COS30031 Games Programming
Research Project Report

Daniel Coady 102084174

18/10/2021

Contents

Abstract

Put the abstract here once we actually have one!

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, 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,

[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

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 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.

```
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

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 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

[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

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.

3.4 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

[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

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

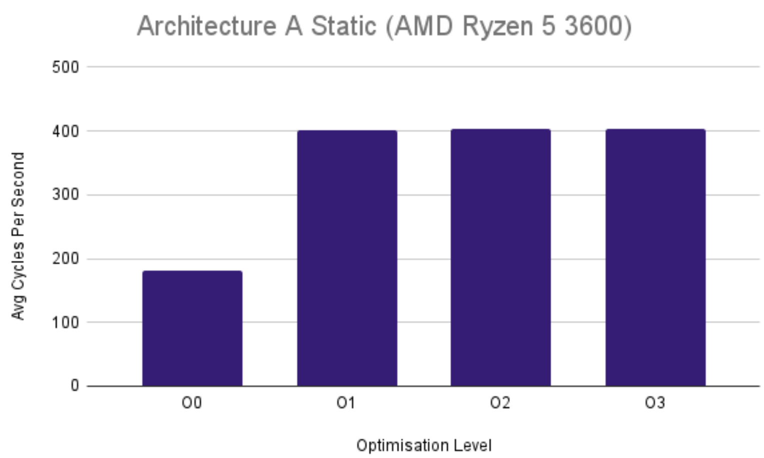

Figure 11: Architecture A Static Test (AMD Ryzen 5 3600)

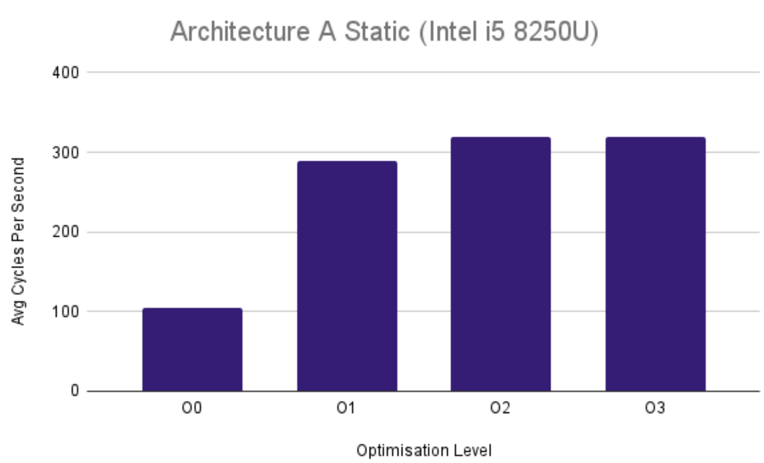

Figure 12: Architecture A Static Test (Intel i5 8250U)

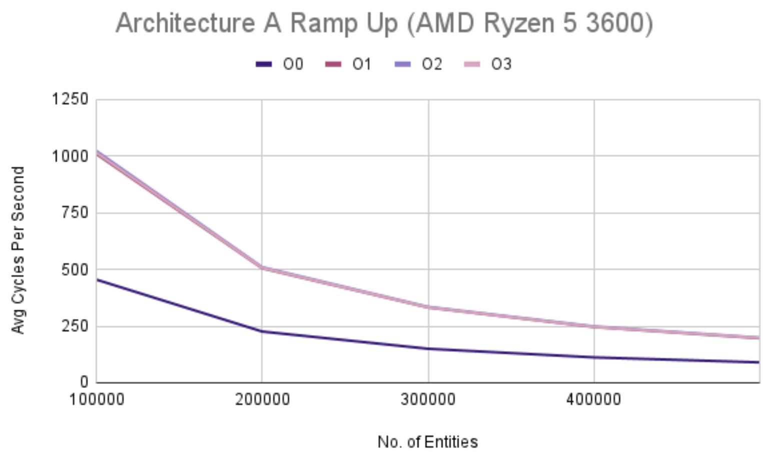

Figure 13: Architecture A Ramp Up Test (AMD Ryzen 5 3600)

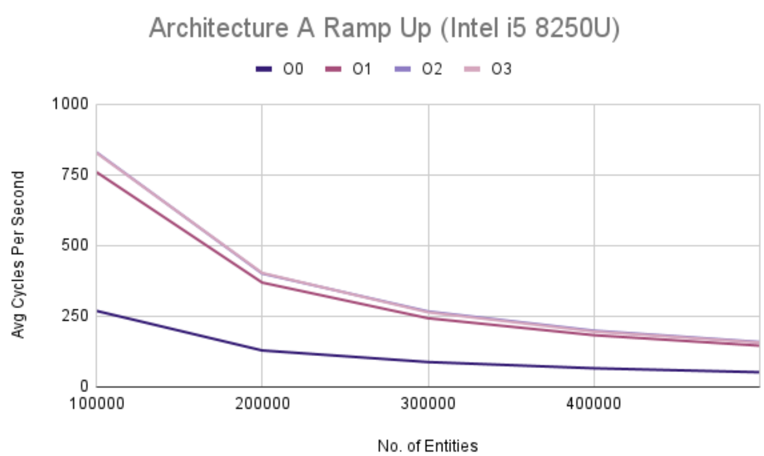

Figure 14: Architecture A Ramp Up Test (Intel i5 8250U)

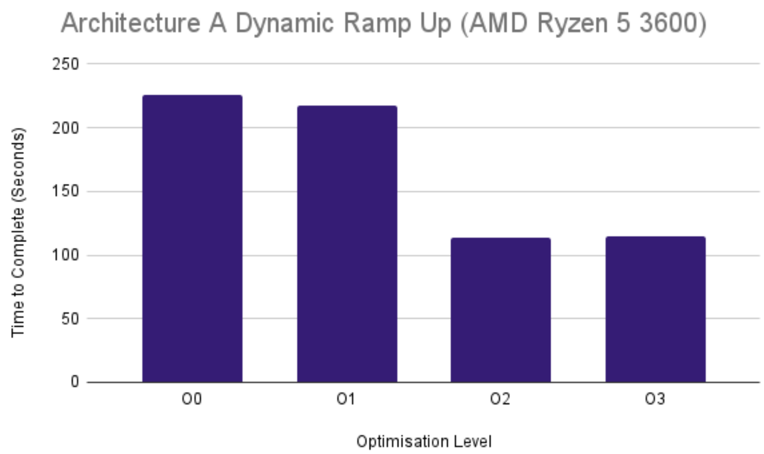

Figure 15: Architecture A Dynamic Ramp Up Test (AMD Ryzen 5 3600)

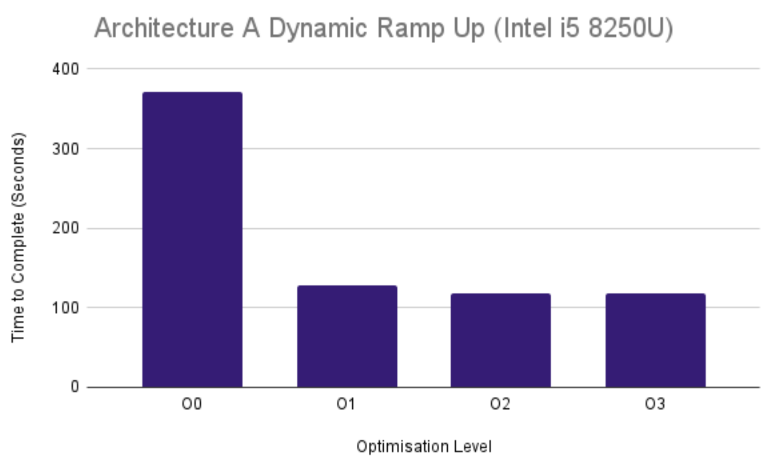

Figure 16: Architecture A Dynamic Ramp Up Test (Intel i5 8250U)

4.2 Architecture B

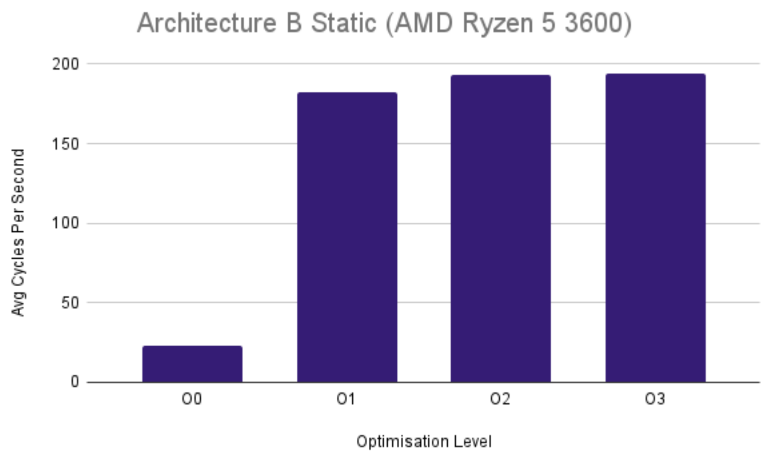

Figure 17: Architecture B Static Test (AMD Ryzen 5 3600)

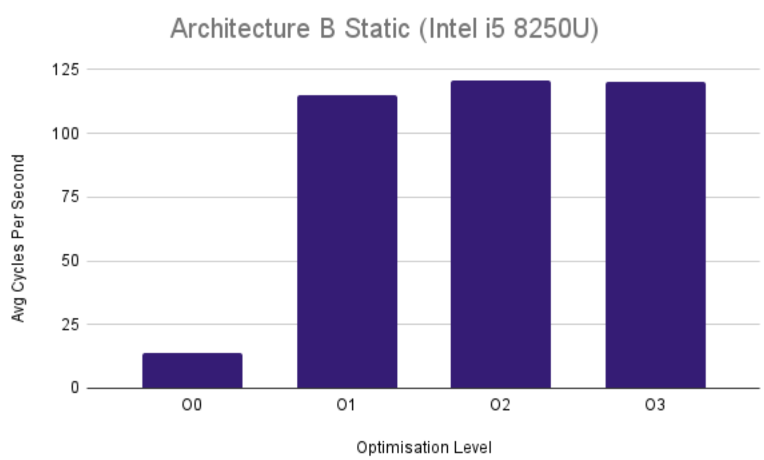

Figure 18: Architecture B Static Test (Intel i5 8250U)

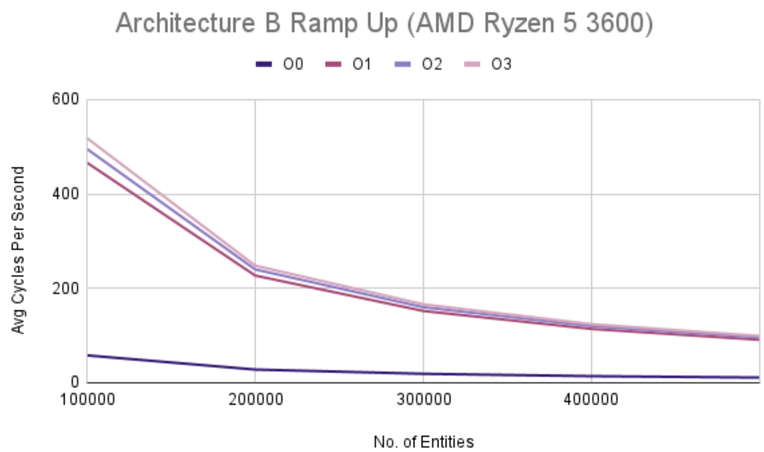

Figure 19: Architecture B Ramp Up Test (AMD Ryzen 5 3600)

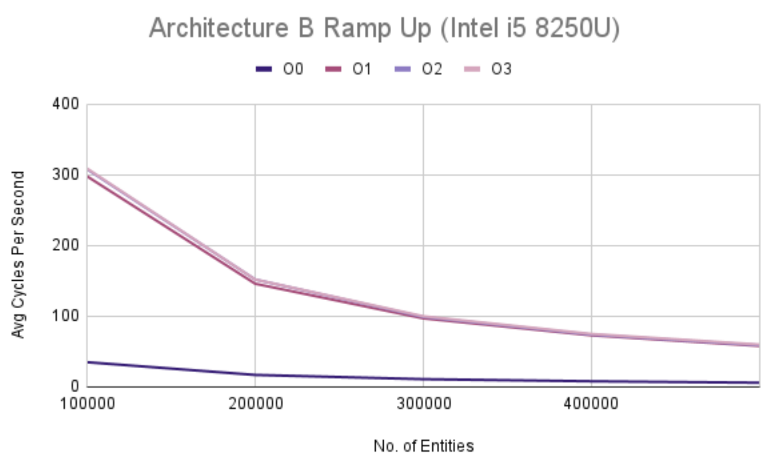

Figure 20: Architecture B Ramp Up Test (Intel i5 8250U)

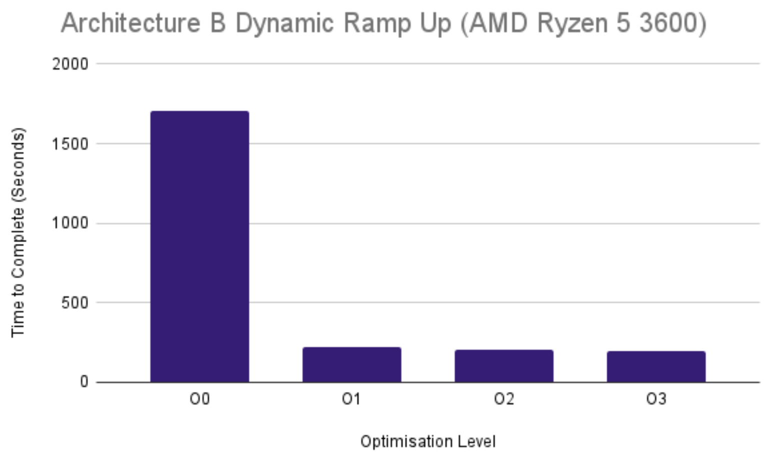

Figure 21: Architecture B Dynamic Ramp Up Test (AMD Ryzen 5 3600)

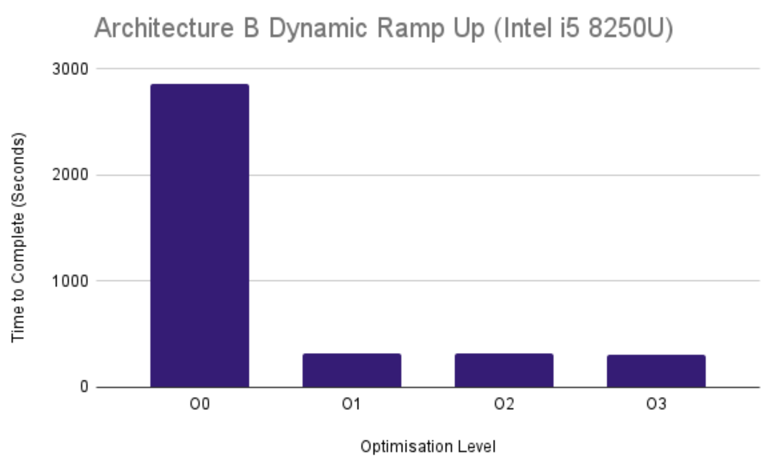

Figure 22: Architecture B Dynamic Ramp Up Test (Intel i5 8250U)

4.3 Architecture C

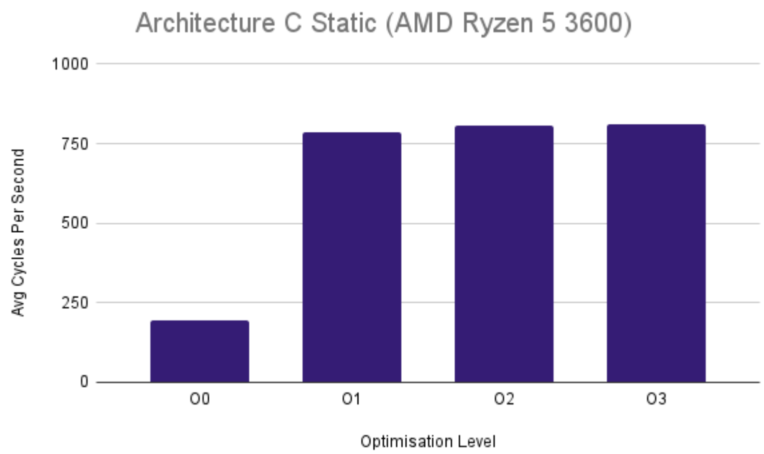

Figure 23: Architecture C Static Test (AMD Ryzen 5 3600)

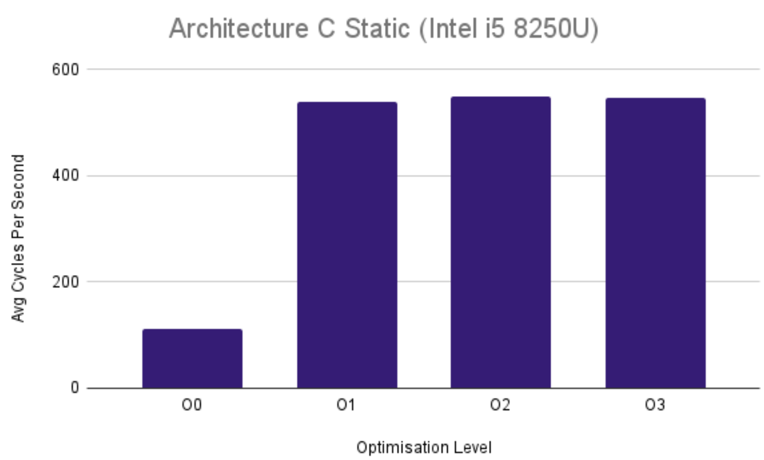

Figure 24: Architecture C Static Test (Intel i5 8250U)

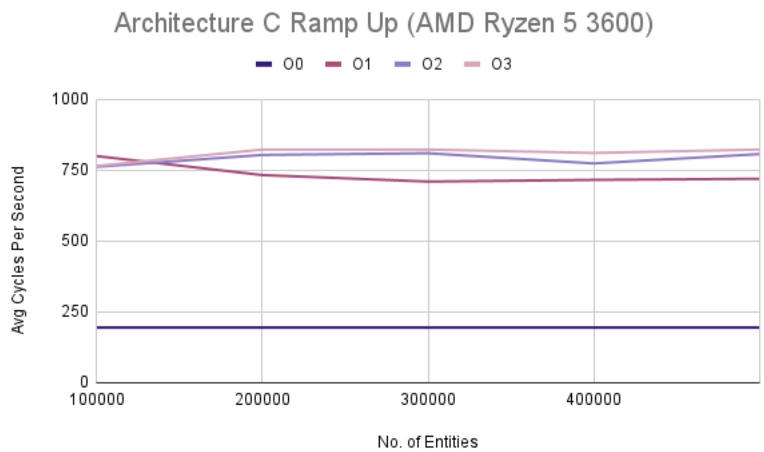

Figure 25: Architecture C Ramp Up Test (AMD Ryzen 5 3600)

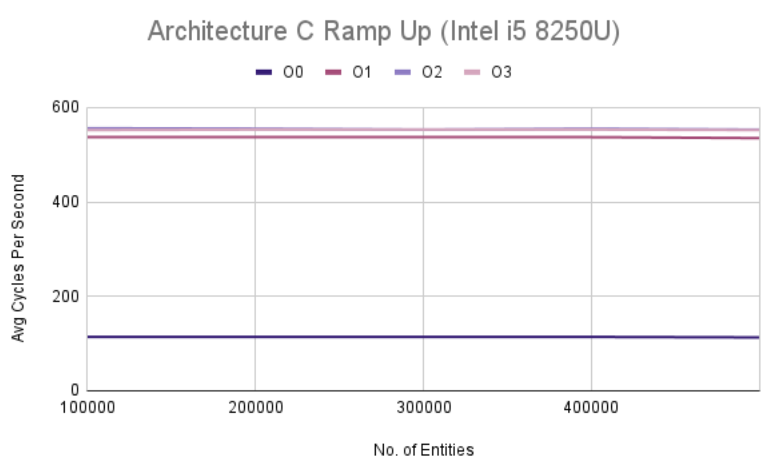

Figure 26: Architecture C Ramp Up Test (Intel i5 8250U)

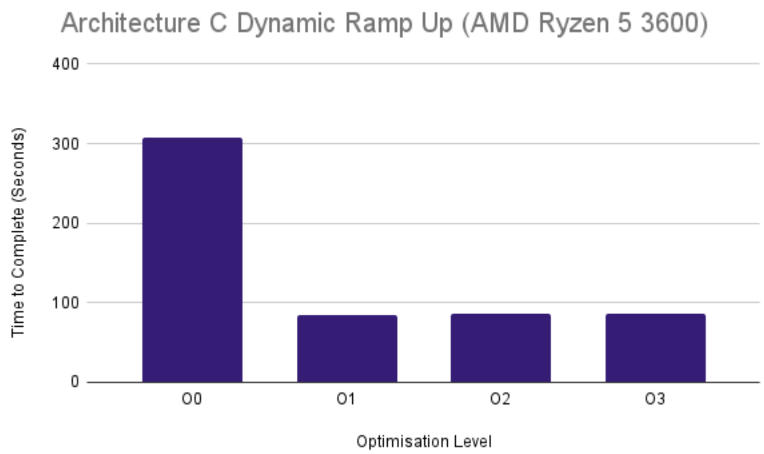

Figure 27: Architecture C Dynamic Ramp Up Test (AMD Ryzen 5 3600)

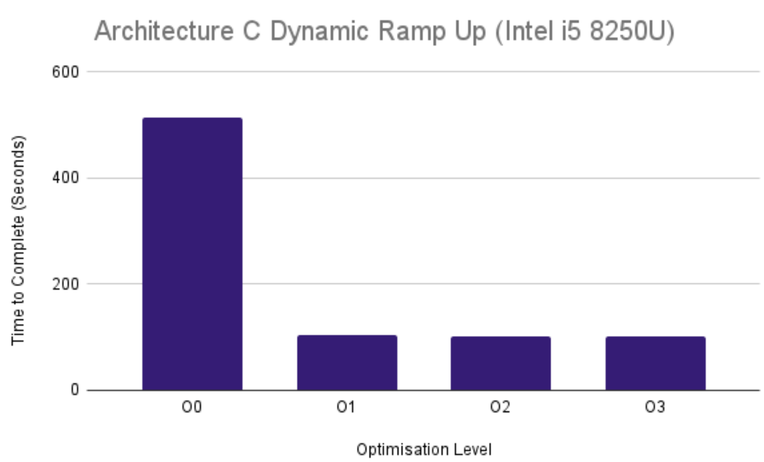

Figure 28: Architecture C Dynamic Ramp Up Test (Intel i5 8250U)

4.4 Architecture Comparisons

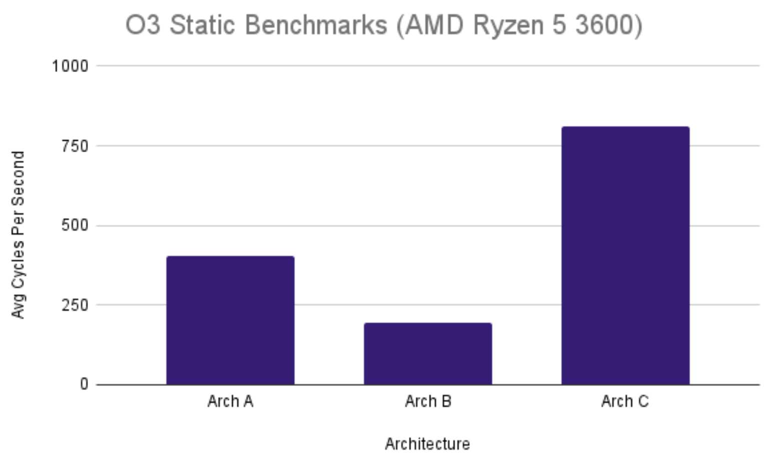

Figure 29: Static Tests (AMD Ryzen 5 3600)

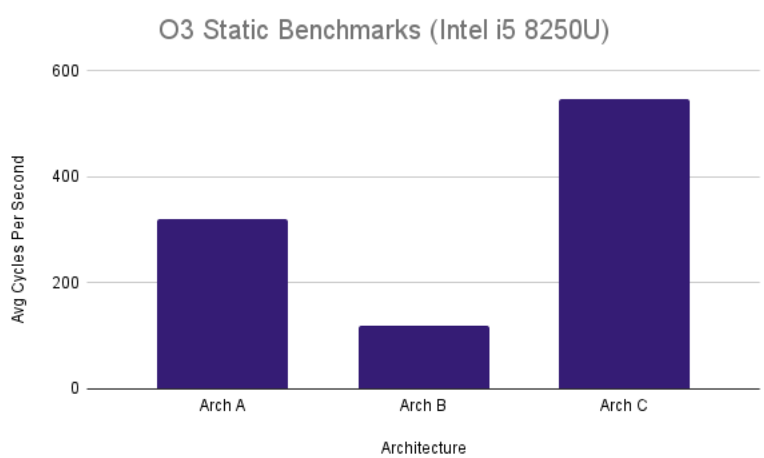

Figure 30: Static Tests (Intel i5 8250U)

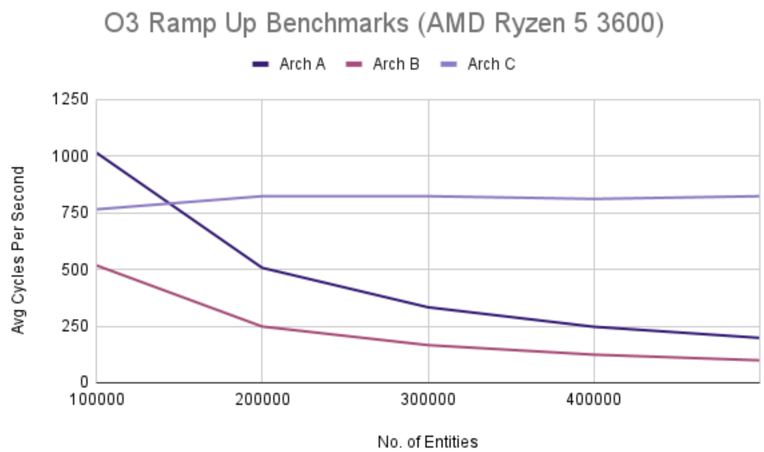

Figure 31: Ramp Up Tests (AMD Ryzen 5 3600)

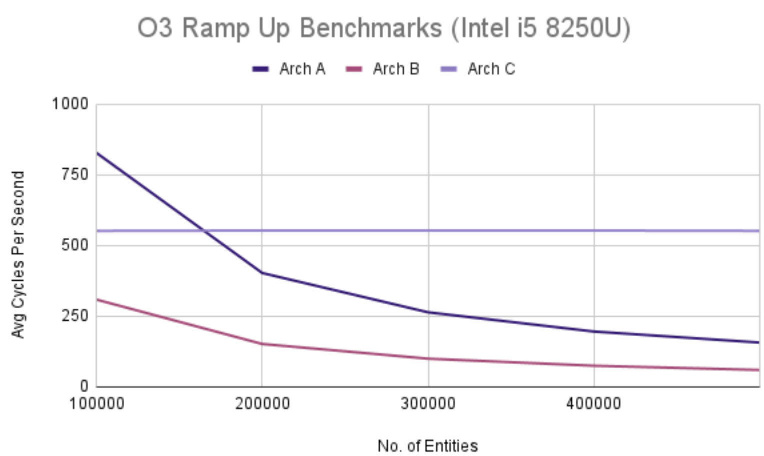

Figure 32: Ramp Up Tests (Intel i5 8250U)

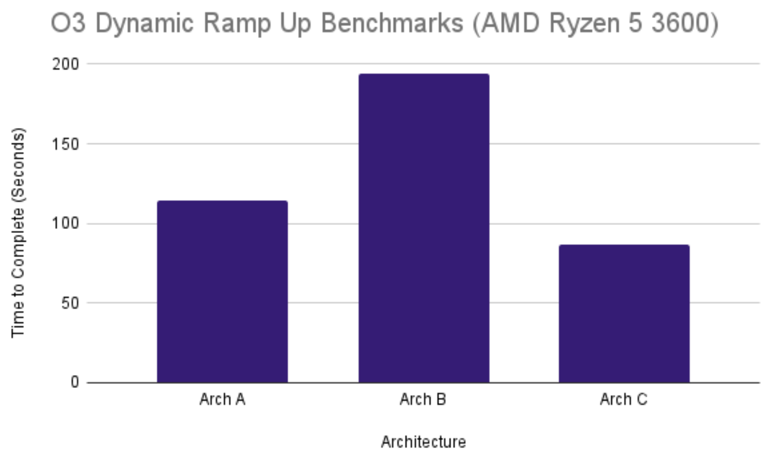

Figure 33: Dynamic Ramp Up Tests (AMD Ryzen 5 3600)

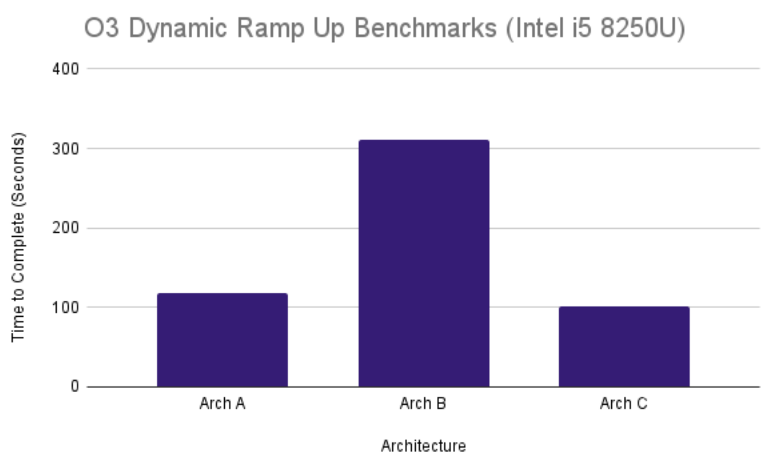

Figure 34: Dynamic Ramp Up Tests (Intel i5 8250U)

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

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.

5.2.2 Usability and Maintainability

5.3 Architecture C

5.3.1 Performance

5.3.2 Usability and Maintainability

5.4 Architecture D

5.4.1 Performance

No performance was measured for this architecture.

5.4.2 Usability and Maintainability

6 Conclusion