IMPLEMENTATION AND ANALYSIS OF THE ENTITY COMPONENT SYSTEM ARCHITECTURE

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Shawn Harris

March 2022

© 2022
Shawn Harris
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	Implementation and Analysis of the Entity Component System Architecture
AUTHOR:	Shawn Harris
DATE SUBMITTED:	March 2022
COMMITTEE CHAIR:	Christian Eckhardt, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Maria Pantoja, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Jonathon Ventura, Ph.D. Professor of Computer Science

ABSTRACT

Implementation and Analysis of the Entity Component System Architecture

Shawn Harris

The entity component system architecture (ECSA) is a data-oriented composition pattern and a data-driven design pattern. Data-oriented software takes into consideration generalized knowledge of hardware. Data-driven design is a methodology used to replace inflexible code with reusable components that can be added, deleted or modified in interactive systems and games.

This thesis explores the ECSA and its alternatives and their strengths and weaknesses. The paper details the creation of an ECSA and bench-marks its performance against object oriented architectures. The hypothesis of this thesis is that the ECSA has CPU cache performance advantages over object oriented architectures as tested by multiple benchmarks.

The results suggest that the ECSA provides superior CPU performance. These results could be valuable for: interactive game developers to get higher frame-rates out of their games, mmorpg server developers to process millions of entities per second, and mobile developers to create battery efficient apps.

ACKNOWLEDGMENTS

Thanks to:

- My Wife and Son, who are my best friends and greatest supporters.
- Dr. Christian Eckhardt, for being my advisor and friend and guiding me through this process.
- Dr. Maria Pantoja, Dr. Jonathon Ventura, for their instruction and support.
- Dr. Zoe Wood, for teaching me computer graphics.
- Andrew Guenther, for uploading this template. [1]

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES xi

CHAPTER

1 Introduction 1

1.1 CPU Cache Performance 1

1.2 Memory Layouts: AoS and SoA 3

1.2.1 Array of Structures 3

1.2.2 Structure of Arrays 5

1.3 Data Oriented Design 6

1.3.1 Contrasting Object Oriented Design 6

1.3.1.1 Polymorphism and Inheritance 7

1.3.1.2 Virtual Functions 7

1.4 Data Driven Design 8

1.4.1 Entity Encoding as a form of Data Driven Design 8

1.5 Inheritance Hierarchy 9

1.6 Game Object Architecture (GOA) 11

1.7 Entity Component System Architecture 14

1.7.1 Entities 14

1.7.2 Components 14

1.7.3 Entity Component Visualization 15

1.7.4 Systems 15

1.8 Summary 17

2 Related Works 18
2.1 Open Source ECSAs 18
2.2 Unity3D and Unreal Engines 18
3 Implementation 19
3.1 Zip 19
3.1.1 zipsrc 19
3.1.2 Entities 20
3.1.3 Components 21
3.1.4 Systems 23
3.1.5 Zip Event Handling 23
3.2 zipgen 25
3.3 zipapp 28
3.4 goa test 29
4 Results 30
4.1 Performance Testing 30
4.1.1 Hardware 30
4.2 Asteroids Benchmark 31
4.2.1 Asteroids Summarized Results 32
4.2.2 Asteroids Rendering Disabled 38
4.2.3 Individual Asteroids Benchmarks 41
4.2.4 Gravity O(n^2) Benchmark 48
4.2.4.1 Combined Benchmark 48
4.2.4.2 Individual Benchmarks 53
4.3 Update x1 System Benchmark 58
4.3.1 Combined Results 58

4.3.2 Individual Benchmarks 64
4.3.3 Individual Results Analysis 65
4.4 Create and Destroy Benchmark 72
4.4.1 Combined Results 73
4.4.2 Individual Benchmarks 78
4.4.3 Individual Results Analysis 78
4.5 Locality of Reference Benchmark 85
4.5.1 Combined Results 85
4.5.2 Individual Benchmarks 90
4.5.3 Individual Results Analysis 90
5 Conclusion 98
5.1 Conclusion 98
6 Future Works 99
6.1 Multi-Threading 99
BIBLIOGRAPHY 101

LIST OF TABLES

Table Page

4.1 Asteroids Rendered Benchmark 34

4.1 Asteroids Rendered Benchmark 35

4.1 Asteroids Rendered Benchmark 36

4.1 Asteroids Rendered Benchmark 37

4.2 Asteroids Data Not Rendered 39

4.2 Asteroids Data Not Rendered 40

4.3 Asteroids Individual Benchmarks 41

4.3 Asteroids Individual Benchmarks 42

4.4 Gravity $O(n^2)$ Combined Benchmark 49

4.4 Gravity $O(n^2)$ Combined Benchmark 50

4.4 Gravity $O(n^2)$ Combined Benchmark 51

4.4 Gravity $O(n^2)$ Combined Benchmark 52

4.5 Gravity $O(n^2)$ Individual Benchmarks 53

4.6 Update x1 Combined Benchmark 59

4.6 Update x1 Combined Benchmark 60

4.6 Update x1 Combined Benchmark 61

4.6 Update x1 Combined Benchmark 62

4.6 Update x1 Combined Benchmark 63

4.7 Update x1 Individual Benchmarks 64

4.8 Create and Destroy Combined Benchmark 73

4.8 Create and Destroy Combined Benchmark 74

4.8 Create and Destroy Combined Benchmark 75
4.8 Create and Destroy Combined Benchmark 76
4.8 Create and Destroy Combined Benchmark 77
4.9 Create and Destroy Individual Benchmarks 78
4.9 Create and Destroy Individual Benchmarks 79
4.10 Locality of Reference Combined Results 86
4.10 Locality of Reference Combined Results 87
4.10 Locality of Reference Combined Results 88
4.10 Locality of Reference Combined Results 89
4.10 Locality of Reference Combined Results 90
4.11 Locality of Reference Individual Tests 91

LIST OF FIGURES

Figure		Page
1.1	CPU vs DRAM Performance	1
1.2	Example AoS code with C structs.	3
1.3	Example AoS code with OOP classes.	4
1.4	Example SoA code.	5
1.5	An example inheritance diagram.	9
1.6	An example game object class diagram.	11
1.7	Example Game Objects Code in C++	12
1.8	An example ecsa schema.	15
1.9	An example system query against an entity.	16
1.10	Entity Encoding Pros and Cons.	17
3.1	zipsrc diagram.	20
3.2	component base class in zip.	21
3.3	example of a component in zip.	21
3.4	example of a system in zip.	23
3.5	Sample YAML config file	25
3.6	zipgen example	26
3.7	Sample YAML config file with modules option	27
4.1	A close up of a planet and randomly generated asteroids.	31
4.2	A zoomed out scene the planet and orbiting asteroids.	32
4.3	Rendered Asteroids Benchmark	33
4.4	Not Rendered Asteroids Benchmark	38

4.5 $1e3$ Asteroids zip ecsa vs game objects oop. 42
4.6 $1e4$ Asteroids zip ecsa vs game objects oop. 43
4.7 $2.5e4$ Asteroids zip ecsa vs game objects oop. 43
4.8 $5e4$ Asteroids zip ecsa vs game objects oop. 44
4.9 $1e5$ Asteroids zip ecsa vs game objects oop. 44
4.10 $2.5e5$ Asteroids zip ecsa vs game objects oop. 45
4.11 $5e5$ Asteroids zip ecsa vs game objects oop. 45
4.12 $1e6$ Asteroids zip ecsa vs game objects oop. 46
4.13 $2.5e6$ Asteroids zip ecsa vs game objects oop. 46
4.14 $5e6$ Asteroids zip ecsa vs game objects oop. 47
4.15 $10e6$ Asteroids zip ecsa vs game objects oop. 47
4.16 Gravity $O(n^2)$ Combined Benchmark 49
4.17 Gravity $O(n^2)$ Benchmark for 100 asteroids comparing the frame-time(s) in ms. 54
4.18 Gravity $O(n^2)$ Benchmark for 500 asteroids comparing the frame-time(s) in ms. 54
4.19 $O(n^2)$ Benchmark for $1k$ asteroids. Lower is better. 55
4.20 Gravity $O(n^2)$ Benchmark for $5k$ asteroids comparing the frame-time(s) in ms. 55
4.21 Gravity $O(n^2)$ Benchmark for $10k$ asteroids comparing the frame-time(s) in ms. 56
4.22 Gravity $O(n^2)$ Benchmark for $50k$ asteroids comparing the frame-time(s) in ms. 56
4.23 Gravity $O(n^2)$ Benchmark for $100k$ asteroids 57
4.24 Combined Update x1 Benchmark. 59
4.25 Update x1 Benchmark $1k$ entities. 66
4.26 Update x1 Benchmark $10k$ entities. 66

4.27 Update x1 Benchmark 25k entities. 67
4.28 Update x1 Benchmark 50k entities. 67
4.29 Update x1 Benchmark 100k entities. 68
4.30 Update x1 Benchmark 250k entities. 68
4.31 Update x1 Benchmark 500k entities. 69
4.32 Update x1 Benchmark 1M entities. 69
4.33 Update x1 Benchmark 2.5M entities. 70
4.34 Update x1 Benchmark 5M entities. 70
4.35 Update x1 Benchmark 10M entities. 71
4.36 Combined Create Destroy Benchmark. 72
4.37 Create Destroy Benchmark 1k entities. 79
4.38 Create Destroy Benchmark 10k entities. 80
4.39 Create Destroy Benchmark 25k entities. 80
4.40 Create Destroy Benchmark 50k entities. 81
4.41 Create Destroy Benchmark 100k entities. 81
4.42 Create Destroy Benchmark 250k entities. 82
4.43 Create Destroy Benchmark 500k entities. 82
4.44 Create Destroy Benchmark 1M entities. 83
4.45 Create Destroy Benchmark 2.5M entities. 83
4.46 Create Destroy Benchmark 5M entities. 84
4.47 Create Destroy Benchmark 10M entities. 84
4.48 Combined Update x1 Benchmark. 86
4.49 Memory Locality Benchmark 1k entities. 92
4.50 Memory Locality Benchmark 10k entities. 92
4.51 Memory Locality Benchmark 25k entities. 93

4.52 Memory Locality Benchmark 50k entities. 93

4.53 Memory Locality Benchmark 100k entities. 94

4.54 Memory Locality Benchmark 250k entities. 94

4.55 Memory Locality Benchmark 500k entities. 95

4.56 Memory Locality Benchmark 1M entities. 95

4.57 Memory Locality Benchmark 2.5M entities. 96

4.58 Memory Locality Benchmark 5M entities. 96

4.59 Memory Locality Benchmark 10M entities. 97

Chapter 1

INTRODUCTION

1.1 CPU Cache Performance

Around the year 1980, microprocessors were still being manufactured without cache memory inside the microprocessor. Over the course of the next several decades the L1, L2, and then L3 cache were invented to solve the problems caused by the disparity between system memory speed and microprocessor computation speed. L4 caches now exist and soon L5 caches will. These cache technologies bridge the performance gap between microprocessor speed and system DRAM memory speed and have become an essential part of the microprocessor [10] [12].

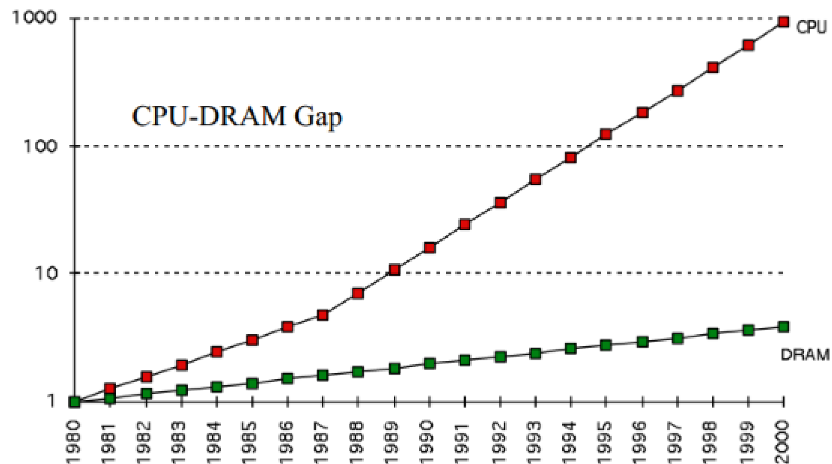

Figure 1.1: CPU vs DRAM Performance
[10]

CPU caches are designed to store the data that is most likely to be needed in the near future of a program. When that data isn't present in the CPU, it has to be loaded or fetched from memory and is referred to as a cache-miss. Cache-misses are expensive events can cost hundreds of CPU cycles [7]. A cache-miss event requires loading the data from the much slower system memory [16]. Software that is created with an awareness of these issues can reduce cache-misses improve the CPU performance, resulting in faster run-time, and in the case of mobile devices, lower power consumption.

1.2 Memory Layouts: AoS and SoA

The terms Array of Structures, and Structure of Arrays refer to layouts for storing data in memory. These memory layouts can directly enhance or hinder CPU cache performance in a software program. These terms were originally documented by Intel as methods for optimizing the performance of SIMD instructions but they also have usefulness for reasoning about vectorizing memory and data operations in general [2].

1.2.1 Array of Structures

Array of Structures is a type of layout where data or records are organized at the lowest level by the structure or individual record or row as they are in a database table. This memory layout is commonly used in object oriented software because that is how classes organize their data members by definition [18].

```
struct Person {
    char name[32];
    uint height;
    uint weight;
    uint age;
};

Person persons[COUNT];
```

Figure 1.2: Example AoS code with C structs.

This is an intuitive and natural way for many developers to organize code but it may be very inefficient in the CPU when thousands or millions records are processed such as in a game loop. Of course, this depends on the application. For example, if some operation is performed with each field independently such as first processing the

name, then the height, and then the weight and age. If too much unrelated data is loaded during the operation that means cache misses will happen more often. When object oriented programming is introduced, as in Figure 1.3, then the performance can get much worse.

```
class Shape {
public:
    virtual uint area() = 0;
};

class Square : public Shape {
public:
    virtual uint area() { return height * width; }
    uint height;
    uint weight;
    ...
    \\ other data
};

class Circle : public Shape {
public:
    virtual uint area() {return PI * radius * radius;}
    uint radius:
    ...
    \\ other data
}

Shape *shapes[COUNT];
```

Figure 1.3: Example AoS code with OOP classes.

Additional data is now being loaded when only the members for computing area are requested and the object instances are polymorphic and can no longer be stored in a contiguous array and must be stored as an array of pointers. Each pointer will have to be loaded to find the objects memory address and then that address loaded to get the class data into memory.

1.2.2 Structure of Arrays

The Structure of Arrays layout method is where the fields of a record or columns are organized like the database into individual columns with each column being a contiguous array. Figure 1.4 shows example code for the SoA style. The data is now organized in memory in a way that will be more beneficial to CPU performance. Actual performance will always vary based on the data and the application.

```
struct People {
    char names[COUNT][32];
    uint heights[COUNT];
    uint weights[COUNT];
    uint ages[COUNT];
};

People people;

for (int i = 0; i < COUNT; i++)
    \\ do something with people.names[i]
```

Figure 1.4: Example SoA code.

1.3 Data Oriented Design

Data oriented design [3] [8] is the practice of designing software with some general notion of the hardware it will execute on and organizing the code and storage layout in a way that will perform efficiently rather than just treating the hardware as a virtualized unknown black box and writing code with no consideration of hardware. Software that requires high performance usually runs on hardware reasoning about that hardware can make a non trivial difference in how well the software can perform, or in the case of mobile software, how efficiently it uses the battery. Data oriented code generally utilizes the SoA memory layout. Data oriented programs focus on a separation between data and logic, and on utilizing data efficiently on the CPU. Data oriented design is quite different from object oriented design and it is important to understand the contrast between the paradigms.

1.3.1 Contrasting Object Oriented Design

For the purpose of this thesis, Object oriented programs [18] or OOP will be taken to mean software that has one or more of these features:

1. Classes
2. Inheritance
3. Polymorphism

OOP programs usually use an AoS memory layout by virtue of their very nature and can have inefficient cache performance when it comes to dealing with thousands or millions of objects in a program. OOP programming can be very useful for some problems but it is important to understand which of the features can effect program

performance and be able to design program structure to work around those issues and evaluate them on a case by case basis. This thesis is not attempting to villianize OOP and in Chapter 3, it will be shown some OOP concepts can be useful for implementing an ECSA.

1.3.1.1 Polymorphism and Inheritance

Inheritance in and of itself does not cause performance problems in game programming. When polymorphism is introduced the program can no longer store large collections of object instances in contiguous arrays. Those objects will have to be referred to by pointer. Dereferencing many pointers in a loop can cause an increase in cache-misses in the CPU.

1.3.1.2 Virtual Functions

Virtual functions are useful but they do have a drawback, their address and exact code can not be deduced by the compiler and therefore they are not candidates for in-lining. This may introduce a performance problem if that function is being called on thousands or millions of entities, 60 times per second. Performance of virtual functions vs non virtual functions is not a focus of this thesis but are mentioned here because they are an area where data oriented design differs from object oriented design.

1.4 Data Driven Design

Data driven design is a methodology used in the creation of game engines and other related software that empowers designers to create games and other content. The core idea is to remove the schema from the code and reduce dependencies on engineers. Modern databases are a good example of this principle. The schema of a table is not part of the code for the database. [4]. Game engine designers have been shifting towards data driven design and this can be easily observed by looking at the popularity of game engines like Unity3D [19] and Unreal [14]. Game engines have become popular tools that empower programmers and non programmers alike to create interactive games and simulations, sometimes with no engineers and little to no programming experience.

1.4.1 Entity Encoding as a form of Data Driven Design

What is an Entity? An entity is a general purpose simulation unit. In a game or interactive simulation an entity is a 'thing' that is simulated each frame. In Unity3D engine, an entity is a game object, in Unreal Engine an entity is an actor. Examples of entities in a video game could include: players, enemies, bullets, etc. With this idea in mind, the general concept of an entity encoding architecture can be introduced where the entities properties or data are known as components, and those components can add or modify the logical behavior of an entity. The three entity encoding architectures covered in this thesis are: Inheritance, Game Objects Architecture and Entity Component Systems Architecture.

1.5 Inheritance Hierarchy

One way to represent entities is with objects using an object oriented language like C++ [18]. Object oriented programming is very popular and for some people it might even be the natural way to reason about programs because it is how they learned to program. Consider the hypothetical made up inheritance tree in Figure 1.5. It shows a very small set of the classes that might exist in a game. There are some problems here. First of all, the behavior is not dynamic, code cannot be dynamically inherited and uninherited at run-time so this is already violating the idea of data driven design. Second, what happens when the developer wants to make a new monster that is a skeleton and fights with both a sword and magic spells? They can make a new C++ class but not reusing code is bad software engineering and will lead to more problems in the future. They could use multiple inheritance to get code reuse but then they would inherit (pun intended) even more problems. Multiple inheritance is usually considered a bad software engineering practice, and for good reason.

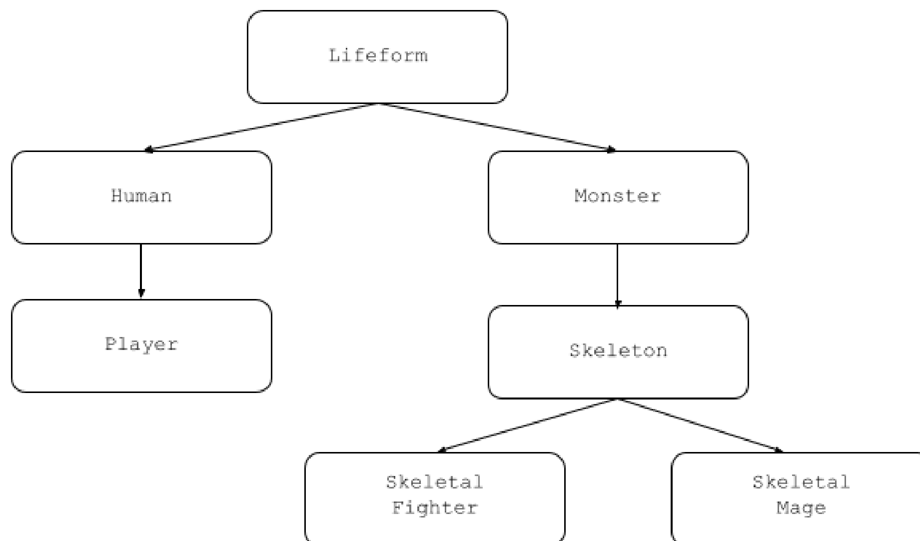

Figure 1.5: An example inheritance diagram.

This is a small and contrived example and the developer might be able to come up with a solution that involves refactoring the code that would make sense. In real games, designs are more complex and could have many hundreds if not thousands of entities and therefore C++ classes. Large and deep inheritance trees are difficult to maintain and hard to refactor and can become a software engineering nightmare.

1.6 Game Object Architecture (GOA)

At a GDC talk in 2002, Scott Bilas [4] described how the popular action RPG, ”Dungeon Siege”, was created and described a data driven design architecture based on what we would now call game objects architecture. This was an object oriented architecture where the core idea was to have one class with common functions that all game entities would share and an abstract class interface called component that would be attached to the game object using composition to create data driven code. These components could be attached, or removed at run-time during the game and could be created by designers without need of engineers using a supported scripting language [5].

This type of architecture has become popular in game engines. It can now be found in the core object component model in the Unity3D [19] and Unreal Engine [14].

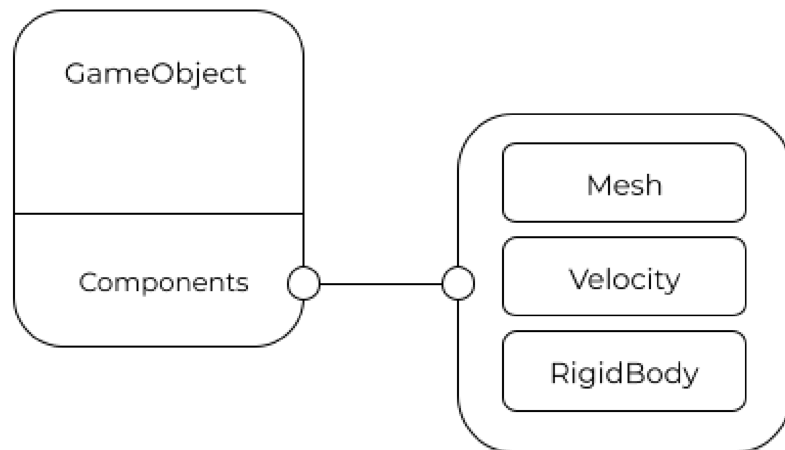

Figure 1.6: An example game object class diagram.

```
class Component {
public:
    virtual void Update() = 0;
    ...
}

class GameObject
{
public:
    void AddComponent(Component *c) {
        components.push_back(c);
    }
    void Update() {
        for (auto c : components)
            c->Update();
    }
    ...
private:
    std::vector<Component *> components;
}
```

Figure 1.7: Example Game Objects Code in C++

Figure 1.6 shows a class diagram for a sample game objects architecture. Note that the game object class is mostly just a container for the components. Some game object architecture implementations put common components like the transform into the game object. The core idea can be understood from the example code in Figure 1.7. To implement a new component, just inherit from 'Component' and implement the new custom behavior. Components can interact with the game object and even other components. Component to component access methods are not detailed here for brevity.

Game objects architecture can meet the data oriented design requirements for interactive flexibility and modification of behavior at run-time but unfortunately it is not very good when it comes to CPU cache performance. GOA is an object oriented design and uses AoS memory layout and polymorphic virtual pointers by necessity of

its design and cache performance will suffer when simulating thousands or millions of entities.

1.7 Entity Component System Architecture

Entity Component System Architecture [13] is a data oriented software paradigm and architecture pattern useful for encoding and processing entities and it solves many of the problems discussed above that object oriented architectures introduce. The ECSA software pattern has three main features:

1.7.1 Entities

In an ECSA, Entities nothing more than identifiers. Entities do not hold data and entities implement no logic. An entity is not a 'container'. The datatype of an entity is dependent on implementation. In this introduction, entities will be treated as integers. The entity signature and the entity id are separate, both will be integers in this introduction. The entity signature is a bit-set used to indicate which components an entity has.

1.7.2 Components

In an ECSA, components only store data and never have any functions nor implement any logic. Components are just data. Components can be structs or scalar types and are stored in arrays of contiguous memory. If an entity 'has' a component then the entity id can be used as an index into the component array to get the value of the component belonging to that entity. Every component has a signature value that can be used to query against the entity signature.

1.7.3 Entity Component Visualization

The relationship between entities and components in an ECSA can be visualized as shown in Figure 1.8.

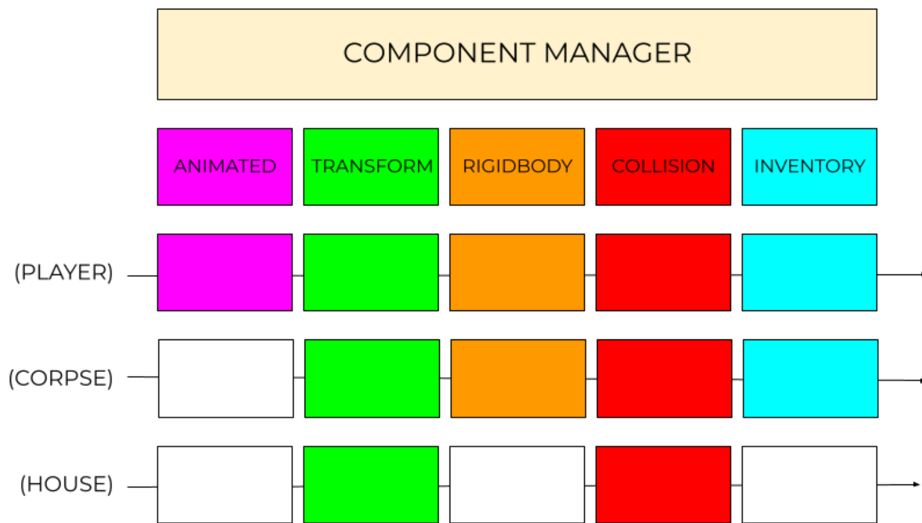

Figure 1.8: An example ecsa schema.

1.7.4 Systems

Systems are functions. Systems are meant to have no data or state. Systems implement logic to mutate components, transform components, add new components, and create or destroy entities. Every system has a unique signature and systems can query against entities and check if an entity signature matches the system's own signature. Figure 1.9 provides a visual representation of the query operation. Each entity signature is checked and if the key fits then a match is made and the entity will be processed by the system. Any entity that matches the query can be assumed to have valid values for the components that were queried. It is dependant

on the implementation but usually there is nothing to prevent code from accessing a component value that the entity doesn't have but if it does it will get invalid data.

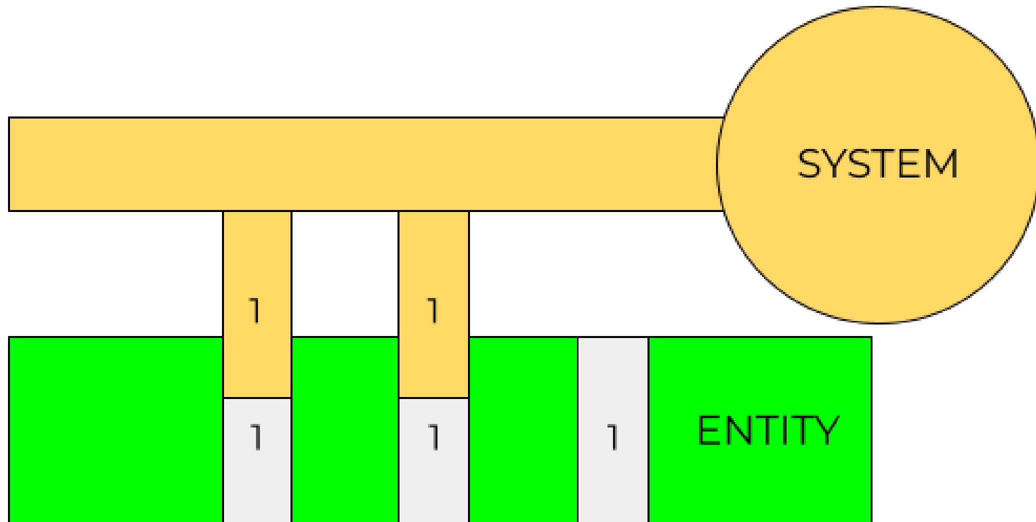

Figure 1.9: An example system query against an entity.

1.8 Summary

Figure 1.10 shows pros and cons of each approach. OOP inheritance is easy to implement but doesn't have the flexibility or good cache performance. Game objects are easy to implement and have flexibility but not good cache performance. Finally, ECSA has flexibility and good cache performance but is harder to implement. The requirement for this research is flexibility and good cache performance so ECSA is a good choice.

	FLEXIBLE	CACHE	EASY
(OOP)			✓
(GO)	✓		✓
(ECSA)	✓	✓	

Figure 1.10: Entity Encoding Pros and Cons.

Chapter 2

RELATED WORKS

2.1 Open Source ECSAs

There are many interesting open source C++ ECSAs, just google, ”open source c++ entity component system”, and many will come up in the results. Since one of the goals of this thesis was to learn and understanding how to create an ECSA, other open source ECSAs were not thoroughly investigated.

2.2 Unity3D and Unreal Engines

Both of Unity3D and Unreal Engine have been referenced in the introduction of this thesis [19] [14] as engines that have game object type component systems and both of these were used as models for the simple game object architecture created for this thesis.

Chapter 3

IMPLEMENTATION

3.1 Zip

Zip is a set of tools and an ecsa implementation created for this thesis. Zip introduces hybrid OOP concepts not usually found in an ecsa. These are convenience features that could be implemented without any using OOP. Using a class with some methods and static members makes the code more concise and does not violate any of the data oriented rules of ecsa.

3.1.1 zipsrc

Zipsrc represents the core source files of zip, which contain generic ecsa code and no component or system definitions. Zipsrc contains the common types of the zip namespace and defines the type of entities, components and systems. It provides an entity manager class for managing the entity component relationships and issuing queries against them.

Figure 3.1 shows the class diagram of zipsrc. EntityManager has no knowledge of components, and its only job is to manage the bit-flags of entities and provide an API to set and clear those flags. EntityManager accepts a template parameter of EntityHandle type so that it can agnostically return handles to entities without knowing how those handles work. EntityHandle type knows about all the components that exist and provides an API for setting and getting their values and testing if an entity has a component. World inherits from EntityManager and world knows about all

components and systems and can be thought of as the component and system manager. All components must be registered with world to set their bit and all systems must be registered with world to establish their call order. Registering a component or system also sets it's signature. Component signatures are always powers of 2, with each successive one increasing by a multiple of 2. System signatures are bit-wise or expressions indicating which components the system wants to match.

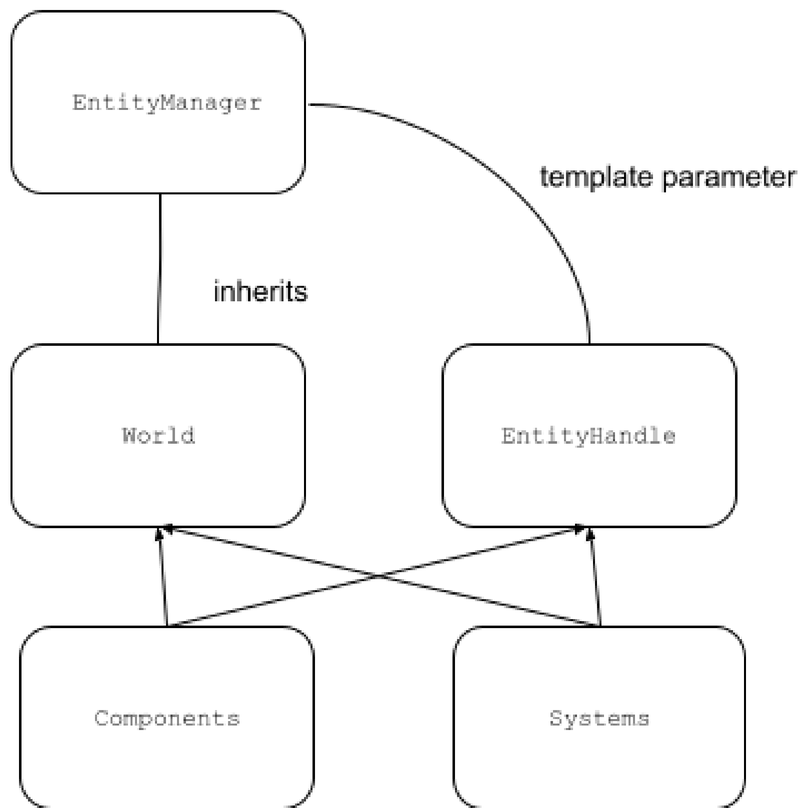

Figure 3.1: zipsrc diagram.

3.1.2 Entities

Entities or EntityIds as they are called in zip are identifiers and in zip they are unsigned 32 bit integers. EntityIds index into an existence array where each stored

value is a 64 bit unsigned integer and each bit is used to store membership by a possible component from the components available to the application. These existence values are called called signatures in zip and they are important for entities, components and systems.

```
using EntityId = uint32_t;
using EntitySig = uint64_t;
```

3.1.3 Components

Components inherit from a base template class and make use of the CRTP template pattern [6].

Base class for the component:

```
template <typename T, typename V>
struct Component {
    inline static Signature sig = 0;
    V value;
};
```

Figure 3.2: component base class in zip.

A component definition:

```
struct VelocityComponent :
    public Component<VelocityComponent, glm::vec3>
{
};
```

Figure 3.3: example of a component in zip.

This still follows the rules of ecsa but the template base class provides some nice conventions the rest of the code can rely on. Important to understand is that although

the component inherits from a template, it has no logic and it is exactly the same size as the value, in this case 12 bytes or a vec3. By using the CRTP pattern every component gets a static member to store the component signature value, and this value is set by the world class. The second template parameter V provides separation between the component type and its value type and establishes a convention that all component values can be retrieved from the value() function regardless of the component type.

3.1.4 Systems

Figure 3.4 provides a sample system definition and an example of using the world/entity manager to query entity signatures for components by that system.

```
void VelocitySystem::update(World* w, float dt) {
    auto count = w->active_count();
    for (EntityId i = 0; i < count; i++) {
    auto eh = w->get_entity_handle(i);
    // sig set elsewhere to
    // PositionComponent::sig | VeloctiyComponent::sig
    if (!eh.has(sig))
        continue;

    auto& pos = eh.position();
    auto& vel = eh.velocity();
    pos = glm::vec3{ pos.x + (dt * vel.x),
        pos.y + (dt * vel.y),
        pos.z + (dt * vel.z) };
    }
}
```

Figure 3.4: example of a system in zip.

Using this example in Figure 3.4, all the positions and components would be stored in contiguous arrays which are owned by the world class.

3.1.5 Zip Event Handling

Zip ecsa also has the ability to raise events when components are added, replaced, or removed from an entity and allow the systems to query and subscribe or process those specific events. This is handled by just creating additional signature arrays, one for "added", one for "replaced" and one for "removed". This feature should be very useful for trying to develop a functioning game but I did not find an objective way to benchmark it in both frameworks since it was a bit of an advanced feature

and I wanted to keep the game objects test program very simple otherwise I would be venturing into an area of making assumptions about how programmers who use game objects implement more advanced features. Added, replaced and removed can be tested with the world functions has_added, has_replaced, and has_removed. These work just like the has function shown in 3.4.

A lot of work went into getting this feature to work correctly and synchronize across all available systems. Zip clears all flags at the end of every update cycle so that events will only be handled once and with that default implementation only a system downstream in the call order would be able to react to an event. This was solved this by using 2 buffers for each of the flag arrays, one is used for queries and the other for writes, they are synchronized at the beginning of each frame. This means when an event is raised it does not actually get processed until the next frame but this does allow every system to see it and respond to it.

3.2 zipgen

In the previous examples there were specific methods for each component type. This makes the code faster than using template meta-programming but it comes with a software engineering cost of maintaining a lot of boilerplate code and that can be painful to write and easy to mess up when copying and pasting. To solve this problem a premake tool called zipgen was created. Zipgen is a tool that generates a zip application (zipapp) from zip source code based on a configuration provided from a YAML config file. The config file can define components and systems which are then turned into full C++ source code files by zipgen.

Zipgen will write out all the source code, including the registration calls in the world class, then the programmer only has to fill in the details inside the system cpp files. zipgen also writes out a main.cpp file and it creates a visual studio solution configured to build the newly created zipapp. A zipapp is the output of zipgen.

Zipgen uses the yaml-cpp library [11] and the YAML file type. YAML is similar to JSON, but it is easier for humans to read and edit. A sample zipgen YAML config file is shown in 3.5.

```yaml
ecs:
    max_entities: 2000000

components:
    - name: Velocity
      value_type: glm::vec3

systems:
    - name: Velocity
      type: execute



```

Figure 3.5: Sample YAML config file

Zipgen reads text templates in from a profile and uses those to generate the zipapp. The profile and text templates allow zipgen to be very flexible and in theory it could generate an entirely different ecsa implementation. Zipgen just has a few rules the text templates need to follow.

Figure 3.6 shows a sample output of the zipgen tool.

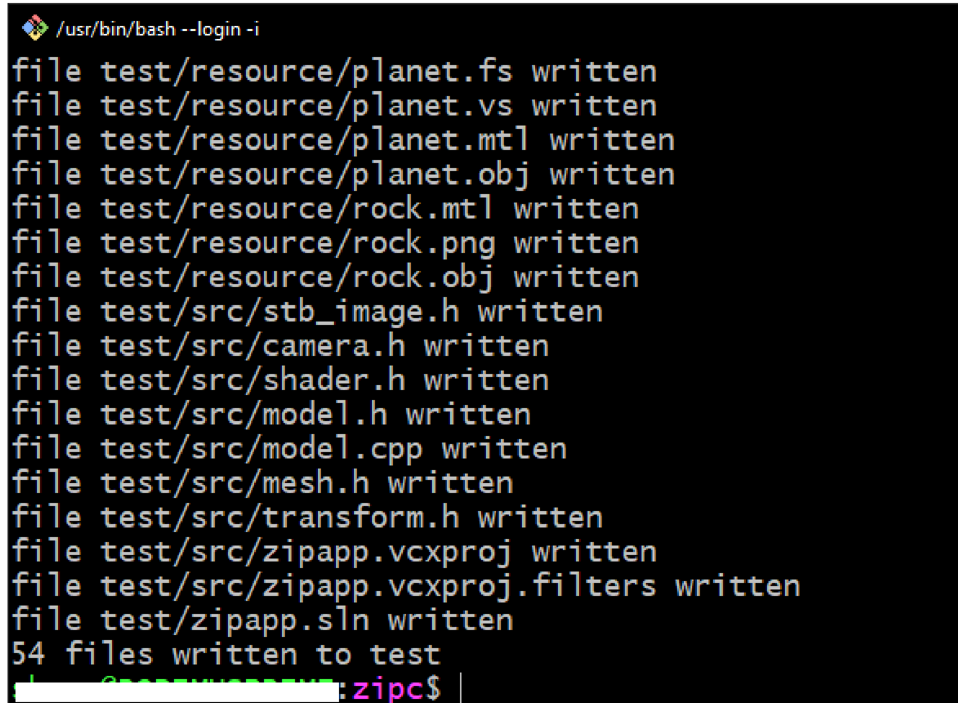

Figure 3.6: zipgen example

Zipgen also includes support for 'modules' that can import prebuilt zip ecsa systems and component code. Modules also have YAML config files and can chain dependencies and will automatically import other modules which they themselves are dependant on. Several reusable are part of zip and were even used in the benchmark results. For example, a glfw module that can be added as an option to the config file, using the modules option as shown in Figure 3.7:

```yaml
ecs:
    max_entities: 2000000

components:
    - name: Velocity
      value_type: glm::vec3

systems:
    - name: Velocity
      type: execute

modules: [glfwWindow]
```

Figure 3.7: Sample YAML config file with modules option

Adding the glfw module adds a blank opengl window and sets up all of opengl to allow rendering. Other modules include the bench-marking module that adds high precision frame-time bench-marking and a types module that imports glm types such as glm::vec3 and glm::mat4 that are used extensively in the result benchmarks.

3.3 zipapp

A zipapp is just an instance of the zip ecsa that has been built by zipgen. All of the benchmark programs used to create benchmarks in chapter 3 are zipapps created with zipgen.

3.4 goa test

Goa test or game objects architecture test is the name of the set of game object test programs used to benchmark against the ecsa zipapps. Goa test programs were created using the exact pattern and class definitions described in the introduction.

An example of a goa test class and base classes for one of the test programs:

```
struct Component {
    Component(GameObject *_go) : go(_go) {}
    virtual void Update(float dt) {}
    GameObject *go;
};

struct GameObject {
    virtual void AddComponent(Component *);
    virtual void Update(float dt);
    TransformComponent *transform;
    std::vector<Component*> components;
};

struct TransformComponent : public Component {
    TransformComponent(GameObject *_go);
    virtual void Update(float dt);
    glm::vec3 position;
    glm::mat4 rotation;
    glm::vec3 scale;
};

struct VelocityComponent : public Component {
    VelocityComponent(GameObject *_go);
    virtual void Update(float dt);
    glm::vec3 velocity;
};
```

Some constructors, destructors omitted for brevity.

Chapter 4

RESULTS

4.1 Performance Testing

Performance of frame-times were tested and recorded for the object-oriented game-objects architecture and the data-oriented entity-component-system architecture. At each benchmark setting, the frame-time was was computed from an average of samples. Each benchmark was run with increasing entity counts to generate curves as well as individual comparison graphs. The parameters for each test will be explained further in their respective sections. The goal of all the tests was to capture and compare the performance differences between the different architectures approaches.

4.1.1 Hardware

All tests were performed on a PC computer with Windows 10, using the following hardware:

1. AMD 5950X CPU
2. GIGABYTE X570 AORUS Master Motherboard
3. a Nvidia RTX 2080ti GPU
4. Corsair Vengeance RGB Pro SL 32GB (2x16GB) DDR4 3600 (PC4-28800) C18 1.35V Optimized for AMD Ryzen - Black (CMH32GX4M2Z3600C18)
5. Samsung 980 PRO SSD 2TB PCIe NVMe Gen 4 Gaming M.2 Internal Solid State Hard Drive

4.2 Asteroids Benchmark

In this test a set of asteroids were created and assigned a random size, rotation, and velocity. On each frame, their positions, rotations were simulated on the CPU and based on then the asteroids were rendered to the screen using the opengl graphics api. Instanced rendering was used so that all asteroids were drawn with a single draw call. The zip asteroids benchmark has 3 update functions and 4 components.

Figure 4.1 is displaying an example generated by the zip asteroids benchmark with $50k$ instanced asteroid meshes with random positions, random scale, random rotation angles and random velocities that are orbiting around a planet.

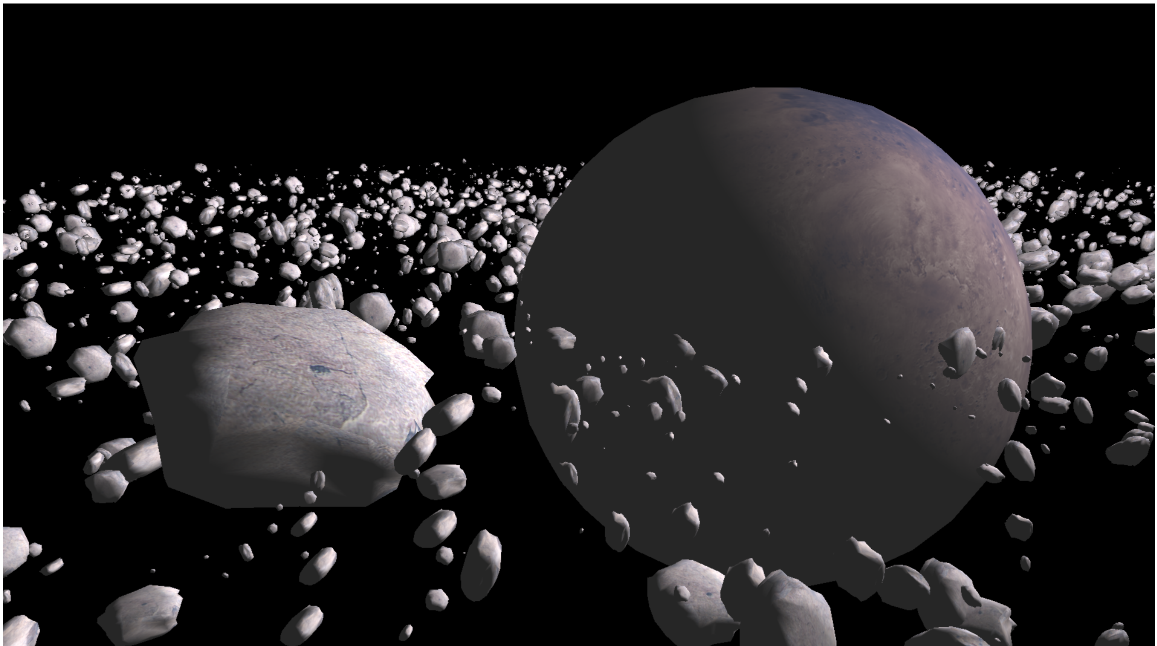

Figure 4.1: A close up of a planet and randomly generated asteroids.

Figure 4.2 shows a zoomed out image of $500k$ asteroids with with random positions, random scale, random rotation angles and random velocities, arrayed in a radial pattern around the planet. The asteroids are simulated each frame based on these initial random properties.

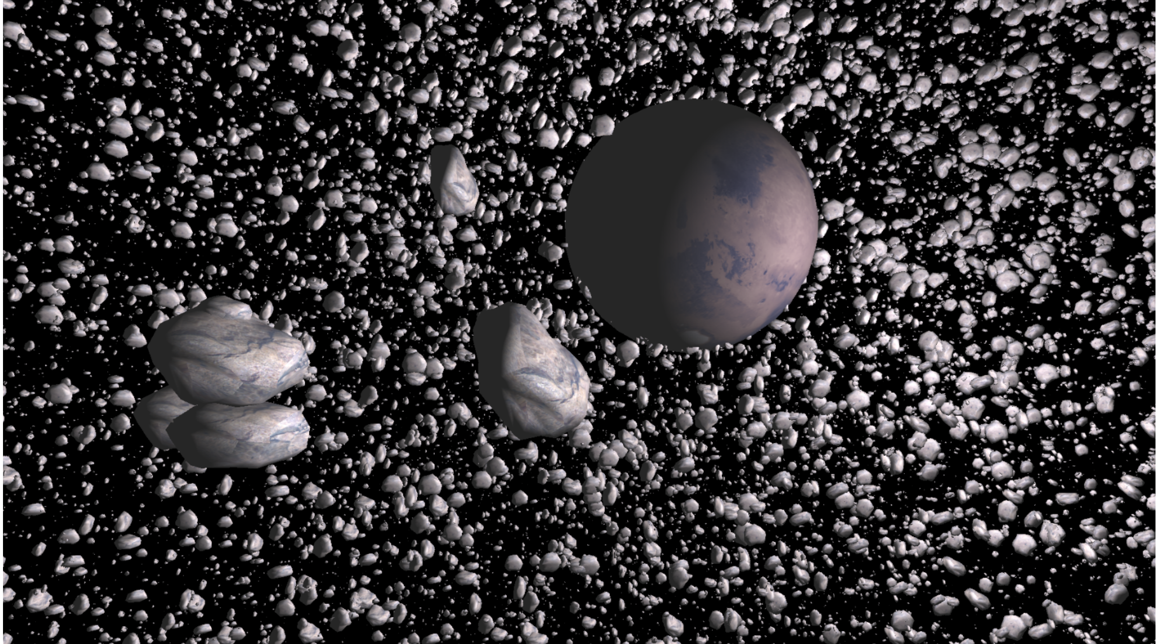

Figure 4.2: A zoomed out scene the planet and orbiting asteroids.

4.2.1 Asteroids Summarized Results

Figure 4.3 shows the plotted curves from Table 4.1 captured at over 50 different settings. Each point on the curve was created from an average of 1000 samples. Lower values are better and the graph shows that over time game objects curve slope is increasing and pulling away from the zip-ecsa curve, indicating that zip-ecsa will scale better for higher entity counts as expected. Frame-times are given in ms for entity totals ranging from 0 to $1e6$. The frame-time is the time to draw everything on the screen once and a lower-frame time is better and indicates better CPU cache performance. Lower frame-times mean higher frames per second, which is important for real time interactive video games and simulations.

The chart shows that there is no significant performance difference at or below $50k$ entities. The performance differences begin to emerge around $120k$ entities and the results rapidly diverge in favor of zip-ecsa for the remaining data-points. The results also show that zip-ecsa has a growing performance advantage as entity count goes up.

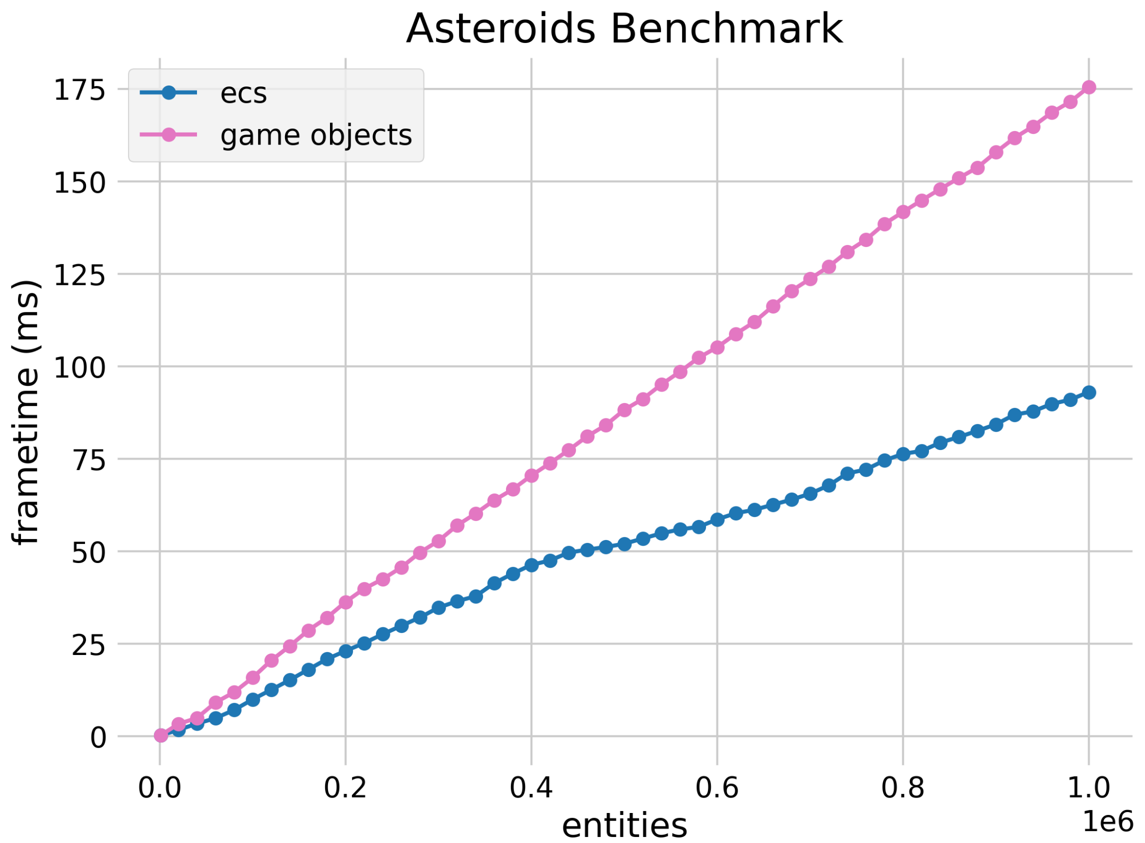

Figure 4.3: Rendered Asteroids Benchmark

Table 4.1: Asteroids Rendered Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+03	0.244544	0.234700
2.0E+04	1.736320	1.620400
4.0E+04	3.429650	3.207110
6.0E+04	4.913530	4.639920
8.0E+04	7.122280	6.280420
1.0E+05	9.986110	7.693620
1.2E+05	12.592700	9.154030
1.4E+05	15.230600	10.806800
1.6E+05	18.067200	12.152000
1.8E+05	20.894600	13.664800
2.0E+05	23.087400	15.606800
2.2E+05	25.209100	17.722800
2.4E+05	27.610000	18.522900
2.6E+05	29.875000	19.941300
2.8E+05	32.129400	21.607700

Table 4.1: Asteroids Rendered Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
3.0E+05	34.790300	22.994900
3.2E+05	36.508800	24.609000
3.4E+05	37.882700	26.550300
3.6E+05	41.393300	28.623400
3.8E+05	43.990700	29.738200
4.0E+05	46.342800	31.066200
4.2E+05	47.529500	32.464100
4.4E+05	49.642700	34.008800
4.6E+05	50.463600	35.838000
4.8E+05	51.154700	37.199900
5.0E+05	52.050900	38.707600
5.2E+05	53.466300	40.397800
5.4E+05	54.874600	41.939600
5.6E+05	55.932000	43.477100
5.8E+05	56.600800	45.101200

Table 4.1: Asteroids Rendered Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
6.0E+05	58.642000	46.677100
6.2E+05	60.312900	49.366300
6.4E+05	61.219600	50.451600
6.6E+05	62.616000	51.351300
6.8E+05	63.971400	53.800200
7.0E+05	65.580900	54.283500
7.2E+05	67.820000	55.833000
7.4E+05	71.068100	60.301900
7.6E+05	72.100100	60.349600
7.8E+05	74.554300	61.205300
8.0E+05	76.350400	63.308800
8.2E+05	77.127000	64.684200
8.4E+05	79.367300	66.306900
8.6E+05	80.892000	67.639500
8.8E+05	82.554900	69.654200

Table 4.1: Asteroids Rendered Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
9.0E+05	84.322000	71.180100
9.2E+05	86.932000	72.655400
9.4E+05	87.824600	74.246400
9.6E+05	89.846300	75.777400
9.8E+05	90.983700	77.072500
1.0E+06	93.053000	78.392300

4.2.2 Asteroids Rendering Disabled

Figure 4.4 shows the results from the previous Figure 4.3 by plotting Tables 4.1 and 4.2. The non-rendered results are overlaid onto the previous graph. These results were gathered by modifying the code to disable the rendering for the asteroids and represent CPU only performance. The game-objects architecture results show that the non-rendered results begin to diverge from the rendered results showing that the CPU is not performing fast enough to let the GPU render at its full speed.

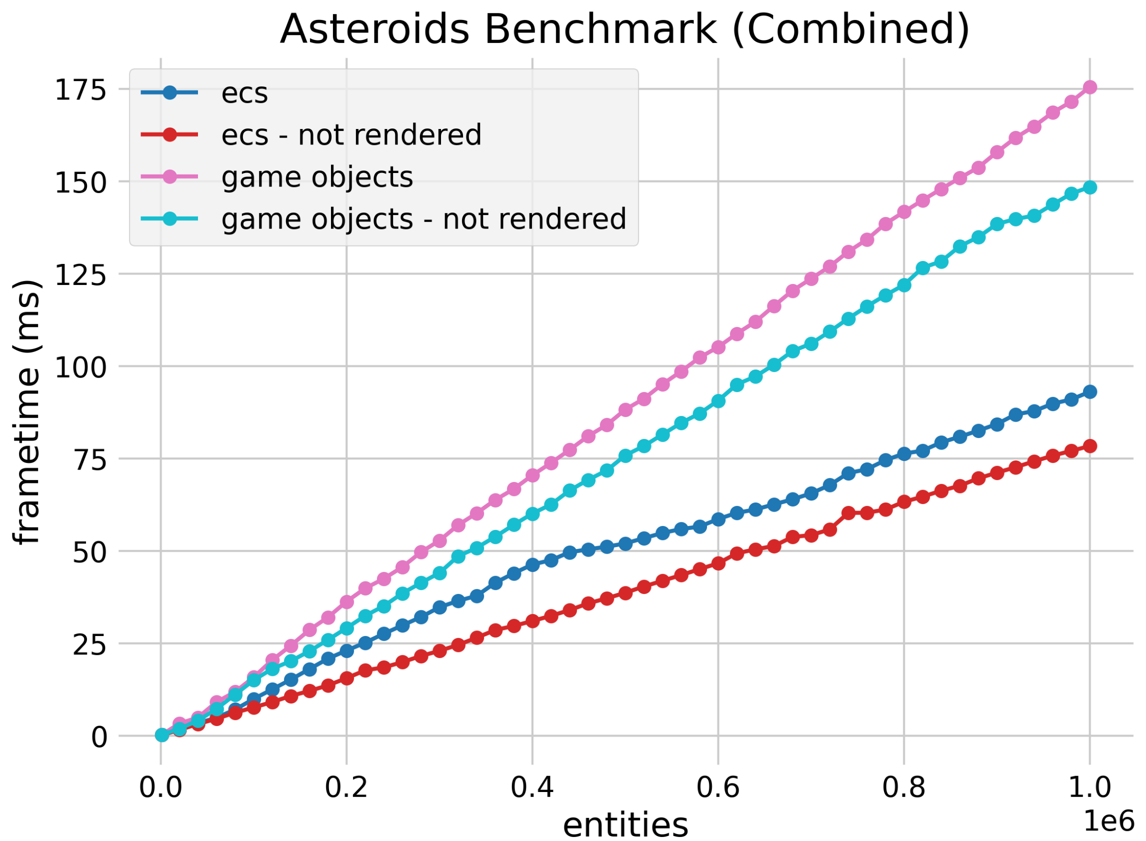

Figure 4.4: Not Rendered Asteroids Benchmark

The zip results show that the non-rendered results and rendered results have a similar slope to their curves, which means the CPU is performing at a level that does not hinder the GPU performance and is not diverging from it. The results show that zip has a clear performance advantage as entity count grows. As entity counts grow, goa test caused an increasing bottleneck to available GPU performance.

Table 4.2: Asteroids Data Not Rendered

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+03	0.233000	0.242000
5.0E+03	0.716000	0.820000
1.0E+04	0.892000	1.158000
2.0E+04	2.652000	2.243000
5.0E+04	4.379000	7.606000
1.0E+05	10.220000	20.920000
2.0E+05	17.690000	42.062000
3.0E+05	26.610000	62.886000
4.0E+05	35.850000	83.328000
5.0E+05	43.700000	102.912000
6.0E+05	52.280000	123.367000
7.0E+05	63.509000	143.260000

Table 4.2: Asteroids Data Not Rendered

entities	zip ecsa (ms)	game objects oop (ms)
8.0E+05	74.774000	164.063000
9.0E+05	79.883000	184.569000
1.0E+06	87.369000	202.423000
2.0E+06	168.651000	407.221000
5.0E+06	434.466000	1013.365000

4.2.3 Individual Asteroids Benchmarks

Frame-times were recorded the for entity counts ranging from $1e3$ to $10e6$ entities plotted from Table 4.3. Individual benchmarks are shown in Figures 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15. These are settings that are orders of magnitude in difference of entity count and they show the evolution of the performance changes.

Table 4.3: Asteroids Individual Benchmarks

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+03	0.235325	0.244544
1.0E+04	1.257210	1.684180
2.5E+04	2.955950	4.176110
5.0E+04	4.070500	7.589860
1.0E+05	9.874400	17.121500
2.5E+05	25.835100	45.485600
5.0E+05	45.010100	89.211200
1.0E+06	87.395500	177.428000
2.5E+06	215.952000	446.873000
5.0E+06	430.308000	891.846000

Table 4.3: Asteroids Individual Benchmarks

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+07	864.655000	1805.270000

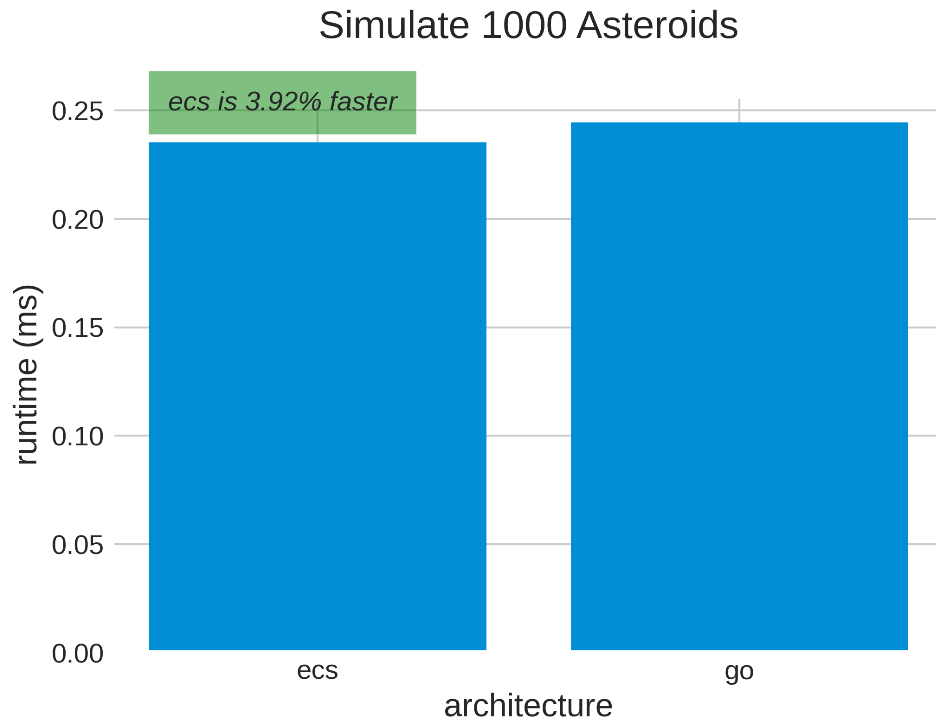

Figure 4.5: $1e3$ Asteroids zip ecsa vs game objects oop.

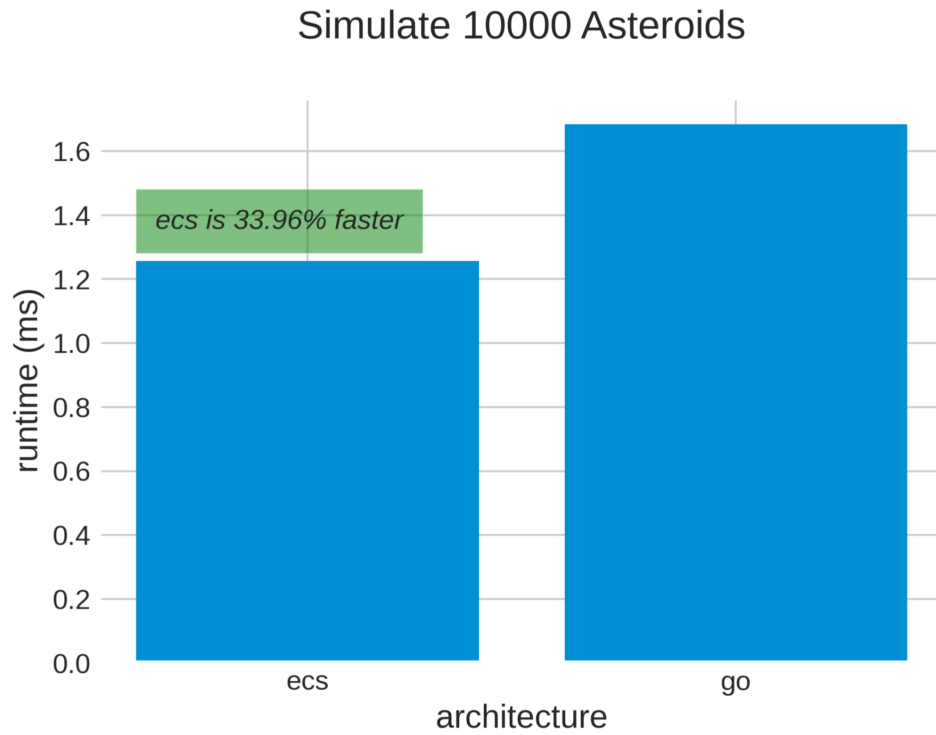

Figure 4.6: $1e4$ Asteroids zip ecsa vs game objects oop.

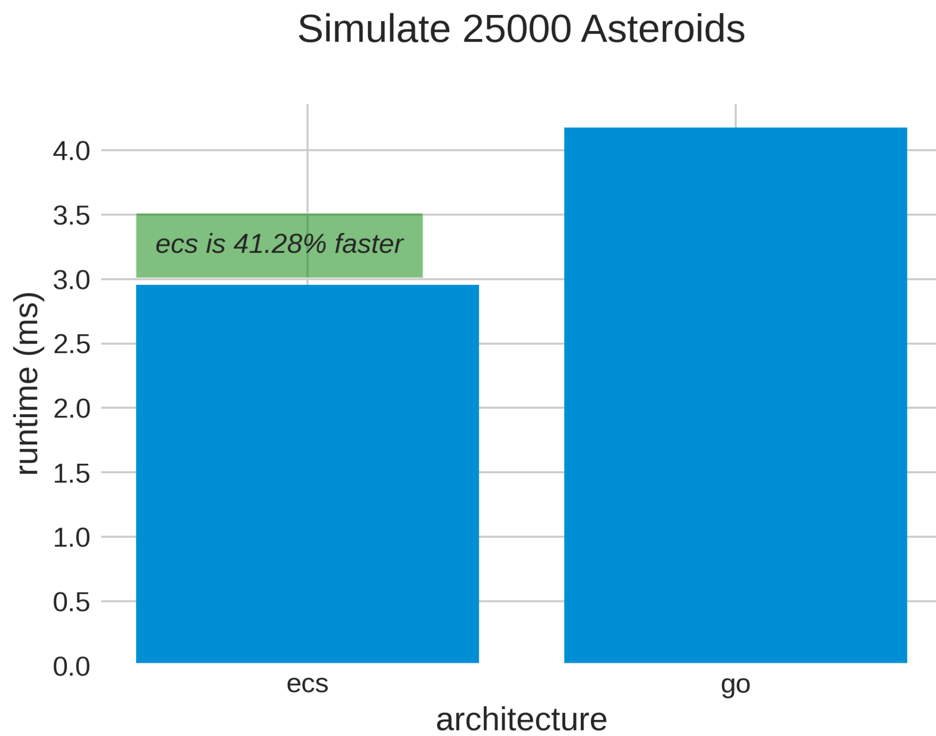

Figure 4.7: $2.5e4$ Asteroids zip ecsa vs game objects oop.

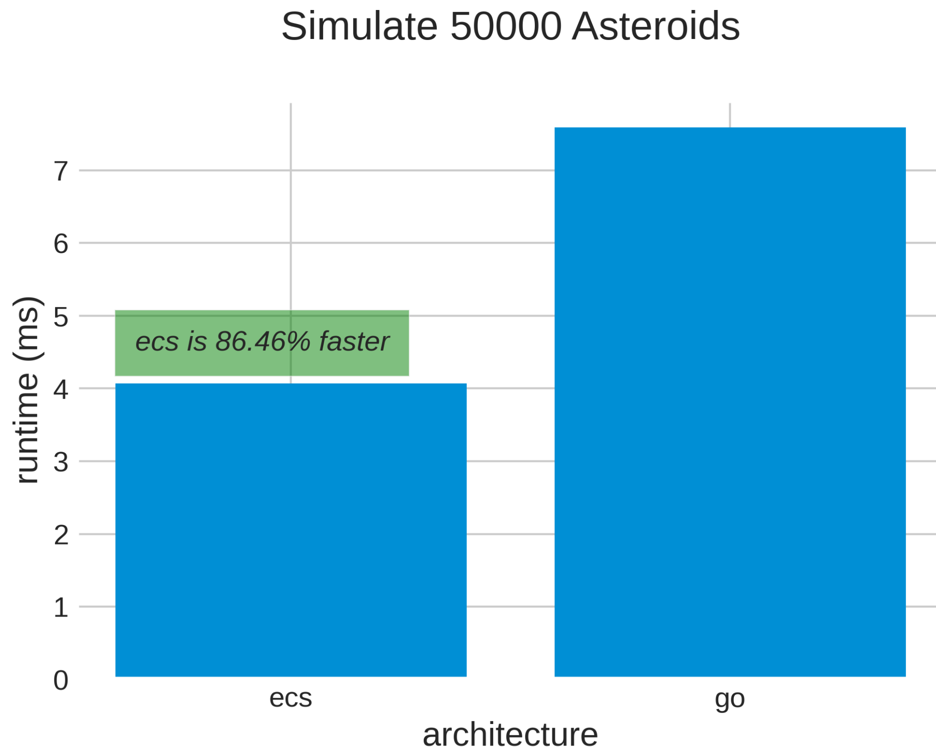

Figure 4.8: $5e4$ Asteroids zip ecsa vs game objects oop.

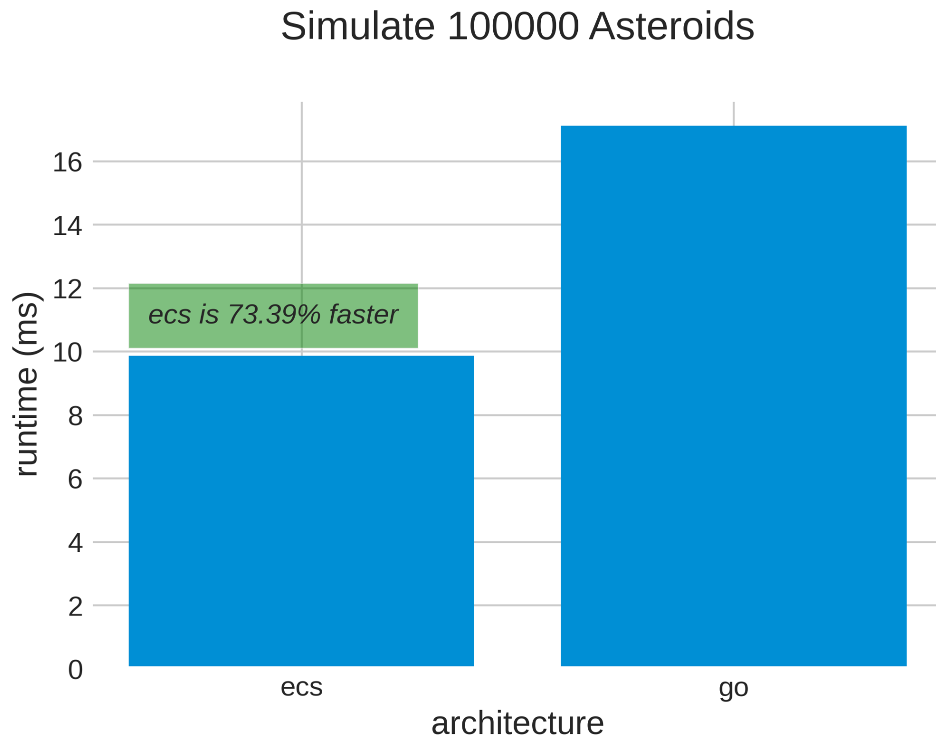

Figure 4.9: $1e5$ Asteroids zip ecsa vs game objects oop.

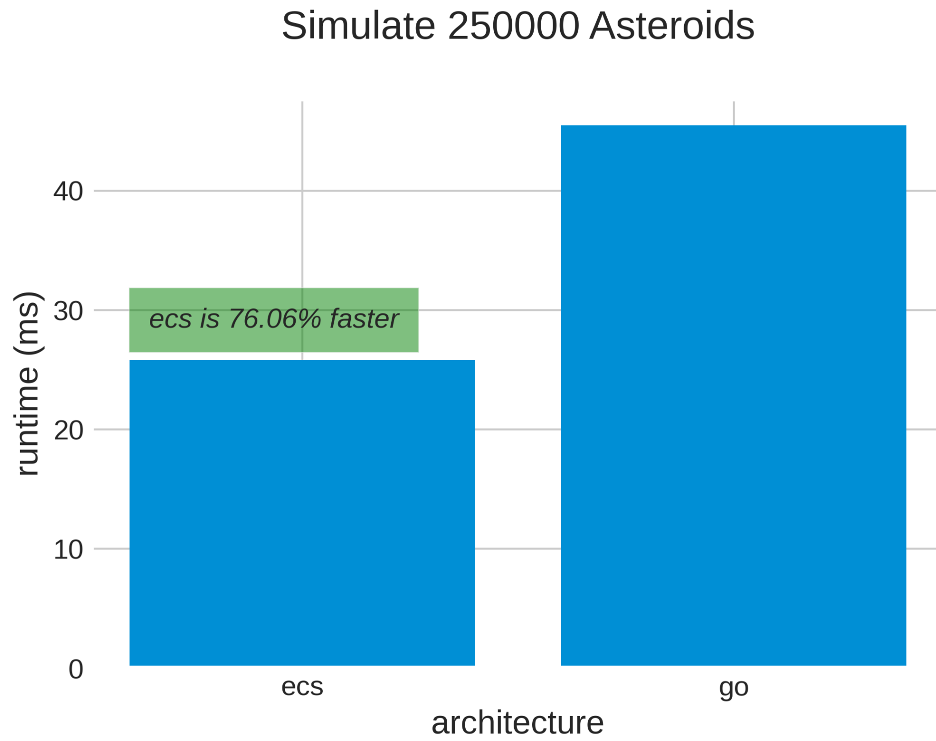

Figure 4.10: $2.5e5$ Asteroids zip ecsa vs game objects oop.

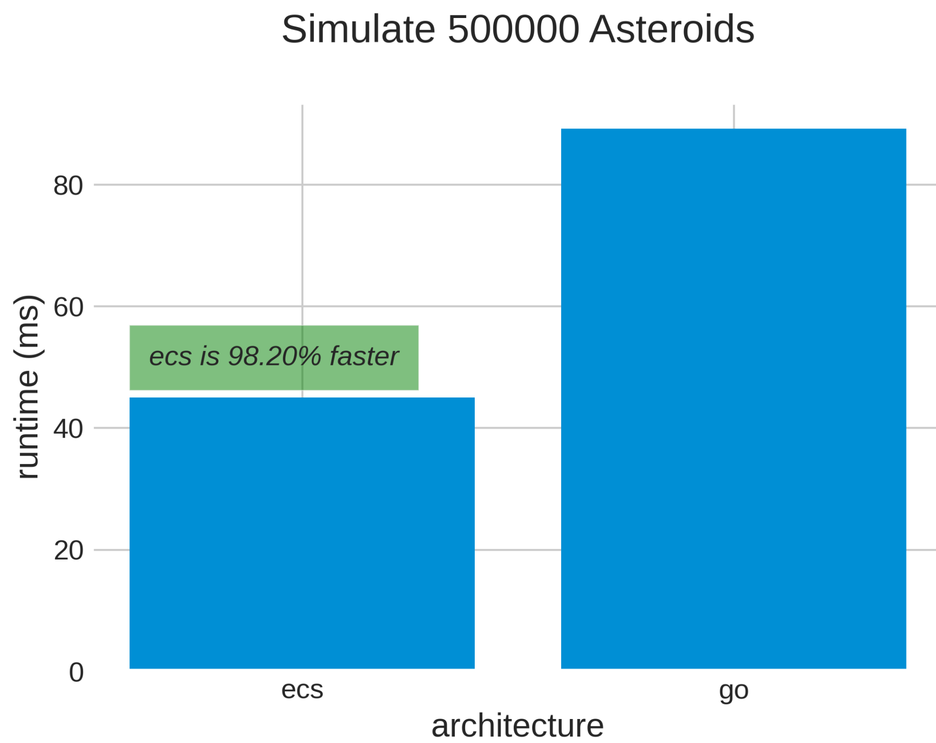

Figure 4.11: $5e5$ Asteroids zip ecsa vs game objects oop.

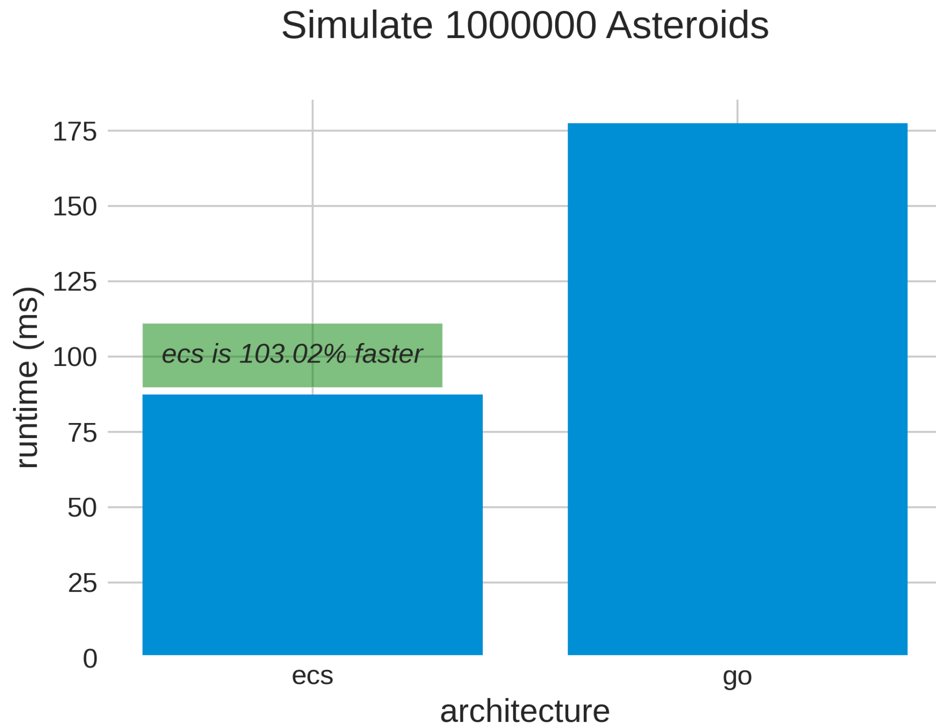

Figure 4.12: $1e6$ Asteroids zip ecsa vs game objects oop.

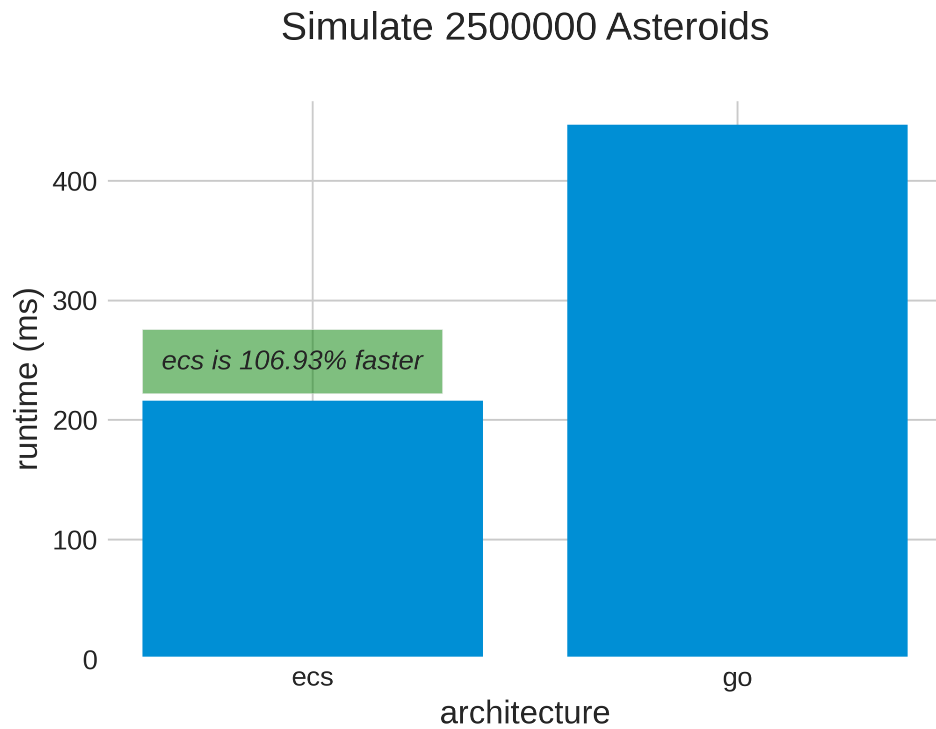

Figure 4.13: $2.5e6$ Asteroids zip ecsa vs game objects oop.

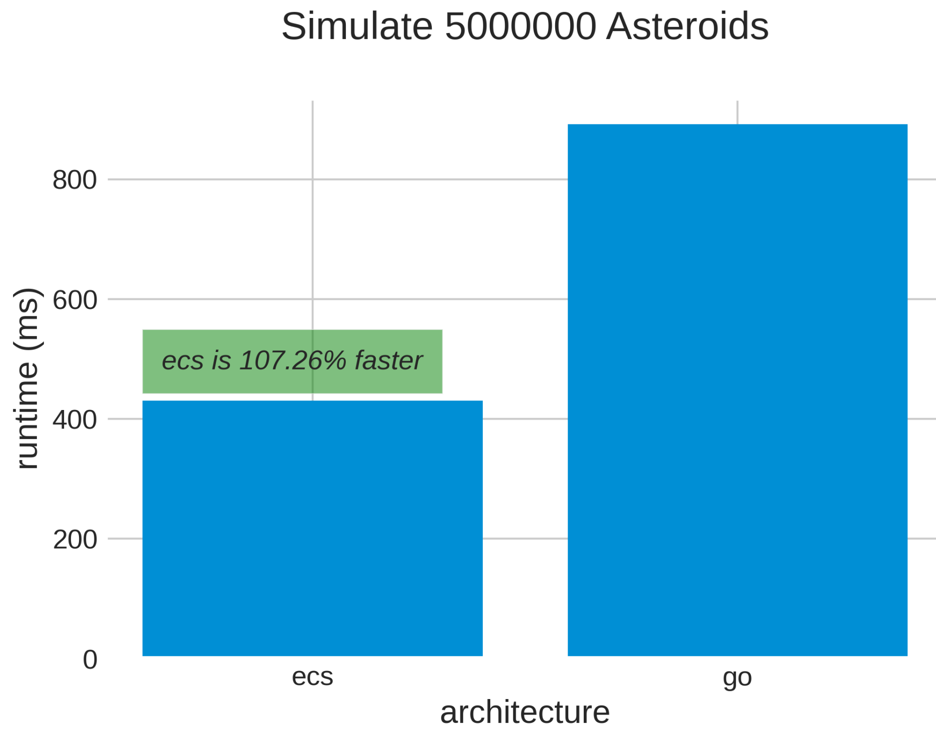

Figure 4.14: 5e6 Asteroids zip ecsa vs game objects oop.

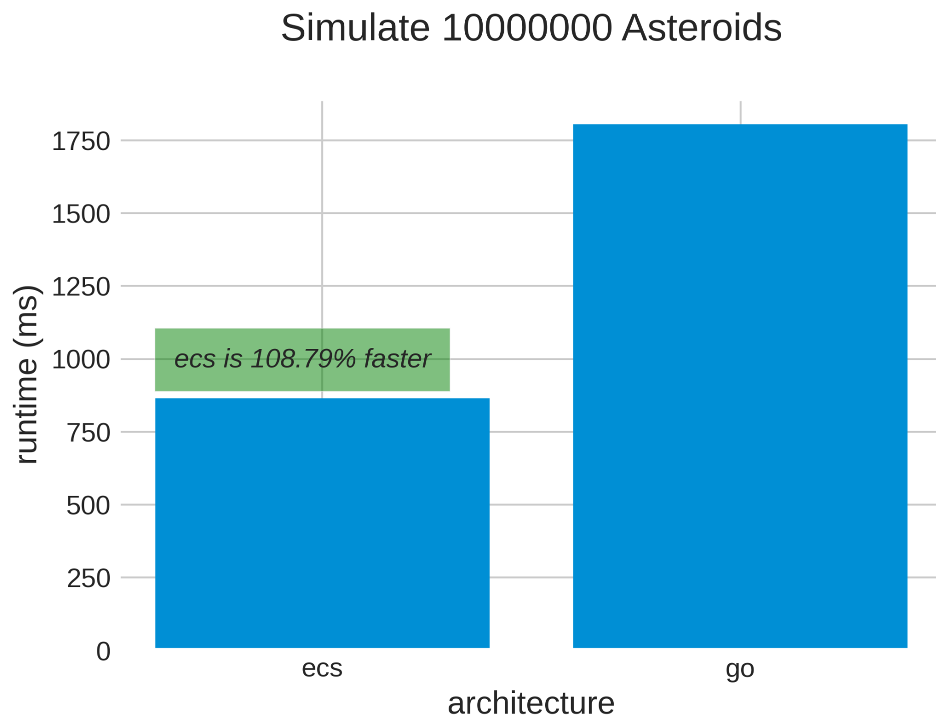

Figure 4.15: 10e6 Asteroids zip ecsa vs game objects oop.

4.2.4 Gravity O(n^2) Benchmark

In this test, a nested loop was run and each asteroid visited every other asteroid and added up simple total and then accumulated and stored that value into a component. The purpose of this test was to measure any cache performance difference between the architectures when using $O(n^2)$ complexity algorithms. The gravity benchmark consists of 4 update functions and 5 components.

4.2.4.1 Combined Benchmark

Figure 4.16 shows the results of plotting data from Table 4.4. The test applied the nested loop to every asteroid for the two architectures. In practical terms this means a nested loop where as each asteroid was processed, it then looped all asteroids again and added up a total based on their positions. The type of operation was kept relatively simple so that CPU-cache performance would be the main cause of performance degradation and not due to complex instructions performed for each visited asteroid.

It is worth noting that both architectures suffered performance degradation and for the combined benchmark the maximum entities used was $5k$ asteroids and at that count, zip ecsa was still performing at 60 frames per second but the game-objects architecture was not and fell to 15 frames per second at $5k$ asteroids which is not quite good enough for an interactive game. Although both suffered a performance decrease from adding the $O(n^2)$ operation, zip ecsa displayed a clear and significant performance advantage in measured frame-times.

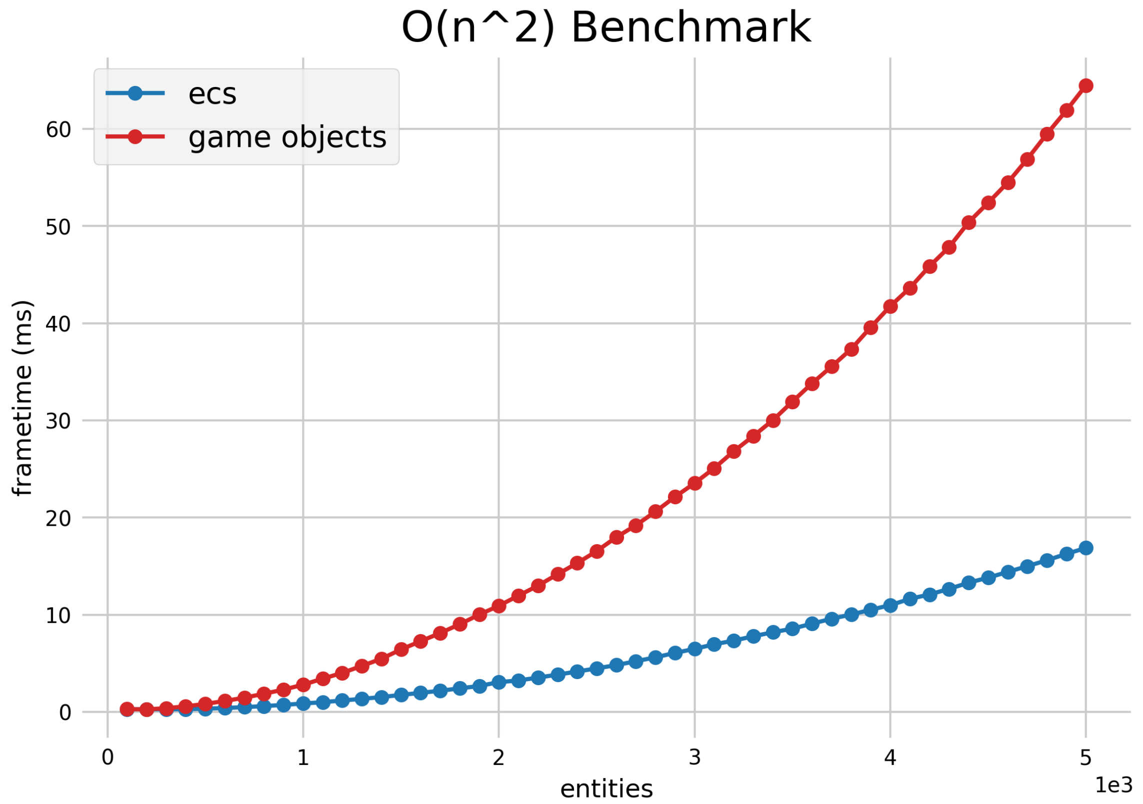

Figure 4.16: Gravity $O(n^2)$ Combined Benchmark

Table 4.4: Gravity $O(n^2)$ Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+02	0.263950	0.290261
2.0E+02	0.281250	0.263624
3.0E+02	0.270220	0.383806
4.0E+02	0.281640	0.572949
5.0E+02	0.337940	0.826125

Table 4.4: Gravity $O(n^2)$ Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
6.0E+02	0.400310	1.127010
7.0E+02	0.497120	1.465680
8.0E+02	0.598850	1.872050
9.0E+02	0.726720	2.296750
1.0E+03	0.864740	2.802760
1.1E+03	1.003410	3.421730
1.2E+03	1.176020	4.018960
1.3E+03	1.347040	4.736020
1.4E+03	1.528150	5.472040
1.5E+03	1.750300	6.427430
1.6E+03	1.963220	7.275630
1.7E+03	2.179350	8.112870
1.8E+03	2.420840	9.028960
1.9E+03	2.669230	10.018400
2.0E+03	3.066830	10.923400

Table 4.4: Gravity $O(n^2)$ Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
2.1E+03	3.234740	11.968400
2.2E+03	3.534870	12.994100
2.3E+03	3.838300	14.178200
2.4E+03	4.158470	15.315500
2.5E+03	4.482860	16.552800
2.6E+03	4.819530	17.984000
2.7E+03	5.217910	19.206900
2.8E+03	5.617380	20.616600
2.9E+03	6.074250	22.138200
3.0E+03	6.489130	23.542500
3.1E+03	6.958960	25.081200
3.2E+03	7.337990	26.824300
3.3E+03	7.785070	28.394400
3.4E+03	8.202750	30.023400
3.5E+03	8.569390	31.942900

Table 4.4: Gravity $O(n^2)$ Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
3.6E+03	9.086520	33.818100
3.7E+03	9.590650	35.549200
3.8E+03	10.023000	37.344500
3.9E+03	10.517700	39.590000
4.0E+03	10.988400	41.765200
4.1E+03	11.635100	43.636200
4.2E+03	12.087200	45.860500
4.3E+03	12.660600	47.828900
4.4E+03	13.301900	50.390200
4.5E+03	13.820600	52.402300
4.6E+03	14.408100	54.480400
4.7E+03	15.001100	56.898900
4.8E+03	15.604900	59.486500
4.9E+03	16.268300	61.909500
5.0E+03	16.874500	64.494000

4.2.4.2 Individual Benchmarks

The individual benchmarks show frame-times in ms from Table 4.5 for zip ecsa compared to goa test. Lower times are better. The percentage % label is automatically generated and indicates which architecture performed with a faster time and by how much.

The individual benchmarks use a lower maximum entity count than the previous asteroids benchmark because the frame-times accelerated to unacceptable levels even for bench-marking if the entity count went beyond $100k$ asteroids and began to get into the region of minutes per rendered frame for the object-oriented game-objects architecture.

Table 4.5: Gravity $O(n^2)$ Individual Benchmarks

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+02	0.256890	0.256120
5.0E+02	0.303340	0.785240
1.0E+03	0.828230	2.697810
5.0E+03	16.539700	63.974900
1.0E+04	64.492700	255.792000
5.0E+04	1594.830000	6312.490000
1.0E+05	6422.370000	25398.600000

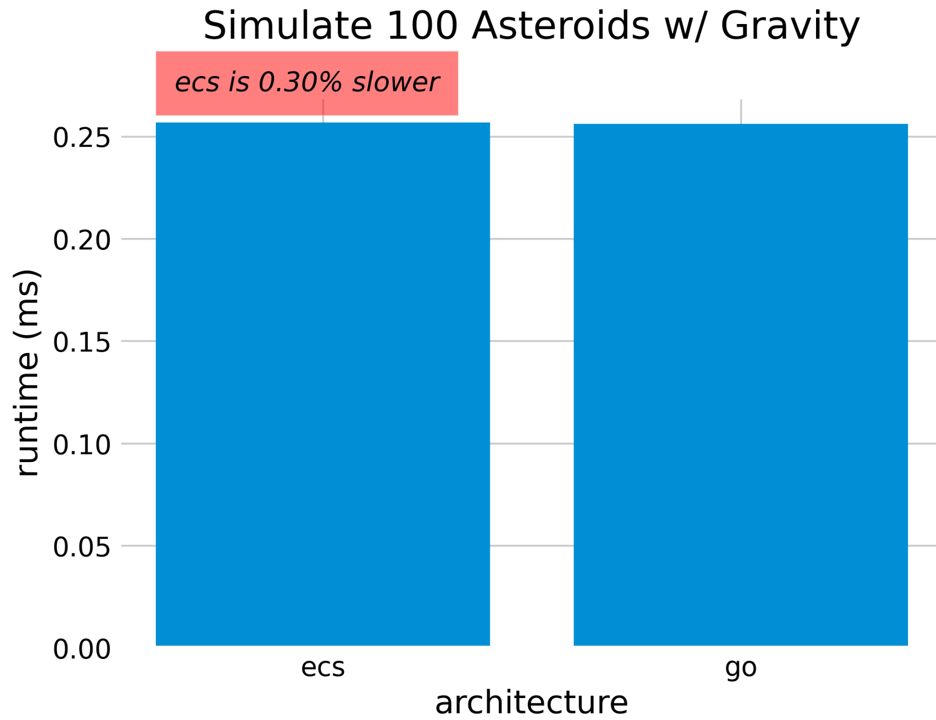

Figure 4.17: Gravity $O(n^2)$ Benchmark for 100 asteroids comparing the frame-time(s) in ms.

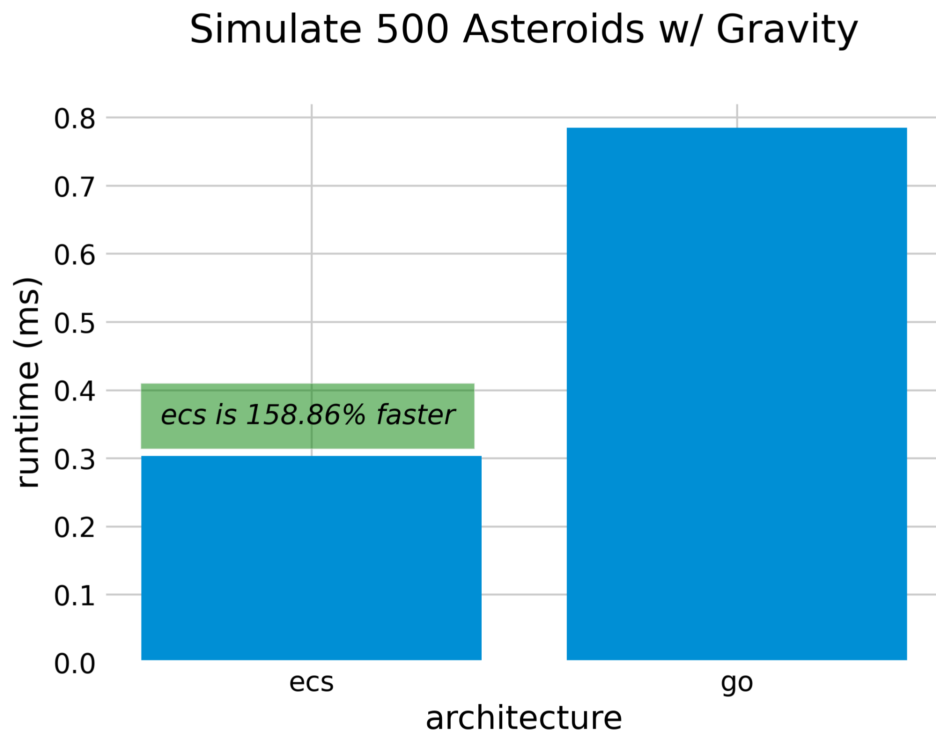

Figure 4.18: Gravity $O(n^2)$ Benchmark for 500 asteroids comparing the frame-time(s) in ms.

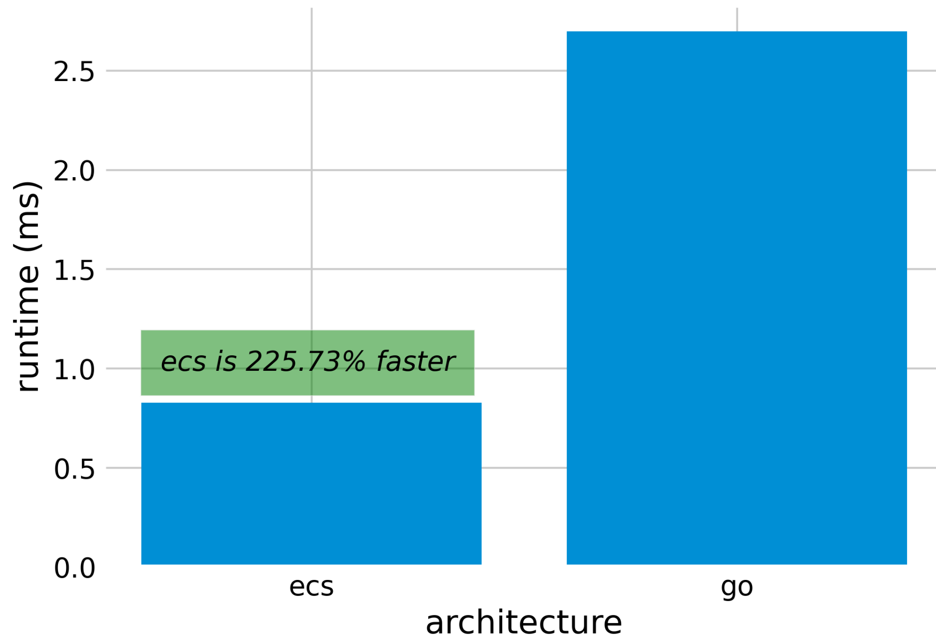

Figure 4.19: $O(n^2)$ Benchmark for $1k$ asteroids. Lower is better.

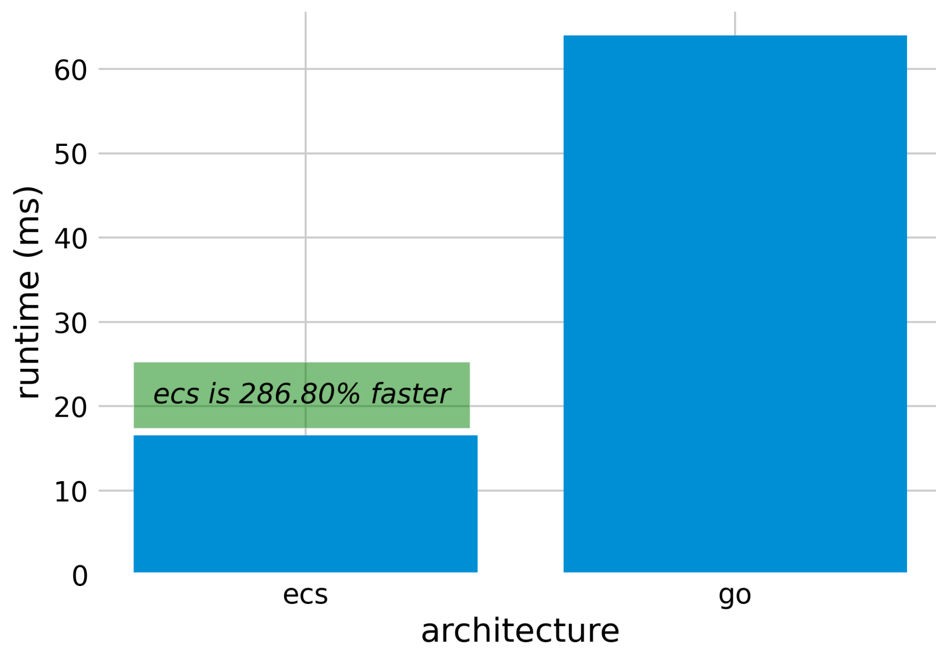

Figure 4.20: Gravity $O(n^2)$ Benchmark for $5k$ asteroids comparing the frame-time(s) in ms.

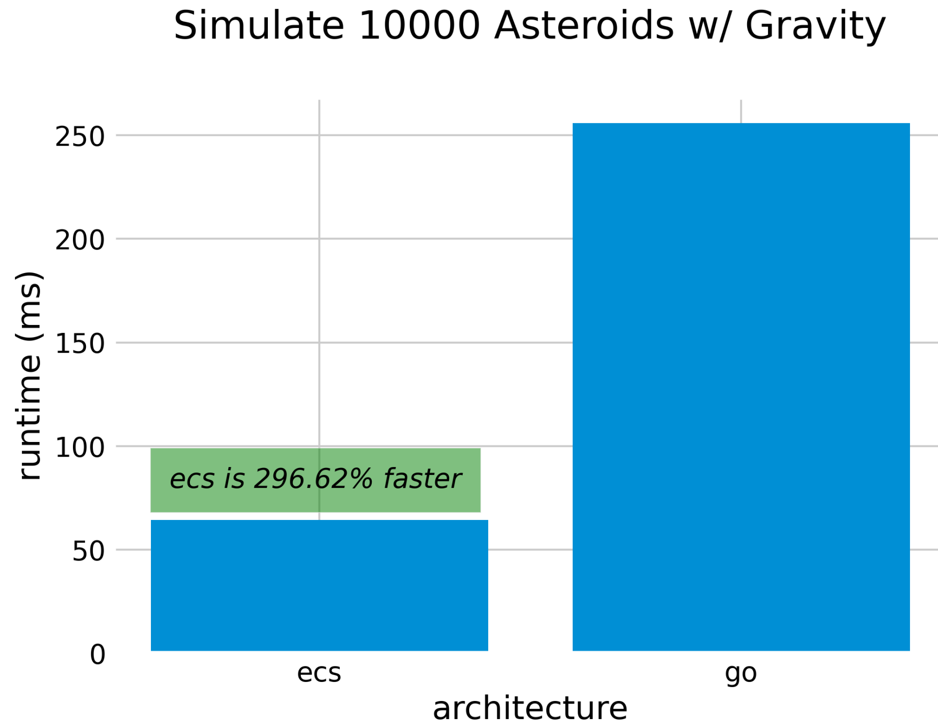

Figure 4.21: Gravity $O(n^2)$ Benchmark for $10k$ asteroids comparing the frame-time(s) in ms.

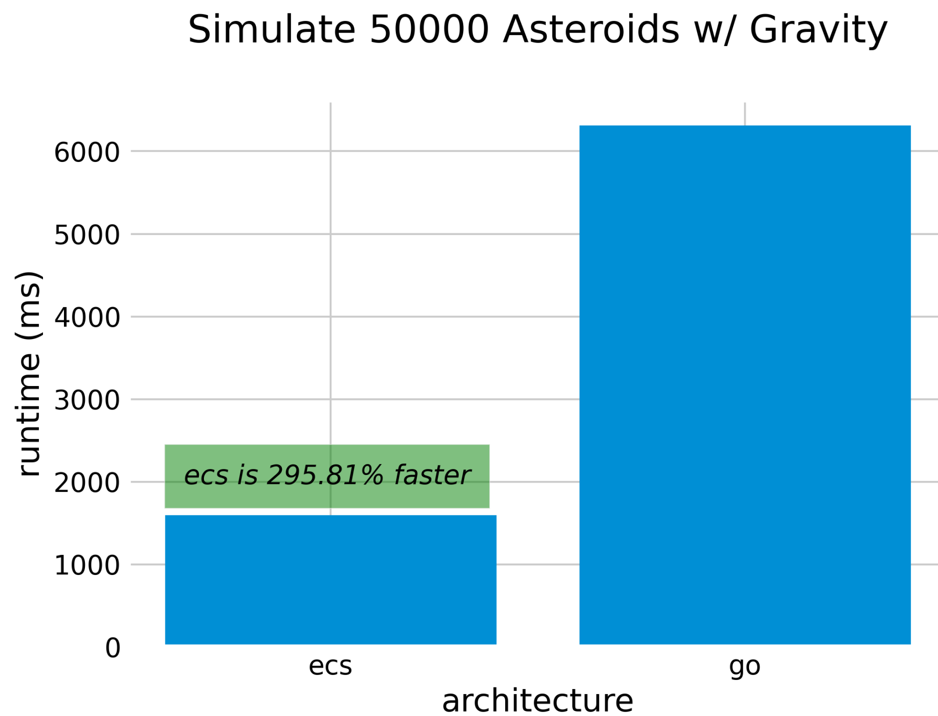

Figure 4.22: Gravity $O(n^2)$ Benchmark for $50k$ asteroids comparing the frame-time(s) in ms.

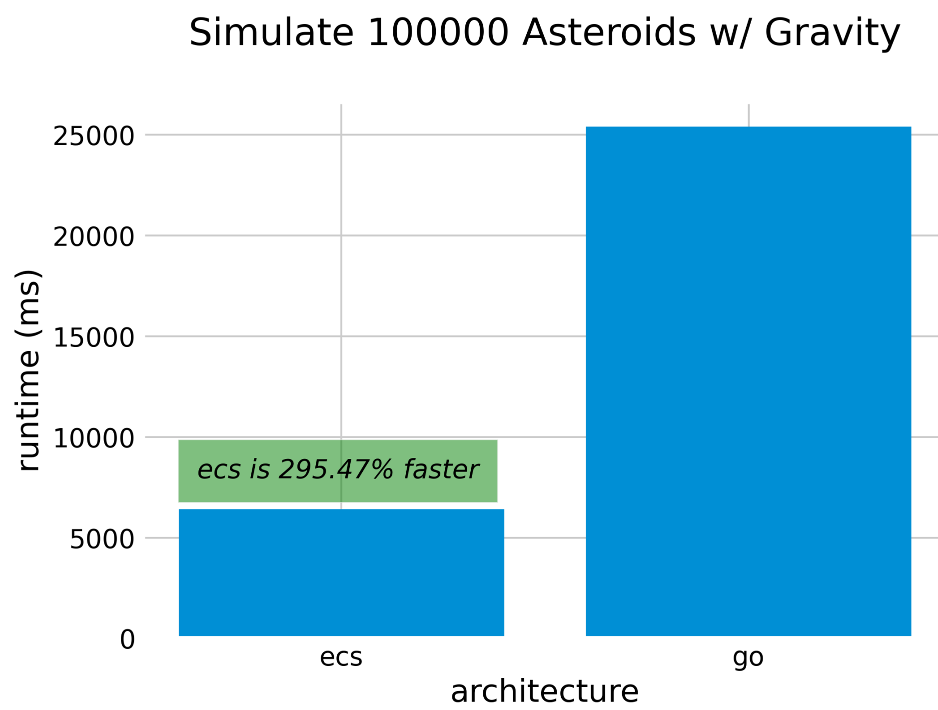

Figure 4.23: Gravity $O(n^2)$ Benchmark for $100k$ asteroids

4.3 Update x1 System Benchmark

This benchmark ran a CPU only update of a single update function with two components. The components are position and velocity, which are both vec3s and are 3 floats with a size of 12 bytes each. The update system updated the position based on the velocity and delta time each call. There was no rendering in this benchmark and the focus is on pure CPU performance and relevant to servers that simulate entities with large game worlds that would need to process thousands or millions of entities.

Example component:

```
struct PositionComponent
{
    float x;
    float y;
    float z;
};
```

4.3.1 Combined Results

The combined results show individual tests plotted as curves for both the data-oriented zip ecsa and goa test. The y axis shows run-time in ms and lower is better. Each test result was averaged from 100 trials at the respective setting. The curve for game-objects is so steep that zip ecsa curve almost looks like a flat line in comparison. The values plotted from data in Table 4.6.

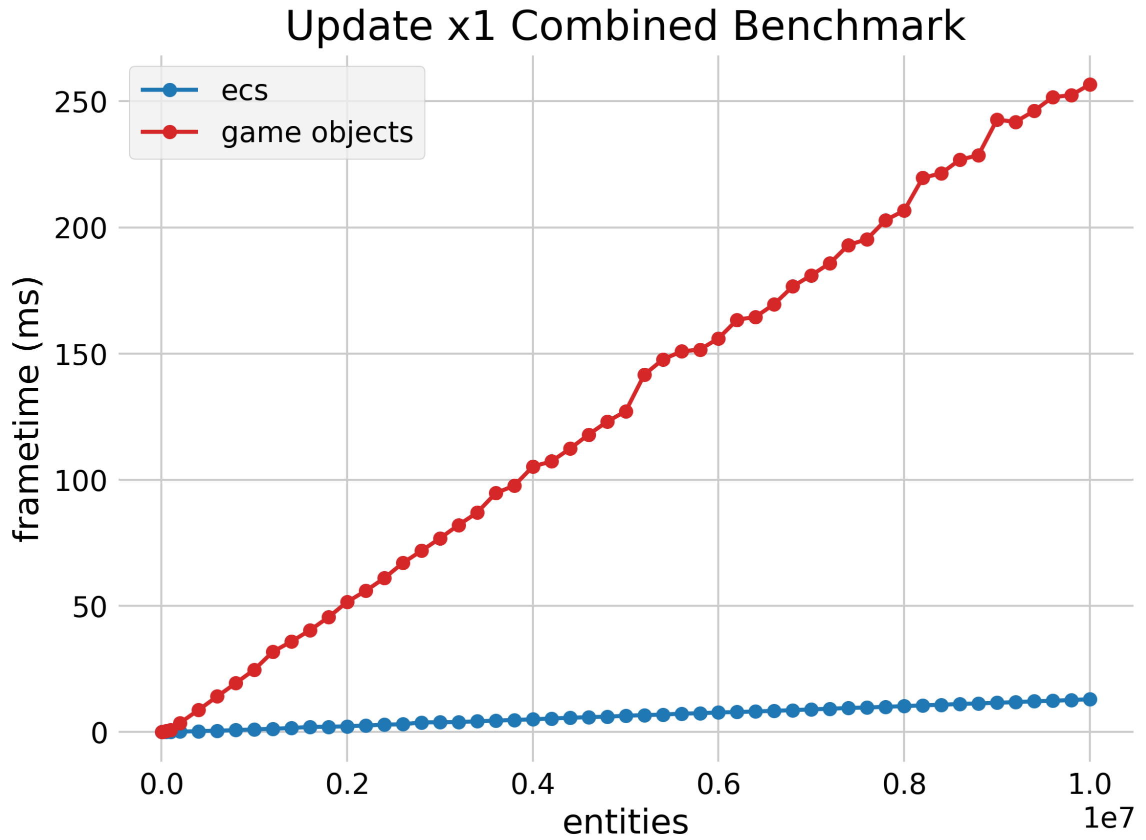

Figure 4.24: Combined Update x1 Benchmark.

Table 4.6: Update x1 Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+03	0.001600	0.005600
1.0E+04	0.019300	0.081900
5.0E+04	0.080700	0.396400
1.0E+05	0.159800	0.869700
2.0E+05	0.197740	3.672300

Table 4.6: Update x1 Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
4.0E+05	0.379300	8.812800
6.0E+05	0.555830	14.148200
8.0E+05	0.779990	19.414100
1.0E+06	1.072680	24.765400
1.2E+06	1.304650	31.863700
1.4E+06	1.652970	35.911500
1.6E+06	2.048380	40.395300
1.8E+06	2.092240	45.585500
2.0E+06	2.320940	51.610800
2.2E+06	2.593760	55.997300
2.4E+06	2.948610	61.014100
2.6E+06	3.124950	67.047900
2.8E+06	3.843160	71.844000
3.0E+06	3.914960	76.820800
3.2E+06	3.973800	82.089900

Table 4.6: Update x1 Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
3.4E+06	4.275340	86.996200
3.6E+06	4.537840	94.692000
3.8E+06	4.825090	97.679400
4.0E+06	5.053870	105.250000
4.2E+06	5.384280	107.293000
4.4E+06	5.678140	112.315000
4.6E+06	5.914630	117.906000
4.8E+06	6.111090	123.023000
5.0E+06	6.468350	127.156000
5.2E+06	6.727520	141.622000
5.4E+06	6.938540	147.647000
5.6E+06	7.309170	150.857000
5.8E+06	7.473480	151.591000
6.0E+06	7.795460	156.046000
6.2E+06	7.941150	163.373000

Table 4.6: Update x1 Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
6.4E+06	8.183430	164.573000
6.6E+06	8.425410	169.615000
6.8E+06	8.699960	176.619000
7.0E+06	8.999300	181.054000
7.2E+06	9.247990	185.819000
7.4E+06	9.579510	193.031000
7.6E+06	9.731730	195.357000
7.8E+06	10.032000	202.889000
8.0E+06	10.283000	206.675000
8.2E+06	10.600700	219.640000
8.4E+06	10.829600	221.461000
8.6E+06	11.164100	226.826000
8.8E+06	11.360800	228.621000
9.0E+06	11.618900	242.682000
9.2E+06	11.887000	241.843000

Table 4.6: Update x1 Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
9.4E+06	12.201300	246.175000
9.6E+06	12.399200	251.641000
9.8E+06	12.729100	252.396000
1.0E+07	13.028000	256.675000

4.3.2 Individual Benchmarks

The following individual benchmarks show the time for both the entity-component system and game-objects architectures to perform a single update with one system. Individual graphs are plotted from Table 4.7. The hypothesis was that the results would be similar to asteroids benchmark.

Table 4.7: Update x1 Individual Benchmarks

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+03	0.001600	0.005400
1.0E+04	0.018300	0.067200
2.5E+04	0.042500	0.225200
5.0E+04	0.085200	0.390300
1.0E+05	0.163100	0.853400
2.5E+05	0.380900	4.379000
5.0E+05	0.598700	11.381400
1.0E+06	1.299300	24.816800
2.5E+06	3.648100	64.257100
5.0E+06	7.361600	128.027000
1.0E+07	14.722200	258.056000

4.3.3 Individual Results Analysis

Zip ecsa performance is faster than expected. The ecsa was significantly faster even for $1k$ entities which shows that the it is useful even for smaller games and simulations and this was unexpected. It was not expected to not pull in performance until higher numbers of entities were tested. The performance lead levels and stays consistent at around $1M$ entities with a 1650% increase over goa test from that point on. Something interesting is that it took less than $5ms$ to update $1M$ entities and that means a game using this architecture could possibly simulate (depending on the game systems) a very high number of entities and still maintain 60 frames per second or higher as long as it was not trying to render too many entities at once. Rendering thousands or millions of entities is a different class of problem and depends heavily on the GPU hardware and graphics code used.

Many of the run-times recorded showed sub $1ms$ timings for both architectures which is so fast that it may not matter that one was faster.

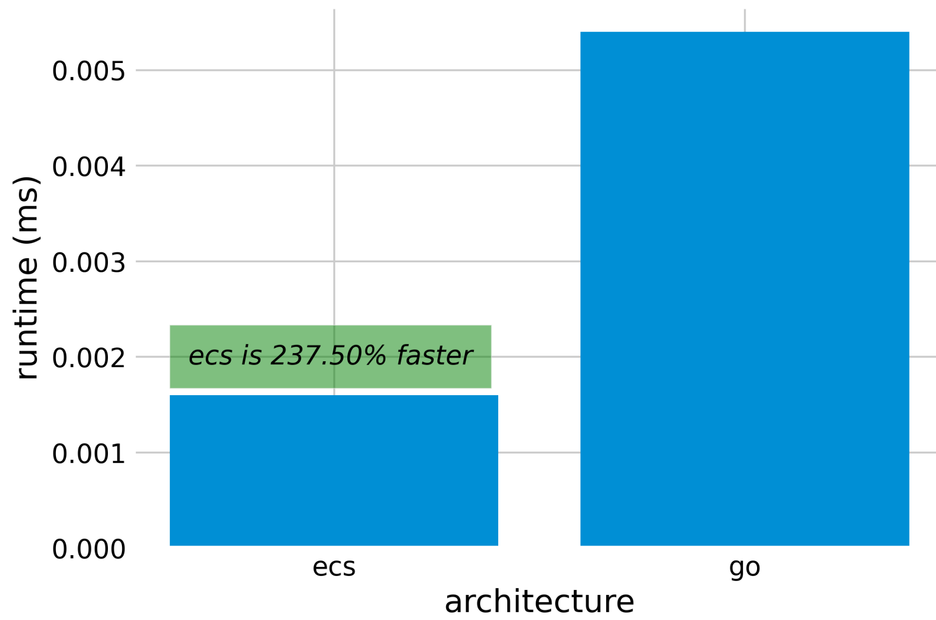

Figure 4.25: Update x1 Benchmark 1k entities.

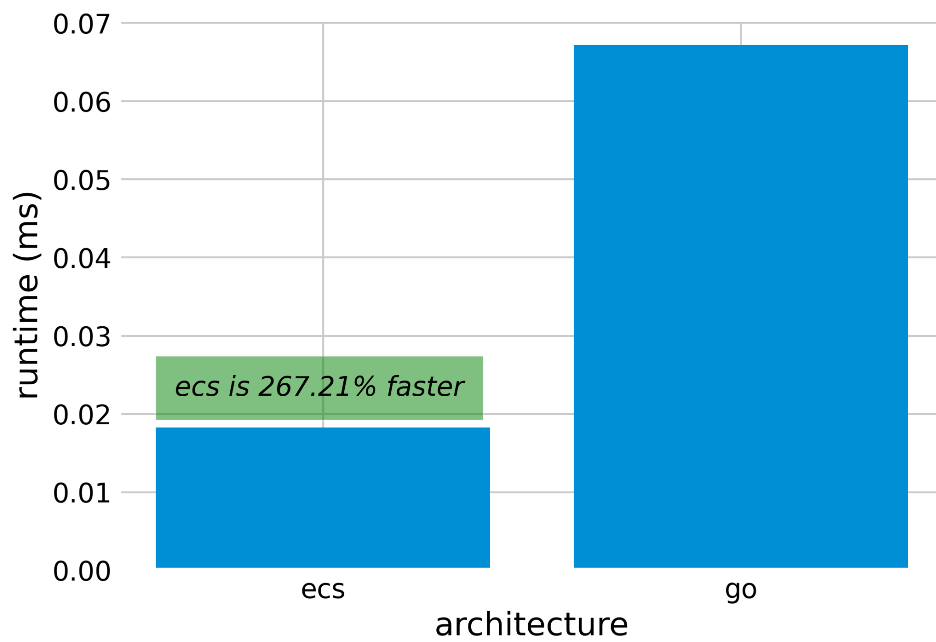

Figure 4.26: Update x1 Benchmark 10k entities.

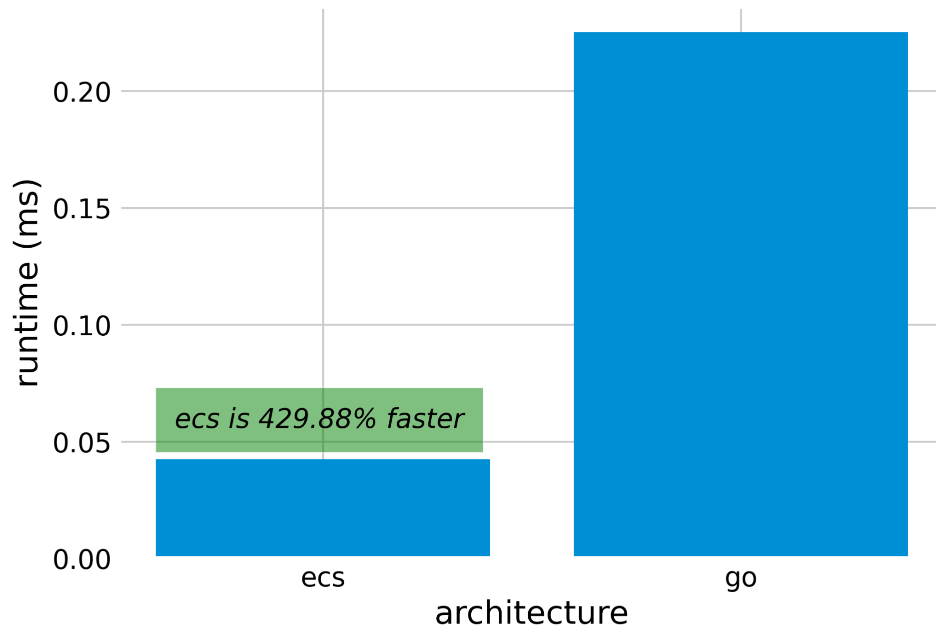

Figure 4.27: Update x1 Benchmark 25k entities.

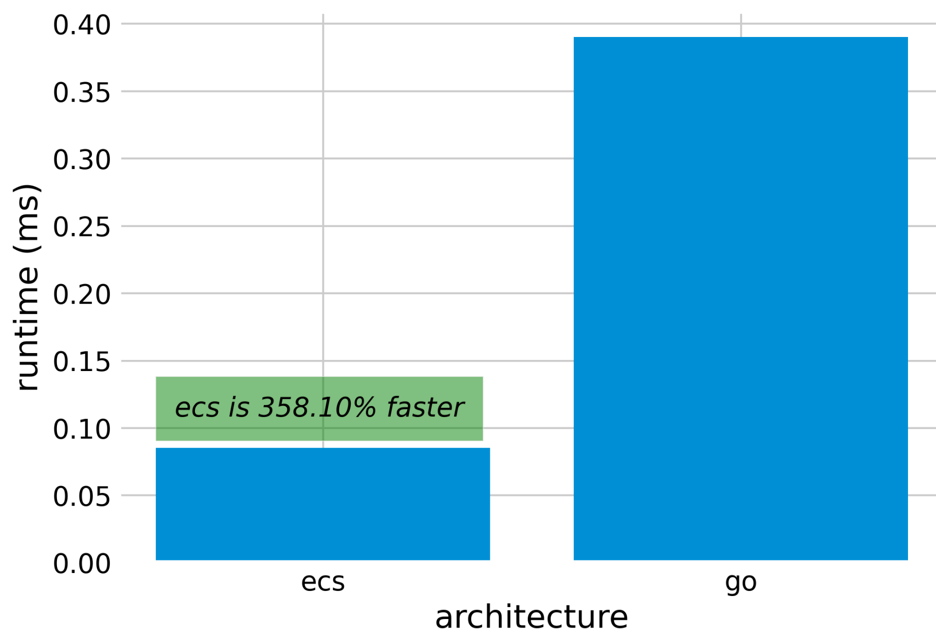

Figure 4.28: Update x1 Benchmark 50k entities.

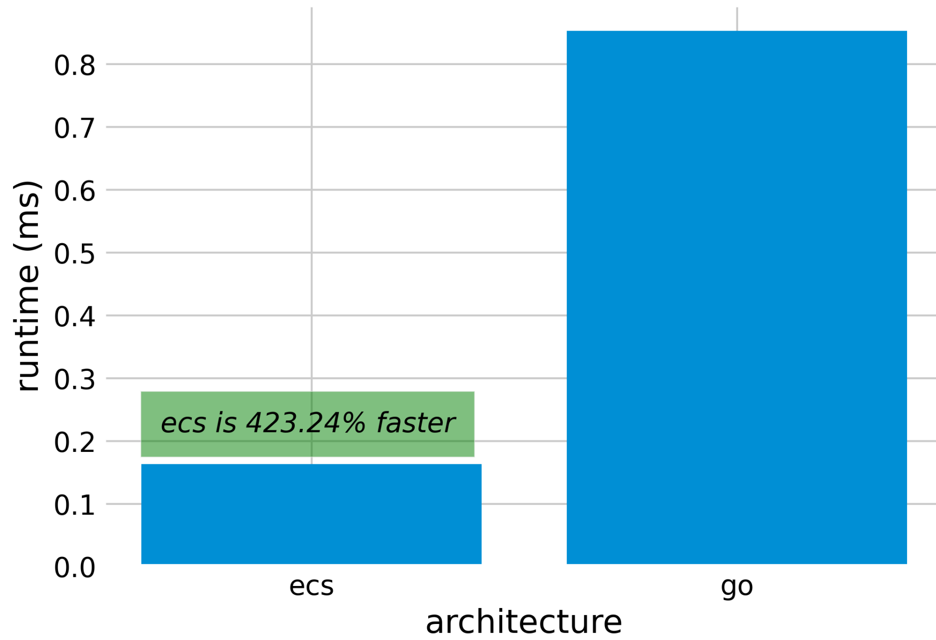

Figure 4.29: Update x1 Benchmark 100k entities.

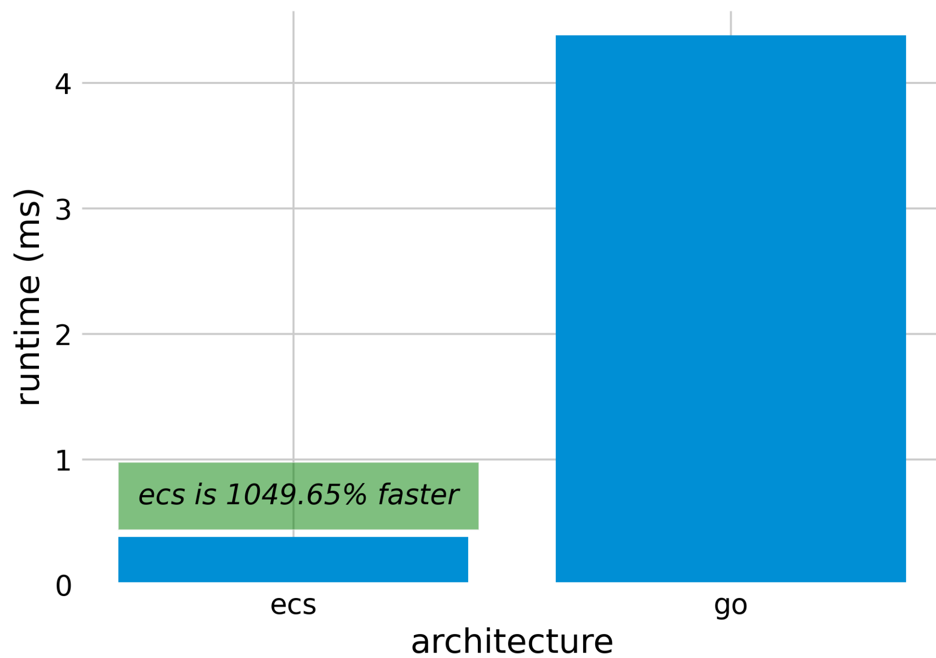

Figure 4.30: Update x1 Benchmark 250k entities.

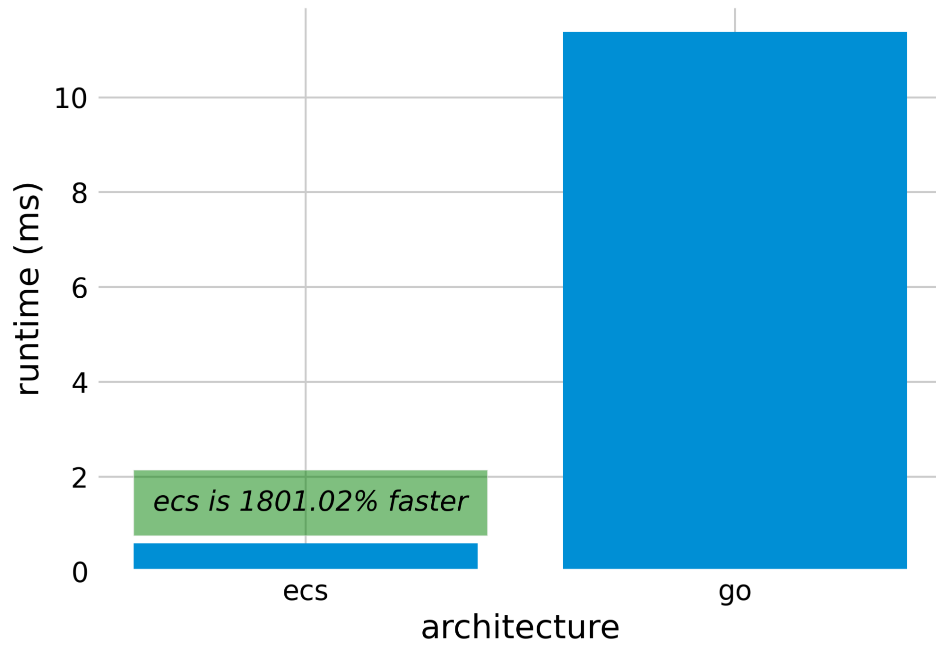

Figure 4.31: Update x1 Benchmark 500k entities.

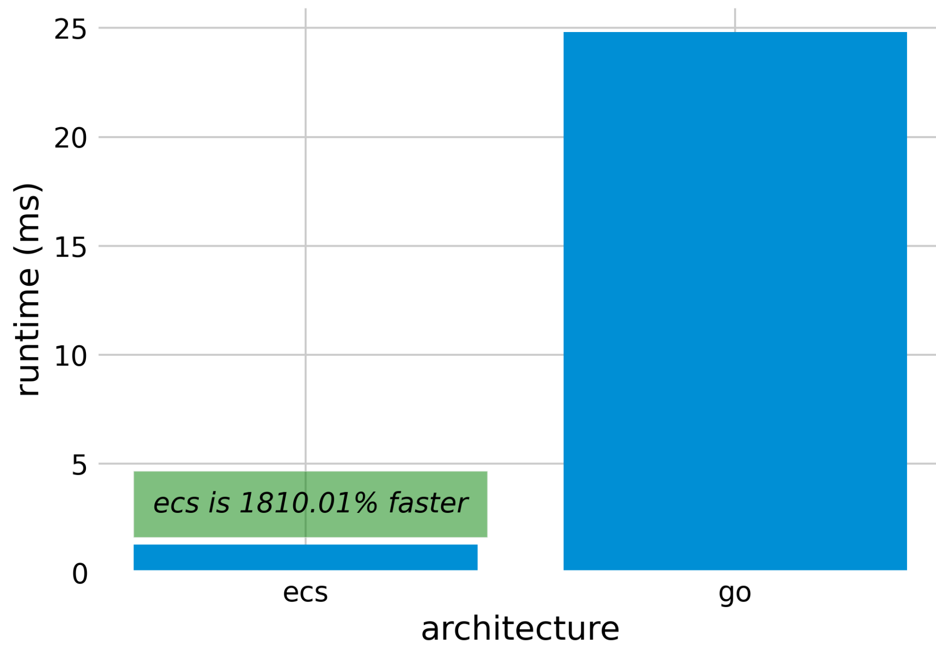

Figure 4.32: Update x1 Benchmark 1M entities.

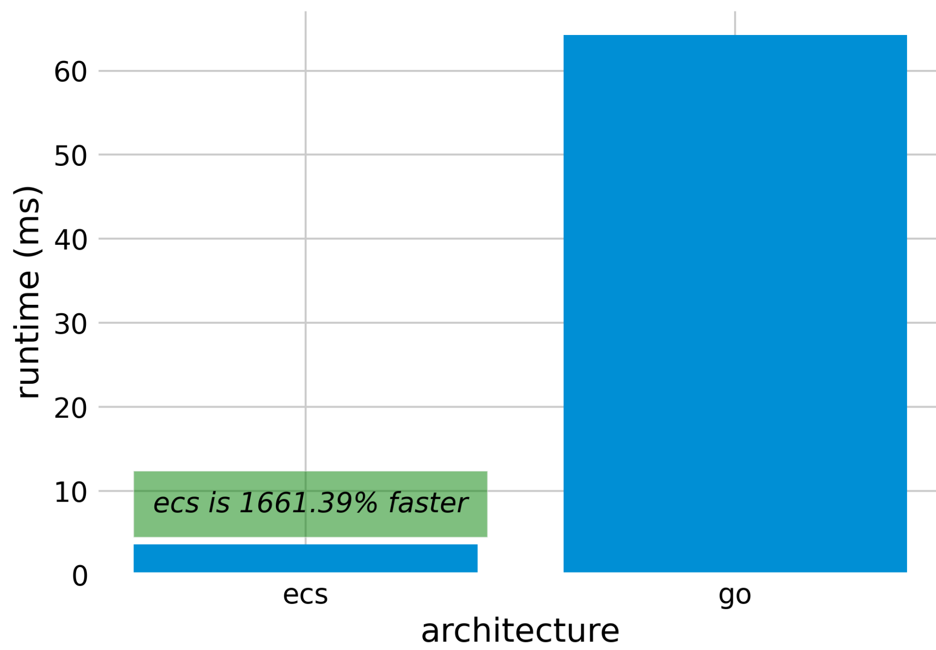

Figure 4.33: Update x1 Benchmark 2.5M entities.

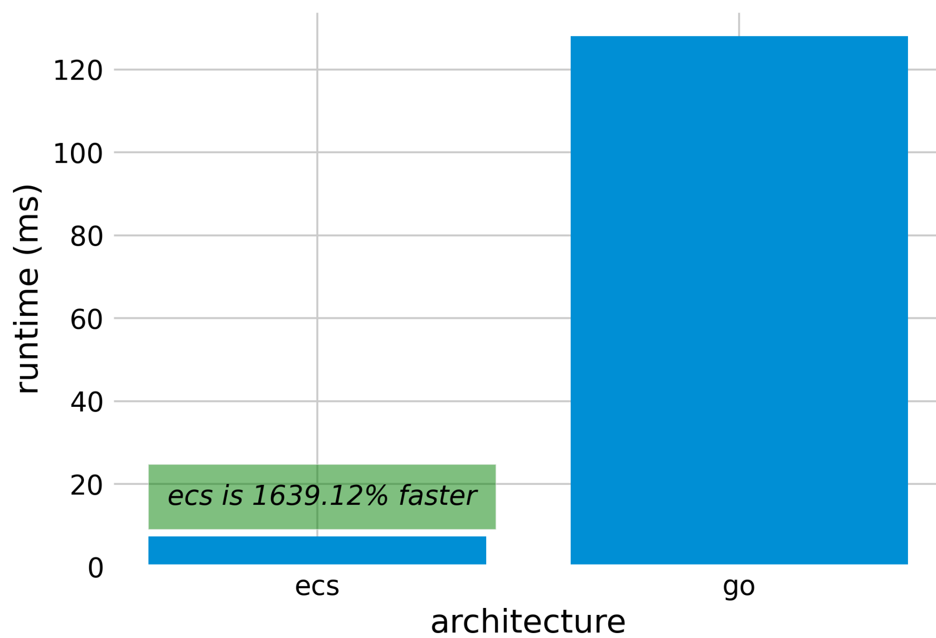

Figure 4.34: Update x1 Benchmark 5M entities.

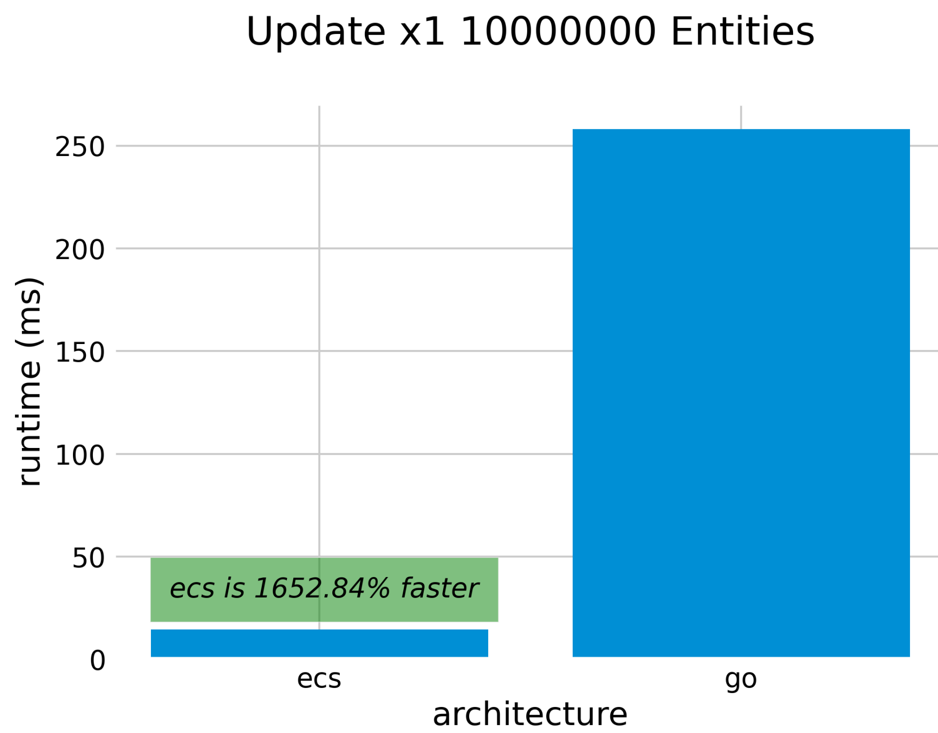

Figure 4.35: Update x1 Benchmark $10M$ entities.

4.4 Create and Destroy Benchmark

This benchmark creates and then destroys entities using both architectures. The expected result is that zip ecsa will be much faster because it does not have to allocate or free any memory and game objects architecture does. Only the creation and deletion API calls were measured and since zip entities are just integers in an array there is really nothing to create or destroy just some flags to clear and so it should be much faster.

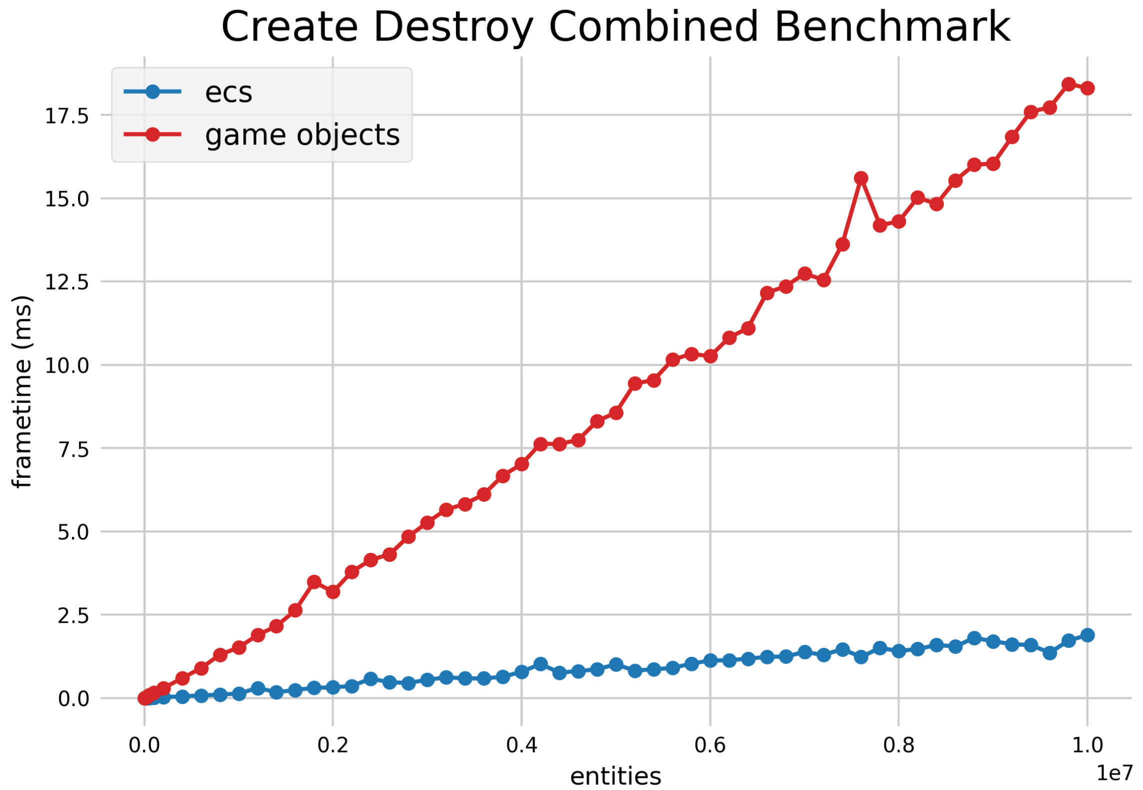

Figure 4.36: Combined Create Destroy Benchmark.

4.4.1 Combined Results

The combined results show individual tests plotted as curves for both the data-oriented zip ecsa and goa test. The y axis shows run-time in *ms* and lower is better. Each test result was averaged from 100 trials at the respective setting. The values were plotted from Table 4.8. As expected, looking at the curve, zip ecsa is at least an order of magnitude faster. It is worth mentioning that it should be possible to reuse game objects and components and speed up this benchmark for game objects but that is a non trivial thing to implement with polymorphic objects and the idea of this benchmark was just to compare the two naive implementations since that is what many developers do use.

Table 4.8: Create and Destroy Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+03	0.000110	0.001670
1.0E+04	0.001750	0.016080
2.5E+04	0.005120	0.034970
5.0E+04	0.005930	0.084240
1.0E+05	0.013550	0.158770
2.0E+05	0.029630	0.300630
4.0E+05	0.051660	0.595150
6.0E+05	0.081550	0.898610

Table 4.8: Create and Destroy Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
8.0E+05	0.102120	1.300750
1.0E+06	0.137950	1.518690
1.2E+06	0.302710	1.894160
1.4E+06	0.177750	2.158540
1.6E+06	0.242870	2.633620
1.8E+06	0.308490	3.487890
2.0E+06	0.322910	3.185880
2.2E+06	0.364990	3.794050
2.4E+06	0.582080	4.145460
2.6E+06	0.475700	4.315440
2.8E+06	0.454470	4.848140
3.0E+06	0.550490	5.272220
3.2E+06	0.621430	5.658440
3.4E+06	0.592560	5.824920
3.6E+06	0.589440	6.117080

Table 4.8: Create and Destroy Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
3.8E+06	0.640770	6.674990
4.0E+06	0.787370	7.022290
4.2E+06	1.025500	7.630580
4.4E+06	0.764820	7.623050
4.6E+06	0.810100	7.747310
4.8E+06	0.862530	8.308010
5.0E+06	1.016460	8.569810
5.2E+06	0.821530	9.439880
5.4E+06	0.866260	9.540220
5.6E+06	0.903280	10.153600
5.8E+06	1.036540	10.326600
6.0E+06	1.127170	10.258900
6.2E+06	1.136130	10.814500
6.4E+06	1.182130	11.102200
6.6E+06	1.238240	12.160600

Table 4.8: Create and Destroy Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
6.8E+06	1.247310	12.355100
7.0E+06	1.384140	12.738200
7.2E+06	1.294580	12.550100
7.4E+06	1.469200	13.615000
7.6E+06	1.234640	15.610900
7.8E+06	1.503580	14.190300
8.0E+06	1.413850	14.302200
8.2E+06	1.471820	15.016900
8.4E+06	1.594440	14.827200
8.6E+06	1.549690	15.534300
8.8E+06	1.808150	16.002800
9.0E+06	1.705790	16.036800
9.2E+06	1.617350	16.837900
9.4E+06	1.591330	17.597400
9.6E+06	1.362170	17.721800

Table 4.8: Create and Destroy Combined Benchmark

entities	zip ecsa (ms)	game objects oop (ms)
9.8E+06	1.724580	18.432700
1.0E+07	1.891340	18.303000

4.4.2 Individual Benchmarks

The following individual benchmarks show the time for both the entity-component system and game-objects architectures to perform a single update with one system. Table 4.9 was used to create the individual plots. The graphs shown focus data with order of magnitude differences in entity counts and how the performance scales based on total entities.

4.4.3 Individual Results Analysis

Zip ecsa was expected to be faster, and it was, but the scaling was not expected. It appears there is no scaling and every individual test had similar performance differences within some variance all the way up to $10M$ entities.

Table 4.9: Create and Destroy Individual Benchmarks

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+03	0.000180	0.001610
1.0E+04	0.003450	0.015690
2.5E+04	0.004390	0.038030
5.0E+04	0.008040	0.080760
1.0E+05	0.016700	0.153380
2.5E+05	0.038340	0.384770
5.0E+05	0.083180	0.762330

Table 4.9: Create and Destroy Individual Benchmarks

entities	zip ecsa (ms)	game objects oop (ms)
1.0E+06	0.122860	1.508810
2.5E+06	0.492690	3.880730
5.0E+06	0.945900	8.357360
1.0E+07	1.692870	18.261600

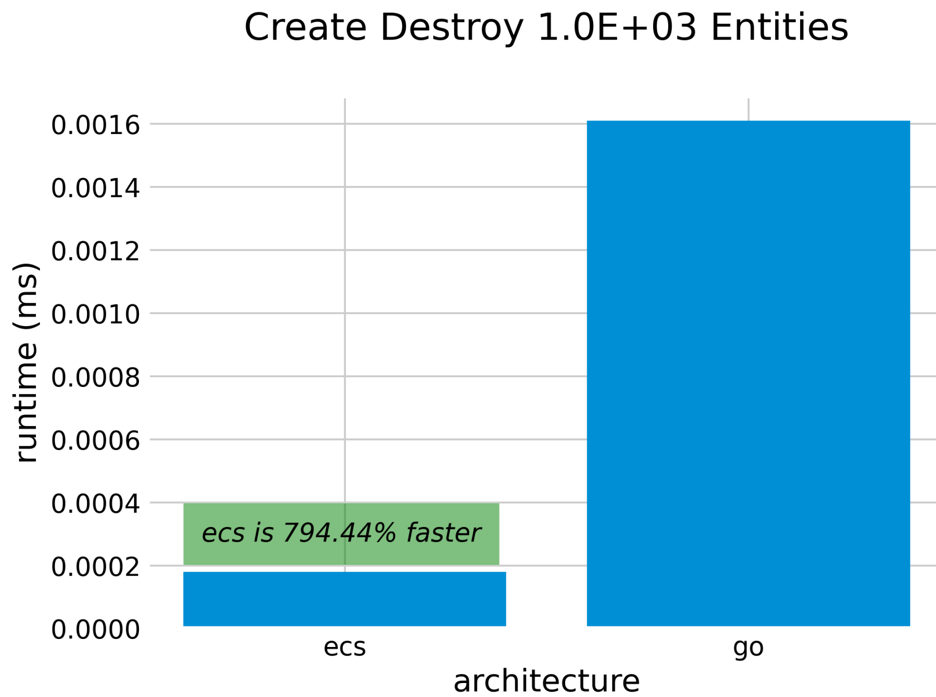

Figure 4.37: Create Destroy Benchmark 1k entities.

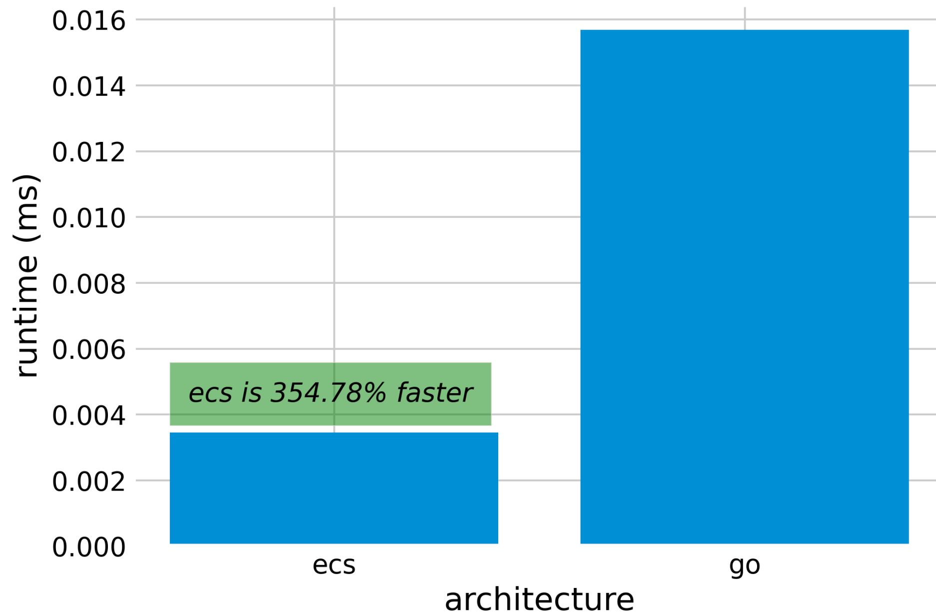

Figure 4.38: Create Destroy Benchmark 10k entities.

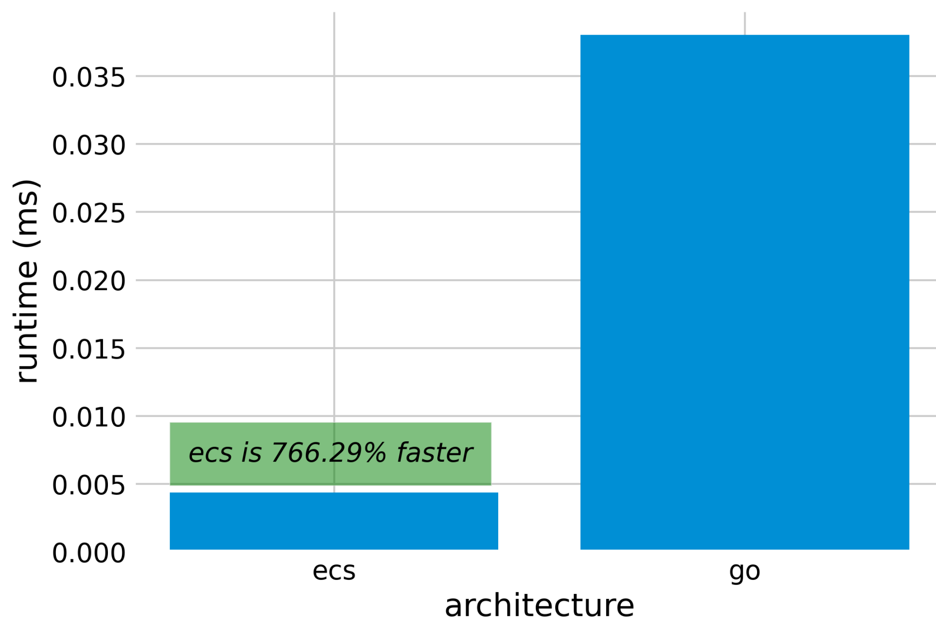

Figure 4.39: Create Destroy Benchmark 25k entities.

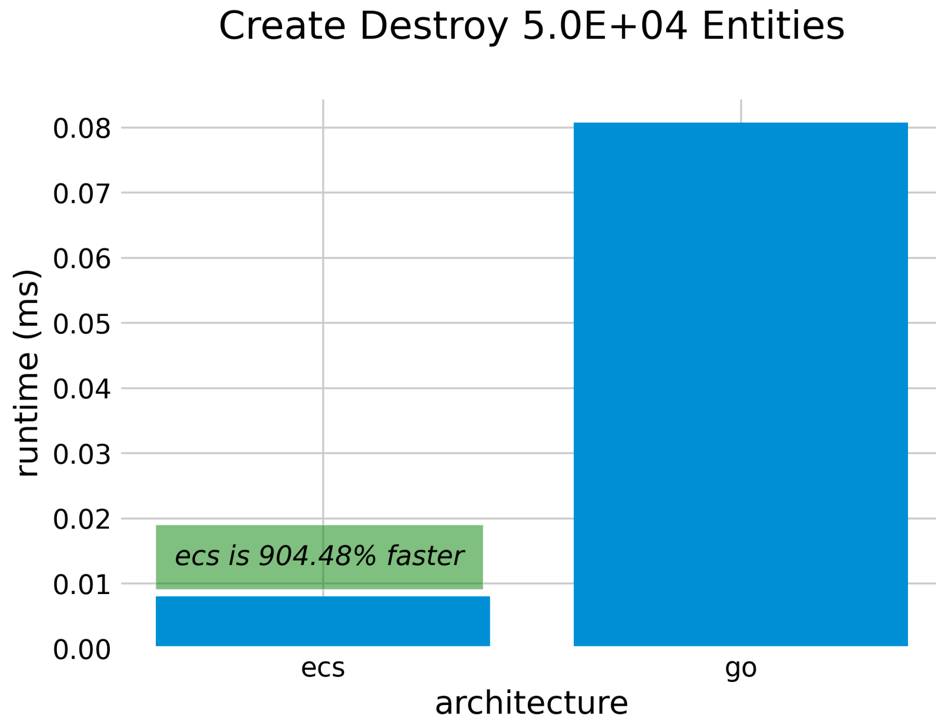

Figure 4.40: Create Destroy Benchmark 50k entities.

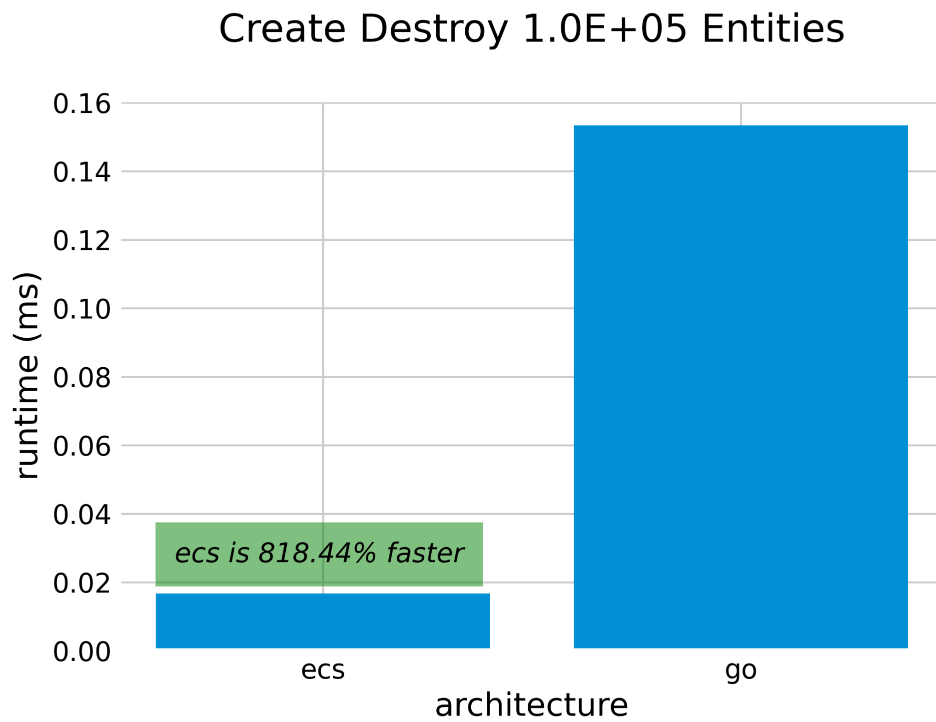

Figure 4.41: Create Destroy Benchmark 100k entities.

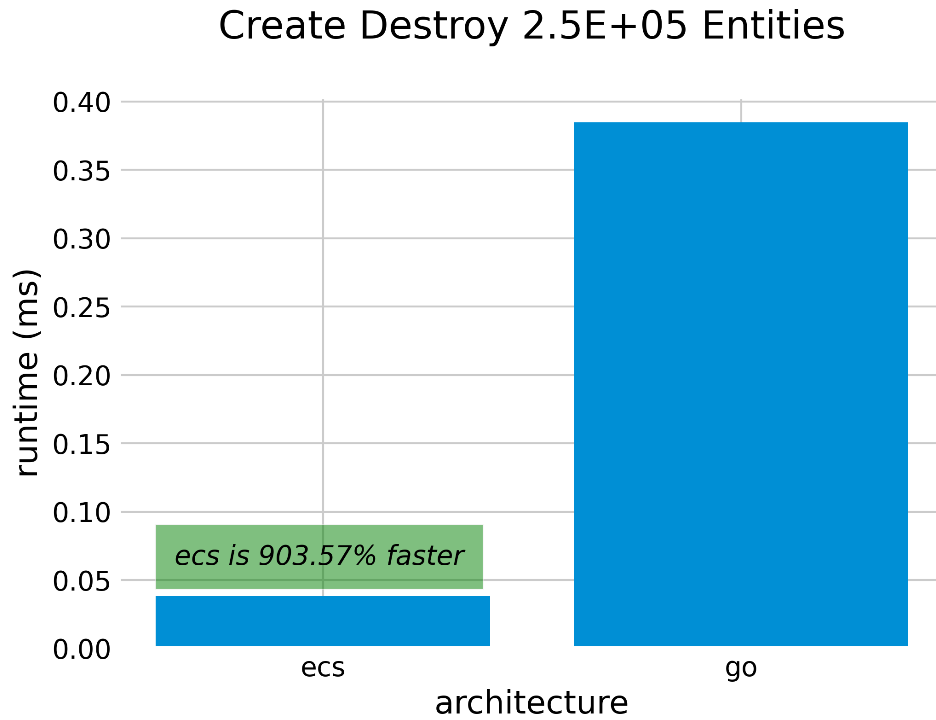

Figure 4.42: Create Destroy Benchmark 250k entities.

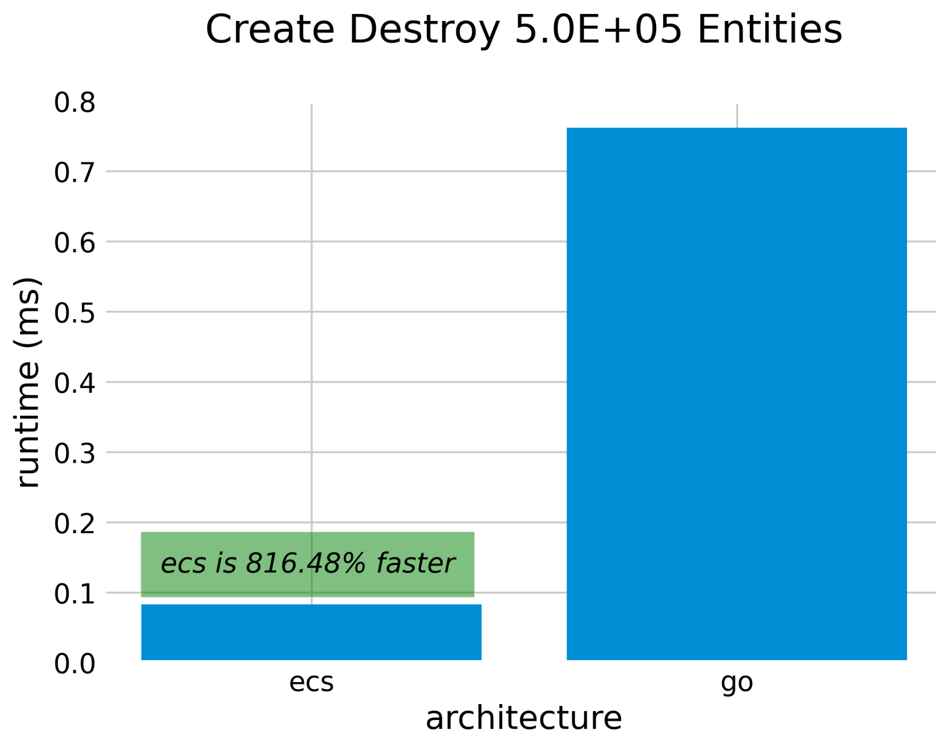

Figure 4.43: Create Destroy Benchmark 500k entities.

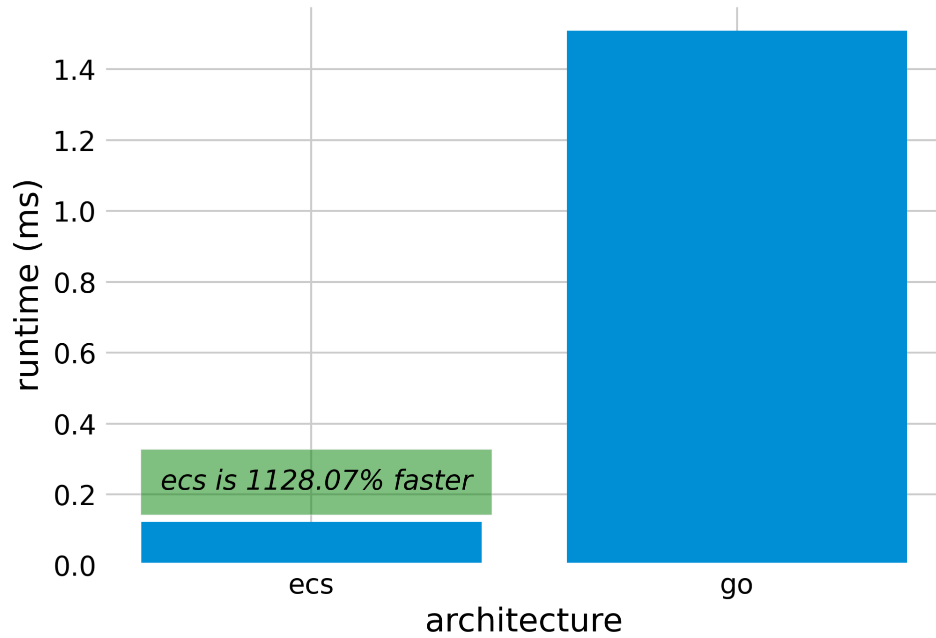

Figure 4.44: Create Destroy Benchmark 1M entities.

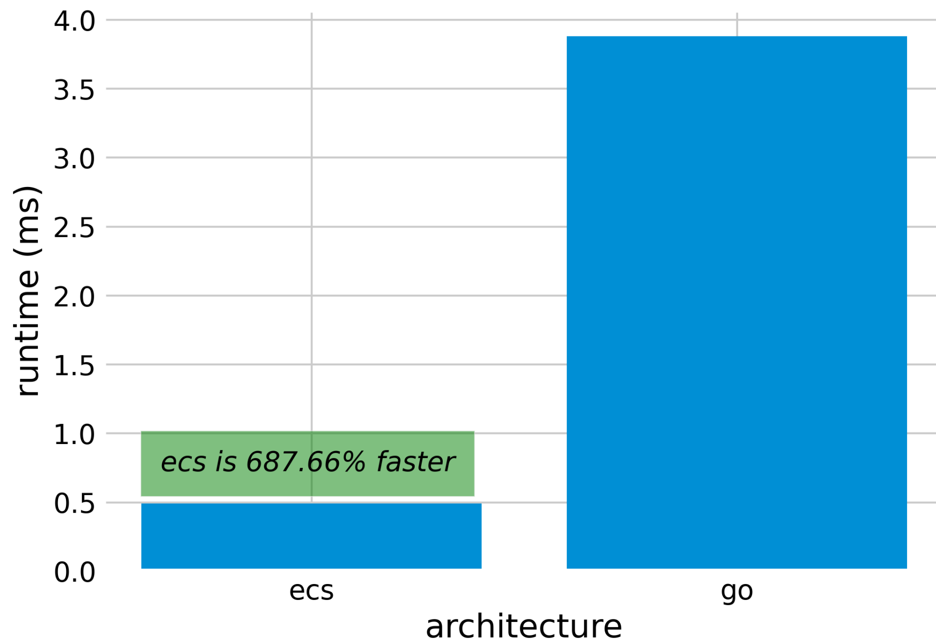

Figure 4.45: Create Destroy Benchmark 2.5M entities.

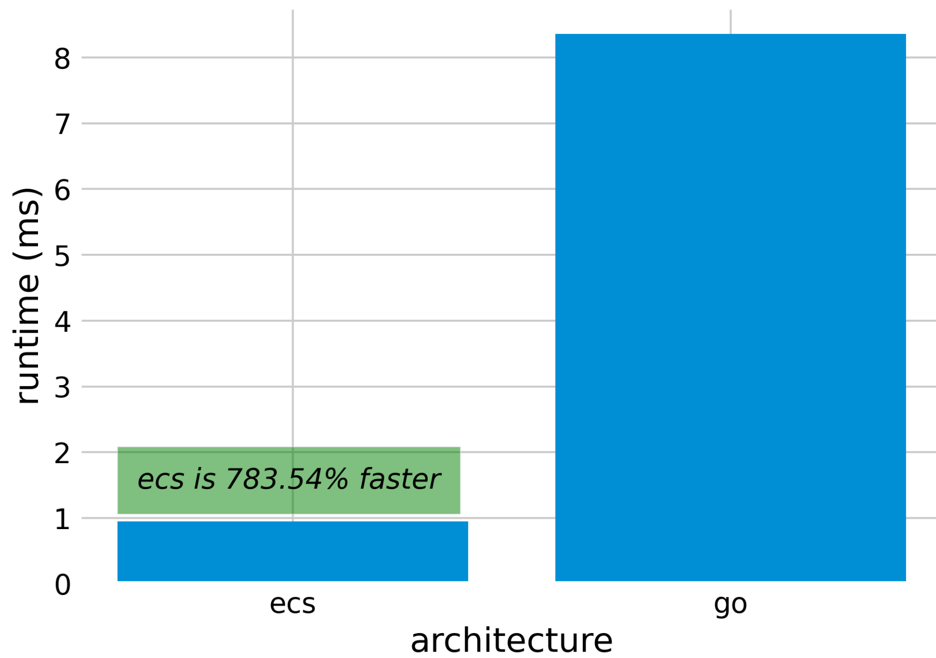

Figure 4.46: Create Destroy Benchmark $5M$ entities.

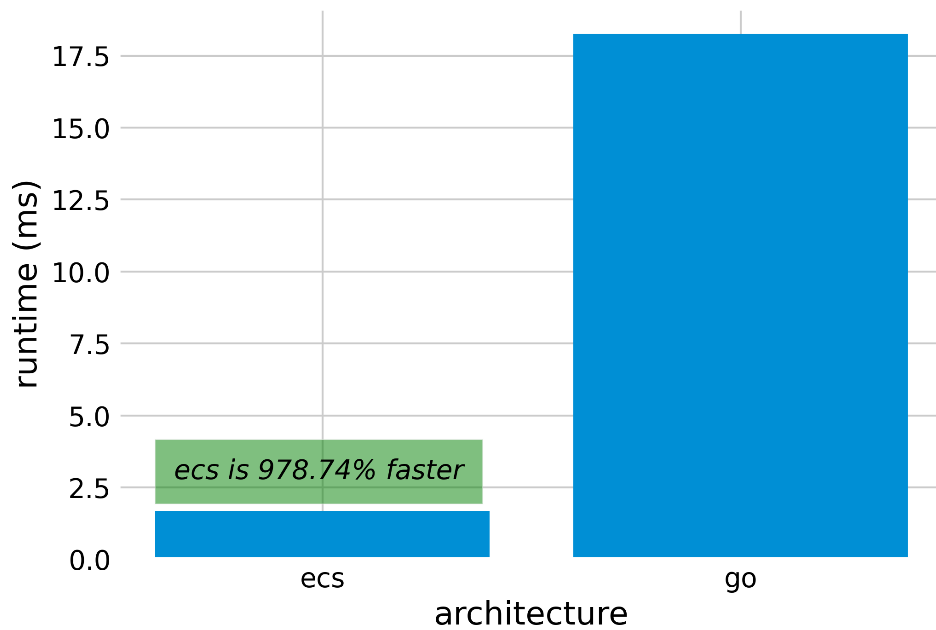

Figure 4.47: Create Destroy Benchmark $10M$ entities.

4.5 Locality of Reference Benchmark

In this benchmark, two versions of zip ecsa were tested against each other to test the effect of how efficient memory layout effects performance. The Update x1 test was reused but with 2 variations:

1. Version 1 - the transform type only contains a vec3 for the position.
2. Version 2 - the transform also contains a vec3 for scale, and 3 vec3's for x, y, z rotation. These are common data that might be stored in a transform data structure. In this test they wont hold any values and they won't be updated but the benchmark recorded the effect of having an additional 12 floats in the transform data structure.

4.5.1 Combined Results

The combined results show individual tests plotted as curves based on data from Table 4.10 for both zip ver1 ecsa and zip ver2 ecsa. The y axis shows run-time in ms and lower is better. Each test result was averaged from 100 trials at the respective setting. As expected, zip v1 with 12 less floats per record was significantly faster for high entity counts.

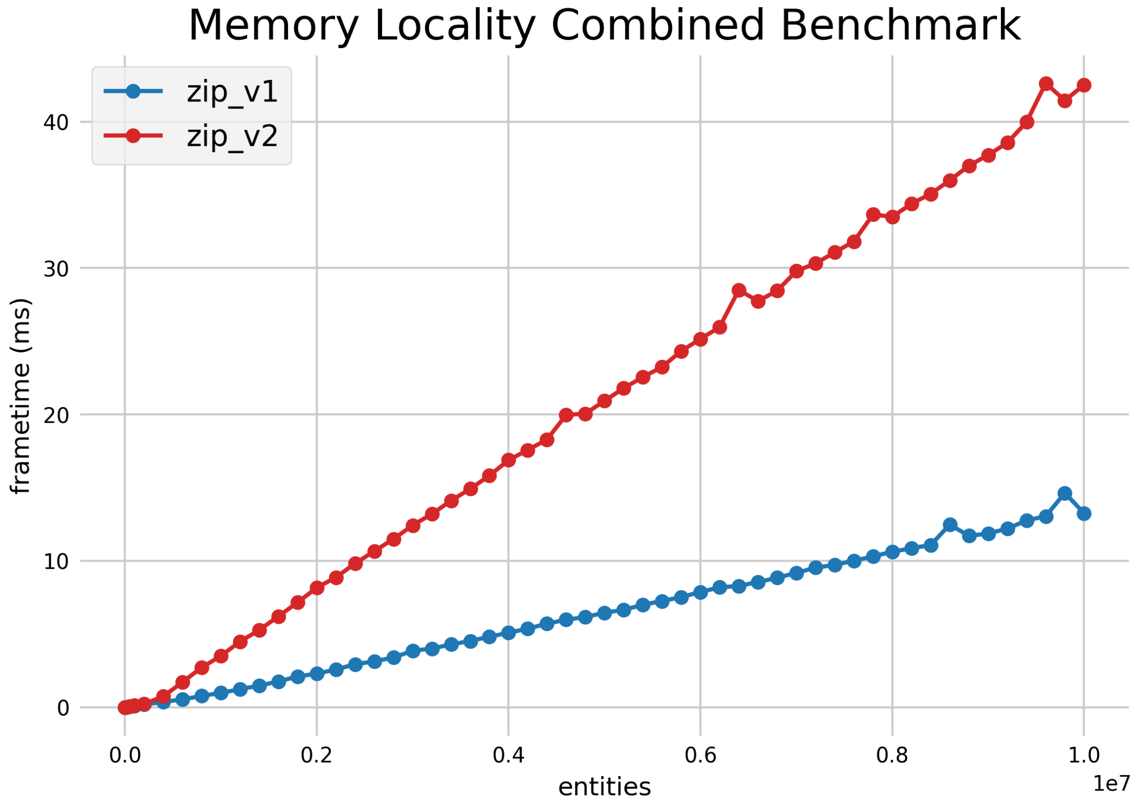

Figure 4.48: Combined Update x1 Benchmark.

Table 4.10: Locality of Reference Combined Results

entities	zip ecsa ver1 (ms)	zip ecsa ver2 (ms)
1.0E+03	0.001370	0.001800
1.0E+04	0.014640	0.016300
2.5E+04	0.032160	0.033120
5.0E+04	0.055090	0.066780
1.0E+05	0.106080	0.125920

Table 4.10: Locality of Reference Combined Results

entities	zip ecsa ver1 (ms)	zip ecsa ver2 (ms)
2.0E+05	0.193580	0.230760
4.0E+05	0.375440	0.748770
6.0E+05	0.543760	1.726830
8.0E+05	0.779770	2.721760
1.0E+06	0.990580	3.513080
1.2E+06	1.241220	4.494940
1.4E+06	1.490230	5.277230
1.6E+06	1.770750	6.225330
1.8E+06	2.111400	7.163480
2.0E+06	2.304680	8.175450
2.2E+06	2.592570	8.877330
2.4E+06	2.944780	9.833970
2.6E+06	3.140440	10.669300
2.8E+06	3.405860	11.501000
3.0E+06	3.866230	12.435300

Table 4.10: Locality of Reference Combined Results

entities	zip ecsa ver1 (ms)	zip ecsa ver2 (ms)
3.2E+06	4.015200	13.207200
3.4E+06	4.297880	14.113600
3.6E+06	4.536580	14.926500
3.8E+06	4.822360	15.825800
4.0E+06	5.099370	16.890100
4.2E+06	5.375420	17.577100
4.4E+06	5.716250	18.294500
4.6E+06	5.986780	19.969900
4.8E+06	6.172700	20.059400
5.0E+06	6.464710	20.930800
5.2E+06	6.661400	21.800100
5.4E+06	7.010530	22.554000
5.6E+06	7.261670	23.247900
5.8E+06	7.533730	24.337000
6.0E+06	7.880040	25.156900

Table 4.10: Locality of Reference Combined Results

entities	zip ecsa ver1 (ms)	zip ecsa ver2 (ms)
6.2E+06	8.195060	25.968500
6.4E+06	8.296690	28.509600
6.6E+06	8.557360	27.747100
6.8E+06	8.852270	28.452000
7.0E+06	9.183500	29.795500
7.2E+06	9.542500	30.317600
7.4E+06	9.732200	31.072400
7.6E+06	10.012500	31.818100
7.8E+06	10.313300	33.660100
8.0E+06	10.628100	33.501000
8.2E+06	10.867600	34.391800
8.4E+06	11.085800	35.068200
8.6E+06	12.480300	35.987600
8.8E+06	11.714700	36.997900
9.0E+06	11.872700	37.720900

Table 4.10: Locality of Reference Combined Results

entities	zip ecsa ver1 (ms)	zip ecsa ver2 (ms)
9.2E+06	12.208900	38.559400
9.4E+06	12.764900	39.969700
9.6E+06	13.058600	42.624700
9.8E+06	14.643500	41.447500
1.0E+07	13.256200	42.513000

4.5.2 Individual Benchmarks

The individual benchmarks show the time for both zip v1 ecsa and zip v2 ecsa based on plots of the data from Table 4.11. The hypothesis was that version 1 would be faster with 12 floats in all cases.

4.5.3 Individual Results Analysis

The hypothesis was proven correct across the board although at the lower entity counts the performance difference fluctuated between 5% and 50%. At higher entity counts the difference stabilized at around 200%.

Table 4.11: Locality of Reference Individual Tests

entities	zip ecsa ver1 (ms)	zip ecsa ver2 (ms)
1.0E+03	0.001380	0.001380
1.0E+04	0.015960	0.015960
2.5E+04	0.028600	0.028600
5.0E+04	0.057230	0.057230
1.0E+05	0.102570	0.102570
2.5E+05	0.236280	0.236280
5.0E+05	0.458310	0.458310
1.0E+06	1.013620	1.013620
2.5E+06	2.982550	2.982550
5.0E+06	6.515930	6.515930
1.0E+07	13.270900	13.270900

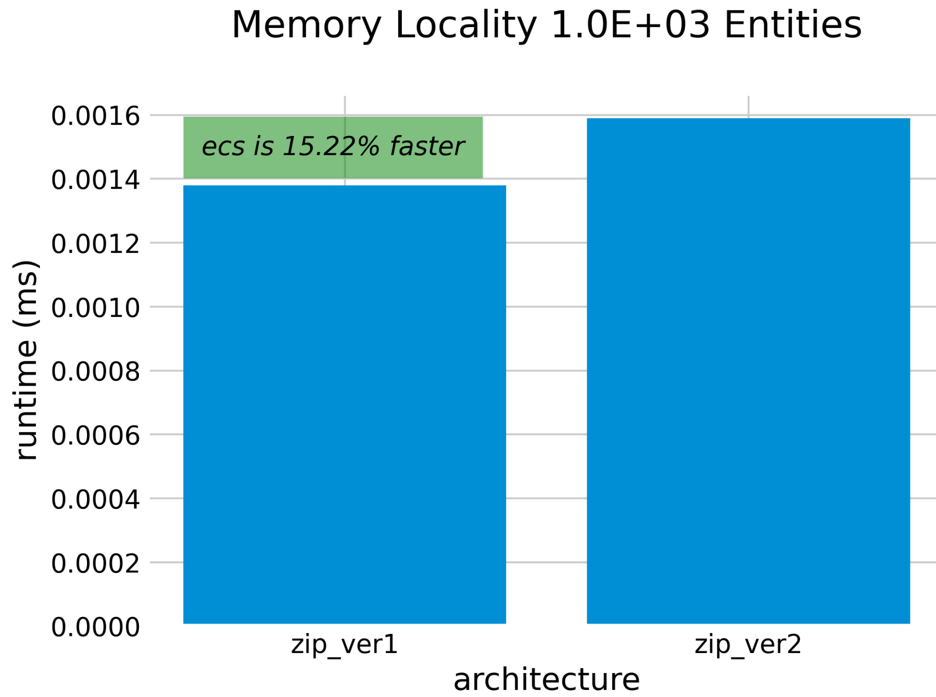

Figure 4.49: Memory Locality Benchmark $1k$ entities.

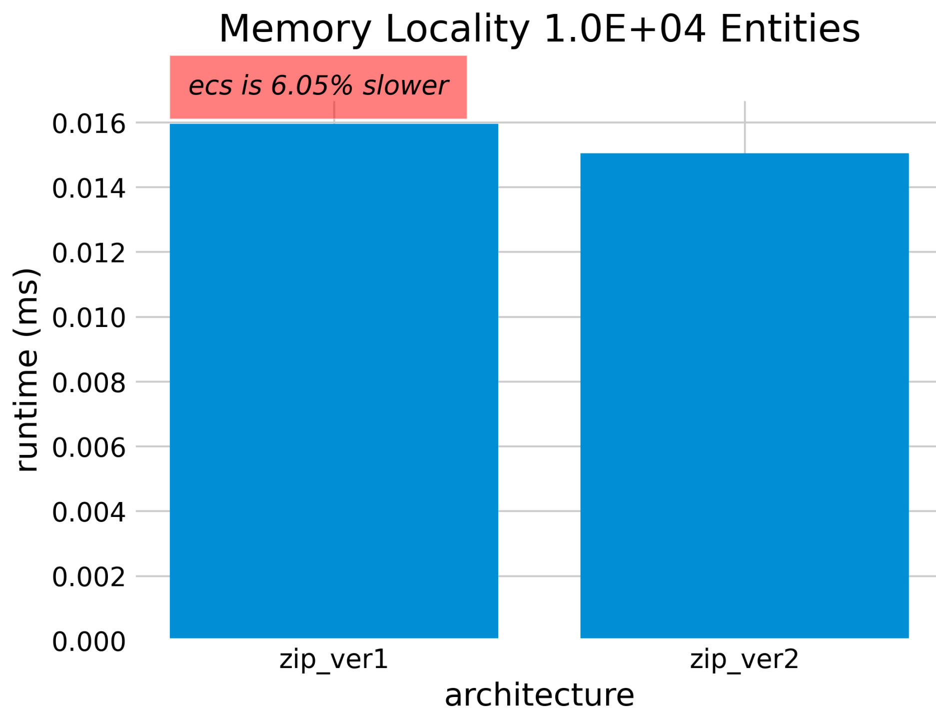

Figure 4.50: Memory Locality Benchmark $10k$ entities.

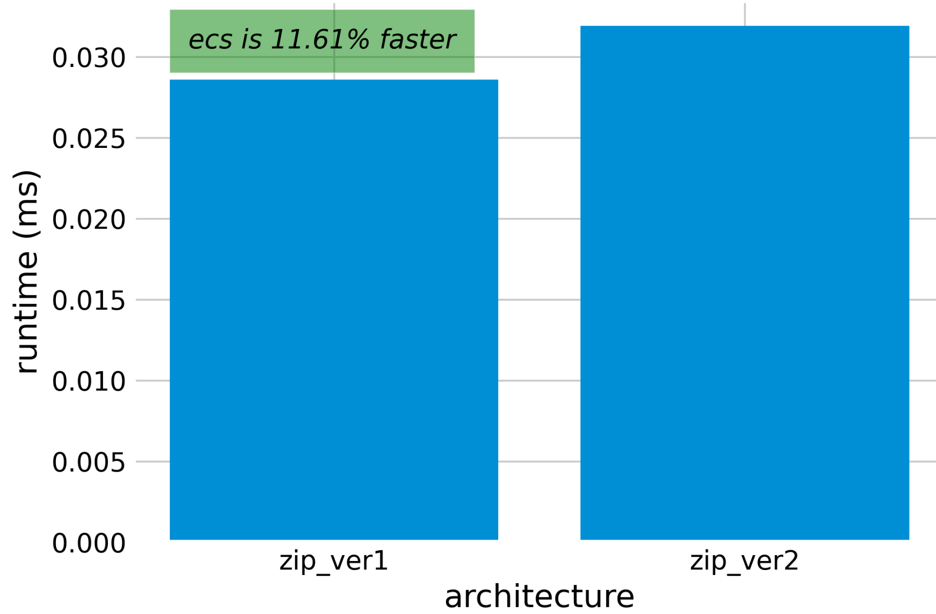

Figure 4.51: Memory Locality Benchmark 25k entities.

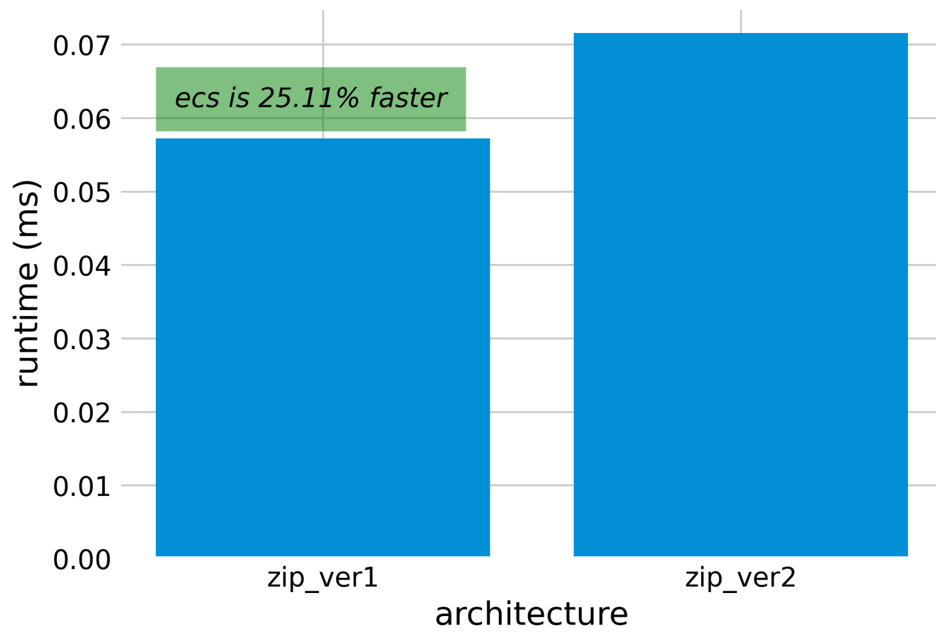

Figure 4.52: Memory Locality Benchmark 50k entities.

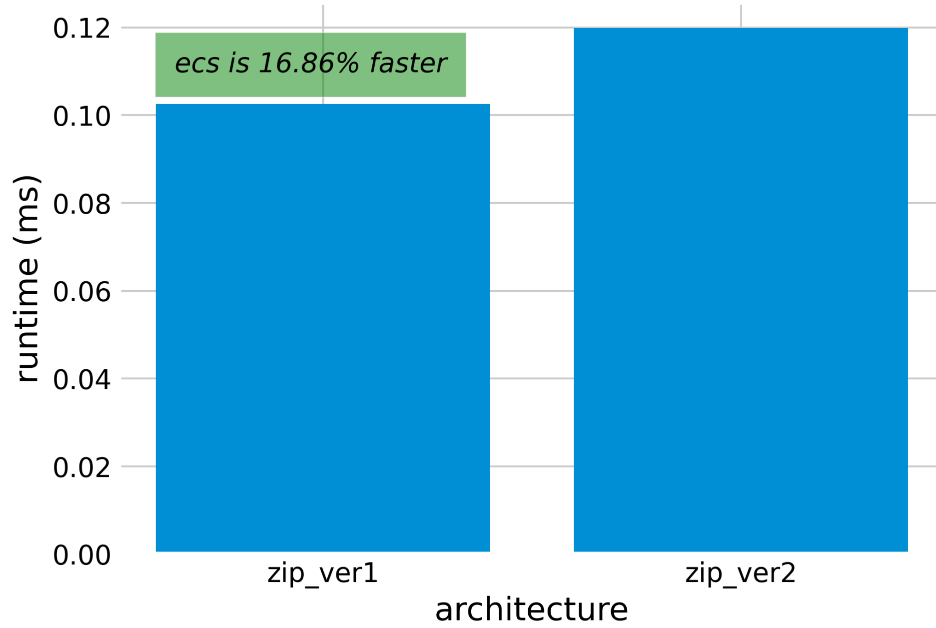

Figure 4.53: Memory Locality Benchmark 100k entities.

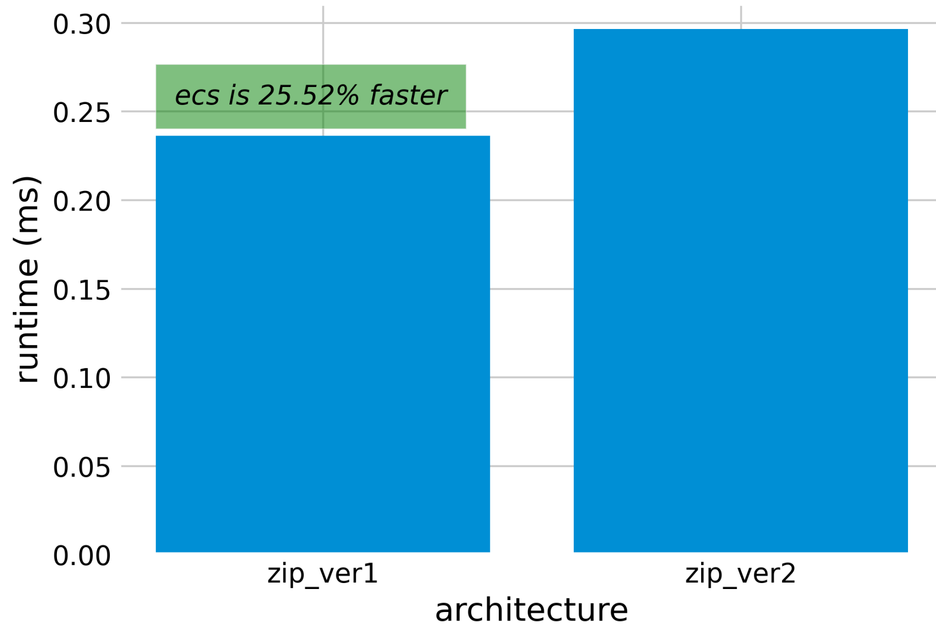

Figure 4.54: Memory Locality Benchmark 250k entities.

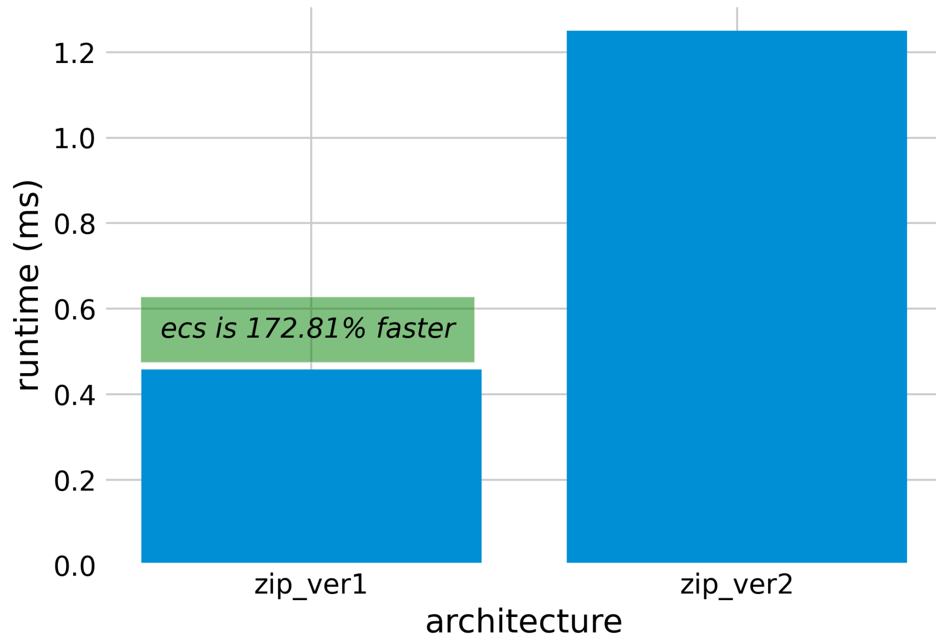

Figure 4.55: Memory Locality Benchmark 500k entities.

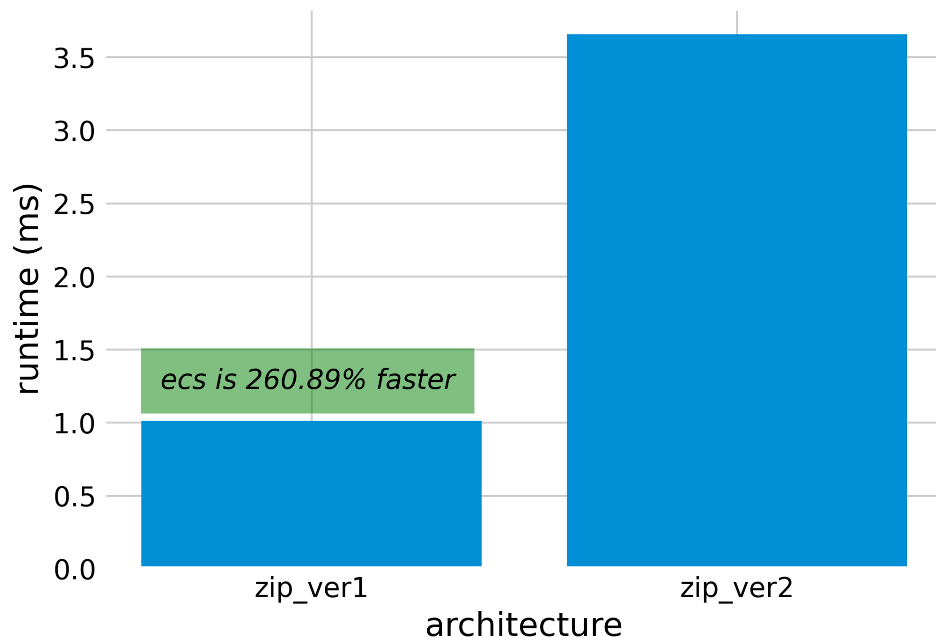

Figure 4.56: Memory Locality Benchmark 1M entities.

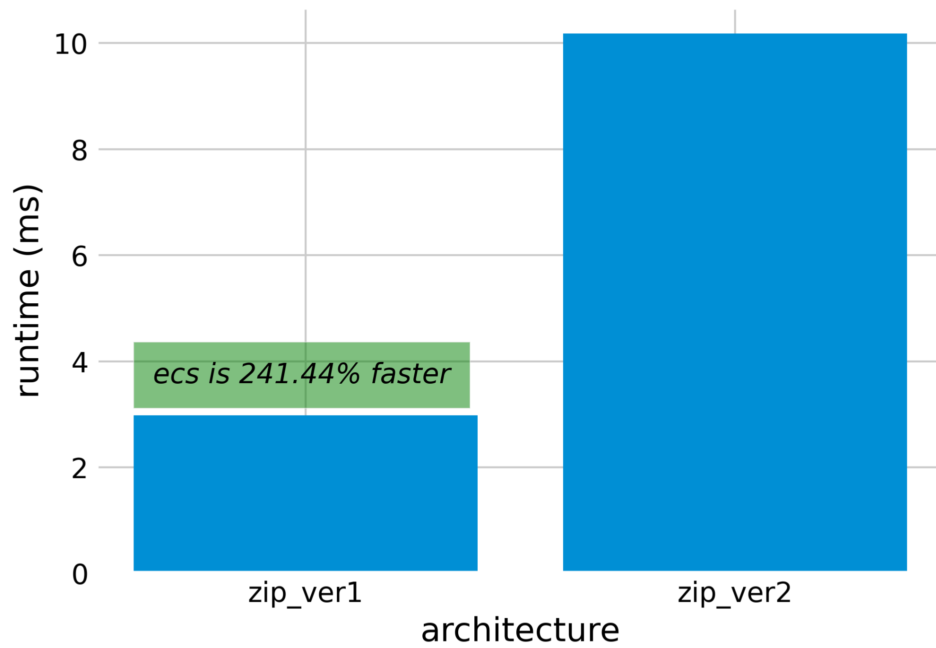

Figure 4.57: Memory Locality Benchmark $2.5M$ entities.

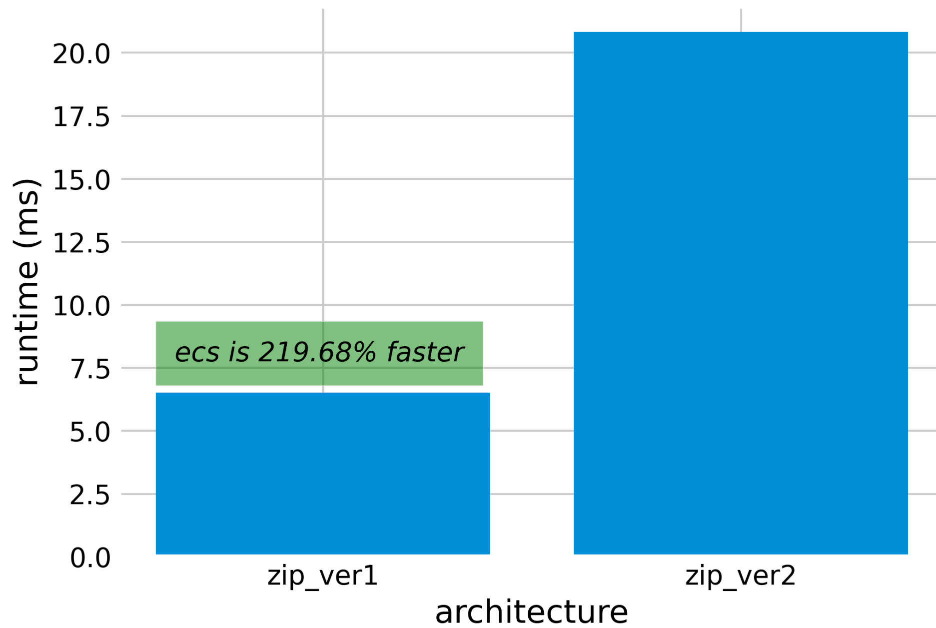

Figure 4.58: Memory Locality Benchmark $5M$ entities.

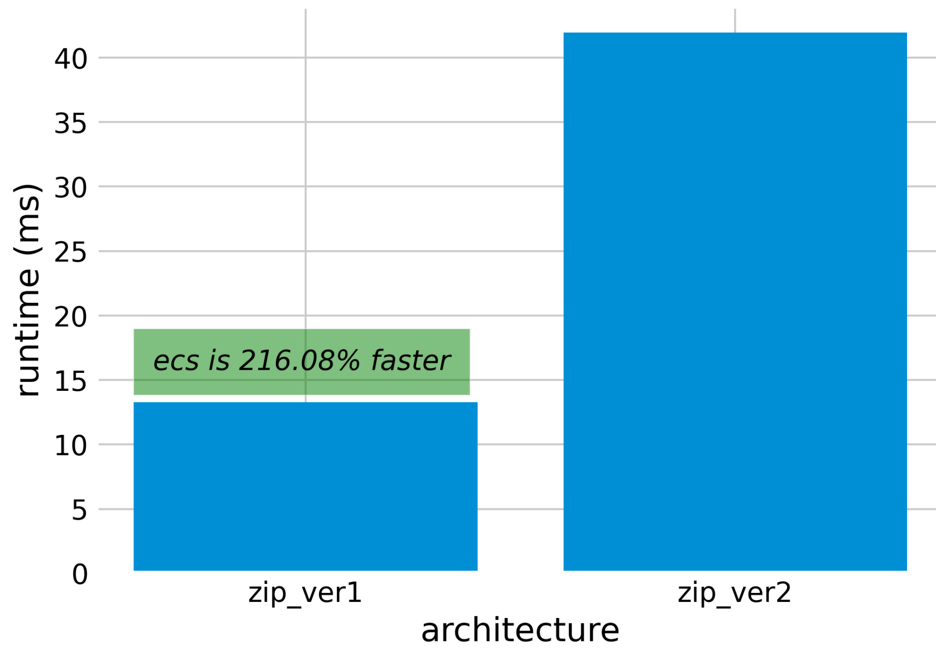

Figure 4.59: Memory Locality Benchmark $10M$ entities.

Chapter 5

CONCLUSION

5.1 Conclusion

Based on the research, implementation and results it can be concluded that ECSA is flexible enough to be used in the same data driven context as object oriented programming and it is suitable for use in real time interactive games. The research showed that ECSA had a significant performance advantage at large entity counts and equivalent performance at lower entity counts. It showed results that were expected but the magnitude of those results were impressive.

The implementation used in zip-ecsa was a naive one and making more specific optimization choices would require a non trivial application and more tests to be evaluated on a case by case basis. This thesis showed that the tools to make that evaluation exist and could be customized for an application to allow the software developers to test different scenarios and evaluate them for the best performance outcomes.

Chapter 6

FUTURE WORKS

6.1 Multi-Threading

This research focused on optimizing single threaded performance. Multi-threading in this type of workload is non trivial since threads will generally need to be kept running and synchronize data workloads with the main thread and there are issues of cache coherency between threads to research and solve. Once a multi-threading strategy is found that is effective, it could be built into the to intelligence of the premake zipgen tool so that cpp files for systems in the ECSA could have the multi-threading code automatically generated based on the config YAML file.

Multi-threading would be applied to the systems and would be broken down into two categories:

1. Outer Parallelism - this means parallelism at the world call level, or allowing the world to execute more than one system in parallel. Allowing this would require some kind of dependency information between the systems so we could know which systems need to be executed serially and which do not. That decision might be based wholly or in part on what components each system modifies since synchronising writes to components between systems in different threads could be very complicated and reduce performance if not handled well.

2. Inner Parallelism - this means parallelism inside the system. Consider the velocity example shown earlier, each velocity is updating 1 position that is on the same entity index and so that should be a highly parallelism operation. On the

surface, this seems straight forward but performance would be bad if we tried to create and destroy threads 60x per second so we need some kind of reusable thread pool or job queue to push work from the system into worker threads and we may need synchronization when modifying component data.

More tests and benchmarks would have to be done to study if adding more threads scales and by how much, or if it causes unexpected issues because of how memory cache is shared between cores.

BIBLIOGRAPHY

[1] Cal Poly Github. http://www.github.com/CalPoly.

[2] How to manipulate data structure to optimize memory use on 32-bit...

[3] M. Acton and I. Games. Data-oriented design and c++. *Luento. CppCon*, 2014.

[4] S. Bilas. A data-driven game object system. In *Game Developers Conference Proceedings*, 2002.

[5] S. Bilas and G. P. Games. The continuous world of dungeon siege. In *Proceedings of the Game Developers Conference*, volume 3, 2003.

[6] J. Boccara. The curiously recurring template pattern (crtp), Sep 2017.

[7] U. Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.

[8] R. Fabian. Data-oriented design. *framework*, 21:1–7, 2018.

[9] J. Gregory. *Game engine architecture*. AK Peters/CRC Press, 2018.

[10] J. Hruska. How l1 and l2 cpu caches work, and why they're an essential part of modern chips. *Extremetech, August*, 30, 2018.

[11] Jbeder. Jbeder/yaml-cpp: A yaml parser and emitter in c++.

[12] N. R. Mahapatra, V. Profile, B. Venkatrao, and O. M. A. Metrics. The processor-memory bottleneck: Problems and solutions: Xrds: Crossroads, the acm magazine for students: Vol 5, no 3es.

[13] A. Martin. Entity systems are the future of mmog development – part 2, Nov 2007.

[14] E. Megagames. The most powerful real-time 3d creation tool.

[15] F. Messaoudi, G. Simon, and A. Ksentini. Dissecting games engines: The case of unity3d. In *2015 international workshop on network and systems support for games (NetGames)*, pages 1–6. IEEE, 2015.

[16] J.-K. Peir, W. Hsu, and A. Smith. Functional implementation techniques for cpu cache memories. *IEEE Transactions on Computers*, 48(2):100–110, 1999.

[17] A. Sanders. *An introduction to Unreal engine 4*. AK Peters/CRC Press, 2016.

[18] B. Stroustrup. What is object-oriented programming? *IEEE software*, 5(3):10–20, 1988.

[19] U. Technologies.

[20] A. Zarrad. Game engine solutions. *Simulation and Gaming*, pages 75–87, 2018.