CS-350 - Fundamentals of Computing Systems
Homework Assignment #6 - EVAL

Due on November 8, 2023 — Late deadline: November 9, 2023 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

EVAL Problem 1

In this EVAL problem, we will study the actual request lengths that are now unknowns that depend on the interaction between the workload created by the client and the server operating on the hardware at hand.

a) For this part, we are interested in the distribution of request lengths as measured by the server on your machine.

Here, it is immediately important to remember that your results might wildly vary when you compare them with those obtained by your colleagues, because you will be running your code on very different machines. The only good point is that now you can compete with your friends about who has the best machine!

To carry out these experiments, run the following command TWO times. The first time, run:

```
./build/server_img -q 100 2222 & ./build/client -a 30 -I images_small/ -n 1000 2222
```

(NOTE: if your machine is too weak—#noshame—and starts rejecting requests with the parameters above, you can reduce the arrival rate of requests sent by the client, but keep the same total number `-n 1000`).

In the `images_small` only keep the two smallest images, i.e. `test1.bmp` and `test2.bmp`. Collect the output of the server and client, and set them aside.

Next, run:

```
./build/server_img -q 100 2222 & ./build/client -a 30 -I images_all/ -n 1000 2222
```

In the `images_all` folder, keep all the images in the dataset. Once again, Collect the output of the server and client, and set them aside.

Post process the outputs from RUN1 and RUN2 and for each run, produce a plot of the CDF (with average and 99% tail latency, as we did in HW5) of the amount of time taken by EACH image operation, but exclude `IMG_RETRIEVE`—we do not trust the timing reporting of that one! So you should have 6 plots per run, one per operation for a total of 12 plots. Keep them small enough so that you can fit all of them in a single page, maybe 3 plots per row. Here I strongly encourage you not to generate each plot manually, but rather use some form of scripting.

Now look at the plots, and answer the following: (1) within the same run, what operations are similar in behavior and which ones behave differently? (2) Within the same run, which ones are the least predictable operations? (3) Across runs, by how much does the average response time increase for each operation? (3) Across runs, by how much does the 99% tail latency increase for each operation?

Problem 1 continued on next page...

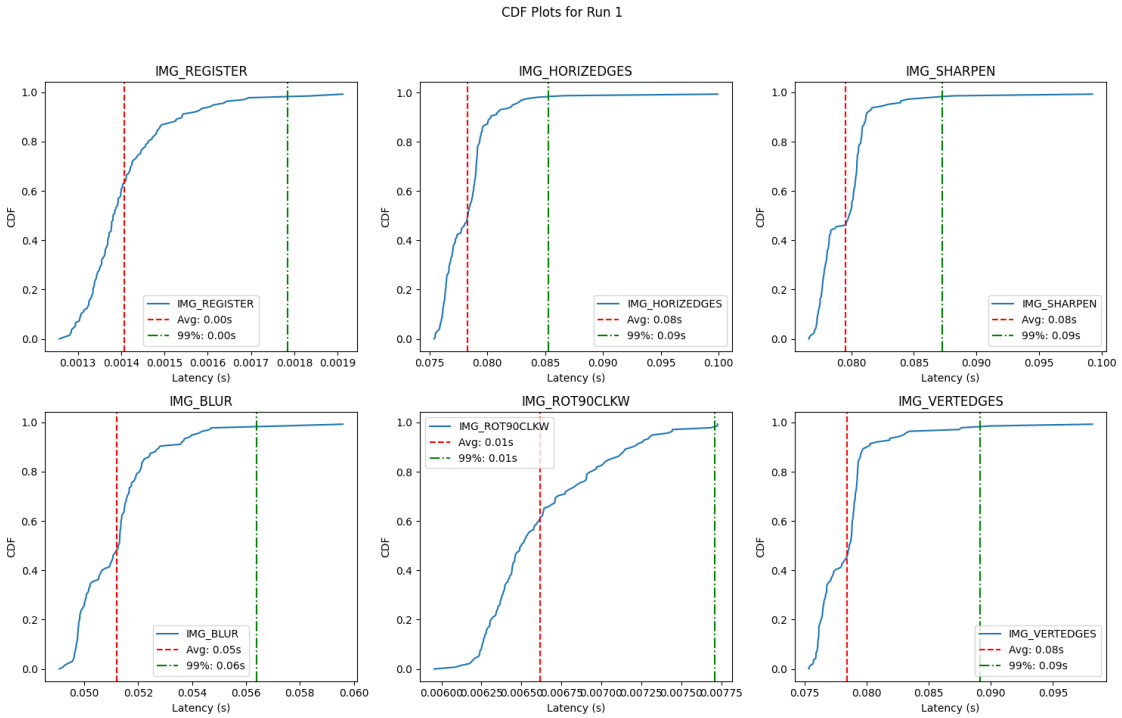

Figure 1: Run 1_cdf_plots

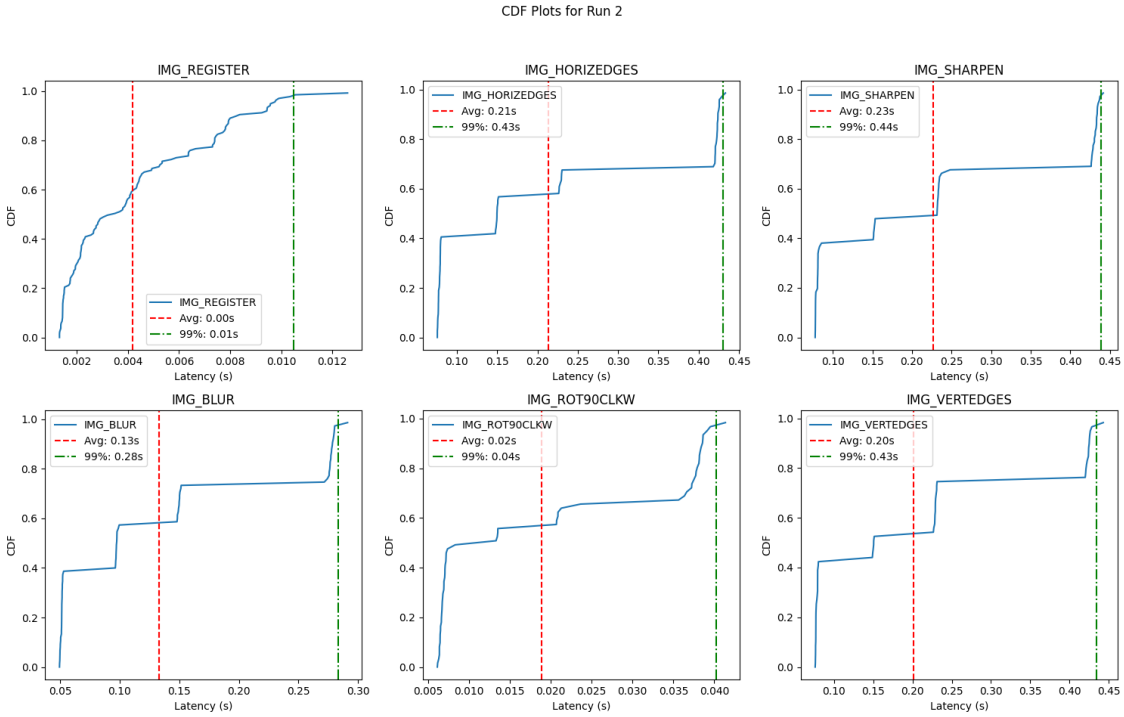

Figure 2: Run 2_cdf_plots

Problem 1 continued on next page. . .

Solution.

(1) within the same run, what operations are similar in behavior and which ones behave differently?

- Operations `IMG_HORIZEDGES`, `IMG_VERTEDGES`, and `IMG_SHARPEN` behave similarly since their latencies are close to each other, which indicates that they may have similar computational complexity and/or resource requirements.
- Operations like `IMG_REGISTER` and `IMG_ROT90CLKW` have significantly lower latencies, suggesting they are less complex or quicker to execute.
- `IMG_BLUR` is in between, indicating it's more complex than image registration or rotation but less so than edge detection or sharpening.

(2) Within the same run, which ones are the least predictable operations?

For Run 1, the standard deviation for all operations is reported as 0.00s, suggesting an extreme level of consistency in their execution times, so every operation is equally predictable.

In the case of Run 2, the standard deviations for the operations `IMG_HORIZEDGES`, `IMG_SHARPEN`, and `IMG_VERTEDGES` are 0.15s, 0.15s, and 0.14s respectively, which are higher in comparison to `IMG_BLUR` and `IMG_ROT90CLKW`, which have standard deviations of 0.09s and 0.01s respectively. Consequently, `IMG_HORIZEDGES`, `IMG_SHARPEN`, and `IMG_VERTEDGES` are identified as the least predictable operations for Run 2 due to the greater variability in their latencies.

(3) Across runs, by how much does the average response time increase for each operation?

- `IMG_REGISTER`: No increase.
- `IMG_ROT90CLKW`: Increase of 0.01s.
- `IMG_BLUR`: Increase of 0.08s.
- `IMG_HORIZEDGES`: Increase of 0.13s.
- `IMG_VERTEDGES`: Increase of 0.12s.
- `IMG_SHARPEN`: Increase of 0.15s.

(4) Across runs, by how much does the 99% tail latency increase for each operation?

- `IMG_REGISTER`: Increase of 0.01s.
- `IMG_ROT90CLKW`: Increase of 0.03s.
- `IMG_BLUR`: Increase of 0.22s.
- `IMG_HORIZEDGES`: Increase of 0.34s.
- `IMG_VERTEDGES`: Increase of 0.34s.
- `IMG_SHARPEN`: Increase of 0.35s.

The operations with the most significant increases in both average and 99% tail latency are the ones that are more computationally intensive (`IMG_SHARPEN`, `IMG_HORIZEDGES`, and `IMG_VERTEDGES`). This could be due to the increased complexity of the operations and perhaps increased resource contention or different characteristics of the input data set in Run 2.

b) For this part, no need to run new commands! Yay! Instead, retrieve the output you generated in the previous part. The one for RUN2 in particular.

As you parse the output produced by your server, construct an online estimator for the length of each type of request. In other words, consider each type of image request (e.g., `IMG_ROT90CLKW` vs. `IMG_HORIZEDGES` etc.) as a separate task. Every time you observe an operation of a given type, consider the measured length as a new job length sample. Build an exponentially weighted moving average estimator (EWMA) with parameter $\alpha = 0.7$ for each request type. Use the estimator to predict the length of the next operation of the same type.

So, say you have observed n operations of type `IMG_ROT90CLKW`, you will construct the estimator $\bar{C}(n)_{\texttt{IMG\_ROT90CLKW}}$ for the $(n+1)^{th}$ operation of the same type.

For each request type, compute the misprediction error as the absolute value of the difference between your prediction and the observed n^{th} job length. Take an average of that error and reason on how good your predictions are for each image operation (i.e., for each task).

Solution. The efficacy of the EWMA predictions for each image operation can be computed by examining the average misprediction error relative to the EWMA estimator value.

$$\text{Error Percentage} = \left(\frac{\text{Average Misprediction Error}}{\text{EWMA Estimator}}\right) \times 100$$

Operation: `IMG_REGISTER`
EWMA Estimator: 0.0031
Average Misprediction Error: 0.0028
Error percentage: 90.32%
This high error percentage suggests that the EWMA predictions for `IMG_REGISTER` are not very reliable.

Operation: `IMG_HORIZEDGES`
EWMA Estimator: 0.0861
Average Misprediction Error: 0.1359
Error percentage: 157.84%
The error exceeds 100%, indicating that predictions are significantly less reliable for this operation.

Operation: `IMG_SHARPEN`
EWMA Estimator: 0.2315
Average Misprediction Error: 0.1517
Error percentage: 65.53%
Although the error is a substantial portion of the estimate, it suggests a moderate level of prediction reliability.

Operation `IMG_BLUR`
EWMA Estimator: 0.2648
Average Misprediction Error: 0.0862
Error percentage: 32.56%
This is the lowest error percentage among the operations, indicating the most reliable predictions.

Operation `IMG_ROT90CLKW`
EWMA Estimator: 0.0111
Average Misprediction Error: 0.0135
Error percentage: 121.62%
Despite the absolute error being small, the error percentage is quite high, leading to less reliable predictions.

Operation: `IMG_VERTEDGES`
EWMA Estimator: 0.0880
Average Misprediction Error: 0.1372
Error percentage: 155.91%
Similar to `IMG_HORIZEDGES`, this high error percentage indicates less reliable predictions.

Therefore, the predictions for `IMG_BLUR` are the most reliable with the lowest relative error, while `IMG_HORIZEDGES` and `IMG_VERTEDGES` show the least reliability. The other operations fall in between, with `IMG_SHARPEN` showing moderate reliability and `IMG_ROT90CLKW` having reliable absolute predictions but less so when considering the relative error.

Problem 1 continued on next page...

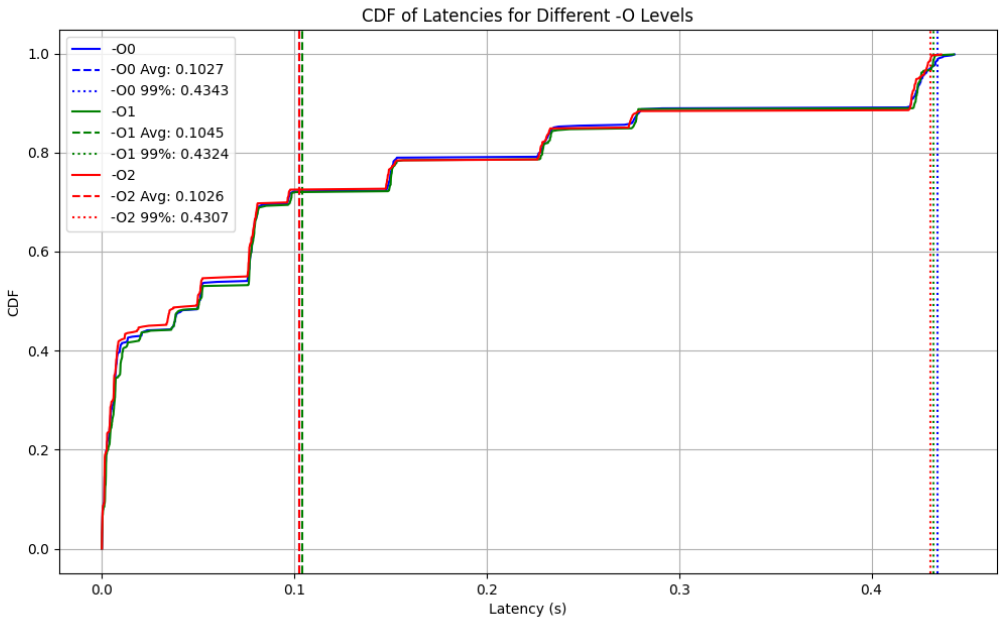

Figure 3: optimization_levels_cdf

c) Okay, here we need to re-run some experiments. First off, set aside the output you produced for RUN2 in part (a). Now, take a look at the Makefile included in the template files and locate the following line: `LDFLAGS = -lm -lpthread -O0`. Focus on the `-O0` (minus capital "o" zero).

First, set that to `-O1`, then run the command `make clean` (careful! this will remove the entire build folder), and then `make` again to recompile the server and all the libraries from scratch. Next, rerun the same command you used for RUN2 (the one with all the images) in part (a).

Now, do everything once again with `-O2`. At this point, you should have 3 outputs in total.

Post-process those outputs to generate 1 plot per output depicting the CDF (with average and 99% tail latency) of the length of any image operation. In other words, do not differentiate by operation type, but treat all the operations from the same output as if they were all just generic image processing operations. Thus, you should just produce 3 plots, one for the `-O0` case, one for the `-O1` case, and one for the `-O2` case.

First, try to figure out what the `-O<value>` flag is supposed to do in theory by looking up (or remembering from CS210) what that is. Next, with plotted data at hand, answer the question of whether or not that flag is behaving as it is supposed to.

Solution. The `-O<value>` flag in a compiler specifies the level of optimization to apply during compilation to improve the performance of the resulting binary.

- `-O0` disables the optimizations option often for debugging purposes, but it may slower the program execution.
- `-O1` enables some optimizations, which improves performance and reduces code size without drastically increasing the complexity of the compilation.
- `-O2` adds to additional optimizations to the ones provided at the `-O1`, which increases the execution speed without increasing the size of the code unreasonably.

Problem 1 continued on next page...

As the plot above suggests, the flag is behaving at it supposed to, as we do see the expected behaviors of `-O<value>` flags:

- The CDF plot for `-O0` is the rightmost curve, and its average and 99 percentile tail latencies is furthest to the right, representing the highest latencies and the least optimized code.
- The CDF plot for `-O1` shifts to the left compared to `-O0`, and its average and 99 percentile tail latencies are between `-O0` and `-O2`, indicating some level of optimization.
- The CDF plot for `-O2` is the furthest to the left, and its average and 99 percentile are also the leftmost lines, representing the most optimized code among the three levels.

d) (NOT GRADED! JUST SOME FOOD FOR THOUGHT. No need to answer.) If, with the current setup, you wanted to design an experiment to understand what is the image processing **capacity** of your machine/server, how would you design that experiment?