CECS: A Concurrent Entity Component System

Voor
College of Eng. and Computer Science
University of Central Florida
Orlando, Florida
jamesvoor@knights.ucf.edu

Bramham
College of Eng. and Computer Science
University of Central Florida
Orlando, Florida
connor.bramham@knights.ucf.edu

Vargas
College of Eng. and Computer Science
University of Central Florida
Orlando, Florida
angel0615@knights.ucf.edu

***Abstract*—The CECS is a project which implements concurrent programming concepts to meet the performance needs of modern simulated world applications. We discuss the correct program behavior of an Entity-Component-System, and the challenges associated with parallelizing this program behavior. We discuss the successes of previous entity-component-system projects, and opportunities we identified which could improve performance. We discuss how our ECS system compares to established work in the field.**

***Index Terms*—concurrent, entity-component-system, concurrency, archetypes, Rust language**

I. Introduction

An Entity-Component-System (ECS) is a software pattern used to simulate a model of some defined world. It is most often used in video games to simulate the game world. There are use cases for the ECS pattern outside of games as described in the "Review of Related Work" section. There is much debate over what an ECS actually is, but we will be using the model as described by Martin [1]. In this model, there are three core concepts: the entity, the component, and the system. To demonstrate these concepts, I will give examples of practical usages.

A. Entity

An entity is a "thing" within the world. They contain no data and have no logic. In essence, they are an identifier for some object, and because of this they are usually implemented simply as a unique integer identifier.

As an example of what an entity is, consider the game Flappy Bird. The bird and the pipes can be modeled as entities.

B. Component

A component is a piece of data that is associated with an entity. They are what give entities meaning. Components, like entities, have no logic. They are simply containers for data. An entity may have any number of components, including none at all (although the use cases for such entities is limited). Entities may also have many different kinds of components. Some implementations also allow for entities to have multiple instances of the same component type.

As an example, consider our previous example, Flappy Bird. As established before, the bird is an entity. To describe the bird, we may use the following components:

- A `Controllable` component which acts as a flag to indicate that the bird can be controlled by the player.
- A `Renderable` component which describes how the bird should look on screen.
- A `Collider` component which enables collision detection with the pipes.
- A `RigidBody` component which allows the bird to fall following the laws of physics.
- A `Position` component which locates the bird in space.

We could describe the pipes using these components:

- A `Renderable` component to show the pipe on screen.
- A `Collider` component which enables collision detection with the bird.
- A `Sliding` component to move the pipe from the right side of the screen to the left over time.
- A `Position` component which locates the pipe in space.

Notice that the pipes have a `Sliding` component instead of a `RigidBody` because they do not behave using the laws of physics. Additionally, since the pipes are not controlled by the player, they do not need a `Controllable` component.

C. System

Systems are the logic that give entities behavior. They contain no data. A system mutates a subset of entities based on which components they have. The set of entities that a system operates on is typically called a query. The types of components requested in the query, including whether or not the component type will be mutated or only read, is called a filter.

For our Flappy Bird example, we might have the following systems:

- A `Physics` system which operates on entities with both the `RigidBody` and `Position` components to apply physics computations. It may also operate on entities with both the `Collider` and `Position` components to resolve collisions.
- A `Renderer` system which operates on entities with both the `Renderable` and `Position` components to draw objects on the screen.
- A `PipeMover` system which operates on entities with both the `Sliding` and `Position` components to move them right to left.
- A `PlayerControl` system operates on entities with both the `Controllable` and `Position` components to allow the player to move them.

Notice that the `Physics` system uses two queries. One is the set of entities with both the `RigidBody` and `Position` components. The other is the set of entities with the `Collider` and `Position` components.

D. Review

With only these three concepts it is clear that an ECS can adequately describe a real-world system. The role of the programmer is to describe the world they wish to model using these ideas.

It should also be clear that this model differs heavily from traditional object-oriented programming paradigms. The ECS pattern prefers composition over inheritance and the separation of logic and data.

Beyond this brief overview, there are other concepts with appear often in the ECS ecosystem, including the idea of shared resources and events. The focus of this paper is on the idea of a dispatcher, which is the logic used to run the code associated with systems. Our goal was to create an algorithm for dispatchers which enables highly parallel execution of systems with as little overhead for the programmer using the ECS as possible.

II. Review of Related Work

A good reference ECS, built with Rust, is the Specs project, maintained by the Amethyst organization. "Specs is close to the design of a classic ECS. Each component is stored in a Storage that contains a collection of like elements" [6]. It was built to prioritizes flexibility and achieves a baseline performance relative to other ECS projects discussed here.

The Amethyst organization built a successor to the Specs ECS, called Legion. "Legion aims to be a feature rich high performance Entity component system (ECS) library for Rust game projects" [1]. The biggest change between Specs and Legion is the use of a structure called an Archetype. This archetype structure improves entity search and filtering performance, "giving us contiguous runs of components which can be indexed together to access each entity" [4]. For a visual aid on ECS archetypes, refer to figure 1 on page 2.

The Concurrent Entity Component System takes inspiration from Legion's Archetype implementation innovation, and attempts to improve it's dispatcher algorithm, increasing parallelization of work, and performance overall. The dispatcher algorithm can be thought of as a work scheduling algorithm, to coordinate thread usage of resources. To this end, our algorithm implements a combination of the fork-join model and the topological sorting model, for organizing concurrent threads.

The ECS systems from which our concurrent entity component system takes inspiration from are designed specifically for use in video game engines. However entity component systems have a much broader application than video games alone, and can be found in such applications as Geographical Wildlife Epidemiological Modeling [2], radar simulation [3] and graphical user interfaces [5].

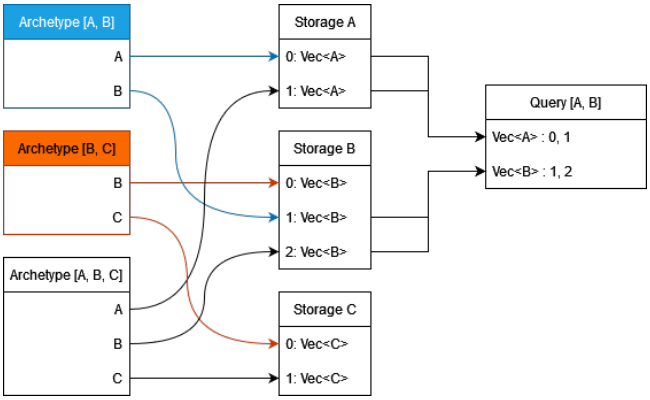

Fig. 1. ECS Archetypes diagram

III. Our Entity-Component-System Model

In an Entity Component System the purpose for using the lower level overview would be for simple components to make up the systems that they are categorized into rather than to have the control be completely delegated to the super ceding set. Components by themselves have no descriptive factor that tell what they are to do. For this we must use a connector to associate the different types of components that are used by an entity. What we call these varieties are Archetypes. Archetypes are ways to categorize the components into an overarching set that contains the same characteristics. In addition to the abstraction of smaller components to Archetypes as a classification method, the use of Archetypes will create order from a memory standpoint. With an ECS, due to the the varying amount of components storage for these could become very expensive in storing. Using the Archetype model, we can create a bound of layered space so that the individual components are fitted into a Table like structure. This will save space and can create better performance for when the computer architecture has to fetch data by words. It is a more effective approach than having to frequently getting non-contiguous data blocks.

Here the diagrams illustrate the way that the Archetypes relate to their data models for storage

For our ECS we were trying multiple methods of ensuring that the different components may follow sequential consistency whilst following our goal of improved performance by using parallelism and multi-threading. Prior to this, we had implemented multiple models that were used to attempt to create a feasible process. Here we will list out some of our implementations that did not work so well:

A. Methods tried

- The Fork Join Model —
  We initially looked into the use of the Fork-Join model as a viable option for setting up the execution of concurrent blocks within our systems. From an operating standpoint this would be a legitimate method of organizing operations. Alternatively, when looking at the standpoint of

- Topological Mapping structure —
  Another method that had come to mind was the use of a mapping structure that would look at the hierarchy of the program and create a correct structure in which to run the program. What we had in mind was the use of a topologically sorted system. This would create a set of rules that indicate what is being written to compared to what is being read to and give precedence to those commands that came in sequential order. All whilst still allowing for reordering of operations that carried no sequential read/write dependencies. An example of this would be the in the scenario in which we have three components, two which rely on registers A & B, and one that pertains to only register C. Here we would have the freedom to run the third system in any execution that we like, as it is independent. On the contrary, since we have the first two that rely on the same register, we must preserve order so that they can access and write data in a historically consistent manner. The main drawback that prevents us from using an analysis method to do this is that we can not accurately predict the manner in which the program will run. This would obviously be a solution to the Halting problem, so we can tell that this is an unsolvable situation. We must therefore take a different approach.

In the end we decided to use the solution that combined some of both aspects. We will be taking into account the aspect of the fork-join model that has to deal with the order of operations in which different systems are run. We will do this by giving the user implicit declarations in the code. In the topologically sorting model, one would have to predict how the code would be running during compile-time, but if the user has some awareness in how the systems and their component's order of operations will occur this will much simplify things. Another aspect that we will be implementing atop this will be the ability to order a system with a given priority, similar to how an operating system will give user defined priority to certain threads that are either system or chosen by the user. Another term that is known widely is "work stealing" which is the act of load balancing amongst threads by the use of a thread-pool to ensure that unlike the fork-join model, there is always constant work. In using queues of sorts, the system can keep popping more systems and operations that are available to be executed.

IV. Correctness

EXAMPLE TEXT, to complete after project completion

V. Performance

EXAMPLE TEXT, to complete after project completion

VI. Appendix

The Concurrent Entity Component System has some features complete, but much is still left to be done. Completed modules include the panic-read-write-lock, the archetype container system, the component filter and buffer are complete. The systems dependency logic module needs to be completed, as well as the major feature of CECS, the dispatcher. Additionally, a testing interface which will measure the performance of CECS needs to be implemented. Performance will be evaluated in such a way that we can directly compare our program's performance to other ECS projects in the field of gaming applications.

References

[1] Amethyst. High performance rust ecs library. github, oct 2021.

[2] Austin V. Davis and Shaowen Wang. A concurrent entity component system for geographical wildlife epidemiological modeling. *Geographical Analysis*, 53(4):836–868, 2021.

[3] Brennen Garland. Designing and building a radar simulation using the entity component system. Thesis, Air Force Institute of Technology, jul 2021.

[4] Tom Gillen. Legion ecs - v0.3, Sep 2020.

[5] Thibault Raffaillac and Stéphane Huot. Polyphony: Programming interfaces and interactions with the entity-component-system model. *Proc. ACM Hum.-Comput. Interact.*, 3(EICS), jun 2019.

[6] Cora Sherratt. Specs and legion, two very different approaches to ecs. blog, mar 2020.