Handwriting Detection on the go!

Pingali V Kalyan

Final Project Report

ECEN 5623 Real time Embedded Systems

University of Colorado, Boulder

August 7th 2016

Contents

Introduction:

My project proposal is for the design of a real-time system that detects handwriting on the go. Unlike the traditional handwriting detection systems which process an image of already written characters, my system would capture a live video (typically someone writing on a board, ex: a classroom setting) and detect what is being written.

This system works similar to the Google Handwriting input [1], only this does not need to be written on your tablet or smartphone. All you need is a camera that points in the direction of the surface you want to write on! (and the system to process all the information)

The scope of the project includes:

- Students who find it difficult to view the board in class could benefit through this system by directly viewing the text on their personal computer (obviously, no bounds on how large you would want the text on your system). The advantage of this system would be that a text feed is compact as opposed to a raw video feed (that cannot be compressed due to the need to maintain pace with the lecture) which would eat up a huge amount of bandwidth.
- A really good alternative for the professors who like the blackboard approach to writing with a stylus on a screen.

Recap of request for proposal:

Note: Important revisions include stating deadlines (outlined in **bold**; with previous invalid assertions or obsolete information ~~crossed out~~)

Handwriting detection on the go! MINIMUM REQUIREMENTS:

1. Resolution should be 640x480 (flexible, but limited to only 4:3 pixel ratio)
2. Video should be captured live through a webcam (or a camera that can be interfaced through a USB, compatible with Linux).
3. Individual frames are to be acquired from the live video at a fixed **frequency (1 frame every 3 seconds)**. Frequency selected would also impose a constraint on the writing speed to ascertain real-time detection. This is identified as a deadline. The deadline would be that the frame acquired should be within a delta of the time at which the frame should be captured. For example, if the system requires frames to be captured every 2 seconds, the frames that is acquired should be timestamped within 2.1 seconds, 4.1 seconds, etc from the start of the process (here the delta is 0.1 seconds).
4. Comparison between Live stream and detected characters should exist. There should be at least two separate windows showing what is being captured (live stream) and what each character is being detected (on the console) as in real-time. This also implies that, the handwriting detection of the previous frame should happen before the next frame is acquired.

5. Timing for processing of each acquired frame **(the frame_grabber service)** should be printed to the console. ~~This is identified as a deadline~~. **The request period is 1 per 3 seconds.** The processing time should be limited (hard real-time) **to 300 milliseconds (deadline identified)**. There should be no deadlines missed in this. The reason for hard real-time imposed is that, once a frame misses a deadline, the next frame would need to process twice the amount of information that one frame would have to. This would ensure that all following processings would also miss the deadline too, every single time **(or maybe not after analysis)**!

6. All capitalised english alphabets should be detected with considerable accuracy.

Handwriting detection on the go! VERIFICATION MINIMUM REQUIREMENTS:

1. Time stamps should exhibit that every deadline as mentioned in the MINIMUM REQUIREMENTS is met with high accuracy (in case of soft real-time and every deadline in case of hard real-time).

2. A video of live stream along with the detection should be compiled to verify that it actually happens on the go! and is real-time without missing the deadline (next frame capture, as specified in point 1 in the MINIMUM REQUIREMENTS).

3. The video, algorithm, any other tools or code used in the completion of said system should be included in the submission.

Handwriting detection on the go! TARGET REQUIREMENT:

1. Make the output more realistic by displaying the detected text on the acquired frames.

Handwriting detection on the go! STRETCH REQUIREMENT:

1. Create a thread only to print at the end of the frame_grabber service. This would eliminate any delays caused due to printf statements!

Handwriting detection on the go! FUTURE SCOPE:

1. Using conventional ways to implement handwriting detection from images at the end of a session and match the above system's output for better accuracy of detection. This could help in maintaining notes of a lecture with high accuracy.

2. Adding digits and spaces for making the system complete for most classes.

References:

(The application is inspired by my interest in teaching on a blackboard and by the Google handwriting app)

[1] http://www.digitaltrends.com/mobile/google-handwriting-android-app/

[2] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_ml/py_knn/py_knn_opencv/py_knn_opencv.html#knn-opencv

[3] https://arxiv.org/ftp/arxiv/papers/1004/1004.3257.pdf

[4] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.259.3553&rep=rep1&type=pdf

Approach to given problem:

3 services have been defined:

- live_video : captures live video feed at 5 frames per second
- frame_grabber : initialized every 15 frames (or 3 seconds), which performs image processing and calls for the kNN comparison (with 5 nearest neighbours)
- print_it : prints out all processed information at once (after frame_grabber is executed)

The main() function runs a while loop which calls the live_video() service, through a binary semaphore.

The live_video() after execution, clocks the time for execution and sleep upto 200 milliseconds including implementation. The nanosleep upto 200 ms ensures that video is streamed at 5 frames per second. Just before the end of the service, a binary semaphore is unlocked to initiate execution of the frame_grabber() service.

The frame_grabber() service completes its execution and in a similar way as the above threads, calls the print_it() service using a binary semaphore.

The flow diagram of this basic structure is outlined below.

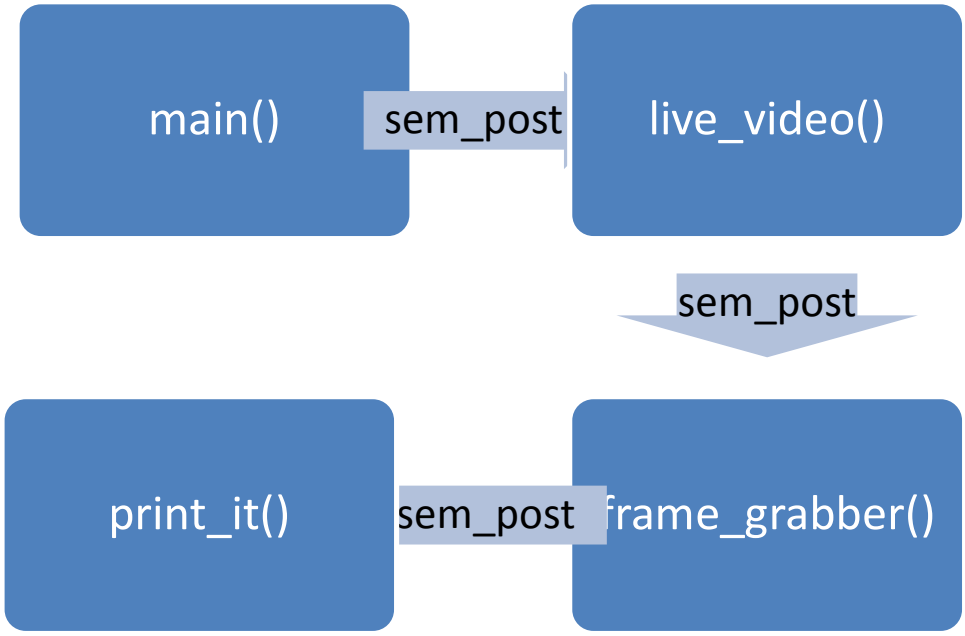


Figure1 : Basic block diagram of process flow

All the services run with a while loop in place, to ensure continuous running of every process. The only time it pauses is, when it waits for a semaphore to be posted!

Concepts overview:

Semaphore:

A very easy way of understanding a semaphore is through this enjoyable excerpt which talks about toilet keys analogous to semaphores:

Is the number of free identical toilet keys. Example, say we have four toilets with identical locks and keys. The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0. Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue.

– *Source : http://niclasw.mbnet.fi/MutexSemaphore.html*

A semaphore which has only one toilet key is a binary semaphore! So, only one person can be using it at once.

Wikipedia says: Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

– *Source : https://en.wikipedia.org/wiki/Semaphore_(programming)*

k – Nearest Neighbours (Machine Learning algorithm):

This is one of the simplest algorithms available for classification of data. It stresses on finding the closest match from a feature space depending on the distances of each point from the test data.

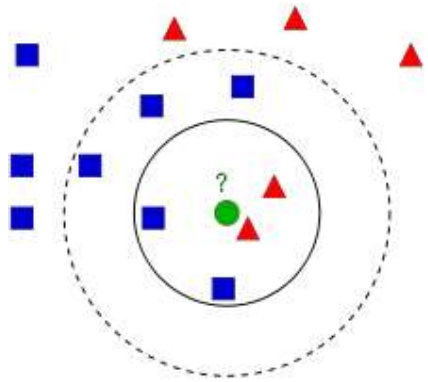


Figure 2: Classification using kNN (Source: http://docs.opencv.org/3.0-beta/_images/knn_theory.png)

From the opencv documentation, "In the image, there are two families, Blue Squares and Red Triangles. We call each family as Class. Their houses are shown in their town map which we call feature space. (You can consider a feature space as a space where all datas are projected. For example, consider a 2D coordinate space. Each data has two features, x and y coordinates. You can represent this data in your 2D coordinate space, right? Now imagine if there are three features, you need 3D space. Now consider N features, where you need N-dimensional space, right? This N-dimensional space is its feature space. In our image, you can consider it as a 2D case with two features)."

Source: http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_ml/py_knn/py_knn_understanding/py_knn_understanding.html

Now when a new member (green circle) moves into this town, according to 2NN should be classified as red (triangle); but should be classified as blue (square) according to 7NN (or 5NN or 6NN). Generally an odd number is chosen for k to avoid conflicts when two different classes have equal number of nearest neighbours to a test data.

In my implementation, I use the kNN algorithm with k = 5.

As for any machine learning algorithm, certain amount of train data is required. The train data was obtained from http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/.

The capitalised alphabet set had to be pre-processed to obtain a decent execution time for the algorithm. To do this, a code "thins.cpp" was coded. The code initially included thinning, which later was deemed unnecessary as it only worsened the accuracy (for the train data used).

Fast thinning algorithm by Zhan-Suen:

Initially, I had planned to incorporate a thinning algorithm to enhance the accuracy in detection.

Having implemented and executed the thinning algorithm, I found that the thinning wasn't exactly giving desired results.

Hence, I completely ignored this and went on to implement an algorithm that does not have any thinning involved.

Inspired from "A Fast Parallel Algorithm for Thinning Digital Patterns" by Zhan and Suen, a github user *(Source : https://github.com/bsdnoobz/zhang-suen-thinning)* coded the algorithm on c++. The code has been incorporated as is into my code and has been referenced as a comment next to the function.

From the abstract of the paper, "It consists of two subiterations: one aimed at deleting the south-east boundary points and the north-west corner points while the other one is aimed at deleting the north-west boundary points and the south-east corner points. End points and pixel connectivity are preserved. Each pattern is thinned down to a "skeleton" of unitary thickness."

The above paragraph gives a gist of the algorithm.

Software Implementation:

thinss.cpp: (Code in appendix directory)

The main function does a system calls to output all filenames from the sample data (train data). Then an Output directory is created to store the processed images.

Next, each (image) file is opened by read "files.txt", converted to grayscale and passed through binary thresholding. The thresholded image is tested for the largest contour that is smaller than the frame size itself (to avoid edges at the frame, due to reflections being detected as a contour).

Later, each is stored as a 40x40 image (1 channel, binary image).

ocr.cpp: (Code in appendix directory)

The code consists of a main() function, the 3 services as described in approach to given problem, a knn_init() function, delta_t(), print_scheduler(), thinning() and thinningIteration().

Out of these the thinning() and thinningIteration() functions are not used (read this).

main():

- Starts with defining a cpuset with only a single CPU (CPU2). This is later used to set affinity to all threads, i.e., all threads are tied to CPU2.
- Scheduling policy is read and updated to FIFO (irrespective of the initial scheduling policy).
- Semaphores, threads, thread attributes and parameters are initialised.
- kNN is initialised.
- Threads are created.
- while(1) is run posting a semaphore to execute the live_video service each time it runs. To ensure a semaphore is posted only after the execution of the live_video service, a semaphore indicating completion is implemented (in sem_live_video_ack).

- threads are joined to ensure all threads complete execution before exiting.

knn_init():

- opens and prints out all filenames from the Output directory as created in the pre-processing using "thins.cpp"
- Every file is opened, reshaped to a row, converted to gray and stored in a training matrix (train_data)
- The identifier to each row in the train_data is stored in the mtrain_class as ascii values of alphabets.
- knn.train() is called to train the data using the train_data and mtrain_class matrices.
- A test image (containing the letter "I") is tested against the train data to ensure that the training was successful.

live_video():

- The thread start function of the service live_video.
- Timing is synchronized to ensure a 200 ms run time of the entire thread.
- This is done by implementing dynamic sleep time (subtracting the execution time from 200 ms and sleeping for the remainder of the time).
- This is done to ensure that one frame is sent to the frame_grabber every 3 seconds (if statement to keep a track of count to 15 frames; 15 frames * 200 ms = 3 seconds).
- The thread mainly streams the video to a Live display.

frame_grabber():

- Processes the frame passed by the live_video service in the same way that the train_data is compiled in the knn_init() function.
- The processed frame is then fed to the kNN algorithm to obtain the output.

print_it():

- Majorly responsible for printing out all the timings (including itself).

delta_t():

- Calculates the difference in times from the first two arguments and stores the result in the third.
- All arguments must be passed as pointers.

print_scheduler():

- Prints out the current scheduling policy of the process that calls the function (current => when the function was called).

thinning() and thinningIteration():

- Implements the Zhan and Suen algorithm.
- Sourced from https://github.com/bsdnoobz/zhang-suen-thinning

Hardware Implementation:

The NVIDIA Jetson TK1 was used as the target system. My HP Laptop acted as the host. The host was connected to the target via Wi-Fi and Ethernet, using WinSCP (for sftp file transfer), putty (for ssh command line) and TightVNC Java Viewer (for interactive GUI).

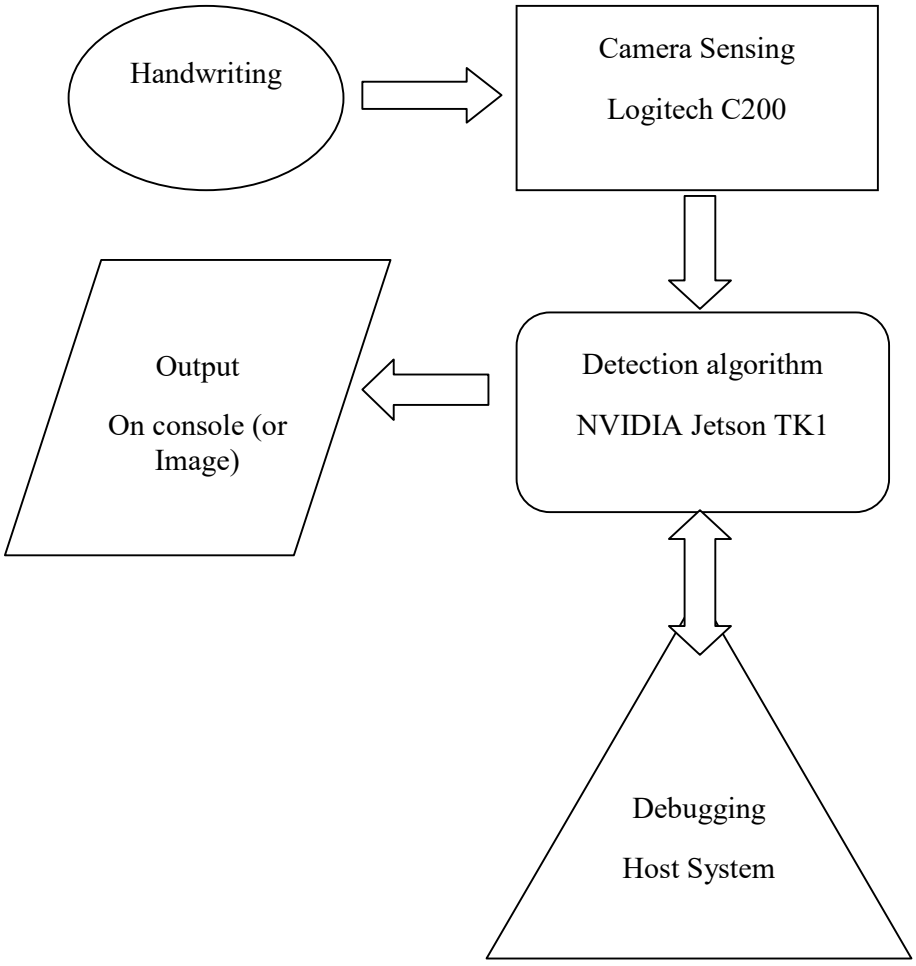


Figure 3: Block Diagram of the Hardware process flow

Analysis:

Minimum Requirements:

live_video():

Average execution time (C_{LV}): 40 ms

WCET (W_{LV}): 80 ms

Request Period (T_{LV}): 200 ms

frame_grabber():

Average execution time (C_{FG}): 55 ms

WCET (W_{CG}): 165 ms

Request Period (T_{FG}): 3000 ms

print_it():

Average execution time (C_{PI}): 1 ms

WCET (W_{PI}): 1 ms

Request Period (T_{PI}): 3000 ms

Cheddar Analysis:

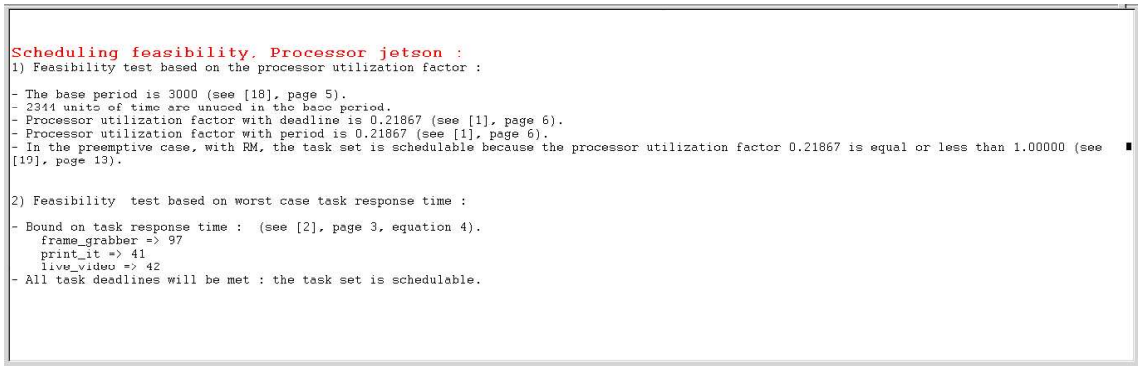


Figure 4: Scheduling feasibility as per Cheddar

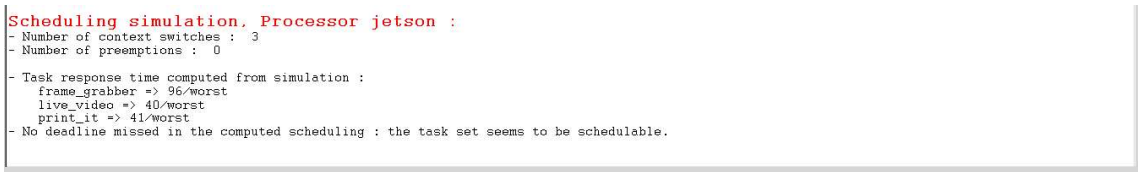


Figure 5: Scheduling simulation of service set

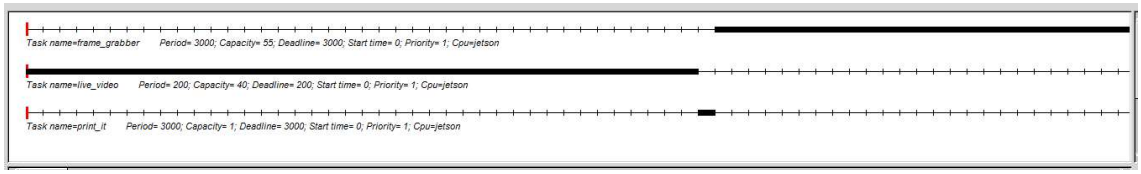


Figure 6: RM scheduling of the service set

Although Cheddar (*including jitter for calculations*) shows print_it to run before frame_grabber, it would not make any difference as such. This is because the WCET as execution times, is still schedulable as shown in the figure below.

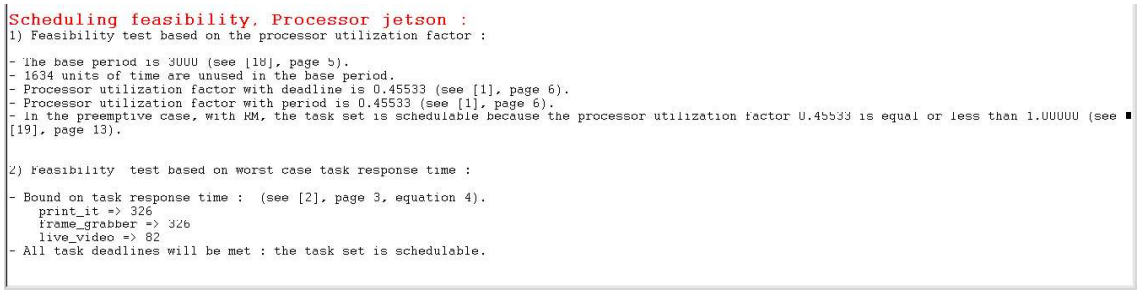


Figure 7: Scheduling feasibility of service set (C = WCET)

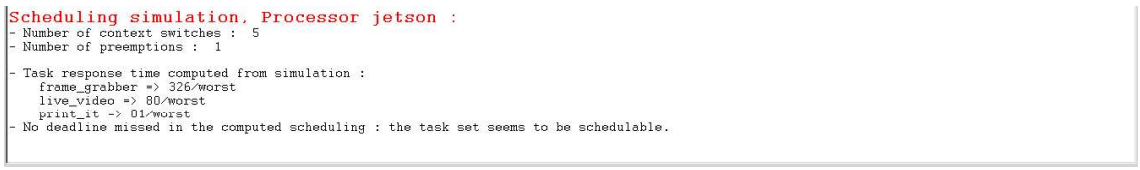


Figure 8: Scheduling simulation of service set (C = WCET)

CPU utilization in case of average execution times used as completion time = 21.87%

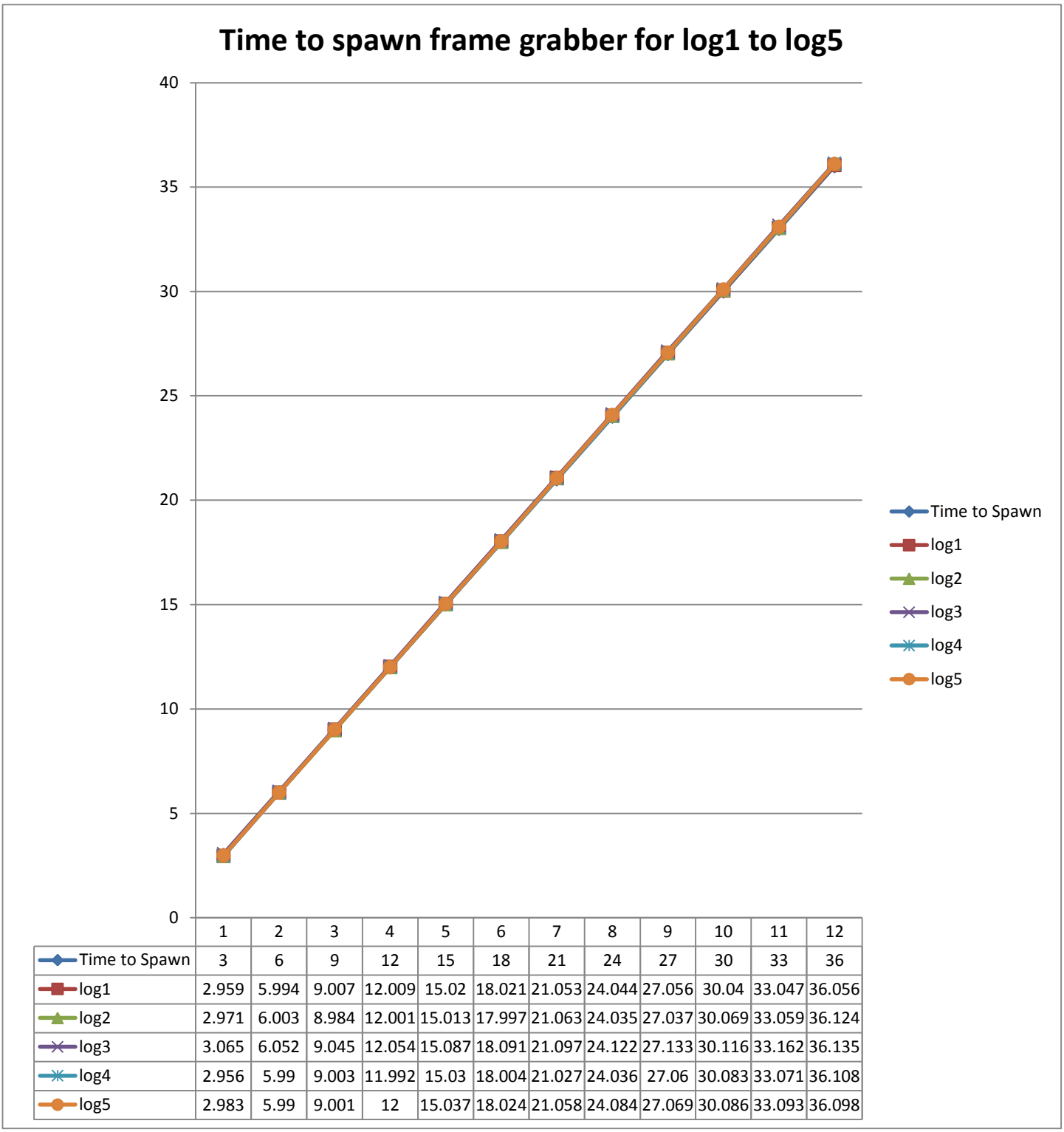
CPU utilization in case of worst case execution times used as completion time = 45.53%

Analysis from run information:

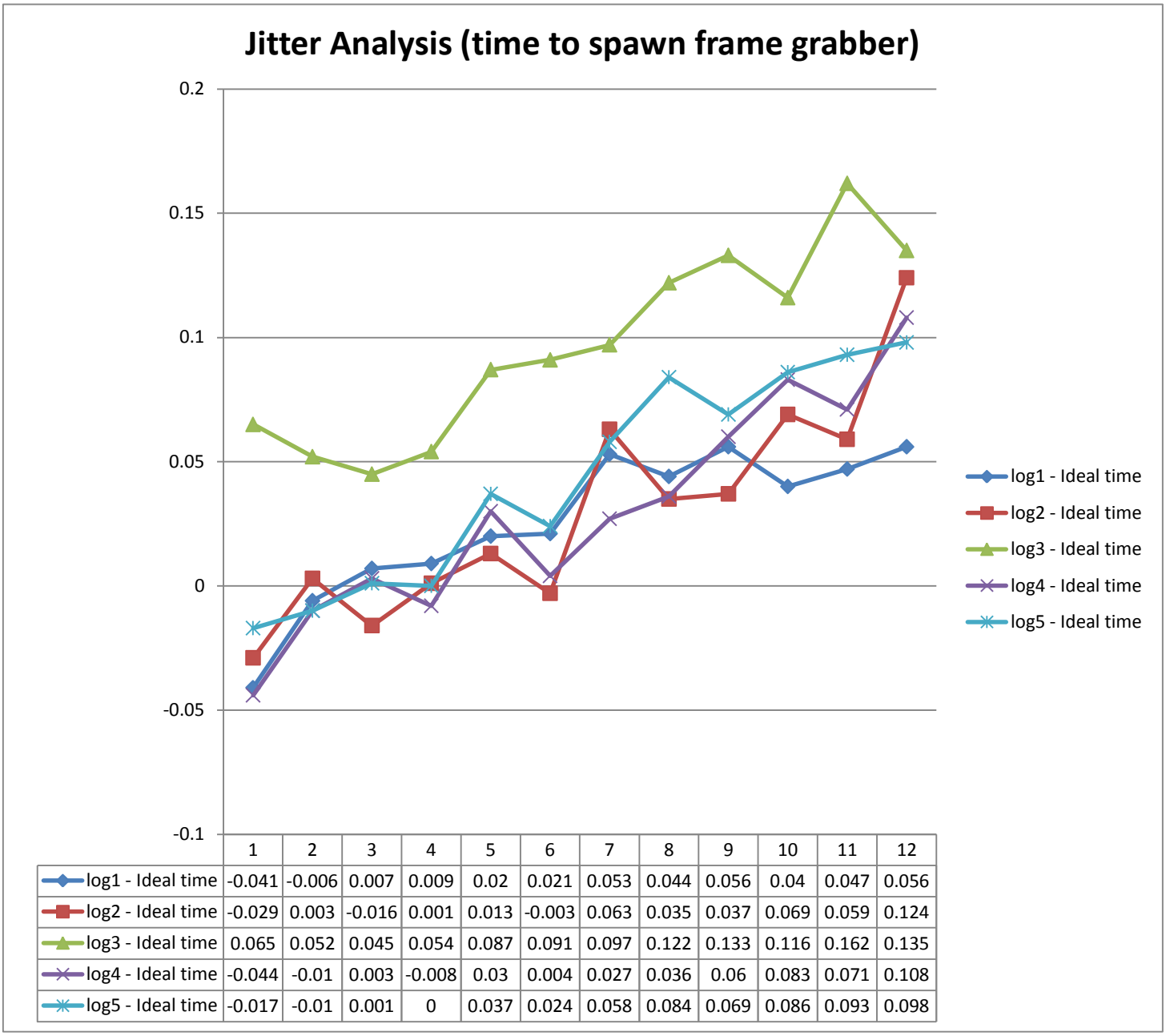
Please refer to log1.txt through log5.txt in the submission .zip for data collected for majority of the analysis attached below.

Each of the graphs below either points out the execution times vs the average or the expected execution time.

Jitter is calculated as the difference between the execution time and the expected/average execution time.



	1	2	3	4	5	6	7	8	9	10	11	12
Time to Spawn	3	6	9	12	15	18	21	24	27	30	33	36
log1	2.959	5.994	9.007	12.009	15.02	18.021	21.053	24.044	27.056	30.04	33.047	36.056
log2	2.971	6.003	8.984	12.001	15.013	17.997	21.063	24.035	27.037	30.069	33.059	36.124
log3	3.065	6.052	9.045	12.054	15.087	18.091	21.097	24.122	27.133	30.116	33.162	36.135
log4	2.956	5.99	9.003	11.992	15.03	18.004	21.027	24.036	27.06	30.083	33.071	36.108
log5	2.983	5.99	9.001	12	15.037	18.024	21.058	24.084	27.069	30.086	33.093	36.098



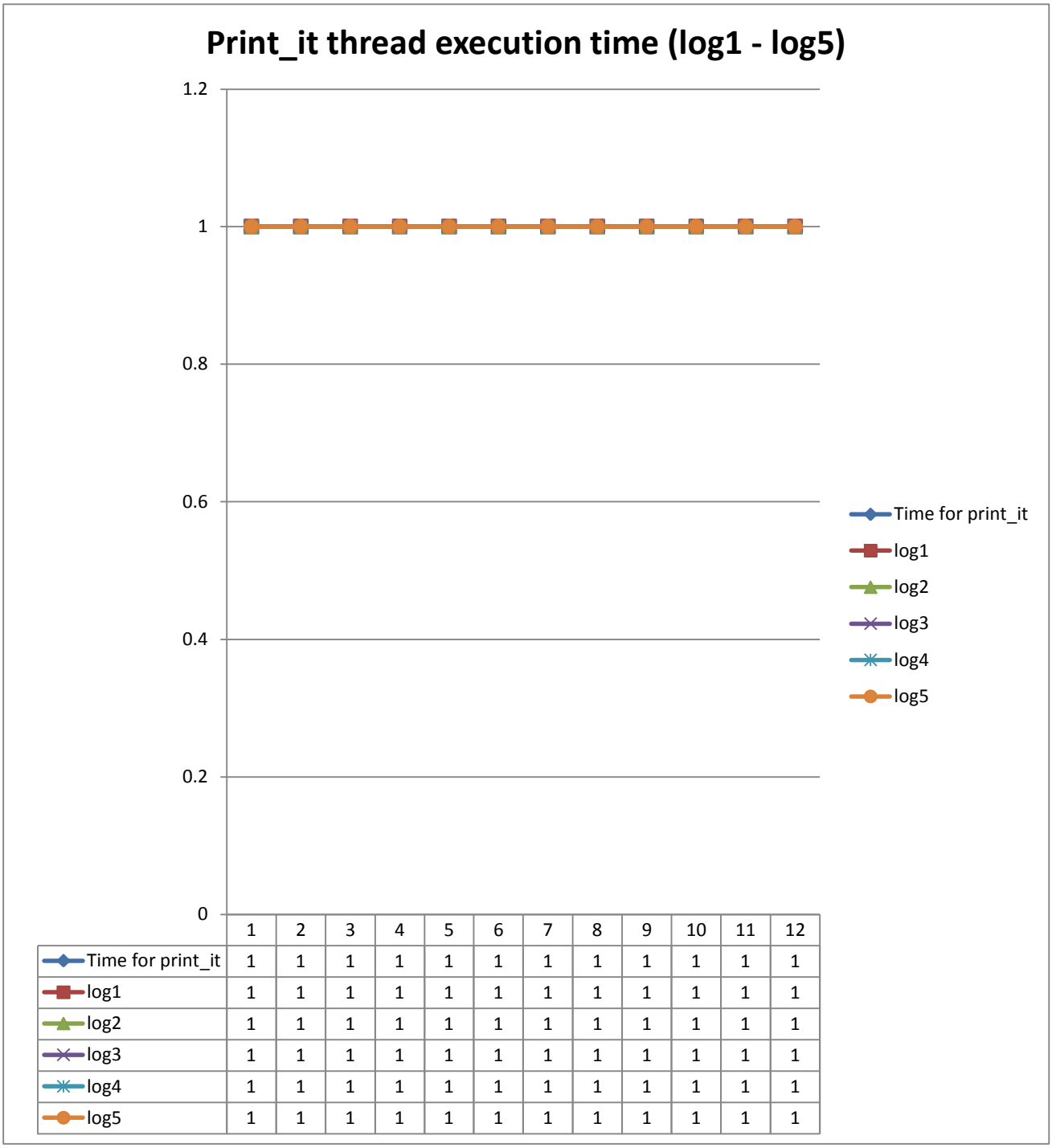
	1	2	3	4	5	6	7	8	9	10	11	12
log1 - Ideal time	-0.041	-0.006	0.007	0.009	0.02	0.021	0.053	0.044	0.056	0.04	0.047	0.056
log2 - Ideal time	-0.029	0.003	-0.016	0.001	0.013	-0.003	0.063	0.035	0.037	0.069	0.059	0.124
log3 - Ideal time	0.065	0.052	0.045	0.054	0.087	0.091	0.097	0.122	0.133	0.116	0.162	0.135
log4 - Ideal time	-0.044	-0.01	0.003	-0.008	0.03	0.004	0.027	0.036	0.06	0.083	0.071	0.108
log5 - Ideal time	-0.017	-0.01	0.001	0	0.037	0.024	0.058	0.084	0.069	0.086	0.093	0.098

Maximum negative jitter: -0.044 seconds

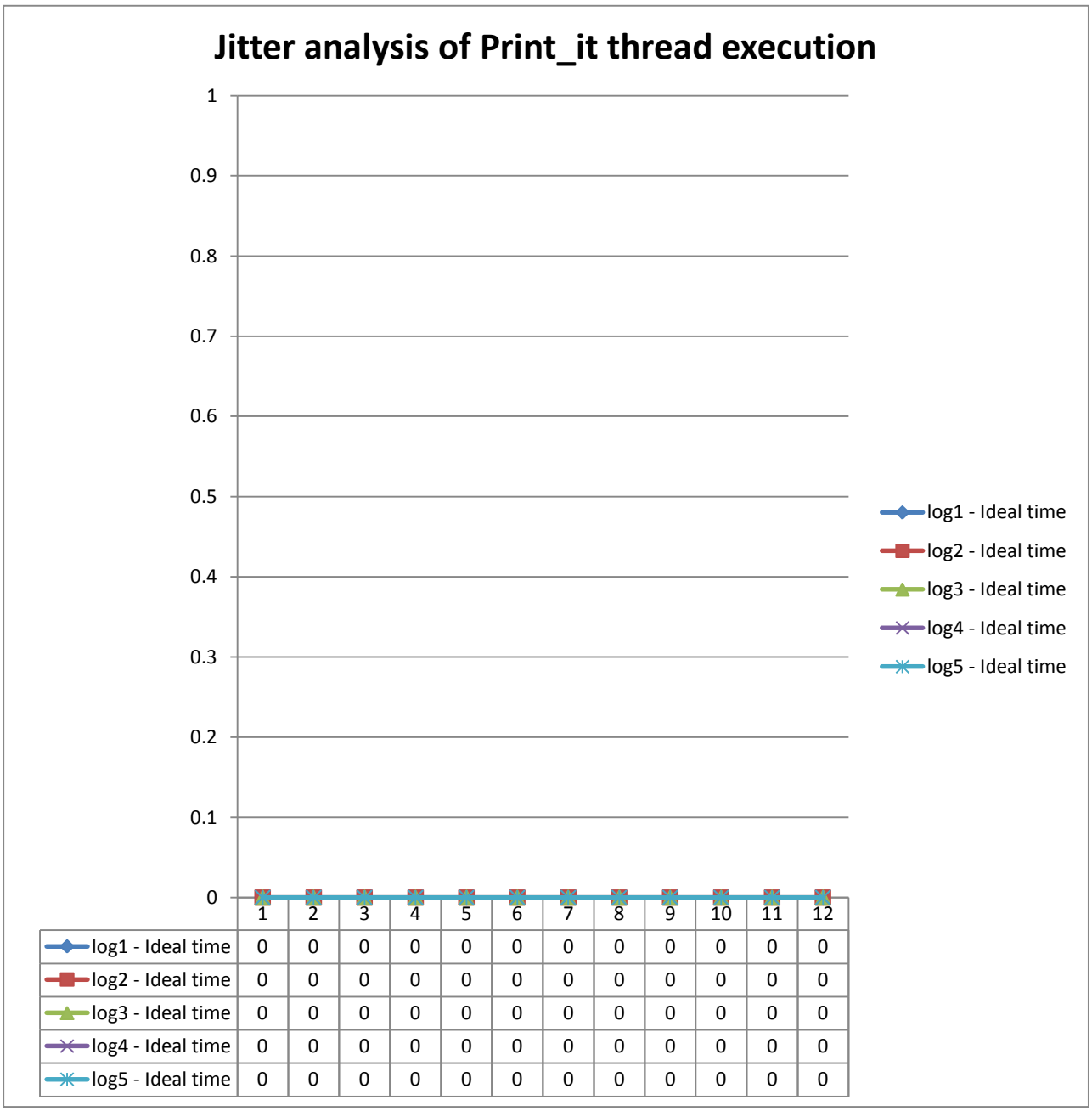
Maximum positive jitter: 0.162 seconds

Average jitter: 0.045 seconds

Very low jitter as no external processes influence timing.



	1	2	3	4	5	6	7	8	9	10	11	12
Time for print_it	1	1	1	1	1	1	1	1	1	1	1	1
log1	1	1	1	1	1	1	1	1	1	1	1	1
log2	1	1	1	1	1	1	1	1	1	1	1	1
log3	1	1	1	1	1	1	1	1	1	1	1	1
log4	1	1	1	1	1	1	1	1	1	1	1	1
log5	1	1	1	1	1	1	1	1	1	1	1	1

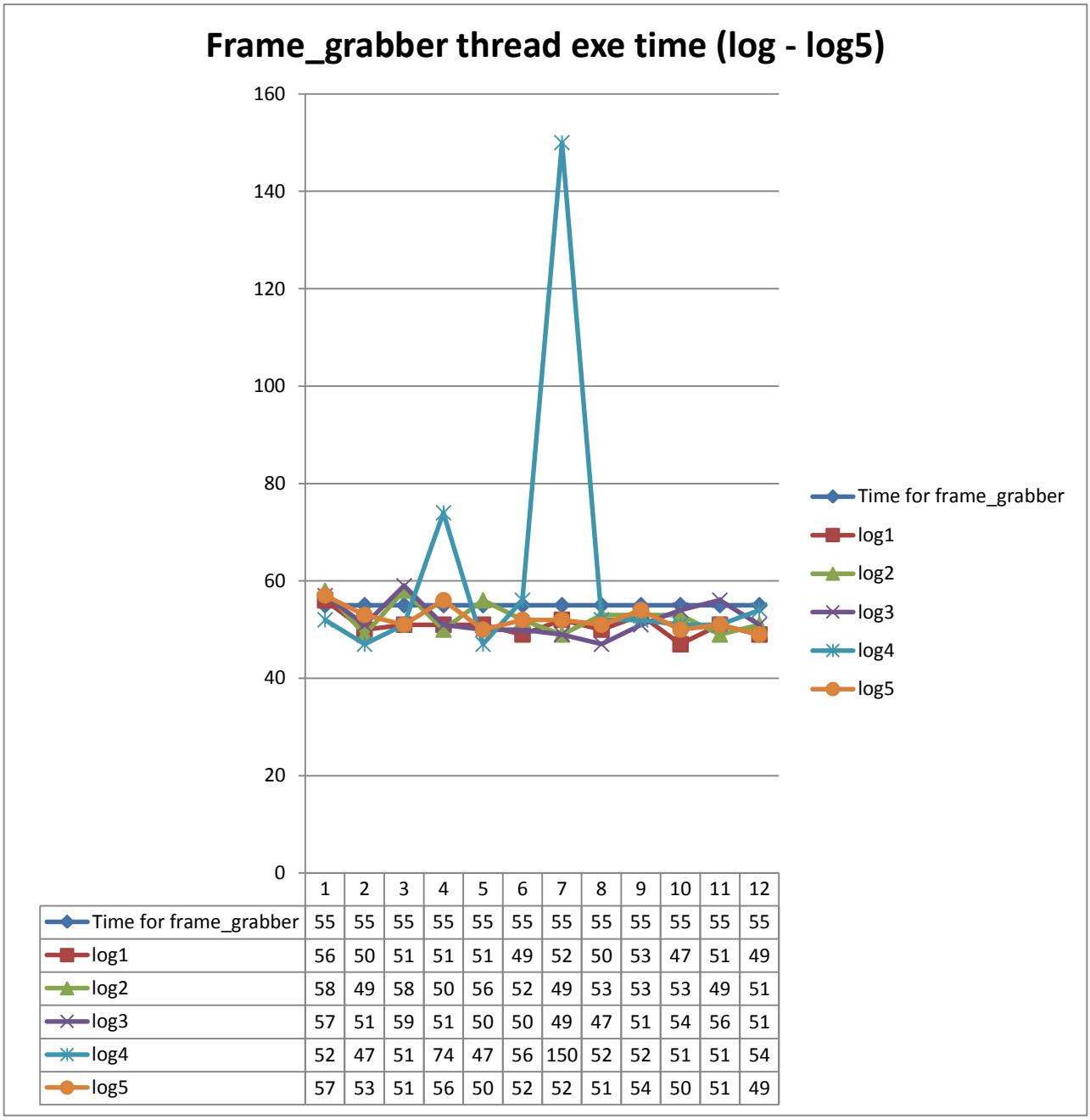


	1	2	3	4	5	6	7	8	9	10	11	12
log1 - Ideal time	0	0	0	0	0	0	0	0	0	0	0	0
log2 - Ideal time	0	0	0	0	0	0	0	0	0	0	0	0
log3 - Ideal time	0	0	0	0	0	0	0	0	0	0	0	0
log4 - Ideal time	0	0	0	0	0	0	0	0	0	0	0	0
log5 - Ideal time	0	0	0	0	0	0	0	0	0	0	0	0

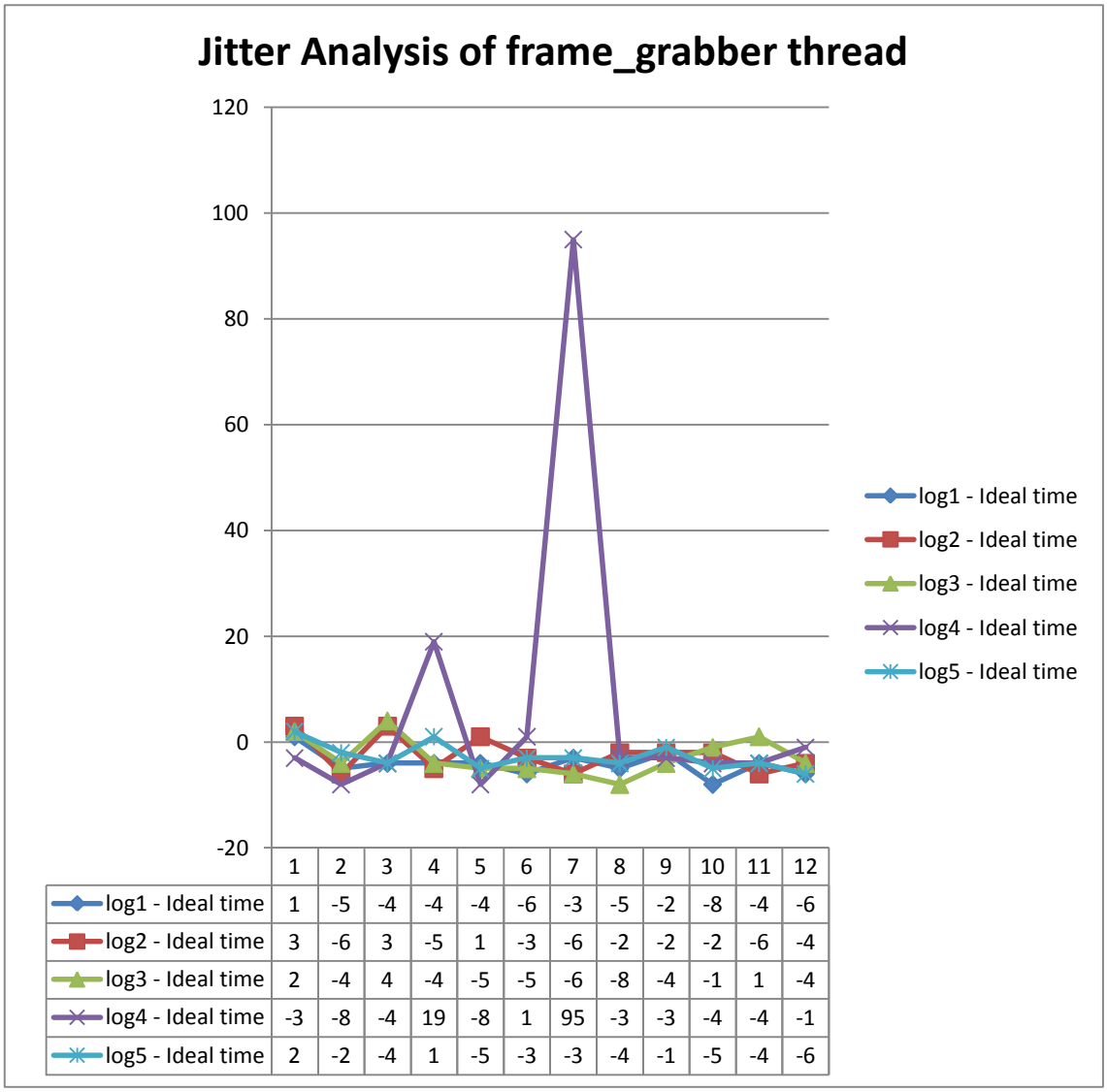
Maximum negative jitter: 0 seconds

Maximum positive jitter: 0 seconds

Average jitter: 0 seconds



	1	2	3	4	5	6	7	8	9	10	11	12
Time for frame_grabber	55	55	55	55	55	55	55	55	55	55	55	55
log1	56	50	51	51	51	49	52	50	53	47	51	49
log2	58	49	58	50	56	52	49	53	53	53	49	51
log3	57	51	59	51	50	50	49	47	51	54	56	51
log4	52	47	51	74	47	56	150	52	52	51	51	54
log5	57	53	51	56	50	52	52	51	54	50	51	49



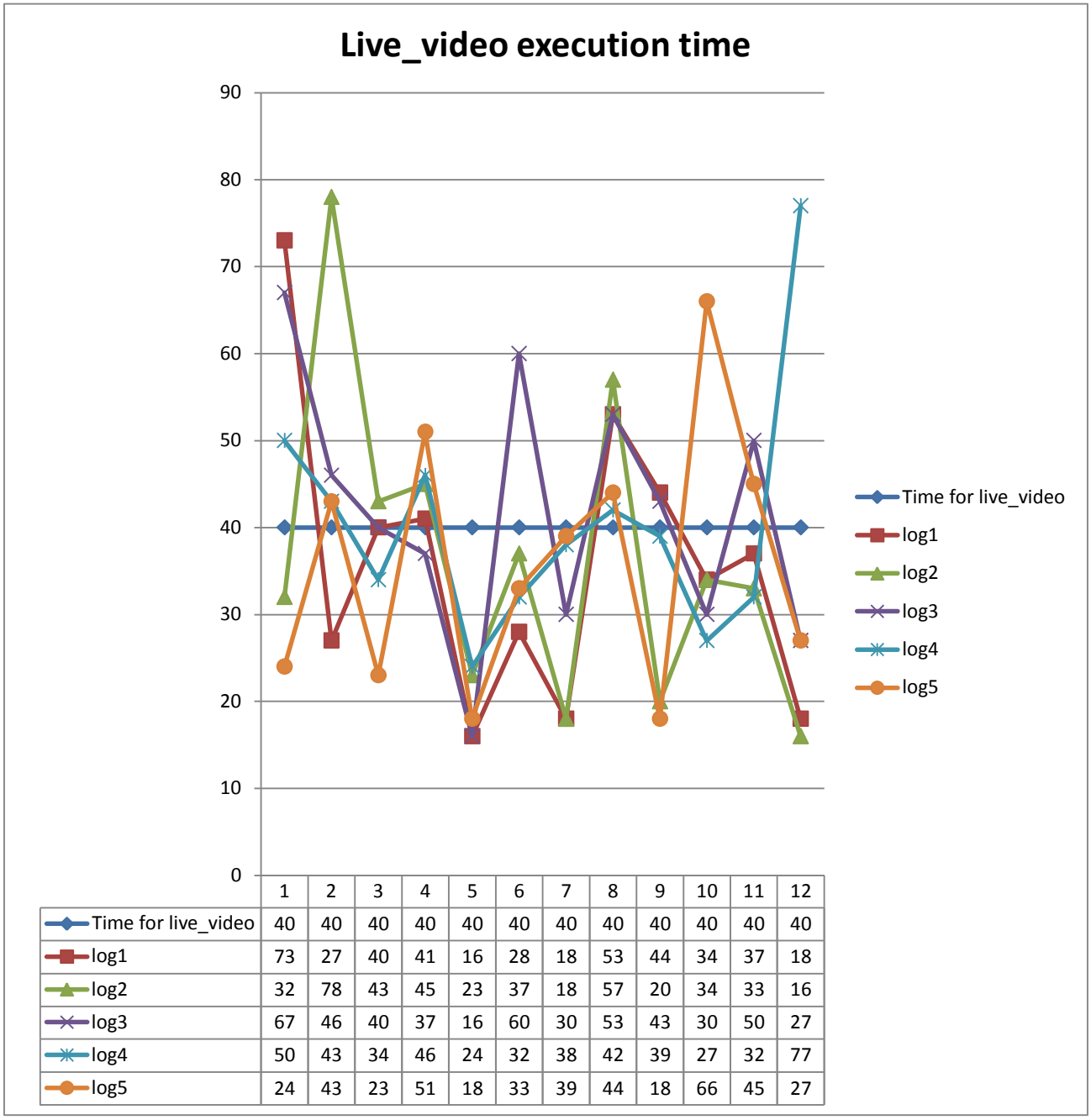
	1	2	3	4	5	6	7	8	9	10	11	12
log1 - Ideal time	1	-5	-4	-4	-4	-6	-3	-5	-2	-8	-4	-6
log2 - Ideal time	3	-6	3	-5	1	-3	-6	-2	-2	-2	-6	-4
log3 - Ideal time	2	-4	4	-4	-5	-5	-6	-8	-4	-1	1	-4
log4 - Ideal time	-3	-8	-4	19	-8	1	95	-3	-3	-4	-4	-1
log5 - Ideal time	2	-2	-4	1	-5	-3	-3	-4	-1	-5	-4	-6

Maximum negative jitter: -8 milliseconds

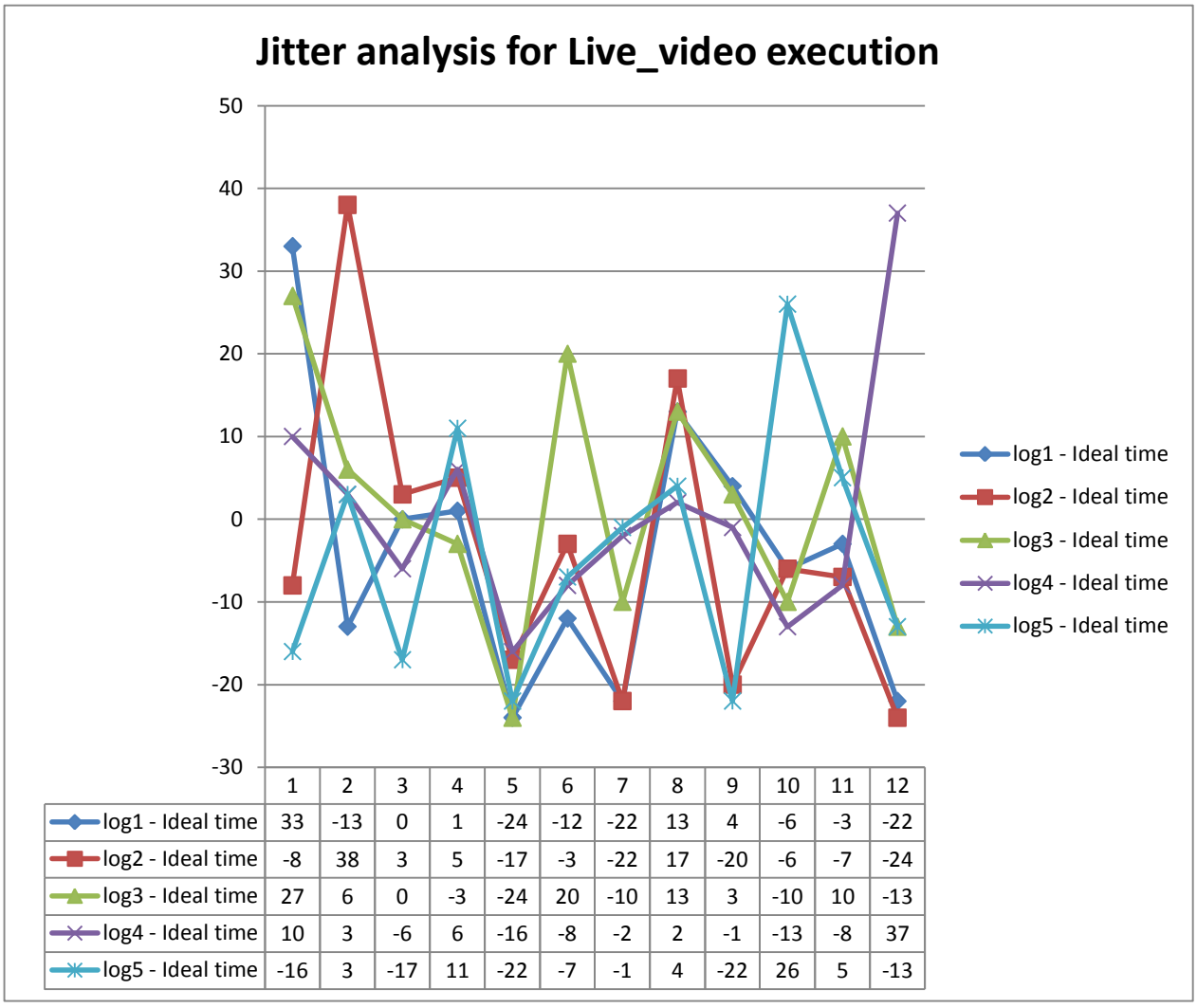
Maximum positive jitter: 95 milliseconds

Average jitter: -1.167 milliseconds

High jitter due to the dependency on the number of contours detected per frame.



	1	2	3	4	5	6	7	8	9	10	11	12
Time for live_video	40	40	40	40	40	40	40	40	40	40	40	40
log1	73	27	40	41	16	28	18	53	44	34	37	18
log2	32	78	43	45	23	37	18	57	20	34	33	16
log3	67	46	40	37	16	60	30	53	43	30	50	27
log4	50	43	34	46	24	32	38	42	39	27	32	77
log5	24	43	23	51	18	33	39	44	18	66	45	27



	1	2	3	4	5	6	7	8	9	10	11	12
log1 - Ideal time	33	-13	0	1	-24	-12	-22	13	4	-6	-3	-22
log2 - Ideal time	-8	38	3	5	-17	-3	-22	17	-20	-6	-7	-24
log3 - Ideal time	27	6	0	-3	-24	20	-10	13	3	-10	10	-13
log4 - Ideal time	10	3	-6	6	-16	-8	-2	2	-1	-13	-8	37
log5 - Ideal time	-16	3	-17	11	-22	-7	-1	4	-22	26	5	-13

Maximum negative jitter: - 24 milliseconds

Maximum positive jitter: 38 milliseconds

Average jitter: -2.0167 milliseconds

High jitter may be attributed to varying speeds over interfacing with the camera.

For tables to each of the above graphs, please refer to Book1.xls found in the submission .zip.

Graph generations are also clearly detailed in the excel document specified above.

Conclusion:

A video was recorded to check for detections and has been added to the appendix directory in the submission.

Considerable accuracy was achieved in detection and all deadlines were consistently met.

Therefore, the requirement of functional correctness and meeting deadlines implies real-time accuracy!