February 25, 2025

Louis Cox

Benjamin Williams

University of Staffordshire

ECS Comparison: Sparse Sets vs Archetypes

GDEV60001 Games Development Project

**Contents**

[1 Introduction 2](#_Toc191329634)

[2 Aims and Objectives 3](#_Toc191329635)

[3 Literature Review 3](#_Toc191329636)

[3.1 Object Oriented Programming 3](#_Toc191329637)

[3.2 Data Oriented Design 4](#_Toc191329638)

[3.3 Entity Component Systems 4](#_Toc191329639)

[3.3.1 History 5](#_Toc191329640)

[3.3.2 Sparse Set Based 5](#_Toc191329641)

[3.3.3 Archetype Based 7](#_Toc191329642)

[3.3.4 Hybrid Component Storage 8](#_Toc191329643)

[3.3.5 Component Communication 8](#_Toc191329644)

[3.3.6 Current Literature 9](#_Toc191329645)

[4 Research Methodologies 9](#_Toc191329646)

[4.1 Implementation 9](#_Toc191329647)

[4.1.1 Sparse Set Based 9](#_Toc191329648)

[4.1.2 Archetype Based 10](#_Toc191329649)

[4.2 Benchmarking 10](#_Toc191329650)

[5 Results and Findings 11](#_Toc191329651)

[5.1 Frame Update Times 11](#_Toc191329652)

[5.1.1 100 Entities 11](#_Toc191329653)

[5.1.2 1000 Entities 11](#_Toc191329654)

[5.1.3 10000 Entities 11](#_Toc191329655)

[5.1.4 50000 Entities 12](#_Toc191329656)

[5.2 Entity Setup 12](#_Toc191329657)

[5.3 Summary of Findings 13](#_Toc191329658)

[6 Discussion and Analysis 13](#_Toc191329659)

[6.1 Performance Trends and Scalability 13](#_Toc191329660)

[6.2 Practical Implications 14](#_Toc191329661)

[6.3 Potential Optimisations 14](#_Toc191329662)

[7 Conclusion 15](#_Toc191329663)

[7.1 Achievement of Research Aims 15](#_Toc191329664)

[7.2 Benefits of Research 15](#_Toc191329665)

[7.3 Limitations of Research 16](#_Toc191329666)

[7.4 Final Thoughts 16](#_Toc191329667)

[8 Recommendations 16](#_Toc191329668)

[9 References 18](#_Toc191329669)

**Abstract**

Entity Component System (ECS) architecture has emerged as a powerful alternative to traditional object-oriented programming in game development and real-time simulations. By decoupling data from logic and optimising memory access patterns, ECS enables improved performance and scalability. However, different ECS implementations present distinct trade-offs in iteration speed, entity modification costs, and memory efficiency. Despite its growing adoption, comparative research on the performance characteristics of different ECS storage architectures remains limited.

This study provides a direct performance comparison of the two most widely used ECS architectures: Sparse Set-based and Archetype-based implementations. To facilitate this, a C++20 implementation of each architecture was developed, and their performance was measured under identical conditions. The benchmarking process focused on two key metrics: iteration speeds, and entity modification costs.

The findings reveal that Sparse Set ECS offers reduced entity modification costs but struggles with iteration efficiency, particularly as entity counts increase. Conversely, Archetype ECS exhibits superior iteration performance at larger scales, benefiting from improved cache locality, though it incurs higher costs when modifying entity compositions.

By analysing these trade-offs, this research provides critical insights into the scalability and efficiency of each approach, helping developers make informed decisions when selecting an ECS architecture.

# Introduction

The Entity Component System (ECS) pattern is an increasingly popular architecture used predominantly for real-time simulations such as games. It offers a highly flexible and performant data-oriented alternative to the traditional object-oriented architectures used for game development. Instead of Object Oriented Programming (OOP), which organises its data and logic around objects and class hierarchies, ECS adheres to Data Oriented Design (DOD) principles. This primarily means decoupling data and logic for improved scalability, modularity, and considerable performance benefits. By structuring data to take advantage of modern CPU architectures, ECS can enable more efficient memory access, improved cache coherency, and simpler parallelisation logic. (Webster, 2015)

The adoption of ECS has grown significantly in recent years, with the paradigm influencing both proprietary and open-source game engines. Major industry players, including Unity with its Data-Oriented Technology Stack (DOTS), Mojang with its Bedrock Edition of Minecraft, and Blizzard with their popular multiplayer hero shooter Overwatch (Ford, 2017), have integrated ECS-based architectures to optimize game performance. Additionally, open-source ECS frameworks such as EnTT[[1]](#footnote-2), Flecs[[2]](#footnote-3), and Bevy[[3]](#footnote-4) have demonstrated the flexibility of the ECS paradigm across different languages and development environments. Despite these advancements, ECS is not a one-size-fits-all solution. The performance and usability of ECS implementations vary based on how they store and process component data, and selecting the right approach requires an understanding of the trade-offs inherent to different ECS architectures.

Whilst the same basic principle of decoupling data and logic applies to all ECS architectures, the difference lies in how data is stored. The details of how entities and components are stored can dramatically affect cache coherency, iteration speed, and modification costs. As a result, ECS implementations often fall into different categories based on their approach to component storage and retrieval, leading to ongoing discussions in industry about which strategies best suit different types of games and simulations.

Despite the growing use of ECS, there remains a lack of consensus on the optimal way to structure and manage component storage, leading to fragmentation in how different engines and frameworks implement the pattern. Developers must often make trade-offs between iteration speed, memory efficiency, and flexibility without a clear, standardized methodology for determining which approach is best suited for their needs. This uncertainty makes it crucial to analyse and compare different ECS storage architectures to better understand their strengths, weaknesses, and ideal use cases. Without this understanding, developers risk implementing inefficient ECS solutions that fail to fully leverage the performance benefits of data-oriented design, leading to unnecessary bottlenecks or overly complex architectures. As ECS continues to gain traction, a more refined understanding of its internal mechanics is necessary to ensure its effective application in modern game development. Insights drawn from this study could help to inform developers of best practices when it comes to implementing ECS themselves, as well as forming a base for future work into this underexplored area of research.

# Aims and Objectives

The aim of this thesis is to provide insight into the differences between the two most widely used ECS architectures: sparse set based and archetype based implementations. While both approaches adhere to the core principles of data-oriented design and aim to maximise performance by optimising memory access patterns and reducing cache misses, they do so using fundamentally different storage models. These architectural differences introduce trade-offs that impact iteration speed, memory efficiency, and entity modification costs. This study seeks to establish a clearer understanding of these trade-offs by benchmarking both approaches in isolation, offering developers a more comprehensive perspective on their performance characteristics.

To achieve this goal, the research is structured around the following key questions:

1. How does each ECS perform during iteration? Is there a notable difference in the update times demonstrated by the two architectures?
2. How does each ECS handle entity composition changes? What are the relative costs associated with adding or removing components, and how does this impact the efficiency of entity setup in a large-scale simulation?
3. How well does each ECS scale? Does performance degrade at higher entity and component counts, and is there an optimal simulation size for each implementation?

By addressing these questions, this research will contribute valuable insights into ECS performance, aiding developers in selecting the most suitable architecture for their game engines and simulation frameworks. Additionally, the findings will help inform potential optimisation strategies that could further enhance ECS efficiency in real-world applications.

# Literature Review

## Object Oriented Programming

Object Oriented Programming (OOP) is a programming paradigm that divides a program into ‘objects’ – self-sufficient units of data and code that can interact with each other (Rentsch, 1982). The core idea is a separation of concerns, called encapsulation, which has often been cited as being important in creating clean, maintainable source code (Snyder, 1986). Objects do not need to know everything about other objects, only how they interact with them; this idea means OOP code is highly modular and maintainable. At a conceptual level OOP also seems a perfect fit for game development, as games in their most basic form are just simulations of interacting entities, the player usually being one of them. This clear conceptual mapping between game entities and code objects is likely a large part of why OOP is the industry standard for game development (Gregory, 2014).

Another key pillar of OOP that proves valuable for game development is inheritance – the idea that classes can inherit from other classes, using them as a base and expanding on them. In this way classes can be extended to fit a variety of use cases incrementally without modifying existing code (Taivalsaari, 1996). This means that code can be easily shared between objects that require the same logic but can result in deep class hierarchies that become difficult to maintain and test (Ryant, 1997).

A common method to avoid these issues employed by modern game engines such as Unity is the ‘component pattern’ (Gold, 2004). In this pattern, game objects simply become containers for components, and each component is responsible for their own functionality. Where inheritance models an ‘is a’ relationship between classes, the component pattern utilises ‘has a’ relationships. This approach makes it easy to share code between objects whilst avoiding the pitfalls of inheritance. It also adheres to OOP’s principle of encapsulation, decoupling unrelated functionality. Decoupled communication between components is still entirely possible using the observer pattern (Nystrom, 2011).

This approach was popularised in 2002 when Scott Bilas delivered a talk outlining the object system for Dungeon Siege (Bilas, 2002). The idea behind this implementation was to put as much power into the designer’s hands as possible by loading components and entities from a database that could be edited with little to no programming knowledge.

## Data Oriented Design

An important alternative programming paradigm is Data Oriented Design (DOD), which is oriented around organizing data for efficient processing. This means that DOD is a paradigm that changes with the hardware it is being applied on, but for the average modern computer it means that data is organised to minimise CPU[[4]](#footnote-5) cache misses.

In 1965 Gordon E. Moore (co-founder of Intel), observed that the number of transistors on a computer chip doubles every two years with minimal cost increase (Moore, 1965). Moore’s Law resulted in CPUs becoming exponentially faster over time, meaning the average computer’s bottleneck today has become memory access speed. Later than same year Maurice Wilkes conceptualised a solution to this problem (Wilkes, 1965) and in 1968 IBM[[5]](#footnote-6) introduced the /360 Model 85, with the first cache memory as we know it today based on Wilkes’ paper. Cache memory is a very small store of memory inside the CPU with faster access time than main memory by a factor of 5 to 10 (Hossain, et al., 2015). When a program attempts to access a variable, the CPU will first check cache memory for it. If it can be found in cache memory, that is referred to as a ‘cache hit’, otherwise it is a ‘cache miss’. (Acton, 2014)

In the event of a cache miss a ‘cache line’ containing that targeted variable will be loaded from RAM[[6]](#footnote-7) into the cache. This means that data stored close to the initial variable is now also available in cache memory. By storing data that is accessed at the same time continuously in memory, it can be ensured that the data loaded into cache lines is relevant to the current logic, maximising cache hits and improving memory access times greatly (Nyberg, 2021). This is the core principle behind modern data-oriented design and is what enables it to outperform OOP even with comparable runtime complexity[[7]](#footnote-8) in its operations.

Another benefit of DOD in the context of game development is that whilst code may only be provided by programmers, data can come from designers, artists, or anybody else involved in the game development process. Orienting systems around data allows for those without programming knowledge to have greater freedom without requiring programmer input (Gregory, 2014).

## Entity Component Systems

ECS combines the architectural benefits of the component pattern with the performance of DOD. It does this by removing the logic from components and storing the pure data in cache-friendly data structures, which results in far quicker memory access and improved performance. ECS implementations vary, mostly in the way they store their components, but all have the same three core elements.

**Entities**: usually only a unique identifier or handle that can be used to index components.

**Components**: the pure data associated with an entity. For example, a position component could hold an X, Y, and Z coordinate. There is no logic associated with components.

**Systems**: the logic that operates on all entities with the corresponding components. For example, a physics system could iterate through all objects with the position, rotation, and rigidbody components and alter their values accordingly.

Contrary to OOP, ECS focuses on composition over inheritance (Webster, 2015). Instead of objects inheriting from other objects to gain their functionality, entities simply use the components with the required functionality. This prevents the complicated class hierarchies and tight coupling associated with inheritance. This approach also proves far more modular than a standard OOP architecture, as systems and components are entirely decoupled from and unaware of one another (Härkönen, 2019).

An important benefit of this approach is simplified parallelisation. For systems where each iteration over the array of compatible components happens without an impact on the following iterations, it becomes simple to dispatch separate threads to tackle them each separately. This approach can aid in avoiding standard multithreading pitfalls such as race conditions or data corruption.

### History

The first known implementation of an ECS in a game engine is the Dark Object System (DOS), created for 1998’s Thief: The Dark Project (LeBlanc, 2017). As previously discussed, the primary difference between ECS implementations is the method through which component data is stored. The DOS had an interesting approach to this issue. Each component, called properties here, is a map linking an entity to a value; the key difference being that the map implementation changed based on the needs of the property. For example, a common property used by most entities could simply use IDs to index an array, where a less frequent property could use a hashmap to limit wasted storage for in exchange for slower lookup times and decreased cache coherency.

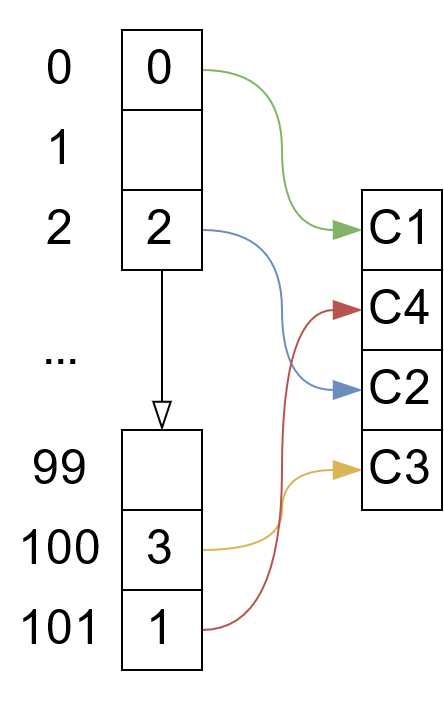
The DOS also implemented a form of inheritance, with some classes inheriting from an archetype and gaining all their properties. On top of this there was a concept of metaproperties, which were groups of properties that could be applied to objects. Whilst the archetypes modelled an ‘is a’ relationship between objects and objects could only inherit from one archetype; metaproperties used a ‘has a’ relationship, allowing for multiple. This idea brought with it many of the architectural issues with standard OOP inheritance but did come with one important benefit: performance. Archetype searches were cached at runtime, making it highly performant for systems to iterate over them.

In 2007 Adam Martin popularised the modern idea of an ECS (Martin, 2007). He put forth a guideline on the pattern which separated entities, components, and systems. This differed from prior implementations such as Dungeon Siege, which were closer to the component pattern in that their components contained functionality as well as data. Whilst the ideas here are largely relevant to any ECS, Martin’s discussion was focused on application in multiplayer games, and as such his ideas naturally gravitated to using a relational database to store components.

### Sparse Set Based

In ECS the decision with the most dramatic effect on performance is how components are stored. This decision is what determines the cache coherency of the architecture, and as such the performance. One option would be to use a standard array for each component, but the issue there becomes the potential for large empty spaces in arrays for uncommon components, which results in wasted storage and lost cache coherency. Another option would be a map of entity ID to component, but there is no guarantee map values are stored continuously in memory, and the hashing overhead for lookups on a map would quickly accumulate.

One solution to this issue is sparse sets. (Briggs & Torczon, 1993) The sparse set is a data structure of two arrays, the sparse list and the dense list. The dense list is what stores the component data, and the sparse list maps the entity ID to the component. This is represented in Figure 1, in which the indices to the left represent an index into the sparse set, the integers shown in the set represent indexes into the dense list, and C1, C2, and so on represent component data. Although represented as blank cells here, indices in the sparse list with no corresponding component will usually contain a flag value such as -1 or the maximum value of the integer type used.



Figure

This approach ensures that components are packed tightly in memory (Redmond, et al., n.d.), but can still be searched in constant time quickly without lookup overhead from hashing. It also limits wasted data, as the sparse list now only needs to save space to store unsigned integers as opposed to storing whole components.

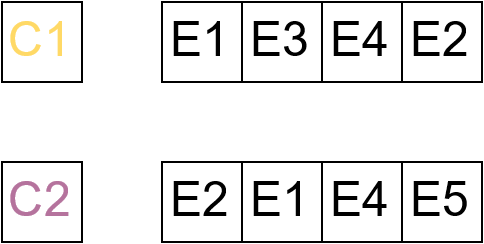
Adding components to this data structrue is relatively easy, components are simply added to the end of the dense list and the gap at the appropriate entity index is filled with the new component’s index in the dense list (Mroz & Pietkiewicz, 2023).

Accessing components is also simple. The entity index you wish to query is checked in the sparse list, and if the value is not null for that entity a pointer to the component at that index in the dense list is returned .

Removing entities from a sparse set however is a little more complex. If the entry you wish to query is simply removed in the dense list it would result in gaps throughout, effecting cache coherency. Instead, we first swap the component at the end of the list with the component we want to delete, before updating the indexes stored in the sparse list to match and deleting the desired component.

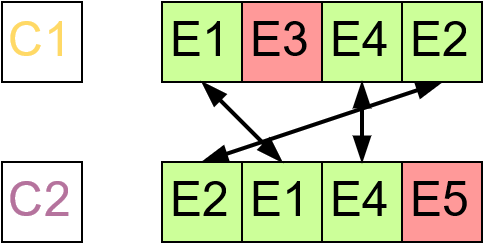
We can also optimise on this further. Whilst wasted space is limited by the sparse set only storing a simple index, in situations with lots of entities that can still add up, so we can employ a concept known as pagination. Because the cache coherency of the sparse list doesn’t matter, we can divide it up into ‘pages’, only actually creating arrays for the pages required. Theoretically this could result in larger sparse sets due to the memory overhead of 2D arrays, but on average the memory saved when not tracking unnecessary pages far outweighs this cost. With this approach, instead of accessing the sparse set directly, we access the page first using the index divided by the maximum page size, and access the desired point in that page using the remainder of the division.

There is one notable issue with the sparse set implementation. In a sparse set based ECS using two component types, 1 and 2, the data would likely stored as seen in Figure 2, with each component type (C1 and C2), holding an array of component data for each entity (E1, E2, and so on).



Figure

For accessing one component type at a time this memory structure is ideal, but a lot of systems will search for entities with multiple matching components. In a situation like this, a sparse set ECS will iterate over the entities in the shortest component list, checking each of them for the other requested components. For complex queries, this can become very slow. Additionally, components not matching the target query will take up space in cache memory regardless of whether the data is used. This can be seen in Figure 3, in which the red cells represent components unnecessarily evaluated, and the green cells represent components from entities matching the query. It can also be noted that the sparse set implementation does not guarantee component order, meaning that whilst iteration over the smaller list will be sequential and cache cohesive, it is likely that elements requested from other sets could be stored far apart from the last element referenced, making it more likely to cause a cache miss.



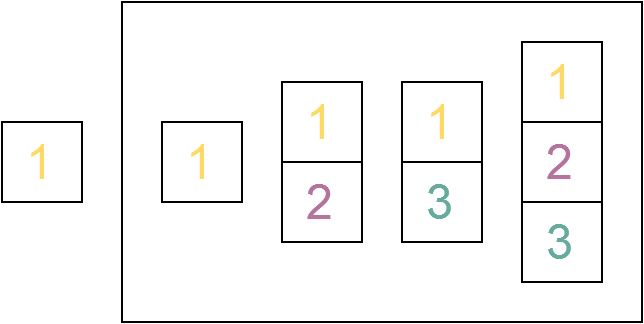
Figure

One popular implementation of this architecture is EnTT, a lightweight open-source header only C++17 ECS library, notable used by Mojang for the Bedrock version of Minecraft. EnTT implements a concept it called groups to help speed up iteration times (Caini, 2019). Essentially, grouping two component types means that when an entity has components of both types, they are moved to the front of the list, and shared entities are placed in the same order in both component sets. This means that instead of iterating over the entire list to check for matches, systems requiring a predefined group only need to search to the final matching entity in the two sets. This does add some overhead when removing or adding components from entities that are in a group, as the other components in that group will have to be reordered to match the change.

### Archetype Based

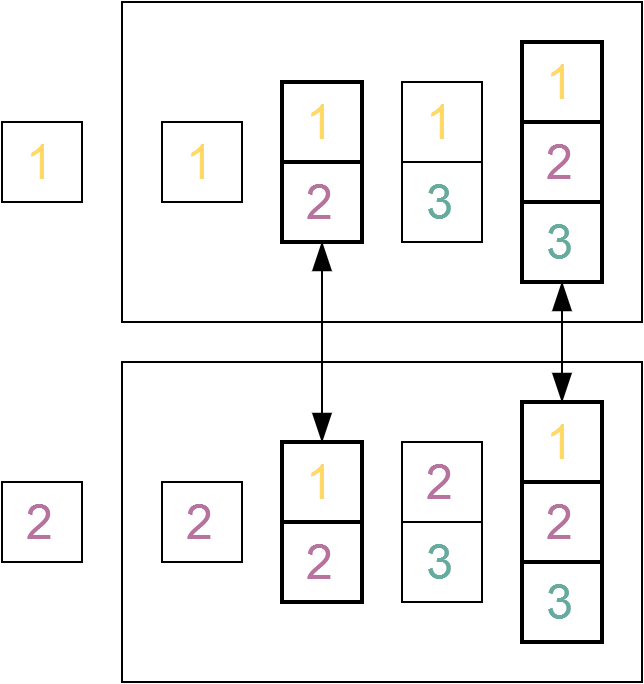
Archetype based ECS was designed to combat the sparse set approach’s primary drawback: slow iteration on multiple component type systems. Components are no longer stored in individual arrays, but are instead grouped into arrays of entities that share the same components, called archetypes. This means that when iterating over an archetype matching the components being queried, no check is required to see if each entity has the correct components. This is because only archetypes that already garauntee the presence of these components are included in iteration.

This type of ECS is particularly suitable for cache coherency, but does face another issue. When looking for entities with components 1 and 2, it is also necessary to find entities with components 1, 2, and 3. To solve this issue, each component keeps a list of pointers to the archetypes it is used in, as seen in Figure 4, which represents the archetypes relevant to component 1.



Figure

Now, when searching for entities with multiple components matching a given query, their archetype lists may be accessed and an intersection operation using them will return the archetypes matching the query. This approach makes it easy to write complex entity queries that include or exclude desired or undesired components as required, all whilst maintaining impressive cache coherency. In Figure 5 the query can be seen, which will return archetypes [1, 2] and [1, 2, 3].

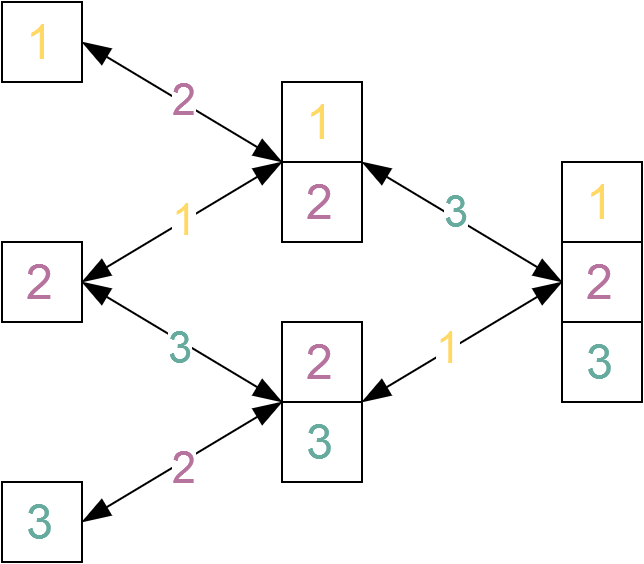


Figure

There is, however, an important downside to this approach. Due to the complexity of moving component data from one archetype to another, adding or removing components or entities at runtime can become quite costly. This is because all component data associated with the entity must be moved to the array of its new archetype. Because of this, it is best advised to batch any edits to entities to be performed at the end of an update cycle.

Another considerable factor in the speed of adding or removing components is the cost of finding the new archetype of the entity. The initial system proposed suffers hashing overhead when searching the hashmap for an appropriate archetype, and as such running it for every entity with a change in components can be very costly. Fortunately, there is something that can be done to offset this drawback. Sander Mertens, author of the FLECS library, has put forth a system called the ‘archetype graph’ (Mertens, 2020).

The structure Sanders proposes tracks archetypes in a graph, where edges represent the component difference between them, meaning we can quickly find the new archetype of a changed entity by simply following the edge of the component that has been added or removed. Because many components will not be changed at runtime, this graph can be created lazily on the fly, effectively caching the results of the first search of each entity change, allowing the same change to be performed far more quickly in the future. An example of the archetype graph can be seen in Figure 6.



Figure

One popular archetype ECS implementation is Unity’s Data Oriented Tech Stack (DOTS), which is an optional data-oriented framework build on top of Unity’s normal OOP systems. DOTS is designed with parallel processing in mind and integrates seamlessly with Unity’s ‘jobs’ system to allow for simpler parallelisation logic. Unity also carries its concept of enabled and disabled components into its ECS, which functionally allows for components to be added and deleted at runtime without suffering the same performance hits. The drawback of this is the slight overhead incurred by checking if components are enabled during a query, which is why only components with the appropriate interface are checked. This allows programmers to select only the components that require frequent enabling and disabling, leaving unrelated systems unimpacted.

### Hybrid Component Storage

We now have two popular ECS variations with their own advantages and drawbacks. Archetypes allow for fast iteration but slow editing of entities, whereas sparse sets offer the reverse: faster editing of entities in exchange for slower system iteration. Bevy is an open source Rust based ECS framework that puts forth a unique solution: using both (Anderson, 2021). By default, the system makes use of an archetype based ECS, however when registering a component with the ECS core, users are given the option of storing it using sparse sets. Queries involving only archetype or sparse set based components are treated as their architecture demands, but queries involving both types must be treated differently. Firstly, the archetype based components are queried as normal, and then the sparse set based components will be iterated over, checking the returned archetype list for matching entities. This system means allows for components commonly added and removed at runtime, such as temporary tags, to make use of the fast editing times of a sparse set architecture without impacting the iteration speeds of unrelated systems; much like Unity’s concept of disabled components.

### Component Communication

In games different entities are often required to interact with each other. In OOP the observer pattern is commonly utilised to make that possible, with objects waiting for triggers from other objects, such as collisions or inputs, before acting. From an architecture perspective this approach is clean and simple, allowing for decoupled communication and modularity; however, from a performance perspective there is an issue. In OOP indirection is rarely considered as a factor and moving between objects in inconsequential. However, with DOD we aim to take advantage of the information currently stored in the cache before moving to another area of memory. Since it is almost a guarantee that the observer pattern will cause a cache miss, most ECS architectures make use of an ‘event queue’, essentially batching those cache misses by storing events and calling them together at the end of a frame.

Whilst this is one viable way of approaching the problem, another is to leverage the architecture already created for the ECS. By adding a component to an entity to represent a given event, we can use systems to query for entities with those components to handle that event. This approach maintains our cache coherency and simplifies our architecture, at the cost of the overhead of adding and removing components are runtime. This cost can become quite considerable depending on the component storage methods employed and the frequency of components added or removed. There is also some overhead from the systems added to handle events with this approach, as they will continue polling for events each frame even without any.

It is also possible to combine these two approaches. Bevy does this by implementing the observer pattern to watch for the addition and removal of components; this way it is possible to create event handler system that only run on frames where at least one entity contains the targeted components.

### Current Literature

Following the relatively recent emergence of ECS architecture as an increasingly popular alternative to the traditional object-oriented approaches, papers released have largely focused on finding potential use cases as well as demonstrating the performance advantages of ECS over conventional paradigms. Several studies have established its effectiveness in game engine architecture (Hollmann, 2019), forestry simulation (Williams, et al., 2024), virtual reality (Fischbach, et al., 2017), machine learning (Brennan, et al., 2023), and a multitude of other computing applications.

While these studies have successfully highlighted the power and versatility of ECS, especially in comparison to OOP (Tarifa Mateo, 2024), much of the existing research has been centred on proving its viability rather than analysing the impact of different implementation strategies. There remains a notable gap in literature regarding the comparative performance of various ECS architectures, including optimizations for specific workloads, memory management strategies, and the trade-offs between Archetype-based and Sparse Set implementations. As ECS continues to gain traction in both industry and academia, further research into these areas could provide valuable insights into maximizing its efficiency as well as allowing different implementations to be better tailored to their use case.

# Research Methodologies

To conduct a fair and structured comparison of sparse set based and archetype based ECS implementations, this study relies on performance benchmarks gathered from custom implementations of both architectures. These implementations will be created in C++20 to take advantage of the language’s memory management options and template classes. The benchmarking process is designed to isolate key performance metrics, ensuring an objective evaluation of iteration speed, entity setup efficiency, and scalability across varying entity counts.

By implementing a custom ECS for both variations, this study eliminates external dependencies that could impact results, such as framework-specific optimisations or third-party ECS implementations. This approach provides a clear and direct comparison, enabling a detailed analysis of how each architecture handles entity processing under different workloads.

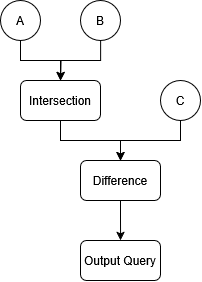
## Implementation

### Sparse Set Based

The sparse set based ECS implementation for this paper will consist of three header files. *“SparseSet.h”*, which will contain the sparse set data structure used to store components; *“World.h”*, which will contain the World class responsible for managing component sets; and *“Query.h”*, which contains the query system which allows the user to access component data.

For this dissertation sparse sets will be implemented as covered in the literature review, including the pagination optimisation. Additionally, an interface abstract class will be used for type erased storage in the world class, as well as granting access to functionality relevant to all component stores, such as deleting a given entity or checking for their existence.

The query system used here will be graph based. A query node abstract class will contain methods to get a list of the valid entities and pointers to the sparse sets it can guarantee the presence of those entities in. A sparse set node abstract class will simply function as a wrapper for a component set, whereas intersection and difference nodes will be capable of comparing entity lists and returning entities that fit the query. For example, in Figure 7 we see a query involving components A, B, and C. The intersection node in this query gives us all entities with both components A and B, whereas the difference node will ensure that no entity returned has component C. The returned query object acts as an iterator that will cycle over the valid entities, getting their component data and passing it into a tuple of references that can be acted on.



Figure

Lastly, the world class will act mostly as a container for component sets, allowing the user to add and remove components from entities and request query objects to iterate over for their systems. The world class will also be responsible for managing entities, which will be purely unsigned integer IDs used to access component sets.

### Archetype Based

The archetype based ECS implementation created for this paper will consist of two main header files: *“Archetype.h”*, which will outline the data storage for each archetype, as well as logic for adding and removing entities; and *“World.h”*, which, much like in the sparse set implementation, acts largely as storage for archetypes and management for entities.

Unlike sparse sets, type erasure here happens directly in the component storage as opposed to coming from an abstract class. Each archetype is initialised using a list of type indexes, which outline the component types that archetype is responsible for. Component data is then stored in a map of type index to void pointer which points to the component vector. These vectors can then be accessed through template functions attached to the archetype, which are used to cast the void pointer to the component type required.

The archetype header file will also contain the definition for an archetype view, which much like sparse set queries will allow us to iterate over component data. Unlike archetypes, archetype views are not type erased, and they contain a reference to the same data as the archetypes, but casted to the correct component type to allow for iteration. Archetype views are also not required to exactly match the components of the archetype they are derived from, but all components used for the view must be available in the archetype. For example an archetype featuring components A, B, and C could have a view referencing only components A and B.

Much like in the sparse set implementation, the world class manages entities and stores the archetypes, providing a clean, central interface for the user to access all ECS data. However instead of returning a query object when a query is made, it returns a world view. A world view simply acts as a container for component views. If the user wishes to query for system A, the world view will collect archetype views for each archetype that also contains component A, allowing for iteration over all relevant archetypes.

## Benchmarking

To evaluate the performance of the different ECS architectures, a controlled benchmarking approach will be used. The primary test will involve running a simulation of John Conway’s Game of Life, a cellular automaton that consists of a grid of entities, each following simple rules to determine their state in the next frame. This simulation is well-suited for benchmarking ECS architectures as it involves frequent component updates, iterative processing of large numbers of entities, and predictable computation patterns, making it an effective way to measure performance in a structured manner.

The performance of each ECS approach will be assessed using two key metrics:

1. Update Performance: The time taken to process each frame of the simulation over 5000 frames. This metric will provide insights into how efficiently each ECS approach handles iteration, cache locality, and component updates.
2. Entity Setup Time: The time required to initialise all entities involved in the simulation. This measurement will help determine the overhead involved in creating and managing entities and components within each ECS architecture.

Using Conway’s Game, each ECS variation will be benchmarked at 100, 1000, 10000, and 50000 entities to highlight how each approach scales at different entity counts, with the widely varied sample sizes helping to provide a more complete picture. All simulations will be entirely single threaded for the sake of simplicity and clarity and all entities will be updated whether relevant or not to ensure a more controlled, level comparison.

By comparing these metrics across different ECS implementations, this benchmark will highlight the strengths and weaknesses of each approach in terms of performance, scalability, and efficiency. The results will provide valuable insights into how different ECS storage models impact real-world simulation workloads, aiding in the selection of the most suitable architecture for various game development scenarios.

# Results and Findings

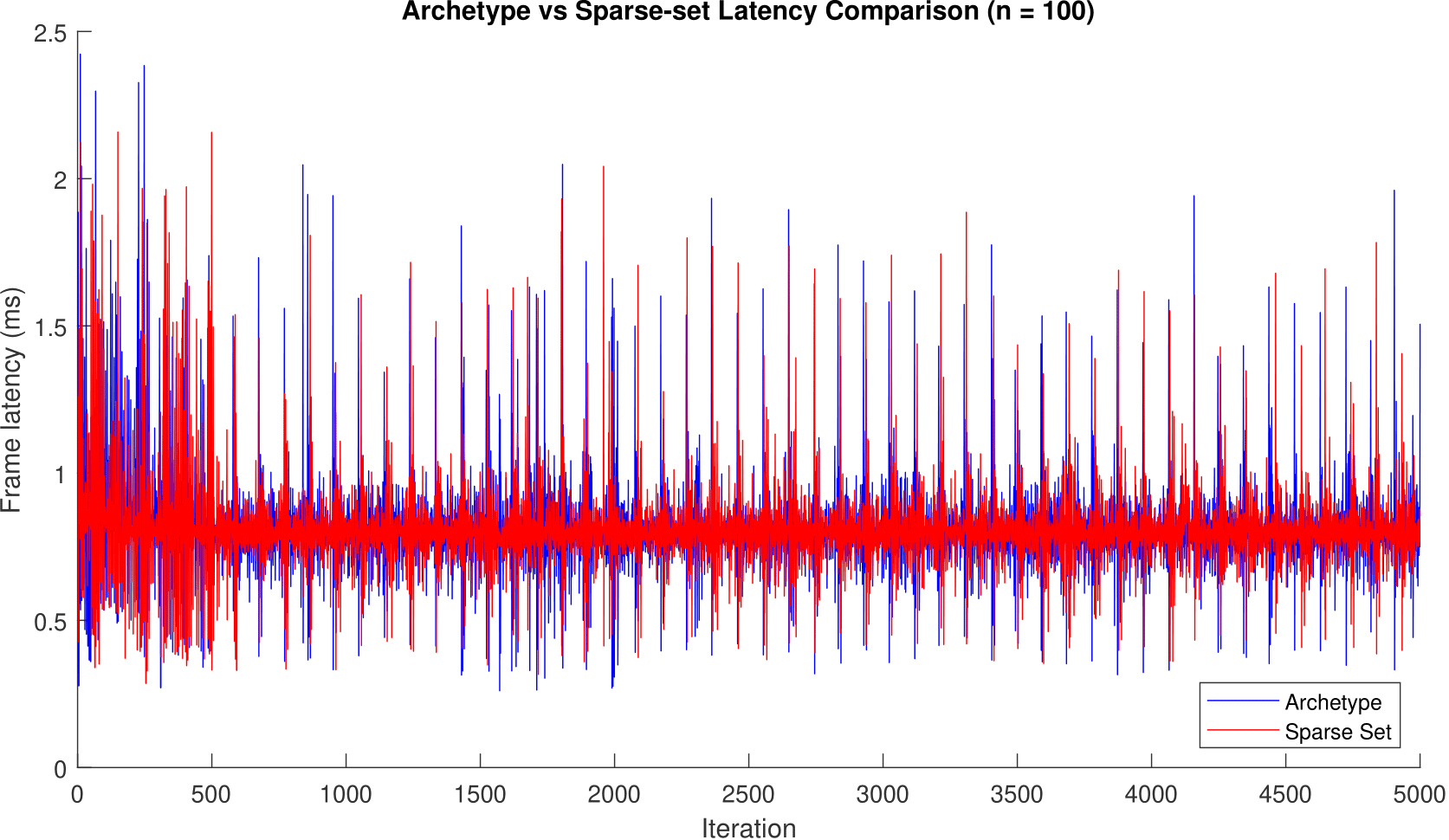
Using the research methods outlined in the section prior, data on the performance of both sparse set and archetype ECS architectures was recorded. This will provide valuable insights into the efficiency and scalability each option has to offer. This section examines two primary metrics: frame update times and entity setup times, measured across varying entity counts.

## Frame Update Times

Update times were recorded for the first 5000 frames each simulation. This information has been gathered for both sparse set and archetype based ECS architectures running at 100, 1,000, 10,000, and 50,000 entities.

### 100 Entities

The first comparison performed was using 100 entities, and as the following graph highlights there was little difference in the update performance of the ECS variations whilst working with an entity count of this size. Figure 8 shows a direct comparison of the performance of the two ECS implementations.



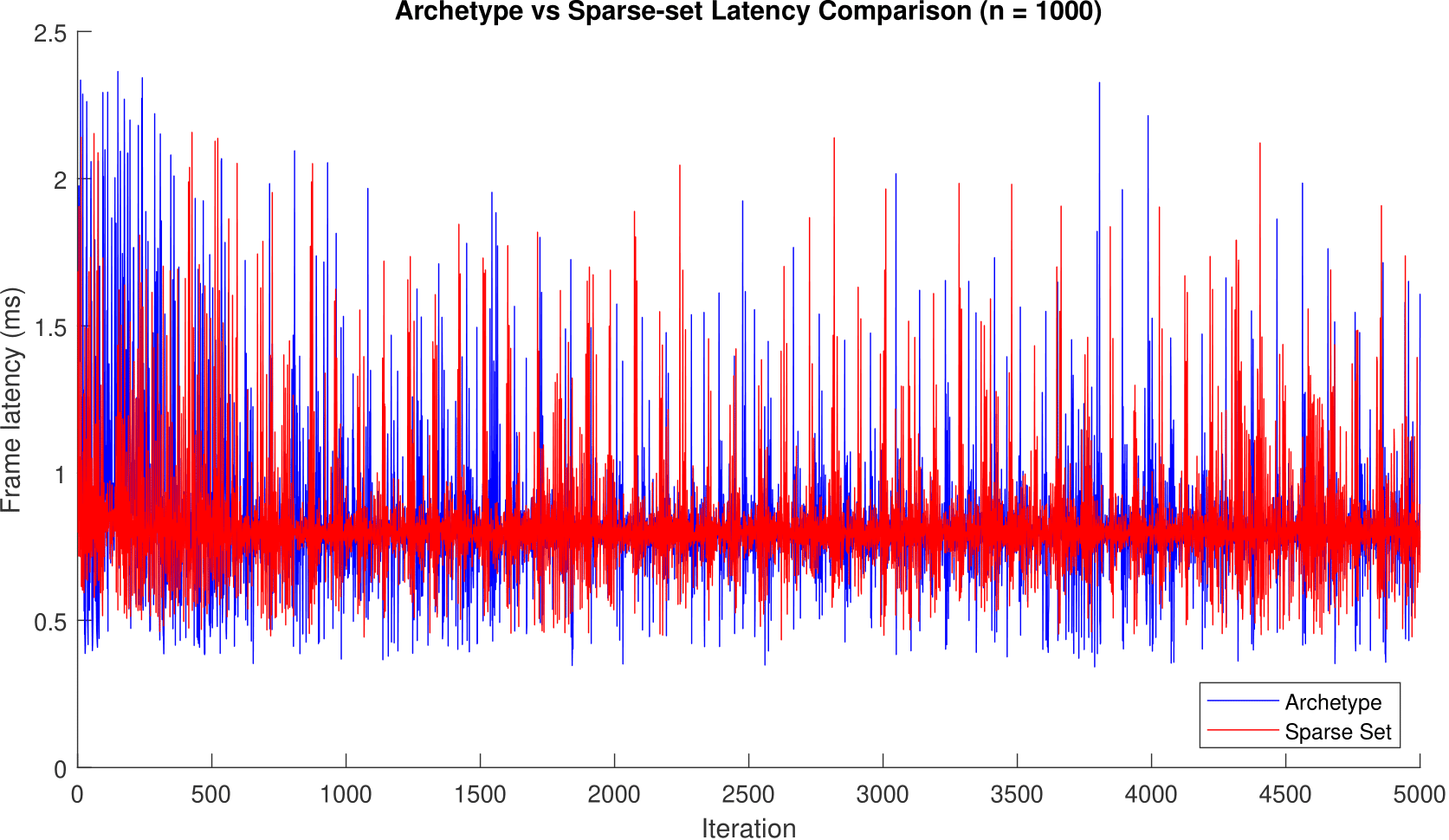
Figure

Using the data gathered from 5000 frames, the iteration speed at 100 entities was found to be marginally faster using sparse sets (*Mdn = .7964ms*) than archetypes (*Mdn = .7989*). A Mann-Whitney U test was performed on these results, finding the difference between iteration speeds for the ECS variations to be statistically insignificant, U (NArch = 5000, NSparse = 5000) = 12392319.5, z = -.746, p = .456.

### 1000 Entities

Following this, the same tests were performed on 1000 entities to remarkably similar results. At 1000 entities iteration over sparse sets (*Mdn = .7975ms*) was faster than iteration over archetypes (*Mdn = .7985ms*), however a Mann-Whitney U test strongly indicated that this result was not statistically significant, U (NArch = 5000, NSparse = 5000) = *12373963.5*, z = -.873, p = .383.

Figure 9 displays the results of this test.



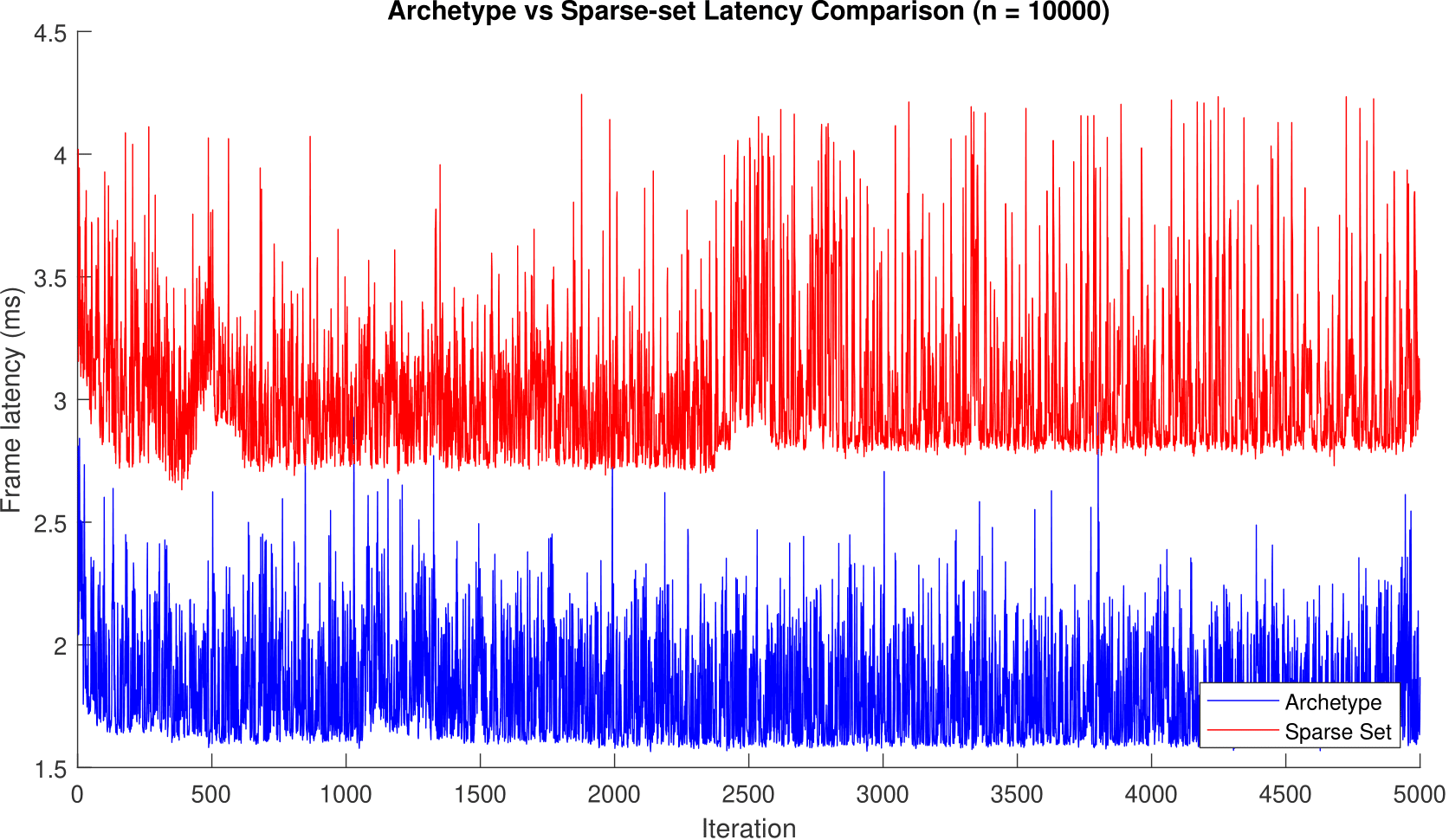
Figure

### 10000 Entities

At 10000 entities, results begin to diverge from the pattern established at smaller entity counts. Iteration for archetypes (*Mdn = 1.7209ms*) was faster on average than sparse sets (*Mdn = 2.9451ms*), on average completing an iteration in 58.432% of the time taken for sparse sets for a median difference of 1.2242 milliseconds.

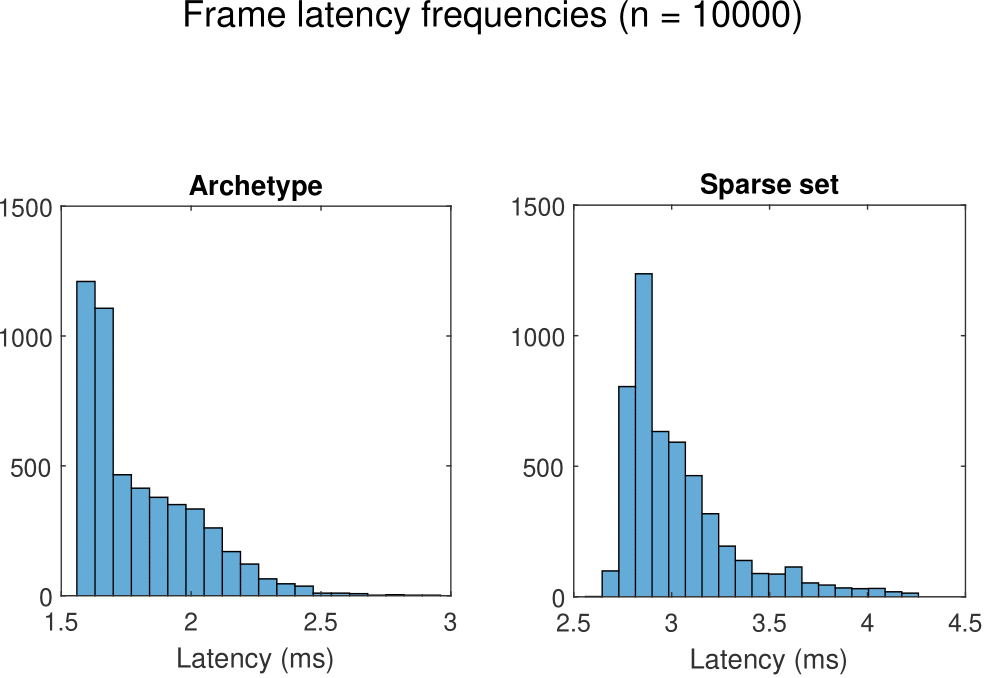
Once again, a Mann-Whitney U test was performed on these results, finding them to be significant, U (NArch = 5000, NSparse = 5000) = *10154.5*, z = -86.598, p = .000.

The clear and significant difference between in the performance of the two ECS implementations is visualised in Figure 10.



Figure

Interestingly, at this entity count sparse sets (*M = 3.0328ms, SD = .2833ms*) also began to demonstrate a greater standard deviation in iteration time than archetypes (*M = 1.806ms, SD = .2148ms*) as can be seen in Figure 11. This increased variation could be indicative of more frequent cache misses.

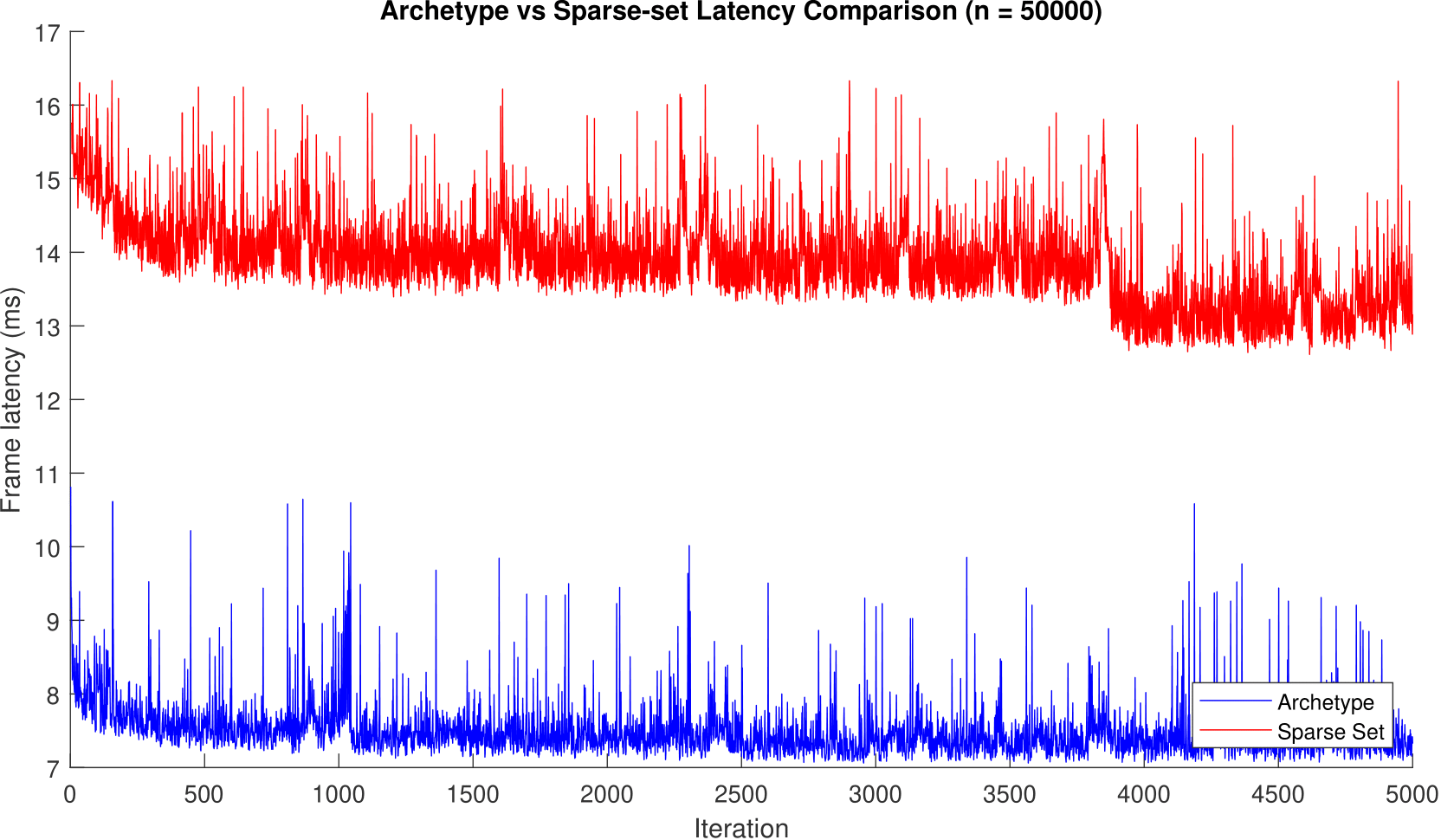


Figure

### 50000 Entities

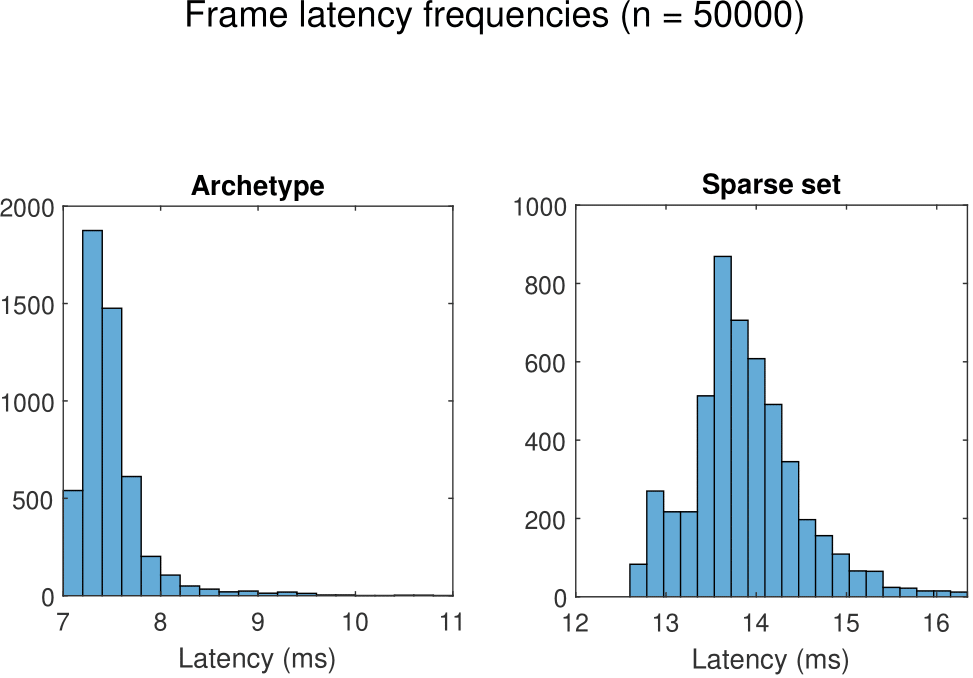
At 50000 entities the differences between the two implementations become increasingly clear. At this level archetypes (*Mdn = 7.4102ms*) iterate pronouncedly faster than the sparse sets (*Mdn = 13.8114ms*), taking on average 53.652% of the time to iterate over the same data, almost doubling the performance. Much like the last set of data, a Mann Whitney U test found the results to be significant, U (NArch = 5000, NSparse = 5000) = .000, z = -86.598, p = .000.

Once again, a comparison of the two implementations is shown in Figure 12, clearly highlighting the performance difference between them.



Figure

Additionally, the difference in standard deviation between sparse sets (*M = 13.8731ms, SD = .6005ms*) and archetypes (*M = 7.4865ms, SD = .3591ms*) continues to grow, as can be seen in Figure 13.



Figure

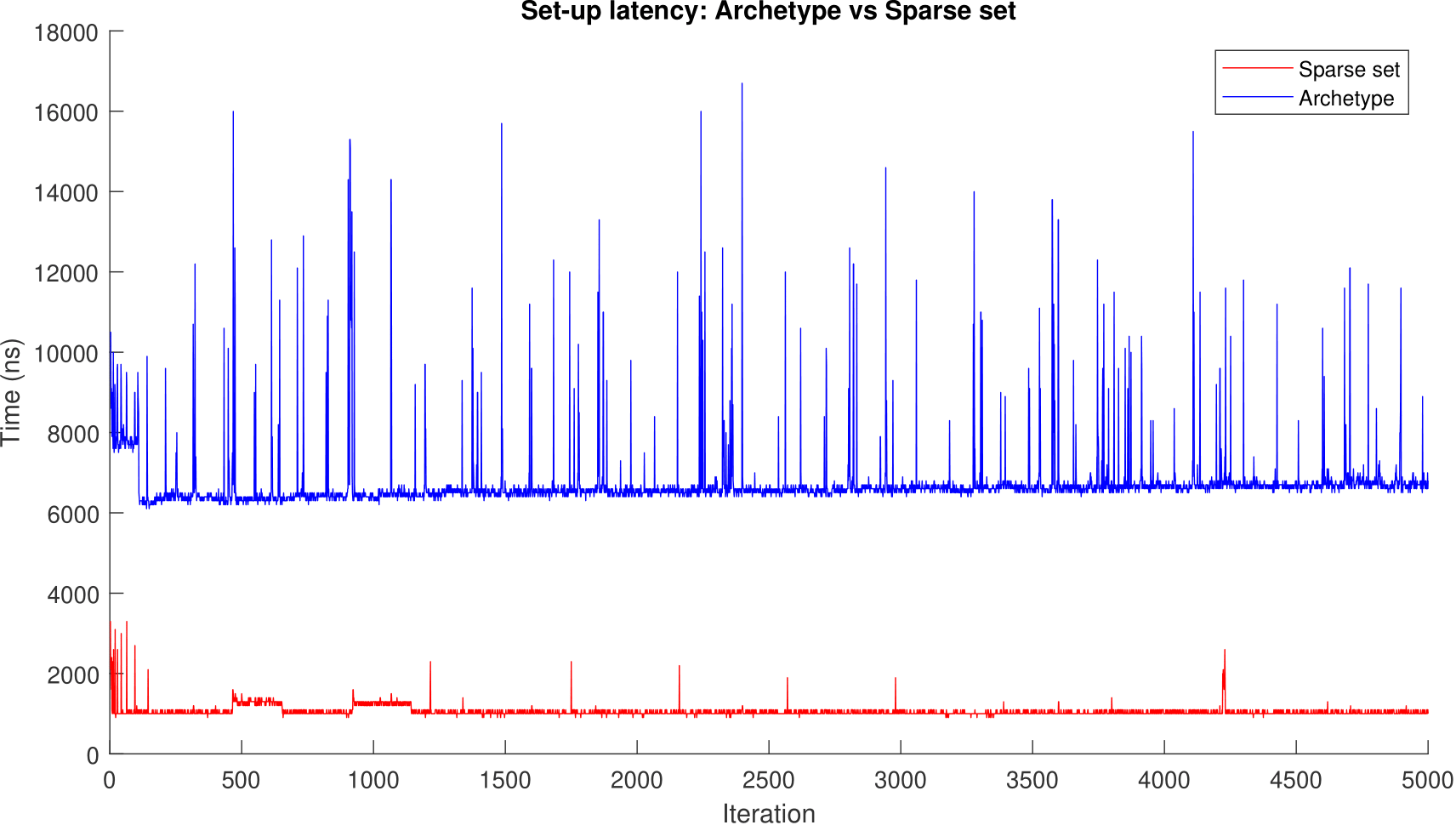
## Entity Setup

Entity setup benchmarks were performed on a sample size of 5000 entities, and the time taken to establish each with the required components for the simulation was measured. Measurements were taken in nanoseconds.

In this regard the sparse set based ECS (*Mdn = 1000ns*) was found to be notably faster than archetype based ECS (*Mdn = 6600ns*). On average the difference between the two approaches was 5600 nanoseconds.

A Mann Whitney U test found these results to be significant, U (NArch = 5000, NSparse = 5000) = *.000, z = -88.613, p = .000.*

Figure 14 demonstrates the clear difference in performance, as well as the significantly greater variation in archetype results.



Figure

Additionally, the standard deviation of archetypes (*M = 6744.2ns, SD = 861.9274ns*) was far greater than that of sparse sets (*M = 1053.18ns, SD = 134.2739ns*), making the variance in time taken to alter entity composition far greater using archetypes than sparse sets. The cause of this could be hashmap lookup overhead and cache indirection, as hashmaps are used to find the appropriate archetype to place a component in.

## Summary of Findings

The benchmarking results demonstrate that as entity counts increase, performance discrepancies between ECS implementations become more pronounced. At lower entity counts, both Sparse Set and Archetype-based ECS approaches perform similarly, with negligible differences in update and setup times. However, as the entity count reaches 10,000 and beyond, distinct performance trends emerge.

Sparse set based implementations show faster initialization times by a factor 6.6. However, as entity counts grow, their update times increase at a greater rate, indicating potential bottlenecks in iteration efficiency. On the other hand, Archetype-based implementations exhibit superior iteration performance at higher entity counts, maintaining more stable frame times even under significant workloads.

Overall, the findings highlight the trade-offs inherent in different ECS architectures. Sparse Set ECS excels in rapid entity setup but may struggle with iteration efficiency at higher entity counts, while Archetype ECS is optimized for efficient iteration but incurs higher setup costs. These results form the basis for the discussion in the next section, where their practical implications and potential optimizations will be explored.

# Discussion and Analysis

The results presented in the previous section highlight key performance characteristics of Sparse Set and Archetype-based ECS implementations. While both approaches demonstrate efficiency at smaller entity counts, performance differences become more pronounced as entity counts increase. This section analyses these findings, discussing the scalability, efficiency, and trade-offs associated with each implementation. Additionally, potential reasons for observed performance trends and considerations for selecting an ECS model in different scenarios are explored.

## Performance Trends and Scalability

The results indicate that as entity counts increase, Archetype-based ECS implementations consistently outperform Sparse Set-based implementations in terms of frame update times. However, at lower entity counts the two approaches exhibit minimal differences in performance, as demonstrated in the 100 and 1,000 entity tests. This suggests that for small-scale applications the choice of ECS implementation has a negligible impact on iteration speed.

However, upwards of 10,000 entities, Archetype based ECS demonstrates significantly better scalability. This is likely due to the more cache-friendly nature of archetype storage, as well as avoiding the overhead of confirming the presence of all required component for each entity iterated over. When component data from an archetype is loaded into cache memory, it can be guaranteed that all surrounding components also loaded into cache memory will be relevant to a given query. As a result, iteration over entities incurs fewer cache misses, leading to sometimes vastly improved performance. Sparse Sets, on the other hand, require more scattered memory access patterns as components are not grouped by the entity’s overall composition. This could mean that when component data is loaded into cache memory there is a chance that some of the components stored are irrelevant to a given query, wasting cache space. Inefficiencies such as this may be the cause of increased cache inefficiency as entity counts grow.

As well as becoming slower with increased entity counts, sparse sets become less consistent. Variance in update latency increased significantly between 10,000 and 50,000 entities, with archetype ECS standard deviation growing by 0.1443ms and sparse set standard deviation growing by 0.3857ms. Once again, could potentially indicate a higher frequency of cache misses.

Despite this, Sparse Set-based ECS implementations appear to maintain an advantage in entity setup times. The results show that Sparse Sets consistently initialise entities more quickly than Archetypes, on average by 6.6 times. This is likely because Sparse Sets store component data individually, ignoring total entity composition as a factor, whereas archetypes require data relocation for all component data related to a given entity when adding or removing any component to or from it. This data relocation will always incur memory copy overheads, and can often cause dynamic reallocation of component vectors, which comes with significant performance costs.

## Practical Implications

The choice between sparse set and archetype based ECS implementations will always depend on the specific requirements of a given application. The results seen in this paper highlight the key characteristics of these two options, showing key factors that should be considered when making this decision.

In environments with large entity and component counts requiring consistent performance, the results here suggest that an archetype ECS is likely better suited. Whilst sparse sets consistently fall behind at scales above 10,000 entities due to their more scattered memory access patterns, archetypes ensure cache locality even in more complex, large scale real-time simulations. This is ideal for games that include many costly systems that must be frequently updated, such as physics simulations, AI pathfinding, and particle systems.

However, in environments requiring frequent editing of entity composition, adding and removing components sparse set based ECS architecture offers significantly reduced modification costs, making it a preferrable choice.

Another advantage of sparse set ECS that has not been discussed yet is ease of implementation. Sparse set based ECS architecture is considerably less complex than the archetype based approach, and in scenarios where the ECS architecture chosen is unlikely to be a performance bottleneck, the simplicity of sparse sets can make it a desirable option.

## Potential Optimisations

Whilst the results presented in this paper form a strong demonstration of the differences in performance of archetype and sparse set-based ECS implementations, it is important to acknowledge that these findings are ultimately dependent on the specific implementations used. There are several optimisations could be applied to potentially offset the drawbacks associated with each ECS variation which were not considered in benchmarks taken for this paper.

One such optimisation is the use of an archetype graph, which was covered in the literature review but not utilised in the benchmarks presented. Implementing this technique could significantly reduce the cost of entity restructures by precomputing transitions between archetypes, thereby reducing the computational burden when components are added or removed from entities.

Another promising optimisation is memory pooling, which can help mitigate the pitfalls of dynamic memory allocation commonly associated with the use of vectors. By pre-allocating fixed-size memory blocks during initialisation, memory fragmentation is minimised, ensuring that component data is stored more efficiently. This approach could potentially lead to significant reductions in entity restructure overhead while also improving cache coherence, which could result in more consistent frame times and enhanced overall performance. (Sperens, 2019)

An additional optimisation that could be employed in a sparse set based architecture is component grouping, as mentioned in the literature review. This improvement could potentially significantly reduce cache misses when operating on multiple component types by ensuring components that are part of entities matching the group are stored close together in memory. This could potentially have an important impact on iteration speeds, helping sparse set based implementations scale to larger entity counts.

# Conclusion

This study set out to compare the performance characteristics of two widely used ECS architectures—sparse set based and archetype based implementations. By leveraging a controlled benchmarking process, this research analysed key performance metrics, including iteration speeds and entity setup times, gathering a total of 50,000 data points during testing. The results highlighted the fundamental trade-offs inherent in each approach, providing valuable insights for developers seeking to optimize ECS usage in game engines and simulation frameworks.

The benchmarking data demonstrated that sparse set based ECS implementations excel in entity setup times, making them well-suited for applications where frequent entity modifications are necessary. Their component storage method allows for rapid insertion and deletion, minimising the overhead typically associated with entity composition changes. However, as entity counts increased, the sparse set implementation exhibited growing iteration inefficiencies, with update times scaling less favourably than the archetype based alternative. This performance degradation was attributed to fragmented memory access patterns and additional query overhead, leading to increased cache misses and reduced processing efficiency at larger scales.

Conversely, the archetype based ECS implementation proved superior in iteration performance, particularly in simulations featuring high entity counts. The organisation of component data into contiguous memory blocks significantly enhanced cache locality, reducing iteration overhead and ensuring more predictable performance across different workloads. However, the cost of modifying entity compositions was notably higher in the archetype based ECS due to data migration overhead, resulting in slower setup and modification times compared to Sparse Sets.

## Achievement of Research Aims

The primary aim of this research was to provide insight into the trade-offs between Sparse Set and Archetype-based ECS architectures. Through a structured benchmarking approach, this study successfully identified the key strengths and weaknesses of each implementation, offering a clear comparison of their relative efficiencies in different scenarios. The objectives outlined at the beginning of this paper were addressed:

1. Iteration Performance: The results confirmed that Archetype ECS significantly outperforms Sparse Set ECS in iteration speed at large entity counts due to superior memory locality and cache efficiency.
2. Entity Composition Modification: Sparse Set ECS demonstrated a lower restructuring cost, making it preferable for applications requiring frequent component additions and removals.
3. Scalability: The study highlighted that whilst the ECS variations exhibit similar performance at low entity counts, sparse set ECS suffers considerable performance degradation at higher entity counts, whereas archetype ECS maintains consistent performance but with higher modification overhead.

By addressing these research questions, this paper provides a comprehensive understanding of how each ECS implementation performs under varying conditions, enabling developers to make informed architectural decisions.

## Benefits of Research

The findings of this study provide critical insights into the performance characteristics of ECS architectures, allowing developers to make informed decisions when selecting an implementation that best suits their needs. By offering a direct comparison between Sparse Set and Archetype ECS under controlled conditions, this research clarifies the specific strengths and weaknesses of each approach, which is essential for game engine developers, simulation designers, and software engineers working with ECS.

One of the most significant benefits of this research is the demonstration of how iteration performance and entity modification costs scale with entity count. The findings highlight that Sparse Set ECS is better suited for applications requiring frequent entity modifications, while Archetype ECS excels in high-performance iteration scenarios. This distinction is crucial for developers optimizing ECS performance for different workloads, as it provides a clear framework for selecting the right architecture based on project requirements.

Additionally, this research sheds light on memory efficiency and cache behaviour, particularly the advantages of contiguous memory storage in Archetype ECS. Understanding how different ECS implementations interact with CPU cache enables developers to design more cache-friendly systems, reducing performance bottlenecks and improving real-time simulation efficiency. The insights gained from this study can contribute to the optimisation of ECS frameworks, leading to more responsive and scalable applications.

Another benefit of this study is its contribution to the broader field of ECS research, where comparative analyses of different implementations are still relatively scarce. By providing quantitative benchmarking data, this research helps to fill a gap in existing literature, offering a valuable reference point for future studies.

## Limitations of Research

Despite its contributions, this study has certain limitations that should be acknowledged. First, the benchmarking results are inherently tied to the specific implementations used in this research. While every effort was made to ensure fair comparisons, other ECS frameworks and implementations may differ in ways that could affect performance outcomes.

Second, the study focused solely on single-threaded performance, omitting considerations for multithreading and parallel processing. Many modern ECS implementations are designed with parallelism in mind, leveraging multithreading to optimise component updates and entity processing. This research does not account for how different ECS architectures perform under parallel execution, which remains an important consideration for future exploration.

Additionally, the benchmarking approach used Conway’s Game of Life as a standardised test case. While this simulation provided a structured environment for comparing ECS architectures, real-world applications often involve more complex dependencies, inter-entity communication, and system interactions. The results of this study may not fully generalise to all ECS use cases, particularly those requiring frequent inter-component communication or external data dependencies.

## Final Thoughts

The results of this study highlight the complexity and versatility of ECS architectures, emphasizing that no single approach is universally superior. Instead, the choice between Sparse Set and Archetype ECS depends on specific use case requirements, with each offering distinct advantages and trade-offs.

As ECS continues to evolve, developers will need to carefully consider scalability, memory access patterns, and modification costs when selecting an ECS implementation. The findings presented in this research offer a valuable framework for understanding the trade-offs involved, helping developers make informed decisions that optimise performance in their specific applications.

Ultimately, this research contributes to the ongoing discourse surrounding ECS development, providing a foundation for future exploration into the evolving landscape of data-oriented design in modern computing.

# Recommendations

While this study provides valuable insights into the performance trade-offs between sparse set and archetype based ECS implementations, there remain numerous opportunities for further research. The growing adoption of ECS in game development, simulations, and real-time computing necessitates continued exploration of new implementations and optimisation strategies. Expanding on the findings of this study will enable developers to better tailor ECS implementations to their different use cases, improving overall efficiency and scalability.

Another potential area for future research is comparing implementations in the context of multithreading. Modern ECS frameworks are often designed with parallelism in mind, leveraging multithreading to distribute computational workloads efficiently. Future research could evaluate how sparse set and archetype ECS architectures perform under parallel execution, investigating how different ECS architectures scale when distributing entity updates across multiple cores. This could provide deeper insights into their practical performance in real-world applications.

Further research papers could also explore the optimisation techniques mentioned in this paper that were not employed during testing. Research into how techniques such as memory pools, component grouping, and archetype graphs affect the performance of each ECS architecture could offer important new context to aid developers in finding the implementation best fitting their use case.

Additionally, there are other ECS architectures that warrant academic exploration. For example, the Hybrid ECS employed by Bevy covered in the literature review. This model allows developers to selectively store components using different methodologies based on usage patterns. Further research into how hybrid approaches can be optimised for different workloads could yield valuable insights into reducing iteration overhead while maintaining efficient entity modification processes.

Whilst there are papers focused on researching how ECS fits a variety of use cases, there is little research comparing differing ECS architectures or optimisations for specific use cases. Whilst the data collected in this paper gives us a good idea of how each ECS performs, real life use cases are likely to be far more complex than the simulation used here. As such, research comparing ECS implementations benchmarked using real world applications could give us even further insight into the performance characteristics of each architecture and optimisation.

Expanding ECS research beyond the foundational benchmarks presented in this study will provide critical advancements in performance, scalability, and efficiency. Investigating multithreading, hybrid storage models, real-world workloads, and memory optimizations will deepen the understanding of ECS architecture and its applications. As ECS continues to be widely adopted across multiple industries, further research will be instrumental in refining best practices and optimising performance for next-generation game engines and simulation frameworks.

# References

Acton, M., 2014. *Data-Oriented Design and C++.* s.l., CppCon.

Anderson, C., 2021. *Bevy 0.5.* s.l.:s.n.

Bilas, S., 2002. *A Data-Driven Object System.* s.l., Game Developers Conference.

Brennan, S. et al., 2023. An extensible, data-oriented architecture for high-performance, many-world simulation.. *ACM Transactions on Graphics (TOG),* 42(4), pp. 1-13.

Briggs, P. & Torczon, L., 1993. An Efficient Representation for Sparse Sets. *ACM Letters on Programming Languages and Systems,* Volume 2, pp. 59-69.

Caini, M., 2019. *Sparse sets and grouping functionalities.* s.l.:s.n.

Fischbach, M., Wiebusch, D. & Latoschik, M. E., 2017. Semantic Entity-Component State Management Techniques to Enhance Software Quality for Multimodal VR-Systems. *IEEE Transactions on Visualization and Computer Graphics,* 23(4), pp. 1342-1351.

Ford, T., 2017. *Overwatch Gameplay Architecture and Netcode.* s.l.:Game Developers Conference.

Gold, J., 2004. *Object-Oriented Game Development.* s.l.:Addison-Wesley.

Gregory, J., 2014. *Game Engine Architecture.* 2 ed. s.l.:CRC Press.

Härkönen, T., 2019. *Advantages And Implementation of Entity-Component-Systems.* s.l.:Faculty of Information Technology and Communication Sciences.

Hollmann, T., 2019. *Development of an Entity Component System Architecture for Realtime Simulation.* s.l.:s.n.

Hossain, M. M., Alam, M. S., Ahmed, J. U. & Bashar, M. K., 2015. Performance Improvement of Average Access Time by Optimization of Cache Memory Access Time. *International Journal of Industrial Engineering and Operations Management,* pp. 756-761.

LeBlanc, M., 2017. *Game Entities in Thief: The Dark Project.* s.l.:Riot Games.

Martin, A., 2007. *Entity Systems are the future of MMOG development.* s.l.:s.n.

Mertens, S., 2020. *Archetypes and Vectorization.* s.l.:s.n.

Moore, G. E., 1965. Cramming more components onto integrated circuits. *Electronics*, 19 April.

Mroz, J. & Pietkiewicz, L., 2023. ECS og Optimaliseringsmønstre.

Nyberg, F., 2021. Investigating the effect of implementing data-oriented design principles on performance and cache utilization..

Nystrom, R., 2011. *Game Programming Patterns.* s.l.:Genever Benning.

Redmond, P., Castello, J., Trilla, J. M. C. & Kuper, L., n.d. Exploring the Theory and Practice of Concurrency in the Entity-Component-System Pattern.

Rentsch, T., 1982. Object oriented programming. *ACM Sigplan Notices,* 17(9), pp. 57-57.

Ryant, I., 1997. Why inheritance means extra trouble.. *Communications of the ACM,* 40(10), pp. 118-119.

Snyder, A., 1986. *Encapsulation and inheritance in object-oriented programming languages.* s.l., Conference proceedings on Object-oriented programming systems, languages and applications.

Sperens, M., 2019. *Dynamic Memory Managment in C++.* s.l.:s.n.

Taivalsaari, A., 1996. On the notion of inheritance.. *ACM Computing Surveys (CSUR),* 28(3), pp. 438-479.

Tarifa Mateo, Y., 2024. Entity component systems and data-oriented design..

Webster, C., 2015. *Entity Component Systems Usefulness, Literature Review,* s.l.: s.n.

Wilkes, M. V., 1965. *Slave Memories and Dynamic Storage Allocation.* s.l.:IEEE.

Williams, B., Oliver, T., Ward, D. & Headleand, C., 2024. *Real-time Data-Oriented Virtual Forestry Simulation for Games.* s.l., Computer Graphics & Visual Computing Conference.

1. https://github.com/skypjack/entt [↑](#footnote-ref-2)
2. https://github.com/SanderMertens/flecs [↑](#footnote-ref-3)
3. https://bevyengine.org/ [↑](#footnote-ref-4)
4. Central Processing Unit [↑](#footnote-ref-5)
5. International Business Machines Corporation [↑](#footnote-ref-6)
6. Random Access Memory [↑](#footnote-ref-7)
7. A measurement of the operations required for an algorithm based on the amount of inputs [↑](#footnote-ref-8)