Memory pool in ECS

GDEV60001 GAMES DEVELOPMENT PROJECT

Jake Hickman
SUPERVISOR: BENJAMIN WILLIAMS
SECOND SUPERVISOR: CRAIG WEIGHTMAN

Contents

Abstract

As the games industry grows, the need for more performant systems increases. To combat this a paradigm was designed that adopts from data-oriented programming called Entity Component System or ECS for short. This study will evaluate the importance and impact of memory management, specifically memory pools on ECS implementations. To do this it will analyse two different implementations, with and without a memory pool, to examine the effects on system latency.

This thesis for the most part will analyse the current state of research within the current ECS space. Due to the state of the subject, the current literature is very limited and must be expanded on by looking at other common paradigms in the industry. To further this research will be done into memory pools as a whole to see how they benefit other paradigms to predict how it will affect the test results.

The aim of this study is to answer whether a memory pool will improve the impact of the ECS architecture.

Introduction

The video game industry grows year on year and the development of frontrunning game engines, such as, Unity and Unreal offer a sophisticated toolkit to build immersive game worlds. As such, there has been an uptick in the interest of game engines with more complex features to meet industry standard. (Ha, T.H., 2022) With this increasing demand, the learning curve of game engines are steadily growing especially in the terms of Unreal engine. (Christopoulou, E. and Xinogalos, S., 2017) Modern games, including large scale games such as *Hyrule Warriors* or even smaller indie games, such as *Totally Accurate Battle Simulator (TABS)* and *Vampire Survivors* are pushing boundaries by using significantly higher entity counts. These games require being able to manage thousands of entities and interactions at runtime.

One paradigm that has gained traction in the games sector known as entity component system (ECS). Some examples of ECS implementations are Unity's data orientated technology stack (DOTS). DOTS promises great performance gains while still working in unison with the current object orientated game development. The paradigm grew from another paradigm, data-oriented programming. As stated before, the paradigm first came from *Thief: The Dark Project 1998*, (LeBlanc, M., 2017) it has only recently started to gain traction for mainstream integration. Even though this is a relatively old paradigm there is a comparative lack of research in the literature, despite the fact that ECS is becoming more frequently used within industry. (Härkönen, T., 2019)

Memory pools are a technique within software development, they are a pre-allocated block of memory that is managed by a program to store and access objects efficiently. Despite their utility within the games industry and software development, there is yet to be a comparative study analysing the performance benefits associated with its usage within an ECS context.

It is with this lack of research in mind that this thesis aims to answer, 'how does a memory pool improve the performance of a sparse set entity component system implementation?' This question will set out to answer:

- RQ1. What is the current state of ECS related research specifically with ties to memory pool usage in optimising ECS?
- RQ2. Does memory pool adaptation improve the performance of a sparse set ECS implementation?

The impacts this research could have on the wider field is that this will provide an understanding of the importance of memory management within an Entity Component System. Understanding, for example, if the application of memory pools provides significant performance benefits would be of great use to games developers.

Aims and Objectives

The aim of this project is to explore the current advancements within the ECS related works. This project will in particular focus on the role of memory pools in optimising implementations of ECS. By examining the relationship between the use of memory pools and ECS, the project seeks to develop a deeper understanding of memory management within ECS based systems.

Comprehensive research will be used to identify existing gaps in existing gaps in the literature regarding entity component systems and their role in game engines. The research will need to cover other methods of programming such as procedural and object orientated programming to signify entity component system's relevance.

This project will also serve as a guild for developers who want to implement an entity component system into their engine. It will provide a basic knowledge of how entity component systems tend to work, and ways of implementing them within an engine environment. This knowledge can hopefully be expanded to develop further research.

Ultimately, it aims to present fundamental understanding of entity component systems while highlighting its relevance within the gaming industry. It also serves to highlight the research domain by providing clear and concise comparison with other methodologies.

The objective of this project is to create/design a basic game engine using entity component system (ECS). This project will need to strike a balance between optimisation and usability. To satisfy the aims of the project; the project is to conduct a thorough review of existing research related to ECS, develop an engine using a sparse set ECS architecture, and finally using this to perform a performance analysis with and without memory pools.

Must have

To achieve the goals set, at minimum an entity component system must be implemented and functional. The system must be able to take in any data type as a component, this is to make the system as resilient and flexible as possible. This design choice is to accommodate for a wide range of gameplay mechanics without core system modifications. Components must be able to be added at runtime and checked for, this is to make sure that systems are able to work. A basic system must also be implemented to show workflow.

Should have

Ability to remove components at run time. This feature would enable more dynamic gameplay such as status effects. Removal also helps avoid memory leaks and overall engine performance. Designing a basic user interface based on feedback from peers.

Could have

Multithreading would improve performance by parallelising systems so that they are able to run asynchronously. Furthermore, developing an event system would allow communications between systems and engine subsystems to further develop code scalability.

Won't have

The project will not implement complex features such as terrain generation or rendering. While they would be beneficial for larger scale games, it would be out of the scope of the project and diverge from the implementation and accessibility.

Why the focus on ECS?

The whole reason for the focus of the entity component system is due the flexibility that the system provides. It allows designers to add the functionality they wish to add and not have to concern themselves with the implementation.

Literature Review

Procedural programming

Procedural programming is a paradigm that focuses solely on the concept of procedure calls. The idea is that a program will run a series of instructions which are executed step by step. Structuring the code in a way that is typically easier for beginners. (Gupta, D., 2004) The code can then be further broken down into functions, each being able to perform a specific task.

Procedural programming is generally classed as one of the easier programming paradigms to learn as it is structured in a way that is similar to how humans follow instructions. To further support this idea, students within a university setting tend to struggle with moving towards a more object orientated programming environment. (Mazaitis, D., 1993)

Issues with paradigm start to arise as the code base gets larger. The key challenge is the lack of being able to structure related code together, unlike in object orientated programming. Without this functionality projects become encumbered with variables and functions as more become introduced. Maintaining a clear separation becomes more challenging and tracking dependencies becomes harder.

Entity component systems improve on procedural logic by implementing code through functions also known as systems. Unlike traditional procedural programming where game logic is tightly coupled systems perform tasks on entities that have a specific component. (Choparinov, E.D., 2024) This task could be anything from physics updates to rendering. This separation of game logic makes the code reusable and malleable. By taking advantage of systems, ECS avoids the pitfalls of nested function calls that is prevalent within procedural programming.

Object Orientated Programming

Object Orientated Programming is the natural progression from procedural programming. It is a form of programming where code is structured around objects. These objects prioritise encapsulation of data in classes that have one or similar reusable functionality. To further increase the usability of code, object orientated programming extends its functionality through; inheritance to provide a structured network of hierarchy of child classes that derive from the base class (Pokkunuri, B.P., 1989), and polymorphism with its ability to deviate from a base classes' implementation. (Buchanan, M., 1994)

Object orientated programming allows for a structured way of modelling real world behaviours, by organising code into classes grouped functionality such as a vehicle class. Moving the code into relevant objects maintains a modular and maintainable codebase.

Typically, within object-oriented game engines, game objects tend to follow the inheritance hierarchy of; *game object -> Character -> enemy or player*. This hierarchy is typically fine however,

each layer of inheritance adds a layer of obscurity to the original implementation. This may lead to unintended side effects such as deep inheritance trees. Unintended tight coupling between classes, means that changes to a base class in this case will have an effect across the entire hierarchy, making debugging harder to locate as more dependencies are added.

To further develop this point, imagine that a base weapon class is designed with two different subclasses, one for melee weapons and the other for ranged. However, problems arise if the designers wish to develop a weapon that can shoot and be used at close range. There are a few options to alleviate this, although, each one has its own drawbacks; choose a singular parent such as melee but add custom functionality this limits the weapon to shoot only when swinging the weapon, duplicate the code from both classes however if one is changed the other class will also need to change causing maintenance issues, and finally use multiple inheritance which introduces conflicts such as the deadly diamond. (Härkönen, T., 2019)

By using ECS the designers can define a strict hierarchy, where each entity is simply a container for components. Instead of forcing a weapon to inherit from a predefined class, ECS allows to construct a hybrid weapon by dynamically assigning components such as melee attack and a ranged attack component.

Data Orientated Programming

In contention to object orientated programming is data orientated programming, this alternative method of programming puts a strict boundary between data and the code. This method prioritises data representation giving designers the ability to easily manipulate data, instead of bundling data and functionality together. Fundamentally data orientated programming handles changes better due to being able to be coupled and decoupled rather than mutating the object. (Fabian, R., 2018)

By organising data into contiguous blocks such as structures or arrays, data orientated programming maximises the use of the CPU's cache. (Wingqvist, D., Wickström, F. and Memeti, S., 2022; Eriksson, B. and Tatarian, M., 2021) Due to modern CPU's processing data in lines, this allows for multiple elements in a single cache fetch. If a cache misses it impacts performance as they force the CPU to fetch data from slower levels of the memory hierarchy. This methodology reduces cache misses leading to faster access and overall performance improvements.

Contiguous data structures not only benefit the performance of the CPU cache but also make it easier to take full advantage of parallel processing. As the data within data orientated programming is immutable, performing multiple processes on it will not cause desync between threads.

Another benefit is that as data being stored sequentially, hardware prefetchers are able to more accurately predict what data will be needed next. This further reduces delay linked to memory access contributing to overall performance gains. (Chen, T.F. and Baer, J.L., 1995)

Entity component system architecture stores components in contiguous memory blocks to make it easier for the systems to read data prioritising performance and cache efficiency. Due to the components being stored together, they can be fetched efficiently and reducing cache misses.

Furthermore, ECS strictly decouples the behaviour of the entities from their data. This separation of provides systems with the only data that they need. These components are easy to manipulate for designers thus adhering to the principle set out by DOP.

As mentioned previously, Systems are able to be parallelizable. As each system processes different components, multiple processes can take place at one time without conflicts. Single instruction,

multiple data optimisations become possible allowing multiple components in a single CPU instruction. (Wingqvist, D., Wickström, F. and Memeti, S., 2022)

Entity Component Systems

Entity component systems avoid a common issue of the object orientated approach, such as ambiguity from inheritance. (Härkönen, T., 2019) ECS can be broken down into three different segments: Entities, which are nothing but a unique identifier; Components, data that is relevant to an entity such as health; finally, there is systems, which define how components are used.

Due to this structure ECS is able to take advantage of parallel processing. Unlike traditional OOP, where inheritance trees and tightly coupled classes tend to get in the way of multithreading, ECS separates the data from the code which allows for multiple processors or threads to operate on them.

For example, a physics system can update the entity positions while the rendering system processes texture since neither system modifies the data at the same time. By leveraging the data-oriented design, ECS implementations can utilise the parallel processing which multi core CPUs greatly benefit from.

In 2017 a talk was released about the game called *Thief: The Dark Project.* This talk was conducted by Marc LeBlanc who was a developer on the game. The talk breaks down how the game engine was made.

Marc begins by outlining the challenges that the developers faced when using traditional OOP programming. He further explains that these paradigms resulted in tight coupling and scalability issues. Not only this, but he also covers memory and compiler issues that would be present for the time.

In the talk Marc provides a detailed explanation on how the games entitles were managed. He highlights how the games architecture used ideas that would later be known as ECS. By decoupling the data from the behaviour, *Thief* was able to achieve a more efficient approach to updating game logic.

A significant part of the talk was dedicated to the memory management techniques. The memory management that is particularly important here is pre allocating blocks of memory and reusing them which aligns with modern memory pooling methods.

Overall, this talk puts in motion the start of ECS. *Thief* will continue to be the influence for modern ECS implementations demonstrating the efficiency of this method.

With the recent growth in popularity that ECS has shown, more modern applications are adopting this paradigm to enhance performance. ECS has shown that developing large scale simulations and game environments can be efficient. These implementations are now becoming available to the wider public, with a few notable implementations:

Unity DOTS (Data Orientated Technology Stack) is a system that leverages ECS to improve scalability and maintainability of games made within Unity. Unfortunately, this system is widely considered difficult to work with and not ‘Prime Time’ ready. It requires a high skill bar to use, rejecting one of the core fundamentals of ECS. (LeBlanc, M., 2017)

Bevy is a modern game engine utilising ECS architecture, to handle data in a cache friendly and parallelizable manner.

EnTT is a widely used library for ECS implementation. This library is used within several indie and commercial game projects. A game known for implementing this is *Minecraft: Bedrock Edition*. The design of this library focuses on performance and a low overhead.

Flecs is a lightweight high performance ECS framework that's been used in various games and tech demos to show how ECS can boost parallel processing and cache efficiency.

These implementations show that ECS has seen a significant increase in appreciation over the past decade, particularly in performance heavy applications such as game development. This rivals traditional OOP approaches while effective leave a lot to be desired terms of performance due to inheritance, and poor cache performance. (Wingqvist, D., Wickström, F. and Memeti, S., 2022) The key drivers towards a more ECS based architecture is due to the growing demand for a more memory and CPU efficient paradigm.

Furthermore, as data grows larger and computational demands rise, not only does ECS support more efficient memory usage but simplifies parallel processing. Decoupling components from the systems, systems are able to process large batches of data without the performance drawbacks of OOP. Therefore, ECS should not be seen as a trend within gaming but rather a necessity to revolutionise the gaming industry as a whole.

Memory Pools

In general memory management within the games industry is imperative due to the intensive real time demands of modern games. (Gregory, J., 2018) This means that memory must be managed efficiently to ensure a smooth gameplay experience. Poor memory management can lead to small issues such as increased latency or at worse game crashes. As games become more demanding memory management techniques become more prominent. (Chen, T.F. and Baer, J.L., 1995)

One of these techniques are memory pools. Unlike standard heap allocation, memory pools pre allocate a large block of contiguous memory. This methodology minimises fragmentation and allocation times during critical operations such as creation and destruction. By organising this memory in a cache friendly manner, memory pools improve data locality, reducing cache misses and boosting overall system performance.

In OOP, objects are usually allocated individually on the heap, leading to fragmentation. Memory pools are able to alleviate these issues by pre allocating a large contiguous block of memory thus storing the objects together. This improves the cache performance while providing a more predictable and consistent allocation and deallocation times.

On the other hand, memory pools within DOP allows the data to be stored to be stored in a contiguous block of memory. This again reduces fragmentation by recycling fixed sized blocks from the pre allocated block. This recycling provides a near constant time allocation which is important when dealing with adding and removing data at runtime.

Memory management is a critical part of any ECS implementation. (Wingqvist, D., Wickström, F. and Memeti, S., 2022) During runtime ECS is creating, removing and updating many components. Memory management reduces the allocation overhead, as it is an expensive operation, which has the effect of leading to memory fragmentation. As the components are added they need to be stored in a contiguous memory to improve data prefetching and reduce cache misses. This would help the support of predictable performance by having consistent allocation and deallocation.

A paper that will help in the development of a memory pool in an ECS setting is one that has been referenced throughout the paper. ‘Evaluating the performance of object-oriented and data-oriented

design with multi-threading in game development' is a paper by Wingqvist, Wickström and Memeti that examines key performance metrics within OOP and data orientated programming.

The results found from the paper, reveal that while OOP is often easier to easier to maintain and develop, the inheritance leads to fragmentation. In contrast DOP demonstrate significant performance benefits such as lower frame latency and more predictable performance under heavy multithreaded workload.

Ultimately, the paper concludes that DOP provides substantial advantages in real time game environments. This research underlines the need for memory management and optimisations. Even though conducted on DOP is still valuable as it ECS is an advanced implementation.

While dedicated peer-reviewed papers focusing solely on memory pools within entity component systems are scarce, the concept is well integrated within industry literature on ECS. This is a problem as there is a lack of formal benchmarking from rigorous academic sources. Without this dedicated study developers rely on trial and error rather than known methods when implementing memory pools.

Research Methodologies

Research Paradigm

The research paradigm used for this is study is a performance-based approach, more specifically the understanding of relevance of entity component system with a game engine setting. This paradigm allows for a clearer understanding between entity component systems and object orientated programming paradigms, also showing insights as to why entity component system is advantageous for modern game environments. The research involves qualitative analysis through literature reviews and industry examples such as unity. It also uses quantitative research to take advantage of performance benchmarks. Benchmarks recorded at runtime will include a range of multiple different entity counts then analyse the frame times to calculate the average frame rate.

The primary benchmark that will be recorded is the latency between frames. This measures the time that is taken for an entire frame update. This will help measure the time that it takes for a system to iterate through all the entities and update the component.

The latency is being recorded each frame guaranteeing high data sets for analysis. The recording will be consistent between runs so if 10 entities run for 800 frames for no memory pool it will also run 800 frames for 10 entities with a memory pool. This allows for direct comparisons between implementations.

There will be two experimental groups used:

- Non memory pooled ECS: this implementation uses std::any to dynamically allocate the entity components within a sparse set
- Memory pooled ECS: this implementation uses a memory pool to pre allocate storage for the entity components

The variable in this experiment is the memory management strategy. This will be whether the ECS implementation uses memory pooling or not.

The benchmarks will be recorded across several different entity counts, 10, 100 and 1000 entities. Testing different scales will allow for proper observation oh how memory pooling impacts performance across workloads.

Mean of the latency (average latency) will also be recorded. This will help smooth out fluctuations to provide a clearer comparison of the two strategies.

To ensure fairness, each recorded run will capture the same number of frames. If variations occur due to different execution speeds, the runs will be capped at the shortest recorded sequence.

Equipment used

To carry out this research, two tools where used. Visual studio served as the primary IDE for coding, debugging and compiling the simulation. MATLAB was used to develop graphs and charts to help illustrate the performance of the benchmarks across different configurations. By using these tools this research provides a theoretical and empirical examination of entity component system optimisations.

Justification

The selected methodology aligns best with what the project sets out to solve. This also aligns with the best practices that Jason Gregory (2018:608) introduces to profiling games. The combination of both quantitative and qualitative research methods ensures that the data provided gives a comprehensive understanding of the effects of entity component systems on game engine performance. Benchmarking is imperative to provide statistical evidence of claims made within the project.

Results and Findings

Tests without memory pooling

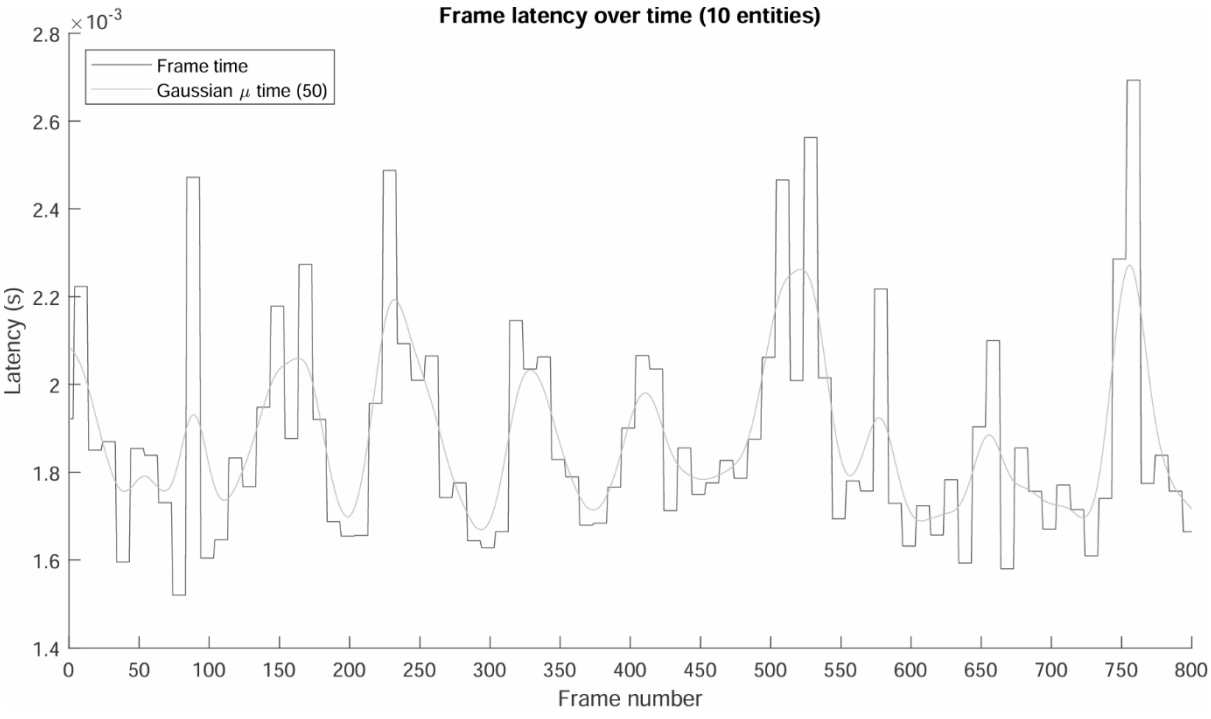

Figure 1

Figure 1 shows fluctuating frame latency from 10 entities using an ECS implementation without a memory pool, instead it uses std::any for component storage. The frame times vary between 1.6ms and 2.8ms with periodic spikes which indicate moments where the system takes significantly longer to process a frame. Within the graph is a Gaussian wave for micro time, revealing an oscillation

pattern. This cyclical performance behaviour suggests that latency variations happen between cycles, further implying that an underlying process is influencing performance at regular intervals.

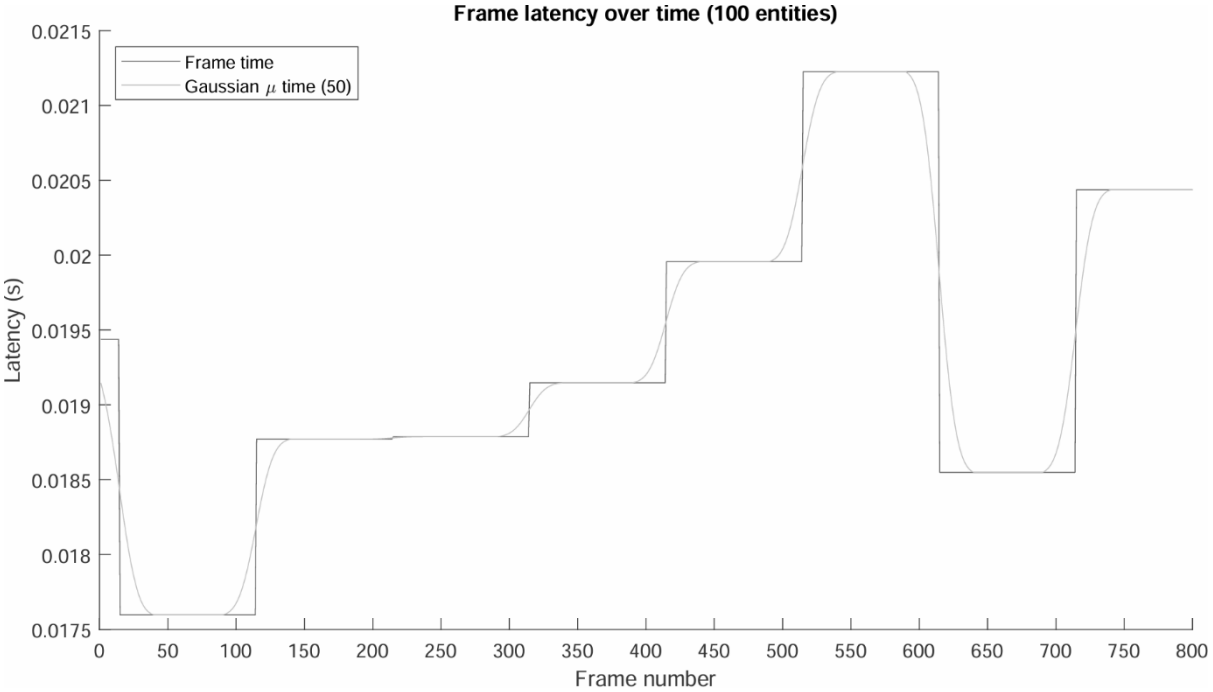

Figure 2

Figure 2 shows frame time latency for 100 entities using ECS with std::any. The frame times recorded exhibit a stepped increase over time. These fluctuations range from 0.0175s to 0.0215s with some plateaus, which suggests that the system maintains a stable processing time before experiencing a jump in latency. This graph also includes a Gaussian smoothed curve that traces micro time revealing a gradual oscillation pattern, suggesting cyclical behaviour in the latency changes.

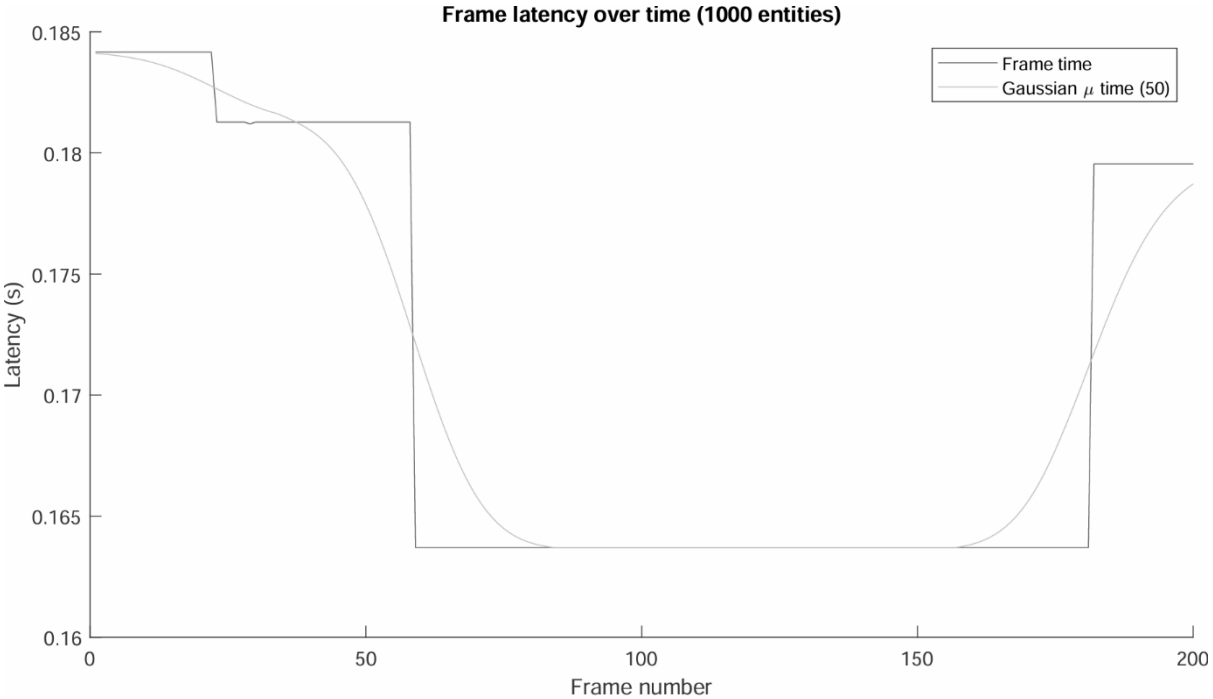

Figure 3

In *Figure 3* is a graph for frame time latency for 1000 entities using ECS with std::any. The frame times range from 0.16s to 0.185s with sudden latency drops for a small duration, in which it then jumps back up to its initial high value. This could suggest some issues with component processing issues. Included in the graph is a Gaussian smoothed curve that traces micro time implying an underlying transition phase rather than an immediate drop in workload.

Tests with Memory pooling

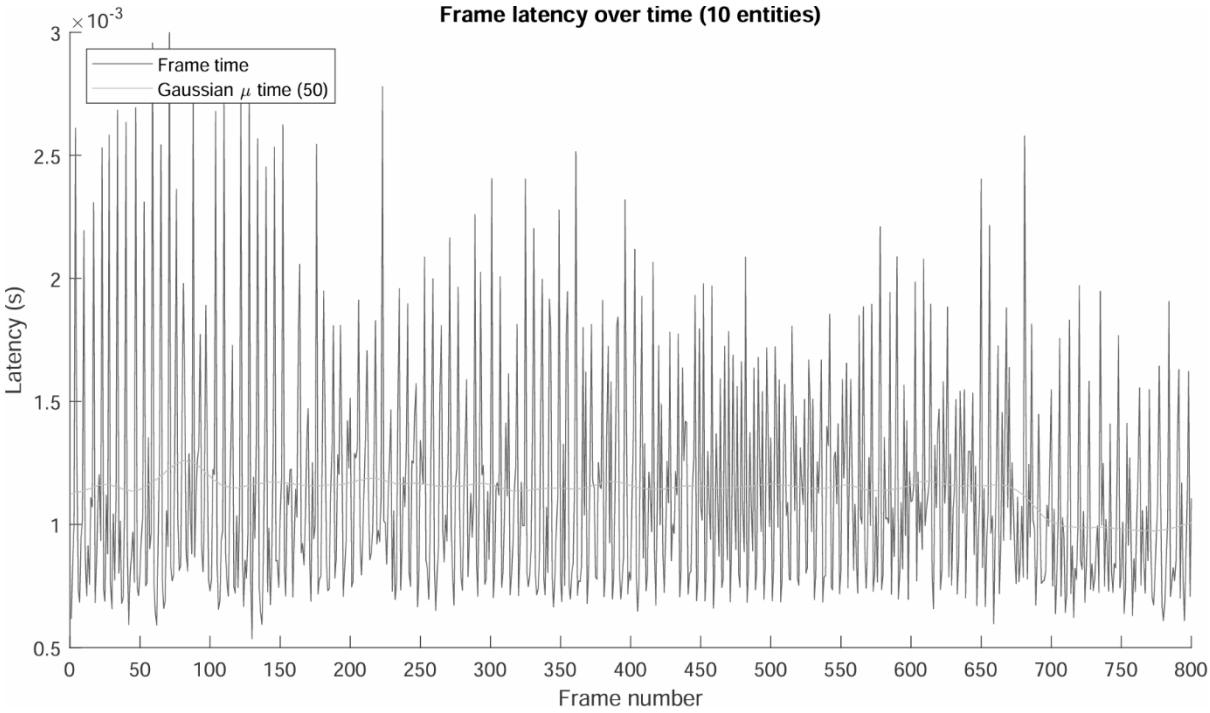

Figure 4

Figure 4 is graph for frame latency times with memory pooling present. The frame times fluctuate between 0.5ms and 3ms. The first ~150 frames are exhibiting a higher latency, with the 200-600 frames stabilising still with noticeable frame times. The last frames from 700 mark appear to be less extreme in variations. The Gaussian smoothed curve remains stable at around 1ms implying that the general frame time is fairly consistent despite the outliers.

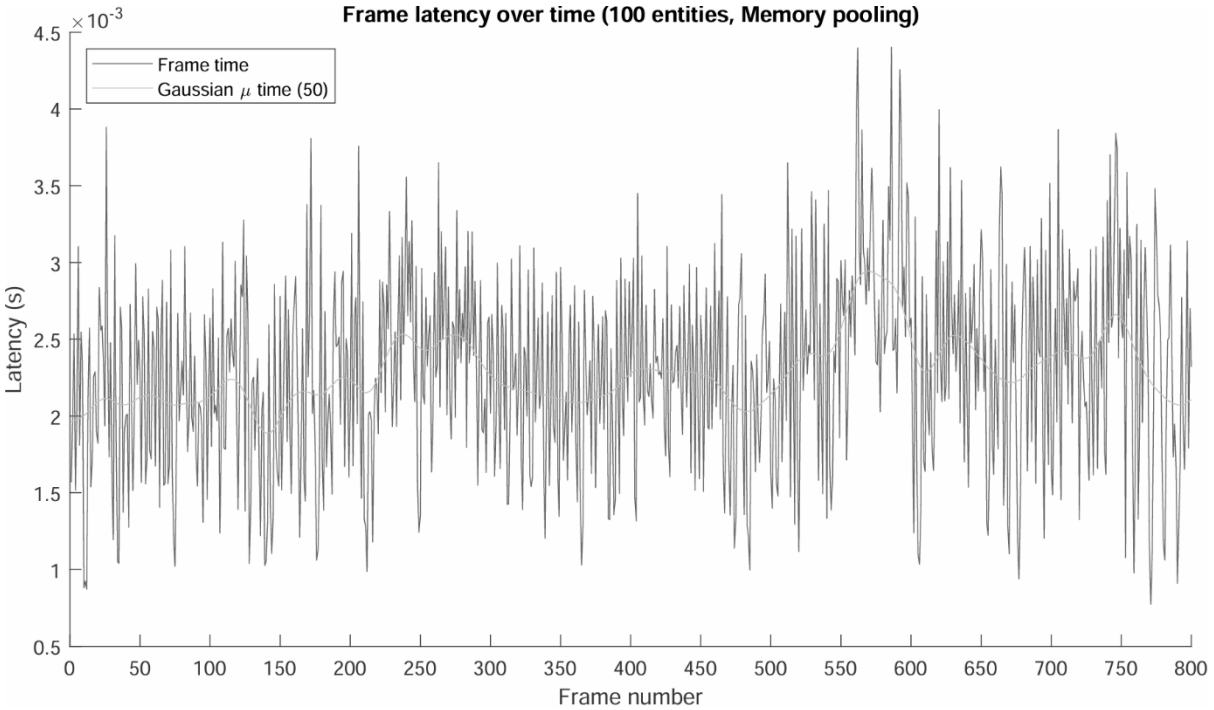

Figure 5

Figure 5 shows a graph for frame time latency over 800 frames with memory pooling. The frames fluctuate between 0.5ms and 4ms. The spikes suggest that either per-frame processing is inconsistent, or certain operations could introduce stalls. The Gaussian smoothing curve shows a wavy pattern suggesting that the frame times are not consistent.

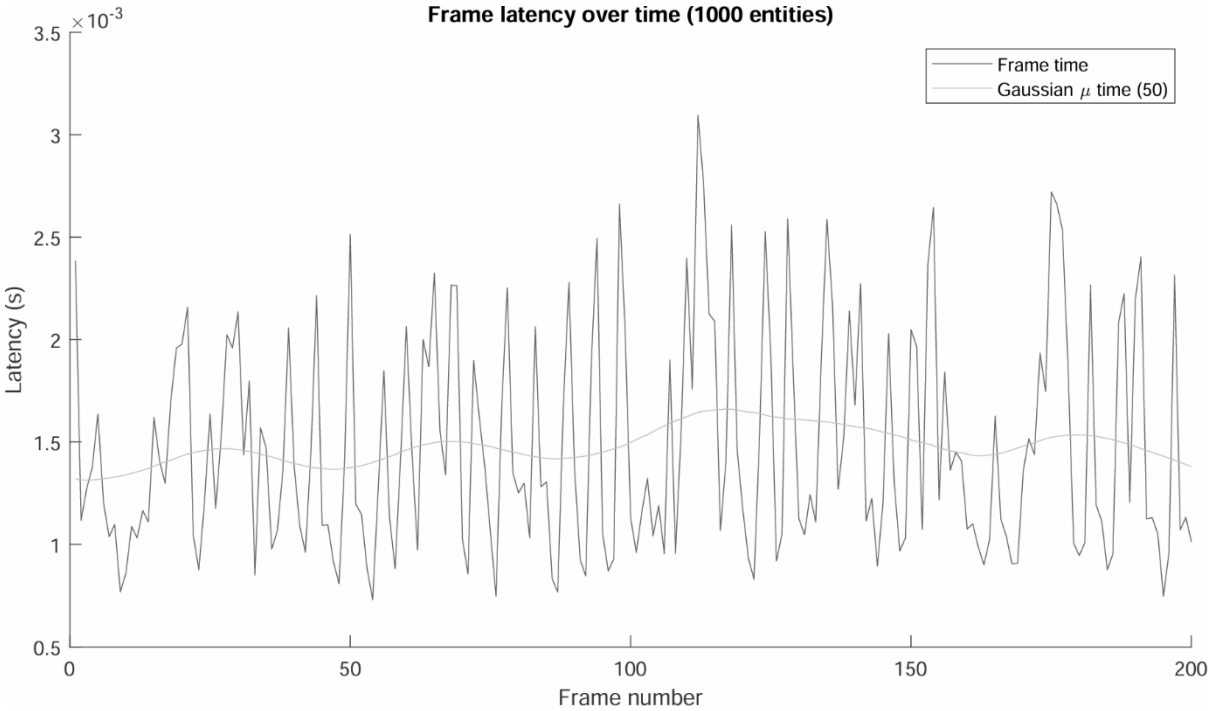

Figure 6

Figure 6 provides a graph of Frame latency over time for 1000 entities with memory pooling. There is fewer latency spikes in this test with them being lower than usual, they fluctuate between 2.5ms to

3ms. The Gaussian time curve suggests that frame latency is relatively constant and seems to stabilise over time.

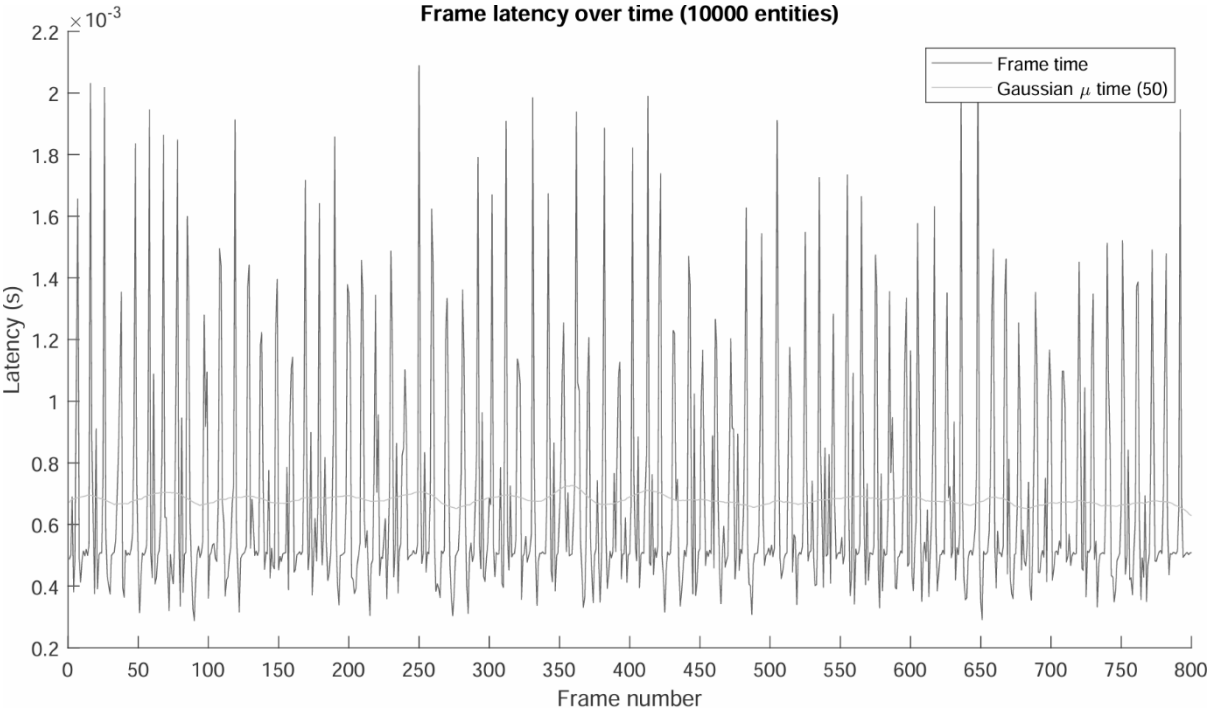

Figure 7

Figure 7 shows a graph of frame latency of simulating 10,000 entities with memory pooling. The frames are fluctuating from 0.1ms to 2ms, this pattern of regular tests suggests systematic stalls. The Gaussian time curve stays relatively constant hovering around 0.7ms, which is lower than the 100 and 1000 entity tests. This suggests that the memory pool is helping.

Comparing pooled vs non-pooled

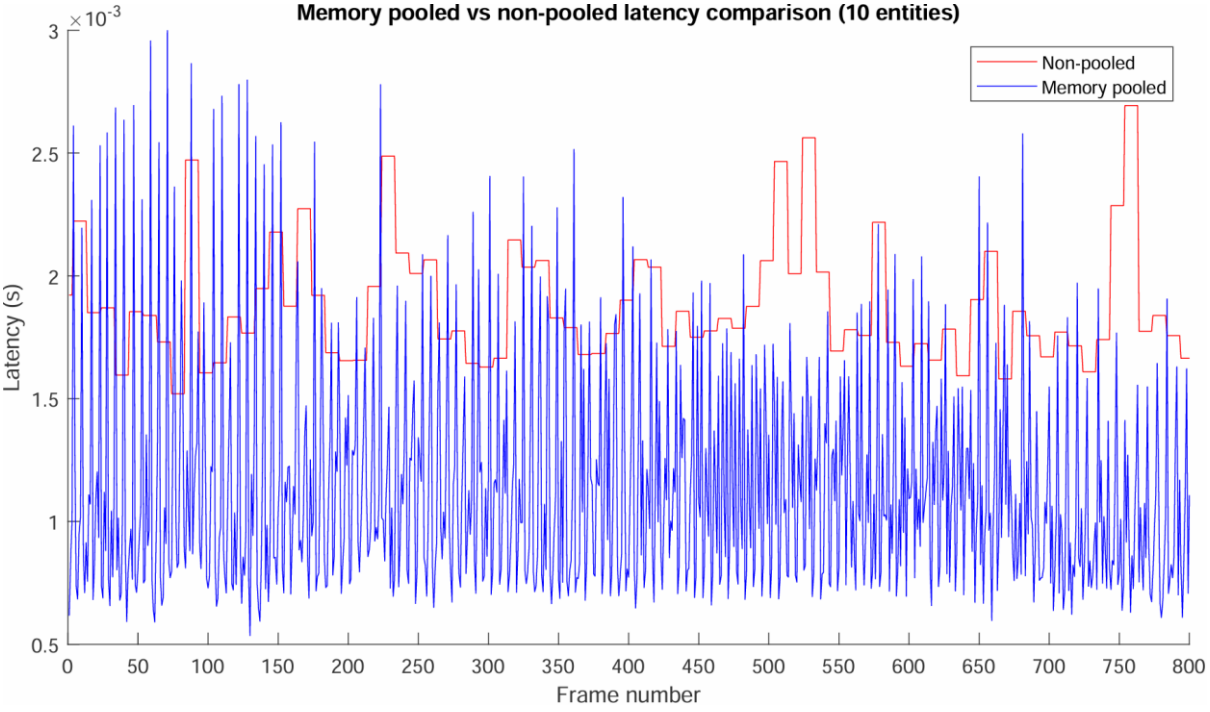

Figure 8

Figure 8 compares the frame latency between a memory pooled and non-memory pooled implementation of ECS with 10 entities. The non-memory pooled ECS implementation (red) has higher latency overall. Some spikes do appear showing inconsistent frame processing. The memory pooled ECS implementation (blue) on the other hand, has a lower latency overall, with more frequent oscillations but stays below the non-memory pooled implementation on average.

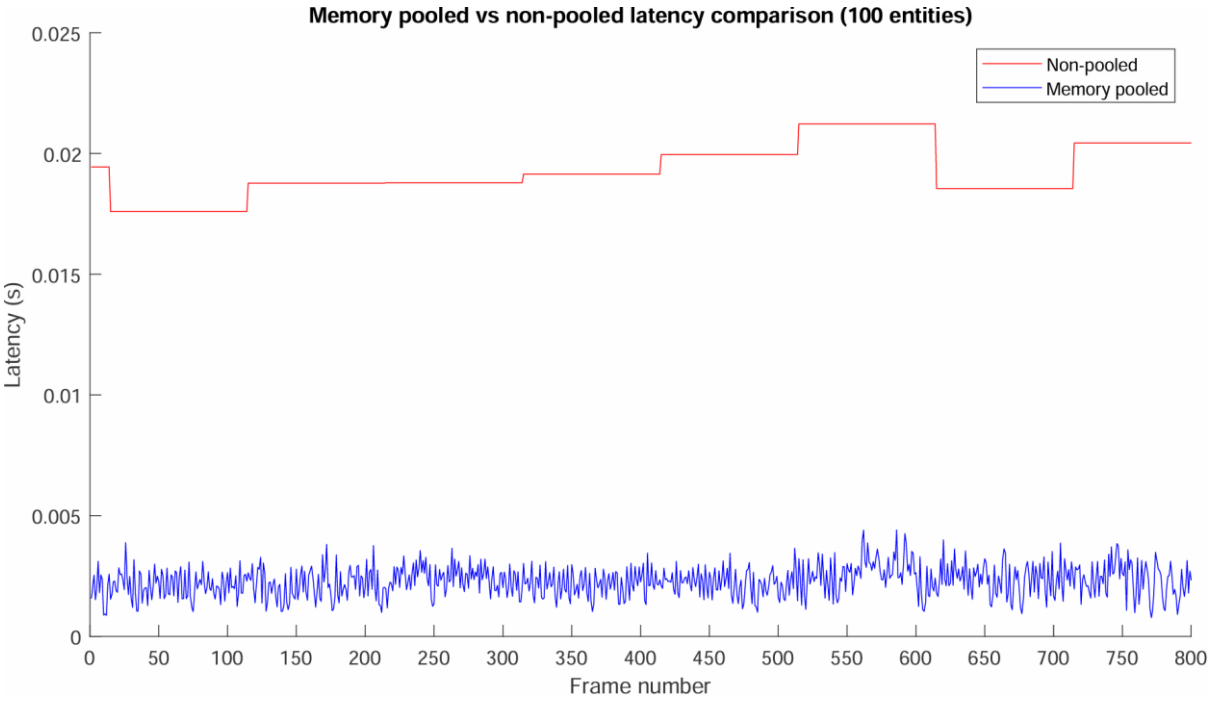

Figure 9

Figure 9 compares the frame latency between a memory pooled and non-memory pooled implementation of ECS with 100 entities. The non-memory pooled ECS implementation (red) has marginally higher latency overall which is consistently over 0.015s. The non-memory pooled ECS shows a step like behaviour. The memory pooled ECS implementation (blue) on the other hand, has much lower latency, typically staying below 0.005s. The graph fluctuates slightly indicating some frame-to-frame variations but remains consistently lower than the non-memory pooled implementation.

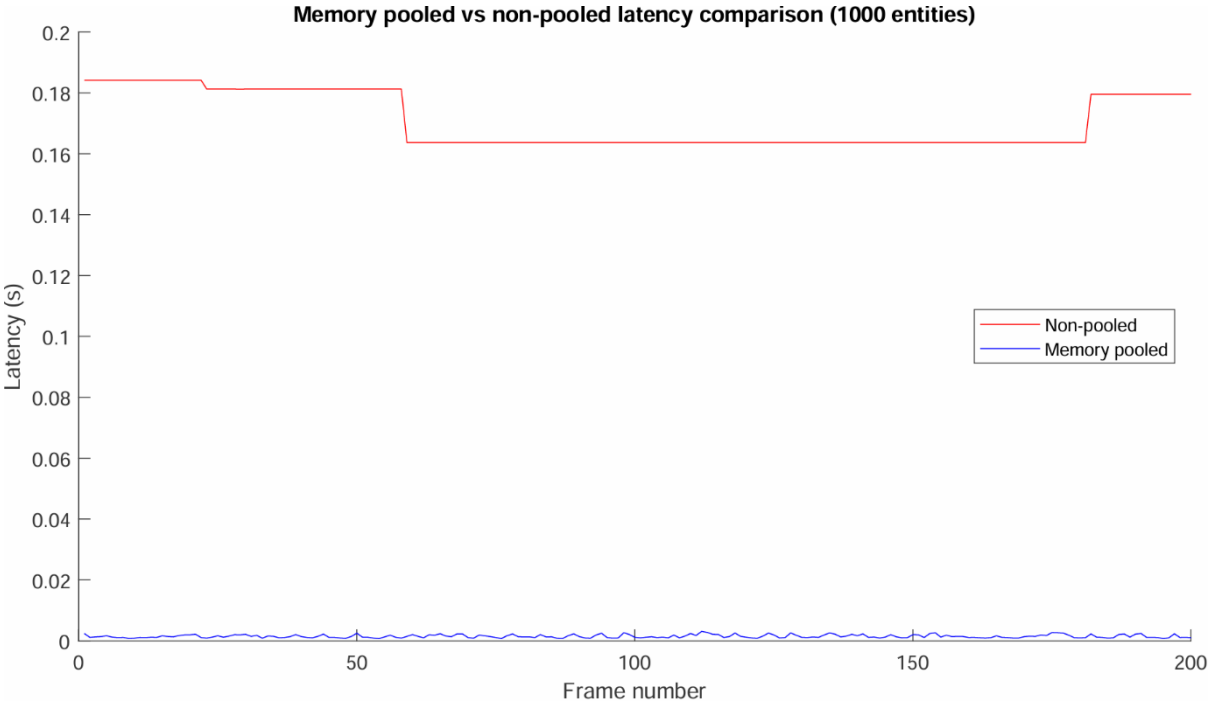

Figure 10

Figure 10 compares the frame latency between a memory pooled and non-memory pooled implementation of ECS with 1000 entities. The non-memory pooled ECS implementation (red) has significantly higher latency overall which is consistently around ~0.18s. The implementation experiences the occasional drop in latency but overall remains inefficient. Compared to the 100 entities test, latency has increased drastically. On the other hand, the memory pooled implementation (blue) consistently has lower latency typically staying below 0.0025s. There are a few fluctuations however it is relatively stable throughout.

Hypothesis Test Summary

	Null Hypothesis	Test	Sig.[a,b]	Decision
1	The distribution of FRAMETIME is the same across categories of GROUP.	Independent-Samples Mann-Whitney U Test	.000	Reject the null hypothesis.

a. The significance level is .050.

b. Asymptotic significance is displayed.

Independent-Samples Mann-Whitney U Test

FRAMETIME across GROUP

Independent-Samples Mann-Whitney U Test Summary

Total N	400
Mann-Whitney U	40000.000
Wilcoxon W	60100.000
Test Statistic	40000.000
Standard Error	1138.659
Standardized Test Statistic	17.565
Asymptotic Sig.(2-sided test)	.000

Figure 11 A

Figure 11 A is a Mann-Whitney U Test to determine if the distributions of frame times significantly differ between the two groups. Mann-Whitney U: 40,000 Wilcoxon W: 60,100 Test Statistic 40,000 Standard Error: 1138.659 Standardised Test Statistic: 17.565 P-value: 0.000. As the p-value is below 0.05, we reject the null hypothesis. This means that there is a statistically significant difference between the two implementations.

Report

FRAMETIME

GROUP	Mean	N	Std. Deviation
1.00	.0011	200	.00060
2.00	.1706	200	.00883
Total	.0858	400	.08511

Figure 11 B

Figure 11 B is statistical evidence that the memory pool implementation significantly improves frame time performance. Memory pool implementation has a mean frame time of 0.0011s with a standard deviation 0.00060s which indicates frame times on average is lower. It also exhibits low variability suggesting consistent performance. The non-memory pooled implementation has a mean frame time of 0.1706 with a standard deviation of 0.00883s. As the deviation is higher this means that the performance is more inconsistent.

Discussion and Analysis

Figure 8 provides a direct comparison of latency times between memory pooled and non-memory pooled ECS implementation for 10 entities. The result highlights the performance advantages of memory pooling by showing the clear difference in frame processing times.

The non memory pool ECS implementation consistently exhibits higher latency with occasional spikes. These could be due to frequent dynamic memory allocations. The uneven pattern also indicates that fragmentation is taking place as the data (components) are not stored together.

The memory pool ECS implementation remains with a consistently lower latency. However, the implementation does show more frequent oscillations which could be caused by ECS processing. Overall, the performance remains more predictable and optimised.

Figure 9 again compares latency times between the same two ECS implementations. However, this time the entity count tested was 100 entities. These results further highlight the performance boost from using a memory pool.

The non memory pool implementation is once again exhibiting higher latency. The step like behaviour suggests that fragmentation could be playing a role in effecting frame processing. The lack of significant variation shows that while latency stays predictably high it's not encountering sudden jumps in latency, like *Figure 8*.

On the other hand, the memory pooled implementation maintains a much lower latency generally staying below 0.005s. however, some fluctuations do exist but suggest some natural frame to frame variations. Still staying well below the non-memory pooled counterpart.

Figure 10 further provides direct comparisons between the two implementations. This time the sample size was 1000 entities for only 200 frames. As the figure shows, the performance gap between memory pools and no memory pool has increased drastically.

The non-memory pooled implementation shows a significant latency increase again, averaging at 0.18s per frame up from 0.015s from the 100 entity test. As the entity count grows it confirms the suspicion that memory fragmentation is in fact the leading cause for the high latency within the ECS implementation.

The memory pool implementation has minor fluctuations, but the overall trend tends to stay stable. The results suggest that the pooling effectively prevents fragmentation, allowing entity updates to be processed quickly and effectively without the overhead of memory allocations.

After judging the trends of the comparative graphs, its clear to see a trend where the memory pool implementation only gets more efficient with more entities. Figure 10 specifically provides the critical importance of memory pooling for handling large numbers of entities. This provides the premise that without memory pools, ECS implementation will struggle to maintain real time performance in large scale applications.

To further back this case up, a statistical test between 1000 entities without a memory pool and 10000 with a memory pool has been conducted.

Report

FRAMETIME

GROUP	Mean	N	Std. Deviation
1.00	.0011	200	.00060
2.00	.1706	200	.00883
Total	.0858	400	.08511

1.00 is the memory pool implementation and 2.00 is the non-memory pool implementation.

These results in the statistical test highlights a substantial performance difference between the two implementations. The memory pool implementation (Group 1) indicates that it has both speed and consistency. Conversely, the non-memory pool implementation (Group 2) suggests that the test was 150 times slower and a greater inconsistency in performance. This statistical test demonstrates that memory pooling is a highly effective optimisation technique for a sparse set ECS implementation. Such improvements are critical when developing games and real time applications. This should provide overall a smoother and more responsive user experience.

Overall, the project confirms with clear evidence that memory management plays a critical role within an ECS implementation. Throughout the performance benchmarks, memory pooling significantly gained the upper hand especially as the entity count increases. Through out the tests the memory pooling implementation retains a consistent and predictable performance unlike the non-memory pooled implementation. The noticeable degradation in iterations could be due to fragmentation of memory.

The key take away from this investigation is that by not having any memory management within the ECS architecture, it directly affects the scalability of the system developed. Reducing the frequency of heap allocations means that memory pooling leads to more stable execution times, not only adhering to date orientated programming principles but also ECS principles.

Besides the performance gains, the investigation also reinforced the idea that memory management directly contributes to the way ECS systems are designed and implemented. By incorporating memory pools into the system, developers gain finer control over memory usage potentially leading to better memory usage predictability.

The investigation confirms that memory management is not just an optional optimisation but a necessary one, as research by Wingqvist, Wickström, and Memeti (2022) demonstrates that data-oriented design relies on effective memory management to yield significantly lower latency. To back this up further Jason Gregory (2014), has argued that that efficient memory management is fundamental to game engine architecture, as frequent heap allocations lead to fragmentation and cache misses with drastically decreases the performance of an ECS system.

Without a proper memory handling even well structured ECS implementations can suffer from severe bottle necks. These finding are particularly relevant when it comes to developing games and simulations where performance is critical.

Despite the project providing insights into memory management within an ECS context, the question on if the project successfully met the objectives is still not answered. To figure this out, comparing the project to the objectives set out earlier is necessary.

RQ1. What is the current state of ECS related research specifically with ties to memory pool usage in optimising ECS?

The literature review conducted as part of this project highlights that ECS is a well integrated paradigm within industry. However as discussed ECS research, particularly in aspect of memory management lacks development. To mitigate this missing information research into memory management as a whole to support the advantages of memory pooling within an ECS setting.

Many ECS frameworks such as Vico also acknowledge the importance of memory efficiency even if not directly. However, directly comparing the impact of memory pooling versus standard allocation strategies within a sparse set ECS implementation remains limited. This study aimed to bridge this gap by highlighting the gaps in knowledge left by other papers.

RQ2. Does memory pool adaptation improve the performance of a sparse set ECS implementation?

The experimental results do in-fact confirm that memory pooling significantly improves the performance in a sparse set implementation. The benchmarks demonstrated that memory pools resulted in consistently lower latency. This is especially prominent with in scenarios with higher entity counts.

The results gathered does align with the memory management theories, particularly those related to cache efficiency and fragmentation. The research states that frequent dynamic allocations can lead to fragmentation and cache misses. In contrast, memory pooling provides a dedicated block of memory and reuses them effectively.

Studies also support the ideology that reducing heap allocations improves overall performance due to excessive heap usage disrupts cache coherence. The results found in the tests further backed up these findings as the memory pool ECS implementation was more stable and predictable compared to the non-memory pooled version.

Research into game engine development suggests that custom memory allocators, including memory pools, are widely used to mitigate memory performance bottlenecks. Which, the results do in-fact back up by showing measurable benefits when applied to an ECS implementation.

Ultimately, the principles discussed in the literature provide a strong support for the findings of this project. Furthermore, there is no critical contradictions between the findings or existing theory.

The most plausible explanation for the observed performance improvements is the fact that the memory pool is actively helping in reducing the number of memory allocations during runtime. Usually without a memory pool memory allocations would result in slower access times.

Memory pooling reduces the number of heap allocations by pre allocating blocks that are reused throughout the applications life cycle. This in turn minimises fragmentation and ensures that the memory remains contiguous. As stated before, this becomes more expressed as more entities get added to the system.

As the memory is stored in contiguous blocks the CPU is able to fetch the data much quicker reducing the need to fetch from the RAM. The experiments result clearly show lower variance in execution times for the pooled implementation.

Even though the results strongly suggest that memory pooling improves ECS performance, there are some potential criticisms to cover. The main criticism is that the study only focuses on latency as the main performance factor. Additional factors such as memory size and scalability of the system should also be examined over runtime to provide more metrics on the implementation.

Another possible criticism is that the implementation of either the non-memory pool or the memory pooled could affect the results. Different memory pool strategies such as fixed or variable might yield varying performance. The specific design choices from this project may not generalise all ECS implementations.

Furthermore, these findings are based on specific hardware specifications. Variations in CPU size and memory could lead to different results. To further expand upon this the results of this test seem to be largely dependent on the architecture of the machine testing with. There could also be some variation based on what compiler the user decides to use. With this in mind the flags and the compiler settings that have been implemented for one implementation may not work under another implementation.

Despite these criticisms, the finding still remains robust providing a strong argument for the inclusion of memory pooling in ECS optimisation. Addressing these limitations in further research could refine the conclusions drawn.

Several factors could affect the results of the project. As previously stated, hardware specifications such as CPU and memory play an important factor in determining the use of memory pools. A system with a larger CPU could potentially see less of an effect than a computer with a smaller CPU.

The ECS implementation itself could affect the results of the tests. The project is designed around sparse set implementation which typically benefit from improved memory locality. (Wingqvist, D., Wickström, F. and Memeti, S., 2022) However, an implementation such as an archetype model would cause the memory pool impact to potentially be different.

The ECS implementation being a stand-alone system could have caused unrealistic results. As the workload is mainly focused on creation, deletion and iteration of entities, if a different use case was implemented performance may have seen greater issues.

Lastly, compiler optimisations and memory allocation strategies employed by the operating system might have influenced the measurements. Some compilers apply aggressive optimisations that could mask or exaggerate the benefits of what has been implemented.

The projects results could very well vary under different conditions. For example, if the entity count was significantly lower the benefits of the memory pool would not be as pronounced as the overhead could undermine them. On the other hand, a test will higher entity counts could show the advantages of memory pooling better.

Different ECS implementations might also influence the results. While sparse sets benefit from memory pools, archetype ECS implementations already optimise for their cache locality.

Additionally on different hardware, as stated would cause the system to vary significantly. Memory prefetching and memory allocation may not be as severe causing less noticeable performance gains.

Conclusion

With the growing need for more performant systems within the games industry, to help develop and create intrinsic worlds. This project set out to find out how memory pools can boost performance of a sparse set entity component system. Given the lack studies within this field, the research aimed to bridge a gap and provide valuable insights into how memory management techniques leveraged ECS architectures.

The project examined existing literature on programming paradigms; procedural, Object oriented, Data oriented and Entity component system. This highlighted the benefits and limitations of each paradigm, demonstrating why ECS is a great next candidate for the games industry to adopt. Specifically, ECS's ability to decouple and couple data while ensuring better performance with cache efficiency.

The study also delved into memory management techniques, especially focusing on memory pools. Information led to an understanding of how memory pools pre allocate a dedicated contiguous block of memory which proceeds to be reused. This reduces fragmentation and improves cache coherency. By implementing these into a sparse set ECS implementation, this project sought to measure the predicted improvements in performance.

Through the development of a custom ECS implementation, the research provided a theoretical demonstration of how ECS operates. The implementation was designed to be flexible, allowing dynamic allocation and removal of components at runtime. Benchmarks were conducted to analyse the impact of the memory pools on the frame latency of the system. This measured a generalised overview of the memory allocation speed and cache efficiency.

The findings ended up showing that memory pools offer substantial improvements in performance. The results showed that specifically that as more entities became involved, the performance boost that the memory pool provided was significantly higher. This showed that the memory pool did in-fact help with removing fragmentation within the memory.

The results of this research could have important implications of game developers. As games become more complex and require greater computational power, the need for optimised memory management is amplified. With the integration of memory pools into ECS, yields significantly higher performance (as shown in *Figure 11*). This is important when developing games with high entity counts.

The main take away from this research is that ECS on its own, although powerful, still requires memory management to be taken full advantage of. This project sets in motion that when implementing ECS memory management is a must.

Additionally, this research contributes to the wider field of ECS architecture by highlighting the key importance of memory management. While industry is adopting the ECS architecture, best practices for memory management within it is scarce. The project aimed to serve a stepping stone for a push towards more standardised ECS memory management.

Final thoughts

The increasing demand for high performance systems especially in real time applications such as a games engine, needs to always keep memory management in mind. This research demonstrated that memory pools can significantly improve ECS efficiency while reducing fragmentation and improving performance within entity heavy environments.

As ECS continues to gain traction in both indie and AAA development, understanding optimisation techniques becomes essential for developers. By addressing memory management challenges, future games engines can achieve greater efficiency and responsiveness.

This study represents a fundamental step towards bridging the gap for ECS research, while providing valuable insights for developers. While there is still a lot to explore, the findings provided should offer a solid foundation.

Recommendations

The experimental setup measured frame latency across varying ECS workloads, however a criticism with this experiment could be that std::any does not represent the optimal ECS implementation. Further experiments from this project could be experimenting the effect of alternative implementations of memory pools or using std::variant to further improve performance.

Upon further inspection, some variance of results seems to be present largely dependent on the machine experimented on. Expanding the results to different computers ended up showing that the memory pool was around the same speed or a little bit quicker. This could be caused by the fact that compiler flags where not transferred over or the data was configured for the specific machine the project was worked on.

For further developments, tests could be conducted on different types of ECS. The one tested within this project was a sparse set implementation. Future tests could test for ECS implementations with architypes or both.

One thing that could increase the efficiency of the implementation is multithreading for the ECS system.

References

Wingqvist, D., Wickström, F. and Memeti, S., 2022, November. Evaluating the performance of object-oriented and data-oriented design with multi-threading in game development. In *2022 IEEE Games, Entertainment, Media Conference (GEM)* (pp. 1-6). IEEE.

Eriksson, B. and Tatarian, M., 2021. Evaluation of CPU and Memory performance between Object-oriented Design and Data-oriented Design in Mobile games.

Mazaitis, D., 1993. The object-oriented paradigm in the undergraduate curriculum: a survey of implementations and issues. *ACM SIGCSE Bulletin*, *25*(3), pp.59-63.

Härkönen, T., 2019. Advantages and Implementation of Entity-Component-Systems.

Gupta, D., 2004. What is a good first programming language?. *Crossroads*, *10*(4), pp.7-7.

Chen, T.F. and Baer, J.L., 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, *44*(5), pp.609-623.

Ha, T.H., 2022. Game development with Unreal Engine.

Christopoulou, E. and Xinogalos, S., 2017. Overview and comparative analysis of game engines for desktop and mobile devices. *International Journal of Serious Games*.

Gregory, J. (2018) *Game engine architecture*. 3rd edition. Boca Raton, FL: A K Peters/CRC Press, an imprint of Taylor and Francis (p. 608).

Pokkunuri, B.P., 1989. Object oriented programming. *ACM Sigplan Notices*, *24*(11), pp.96-101.

Buchanan, M., 1994. Overloading and Polymorphism in the interpretation of Inheritance in C++.

LeBlanc, M., 2017. Gane Entities in Thief: The Dark Project. Available at: https://www.youtube.com/watch?v=5di7jmHKAQs (20th February 2025)

Choparinov, E.D., 2024. *A Graph-Based Approach To Concurrent ECS Design* (Doctoral dissertation, Universiteit van Amsterdam).

Hatledal, L.I., Chu, Y., Styve, A. and Zhang, H., 2021. Vico: An entity-component-system based co-simulation framework. *Simulation Modelling Practice and Theory*, *108*.