The Essence of Entity Component System

Anisha Tasnim
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin, USA
tasnim@uwm.edu

Tian Zhao*
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin, USA
tzhao@uwm.edu

Abstract

Modern game engines increasingly adopt the Entity Component System (ECS) paradigm as a data-oriented alternative to traditional object-oriented architecture. While ECS promotes modularity and performance through the separation of data and behavior, its practical efficiency depends heavily on the underlying data layout. Despite widespread adoption in frameworks, such as Unity DOTS, Bevy, and Flecs, the semantics of the archetype ECS remain informal and implementation-dependent, limiting rigorous reasoning about determinism, system scheduling, and structural mutations.

This work formalizes and experimentally evaluates the archetype ECS. The formal model captures entity creation, component composition, system execution, and archetype migration as compositional state transitions, establishing the core invariants of archetype organization. Using a *Tower Defense* simulation, we compare the archetype ECS with alternative designs under identical conditions. Results show that the archetype ECS achieves higher frame rate and better frame stability than alternative designs, due to improved cache efficiency and consistent entity access. By uniting formal semantics with empirical validation, this study shows that the archetype ECS outperforms traditional architectures and provides a solid foundation for reasoning about correctness and parallelism.

CCS Concepts

• **Software and its engineering → Software performance**; **Semantics**.

Keywords

Entity Component System, Semantics, Type System, Simulation, Computer game

ACM Reference Format:
Anisha Tasnim and Tian Zhao. 2026. The Essence of Entity Component System. In *The 41st ACM/SIGAPP Symposium on Applied Computing (SAC '26), March 23–27, 2026, Thessaloniki, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3748522.3779910

1 Introduction

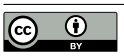
Modern game engines must efficiently manage thousands of simultaneously active objects, such as towers, enemies, bullets, and visual effects along with maintaining smooth, real-time performance. In a *Tower Defense* game, for example, hundreds of towers track moving enemies, bullets fly through the air, and particle systems update simultaneously. Each frame must process every active entity's position, health, and state within milliseconds to maintain 60 frames per second (FPS) rendering speed. The way these entities are represented and accessed in memory critically affects performance.

Historically, most game engines have used Object-Oriented Programming (OOP), where each object (e.g., Tower, Enemy, Bullet) encapsulates its data and methods within hierarchical class structures. While intuitive, this design becomes inefficient as the number of entities increases. Each object instance occupies a scattered region of memory, leading to poor cache utilization when iterating over thousands of objects during game-play. For example, updating all enemy positions requires dereferencing each object individually, a process that repeatedly jumps across memory, resulting in cache misses and reduced throughput [5, 6]. As game complexity and hardware parallelism increase, these unpredictable access patterns become a fundamental bottleneck.

To address these limitations, the Entity Component System (ECS) architecture emerged, representing a shift toward Data-Oriented Design (DOD). In ECS, entities are simple identifiers, components are plain data structures (e.g., Position, Velocity, Health), and systems define the logic that operates over sets of components. For instance, in *Tower Defense*, a movement system updates all entities that have both `Position` and `Velocity` components, and a collision system processes `Health` components. This approach decouples data from behavior, allowing systems to operate over contiguous blocks of homogeneous data and enabling efficient parallelization [7, 13].

However, the performance of ECS frameworks is deeply tied to their internal data layout. Early ECS implementations used an Array-of-Structs (AoS) layout, where each entity's components are grouped together in memory. This still leads to inefficient cache utilization during system-level iteration, such as systems only need one or two components from each entity. Struct-of-Arrays (SoA) layout improves this by storing each component type in its own contiguous array, enabling vectorized operations across entities and more predictable access patterns [1, 2].

Modern archetype-based ECS framework uses the SoA principle by grouping entities with identical component sets into archetypes, dense columnar tables where each column represents a component type and each row represents an entity. This design minimizes pointer chasing, enables constant-time component access, and leverages CPU cache lines efficiently. Frameworks such as Unity DOTS, Bevy, and Flecs adopt this design to achieve significant performance improvements in large-scale simulations [1, 9]. Moreover, recent work has extended ECS principles to high-performance domains. Madrona [14] used GPU to accelerate ECS for reinforcement learning environments, while Vico [8] used ECS for co-simulation across distributed systems, which demonstrated ECS's generality as a concurrent computation model.



This work is licensed under a Creative Commons Attribution 4.0 International License.
SAC '26, Thessaloniki, Greece
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2294-3/2026/03
https://doi.org/10.1145/3748522.3779910

Despite these advances, the semantics of archetype ECS frameworks remain informal, often described operationally without formal grounding. This lack of formalism makes it difficult to reason about determinism, system scheduling, and structural mutations. As ECS designs increasingly influence simulation engines and parallel runtime systems, establishing a formal semantics for archetype ECS becomes crucial.

In this research, we make the following contributions:

(1) An operational semantics is defined to capture the core mechanisms of archetype ECS, such as entity creation, component association, system execution, and archetype migration in a compositional and deterministic manner. This semantics establishes the basic invariants of archetype-based data organization, showing that ECS execution can be described as a series of stable state transitions.
(2) A type system is defined for archetype ECS to ensure that a well-typed ECS program will not have unsafe access to archetype storage during runtime execution.
(3) An archetype SoA framework is implemented in Scala and its performance is compared with that of an object-oriented and an AoS implementation using a *Tower Defense* simulation [16]. The results demonstrate that the archetype SoA achieves higher frame stability.

In the rest of the paper, we discuss related works in Section 2 and present a motivational example in Section 3. In Section 4, we give an operational semantics to model the evaluation of ECS programs. Section 5 defines a type system for ECS programs to prevent runtime errors due to unsafe access to archetype storage. Section 6 describes an implementation of the *Tower Defense* simulation in ECS. Section 7 compares the performance of *Tower Defense* implemented in OOP, AoS, and archetype SoA, which shows that SoA has better gameplay performance than the other two designs.

2 Related Work

Early game engines predominantly relied on object-oriented designs, where data and behavior were encapsulated within rigid inheritance hierarchies. These architectures encouraged modular encapsulation but scaled poorly as game logic became increasingly complex. The deep class hierarchies and runtime polymorphism typical of object-oriented designs produced non-deterministic memory access and poor cache utilization, leading to inefficiencies in highly parallel workloads [5, 6]. As hardware concurrency increased, these limitations prompted researchers to explore data-oriented designs that treat computation as transformations on structured memory layouts rather than class abstractions.

ECS emerged as a response to scalability challenges. ECS decomposes game logic into three dimensions: entities (identifiers), components, and systems (behavioral processors). This decomposition follows the principles of DOD, prioritizing predictable data access patterns and cache locality over inheritance-based flexibility [7, 13]. Unlike traditional object-oriented entities, ECS entities have no intrinsic behavior. In ECS, functionality arises through the combination of components and the systems that operate on them. This architectural separation enables composition over inheritance, enables runtime reconfiguration, and simplifies parallelization across systems.

While ECS successfully decouples data and logic, the efficiency of its implementation depends primarily on the internal data layout used to store components. Early AoS models retained per-entity memory organization, where all components belonging to a single entity were stored contiguously. Although this design simplified access, it resulted in fragmented iteration patterns during system updates. In contrast, in SoA layouts, each component type is stored in a contiguous array. This improves spatial and temporal locality and enables Single Instruction Multiple Data (SIMD) vectorization, which accelerates iteration across large datasets [2].

Modern archetype ECS architectures build upon the SoA model by grouping entities with identical component sets into archetypes. Each archetype is stored as a columnar table, where columns represent component types and rows represent entity instances. This organization eliminates pointer chasing and supports constant-time component lookup through column indexing [2]. Studies have demonstrated that this design achieves significant gains in update throughput and cache coherence compared to sparse-set ECS implementations [4].

Several widely adopted frameworks embody these principles. Unity's DOTS architecture introduced chunk-based archetype storage with Burst-compiled jobs to mitigate cache inefficiency and main-thread contention [15]. Bevy ECS, implemented in Rust, integrates archetype tables with compile-time type safety through the Rust ownership model, achieving memory-safe, cache-optimized traversal [1]. Similarly, Flecs [9, 10], a C-based archetype ECS, organizes entities into dense tables and defers world modifications through command queues, ensuring thread-safe yet deterministic execution in parallel environments. These frameworks illustrate how archetype ECS designs maximize spatial locality and enable predictable and high-throughput updates suitable for large-scale simulations such as Tower Defense [11].

The evolution of ECS has also expanded beyond game engines into broader high-performance and concurrent simulation contexts. Vico [8] applied ECS to distributed co-simulation, showing improved scalability through composition and fine-grained task isolation. Madrona [14] demonstrated a GPU-accelerated ECS runtime capable of executing large batched reinforcement learning environments as unified "megakernels", achieving orders-of-magnitude speedups over CPU baselines. Cox *et al.* [4] benchmarked archetype and sparse-set ECS designs using John Conway's *Game of Life* [3], showing that archetype implementations nearly doubled iteration throughput at high entity counts, while sparse sets excelled in dynamic environments due to lighter update overhead. Fedoseev *et al.* [5] compared Unity's object-oriented and DOD-based ECS prototypes, observing improved frame stability and reduced CPU load in the DOD version. These findings suggest that ECS designs and archetype-based SoA layouts in particular can achieve superior frame consistency and computational throughput in simulation-heavy workloads.

Despite extensive engineering refinement, the formal semantics of archetype ECS remain underdeveloped. Most existing frameworks describe their execution behavior operationally, relying on implementation heuristics rather than rigorous formalism. As ECS systems increasingly influence simulation engines, AI frameworks, and parallel schedulers, a formal model is necessary to reason about

determinism, component migration, and synchronization invariants. Empirical comparisons reinforce the efficiency of archetype-based ECS systems. The absence of such semantics limits theoretical analysis and compiler-level optimization, motivating a structured, language-theoretic definition of ECS execution.

The most closely related work is Core ECS [12], which is a formalism that captures the semantics of ECS system focusing on concurrency and deterministic scheduling. It showed how safe schedule can be constructed and that schedule safety implies schedule determinism. In contrast, we model the archetype-based ECS architecture and provide an operational semantics along with type system that exposes precise read-write sets for every system. This enables us to reason statically about conflict freedom and system compatibility. While Core ECS focuses primarily on scheduling and deterministic concurrency, our approach provides a type-directed mechanism for detecting structural conflicts at compile time that is not expressible in Core ECS. Furthermore, our model directly reflects the concrete memory layout and archetype transitions used in real-world ECS engines.

3 Examples

This section introduces an example that illustrates the challenges addressed by our formalism. In particular, it demonstrates how ECS systems interact through shared archetypes, how structural mutations must be deferred to preserve iteration stability, and how read-write overlaps create the conflicts formalized in Section 4.

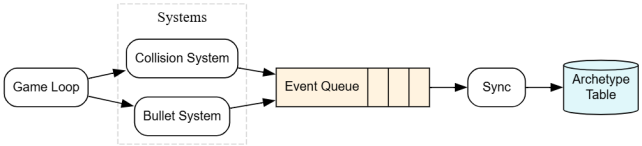


Figure 1: Illustration of an ECS game loop, where queued events are synchronized to archetype table.

In *Tower Defense*, the efficiency of the game loop depends on how entities and their data are stored in memory. Figure 1 illustrates a part of the per-frame execution workflow of the ECS, which runs a `BulletSystem` and a `CollisionSystem`. The systems iterate over stable archetype tables, generate structural events, and defer these mutations until a subsequent synchronization step. The figure shows three main properties of archetype ECS: (1) stable iteration sets, ensuring that systems observe a consistent snapshot of entity data; (2) event staging, which defers all structural modifications until after system execution; (3) all deferred mutations synchronized in an ordered batch to reestablish archetype consistency. These mechanisms illustrate the interaction between entity iteration and structural mutation that our semantics captures through conflict modeling and tracking of archetypes dirtied by deferred mutations.

Each ECS system operates independently over entities with matching component sets. For example, `BulletSystem` in Listing 1 takes all entities with `Bullet` component. When a bullet exits the game boundary, it must be removed but directly modifying the world during iteration can corrupt archetype tables. To maintain structural integrity, such changes are issued as queued events and processed in a later synchronization (`world.sync()`) phase. This decoupling between event generation and structural mutation is essential for maintaining stable SoA memory layout and preventing mid-iteration inconsistencies.

Listing 1: Bullet system

```
class BulletSystem(mapWidth=200f, mapHeight=200f) extends Systems {
  override val queryComponents = Set(classOf[Bullet])

  override def update(world: World, dt: Float): Unit = {
    val view = world.queryRows(queryComponents)

    view.foreach { r =>
      val b = r.table.getAt[Bullet](r.row, classOf[Bullet])

      b.x += b.dx * b.speed * dt
      b.y += b.dy * b.speed * dt
      b.ttl -= dt

      // Bounds & lifetime
      val outOfBounds =
        b.x < 0f || b.y < 0f || b.x > mapWidth || b.y > mapHeight

      if (b.ttl <= 0f || outOfBounds)
        world.enqueueEvent(Destroy(r.entity))
    }
  }
}
```

`BulletSystem` and `CollisionSystem` illustrate how our semantics handles system interactions and detects conflicts. The bullet archetype contains entities with the components `Bullet` and `Position` and the enemy archetype contains entities with the components `Position`, `Health`, and `Speed`. `BulletSystem` iterates over the bullet archetype, reading each bullet's position and time-to-live, updating its state, and staging `Destroy(bullet)` events when a bullet expires or leaves the map bounds. `CollisionSystem` reads the enemy archetype to identify alive enemies, and reads the bullet archetype to test proximity between each bullet and its targeted enemy. When a hit is detected, it writes to the enemy archetype by decrementing enemy health and stages a `Destroy(bullet)` event.

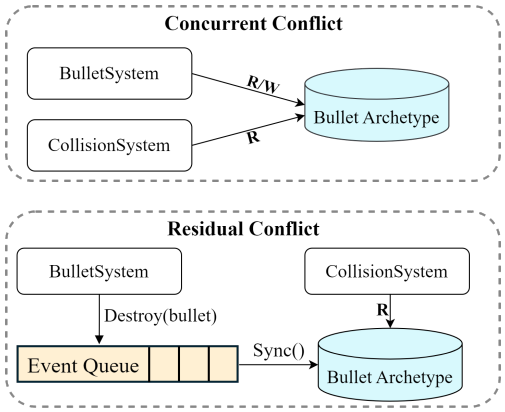


Figure 2: Illustration of conflicts during the execution of `BulletSystem` and `CollisionSystem`.

As illustrated in Figure 2, though the two systems do not iterate the same entities, their access patterns over the bullet archetype produce conflicts. For example, `BulletSystem` writes to the bullet archetype, while `CollisionSystem` reads from it, creating a *concurrent conflict*. The systems also stage `Destroy(bullet)` events, which leave the bullet archetype in a temporary dirty state until a synchronization step commits the structural updates, creating a *residual conflict*. To prevent concurrent conflict, `BulletSystem` and `CollisionSystem` cannot run in parallel. To prevent residual conflict, `Destroy(bullet)` events must be synchronized before running systems that access the bullet archetype.

4 Semantics

In this section, we model the behavior of ECS programs via an operational semantics, which is a formal model describing how each "step" in the system modifies stored data or interacts with entities and components. The semantics model terms, systems, and game state. By specifying precise semantic rules, we clarify how systems iterate over entities, change state, and maintain the ECS data reliably. Runtime errors include *conflicts* caused by unsafe access to archetypes in systems. A type system is defined for terms, systems, and game state so that a well-typed ECS program will not have conflicts and other runtime errors.

4.1 Syntax

Our ECS model adopts the syntax in Figure 3 to formally describe entities, components, and systems. Primitive types P represent the basic data types available in the ECS, such as Int, Float, and Boolean.

Entities are represented by unique integers. Component types C define data objects, each encapsulating multiple fields of primitive types P_i. Components consist of primitive values v_i corresponding to their declared types. An archetype is a set of component types.

A system $(A, \lambda x.t)$ includes an archetype A and a function that processes all entities of A. In $\lambda x.t$, x is the variable for entities and t is the operation performed on the entities and their components. Although a system can have multiple archetypes, this syntax only have one for simplicity. The systems can be sequential, data parallel, or task parallel. A sequential system will run in a single thread. A data parallel system can run in multiple threads by splitting the entities of the system among the threads. The task parallel system represents two or more systems running in parallel.

ECS libraries like Flecs [9] use query mechanism to retrieve archetypes that satisfy some query conditions. We do not model query mechanism since the queries of task-parallel systems may return the same archetypes at runtime and safe treatment of queries complicates the formalism.

Term t in Figure 4 defines computational expressions within our ECS model. The terms include values, variables, conditionals, sequence, read and write entity components, and events. For simplicity, we omit computations such as function calls, local variables, and assignments. Event *evt* is the entity's life-cycle operations. An entity can be created or destroyed; a component can be added to or removed from an entity. The events may be staged (*lz evt*) or immediate (*im evt*). The immediate events can only be safely evaluated in a single-threaded system. Thus, most events are staged, which are placed in the event queue until they can be safely processed.

P	$\in$	$Primitive$	$=$	$Int \mid Float \mid \ldots$	
C	$\in$	$CType$	$=$	$\{P_1, \ldots, P_n\}$	Component type
A	$\in$	$AType$	$=$	$\{C_1, \ldots, C_n\}$	Archetype
e	$\in$	$Entity$	$=$	n	Entity ID
c	$\in$	$Component$	$=$	$\{v_1, \ldots, v_n\}$	Component value
s	$\in$	$System$	$=$	$(A, \lambda x.t)$	Sequential system
			$\mid$	$\texttt{par}\ (A, \lambda x.t)$	Data parallel system
			$\mid$	$[s_1, s_2]$	Task paralllel system

Figure 3: The syntax of entities, components, and systems.

t	$\in$	$Term$	$=$	v	Value
			$\mid$	x	Variable
			$\mid$	$\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2$	Conditional
			$\mid$	$t;\ t'$	Sequence
			$\mid$	$t.C$	Read component
			$\mid$	$t.C := t'$	Update component
			$\mid$	$lz\ evt$	Staged event
			$\mid$	$im\ evt$	Immediate event
v	$\in$	$Value$	$=$	n	Primitive value
			$\mid$	e	Entity
			$\mid$	c	Component value
evt	$\in$	$Event$	$=$	$create\ (C, c)$	Create entity
			$\mid$	$destroy\ x$	Destroy entity
			$\mid$	$add\ (x, C, c)$	Add component
			$\mid$	$remove\ (x, C)$	Remove component

Figure 4: The syntax for terms, values, and events.

4.2 Archetype and Storage

In an archetype ECS model, an entity is grouped with others sharing the same archetype, allowing for efficient processing. Archetype stores similar entities in table-like storage, where each row is dedicated to an entity and component values are stored in columns. For convenience, we use $\mathcal{H}$ to represent the heap storage for archetypes, where $\mathcal{H}[A]$ returns the entities of an archetype A and $\mathcal{H}[e][C]$ returns the component C of entity e.

$$\begin{array}{lcll} \mathcal{H} & \in & Heap & \stackrel{def}{=} & AType \rightarrow \{Entity\} \\ & & & \cup & Entity \rightarrow (Ctype \rightarrow Component) \end{array}$$

4.3 Game State and Frames

The game state is defined as a tuple of heap, frame, and event queue, $(\mathcal{H}, fr, q)$. A frame contains the scheduled systems or synchronization steps that execute the staged events in the event queue.

w	$\in$	$State$	$=$	$(\mathcal{H},\ fr,\ q)$	Game state
fr	$\in$	$Frame$	$=$	$[sync]$	sync before next frame
			$\mid$	$s :: fr$	run a system s
			$\mid$	$sync :: fr$	execute staged events
q	$\in$	$Queue$	$=$	$[evt_1, \ldots, evt_n]$	Event queue

4.4 Conflicts

ECS supports data parallelism (a system runs in parallel threads where each thread operates on a separate set of entities) and task parallelism (multiple systems run in parallel). Since a system reads and/or writes the entities of archetypes or modify the archetypes (events), ECS must ensure that there is no conflicts between the threads that can lead to concurrency errors.

The events cannot be executed in parallel since they modify archetype tables, which are not thread safe. Thus, events are often staged in a queue, which can be applied safely in a single thread between the run of the systems. If a system does not have immediate events, it is safe to run it in parallel threads since each thread operates on a separate partition of the archetype table and the staged events do not affect the archetype tables until they are applied.

In practice, not all events can be staged. For example, in the `CollisionSystem`, a bullet that collided with an enemy should be destroyed immediately so that the same bullet cannot hit multiple enemies. In this case, it is not safe run `CollisionSystem` in parallel since race condition will occur due to concurrent modification to bullet archetype table. Two or more systems can run in parallel if they do not have read/write access to the same archetype tables.

The staged events must be applied before running a system that depends on the archetypes modified by the events. For example, if a bullet entity is destroyed in `CollisionSystem` but the render system runs before the destruction event is applied, then the destroyed bullet will be rendered.

In summary, there are two types of conflicts.

Concurrent Conflict. This occurs when the systems running in parallel threads have concurrent access to the same archetype tables (read/write the same entities or add/remove/modify entities). To avoid this, systems that have conflicting access to the same archetype tables cannot run in parallel. Also, a system with immediate events cannot run in parallel.

Residual Conflict. This occurs when a system uses an archetype that was dirtied by a previous system via staged events. Before executing a system, we must ensure that no archetype required by the system is dirtied. A synchronization barrier can also be placed before a system that accesses previously dirtied archetypes, ensuring that all staged events are fully applied and the archetype becomes clean again.

4.5 Event Evaluation

Figure 5 shows the evaluation rules for events, which include entity creation, entity destruction, and adding component to or removing component from an entity. $\mathcal{H}, evt \to \mathcal{H}', v$ represents the evaluation of evt that changes the storage from $\mathcal{H}$ to $\mathcal{H}'$.

$$\frac{e \text{ is a fresh integer} \quad \mathcal{H}_1 = \mathcal{H}[e \mapsto \{C \mapsto c\}] \quad A = \{C\} \quad \mathcal{H}_2 = \mathcal{H}_1[A \mapsto \mathcal{H}[A] \cup \{e\}]}{\mathcal{H}, create(C, c) \to \mathcal{H}_2, ()} \text{ E-Cre}$$

$$\frac{e \in \mathcal{H}[A] \quad \mathcal{H}' = \mathcal{H}[A \mapsto \mathcal{H}[A] - \{e\}]}{\mathcal{H}, destroy\ e \to \mathcal{H}', ()} \text{ E-Des}$$

$$\frac{e \in \mathcal{H}(A) \quad \mathcal{H}_1 = \mathcal{H}[e \mapsto \mathcal{H}[e][C \mapsto c]] \quad \mathcal{H}_2 = \mathcal{H}_1[A \mapsto \mathcal{H}_1[A] - \{e\}] \quad A_1 = A \cup \{C\} \quad \mathcal{H}_3 = \mathcal{H}_2[A_1 \mapsto \mathcal{H}_2[A_1] \cup \{e\}]}{\mathcal{H}, add(e, C, c) \to \mathcal{H}_3, ()} \text{ E-Add}$$

$$\frac{e \in \mathcal{H}(A) \quad \mathcal{H}_1 = \mathcal{H}[e \mapsto \mathcal{H}[e][C \mapsto \bot]] \quad \mathcal{H}_2 = \mathcal{H}_1[A \mapsto \mathcal{H}_1[A] - \{e\}] \quad A_1 = A - \{C\} \quad \mathcal{H}_3 = \mathcal{H}_2[A_1 \mapsto \mathcal{H}_2[A_1] \cup \{e\}]}{\mathcal{H}, remove(e, C) \to \mathcal{H}_3, ()} \text{ E-Rem}$$

Figure 5: The rules for event reduction.

A new entity is created with a component type and its initial value. When an entity is destroyed, it is removed from the archetype table where it was stored. Adding or removing a component from an entity changes the archetype of the entity. Thus, this entity is first removed from the old archetype table and then added to the new table. The component value of the entity is also updated.

4.6 Term Evaluation

Figure 6 shows the small-step evaluation rules for terms, where $\mathcal{H}, t, q \to \mathcal{H}', t', q'$ reduces term t to t' with possibly new storage and queue. Most rules are standard including reading/writing components of an entity. A staged event *lz evt* (*lz* is short for lazy) is added to the event queue. An immediate event *im evt* is evaluated to a value resulting a new archetype storage.

4.7 System Evaluation

Figure 7 shows the evaluation rules for systems, which can be sequential, data parallel, or system parallel. A system $(A, \lambda x.t)$ is evaluated by calling $\lambda x.t$ with each entity of the archetype A. Rule (S-Iter) describes how $\lambda x.t$ iterates over entity list es, denoted by $(\lambda x.t)@es$. The notation $\to^*$ means transitive closure of $\to$.

Sequential system. A system $s = (A, \lambda x.t)$ runs over the entities es of A. In the sequential mode, it simply iterates over each entity in es, applying t, and possibly staging events.

Data parallel system. If a system needs to process a large number of entities, it can process partitions of the entities in separate threads. For simplicity, Rule (S-Data-P) only shows two partitions. Each thread returns a new archetype heap $\mathcal{H}_i$ and queue q_i. The queues are combined. The heaps are merged, where the archetype tables must not change and only the component values of entities may be updated. This rule precludes immediate events in parallel threads.

$$\mathcal{H},\ \texttt{if}\ \mathit{true}\ \texttt{then}\ t\ \texttt{else}\ t',\ q \to \mathcal{H},\ t,\ q \qquad \text{R-If-True}$$

$$\mathcal{H},\ \texttt{if}\ \mathit{false}\ \texttt{then}\ t\ \texttt{else}\ t',\ q \to \mathcal{H},\ t',\ q \qquad \text{R-If-False}$$

$$\mathcal{H},\ ();t,\ q \to \mathcal{H},\ t,\ q \qquad \text{R-Seq}$$

$$\frac{e \in \mathcal{H}[A] \quad C \in A}{\mathcal{H},\ e.C,\ q \to \mathcal{H},\ \mathcal{H}[e][C],\ q} \qquad \text{R-Read-Comp}$$

$$\frac{e \in \mathcal{H}[A] \quad C \in A \quad \mathcal{H}' = \mathcal{H}[e \mapsto \mathcal{H}[e][C \mapsto v]]}{\mathcal{H},\ e.C := v,\ q \to \mathcal{H}',\ (),\ q} \qquad \text{R-Write-Comp}$$

$$\mathcal{H},\ lz\ evt,\ q \to \mathcal{H},\ (),\ evt :: q \qquad \text{R-Staged-Event}$$

$$\frac{\mathcal{H},\ evt \to \mathcal{H}', v}{\mathcal{H},\ im\ evt,\ q \to \mathcal{H}',\ v,\ q} \qquad \text{R-Immedi-Event}$$

$$\frac{\mathcal{H},\ t,\ q \to \mathcal{H}',\ t',\ q'}{\mathcal{H},\ \mathcal{E}[t],\ q \to \mathcal{H}',\ \mathcal{E}[t'],\ q'} \qquad \text{R-Context}$$

$$\mathcal{E}[\cdot] \quad = \quad \cdot \mid \mathcal{E}[\cdot];t \mid \texttt{if}\ \mathcal{E}[\cdot]\ \texttt{then}\ t\ \texttt{else}\ t' \mid \mathcal{E}[\cdot].C \mid \mathcal{E}[\cdot].C = t \mid e.C = \mathcal{E}[\cdot]$$

Figure 6: The rules for term reduction.

$$\frac{es = [e_1, \ldots, e_n] \quad \forall i \in [1..n].\ \mathcal{H}_{i-1},\ [e_i/x]t,\ [] \to^* \mathcal{H}_i, v_i,\ q_i}{\mathcal{H}_0, \lambda x.t\ @\ es \to \mathcal{H}_n,\ q_1 + \ldots + q_n} \qquad \text{S-Iter}$$

$$\frac{es = \mathcal{H}[A] \quad \mathcal{H}, \lambda x.t\ @\ es \to \mathcal{H}',\ q}{\mathcal{H},\ (A, \lambda x.t) \to \mathcal{H}',\ q} \qquad \text{S-Seq}$$

$$\frac{es = es_1 \cup es_2 = \mathcal{H}[A] \quad es_1 \cap es_2 = \emptyset \quad i \in \{1,2\}.\ \mathcal{H},\ \lambda x.t\ @\ es_i \to \mathcal{H}_i,\ q_i}{\mathcal{H},\ \texttt{par}\ (A, \lambda x.t) \to merge(\mathcal{H}, \mathcal{H}_1, \mathcal{H}_2),\ q_1 + q_2} \qquad \text{S-Data-P}$$

$$\frac{\forall i \in \{1,2\}.\ \mathcal{H},\ s_i \to \mathcal{H}_i,\ q_i \quad query(s_1) \cap query(s_2) = \emptyset}{\mathcal{H},\ [s_1, s_2] \to merge(\mathcal{H}, \mathcal{H}_1, \mathcal{H}_2),\ q_1 + q_2} \qquad \text{S-Task-P}$$

$$\frac{\mathcal{H},\ s \to \mathcal{H}',\ q' \quad no_conflict(\mathcal{H}, s, q)}{\mathcal{H},\ s :: fr,\ q \to \mathcal{H}',\ fr,\ q + q'} \qquad \text{F-System}$$

$$\frac{\forall evt_i \in q = [evt_1, \ldots, evt_n].\ \mathcal{H}_{i-1},\ evt_i \to \mathcal{H}_i, ()}{\mathcal{H}_0,\ sync :: fr,\ q \to \mathcal{H}_n,\ fr,\ nil} \qquad \text{F-Sync}$$

Figure 7: The rules for system and frame evaluation.

Task parallel system. A list of systems can run in parallel threads. The restriction is similar to that of the data parallel systems except that task parallel systems have different archetypes. For simplicity, Rule (S-Task-P) only shows two systems.

$$query(A, \lambda x.t) = \{A\} \quad query(\texttt{par}\ s) = query(s)$$

$$query([s_1, s_2]) = query(s_1) \cup query(s_2)$$

$$\frac{\forall A.\ \mathcal{H}'[A] = \mathcal{H}[A] = \mathcal{H}_1[A] = \mathcal{H}_2[A] \quad \forall e, i.\ \text{if}\ \mathcal{H}_i[e] \neq \mathcal{H}[e]\ \text{then}\ \mathcal{H}'[e] = \mathcal{H}_i[e]\ \text{else}\ \mathcal{H}'[e] = \mathcal{H}[e]}{merge(\mathcal{H}, \mathcal{H}_1, \mathcal{H}_2) = \mathcal{H}'}$$

$$\frac{A \notin atype(\mathcal{H}, q)}{no_conflict(\mathcal{H},\ (A, \lambda x.t),\ q)}$$

$$\frac{no_conflict(\mathcal{H}, s, q)}{no_conflict(\mathcal{H}, \texttt{par}\ s, q)} \qquad \frac{\forall i \in \{1,2\}.\ no_conflict(\mathcal{H}, s_i, q)}{no_conflict(\mathcal{H}, [s_1, s_2], q)}$$

$$\begin{aligned}
atype(\mathcal{H}, create(C, c)) &= \{\{C\}\} \\
atype(\mathcal{H}, destroy\ e) &= \{A\}\ \text{where}\ e \in \mathcal{H}(A) \\
atype(\mathcal{H}, add(e, C, c)) &= \{A, A \cup \{C\}\}\ \text{where}\ e \in \mathcal{H}(A) \\
atype(\mathcal{H}, delete(e, C)) &= \{A, A - \{C\}\}\ \text{where}\ e \in \mathcal{H}(A) \\
atype(\mathcal{H}, q) &= \bigcup_{evt \in q} atype(\mathcal{H}, evt)
\end{aligned}$$

Figure 8: Auxiliary functions.

4.8 Frame Reduction

A game state is a triple of archetype heap $\mathcal{H}$, a frame fr, and an event queue q. A frame is a list of systems and sync operations. The transition of a state is the execution of the system or sync on top of the frame, which is defined in Figure 7. The predicate $no_conflict(\mathcal{H}, s, q)$ ensures that the entities of the archetype accessed by s is not dirtied by events in q.

The sync operation applies all queued events in order and then clears the event queue. This design is chosen for simplicity. In practice, an ECS library can automatically execute events if they are in conflict with the next system in the frame. When a frame completes, the game loop can restart with the same frame or dynamically reconfigure the frame with new systems.

5 Type System

Our ECS model has a simple set of types τ, which include primitive types, unit type, component types, and archetypes, which are sets of component types.

$$\begin{aligned}
\tau \quad = \quad & P && \text{Primitive type} \\
\mid \quad & () && \text{Unit type} \\
\mid \quad & C && \text{Component type – set of primitive types} \\
\mid \quad & A && \text{Archetype – set of component types}
\end{aligned}$$

We use a type and effect system to track the read and write operations to archetypes during system computation. The type judgment for term has the form $\Gamma \vdash t : \tau\ \&\ W$, where given the variable environment Γ, τ is the type of the term t and W is the set

$$\Gamma \vdash x : \Gamma(x) \ \& \ \emptyset \qquad \text{T-Var}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \ \& \ W_1 \quad \Gamma \vdash t_2 : \tau_2 \ \& \ W_2}{\Gamma \vdash (t_1; t_2) : \tau_2 \ \& \ W_1 \cup W_2} \qquad \text{T-Seq}$$

$$\frac{\Gamma \vdash t : \mathit{Bool} \quad \Gamma \vdash t_1 : \tau \ \& \ W_1 \quad \Gamma \vdash t_2 : \tau \ \& \ W_2}{\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2 : \tau \ \& \ W_1 \cup W_2} \qquad \text{T-If}$$

$$\frac{C \in \Gamma(x)}{\Gamma \vdash x.C : C \ \& \ \emptyset} \qquad \frac{C \in \Gamma(x) \quad \Gamma \vdash t : C \ \& \ W}{\Gamma \vdash x.C = t : C \ \& \ W} \qquad \text{T-Comp}$$

$$\frac{\Gamma \vdash \mathit{evt} : () \ \& \ W}{\Gamma \vdash \mathit{lz}\ \mathit{evt} : () \ \& \ W} \qquad \frac{\Gamma \vdash \mathit{evt} : () \ \& \ W}{\Gamma \vdash \mathit{im}\ \mathit{evt} : () \ \& \ \emptyset} \qquad \text{T-Event}$$

Figure 9: Term typing rules.

$$\Gamma \vdash \mathit{create}(C, c) : () \ \& \ \{\{C\}\} \qquad \text{T-Create}$$

$$\frac{\Gamma \vdash t : \tau \ \& \ W}{\Gamma \vdash \mathit{destroy}\ t : () \ \& \ W \cup \{\tau\}} \qquad \text{T-Destroy}$$

$$\frac{\Gamma \vdash t : \tau \ \& \ W \quad C \notin \tau \quad \tau' = \tau \cup \{C\}}{\Gamma \vdash \mathit{add}\ (t, C, c) : () \ \& \ W \cup \{\tau, \tau'\}} \qquad \text{T-Add}$$

$$\frac{\Gamma \vdash t : \tau \ \& \ W \quad C \in \tau \quad \tau' = \tau - \{C\}}{\Gamma \vdash \mathit{remove}\ (t, C) : () \ \& \ W \cup \{\tau, \tau'\}} \qquad \text{T-Remove}$$

Figure 10: Event typing rules.

$$\frac{x : A \vdash t : \tau \ \& \ W \quad t \text{ has no staged events}}{\vdash (A,\ \lambda x.t) : (\{A\}, W)} \qquad \text{T-Seq}$$

$$\frac{\vdash s : (R, W) \quad s \text{ has no immediate events}}{\vdash \texttt{par } s : (R, W)} \qquad \text{T-Data-P}$$

$$\frac{\begin{array}{c} s_1,\ s_2 \text{ have no immediate events} \\ \vdash s_1 : (R_1, W_1) \quad \vdash s_2 : (R_2, W_2) \quad R_1 \cap R_2 = \emptyset \end{array}}{\vdash [s_1, s_2] : (R_1 \cup R_2, W_1 \cup W_2)} \qquad \text{T-Task-P}$$

$$\frac{\vdash fr : \emptyset}{\vdash (\mathit{sync} :: fr) : \mathcal{D}} \qquad \text{T-Sync}$$

$$\frac{\vdash s : (R, W) \quad R \cap \mathcal{D} = \emptyset \quad \vdash fr : W \cup \mathcal{D}}{\vdash (s :: fr) : \mathcal{D}} \qquad \text{T-System}$$

Figure 11: Frame and system typing rules.

of archetypes that may be written by staged events in t. The typing rules in Figure 9 and 10 collect the archetypes of the staged events and put them in the write set W. The archetypes of the immediate events are not tracked since they are evaluated sequentially.

The type judgment for systems has the form $\vdash s : (R, W)$, which says that the system s will access entities of archetypes in R and produce staged events that can change archetypes in W. Rule (T-Seq) checks the type of a sequential system, where the parameter x has the type A since the entities passed to x are in the archetype A. Mixing immediate and staged events in the same system can be problematic since immediate events may change the heap structure that the staged events depend on. Thus, for simplicity, we require sequential system to have no staged events while data/task parallel systems to have no immediate events.

The type rules for frames in Figure 11 tracks the set of dirtied archetypes $\mathcal{D}$. Rule (T-System) ensures that the archetypes in R are not in $\mathcal{D}$ and combines the write set W with $\mathcal{D}$ in checking the rest of the frame. Rule (T-Sync) clears $\mathcal{D}$ for systems after a sync.

5.1 Properties

$$\Gamma \vdash e : \Gamma(e) \ \& \ \emptyset \qquad \frac{\Gamma \vdash t : \tau \ \& \ W \quad W \subseteq W'}{\Gamma \vdash t : \tau \ \& \ W'} \qquad \text{T-Sub}$$

$$\frac{\forall e_i \in es. \quad e_i \in \mathcal{H}[A] \quad e_i : A \vdash [e_i/x]t : \tau \ \& \ W}{\vdash \mathcal{H}, \lambda x.t \ @ \ es : (\{A\}, W)} \qquad \text{T-Iter}$$

$$\frac{\vdash c_1 : C_1 \ldots \vdash c_n : C_n}{\vdash \{C_1 : c1, \ldots, C_n : c_n\} : \{C_1, \ldots, C_n\}} \qquad \text{T-Entity}$$

$$\frac{\forall A.\ \forall e \in \mathcal{H}[A].\ \vdash \mathcal{H}[e] : A}{\vdash \mathcal{H}} \qquad \text{T-Heap}$$

$$\frac{\vdash \mathcal{H} \quad \mathcal{D} \supseteq \mathit{atype}(\mathcal{H}, q) \quad \vdash fr : \mathcal{D}}{\vdash \mathcal{H}, fr, q} \qquad \text{T-Frame}$$

Figure 12: Type rules for runtime values.

In this section, we state the progress and type preservation lemmas to show that a well-typed ECS program will not get stuck. The proof, omitted here, uses the typing rules of runtime values in Figure 12. Rule (T-Frame) says that a game state $\mathcal{H}, fr, q$ is well-typed if $\mathcal{H}$ is well-typed and the frame fr is well-typed with respect to the archetypes dirtied by q. Rules (T-Heap) and (T-Entity) check that every entity of every archetype is well-typed.

Lemma 1 (Progress). *If $\vdash \mathcal{H}, fr, q$, then there exists $\mathcal{H}', fr' \neq fr$, and q' such that $\mathcal{H}, fr, q \to \mathcal{H}', fr', q'$.*

The interesting part of the proof is that if $fr = s :: fr'$, then for s to reduce, it cannot access the archetypes dirtied in q. This can be shown from the typing rules for frame and system.

Lemma 2 (Preservation). *If $\vdash \mathcal{H}, fr, q$ and $\mathcal{H}, fr, q \to \mathcal{H}', fr', q'$, then $\vdash \mathcal{H}', fr', q'$.*

The interesting part of the proof is that if $fr = s :: fr'$ and $\vdash s : (R, W)$, then W includes the archetypes dirtied by staged events in s. Together with the typing rules for frame and system, we can show that the new game state is well typed.

Theorem 1 (Soundness). *If $\vdash \mathcal{H}, fr, q$, then there exists $\mathcal{H}'$ such that $\mathcal{H}, fr, q \to^* \mathcal{H}', [], \mathit{nil}$.*

The progress and preservation lemmas ensure that a well-typed game state will evaluate its frame until it is empty. Since a frame always ends with a *sync*, the event queue will be cleared in the end.

6 Experimental Implementation

We implemented a *Tower Defense* simulation to compare four designs: OOP, Array-of-Structs (AoS) ECS, and single/multi-threaded archetype Struct-of-Arrays (SoA) ECS. Archetype SoA organizes entities into archetypes (sets of component types) and stores their data in column-major layout, enabling efficient bulk operations. All structural mutations, including component addition/removal and entity creation/destruction, are deferred and queued during system execution, and synchronized before executing the next system.

6.1 Tower Defense Architecture

Tower Defense models waves of enemies moving along a path while stationary towers automatically detect and attack them. The simulation advances in a deterministic main loop that runs at fixed time steps. Each frame executes a predefined sequence of systems and synchronization points. Systems are pure functions that operate over component data matched by a query. All game state resides in components, which are plain data classes without embedded behavior accessed via stable entity IDs.

6.2 Archetype Storage

In archetype SoA, each archetype is defined as a unique set of component classes, mapped to an `ArchetypeTable`, which is a columnar store that maintains all entities sharing that exact component sets. A global index maps each entity to its current table and row, enabling $O(1)$ component lookup. Structural mutations to archetypes may be deferred as queued events, which can be synchronized before the next system executes. During synchronization, affected entities are migrated between archetype tables using a swap-and-pop algorithm to maintain dense storage. Once all structural updates have been applied, the archetype-query cache is invalidated so that subsequent systems operate on up-to-date data.

6.3 Core Systems

The game is driven by a sequence of systems, each operating over component arrays selected by its query signature. A system declares the components it reads or writes, and at runtime the ECS engine identifies the matching archetype tables and executes the system's update function directly over the relevant columns.

Tower Defense includes position updates (`MovementSystem`), combat logic (`BulletSystem`, `CollisionSystem`, `ShootingSystem`), enemy spawning (`EnemySpawning`), enemy health (`HealthSystem`), collision effects (`ParticleSystem`), a path following algorithm (`PathFollowingSystem`) and visualization (`RenderSystem`). Each system is implemented as a function that takes the current world state and delta time, and mutates only the components it explicitly queries. For example, `MovementSystem` iterates over all entities with `Position` and `Velocity`, updating coordinates by applying the velocity scaled by delta time.

The execution order of systems is defined statically, which sequences system steps and synchronization points. This design enables fine-grained control over update dependencies and makes system behavior reproducible. Dynamic scheduling of systems is possible for more complex behavior though each new schedule must be verified at runtime for safety.

6.4 Query Interface

In archetype SoA, the systems access the archetype tables through a query mechanism. A system's query specifies the components it requires and at runtime the engine resolves this signature to the archetype tables that contain exactly those components.

To ensure performance and cache locality, the query mechanism looks up entities using archetype indices via hash maps and the system receives a map from component type to data, enabling in-place mutation. The query interface provides critical functionality to systems, enabling them to retrieve the number of matching entities, iterate efficiently over entity IDs alongside their component data. Query results are stable within a single frame execution, the interface ensures that iteration sets remain valid throughout the execution of each system.

6.5 Parallelism

SoA-PAR is our parallel version of the archetype SoA design. It includes data and task parallel execution over archetype-structured storage.

Task Parallelism. SoA-PAR employs task parallelism at the system level to exploit coarse-grained concurrency across independent subsystems. Each system encapsulates a distinct unit of game logic operating over a subset of components. These systems declare their memory access patterns through an access descriptor that specifies which component types are read, written, or structurally modified during execution.

At runtime, `FrameExecutorParallel` class analyzes each system's access sets and organizes them into waves of conflict-free execution. Systems whose read-write do not overlap are grouped into the same wave, and executed concurrently using the shared worker pool managed by the Scheduler. Systems that perform structural updates are placed in their own serialized waves to ensure a consistent view of archetype state.

The scheduler executes waves sequentially. After running all systems in a wave in parallel, it immediately invokes `sync()` to apply the staged events. Each wave boundary serves as a synchronization point for both execution and structural consistency, ensuring that subsequent waves observe a fully updated world state. This design provides deterministic ordering, eliminates residual conflicts across waves, and achieves scalable parallelism for system-level tasks.

Data Parallelism. Within each system, data parallelism is implemented with a lightweight construct, `ParFor`, which partitions an entity range into contiguous chunks distributed across worker threads. Each thread executes the same update function on its assigned subset, achieving SIMD-like parallelism.

For instance, the `MovementSystem` updates entity positions by applying a uniform computation over all entities with position and velocity. Using `ParFor.parFor`, this loop is partitioned into chunks that execute concurrently on all available processor cores. The chunk size is dynamically chosen based on the number of hardware threads and a configurable granularity parameter, balancing load distribution and scheduling overhead. This model yields near-linear speedup for data-oriented systems dominated by arithmetic or memory-bound operations.

In summary, SoA-PAR supports two-level parallelism:

(1) Task-level parallelism between systems whose data dependencies permit concurrent execution.
(2) Data-level parallelism within each system across independent entity records.

By nesting these two forms of concurrency, the runtime maximizes CPU utilization on multi-core architectures while preserving deterministic execution.

7 Performance Comparison

We compared OOP, AoS, archetype SoA, archetype SoA-PAR by running *Tower Defense* implemented in each design, where only SoA-PAR uses multiple threads. Although it is well established in the literature and in industrial practice that ECS outperforms OOP, our experiments quantify the performance gain obtained specifically from an implementation based on our ECS semantics. We use OOP as the baseline because OOP remains the common architecture in game development and interactive simulations, especially in commercial engines and educational materials. By comparing against OOP, we show how much of the performance improvement comes directly from our formal model.

We ran each design under the same game-play conditions (max entities 20000, max enemies 15000, enemy spawn interval 0.05s, turret firing interval 0.02s), targeting a fixed frame rate of 60 FPS. Performance data were collected through an integrated profiler. Experiments were conducted on a system equipped with Intel Core i7-12650H CPU, 32 GB DDR5 RAM, and NVIDIA GeForce RTX 4060 GPU. All ECS computation and system scheduling run exclusively on the CPU and the GPU is used only for rendering. To ensure stability of the performance, each configuration was executed multiple times. Across these runs we observed minimal variance, and all runs produced consistent timing behavior. Because the differences were negligible, we report a representative run for each configuration.

We evaluated two foreground loop policies in `LibGDX/LWJGL`: an uncapped run with `foregroundFPS=0`, which performs no throttling and renders as fast as the hardware allows, and a capped run with `foregroundFPS=60`, where the engine uses cooperative sleep/yield to target 60 FPS without syncing to the monitor. Each configuration was executed several times and the variance of the results were minimal. A full scalability study with varying entity count and different map size is deferred to future work.

Frame Rate. Table 1 represents the average FPS achieved among four designs under the two rendering settings. OOP exhibits the lowest performance in both settings and shows marked frame-time instability. The AoS design nearly doubles OOP throughput, while the archetype SoA yields an additional improvement. The parallel variant (SoA-PAR) performs the best overall, sustaining over 100 FPS with an unconstrained foreground rate and maintaining close to the 60 FPS target under capped conditions.

The cumulative FPS distributions in Figure 13 illustrates the stability differences among architectures. Under the uncapped setting (*foregroundFPS=0*), shown in Figure 13a, SoA-PAR consistently dominates the entire CDF curve, reflecting both higher throughput and smoother frame pacing. The standard SoA configuration ranks second, outperforming AoS and OOP due to improved cache locality and reduced memory misses. In the capped setting (*foregroundFPS=60*), shown in Figure 13b, the SoA-PAR curve rises steeply near the 60 FPS mark, indicating minimal variance and stable frame timing close to the target rate. By contrast, OOP exhibits a much flatter distribution, revealing substantial frame-to-frame fluctuations and degraded temporal stability.

Table 1: Average FPS across architectures & foreground FPS.

Architecture	Avg FPS {*foreground FPS = 0*}	Avg FPS {*foreground FPS = 60*}
OOP	36.13	25.49
AOS	61.51	47.71
Archetype SoA	65.81	48.76
Archetype SoA-PAR	109.54	58.54

8 Conclusion

In this paper, we presented a formal semantics and a type system for an archetype-based ECS framework. Our semantic model captures the essence of archetype ECS computation in that entities can be read and updated in parallel by stateless systems while mutations to archetype tables are delayed until they can be safely processed. We also evaluated the performance of four architectural designs, including OOP, AoS, archetype SoA, and its parallel extension (SoA-PAR) using a *Tower Defense* simulation. The results align with prior findings: the SoA architecture consistently outperforms OOP and AoS due to its contiguous memory layout and improved cache locality. The parallel SoA design further amplifies these benefits by exploiting multi-core hardware, achieving both higher throughput and stable frame pacing under load.

As future work, we will conduct a scalability study with varying entity counts, game maps, and system complexity. We will also extend our formalism to model system query to retrieve entities using filters. While queries are flexible, they introduce challenges to static checking of task-parallel systems since the sets of queried archetypes are dynamic. Another interesting feature is entity relationships [9]. For example, `Turret` entities can relate to `Bullet` entities via a `Fires` relationship, which enables more capable queries than component-based ones. However, cleaning up relationships after entity destruction is a challenge. Other design choices include automatic execution of staged events based on the archetypes of the next system and dynamic scheduling of parallel systems based on their dependencies and the availability of threads [13].

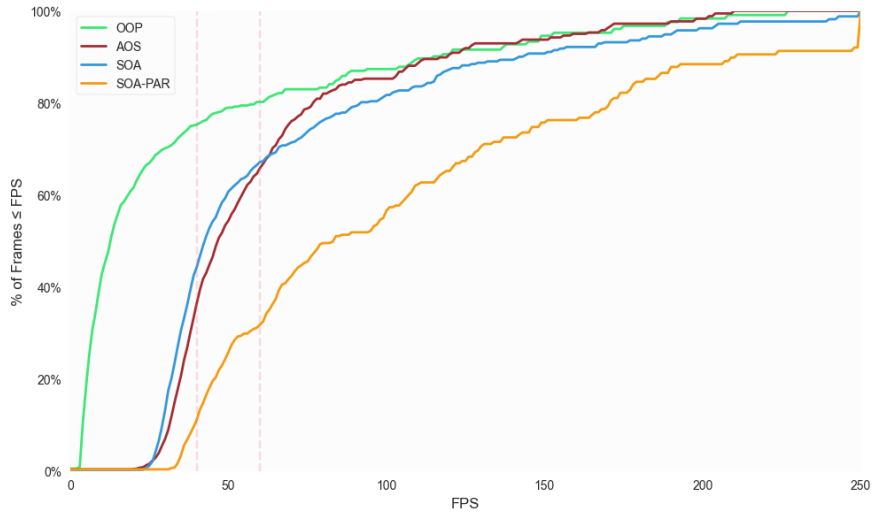
The source code is available at https://github.com/uwm-se/ecs.

Acknowledgments

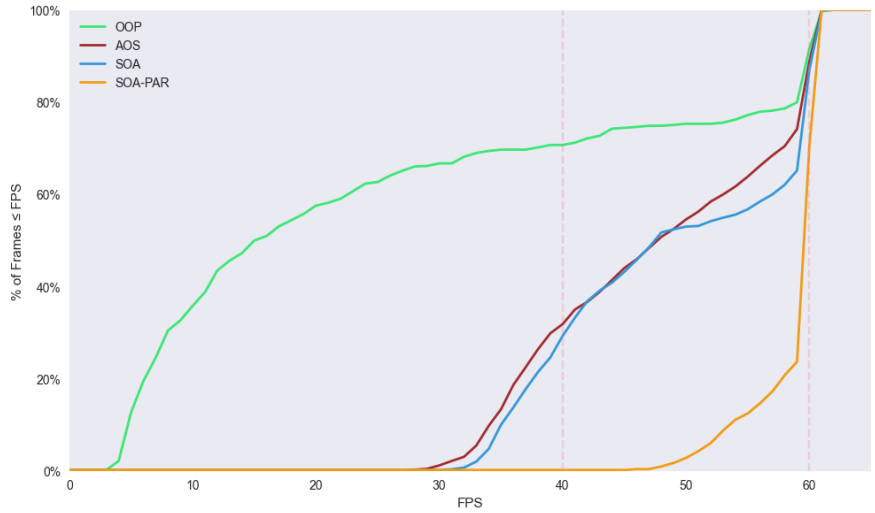
This work is partially supported by the Northwestern Mutual Data Science Institute (NMDSI) under grant number SS136.

References

[1] Bevy. 2025. bevy_ecs::archetype - Rust. https://docs.rs/bevy_ecs/latest/bevy_ecs/archetype/index.html [Online; accessed 2025-10-09].
[2] Bailey V. Compton. 2022. *An Investigation of Data Storage in Entity-Component Systems.* Master's thesis. Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio. https://scholar.afit.edu/etd/5353 AFIT-ENG-MS-22-M-018; DTIC Accession No. AD1166837.
[3] John Conway. 1970. Conway's Game of Life: Scientific American, October 1970. https://www.ibiblio.org/lifepatterns/october1970.html [Online; accessed 2025-10-08].
[4] Louis Cox, Benjamin Williams, James Vickers, Davin Ward, and Christopher Headleand. 2025. Run-time Performance Comparison of Sparse-set and Archetype



(a) Cumulative FPS distribution (*foregroundFPS=0*).



(b) Cumulative FPS distribution (*foregroundFPS=60*).

Figure 13: Comparison of cumulative FPS distributions for OOP, AoS, SoA, SoA-PAR with or without capping FPS at 60.

Entity-Component Systems. In *Computer Graphics and Visual Computing (CGVC)*, Yun Sheng and Aidan Slingsby (Eds.). The Eurographics Association. doi:10.2312/cgvc.20251224

[5] Kirill Fedoseev, Nursultan Askarbekuly, Ekaterina Uzbekova, and Manuel Mazzara. 2020. A Case Study on Object-Oriented and Data-Oriented Design Paradigms in Game Development. doi:10.13140/RG.2.2.16657.66405

[6] Daniel Masamune Hall. 2014. *ECS Game Engine Design.* Bachelor's thesis. California Polytechnic State University - San Luis Obispo. https://digitalcommons.calpoly.edu/cpesp/135/

[7] Toni Harkonen. 2019. Advantages and Implementation of Entity-Component-Systems - Trepo. https://trepo.tuni.fi/handle/123456789/27593

[8] Lars I. Hatledal, Yingguang Chu, Arne Styve, and Houxiang Zhang. 2021. Vico: An Entity-Component-System Based Co-simulation Framework. *Simulation Modelling Practice and Theory* 108 (2021), 102243. doi:10.1016/j.simpat.2020.102243

[9] Sander Mertens. 2020. Building an ECS #2: Archetypes and Vectorization | by Sander Mertens | Medium. https://ajmmertens.medium.com/building-an-ecs-2-archetypes-and-vectorization-fe21690805f9 [Online; accessed 2025-10-09].

[10] Sander Mertens. 2025. SanderMertens/flecs: A Fast Entity Component System (ECS) for C & C++. https://github.com/SanderMertens/flecs [Online; accessed 2025-10-09].

[11] Sander Mertens. 2025. SanderMertens/tower_defense: Tower defense game written in Flecs. https://github.com/SanderMertens/tower_defense [Online; accessed 2025-10-09].

[12] Patrick Redmond, Jonathan Castello, José Trilla, and Lindsey Kuper. 2025. Exploring the Theory and Practice of Concurrency in the Entity-Component-System Pattern. doi:10.48550/arXiv.2508.15264

[13] Vittorio Romeo. 2016. *Analysis of Entity Encoding Techniques, Design and Implementation of a Multithreaded Compile-time Entity-Component-System C++14 Library.* Ph. D. Dissertation. Università degli Studi di Messina. doi:10.13140/RG.2.1.1307.4165

[14] Brennan Shacklett, Zhiqiang Xie, Bidipta Sarkar, Andrew Szot, Erik Wijmans, Vladlen Koltun, Dhruv Batra, and Kayvon Fatahalian. 2023. An Extensible, Data-Oriented Architecture for High-Performance, Many-World Simulation. *ACM Transactions on Graphics* 42 (07 2023), 1–13. doi:10.1145/3592427

[15] Šimon Tichý. 2023. The Last Clan - RTS game in Unity. https://dspace.cuni.cz/handle/20.500.11956/188235

[16] Wikimedia. 2007. Tower defense - Wikipedia. https://en.wikipedia.org/wiki/Tower_defense [Online; accessed 2025-10-09].