COS30031 Games Programming Research Project Report

Daniel Coady 102084174

18/10/2021

Contents

Abstract

Game architecture design is not a problem with a one size fits all solution. There are many factors that dictate how ”good” a design is, but most of them are entirely subjective and/or situational. What might work for one person or problem, may not work for another. As such, it’s important to assess the advantages and disadvantages of each design so that you may utilise the ones that best apply to your situation.

1 Introduction

1.1 Background

Game architecture is an art, one that is truly hard to learn and master. In fact, there are many experts and experienced developers who disagree frequently on the topic of how the backbone of games should be structured. This happens for a variety of reasons, but more often than not we will see two aspects of various architectures compared: the usability or maintainability, and the performance.

When we talk about the usability or maintainability of something, we refer to how easy it is to work with. Of interest to us is the ergonomics related to developing with and extending on a given architecture as the needs of a solution expands in scope. Performance on the other hand is far more straightforward–how fast is it? Both aspects are incredibly important when weighing up which architecture is most appropriate to you, your project, and your needs.

1.2 Purpose of This Research

Using the metrics outlined earlier, I aim to compare and contrast four different architectures. This will involve deep dives into every architecture, dissecting how all of them work and what their purposes are. My hope is that through this research presented in this report, the reader may be able to:

- Understand what each architecture is, and how they work
- Understand the purpose of each architecture
- Understand the use case for each architecture
- Come to their own conclusions regarding choices in game architecture

2 Methodology

In order to test the various aspects of each of these architectures, I have had to formulate a series of tests and establish a common testing environment for each of these tests to be run within.

2.1 Tests

All tests will be done with a simple set of entities. These entities will be represented by squares in a graphical window, and they will move with a fixed velocity. When an entity reaches the edge of the window, it will then loop back around to the other side of the window. To add an extra layer of complexity to the processing of the entities, a random amount of entities will also colour shift while moving. Data will be tracked in the form of average cycles per second and time to complete in seconds.

To ensure we collect as much useful data as possible, I will extend this basis for testing in three key ways:

2.1.1 Static

The test will be run with 250,000 entities in the simulation over the course of one minute. The purpose of this test is to understand at a high level how compiler optimisation level affects the speed of each architecture. Additionally, the test will be run through valgrind's cachegrind tool in order to obtain cache profiling information for each architecture.

2.1.2 Ramp-up

There will be multiple tests run for each architecture, starting at 100,000 entities and adding 100,000 with each subsequent test until 500,000 entities is reached. Each test will be run over the course of one minute. The purpose of this test is to understand at a high level how a given architecture scales up given n amount of entities.

2.1.3 Dynamic Ramp-up

A single test will be run for each architecture that starts with 0 entities. For each cycle performed, an entity will be added to the architecture until a limit of 100,000 entities is hit. Once this limit has been hit, entities will start being removed from the architecture until there are none left, and the time taken to execute will be measured. The purpose of this test is to understand at a high level how the overhead introduced by the creation and removal of entities from an architecture affect the execution time of an application.

2.2 Environment

In order to test each of these architectures, a common environment for them to run in has been established. This will take form of a simple front-end abstraction I have written on top of the OpenGL 4.3 Core API, using GLFW for windowing and other miscellaneous functionality. All code used for the environment and development of tests will be written in C++, targeting the C++17 standard at a maximum. Something to note here is that I have elected to use OpenGL 4.3 Core. This is because it is the first version of the OpenGL specification to add compute shaders to the core profile, which will become important later on for one of our architectures.

3 Introduction to Tested Architectures

This research has chosen to focus on four key architectures. The rationale behind this is while it might not be exhaustive, it will provide meaningful data points to inform how different styles and implementations of architecture can affects the usability/maintainability and performance of your game. All code mentioned will be available on GitHub[1] under the Unlicense License.

3.1 Architecture A–Pure Object Oriented

Taking a more traditional and plain object oriented approach, we see what you might expect from a more standard application's codebase. Since it is very pure as far as object oriented architecture goes, much of the same goals of the paradigm carry over to this architecture–that is, we want to maximise code cohesion, minimise code coupling, and reduce overall code duplication. Most of this is to ensure that at a high level, the codebase is as maintainable as possible. There are a few notable parts that make up this architecture:

- Engine

  The brains of the operation. Manages entities and their associated memory.

- Entity

  A base class for all entities in the architecture to inherit from.

- Colour Shift Entity

  A class which derives from the entity base class to add colour shifting functionality.

Starting with the Entity class in figure 1, it contains some basic information required to render a given entity to the provided render target–a 2D position, a 2D velocity, and a colour with RGB components. We also have some simple getter methods for acquiring the private and protected members of the class, and an update method which simply will move the entity's position by the given velocity vector.

The ColorShiftEntity class in figure 2 then inherits from the Entity class and adds a new member to store colour velocity. This colour velocity is also an RGB value which dictates how the colour of an entity should shift with every update. It's for this reason that we have an override for the update method,

[1] https://github.com/pondodev/research-project

```
class Entity {
public:
    Entity(
        unsigned int _id,
        glm::vec2 _position,
        glm::vec2 _velocity,
        glm::vec3 _color );
    virtual void update();
    unsigned int get_id();
    glm::vec2 get_position();
    glm::vec3 get_color();

protected:
    glm::vec3 color;

private:
    unsigned int id;
    glm::vec2 position;
    glm::vec2 velocity;
};
```

Figure 1: Architecture A Entity class declaration

```
class ColorShiftEntity : public Entity {
public:
    ColorShiftEntity(
        unsigned int _id,
        glm::vec2 _position,
        glm::vec2 _velocity,
        glm::vec3 _color,
        glm::vec3 _color_velocity );
    void update() override;

private:
    glm::vec3 color_velocity;
};
```

Figure 2: Architecture A ColorShiftEntity class declaration

```
class Engine {
public:
    ~Engine();
    unsigned int add_entity(
        glm::vec2 _position,
        glm::vec2 _velocity,
        glm::vec3 _color );
    unsigned int add_entity(
        glm::vec2 _position,
        glm::vec2 _velocity,
        glm::vec3 _color,
        glm::vec3 _color_velocity );
    void remove_entity( unsigned int id );
    void pop_entity();
    std::vector<Entity*> get_entities();
    void update();

private:
    unsigned int current_entity_id = 0;
    std::vector<Entity*> entities;
};
```

Figure 3: Architecture A Engine class declaration

which will perform the same update functionality as the base Entity class but will also apply the colour velocity.

Finally, the Engine class in figure 3 which uses a factory-like pattern. The engine is what a programmer would primarily be interfacing with in order to operate the architecture. This is because it is where one would create, manage, and update entities. It provides a method with two overrides to create an entity–one override for regular entities and one override for colour shifting entities. There is also a method to update every single entity managed by the engine.

3.2 Architecture B–Object Oriented Component Pattern

This is an architecture many might be familiar with from general purpose engines such as Unity or Unreal Engine 4. The core idea at play is that pure object oriented approaches to game architecture aren't particularly conducive to how games are generally programmed. In particular, there is the idea of an entity having a series of "traits" which apply to it which in a purely object oriented architecture would normally be implemented through means of inheritance. However, this can be prone to many different types of issues such as deep, complicated inheritance trees or ambiguity introduced through diamond inheritance. To add to this, it becomes significantly less feasible to implement this kind of architecture with a language such as C# which does not allow for multiple inheritance and would instead require you to use interfaces.

This is the core rationale behind the component pattern in object oriented game architecture design–reduce the amount of inheritance required by storing a collection of components which define traits of an entity, inside of an entity. This also means that in languages like C# which do not support multiple inheritance, we can still implement this pattern in a clean and maintainable manner. My implementation has 4 key parts:

- Engine

  Much like the pure object oriented architecture, manages the creation, removal, and updating of entities.

- Entity

  A very bare bones class that has an id and collection of components.

- Component Base

  The base class from which all components that can define traits or behaviour of an entity will inherit from.

- Components

  Child classes of the component base class that will define specific traits or behaviour for entities it is applied to.

The Entity class is incredibly simple for the most part: an id, a vector of components belonging to this entity, and methods to add/remove components from an entity. There is one rather complex thing, however, and that is the template method for getting a component from an entity. This method uses run time type information (RTTI for short) to get an arbitrary component of a given type from the vector of components.

The ComponentBase class in figure 5, as previously mentioned, will be the base class that all components inherit from. It's simple for the most part, only really providing one getter method for the component's id. However one curiosity here is that the method to get the id is virtual, but we never override it in any of the classes that derive from it. This is due to a quirk of C++ and it's design since type information isn't available during run time in the same way we might expect it to be in something like C# and it's reflection feature. To get around this, there is RTTI which can give us partial information on types during runtime, but it requires us to do two things: provide a pointer, and implement at least one virtual method on the base class to be inherited from.

Components as seen in figure 6 all inherit from the ComponentBase class. This allows us to polymorphically store them in a collection (which is exactly what we do in the Entity class) while still providing unique functionality per component.

Finally there is the Engine class in figure 7. Much like the pure object oriented architecture, the engine will manage all entities and their associated

```
class Entity {
public:
    Entity();
    ~Entity();
    unsigned int get_id();
    void add_component( Component* c );
    void remove_component( unsigned int id );
    template <typename T>
    std::optional<T*> get_component() {
        std::optional<T*> to_return;

        for ( auto c : components ) {
            if ( typeid(*c) == typeid(T) ) {
                to_return = (T*)c;
                break;
            }
        }

        return to_return;
    }

private:
    static inline unsigned int next_id;
    unsigned int id;
    std::vector<Component*> components;
};
```

Figure 4: Architecture B Entity class declaration

```
class Component {
public:
    Component();
    virtual unsigned int get_id(); // virtual so RTTI works

private:
    static inline unsigned int next_id;
    unsigned int id;
};
```

Figure 5: Architecture B ComponentBase class declaration

```
class MovementComponent : public Component {
public:
    MovementComponent( glm::vec2 _pos, glm::vec2 _vel );
    void move();
    glm::vec2 pos;

private:
    glm::vec2 vel;
};

class ColorComponent : public Component {
public:
    ColorComponent( glm::vec3 _value );
    void apply_velocity( glm::vec3 vel );
    glm::vec3 value;
};

class ColorVelocityComponent : public Component {
public:
    ColorVelocityComponent( glm::vec3 _value );
    glm::vec3 value;
};
```

Figure 6: Architecture B Component class declarations

```
class Engine {
public:
    ~Engine();
    void add_entity( Entity* entity );
    void remove_entity( unsigned int id );
    void pop_entity();
    std::optional<Entity*> get_entity( unsigned int id );
    std::vector<Entity*> get_all_entities();
    void update();

private:
    std::vector<Entity*> entities;
};
```

Figure 7: Architecture B Engine class declaration

```
typedef uint32_t Entity;
typedef enum {
    Movable       = 0b100,
    Color         = 0b010,
    ColorVelocity = 0b001
} ComponentFlag;
```

Figure 8: Architecture C typedefs

memory appropriately. However, a key difference is that creation of the entity is now an external responsibility to the engine and a programmer would then need to register the created entity with the engine. This is something I would classify as a design flaw in the implementation which, if used in the real world, should be remedied. However for the purposes of these tests it will be more than adequate to collect the necessary data.

3.3 Architecture C–Entity Component System

While by no means a new approach to game architecture with uses of it dating back to 2001-2003[2], it is only in recent times where we have seen it truly come into it's own. Entity component systems, often shortened down to ECS, are a performance first architecture design which foregoes object oriented design in favour of data oriented design. The key difference between the two paradigms is that while object oriented design prioritises concepts such as ownership through means of abstraction and encapsulation, data oriented design chooses to separate data entirely from it's functionality. This is done through the 3 parts of an ECS implementation:

- Entities

  Abstract identifiers that are used to denote ownership over a registered component in the architecture.

- Components

  Simple, tightly packed data packets that can be considered a "state" belonging to an entity.

- Systems

  Functionality which operates over a collection of entities and it's select components that are registered to a given system, applying logic and functionality to the components.

The first thing of note are a couple of simple typedefs that have been created, as seen in figure 8. The first is for entities, which defines their id as an unsigned

[2]http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/

```
struct MovableComponent {
    float pos_x;
    float pos_y;
    float vel_x;
    float vel_y;
};

struct ColorComponent {
    float r;
    float g;
    float b;
};

struct ColorVelocityComponent {
    float r;
    float g;
    float b;
};
```

Figure 9: Architecture C component struct declarations

32-bit integer. The second is an enum flag set which we can bitwise-or together to indicate what components have been set on an entity.

Components, as mentioned before, are incredibly simple. Figure 9 shows that each component is nothing more than a struct with primitive data within.

The Engine class in figure 10 is actually somewhat of a simplification of what you might expect in a more general purpose ECS implementation. This is because normally there are separate manager classes for each of the parts of an ECS, however for the purpose of this research I have elected to simplify it down for demonstration purposes and simplicity.

Within this Engine class we have some methods for adding and removing entities, functionality normally delegated to an entity manager. We also have parts of what might be expected from a component manager in the form of methods to add/get components for an entity and tightly packed arrays which contain the components. The final piece of the puzzle then are the systems, which come in two parts. First are the two system methods which perform the actual functionality of a given system, and second are the vectors which contain all the entity ids which are registered to be worked on by a given system.

As a final note, I'll mention that I owe a great deal to Austin Morlan as their ECS implementation[3] inspired much of my own.

[3]https://austinmorlan.com/posts/entity_component_system/

```
class Engine {
public:
    Engine();
    ~Engine();
    std::optional<Entity> add_entity();
    void remove_entity( Entity id );
    int entity_has_component( Entity id, ComponentFlag component );

    void movement_system();
    void color_shift_system();

    MovableComponent* add_movable_component( Entity id );
    ColorComponent* add_color_component( Entity id );
    ColorVelocityComponent* add_color_velocity_component( Entity id );

    MovableComponent* get_movable_component( Entity id );
    ColorComponent* get_color_component( Entity id );
    ColorVelocityComponent* get_color_velocity_component( Entity id );

private:
    std::queue<Entity> available_entities;
    std::vector<Entity> movement_system_entities;
    std::vector<Entity> color_shift_system_entities;

    ComponentFlag* entity_component_flags;
    MovableComponent* movable_components;
    ColorComponent* color_components;
    ColorVelocityComponent* color_velocity_components;
};
```

Figure 10: Architecture C Engine class declaration

3.4 Architecture D–Entity Component System With Compute Shaders

This is identical to Architecture A with one key difference: we use the GPU to run the systems in the ECS implementation. In theory this should provide incredible performance benefits to our architecture since GPUs are incredibly fast at parallel floating point computations. However, much to my own disappointment, I was not able to implement it fully into an ECS implementation for reasons that will be mentioned later. I have, however, successfully written computer shaders in OpenGL[4] so will be able to comment more on other aspects of it. This does unfortunately mean that I have been unable to collect any data on it and as such, will not be able to report on the performance of it.

[4] https://github.com/pondodev/opengl_compute

4 Data

The following sections display the data visualisations for each test run.

4.1 Architecture A

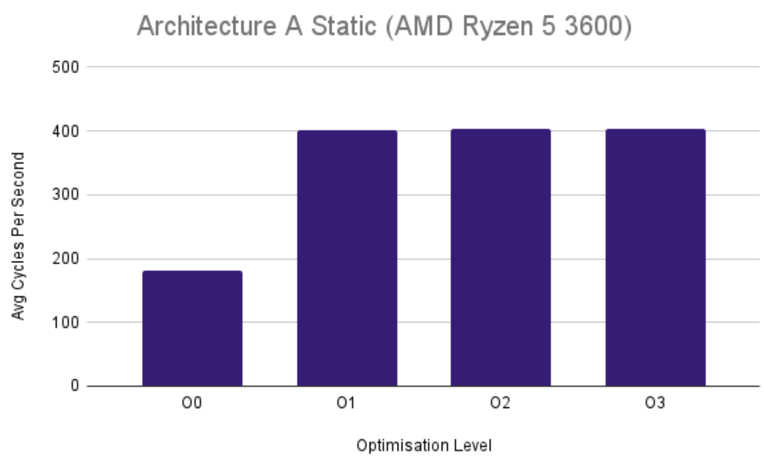

Figure 11: Architecture A Static Test (AMD Ryzen 5 3600)

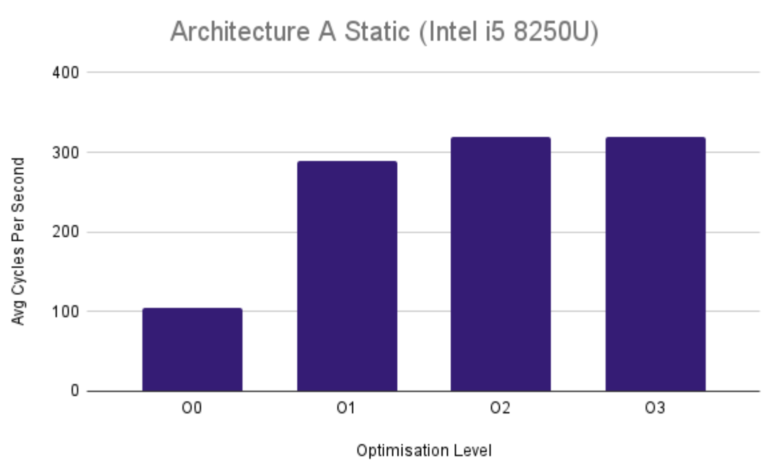

Figure 12: Architecture A Static Test (Intel i5 8250U)

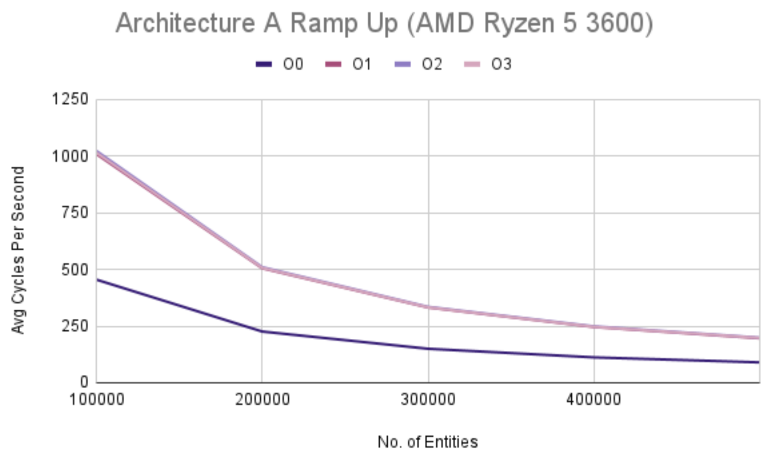

Figure 13: Architecture A Ramp Up Test (AMD Ryzen 5 3600)

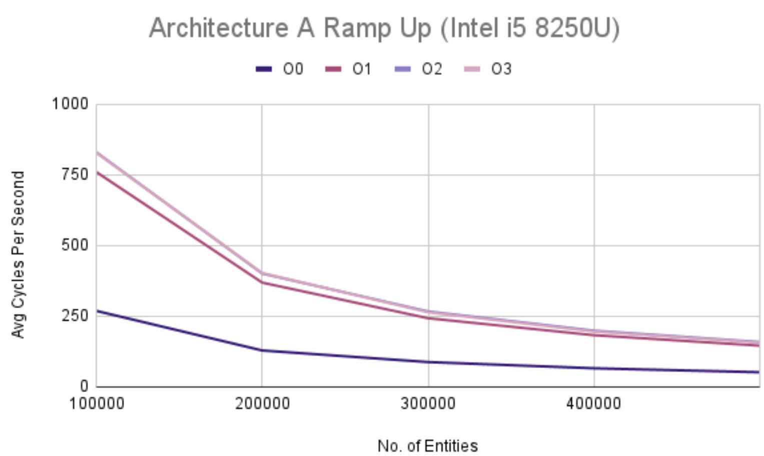

Figure 14: Architecture A Ramp Up Test (Intel i5 8250U)

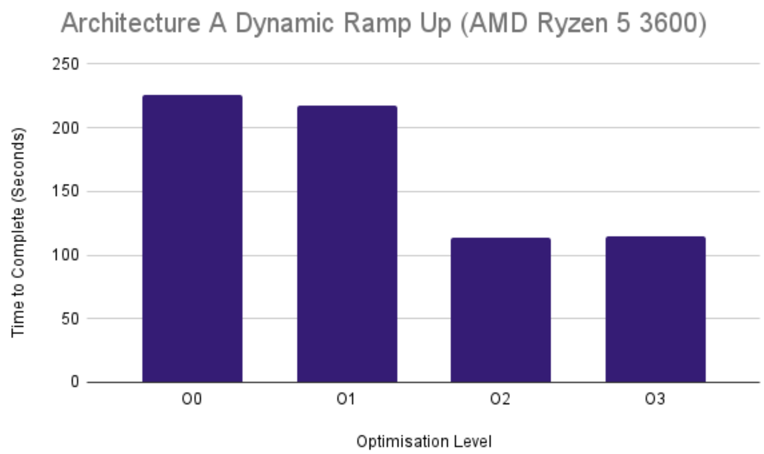

Figure 15: Architecture A Dynamic Ramp Up Test (AMD Ryzen 5 3600)

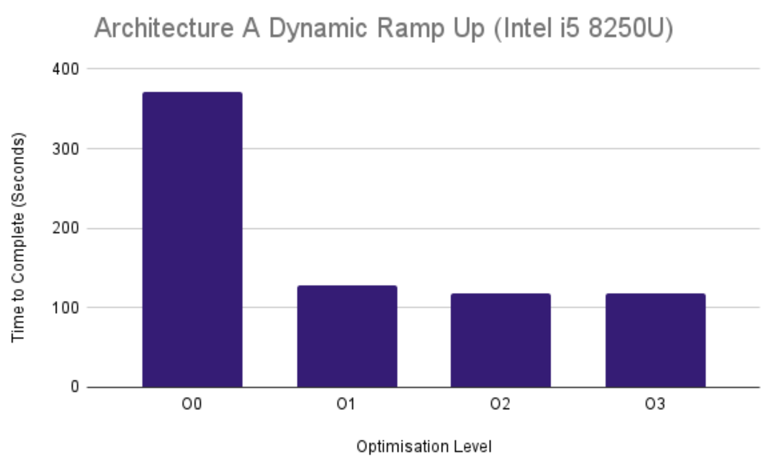

Figure 16: Architecture A Dynamic Ramp Up Test (Intel i5 8250U)

4.2 Architecture B

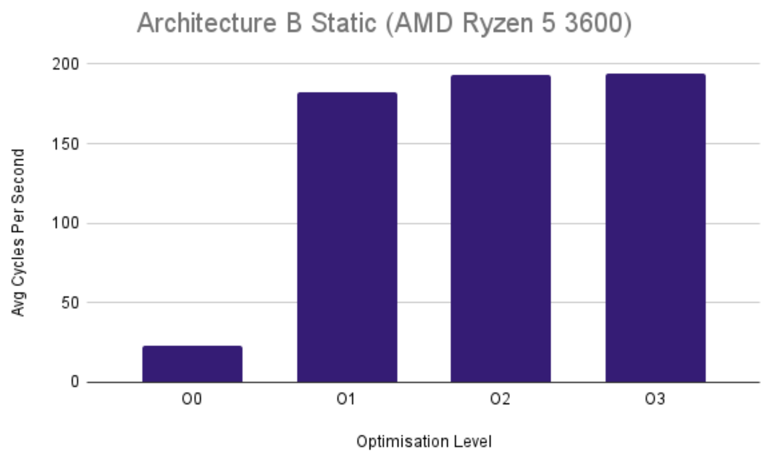

Figure 17: Architecture B Static Test (AMD Ryzen 5 3600)

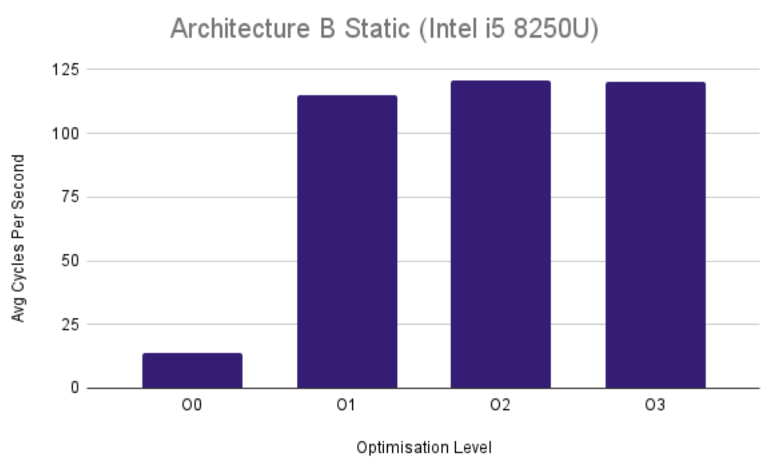

Figure 18: Architecture B Static Test (Intel i5 8250U)

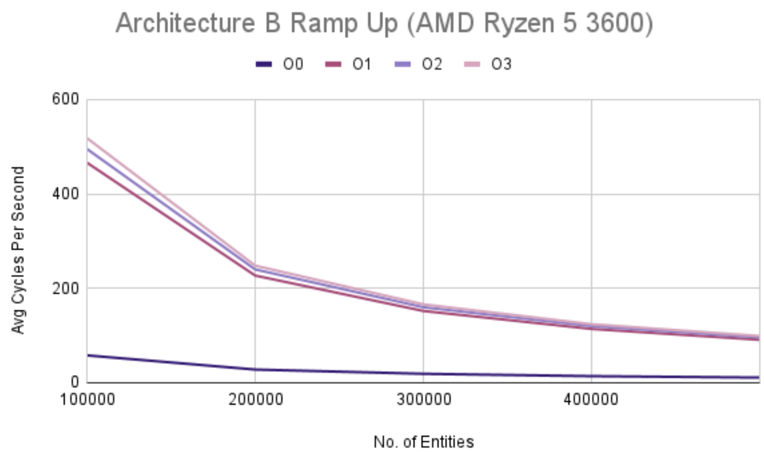

Figure 19: Architecture B Ramp Up Test (AMD Ryzen 5 3600)

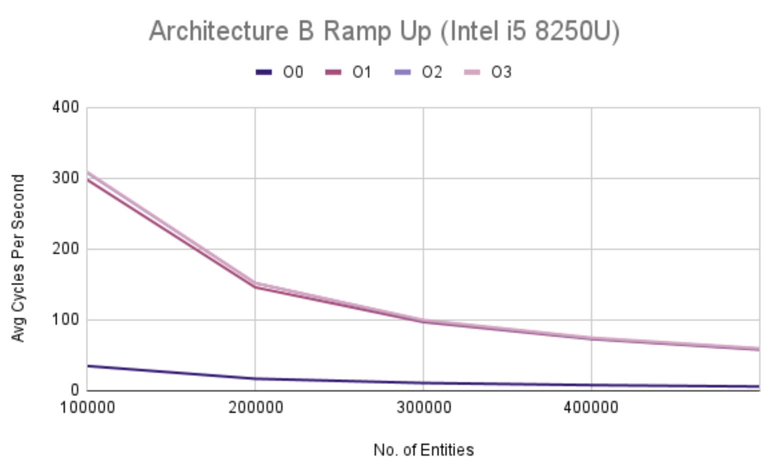

Figure 20: Architecture B Ramp Up Test (Intel i5 8250U)

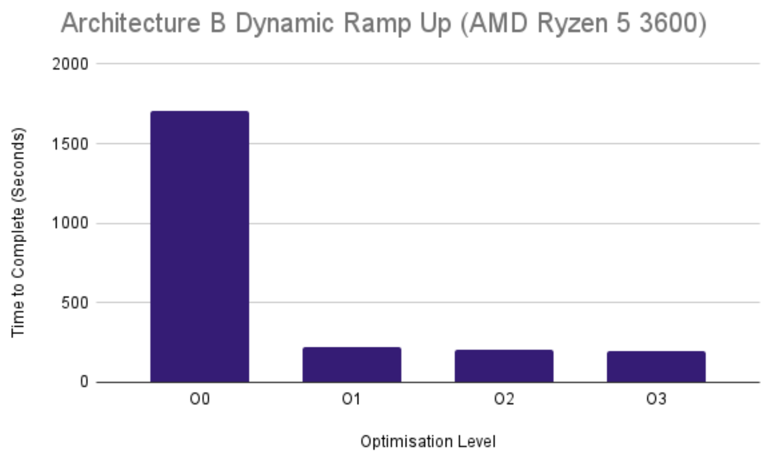

Figure 21: Architecture B Dynamic Ramp Up Test (AMD Ryzen 5 3600)

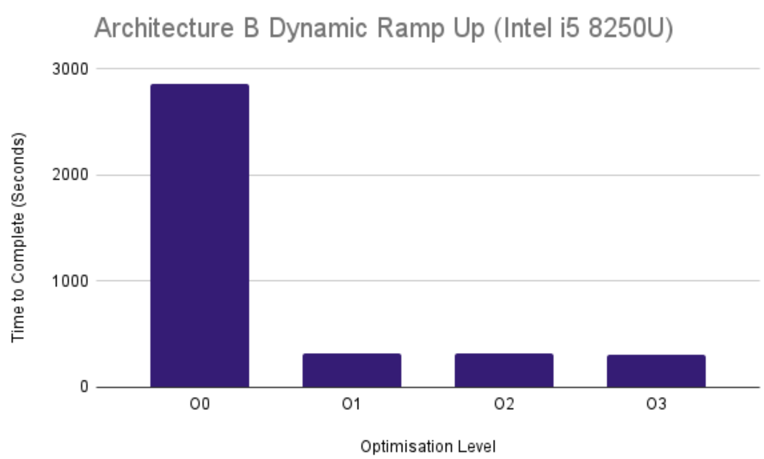

Figure 22: Architecture B Dynamic Ramp Up Test (Intel i5 8250U)

4.3 Architecture C

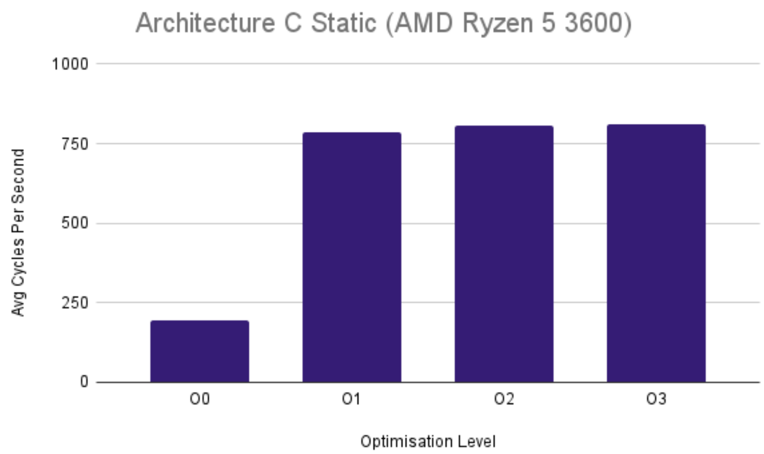

Figure 23: Architecture C Static Test (AMD Ryzen 5 3600)

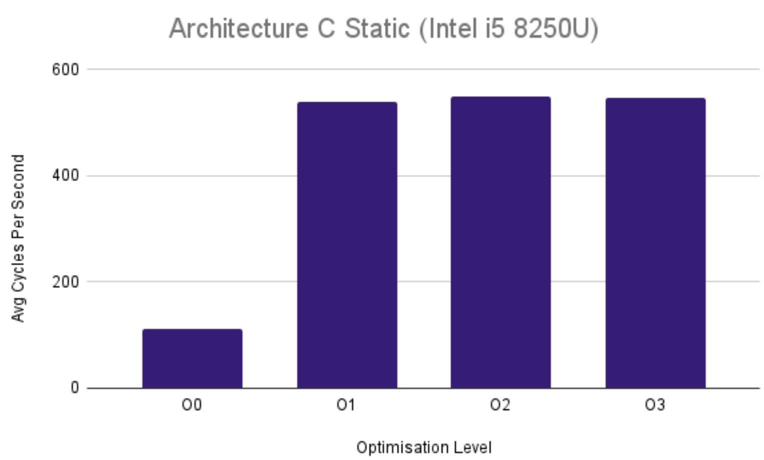

Figure 24: Architecture C Static Test (Intel i5 8250U)

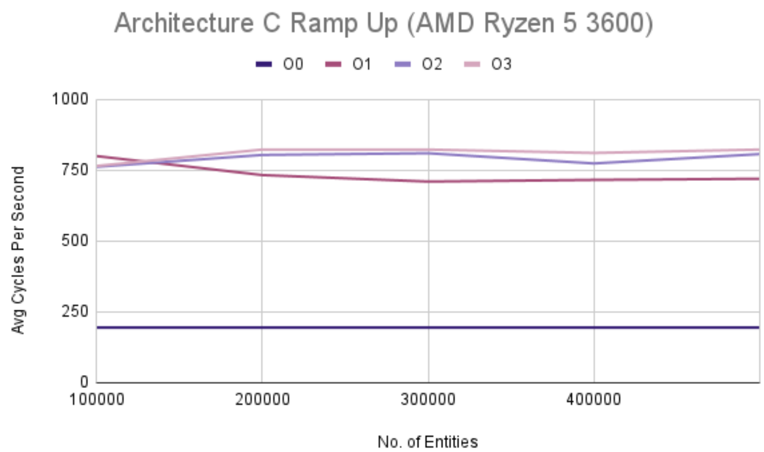

Figure 25: Architecture C Ramp Up Test (AMD Ryzen 5 3600)

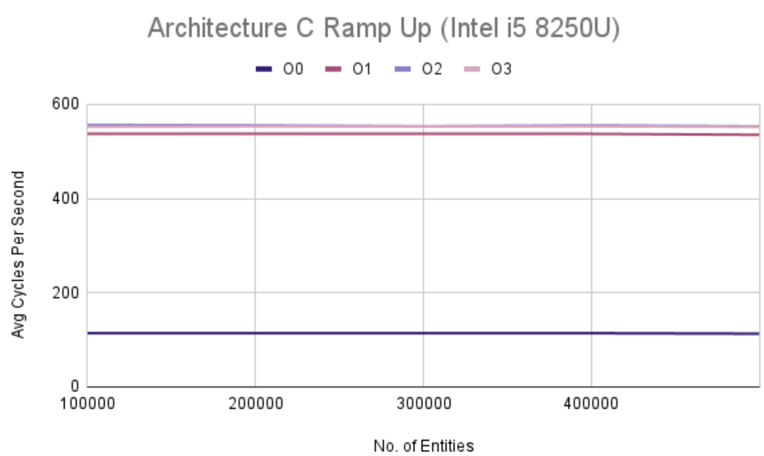

Figure 26: Architecture C Ramp Up Test (Intel i5 8250U)

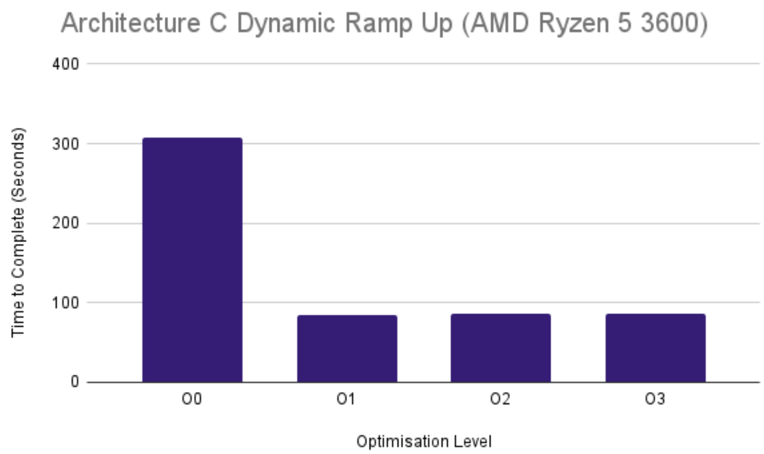

Figure 27: Architecture C Dynamic Ramp Up Test (AMD Ryzen 5 3600)

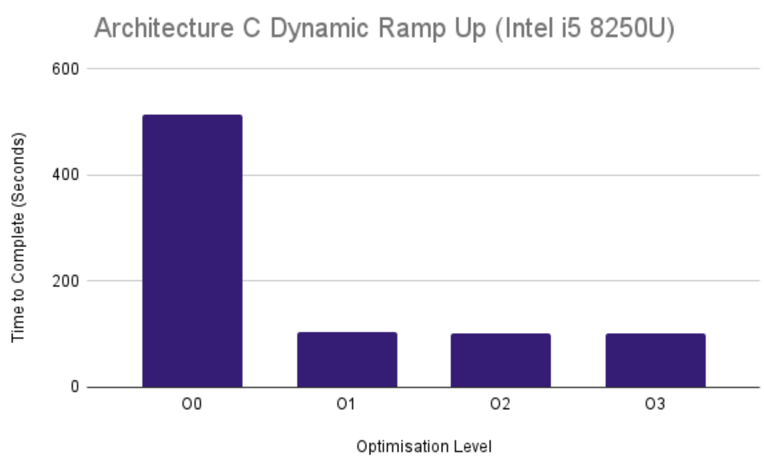

Figure 28: Architecture C Dynamic Ramp Up Test (Intel i5 8250U)

4.4 Architecture Comparisons

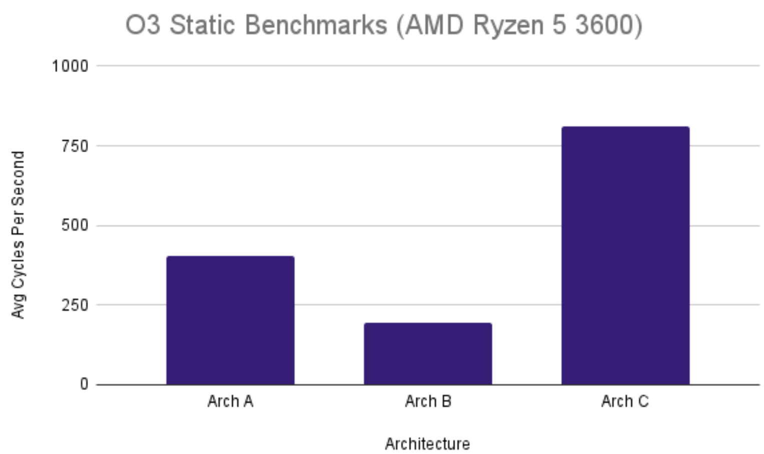

Figure 29: Static Tests (AMD Ryzen 5 3600)

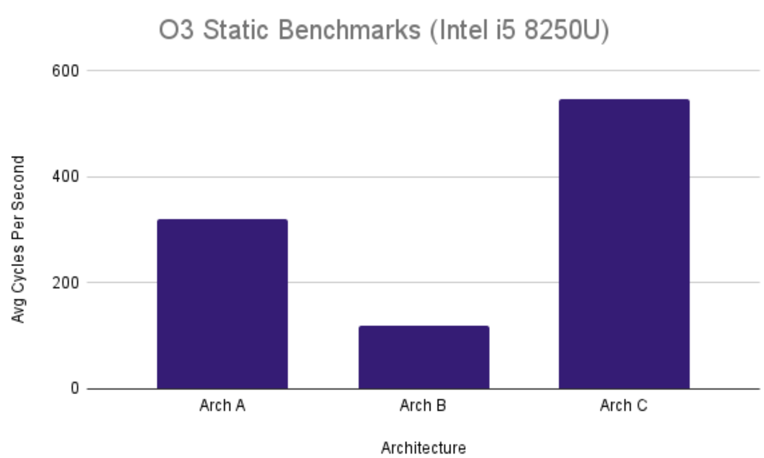

Figure 30: Static Tests (Intel i5 8250U)

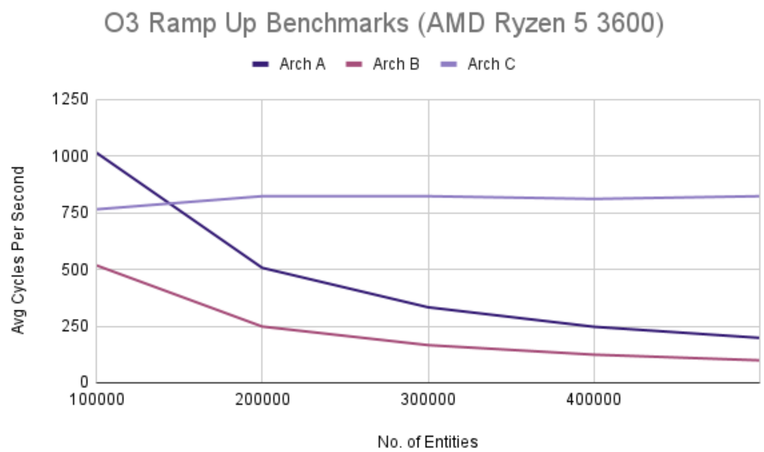

Figure 31: Ramp Up Tests (AMD Ryzen 5 3600)

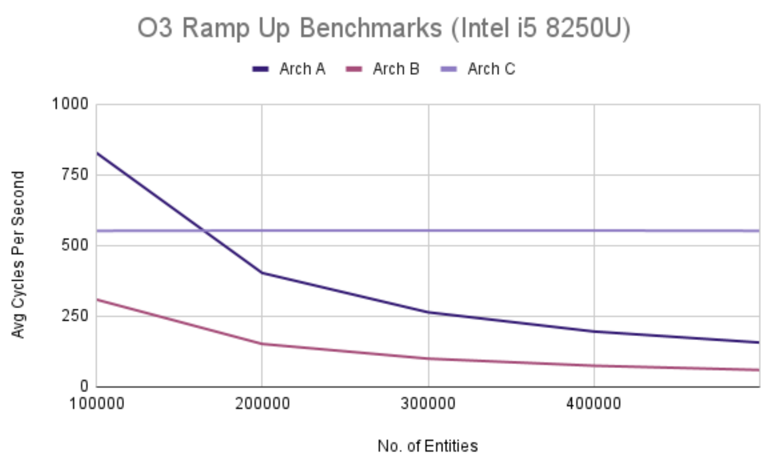

Figure 32: Ramp Up Tests (Intel i5 8250U)

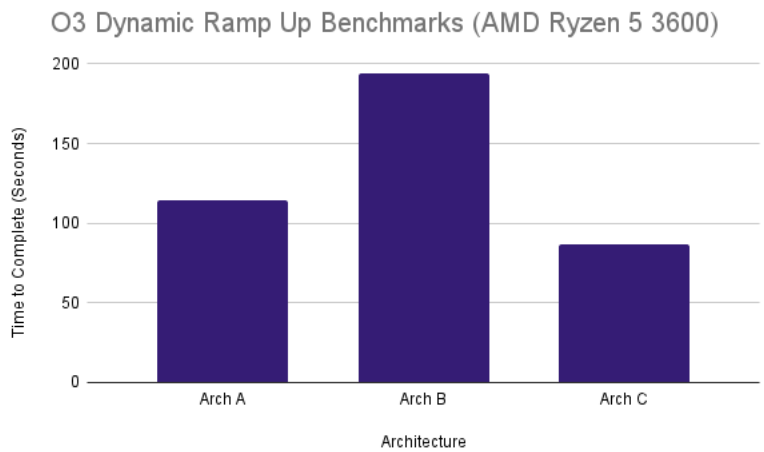

Figure 33: Dynamic Ramp Up Tests (AMD Ryzen 5 3600)

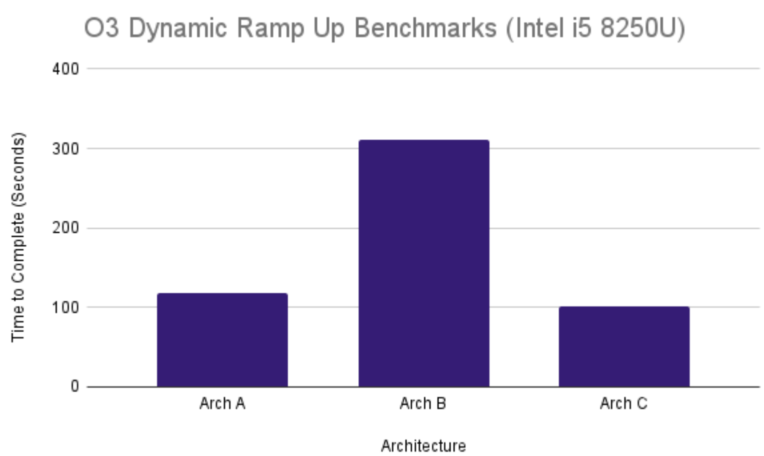

Figure 34: Dynamic Ramp Up Tests (Intel i5 8250U)

4.5 Cachegrind Output

```
I   refs:      15,279,334,589
I1  misses:           768,022
LLi misses:           574,626
I1  miss rate:           0.01%
LLi miss rate:           0.00%

D   refs:       5,911,147,698  (4,233,875,171 rd   + 1,677,272,527 wr)
D1  misses:       230,427,009  (  229,725,605 rd   +       701,404 wr)
LLd misses:        65,787,658  (   65,326,385 rd   +       461,273 wr)
D1  miss rate:            3.9% (          5.4%     +           0.0%  )
LLd miss rate:            1.1% (          1.5%     +           0.0%  )

LL refs:          231,195,031  (  230,493,627 rd   +       701,404 wr)
LL misses:         66,362,284  (   65,901,011 rd   +       461,273 wr)
LL miss rate:             0.3% (          0.3%     +           0.0%  )
```

Figure 35: Architecture A cachegrind output (AMD Ryzen 5 3600)

```
I   refs:      11,673,979,882
I1  misses:           108,417
LLi misses:            45,870
I1  miss rate:           0.00%
LLi miss rate:           0.00%

D   refs:       4,506,264,005  (3,233,764,491 rd   + 1,272,499,514 wr)
D1  misses:       174,092,870  (  173,478,074 rd   +       614,796 wr)
LLd misses:       172,471,117  (  171,946,373 rd   +       524,744 wr)
D1  miss rate:            3.9% (          5.4%     +           0.0%  )
LLd miss rate:            3.8% (          5.3%     +           0.0%  )

LL refs:          174,201,287  (  173,586,491 rd   +       614,796 wr)
LL misses:        172,516,987  (  171,992,243 rd   +       524,744 wr)
LL miss rate:             1.1% (          1.2%     +           0.0%  )
```

Figure 36: Architecture A cachegrind output (Intel i5 8250U)

```
I   refs:      17,093,466,994
I1  misses:           285,064
LLi misses:           238,882
I1  miss rate:           0.00%
LLi miss rate:           0.00%

D   refs:       6,048,221,487  (5,025,865,701 rd   + 1,022,355,786 wr)
D1  misses:       221,749,747  (  220,647,937 rd   +     1,101,810 wr)
LLd misses:       124,021,475  (  123,094,901 rd   +       926,574 wr)
D1  miss rate:            3.7% (          4.4%     +           0.1%  )
LLd miss rate:            2.1% (          2.4%     +           0.1%  )

LL refs:          222,034,811  (  220,933,001 rd   +     1,101,810 wr)
LL misses:        124,260,357  (  123,333,783 rd   +       926,574 wr)
LL miss rate:             0.5% (          0.6%     +           0.1%  )
```

Figure 37: Architecture B cachegrind output (AMD Ryzen 5 3600)

```
I   refs:      14,178,150,149
I1  misses:           110,258
LLi misses:            47,501
I1  miss rate:           0.00%
LLi miss rate:           0.00%

D   refs:       5,006,378,567  (4,162,894,790 rd   + 843,483,777 wr)
D1  misses:       181,408,674  (  180,305,799 rd   +   1,102,875 wr)
LLd misses:       179,883,519  (  178,870,475 rd   +   1,013,044 wr)
D1  miss rate:            3.6% (          4.3%     +         0.1%  )
LLd miss rate:            3.6% (          4.3%     +         0.1%  )

LL refs:          181,518,932  (  180,416,057 rd   +   1,102,875 wr)
LL misses:        179,931,020  (  178,917,976 rd   +   1,013,044 wr)
LL miss rate:             0.9% (          1.0%     +         0.1%  )
```

Figure 38: Architecture B cachegrind output (Intel i5 8250U)

```
I   refs:      20,674,747,170
I1  misses:         1,059,201
LLi misses:           252,983
I1  miss rate:           0.01%
LLi miss rate:           0.00%

D   refs:       6,047,394,193  (4,392,478,824 rd   + 1,654,915,369 wr)
D1  misses:       227,381,637  (  226,639,670 rd   +       741,967 wr)
LLd misses:           826,630  (      419,239 rd   +       407,391 wr)
D1  miss rate:            3.8% (          5.2%     +           0.0%  )
LLd miss rate:            0.0% (          0.0%     +           0.0%  )

LL refs:          228,440,838  (  227,698,871 rd   +       741,967 wr)
LL misses:          1,079,613  (      672,222 rd   +       407,391 wr)
LL miss rate:             0.0% (          0.0%     +           0.0%  )
```

Figure 39: Architecture C cachegrind output (AMD Ryzen 5 3600)

```
I   refs:      13,928,271,322
I1  misses:           108,626
LLi misses:            45,786
I1  miss rate:           0.00%
LLi miss rate:           0.00%

D   refs:       4,087,370,472  (2,971,977,825 rd   + 1,115,392,647 wr)
D1  misses:       152,070,396  (  151,456,130 rd   +       614,266 wr)
LLd misses:       150,530,736  (  149,987,229 rd   +       543,507 wr)
D1  miss rate:            3.7% (          5.1%     +           0.1%  )
LLd miss rate:            3.7% (          5.0%     +           0.0%  )

LL refs:          152,179,022  (  151,564,756 rd   +       614,266 wr)
LL misses:        150,576,522  (  150,033,015 rd   +       543,507 wr)
LL miss rate:             0.8% (          0.9%     +           0.0%  )
```

Figure 40: Architecture C cachegrind output (Intel i5 8250U)

5 Analysis

5.1 Architecture A

5.1.1 Performance

It would appear through the data collected and presented in section 4.4 that Architecture A sits squarely in the middle of the road between all of the architectures being compared. No doubt, a part of this lies within the relative simplicity of the implementation compared to the others. Most notably, within the engine every entity is stored inside of the same vector. Since vectors are, according to the C++ standard, tightly packed collections, in theory the data contained within each entity could be dispatched to the cache for faster access. I must stress though that this is entirely in theory, since cache performance is not only CPU architecture dependant, but also manufacturer dependant. Differences in how cache pre-fetching occurs will massively affect whether a specific collection of data ends up being dispatched to the cache or not, which in turn can have a large impact on performance.

There's also very little in the way of algorithmic complexity since all difference in functionality between the entities are handled through means of polymorphism. This allows us to simply perform a single loop over the collection of entities to achieve the desired functionality in O(n) time.

5.1.2 Usability and Maintainability

This is perhaps the largest drawback to a purely object oriented approach to game architecture. The problem is that the paradigm doesn't lend itself entirely to how games are generally structured, with what might be considered a many to many relationship between entities and traits. That is, many entities may have many traits, and many traits may be relevant to many entities. While this makes sense in more specific applications such as our entities that can either move or move and colour shift, as soon as you start scaling up to larger and larger scopes this quickly becomes unfeasible for everything.

To be clear, I don't think this makes it an entirely useless idea to keep in mind while designing game architecture–quite the contrary actually. One of the core goals of object oriented programming is to reduce code repetition in order to increase maintainability of a given codebase. It's for this reason that when designing larger, encapsulating aspects of an architecture this way of thinking often becomes indispensible. In fact, it's something we see frequently in real world applications of component patterns or ECS. This is because of how well it lends itself to not specific parts of architecture, but rather the larger picture. Add to this that it seems like it can still maintain reasonable performance, and it makes for an exceptional candidate for the backbone of any other core architecture used for managing entities in a game.

5.2 Architecture B

5.2.1 Performance

Again referring to section 4.4, it's incredibly clear just how much worse this architecture performs. It reliably comes last in every test, and by incredibly large margins. Analysing figures 37 and 38 we can see that there are a few cache misses which no doubt has an impact on performance, but I posit that the biggest impact on performance is actually the algorithmic complexity of the implementation. Consider the following implementation of the Engine class' update method:

```
void Engine::update() {
    for ( auto e : entities ) {
        auto mov = e->get_component<MovementComponent>();
        auto col = e->get_component<ColorComponent>();
        auto col_vel = e->get_component<ColorVelocityComponent>();

        if ( mov.has_value() ) mov.value()->move();

        if ( col.has_value() && col_vel.has_value() ) {
            col.value()->apply_velocity( col_vel.value()->value );
        }
    }
}
```

This is a simple loop in O(n) time over a collection of entities, which in of itself is not unusual–every implementation does this for all of it's entities as well. The more interesting part for us however is the get get_component calls, of which there are 3 of them. If we have a look at the implementation of this template method then we would find the following:

```
template <typename T>
std::optional<T*> get_component() {
    std::optional<T*> to_return;

    for ( auto c : components ) {
        if ( typeid(*c) == typeid(T) ) {
            to_return = (T*)c;
            break;
        }
    }

    return to_return;
}
```

This is another algorithm which in this case has a worst case time complexity of O(n). This means that our update method now has a worst case time complexity of O(n(x+y+z)) which is, to put it bluntly, real bad. This is only compounded by the fact that not every entity will have a ColorVelocityComponent, which means that the search time for it on those entities will always be O(n). There are absolutely improvements that can be made to this such as a flag system similar to my ECS implementation's which will add a very fast check to see if a component exists on an entity. This gets complicated quickly however and can affect extensibility due to a need to register components somewhere in the architecture. This is absolutely possible, but still not entirely ideal.

As an aside, this is interesting to note when thinking in the context of a general purpose engine such as Unity. Unity uses a component pattern for it's GameObjects, and it's suggested in many development communities that the GetComponent method's return value be cached in a variable if needed multiple times due to how expensive it is. This exploration into a potential implementation of the method shows how and why it can be so expensive to call, and should serve as a reminder to do such caching if you end up implementing a component pattern in your own game architecture.

5.2.2 Usability and Maintainability

This is what I would consider the "middle ground" of the two extremes discussed in this report. It is still at it's core object oriented, defining an entity as an object that has a set of responsibilities. The key difference between this and a pure object oriented approach such as what has been seen in Architecture A is that the responsibilities are now contained within objects belonging to the entity. This still maintains much of what we set out to do with object oriented design, but now removes the need for deep inheritance trees for more complex entity behaviour. This alone makes the maintainability of the architecture far easier, since now there are less opportunities for bugs that may occur through deep inheritance trees or multiple inheritance.

Add to this that the extensibility is incredibly simple due to realistically only needing to add components to introduce new behaviour. This all makes for an incredibly sensible architecture for the way most developers might think about a game's logic, in turn making it easy for many to work with. To once again analyse this in the context of general purpose engines, it makes sense why this is a pattern so often seen in their designs. While it is most certainly not the most performant (though as previously mentioned, there are absolutely ways in the real world to optimise this) it allows for faster development which means faster iteration times. Often this can outweigh the performance cost, especially in situations where money is at stake since you want to ensure you can iterate quicker to get your product completed sooner. It is also what I would describe as being the most general purpose, being excellent for smaller sets of entities or even one use entities, but still having the ability to scale up to manage larger sets of entities.

5.3 Architecture C

5.3.1 Performance

Once again referring to section 4.4 we are able to see just how much faster ECS is over any other architecture. In fact not only is it faster, but it's *significantly* faster than the next fastest architecture. The reasoning behind this is rather interesting, because of how low level this ends up getting. For starters we have our two systems which are always guaranteed to complete in O(n) time. Since they're both constant and have no bearing on each other, this makes the overall time complexity of the update algorithm O(n). So this would make it the same time complexity as Architecture A, right?

This is where things start to get complicated when talking about time complexity measurements, and also serves as a good example of why big O notation and other methods of measuring time complexity shouldn't be taken as gospel and instead need to be understood in context. You see, in Architecture A we iterate over every entity and perform a simple operation on each of them, making the execution time O(n). In this architecture we perform two separate O(n) time operations one after another, which will then simplify down to completing in O(n) time due to both being constant. The difference between the two however is what n actually is. In Architecture A n is the number of entities, while in this architecture n is both the number of entities registered for the movement system and the number of entities registered for the colour shift system. The reason why this is important is because in this architecture, n is almost certainly going to be bigger than in Architecture A given the same data set. This is due to the two systems potentially working on the same entities multiple times for different things.

So this architecture is slower in absolute terms when looking at the time complexity, and yet it's faster than Architecture A. Why is this? This has already been alluded to a few times earlier in this report, but cache is important here. Since all levels of cache are significantly faster to access than RAM, this makes it a prime area for optimisation with data intensive tasks. So how can we improve performance with the cache? The simple answer is that we want to make sure our data is sent to the cache by the CPU, and that as much of it is possible is sent. That way when it comes the time to actually perform these operations on the data, reads and writes on the data will be significantly faster which in turn makes for a faster overall algorithm.

There are a few things which can influence the sending of data to the cache and amount that is sent, many of which ECS at it's core exploit. One such thing is the tesselation of data in the cache, which simply means how tightly packed the data can be. This is why we always ensure that the data we use in an ECS implementation is simple, small, and in tightly packed arrays. This increases the tesselation ability of a given set of data, allowing us to send more of it to the cache at the same time.

This leads nicely onto a problem that we aim to avoid when using an ECS–cache misses. There are two types of cache misses that we care about: data cache misses and instruction cache misses. In both cases a miss happens when something is requested from the cache but it isn't there yet. Avoiding misses on instructions generally means that you want to ensure that the systems you write are as simple as possible. This ensures that as much of the code related to the system is sent into cache as possible, which makes reading what actions the CPU must perform next far quicker.

Data misses would generally speaking be less impactful than instruction misses, but when the data set scales to larger and larger sizes we start to see increasing performance impacts due to data misses. Avoiding such misses is potentially more complicated for a variety of reasons. There are simpler things we can address such as how much data can be sent to the cache in a given call, which is why we try to ensure data can be packed together as tightly as possible. There's a much more complex and important problem we face however, and that's cache pre-fetching[5]. The reason why this gets so complicated is because of variance in CPUs. You see, normally we might talk about how broader architectures differ such as comparing x86_64 (which is mostly used in desktop computers) to the ARM family of architectures (which is mostly used in mobile/low power computers). However when we start talking about aspects of CPUs that are as low level as how we load data into the cache, we then need to start discussing differences in CPU manufacturers. You see while the specs of two CPUs, one from Intel and one from AMD, might look the same on paper, they can perform wildly different in different scenarios. This comes down to the implementation details of a CPU, decided by the engineers that designed the CPU. Of interest to us is the method of cache pre-fetching, because this indicates to us how we might want to write our code in order to make sure that data is reliably dispatched to the CPU. Admittedly, this does get into the realm of crazy low level optimisation so most programmers will not need to think about this. However, when your cache optimisations don't behave as you expect them to then this is something that you should bear in mind as a potential reason for the unexpected behaviour.

A final thing I'll mention is that ECS lends itself particularly well to a compiler optimisation called inlining. The idea behind inlining is simple: move the contents of a function call to where the function should be normally be called from. I'm of course glossing over this massively, but this is the general idea behind it, and lets us understand the kind of performance benefits that it can bring. At a low level, functions don't exist in the way that we think of them as existing. This is because assembly, the higher level abstraction of raw machine code, generally only has a very bare bones selection of instructions at it's disposal. This allows it to do everything that we expect our computers to do, but does mean writing code can be difficult due to how verbose and difficult to maintain it can be (which is why languages such as C first came to be). So functions don't really exist in assembly, but we do have ways around this so that we can still reuse generic code for multiple purposes. This is done with a

[5]This is something that I'm still not entirely clear on. This lecture provided a good starting point however: http://home.eng.iastate.edu/ zzhang/cpre581/lectures/Lecture17-1p.pdf

structure implemented on the hardware (though sometimes you find software implementations too) called a stack, which is pretty much the same as something like the stack collection found in the C++ standard library. At a high level, when we wish to call a function we will push the current memory address found at the program counter onto the stack. Then we will jump to the address of the function, execute the code, and then upon returning we will pop the address off the stack and set the program counter to that address. This works really well (and might I add, is kinda cool) but comes with associated overhead. If you're constantly calling functions or methods then you run into the issue of constantly jumping to arbitrary memory addresses, pushing and popping off the stack, and some other things I've not mentioned for the sake of simplicity. So by inlining a function call, we now end up increasing the speed by virtue of removing overhead associated with calling a function.

There are, of course, caveats. Inlining isn't some pancea that can solve your performance issues. In fact, it's generally a fairly niche usecase where inlining can improve performance by a significant amount. This is compounded by the fact that inlining manually, if you're not careful, can have knock-on effects which negatively affect your code. Generally speaking, programmers will simply allow the compiler to automatically inline what it deems as reasonable (and indeed, they're all pretty good at that these days). This all does come at a cost, and that cost is space. Due to the literal duplication of code, you'll very likely see some amount of increase in size of your compiled binary. This generally might not be an issue on modern PCs, but is all the same something to consider depending on the application you're writing for (eg. if you're writing a demo for some niche hardware then size will become very important).

5.3.2 Usability and Maintainability

At a high level, it can look like there are many parallels with Architecture B when it comes to the usability and maintainability. If we wish to add new functionality, then all we really need to do is add new components and systems to act on them–not dissimilar from how we add functionality through a component pattern based architecture. And indeed, compared to something like Architecture A I would argue that it is easier to add functionality to, especially at scale. The problem, however, lies in the fact that ECS is really designed for large scale architectures. The entire idea behind ECS is performance at scale, which is something we absolutely see when compared directly against all other architectures discussed in this report. When applied to smaller scale applications it still absolutely is performant, but it becomes less reasonable to maintain. This is because before one is able to create game logic using ECS, you must implement the supporting architecture.

Not only this, but since the key advantage it has over it's contemporaries is the performance you need to be constantly vigilant of how you've implemented features. An excellent example of this lies within an early attempt at optimising my implementation using Austin Morlan's solution for ensuring all data is tightly packed. In their implementation, they created a template class that could hold a generic collection of any type and would always ensure tight packing of data[6]. In theory this would be an excellent boost to speed since we can now always ensure that the data within the collection is tightly packed, allowing us to send more data during pre-fetching. However in my testing this ended up making my ECS implementation slower than every single other architecture tested, and by a very significant margin too. I've some thoughts on why this might have happened, but I ultimately think the largest hit to performance is the indexing of an unordered map. While it is absolutely possible that we've aided the pre-fetching prediction algorithms (though there are some arguments to the contrary) there is now added computation time from attempting to index an unordered map. In the pursuit of optimising cache usage, we have added an extra burden on the CPU which ultimately outweighed what benefits were gained. It's this kind of thought and care that is required when writing your own ECS implementation that can ultimately make it a hard architecture to write and use, at least initially. In theory most of this should be a non-issue once robust implementations are in place, but to get to that point is rather difficult.

5.4 Architecture D

5.4.1 Performance

No performance was measured for this architecture.

5.4.2 Usability and Maintainability

Since this is more or less the same as Architecture C, many of the same points apply here. There is one very key drawback to this though, and that is the complexity introduced through interacting with low level graphics APIs. Being as low level as it is, there's already a level of verbosity and technicality that makes it difficult to work with unless you have much prior knowledge. For example, you have to understand how the OpenGL state machine operates, how textures are created/used in the state machine, how to dispatch compute programs, how to appropriately designate work group size, how to batch dispatches, etc. This is all already rather difficult, but to make matters worse you also get very little in the way of feedback from your shader programs bar the input and output. It makes for not only a complex architecture, but one that is incredibly hard to debug due to the nature of what backs it.

This also feeds into another issue with the usability and maintainability of this architecture once we add compute shaders to it. You see, previously we had spoken about the kind of challenges one would face when creating and improving upon an ECS implementation, and of course the same challenges apply here. There is a new challenge that arises however, and is actually what stopped me from being able to complete the architecture for this report. There

[6] I won't go into detail here as to how it works for the sake of brevity. If you're interested, however, you can find my interpretation at https://github.com/pondodev/research-project/blob/a0a75d5856a007baaeec22d9ce82fcad918fb5ee/program/architecture_c/component_container.h

is a long and complex set of tasks you need to complete in order to prepare OpenGL to process arbitrary data through compute shaders, but of note to us is the preparation of input/output buffers and scaling work group sizes during runtime.

There are a variety of methods that one can use to send and retrieve data from a compute shader, but for the sake of simplicity I have opted to use textures. This is mostly because there are already many abstractions that exist within OpenGL to work with textures which makes the writing and blitting to and from them incredibly trivial. Textures do however have a maximum size that they can be (my running theory is because of 16-bit addressing/indexing, but I can't be sure) which means that you can only send so much data at a time to and from the GPU. To add to this, compute shaders also suffer from the same problem (again, likely because of 16-bit addressing/indexing) when defining work group sizes. This does mean that neither one of these aspects limits the other, which is some cold comfort, but does mean we now need to implement dynamic batch dispatching.

The idea behind this is simple: dispatch data and the program in subsections of the larger data set so that everything can be processed without extending past the technical limits of the hardware and software used. Add to this that you need to be incredibly smart about what you do/don't do, because most any operation in this chain that makes up a compute shader bears a heavy amount of overhead. Because of this, if you're not careful then you may end up with worse performance, therefore defeating the purpose of implementing compute shaders in the first place.

So in summary, the theory tells us that compute shaders should increase performance by a significant amount over a traditional CPU implementation. Whether or not this is the case, unfortunately I have been unable to test. What can be said for certain though is that by introducing compute shaders you increase architectural complexity by a very, very significant amount. It's for this reason that I see it as a very nuclear solution–that is, it should not be your first option. Turn to this if you find that this is the only option left, because if you are to delve into this rabbit hole then know that there be dragons ahead.

6 Conclusion

Perhaps this is not the most exciting conclusion, especially for those who feel most passionate about any one of these architectures. I do however strongly believe that there is no single "best" architecture design, and my findings only further reflect this. Not one of these designs casts a wide enough net over each use case and problem to allow me to point at it and say "this is the one". Instead, I would posit that each of these designs compliment each other. Each can account for another's shortcomings, and some may even serve as a way to assist the design of others.

So ultimately, it depends. Refer to the analyses of each architecture design to understand what may fit your needs best, and whether you may need to instead combine two or more together to create a solution that works best for you.