Performance and Complexity Trade-offs in Archetype-based Entity-Component-System

Laurent Voisard[1]

Supervisors:

Dr. Yann-Gaël Guéhéneuc[1], Dr. Fabio Petrillo[2] & Dr. Cristiano Politowski[3]

[1]Concordia University
[2]École de technologie supérieure (ÉTS)
[3]Ontario Tech University

June 19, 2026

Overview

1. Introduction
2. Background
3. Mapping Study
4. Empirical Case Study
5. Analytic Hierarchy Process Recommendation
6. Discussion

Introduction

- The video game industry increasingly relies on high-performance real-time systems
- Traditional object-oriented design can become limiting for large-scale simulations in terms of performance and scalability
- The Entity-Component-System (ECS) pattern has been widely adopted in industry:
  - Bevy
  - Unity DOTS
  - Unreal Mass Entity

Problem Statement

Despite ECS's growing adoption in commercial game engines and independent projects, existing academic research remains relatively limited in scope

- Existing ECS research focuses primarily on:
  - High-level ECS concepts
  - Software qualities
  - Conceptual discussions

Problem

This lack of empirical and implementation-level analysis limits our understanding of how ECS behaves in practice and hinders the development of informed design guidelines for practitioners.

ECS Overview

Entity

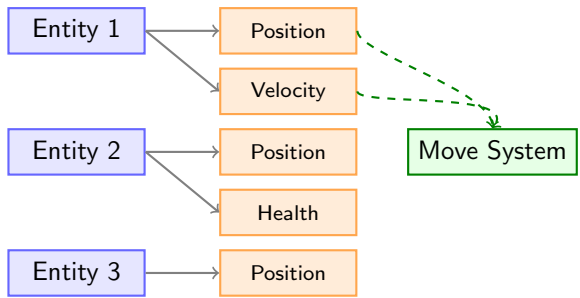
- Unique identifier
- Represents a game object

Component

- Plain data container
- No behaviour

System

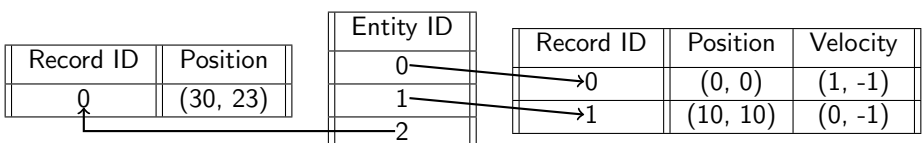
- Processes entities
- Executes logic on matching components



ECS Storage Models

Archetype-based ECS

- One archetype per unique component combination
- Cache-friendly iteration



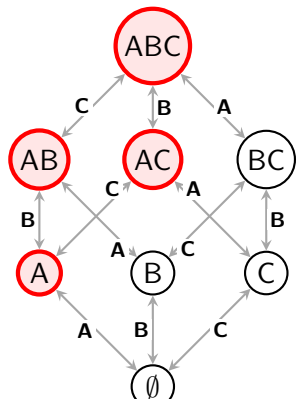
Archetype Fragmentation

Exponential Growth

- Number of potential archetypes scales at 2^N
- With just 3 components (A, B, C), up to $2^3 = 8$ archetypes
- With 10 components, up to $2^{10} = 1024$ archetypes

Performance Overhead

- Systems iterate through a high count of near-empty archetypes, adding overhead



Relationships Components

Relationship = Relation + Target

- Relationships are represented as pairs:

  `.Relation(Target)`

- Example:
  - Relation: `Owns`
  - Target: `Car`

Adding the relationship to `Bob`

```
Bob.Owns(Car)
...
Bob.Owns(Bicycle)
Bob.Owns(House)
```

Each unique pair is treated as a distinct component

Note

Each distinct relationship becomes a unique component. Can lead to high archetype fragmentation if there are many new components.

Mapping Study [1]

Before investigating ECS, we first needed to understand:

- What has already been studied?
- What research gaps remain?

Research Questions

In software engineering,

- **RQ1.1**: what are the **publication trends** on ECS?
- **RQ1.2**: what are the **research domains** of ECS?
- **RQ1.3**: what are the **benefits** and **drawbacks** of developing software using ECS?

[1] Voisard, L., Serra, H. D. F., Politowski, C., Petrillo, F., & Géhéneuc, Y.-G. (2025, April). A mapping study of the entity component system pattern. In 2025 IEEE/ACM 9th International Workshop on Games and Software Engineering (GAS) (pp. 33-40). IEEE.

Benefits vs. Drawbacks in ECS Research

Out of 35 primary studies

Reported Benefits

- Performance (26 studies)
- Modularity (21 studies)
- Maintainability (18 studies)
- Scalability (17 studies)
- Reusability (16 studies)

Reported Drawbacks

- Usability (5 studies)
- Limited applications (4 studies)
- Complexity (1 study)
- Memory overhead (1 study)
- Maintainability (1 study)

Research Gap 1

Current research on ECS focuses on benefits above the drawbacks, suggesting a positive bias in the field.

Surface Level Understanding of ECS

Most papers (27/35) discuss ECS at a high-level

- Entity, components, systems
- Data and behaviour separation
- Software qualities

Fewer studies (8/35) investigate lower-level concepts

- Storage models
- Memory layouts
- Cache behaviour

Research Gap 2

Current ECS literature provides limited low-level empirical analysis of ECS runtime behaviour.

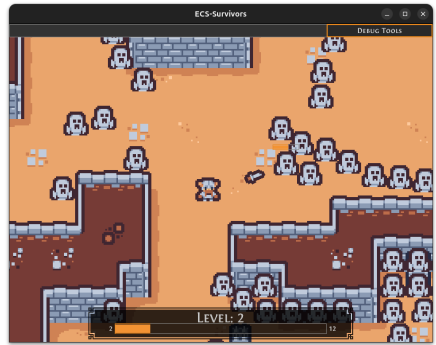
ECS-Survivors: Case Study & Motivation

About the project

- Real-time survivors-like game built using the Flecs ECS framework
- Featured on the official Flecs Web site
- Hundreds to thousands of dynamic entities with collision-heavy gameplay

Why collision handling?

- Continuous, high-frequency updates
- Forces inter-entity and cross-system interaction, stressing ECS's low coupling

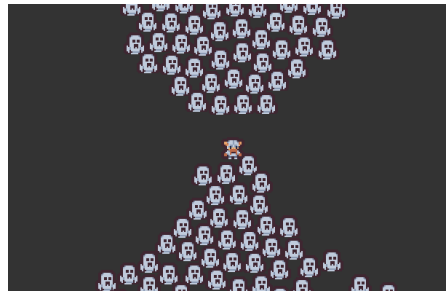


Scope

Collision handling is a mechanism for evaluating ECS runtime performance, not a rigorous physics engine.

Experimental Setup

- Six implementations
- 30 repetitions
- Metrics collected every frame:
  - Frame time & entity counts
  - Cache behaviour
- Cognitive complexity
- Cyclomatic complexity



Research Questions

- **RQ2.1** How do different ECS collision-handling methods exhibit measurable runtime performance variation?
- **RQ2.2** How do the different integrations of these methods introduce code complexity that can be quantified using cognitive and cyclomatic metrics?

Collision-handling Implementations (1/2)

Relationship-based

I1 Relationships

- On collision: add `CollidedWith(B)` to entity A and vice versa
- Each distinct pair → new archetype
- High fragmentation risk

I2 Non-fragmenting Relationships

- Same as I1 but with `DontFragment` trait
- All `CollidedWith(*)` stored in a single archetype
- Eliminates archetype explosion

Record-based

I3 Record Entity

- On collision: spawn a new entity with `CollisionRecord(A, B)`
- Resolve, then delete all record entities
- No structural changes on colliders

I4 Record List

- On collision: append to the `RecordList` global component
- Resolve, then clear the list
- No structural changes on colliders

Collision-handling Implementations (2/2)

Spatial Partitioning
I5 Per-cell hashing

- Entities grouped into spatial cells
- Iterates cell by cell
- Collides owned entities with neighbours

I6 Per-entity hashing

- Entities grouped into spatial cells
- Iterates entity by entity
- Lookup entity cell
- Collide with neighbours

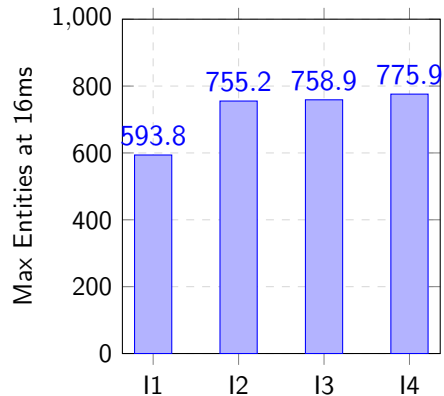
Ericson, C. (2004). *Real-time Collision Detection*. CRC Press.

Performance Results: ECS Mutation Trade-offs

- Relationship implementation I1 scaled poorly as entity counts increased
- Frequent structural changes generated archetype fragmentation
- Fragmentation introduced measurable iteration overhead

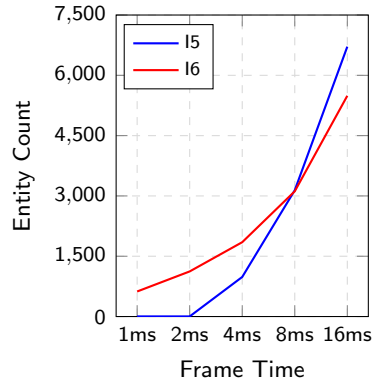
Key Finding

Archetype mutations and fragmentation introduced measurable runtime costs.



Performance Results: Spatial Partitioning

- Both implementations significantly outperformed naive approaches
- I5 scaled better at higher entity counts due to improved cache locality
- I6 performed better at lower entity counts from lower total work



Key Finding

Different memory-access patterns influenced scalability behaviour.

Impl.	16ms	8ms	4ms
I5	6,713.2	3,141.8	984.3
I6	5,491.2	3,115.3	1,849.3

Complexity Results

- However, complexity must be considered relative to performance gains
- We therefore computed an efficiency index:

$$\text{Eff} = \frac{E}{\sqrt{C_c C_g} \cdot S}$$

where:

- E = supported entities
- C_c = cyclomatic complexity
- C_g = cognitive complexity
- S = system count

- Best balance: I3, I2, I5

Impl.	Cyclo.	Cogn.	Systems
I1	9	8	3
I2	9	8	3
I3	9	8	3
I4	13	14	4
I5	23	38	8
I6	23	32	8

Impl.	Eff (16 ms)
I1	23.33
I2	29.68
I3	29.81
I4	14.38
I5	29.02
I6	25.87

AHP Results

S1 - Performance first

- **I5** wins (0.346)
- Spatial hashing scales best

S2 - Complexity first

- **I2 & I3** tie (0.205)
- Simpler naive approaches

S3 - Balanced

- **I5** still leads (0.246)
- Spatial hashing offers the best overall trade-off

Scenario	Priority	λ
S1	Performance $\gg$ Complexity	3
S2	Complexity $\gg$ Performance	1/3
S3	Performance $=$ Complexity	1

Impl.	S1	S2	S3
I1	0.093	0.199	0.146
I2	0.112	0.205	0.158
I3	0.112	0.205	0.158
I4	0.086	0.139	0.107
I5	0.346	0.147	0.246
I6	0.253	0.116	0.184

Best 2nd Worst

Discussion

Archetype Fragmentation

- Relationship-based approaches introduce fragmentation silently at the API level
- `DontFragment` is a simple fix with $\sim$25% gains, but requires awareness
- Ties in to RG2

Spatial Hashing Trade-offs

- I5 is better at high entity counts, but fails under tight budgets (1–2ms)
- I6 is more consistent across scales
- Crossover point $\approx$ 3,100 entities

Performance vs. Complexity

- Spatial hashing is nearly an order of magnitude faster than naive approaches
- But 2–3$\times$ more systems, higher cyclomatic and cognitive complexity
- Vampire Survivors caps enemies at 300 entities, I2 and I3 may be sufficient

Future Work

Short Term

- Extend mapping study to grey literature
- Support diverse collision shapes and entity sizes
- Replicate study across other ECS frameworks
- Compare ECS implementations against OOP/component-based approaches

Long Term

- **ECS-specific static analysis tooling**
  - OOP-to-ECS automated migration
  - Accidental system matching detection
- **Unified CPU–GPU ECS framework**
  - Runtime dynamism + GPU parallelism
  - GPU code-generated systems
  - System scheduler and CPU-GPU synchroniser

Conclusion

Thesis Takeaway

This work provides concrete, empirical evidence to complement the literature, and practical guidance for developers in performance-sensitive ECS domains.

Main Contributions:

- Systematic mapping study
- Empirical case study
- Practical guidance (AHP)
- Open source code for ECS-Survivors
- Open source code experimentation for case-study

Appendix

Quicklinks:
https://github.com/ptidejteam/ecs-survivors/tree/main
https://laurent-voisard.itch.io/ecs-survivors
https://github.com/sandermertens/flecs#ecs-survivors
https://github.com/LVoisard/ecs-survivors-collision-test

Mapping Study Methodology

Databases

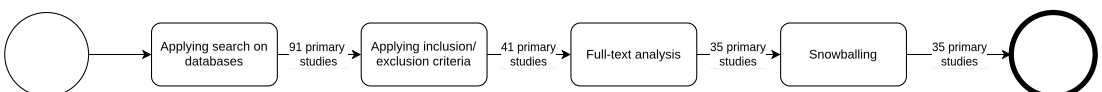
- ACM Digital Library
- IEEE Xplore
- Scopus

Search String

Entity Component System"*
OR
Entity-Component-System"*

Selection Process

- Inclusion/exclusion criteria
- Full-text review
- Snowballing process



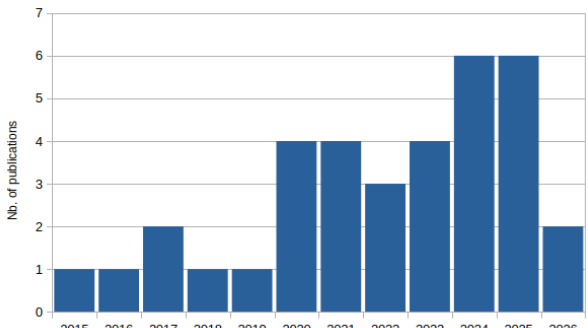
Result

91 initial studies $\rightarrow$ 35 final primary studies

Mapping Study Findings

Research Domains

- Real-time applications
- Video games
- Game engines
- Virtual reality
- Emerging non-game domains



ECS Storage Models

Basic ECS

- Simple implementation
- Memory waste

Entity ID	Position	Velocity
0	(0, 0)	(1, -1)
1	(10, 10)	(0, -1)
2	(30, 23)	null

ECS Storage Models Cont'

Sparse-Set ECS

- Fast component additions/removals
- Lower structural-change cost

Entity ID	Sparse Positions	Sparse Velocities
0	0	0
1	1	1
2	2	null

Dense Positions
(0, 0)
(10, 10)
(30, 23)

Dense Velocities
(1, -1)
(0, -1)

AHP Decision Framework

- Supplement Eff. index
- Structured multi-criteria decision making
- Balances **performance** vs. **complexity**
- Produces ranked scores per scenario

Three Developer Scenarios

Scenario	Priority	λ
S1	Performance $\gg$ Complexity	3
S2	Complexity $\gg$ Performance	$1/3$
S3	Performance $=$ Complexity	1

Criteria Pairwise Table

	Perf.	Complexity
Performance	1	$1/\lambda$
Complexity	λ	1

Consistency Ratios (CR)

- Performance CR: 2.6%
- Complexity CR: 0.6%

Both pass the 10% threshold.

AHP Performance Table

Table: Pairwise comparison of methods under the *performance* criteria

Implementation	**I1**	**I2**	**I3**	**I4**	**I5**	**I6**
I1	1	1/2	1/2	1/2	1/8	1/7
I2	2	1	1	1/2	1/7	1/6
I3	2	1	1	1	1/7	1/6
I4	2	1	1	1	1/7	1/6
I5	8	7	7	7	1	2
I6	7	6	6	6	1/2	1

AHP Table Complexity

Table: Pairwise comparison of methods under the *complexity* criteria

Implementation	**I1**	**I2**	**I3**	**I4**	**I5**	**I6**
I1	1	1	1	2	5	5
I2	1	1	1	2	5	5
I3	1	1	1	2	5	5
I4	1/2	1/2	1/2	1	4	4
I5	1/5	1/5	1/5	1/4	1	1
I6	1/5	1/5	1/5	1/4	1	1