Lab 1: Simulating and Exploring Cache Behavior

Introduction

In this lab, I extended a timing simulator for a five-stage pipelined MIPS processor with separate L1 instruction and data caches. I wrote a benchmarking framework to analyze the effects of various cache design parameters and replacement policies. To test the cache's effectiveness in representative and meaningful situations, I extended the provided set of test cases.

Cache Implementation Details

Cache Data-Structure The instruction cache (I-Cache) and data cache (D-Cache) are based on a generic implementation of a set-associative cache in `src/cache.c` and `src/cache.h`. Since the timing simulator doesn't need to model the full data-path details, the cache data structure only tracks the most necessary metadata. This includes an array of the cache's sets, the blocks inside the sets and the related metadata for correctly handling cached data depending on the replacement policy. A cache access function checks if the memory address in its argument is a hit or a miss, updates the internal data structures and returns the incurred number of cycles to stall the calling stage. The data to be managed is handled directly using the provided `mem_read_32` and `mem_write_32` functions.

`Pipe_State` Integration The pipeline data-structure `Pipe_State` is extended with a I-Cache and D-Cache `struct`. Two stall cycle counters are added to track the remaining number of cycles to stall the fetch or memory stages respectively. The cache organization, miss timings, dirty evictions and assumptions about the I-Cache / D-Cache are set as specified in the task requirements. Pipeline flushes to stages waiting on memory requests do not cancel the memory requests, the stall cycle counters are not reset.

Instruction Cache The I-Cache is used in the fetch stage of the pipeline. If the cache access determines a stall is necessary, the stage's stall cycle counter is set and the stage returns early. In later cycles, the stage only progresses once the stall cycle counter has been decremented till zero. A quirk of this mechanism is that the program counter `PC` can't be incremented during a `syscall` if the fetch stage is stalled. To ensure the program terminates with the correct `PC` value if the fetch stage is stalled, it is manually incremented in the `syscall` handler.

Data Cache The D-Cache is used in the mem stage of the pipeline and features the same stalling mechanism as the fetch stage. Due to the given specification, that dirty blocks *"are written immediately into main memory in the same cycle as when the new block is inserted into the cache"*, stores don't stall the pipeline beyond the initial read at the beginning of the stage.

Replacement Policies I implemented three different replacement policies, which are activated with compiler flags.

1. LRU (Least Recently Used): Maintains a recency counter for each block. On insertion, evicts the block with the highest (oldest) counter value. On access, resets the accessed block's counter and increments all others.
2. RRIP (Re-Reference Interval Prediction): Implements a scan-resistant policy using 2-bit counters. New blocks are inserted with a distant re-reference prediction (value 2), and blocks are promoted on hits. This protects the cache from streaming access patterns that pollute LRU caches (Jaleel et al. 2010).
3. Random: Selects a victim block uniformly at random from the set. This provides a simple baseline, which demonstrates the other policies' ability to exploit sequential access patterns.

Verification I verified correctness by comparing the register dumps with the baseline sim. I validated the cycle counts by checking the miss penalties (as multiples of 50) for the individually activated I-Cache and D-Cache on simple input programs.

Experimental Methodology

Benchmark Suite

Input Programs To comprehensively explore the cache behavior, I prepared a set of program that test cache performance under a broad spectrum of memory access patterns. Each program is designed to determine the effect of a specific cache parameter or replacement policy. I reused some programs provided in the base project folder.

Other programs I wrote in C and converted to machine code using godbolt.org's `MIPS GCC 15.2.0` compiler and the MARS simulator.

1. **Sequential-8**: Performs contiguous sequential memory access with high spatial locality. This scenario lets the D-Cache demonstrate it's ability to exploit sequential access patterns.
2. **Pointer Chase (custom)**: Traverses a large linked list with pseudo-random memory accesses (using the MINSTD LCG). This represents irregular D-Cache access with low spatial locality.
3. **Strided Access (custom)**: Accesses array elements with a fixed stride, not aligned with the block size. This tests the effect of the D-Cache block size when only a small amount of data from a block is used.
4. **Stream Reuse (custom)**: Combines streaming accesses with temporal reuse by revisiting previously accessed data. This tests replacement policies' ability to distinguish between data that will be reused versus data accessed only once, making it particularly relevant for evaluating LRU versus scan-resistant policies like RRIP.
5. **Looped Random 1k (custom)**: Repeatedly executes randomly generated instructions within a ~6KB instruction footprint, demonstrating the effect of I-cache size. This stresses the I-Cache with a working set that fits in larger I-caches but thrashes smaller ones.
6. **Primes (Sieve)**: Implements the Sieve of Eratosthenes for finding prime numbers. This represents a "real-world" computational workload with a mixed memory access pattern.

Parameter Sweep I parametrized the sim implementation to support runtime configuration of the caches using a custom command defined in `shell.c`. A modified version of the `run.py` file generates the parameter sweep and runs the simulators in parallel. Results were collected in CSV format for analysis, tracking `IPC` and cycle counts for each program/configuration pair. All powers of two within the following parameter ranges were tested:

- Cache size : [`1KB`, `2KB`, …, `512KB`, `1024KB`]
- Block size: [`4B`, `8B`, …, `256B`, `512B`]
- Associativity: [1, 2, …, 8, 16]
- Replacement policy (LRU, Random, RRIP)

The sweep includes only the physically valid combinations of the cache parameters, i.e. where the set size is at least the block size. For simplicity, D-Caches and I-Caches with different block sizes or different replacement policies aren't considered. This doesn't seem to be common practice in real world L1 caches.

Results and Discussion

To compare cache parameters with a baseline, a reference configuration was used (LRU replacement policy, 64 KB 8-Way D-Cache and 2KB 4-Way I-Cache with a 32 byte block size). Then with the remaining parameters kept fixed, the cache size, associativity, block size and replacement policy were varied.

Effect of Cache Size

fig. 1 shows the effect of cache size on IPC across different workloads. For the I-Cache, the roughly 1.5k instructions fit in about 6KB of memory. So the IPC gets a lot better on repeated access after the I-Cache size goes from 4KB to 8KB. Most of the programs in the benchmarking suite fit within 2KB of I-Cache, so the I-Cache size has to be selected to cache the typical program size. Regarding the Data Cache

Effect of Block Size

Effect of Associativity

Replacement Policy Comparison

Discussion of Observations

Citations

Jaleel, Aamer, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)." *SIGARCH Comput. Archit. News* (New York, NY, USA) 38 (3): 60–71. https://doi.org/10.1145/1816038.1815971.

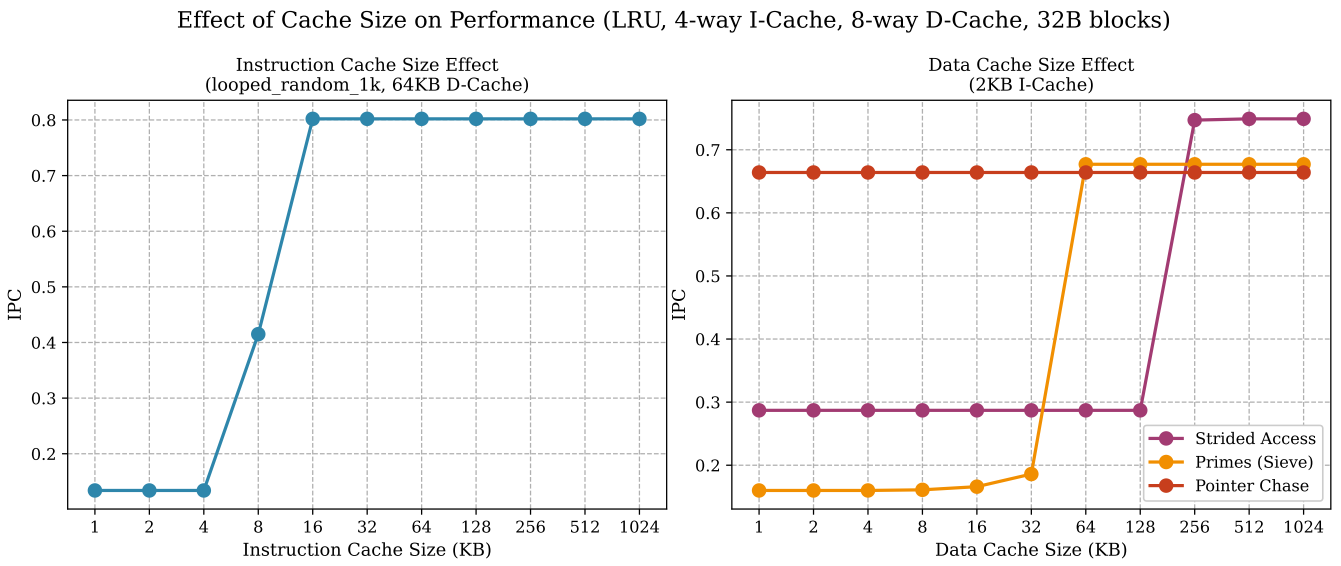

Figure 1: Effect of Cache Size on IPC (LRU, 4-Way I-Cache, 8-Way D-Cache, 32B Blocks)