EG UK Computer Graphics & Visual Computing (2025)
Y. Sheng and A. Slingsby (Editors)

Run-time Performance Comparison of Sparse-set and Archetype Entity-Component Systems

L. Cox[1], B. Williams[1], J. Vickers[1], D. Ward[1] and C. Headleand[1]

[1]Staffordshire Games Institute
School of Digital, Technologies, Innovation and Business
University of Staffordshire, UK

Abstract

Entity-Component System (ECS) architectures have emerged as a powerful alternative to traditional object-oriented solutions in video games and real-time simulations. However, different ECS implementations present distinct trade-offs between iteration speed and modification costs. Despite its growing adoption, a comparative analysis on the performance characteristics of ECS implementation types has yet to be conducted. This study compares the performance of two widely-used ECS implementations: sparse-set and archetype-based. To facilitate this, an implementation of each architecture was developed in C++20 and their performance was examined in terms of iteration speed and entity modification costs. The results show sparse-set ECSes enable cheaper entity modifications but scale poorly during iteration, while archetypes excel at large-scale iteration through cache efficiency but incur higher composition change costs. These findings provide valuable and actionable guidance for developers selecting ECS architectures for their specific applications.

CCS Concepts

• ***Applied computing*** → ***Computer games;*** • ***General and reference*** → ***Performance;*** • ***Computing methodologies*** → ***Real-time simulation;*** *Computer graphics;*

1. Introduction

Video games have become the largest sector in the global entertainment industry [DlTSGR25], with recent figures showcasing that one-third of people engage with video games daily [BR24]. As gaming continues to grow in popular culture, consumer demand for bigger, better, and more technically ambitious game experiences has intensified. In response, game development studios are continuously pushing the fringes of what is computationally possible. One games development technology which has gained considerable traction in recent years is the Entity-Component System (ECS) engine architecture. Initially introduced in commercial games development as early as 1998 [Här19], ECS has recently found renewed interest in modern video games. As an example, the front-running *Unity* and *Unreal* game engines now offer native ECS support for developers [TM24], founding a significant rise in recent interest for this technology. Interest in ECS architectures is largely spurred by its efficiency in simulating massive numbers of entities in real-time. For example, an archetype-based ECS approach enables easy computational parallelisation (via vectorisation) across entities in a game world, which has been shown to have significant performance benefits [WOWH24]. Due to these benefits, many game studios have integrated ECS into their development process. One notable example is *Minecraft: Bedrock Edition*, a video game by *Mojang*, which uses *EnTT* (a C++ sparse-set ECS library) to drastically improve computational performance [Här19].

Despite the growing interest and adoption of ECS in the games industry, there remains comparatively little research concerning this architecture in the literature. Whilst some papers demonstrate the utility of applying ECS for computational efficiency [WOWH24], notable gaps are still evident in the literature, with one of these gaps concerning the comparative analysis of ECS implementations. Entity-Component Systems can be implemented in several ways, with sparse-set and archetype-based being two of the most common implementation patterns. Comparing these two implementations in terms of frame latency would, for instance, provide valuable insights into the benefits and disadvantages of either implementation; potentially informing developers which is most appropriate given their specific application. It could also provide developers with actionable insights and prompt engine design decisions prior to implementation, significantly reducing production costs.

It is with this motivation in mind that this paper presents a performance-based analysis between two prevalent ECS implementation paradigms: sparse-set and archetype-based architectures. We provide an initial analysis of the two in terms of frame latency performance and establish practical guidelines to aid developers in

DOI: 10.2312/cgvc.20251224

selecting the optimal ECS paradigm for specific use cases. We also additionally highlight the urgency for further work in this currently underexplored area.

2. Related Work

Video games have dramatically shifted from a niche hobby into a mainstream form of entertainment in recent decades [EW22]. The sector has eclipsed traditional forms of media, with revenues surpassing the music and video industry combined [PPM*21]. The rapid growth of the industry has brought about more discerning consumers, demanding more technologically robust games year-on-year [CJV*24]. With this increasing demand, and game success being heavily dependent on innovation [AL14], developers face unrelenting pressure to provide richer experiences which fully utilise consumer hardware.

2.1. Architectural Engine Patterns

To meet these demands, game engine architectures have evolved significantly over the years. Object-oriented Design (OOD) remains the most dominant paradigm for modelling game engines [Gre18], primarily due to its benefits in maintainability and abstraction [WBW89], and the clear conceptual mapping between game entities and code objects [Gre18]. OOD also finds applicability throughout several engine subsystems, such as the graphics pipeline or game logic [AC07]. Game engines have naturally adopted architectural OOD patterns through their evolution. Amongst these is the prevalent example of the *component* pattern, popularised by the front-running Unity and Unreal game engines [Gol04]. By opting for composition rather than inheritance, entity behaviour emerges from a distinct set of reusable components instead of tightly-coupled objects [Nys14]. This approach promotes more comprehensible, maintainable and decoupled architectures [Rau18, Nys14]. Similar patterns can also be leveraged in intercomponent communication, via the observer pattern [EE05], in further decoupling efforts.

Whilst OOD offers advantages in maintainability and code comprehension, its performance limitations can present challenges for game engines, such as the overhead associated with dynamic object allocation [Yli25], poor cache coherence [Far18] or more unstable performance [FAUM20]. These limitations have prompted a shift of industry interest in Data-oriented Design (DOD) as a viable alternative [Yli25]. Where OOD prioritises abstraction and encapsulation, DOD reorganises systems around data access patterns, transformations and hardware utilisation, for efficient processing [Fab18]. Whilst the definition of DOD can vary between sources, the core tenets of DOD focuses on performance first and architectural decisions being driven by data [Str19]. The key motivator in the adoption of DOD principles in games development is largely spurred by the inherent performance benefits [FAUM20]. Traditional OOD approaches are mostly incompatible with the demands of complex games, where several distinct object types exist with overlapping functionalities; making deep inheritance trees brittle, inflexible and cumbersome for developers [Joh25]. Outside of the context of performance, DOD also has other benefits, such as enabling non-technical designers greater freedom without requiring programmer input [Gre18].

2.2. Entity-Component Systems

The Entity-Component System (ECS) paradigm combines the architectural benefits of the component pattern with the performance of Data-oriented Design [Han16, TM24]. The ECS pattern achieves this by distinctly separating behaviour from data [Joh25], into three categories [TM24, Här19]:

- **Entities**: Typically a unique numerical identifier use to compose related components together [Com22]. More abstractly, entities are the fundamental building blocks of the simulation [Här19].
- **Components**: The containers of plain data associated to an entity, with no attached logic or behaviour [TM24, Här19].
- **Systems**: The processes that holistically simulate all entities matching a set of components, with no associated state or data [Här19].

The decoupling of logic from behaviour in ECS not only fosters architectural modularity [TM24], but additionally yields structures which are readily adaptable to computational parallelisation [Gar21]. Previous work also confirms that parallelised ECS approaches can significantly outpace serialised counterparts in terms of performance [WOWH24]. The key differences between OOD and ECS are illustrated in Figure 1.

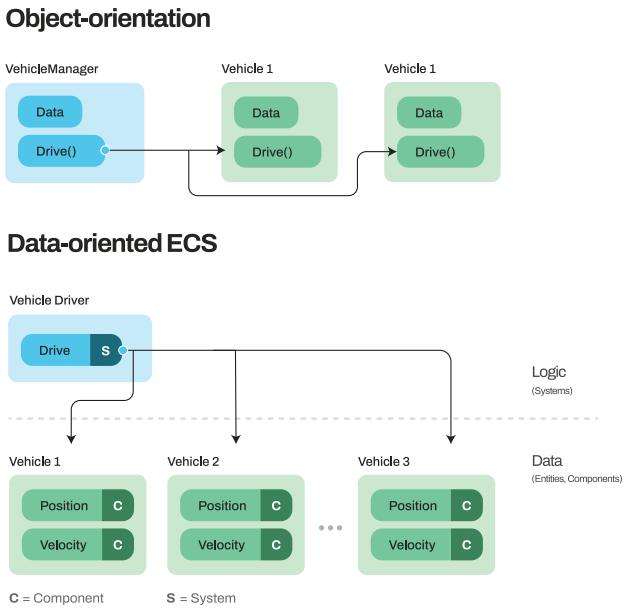


Figure 1: *A diagram showcasing the conceptual differences in OOD vs ECS. In OOD, objects encapsulate both logic and data. In ECS, components represent data and are clearly separated from program logic.*

Recognising these advantages, industry-leading engines such as *Unity* and *Unreal* have recently offered toolkits for developers interested in leveraging the power of ECS [TM24]. The application of ECS principles have for existed for several years prior to the coining of the term 'ECS', however. For example, early DOD works highlight the utility of separating data from logic, along with its inherent parallelisation benefits [Sha80, SH84]. In games, one of the

earliest implementations can be found as far back as 1998, in *Thief: The Dark Project*, a stealth-based video game [Leb17, Här19]. The paradigm later gained further traction in the industry, notably in the 2002 title *Dungeon Siege*, which leveraged ECS for significant performance gains [Fre16]. Today however, ECS adoption expands rapidly as developers look to build more performant games, with recent papers suggesting ECS as the backbone of bespoke modern engines [Hol19, Hal14]. The traction of ECS has also spread to mainstream titles, such as the popular *Minecraft: Bedrock Edition*, which leverages the popular *EnTT* C++ ECS library [HÖ20]. Outside of games, ECS also finds utility in a large array of real-time applications [Com22]; in simulating virtual forestry [WOWH24], maritime systems [HCSZ21] or GUI applications [RH19], to name a few examples.

2.3. Entity-Component System Implementations

ECS is an architectural paradigm and is fundamentally implementation agnostic; there are several ways in which an ECS can be made. For example, archetype-based ECSes collate entities with identical component compositions into contiguously stored *'archetypes'*, which helps with cache coherency and parallelisation efforts [Cho24]. Three notable examples of archetype-based ECSes are Unity's *DOTS* [Tic23], Unreal's *MassEntity Plugin* [TM24] and Mertens' *FLECS* [Mer20] packages for ECS. This approach lends itself well to easy parallelisation via multithreading, making it a popular implementation type [TM24]. One downside of an archetype-based ECS concerns structural changes: when an entity's archetype is modified, memory is restructured, incurring drastic costs [Cai21]. Batching structural changes at well-defined points or utilising an archetype graph can help reduce this, however [Cho24, Mer20].

Another popular ECS implementation type are sparse-set ECSes, popularised by the *EnTT* library [Cho24]. Sparse-set ECSes utilise a sparse set data structure [BT93], which is a data structure of two arrays: the sparse list and the dense list. The dense list stores component data, and the sparse list maps the entity identifier to components. The advantages of this approach are constant $\mathcal{O}(1)$ search and modification time [Cai21, TM24]. The drawbacks to this approach however concern direct indexing domain constraints [Cho24] and inflexibility to query multiple components in iteration [TM24].

2.4. Comparative Studies

Considering which type of ECS implementation may be most appropriate is an inherently complex problem, as each implementation has unique advantages and disadvantages. Other types of ECS implementations, such as graph-based implementations [Cho24] or bitset ECSes [Cai21] also exist, which further complicates decision making. Despite the clear growth of the adoption of ECS in industry, the literature unfortunately remains largely unexplored. As such, there has yet to be a direct performance-based comparison between the two most prevalent ECS types: sparse sets and archetypes. Similar comparisons exist, for example, in investigating serial vs parallelised ECS simulations [WOWH24, Cho24], or in contrasting object-oriented methods to ECSes [Här19, TM24, AA24], but there is little-to-no work examining ECS implementations.

To date, the only analysis which indirectly examines ECS types appears in the work of Hansen & Öhrström [HÖ20], who contrasted several open-source ECS libraries. They identified several key insights, notably that entity lookups constitute a major source of simulation overhead, regardless of the library. Secondly, object-oriented approaches scale poorly in comparison to ECS approaches, due to a lack of cache coherence [AA24]. Unfortunately, much richer insights concerning cross-implementation contrasts, e.g. between archetypes or sparse-set ECSes, are absent in the literature. Given the widespread adoption of these two approaches in modern engines and libraries (e.g. *Unity*, *Unreal*, *EnTT*), the lack of comparative studies deserves consideration.

3. Methodology

To explore the key differences between archetypes and sparse-sets, two minimal C++20 ECS prototypical implementations were developed. The exploration of current open-source solutions would not allow us to accurately contrast the inherent architectural differences of each. Instead, we opted for a comparison of purpose-built minimal prototypes as it enables a direct comparison of architectures between the two implementations in a controlled environment. The C++ source code of the minimal ECS prototypes is also open-source and freely available on GitHub[†].

3.1. Implementation Agnostic Features

Both of the sparse-set and archetype-based ECS types were built within a single framework, with some features which were independent across each implementation. Performing a comparative analysis is easily achievable through this approach, as boilerplate code and utilities could be implemented which work across both prototypes. This not only aids in undertaking the comparison, but additionally in reducing covariates which may affect performance results. For example, a separate class for logging performance data was developed and functioned identically across both sparse-set and archetype ECS implementations, which is discussed later.

An abstract *World* class was developed to manage component sets and entities across both ECSes, as a common interface for containing and managing entities. This abstract layer provides common functionality relevant to both ECS types regarding entity management, and is minimal in its design. It provides the following methods which each ECS can utilise:

- `DeleteEntity(E)`: To delete an entity `E` from the world.
- `IsEntityRegistered(E)`: To determine if an entity, given an identifier `E`, exists in the underlying ECS.
- `AddComponent(T, E)`: Used in adding components of type `T` to an existing entity `E` at run-time.
- `RemoveComponent(T, E)`: Removes a component type `T` from an entity at run-time.
- `GetComponent(T, E)`: Given a component type `T`, attempts to retrieve the values of a component associated with an entity `E`.

[†] https://github.com/StaffsUniGames/cgvc25-ecs-comparison

A separate *Query* system was also developed which enabled the look-up and selection of entities in the world by component types, a common feature of several ECS implementations. Entities which match the component-based criteria can be queried via `GetView()`, in which variadic templated component types can be supplied. The query system searches the world for entities matching the set of components, and returns several results. The method can be used with a *WorldViewIterator* to iterate entities which match the criteria given by the set of components passed. The concrete implementation details of these two systems differed depending on the ECS type, and are discussed in more detail in the following sections. A diagram illustrating the role of the *World* and *Query* classes is shown in Figure 2 for further clarity.

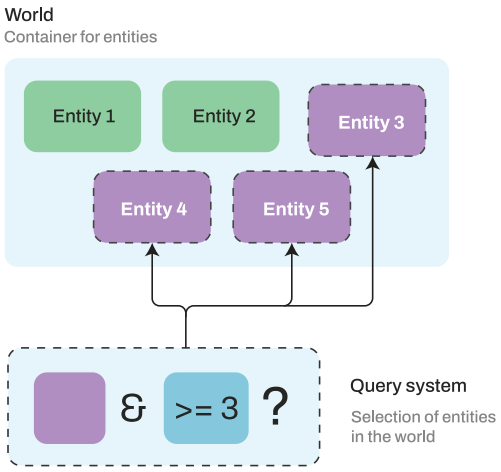


Figure 2: *A diagram showcasing the World and Query systems – the world contains entities and their associated components, and the query system enables the selection of entities matching a set of components.*

3.2. Sparse Set Implementation

The implementation of the sparse-set ECS prototype closely drew on the related work; at its core a sparse-set data structure is used to store components. The *World* was implemented utilising this sparse-set structure. The sparse-set approach can typically be implemented through two `std::vector` instances – the dense list $\mathcal{D}$ and sparse list $\mathcal{S}$. The list $\mathcal{D}$ contains component data of type T, whereas $\mathcal{S}$ contains integer indices mapping entities to components in $\mathcal{D}$. For a given entity identifier E, its index into $\mathcal{D}$ can be found as $\mathcal{S}[E]$, where $\mathcal{S}[n]$ returns the nth element of $\mathcal{S}$. This can be used as an index into $\mathcal{D}$ to retrieve the component data as $\mathcal{D}[\mathcal{S}[E]]$.

The implementation used in the sparse-set ECS prototype uses a pagination optimisation, which adds a further depth to the indexing procedure. Sparse-mapped indices are paginated by a particular size of n entities. This approach means that a single contiguous list of indices $\mathcal{S}$ is not used, instead, there are several lists (or 'pages'). This approach is useful for large entity counts, where allocating huge regions of contiguous memory is infeasible. Similar pagination strategies are found outside of ECS literature, especially where large-scale management and allocation of memory are concerned – for example, in virtual memory [Den70].

A list of pages $\mathcal{P}$ contains several fixed size arrays of indices (of length n) into the dense list $\mathcal{D}$. In this approach, there are two indices (p,w) into a page: p as the page number and w as the in-page index (word). These can be computed, given a contiguous entity index E and page size N, as $p = E/n$ and $w = E \mod n$. An illustration of this strategy can be seen in Figure 3 for further clarity.

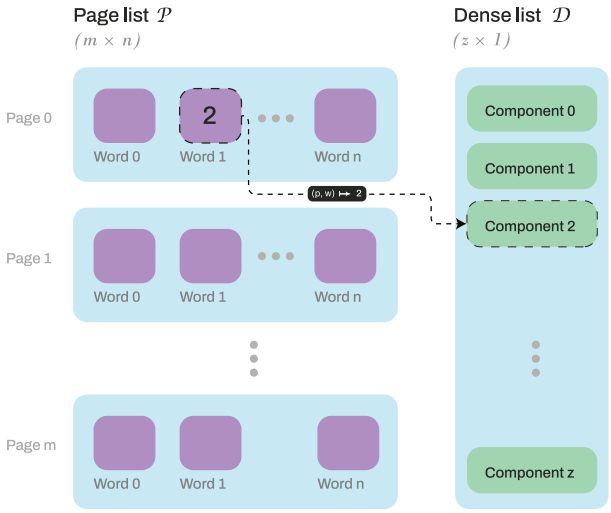


Figure 3: *A visual outline of the pagination optimisation used in the sparse-set implementation. The page list $\mathcal{P}$ stores several fixed size arrays of length n, which contain indices into $\mathcal{D}$. In this diagram $(p,w) = (0,1)$ which returns the data for component index 2 in the dense list $\mathcal{D}$.*

The query system for sparse-sets utilised a graph-based approach for creating complex queries. A *Node* abstract class acts as a wrapper for a component set, along with intersection and difference nodes. A tree of these nodes can be built and supplied to the query system, which returns a collection of entities matching the query. The sparse set is searched linearly as it is a flat set of indices; those matching the graph-based query are included in the return set.

3.3. Archetype-based Implementation

The archetype-based prototype was implemented within the same framework as its sparse-set counterpart, but is very different in its design. Instead of utilising two sets of indices, archetype-based ECSes typically organise entities by their componental composition or *archetypes*. Archetypes are unique sets of components which define a particular class of entity, as illustrated in Figure 4.

For example, the composition of an archetype $A = \{C_1, C_2\}$ contains two components and $B = \{C_1, C_2, C_3\}$ three. Entities in the world can be of these archetypal types; their set of distinct components forms the basis of their archetype. Note in this case A and B are compositionally different, but components C_1 and C_2

are shared, reducing unnecessary repetition of data and enabling reusable components across several archetypes.

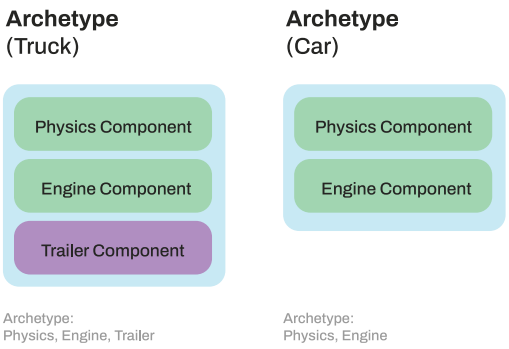


Figure 4: *A diagram illustrating the archetypal ECS paradigm, namely, that archetypes are formed by their set of components. The two archetypes used here are distinct as they differ in componental composition.*

In our implementation of the archetype-based prototype, an *Archetype* class is used which holds a vector of entities $\mathcal{E}$. The class also holds a set of valid componental types $\mathcal{T}$ and unordered map $\mathcal{M}$ which maps types T to component data C with $\mathcal{M}[T]$. Component data is stored in a *ComponentStore* instance which has an integer size and raw `void*` data buffer. The *Archetype* class implements $\mathcal{M}$ as an `std::unordered_map<std::type_index, ComponentStore>`, mapping types to a set of components. This enables *Archetype* instances to be formed of a unique set of components, embodying the paradigm.

This archetype implementation shares some similarities with the sparse-set prototype. The archetype ECS contains an *ArchetypeView* for usage with viewing the results of an archetype-based query, along with an *Iterator* class for iteration. This is similar to the sparse-set prototype where a similar approach is employed. These views are not required to match the components precisely from the archetype from which they are derived; however, all components used for said view must be available in the archetype. For instance, an archetype comprising of components $\{C_1, C_2, C_3\}$ may have a view referencing only components $\{C_1, C_2\}$. Another similar feature between the two implementations is the management of entities via the *World* class. However, the *World* in the archetype approach also stores and manages the archetypes alongside entities. This provides a central interface for the access of all ECS data – should a user request a query for a system A, the view will collect archetype views for each archetype containing component A – enabling the iteration over all relevant archetypes.

4. Benchmarking

The two ECS implementations were benchmarked using John Conway's Game of Life [C*70] using a uniform grid of different sizes. This cellular automaton is well-suited for benchmarking ECS architecture as it involves frequent component updates, iterative processing of large numbers of entities, and predictable computation patterns. It is often used as a benchmarking problem in various computing contexts e.g. the work of Lucas et al. [LDV*19] in which the authors adopt the cellular automaton in benchmarking algorithms. Each cell was assigned as a particular entity, with a *CellData* component for each. An example screenshot can be seen in Figure 5

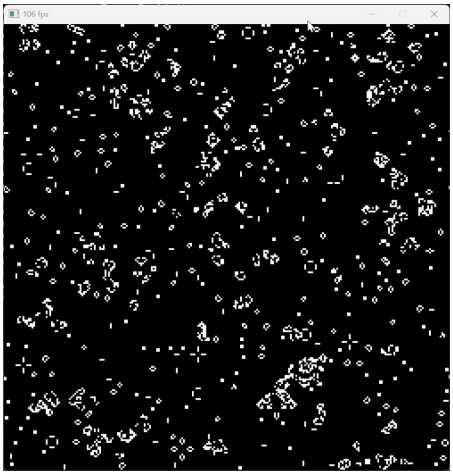


Figure 5: *A screenshot of the implemented benchmark using Conway's Game of Life. In this case, $n = 62500$ (250×250).*

The first metric benchmarked in the comparison concerns frame latency, namely, the time taken to process each frame of the simulation. Benchmarking frame latency provides insights into how efficiently each ECS approach handles iteration, cache locality, and component updates. Each of the simulations was run for a total of 5,000 frames. The second metric concerns latency associated with entity instantiation: the time required to initialise all entities involved in the simulation. This measurement helps determine the overhead involved in managing entities and components within each ECS architecture.

An entity was assigned to each cell in order to reliably examine performance across a consistent number of entities. Using Conway's Game of Life, each of the two ECS variations was benchmarked at increasing entity counts. To achieve this, the uniform grid size of the simulation was altered such that it satisfied the number of entities. Four entity counts and grid sizes were used: 100 (10×10); 1,000 (50×20); 10,000 (100×100); and 50,000 (500×100). Neighbourhood cell updates on the border of the grid wrapped around the space by using modulo arithmetic. The simulations were single-threaded and all entities were updated to ensure a more controlled, level comparison. In addition, no spatial partitioning optimisations were employed. By evaluating these metrics across both ECS implementations, we aim to reveal the distinct performance characteristics of each architecture. This includes their relative strengths in iteration efficiency, data locality, and setup overhead, as well as their scalability across varying entity counts.

5. Results

5.1. Frame Latency

Frame latency was benchmarked between the two ECS implementations with progressively increasing entity counts, to capture their

Table 1: *Descriptive statistics for frame latency delta F_Δ between the sparse-set and archetype-based ECSes. For all cumulative metrics, $N = 5000$. For the sake of brevity, the archetype-based ECS is abbreviated to 'Arch' and the sparse-set ECS as 'Sparse' in this table. Frame latency here is expressed in milliseconds.*

	Mean		Median		Std. Dev		Range	
Entity count	Arch	Sparse	Arch	Sparse	Arch	Sparse	Arch	Sparse
100	.828	.822	.797	.799	.571	.449	[.260, 38.650]	[.290, 30.140]
1000	.846	.837	.799	.798	.514	.446	[.340, 28.620]	[.430, 28.280]
10000	1.812	3.043	1.721	2.945	.403	.401	[1.560, 25.490]	[2.630, 21.660]
50000	7.510	13.941	7.410	13.819	1.176	.804	[7.060, 85.160]	[12.610, 20.050]

runtime performance under stress. Each benchmarked run captured latency data for the first 5,000 iterations using `std::chrono` for high-resolution timing. Frame timestamps were recorded at the start (T_1) and end (T_2) of the frame and latency delta calculated as $F_\Delta = (T_2 - T_1)$. As $N = 5,000$, a total of 5,000 measures of frame latency were recorded for each of the ECS types. A full table of descriptive statistics can be found in Table 1.

For 100 entities, frame latency was marginally higher for the sparse set implementation ($Mdn = .799$ ms) against the archetype ECS ($Mdn = .797$ ms). This difference did not achieve statistical significance however, U($N_A = 5000$, $N_S = 5000$) = 12400341.5, $z = -.691$, $p = .490$. In the case of 1,000 entities, frame latency was almost identical between the sparse set ECS ($Mdn = .798$ ms) and archetype implementations ($Mdn = .799$ ms). With such a negligible difference, no statistical significance was achieved, U($N_A = 5000$, $N_S = 5000$) = 12372266.5, $z = -.885$, $p = .376$.

With 10,000 simulated entities the difference in F_Δ was more pronounced between the two ECSes, with significantly lower latency in the archetype ($Mdn = 1.721$ ms) against the sparse-set ($Mdn = 2.945$ ms) implementation, U($N_A = 5000$, $N_S = 5000$) = 40203.5, $z = -86.320$, $p < .001$. A similar effect was found with 50,000 entities, with latency significantly lower in the archetype ECS ($Mdn = 7.410$ ms) against the sparse set implementation ($Mdn = 13.819$ ms), U($N_A = 5000$, $N_S = 5000$) = 11738.0, $z = -86.517$, $p < .001$. A visualisation of frame latency times as n increases can be found in Figure 6.

Aggregating frame-latency alongside entity count enabled correlation testing between the two variables. Entity count was found to be positively correlated to frame-latency, achieving significance in both the archetype-based ($r_s = .866$, p < .001, $N = 20000$) and sparse-set ($r_s = .869$, p < .001, $N = 20000$) prototypes.

5.2. Entity Instantiation Latency

A similar analysis was conducted with respect to the latency of instantiating entities initially. The latency I_Δ recorded for a given entity includes the entity's construction, association with components, and emplacement in the grid. Measurement of the setup latency I_Δ were calculated similarly to frame latency; `std::chrono` timestamps were recorded prior to entity instantiation (T_1) and immediately afterwards (T_2). The instantiation latency was calculated as $I_\Delta = (T_2 - T_1)$, and expressed in nanoseconds: this was conducted across 5,000 entities at the start of the simulation. The sparse-set implementation was found to have significantly lower instantiation latency ($Mdn = 1000.0$) compared to the archetype ECS across constructed entities ($Mdn = 6600.0$), U($N_A = 5000$, $N_S = 5000$) = 49047.0, $z = -88.263$, $p < .001$. The stark difference in latency between the two can be visualised in Figure 7. Additionally, full descriptive statistics between the two ECS types can be be found in Table 2.

Table 2: *Descriptive statistics of initial entity set-up times, across $N = 5000$ entities, for both ECS implementations. Values of set-up latency (I_Δ) are expressed in nanoseconds.*

	Archetype I_Δ	Sparse-set I_Δ
Mean	6883.280	1093.920
Median	6600.0	1000.0
Std. Dev	3373.053	1081.525
Range	[6100.0, 108000.0]	[900.0, 61400.0]

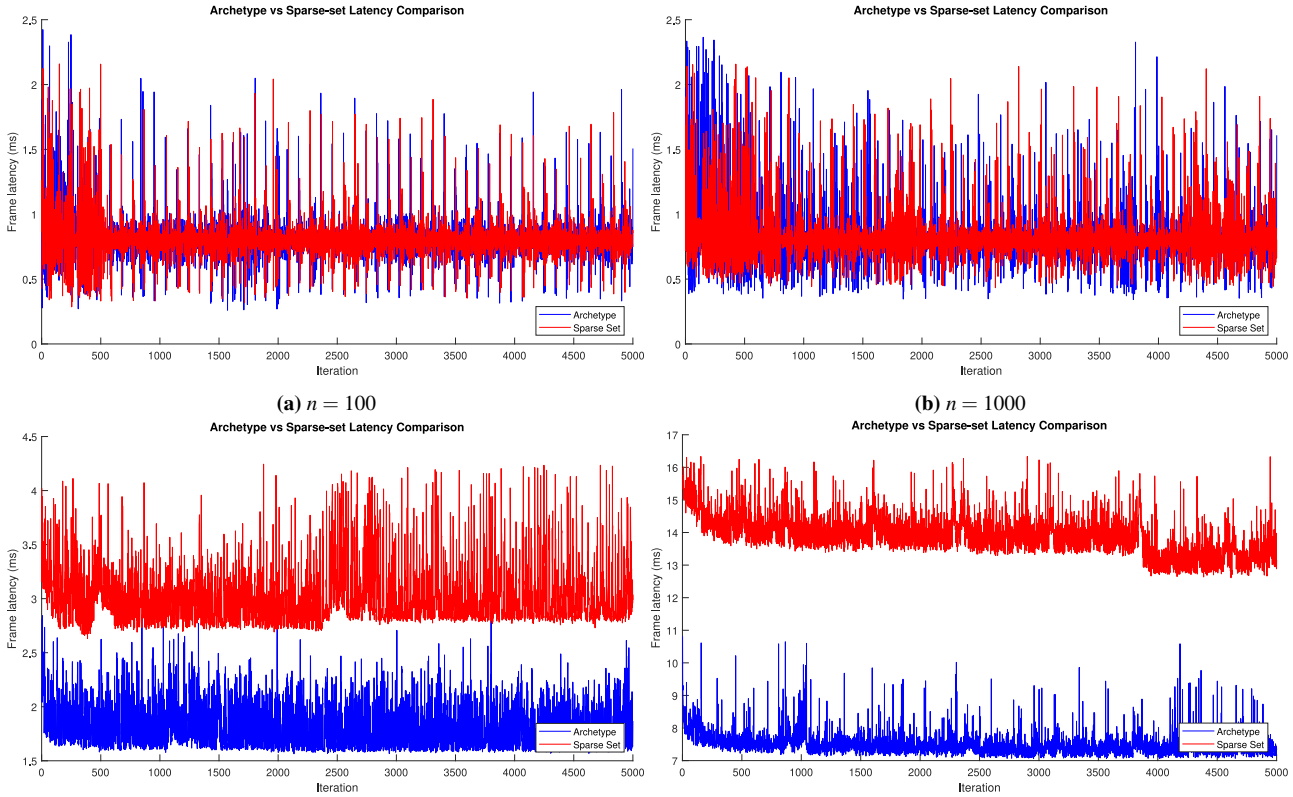
6. Discussion

6.1. Performance and Scalability

At lower entity counts (100 and 1,000 entities), there were no statistically significant differences in frame latency between the two implementations. This suggests that for small-scale simulations or systems where entity count remains low, either architecture may be equally suitable for runtime iteration efficiency. It also highlights the lack of utility the ECS paradigm has at small scales, something that has been seen in previous studies comparing ECS to OOD approaches [HÖ20].

For large entity counts however (10,000 entities and upwards), performance divergence is much more pronounced. This is seen visually in Figure 6. The archetype-based ECS consistently outperformed sparse-set implementations in frame update time at these higher scales. At 50,000 entities, the archetype ECS achieved nearly twice the performance of the sparse-set equivalent over the 5,000 frames. The trend observed here is likely attributed to the cache-coherent nature of archetypal storage of components. By storing entities with identical component compositions contiguously, archetypes can reduce cache misses and improve iteration throughput, especially under larger workloads. Sparse-set ECS variants, in contrast, can store componental data non-contiguously, requiring more indirect memory access, potentially leading to inefficiencies as entity counts grow.

Additionally, the sparse-set ECS demonstrated increased variance in update latency at higher scales. Between 10,000 and 50,000



(a) $n = 100$ **(b)** $n = 1000$

(c) $n = 10000$ **(d)** $n = 50000$

Figure 6: *A comparison of frame latency times between the two ECS implementations, for the first 5,000 iterations. In each of the figures, n refers to the number of simulated entities. Note the growing separation in frame latency as n increases. Please note that a extreme outliers have been filled to the nearest mean in these plots; the full data is available as additional material.*

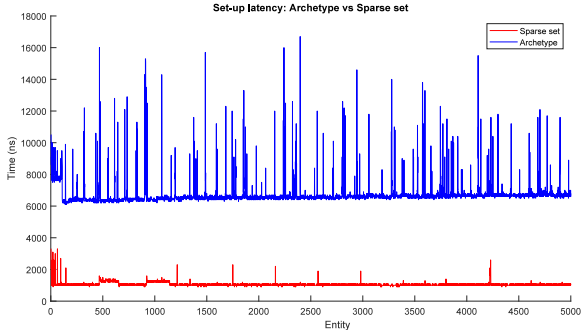


Figure 7: *Entity instantiation latency times (I_Δ) between the sparse-set and archetype ECS implementations. Latency is expressed in nanoseconds in this plot.*

entities, the deviation of frame times grew significantly more for the sparse-set ECS than for the archetype ECS, indicating less consistent run-time behaviour. Again, this is likely a result of less predictable memory access patterns and less efficient cache usage.

6.2. Entity Setup and Modification

In contrast to runtime iteration, the sparse-set ECS implementation consistently demonstrated lower entity instantiation costs. Across all benchmarks, the sparse-set ECS instantiated entities significantly faster than the archetype implementation—by an average factor of 6.6×. This difference reflects the lower overhead of sparse-set systems when adding or removing components, as entities are not grouped by composition and require no restructuring of memory. This makes the sparse-set ECS most suitable in contexts where structural changes are frequent, e.g. adding entities to the world, or restructuring the componental composition of entities. Sparse-set ECS types appear to be more suitable to dynamic environments where an entity's behaviour and composition can mutate frequently. However, where sparse-sets have lower costs associated with structural changes, they have conversely higher frame latency costs at large entity scales, and higher variance in update performance.

6.3. Summary of Trade-offs

The findings presented here reinforce the view that the optimal choice of ECS architecture is context dependent, a view shared by the literature [TM24,Här19,Cai21]. There is no clear choice of ECS

implementation: the optimal selection of sparse-sets or archetypes depends ultimately on the computational scenario. The careful consideration of the simulation and its performance is therefore crucial for any developer wishing to implement a form of ECS.

Archetypal ECSes exhibit higher performance in scenarios with high entity counts and little run-time structural changes. They benefit from strong cache coherency and produce generally consistent iteration speeds across a large amount of entities in the simulated world. If stability is paramount in the face of large entity counts, and entity composition is unlikely to change, archetype-based ECSes appear to be a good candidate for performant real-time simulations. They also suit video games which involve a large number of simulated elements of a few distinct archetypes. This makes archetypal ECSes a prime candidate for performance-critical systems such as physics engines. Sparse-set ECSes, by contrast, have lower costs associated with structural changes. When entities are created, destroyed, or modified in some way, the cost of doing so is drastically reduced in comparison to an archetypal ECS. From our findings, sparse set ECSes appear to be most suitable in scenarios where compositional modification is likely to occur. One example may be in the simulation of non-pooled particle systems, which likely have thousands of entities being modified in real-time. Sparse-sets also have lower overheads associated with faster initial set-up and instantiation times. This makes them an ideal candidate for compositional flexibility. Furthermore, sparse-set architectures are generally simpler to implement and maintain. In scenarios where ECS performance is not the primary bottleneck, this simplicity may make them a preferable design choice.

The primary trade-off between the two implementations is between compositional flexibility and scaling frame latency costs. Archetype-based ECSes perform well at scale, but have high costs associated with structural changes: they are inflexible but performant under stress. In contrast, the performance of sparse-set ECSes does not scale well with entity count, but they have significantly lower costs associated with compositional restructuring: they are flexible and less performant under stress. This trade-off must be carefully considered for the most suitable candidate to be selected, depending on the computational scenario the developer wishes to tackle.

7. Conclusion

This paper set out to compare the efficiencies of two ECS architectures in the creation and iteration of up to 50,000 unique entities. A custom-built C++ implementation of sparse set and archetype architectures was developed to leverage a controlled benchmarking process. The results reveal the key trade-offs inherent in each approach, providing valuable insights for developers seeking to optimise ECS usage.

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. Update latency scaled poorly in contrast to the archetype-based prototype. This could be attributed to more indirect memory access 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 enhanced cache coherence and reduced iteration overhead. This evidently resulted in more predictable performance under stressful computational workloads. However, the cost of modifying entity compositions was notably higher in the archetype-based ECS due to the overhead of structural changes. This resulted in slower setup and modification times compared to sparse sets.

Our findings offer practical insights into how different ECS storage models perform under real-world simulation workloads, supporting more informed architectural decisions for game engine developers and systems designers. We hope to inspire future work in this rapidly-growing field of Entity-Component Systems, which currently remains largely unexplored in the literature.

7.1. Future Work

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 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.

For future directions of research, an examination of other types of ECS implementations would be useful to the field. The consideration of parallelisation and the performance benefits it may bring into real-time ECSes would be another interesting avenue of work. Furthermore, a more robust comparison of performance data would provide value to the field: examining memory footprints between archetype-based and sparse-set ECSes, for example. In addition, examining if the effects observed in this paper are found with other benchmarks would be an interesting future direction. Finally, the comparison between hybrid and non-hybrid ECS variants may produce interesting findings.

References

[AA24] ANDERSSON G., ANDERSSON E.: *A Bloat-Free 3D Game Engine*. Bachelor's thesis, Halmstad University, 2024. 3

[AC07] AMPATZOGLOU A., CHATZIGEORGIOU A.: Evaluation of object-oriented design patterns in game development. *Information and Software Technology 49*, 5 (2007), 445–454. 2

[AL14] ASADUZZAMAN A., LEE H. Y.: Gpu computing to improve game engine performance. *Journal of Engineering & Technological Sciences*, 2 (2014). 2

[BR24] BREWER J., ROMINE M.: Breaking barriers in mobile game development. *Proceedings of the ACM on Human-Computer Interaction 8*, CHI PLAY (2024), 1–20. 1

[BT93] BRIGGS P., TORCZON L.: An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems (LOPLAS) 2*, 1-4 (1993), 59–69. 3

[C*70] CONWAY J., ET AL.: The game of life. *Scientific American 223*, 4 (1970), 4. 5

[Cai21] CAINI M.: Ecs back and forth. https://skypjack.github.io/2021-08-29-ecs-baf-part-12/, 2021. Accessed: June 2025. 3, 7

[Cho24] CHOPARINOV E. D.: *A Graph-Based Approach To Concurrent ECS Design*. PhD thesis, Universiteit van Amsterdam, 2024. 3

[CJV*24] CORREIA R. F., JAKUTIS L., VILKAITIS K., SURVILAITĖ E., VENCIUTE D., DE SOUSA J. P. P.: The effect of consumer participation during the new product development process on consumer brand identification: A gaming industry study. *Business Strategy & Development 7*, 2 (2024), e391. 2

[Com22] COMPTON B. V.: *An Investigation of Data Storage in Entity-Component Systems*. Master's thesis, Air Force Institute of Technology, 2022. 2, 3

[Den70] DENNING P. J.: Virtual memory. *ACM Computing Surveys (CSUR) 2*, 3 (1970), 153–189. 4

[DlTSGR25] DE LA TORRE-SIERRA A. M., GUICHOT-REINA V.: Women in video games: An analysis of the biased representation of female characters in current video games. *Sexuality & Culture 29*, 2 (2025), 532–560. 1

[EE05] EALES A., EALES A.: The observer pattern revisited. *Educating, Innovating & Transforming: Educators in IT* (2005). 2

[EW22] ENGELSTÄTTER B., WARD M. R.: Video games become more mainstream. *Entertainment Computing 42* (2022), 100494. 2

[Fab18] FABIAN R.: *Data-Oriented Design*. 2018. 2

[Far18] FARYABI W.: *Data-oriented Design approach for processor intensive games*. Master's thesis, Norweigian University of Science and Technology (NTNU), 2018. 2

[FAUM20] FEDOSEEV K., ASKARBEKULY N., UZBEKOVA A., MAZZARA M.: Application of data-oriented design in game development. In *Journal of Physics: Conference Series* (2020), vol. 1694, IOP Publishing, p. 012035. 2

[Fre16] FREIBERG S.: *Procedural generation of content in video games*. PhD thesis, Hochschule für angewandte Wissenschaften Hamburg, 2016. 3

[Gar21] GARLAND B. T.: *Designing and Building a Radar Simulation Using the Entity Component System*. Master's thesis, Air Force Institute of Technology, 2021. 2

[Gol04] GOLD J.: *Object-oriented game development*. Pearson Education, 2004. 2

[Gre18] GREGORY J.: *Game engine architecture*. AK Peters/CRC Press, 2018. 2

[Hal14] HALL D. M.: *ECS game engine design*. Master's thesis, California Polytechnic State University, 2014. 3

[Han16] HANISCH M.: *Konzeption und Evaluation eines Entity-Component-Systems anhand eines rundenbasierten Videospiels*. PhD thesis, Hochschule für angewandte Wissenschaften Hamburg, 2016. 2

[Här19] HÄRKÖNEN T.: *Advantages and Implementation of Entity-Component-Systems*. Bachelor's thesis, Tampere University, 2019. 1, 2, 3, 7

[HCSZ21] HATLEDAL L. I., CHU Y., STYVE A., ZHANG H.: Vico: An entity-component-system based co-simulation framework. *Simulation Modelling Practice and Theory 108* (2021), 102243. 3

[HÖ20] HANSEN H., ÖHRSTRÖM O.: *Benchmarking and Analysis of Entity Referencing Within Open-Source Entity Component Systems*. Bachelor's thesis, Malmö Universitet, 2020. 3, 6

[Hol19] HOLLMANN T.: *Development of an Entity Component System Architecture for Realtime Simulation*. PhD thesis, Bachelorarbeit, Koblenz, Universität Koblenz-Landau, Campus Koblenz, 2019. 3

[Joh25] JOHNSON R.: *Entity-Component System Design Patterns: Definitive Reference for Developers and Engineers*. HiTeX Press, 2025. 2

[LDV*19] LUCAS S. M., DOCKHORN A., VOLZ V., BAMFORD C., GAINA R. D., BRAVI I., PEREZ-LIEBANA D., MOSTAGHIM S., KRUSE R.: A local approach to forward model learning: Results on the game of life game. In *2019 IEEE Conference on Games (CoG)* (2019), IEEE, pp. 1–8. 5

[Leb17] LEBLANC M.: Game entities in thief: The dark project, 2017. 3

[Mer20] MERTENS S.: Building an ecs: Archetypes and vectorization. https://ajmmertens.medium.com/building-an-ecs-2-archetypes-and-vectorization-fe21690805f9, 2020. Accessed: June 2025. 3

[Nys14] NYSTROM R.: *Game programming patterns*. Genever Benning, 2014. 2

[PPM*21] POLITOWSKI C., PETRILLO F., MONTANDON J. E., VALENTE M. T., GUÉHÉNEUC Y.-G.: Are game engines software frameworks? a three-perspective study. *Journal of Systems and Software 171* (2021), 110846. 2

[Rau18] RAUTAKOPRA A.: *Game Design Patterns: Utilizing Design Patterns in Game Programming*. Bachelor's thesis, 2018. 2

[RH19] RAFFAILLAC T., HUOT S.: Polyphony: Programming interfaces and interactions with the entity-component-system model. *Proceedings of the ACM on Human-Computer Interaction 3*, EICS (2019), 1–22. 3

[SH84] SAKAI H., HORIUCHI H.: A method for behavior modeling in data oriented approach to systems design. In *1984 IEEE First International Conference on Data Engineering* (1984), IEEE, pp. 492–499. 2

[Sha80] SHARP J. A.: Data oriented program design. *ACM SIGPLAN Notices 15*, 9 (1980), 44–57. 2

[Str19] STRAUME P.-M.: *Investigating data-oriented design*. Master's thesis, Norweigian University of Science and Technology (NTNU), 2019. 2

[Tic23] TICHÝ Š.: *The Last Clan-RTS game in Unity*. Bachelor's thesis, Charles University, 2023. 3

[TM24] TARIFA MATEO Y.: *Entity component systems and data-oriented design*. Bachelor's thesis, Universitat Politècnica de Catalunya, 2024. 1, 2, 3, 7

[WBW89] WIRFS-BROCK R., WILKERSON B.: Object-oriented design: A responsibility-driven approach. *ACM sigplan notices 24*, 10 (1989), 71–75. 2

[WOWH24] WILLIAMS B., OLIVER T., WARD D., HEADLEAND C.: Real-time Data-Oriented Virtual Forestry Simulation for Games. In *Computer Graphics and Visual Computing (CGVC)* (2024), Hunter D., Slingsby A., (Eds.), The Eurographics Association. doi:10.2312/cgvc.20241218. 1, 2, 3

[Yli25] YLIKOSKI J.: *Transferring game mechanics to Unity DOTS and ECS after release*. Master's thesis, Lahti University of Technology (LUT), 2025. 2