RS: First Homework

Mark Loboda in Simon Goričar

1. 4. 2025

1 Introduction

In this project, we used the `GEM5` simulator, an open-source computer-system architecture simulator. It allows users to analyze and test hardware configurations and architectures without actually developing or having to produce any hardware physically. In this report, we're evaluating the performance of the *O3 processor* model using the *Whetstone benchmark* and analyzing the influence of cache size and associativity on the performance of a tiled matrix multiplication program.

2 Benchmarks

2.1 First task: O3 processor

2.1.1 The effect of issue width and reorder buffer size on instructions per cycle

We used the *GEM5* simulator to run the *Whetstone* benchmark on different configurations of the *O3* processor. We performed the benchmark using the combination of the following parameters:

- **Issue Width:** 1, 2, 4, 8
- **Reorder Buffer (ROB) Size:** 2, 4, 8, 16, 32

The parameters we configured will likely directly affect how well the processor can exploit instruction-level parallelism (ILP) in out-of-order execution. The **issue width** is the maximum number of instructions that can be issued in the same cycle. The **reorder buffer** (ROB) is used to store instructions that have been issued but not yet committed. This allows out-of-order execution while giving the illusion of in-order execution. A larger ROB size gives the processor a larger window for finding independent instructions.

We expect the performance of the **O3 processor** to increase with **ROB size** and **issue width**. When running the benchmark on all the given configurations, we got the following results:

The first thing we notice is that the IPC of the processor increases with issue width and ROB size. We expected this behavior.

In the runs with a low ROB size of only 2 or 4, we can see that the increase of issue width doesn't provide any gain in performance. We can conclude that the ROB size of less than 2 is extremely restrictive. The processor quickly runs out of space to queue instructions.

When increasing the ROB size from 4 to 8 while keeping the issue width at 1, we can see a substantial gain in IPC. The gain quickly becomes negligible, increasing the ROB size past 8 while leaving the issue width at 1.

We think we can therefore conclude that increasing the issue width without increasing the ROB size doesn't really make a difference at low ROB sizes. After reaching some minimum ROB size, increasing the issue width proves extremely beneficial for the IPC of the processor. The best gain proves to be when ROB = $4 \cdot \text{IssueWidth}$.

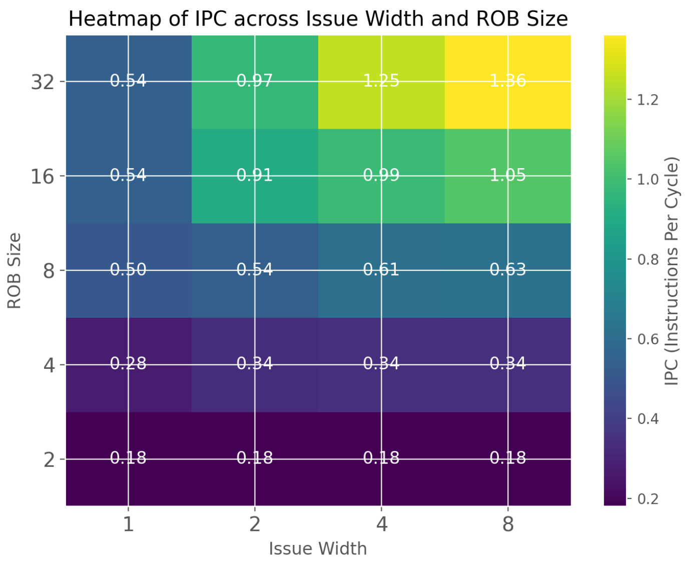


Figure 1: IPC Heatmap Visualization

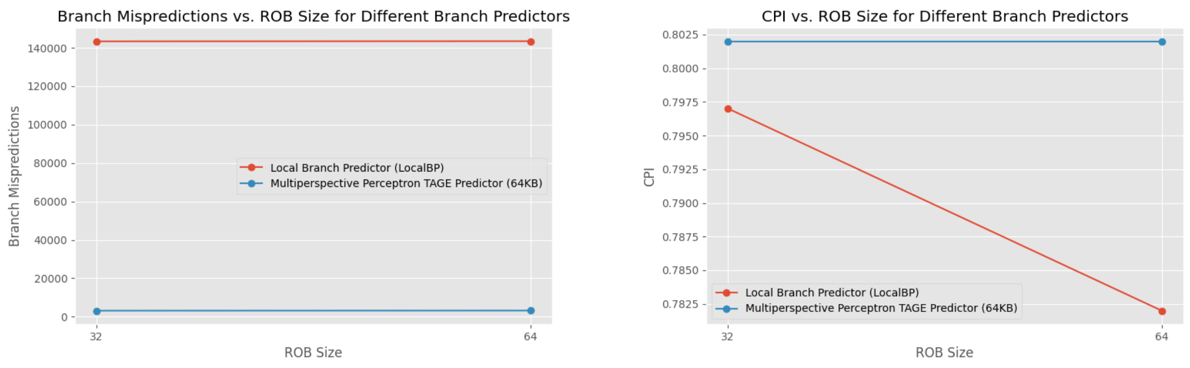
2.1.2 The effect of issue width and ROB size on CPI and branch misprediction

We ran the same *Whetstone* benchmark on the `GEM5` simulator. We tested two different implementations of branch predictors:

- **Local Branch Predictor (LocalBP)**
- **Multiperspective Perceptron TAGE Predictor (64KB) (TAGE)**

Additionally, we tested the predictors using the following two configurations with the goal of testing whether using a different and more complex predictor can increase the performance:

- **Issue Width:** 4, **ROB size:** 32
- **Issue Width:** 4, **ROB size:** 64



(a) (b)

Figure 2: Results of the *Whetstone* benchmark with different branch predictors.

The branch predictor **LocalBP** suffers from a very high branch misprediction rate, whereas the **TAGE** predictor reduces mispredictions by nearly two orders of magnitude. Despite the drastic difference in misprediction counts, CPI remains roughly the same between branch predictor implementations. Doubling the ROB size from 32 to 64 slightly increases the CPI for the LocalBP but yields virtually no change when using the TAGE predictor. The *branch misprediction count* is not affected by the size of the ROB.

The **TAGE** predictor's higher complexity adds a latency overhead on the prediction, causing the overall CPI to stay roughly the same despite the drastic improvement in *branch misprediction count*. Another reason for the overall CPI staying the same could be a compiler optimization of some parts of the benchmark, where the results are never used (e.g., not printed at the end), therefore not being calculated at all. We would have to check the assembly code of the compiled source code to verify.

2.2 Second task: cache

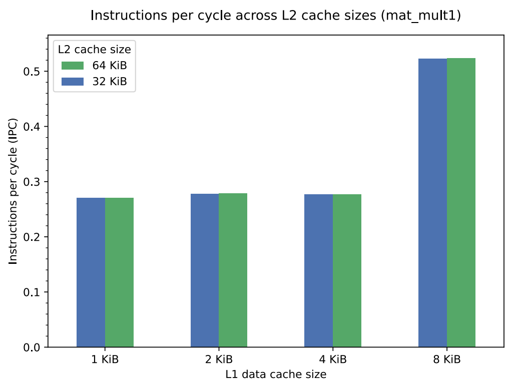
Another benchmark we performed using `gem5` was one of impact of cache sizes and associativities on instructions per cycle (IPC), L1 miss rate, L2 miss rate, and total cycle count. The workloads we used were three similar matrix multiplication programs, named `mat_mult1.c`, `mat_mult2.c`, and `mat_mult3.c`. The workloads differed only in the order in which they performed matrix multiplication: they used the `iijjkk`, `kkiijj`, and `iikkjj` order, respectively. The matrices were of size 256 and the block size used for the calculation was 64.

2.2.1 Impact of L1 and L2 cache size

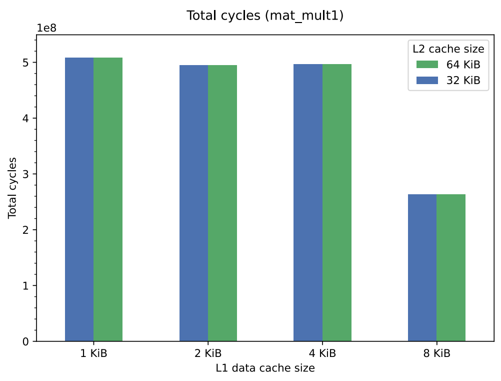
We performed 24 different runs with varying L1 data cache and L2 cache sizes, keeping the set associativity of both caches at 16. The tested L1 data cache sizes were 1 KiB, 2 KiB, 4 KiB, and 8 KiB. The tested L2 cache sizes were 32 KiB and 64 KiB; all combinations of those L1 and L2 sizes were benchmarked.

We observed practically zero meaningful difference between the three matrix multiplication workloads, at least in the values we measured, so in plots where results aren't shown for all three workloads we'll show only `mat_mult1`.

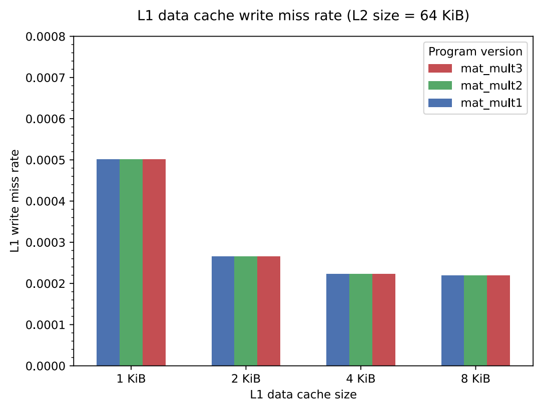
First off, let's have a look at the changes in **instructions per cycle** (figure 3a). As we can see on the plot, we L2 cache size had essentially zero impact on the overall IPC, while increasing the L1 data cache size had some nearly negligible impact using small sizes, but jumped considerably when set to 8 KiB. For context, the size of each of the three matrices used in the workload is 256 KiB, so it is possible that, when using 8 KiB, the reduced miss rate is finally large enough to fit a large chunk of the matrices into L1, and therefore results in significantly less waiting for the cache, in turn raising the IPC value. The L2 cache size doesn't seem to impact the average IPC.



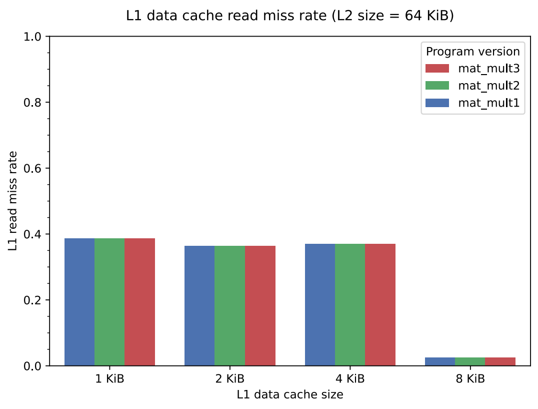
(a) Instructions per cycle across L1 and L2 cache sizes.



(b) Total cycles across L1 and L2 cache sizes.



(c) L1 data cache write miss rate using 64 KiB of L2 cache and varying L1 data cache sizes.



(d) L1 data cache read miss rate using 64 KiB of L2 cache and varying L1 data cache sizes.

Figure 3: IPC, cycle count, and L1 results of our cache benchmark.

Secondly, **total cycles** (figure 3b). The results mirror the improvements seen in the instruction per cycle plot: as the IPC increases, the total cycle count decreases, as we'd expect. As before, the L2 cache size doesn't seem to impact total cycles (nor IPC); it seems the L2 misses are too rare to show up.

Thirdly, **L1 miss rate** (figures 3d and 3c). In the read miss rate plot, a similar jump can be observed as with the IPC value. However, the write miss rate shows a much more gradual decrease over the L1 data cache size from 1 KiB to 8 KiB. Our reasoning is that writes are only done into a single matrix, while reads are done from three (A, B, and the destination matrix). We think this causes the write misses to be more easily optimizable by increasing the L1 size, because the destination matrix likely isn't competing for cache as heavily as the two source matrices.

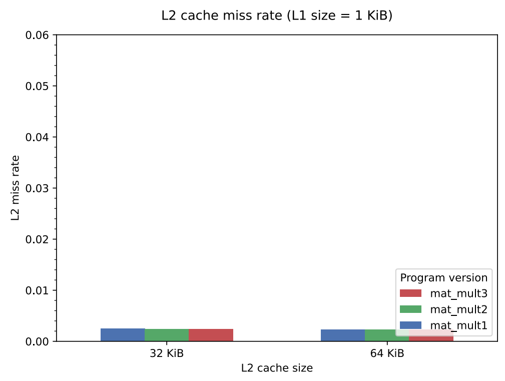
Lastly, **L2 miss rate** (figure 4). Here we see a similar pattern as with the L1 miss rate plot, just in reverse. Interestingly, unless L1 is very large, L2 size doesn't seem to actually impact L2 miss rate. As L1 increases to 8 KiB, we see the same jump we saw in the L1 data cache read miss rate. At that L1 size, we even see the first impact of L2 and matrix multiplication strategy choices: an increased L2 size actually improves the L2 miss rate, and we can notice a small change in performance of different matrix multiplication implementations — the `iijjkk` order seemingly performing the worst — though we don't think the results provide a clear enough distinction to warrant a conclusion. What we should take away from this is more of a confirmation of the jump we saw in the L1 read miss rate graph: as L1 reads start missing less frequently, the L2 cache gets used more and more, warranting more cache misses, and the increased L2 size finally making an impact.

2.2.2 Impact of L1 and L2 associativity

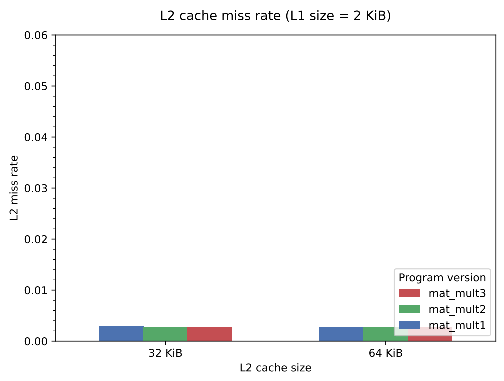
The second cache benchmark we performed was with a fixed L1 data cache of size 4 KiB and a fixed L2 size of 256 KiB, while varying the cache associativities across 1, 2, 4, 8, and 16. As such, we performed 75 runs with different combinations of these parameters and while running all three workloads.

Once again, we observed a minimal, if any, difference between the three matrix multiplication workloads, so once again, we'll show only `mat_mult1` in the plots.

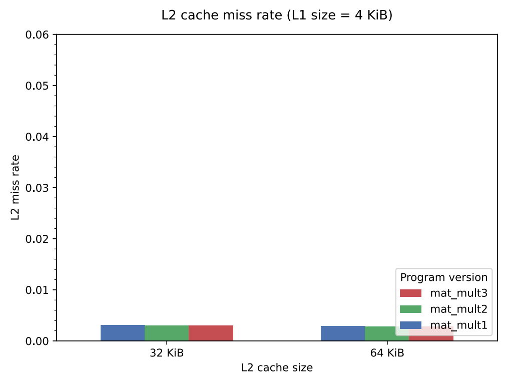
First off, let's look at how **IPC** and **total cycles** are impacted. As we can see in figures 5c and 5a, both IPC and total cycles benefit positively from an increase in both L1 and L2 cache associativity, though the improvements fall off at a certain point; for example, note how — at least for our workload — the optimal IPC was achieved with the combination of 2-way associative L1 cache and 4-way-or-more associative L2 cache.



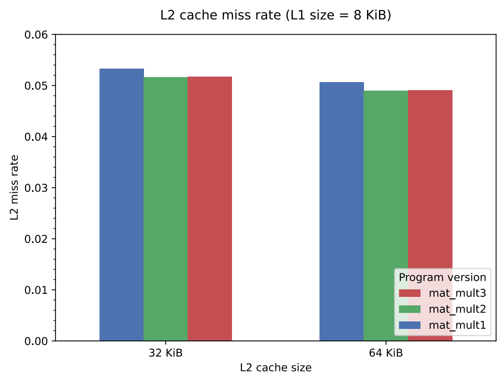
(a) L2 cache miss rate when using an L1 data cache of size 1 KiB



(b) L2 cache miss rate when using an L1 data cache of size 2 KiB



(c) L2 cache miss rate when using an L1 data cache of size 4 KiB



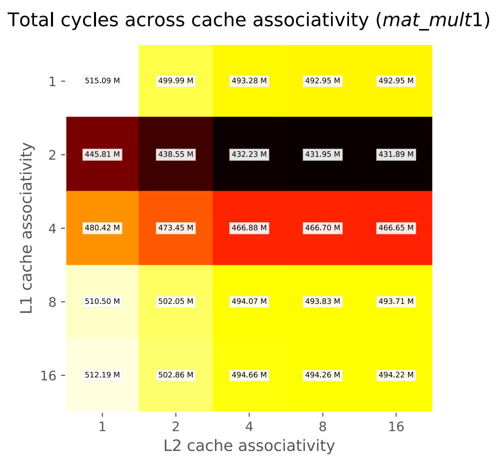
(d) L2 cache miss rate when using an L1 data cache of size 8 KiB

Figure 4: L2 results of our cache benchmark.

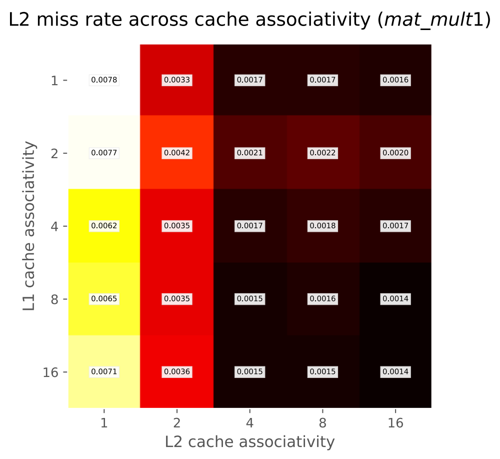
The same is true for the total cycle count. We think a different pattern would emerge under a different — perhaps more allocation-heavy — workload.

Secondly, **L1 read and write miss rates** (figures 6a and 6b) can be seen to be somewhat decreasing as L1 cache associativity is increased. For read misses, this is only true up to 2-way associativity, after which the miss rates rise up to nearly the initial rate again. For write misses, it is evident that higher associativity is better, with no apparent downside.

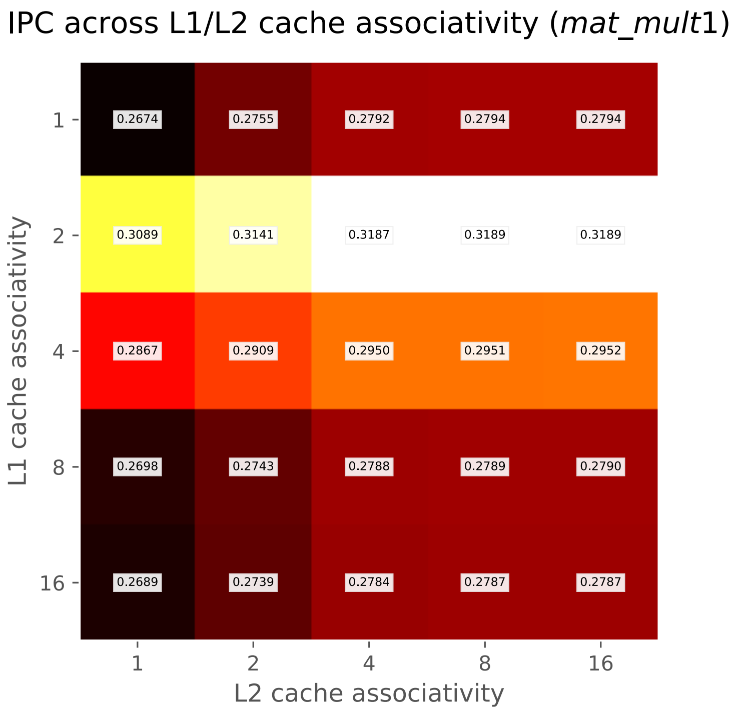
Lastly, **L2 miss rate** (figure 5b) seems to be influenced primarily by an increase in L2 associativity. However, due to L1 being in front of L2 and taking some misses, we can see a very slight improvement as L1 associativity is increased as well and the underlying L1 miss rate falls.



(a) Total cycle count across cache associativity combinations.

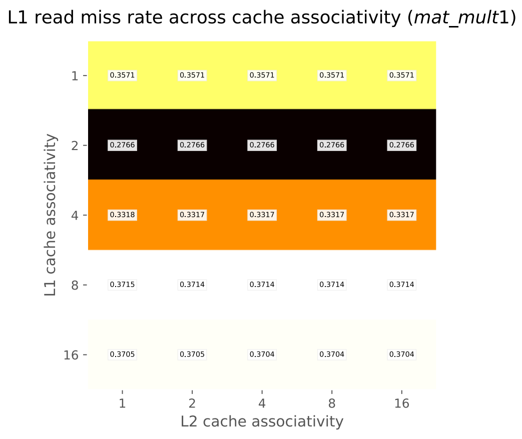


(b) L2 cache miss rate across cache associativity combinations.

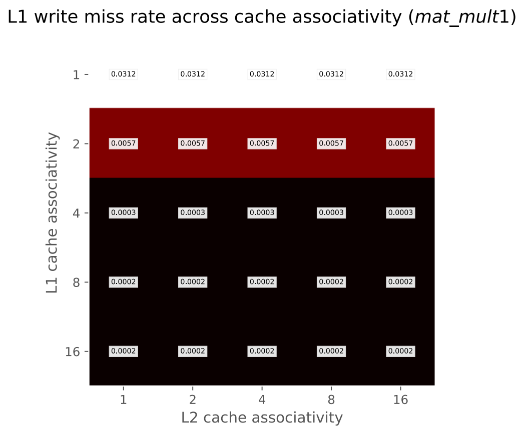


(c) IPC across cache associativity combinations.

Figure 5: Total cycles, L2, and IPC results of our cache associativity benchmark.



(a) L1 read miss rate across cache associativity combinations.



(b) L1 write miss rate across cache associativity combinations.

Figure 6: L1 miss rate results of our cache associativity benchmark.