Analysis of Profiler Output and the Rube Goldberg Machine

Vivek Agarwal
110050042 vivekcse@cs.iitb.ac.in

Manoj Tadepallilakshmi Datta
110050063 manojtld@cse.iitb.ac.in

C. Yeshwanth
110050083 yeshwanth@cse.iitb.ac.in

April 8, 2013

1 Introduction

In this report, we analyze the profiler output obtained in the same way as was obtained in Lab 9 except that the Rube Goldberg machine code was used instead of the dominos code that was used in the previous profiling exercise. We also compare our target Rube Goldberg machine to the one we've built and present reasons for the deviation.

2 Plot Analysis

2.1 Plot 1

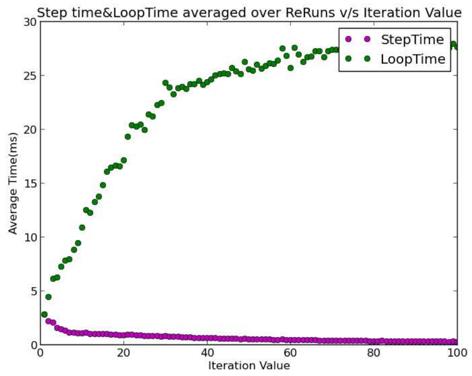


Figure 1: Plot - 1

In the plot for lab 9, we observed that the average Step time decreases while the total loop time increases with the number of iterations. The graph obtained in the case of the Rube Goldberg machine was very similar with a change in slope being observed in the total loop time as was the case with the dominos code. Again, we attribute both to the fact that linux prioritizes programs. Again programs with higher priorities seem to be given more resources and we believe that this is the cause of the speed up. The more resources for roughly the same computation per iteration cause the program to run faster and this results in the change in slope and the decreasing average Step time per iteration.

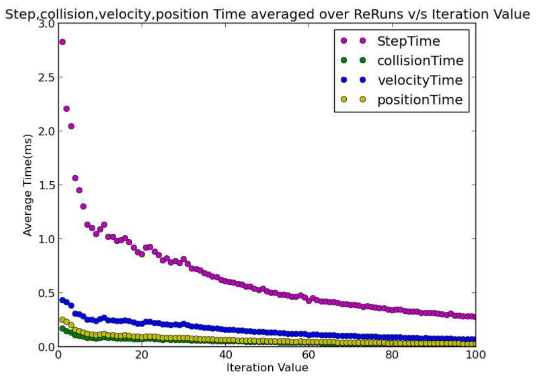
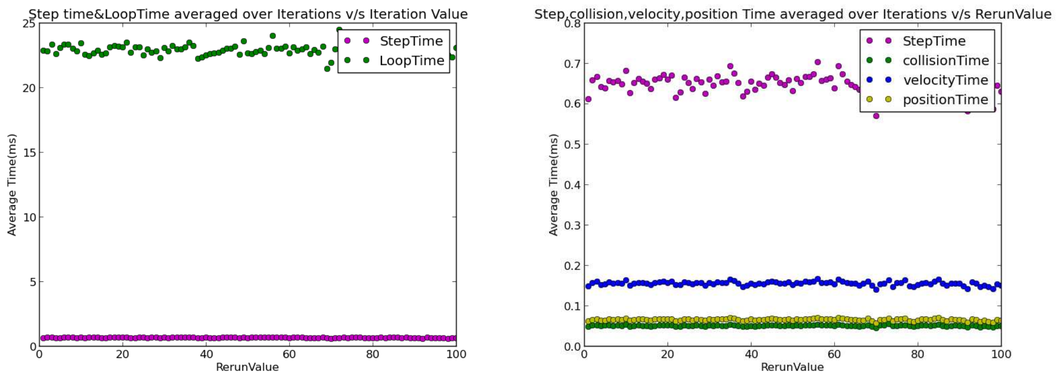


Figure 2: Plot - 2

2.2 Plot 2

Again, the results of the profiling data for the Rube Goldberg machine are very similar to the results of Lab 9. The reason for this behavior is again attributed to the same causes that were mentioned in the previous section that Linux prioritizes it's programs and allocates more resources to the ones with higher priority.

2.3 Plots 3a and 3b



(a) Plot 3a (b) Plot 3b

Figure 3: Plots 3a and 3b

The results of this plot yield little information about the functioning of the linux kernel. The plots are basically straight lines which shows that re-run values seem to have little to no effect on the performance of the program. This could maybe show that linux does not maintain information about the previous time that a program was run and the type of computations that the program carried out.

2.4 Plot 4

This particular graph yields very interesting results as it shows that it is not only the average step time that drops with iteration values but also the variance in the average loop time. The decrease in average step time has been explained in the preceeding

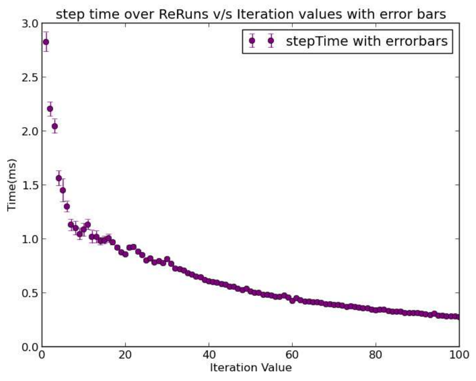


Figure 4: Plot - 4

sections. We believe the variance decreases with the number of iteration values because of firstly, the fact that the variance of the mean decreases with the number of samples that were taken and secondly as the number of iteration values increases, the resources allocated to the program running the same iteration over different reruns tend to stabilize and this reduces the variance in the average step time. This is similar to the results of Lab 9 and while they in no way prove the statements that we have just made, it at least proves that the observations are general and not something specific to the dominos code.

2.5 Plot 5

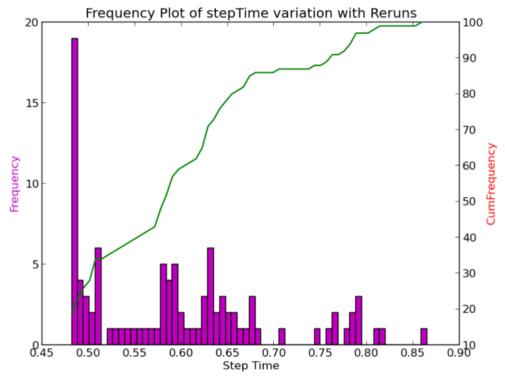


Figure 5: Plot - 5

In this graph, the distribution of average step times for an iteration number of 42 is a lot more random than the one we had obtained for Lab 9. A particularly high peak is noticed near the lowest range of average step times but nothing substantial enough for us to draw reasonable conclusions.

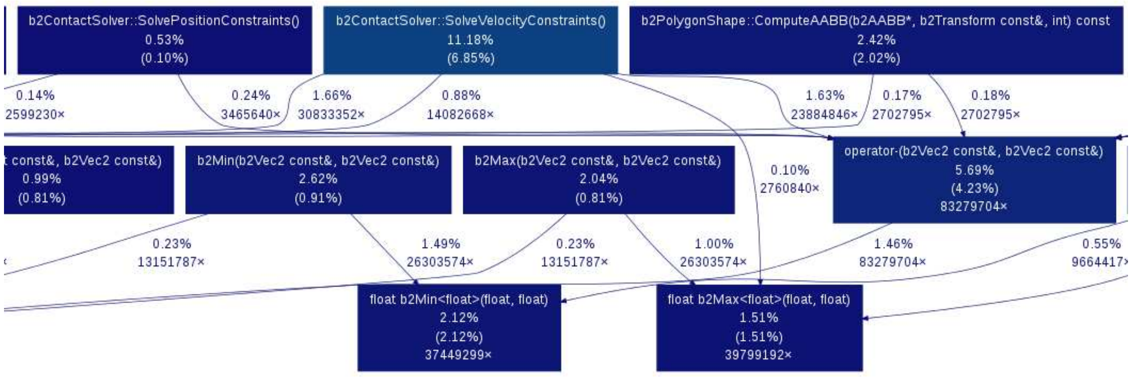
3 Performance Gains Upon Using the -O3 Flag

Based on the results of Lab 9, we ran the simulation for 50000 iterations and we found that this was sufficient to differentiate the optimized version of the code from the non-optimized or the debug version. The results of the profiler for 50000 iterations yielded a speed up of about 5 times and significant differences were observed in the call graphs. Since the behaviour of the graph is not much different between the two cases (dominos and the Rube Goldberg Machine), we present similar arguments for the case of the Rube Goldberg Machine as well. As we saw from the data the function solve velocity constraints which took up 0.68 seconds to run for the dubug-versiion of the code in one run took 0.25 seconds to run in the optimized version and comprised 12.5 % of the total running time.

3.1 Difference in the Analysis Files

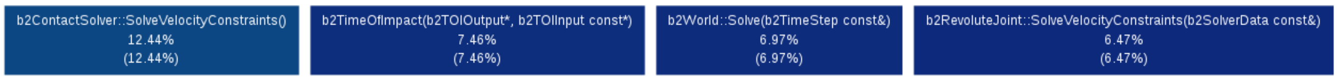
In both versions the function taking the largest amount of time was the SolveVelocityConstraints() in b2ContactSolver.cpp. The distinction between the two runs with iteration values at and above 5000 was the absence of several functions relating to the Vector class b2Vec2. One of the optimizations of the -O3 tag is the one that includes smaller functions as inline functions. This increses the size of the executable but greatly speeds up the execution of the program as the program pointer does not need to be redirected each time a function is called. There are several pointer references to the points variable within the function which are within an object called vc which is accessed repeatedly and the optimized version does not duplicate the work performed in this process. There are also several cases within the function where objects are created within the loop body but are destroyed subsequently. Memory is allocated for these objects during an iteration and is deallocated after the current iteration. This can be avoided by putting the objects outside of the loop as memory would not need to be allocated repeatedly within the loop. There is also the case where a memory address is accessed repeatedly within a loop and this causes repeated computation that can be avoided.

3.2 Differences in the Call Graphs



b2ContactSolver::SolvePositionConstraints() 0.53% (0.10%)
b2ContactSolver::SolveVelocityConstraints() 11.18% (6.85%)
b2PolygonShape::ComputeAABB(b2AABB*, b2Transform const&, int) const 2.42% (2.02%)
0.14% 2599230×
0.24% 3465640×
1.66% 30833352×
0.88% 14082668×
1.63% 23884846×
0.17% 2702795×
0.18% 2702795×
const&, b2Vec2 const&) 0.99% (0.81%)
b2Min(b2Vec2 const&, b2Vec2 const&) 2.62% (0.91%)
b2Max(b2Vec2 const&, b2Vec2 const&) 2.04% (0.81%)
0.10% 2760840×
operator-(b2Vec2 const&, b2Vec2 const&) 5.69% (4.23%) 83279704×
0.23% 13151787×
1.49% 26303574×
0.23% 13151787×
1.00% 26303574×
1.46% 83279704×
0.55% 9664417
float b2Min<float>(float, float) 2.12% (2.12%) 37449299×
float b2Max<float>(float, float) 1.51% (1.51%) 39799192×

Figure 6: Non-Optimized



b2ContactSolver::SolveVelocityConstraints() 12.44% (12.44%)
b2TimeOfImpact(b2TOIOutput*, b2TOIInput const*) 7.46% (7.46%)
b2World::Solve(b2TimeStep const&) 6.97% (6.97%)
b2RevoluteJoint::SolveVelocityConstraints(b2SolverData const&) 6.47% (6.47%)

Figure 7: Optimized

A single block of the call graph shows the fraction of the time that the function was running as a fraction of the total running time of the program. Outgoing edges represent the calls made by the function within the block to other functions that were called during the running of the function. The two values that are included with the block of a function are the amount of time the function was on the program stack and the amount of time it was on top of the program stack. The call graphs also convey the same information as the analysis files. The show that a large number of functions relating to b2Vec2 have been made inline at the cost of space but improve the running times of the program. The Call Graphs do not show much distinction between the two versions for small iteration values similar to the Analysis files but on much larger iteration values above 5000, a difference was observed in the distribution of time for various times. It showed that some of the functions mentioned above (operator+, operator*, dot products, cross products, etc) were called in the order of 10^6 times and the time spent in referncing these functions was a significant contributor to the discrepency between the two versions. Both the Call Graphs again showed SolveVelocityConstraints() to be the most expensive function and this was the function that we investigated primarily to suggest improvements to the existing code.

4 Optimizations

Several optimizations can be made to the code to improve it's running time. We found instances within the function SolveVelocityConstraints() where clever optimizations could drastically speed up run times. Some of the optimizations that we found are listed below:

4.1 Inline functions for b2Vec2 classes

As noticed in both the call graphs and the analysis files, we saw that an enormous amount of time was wasted when the functions weren't made inline as these functions were called on the order of 10^6 times. This situation can be greatly improved my making certain functions corresponding to the b2Vec2 class like the constructor, dot product, the operator* and the cross products inline as these would reduce the need to redirect the program pointer to the code that corresponds to these functions and reduce the load on the program stack. This simple optimization is responsible for a large amont of the savings that are obtained with the -O3 flag.

4.2 Reducing Object Member Access Within Loops

The points member of vc is accessed repeatedly within the for loop corresponding to SolveVelocityConstraints(). This situation can be made better when the value is stored before the loop and is incremented with a ++ increment operator within the for loop as this is a cheaper operation that incrementing the value of the pointer with a ”+ (int)variable” operation. The for loop can be re-written to remove the repeated access of the points variable and to reduce the number of ”+ j” operations performed on it. This occurs in several other cases in the code following the one mentioned in SolveVelocityConstraints(). This can be changed to the more conservative version mentioned here.

4.3 Memory Allocation within Loops

Memory is allotted for a lot of variables within the loop and this memory is allocated dynamically (ex. indexA and indexB). This memory is re-allocated in each iteration of the for loop. This memory is allocaed and de-allocated in each iteration. This work can be reduced by allocating space for the variables before the loops as this reduces the need to re-allocate memory during the iteration and these variables are accessible inside the for loop for any manipulation that might be needed. The downside to this process is that this memory will not be de-allocated as long as this particular function call is within the program stack and this could cause problems for memory intensive processes later in the function.

4.4 Mathematical Simplifications

There are also instances within the program where the amount of calls to comparatively heavy multiplication and cross product computations can be reduced by utilizing basic mathematical properties of the operations mentioned before. The distributive properties of multiplication over addition and the formula for computing triple cross can simplify the computation of certain values within the SolveVelocityConstraints() function. Specifically, the vA, wA, vB and wB can be simplified by using the distributive properties of integer multiplication and scalar cross product. The computation of dv1 and dv2 in the later segments of the code involve a triple scalar product upon substitution of the value of wB and similar variables in the code and this can be simplified further leading to a more efficient implementation. In the first case the value of P can be accumulated through the loop and the cross product can be calculated once after the loop as cross products are expensive computations and additions are relatively faster (O(n) for addition vs O($n^{1.59}$) for multiplication).

$$a \times (b + c) = a \times b + a \times c \tag{1}$$

$$a \times (b \times c) = (a.c)b - (a.b)c \tag{2}$$