UCLouvain

École polytechnique de Louvain

LINFO2241 - Architecture and performance of computer systems

Project 2024-2025 : Part 3

Group #23

Authors *:*
GUERRERO Anthony - 1905-2000
KHEIRALLAH Cédric - 0462-1900

Teacher *:*
BARBETTE Tom

Teaching Assistants *:*
EVRARD Colin
INGENZI Vany
TYUNYAYEV Nikita
VANLIEFDE Maxime

2024 - 2025

1 Measurement setup

Measurements were made on a laptop with Ryzen 7 4800HS 8 core with 16GB of RAM running under Linux Mint 22 and also tested on a Lenovo IdeaPad 5 with 16GB of RAM and an AMD Ryzen 5 4600u CPU 6 core running under Linux/Ubuntu 22.04.5 LTS 64-bit.

2 Workload used in experiments and choosing features

For the workload we used WRK with a rate of -1 and used the parameters and features from the given test cases 1, 2 and 3 that were as follows :

- **Test case 1 : Small and big matrices**
  - ✓ **MATSIZE = 64**, NBPATTERNS = 1, PATTERNSIZE = 4
  - ✓ **MATSIZE = 512**, NBPATTERNS = 1, PATTERNSIZE = 4
- **Test case 2 : Small and big patterns**
  - ✓ MATSIZE = 64, NBPATTERNS = 16, **PATTERNSIZE = 32**
  - ✓ MATSIZE = 64, NBPATTERNS = 16, **PATTERNSIZE = 128**
- **Test case 3 : Small and large amount of patterns**
  - ✓ MATSIZE = 64, **NBPATTERNS = 8**, PATTERNSIZE = 32
  - ✓ MATSIZE = 64, **NBPATTERNS = 128**, PATTERNSIZE = 32
- **Test case 4 : Resource utilization**
  - ✓ MATSIZE = 512, NBPATTERNS = 1, PATTERNSIZE = 8, **NB_WORKER = 1**
  - ✓ MATSIZE = 512, NBPATTERNS = 1, PATTERNSIZE = 8, **NB_WORKER = 2**
  - ✓ MATSIZE = 512, NBPATTERNS = 1, PATTERNSIZE = 8, **NB_WORKER = 4**
  - ✓ MATSIZE = 512, NBPATTERNS = 1, PATTERNSIZE = 8, **NB_WORKER = 8**

The first three tests were run for every version of our server (non-optimised, with cache awareness, with loop unrolling and with best optimisations) and repeated three times each to avoid dispersion of the results.
The fourth test was also run for every version of our server but with the additional parameter `NB_WORKER` of NGINX set to every value from 1 to the number of CPU cores on the machine.

3 Optimisations

3.1 Loop unrolling

As seen in the course we used loop unrolling on the inner loop of the two functions where loops seemed to slow down the program the most : `multiply_matrix` and `test_patterns`.
By looking at most of the test cases we decided to assume that the unrolling factor should be of 8 which improved performances significantly.
Later on we tried adding another loop unrolling of factor 4 right after it to catch the few cases where the sizes where not a multiple of 8 and avoid doing a full loop at factor 1 and it (slightly) improved performances again.

3.2 Line-by-line matrix multiplication

The matrix multiplication occurring in the `multiply_matrix` function is by far what slows the program the most and using cache-aware line-by-line matrix multiplication gave us the biggest improvement during this optimisation process.
We simply fetched and kept the value of the `matrix1` element currently being focused on during the calculation in a variable before reaching the innermost loop ; it can now be used multiple times in a row without the need to be re-fetched and and is kept in cache while we browse `matrix2` to calculate the values together.

3.3 Others

3.3.1 *__builtin_expect*

`__builtin_expect` is used in `test_patterns` for the loop after the unrolled loop, it is used since it's almost always positive for the test cases, the difference of performance although positive wasn't significative measure it rigorously.

3.3.2 Avoiding *pcalloc*() initialization of *parsed* structure

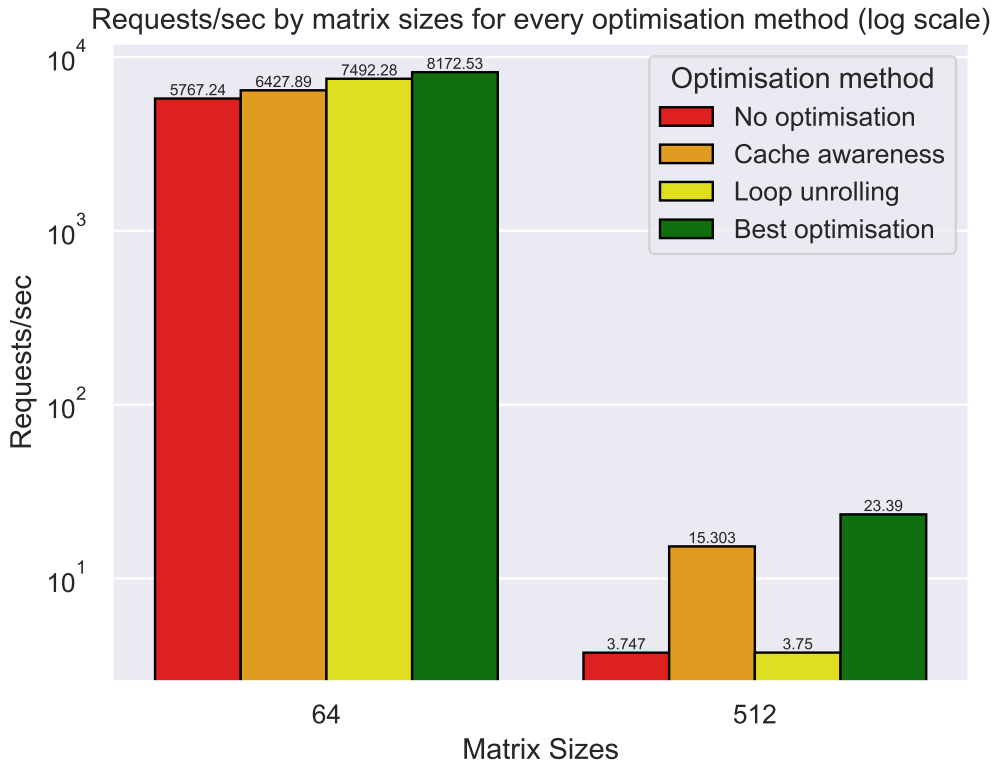
Since the `parsed_request` structure contains multiple different type of elements (3 `uint32_t` variables and 3 arrays of `uint32_t`), using `palloc` instead of `pcalloc` avoids the initialisation of its element which are directly set in the `parse_request` function of our server using the pointer to the start of the structure.

3.3.3 Skipping initialization of result matrix in *multiply_matrix*

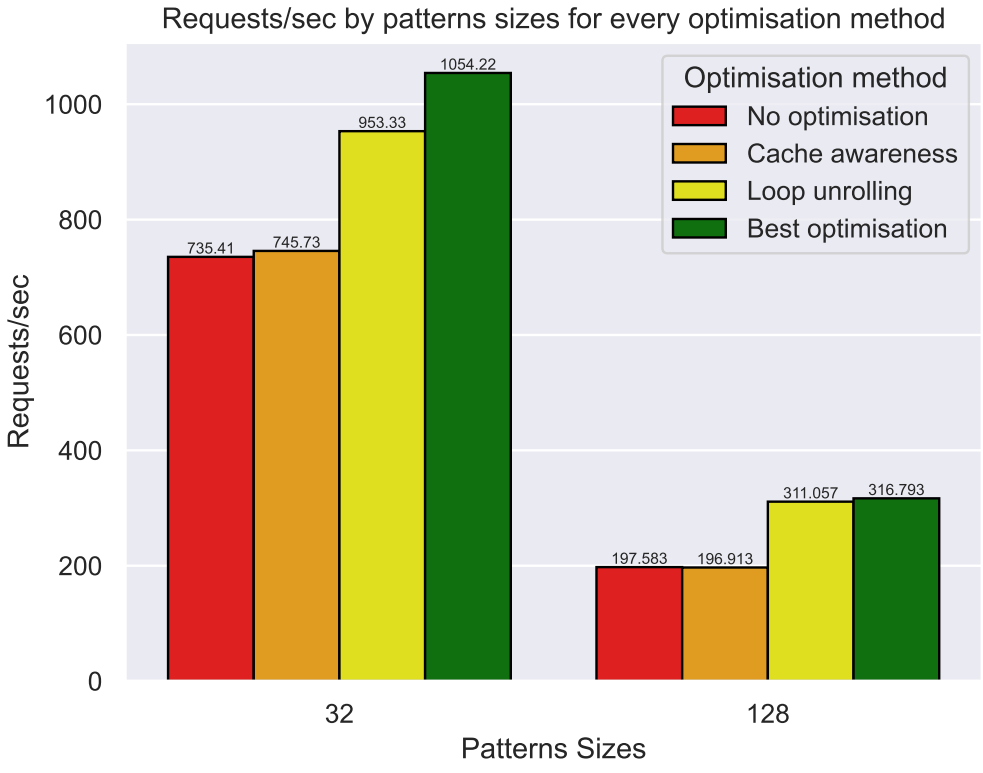
We found out that in our original code we actually initialized the result matrix in `multiply_matrix` twice, once in the main code using `pcalloc` and once in the function `multiply_matrix` so we removed the second one for a tiny optimization.

4 Graph results

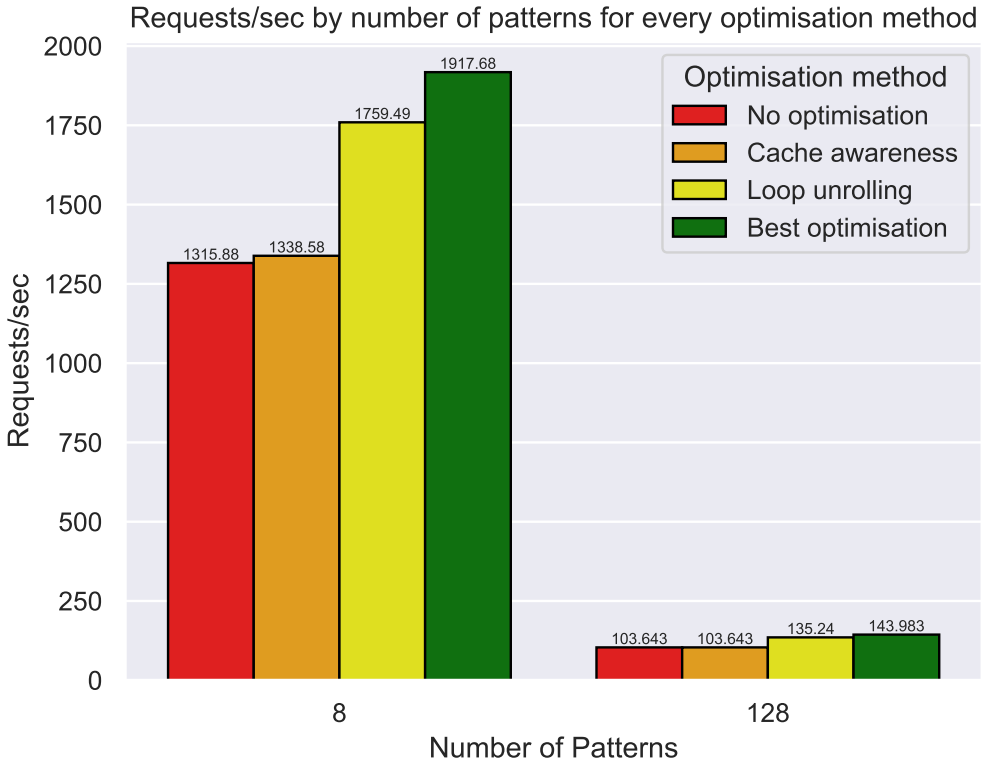
4.1 Results for test case 1



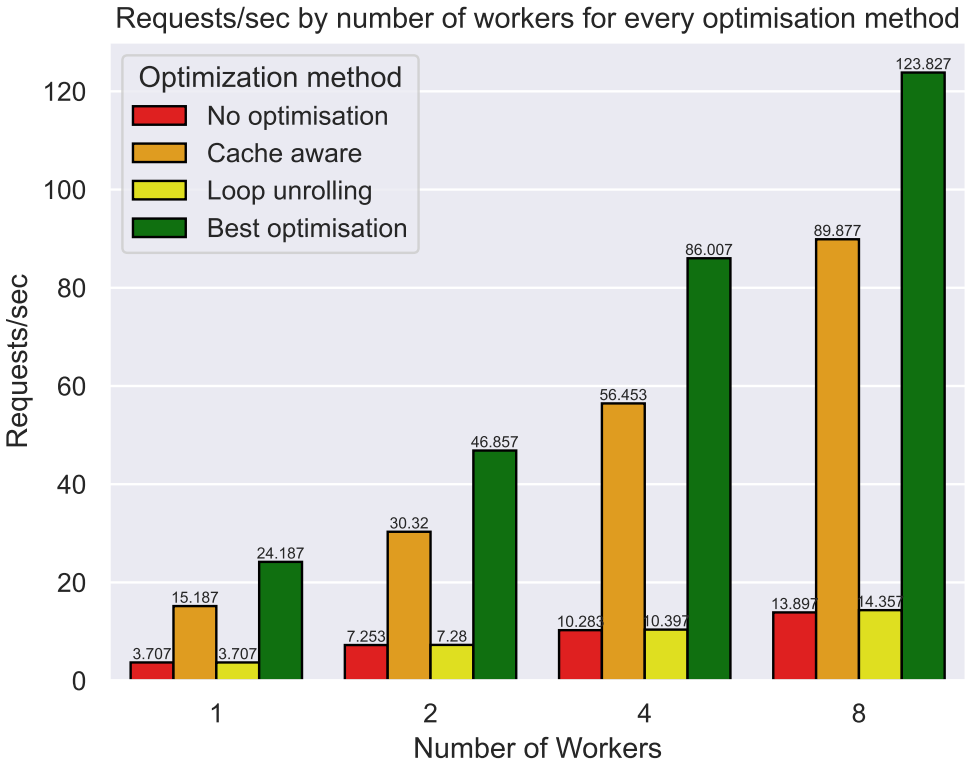
4.2 Results for test case 2



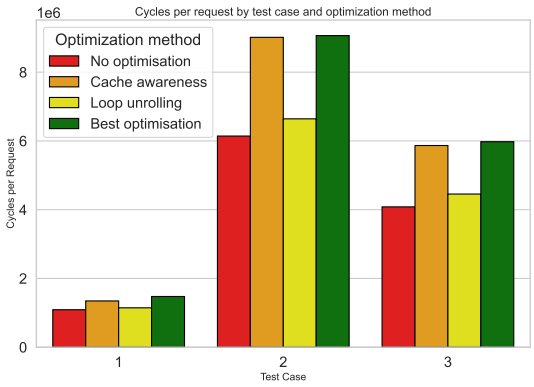
4.3 Results for test case 3

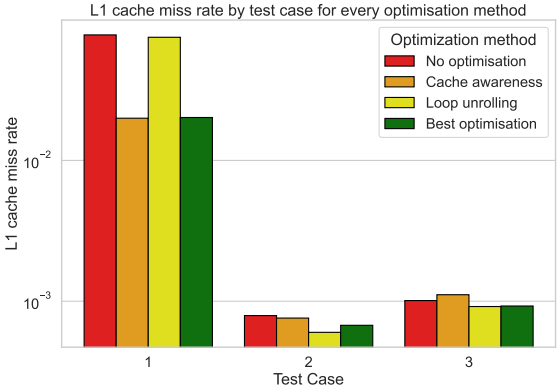


4.4 Results for test case 4

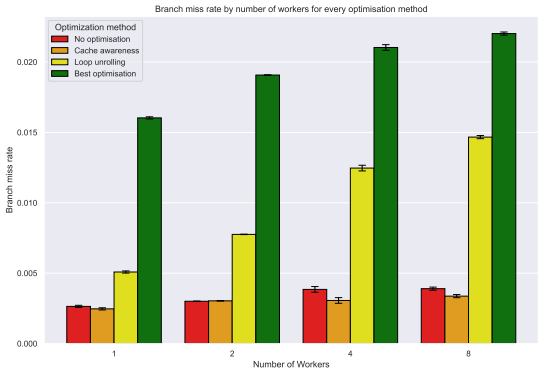


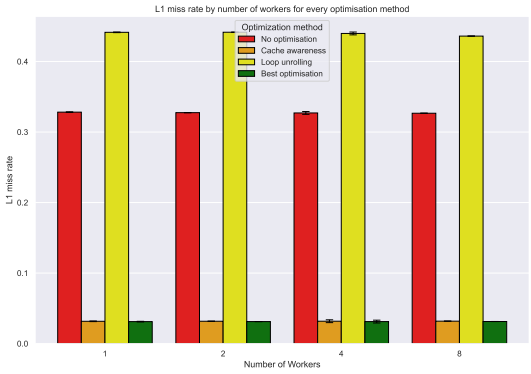
4.5 Results for perf in test case 1, 2, 3





4.6 Results for perf in test case 4





5 Results interpretation

5.1 Test case 1 : Matrix Sizes

This test varied matrix sizes which, as we saw in the second part of this project, has the most impact on performances. Running all our versions of the code with a small matrix size of 64 gave us good performances on all optimisation method with best optimisation being better than loop unrolling being better than cache awareness itself being obviously better than no optimisation whatsoever.
When inputting a much larger matrix size of 512 results got more interesting. Due to the large impact of this parameter on the performances we saw a global diminution of Requests per second with loop unrolling now outputting results as terrible as no optimisation while the best and cache aware versions managed to somewhat keep up.
This is probably due to the fact that the matrix size mostly impact the `multiply_matrix` function where we implemented line-by-line matrix multiplication.

5.2 Test case 2 : Patterns Sizes

This test varied the size of patterns parameter which has much less impact than the others (but still some impact as we'll see here). No matter the pattern sizes best and loop unrolling optimisation are consistently significantly better than using cache awareness or no optimisation. This makes a lot of sense since patterns sizes mostly impact the `test_patterns` and `res_to_string` functions where the main optimisation was loop unrolling thus this optimisation method having the best performances when increasing this parameter.
While loop unrolling increased execution speed it was at the cost of more instructions (??).

5.3 Test case 3 : Number of Patterns

This test varied the number of patterns and has very similar results to the second test of varying patterns sizes where the loop unrolling optimisation is better than the cache awareness optimisation since it's mostly affecting the `test_patterns` and `res_to_string` functions where we implemented loop unrolling optimisation.

5.4 Test case 4 : Number of Workers

We can observe that in this scenario cache awareness has way more impact than loop unrolling due to a matrix size of 512 being used. The increase in performance is linear from 1 to 4 workers since the computer just does the tasks in parallel but for the case with 8 workers the increase isn't as large, this can be attributed to the OS doing other tasks (`wrk` sending requests, the computer's GUI etc...) those background tasks limit the capabilities of the computer to use all of it's cores for this specific task.

5.4.1 L1 cache miss rate

We observed that by far the cache aware and best implementation have way better cache hit rate than the basic and loop unrolling scenario which is what we expected behaviour.

5.4.2 Branch miss rate

We observed that the branch miss rate greatly increases for loop unrolling and best, this can be seen as an issue but we can observe that the highest miss rate is a bit higher than 2% so it

doesn't affect performance enough for it to overshadow the other improvements in the code.

6 Group work evaluation

Concerning group work we made regular meetings and separated work between ourselves depending of our progress in the moment.
We spent the first week on code optimisation and discussing how we could optimise it furthermore. We then started making the scripts for analysing the performances of the different test cases.
We divided the scripts between ourselves one creating the scripts to run the commands, retrieving the results into csv files and plotting the results and the other doing the same for running `perf` on the tests cases.
We ended up with two different ways of doing the same thing but I'd say this made us learn a few tricks from each other despite the few merge conflicts on the git.