Addendum: Revisiting the Runtime Efficiency of Entity Component System Architectures

Alexander Kyriacou, 103059830
Object Oriented Programming - COS20007
Swinburne University of Technology

Abstract – **While the original paper came to an inconclusive result it was hypothesized that there is likely a performance bottleneck negating any definitive conclusions on performance. This research project aims to achieve two things. The first is that we will investigate the source of possible confounding variables. Secondly we will adjust the simulation parameters to gain a better look at the runtime efficiency of various ECS architectures. It was found that the rendering functionality of the original papers simulation severely limited performance of the simulation. After adjusting the test accordingly it was found that entities through inheritance was the most efficient architecture. However, it is possible that the test used was not a fair comparison between inheritance and composition. Further research is required to test a more complex simulation, ensuring that inheritance based architectures are not being disproportionately optimized during compile time.**

I. Introduction

The previous research report investigates the runtime efficiency of entity component system architectures compared to inheritance based methods. After refining the methodology in order to reduce random variations in results, it was found that there was no noticeable difference in framerate efficiency. However, it is improbable that such vastly different implementations share almost identical performance metrics. As such, it was concluded that there is likely a performance bottleneck that is severely hindering the performance differences between different architectures. It was concluded that the most likely cause of such a bottleneck was the rendering functionality of the simulation. More specifically, the Splashkit libraries "FillCircle()" function used to render the ball objects to the screen during testing. This was concluded due to it being "one of very few similarities between testing architectures" making it the most obvious problem point. As such this report aims to investigate the hypothesis that rendering functionality of the given simulation is causing inconclusive results. In addition this report will also complete a third iteration of testing to see if more definitive results can be achieved regarding architecture performance.

A. Visual Studio Performance Profilers

Visual Studio offers a number of tools that allow users to visualize and quantitatively see how a program is being run. This includes CPU profilers, Memory profilers, object allocation tracking and more. The aspect that will be highlighted in this report is the CPU profiling capability as that is what will allow us to identify function and processes monopolizing performance. By recording CPU performance during a program, Visual Studio measures the execution time of each function and provides a breakdown of where runtime is being spent (Figure 1).

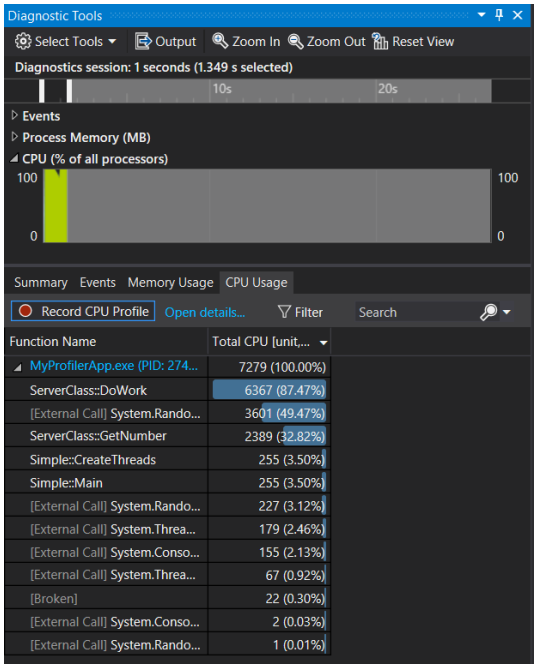

Figure 1: Example of CPU profiling output. Top time series plot shows the percentage of CPU usage at a given time. Bottom list shows a breakdown of the CPU resources being used on each function.

II. Method

A. Investigating previous simulation

The first step in this report is to investigate the performance of the original simulation. This was done by running the original simulation again. This time, a CPU performance profiler will gathering data during runtime on the resource consumption of different function. In this case, the entire simulation wasn't run again as full simulations take over 15 hours. Rather, it was run for the 5 second duration element of the simulation as it would be relatively reflective of the performance breakdown of a full simulation (as seen from the results within the original report).

B. Third Round Testing

After analyzing the results found from the performance profiling of the previous simulation (Such results will be

discussed within the discussion portion of this report) several changes were made to the simulation. The first of these changes was that all rendering functionality was removed from the simulation. The rendering functions of each architecture themselves were kept, however all 'FillCircle()' function calls were deleted (Example seen in Table 1). In addition, all Splash kit functionality within the game loops were removed leaving just the updating of the simulations themselves (Example seen in Table 2).

In addition to these function process changes, a number of design changes were made to two simulation architectures. Firstly, the 'GetComponent()' function of the entity component system architecture was optimized. In the second iteration of the simulation, this was achieved through typecasting each of an entities components against the component type 'T' provided within the generic function call. This is an unreasonably slow process and stood to unfairly represent the runtime efficiency of the entity component system architecture. As such, a new enumeration 'ComponentTypes' was created. This lists all possible components within the program allowing each component class to assign their type by including a 'componentType' field (Figure 2). This way the 'GetComponent()' function only requires a much more efficient enumeration lookup when requesting components from an entity.

The second architectural change that was made was the addition of pseudo inheritance classes within the entities through inheritance architecture. It was decided that the three class hierarchy used within the first two simulation iterations unreasonably favored the inheritance approach. Having such a simple inheritance tree would be unrealistic in a real world scenario and consequently, would discredit any comparisons made on architecture performance. To solve this issue, four 'PsuedoInheritance' classes were made for the final 'Ball' class to inherit from (Figure 3). These classes each contain empty wrapper functions for each of the 'DynamicEntity' functions, and aim to make the inheritance model of the simulation a more realistic example of how it might be seen in a real world scenario.[1]

Table 1: Changes made to render function within the third iteration of testing. The above cells show the original render functionality used in the original simulation. Below cells illustrate the new render functionality with an absence of Splashkit calls. Note: While the table shows the changes made specifically to the entities as a component bucket approach it should be trivial to extrapolate these changes to the other architectures.

Test Iteration 2 Render Function

```
public void Update()
{
     ECB_CTransform transform = ↲
             entity.GetComponent<ECB_CTransform>();

     ECB_CColor color = ↲
             entity.GetComponent<ECB_CColor>();

     SplashKit.FillCircle(
                color.color,
                transform.x,
                transform.y,
                5);
}
```

Test Iteration 3 Render Function

```
public void Update()
{
     ECB_CTransform transform = ↲
             entity.GetComponent<ECB_CTransform>();

     ECB_CColor color = ↲
             entity.GetComponent<ECB_CColor>();

}
```

Table 2: Changes made to simulation game-loop function within the third iteration of testing. The above cells show the original test loop used in the first research reports simulation. Below cells illustrate the new test process with an absence of Splashkit function calls. Note: While the table shows the changes made specifically to the entities as a component bucket approach it should be trivial to extrapolate these changes to the other architectures.

Test Iteration 2 Game-Loop

```
int frames = 0;
stopwatch.Start();
do
{
    SplashKit.ProcessEvents();
    SplashKit.ClearScreen();
    ECB_Entities.Update();
    SplashKit.RefreshScreen();
    frames++;
} while (ElapsedTime < TestDuration);
AddRecord(ballcount, time, "ECB", frames);
```

Test Iteration 3 Game-Loop

```
int frames = 0;
stopwatch.Start();
do
{
    ECB_Entities.Update();
    frames++;
} while (ElapsedTime < TestDuration);
AddRecord(ballcount, time, "ECB", frames);
```

[1] Full code for entire simulation can be found at the link provided in Appendix 2.

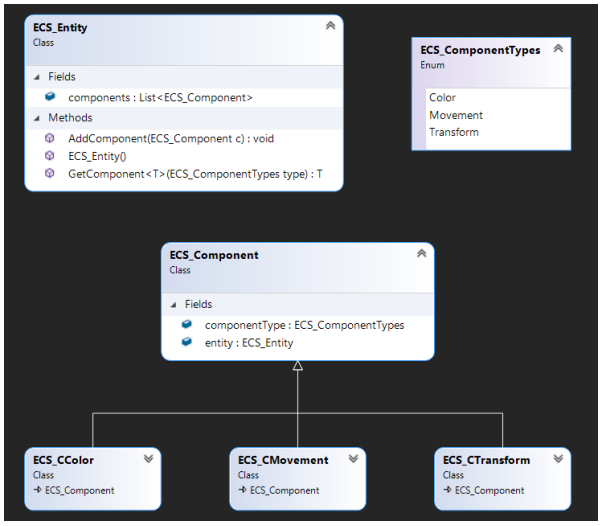

Figure 2: UML of test iteration 3 changes to entity component system architecture. Note the new enumeration and its use within the Entity classes 'GetComponent' functions parameters.

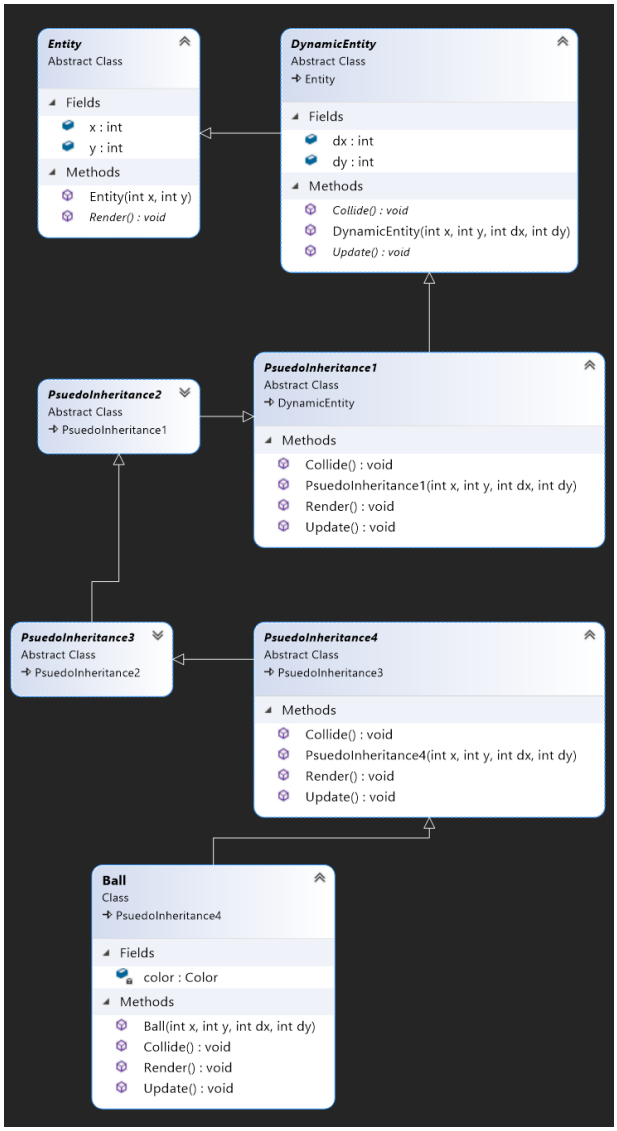

Figure 3: UML of test iteration 3 changes to entities through inheritance architecture.

C. Simulation method and data gathering

Some adjustments had to be made to the simulation parameters now that rendering had been removed from the simulation. As will be further discussed in the discussion portion of this report, it was expected that the simulations would be orders of magnitude more efficient with the removal of rendering. As such, the increase in entity count was adjusted accordingly. This set of tests was run with entity counts in 100 increments of 1000 entities starting at 1000 and ending at 100,000 entities. Each test increment was run at intervals of 5, 10 and 15 seconds with each parameter set being repeated 5 times to minimize random variations for a total of 1500 simulations per architecture.

Besides these changes, the simulation was run the same way as described in the second iteration of testing. This includes calling the C# Garbage Collector's 'GC.Collect()' method at the end of each simulation. In addition, all power-saving, time-out and other applications were closed/disabled throughout the testing duration.

CPU Usage | ECS_CMovement.cs | ECS_Component.cs | Entity.cs | DynamicEntity.cs | Ball.cs | CMA_ComponentManager.cs

Current View: Functions

Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
ResearchProject.exe (PID: 20036)	27687 (100.00%)	0 (0.00%)	ResearchProject.exe
[Native]	27221 (98.32%)	12063 (43.57%)	Multiple modules
Program.Main()	15100 (54.54%)	11 (0.04%)	ResearchProject.dll
ResearchProject.TestRunner.RunTest(int, int)	13744 (49.64%)	47 (0.17%)	ResearchProject.dll
dynamicClass.IL_STUB_PInvoke()	9361 (33.81%)	9196 (33.21%)	ResearchProject.dll
SplashKitSDK.SplashKit.RefreshScreen()	9058 (32.72%)	1 (0.00%)	ResearchProject.dll
SplashKitSDK.SplashKit.FillCircle(SplashKitSDK.Color, float64, float64, flo...	3881 (14.02%)	8 (0.03%)	ResearchProject.dll
dynamicClass.IL_STUB_PInvoke(SplashKitSDK.__sklib_color, float64, float...	3853 (13.92%)	3853 (13.92%)	ResearchProject.dll
ResearchProject.ECB_EntityManager.Update()	1596 (5.76%)	17 (0.06%)	ResearchProject.dll
ResearchProject.ECB_Entity.Update()	1571 (5.67%)	25 (0.09%)	ResearchProject.dll
ResearchProject.ECB_CRender.Update()	1432 (5.17%)	3 (0.01%)	ResearchProject.dll
SplashKitSDK.Window..ctor(string, int, int)	1340 (4.84%)	3 (0.01%)	ResearchProject.dll
SplashKitSDK.SplashKit.OpenWindow(string, int, int)	1337 (4.83%)	49 (0.18%)	ResearchProject.dll
dynamicClass.IL_STUB_PInvoke(__sklib_string, int, int)	1274 (4.60%)	1264 (4.57%)	ResearchProject.dll
ResearchProject.Ball.Render()	921 (3.33%)	1 (0.00%)	ResearchProject.dll
ResearchProject.ECS_SRender.Update()	892 (3.22%)	6 (0.02%)	ResearchProject.dll
ResearchProject.CMA_SRender.Update()	724 (2.61%)	8 (0.03%)	ResearchProject.dll
[Unwalkable]	461 (1.67%)	0 (0.00%)	Multiple modules
[External Code]	284 (1.03%)	284 (1.03%)	Multiple modules
SplashKitSDK.SplashKit.ProcessEvents()	171 (0.62%)	3 (0.01%)	ResearchProject.dll
SplashKitSDK.SplashKit.ClearScreen()	140 (0.51%)	1 (0.00%)	ResearchProject.dll
dynamicClass.IL_STUB_PInvoke()	101 (0.36%)	101 (0.36%)	ResearchProject.dll
ResearchProject.ECB_CCollide.Update()	72 (0.26%)	7 (0.03%)	ResearchProject.dll
ResearchProject.ECB_Entity.GetComponent<T>()	68 (0.25%)	44 (0.16%)	ResearchProject.dll
ResearchProject.ECS_Entity.GetComponent<T>()	65 (0.23%)	36 (0.13%)	ResearchProject.dll
SplashKitSDK.SplashKit.CurrentWindowWidth()	65 (0.23%)	7 (0.03%)	ResearchProject.dll

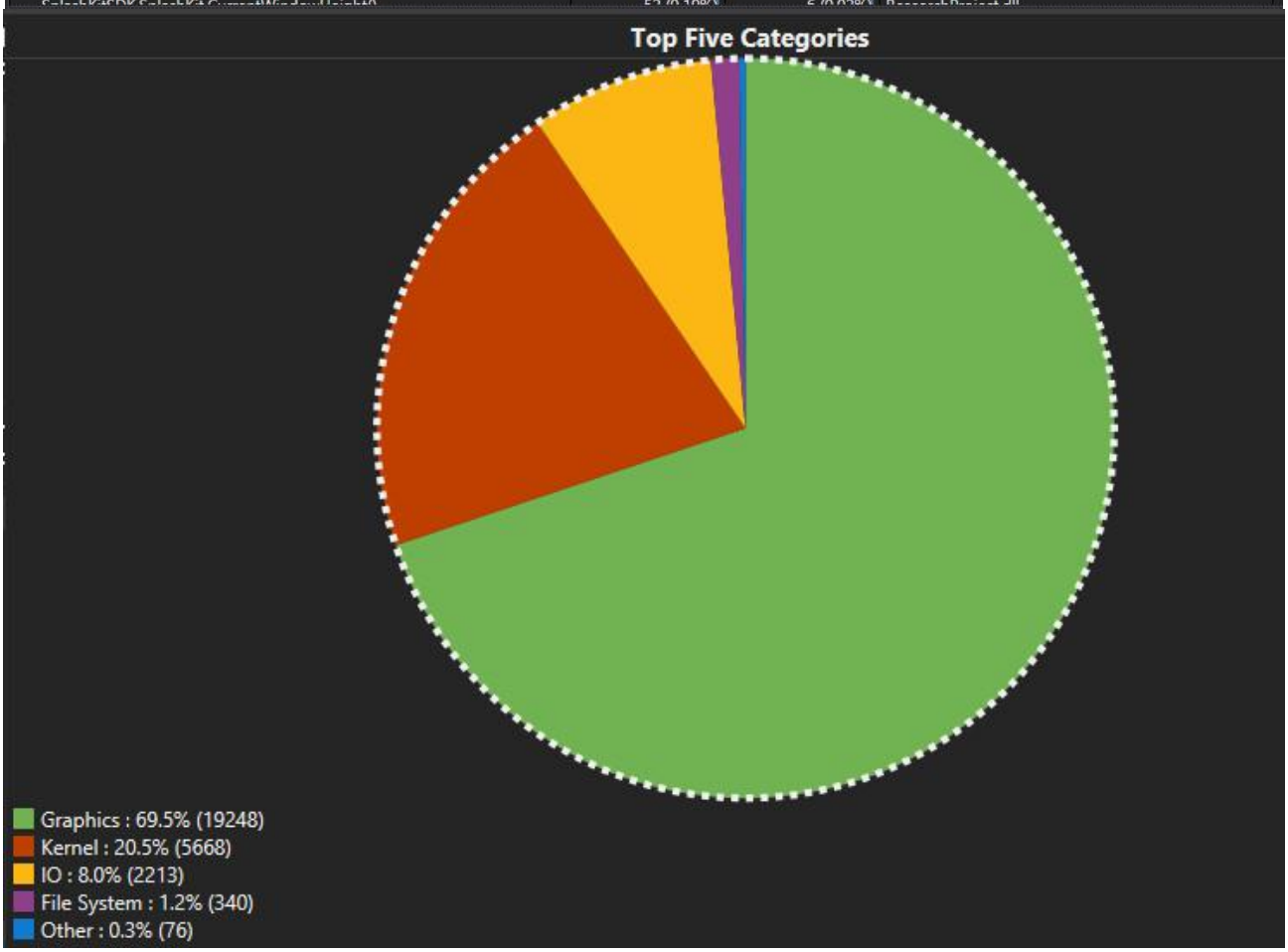

Figure 4: CPU diagnostic profile of test iteration two. Seen above is a breakdown of the most used functions within the simulation sorted by CPU usage. Seen below is a pie chart of the runtime CPU usage by function category. Note how IO is responsible for just 8% of all CPU usage within the simulation, highlighting the dominance that rendering had over the first two iterations.

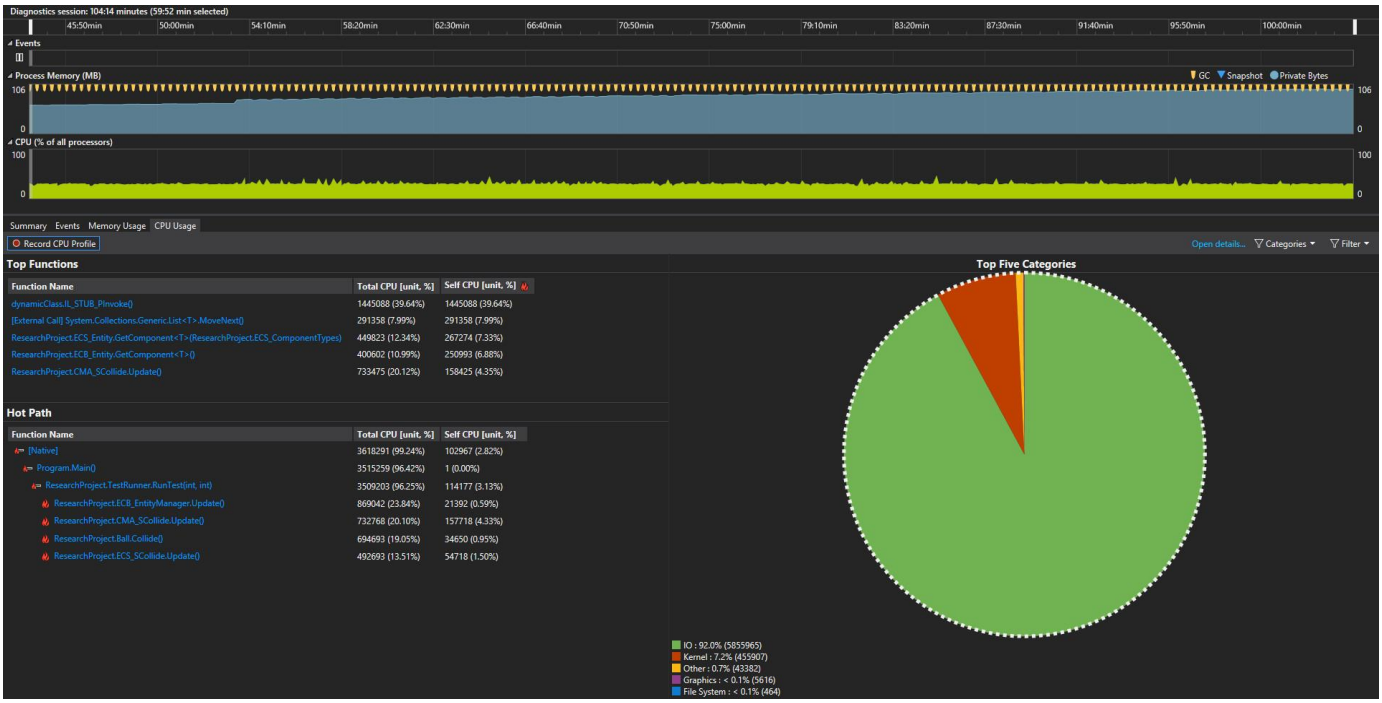

Figure 5: CPU diagnostic profile of test iteration three. Seen to the left is a breakdown of the most used functions within the simulation sorted by CPU usage. Seen to the right is a pie chart of the runtime CPU usage by function category. Note how IO is responsible for 92% of all CPU usage within this simulation, showing the improvements of this simulation over the previous tests. Also note how IO is represented in green in the right plot despite it being represented in yellow in Figure 4.

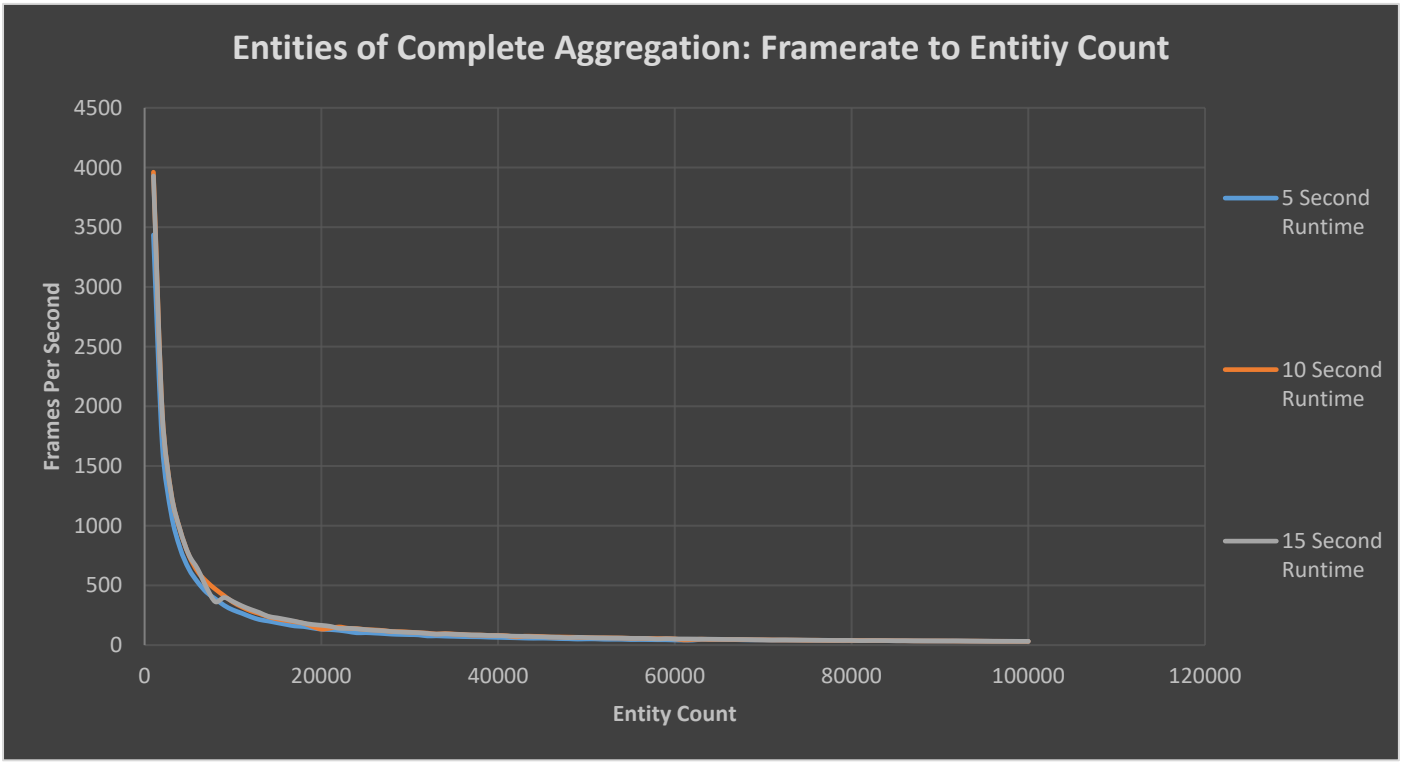

Figure 6: Entity of complete aggregation architecture results with different simulation runtimes derived from third round of testing.

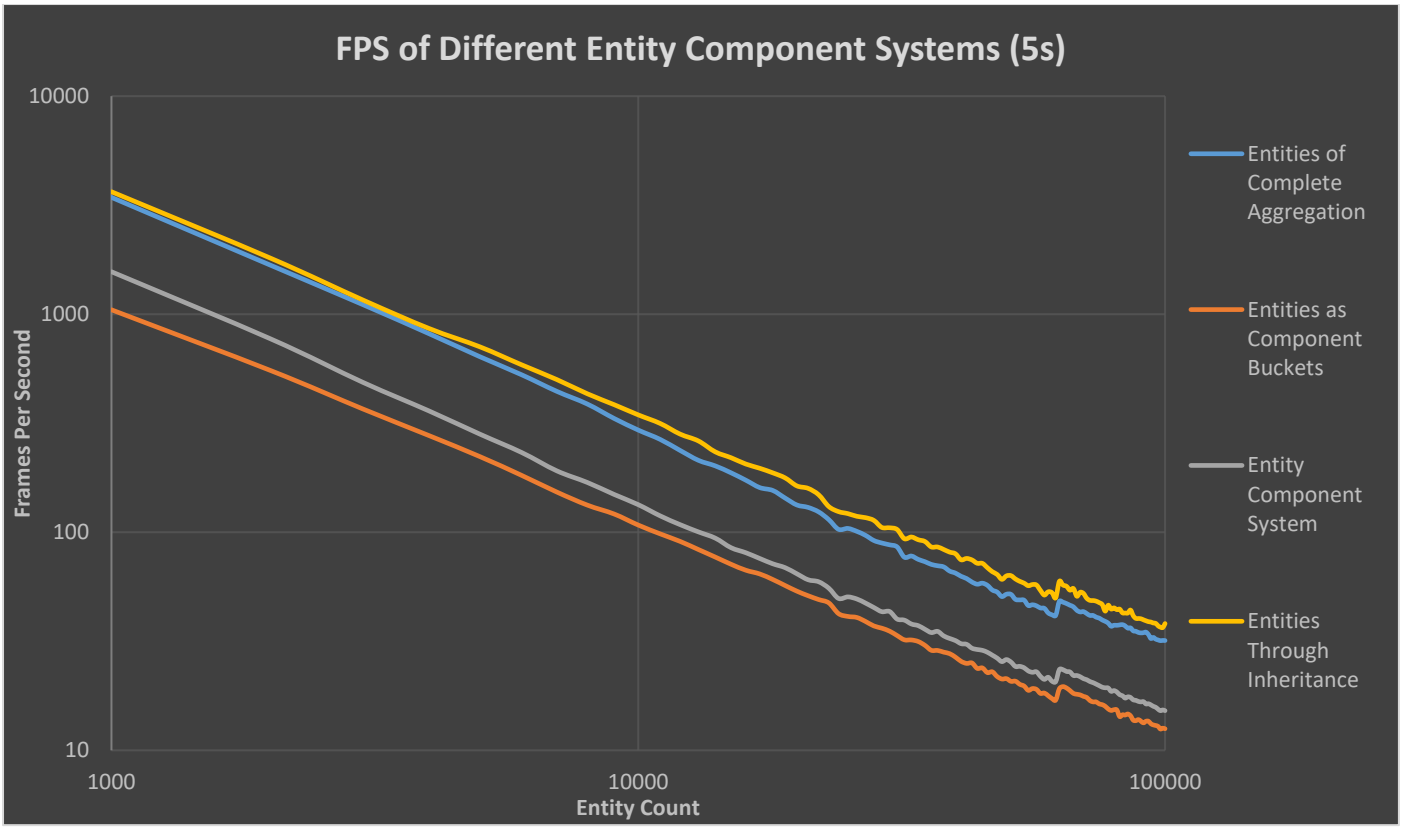

Figure 7: Framerate of different simulation architectures for a duration of 5 seconds. Note that both the x and y axis use logarithmic scales to better show differences between the variables.

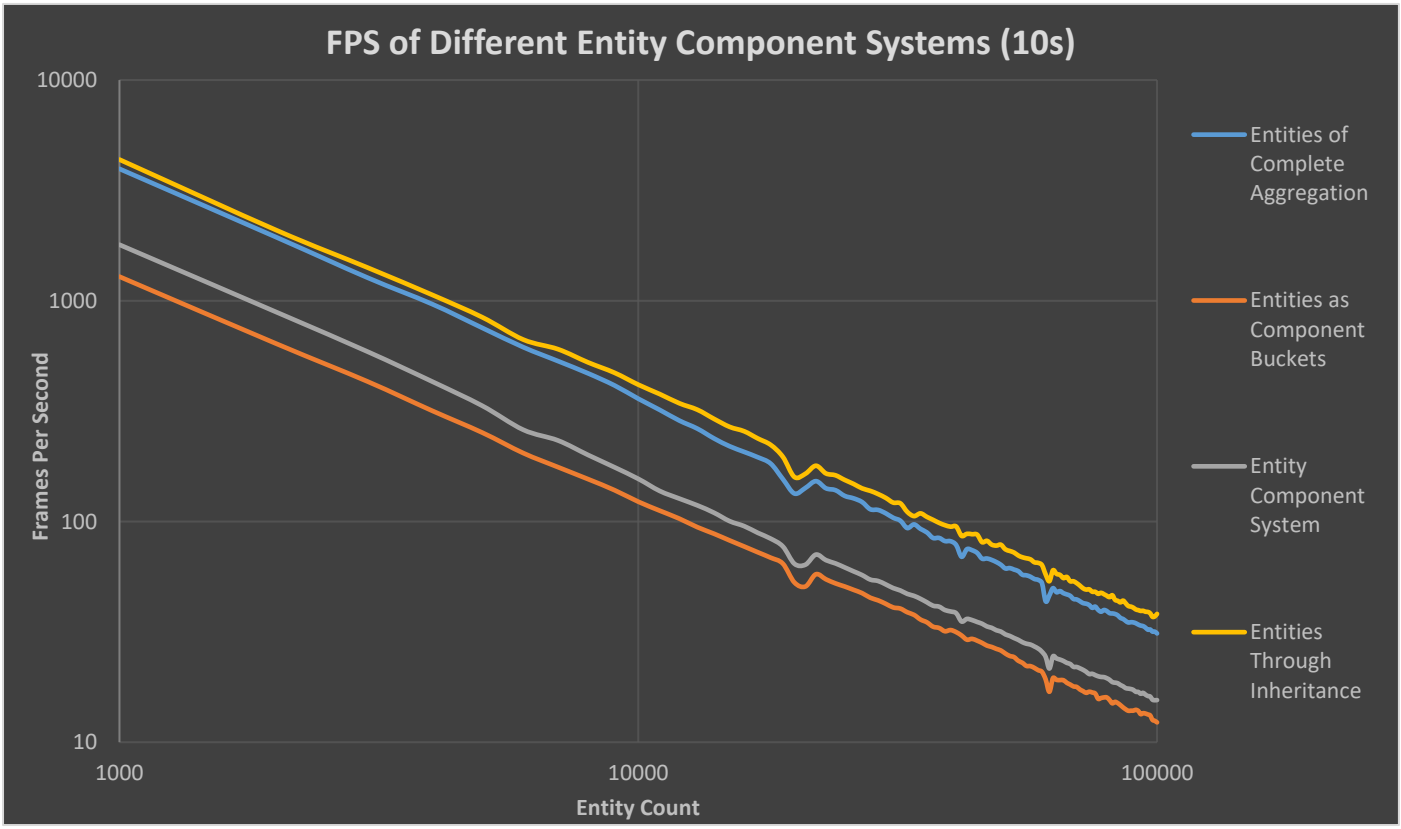

Figure 8: Framerate of different simulation architectures for a duration of 10 seconds. Note that both the x and y axis use logarithmic scales to better show differences between the variables.

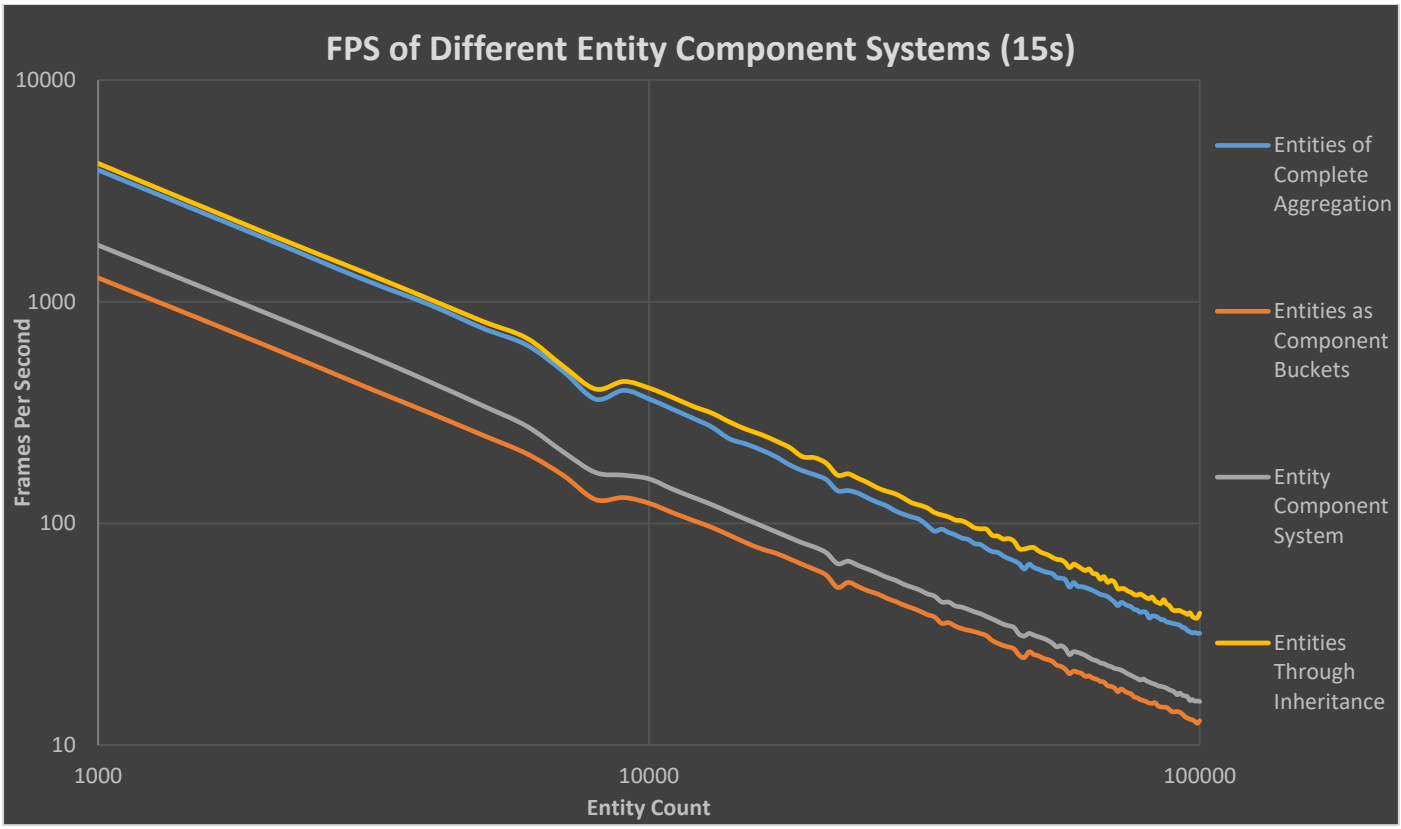

Figure 9: Framerate of different simulation architectures for a duration of 15 seconds. Note that both the x and y axis use logarithmic scales to better show differences between the variables.

Table 3: Average performance differences between each architecture. Example: The entity component system architecture had an average framerate 1.25x that of the entities as a component bucket approach.

	Entities of Complete Aggregation	*Entity Component System*	*Entities Through Inheritance*	*Entities as a Component Bucket*
Entities of Complete Aggregation	*1*	*2.074*	*0.847*	*2.563*
Entity Component System	*0.482*	*1*	*0.408*	*1.235*
Entities Through Inheritance	*1.181*	*2.449*	*1*	*3.024*
Entities as a Component Bucket	*0.391*	*0.810*	*0.331*	*1*

IV. Discussion

A. Analysis of performance bottlenecks within second iteration of testing

After analyzing the results from the CPU diagnostic profiling of the second iteration of testing (seen in the original research report) we get the results seen in Figure 4. Here we can see that the rendering functionality of the simulation accounted for a minimum of 69.5% of the total CPU usage. In addition to this, the kernel usage being 20.5% is abnormally high for a simulation that has the purpose of testing the efficiency of data transfer between architectures. We can also see in Figure 4 that Splashkits 'RefreshScreen' accounts for over 32% of CPU usage and 'FillCircle' accounts for 28% of usage. Therefore, the previous reports hypothesis that the rendering functionality is bottlenecking test performance is clearly substantiated. By removing such functionality we can expect a 69.5% minimum performance increase within the simulation. It was this result that dictated the decision to remove both the FillCircle function calls as well as all Splashkit game-loop functions described within the method of this report.

After making the design changes to mitigate the previously described performance bottlenecks, we can see the CPU diagnostic profile of the third simulation in Figure 5. Here we can see that the CPU usage of graphical functionality has plummeted to a value less than 0.1% of total resource consumption. In addition, the kernel usage has also dropped down to 7.2% of CPU usage. Most importantly, IO CPU consumption has increased a massive 1150% to representing 92% of the runtime utilization. As such, the majority of testing runtime is being consumed with the transfer of data between classes. This result reflects the simulations focus on testing the reports goal of the runtime efficiency of the program architectures themselves. Consequently the removal of rendering from the simulation also resulted in an over 23,000% increase in the amount of processes completed (Derived using data in Figures 4 and 5). This result influenced the adjustment to simulation entity counts described within the method section of this report. That way each architecture was able to be strained to its maximum capability, allowing for more substation comparison between them.

B. Analysis of test iteration 3 results

As can be seen in Figure 6, the third iteration of testing showed little data volatility between different test runtime of the same architecture. This mean that simulation framerate was not affected by an increase in simulation runtime, as was identified from the results from the first iteration of testing. Consequently this represents this simulations ability to remove random variations in resource usage and memory management processes, thus giving confidence in the validity of subsequent data.

As can be observed in Figures 7, 8 and 9 the data shows substantial differences between the framerate performances of different architectures. Here we can see that both the entities through inheritance approach and the entities of complete aggregation approach shows significant performance over both the entity component system and entities as component buckets approach. It is observed that on average, the entities of complete aggregation approach is 256% faster than that of the entities as component bucket approach and 207% faster than the full entity component system approach (Table 3). Interestingly, the entities through inheritance approach saw the greatest runtime efficiency operating 18.1% faster that the entities of complete aggregation approach and 302% that of the component bucket architecture. It is likely that we observe both the inheritance and complete aggregation architectures being at least 2-fold faster than the other methods due to an absence of cache misses during runtime. As discussed in the original report, both methods benefit from being able to organize their object data such that all information is conveniently stored within the CPU cache. Consequently, it is expected that both methods would experience much greater runtime efficiency than the other component methods as realized. While this data would seemingly suggest that inheritance based models are the most performance efficient, it is possible there were some oversights within the test design. The inheritance based architecture was 'padded' with wrapper classes in order to add realism to how it might be used in industry. However, it was not checked if such pseudo-inheritance classes were ignored and removed as a compilation optimization. Accordingly, it is very possible that the entities through inheritance architecture was compiled in a way that disproportionately represented the method. As such, the method may have performed more favorably than how it would appear in a real industry use-case.

V. Conclusion

This research report set out to serve as an extension of the research conducted in the previous report. This was done by testing the hypothesis that rendering functionality was creating a performance bottleneck causing inconclusive result. To do this, the original simulation was run with a CPU diagnostic profiler in order to examine where processing time were being consumed. Results from this test found that a majority of CPU resources were being consumed by rendering functionality as hypothesized. Subsequently, it was found that removing the rending function calls within the simulation, resulted in performance of the test increasing by 23,000% showing an obvious bottleneck in how the simulation was originally run. After analyzing this data, a third iteration of testing was designed with this conclusion in mind. This simulation involved a removal of rending from the simulation as well as design adjustments being made to both the inheritance architecture and entity component system method. Results from this simulation found that inheritance based architectures had the highest performance alongside the entities of complete aggregation approach. Both the entity component system and entity as a component bucket approach lagged behind with performance at least half that of the most

efficient two approaches. However this third round of testing reveals a possible flaw in the testing process. It is possible that the comparisons between the inheritance and composition based approaches was not a fair one. It is hypothesized that the inheritance based architecture was heavily optimized during compile time such that it appears as the favorable option. Further testing is required to investigate and test C# optimizations during the programs compilation to ensure that different architectures are not being disproportionately favored. This way more objective results can be drawn on runtime efficiency regardless of the framework of the simulation itself. Overall, this follow-up research was able to achieve much more conclusive results regarding the original reports goal of contrasting performance differences between design architectures.

Appendix I. Raw test data

The raw first round testing data can be found at:

https://docs.google.com/spreadsheets/d/1Qcog8en6GuUIv5jA2O2T3GLUjB9fATm4/edit?usp=sharing&ouid=113435264555937625106&rtpof=true&sd=true

Appendix II: Code used within research

Full code used within this research addendum can be found at the below link. Note what is referred to as "Cache Miss Adjustment" within the code solution is referred to as Entities of Complete Aggregation within this report.

https://github.com/AlexKyriacou/ResearchProject---Addendum