NC State University
Department of Electrical and Computer Engineering
ECE 785: Spring 2019 (Dr. Dean)
Project #1:
Optimizing the Spherical Geometry Code

by

<< SALONI SHAMBHUWANI >>

NCSU Honor Pledge: "I have neither given nor received unauthorized aid on this test or assignment."
Student's electronic signature: __Saloni Shambhuwani____
(sign by typing your name)
Course number: _____ECE-785_____

Introduction

In this project I tried to optimize the execution speed of a program which find the closeness of capital from each waypoint. The base code gave nearly `74.821 us 73.250 us (average and minimum resp.) which is optimized to nearly 45 us and 43us`. The function which were touched and optimized are: Calc_Closeness, sincos.c, Calc_Bearing.

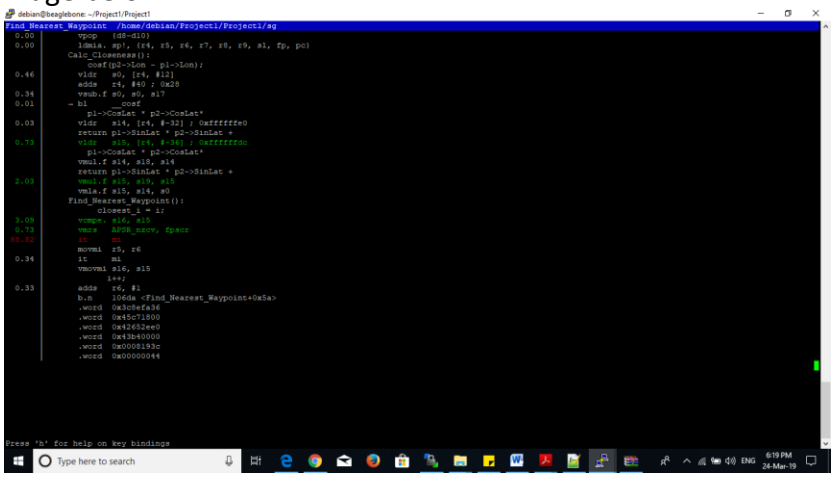
Execution Time Optimization

Below is the table of all the optimizations performed to get the most optimized code.

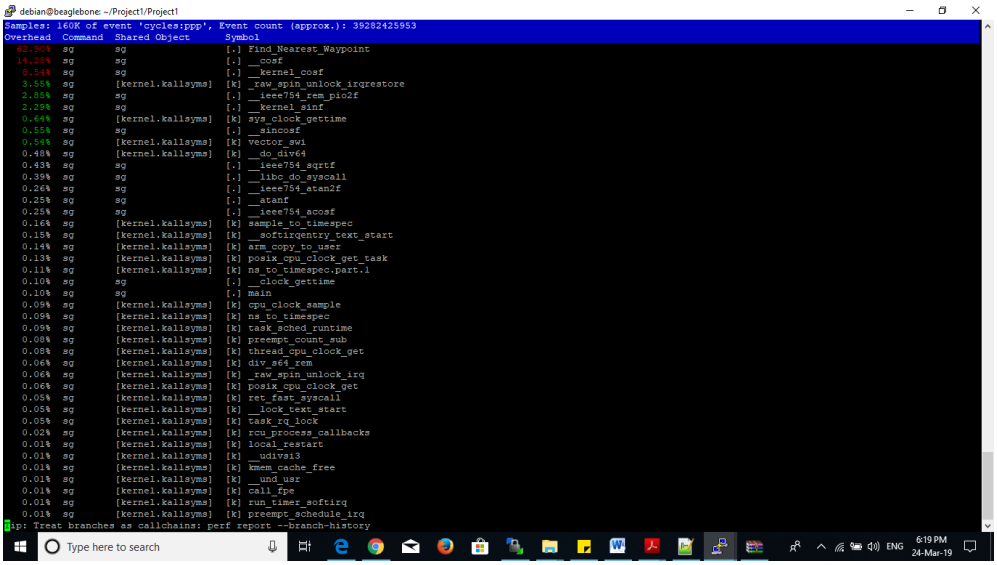
Step	Otimization description	Time taken for optimization
1	Starter code	
2	Optimization1: Reducing Pipeline stall	5 hrs
3	Optimization2: Optimizing on cos function(cos approximation)	0.50 hr
4	Optimization3:Filteration of close points and optimization on fmod in cos function.	7-9hrs

#1: Starter code

The starter code gave `74.821 us 73.250 us` as average and minimum time. So to start the optimization I looked into the perf record and found that the pipeline stalls to compare the max_c value and waiting for the comparison value to get ready. The stall can be seen in the image below:



Here the pipeline stall can be seen to have spend about 89% of execution time.



Also on looking at the report of performance we find that the program takes about 62.90% of time in function Find_Nearest_Waypoint(). So the optimization has to start from here. So below optimization is to reduce the pipeline stall and do some independent work to hide the memory latency.

- Also Highly accurate cosf function being used to calculate everything thus taking more time

VALIDATION

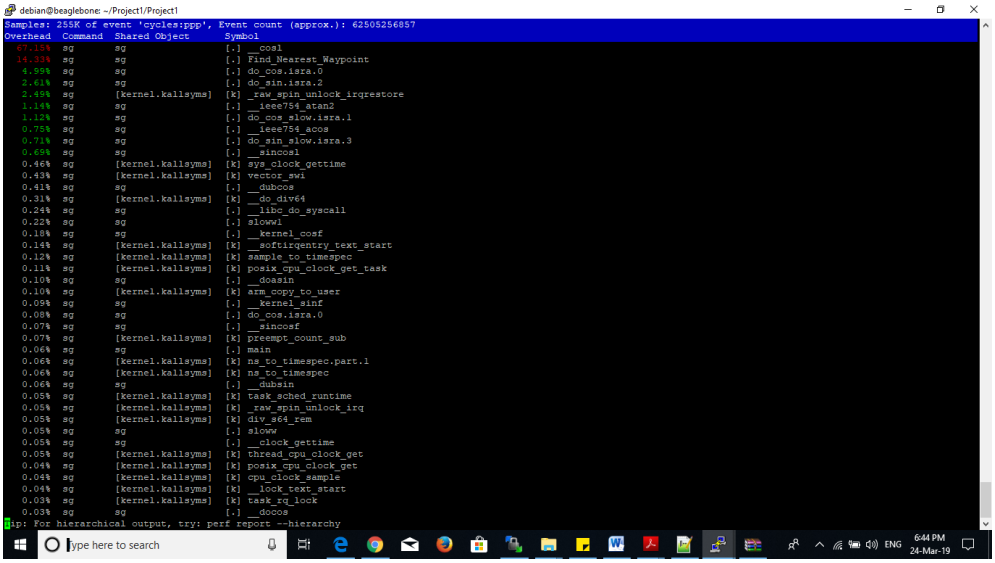
The validation code is made with double precision calculation of closeness and bearing and distance and is compared for any error in single precision code. The validation bit is set to zero in the code for now which is in geometry.h. but on making it 1 we can see the errors in distance and bearing for all capitals

Optimization #1: Reducing Pipeline stall

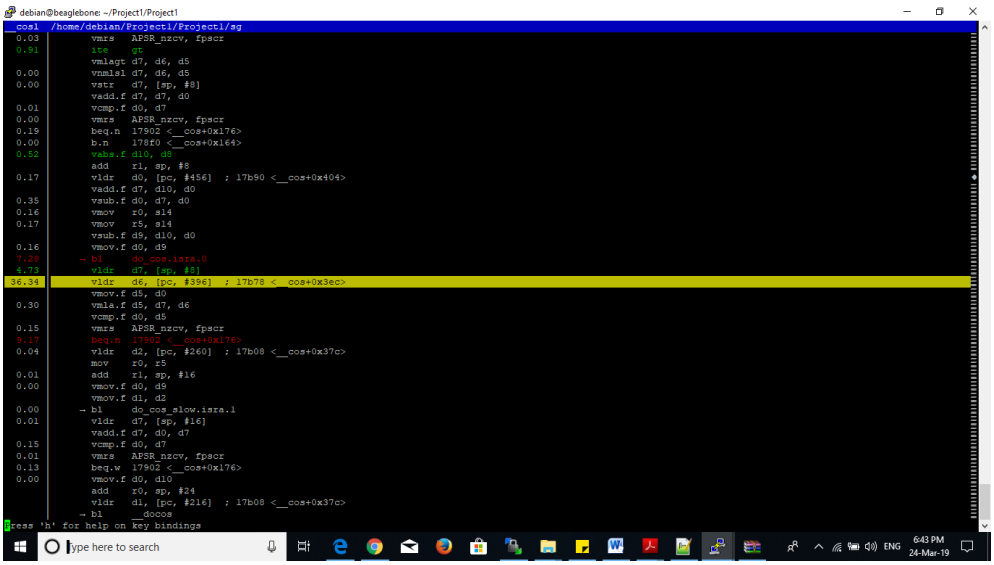
In this optimization, I tried to calculate the closeness from second waypoint for the capital while the program is calculating closeness for first waypoint(where the pipeline is stalling). On doing so I got the reduction in % time the program uses to Find_Nearest_Waypoint(). However the minium and average time remains the almost the same. This is because we are still calculating for all the waypoint it's just that we are doing two in one loop iteration. How much ever reduction is there is because I have removed one extra mov operation by using union of int and float value. Also using int reducing some of the floating point overhead and if you just

compare the two values by subtracting them then checking the sign of the value on the integer side it eliminates the stall. Since there is no way to get the FPSCR to the integer unit without stalling the pipeline, interacting with the FPSCR requires VMRS or VMSR instructions, which according to documents they stall the pipeline until all floating point instructions have completed. Thus the above optimization.

The perf report below shows the huge reduction from 89% for finding nearest point function. But cos looks to increase the overhead.



From the perf record we can see that now the pipeline stalls in calculation of cosf() function. This is our next optimization then.



This is because the cos function is not optimized and floating point operations take a long time to perform as mentioned above.

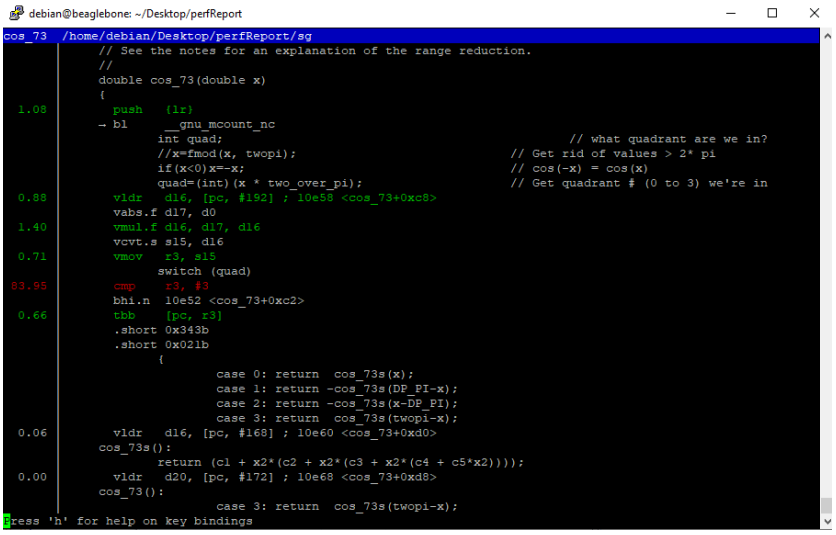
Code change is shown below:

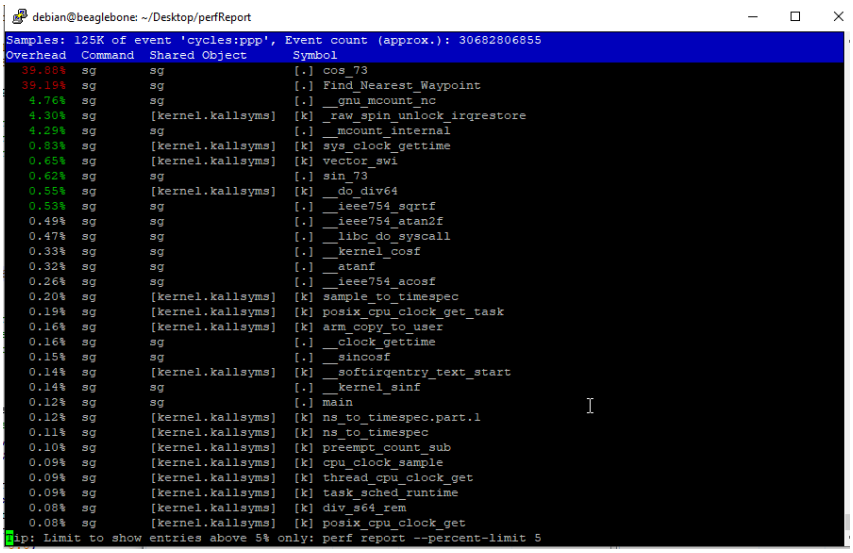
```
    c = Calc_Closeness(&ref, &(waypoints[i]) );
(p.diff_float) = c - max_c;
arr[i]=c;
c1= Calc_Closeness(&ref, &(waypoints[i+1]) );
(q.diff_float) = c1 - max_c;
arr[i+1]=c1;
 //    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &end
    if (!((p.diff_int) & 0x80000000)) {
      max_c = c;
      closest_i = i;
    }
     if (!((q.diff_int) & 0x80000000)) {
      //max_c = c1;
      closest_i = i+1;
    }
    if(max_c<c1)
    {
        max_c=c1;
    }
```

Optimization #2: Optimizing on cos function(cos approximation)

For cos approximations, I tried cos_32, cos_52, cos_73, cos_121 from sincos.c file and found that **cos_73 gives the lowest accuracy (and therefore fastest) version which provides correct results in validation**. The data for all 3 is attached in .txt files. It gave quite a good reduction in time and as shown below:

Now we see that the Find_Nearest_Waypoint() still take about 39% of program execution time So the next optimization goes for filtering the waypoints to closest with higher precision and finding these closest points as fast as possible .





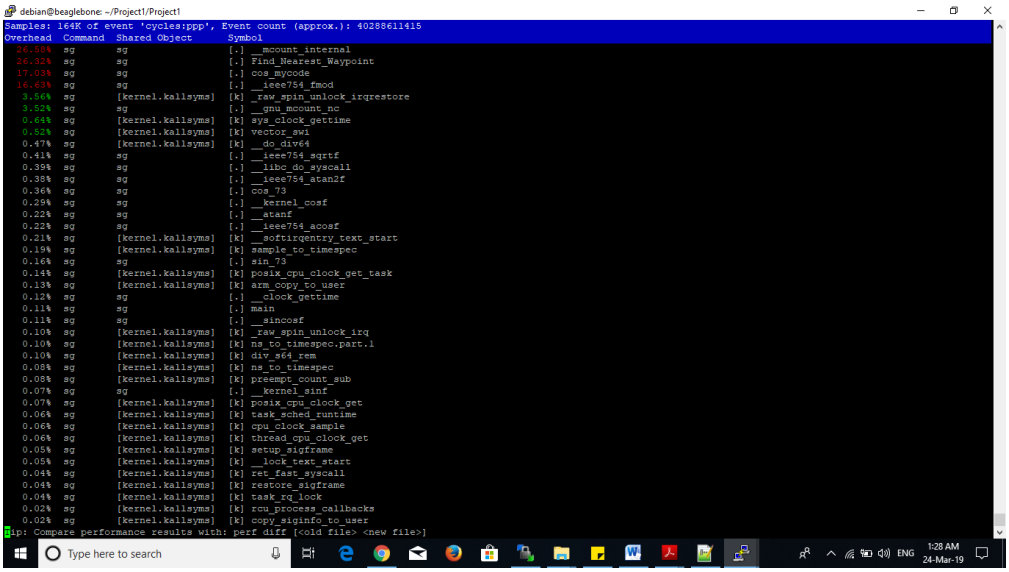
Optimization #3: Filteration of close points and optimization on fmod in cos function.

1. Refine your code to use two passes to filter out distant points.
 The first pass calculates the “closeness” to all the points using a fast but inaccurate cos approximation. Save the closeness of each point, and also the closeness of the closest point.
 The second pass calculates closeness for all points with a closeness within ε of (apparent) closest point. The second pass calculates the “closeness” with a slower but more accurate cos approximation. You will need to calculate a valid error limit ε based on the maximum error of your approximation

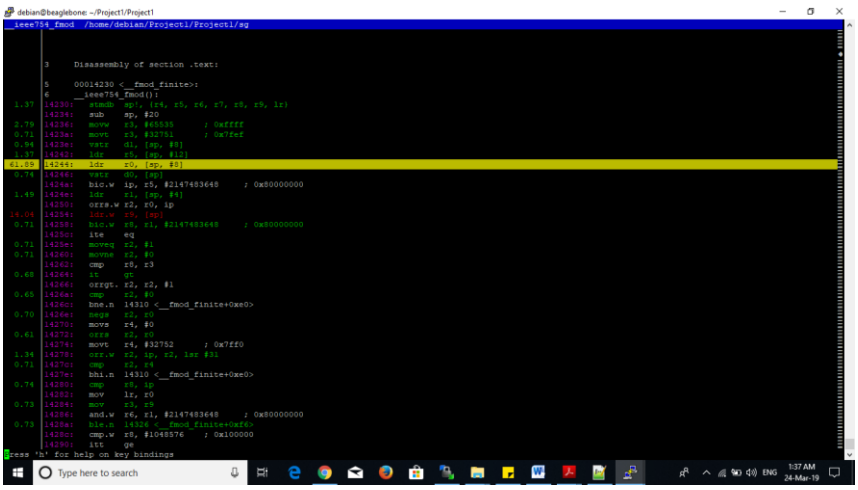
On first pass of the code I used cos_mycode which gives 1.2 decimal precision for cos to find the closest points. And then using cos_73 (précised and fast calculation version of cos) for

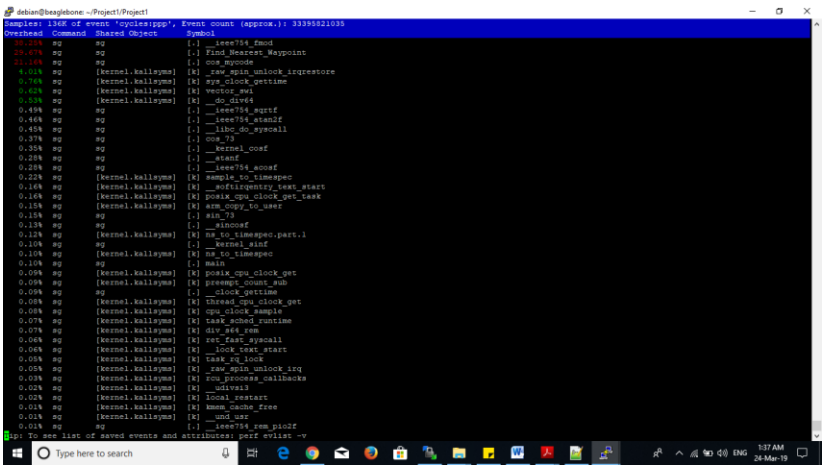
finding the closest point to capital point. Doing this reduced the overhead of Find_Nearest_Waypoint()

On looking as the perf annotate now I found that the overhead on mcount function is highest. To overcome this I removed the –pg from the Makefile which reduces the compiler overhead of profiling for gperf. This gave the reduction in time as shown below:



Below figure shows that the program time is most spent in calculation of fmod. Now still the overhead on cos was found to more. That means cos optimization still needs to be done. On looking at the object code of cos_73(), fmod is not required since the values are within 0 to pi/2. So on reducing fmod overhead we get the time reduced to

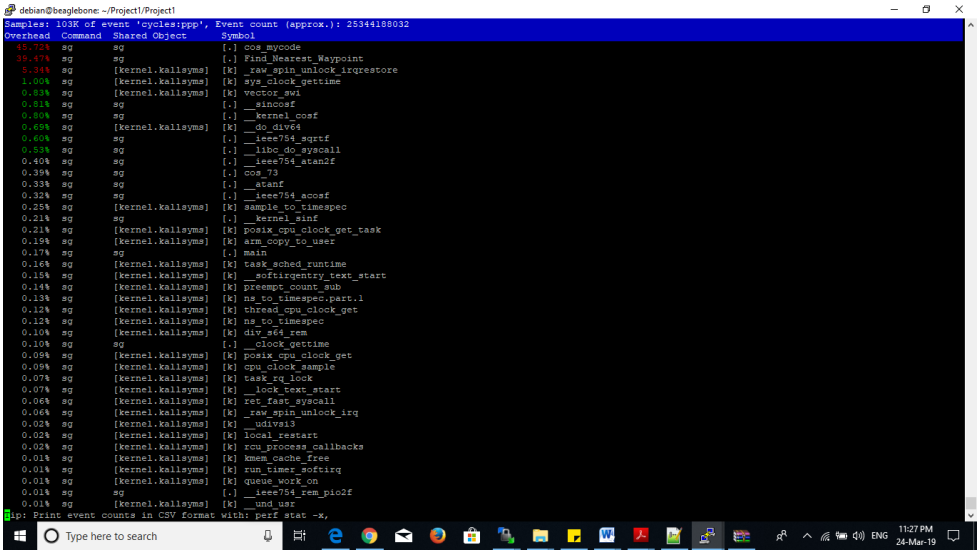


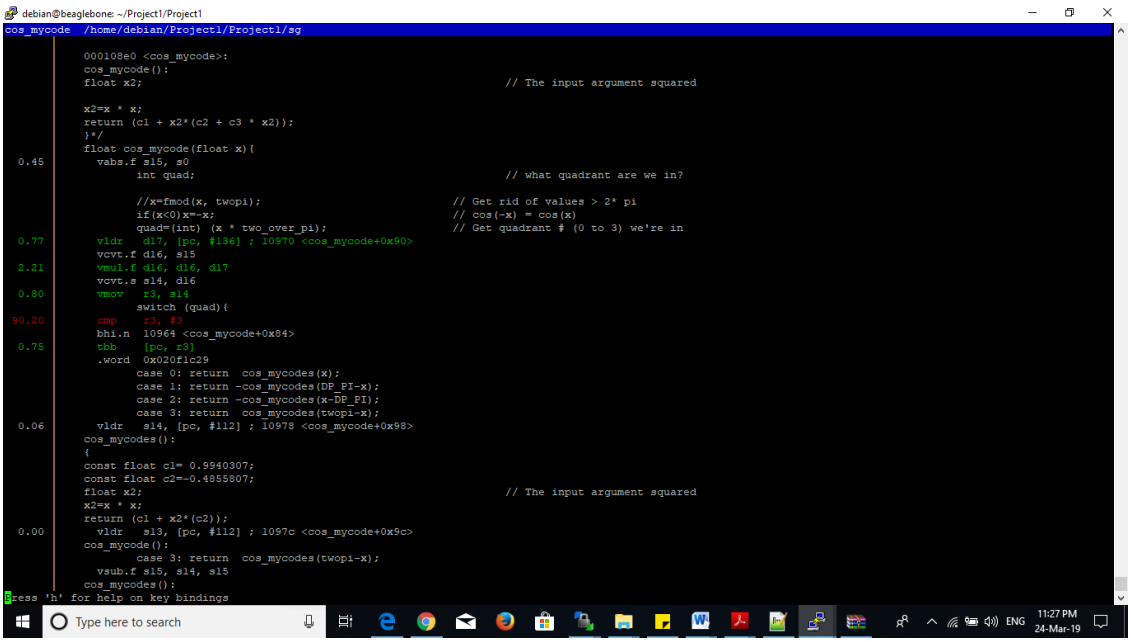


The approximation table is shown below:

Function	Degree	Approximation	Epsilon (range)	# of points in pass 2
cos_mycode	2	$c_1 + c_2x^2$	0.011 – 0.6	7-8
cos_32	4	$c_1 + c_2x^2 + c_3x^4$	0 – 0.011	7-8
cos_52	6	$c_1 + c_2x^2 + c_3x^4 + c_4x^6$	0 – 0.003	5-6
cos_73	8	$c_1 + c_2x^2 + c_3x^4 + c_4x^6 + c_5x^8$	0 – 0.001	4-5

The final perf annotate and report can be seen below:





Lessons learned:

The independent work can be done while the pipeline is stalling as the issue queue still has space to take the instructions.

The cos and sin approximations arer the most important while dealing with floating point operations.

Floating point operations should be avoided as much as possible.