11-Aug-17

ECEN 5623 – Real Time Embedded Systems

Summer 2017

TIME-LAPSE

Project Report

Jeet Baru

Email: jeet.baru@colorado.edu

ID Number: 107189037

Introduction

The main aim of the project is to generate an accurate time lapse video, thereby applying concepts of real time scheduling learnt in class. Time lapse is popular term for a fast motion video. Time lapse photography is a technique whereby frames are captured at a very small frame rate and is played at faster rate hence time appears to move faster hence lapsing.

The specialty of it is that the video is of slowly changing object/scene captured and played so that clear changes can be noted quickly. Hence time lapse videography has found application in recoding slow changes that would have been otherwise difficult to notice. For example, time lapse videos have been used to monitor phenomenon like germinating of seeds, blossoming of flowers, formation of clouds etc. Time lapse videos have also found its application in movies hence now have become a popular feature of this generation of smart phones.

Time lapse requires the frames to be captured at a specific and constant frame rate. Also the processing on the frames must be performed before acquiring a new frame, or at least must be accounted for without hindering or delaying a new frame capture. This deadline based nature of this application forms an ideal project to apply and test concepts about real time scheduling.

The development board I have used for the project is the BeagleBone Black which has a ARM cortex A8 processor, 512 MB RAM, 4GB on noard memory. It also has USB host for connecting Camera, Keyboard and Mouse. It has a mini USB client for power and display, HDMI out for display and Ethernet connectivity. I had Debian Linux loaded onto the board. I feel this board provided just enough firepower for my application. I used POSIX API to be able to schedule tasks and manage resource allocations and have deterministic timing. I have also used a mouse, keyboard and display to setup the native embedded environment. Also I am using a Logitech C200 camera to capture images. I am also using the OpenCV API to capture, process and store the time lapse video.

Functional Requirements

There are 6 main functional requirements of the Time lapse video system that I have proposed. I have listed and described them before:

1. **Camera Capture**: Capture frames of the target at a specific frame rate at a resolution of 640 x 480. The resolution of 640 x 480 is high enough for any standard TV monitor. The frame rate selected is 1 Hz. Hence the camera shall capture a frame every 1 sec based on REALTIME TIMER used from the POSIX API
2. **Save Images in its raw form as .ppm**. I have saved image as P6 encoded ppm. These images are uncompressed binary representation of the captured frame stored as .ppm file. The size of these files is large as there is no loss of information. PPM files form an integral intermidiate step for processing before saving them as a more efficient format.
3. **Embed important information of device, the user and the timestamp into the ppm file**. I have also embedded the output from system command 'uname' about, the user running the application, the device on which the application is being run and the Timestamp. The timestamp forms a nice way to ensure that the captured images are at an interval of 1 sec.
4. **Compress the captured images as .jpg** to reduce the amount of space used. My implementation also incorporates a feature to compress the processed ppm file and save it as more storage efficient jpg file.
5. **Generate a video from the captured frames**. Finally, I have implemented a thread that after compression generates a video from the captured frames and store it as a .avi file. The codec used for this purpose was DIVX. With this added functionality we are able to observe and verify how the code was able to capture frames with high accuracy.
6. **Store the images and the video off board in an SD card**. I have added a functionality of offloading the generated images and video off chip in the SD card. This helps to save space on chip and hence allows the application to run for a longer period breaking the shackles of storage. BeagleBone Black has an onchip memory of just 4 GB which becomes insufficient after installing OpenCV libraries. Hence I am able to record long videos without worrying about storage.
7. **Run the timelapse at a higher frame rate.** Running the time lapse video recording at a higher frame rate puts the application to a real test. I have set a requirement of being able to achieve at least 5 times faster speed than the initial 1Hz. Trying to achieve this goal requires the code to be optimal with least lag.
8. **Save CSV file of times for further analysis.** I have saved csv file of timestamp to further analyse the software in Microsoft Excel

I have performed timing analysis on each functionality and used Rate Monotonic Policy to determine threads and its schedule and making it almost deterministic.

Real Time Requirements

There are 2 hard real time service requirements and 1 best effort service. I have described each of them below:

- Capture Image: This service is mainly concerned with monitoring timing and capturing images at an exact rate of 1 Hz. I have used the OpenCV API to capture the image as an IplImage*. This is defined globally and hence is available to be used by the save image thread to save it. A track of time is kept to ensure that the next frame is captured only when 1 sec of a difference is achieved between previous frame and next frame. Hence I poll for the right time and grab the frame
- Save Image: This service saves the captured frame as .jpg file. It is also involved in modifying the file and hence embedding the information of user, device and timestamp. The generated IplImage* is saved as jpg using cvSaveImage()

The above two services form the crux of the hard real time requirements. It is essential that the above two services together take up less than 1 sec to run. Hence instead of having a deadline for each of the above threads, in my implementation I have set common deadline of 1 sec. The above 2 functions are made to run one after the other keeping the contents of each functions loop critical by using semaphore locks. The extended goal for the project requires us to try and run our application at 10 Hz as compared to current 1 Hz. I have tested my application for 1 Hz, 5 Hz and 10 Hz requirement as described later.

- PPM and Generate Video: The third thread in my implementation involves generation of ppm images from compressed file and add timestamp information to the file. It also loads frames from saved ppm files and adds them to the video file. The video file is saved in avi format using the DIVX codec. I have put deadline on execution time of this thread because it is not integral to the basic functionality of my application. I have written code such that is thread runs only after the first 2 services have completed execution. Thus this service acts as a post capture processing delay before the user has the final time lapse video.

One major constraint for my application to run on the BeagleBone was insufficient memory and slow write speeds for writing to SD card. PPM files are raw image data hence each frame requires atleast 1 Mega Byte of storage space. Thus saving 2000 frames impossible on the BeagleBone. I hence decided to save compressed images first and keep a log of timestamps and save ppm files once all images are captured.

I had to take the decision of excluding the PPM and Video generation thread from 1 sec deadline as the time taken to save as video and compress the images was too much. This made meeting the 1 sec deadline difficult. Also compression and video generation could be done later and was not integral to capturing frames in 1 sec.

Functional Design Overview

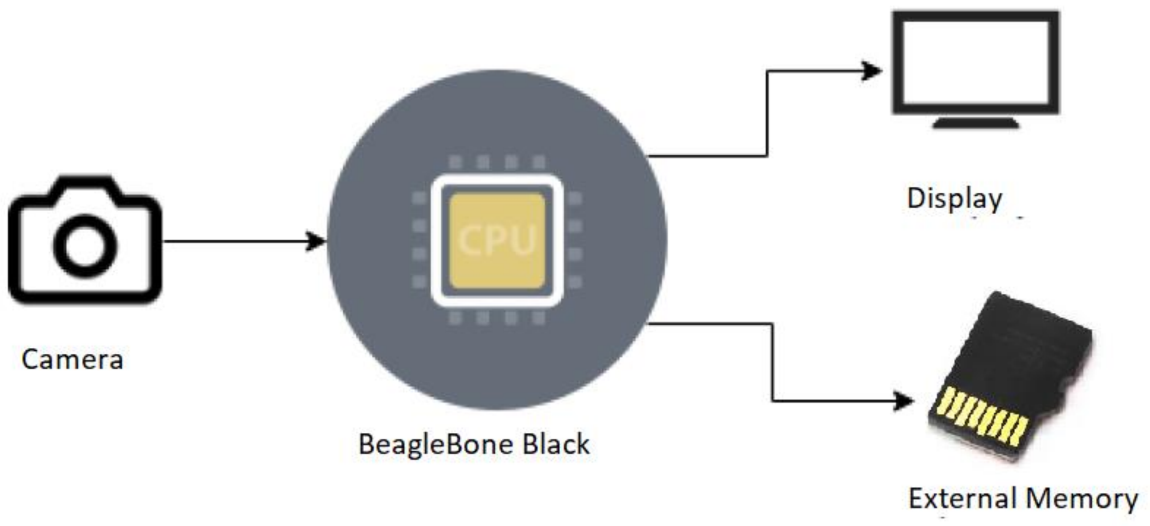
Hardware Design:

I have used the following hardware components for the implementation:

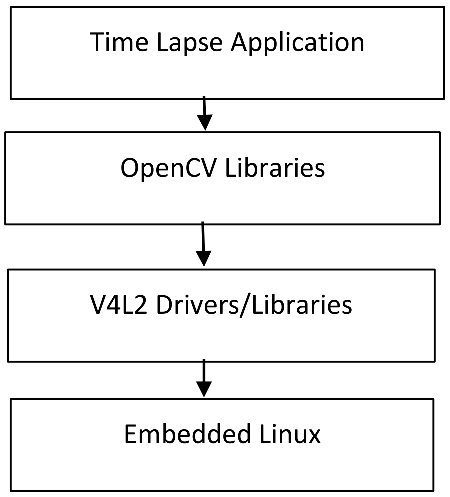
- BeagleBone Black
- USB VGA Camera Logitech C200
- Embedded Linux OS made real by use of POSIX extensions
- OpenCV libraries for image processing
- External SD card to offload images and video
- Keyboard, Mouse and Display to work on the embedded system natively

My project makes use of the Embedded Linux OS to make my system deterministic so that most of the deadlines can be met for my application to work smoothly. Using the standard POSIX thread API, I achieve this determinism and partial parallelism by making use of threads. BeagleBone provides for an ideal embedded environment. It has the ideal hardware specifications that make this application a challenge by not boasting an overkilling hardware. Having known the deadline for my real-time system to be 1 sec and offloading inessential functionality, I hope to have the various system components work in sync and meeting the deadline.

Hardware Design Block Diagram:



Software Design:



Drawn above is the Layer diagram of software system. Above the OS layer there is an installed layer of V4L2 libraries and Camera drivers. This allows for functioning of attached camera hardware. On top of this layer is the OpenCV libraries that provide for functionality like image capture, video write, image compression. My Application is on the topmost layers closest to user and makes use of the OpenCV APIs to achieve the goal.

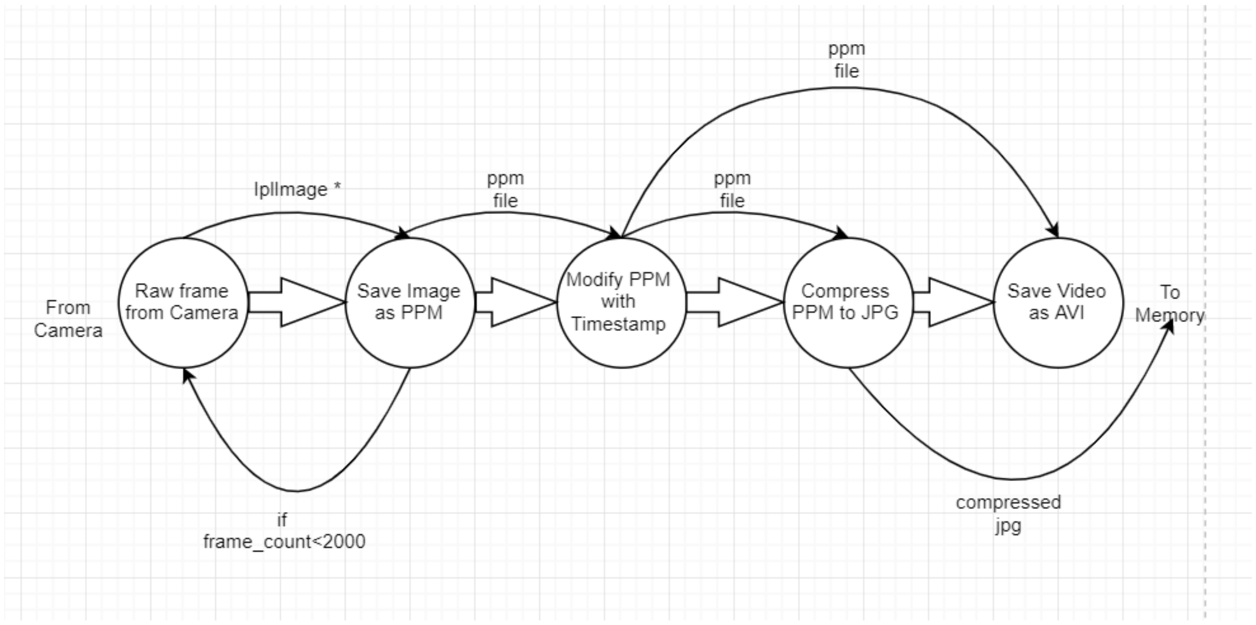
Application Design:

As described earlier, in my application I have made use of 3 threads, Image capture, save image and compress video. Of these image capture and save image threads have a combined deadline of 1 sec. That is both these threads must complete 1 iteration of capturing and saving image in 1 second. The third thread however I have kept as best effort and has no deadline and is executed at the end.

Application Software Design Explanation: The order of events explaining the functioning of my application and the logic is as follows:

- First all three threads are created and thread 1 has highest priority followed by thread 2 and thread 3.
- Thread 1 (image capture) locks the semaphore and captures 1st frame records the timestamp and releases semaphore.
- Thread 2 (save image) after waiting on the semaphore saves the image as a jpg file. It releases semaphore for thread 1 again.
- Now the thread 1 records time and continues monitor it till 1 second has elapsed. It then captures the image and passes control onto thread 2.
- This continues till all frames are captured.
- Once all frames are received Thread 3 (compress video) is executed. Here jpg images are read and ppm files are saved. Also files are manipulated to save information of timestamp in the ppm file. Finally, all images and videos are saved onto the external SD card.

A state flow diagram explaining functioning of each thread and hence the functioning of the application as a whole is described below:



Challenges Faced

Storage Space in Beagle Bone

BeagleBone is relatively a less powerful device in terms of Memory. Considering that it has Linux running on it, on board memory of 4 GB seems insufficient. Installing OpenCV libraries took up more space leaving me with less than 200 MB to work with. I changed my initial plan of saving ppm files to on board storage to saving them on the SD card.

Randomly varying write time for SD card

Writing files to SD card made took random times. Many files satisfied my required deadline of 1 sec but I observed frames to take randomly large time to write to SD card. As a possible solution I changed the IO scheduler for Linux from default cfq to deadline. This improved situation for frames with small delays in writing but the frames that gave large delays in writing still persisted. Another solution I implemented was a custom function to write file to the SD card. I took this approach to reduce the number of writes to a minimum of 2 and thereby improving the response. On testing I observed that the writes for inbuilt OpenCV function were comparable to the writes for my function hence not improving time.

Hence I decided to write compressed files to the system memory giving more predictable write times. Also I stored timestamps associated to each frame in RAM and used it later while modifying the then generated ppm files

Large time required for storing PPM and writing Video file

I integrate video writer in my thread itself hence adding functionality to my application. The time take to write frames to the video was also significant hence meeting the 1 sec deadline was difficult. Hence I chose to decouple writing ppm and generating video thread from the thread that captured frames and stored the compressed image onto the SD card.

Real Time Analysis

To verify the thought approach, I proceeded with calculation of execution time for each thread. For this I calculate the average execution time for each thread, This, was done by sampling times for each execution of while loops, for frame capture and saving ppm files.

Profiling each of the service threads individually for one execution, I determine the execution times and the longest path through the code represents the WCET. I used the timespec structures to store the timestamp in seconds and nanoseconds. Creating such structures for start time, stop time and the difference, I placed the clock_gettime function at the start of the service we want to profile by using the CLOCK_REALTIME macro. I also placed this function at the end of the service and the difference in the two timestamp values gave me the execution time.

The average execution times were calculated for **2000** samples

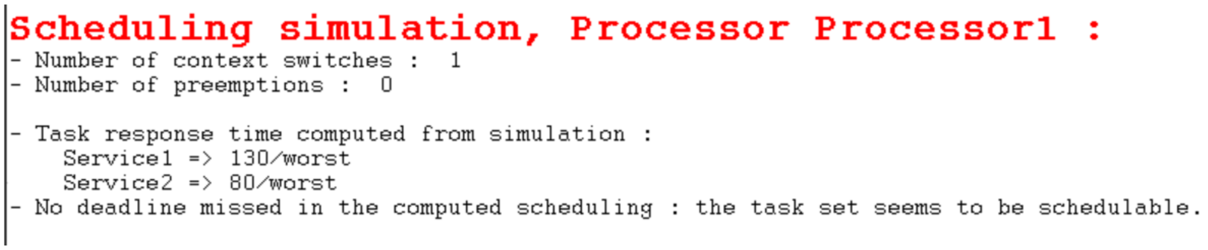
- The execution time for thread 1, capturing images is **32ms**. In this this thread I just captured a frame as IplImage*.
- The execution time for thread 2, writing ppm files is **64ms**. In this thread I saved the obtained frame as a compressed jpg on the device, I also keep a log of timestamps so as to add them to the ppm file generated later. Since the files were being written to the memory these times were deterministic.
- The execution time for thread 3, was **1.63 seconds**. In this thread I create a raw image file from the saved jpg and add timestamp and other device information to it as headers. In this thread, I also generate a video and add frames to it. Since the writing for this thread was on the SD card, the timings varied by great amounts. The average execution was 1.63 secs which is beyond the requirement of 1 sec. Hence I decoupled the thread and make this module run independently as best effort service.

I have considered the worst case execution time for each of the above average execution times to be 20 ms more. This assumption I feel is reasonable as the variance in recorded times while profiling wasn't bigger than 20 ms.

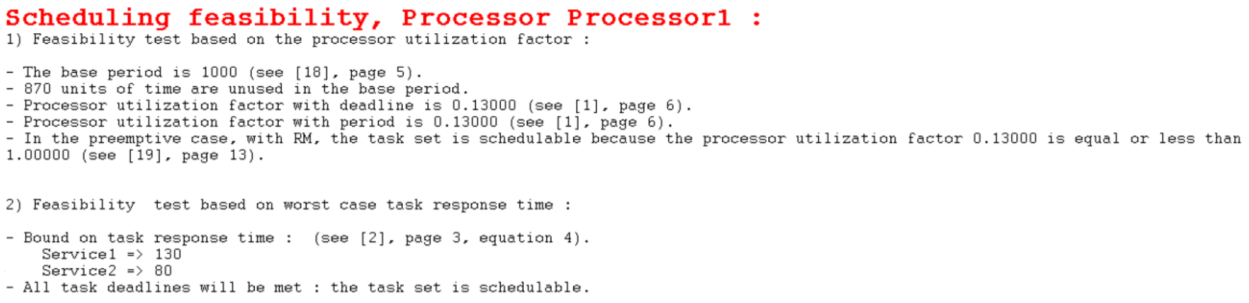
As described before only thread 1 and 2 have a combined deadline of 1/(Required Frequency). I ran test for Required Frequency of 1 Hz, 5 Hz and 10 Hz.

For 1 Hz:

I have 2 real time services S_1 and S_2. Completion Time C_1 = 50ms, Period and Time to deadline are $T_1 = D_1$ = 1 sec. S_2 has C_2 = 80ms and $T_2 = D_2$ = 1 sec Applying Rate monotonic policy, I simulated this condition on cheddar and obtained the following results.



```
Scheduling simulation, Processor Processor1 :
- Number of context switches :  1
- Number of preemptions :  0

- Task response time computed from simulation :
    Service1 => 130/worst
    Service2 => 80/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.
```



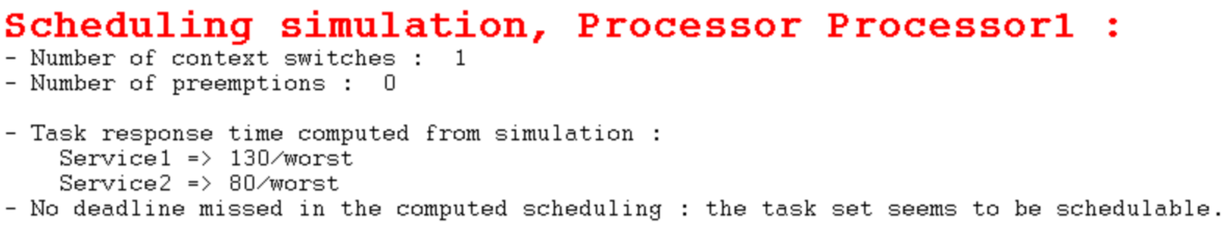
```
Scheduling feasibility, Processor Processor1 :
1) Feasibility test based on the processor utilization factor :

- The base period is 1000 (see [18], page 5).
- 870 units of time are unused in the base period.
- Processor utilization factor with deadline is 0.13000 (see [1], page 6).
- Processor utilization factor with period is 0.13000 (see [1], page 6).
- In the preemptive case, with RM, the task set is schedulable because the processor utilization factor 0.13000 is equal or less than
1.00000 (see [19], page 13).

2) Feasibility  test based on worst case task response time :

- Bound on task response time :  (see [2], page 3, equation 4).
    Service1 => 130
    Service2 => 80
- All task deadlines will be met : the task set is schedulable.
```

We see above that the service is both schedulable and feasible and very safe. It gives a CPU margin of 870 ms. Polling for the deadline is a bad approach as we may end up wasting the CPU as a resource. I decided to make the CPU enter sleep mode for this time hence saving power.

Next we analyze for 5 Hz.

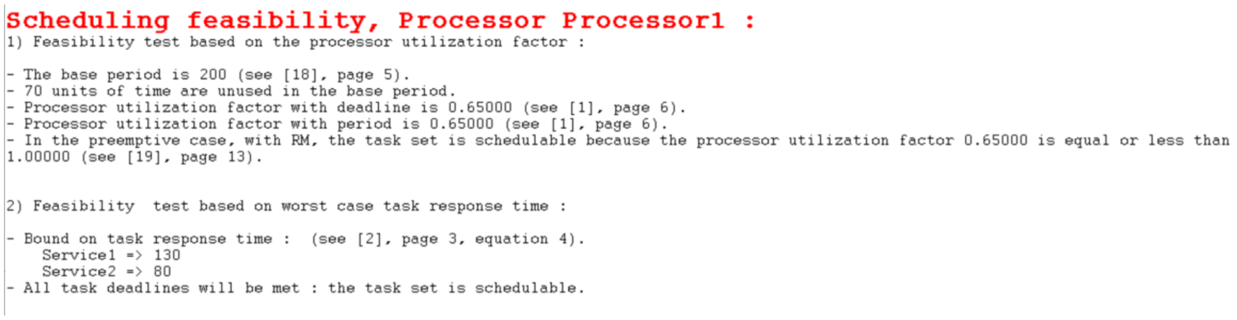
For 5 Hz:

The services in this case have the same Completion Times C_i. But the Deadline and Period of request has reduced significantly to 200ms. Therefore, $T_1 = D_1 = T_2 = D_2 = 200$ms. The simulation in cheddar setting RM as policy yielded following results:



```
Scheduling simulation, Processor Processor1 :
- Number of context switches :  1
- Number of preemptions :  0

- Task response time computed from simulation :
    Service1 => 130/worst
    Service2 => 80/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.
```

The feasibility and schedulablility test gave the following results:



```
Scheduling feasibility, Processor Processor1 :
1) Feasibility test based on the processor utilization factor :

- The base period is 200 (see [18], page 5).
- 70 units of time are unused in the base period.
- Processor utilization factor with deadline is 0.65000 (see [1], page 6).
- Processor utilization factor with period is 0.65000 (see [1], page 6).
- In the preemptive case, with RM, the task set is schedulable because the processor utilization factor 0.65000 is equal or less than
1.00000 (see [19], page 13).

2) Feasibility  test based on worst case task response time :

- Bound on task response time :  (see [2], page 3, equation 4).
    Service1 => 130
    Service2 => 80
- All task deadlines will be met : the task set is schedulable.
```

We see that the services are schedulable and feasible having a CPU utilization factor of 65%. Hence even at 5 Hz the application can run safely.

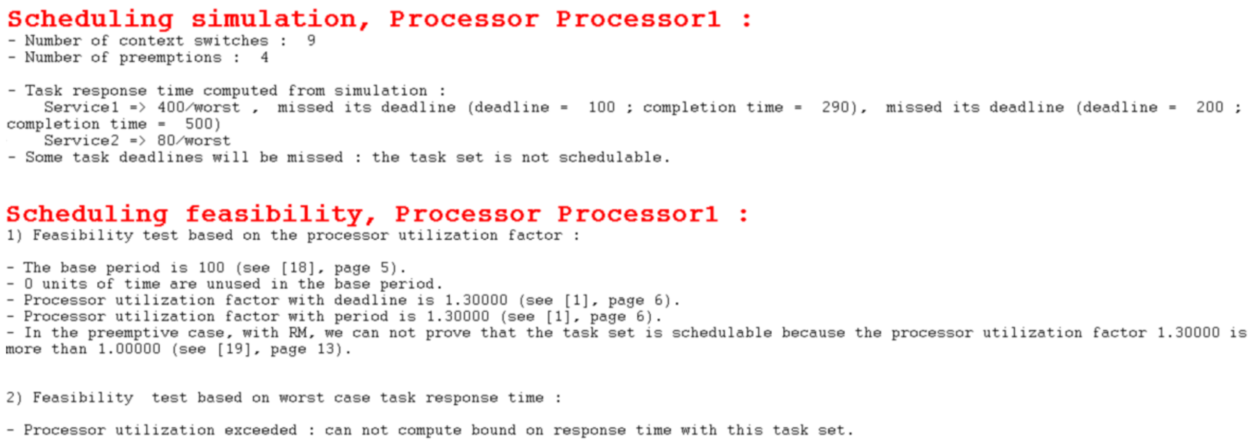
For 10 Hz:

The Completion Times are still same but the deadlines and periods are now mere 100ms. We have

For S_1 : $C_1 = 50$, $D_1 = T_1 = 100$ms

For S_2 : $C_2 = 80$, $D_2 = T_2 = 100$ms

On running the simulation and feasibility test we observe:



```
Scheduling simulation, Processor Processor1 :
- Number of context switches :  9
- Number of preemptions :  4

- Task response time computed from simulation :
    Service1 => 400/worst ,  missed its deadline (deadline =  100 ; completion time =  290),  missed its deadline (deadline =  200 ;
completion time =  500)
    Service2 => 80/worst
- Some task deadlines will be missed : the task set is not schedulable.

Scheduling feasibility, Processor Processor1 :
1) Feasibility test based on the processor utilization factor :

- The base period is 100 (see [18], page 5).
- 0 units of time are unused in the base period.
- Processor utilization factor with deadline is 1.30000 (see [1], page 6).
- Processor utilization factor with period is 1.30000 (see [1], page 6).
- In the preemptive case, with RM, we can not prove that the task set is schedulable because the processor utilization factor 1.30000 is
more than 1.00000 (see [19], page 13).

2) Feasibility  test based on worst case task response time :

- Processor utilization exceeded : can not compute bound on response time with this task set.
```

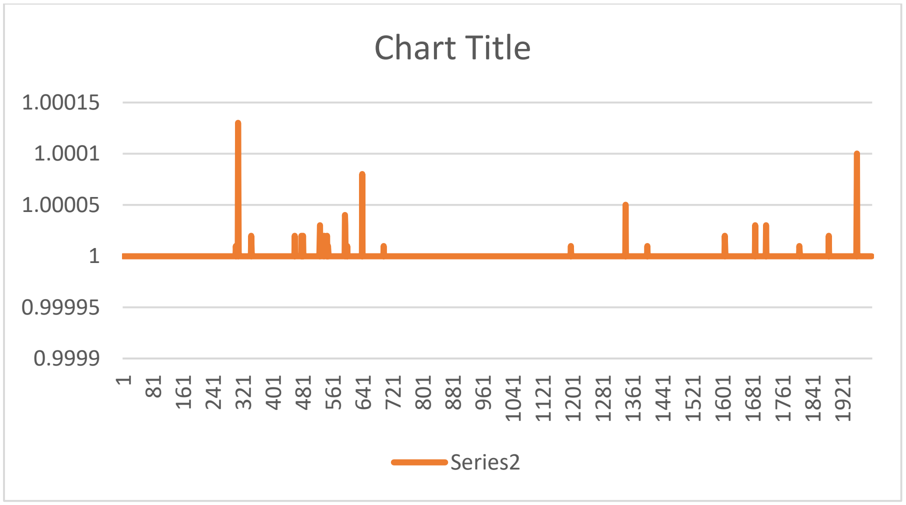
We see that scheduling the above services at a deadline of 100ms will make it miss deadline and hence is not feasible. From the simulation we can observe and expect an average jitter of 30ms.

To test my application effectively I also added a functionality of saving comma separated value files. Advantage of this was I could extract all data and use softwares like Microsoft Excel to analyze it. I have tested. I used these to perform Jitter analysis.

Jitter Analysis

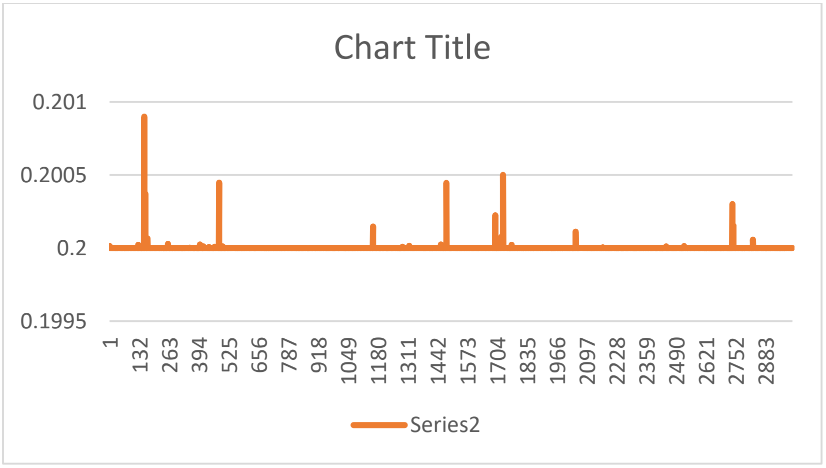
For 1 Hz:

- Total number of frames = 2000
- Total Time taken for execution = 2000.01 secs
- Deadline = 2000 secs
- Jitter = 0.01s
- Jitter per frame = 5us
- Number of deadline misses = 24
- Percentage deadline misses = 1.2%



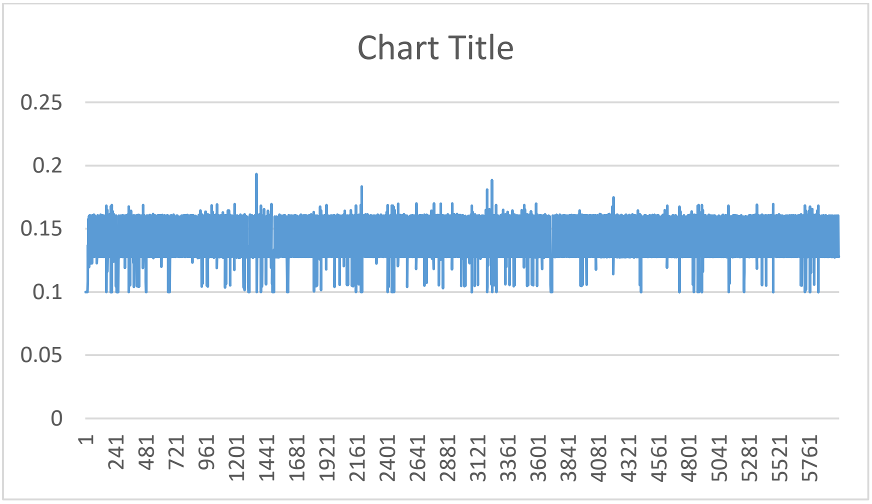
For 5 Hz:

- Number of frames = 3000
- Total Execution Time = 600.038 secs
- Deadline = 600 secs
- Total Jitter = 0.038 secs
- Jitter per frame = 12 us
- Number of Deadline Misses = 1778
- Percentage Deadline Missed = 59%



For 10 Hz:

- Number of frames = 6000
- Total Execution Time = 800.1038 secs
- Deadline = 600 secs
- Total Jitter = 200.1038 secs
- Jitter per frame = 0.3ms
- Number of Deadline Misses = 5852
- Percentage Deadline Missed = 97%



Analysis:

From the above data, I have inferred the following:

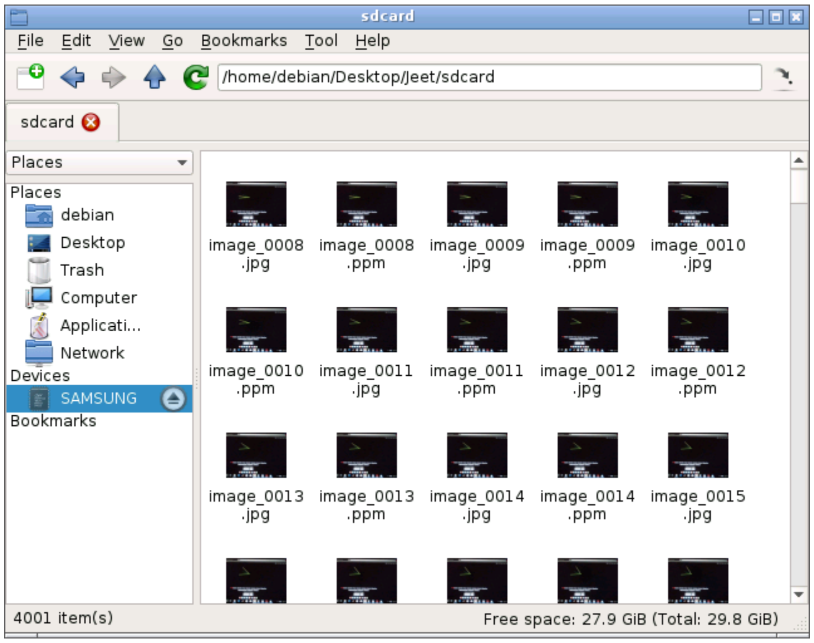
For 1 Hz frame rate there is very less jitter and the jitter per frame is also negligible. The percentage deadline missed is just 1.2% with an average jitter of 5us. Hence we see that the CPU has ample time post the two services' execution.

For the 5 Hz frame rate the average jitter is 12us which is small. But the percentage deadline missed is 59 percent. This shows that at 5Hz we are running close to max frame rate possible for my application. We can also infer that although the number of deadline missed is large the average jitter is less thus meaning that if the application has soft real time target, this application is able to achieve it.

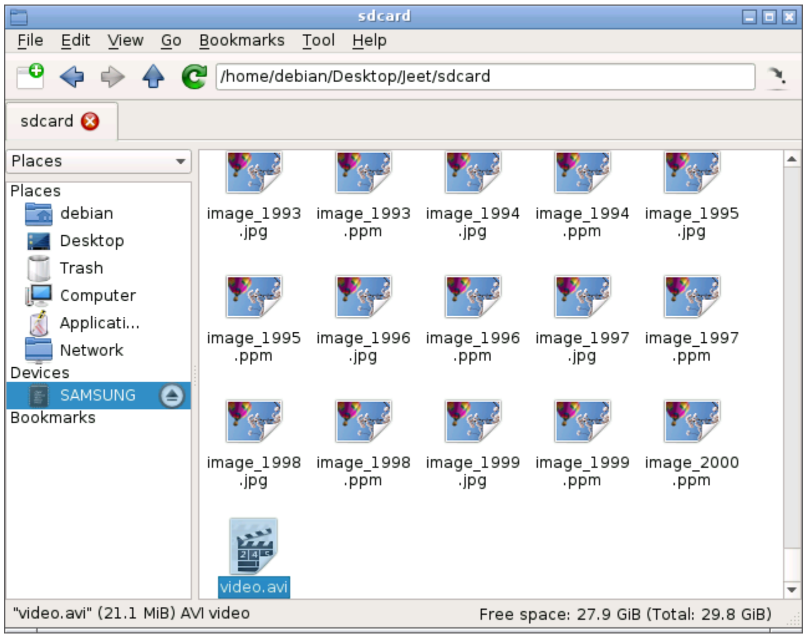
For the 10 Hz frame rate we observe that 97% of deadlines are missed with an average jitter of 30ms. This result is expected from our analysis done on cheddar. We had shown how total execution time for both services is 130ms. Here the deadline for each frame is 100ms. Thereby giving an average jitter of 30ms. I also made an observation that this result could be improved if a more powerful hardware is used or saving of frames is skipped and only the video is generated.

Proof of Concept

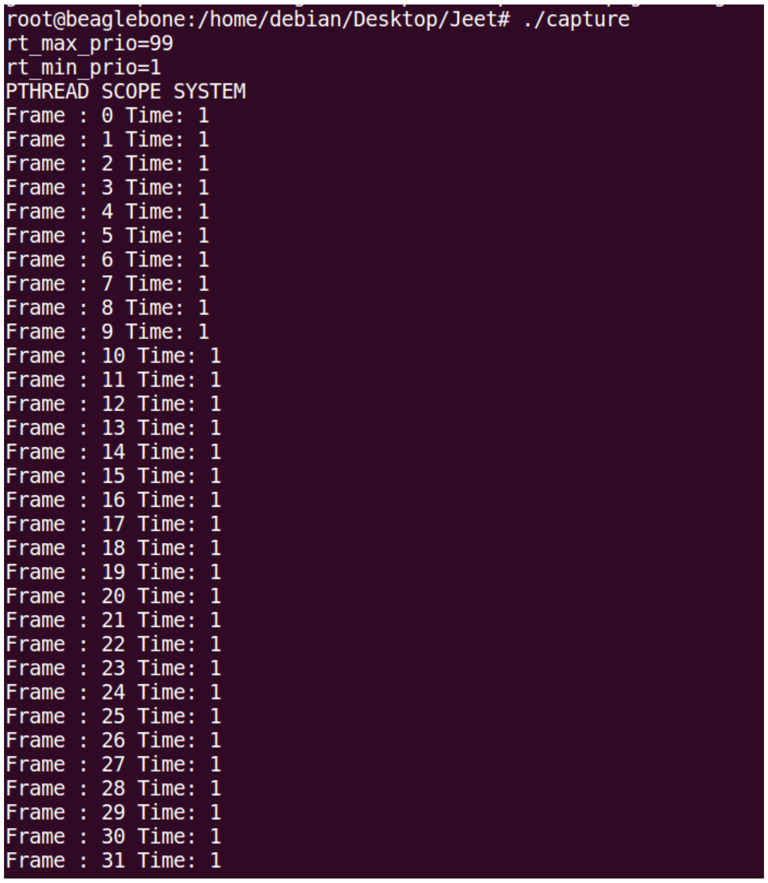
After extensive testing and jitter analysis, we see that the application performs well framerates of 1 Hz and 5 Hz. Although deadlines are missed for 10 Hz we still obtain images and video but these are skewed compared to required 0.1ms time delay. My code, with 2 threads, SCHED_FIFO scheduling policy generates timestamped images and a time lapse video using the Logitech C200 camera and BeagleBone Black. Screenshots of execution of code and outputs are shown below:



4001 item(s) Free space: 27.9 GiB (Total: 29.8 GiB)

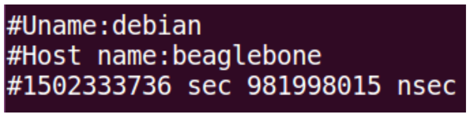


"video.avi" (21.1 MiB) AVI video Free space: 27.9 GiB (Total: 29.8 GiB)

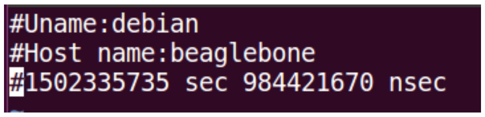


```
root@beaglebone:/home/debian/Desktop/Jeet# ./capture
rt_max_prio=99
rt_min_prio=1
PTHREAD SCOPE SYSTEM
Frame : 0 Time: 1
Frame : 1 Time: 1
Frame : 2 Time: 1
Frame : 3 Time: 1
Frame : 4 Time: 1
Frame : 5 Time: 1
Frame : 6 Time: 1
Frame : 7 Time: 1
Frame : 8 Time: 1
Frame : 9 Time: 1
Frame : 10 Time: 1
Frame : 11 Time: 1
Frame : 12 Time: 1
Frame : 13 Time: 1
Frame : 14 Time: 1
Frame : 15 Time: 1
Frame : 16 Time: 1
Frame : 17 Time: 1
Frame : 18 Time: 1
Frame : 19 Time: 1
Frame : 20 Time: 1
Frame : 21 Time: 1
Frame : 22 Time: 1
Frame : 23 Time: 1
Frame : 24 Time: 1
Frame : 25 Time: 1
Frame : 26 Time: 1
Frame : 27 Time: 1
Frame : 28 Time: 1
Frame : 29 Time: 1
Frame : 30 Time: 1
Frame : 31 Time: 1
```

The correct functioning was verified by observing the timestamps of the images for 1st and last frame.



```
#Uname:debian
#Host name:beaglebone
#1502333736 sec 981998015 nsec
```



```
#Uname:debian
#Host name:beaglebone
#1502335735 sec 984421670 nsec
```

Future Work

Future work on this project shall concentrate on improving performance for higher frame rates. The camera gave an average FPS 15 Hz. To capture and save images at rates close to this we can implement a heap queue based design. The queue shall act as a buffer for all captured frames. The thread capturing frames and adding to the buffer shal have a hard deadline of 1 sec. Other threads shall involve emptying the buffer and saving the frames as compressed images and generating a video.

The above mentioned approach shall surely improve the timing for the application. Another feature addition that can be made to the project is a File Transfer Protocol. This shall help evade any storage constraints. Adding any 'download over Ethernet' feature shall pose a great challenge to meeting the set deadlines

Conclusion

Using all the core concepts taught throughout the course, I successfully prototyped my Time Lapse Video Recorder using OpenCV. I first listed the real time requirements to do thorough real time analysis by profiling each service and used Cheddar for timing diagram analysis and verify feasibility.

The application was run to capture 2000 frames. Frames were captured and saved as jpg. Later these were saved as ppm(with timestamps) and a video file was also generated. The source code, image files and the video has been attached as a zip file.

The code consisted of two RT Threads one for capturing frames and other for saving compressed images. Another thread was used to save video to the SD card. Observation regarding jitter and deadline misses were made. We see that for 1Hz deadline is easily met, for 5Hz the application is on the brink of becoming unsafe. For 10Hz deadlines were missed consistently.

Solution to missed deadline for 10Hz was suggested and future work on the project has also been described. The project gave us students an environment of working with constraints of limited CPU, memory and IO latency, and pushed us to optimize our code so as to obtain maximum framerate and still meet deadlines.

References

- http://ecee.colorado.edu/~ecen5623/index_summer.html
- http://docs.opencv.org/
- http://www.cplusplus.com/doc/tutorial/
- http://computing.llnl.gov/tutorials/pthreads/
- Real Time Embedded Components and Systems with Linux and RTOS [Book]
- POSIX API reference manual