Synchronome Final Project Report

ECEN 5623 - Real-Time Embedded Systems

Brian Ibeling and Josh Malburg

8/7/2020

https://github.com/ibelingb/ecen5623_final_project.git

Project Overview

The goal of this project is to use a camera to observe a periodic physical process (a clock), synchronize with that process, controlling jitter and drift, and experience no glitches (frame blurring, skipping, or missing) over a 30 minute period. This needs to be observed at a minimum periodic rate of 1 Hz, and produce a predictable response over a long period of time. A stretch goal will be to observe the changes of a clock at a 10 Hz rate, showing the subsecond changes on a stopwatch over a 3 minute period. In either scenario (1 Hz monitoring for 30 minutes vs 10 Hz monitoring for 90 seconds), a total of 1800 frames will be captured, for which there should be no two frames which are the same or have any frames with blurring or distortion.

The goal of this project is characterize, analyze, and verify a real-time system, as well as to simulate a mission critical real-time system. A glitch or failure could result in loss of the asset and/or potentially loss of human life, so we need to be able to verify our software will work on a consistent, deterministic, and predictable manner.

Project Requirements

The following list provides requirements for completing the standard extended lab as defined here.

1. The system shall monitor the physical process of an Analog Clock or a Digital Stopwatch.
2. The system shall receive frame images from a camera at a rate of approximately 30 Hz.
3. The system shall acquire individual frames at a minimum resolution of 640 x 480.
4. The system shall acquire frames at a minimum rate of 20 Hz.
5. The system shall add a timestamp to each acquired frame in the frame header or onto the image directly.
6. The system shall add the target platform name to each acquired frame in the frame header or onto the image directly.
7. The system shall save frames in a PPM frame format.
8. The system shall save frames to persistent storage memory at a minimum rate of 1 Hz.
9. The system shall save frames to persistent storage memory at a maximum rate of 10 Hz.
10. The system shall operate for a maximum duration of 30 minutes when saving frames to persistent memory at a 1 Hz rate.
11. The system shall operate for a maximum duration of 1.5 minutes when saving frames to persistent memory at a 10 Hz rate.
12. The system shall only save unique frames (no duplicate frames).
13. The system shall only save frames without blurring or distortion.
14. The system shall save a frame for every unique position of the Analog Clock or Stopwatch.

15. The system shall have less than 1 second of error over a 30 minute duration.
16. The system shall capture logging data to determine system jitter and drift.
17. The system shall support an additional image processing feature for edge detection.
18. The system shall add hough circles/line transforms to find/draw the hands and border of the clock.
19. The system shall support the ability to enable or disable image processing features via a runtime flag passed to the application when it is started.

Project Design and System Analysis

To meet the project requirements as listed above, the following hardware and software services will be utilized.

Hardware Block Diagram

Figure 1 below shows the hardware that will be utilized for the Standard Final Project, as well as the interfaces connecting the various components. Capture of frames from a standard webcam (C270 or better) will be captured, stored, and analyzed on the Raspberry Pi 3B+ (RPi), with system logging data also captured for analysis of system performance as well as timing jitter and drift. Frames and logging data will then be passed to a remote server (desktop computer) for further analysis.

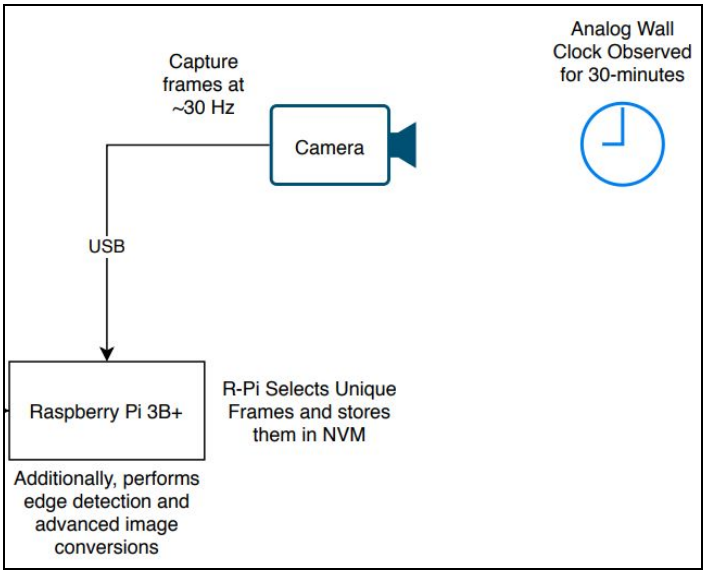

Figure 1: Key Hardware Elements for Standard Final Project

Software Source Diagram

Figure 2 below shows the Software-Hardware Stackup for the project, highlighting the various software services that will be developed, the utilized software libraries (OpenCV) and Linux

drivers, and hardware used. The use of OpenCV, a widely used machine vision tool, will greatly reduce complexity of developing image processing functionality with the limited schedule remaining (see Project Schedule section).

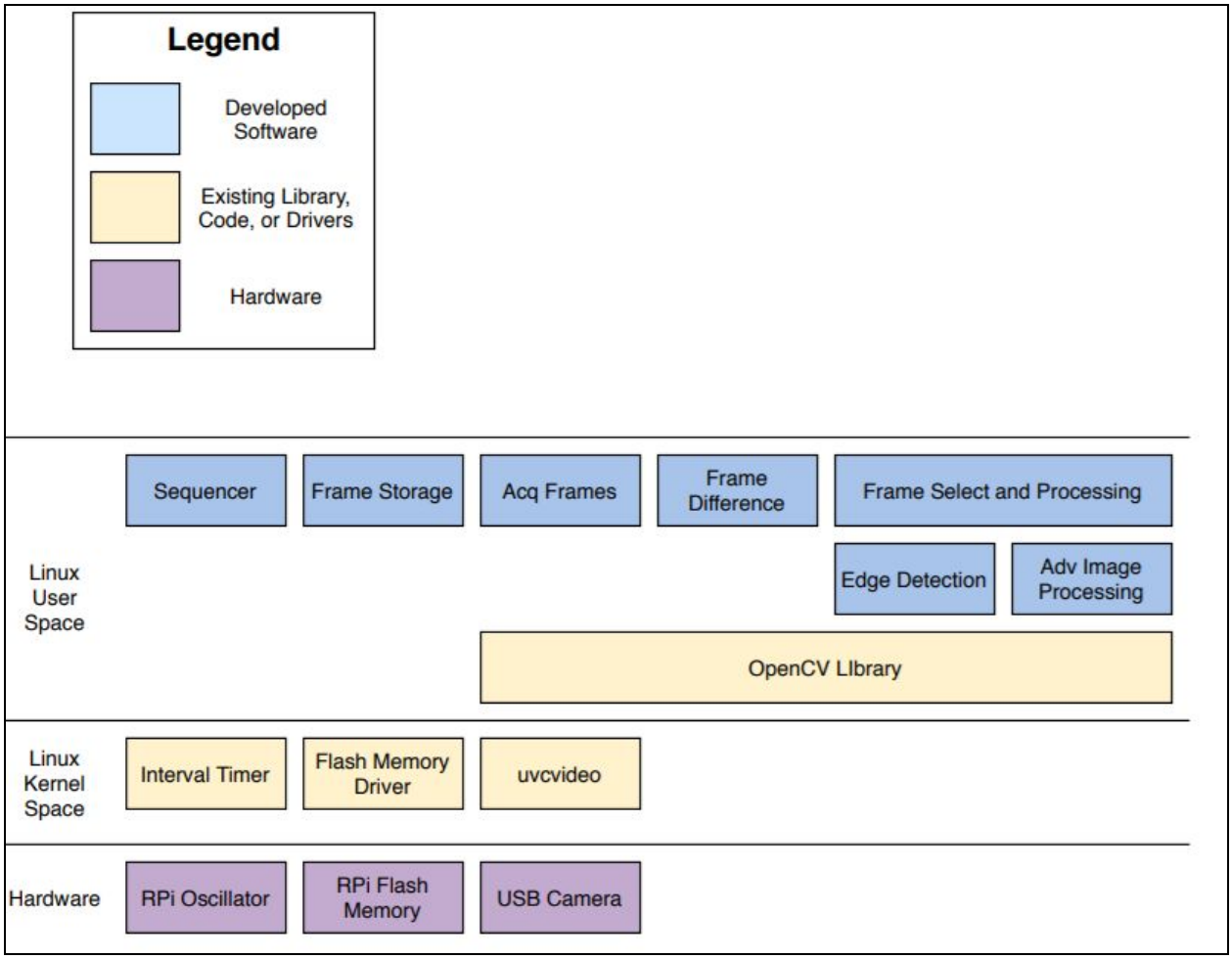

Figure 2: Software-Hardware Stackup for Standard Final Project

Process/Thread Diagram

Figure 3 presents a block diagram view of our software process. Our current concept includes 4 real-time tasks and one best-effort task. A timer will be used to run the sequencer and the sequencer will synchronize all other RT tasks using semaphores. The Frame Acquisition (FA) thread will read 24 frames a second which should be sufficient to capture each tick of a clock's second hand. The FA thread will add frames to a circular buffer that will be periodically processed by the Frame Difference (FD) thread. The FD thead's primary task is to identify frames in which the second hand moved; the thread will threshold the difference between subsequent frames to detect motion of the clock hands. Frames for each unique second hand position will be inserted into the Select message queue (selectQueue). Periodically, as dictated by the sequencer, the Frame Processing thread will read the Select MQ, use Hough line and circle transforms to color the clock arms and boundary and add the enhanced image to the Write message queue (writeQueue). The best-effort Frame Writer thread will block on reads to the write queue, waiting for new frames to write; before writing frames to file it will add text overlay for the frame time and target name. Since this thread is best effort it will be preemptible by the RT tasks and we need to ensure there's adequate slack time for writes to occur and the queue doesn't fill.

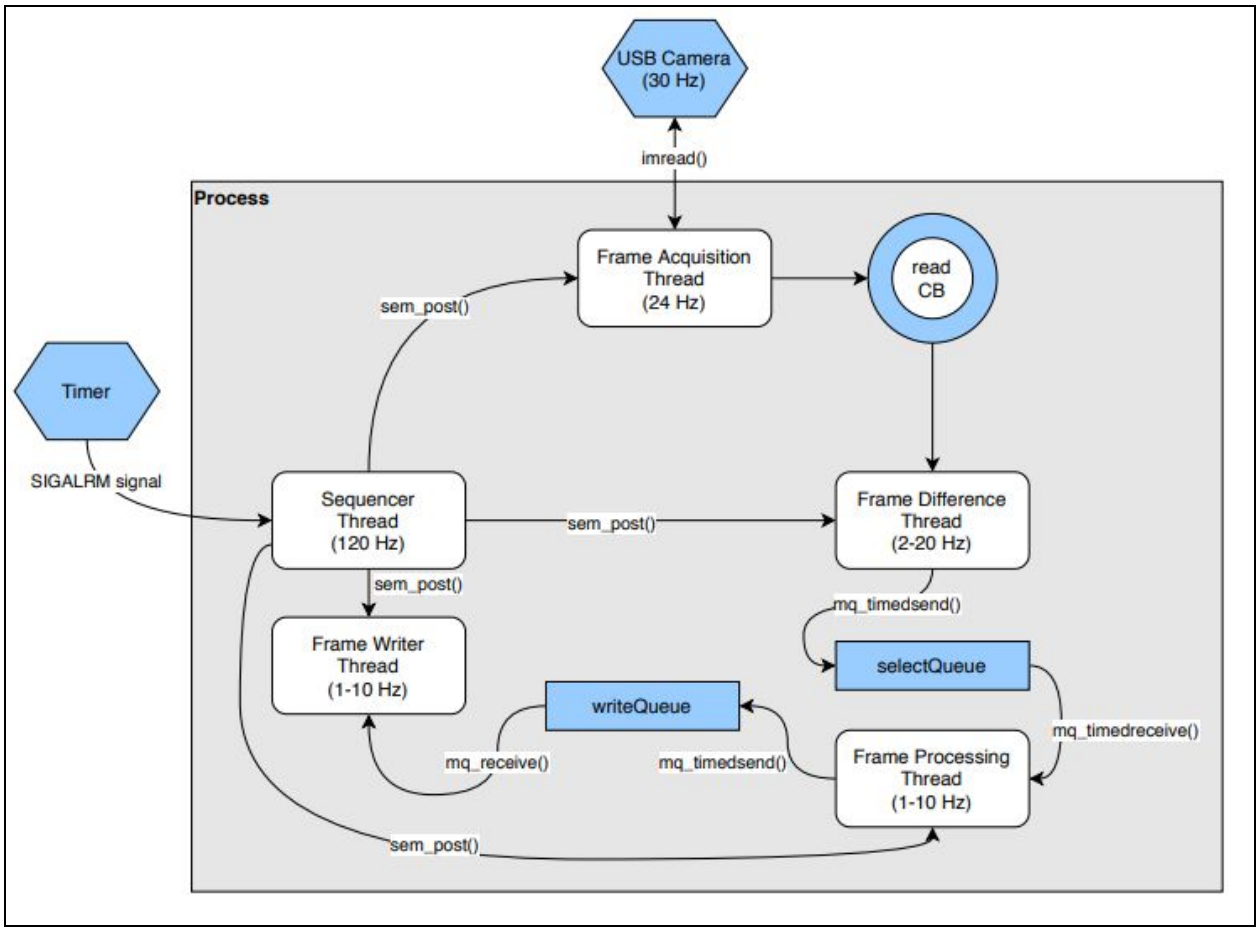

Figure 3: Software System Synchronization and Block Diagram

Image Processing Overview

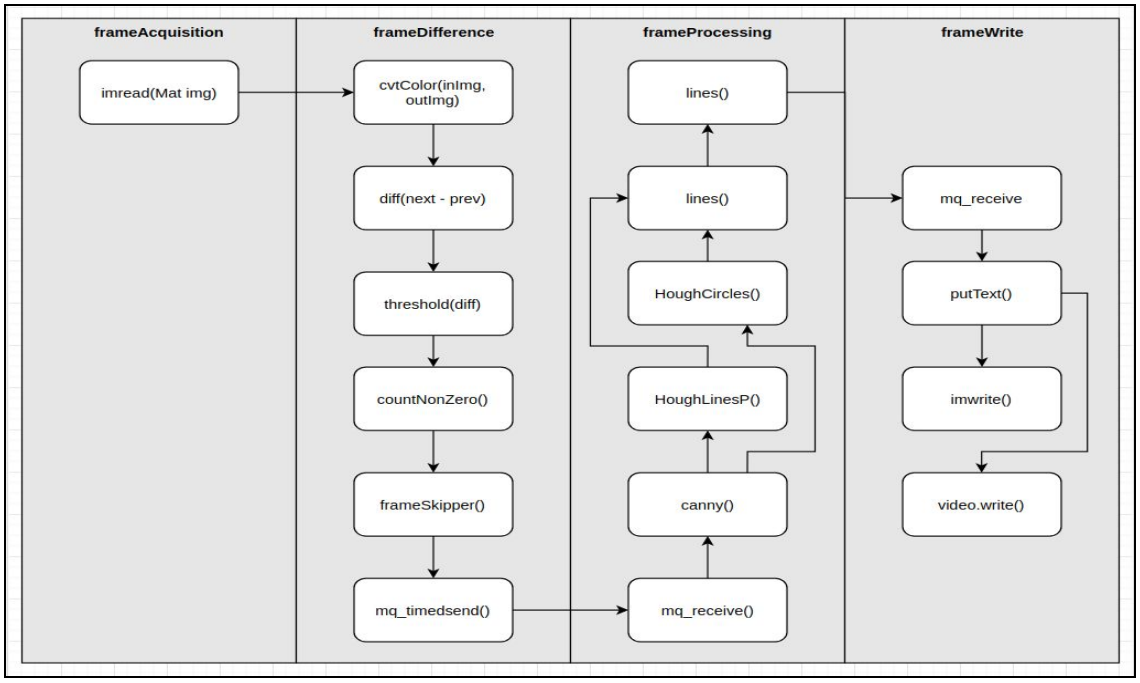

Figure 4: Image Processing Pipeline

The preceding diagram illustrates the various transformations applied to the image data. The frame acquisition simply reads frames and inserts them in the circular buffer. The frame difference thread, who's primary objective is to detect second hand motion first converts the image to grayscale then creates a difference frame using the two frames. The difference frame is then passed through a binary threshold function; if the pixel intensity of the difference frame is greater than a threshold value it is set to 0xFF (white) and if the pixel is below the threshold value it gets set to 0x00 (black). We then count the number of white pixels and if the number exceeds 100 we decide the second hand has changed and skip the next several frames before inserting a frame into the select queue. The processing queue then reads frames from the select queue and applies the advanced image processing algorithms. The thread first applies a canny filter to extract the clock edges and pass that frame to the hough transforms: HoughLinesP and HoughCircles. The purpose of these two functions is to highlight the clock hands and circular boundary. After the advanced processing has been applied the enhanced frames are passed through the write queue to the frameWrite thread. The frameWrite thread reads the frame time from the message structures to apply a text overlay before writing the final image to a file and video. The application of the canny filter and hough transforms can be enabled/disabled by command line arguments. Here's an example of the output:

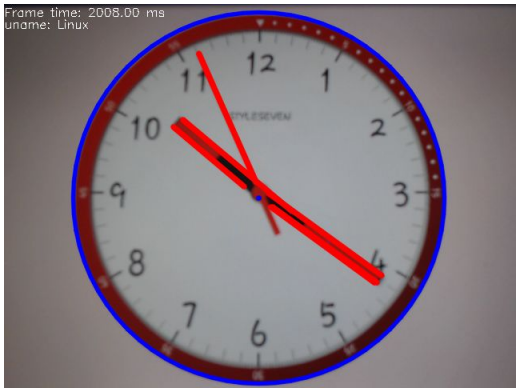

Figure 5: Advanced Processing Example

Thread State Diagrams

We are planning to assign thread priorities according to the Rate Monotonic policy which means the threads from highest to lowest priority will be: Sequencer, Frame Acquisition, Frame Difference, Frame Processing and Frame Storage. All of these threads will implement the state diagram shown below. When executing lower-priority threads are preempted by higher-priority threads they will move into the Ready state. Executing threads that need an unavailable resource (e.g. semaphore or queue) will transition to the pending state and when the resource becomes available the thread moves into the ready queue waiting to be dispatched by the scheduler. The schedule of course dispatches threads in the ready queue by priority. We are planning to either use clock_nanosleep (w/ absolute time to avoid drift) or a signal with a hardware time to trigger the sequencer. When waiting on a signal a thread goes to the pending state, but when using a sleep function it goes to the delayed state.

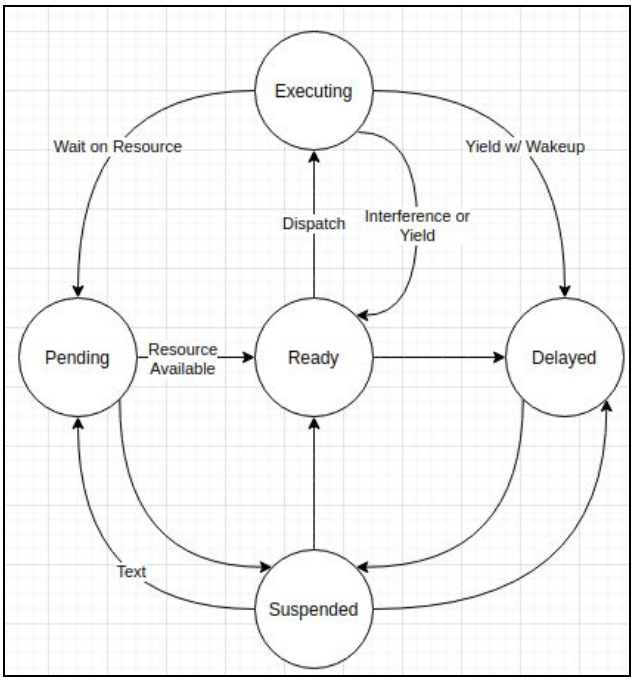

Figure 6: SW Service State Machine Diagram

Software Services

The list below provides definitions of all software services along with Rate Monotonic Analysis (RMA) details implemented to meet the requirements as detailed above. All services write logging data to syslog to allow for tracing and validation of the system, as well as analyze execution time of each service.

1. **Sequencer (S1) service thread (120 Hz RT task)**

The Sequencer service thread acts as an executive dispatch service, updating blocking semaphores to all other service threads within the system at specified intervals. These dispatch

intervals are a substrate of the Sequencer's base rate, which is set to 120 Hz. This service is driven from the high frequency oscillator provided by the RPi3, checking a comparator against the Interval Timer (IT) and sending an interrupt signal to the Sequencer at the desired frequency.

Table 1 shows a breakdown of the RMA components for this S1 service.

Table 1: Sequencer Service (S1) RMA Details

Execution Frequency	Period (T)	Deadline (D)	WCET (C)
120 Hz	8.33 msec	8.33 msec	2.25 msec

The execution frequency of 120 Hz was determined by the need to have a high-rate scheduler with a rate that is a least common multiple (LCM) of all services being supported in this application. This includes sampling an analog clock at 1 Hz and sampling a digital stopwatch at 10 Hz. Rates of 24 Hz, 20 Hz, 10 Hz, 2 Hz, and 1 Hz resulted in a selection of 120 Hz for the base rate.

The Service Period and Deadline were determined based on the Execution Frequency.

The Worst Case Execution Time (WCET) was determined by analyzing timestamps written to syslog at the start and end of each sequencer execution cycle. In excel, the difference in time between these two events resulted in an observed WCET of 2.25 msec (0.10 msec average). The analysis and supporting data for this result can be found here.

2. **Frame Acquisition (S2) service thread (24 Hz RT task)**

The Frame Acquisition (FA) thread is responsible for reading frames from a USB camera at a desired frame rate of 24 frames per second. The thread uses OpenCV VideoCapture class to open the device, configure the frame size and turn off the autofocus feature. The FA thread will insert read frames into a circular buffer to be read and processed by the Frame Difference (FD) thread.

Table 2 shows a breakdown of the RMA components for this S2 service.

Table 2: Frame Acquisition Service (S2) RMA Details

Execution Frequency	Period (T)	Deadline (D)	WCET (C)
24 Hz	41.666 msec	41.666 msec	12 msec

The execution frequency of 24 Hz was determined by the requirement to process frames at a max rate of 10 Hz using the digital stopwatch. The Nyquist Sampling Theorem specifies that a sampling rate of greater than 2x the base rate of the event being sampled. To ensure that no

frames are missed, a slight increase in frames received is used, resulting in the 24 Hz rate being selected. This also supports both clock types of a 1 Hz Analog Clock and a 10 Hz Digital Clock.

The Service Period and Deadline were determined based on the Execution Frequency, with a requirement to acquire frames from a camera at a minimum rate of 20 Hz.

The Worst Case Execution Time (WCET) was determined by analyzing timestamps captured from syslogs written when the service begins reading a frame from the camera and when the service has written that frame to the Circular Buffer (CB). Viewing this data in Excel and calculating the time difference between each event, values range from 4-12 msec (4.24 msec average), with the max of this range used for the WCET of S2.

3. Frame Difference (S3) service thread (2 Hz RT task)

The Frame Difference (FD) thead's primary task is to identify frames in which the second hand moved; the thread thresholds a difference frame to detect motion of the clock hands. Frames for each unique second hand position will be inserted into the Select message queue (selectQueue).

Table 3 shows a breakdown of the RMA components for this S3 service.

Table 3: Frame Difference Service (S3) RMA Details

Execution Frequency	Period (T)	Deadline (D)	ACET (C)	WCET (C)
(1 Hz Analog Clock) 2 Hz	500 msec	500 msec	64.5 msec	308 msec
(10 Hz Digital Stopwatch) 20 Hz	50 msec	50 msec	14.66 msec	28.25 msec

The execution frequency for both cases was determined by a desire to run this service at twice the minimum sample rate required based on the input device (analog vs digital clock).

Similarly, the Service Period and Deadline for each case was determined based on the requirement to detect a change at the needed rate based on the input device.

The Worst Case Execution Time (WCET) for the 1 Hz Analog Clock case was determined by analyzing timestamps captured from syslogs written when the service begins processing a frame and when the service has written a frame to the selectQueue. Viewing the data in Excel, the time difference was calculated between each event. There are two events to account for: calculating differences between frames and writing a found difference to the selectQueue. The execution time for determining differences between frames ranges from 4-12 msec (4.21 msec average) to complete processing per frame. For the sake of worst-case, we'll use 12 msec across 24 frames per 1 sec received from the Acquisition Thread, which gives 144 msec over the expected period. If a difference is found between frames, this needs to be written to the

selectQueue. Again, viewing the data in excel and comparing the time offset from when frame processing started to when it completed writing data to the selectQueue, processing time ranged between 8-20 msec (13.98 msec average). Again, taking the high end of this gives 20 msec per frame write to the selectQueue. The following equations summarizes the analysis above for S3 ACET and WCET for an Analog Clock:

$$((4.21\ msec\ frame\ difference\ calc) * (24\ frames\ per\ 1\ sec)) + (13.98\ msec\ insert\ to\ MQ) = 64.5\ msec\ S3\ ACET$$
$$((12\ msec\ frame\ difference\ calc) * (24\ frames\ per\ 1\ sec)) + (20\ msec\ insert\ to\ MQ) = 308\ msec\ S3\ WCET$$

The ACET was included with this analysis due to the WCET likely being too extreme in the estimates made, and wanted to highlight that deadlines are still likely to be met for both cases.

WCET and ACET analysis for the 10 Hz Digital Stopwatch will be completed in the coming week.

4. Frame Processing (S4) service thread (1 Hz RT task)

The Frame Processing thread reads frames from the Select message queue and applies Hough line and circle transforms to find the clock hands and circular border. A canny filter is also used to enhance edges making it easier for lines and circles to be detected. Enhanced images are added to the Write message queue.

Table 4 shows a breakdown of the RMA components for this S4 service.

Table 4: Frame Processing Service (S4) RMA Details

Execution Frequency	Period (T)	Deadline (D)	ACET (C)	WCET (C)
(1 Hz Analog Clock) 1 Hz	1000 msec	1000 msec	48.07 msec	96 msec
(10 Hz Digital Stopwatch) 10 Hz	100 msec	100 msec	2.58 msec	8.82 msec

The execution frequency for both cases was determined by the requirement to sample frames based on the input device (analog vs digital clock) at the specified rate of 1 Hz and 10 Hz respectively.

Similarly, the Service Period and Deadline for each case was determined based on the requirement to detect a change at the needed rate based on the input device.

Similar to analysis of S3, the Worst Case Execution Time (WCET) for the 1 Hz Analog Clock case was determined by analyzing timestamps captured from syslogs written when the service

begins processing a frame and when the service has written a frame to the writeQueue. Analysis for S4 is simpler than compared to S3, which saw a 1:1 relation between the start of frame processing and frame written to the writeQueue.

WCET and ACET analysis for the 10 Hz Digital Stopwatch will be completed in the coming week.

5. **Frame Write (S5) service thread (1 Hz, 10 Hz Best Effort task)**

The Frame Write service thread is being implemented in a non-RT fashion, writing received frames to memory using a Round Robin scheduler and not a FIFO scheduler like the other services listed above. Frames will be received from the S4 Frame Processing service thread via a writeQueue, a text overlay with the frame time and device name is added before the frame is saved to non-volatile memory (NVM) in a best-effort manner. This was done to assist with decoupling memory-bound processes such as this one from other CPU-based processes such as S2-S4 above. This also reduces the application blocking due to memory input/output (IO) intensive events while frames are saved. Additionally, the writeQueue provides additional buffering to help ensure frames are not dropped or overwritten before the low-priority Frame Write service is able to process them.

CPU Affinity and RT Priority

The Raspberry Pi 3B+ has a total of 4 cores available. CPU Affinity for the services above is detailed in the Table 5 below:

Table 5: Service Threads CPU Affinity and Priority

Services	Priority	Assigned CPU Core	Justification
S1: Sequencer (120 Hz)	RT_MAX - 1	Core 4	Highest priority as this drives execution timing for all other services. Isolated to its own CPU core due to high execution rate.
S2: Frame Acquisition (24 Hz)	RT_MAX - 2	Core 3	Higher priority to ensure frames are captured into system at the relatively fast rate needed. Isolated to its own CPU core due to high execution rate and importance to capture frames.
S3: Frame Difference (2 Hz for 1 Hz Clock)	RT_MAX - 3	Core 2	Medium priority to process buffered data received.

(20 Hz for 10 Hz Clock)			Shared on CPU Core 2 due to low frequency of execution.
S4: Frame Select (2 Hz for 1 Hz Clock) (20 Hz for 10 Hz Clock)	RT_MAX - 4	Core 2	Medium priority to process buffered data received. Shared on CPU Core 2 due to low frequency of execution and analysis from 1 Hz testing.
S5: Frame Write (10 Hz for 1 Hz Clock) (20 Hz for 10 Hz Clock)	Round Robin, RT_MAX - 5	Core 1	Lowest priority executed as a Best Effort thread to write frames to NVM memory. Done so to help isolate memory IO from rest of system processing.

Software Verification Methods

Our system will use syslog messages to verify our real-time performance. Each thread will add a comma-delimited log message to the log at the start and end of execution time and when key events occur. This will enable us to evaluate that the periodicity of our threads agrees with our schedule and enables us to find each thread's worst-case execution time over many iterations. Our log file will contain messages when each read frame was written to the circular buffer, messages when the Clock Sync thread started, found a new time frame and stopped execution; messages when the Frame Processing thread started, read a message from the Select queue, inserted a message in the Write queue and ended execution; and messages from the best-effort write back thread showing when the thread was able to perform its work. All these log messages will be imported as a CSV file into a spreadsheet tool and we'll plot the timestamps to verify all thread start times are monotonically increasing and calculate the start time jitter.

During implementation we'll occasionally plug our thread statistics into Cheddar to verify our schedule is feasible. Since our only thread accessing I/O is best effort, we will set our timeline equal to our deadline. We will also evaluate the impact to CPU Utilization with the image processing features On/Off.

We also plan to use htop to verify thread priorities and average CPU loading for each thread. And time-permitting try to include ftrace, sysprof and/or gpro.

System Performance and Analysis

Sample Output Tests

Our system was able to successfully detect all changes in motion of the analog clock updating at a 1Hz rate over a 30 minute time period, capturing 1800 frames total. The data for the individual frames can be found here, and a diff from the starting frame to the final frame is shown in Figure 7 below. This test case was run with filtering enabled, video output at runtime disabled, and hough image processing disabled.

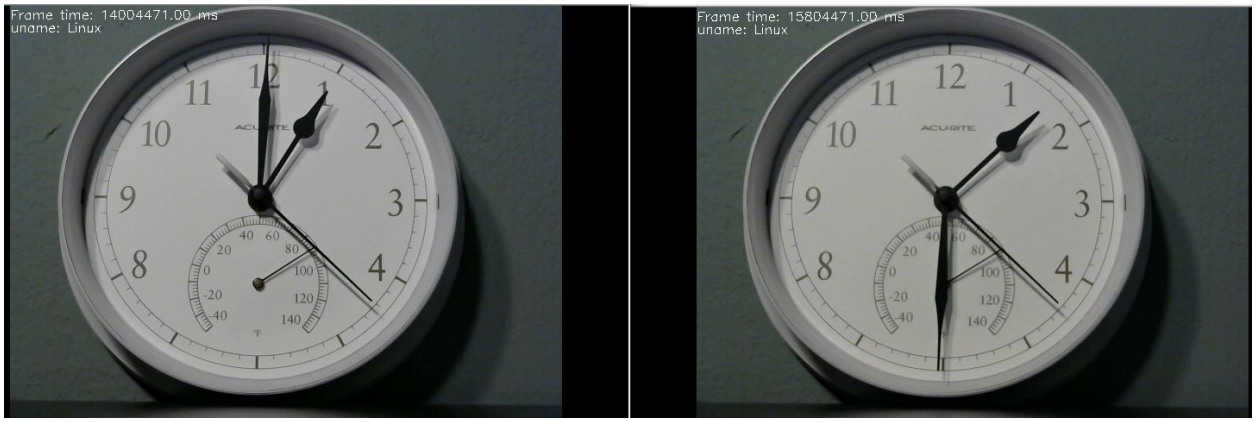

Figure 7: Diff of Starting Frame (Left) and End Frame (Right) for 1Hz Data w/ Analog Clock after 30 min (Filter Enabled, Video output and Hough filter disabled)

The next test case was run with filtering enabled, video output at runtime enabled, and hough image processing disabled. The data for the individual frames (first and ending 150 frames) can be found here, the video produced by the application while the frames were being processed can be found here, and a diff from the starting frame to the final frame is shown in Figure 8 below.

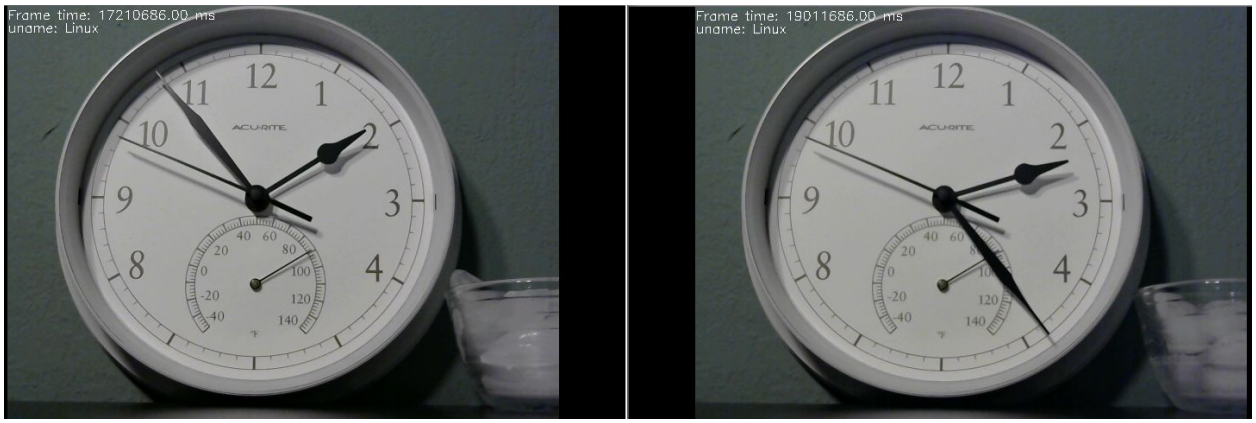

Figure 8:Diff of Starting Frame (Left) and End Frame (Right) for 1Hz Data w/ Analog Clock after 30 min (Filter Enabled and Video output enabled, Hough filter disabled)

The next test case was run with filtering enabled, video output at runtime enabled, and hough image processing enabled. The data for the individual frames (first and ending 150 frames) can be found here, the video produced by the application while the frames were being processed can be found here, and a diff from the starting frame to the final frame is shown in Figure 9 below.

Figure 9: Diff of Starting Frame (Left) and End Frame (Right) for 1Hz Data w/ Analog Clock after 30 min (Filter Enabled, Video output enabled, and Hough filter enabled)

A baseline test case was run with filtering disable, video output at runtime enable, and hough image processing disable. The data for the individual frames (first and ending 150 frames) can be found here, the video produced by the application while the frames were being processed can be found here, and a diff from the starting frame to the final frame is shown in Figure 9a below.

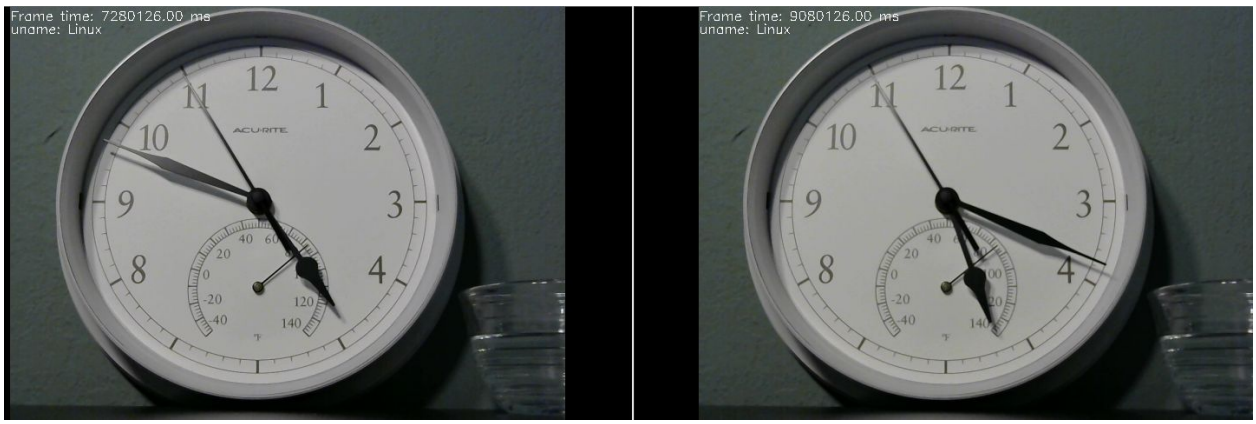

Figure 9a: Diff of Starting Frame (Left) and End Frame (Right) for 1Hz Data w/ Analog Clock after 30 min (Filter disabled, Video output disabled, and Hough filter disabled)

System Performance and 1 Hz Analog Clock Testing

The ACET and WCET for all threads across the various configurations for the successful 1 Hz analog clock testing can be found in Table 6 below. This was captured by recording the timestamp at the start and end of each service thread execution cycle, then finding the difference between those values using excel.

Table 6: Service Threads Recorded ACET and WCET from 1Hz Data

App Type	Service	ACET	WCET	Period	Deadline
Filter: Off Hough: Off Video: **On**	S2: FrameAcq	7.20 msec	96 msec	41.66 msec	41.66 msec
Filter: **On** Hough: Off Video: Off	S2: FrameAcq	7.43 msec	96 msec	41.66 msec	41.66 msec
Filter: **On** Hough: Off Video: **On**	S2: FrameAcq	7.26 msec	76 msec	41.66 msec	41.66 msec
Filter: **On** Hough: **On** Video: **On**	S2: FrameAcq	7.06 msec	74 msec	41.66 msec	41.66 msec
Filter: Off Hough: Off Video: **On**	S3: FrameDiff	30.10 msec	57 msec	500 msec	500 msec
Filter: **On** Hough: Off Video: Off	S3: FrameDiff	30.08 msec	55 msec	500 msec	500 msec
Filter: **On** Hough: Off Video: **On**	S3: FrameDiff	25.14 msec	56 msec	500 msec	500 msec
Filter: **On** Hough: **On** Video: **On**	S3: FrameDiff	28.62 msec	56 msec	500 msec	500 msec
Filter: Off Hough: Off Video: **On**	S4: FrameProc	3.04 msec	12 msec	1000 msec	1000 msec

Filter: **On** Hough: Off Video: Off	S4: FrameProc	14.86 msec	57 msec	1000 msec	1000 msec
Filter: **On** Hough: Off Video: **On**	S4: FrameProc	9.71 msec	70 msec	1000 msec	1000 msec
Filter: **On** Hough: **On** Video: **On**	S4: FrameProc	487.93 msec	932 msec	1000 msec	1000 msec
Filter: Off Hough: Off Video: **On**	S5: FrameWrite	185.00 msec	4386 msec	1000 msec	1000 msec
Filter: **On** Hough: Off Video: Off	S5: FrameWrite	113.05 msec	4855 msec	N/A	N/A
Filter: **On** Hough: Off Video: **On**	S5: FrameWrite	171.75 msec	4674 msec	N/A	N/A
Filter: **On** Hough: **On** Video: **On**	S5: FrameWrite	220.51 msec	9978 msec	N/A	N/A

It can be noted from Table 6 that numerous services across the 3 separate cases tested (different features enabled) have recorded WCET values that exceed the expected deadline for that service execution frequency. Let's analyze each of these individually for the different services.

For S2 (FrameAcquisition), this likely means that a frame arrived later than the 24 Hz expected rate compared to the sequencer driving rate. This likely results in a single frame being missed out of the 24 frames being received each second, and has a minimal or negligible consequence for this period deadline being missed, since the clock update rate is so slow at 1Hz, and thus not seeing a missed frame within the output frame data captured. Figure 10 below shows the execution time of the S2 service for the duration of the 30-minute test. The worst case 96 msec cycle only occurs twice at the beginning of the test, with an average execution time falling below 10 msec.

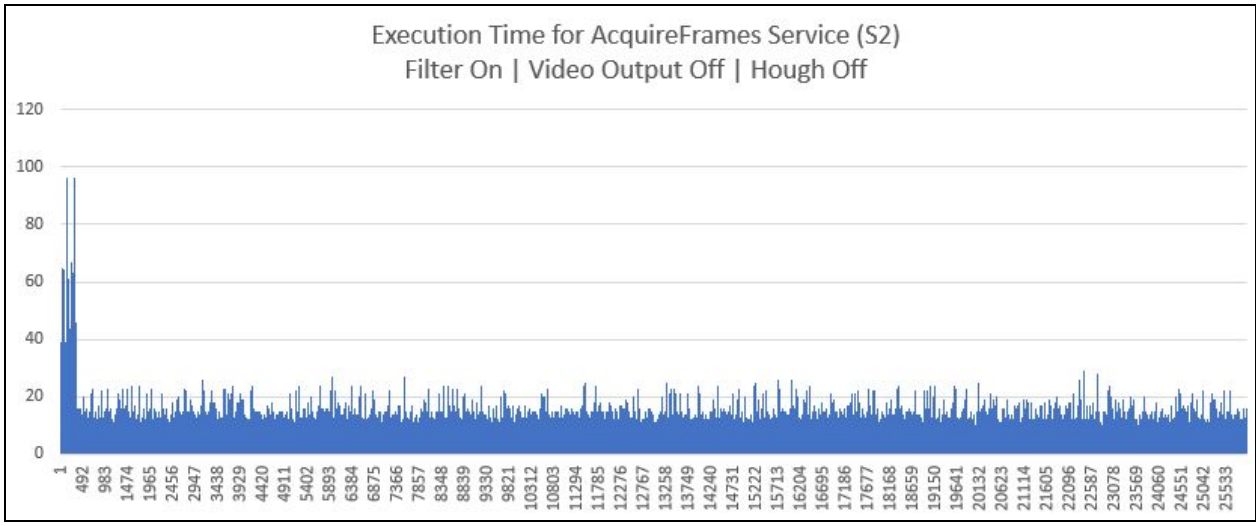

Figure 10: Execution Time for FrameAcquire S2 Service (Filter On, Video Off, Hough Off)

Similar for the other test cases where the S2 service missed deadlines; due to the higher than necessary sampling rate for the 1 Hz analog clock, a missed deadline compared to the sequencer execution rate. Figure 11 shows the next test case, and more noise associated with its execution. This is perhaps due to the S4 service sharing the same core as S2, and additional frame processing perhaps being needed due to the addition of ice melting to the camera field of view.

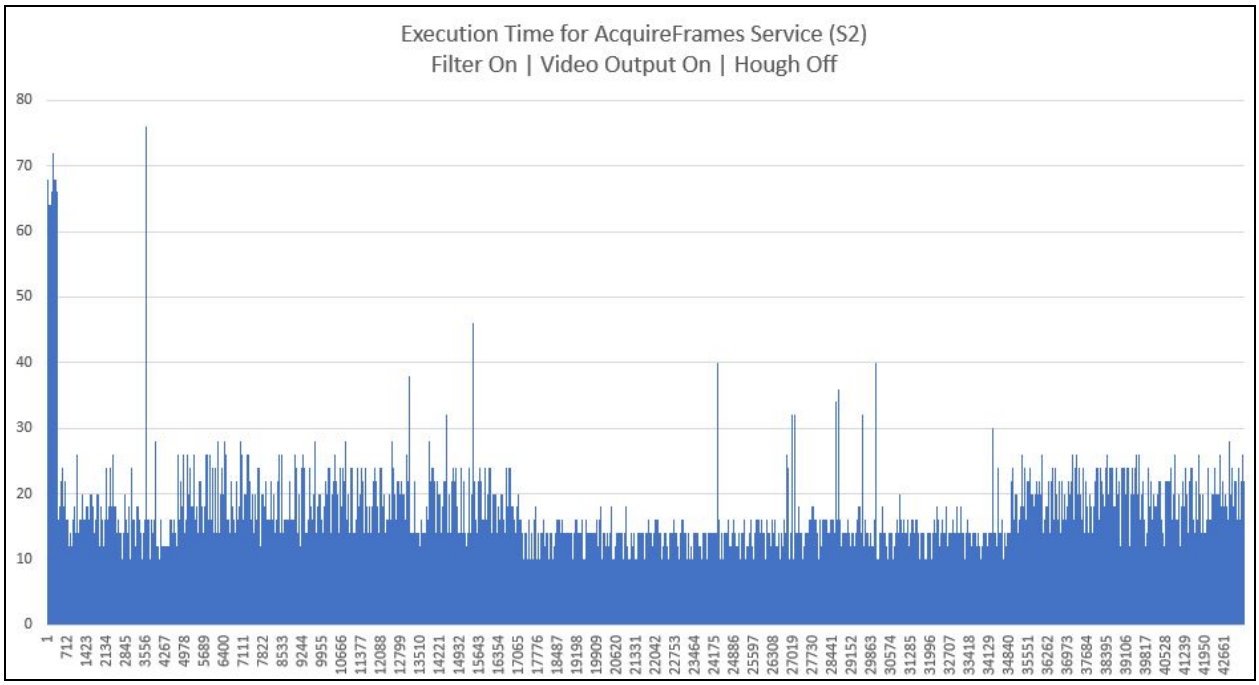

Figure 11: Execution Time for FrameAcquire S2 Service (Filter On, Video On, Hough Off)

The final case shown in Figure 12 has less noise floor compared to Figure 11 than expected based on the additional processing required from the core-shared FrameProcessing S4 service. However, it seems S4 might be causing longer and more frequent processing delays for the S2 service, as displayed when comparing Figure 12 to Figure 10. Again as before, missed frames

at such a high rate for a low-change event does not result in frames being missed by the system.

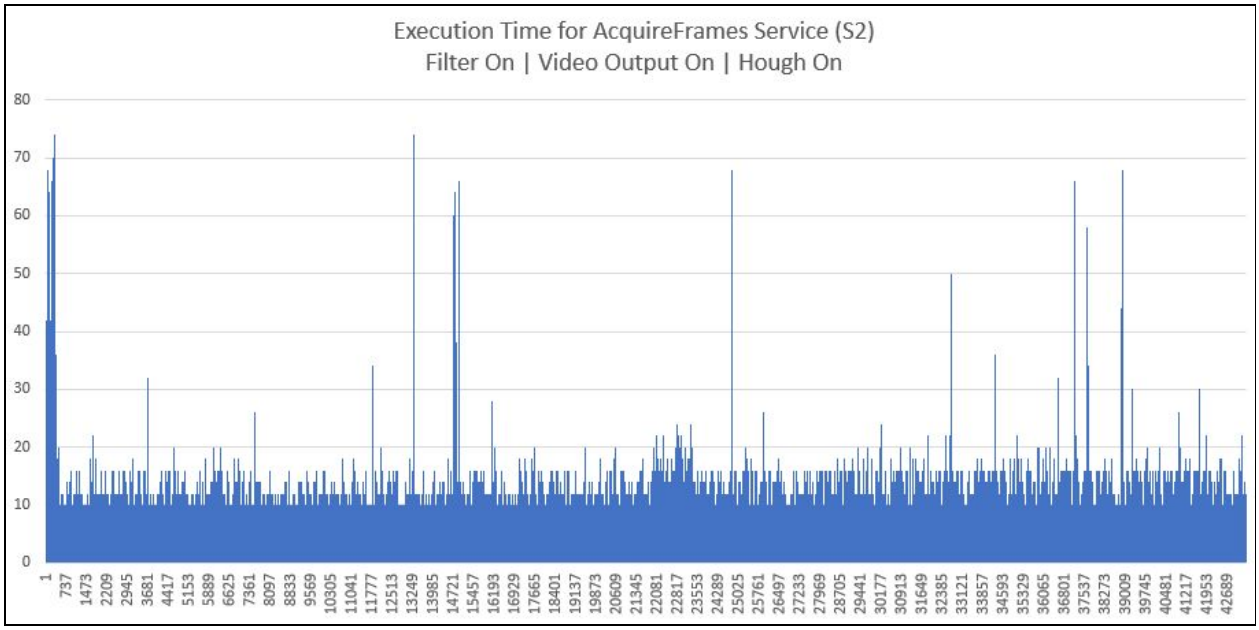

Figure 12: Execution Time for FrameAcquire S2 Service (Filter On, Video On, Hough On)

Another interesting case to compare is the FrameWrite (S5) service, shown in Figure 13 and Figure 14. From Table 6, the average execution time with output video and Hough image processing enabled is significantly higher compared to the base case where only filtering is enabled. In comparing these cases, it's interesting to note the noise floor of Figure 14 is visibly higher than the noise floor of Figure 13, which essentially falls to the average time of ~113 msec. As expected, the Hough information added to frames stored results in more high-end FrameWrite execution times, as well as an overall higher average execution time.

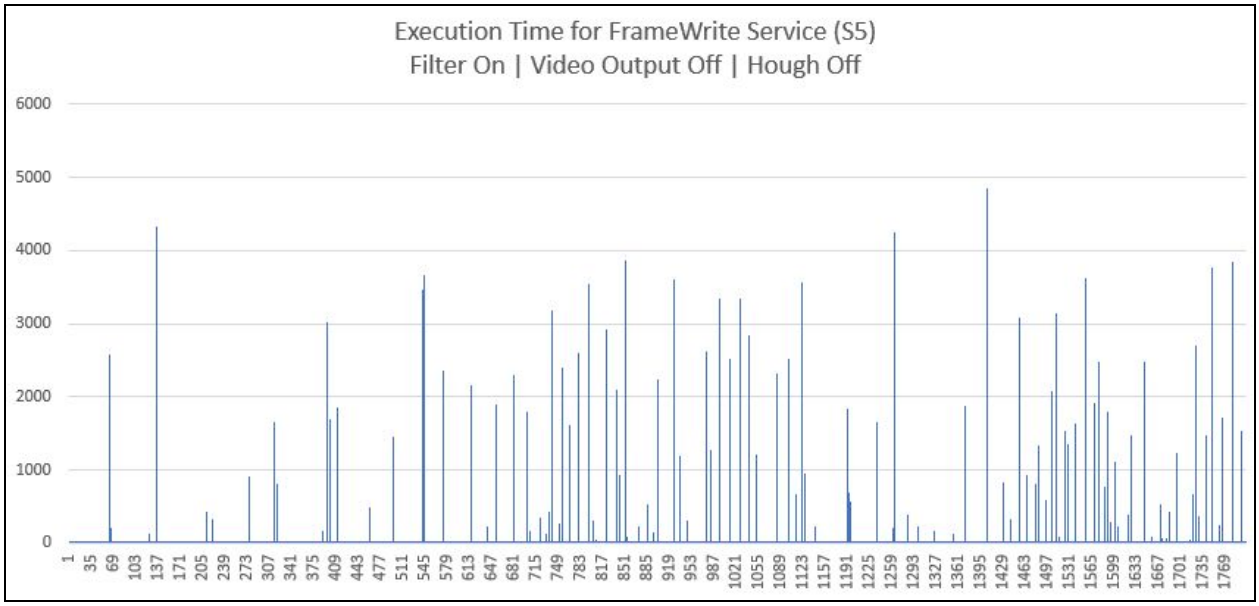

Figure 13: Execution Time for FrameWrite S5 Service (Filter On, Video Off, Hough Off)

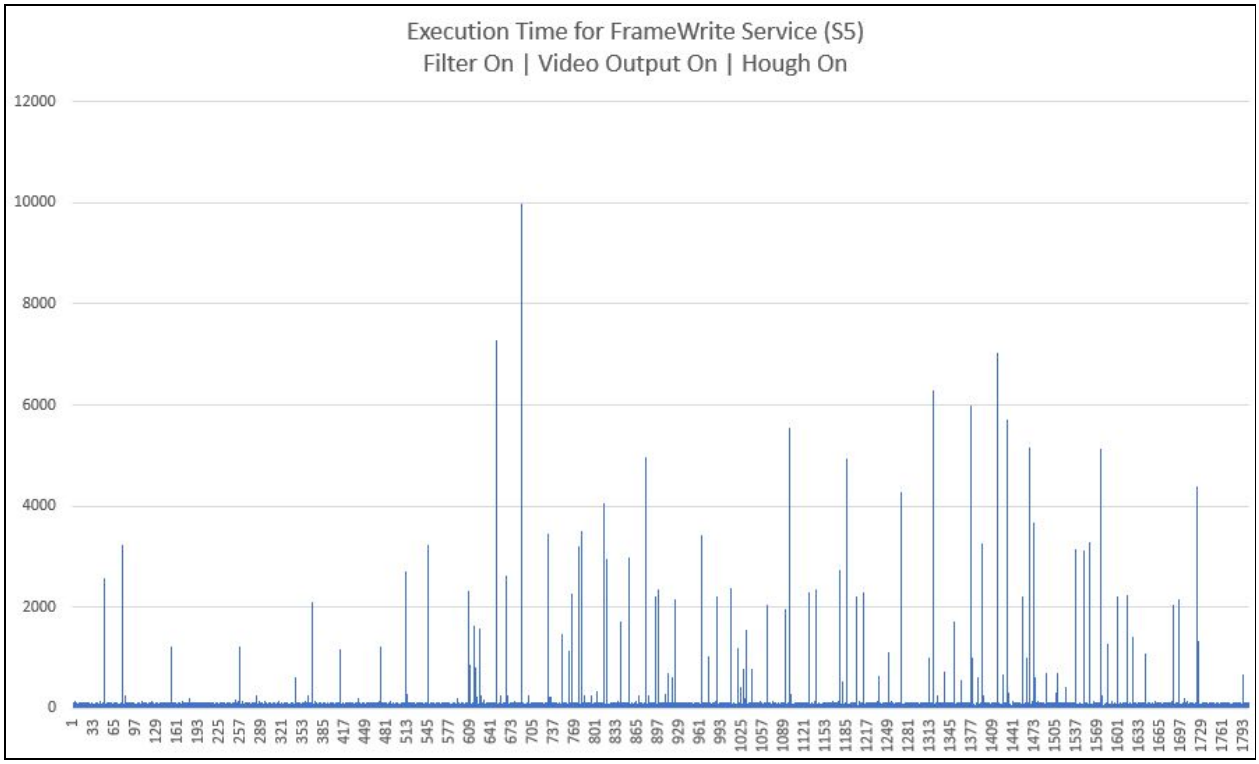

Figure 14: Execution Time for FrameWrite S5 Service (Filter On, Video On, Hough On)

System Performance and 10 Hz Digital Clock Testing

Our system was able to receive and process frames at the desired 10 Hz and save 1800 frames from a digital stopwatch, but was unable to do so without jumps in time and missed frames, and was unable to do so on a consistent basis. A semi-successful capture of 1800 frames is shown in Figure 15a and Figure 15b below, showing the start and end frames of the 1800 captured.

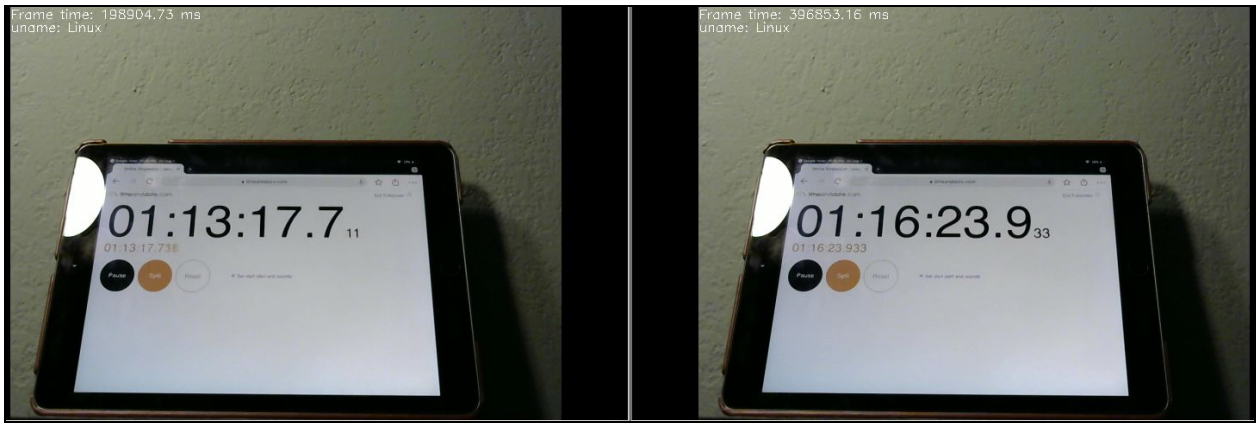

Figure 15a: Diff of Starting Frame (Left) and End Frame (Right) for 10Hz Test1 Data w/ Digital Stopwatch after 3 min (Filter Disable, Video output disabled, and Hough filter disabled)

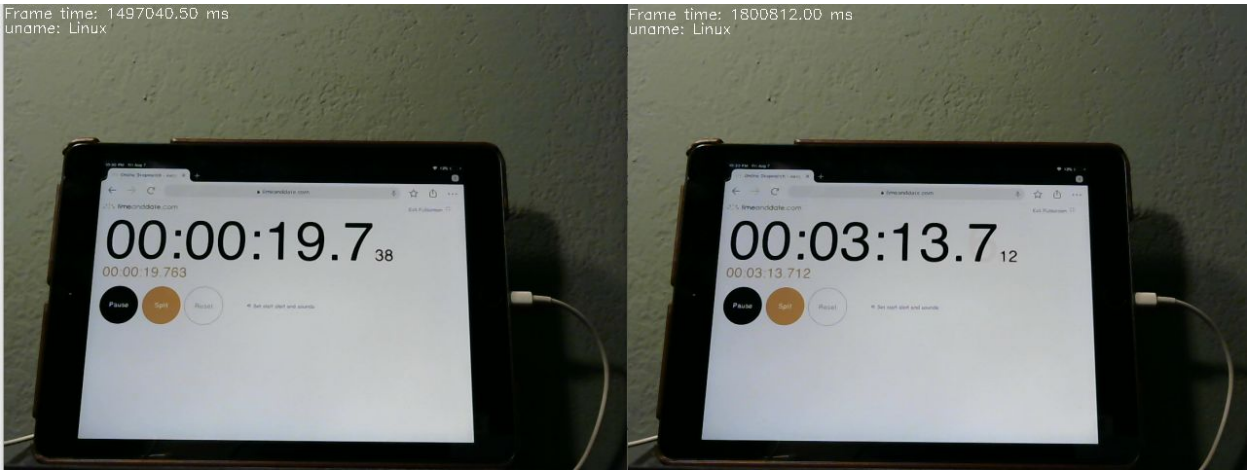

Figure 15b: Diff of Starting Frame (Left) and End Frame (Right) for 10Hz Test2 Data w/ Digital Stopwatch after 3 min (Filter Disable, Video output disabled, and Hough filter disabled)

Frames captured from this test run can be found here, but it should be noted that frames are missing at the end of the test. This is likely due to frames being overwritten and/or dropped as they get backed up in the WriteQueue between the S4 Frame Process service and the S5

Frame Write service. This can be seen from the syslog statements shown in Figure 16, which begin flooding our system logging as our system fails to keep up with the heavy amount of data to process.

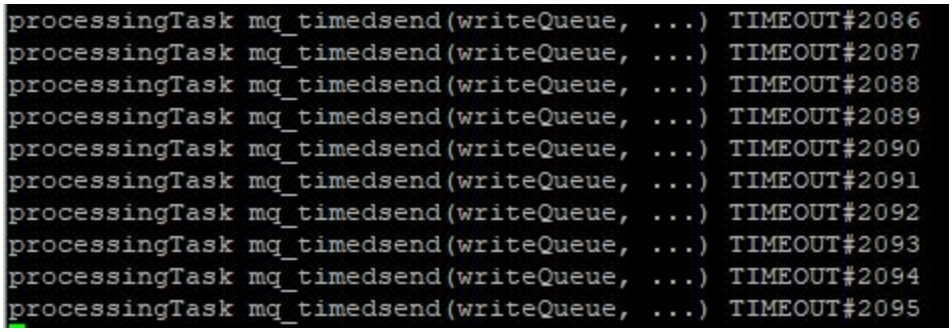

Figure 16: S4 Process Task Failing to Write Frames to Write Service WriteQueue

This was in large part due to our system not being able to write data fast enough to the system flash memory, and not properly configuring our system to separate memory IO from processing IO. Additionally, we ran out of time to properly assess the performance of the system using more powerful tool such as sysprof and gprof. It would be good to look at the write speed of flash memory, and to see how data flows from user space to kernel space and saved to system memory. Figure 17 shows the execution time of all frame processing services for the partially successful test shown in Figure 15a above.

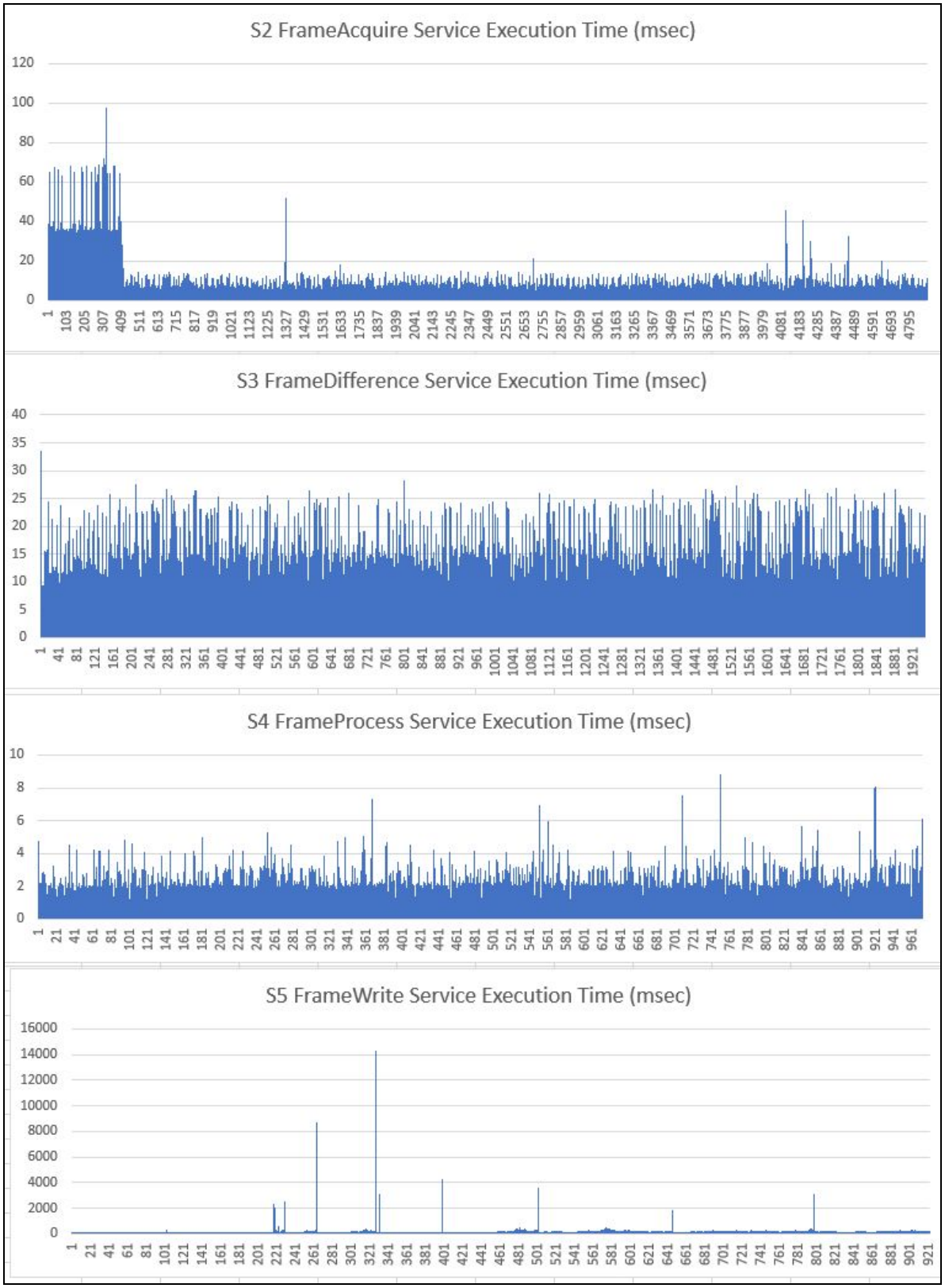

Figure 17: Execution time of partially successful 10Hz Test Run across all frame processing services

System Jitter

Jitter for the system was measured by looking at the frequency at which the S2 Frame Acquisition task was triggered, capturing the rate at which our system was receiving frames from the camera. This was done by capturing a syslog timestamp each time the service was triggered (semaphore received from the sequencer), then calculating the delta time difference between each event. The expected frequency assuming a 24 Hz signal is received is 41.666 msec, with Figure 15 showing the delta time between each event in a 2-D column graph. This graph helps to show events which were early as well as late compared against the 41.666 msec intervals. Figure 16 shows the same analyzed data, but simply the time difference from the expected 41.666 msec periodic events.
It should also be noted that this service is driven by both the sequencer semaphore being triggered at 24 Hz as well as the USB camera providing frames to our system at a rate ranging from 20-30 Hz. A high level of lighting was used to reduce camera exposure/integration time so frames could be input to the system at the fastest rate possible.

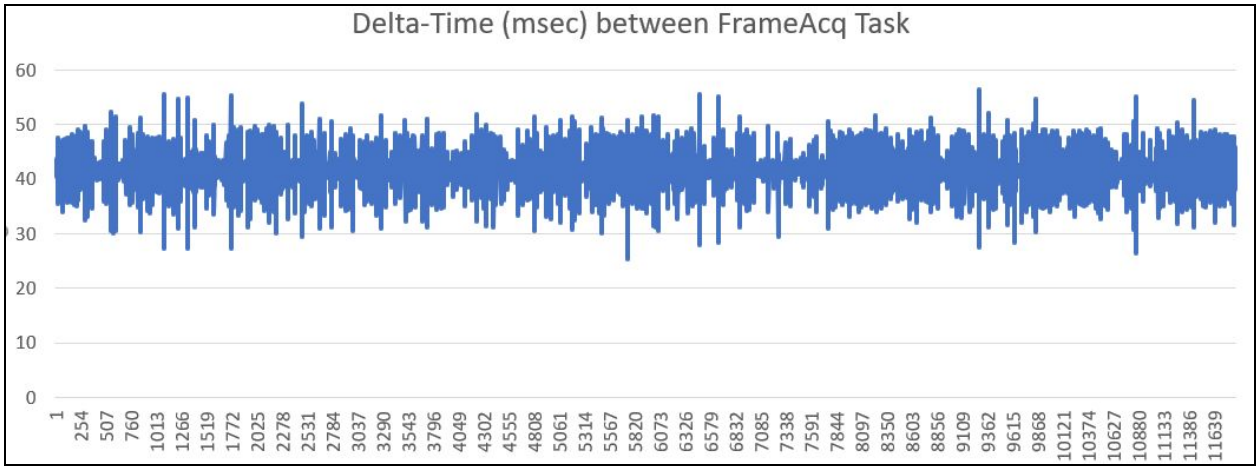

Figure 15: Time difference (msec) between start of S2 Frame Acquisition Task

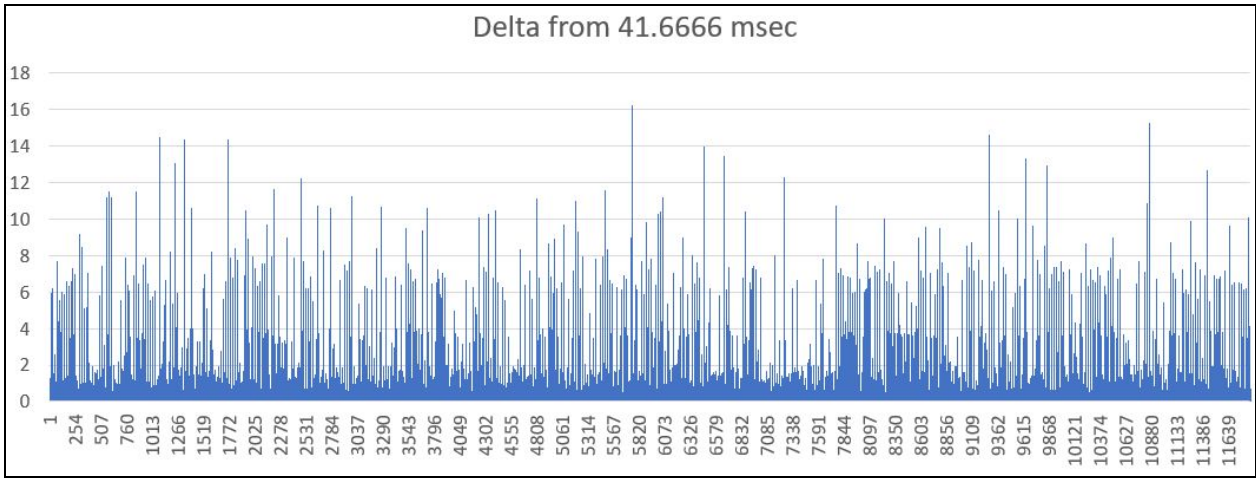

Figure 16: Execution time delta (msec) from 41.666 msec rate of S2 Frame Acquisition Task

From this data, it can be seen that system jitter ranged from 0-16 msec (0.918 average msec delta) for the 1 Hz clock change with a 24 Hz frame acquisition. Data for this analysis can be found here.

Jitter ranging up to 16 msec with an average jitter of less than 1 msec is an acceptable and expected range for our system to tolerate jittering in timing ranging in these values. Timing should be fairly predictable given that interrupts are driven from the RPi's hardware oscillator, which drives the sequencer at the desired 120 Hz base rate. Additionally, this sequencer is executing on its own separate processing core, isolated from the CPU intensive operations of the other frame handling services in the system.

Drift Analysis

Drift within the system was determined by comparing the start time of each FrameAcquire (S2) execution cycle against the expected 41.666 msec offset for each subsequent event from the start of each test. It was somewhat difficult to see the total drift or a major deviation from the expected time delta, so a delta from 41.666 for each event was used. Over the course of a 30-minute test, the AcquireFrame service saw an drift value of 0.68 msec from 41.666 msec, with the individual drift events shown in Figure 17 below. Ultimately, it would seem that due to the use of the HW oscillator provided on the RPi, remaining synced with the expected timing and triggered events is fairly consistent, and likely a longer duration test would experience a greater amount of drift over time.

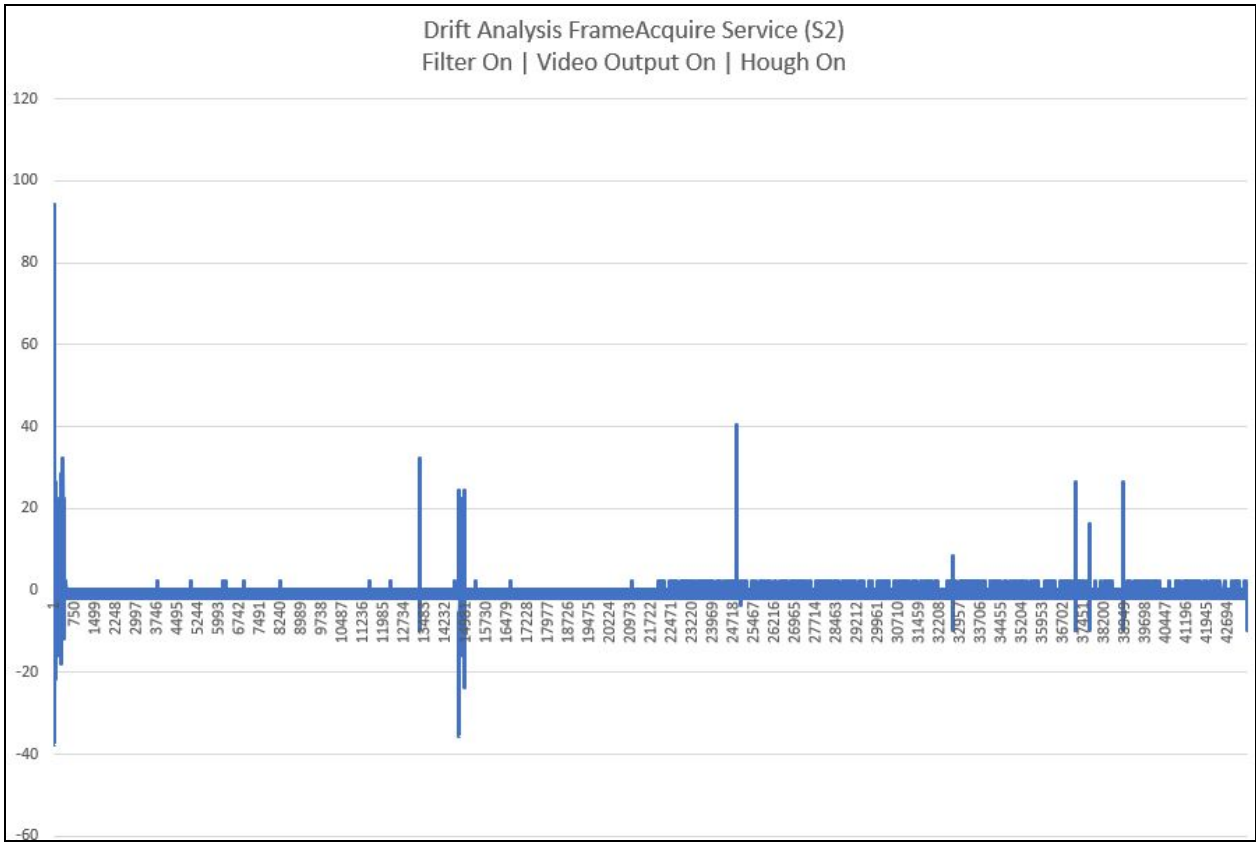

Figure 17: Drift Analysis for execution time delta (msec) from 41.666 msec rate of S2 Frame Acquisition Task for

CPU Utilization

System CPU utilization was monitoring by using the Linux htop process viewer. Figure 18, Figure 19, and Figure 20 below show the CPU utilization across the 4 RPi cores throughout a 30-minute 1 Hz Clock update rate. As seen, CPU utilization does not exceed 40% in the 3 snapshots taken and swap space experiences no spikes throughout the testing period.

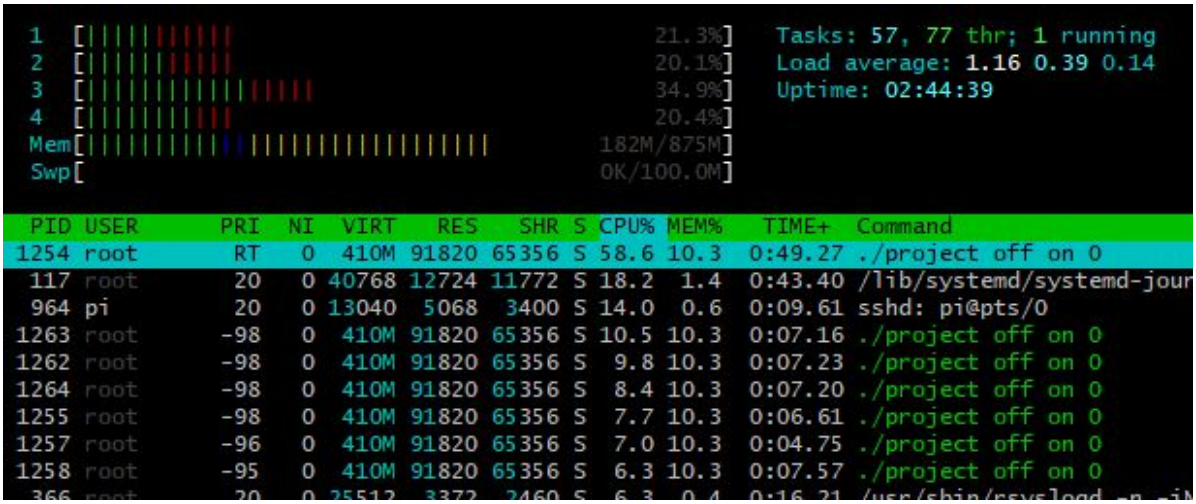

Figure 18: RPi CPU Utilization for 1 Hz Clock Difference Detection Test (Beginning)

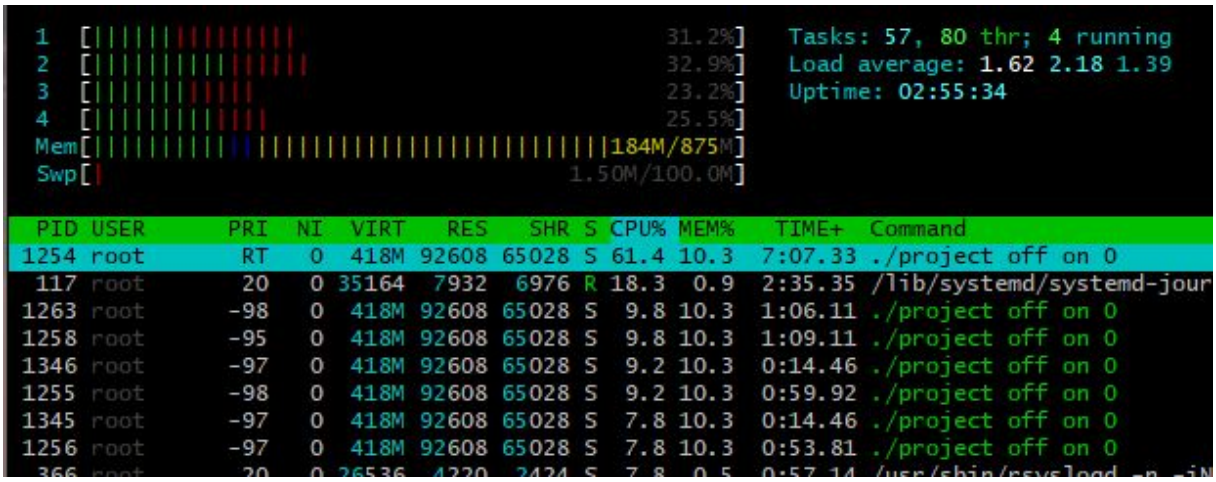

Figure 19: RPi CPU Utilization for 1 Hz Clock Difference Detection Test (Middle)

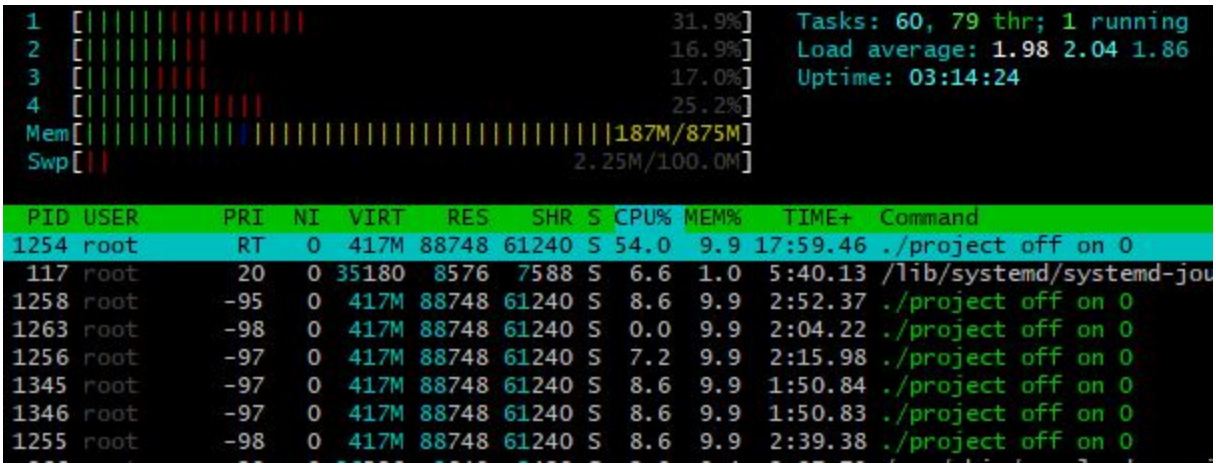

Figure 20: RPi CPU Utilization for 1 Hz Clock Difference Detection Test (End)

Cheddar Simulation Analysis

Based on the WCET defined in Table 1 - Table 4 above, the Cheddar analysis shown in Figure 21 below was generated. This shows the execution of services S1 - S5 across the 4 cores available on the RPi, with services S3 and S4 sharing core 2. Due to Cheddar being able to only receive whole numbers and not fractional values, some values from the tables above needed to be adjusted to allow the simulation to complete. These were modified to provide a worst case of execution, rounding C-values up and T-periods down. The Cheddar results show that our system is able to meet the minimum requirements for this system and will have an estimated 29% of free CPU time for safety margin. Since the WCET and shortest request period were used for RMA the safety margin should actually be 82% on average with a standard deviation of 18%.

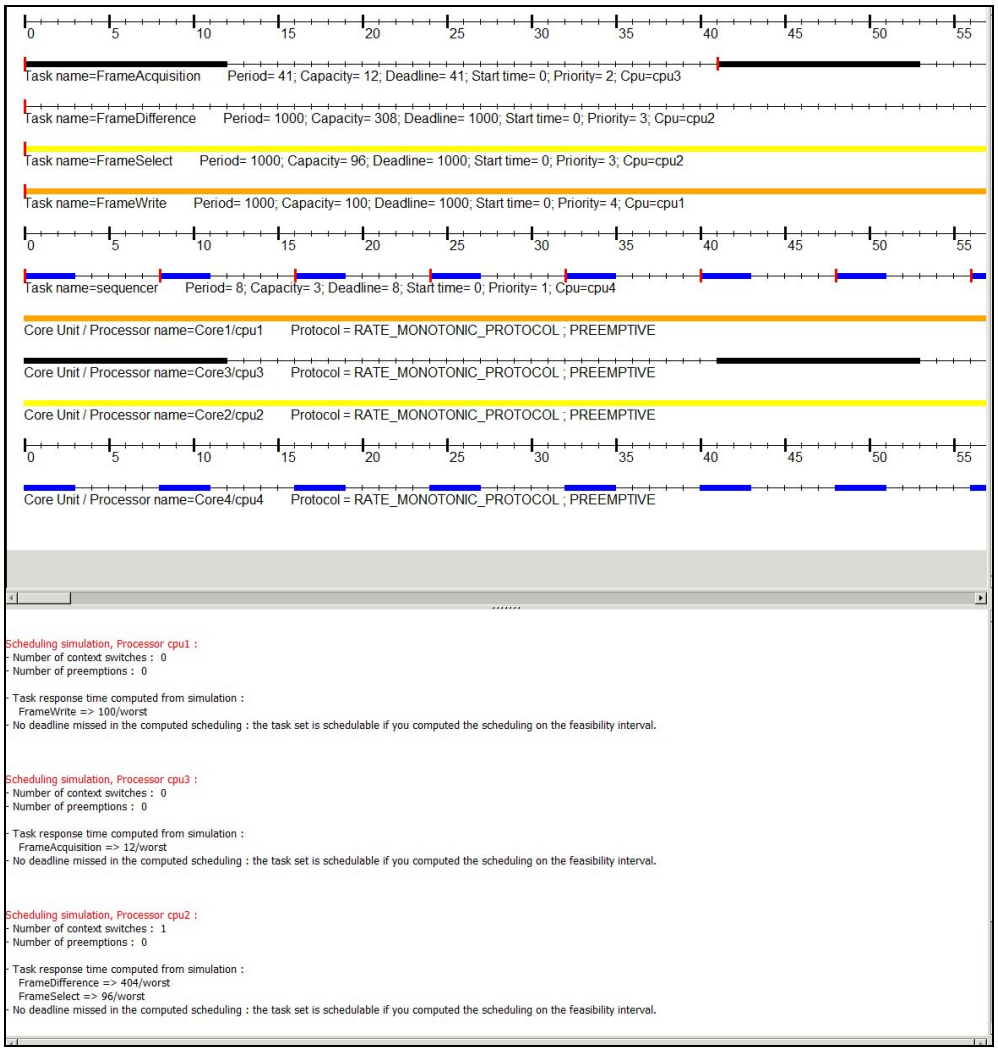

Figure 21: System WCET Cheddar Analysis for 1 Hz Digital Clock Test Case (Simulation Successful)

The stretch requirements to capture all changes from a 10 Hz stopwatch in a 3 minute timespan causes the timing requirements to tighten on this system. This adjusts the period which the frame processing services are required to complete their frame processing. Based on the WCET analysis provided in Table 3 and Table 4, it's clear that these CPU intensive processes sharing the same CPU core is not the ideal design approach. Additionally, if the estimated WCET for the S3 Frame Difference Service is true, the efficiency of this service will need to be either improved or modified so that processing within this thread is handled in a different service with more bandwidth. Figure 22 shows the predicted simulation failure result that services on Core 2 would fail to meet processing deadlines.

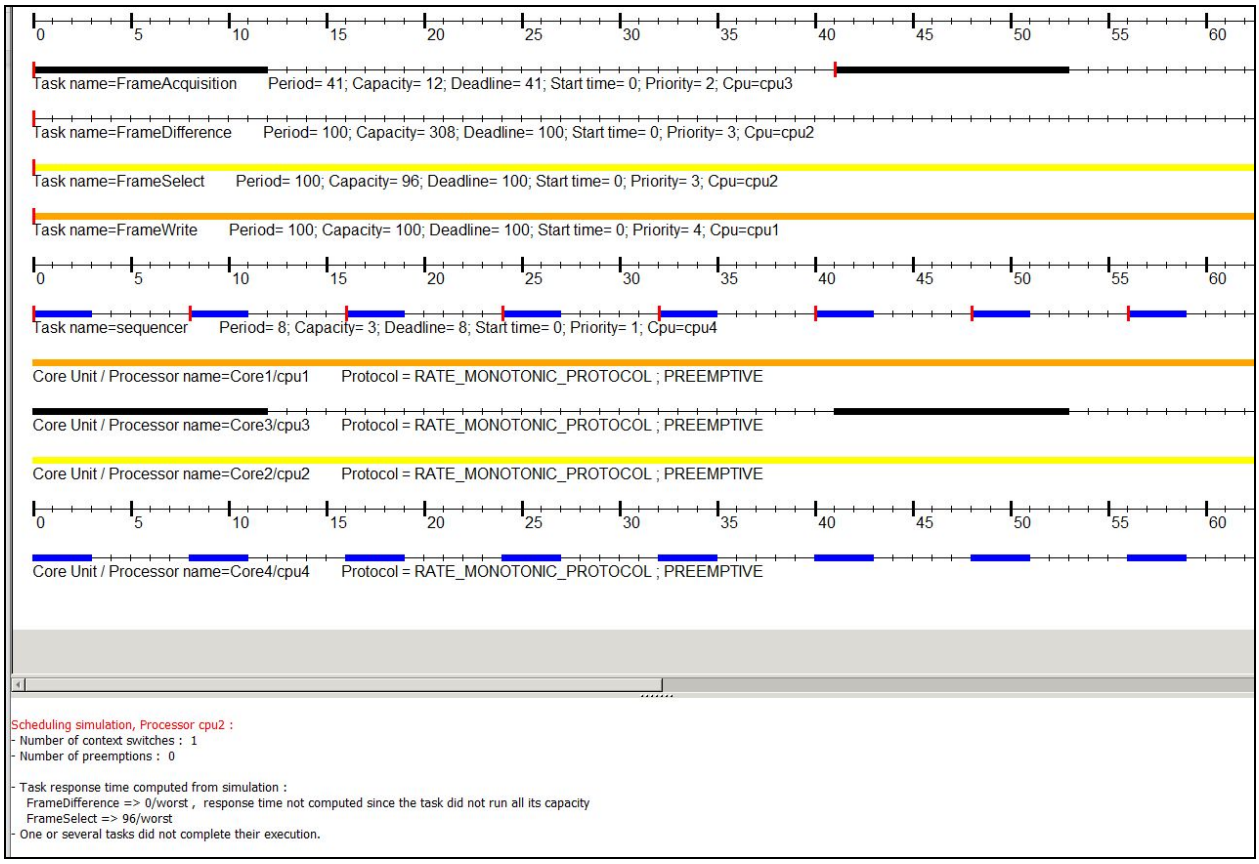

Figure 22: System WCET Cheddar Analysis for 10 Hz Digital Stopwatch Test Case (Simulation Failure)

Perhaps a more accurate representation of the system would be to use the average case execution time (ACET) for the services in question. Figure 23 shows that given the current CPU affinity and priorities set for the system service, even ACET would fail under the current design.

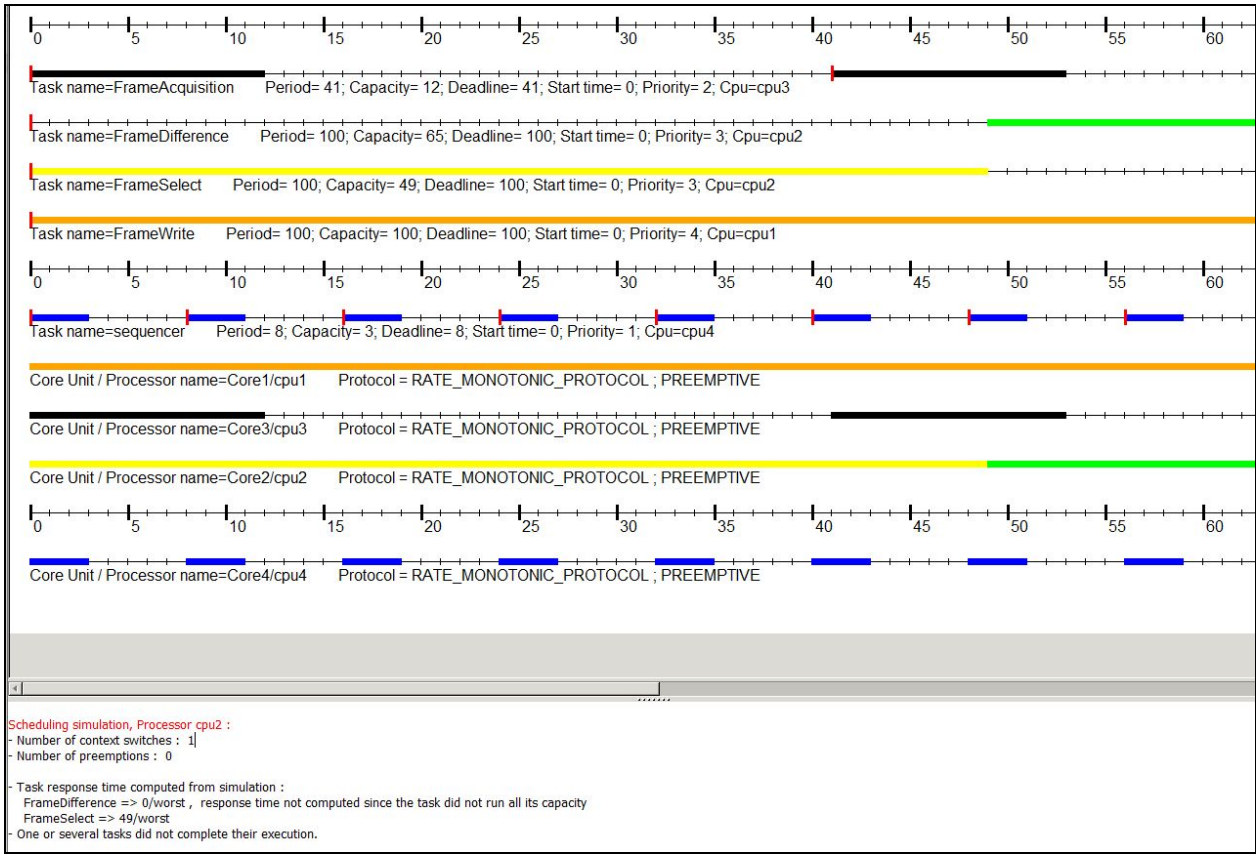

Figure 23: System ACET Cheddar Analysis for 10 Hz Digital Stopwatch Test Case, S2+S3 Shared on Core 2 (Simulation Failure)

Moving S3 (Frame Processing) to core 3 to share CPU time with S1 (Frame Acquisition) and again using ACET provides a simulation that successfully complete, as shown in Figure 24. Given this, the CPU affinity presented in Table 5 will be updated to move S3 to Core 3 prior to completing testing for the 10 Hz stopwatch testing. With this thread configuration, using ACET, we would have an estimated 42% of free CPU time for safety margin.

Figure 24: System ACET Cheddar Analysis for 10 Hz Digital Stopwatch Test Case, S1+S3 Shared on Core 3 (Simulation Successful)

We'll also note that improvements can still be made to the system (Single User Mode, using Neon Instructions) to improve overall system performance, and will be investigated in the coming week. Additionally, all 1 Hz testing data was captured with the filtering feature and syslog data being captured, as well as video being recorded from the saved frames. Performance can be improved further by adjusting these features as well.

Conclusion

Using OpenCV 3.X with message queues and circular buffers was a challenge because the Mat structure is similar to a C++ smart pointer with automatic memory management. Each time the class object is passed, it increments its internal self-reference count and will only destroy itself when the reference count is zero. Additionally, the class doesn't contain the pixel data, instead it just contains a pointer to a dynamically allocated block of memory containing the pixel data. This block of memory gets destroyed when the Mat object's reference count is zero. Unfortunately when a Mat object is inserted into a Linux message queue, the object's self-reference count is NOT incremented so when the object goes out of local scope (end of loop) the pixel data is destroyed. Consequently, to pass the pixel data between threads it must be copied to a dynamically allocated memory block to be freed by the queue reader when it's

finished with the data. Dealing with C++ smart pointer memory management was also a challenge when implementing the circular buffer for the same reasons. Message queues and circular buffers were primarily picked because they were a part of the course materials; if not for this, using standard C++ containers and OpenCV's cv::Ptr<Mat> type would have been the preferred method.

A significant portion of our time was spent tuning the frame difference thread's parameters to accurately detect each second hand tick. It was a challenge to identify the best threshold value to reject false positives and detect motion of the black second hand when its motion was less detectable (i.e. while moving in front of other hands). It was also a challenge to find the right lighting conditions to enable higher camera frame rates. In hindsight simply periodically detecting a single tick over a small window would have been an easier approach because we wouldn't have had to worry about hand obscurities / corner cases.

In the end, our system was successfully able to consistently and repeatedly receive, analyze, process, and write unique frames viewing an analog wall clock updating at a 1 Hz rate. Using syslog print statements, we were also able to determine the average execution time for the various services within our system.

References and Resources

1. 4-part video series Professor Siewert provided in the Lightboard Studio Rough Cuts
    a. http://ecee.colorado.edu/~ecen5623/ecen/video/lecture/Lightboard-Studio-rough-cuts/L-N9.1-Final-Project-Services-Decomp-Does-Not-Work.mp4
    b. http://ecee.colorado.edu/~ecen5623/ecen/video/lecture/Lightboard-Studio-rough-cuts/L-N9.2-Final-Project-Services-Decomp-Not-Fault-Tolerant.mp4
    c. http://ecee.colorado.edu/~ecen5623/ecen/video/lecture/Lightboard-Studio-rough-cuts/L-N9.3-Final-Project-Services-Decomp-Somewhat-Fault-Tolerant.mp4
    d. http://ecee.colorado.edu/~ecen5623/ecen/video/lecture/Lightboard-Studio-rough-cuts/L-N9.4-Final-Project-Services-Decomp-Best.mp4

Appendix

- Links to our project code can be found at the following links:
    - Project GitHub: https://github.com/ibelingb/ecen5623_final_project.git
    - 1Hz Analog Clock Code
        - GitHub: https://github.com/ibelingb/ecen5623_final_project
        - Google Drive: https://drive.google.com/file/d/1lUH1E0oUsM9kn_reyI1PQdCqN_Icbgxy/view?usp=sharing
    - 10Hz Digital Clock Testing Code (Branch from Master)

        - GitHub: https://github.com/ibelingb/ecen5623_final_project/tree/10hz_testing
        - Google Drive: https://drive.google.com/file/d/1GPrxcXU1_IqYRjKYyvpaX5b1OhNdJyYq/view?usp=sharing
        -
- Logging data as well as WCET and ACET analysis for services for the Software Services section can be found here.
- Logging data, captured frames, and the recorded video for the 1Hz Analog Clock sampling for the various application configurations is found below:
    - Filtering enabled, video generation at runtime disabled, with Hough filtering disabled is found here.
    - Filtering enabled, video generation at runtime enabled, with Hough filtering disabled is found here.
    - Filtering enabled, video generation at runtime enabled, with Hough filtering enabled is found here.