Project Title:
Memory Latency Profiling and Software Prefetching on the Raspberry Pi 5

Project Goal:
The goal of this project is to empirically characterize the memory hierarchy of the Raspberry Pi 5 and to evaluate the effectiveness of software prefetching as a technique to mitigate memory access latency. Using custom C++ microbenchmarks, the project will measure memory access behavior across cache levels and main memory, then analyze how explicit software prefetching impacts performance on memory-bound workloads.

Hardware Requirements: - Raspberry Pi 5 (ARM Cortex-A76 CPU) - Active cooling solution (heatsink and/or fan) to prevent thermal throttling during sustained measurements - Optional: USB power meter for optional energy/performance discussion

---

Software Requirements: - Operating System: Raspberry Pi OS (64-bit recommended) - Programming Language: **C++** (for all benchmarks and experiments) - Tools: - `perf` (hardware performance counters for cycles, instructions, cache misses, IPC) - Python with matplotlib (for plotting and visualization of results) - Optional: `valgrind` / `cachegrind` for supplementary memory behavior analysis (non-cycle-accurate)

Project Deliverables

Deliverable 1: Memory Latency Profiler (C++ Microbenchmark)

- A custom C++ microbenchmark that:
- Allocates memory buffers of increasing working set sizes
- Uses a controlled access pattern (e.g., dependent accesses or pointer chasing) to expose true memory latency
- Measures average memory access latency for each working set size
- Output data recorded in a structured format (e.g., CSV)

Deliverable 2: Memory Hierarchy Characterization

- Plots of average memory access latency versus working set size
- Identification of distinct latency regions corresponding to:
- L1 cache
- L2 cache
- Main memory (DRAM)
- Discussion of observed transition points between cache levels

---

Deliverable 3: Baseline Memory-Bound Workload

- Selection and implementation of a memory-sensitive workload in C++ (e.g., matrix multiplication variant, array traversal kernel, or graph-style access pattern)
- Baseline version without explicit software prefetching

Deliverable 4: Software Prefetching Implementation

- A modified version of the same workload that includes manual software prefetching
- Clear documentation of:
- Prefetch placement within the code
- Chosen prefetch distance
- Rationale for the chosen strategy

Deliverable 5: Performance Evaluation and Comparison

- Quantitative comparison between baseline and prefetch-enabled implementations using:
- Execution time
- Instructions per cycle (IPC)
- Cache and memory-related performance counters from `perf`
- Plots and tables summarizing performance differences
- Notes on experimental controls (consistent runs, cooling, CPU frequency behavior, optional core pinning)

Deliverable 6: Analysis and Final Report

- Interpretation of measured results in terms of computer architecture concepts
- Discussion of:
- When and why software prefetching improves performance
- Cases where prefetching provides little benefit or degrades performance
- The relationship between memory access patterns, cache behavior, and latency hiding
- Connection of observed behavior to underlying microarchitectural design principles

---

Optional Extension: Adaptive Software Prefetching

- Implementation of a simple adaptive prefetcher that:
- Detects memory access stride at runtime
- Adjusts prefetch distance dynamically
- Comparison against static prefetching strategies
- Extended analysis discussing trade-offs and effectiveness

Project Summary:

This project will use carefully designed C++ microbenchmarks on real ARM hardware to expose memory hierarchy behavior and evaluate software-based techniques for mitigating memory latency. The results will demonstrate how architectural constraints influence performance and how software can be structured to better exploit modern CPU designs.