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 progressively larger strides (1, 2, 4, 8, 16…), creating a pattern that causes extensive thrashing in smaller D-Caches.
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 Observations

To systematically evaluate cache performance, I established a baseline configuration and varied one parameter at a time while holding others constant. The baseline configuration consists of an LRU replacement policy with a 64 KB 8-way D-Cache, a 2 KB 4-way I-Cache, and 32-byte blocks for both caches.

Effect of Cache Size

I-Cache In fig. 1, the effect of cache size on IPC across different workloads is demonstrated. Without jumps or branches, the I-Cache is generally accessed sequentially when the program counter gets incremented. Therefore the I-Cache size yields a sharp performance increase when the cache becomes large enough to hold a program's instruction footprint. The Looped Random 1k program, contains approximately 1,500 instructions occupying roughly 6 KB of memory. So the IPC improves as I-Cache size increases from 4 KB to 8 KB, showing the benefit of eliminating instruction cache misses once the entire working set fits in cache.

Most programs in the benchmarking suite have small instruction footprints (<2KB), showing minimal performance variation across I-Cache sizes. This suggest that in the real-world, programs with larger instruction working sets require proportionally larger I-Caches to avoid performance degradation.

D-Cache The varying effects of D-Cache size across different memory access patterns are shown in fig. 1. Two performance trends are discernible:

Performance-scaling programs: Primes, Strided Access, and Stream Reuse show significant performance gains with increasing cache size. For these programs, performance exhibits a sharp transition when the cache capacity exceeds the working set size.

The Primes benchmark has a working set of about 64KB (the sieve array), resulting in improved IPC when D-Cache size reaches this threshold.

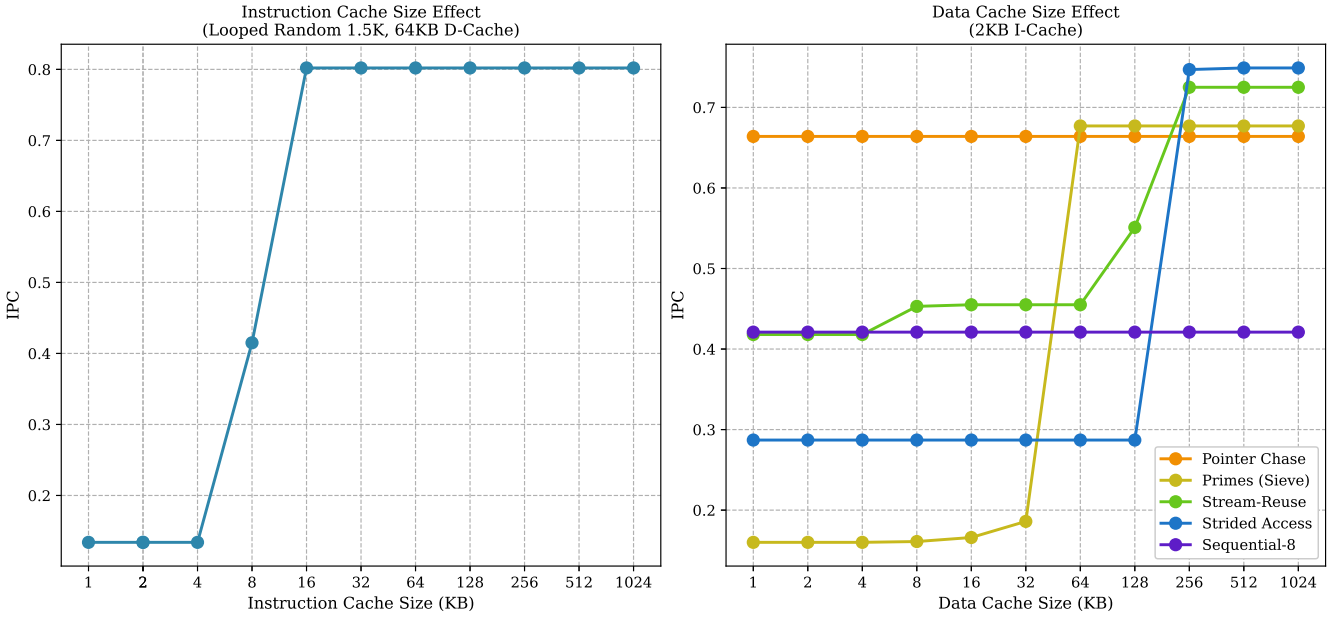

Figure 1: Effect of Cache Size on IPC (LRU, 4-Way I-Cache, 8-Way D-Cache, 32B Blocks)

Similarly, Strided Access benefits once the cache can hold its entire array ($64 \cdot 1024 \cdot 4B = 256KB$). In configurations where the cache capacity is insufficient, thrashing occurs as each new stride pattern walks through memory faster than the cache can accommodate, causing previously cached blocks to be evicted before they can be reused. When cache size increases to match the working set (256KB), thrashing is eliminated because all array elements can remain resident simultaneously.

Stream Reuse's IPC performance increases once after the 8KB of the frequently accessed working set fit inside the cache. The performance increases again once the 128KB of streaming data fit inside the cache too. This eliminates most capacity misses and reduces thrashing from the streaming accesses displacing hot data.

Limited-benefit programs: Pointer Chase shows no improvement with larger cache sizes. Due to the random access pattern, there is little temporal locality for the cache to exploit. Even when the entire working set fits inside the cache, the random access pattern creates frequent conflict misses among the sets. Furthermore, only 2B out of the 32B blocks are actually used per access, leading to cache pollution. In fact 93.75% of each 32B block contains irrelevant information. This combination of random access pattern and poor use of cache space means even large caches provide limited benefit.

Takeaway These observations highlight how memory access patterns determine the relationship between cache size and performance. Programs with well-defined working sets and some degree of locality benefit most from appropriately sized caches, while those with no locality don't benefit from cache size.

Effect of Block Size

As shown in fig. 2, increasing the block size generally improves performance up to 128B for most programs, after which performance drastically declines. This pattern reflects the trade-off in block size selection:

Size <= 128B: Most programs benefit from larger block sizes (until 128B) for two main reasons: First, the 50-cycle miss penalty is shared by more bytes per miss. Second, in sequential access patterns the data in the next few requests effectively gets prefetched.

Size > 128B: The performance collapses beyond a block size of 128B, because larger blocks mean fewer sets. This creates conflict misses, since unrelated addresses map to the same set. Furthermore, programs with irregular access patterns like Pointer Chase effectively waste a lot of the space in a block.

Takeaway These observations demonstrate that optimal block size depends on balancing spatial locality benefits against set reduction costs.

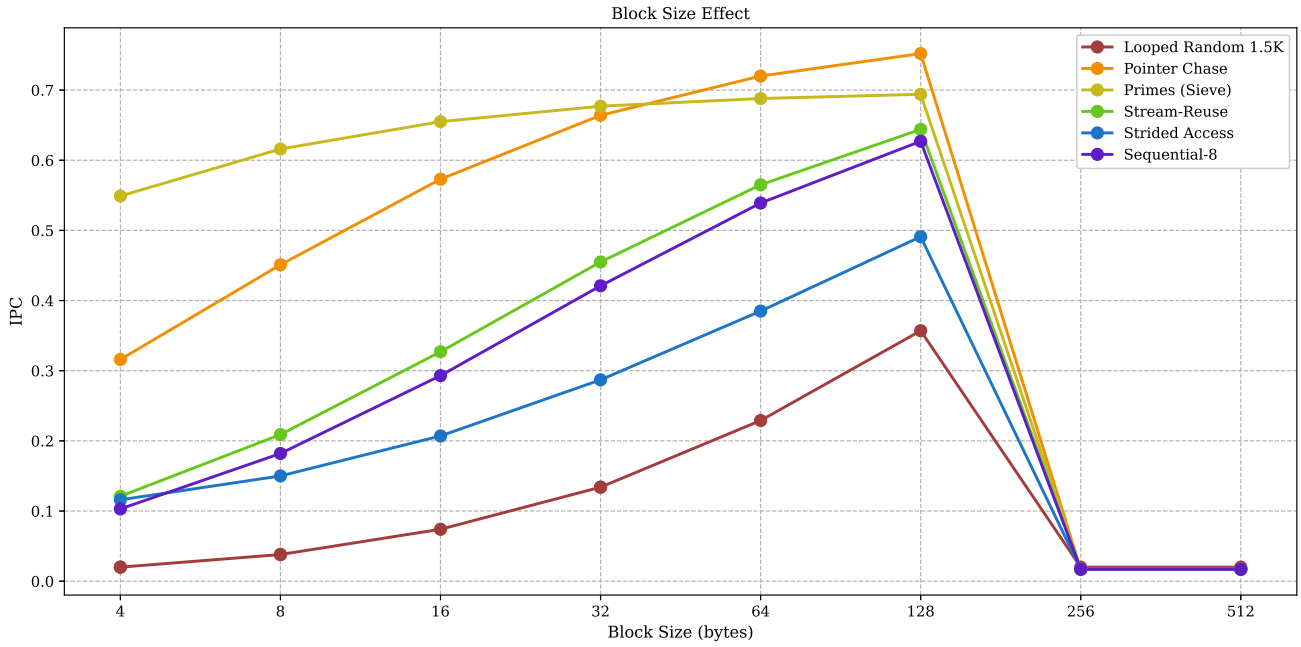

Figure 2: Effect of Block Size on Performance (LRU, 2KB I-Cache 4-way, 64KB D-Cache 8-way)

Effect of Associativity

As can be seen in tbl. 1, most programs (Pointer Chase, Primes, Sequential-8) show essentially no performance variation across associativity levels. For these workloads, going from direct-mapped (1-way) to fully associative (16-way) produces identical cycle counts. This suggests that conflict misses are not a significant performance factor compared to capacity misses or compulsory misses for these programs.

Table 1: Effect of D-Cache Associativity on IPC (LRU, 2KB 4-way I-Cache, 64KB D-Cache)

Program	1-way	2-way	4-way	8-way	16-way
Pointer Chase	0.664	0.664	0.664	0.664	0.664
Primes	0.677	0.677	0.677	0.677	0.677
Sequential-8	0.421	0.421	0.421	0.421	0.421
Stream Reuse	**0.453**	0.455	0.455	0.455	0.455
Strided Access	0.287	0.287	0.287	0.287	**0.286**

Interestingly, Stream Reuse shows a small approximately 0.25% increase in performance when increasing from 1-way to 2-way associativity. This stems from its use pattern, where the streaming data can share a set with the frequently accessed data on tag conflicts.

Takeaway This demonstrates that higher levels of D-Cache associativity aren't justified in their added complexity. The performance gain is minimal at beyond 2-way associative sets.

Replacement Policy Comparison

Table 2: Effect of Replacement Policy on IPC (2KB 4-way I-Cache, 64KB 8-Way D-Cache)

Program	Random	LRU	RRIP
Pointer Chase	0.664	0.664	0.664
Primes	0.677	0.677	0.677
Sequential-8	0.421	0.421	0.421
Stream Reuse	**0.484**	0.455	0.455
Strided Access	**0.290**	0.287	0.287

As shown in tbl. 2, changing the caching policy makes no difference in cache IPC performance for the Pointer Chase, Primes and Sequential-8 programs. Meanwhile Stream Reuse and Strided Access, the programs that benefit from high cache scan resistance, show a 6.02% and 0.99% increase in IPC by using the LRU and RRIP policies instead of evicting a random block. This shows that they benefit from trying to keep temporally local data inside the cache.

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.