Fundamentals of Computing Systems Homework Assignment #6 - EVAL

---

1. EVAL Problem 1

   (a) **Manipulated Image**
   I added one line of code "uint8_t result = saveBMP("Eval_a.bmp", img)" in between send response and send image code. The image that I saved is following:

November 1, 2024

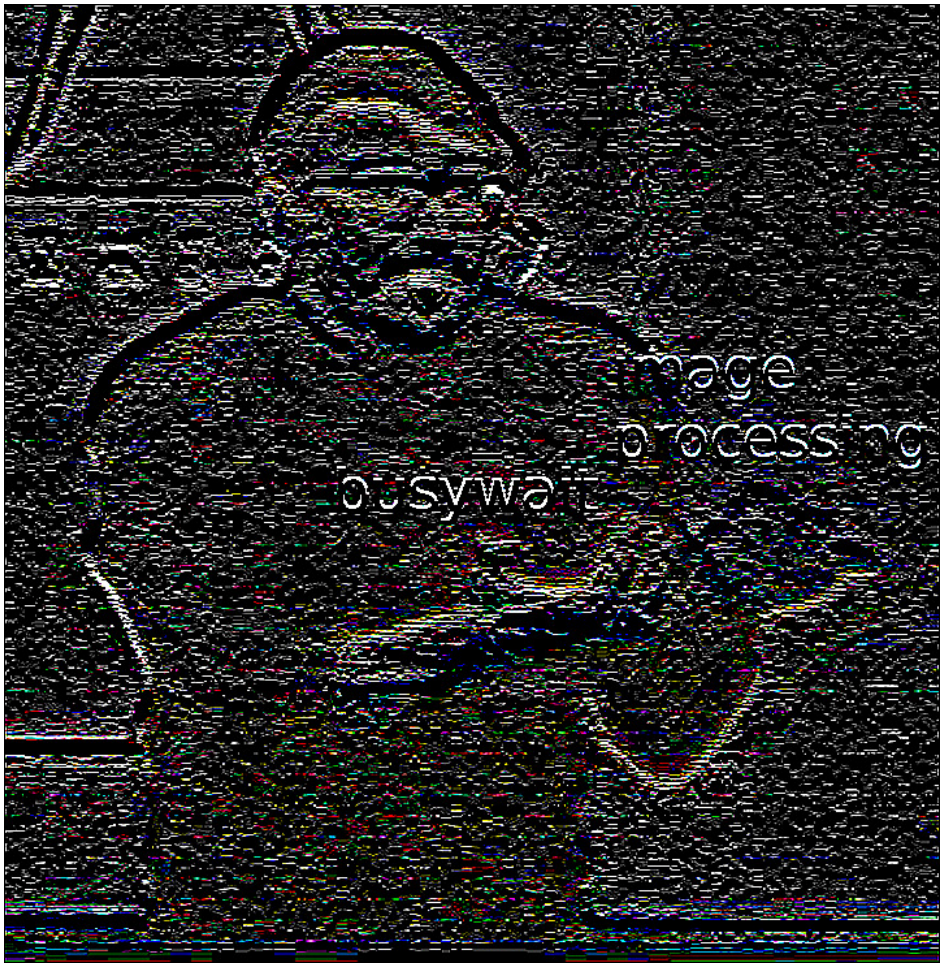

Figure 1: Eval a) BMP file image

Wow, it seems a little scary, but I can see the professor considering whether to use busywait or image processing. Let's examine the client's requests in order to better understand the server's behavior. (There are only 5 requests, so it's not difficult to check the requests.)

The 5 operations of clients are as following:

Listing 1: Client Requests

```
[#CLIENT#] SENT REQ 0 [OPCODE = IMG_REGISTER, OW = 1,
    IMG_ID = 4]
```

[#CLIENT#] SENT REQ 1 [OPCODE = IMG_HORIZEDGES, OW = 1, IMG_ID = 0]
[#CLIENT#] SENT REQ 2 [OPCODE = IMG_SHARPEN, OW = 1, IMG_ID = 0]
[#CLIENT#] SENT REQ 3 [OPCODE = IMG_SHARPEN, OW = 1, IMG_ID = 0]
[#CLIENT#] SENT REQ 4 [OPCODE = IMG_RETRIEVE, OW = 1, IMG_ID = 0]

The first step is for the client to register the original image of the professor. Afterwards, the image is subjected to "IMG_HORIZEDGES" and "IMG_SHARPEN" twice, before being sent back to the client (save as a BMP file). The "IMG_HORIZEDGES" function returns "This function applies the Sobel horizontal operator to each pixel in the image to detect horizontal edges. Edge pixels are not processed to keep the implementation simple."

This function implements the Sobel operator for horizontal edge detection in images. It uses a 3x3 horizontal Sobel kernel '[-1 -2 -1; 0 0 0; 1 2 1]' to detect significant brightness changes in the horizontal direction. The function processes each pixel by examining its 3x3 neighborhood and applying the Sobel convolution operation separately to each RGB channel. For each pixel, it calculates the weighted sum of neighboring pixels using the kernel, then clips the results to ensure they stay within the valid range of 0-255. Edge pixels (those on the image border) are set to black for simplicity. The function creates and returns a new image containing the detected edges while leaving the original image unchanged.

After changing light edges to the dark, the server runs IMG SHARPEN twice. The image sharpen function implements an image sharpening operation using a 3x3 sharpening kernel with values [-1 -1 -1; -1 9 -1; -1 -1 -1]. The kernel is designed to emphasize the center pixel while subtracting weighted values from its neighbors, which creates a sharpening effect by increasing local contrast. The function processes each pixel by convolving it with this kernel, handling the RGB channels separately - it extracts the RGB components from each pixel in the 3x3 neighborhood, applies the kernel weights to each component, clips the resulting values to ensure they stay within the valid range (0-255), and combines the RGB components back into a single pixel value. Edge pixels are left unmodified to maintain simplicity.

After these processes the outcome of the image is as figure 1.

(b) **CDF plots of Images Small and Image All**

The two CDF Plots from two experiments are as following:

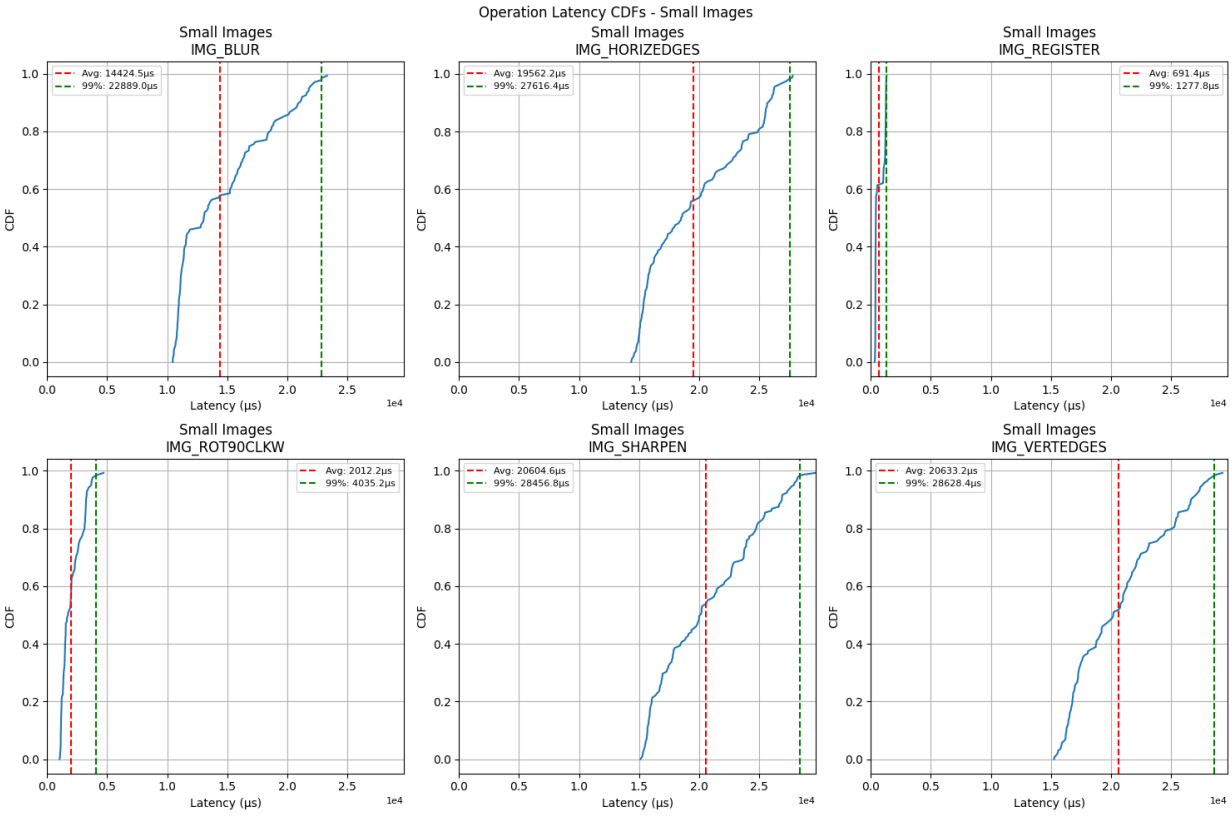

Figure 2: Small Images CDF

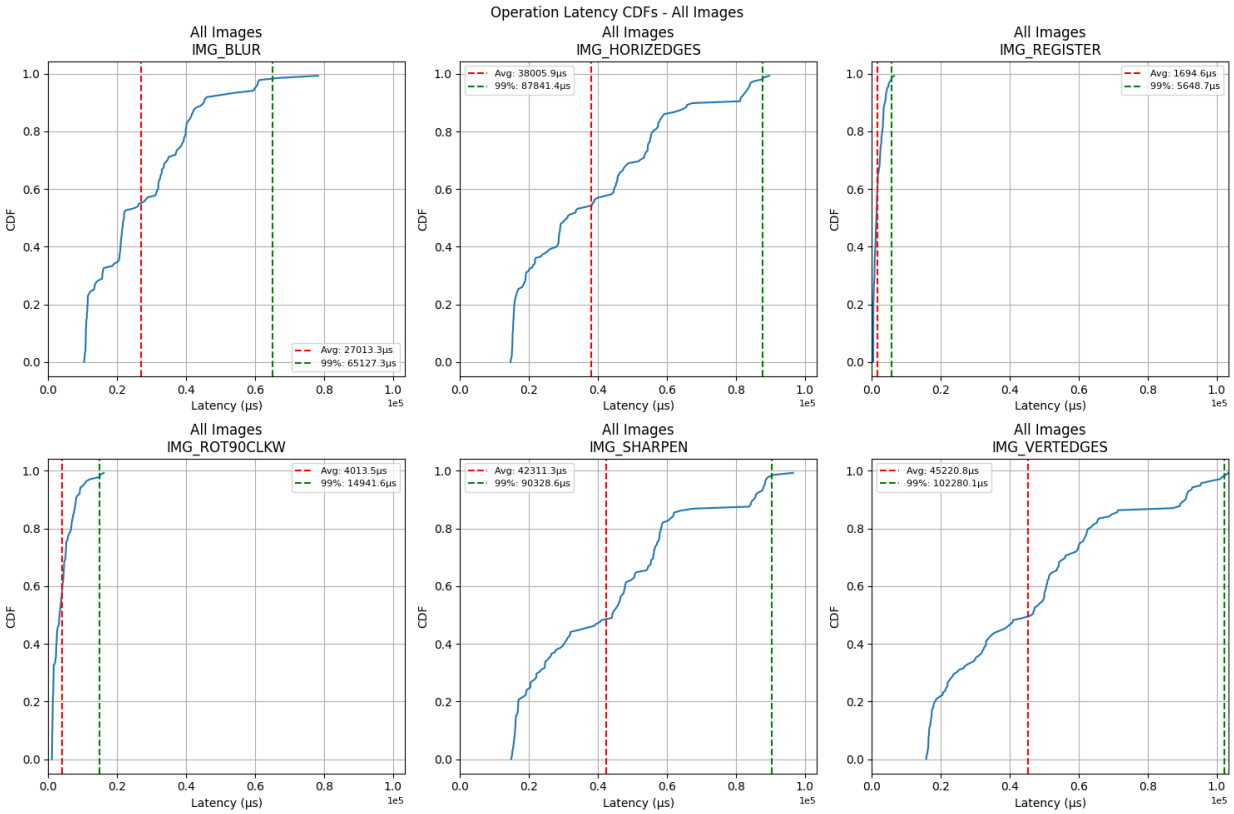

Figure 3: All Images CDF

To make the analysis easier, I recorded the execution times for each operation across different runs. Examining the behavior within the same run (small images) reveals three distinct groups: IMG_HORIZEDGES, IMG_VERTEDGES, and IMG_SHARPEN exhibit similar patterns with average latencies around 19-20ms and BCETs around 14-15ms; IMG_BLUR operates slightly faster with an ACET of 14.4ms and BCET of 10.4ms; and notably different, IMG_REGISTER and IMG_ROT90CLKW demonstrate significantly lower execution times, with IMG_REGISTER averaging 0.69ms and IMG_ROT90CLKW at 2.01ms.

When comparing runs between small and all images, we observe substantial increases in processing times across all operations. For predictability, measured as the difference between WCET and BCET, **IMG_VERTEDGES** is the least predictable operation in the **small images run** with a variation of 13,357.4μs,

while **IMG_SHARPEN** becomes the least predictable in the **all images run** with a difference of 75,512.6μs between its WCET and BCET. IMG_REGISTER maintains the most predictable behavior in both runs, with differences of 975.8μs and 5,296.7μs respectively. The average response time increases vary significantly across operations, with IMG_VERTEDGES showing the largest ACET increase of 119.2% (from 20.6ms to 45.2ms). This stark contrast in predictability between operations can be attributed to their computational nature, with I/O-bound operations like IMG_REGISTER showing more consistent behavior compared to CPU-intensive operations like edge detection and image sharpening.

Table 1: Statistics Comparison

Operation	**Average response time**	**99% tail latency incease time**	**Increase(average, 99% tail)**
IMG_BLUR	14424.5μs $\rightarrow$ 27013.3μs	22889.0μs $\rightarrow$ 65127.3μs	87.3% increase, 184.5% increase
IMG_HORIZEDGES	19562.2μs $\rightarrow$ 38005.9μs	27616.4μs $\rightarrow$ 87841.4μs	94.3% increase, 218.1% increase
IMG_REGISTER	691.4μs $\rightarrow$ 1694.6μs	1277.8μs $\rightarrow$ 5648.7μs	145.1% increase, 342.0% increase
IMG_ROT90CLKW	2012.2μs $\rightarrow$ 4013.5μs	4035.2μs $\rightarrow$ 14941.6μs	99.5% increase, 270.3% increase
IMG_SHARPEN	20604.6μs $\rightarrow$ 42311.3μs	28456.8μs $\rightarrow$ 90328.6μs	105.3% increase, 217.4% increase
IMG_VERTEDGES	20633.2μs $\rightarrow$ 45220.8μs	28628.4μs $\rightarrow$ 102280.1μs	119.2% increase, 257.3% increase

(c) **Estimator** $\hat{C}(n)$

Function	**Average Prediction Error (ms)**	**Number of Samples**	**Final EWMA Estimate (ms)**	**Relative Prediction Error (%)**
IMG_REGISTER	1.29ms	137	2.19ms	75.9%
IMG_HORIZEDGES	22.62ms	158	37.10ms	59.5%
IMG_SHARPEN	22.68ms	145	21.82ms	53.6%
IMG_BLUR	14.29ms	135	33.67ms	52.9%
IMG_ROT90CLKW	2.64ms	138	2.76ms	65.9%
IMG_VERTEDGES	25.36ms	139	85.86ms	56.1%

Table 2: Prediction Error Analysis for Image Processing Functions

In order to predict accuracy across different image operations, EVAL said to use 0.7 as an alpha value for the EWMA predictor. IMG_REGISTER demonstrates the lowest absolute prediction error of 1.29ms, though its relative error (75.9%) is high due to its short processing time. In contrast, more complex operations like IMG_VERTEDGES, IMG_SHARPEN, and IMG_HORIZEDGES show larger absolute errors (22-25ms) but significantly better prediction accuracy (53-60%). IMG_ROT90CLKW shows moderate absolute error at 2.64ms with a relative error of 65.9%. IMG_BLUR demonstrates intermediate predictability with an average prediction error of 14.29ms and a relative error of 52.9%. Overall, while the absolute prediction errors are higher for complex image operations, their relative

errors are generally lower than simpler operations like IMG_REGISTER, indicating that the EWMA predictor ($\alpha = 0.7$) performs more consistently with complex operations that have more stable behavior patterns. The final EWMA estimates show significant variation, with IMG_VERTEDGES having the highest estimate at 85.86ms, while IMG_REGISTER and IMG_ROT90CLKW maintain lower estimates around 2-3ms.

(d) **LDFLAGS -O Flags**

The 3 CDF plots based on the -O flags in the Makefile is as follows:

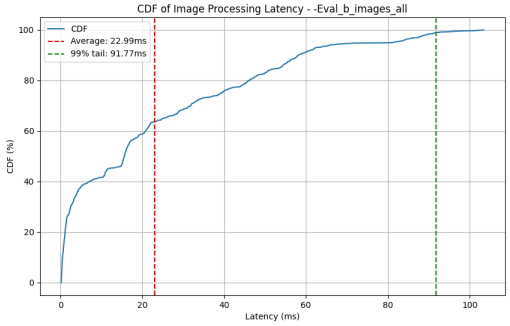

Figure 4: -O0

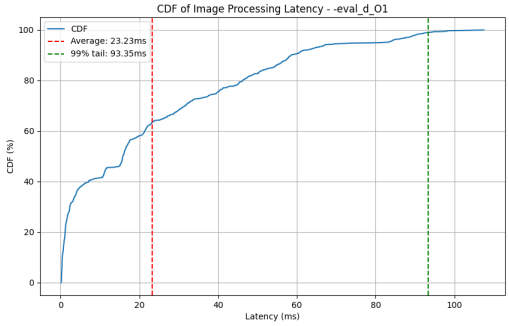

Figure 5: -O1

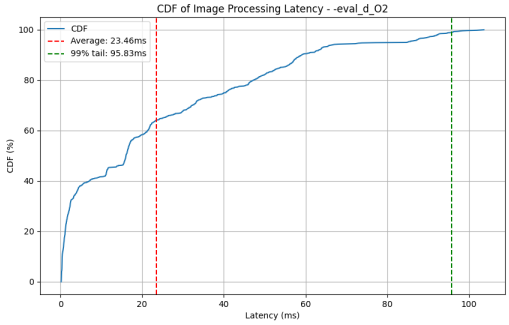

Figure 6: -O2

Figure 7: -O flag variation

The -O flag in compiler settings controls optimization levels applied during code compilation. It ranges from -O0 (no optimization) to higher levels like -O1 and

-O2, which enable increasingly aggressive optimizations. These optimizations focus on improving execution speed, reducing code size, and making better use of CPU resources. They do this by inlining functions, unrolling loops, and simplifying instructions. Typically, as the optimization level increases (e.g., moving from -O0 to -O1 and then to -O2), one would expect a reduction in average latency and improved performance.

However, the attached plots show that the CDF of image processing latency for -O0, -O1, and -O2 remains similar, with only slight differences in the average latency and 99% tail values. This indicates that the optimizations provided by the higher -O levels did not significantly affect the overall latency of operations. The expected performance improvements from increased optimization levels are not realized. This could suggest that the nature of the image processing operations or the specific codebase is not highly sensitive to the types of optimizations applied. Therefore, the -O flag in this scenario is not behaving as typically expected, as there is no substantial improvement in runtime performance at higher optimization levels.