The Semantics of Entity Component System

Anisha Tasnim
Department of Computer Science
University of Wisconsin – Milwaukee
Milwaukee, Wisconsin, USA
tasnim@uwm.edu

Tian Zhao*
Department of Computer Science
University of Wisconsin – Milwaukee
Milwaukee, Wisconsin, USA
tzhao@uwm.edu

ABSTRACT

Entity Component System (ECS) architecture has become a widely adopted foundation for large-scale simulation and game-engine design because it separates identity, state, and behavior in a form that supports data-oriented execution. Among its variants, archetype-based ECS is especially effective because entities with identical component sets are organized into dense columnar tables, enabling regular iteration and efficient memory access. Despite its practical success, the semantics of archetype-based ECS remain only partially formalized, and it is still unclear to what extent its execution discipline can be preserved beyond CPU-oriented runtimes.

This paper studies archetype-based ECS from both semantic and architectural perspectives. First, it formalizes the core mechanisms of archetype ECS, including entity creation, component association, system execution, deferred structural mutation, and archetype migration, and presents a type system that prevents unsafe access to archetype storage during execution. Second, it investigates how this model can be extended to GPU-resident execution. To this end, we implement a reference CPU realization and a GPU-resident realization using a Tower Defense simulation. The GPU design preserves stable table iteration, query-driven execution, and deferred structural mutation, while adapting storage and synchronization to device-resident execution through GPU SoA buffers, device-side cardinality tracking, and explicit reconciliation passes. Experimental results show that the GPU-resident realization achieves substantially higher steady-state throughput than the CPU baseline, with approximately 6.4-6.81× speedup across two stress scenarios and low run-to-run variability. These results indicate that the essential execution discipline of archetype-based ECS can be preserved in a constrained GPU-resident setting and can provide a practical foundation for high-throughput simulation.

CCS CONCEPTS

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

KEYWORDS

Entity Component System, Semantics, Type System, Simulation, GPU

*Corresponding author.

1 INTRODUCTION

Modern game engines and simulation systems must manage large populations of active entities while sustaining smooth real-time performance. In a *Tower Defense* simulation, for example, towers continuously track moving enemies, bullets traverse the map, and particle effects update every frame. To maintain 60 frames per second (FPS), the system must advance the position, health, and state of every active entity within a few milliseconds. Under these conditions, performance depends critically on how entity state is represented and accessed in memory. This same performance challenge extends beyond games to emerging real-time autonomous systems, including robot localization [23], multi-robot patrolling [25], and drone-car collaborative mobility [20], which have created growing demand for efficient simulation infrastructures that can manage large populations of dynamic entities while maintaining predictable performance.

Historically, most game engines and simulation tools have been built using Object-Oriented Programming (OOP), where objects such as `Tower`, `Enemy`, and `Bullet` encapsulate both data and behavior within hierarchical class structures. Although intuitive, this design becomes inefficient at scale. Object instances are typically scattered throughout memory, so iterating over large numbers of entities leads to poor cache utilization and frequent pointer chasing. For example, updating enemy positions requires dereferencing each object individually, producing irregular memory access, cache misses, and reduced throughput [4, 6]. As game complexity and hardware parallelism continue to increase, these access patterns become a major performance bottleneck.

Entity Component System (ECS) architecture addresses these limitations by shifting toward Data-Oriented Design (DOD). In ECS, entities are lightweight identifiers, components are plain data structures such as `Position`, `Velocity`, and `Health`, and systems implement behavior over sets of components. In a *Tower Defense* simulation, for instance, a movement system updates all entities that carry both `Position` and `Velocity`, while a collision system processes entities with `Health`. By separating data from behavior, ECS enables systems to operate over homogeneous data in bulk and exposes opportunities for efficient parallel execution [7, 17].

The efficiency of ECS, however, depends heavily on data layout. Early ECS implementations often used an Array-of-Structs (AoS) organization, where all components of an entity are stored together. While simpler than object hierarchies, AoS still performs poorly when a system needs to access only a subset of an entity's components. Struct-of-Arrays (SoA) improves this organization by storing each component type in its own contiguous array, resulting in more regular access patterns and better support for vectorized operations [1, 2].

Modern archetype-based ECS extends the SoA principle by grouping entities with identical component sets into archetypes, represented as dense columnar tables in which each column stores one component type and each row corresponds to an entity. This organization minimizes pointer chasing, supports constant-time component access, and uses CPU caches more effectively. Frameworks such as Unity DOTS, Bevy, and Flecs adopt this design to improve performance in large-scale simulations [1, 11]. More broadly, ECS has been used in simulation [8, 18], game development [1, 5, 6, 11, 16, 22], and interactive programming environments [9, 13, 14]. Prior work has also compared key implementation strategies, such as sparse sets and archetype tables, showing that SoA layouts and table-centric iteration improve cache efficiency and expose data parallelism [5, 15, 24].

Despite these advances, the semantics of archetype-based ECS remain largely informal and implementation dependent. This makes it difficult to reason about determinism, scheduling, and structural mutation. Recent work formalized the semantics of archetype ECS and established CPU baselines under a disciplined mutation regime, in which systems iterate over stable tables while structural changes are deferred and applied only at explicit synchronization barriers [21]. That work, however, does not address GPU execution.

Extending archetype-based ECS to the GPU is not straightforward. GPUs offer massive data-parallel throughput, but ECS workloads introduce challenges that do not arise in conventional bulk-synchronous kernels. Systems are often lightweight, query cardinalities vary dynamically with archetype occupancy, and structural mutations such as entity creation, destruction, and archetype migration require coordinated updates to storage and indexing metadata. As a result, a naive translation of a CPU-oriented ECS runtime to the GPU can lose much of its expected benefit through kernel-launch overhead, host-device synchronization, and unstable iteration domains. Recent GPU-resident ECS engines suggest that device-native storage and scheduling can mitigate some of these costs [19], but the effectiveness of archetype-based ECS as a GPU substrate under practical simulation conditions remains insufficiently characterized.

In this work, we formalize the operational semantic of archetype-based ECS in CPU, an experimental implementation and evaluation of archetype-based layout, and extend the CPU design to investigate whether archetype-based ECS can serve as an effective foundation for GPU-accelerated simulation, and under what conditions it yields end-to-end performance gains. We study a *Tower Defense* simulation [26] to analyze the performance.

In this research, we make the following contributions:

(1) We formalize the core mechanisms of archetype ECS, including entity creation, component association, system execution, and archetype migration, as compositional and deterministic state transitions.
(2) We define a type system for archetype ECS that prevents unsafe access to archetype storage during execution.
(3) We implement an archetype SoA framework in Scala and compare it with object-oriented and AoS implementations using a *Tower Defense* simulation [26], showing that archetype SoA achieves higher frame stability.
(4) We present a GPU-resident backend in *wgpu* that stores archetype tables as GPU SoA buffers, maintains device-side cardinality counters, and executes per-frame update through WGSL compute kernels with instanced rendering.
(5) We provide an end-to-end GPU evaluation of performance, scalability, and generality against a CPU baseline in Python.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces the ECS background, and Section 4 presents a motivating example. Section 5 defines the operational semantics of ECS programs, and Section 6 presents the type system used to prevent unsafe access to archetype storage. Section 7 describes the CPU implementations of the *Tower Defense* simulation and its performance evaluation. Section 8 discusses the challenges of extending ECS from CPU execution to GPU-resident execution and the performance of *Tower Defense* on GPU.

2 RELATED WORK

Early game engines relied heavily on object-oriented design, where data and behavior were organized through inheritance hierarchies. Although this style encouraged modular encapsulation, it scaled poorly as game logic became more complex. Deep class hierarchies and runtime polymorphism led to irregular memory access and poor cache utilization, which in turn limited performance in highly parallel workloads [4, 6]. These limitations motivated a shift toward data-oriented design, where computation is organized around structured memory layouts rather than object abstractions.

ECS emerged as a response to these scalability challenges. By decomposing game logic into entities, components, and systems, ECS favors predictable data access patterns and cache locality over inheritance-based flexibility [7, 17]. This separation of identity, state, and behavior enables runtime reconfiguration and simplifies parallel execution. However, the efficiency of ECS depends strongly on how component data are stored. Early Array-of-Structs (AoS) designs retained per-entity grouping, which simplified access but still fragmented system-level iteration. Struct-of-Arrays (SoA) layouts improved on this by storing each component type in a contiguous array, thereby improving spatial locality and enabling vectorized traversal across large datasets [2]. Modern archetype-based ECS extends this idea further by grouping entities with identical component sets into dense columnar tables. This organization minimizes pointer chasing, supports constant-time component lookup, and improves update throughput and cache coherence relative to alternative designs such as sparse sets [2, 3].

These principles are reflected in several widely used frameworks. Unity DOTS employs chunk-based archetype storage together with Burst-compiled jobs to reduce cache inefficiency and main-thread contention [22]. Bevy combines archetype tables with Rust's ownership model to provide memory-safe and cache-efficient traversal [1]. Flecs likewise organizes entities into dense tables and defers world modifications through command queues, enabling thread-safe and deterministic parallel execution [11, 12]. Beyond game engines, ECS has also been adopted in broader simulation contexts. Vico applies ECS to distributed co-simulation [8], while Madrona demonstrates a GPU-accelerated ECS runtime for large batched reinforcement-learning environments [18]. Other empirical studies have shown that archetype-based designs can substantially improve throughput and frame stability in simulation-heavy workloads [3, 4].

Despite this progress, the formal semantics of archetype-based ECS remain underdeveloped. Most existing frameworks describe execution operationally rather than through a rigorous semantic model, making it difficult to reason precisely about determinism, component migration, and synchronization invariants. This gap has motivated recent efforts to formalize ECS execution. The closest related work is Core ECS [15], which focuses on concurrency and deterministic scheduling, showing how safe schedules can be constructed and how schedule safety implies determinism. Our work differs in emphasis by modeling archetype-based ECS directly, including its memory layout, archetype transitions, and structural effects. In particular, we provide an operational semantics together with a type system that exposes read-write sets for systems and supports static reasoning about structural conflicts and compatibility. A recent mapping study further highlights the importance of SoA layouts and table-centric iteration in exposing cache efficiency and data parallelism [24], while recent formal work identifies stable-table iteration with deferred structural mutation as a key invariant of high-performance archetype ECS [21].

The extension of ECS to GPU execution introduces a different set of concerns. Recent GPU-resident systems show that ECS-like organizations can map well to the GPU when storage, scheduling, and mutation handling are designed natively for device execution. Isaac Gym, although not an ECS engine, demonstrates the broader value of keeping simulation and downstream learning pipelines resident on the GPU, thereby minimizing CPU bottlenecks and synchronization overhead [10]. The closest work in this space is Madrona, which represents archetypes as GPU-resident column stores and executes ECS computation graphs as GPU megakernels [18, 19]. By combining GPU-native storage, compaction, and scheduling, Madrona reduces kernel-launch overhead and avoids repeated CPU read-backs when query cardinalities change dynamically.

Our work differs from these systems in both scope and emphasis. Rather than targeting many-world batch simulation through monolithic GPU kernels, we focus on single-world simulation and on preserving the execution discipline of archetype-based ECS under explicit system-level GPU execution. In particular, we retain stable table iteration, deferred structural mutation, and explicit synchronization boundaries, and study how these principles can be realized in a GPU-resident setting. In this sense, the present work lies at the intersection of two lines of research: the formalization of archetype ECS and the design of GPU-resident simulation runtimes.

3 ECS FOUNDATION

ECS is a data-oriented design for structuring simulations and interactive systems. ECS decomposes program state and behavior into three orthogonal concepts: entities, components, and systems. Entities are opaque identifiers with no intrinsic data or behavior. They serve solely as stable references that associate component data across subsystems. Components are typed, homogeneous data records that store all mutable state. Each component type represents a single aspect of state, such as position, velocity, health, and component instances are associated with entities. Components contain no behavior and are typically stored in contiguous arrays to enable efficient iteration. Systems implement behavior as transformations over sets of components. A system declares the component types it

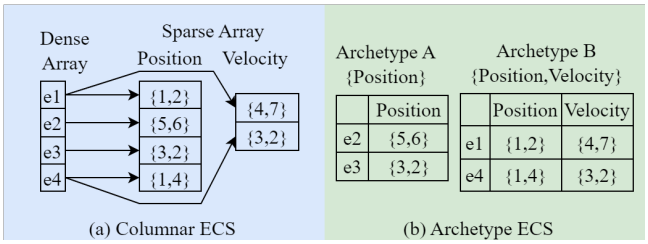


Figure 1: ECS variants based on how components are stored.

operates on and executes by iterating over all entities that possess the required components. Systems are expressed independently of entity identity or ownership, enabling flexible scheduling and parallel execution.

Execution in ECS is organized around query-driven iteration. Rather than invoking methods on objects, systems select entities by component presence and operate directly over component data. This formulation shifts the primary abstraction from control flow to data access, allowing implementations to exploit memory locality, vectorization, and data parallelism. The performance characteristics of an ECS implementation are therefore dominated by how component data is organized in memory and how component-set queries are resolved.

3.1 ECS Variants

While ECS defines a programming model, it does not prescribe a single memory layout. In practice, the dominant performance differences among ECS engines arise from how component data are organized in memory and when component-set membership is resolved for system execution. Two widely used implementation families are *columnar* and *archetype* ECS distinguished by their storage designs.

Columnar ECS. Each component type of columnar ECS is stored in its own dense array, typically accompanied by a mapping from entity identifiers to array indices. An entity is therefore represented implicitly by the set of component entries associated with its identifier across stores. Systems execute by selecting entities that possess the required components and iterating over the corresponding component arrays. When a system requires multiple components, the runtime coordinates access across stores via identifier-based joins or sparse-to-dense index indirection. Fig. 1(a) illustrates how entities are mapped to dense component arrays. This design emphasizes structural flexibility: adding or removing a component affects only the corresponding store, without relocating unrelated component data. For example, adding a new component (Velocity) to $e2$ is inexpensive: the runtime simply appends (or allocates) a new entry for $e2$ in the Velocity component store and updates the entity-to-index mapping. No other component arrays are moved or structurally modified.

Archetype ECS. Entities in archetype ECS are grouped by their exact component sets. Each distinct component combination defines an archetype, and entities sharing that combination are stored together in a dense table. Within a table, components are commonly laid out in SoA form, with one contiguous column per component

type. Systems execute by selecting the archetype tables that satisfy their required component set and iterating directly over their rows. Because all rows in a table share an identical layout, component access is regular and does not require per-entity presence checks or execution-time joins. Structural changes that alter an entity's component set are implemented as table migration, moving the entity's data from one archetype table to another. Fig. 1(b) shows how the entities are stored in archetype tables. Suppose that *Velocity* is added to $e2$. Since Archetype B $\{Position, Velocity\}$ already exists, $e2$ will be moved from Archetype A to Archetype B. If Archetype B does not exist, a new archetype table will be created to store $e2$.

4 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 5.

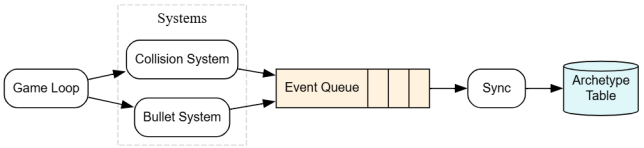


Figure 2: Illustration of an ECS game loop, where queued events are synchronized to archetype table.

In *Tower Defense*, the efficiency of the game loop depends on how entities and their data are stored in memory. Figure 2 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.

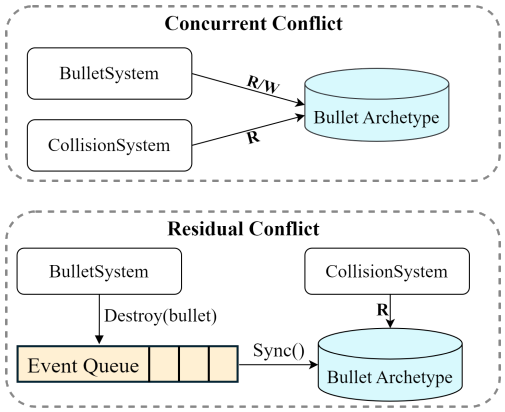


Figure 3: Illustration of conflicts during the execution of `BulletSystem` and `CollisionSystem`.

As illustrated in Figure 3, 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.

5 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.

5.1 Syntax

Our ECS model adopts the syntax in Figure 4 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 [11] 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.

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 4: The syntax of entities, components, and systems.

Term t in Figure 5 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.

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 5: The syntax for terms, values, and events.

5.2 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.

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

5.3 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

5.4 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.

5.5 Event Evaluation

Figure 6 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 $\mathcal{H}'$.

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.

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

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

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

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

Figure 6: The rules for event reduction.

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

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

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

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

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

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

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

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

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

Figure 7: The rules for term reduction.

5.6 Term Evaluation

Figure 7 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.

5.7 System Evaluation

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

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

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

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

$$\frac{\mathcal{H},\ s \rightarrow \mathcal{H}',\ q' \quad no\_conflict(\mathcal{H}, s, q)}{\mathcal{H},\ s :: fr,\ q \rightarrow \mathcal{H}',\ fr,\ q + q'} \quad \text{F-System}$$

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

Figure 8: The rules for system and frame evaluation.

$$query(A, \lambda x.t) = \{A\} \quad query(\mathsf{par}\ s) = query(s)$$
$$query([s_1, s_2]) = query(s_1) \cup query(s_2)$$

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

$$\frac{A \notin atype(\mathcal{H}, q)}{no\_conflict(\mathcal{H},\ (A, \lambda x.t),\ q)}$$

$$\frac{no\_conflict(\mathcal{H}, s, q)}{no\_conflict(\mathcal{H}, \mathsf{par}\ s, q)} \qquad \frac{\forall i \in \{1,2\}.\ no\_conflict(\mathcal{H}, s_i, q)}{no\_conflict(\mathcal{H}, [s_1, s_2], q)}$$

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

Figure 9: Auxiliary functions.

Figure 8 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 $\rightarrow^*$ means transitive closure of $\rightarrow$.

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.

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.

5.8 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 8. 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.

6 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.

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

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 of archetypes that may be written by staged events in t. The typing rules in Figure 10 and 11 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 A. Mixing

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

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

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

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

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

Figure 10: 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 11: 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 12: Frame and system typing rules.

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 12 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.

6.1 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 13: 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 uses the typing rules of runtime values in Figure 13. 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 (Heap Canonical Forms). *If $\vdash \mathcal{H}$ and $e \in \mathcal{H}[A]$, then $\vdash \mathcal{H}[e] : A$. In particular, for every $C \in A$, the component value $\mathcal{H}[e][C]$ is defined.*

Proof. A direct proof by T-Heap and T-Entity. □

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

Proof. By case analysis on the derivation of $\mathcal{H}, \mathit{evt} \to \mathcal{H}', ()$.

Case E-Cre. A fresh entity e is created with component map $C \mapsto c$ and inserted into the archetype table C. By construction, the new entity has exactly the components required by its new archetype. All existing entities remain unchanged. Hence $\vdash \mathcal{H}'$.

Case E-Des. The entity e is removed from its archetype table. No remaining entity changes type and therefore the $\mathcal{H}$ invariant is preserved.

Case E-Add. The entity e is moved from archetype A to archetype $A \cup C$, and its component map is extended with the new component C. Thus the shape of the entity remains consistent with the archetype table in which it is stored.

Case E-Rem. Symmetrically, the entity e is moved from archetype A to archetype $A - C$, and its component map is updated accordingly. Again, the entity shape matches the destination archetype.

In every case, each entity remains stored in a table that matches its component structure, so $\vdash \mathcal{H}'$. □

Lemma 3 (Iteration Preservation). *If $\vdash \mathcal{H}$, every entity in the list es belongs to archetype A, and $x : A \vdash t : \tau \& W$, then there exist $\mathcal{H}'$ and q' such that $\mathcal{H}, \lambda x.t@es \to \mathcal{H}', q'$ and $\vdash \mathcal{H}'$.*

Proof. **Base case.** If $es = []$, the iteration is empty, so the result is immediate.

Inductive case. Let $es = e :: es'$. Since $e \in \mathcal{H}[A]$, by Lemma 1, the entity e has type A. Hence the substituted body $[e/x]t$ is well-typed. The term reduction rules in Figure 7 either leave the heap unchanged, update an already existing component in place, enqueue a staged event, or evaluate an immediate event. In the last case, heap being well-formed is preserved by Lemma 2. Therefore evaluating $[e/x]t$ preserves heap typing. Applying the induction hypothesis to the remaining entities es' yields the result. □

Lemma 4 (Merge Preservation). *If $\vdash \mathcal{H}$, $\vdash \mathcal{H}_1$, and $\vdash \mathcal{H}_2$, and $\mathcal{H}' = merge(\mathcal{H}, \mathcal{H}_1, \mathcal{H}_2)$, then $\vdash \mathcal{H}'$.*

Proof. By definition of *merge*, the archetype tables of $\mathcal{H}'$ are identical to those of $\mathcal{H}$, $\mathcal{H}_1$, and $\mathcal{H}_2$, and only component values of existing entities may differ. Since entity type in archetype tables is unchanged, each entity in $\mathcal{H}'$ still has the same archetype shape as required by Rule T-Heap. Therefore $\vdash \mathcal{H}'$. □

Lemma 5 (Progress). *If $\vdash \mathcal{H}, fr, q$, then there exists $\mathcal{H}', fr' \neq fr$, and q' such that $\mathcal{H}, fr, q \to \mathcal{H}', fr', q'$.*

Proof. By inversion on Rule T-Frame, there exists a dirty set $\mathcal{D}$ such that $\vdash \mathcal{H}$, $\mathcal{D} \supseteq atype(\mathcal{H}, q)$, and $\vdash fr : \mathcal{D}$.
Case analysis on fr:

- **Case** $fr = sync :: fr$ By Rule F-Sync, the queued events in q are executed in order, producing some heap $\mathcal{H}'$ and clearing the queue: $\mathcal{H},\ sync :: fr,\ q \to \mathcal{H}_n,\ fr,\ nil$.
- **Case** $fr = s :: fr$ By inversion on Rule T-System, there exist sets R and W such that $\vdash s : (R, W)$, $R \cap \mathcal{D} = \emptyset$, and $\vdash fr : W \cup \mathcal{D}$. Since $\mathcal{D} \supseteq atype(\mathcal{H}, q)$, no archetype accessed by s is dirtied by the pending queue. Therefore $no\_conflict(\mathcal{H}, s, q)$ holds in F-System.
  Case analysis on s:
  - **case** $s = (A, \lambda x.t)$ By inversion on the system typing judgment, we have $x : A \vdash t : \tau \& W$ for some τ. Since $\vdash \mathcal{H}$, every entity in $\mathcal{H}[A]$ is well-typed at archetype A by Lemma 1. By Lemma 3, the iteration over $\mathcal{H}[A]$ reduces: $\mathcal{H}, \lambda x.t@\mathcal{H}[A] \to \mathcal{H}', q'$, which is proved by Rule S-Seq.
  - **Case** $s = par(A, \lambda x.t)$ By inversion on Rule T-Data-P, the underlying body is well-typed and contains no immediate events. Partition $\mathcal{H}[A]$ into disjoint subsets es_1 and es_2. By Lemma 3, each partition reduces: $\mathcal{H}, \lambda x.t@es_1 \to \mathcal{H}_1, q_1$ and $\mathcal{H}, \lambda x.t@es_2 \to \mathcal{H}_2, q_2$. By Rule S-Data-P, we have $\mathcal{H}, par(A, \lambda x.t) \to merge(\mathcal{H}, \mathcal{H}_1, \mathcal{H}_2), q_1 + q_2$
  - **Case** $s = [s_1, s_2]$ By inversion on Rule T-Task-P, both systems are well-typed, neither contains immediate events, and their accessed archetypes are disjoint. Hence each system reduces: $\mathcal{H}, s_1 \to \mathcal{H}_1, q_1$ and $\mathcal{H}, s_2 \to \mathcal{H}_2, q_2$. By Rule S-Task-P, $\mathcal{H}, [s_1, s_2] \to merge(\mathcal{H}, \mathcal{H}_1, \mathcal{H}_2), q_1 + q_2$.

  In every cases, there exist $\mathcal{H}_s$ and q_s such that $\mathcal{H}, s \to \mathcal{H}_s, q_s$. Since $no\_conflict(\mathcal{H}, s, q)$ holds, by Rule F-System, $\mathcal{H},\ s :: fr,\ q \to \mathcal{H}_s,\ fr,\ q + q_s$.

In all cases, there exist $\mathcal{H}', fr', q'$ such that $\mathcal{H}, fr, q \to \mathcal{H}', fr', q'$. □

Lemma 6 (Preservation). *If $\vdash \mathcal{H}, fr, q$ and $\mathcal{H}, fr, q \to \mathcal{H}', fr', q'$, then $\vdash \mathcal{H}', fr', q'$.*

Proof. We prove by induction on $\mathcal{H}, fr, q \to \mathcal{H}', fr', q'$.

- **Case** $fr = sync :: fr$ In this case, $fr' = fr$, and $q' = nil$. By inversion on $\vdash \mathcal{H}, sync :: fr, q$, there exists $\mathcal{D}$ such that $\vdash \mathcal{H}$, $\mathcal{D} \subseteq atype(\mathcal{H}, q)$, $\vdash sync :: fr : \mathcal{D}$.
  By inversion on Rule T-Sync, $\vdash fr : \emptyset$. The queue is executed event-by-event. By Lemma 2, well-formed heap is preserved throughout the synchronization step, so $\vdash \mathcal{H}'$. Since $q' = nil$, we have $\emptyset \subseteq atype(\mathcal{H}', q')$. Therefore by Rule T-Frame, $\vdash \mathcal{H}', fr, nil$.
- **Case** $fr = s :: fr$ In this case, $\mathcal{H}, s \to \mathcal{H}, q_s$, $q' = q + q_s$, and $no\_conflict(\mathcal{H}, s, q)$. By inversion on $\vdash \mathcal{H}, s :: fr, q$ and Rule T-System, there exists $\mathcal{D}, R, W$ such that $\vdash \mathcal{H}$, $\mathcal{D} \supseteq atype(\mathcal{H}, q)$, $\vdash s : (R, W)$, $R \cap W = \emptyset$, and $fr = W \cup \mathcal{D}$.
  We show that $\vdash \mathcal{H}', fr, q + q_s$.
  To prove $\vdash \mathcal{H}'$, case analysis on $\mathcal{H}, s \to \mathcal{H}, q_s$: if s reduces by Rule S-Seq, heap preservation follows from Lemma 3.
  If s reduces by Rule S-Data-P, each branch preserves heap typing by Lemma 2 and the merged heap is well-typed by Lemma 4.
  If s reduces by Rule S-Task-P, each branch preserves heap typing by Lemma 2 and the merged heap is well-typed by Lemma 4.
  Hence $\vdash \mathcal{H}'$ in all cases.
  The dirty set for the continuation frame is preserved. The old queue contribution remains covered by $\mathcal{D}$, since $\mathcal{D} \supseteq atype(\mathcal{H}, q)$. The newly produced staged events are covered by W, because W is precisely the effect set collected by the typing rules for the system body and its staged events. Therefore $atype(\mathcal{H}', q + q_s) \subseteq W \cup \mathcal{D}$
  Since $\vdash fr : W \cup \mathcal{D}$, by Rule T-Frame, $\vdash \mathcal{H}', fr, q + q_s$.

□

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

The progress and preservation lemmas ensure that a well-typed 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.

7 CPU IMPLEMENTATION OF ECS

This section presents the implementation of the archetype-based ECS used in this study. 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.

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.

7.1 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.

7.2 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.

7.3 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 inplace 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.

7.4 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 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.

7.5 Performance Evaluation

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. 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.

Table 1: Average FPS across architectures & foreground FPS.

Architecture	Avg FPS {*foreground FPS = 0*}	Avg FPS {*foreground FPS = 60*}
OOP	45.48	27.59
AOS	61.51	47.71
Archetype SoA	65.81	48.76
Archetype SoA-PAR	109.54	58.54

The cumulative FPS distributions in Figure 14 illustrates the stability differences among architectures. Under the uncapped setting (*foregroundFPS=0*), shown in Figure 14a, 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 14b, 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.

8 GPU IMPLEMENTATION OF ECS

The semantics in Section 5 and implementation in Section 7 characterize archetype-based ECS as an execution model organized around stable archetype tables, query-driven system execution, deferred structural mutation, and explicit synchronization. The question addressed in this section is whether this execution discipline can be carried into a GPU-resident setting without abandoning its central invariants. Our aim is not to redefine archetype ECS for the GPU, but to determine how its core structure must be adapted so that GPU execution remains both practical and semantically meaningful.

8.1 Challenges and Solutions

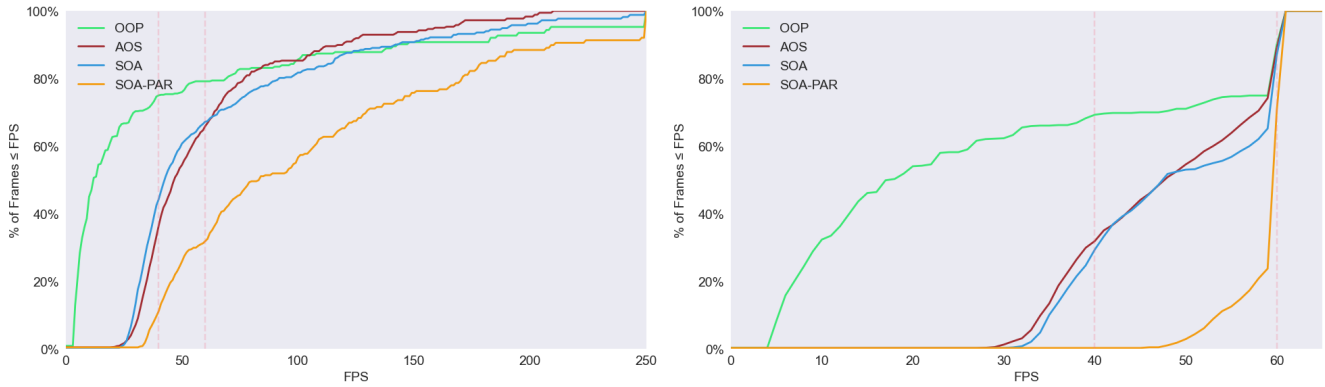
A direct translation of a CPU-oriented ECS runtime to the GPU is problematic. In a conventional CPU realization, systems execute as host-controlled loops over archetype tables and synchronization occurs between systems as deferred structural events are flushed. On the GPU, however, this organization introduces several challenges. (1) ECS workloads often consist of many lightweight systems, so launching a separate kernel for each system may incur overhead disproportionate to the work performed. (2) The iteration domain of a system depends on the dynamic cardinalities of the archetype tables matched by its query. If these cardinalities are maintained on the device, host-side dispatch sizing can require read-back or synchronization, undermining the latency-hiding advantages of GPU execution. (3) Structural mutation, including creation, deletion, and archetype migration, requires coordinated updates to storage and indexing metadata and therefore introduces additional synchronization pressure.

The guiding idea of our extension is to preserve the semantic discipline of archetype ECS while relocating its storage and steady-state execution to the GPU. Archetype tables remain the basic unit of organization, but are represented as device-resident SoA buffers. Table cardinalities are tracked using device-side counters, allowing kernels to derive effective iteration domains directly from GPU-resident state. Systems continue to execute over stable tables selected by component requirements, and structural effects are still deferred rather than applied during iteration. Instead of immediate structural mutation, system passes record logical consequences into device-resident state, and an explicit reconciliation phase restores a consistent table configuration before subsequent dependent computation proceeds. In this way, the role played by synchronization in the semantics is retained, but its realization is adapted to the GPU execution model.

Our implementation is not a fully dynamic ECS runtime on the device. It fixes the set of archetypes in advance and restricts structural mutation to a bounded set of operations that can be reconciled efficiently on the GPU. A fully general realization would require substantially more dynamic allocation, more complex migration logic, and tighter host-device coordination, which is not needed for determining whether the essential structural advantages of archetype-based ECS survive under GPU-resident execution.

8.2 Archetype-based ECS on GPU

We implement a *Tower-Defense* simulation in Python with a CPU and a GPU backend, which are designed to preserve the same ECS invariants so performance differences can be attributed fairly. Both



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

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

Figure 14: Comparison of cumulative FPS distributions for OOP, AoS, SoA, SoA-PAR in CPU with or without capping FPS at 60.

backends use dense archetype-style storage, query-driven iteration, and deferred structural changes that are only applied at explicit synchronization points. On CPU backend, these rules are implemented directly in the runtime via a *sync* operation. On GPU backend, the same constraints are encoded as *WGSL* compute kernels that operate over GPU tables and perform explicit reconciliation steps using bounded structural primitives. For visualization, *Pygame* is used for windowing and event handling in CPU backend, while the GPU backend uses *wgpu* for both compute and rendering.

We use *Tower Defense* for comparing ECS implementations because it generates large, dynamic populations of similar entities, such as enemies, towers, and projectiles updated by a stable set of per-frame systems such as path following, target acquisition, collision, damage, and rendering. These characteristics stress exactly the performance factors ECS is meant to optimize: data layout, iteration efficiency, and scheduling, while also making the CPU-GPU trade-offs visible (cache behavior and multi-threading versus data-parallel updates and synchronization overhead).

GPU mapping of archetype storage. This implementation uses a fixed tower-defense archetype set and *GameState* with map-driven *Spawner*, static and dynamic *Turret*, *Enemy*, *Bullet*, and predeclared *Particle* capacity. Each archetype is a GPU-resident SoA table: every component field is stored in a dedicated storage buffer (`f32`, `i32`, or `u32`) and each table has a device-resident `u32` size counter for live row count, which is separate from the fixed buffer capacity.

Dense entity indices of length `MAX_ENTITIES` in `entity_table`, `entity_row`, and `entity_alive` support targeted resolution of logical entity IDs to storage rows. Logical deletion clears liveness and invalidates mappings. Bullet state uses double SoA buffers (ping–pong) so compaction after movement can stay on the GPU without reading sizes back to the CPU.

Layout planning and one-time uploads. A dynamic planner derives capacities from the map and configuration (grid dimensions, per-table row caps, and `MAX_ENTITIES` headroom) and allocates all GPU buffers once. Static navigation data is computed on CPU as BFS "next step" arrays (`map_next_x`, `map_next_y`) plus goal mask and tile types, then uploaded once. Initial static entities such as turrets are uploaded once and inserted into the GPU entity index.

Initial static entities (*Turret*, *Spawner*, and *GameState*) are written once, registered in the entity index, and the global next-entity ID in `sim_state` is seeded so subsequent spawns allocate disjoint IDs.

GPU execution model and dispatch granularity. The backend keeps stable SoA tables and *delayed* structural effects. Each frame records a fixed sequence of *WGSL* compute passes followed by a render pass. The compute pipeline follows a fixed schedule:

(1) indirect-argument setup from live size counters,
(2) enemy spawning (atomic append into the enemy table and entity index), grid clear
(3) path following from the BFS next-step fields, goal detection,
(4) construction of a uniform-grid linked list of enemies, turret shooting with buffer selection matching the current bullet,
(5) bullet integration and compaction, bullet-enemy collision,
(6) a `sync` pass that applies consequences of logical structural changes, game management, and
(7) small passes that fill indirect draw/dispatch arguments for the subsequent render and next sub-step.

Kernels use `global_invocation_id.x` to index SoA elements and guard on the live size. When indirect dispatch is enabled, helper kernels convert those sizes into workgroup dispatch counts so work tracks packed population up to the predeclared per-table ceiling.

Memory transfer and synchronization control. The steady-state design targets fully GPU-resident execution (no per-frame CPU-GPU updates for simulation or rendering).

To retain per-frame metrics without per-frame stalls, the GPU writes a fixed-size FrameStat record (48 bytes) into a GPU-resident log buffer each step. At termination, the log is bulk-copied once and decoded on CPU to finalize per-frame CSV output.

The backend logs per-pass CPU-side timing around command recording and submission. Key tradeoffs are explicit: the GPU backend fixes the archetype set and relies on bounded atomic allocation and constrained structural primitives rather than fully general component-set transitions, thereby prioritizing reproducible measurement of dispatch overhead, synchronization costs, and

structural-mutation handling under a controlled workload. Dynamic archetype creation is not supported on the GPU. Only predefined archetype tables are allocated.

8.3 Performance Evaluation

This section evaluates whether the advantages of archetype ECS remain effective for GPU-resident execution. We compare the performance of *Tower Defense* in CPU and GPU backends under the identical configurations.

Hardware configuration. All experiments were conducted on a Windows 11 system equipped with an NVIDIA RTX 4060 GPU and a 12th Gen Intel Core i7-12650H CPU. The system was configured with 32 GB of DDR5 memory running at 4800 MT/s.

GPU-resident Execution. GPU execution of archetype ECS evalutes with respect to performance, scalability, and generality. We compare a CPU-only backend implemented in python against a fully GPU-resident backend in Python under matched workloads, reporting steady-state FPS, workload size signals such as alive enemies, and run-to-run variability. All results are aggregated across 5 independent runs for each backend and summarize steady-state performance using the per-run aggregated logs.

Game Configuration. We evaluate 2 stress configurations that differ only in enemy spawning intervals. In Table 2, *Stress1* uses enemy-spawning interval of 0.01s while *Stress2* increases structural pressure by halving enemy spawning interval. Both configurations use a fixed-timestep simulation targeting 60 FPS and the same map (Fig. 15). Additionally, during GPU execution, we observe that the CPU utilization is always below 5%.

Table 2: Stress Test Configuration

Parameters	Stress1	Stress2
Max Enemies	80000	80000
Enemy Spawn Interval	0.01 s	0.005 s
Enemy Health	100	100
Enemy Speed	0.7 unit/s	0.7 unit/s
Turret Range	5	5
Turret Fire Interval	0.01 s	0.01 s

To mitigate initialization effects, we discard the first 10 frames of each run and analyze only steady-state execution. Metrics are computed per run and summarized across runs using the mean and standard deviation. In addition to mean FPS, we report maximum alive enemies to verify that backends are compared under closely matched workload intensity.

FPS Comparison. GPU-resident execution achieves substantially higher steady-state FPS than CPU execution in both stress scenarios. Tab. 3 shows the per-run performance. Under *Stress1*, the CPU backend sustains 65.24 FPS on average across runs, whereas the GPU backend sustains 418.0 FPS, yielding a 6.4× throughput improvement. Under *Stress2*, the CPU backend sustains 38.17 FPS and the GPU backend sustains 260.07 FPS, corresponding to a 6.81× improvement. Overall, GPU-resident SoA execution improves

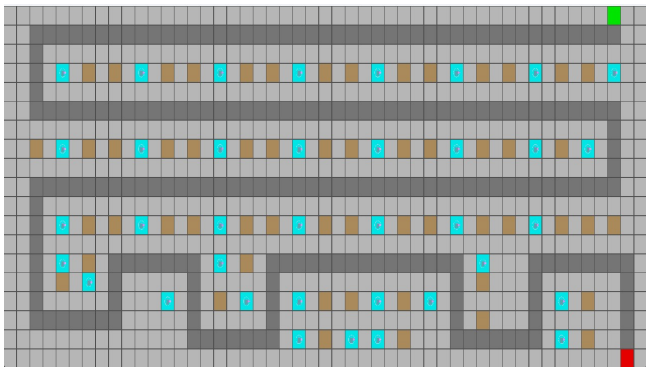


Figure 15: Tower defense game map.

throughput by approximately 6.4-6.81× over CPU execution for the evaluated stress conditions.

GPU frame-time composition (per-pass cost). We complement throughput with per-pass *GPU* times from hardware timestamp queries around each instrumented pass (`gpu_*_ms_est` in the frame log). Table 4 reports means and standard deviations across five GPU runs for *Stress1* and *Stress2* under `systemStress` with timestamps enabled. The largest measured contributors are `ShootingSystem` and `GameManagement`.

Distribution of instantaneous FPS.. To complement mean FPS, Fig. 16 reports the survival function of instantaneous per-frame FPS, which is the fraction of frames meeting a given FPS threshold. GPU execution dominates CPU execution across thresholds in both stress scenarios, indicating that the improvement is not limited to the mean but holds across the frame distribution.

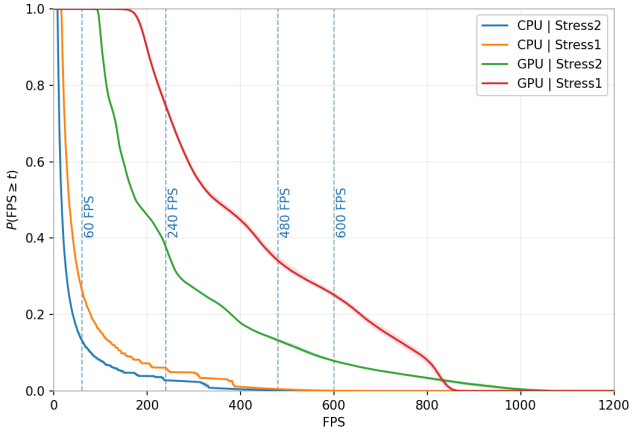


Figure 16: Survival plot: Fraction of frames meeting FPS targets under varying stress.

Scalability. Increasing stress from *Stress1* to *Stress2* approximately doubles the steady-state active population (CPU peak alive: 29853 to 59707 and GPU peak alive: 29843 to 59703), and both backends exhibit a commensurate throughput reduction. The CPU backend

Table 3: Per-run average FPS for CPU and GPU backends under two Stress scenarios (5 runs each).

CPU Backend					GPU Backend				
Scenario	**Backend**	**Run**	**Max Alive**	**Avg FPS**	**Scenario**	**Backend**	**Run**	**Max Alive**	**Avg FPS**
Stress1	CPU	1	29 853	65.33	Stress1	GPU	1	29 841	417.80
	CPU	2	29 853	64.99		GPU	2	29 842	419.10
	CPU	3	29 853	65.00		GPU	3	29 838	420.75
	CPU	4	29 853	65.13		GPU	4	29 840	417.82
	CPU	5	29 853	65.70		GPU	5	29 840	414.71
		Average	29 853	65.23			**Average**	29 843	418.0
Stress2	CPU	1	59 707	39.26	Stress2	GPU	1	59 700	262.93
	CPU	2	59 707	38.86		GPU	2	59 701	260.71
	CPU	3	59 707	39.11		GPU	3	59 703	258.68
	CPU	4	59 707	37.74		GPU	4	59 697	259.35
	CPU	5	59 707	35.88		GPU	5	59 700	258.67
		Average	59 707	38.17			**Average**	59 703	260.07

Table 4: Per-pass mean GPU time (ms, `gpu_*_ms_est`) and variability across five runs. Percentage is share of the mean sum of all passes for *Stress1* (0.01s, `systemStress` with timestamps) and *Stress2* (0.005s). The %∑ column is the share of the mean sum of all timestamped passes in that scenario.

	Stress1 (0.01s)			Stress2 (0.005s)		
Category	Mean	SD_5	%∑	Mean	SD_5	%∑
ShootingSystem	1.847	0.025	63.3	3.776	0.001	67.7
Game management	0.796	0.002	27.2	1.547	0.004	27.8
Rendering	0.067	0.001	2.3	0.128	0.000	2.3
All other gpu (sum)	0.210	—	7.2	0.123	—	2.2
Sum of all gpu passes	**2.920**	0.027	100.0	**5.574**	0.004	100.0

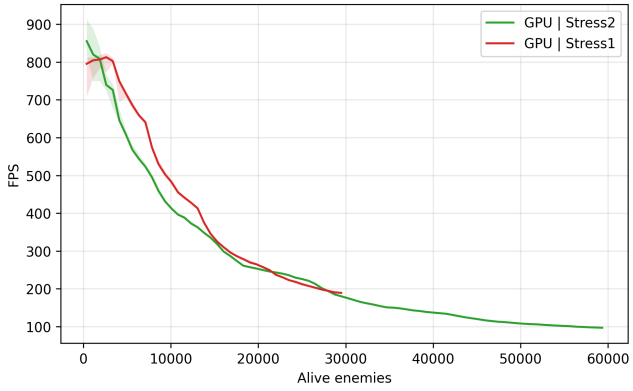


Figure 17: GPU scalability under increasing entity stress.

decreases from 65.24 FPS to 38.17 FPS (0.585× retention), while the GPU backend decreases from 418.0 FPS to 260.07 FPS (0.62× retention). Performance degrades under increased entity stress for both backends. However, the GPU backend maintains substantially higher absolute FPS throughout, and the GPU/CPU speedup remains stable as stress increases. To directly evaluate GPU-only scalability, Fig. 17 plots GPU FPS as a function of alive enemies with equal weight across the 5 runs (median with variability band). FPS decreases monotonically as the active set increases and the *Stress2* curve is shifted downward relative to *Stress1* at comparable entity counts, consistent with higher structural intensity and the larger steady-state population.

Generality. The principal trends are consistent across both stress scenarios. GPU execution exceeds CPU execution by a similar multiplicative factor (6.4× vs. 6.81×), and the direction of scaling with increased stress is the same (FPS decreases as the peak alive population increases). Run-to-run variability is low, particularly for the GPU backend, with FPS coefficients of variation below 1% (*Stress2*: 0.0069, *Stress1*: 0.0053), indicating that the observed performance trends are not driven by outlier runs. Additionally, peak alive enemies are stable within each scenario (Table 3), supporting that comparisons are made under reproducible workload conditions rather than drifting simulation behavior.

9 CONCLUSION

This research presents a formal semantics and a type system for an archetype-based ECS framework, and demonstrates how these semantics can be leveraged to enable GPU-resident execution with minimal modifications. 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 ECS in CPU under 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. Additionally, this work evaluates whether archetype ECS provides a suitable substrate for GPU-resident simulation. To prove that, we implemented a *Tower Defense* simulation with two matched backends: a CPU baseline with *Pygame* and a GPU-resident backend built using *wgpu*. Both backends preserve identical ECS invariants, including dense archetype tables, query-driven iteration, and deferred structural mutation. As a result, observed performance differences primarily reflect the execution substrate rather than algorithmic divergence. Across five independent runs and two stress scenarios, GPU-resident execution consistently outperformed CPU execution. Mean frame rate increased from 65.24 to 418.0 FPS in *Stress1* and from 38.17 to 260.07 FPS in *Stress2*, yielding a stable speedup of approximately 6.4-6.81×. Increasing stress approximately doubled the peak number of alive enemies and reduced performance in both backends. GPU throughput degraded monotonically with increasing entity count, however maintained substantially higher absolute FPS than the CPU across the entire workload range. These trends were consistent across scenarios and exhibited low run-to-run variability.

As future work, we will extend our formalism to model system query to retrieve entities using filters. Additionally, a formalism for GPU and generalizing GPU-side structural transitions and evaluating additional workloads with more diverse mutation patterns. 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 [11]. 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 [17].

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

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] Louis Cox, Benjamin Williams, James Vickers, Davin Ward, and Christopher Headleand. 2025. Run-time Performance Comparison of Sparse-set and Archetype Entity-Component Systems. In *Computer Graphics and Visual Computing (CGVC)*, Yun Sheng and Aidan Slingsby (Eds.). The Eurographics Association. https://doi.org/10.2312/cgvc.20251224

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

[5] Paul Gestwicki. 2012. The entity system architecture and its application in an undergraduate game development studio. In *Proceedings of the International Conference on the Foundations of Digital Games (FDG '12)*. Association for Computing Machinery, New York, NY, USA, 73–80. https://doi.org/10.1145/2282338.2282356

[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. https://doi.org/10.1016/j.simpat.2020.102243

[9] Patrick Lange, Rene Weller, and Gabriel Zachmann. 2016. Wait-free Hash Maps in the Entity-Component-System Pattern for Realtime Interactive Systems. In *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. 1–8. https://doi.org/10.1109/SEARIS.2016.7551583

[10] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. 2021. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning. arXiv:cs.RO/2108.10470 https://arxiv.org/abs/2108.10470

[11] 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].

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

[13] Thibault Raffaillac and Stéphane Huot. 2018. Applying the Entity-Component-System Model to Interaction Programming. In *Proceedings of the 30th Conference on l'Interaction Homme-Machine (IHM '18)*. Association for Computing Machinery, New York, NY, USA, 42–51. https://doi.org/10.1145/3286689.3286703

[14] Thibault Raffaillac and Stephane Huot. 2019. Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model. *Proc. ACM Hum.-Comput. Interact.* 3, EICS, Article 8 (June 2019), 22 pages. https://doi.org/10.1145/3331150

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

[16] Christopher Reilly and Kevin Chalmers. 2013. Game Physics Analysis and Development – a Quality-driven Approach Using the Entity Component Pattern. *The Computer Games Journal* 2, 2 (2013), 125–153. https://doi.org/10.1007/BF03392345

[17] 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. https://doi.org/10.13140/RG.2.1.1307.4165

[18] 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. https://doi.org/10.1145/3592427

[19] B. P. Shacklett, K. Fatahalian, F. Kjoelstad, and C.-Y. K. Liu. 2025. *High-throughput, Programmable Batch World Simulation on the GPU.* Dissertation. Stanford University. https://purl.stanford.edu/js274bf0548 Stanford University School of Engineering and Stanford University Computer Science Department.

[20] Sayma Tamboli, Kiran Davuluri, Khairul Mottakin, and Zheng Song. 2026. Real-Time Drone-Car Collaboration for Future Mobility: Vision and Challenge. *SIGAPP Appl. Comput. Rev.* 26, 1 (May 2026), 5–19. https://doi.org/10.1145/3807408.3807409

[21] Anisha Tasnim and Tian Zhao. 2026. The Essence of Entity Component System. In *Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing (SAC '26)*. ACM, Thessaloniki, Greece, 1–10. https://doi.org/10.1145/3748522.3779910

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

[23] Jiří Ulrich, Jan Blaha, Ahmad Alsayed, Tomáš Rouček, Farshad Arvin, and Tomáš Krajník. 2023. Real Time Fiducial Marker Localisation System with Full 6 DOF Pose Estimation. *SIGAPP Appl. Comput. Rev.* 23, 1 (April 2023), 20–35. https://doi.org/10.1145/3594264.3594266

[24] Laurent Voisard, Henrique Serra, Cristiano Politowski, Fabio Petrillo, and Yann-Gaël Guéhéneuc. 2025. A Mapping Study of the Entity Component System Pattern. In *2025 IEEE/ACM 9th International Workshop on Games and Software Engineering (GAS)*. 33–40. https://doi.org/10.1109/GAS66647.2025.00010

[25] James C. Ward, Ryan McConville, and Edmund R. Hunt. 2025. Learning Minimalist Strategies for Decentralized Multi-Robot Patrolling. *SIGAPP Appl. Comput. Rev.* 25, 2 (July 2025), 18–30. https://doi.org/10.1145/3746626.3746628

[26] wikipedia. 2007. Tower defense - Wikipedia. https://en.wikipedia.org/wiki/Tower_defense [Online; accessed 2026-02-09].