Computer Architecture HS2025 Lab1 Report

Frederic zur Bonsen

October 2025

1 Introduction

In this lab, we were asked to extend a timing simulator to model instruction/data caches. The timing simulator works on the MIPS instruction set and is written in C. It implements a five-stage CPU with adjacent L1 instruction and data caches. These caches were benchmarked with a diverse set of test cases. Parameter variations of the cache size, block size, and associativity of the cache were performed to understand the respective impact on the caches on the performance. Several cache eviction policies were implemented and benchmarked to understand their impact on cache performance.

2 Cache Implementation

2.1 Cache

As both an instruction and a data cache had to be implemented a generic dynamic data structure was chosen given in src/cache.h and src/cache.c that allows for variation of parameters. As the caches only need to implement the timing simulation and not the memory path, the implementation only implements the metadata, i.e. the position and state of memory in the cache and the state of the cache instead of the full cache with memory. In src/cache.c there are multiple implementations of the simulator each appertaining to an eviction policy. The functionality of the cache simulator is as follows. It is called by the pipeline and given an address, as well as the cache which is being accessed. It then checks whether this address is a cache hit or miss, if it is a miss it adds it to the cache and if necessary evicts an entry following the selected policy. It then returns the number of cycles necessary to get the memory. This is zero cycles in case of a cache hit and fifty cycles in case of a cache miss.

2.2 IF-Stage

In the instruction fetch stage, the CPU fetches an instruction either from memory or from the instruction cache. In case of a cache miss the entire stage is stalled for fifty cycles. This is handled with a simple flag and counter. Where the flag is used to indicate if the stall is to be resolved in this cycle or not and the counter is used to keep track of the remaining cycles the stage needs to be stalled for.

2.3 MEM-Stage

In the memory stage, the CPU reads/writes from/to memory or the data cache. The cache handling is a bit more involved than in the instruction fetch stage. In case of a memory instruction, a cache access is performed resulting in either a hit or a miss. A hit has no consequence, but a miss stalls any successive memory instructions. Any non-memory instructions before the next memory instruction do not get stalled. This is handled with the same counter and flag logic used in the instruction fetch stage.

2.4 Pipeline Remarks

To fully understand the implementation a few additional remarks are necessary. This implementation of the pipeline and caches does not reset cache stalls in case of a pipeline flush. Also dirty evictions are assumed to take zero time to write back into memory.

2.5 Eviction Policies

LRU Least-Recently-Used evicts the entry that was used the longest time ago. Each entry has a counter indicating its order in the LRU chain, which is updated whenever an entry is accessed.

FIFO First-In-First-Out evicts the element that has been in the cache the longest. Each set has a global counter to assign positions to new entries, and each entry stores its internal position to determine eviction.

RANDOM Randomly selects an entry to evict. This is a good baseline as it is a "stupid" policy. Other policies can be compared against it to assess their performance.

3 Methodology

3.1 Verification

The code was verified by using the run.py script provided, which compares a set of test cases between a baseline implementation without cache simulation and the version with implemented cache simulation. This allows for verification of functional correctness of the simulator. To verify the correctness of the timing simulation, direct comparison of the performance of some of the simple test cases with an expected result were used. In addition, for the more complex test cases the results were compared with peers.

3.2 Benchmark

To perform comprehensive tests on the performance of different parameter configurations and eviction policy choice a set of tests were selected to be used as a benchmark. This benchmark consists of several of the provided tests as well as some tests that were implemented in C and then compiled with https://godbolt.org's MIPS GCC 15.2.0 compiler and the MARS simulator as provided, and some that were purpose built by hand and by python-script.

icache_fill (Custom) This test performs successive similar operations. The goal of this test is to fill and test the instruction cache. This is spatially local in the instruction cache.

sequential_access (Custom) This test performs sequential access to memory by iterating over different sequence lengths n and then iterating over an array always accessing the next n memory entries in sequence. This has high spatial locality and moderate temporal locality as for shorter sequences big parts of these sequences are repeatedly accessed in a short time.

strided_access (Custom) This test performs strided access to memory by iterating over different stride distances and then accessing memory in these stride intervals. This simulates access to matrix like data structures of variable sizes. This has neither temporal nor spatial locality. And is used to test the data cache.

pointer_chase (Custom) This test performs a random pointer chase. The memory accesses are neither temporally nor spatially local. This emulates the accessing of graph structures.

matmul (Custom) This test performs matrix multiplication. Matrix multiplication is a common application. This test therefore emulates a real world use case.

levenshtein (Custom) This test calculates the Levenshtein distance, also known as edit distance, of two strings. This is operation is performed very often in genomics, as part of non gap-affine sequence-to-sequence alignment. It is a good test of a realistc workload and emulates a real world use case.

primes This is the provided primes.x. It implements the sieve of Eratosthenes which is used to represent a normal program that could realistically be run on the CPU.

random1 This is the provided random1.x. This is used to test the instruction cache.

3.3 Parameter Sweep

A parameter sweep was performed on the cache size, block size and associativity of the caches. This was implemented by using the dynamic cache structure that allows for parameter variation. A parameter sweep was performed by running a python-script to run a modified version of the simulator that requests the parameters as additional inputs.

- cache sizes: [1*1024, 2*1024, ..., 1024*1024]
- block sizes: [4, 8, ..., 256]

- associativities: [2, 4, ..., 64]

These parameter sweeps run on all the implemented eviction policies to get a comprehensive picture of the performance of each policy.

- policies: [LRU, FIFO, RANDOM]

To perform the parameter sweep in a reasonable time frame only one parameter was varied at once and the other two were fixed. Cache size was fixed at 32KB, block size at 64B, and the associativity at 4 ways.

4 Results

Not all results will be presented here as this would overwhelm the reader and would neither be reasonable nor interesting. Therefore the here presented results are the ones deemed interesting and worthy of discussion by the author.

4.1 Cache Size

Instruction Cache The access to the I-Cache happens in the IF-stage and is strongly sequential. Therefore the I-Cache sees a strong increase in use for applications that exceed the I-Cache size. Over all the tested applications no big difference in I-Cache performance could be observed.

Data Cache Figure 1 shows the effect of the data cache size on the performance. It can be seen that with increasing cache size the performance also increases up to a limit which is the optimal performance of the algorithm , i.e. the version where the cache is big enough so that there are no cache misses.

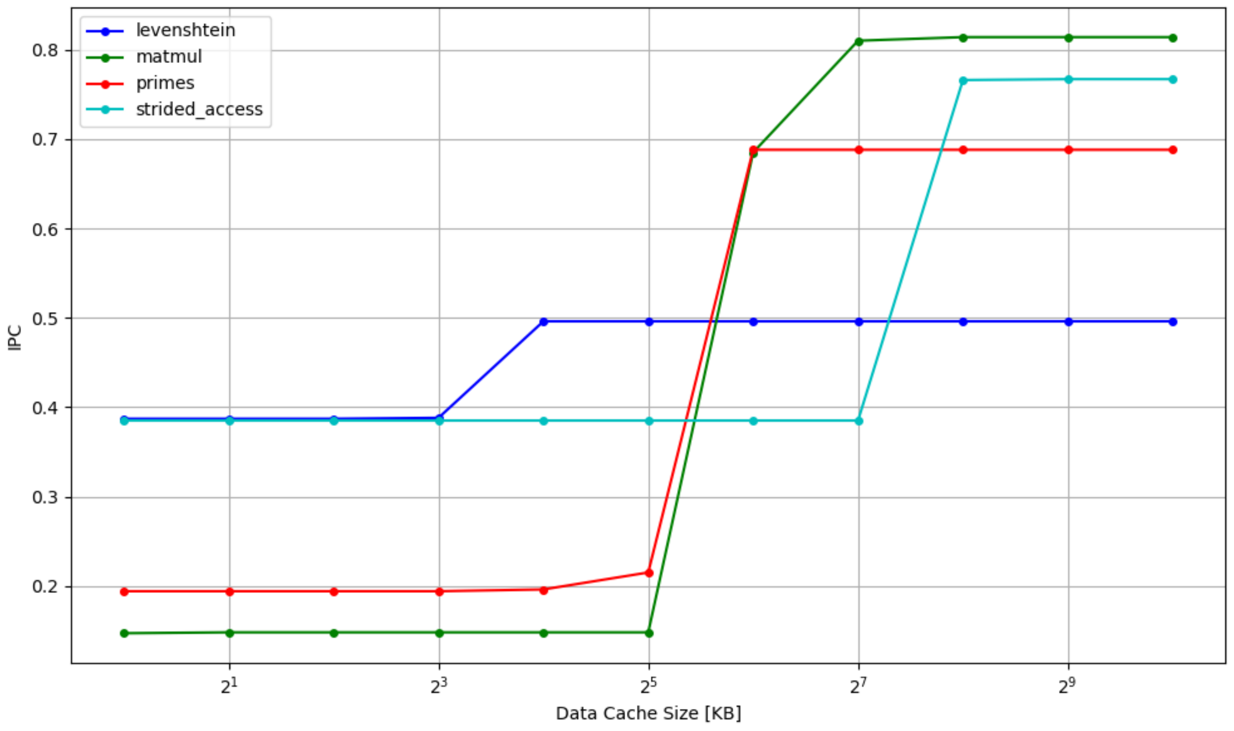


Figure 1: Plot of IPC in respect to the data cache size for strided_access and primes using the LRU eviction policy.

4.2 Block Size

Figure 2 shows the impact of variable block sizes on the performance in a spatially local process. We see that in both the increase in instruction cache and data cache size improve the performance significantly.

Figure 3a and Figure 3b demonstrate the impact that processes with over proportional usage either of the caches have on the respective requirements. Figure 3a shows that random1 which very intensively uses the instruction caches benefits extremely from instruction cache block size increases while not being affected

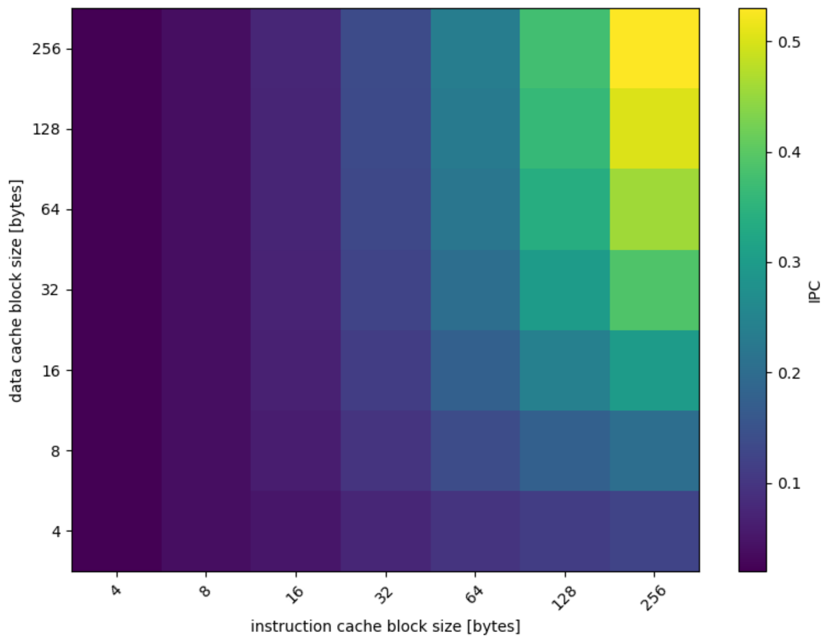
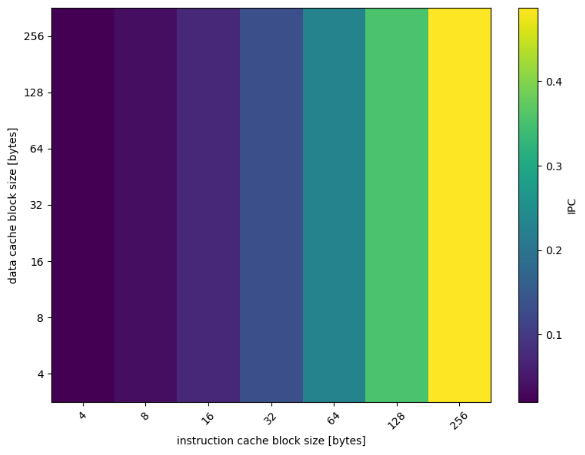
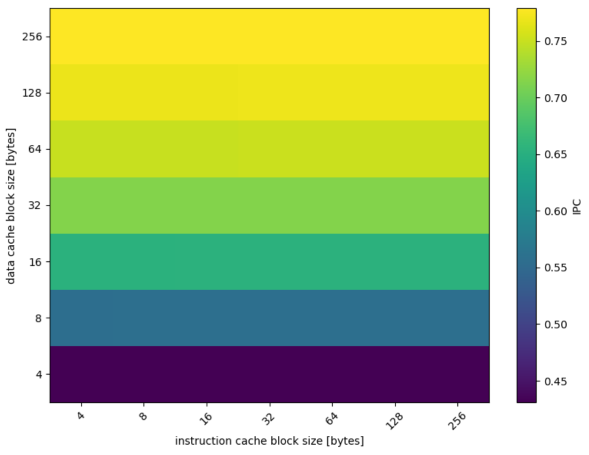


Figure 2: Heatmap of the cache block sizes of both the data and the instruction caches using the LRU eviction policy performing icache_filler.

by data cache block size. Figure 3b on the other hand shows that a sequential_access which is a very data cache intensive process with a lot of memory accesses benefits only from data cache block size increases and not from instruction cache block size increases.



(a) Heatmap of the cache block sizes of both the data and the instruction caches using the LRU eviction policy performing random1.



(b) Heatmap of the cache block sizes of both the data and the instruction caches using the LRU eviction policy performing sequential_access.

4.3 Associativity

Figure 4 shows the effect of changing associativity. Here the performance increases with higher associativity. In general higher associativity increases performance for spatially local processes as it decreases conflict misses. And increases the hit rate.

4.4 Eviction Policies

As stated above the three eviction policies LRU, FIFO, RANDOM were implemented. There were no performance differences observed between FIFO and LRU. The RANDOM policy performed slightly worse in the very large test cases such as sequential_access or strided_access but as these differences were about 0.0024% of performance they can be qualified as negligible.

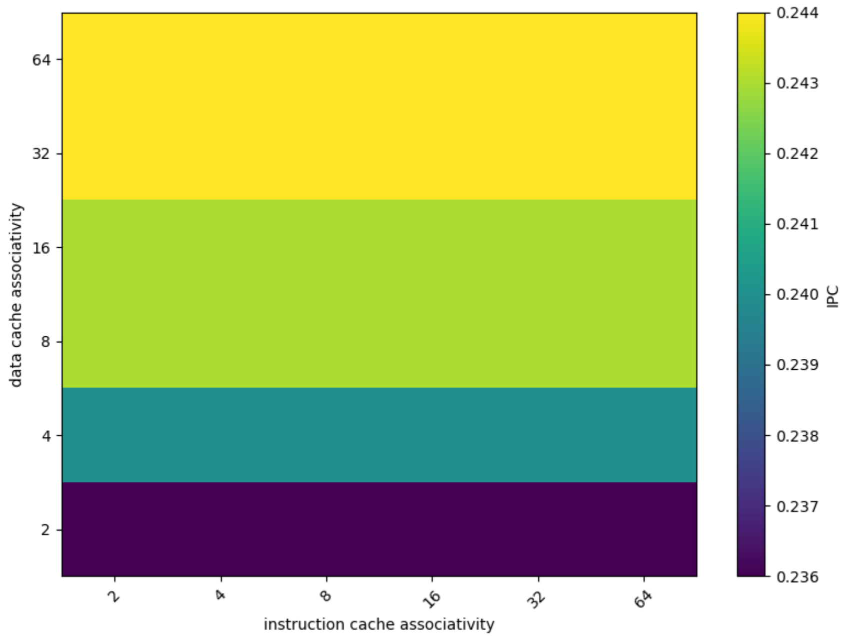


Figure 4: Heatmap of the cache associativity of both the instruction and data cache using the random eviction policy performing primes.

5 Discussion

From the results it can be seen that both for cache size and cache block size an increase also increases performance. But this is a fallacy that stems from the assumptions made when designing the simulator. The simulator does not attribute any cost to bigger caches or bigger block sizes. In reality a bigger caches come with a cost. They incur a higher access latency as well as higher power consumption. Both of which are factors this simulator does not take into account. Bigger block sizes also come with a cost. For a bigger block it takes longer to load the data from memory on a hit and the same goes for a dirty eviction.
The associativity results are as expected. However, the benefit of higher or lower associativity varies with the process that is run. In general an ever increasing associativity will only create diminishing returns which in a real live scenario would be overwhelmed by access latency of having to parse a highly associative cache.
Another surprising result is that the difference in eviction policy has little to no effect on the performance. This can be the consequence of two things. Either a faulty implementation of one of the eviction policies or a set of test cases that do not properly cover the differences in eviction policies.
A further interesting extension of this simulator would be the measuring of hit and miss rates as these would also be interesting values to assess the performance of the caches.