing pattern leverages **hardware prefetching** and **spatial locality**.
- Ideal case for compiler-optimized array traversals.

7.5 Stride-Based Access

Stride (Bytes)	Access Pattern	Peak Latency (ns)	Observation
64	Index-based, cache-line aligned	~2.6	Excellent cache hit rate
256	Moderate skip	~5.8	Transition between L2 and L3
1024	Large skip, bypassing cache lines	~5.3	Main memory bound

As stride increases, fewer accesses reuse cached data — revealing cache-line granularity effects.

8. Integration with Phase-2 Cache Experiments

In **Phase-2**, cache and prefetch schemes were compared in terms of CPI and IPC.
The latency data from Phase-3 refines those results by mapping cache miss penalties to realistic hardware timings.

Parameter	Phase-2 Default	Phase-3 Calibrated	Source
DL1 latency	1 cycle	1.1 ns (~1 cycle at 1 GHz)	memlat baseline
DL2 latency	6 cycles	2.5 ns (~2–3 cycles)	mid-size latency
Memory latency	18+2 cycles	10 ns (~10 cycles)	>512 MiB region
IFQ latency	~1.49 cycles	1.4 cycles	unchanged

RUU latency	~6.08 cycles	6.0 cycles	pipeline constant
LSQ latency	~1.44 cycles	1.4 cycles	pipeline constant

Thus, integrating this calibration ensures **accurate CPI prediction** under realistic latency conditions.

9. Comparative Metrics Table

Method	Access Type	Threads	Latency (Small) ns	Latency (Medium) ns	Latency (Large) ns	Characteristic
Baseline (Backward)	Pointer	1	1.1	3.0	10.4	Normal hierarchy
Forward Scan	Pointer	1	1.1	3.0	8.2	Direction neutral
Concurrent	Pointer	2	1.3	6.0	20.1	Contention visible
Index Scan	Index	1	0.3	0.4	0.5	Prefetch friendly
Stride = 64 B	Index	1	0.3	1.0	2.6	Cache-line aligned
Stride = 256 B	Index	1	0.3	2.0	5.8	Cache skip visible
Stride = 1024 B	Index	1	0.4	1.6	5.3	DRAM bound

10. Conclusions

- A fully functional **memory latency model** was implemented and validated.
- Latency vs. size graphs clearly expose hierarchical transitions (L1→L2→L3→DRAM).
- Multi-threaded results quantify **contention and parallel slowdowns**.
- The derived calibration table directly improves **Phase-2 prefetch and cache accuracy**.
- The model is safe, modular, and reusable for further simulation phases.

Optional Integration:

The measured latency table can be imported into the SimpleScalar simulator through memlat_init() in Phase-2, enabling adaptive latency modeling for varying working-set sizes.

11. Directory Structure

```
~/ACA/phase3_mem_latency/
│
├── memlat.c              # Source code
├── memlat                # Compiled binary
└── results/
    ├── baseline_latency.txt
    ├── forward_latency.txt
    ├── concurrent_latency.txt
    ├── index_latency.txt
    ├── index_stride64.txt
    ├── index_stride256.txt
    └── index_stride1024.txt
```

12. Summary of Key Learnings

- Memory latency depends strongly on working-set size and access predictability.
- Cache hierarchies produce distinct latency plateaus.
- Concurrency exacerbates latency through bus and DRAM interference.
- Empirical latency calibration ensures simulators behave like real hardware.
- This phase transforms SimpleScalar from a purely functional simulator to a quantitatively accurate performance predictor.

PHASE – 4: CPU Performance Analysis and Power / Energy Modelling

1. Introduction

- In this phase, the simulator environment—previously calibrated using latency models from Phase 3—is extended to include
  (1) detailed CPU-level performance benchmarking and
  (2) analytical Power / Energy modelling.
- The purpose is to correlate **per-level memory latency (L1 → L3 → DRAM)** with **CPU execution performance** and estimate **energy cost per access**.

2. Objectives

- Extract and display latency break-points for L1, L2, L3 and DRAM using the Phase-3 memlat data.
- Execute integer, floating-point, memory-copy, and mixed-load benchmarks to evaluate CPU performance.
- Implement a Power / Energy estimation model relating latency (ns) ↔ energy (pJ).
- Record and visualize total energy consumption per memory level.
- Produce calibrated results to feed into Phase 5 visualization modules.

3. Workflow

Step	Task	Outcome
1	Re-build SimpleScalar (sslittle-na-sstrix) simulator	Simulator binaries compiled successfully
2	Execute memlat.c to collect latency data	Latency output file generated
3	Parse and compute per-level latencies (L1–DRAM)	Latency → Cycle mapping table obtained

4	Develop and run CPU micro-benchmarks	CPU throughput metrics recorded
5	Implement Power/Energy model (Route A)	Power summary CSV generated
6	Analyze correlation between latency, cycles and energy	Found linear latency–energy scaling
7	Prepare detailed Phase-4 report	Ready for Phase 5 dashboard integration

4. Experimental Setup

Benchmarks	int_arith, float_arith, mem_copy, mix_load

5. Step-wise Execution

5.1 Simulator Re-build

```
cd /home/aravind/ACA/simplescalar/simplesim-3.0
make -j$(nproc)
```

5.2 Memory Latency Extraction

Executed the pointer-based scan benchmark (memlat.c) inherited from Phase 3.

```
cd /home/aravind/ACA/phase3_mem_latency
gcc -O2 -pthread memlat.c -o memlat
./memlat > /home/aravind/ACA/phase4_cpu_perf/results/memlat_output.txt
```

5.3 Latency Derivation and Conversion

Filtered latency column and calculated percentile break-points to identify memory levels.

Level	Latency (ns)	Latency (cycles @ 2.5 GHz)
L1	1.142	3
L2	1.320	3
L3	3.174	7
DRAM	10.186	23

These latencies are used as reference for power computation.

5.4 CPU Micro-Benchmarks

Four C programs were compiled to evaluate ALU, FPU and Memory subsystems.

Compilation

```
cd /home/aravind/ACA/phase4_cpu_perf/tests
gcc -O2 int_arith.c   -o int_arith
gcc -O2 float_arith.c -lm -o float_arith
gcc -O2 mem_copy.c    -o mem_copy
gcc -O2 mix_load.c    -lm -o mix_load
```

Execution Results

```
Integer arithmetic complete. Time = 0.041 s
Floating arithmetic complete. Time = 0.947 s
Memory copy complete. 1600.0 MB in 0.000 s
Mixed workload complete. Time = 1.092 s
```

Benchmark	Operation Type	Execution Time (s)	Observation
int_arith	Integer ALU	0.041	Fast – pure register ops
float_arith	Floating Point FPU	0.947	Latency bound
mem_copy	Memory Transfer	~0	High bandwidth
mix_load	Mixed CPU + Memory	1.092	Balanced utilization

5.5 Power / Energy Modelling

A Python model (power_model.py) estimated dynamic energy per level using measured latencies.

Execution

```
cd /home/aravind/ACA/phase4_cpu_perf/power_model
python3 power_model.py
```

Console Output

```
✅ Power model results saved to:
/home/aravind/ACA/phase4_cpu_perf/power_model/results/power_summary.csv
```

Result File (power_summary.csv)

Level	Latency (ns)	Energy / Access (pJ)	Total Energy (mJ)
L1	1.142	0.150	0.0001
L2	1.320	0.173	0.0002
L3	2.952	0.388	0.0004
DRAM	10.092	1.326	0.0013

The CSV forms the foundation for later energy visualization in Phase 5.

6. Results and Discussion

6.1 Latency Hierarchy

- Distinct step-wise latency increments confirm correct cache–memory separation.
- L1/L2 latencies ≈ 1 ns → on-chip access; DRAM ≈ 10 ns → off-chip.

6.2 CPU Performance

- Integer operations complete fastest due to register-only dependency.
- Floating point pipeline adds substantial latency.
- Memory copy test validates sustained bandwidth capability.

6.3 Power Analysis

- Energy scales nearly linearly with latency.
- DRAM access consumes ~9× the energy of L1 cache access.
- Confirms typical hierarchical energy gradient:
  L1 < L2 < L3 < DRAM

7. Observations

Aspect	Observation
Latency Calibration	Accurate 4-level hierarchy derived from memlat output.
CPU Tests	Benchmarks confirm simulator integrity and throughput.
Power Model	Produced consistent per-level energy ratios matching literature.
Integration	Phase 4 unifies latency data with power estimation for next-phase dashboard.

8. Directory Structure

```
~/ACA/
└── phase4_cpu_perf/
    ├── results/
    │   ├── memlat_output.txt
    │   ├── latencies_only.txt
    │   └── power_summary.csv
    ├── tests/
    │   ├── int_arith
    │   ├── float_arith
    │   ├── mem_copy
    │   └── mix_load
    └── power_model/
       ├── power_model.py
       └── results/
          └── power_summary.csv
```

9. Summary of Key Learnings

- Verified end-to-end latency, performance and energy linkage.
- Demonstrated that latency impacts both IPC and power consumption.
- Established a realistic per-level energy profile for SimpleScalar.
- Created a modular Power Model script usable for extended studies.
- Validated simulator stability and quantitative correctness across CPU and memory layers.

PHASE – 5 : CPU Performance Dashboard – Visualization and Analysis of CPU Performance and Power Model

Objective

- To develop an **interactive CPU performance and power modeling dashboard** that integrates and visualizes results from all previous phases (1 → 4).
- This phase transitions the project from raw command-line output to a **web-based analytical dashboard** built using **Flask + Plotly Dash + Pandas**, providing a unified visualization of simulation metrics, memory latencies and power/energy data.

Workflow

Step	Task	Outcome
1	Create project structure and sub-directories	Initialized /home/aravind/ACA/phase5_dashboard/ with scripts/, data/, logs/ folders
2	Set up Python 3.12 virtual environment	Isolated environment ready for Flask + Dash
3	Installed all dependencies (pandas, flask, dash, plotly, matplotlib)	Dashboard dependencies verified
4	Collected Phase-4 latency + power data	Created memlat.csv, power_summary.csv, summary_report.txt
5	Developed Dash web app	Integrated all phases into interactive web dashboard
6	Added data logger + launch script	Enabled automatic refresh and one-click startup
7	Validated dashboard output	Interactive graphs + tables confirmed

1. Environment Setup

1.1 Directory Initialization

```
mkdir -p /home/aravind/ACA/phase5_dashboard/{scripts,data,logs,templates,static}
cd /home/aravind/ACA/phase5_dashboard
```

1.2 Virtual Environment Setup

```
sudo apt install -y python3.12-venv
python3 -m venv /home/aravind/ACA/phase5_dashboard/venv
source /home/aravind/ACA/phase5_dashboard/venv/bin/activate
```

1.3 Dependency Installation

```
pip install --upgrade pip setuptools wheel
pip install pandas flask matplotlib plotly dash flask-cors numpy
```

Installed packages verified via `pip list`.

2. Data Preparation

2.1 Data Sources

- **Phase-1/2** IPC & CPI Metrics
- **Phase-3** Latency Hierarchy
- **Phase-4** Latency: `/home/aravind/ACA/phase4_cpu_perf/results/memlat_output.txt`
- **Phase-4** Power: `/home/aravind/ACA/phase4_cpu_perf/power_model/results/power_summary.csv`

2.2 Data Collection Script

`collect_phase4_data.py` automatically:

- Parses Phase-4 latency data
- Converts to CSV → `memlat.csv`
- Copies `power_summary.csv` and `latencies_only.txt`
- Writes `summary_report.txt`

Output Files

/home/aravind/ACA/phase5_dashboard/data/memlat.csv
/home/aravind/ACA/phase5_dashboard/data/power_summary.csv
/home/aravind/ACA/phase5_dashboard/data/latencies_only.txt
/home/aravind/ACA/phase5_dashboard/data/summary_report.txt

3. Dashboard Application

3.1 Toolchain

- **Framework:** Flask + Plotly Dash
- **Frontend:** HTML + Plotly Graphs
- **Backend:** Python (pandas for data parsing + auto-refresh)

3.2 Dashboard Features

Component	Function
Summary Table	Aggregates metrics from Phases 1–4 (IPC, CPI, latency, energy)
Phase-1 Graph	Displays baseline IPC and CPI values
Phase-2 Graph	Compares prefetching schemes (Baseline / Next-Line / OBL / Stride)
Phase-3 Graph	Plots latency vs access type (Pointer, Index, Stride)
Phase-4 Graph 1	Latency vs Memory Size curve
Phase-4 Graph 2	Energy per access bar chart
Phase-5 Graph	Consolidated cross-phase overview (IPC / CPI / Energy / Latency)

3.3 Execution

```
chmod +x /home/aravind/ACA/phase5_dashboard/scripts/dashboard_app.py
python /home/aravind/ACA/phase5_dashboard/scripts/dashboard_app.py
```

Dashboard launches on **http://127.0.0.1:8050**

4. Automation and Logging

4.1 Logger Script

logger.py continuously monitors /data/ directory and logs updates every 30 s to:

```
/home/aravind/ACA/phase5_dashboard/results/dashboard_log.txt
```

4.2 Unified Launch Script

start_dashboard.sh

```
#!/bin/bash
cd /home/aravind/ACA/phase5_dashboard
source venv/bin/activate
nohup python scripts/logger.py >/dev/null 2>&1 &
python scripts/dashboard_app.py
```

Make it executable and run:

```
chmod +x /home/aravind/ACA/phase5_dashboard/start_dashboard.sh
/home/aravind/ACA/phase5_dashboard/start_dashboard.sh
```

5. Results and Visualization

Visualization	Description
📊 Summary Table	All key metrics (IPC, CPI, Latency, Energy) across Phases 1–4
Phase-1 Graph	CPI ≈ 1.07, IPC ≈ 0.94 — baseline performance
Phase-2 Graph	Best IPC = 1.324 with One-Block Lookahead Prefetch
Phase-3 Graph	Latency curve shows L1 ≈ 1.1 ns → DRAM ≈ 10 ns
Phase-4 Graph	Energy hierarchy: L1 ≈ 0.15 pJ → DRAM ≈ 1.32 pJ
Phase-5 Graph	Unified comparison of IPC, CPI, latency & energy trends

All plots auto-refresh every 30 s and update when new Phase-4 data arrives.

6. Directory Structure

```
ACA/
└── phase5_dashboard/
    ├── venv/
    ├── data/
    │   ├── memlat.csv
    │   ├── power_summary.csv
    │   ├── latencies_only.txt
    │   ├── summary_report.txt
    │   └── all_phase_summary.csv
    ├── results/
    │   └── dashboard_log.txt
    ├── scripts/
    │   ├── dashboard_app.py
    │   ├── collect_phase4_data.py
    │   └── logger.py
    ├── start_dashboard.sh
    └── README_phase5.txt
```

7. Observations

- Dashboard provides **real-time performance insight** for all prior phases.
- **Prefetching improvements** and **power trade-offs** are now visually correlated.
- **Latency and Energy** hierarchies validate the simulated cache-memory model.
- System is modular: adding Phase 6 (pipeline viewer/logging) will require only new CSV inputs.

8. Conclusions

Aspect	Observation
Integration	Successfully combined metrics from Phases 1 to 4 into a live dashboard.
Technology Stack	Flask + Dash + Plotly + Pandas enabled responsive visualization.
Visualization	Multi-phase data displayed interactively (auto-refresh + tables).
Power Model	Energy scales logically with memory hierarchy (L1 → DRAM).
Automation	Logger and launcher scripts enable continuous monitoring.
Outcome	A complete, self-contained web dashboard for CPU performance analysis.

9. Summary of Key Learnings

- Integration of analytical and simulation phases via a **unified web interface**.
- **Dash + Plotly** simplifies live plotting of system metrics.
- Linking performance and power/energy provides holistic processor insight.
- Modular pipeline ensures future scalability (Phase 6 and beyond).
- The dashboard marks the transition from raw experimentation to **real-time visual analysis**.