UNIVERSITY OF SUSSEX

CANDIDATE NO. 146782

BSc COMPUTER SCIENCE – INFORMATICS

RUST AS A GAMES PROGRAMMING LANGUAGE

SUPERVISOR: DR. MARTIN BERGER

2018/2019

Declaration

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signed:

Govind Venkatesh

Date:

27/04/19

Acknowledgements

I would like to thank Dr. Martin Berger for his help and guidance throughout the project and for also encouraging me to pursue what I really want in my future career.

0 Summary

This project aims to explore the viability of Rust programming language as a games programming language. This is accomplished through an analysis of the development process and the creation of a sub-module of a game engine in Rust. It documents the research behind learning Rust and implementing such a sub-module (called Entity Component System). The implementation in Rust is explained and compared with modules written in C++ (the industry standard) as an anecdotal measure of performance. Finally, after a discussion and comparison of the results, it is found that Rust is a good language for small to medium sized games but not so much for big budget games.

1 Contents

Rust as a Games Programming language

1 Introduction

The video games industry is currently dominated by one language, C++ [1]. Its speed is unparalleled and fine-grained low-level control is difficult to match, but along with these benefits come some caveats. A less than strict type system, focus on Object Oriented Programming (OOP), and difficult concurrency cause many programming nightmares such as the much feared "undefined behaviour" (UB), when developing large applications such as video games (UB is when the semantics of a programming construct are left intentionally loose [2]). More recent iterations of C++, namely version 11 and 14, have introduced more high-level features such as type inference and lambdas. Despite these new additions, the underlying language does not become any less cumbersome to use. In one sense, it can be said that C++ is no longer fit for service. To help shift the focus of the industry from C++ to another language, substantial work must be done to prove that such change will be beneficial [3].

The reason for the widespread use of C/C++ is their speed and efficiency due to their lack of garbage collection. These languages, however, irrespective of the level of programming experience of the developer, make for a difficult debugging experience. This is due to segmentation faults and other forms of badly formed access to memory resulting in undefined behaviour. Although UB is required to make a language flexible enough to compile to different CPU architectures, debugging UB is time consuming, and time is a precious resource when developing such large-scale software. Furthermore, C, being procedural and also lacking many libraries for data structures, requires a lot of bootstrapping to get started and C++, being an Object-Oriented Programming (OOP) language, makes it difficult to make good, easy to understand abstractions.

1.1 The project

Rust, developed at Mozilla Research, is intended to be a language for highly concurrent and highly safe systems. "Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety." – Mozilla [4].

The focus of this project is to explore the feasibility of the Rust programming language in the context of developing video games. The objective is to determine if Rust could provide a smoother programming experience when developing games and game engines, and at the same time promote better coding practices than C++, the current industry standard.

1.2 Objectives

The main objective will be to use Rust to create a small game engine or sub-component of a game engine, thereby providing an experience of programming such systems in Rust, which can be compared with the experience of programming in C and C++. Furthermore, this implementation will be benchmarked to other solutions in Rust and also benchmarked against solutions in C/C++ for a quantitative comparison on any efficiency gain/loss, albeit anecdotal.

1.3 Structure

The following sections of the report will cover various aspects of the project in the following order: Professional Considerations, Requirements Analysis, Background Research, ECS Implementation, Benchmarking and finally a Conclusion.

2 Professional Considerations

2.1.1 Public Interest

The use of the Rust Language and the creation of an Entity Component System (ECS) keeps the public interest in mind. The process by which the ECS will be created will not threaten or discriminate towards people regardless of their sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability or of any other condition or requirement. Neither the ECS nor the Rust language pose any threat to the public's health, privacy, security and wellbeing [5].

2.1.2 Professional Competence and Integrity

The knowledge and findings within this paper will be provided within the domain of Computer Science and no additional knowledge outside this domain will be claimed to be accurate. The findings presented will be the most up to date at the time of the publishing of this paper, as during the investigation, which is an iterative process, updates will be made alongside the development of the ECS in Rust. Furthermore, the viewpoints of Rust developers in industry will also be considered and explored, in a way which will not bring them any harm or injury to their reputation. [5].

2.1.3 Duty to Relevant Authority

All third-party code used and drawn inspiration from will be accredited correctly within this paper and within the code base of the project. A clear distinction will be made from extrapolations and improvements made from existing solutions if any. Any information gained during the project will be presented solely with the authority of the people from whom the information was obtained. All information will be presented even if it does not support the hypothesis of the investigation. There will be no discrimination against information [5].

2.1.4 Duty to the Profession

No disrepute will be brought to the people whose works and opinions have been used. The work will be conducted in a professional manner. Opinions will be considered equally, even if they were to oppose each other, as this will more accurately represent the language and the various benefits and downsides of using it for games engine programming.

It may be necessary to get the opinions of professionals in the field. These will be obtained through online sources, blogs or interviews and referenced appropriately [5].

3 Requirements analysis

3.1 Project Requirements

The requirements for proper execution of this project are as follows. Code will be version controlled through Git and stored remotely with GitHub. Git will allow for branches, making it easier to track working code and "work in progress" (WIP) code. Pull requests will be made to merge these branches into master once they are completed and tested, ensuring that only working code exists in master. Travis CI will be used as a continuous integration tool (CI) to automate this process. CI is useful when working on multiple features and having tests running on them independently, saving time and smoothening out the workflow. [6] [7] [8]

The objective is to implement an Entity Component System (ECS, game engine submodule used for memory management) in Rust, to gain experience in using this language and to also benchmark the implemented ECS against C/C++ as well as other Rust implementations.

The benchmarking will be done using Criterion-rs which is a Rust benchmarking library able to generate automated reports with many kinds of statistical analysis. This library is based on a Haskell benchmarking library called Criterion. This tool will allow for the ECS to be quantitatively evaluated against other implementations. [9]

4 Background Research

4.1 The current state of industry

"The games industry is dominated by C++" – Mike Acton [10]

The reasons for C++ becoming the industry standard language are its capability to allow one to express very low-level control, a thin runtime environment and cultural influence as well as a lack of competing languages. Other candidate languages like C, C#, Java, Python, and Lua are used in some ways in the games industry for tooling and scripting, however, these are not used for implementing core features. The reason for this is their heavier runtime environment, higher level expressiveness (preventing low level access) and slower speeds [11].

4.2 The Rust Language

4.2.1 Rust primer

This section gives an overview of the Rust programming language. The main novelty in Rust compared to C++ is in its memory management and typing system, which are described in Section 3.3 . The following section highlights the basic features of the language.

4.2.1.1 Basic Syntax

4.2.1.1.1 Mutable and Immutable

Rust provides the concept of mutable and immutable variables. Mutability means that a variable can be changed after being initialized, on the contrary, immutability prevents change after initialization. The primary reason for this is for aliasing control. Aliasing is when variables and pointers overlap the same area of memory. This is an important factor to consider when optimising a program. A mutable variable (marked with "mut") is only allowed to be accessed mutably and an immutable variable is not allowed to be accessed mutably, preventing the scenario where there are two references to an area in memory, one mutable and one immutable [12].

4.2.1.1.2 Control Flow

Rust has some basic control flow constructs such as "if" statement, "if then else" and "if then else ifs". As expected, this code, being fairly standard and common to many other languages is handled in similar ways.

```
pub fn if_statement(){
    let mut x;
    if true {
        x = 1;
        println!("x is always assigned 1");
    }else{
        x = 2;
        println!("x is never assigned 2");
    }
}
```

Figure 1: If statement

```
pub fn complex_boolean(x: i32){
    if (x > 1) && (x < 3) {
        print!("x is 2");
    } else if (x > 0) && (x < 2){
        print!("x is 1");
    }else{
        print!("x");
    }
}
```

Figure 2: if and else statements

```
pub fn if_then_else(x: i32){
    let y = Some(x);
    if y == Some(0){
        println!("x is zero");
    } else if y == Some(1) {
        println!("x is one");
    } else if y == Some(2) {
        println!("x is two");
    }else{
        println!("x is greater than 2");
    }
}
```

Figure 3: if then else-if

It can be noticed from the code snippets that it is not always necessary to enclose the guard in brackets. This is an approach found in functional programming languages, but it does not work as consistently in Rust as it does in Haskell for example. When introducing complex Boolean statements, bracketing helps the Rust compiler to work out the ordering and nesting of the statements.

In addition to conditional control flow, there are also different kinds of loops.

```
pub fn for_loop() -> i32{
    let vector = vec![0, 1, 2, 3, 4];
    let mut sum = 0;
    for element in vector {
        sum += element;
    }
    sum
}
```

Figure 4: for each loop

In the above code snippet, a “for each” loop is encoded via a “for” keyword followed by a variable to represent the element currently being accessed “in” the collection provided, which can be an Iterator or something that can be transformed into an iterator, such as a Vec data structure.

```
pub fn standard_for_loop() -> String{
    let mut ah = String::new();
    for _i in 1..100 {
        ah.add_assign("a");
    }
    ah
}
```

Figure 5: for loop

To achieve the semantics of a "for" loop as found in Java or C++, a range can be used. The _ found before the variable name in this case causes the compiler to ignore it, essentially marking it as unused. The body of the loop in this case is simply iterated x number of times as defined by the range syntax "1..n".

```
pub fn while_loop(){
    let mut index = 0;
    while index < 100 {
        println!("index: {}", index);
        index += 1;
    }
}
```

Figure 6: while loop

"While" loops work exactly as expected where a guard is provided and the body is iterated until the guard becomes false. This control flow is very similar to C++ except it again can forgo the bracketing in simple Boolean statements as seen with the "if" statements.

```
pub fn pattern_match(){
    let result = Some("hello");
    match result {
        Some(st) => println("{}", st),
        None => println("bye")
    }
}
```

Figure 7: pattern match statement

One of Rust's seeming high-level features is the "match" statement. This is the most powerful control flow operator and allows one to match some data against a set of defined patterns. In the case above, we can see the use of this against an "Enum" type "Option". The type Option enumerates two possibilities, a Some(x), indicating the presence of data and a None type indicating the lack of presence of data. When paired with a "match" statement, it is possible to define what should be done in the case of Some(x) and None (or otherwise). This pattern matching is also extremely deep and can even pattern match on the data stored within the Some as shown below [13].

```
pub fn complex_match(){
    let result = Some("hello");
    match result {
        Some("hello") => println!("good bye"),
        Some("good bye") => println!("hello"),
        _ => println!("???")
    }
}
```

Figure 8: Complex pattern match statement

This sort of pattern matching is commonly seen in Haskell and other functional programming languages but is also present in Rust. This is a high-level feature embedded elegantly within a low-level systems programming language. The approach taken within Rust is efficient due to its LLVM backend compiler infrastructure.

4.2.1.2 Traits, Generics and associated types

```
pub trait HelpfulFunctionality{
    type my_type;

    fn some_func(&mut self) -> Self::my_type;
}

struct ConcreteImplementor<'a>{
    my_val: &'a String
}

impl <'a> HelpfulFunctionality for ConcreteImplementor<'a>{
    type my_type = &'a String;

    fn some_func(&mut self) -> Self::my_type {
        self.my_val
    }
}
```

Figure 9: Generics and Associated types

In Rust, reusable functionality is packaged up in what is called a “Trait”. Traits function similarly to Interfaces in that there can be no unimplemented methods (those defined without a body in the interface). An interface can also contain methods with a body that can be invoked from any concrete type. However, unlike in OOP where generics can inherit functionalities of other generics, in Rust, this is not possible as structs (4.2.1.5) are the only constructs that can inherit functionality and only from traits, preventing a deeply nested inheritance hierarchy.

In addition to this, the code snippet above shows an associated type by the “type Item;” declaration within the trait. This is an implementor defined type and can be used internally within the trait as function parameter or return types.

4.2.1.3 Closures

```
pub fn closure(x: i32, y: i32) -> i32{
    let add = |x: i32, y: i32|{
        x + y
    };
    add(x, y)
}
```

Figure 10: Closure

In the above snippet a closure is defined and stored in a variable. A Closure is a concept taken from functional programming as it is commonly referred to as ‘functions first’. These closures can be passed as arguments to functions as well, allowing for increased modularity and reusability. Furthermore, closures are used within the definition of threads in Rust.

```
pub fn thread_and_closure(x: i32, y: i32){
    let add = |x: i32, y: i32|{
        x + y
    };
    let handle = thread::spawn(move || add(x, y));
    handle.join().expect("thread execution error:");
}
```

Figure 11: Closure in a thread

A thread is created as depicted above; when threads are created, they are given a closure. This closure has the ability to capture the variables that are in its parent scope, and this can be seen by the move keyword in the snippet above. The interesting aspect of threads in Rust is how it guarantees that the thread does not live longer than its parent thread. This can be made explicit by the use of the “join” keyword, which waits for the thread to finish execution. Furthermore, when the thread references a variable (not taking ownership), it must be ensured that the references held by the thread do not expire before the thread finishes execution. These are guaranteed by the Borrow Checker, as will be highlighted in the next section of the report (4.3.1).

4.2.1.4 Pointers

Rust provides several useful smart pointers in its standard library. The ones used widely in this implementation are, “Box”, “RefCell” and “RwLock”.

```
pub fn box_usage() -> Box<i32>{
    Box::new(1000)
}
```

Figure 12: Box usage

Box is a very simple pointer that points to data stored on the heap. Its use is common in many programs since not all types are sized at compile time.

```
pub struct IMutateMyself{
    pub data: RefCell<i32>
}

impl IMutateMyself {
    pub fn succ(&self){
        let mut mutable_borrow = self.data.borrow_mut();
        *mutable_borrow += 1;
    }

    pub fn pred(&self){
        let mut mutable_borrow = self.data.borrow_mut();
        *mutable_borrow -= 1;
    }
}

pub fn increment_three_times(immutable_data: IMutateMyself){
    let data = immutable_data;
    data.succ(); //this would be dissalowed if interior mutability did not 
exist
    data.succ(); //the immutable value data is mutated three times by itself 
    through the use of refcell
    data.succ(); //externally, this location of data cannot be manipulated, it 
can be through the use of its own
                 //functions.
}
```

Figure 13: RefCell usage

RefCell is a pointer that makes use of “unsafe” Rust code (Rust with some of the Borrow Checker analysis turned off), forcing the Borrow Checker to run at runtime rather than compile time, allowing for a pattern called “interior mutability”. This design pattern allows for an immutable reference to mutate itself and is incredibly useful when a data structure is immutable externally but can mutate itself [14].

4.2.1.5 Common Data Structures

```rust
pub fn data_structures(){
    let map: HashMap<String, String> = HashMap::new(); //hashmap creation

    let vector: Vec<i32> = Vec::new();//vector creation

    let array = [10, 20, 30]; //fixed size array

    let tuple = (10, "hello");//tuples
}
```

Figure 14: common data structures

The most widely used data structures in Rust are HashMap, Vec and Tuples.

Vec represents a contiguous, indexed, dynamically allocated space of memory, much like a Vector in C++. Arrays can also be used; however, they are statically allocated.

HashMaps function as expected like in any other language. The default hashing function used is “SipHash” which is cryptographically secure. However, this is a slow hashing function and can be swapped for something faster [15].

Tuples are commonly found in functional programming languages such as Haskell and Scala. These data structures are useful when trying to describe a pairing of two types. In this sense, it is a composite data type that composes two or more types into a new one. They are useful when grouping elements together.

```
pub struct Name<'a>{ //lifetime guarentees that the string reference will live
as along as this struct
    pub field1: &'a str,
    pub field2: i32,
    //more fields ...
}

//impl blocks for this struct require a lifetime specifier, to help the borrow
checker how long references live
//declare lifetime next impl in angle bracket for use
impl<'a> Name<'a> {
    pub fn new() -> Name<'a> {
        Name{field1: &"hello", field2: 10}
    }

    pub fn change_str(&mut self, new: &'a str){
        self.field1 = new;
    }
}
```

Figure 15: structs and implementation blocks

The most common data structure in Rust is a struct. A struct is very similar those found in C, however, here there is a clear separation of functionality and data. Structs are treated as a grouping of data where each component type is given a name (field). A struct may also have an "impl" block which associates functionality. These two features are expressed individually creating a clear separation of logic and data.

4.2.1.6 Options

```
pub fn options(){
    let something: Option<i32> = Some(10);
    let nothing: Option<i32> = None;
    matcher(something);
    matcher(nothing);
}

pub fn matcher(input: Option<i32>) {
    match input {
        Some(x) => {/*do something*/},
        None => {/*do something else*/},
    }
}
```

Figure 16: Option data type

An option type in Rust is an enumerated type that is either Some(x) or None. This is a useful type that can be represent the presence (or absence) of data. It is used heavily in the implementation section of this report to signify present and missing data.

4.3 Rust vs C++

Rust and C++ share similarities, as they both aim to empower the programmer to write low-level, fast code. However, even though they have the same goal, they do this in different ways. The novelty behind the Rust approach to writing safe system level code lies in the Borrow Checker, which is described below. This feature does not exist in C++ .

4.3.1 Borrow Checker

The main objective of the Borrow Checker is to ensure memory safety. The Borrow Checker accomplishes this through a set of rules that help keep track of single ownership and references (aliases) to that owned data. These rules also help prevent data races in a concurrent environment [16]. Within the Borrow Checker, to enforce its rules, there is a concept of an “owner”. An owner is a variable that has some data assigned to it. The rules enforce that there can only ever be one owner for data at a time. When data is assigned to another owner the previous one is invalidated. This rule makes it such that a double free (the same memory location is cleared twice) can never occur. Furthermore, the Borrow Checker is lexically scoped, meaning whenever an owner goes out of scope, it and its resources are freed from memory. The advantages of this rule become apparent when using Mutex as it is automatically unlocked and freed when it goes out of scope, preventing common errors when forgetting to unlock a Mutex. To assist the Borrow Checker in analysing the scoping of variables, a term “lifetime” is used. Life time annotations are used to describe to the compiler how long references live for before they are dropped. This is a way to guarantee that the data at the address referenced will be available when accessed, preventing an error called “use after free”.

The following rules are employed by the Borrow Checker [17]:

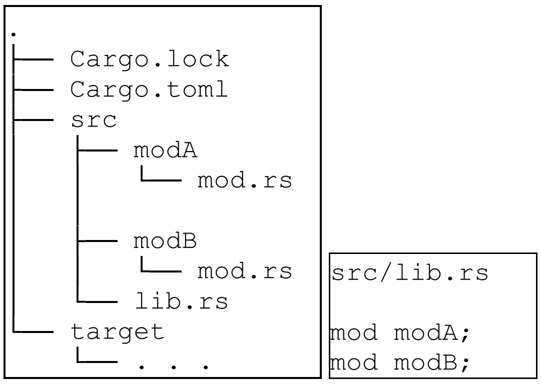
1. Each value in Rust has a variable that’s called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

These rules when put in practice guide the way in which one programs. An “owner” is the first reference to the data on the heap. Every subsequent reference to that data Borrows from the owner. The Borrowing can either be mutable (read/write) or immutable (read only), but both cannot exist at the same time. This is a scenario in which a data race can occur, which will lead to undefined behaviour and Rust will disallow this from happening at compile time. When a variable is moved from owner to owner, the previous owner is invalidated. This stops a bug where a double free can happen. When an owner goes out of scope, the data on the heap that was referenced is freed. This is done automatically by the Rust compiler, when the closing bracket of a scope is reached. Finally, Lifetime annotations, which are elided (implicit), may need to be explicitly mentioned. These annotations help the compiler’s analysis by providing extra information on the scope of variables [17].

Having explained the core difference between Rust and C++, the following sections will discuss the other differences and similarities. The following criteria/sections have been chosen as they provide an overview of both Rust and C++ common features used frequently when programming [18].

4.3.2 Organisational aspects

Workspaces and Modules are concepts that facilitate the development of large systems in Rust. In Rust, code is organised by modules which allow for better readability and maintainability. Each Module has its own namespace, so the variables will not clash when used in other Modules. Furthermore, Rust allows modules to be described as a file hierarchy (see Figure 17). This approach allows for fewer sprawling code files and a neatly organised hierarchy. However, the downside to this approach is that it is not as easy to reorganise the Modules. To counter this, modules allow for re-exports; this means re-declaring the modules in the root of the namespace. In the case of a library, which will need to expose an external API for other programmers to use, a helpful thing to do would be to re-export the useful modules so that they are easier to access, without having to declare a long chain of namespaces [19].



```
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── modA
│   │   └── mod.rs
│   │
│   ├── modB
│   │   └── mod.rs
│   └── lib.rs
└── target
    └── . . .
```

```
src/lib.rs

mod modA;
mod modB;
```

Figure 17: workspace hierarchy and modules

Workspace is another organisation feature in Rust that allows for the definition of sub-projects (crates). A workspace will define interdependencies between different local crates being developed. In the case of a game engine these crates could be the renderer, ECS, Audio engine, etc. This allows for better code separation; having each sub-module of the game engine in its own sub-project means that the code is being designed for modularity. This feature is useful in homebrew/indie engines that require modularity and flexibility as each section of the engine is nicely separated out in its own crate [20].

Furthermore, Rust has three release channels, of which two are predominantly used. The “stable” release channel contains code that has been through rigorous testing and is sure to work as expected, whereas the nightly channel contains new features that are not supported by the stable compiler. Finally, there is a beta release channel which serves as an intermediary step where candidate features from the nightly release channel go to before making it to stable Rust [21].

C++ is mainly described by standards released by the “open-std” organisation. The implementors of these standards produced the conforming compilers, and these are not necessarily open source like rustc (Rust language compiler) [22].

The downside to the C++ approach is confusion due to the competition between implementors. Compilers such as MSVC (Microsoft’s C++ compiler) only targets Windows and other compilers have similar advantages and disadvantages.

4.3.3 Zero Cost Abstractions

Zero Cost Abstractions are abstractions that are not included if not used. An example of such an abstraction are Traits and Templates in Rust and C++ respectively. Traits, like templates can be statically dispatched by having the compiler remove the abstraction for duplication of specialised code [23]. In this sense, Bjarne Stroustrup’s ‘mantra’ that ‘what you do use you cannot hand code better and what you do not use you do not pay for’ applies equally to both C++ and Rust. Due to virtual calls and dynamic dispatch not being efficient enough, should they not be needed they will not be used. While high-level abstractions such as Traits and Interfaces can be used, the compiler will still produce efficient machine code. This is achieved through static, compile time analysis [23].

Even with these Zero Cost Abstractions in Rust, there are edge cases in which a programmer is forced to use and pay for an abstraction unlike in C++. For example, a common difficulty for Rust programmers are circular data structures. The data structures violate the rules imposed by the Borrow Checker, namely “single ownership”. This is due to the first node being owned by the last node and also by the header sentinel or owning variable. Here a programmer is forced to use a Reference Counted pointer and Refcell for unsafe functionality permitting the multiple owners, but with overhead.

On the other hand C++ is more than comfortable to allow a programmer to code for multiple owners, making circular data structures easy to code and all the while upholding Stroustrup’s mantra.

4.3.4 Move Semantics

C++ and Rust tackle ‘move’ semantics in very different ways. Rust’s novelty comes in the form of a built-in static analyser (the Borrow Checker as explained above) that prevents illegal use after move. C++ will not statically check these conditions, but external static code analysers can be used to achieve this. Use-after-move errors are caught at run-time.

4.3.5 Smart Pointers vs Null pointers

Both C++ and Rust prefer the use of smart pointers that are capable of detecting “use after free”, “double free” and “dangling pointer” bugs. C++ does not enforce this rule and will allow users to mix use of the raw pointer and smart pointers. Rust, however, will force users to isolate raw pointers in “unsafe” blocks of code. These blocks of code make it extremely easy to identify where unexpected behaviour originates from, unlike in C++. Furthermore, smart pointers in C++ allow for Null dereference, whereas in Rust, smart pointers return an Option type (4.2.1.6) if the value does not exist, which can be ignored but if handled by a match statement will require an exhaustive pattern match, meaning the “None” case (lack of data) will need to be handled. A null pointer dereference is a bad form of memory access which in C/C++ will produce an exception, crashing the program or in the worst case allowing a hacker to bypass security, potentially causing harm [24].

4.3.6 Buffers

A serious issue in low level programming is a buffer overflow. This is when writing data to a buffer (some sized area of memory) extends beyond the limit of its size. Buffer overflows are possible in C++ code as only wrapper classes on buffers perform range checks such as a Vector (array list guaranteed to be contiguous memory). Rust has a slice (range-checked segment of memory) data type implemented for many data structures which have range checks. In addition to this, Rust promotes the use of Iterators, which perform range checks and iterate through the data buffer. They also provide useful functions such as map and collect to transform the data within the buffer and the buffer itself.

4.3.7 Data Races

Already mentioned above, the difference here lies with the Borrow Checker. C++ requires quite a good level of programming experience to safely implement concurrency. Rust will automatically drop and unlock data when the owning Mutex goes out of scope, preventing many errors that are common to C++ concurrency. Furthermore, the additional checks on “move after use” work in a multi-threaded environment, preventing data races.

4.3.8 Object Initialisation

Rust does not allow for uninitialized variables unlike C++. All variables must be explicitly initialized before use. Further to this, all values have defaults in Rust. C++ however does not have defaults for Primitive types such as integers and requires the user to write constructors for user defined types.

Both C++ and Rust use a technique for initializing variables called “Resource Acquisition is Initialization (RAII)”. The main difference between the Rust and C++ approach comes down to the Borrow Checker and lack thereof. The Borrow Checker can guarantee RAII whereas in C++ it is up to the programmer to make sure that these resources are freed after being used. The Borrow Checker, as explained previously in 4.3.1 provides typed memory management whereas C++ provides manual memory management.

4.3.9 Static Polymorphism

Rust makes use of a feature found in functional programming called traits. Traits allow users to define an interface and provide default methods used to extend structs with functionality. These are referred to Mix Ins in normal OO programming. They provide a way of specifying static and dynamic interfaces. C++ 17 (version released in the year 2017) has a feature called Concepts which provides a similar functionality; however, it is still not included in all compilers and is not fully featured, only included as a “lite” version. Currently, the most common way of achieving this is through Virtual functions and abstract classes where they may be optimized by the compiler.

4.3.10 Adding new traits

Traits in Rust can be added at any point whereas in C++ it is required to inherit the interface and then extend the functionality, which is problematic and is generally avoided in games development. An up and coming feature of C++ would be its unified functional call syntax which allows for an emulation where methods can be extended without the need for inheritance.

The Rust compiler performs some optimizations around the use of traits. These optimizations are done statically (via compile time analysis) where traits are replaced by their concrete implementations. By injecting the concrete implementations of traits, there is less overhead as the code can be called directly. This however, does not happen in all instances. Sometimes the Rust compiler will compile the program to use message passing in order to resolve the correct method at runtime; this is a slower method but does allow for more flexibility.

Going deeper into how Rust handles trait coercion (implicit conversion of type, usually to a more generic type), there is the use of “fat pointers”. These pointers use an additional machine word when pointing to a trait for its vtable. Although this makes the size of the pointer twice as large, it compliments Rust’s method of static trait resolution. In contrast to this, C++ puts the pointer to the vtable in the object rather than alongside the pointer to the object. In this case the pointer to the object is lean but the size of the object grows by one machine word. Had Rust used this method of dynamic dispatch, with its static trait resolution, the sizes of the concrete objects would be larger for no benefit. This is why Rust prefers to use its “fat pointers” rather than the C++ approach of objects carrying pointers to their vtables [25, 26] .

Furthermore, C++ takes after the OOP paradigm which allows for deeply rooted inheritance hierarchy trees. This is bad for maintainability and scalability since a change high up in the inheritance hierarchy propagates as errors in all subsequent levels, making it difficult to change functionality. Rust does not allow for this as explained in section 4.2.1.2.

4.3.11 Switch Statement Branch checking

Rust can determine if a switch statement or pattern match is non-exhaustive. C++ requires external code analysers to spot these situations.

4.3.12 Runtime Environment

Both languages can compile directly to machine code, and they also do not have a garbage collector making the run time environments minimal. It is also possible to exclude the standard library in both languages, making the environment correspondingly smaller.

4.3.13 Foreign Function Interfaces

It is also possible to call C code from both C++ and Rust without any overhead. Other languages require wrappers to be written.

Rust interfaces with foreign interfaces through the Foreign Function Interface (FFI). FFI is code that is written in C or some other language, that is then called by Rust. These foreign functions are assumed to be unsafe since Rust cannot guarantee anything about it.

However, FFI is a powerful feature since one can wrap the foreign interface in a Rust wrapper that will provide a safe API. This is useful in games development as this will allow the interfacing with low level APIs such as Vulkan or OpenGL directly from Rust code [27].

4.4 Entity Component Systems and their potential architecture.

4.4.1 What is an ECS

An Entity Component System (ECS) is an abstraction provided to solve an issue with hierarchical structures and flexibility. This issue is caused by using OOP principles, such as inheritance. In a deep inheritance hierarchy, a change at the root will require significant reworking of code. This issue is also called coupling, where parent classes are coupled to their inheriting children. A principle called composition over inheritance is used to overcome this problem, where functionality is grouped into components or classes on their own; this way inheritance is limited. Instead, the ECS will maintain an index for each Game Object (GO) and have it mapped to its components. A GO is a singular piece of content that can perform tasks like rendering, animating, take input, etc. They may be only logic, meaning they have no visible form in the game world. Examples of GO could be anything from way points to playable characters [28].

4.4.2 Current issues with OO

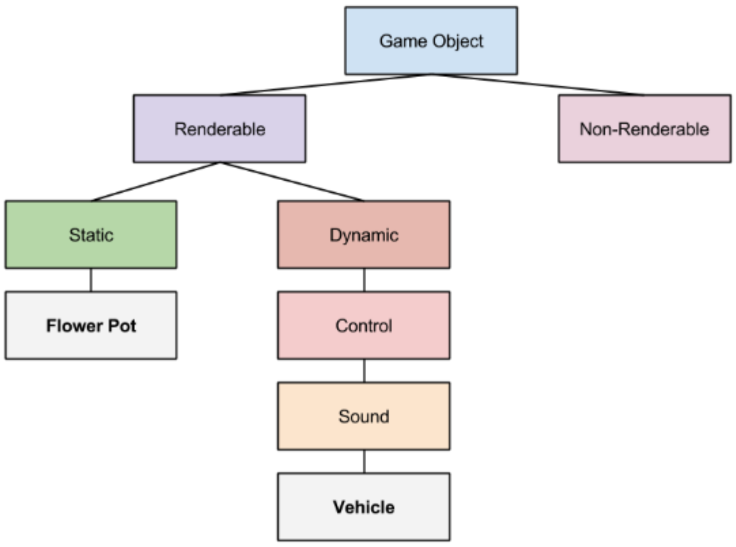


Figure 18 OOP Hierarchy [29]

As requirements come in from the game designers, these new features need to be worked into the inheritance hierarchy. This can cause issues as certain features may require to be, for example, static and non-renderable. This is not possible in the above hierarchy since something that is static, cannot also be non-renderable. This causes an issue called the diamond inheritance problem which occurs when a subclass inherits from a superclass more than once.

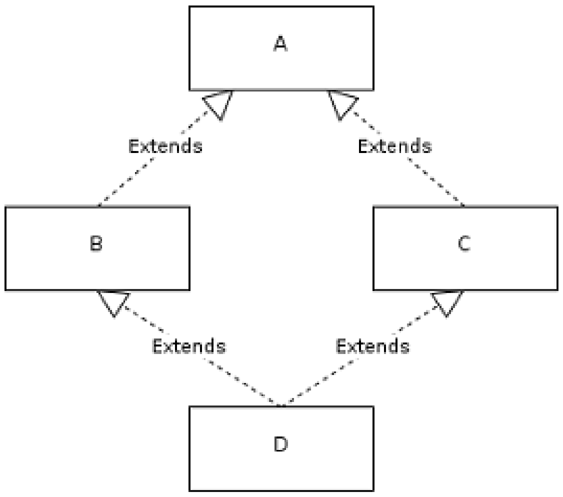


Figure 19 Diamond inheritance problem [30]

The above diagram highlights this issue. When D inherits from both B and C, which in our case can be Static and Non-Renderable respectively, both of which inherit from Game Object, the question that arises is which version of the methods does it use, the ones inherited through Non-Renderable or the ones inherited through Static?

A way around this issue is to use duplication, and have a static class also inherit from Non-Renderable. This is however widely accepted to be bad practice, as one should be cutting down on duplication. This sort of hierarchy can be seen below in Figure 20.

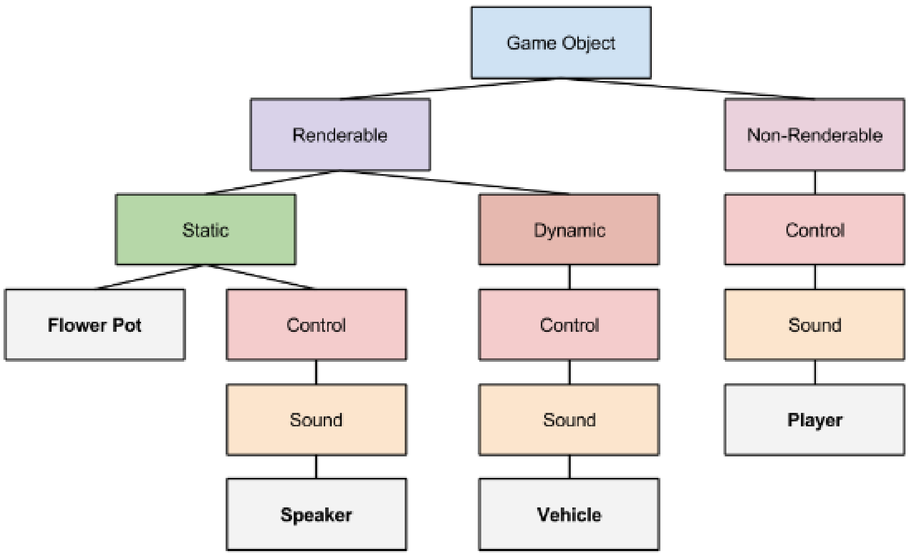


Figure 20 OOP redundancies [29]

Another solution to overcome this sort of tangled messy hierarchy is using a non-intuitive hierarchy. This however, will make maintainability and scalability difficult as the feature set grows to expand on the game's logic and gameplay. Furthermore, as features get added that require more sharing of functionality, more functions will have to be written to accept these new classes [29].

This architecture creates quite high reliance on parent classes. This is an issue when modifications need to be made near the root of the inheritance hierarchy. Taking Figure 18 for example, if a method were to be removed due to being made redundant, but has been overridden in the inheriting classes, the code would need to be removed from all inheriting sub classes.

4.4.3 How the ECS solves these issues

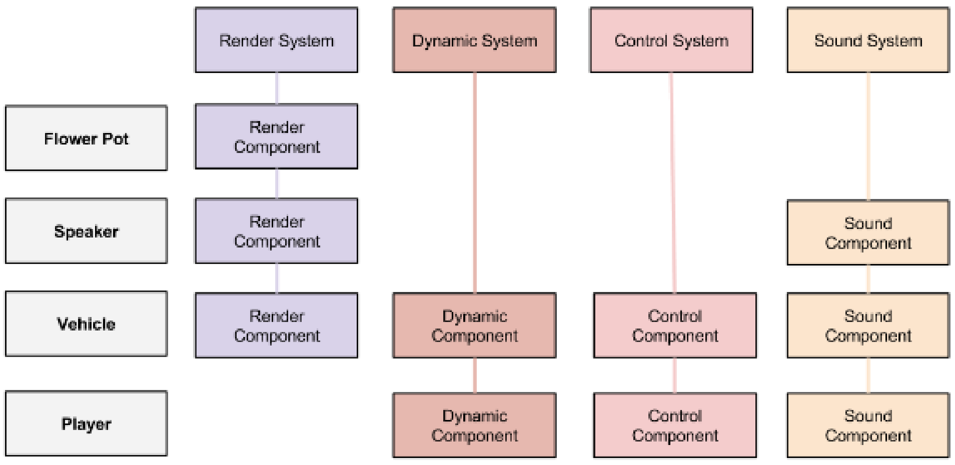


Figure 21 naive ECS architecture [29]

Some common misconceptions about ECS are related to confusion regarding OO principles. It is important to note that an ECS consists of Entities, Components and Systems. Entities and Components are not equivalent to Classes and Objects [31]; however, having said this, an explanation of an ECS is as follows. An Entity is an empty wrapper around its components; it can be thought of as an index to a grouping of components. This can be seen in Figure 21 where a Flower Pot is essentially an index. Components are properties that are associated with an entity; it is just data and this is important and can be seen in Figure 21 as the Render Component, Dynamic Component, Control Component and Sound Component. Finally, we have Systems, the abstraction that separates game logic from the data representation of the game. In Figure 21 the Render System operates on all render components. Similarly, every other System operates on a certain type of component. Systems can also be written to operate on a combination of components, updating in any way seen fit for the game [32].

4.4.3.1 Advantages

One of the advantages of an ECS system is its ability to clearly separate game logic from data. This is a great advantage as developing games is an iterative process and it’s very costly to make changes to logic after creation. This means, that with a normal OO hierarchy, a game logic change would require significant rework especially if the dependencies are in a deep hierarchy tree. Another advantage would be its ability to parallelize the various systems to increase the efficiency of the game, making use of the processor's multiple cores [29].

4.4.3.2 Disadvantages

All these advantages do not come without risk. The ECS, with its high levels of decoupling and encapsulation, makes it difficult for systems to work together without coupling them or having awkward code gluing the systems together. For example, some of the processes where order matters are physics, collision detection and rendering. These processes are interlinked as physics must first move the entities in the game world, then the collision detection must register that they are touching and finally the renderer must display them correctly. However, if the systems are not able to communicate and know about each other, then these steps can occur in the wrong order, which leads to bugs like clipping (overlapping) textures. There is also an issue with shared resources; take for example, a mesh that has 50 instances in the game world. It is highly inefficient and redundant to have 50 separate versions of the same mesh. However, in an ECS, it is no longer clear where shared resources should be maintained [29] [33].

4.4.3.3 Additional problem space

An additional problem in creating an ECS is the inability to easily make communication channels between systems. This can be tackled in several ways, such as implementing a message queue API. This will allow the various systems to pass messages between each other. A solution to the sharing of resources, an approach taken by an ECS called Artemis, was having the Entity Manager, a global resource accessible by all systems, maintain all components. This allows for sharing and syncs all components, overcoming that limitation of the ECS [29].

4.4.4 ECS/Games Engine User Requirements

This section discusses the need for an ECS and then highlights the requirements of a minimal feature game engine.

The users of an ECS are those who are implementing or are creating homebrew video games. They wish to have finer control and better flexibility over what they want their components to do and if they wish to use a third-party library or create their own. Going further, the users of a game engine would want to have a full featured and easy to use framework to make video games.

The current standard in the industry are the game engines provided by Unity (Unity 2018) and Epic Games (Unreal Engine 4 or UE4). These two engines are full featured and have been in development for many years. However, there are many restrictions to using these engines. Unity is a closed source game engine, not allowing for expandability and modification for custom behaviour at the engine level. UE4 has source code available to tweak and modify, however, the framework locks the developer into using C++, introducing the problems discussed earlier. Furthermore, by using these engines, the core features of the game could become deprecated/unsupported in the future. This creates more work during development and later for maintenance and upkeep. The best solution to these problems is to create a game engine in-house to ensure future proofing and more stable development [34] [1].

Considering these drawbacks of current industry standards, an ideal system could be implemented in Rust, providing for a memory safe programming experience (no malformed memory accesses) with modern programming features such as closures and pattern matching, all with zero cost abstractions. Rust, which promotes a data-oriented architecture (see appendix 9.3.1), will be beneficial for designing a performant games engine. Although allowing for hot reloads and code reflection could be difficult, the benefits would make for a system with less of a need for it.

An ECS library will facilitate the ability of other users to implement their own components. For example, the ECS contains an Entity Manager, where each entity is represented by an identifier which maps to its components; this will be implemented and registered by the user of the library. It will comprise some sort of bare bones interface for a component, providing its entity-component mapping which will be registered with the manager. The ECS will also provide an Engine, which will update each entity and its components on every iteration of the game loop.

The benefits of the entity component system are highlighted above. The drawbacks are the high level of isolation and lack of cross communication between these components. This communication is vital for allowing easy programming of game logic, and so will be considered in the implementation of this ECS. A further feature, which is extremely desirable, would be a messages interface, to facilitate communication between components.

4.4.4.1 ECS Requirements:

1. Must provide an interface for implementing Components
2. Must provide an interface for implementing Systems
3. Must provide an API for associating entities to components
4. Must manage a game loop, running within it each registered system

4.4.4.2 Additional Requirements:

- Design and implement a simple rendering system
- Design and implement a simple audio system
- Design and implement a simple physics system
- Develop a small demo using the previously mentioned implementations

This however, does not complete an entire game engine, rather just the backbone. The importance of this backbone cannot be understated. Further features that would be nice to provide are physics, sound, AI, rendering, IO and networking components. These are additional requirements, but they can also be included as third-party libraries as there are plenty of these available for use. An example of such would FMOD for sound, Vulkano for rendering on Vulkan APIs and Glium for OpenGL APIs.

5 ECS Implementation

5.1 Rust ECS Implementation

The architecture of the ECS created in the available time is as follows:

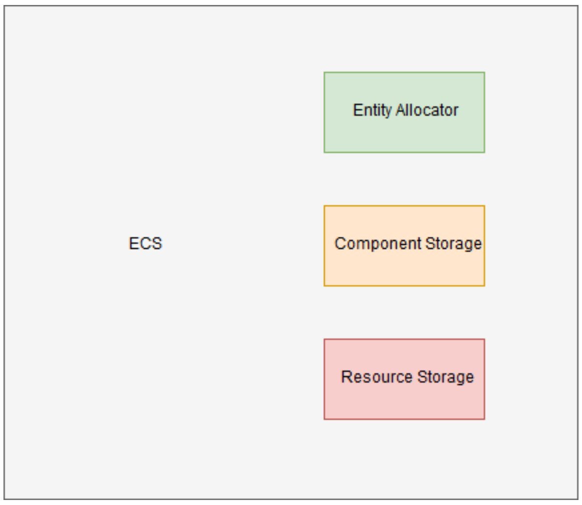


Figure 22: Core ECS architecture

The ECS struct maintains three data structures, which are the Entity Allocator, Component Storage and Resource Storage. They work as follows:

The ECS was designed to have three major components, leaving the fourth as an implementation detail of the user of the library. The three major components are the Component, Resources and Entity management modules. The first consists of ways to define Components and their associated storage method and Entities provide manipulation methods such adding, removing components as well as allocating and deallocating Entities.

The Entity Allocator provides a way to allocate and deallocate entities from the game world. In order to do so, a collection of entities must be maintained and this was accomplished by storing them in a “vec”. However, with such a naïve implementation, an issue arises where a removed entity, which thereafter is replaced with a new entity with a different set of components, is accessed assuming to be the old entity. This is not caught by the Rust compiler and could cause the program to crash. The solution to this problem is to introduce the concept of a generation, which is a value that increments each time an index is reallocated, preventing entities from being illegally accessed. The following image highlights the construction of the generational data structure.

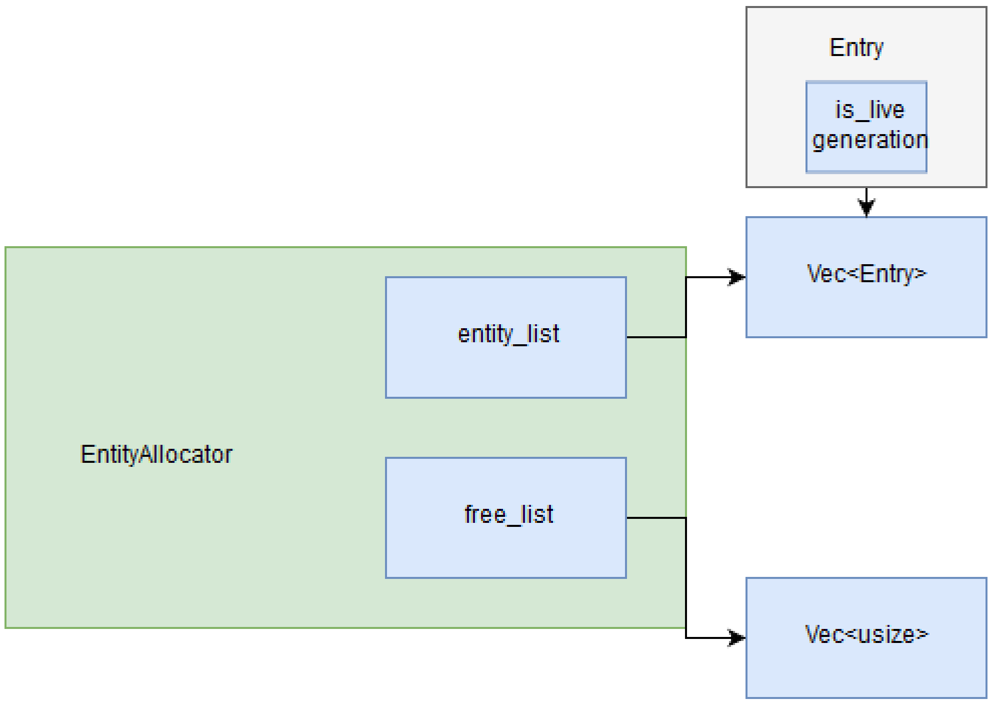


Figure 23: EntityAllocator architecture

The generational index "Entry" in Figure 23 can be marked as "dead" by setting the (is_live) marker to false. This marker is set to false when an entity is deallocated making it impossible to access. It has its index removed and put into a "free list".

When adding a new entity into the generational index it is important to make sure that the array does not grow unnecessarily. If the free_list is empty then a new index will be allocated; if the free_list contains indexes then the new entity will be allocated one from the free_list, marking the corresponding entry as "live" by setting the is_live flag to true. This architecture means that the entity list will never shrink, and as a consequence should only be used for long storage.

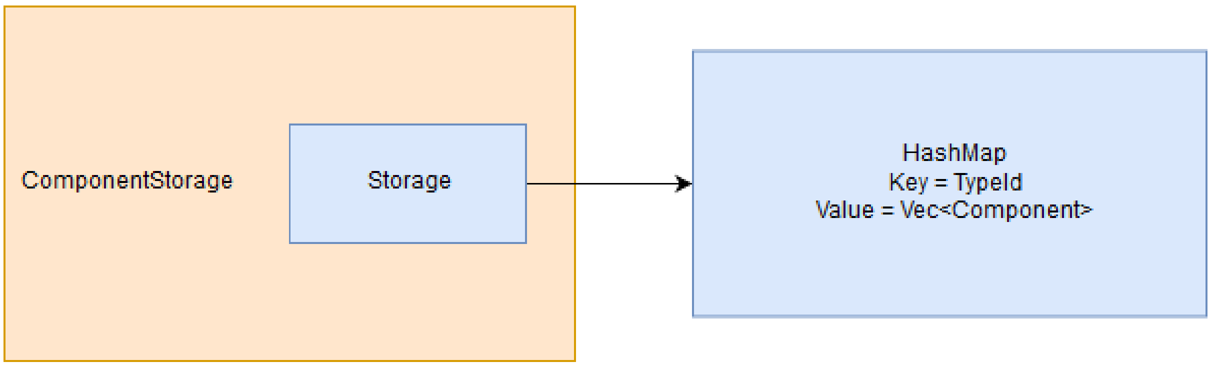


Figure 24: ComponentStorage Architecture

The main idea of the Component Storage module is to use a vec to store components separately, in their own data structures. This architecture is like that seen in Figure 21 where each vec will have one type of component. Rust makes available a standard module called "Any", which simulates runtime reflection. In addition to "Any", there is also a module called "TypeId" which will return the "TypeId" of a given type. A "TypeId" is a "struct" that contains information about a type. Each type will have a unique “TypeId” and consequently, it can be used in a HashMap to store and retrieve data by its type. This allows for the appropriate component storage to be retrieved and cast to the correct type. To aid with this implementation, by making the API as generic as possible all components must implement the Component Trait.

The Component Trait packages two forms of data, one is the type of the component and the other is the type of storage the component will use. By ascribing the Component Trait with an associated type constrained to being an implementer of the Storage trait, it becomes easier to instantiate and initialize a new component in the Component Storage module. Furthermore, when a user of the library implements their own component type, they can also implement their own storage type and the component module functions would work as expected, as long as the logic in the user implemented methods is sound.

A major feature of an ECS is to allow systems to operate over select components. Iterators are used to implement this feature. Iterators are implemented by most data structures in Rust and they allow for safe iteration over its elements. An Iter trait is created, which is a copy of the standard Iterator implementation with some extensions. This is the trait type that will be returned from a Component Storage type. The Iter defines methods for getting the next element, joining this iterator with another (zip like functionality) and an “into_iterator” method which transforms the iterator into an iterator wrapper providing even more of the standard library Iterator functionality such as “map” and “collect".

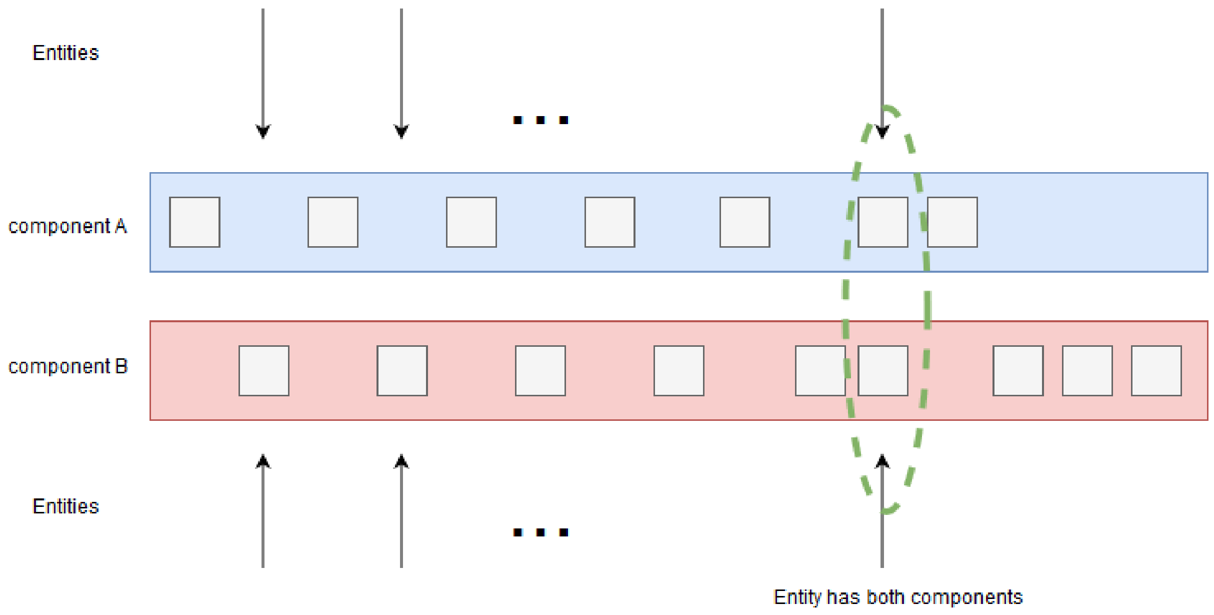


Figure 25: Iterator next_element function

The next_element method of the Iter trait takes as parameter an optional index. This optional index is key when syncing two iterators together when joined. Joining iterators is a key method of searching entities that have the two components. When both iterators find an entity with its component, if the indexes do not match, the lesser one is iterated until it matches the larger one and vice versa. If the two indexes match then the entity has both components, in which case return a reference to both components.

Due to an issue with dynamic typing, when an entity is being deallocated, all of its components need to be set to Empty. In order to do this, each component entry needs to retrieved, downcast to the appropriate type and then indexed. However, this is not possible due to the Any API provided by the standard library. In order to do this, dynamic dispatch was used by implementing a special trait called GenericComponentStorage. This trait defines a remove method, which takes an index and returns a result, ("Ok" for successful remove and Err otherwise). This trait implements Send, Sync and Downcast. The notable feature here is Downcast; this is a crate developed by "Ashish Myles" [35] which is a convenience that allows users to easily implement the Any API for their own uniquely defined types. This way, the GenericComponentEntry allows for both downcasting capability and removal of components, solving the issue with the original Any API whilst not compromising any functionality.

Due to Rust's use of mutable and immutable access to prevent data races, a need for the interior mutability pattern was needed when accessing multiple component storage variable mutably. This sort of functionality can be implemented through using RefCell which makes use of unsafe code to mutably borrow immutable variables or through RwLock which is similar but also allows for concurrency. Going down the route of the RwLock allows for the ECS to allow multithreaded borrowing of its component storage; this is useful for implementing systems that run in parallel.

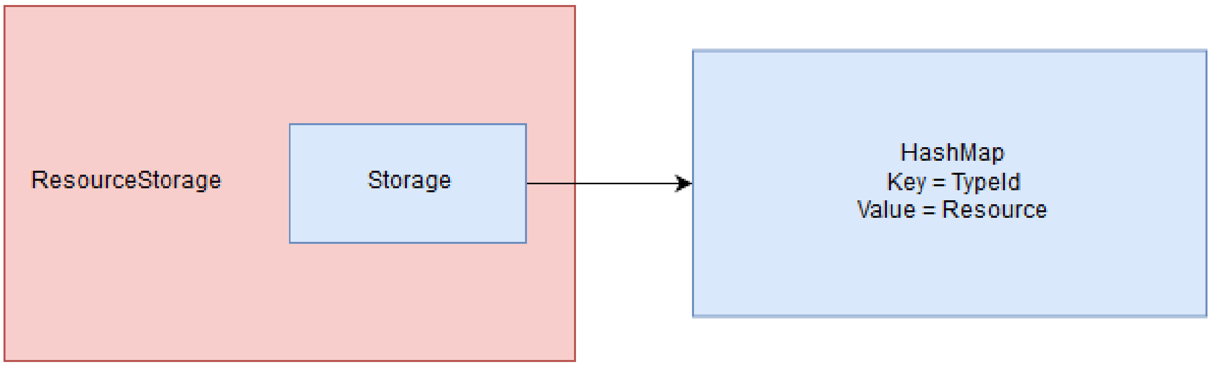


Figure 26: ResourceStorage architecture

The resource management data structure is implemented in the exact same way as the component storage. The difference lies in the API to access the resource management where, instead of storing a TypeId key and Array value (like in the component storage), the TypeId of the resource is used as the key and the Resource itself is used as the value.

The reason for creating resource management is to allow for shared data. This solves the issue with communication that was described as an addition problem area in section 4.4.3.3 by allowing easily accessible data that is not tied to any entity.

Both of these modules are contained within the ECS struct. The ECS struct has an initializer that will default and ready the modules to accept new entities and components. The ECS struct provides higher level methods to access and manipulate the entity allocator and component storage. Some new types are introduced in the implementation of the ECS struct such as the Read and Write handle. These structs wrap the result of acquiring locks from the

RwLock, RwReadGuard and RwWriteGuard. The Handles have methods to conveniently retrieve the underlying component storage iterator. These handles can also be sent across threads safely.

A major limitation of this ECS is that it does not aid the user in any way to implement systems. This can be quite confusing organisationally as there is nowhere to really put the systems. The user is in charge of arranging the order of execution of each system and also which ones are to be run in parallel. In addition, there is a deadlocking situation that is not handled and so the user must also be aware of this.

To improve the current implementation of the ECS, the deadlocking situation can be prevented by implementing a predictable locking order, which is a fairly simple solution to implement but this not an optimized solution. Furthermore, there is currently a stub for a system trait to help organise systems in a game loop. This is a more complicated problem to solve but it can be accomplished through the use of third party libraries like Tokio [36] or SHRED [37], which can also help provide a better solution to deadlocking.

5.2 C++ ECS Implementation

Implementing an ECS in C++ poses the same architectural challenges as those faced above. However, these are exaggerated using OOP style promoted by C++. But this cannot be said for sure since this project primarily focuses on development in Rust and not C++ ; some comparison of the development experience will be provided below.

Despite being an OOP language, C++ can be used to create highly efficient ECSs. A library called EnTT will be used as a comparison for the benchmarks conducted and it is also the fastest open source ECS library in C++.

Inspecting the source code of this library organisationally shows that it is all contained in one file. This is called a header only library and it is one way of organising C++ programs for easier compilation to multiple target systems. Rust is also capable of this, but it is not required to do so. The reason for writing all functionality in one header file in C++ is for better code optimization and easier inclusion. However, there are disadvantages such as long compile times despite the number of changed lines of code and difficulty to read.

To provide a very performant architecture, the EnTT library provides a runtime environment with type reflection. This sort of functionality can be problematic in Rust due to its requirement for lifetime analysis and single ownership. Any form of multiple ownership may be overcome with the use of a reference counted smart pointer but this then causes overhead, which is not good for performance.

Despite the lack of depth in this comparison, it does highlight an issue with Rust which is higher friction when programming. Having to consider only allowing one owner and lifetimes of references, there is a lot more to keep in mind while programming compared to C++ which is comfortable with allowing multiple owners and omitting lifetimes. This might make it easier to program a complicated and highly efficient ECS like EnTT in C++ as opposed to Rust.

6 Benchmarking

6.1 Continuous Integration Pipeline for Benchmarking

Continuous Integration (CI) is a method of developing software where code is built and tested in a parallel manner. It can be considered as the practice of merging in small code changes frequently - rather than merging in a large change at the end of a development cycle. The goal is to build healthier software by developing and testing in smaller increments, providing immediate feedback and helping in writing self-documenting code. [38].

Using Travis-CI enabled me to implement the ECS in stages. I was also able to automate the production of benchmarking reports. For each new feature implemented in the ECS, I would write an appropriate benchmark testing it. Once this work was finished it would be pushed to Version Control provided by Git and GitHub, which integrates nicely with Travis-CI. Travis will then pull the most recent changes from the repository and run various predefined commands which are, “cargo build”, “cargo test” and “cargo bench”. Cargo build will build the project, cargo test will run unit tests to check logic and behaviour and finally cargo bench will run the benchmark tests.

6.2 Benchmarking framework

A framework called Criterion-rs is used to benchmark the ECS functionality. The Benchmark calculates the average time taken per iteration (clock count) for the function to complete. The library provides complex statistical analysis to indicate if performance has regressed or improved from the previous run of the benchmarks. The main motivation of using this library instead of the included benchmark utility tool is for its use of stable Rust. The built-in benchmarking utility makes use of features only found in unstable (or nightly) Rust, which is riskier to use than the stable release channel (4.3.2).

Criterion-rs also contains some other useful functionalities such as providing a detailed report of each benchmark in HTML form. It also provides numerous graphs that can be used to justify performance. Using this feature in combination with Travis-CI's ability to publish HTML pages to Amazon S3 storage allows data to be readily visible. With some simple configuration in the travis.yml config file, it is possible to automatically upload these reports to an S3 bucket. Here is an example of such a report hosted on an S3 bucket:

https://s3-eu-west-1.amazonaws.com/enginebenchmark/17/17.1/report/index.html.

In addition to this benchmark, cache performance was measured.

6.3 Clock count benchmarking

One way of determining whether Rust is a good games programming language is to measure the performance of the ECS implementation. The performance of the non-concurrent ECS implementation will be compared to other ECS implementations in Rust as well as a few C++ implementations.

6.3.1 Understanding the statistics

The graphs shown below are in pairs; the reason for this is to understand how noisy the data is. In each case, the second graph shows the linear regression, which is an attempt to best fit a line through a data set, while optimising the parameters (best fit). This graph is vital in identifying how noisy a data set is and when paired with a R^2 goodness-of-fit value, it is a measure of how accurately the linear regression model fits the data set. If the data set is not too noisy this R^2 value should sit between 0.99 and 1, otherwise it will be less than 0.99 and the analysis becomes more uncertain. Any value lower than 0.9 indicates a lot of noise [39].

6.3.2 Results

6.3.2.1 *Adding new entities*

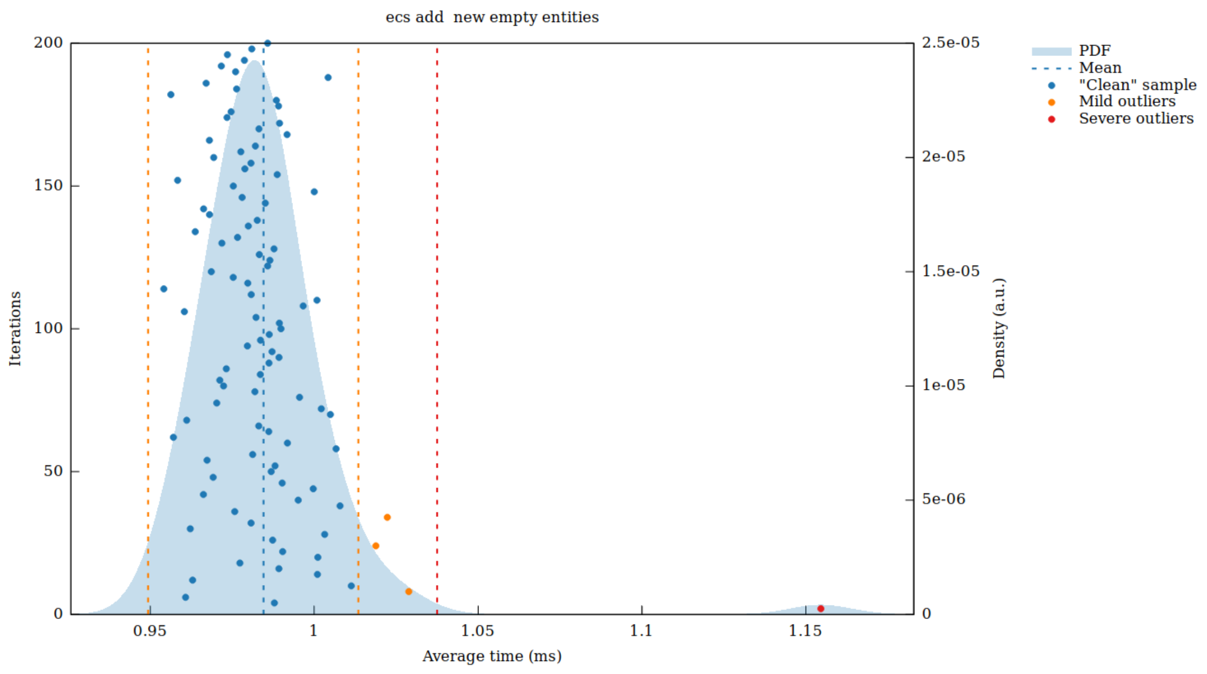


Figure 27: Adding new entities to the ECS - average

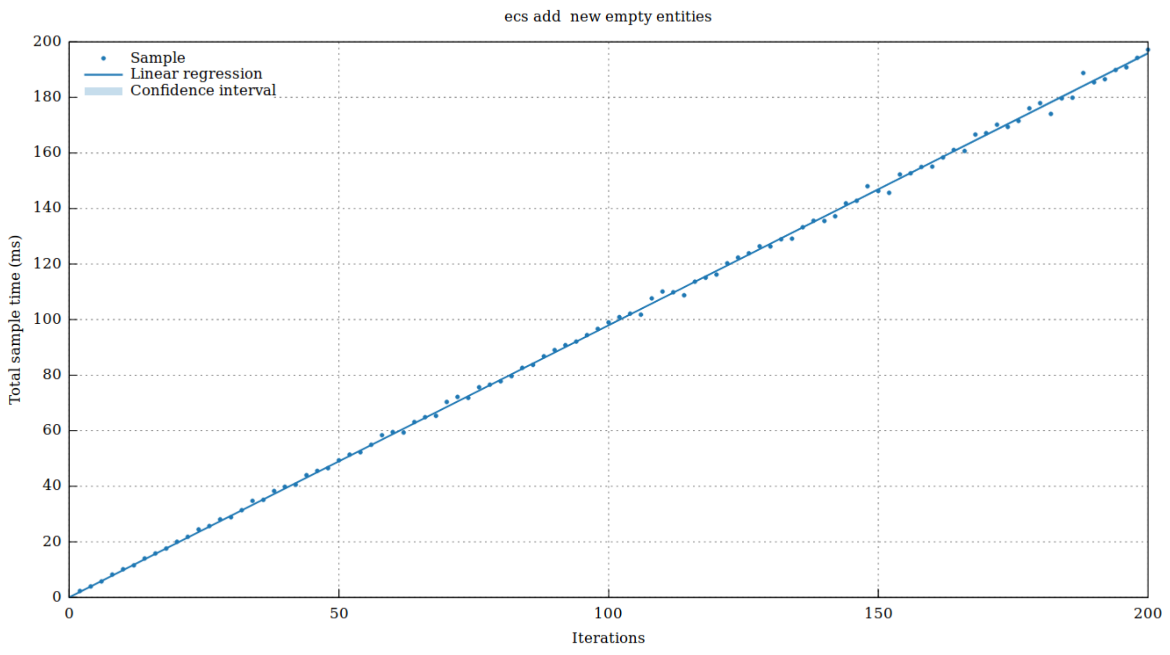


Figure 28: Adding new entities to the ECS - linear regression

The ECS implementation when adding 10000 empty entities to its storage performs at an average 985 microseconds. The R^2 value during this test sits at 0.98, which means that there is some noise but minimal. It can be seen in the linear regression model that the line also fits the data set quite well.

6.3.2.2 Deallocating empty entities

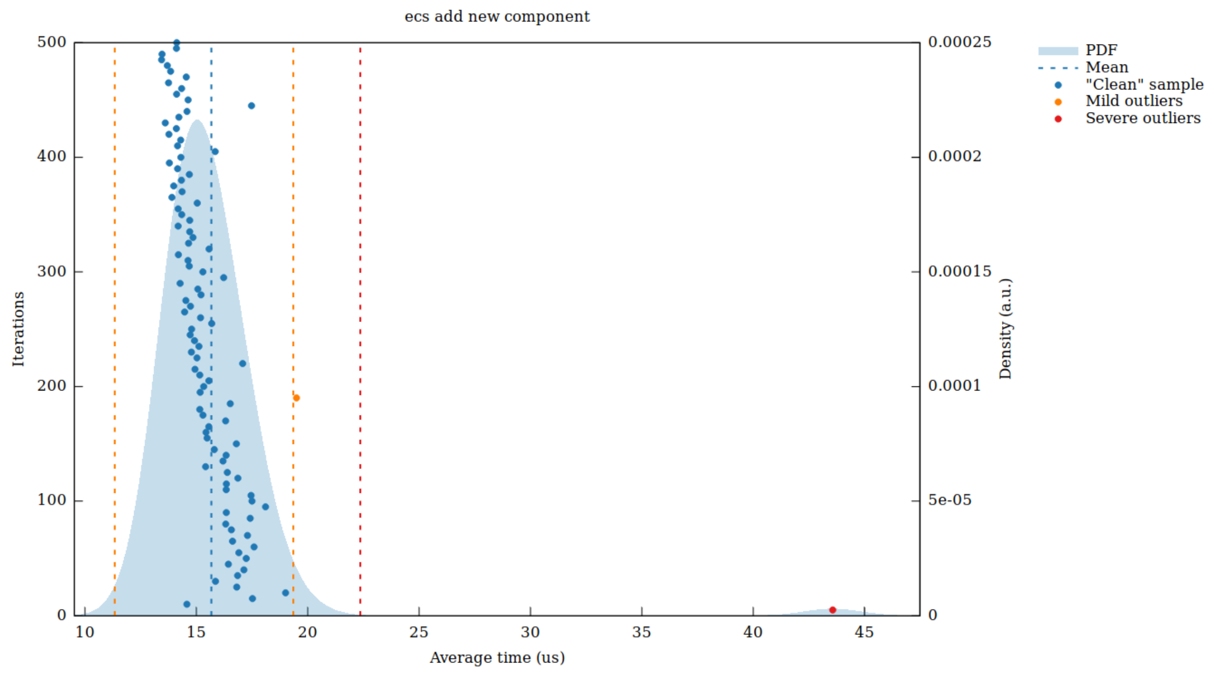


Figure 29: Adding new components to the ECS - average

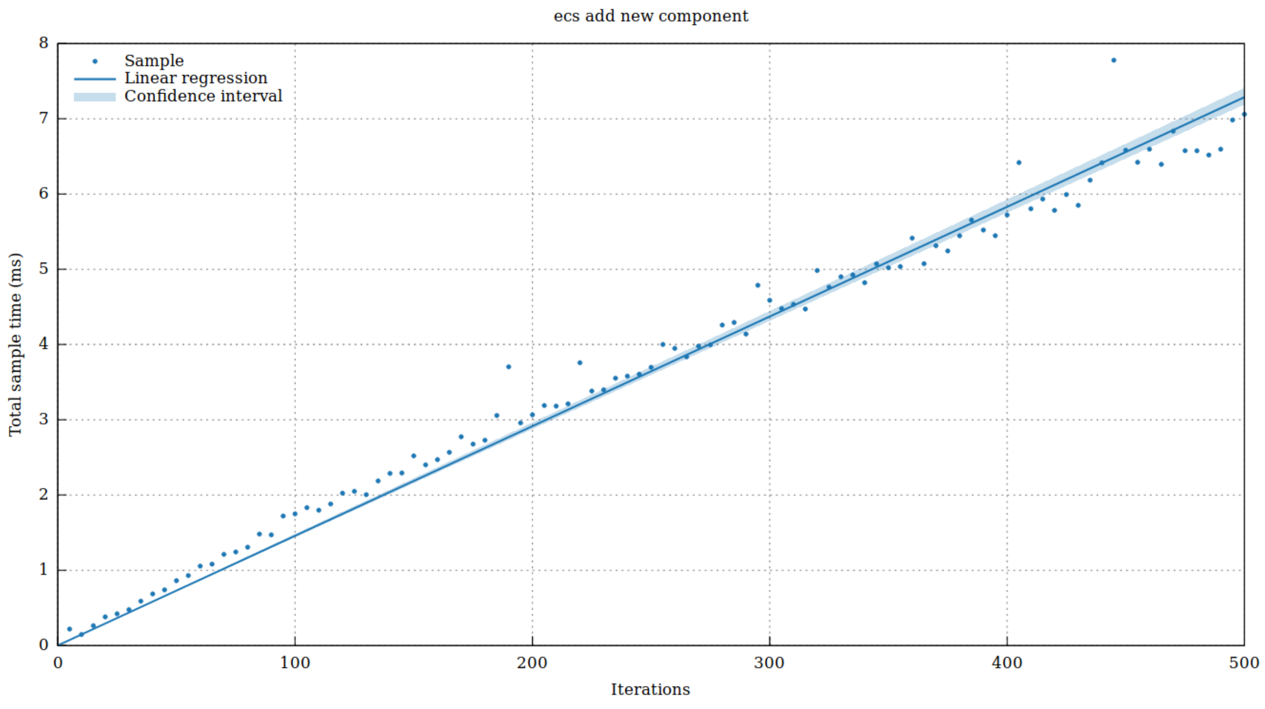


Figure 30: Adding new components to the ECS - linear regression

When deallocating 10000 empty entities, the average time is at 15.4 microseconds. However, the R^2 value for this test is 0.48; it is highly likely that some other process(es) interfered with this test. The time however is not a bad value since most of the operations taking place are within microseconds. In Figure 29 we can see an outlier sitting close to 45 microseconds; this signifies that some other process was prioritised during the benchmarking of this function.

6.3.2.3 Deallocating entities with components

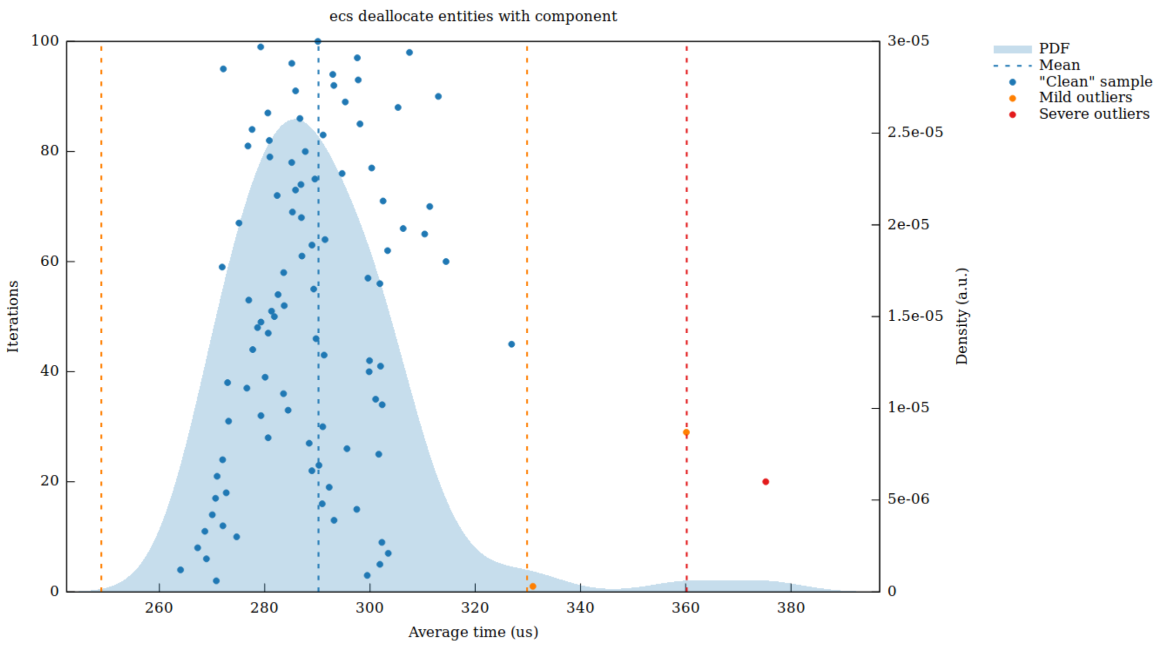


Figure 31: Deallocating entities with component from ECS - average

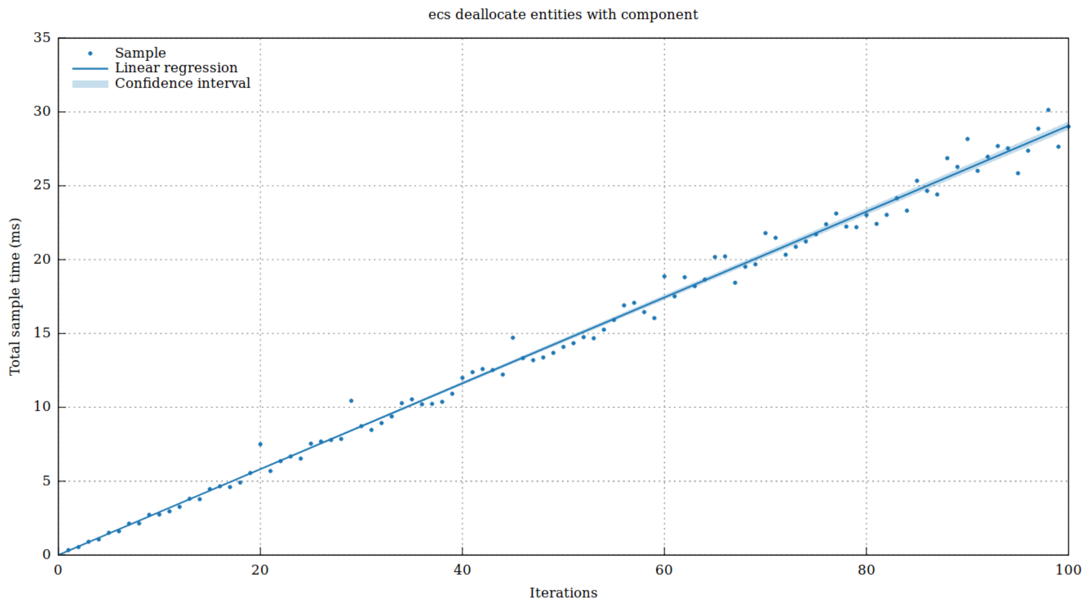


Figure 32: Deallocating entities with component from ECS - linear regression

When deallocating 10000 entities with components the average time taken is 290 microseconds. Figure 32 also shows that the data set is noisy, but not nearly as much as the previous function benchmarked. The R^2 value is 0.81 which also suggests noisiness and so these results must also be looked at sceptically. However, again, the operation of removing 10000 entities and their components is within microseconds, which is still an ideal value.

6.3.2.4 Reading one type of component for all entities

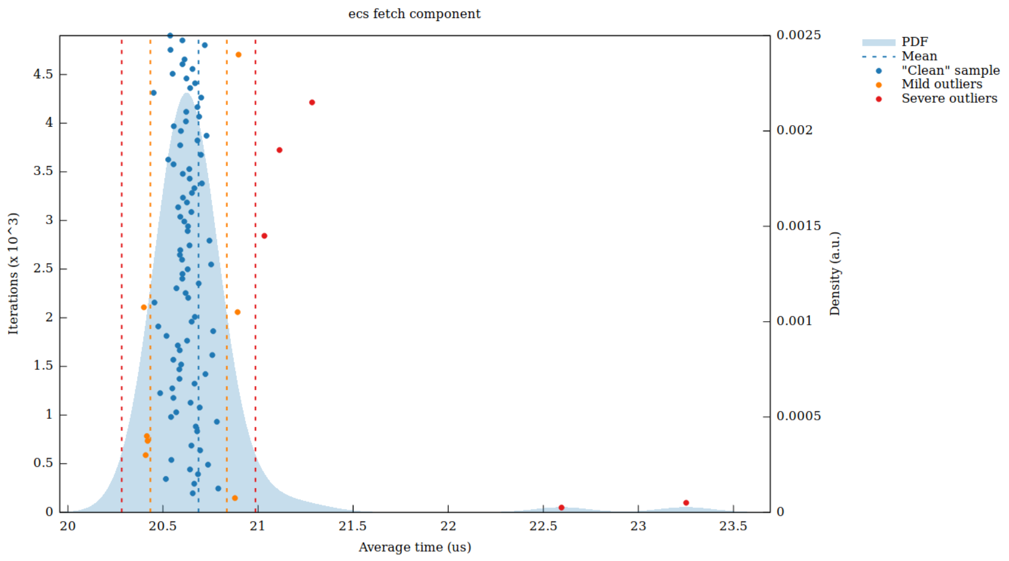


Figure 33: Reading one component from entity in ECS - average

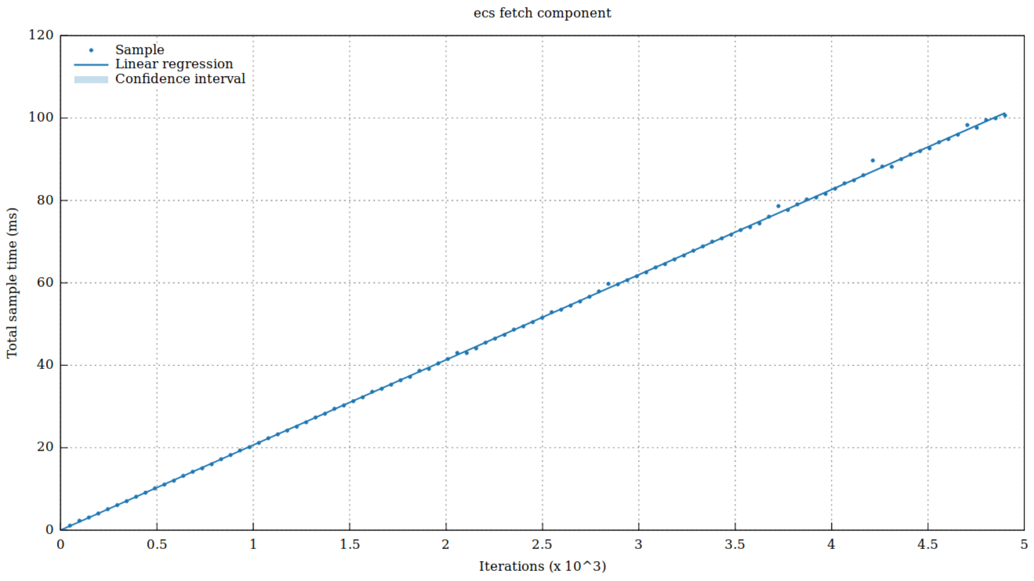


Figure 34: Reading one component from entity in ECS - linear regression

The above graphs highlight the performance of the ECS when reading one type of component for all entities that have it. The average time per iteration is 20.6 microseconds for 10000 entities. Furthermore, Figure 34 highlights the noise in the dataset. There is quite a low amount of noise and this is substantiated with an R^2 value of 0.99. This benchmark did not suffer from much noise and its values are more trustworthy. The 20-microsecond benchmark is highly ideal for such a simple query.

6.3.2.5 Registering a new component

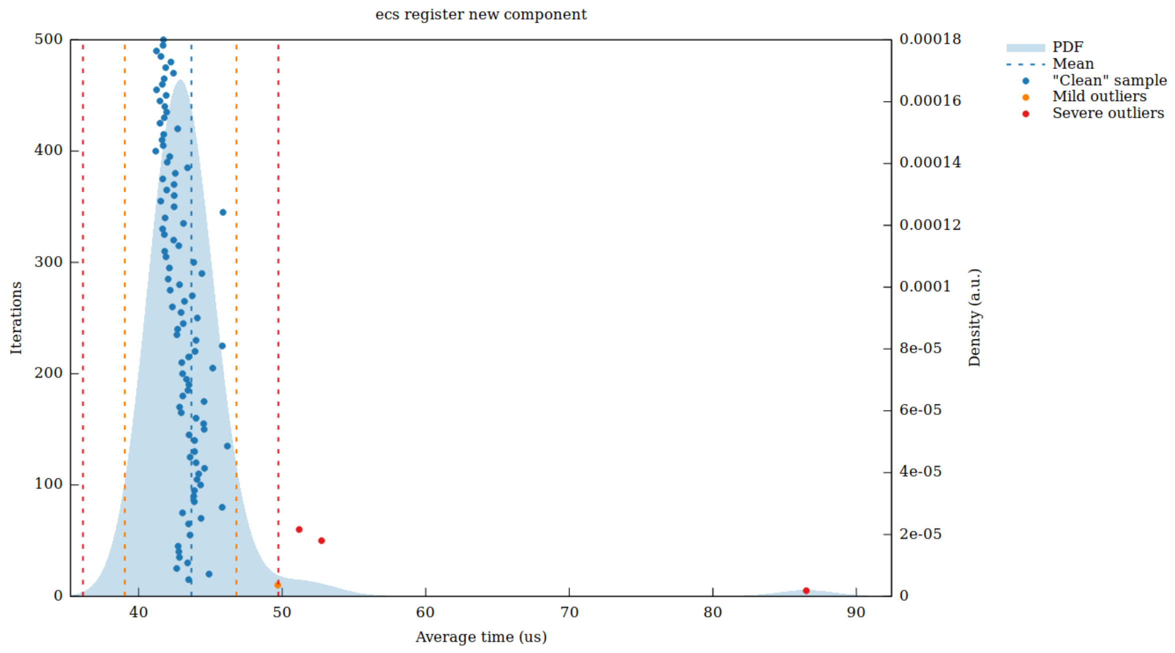


Figure 35: Registering a new component with the ECS - average

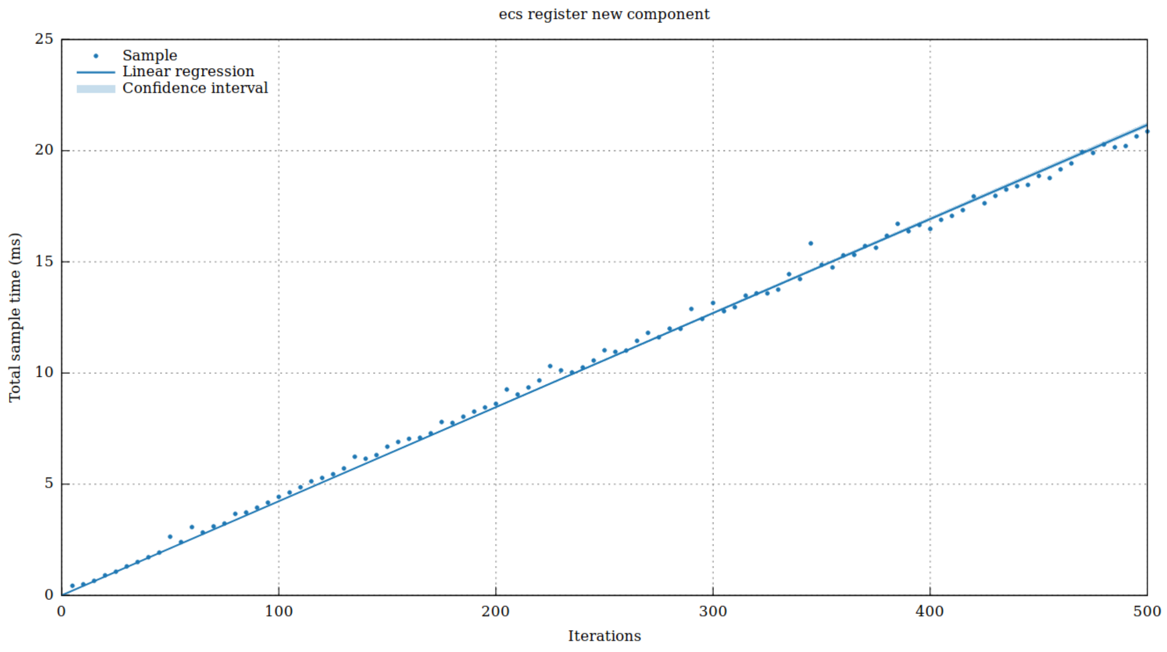


Figure 36: Registering a new component with the ECS - linear regression

When registering a new component, the ECS will create a new entry in the Any map. It will create an empty sized Vec containing the type of component being registered and store the TypeId of the component as its index (key). These operations seemingly take on average 42 microseconds which is quite high for initialising empty data. However, there is a lot of noise present in this dataset as the R^2 value sits at 0.923. This noise can also be seen in Figure 36 and so the average time is probably skewed higher.

6.3.2.6 Removing a component from one entity

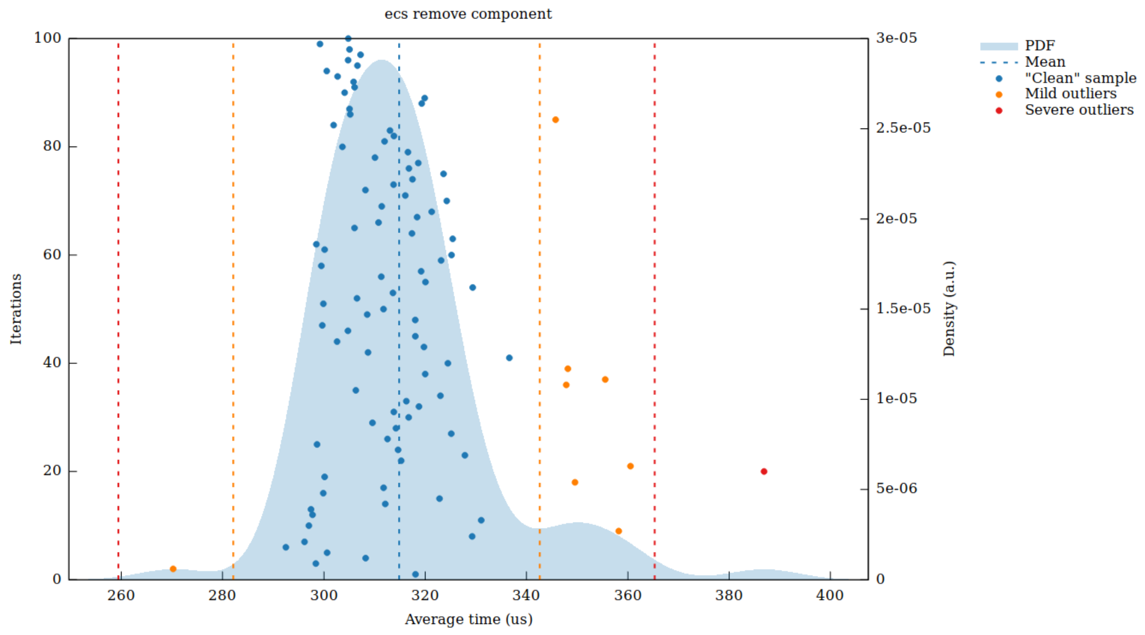


Figure 37: Removing a component from one entity in ECS - average

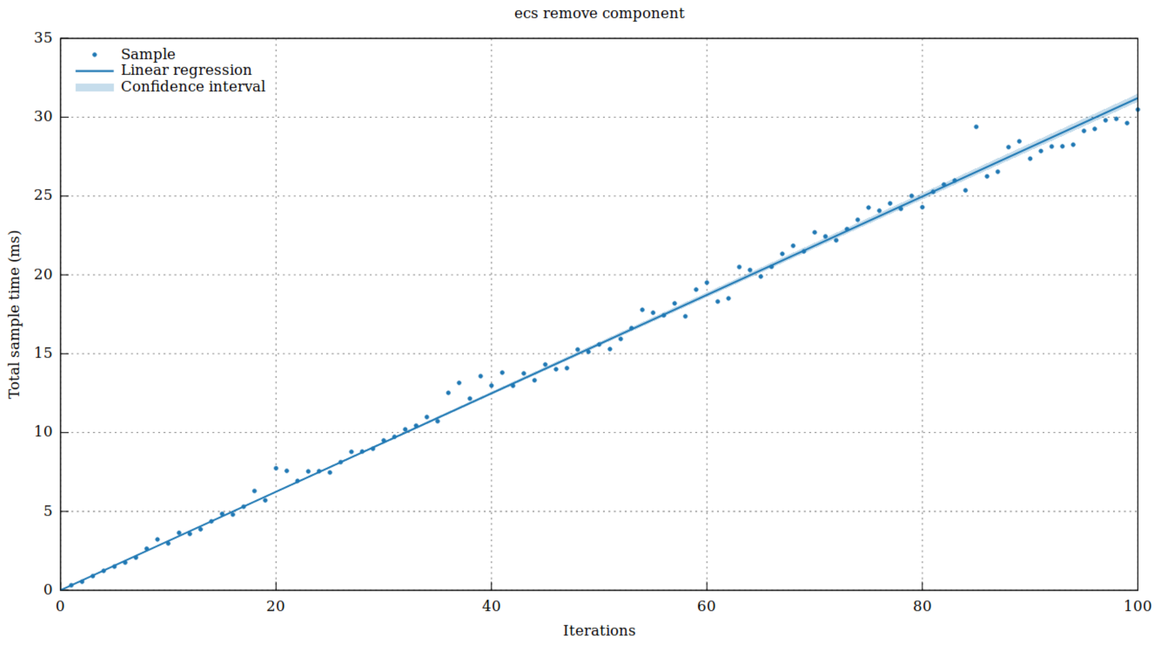


Figure 38: Removing a component from one entity in ECS - linear regression

When removing a component from one entity, the average time taken is 312 microseconds. The specific ECS instance is initialised with 10000 entities and entity no. 66 has its component removed. This operation makes use of direct constant time indexing to do so and no loops are involved. There is no reason for this operation to take 312 microseconds and suggests that the data is simply too noisy to make any sense of. This is also suggested by the very low R^2 value of 0.84. The outliers can also be seen in both graphs, where in Figure 37 the outliers are highlighted in red and Figure 38 shows how far they stray from the regression line.

6.3.2.7 Sequentially updating two components

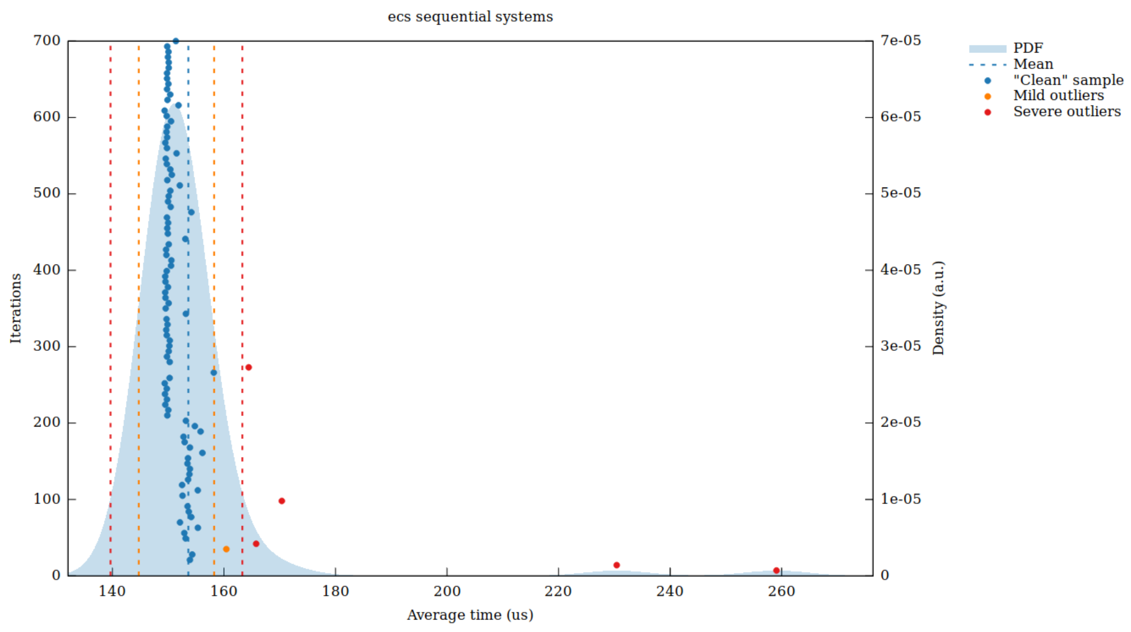


Figure 39: Updating two components sequentially for all entities in ECS - average

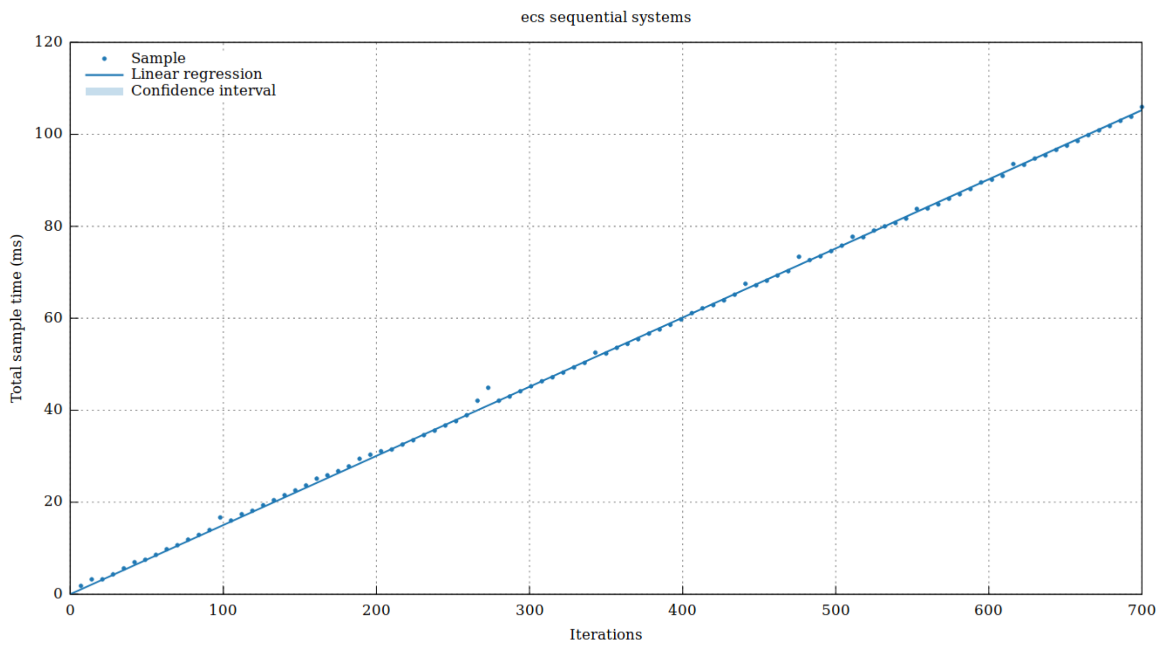


Figure 40: Updating two components sequentially for all entities in ECS - average: Updating two components sequentially for all entities in ECS - linear regression

When sequentially updating two components W1 and W2 based on reading a value from component R, the average time per iteration was 153.1 microseconds. The R^2 value is sitting at 0.984 which indicates that there is some noise but not too much. This noise can also be seen in Figure 40 and as some red outlier dots in Figure 39. This is an acceptable time to update 10000 entities in two different ways.

6.3.2.8 Updating position component if entity has velocity component

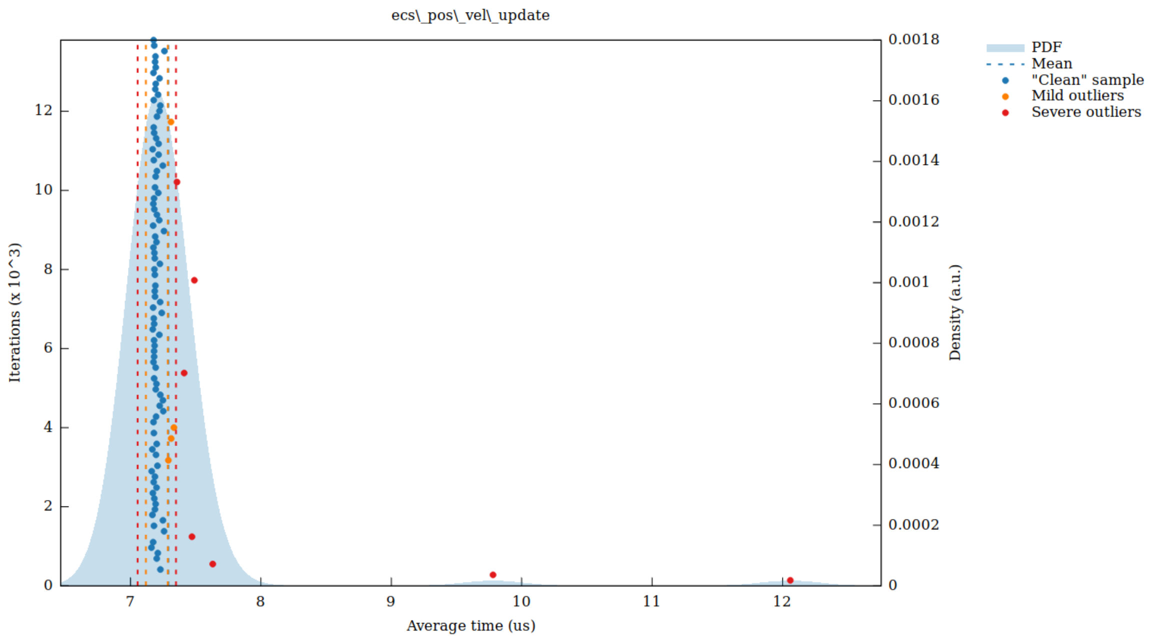


Figure 41: realistic sequential update with two components for all entities - average

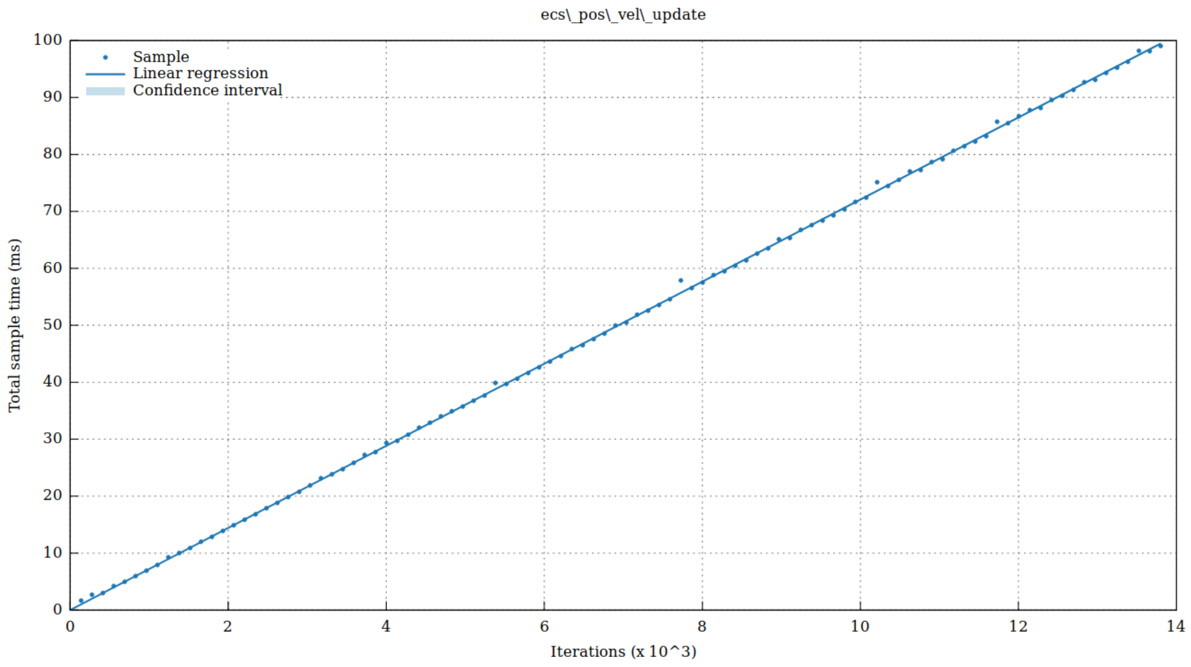


Figure 42: realistic sequential update with two component for all entities - linear regression

Figure 41 and Figure 42 describe the performance in a more complicated search query. In this case, there are 9000 entities of which all have a position component but only 1000 have a velocity component. Here the update loop requires that the velocity component to be read from and the position component to be updated based on the read velocity. In this case the performance of the ECS is at 7.3 microseconds average. This is a good average value to have for such a query.

6.3.3 Comparison with other Rust approaches

A project found on GitHub called ecs_bench by Ischmierer [40] benchmarks the following ECS libraries that are also written in Rust.

Library	pos_vel build	pos_vel update	parallel build	parallel update
calx-ecs	228 µs/iter (+/- 7)	18 µs/iter (+/- 0)	385 µs/iter (+/- 18)	76 µs/iter (+/- 1)
constellation	280 µs/iter (+/- 8)	8 µs/iter (+/- 0)	455 µs/iter (+/- 6)	152 µs/iter (+/- 13)
ecs	1,452 µs/iter (+/- 39)	324 µs/iter (+/- 29)	1,380 µs/iter (+/- 29)	3,644 µs/iter (+/- 74)
froggy	589 µs/iter (+/- 10)	8 µs/iter (+/- 0)	1,429 µs/iter (+/- 65)	73 µs/iter (+/- 2)
specs	261 µs/iter (+/- 7)	3 µs/iter (+/- 0)	675 µs/iter (+/- 88)	87 µs/iter (+/- 1)
trex	1,094 µs/iter (+/- 81)	179 µs/iter (+/- 2)	1,637 µs/iter (+/- 172)	379 µs/iter (+/- 11)

Table 1: Rust ECS performance

Comparing my pos_vel (position and velocity) update average time at 7.3 microseconds with the values in this table, for all the ECS systems listed, the approach implemented in this report outperforms them by at least 1 microsecond (except for "specs", the most performant Rust ECS). This value is an estimate and there are far too many variables involved with clock count benchmarks to be able to tell with certainty that this implementation outperforms the ones listed. However, it does describe the general performance and in this sense the implementation can hold its own against the other implementations listed in the table above. Further to this, the parallel update in the implementation performs at an average of 153 microseconds as described above. This value is also very similar to those found in the table above.

6.3.4 Comparison with C++ ECS results

The following benchmark shows the performance of the Rust implemented ECS and various other C++ ECS. The benchmark was only done up to a sample size of 100000 entities due to hardware limitations. Despite using a different benchmarking framework, the tests have been implemented in the same way as the implementer for the C++ benchmark has done. All values have been recalculated using the same machine.

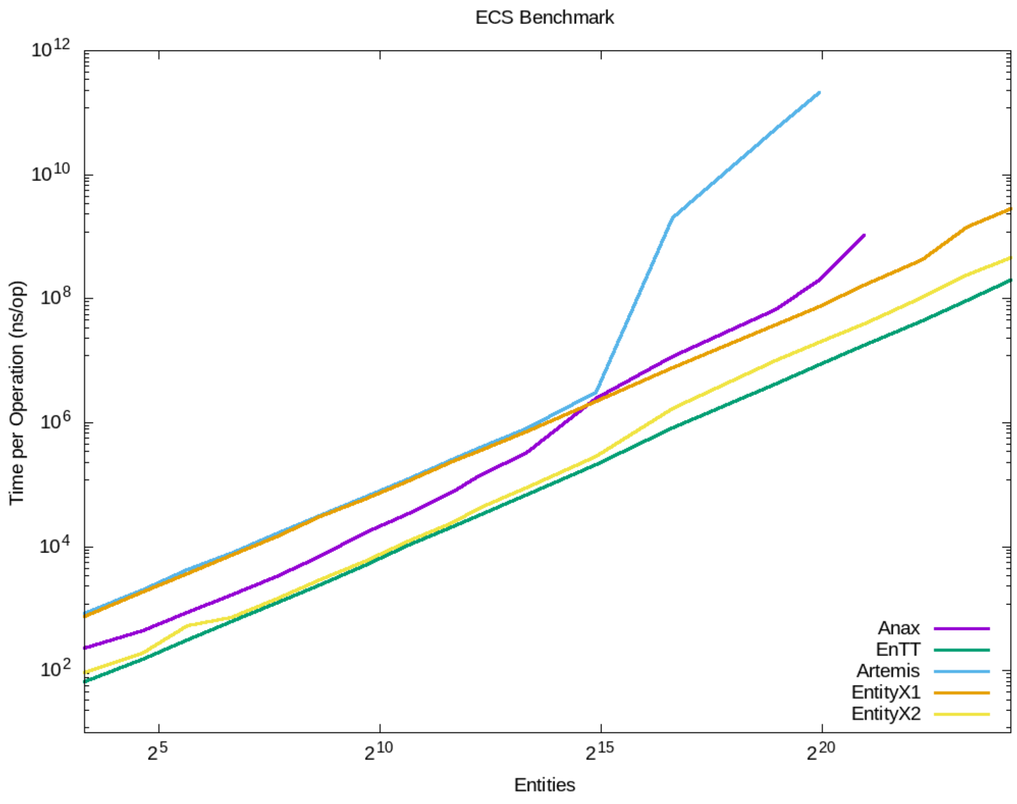


Figure 43: C++ ECS update performance

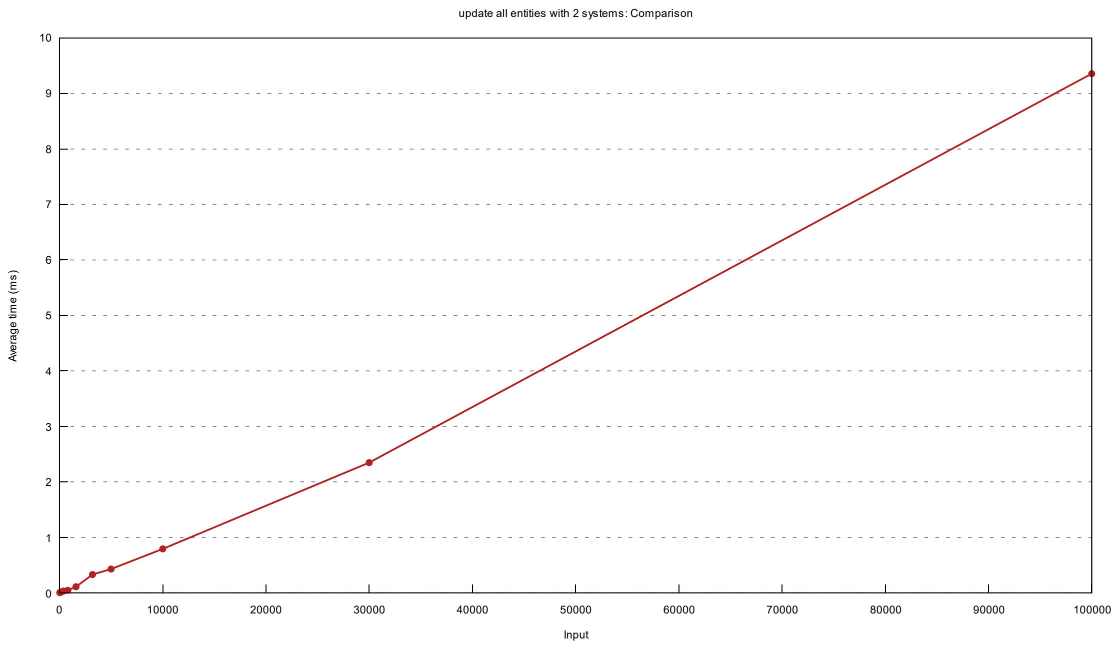


Figure 44: Rust ECS update entities performance

Benchmark	Rust ECS	Anax	EntityX1 (master)	EntityX2 (experimental/compile_time)	EnTT (master)
Update 800 entities with 2 Systems	0.000051s	0.000017s	0.000058s	0.0000058s	0.000005s
Update 1600 entities with 2 Systems	0.00011s	0.000034s	0.00012s	0.000012s	0.000010s
Update 3200 entities with 2 Systems	0.00024s	0.000076s	0.00024s	0.000024s	0.000021s
Update 5000 entities with 2 Systems	0.00037s	0.00014s	0.00036s	0.000042s	0.000034s
Update 10000 entities with 2 Systems	0.00072s	0.00032s	0.00072s	0.000088s	0.000067s

Table 2: Rust ECS vs C++ ECS update performance

In Table 2, the performance of the Rust ECS is shown in comparison to various C++ ECS. The benchmark above was created by a GitHub user called "abeimler" [41].

The above results show that the Rust ECS does not perform as well as the C++ ECS in all cases; however, the results are within the same degree. These are promising results as with no optimisations the Rust ECS already performs well. Some tweaking of the architecture to minimise pointer hopping and further improving cache data locality will produce better results.

6.4 Cache benchmarking

In addition to collecting timing values for the ECS implementation, cache miss values were also collected. The reason for conducting cache analysis is because cache performance is an important factor to consider when developing a game engine or any part of a game engine that deals with allocating and iterating over memory. By having good cache performance there is less time wasted on retrieving data from RAM, or even worse disk storage.

6.4.1 Cache 101

6.4.1.1 Cache architecture

The rationale for introducing cache storage to modern computer architecture is to mitigate effects on performance when accessing RAM. In a nutshell, RAM is slow, CPUs are fast and a lot of time can be wasted waiting for memory retrieval [42].

6.4.1.2 The solution

Introducing cache to the CPU, which is fast on-die memory, reduces the time wasted waiting for RAM. The architecture of cache storage is as follows.

Normally, there are either 2 or 3 levels of cache which increase in size and decrease in speed as one moves further from the CPU. L1 cache is the fastest and smallest cache available to the CPU. Next is L2 cache which is much larger than L1 cache but located further away from the CPU and so is slower. Then there is L3 which is even larger and even slower and finally there is RAM which is the slowest and largest form of memory. When data is required by the CPU, it systematically checks L1 first, then L2 (then L3 if it exists); if the data hasn't been found at this point, it results in a cache miss and the CPU must fetch the data from RAM which is a relatively slow procedure [42].

6.4.1.3 Cache lines

The granularity of data fetched from memory and caches is fixed; this is determined by the cache implementer. A typical size for a cache line is 64 bytes but the cache line varies from CPU to CPU. Furthermore, the granularity of the cache line also determines how much data is invalidated (see next section). When a piece of data is invalidated the whole cache line it belongs to becomes invalidated. This however can cause issues when paired with the multi-level architecture of cache, namely coherency [42].

6.4.1.4 Cache invalidation

Caches are coherent, meaning that when data is read from multiple cores and written to multiple cores, an issue with coherency occurs. Since L1 cache is local to a core, if an update occurs to a piece of data shared between multiple cores, the updated value must be propagated to L2 and L3 cache, such that the data is the same between all levels, thereby upholding coherency. To expand upon this idea and use the correct terminology, when a cache line is written on one core it is invalidated on all others; similarly, when a cache line is removed from an outer cache level then it is removed from all inner cache levels, this is called inclusive caching [42].

6.4.1.5 Prefetching

To mitigate the effects of fetching data from RAM, the cache management algorithm used in the OS will move data surrounding the accessed index to cache as well. This is a property of programs where data found beside an accessed index are likely to be accessed as well in the future. So, all of this data (the amount is dependent on the algorithm used) is fetched and put into cache reducing further main memory accesses [42].

6.4.2 Implementation of Cache benchmarks

6.4.2.1 Intel Hardware Counters

In recent Intel processors, there has been the introduction of hardware performance counters. These counters measure various aspects of underlying hardware's performance.. They are incredibly useful in reliably collecting information about low level hardware such as L1, L2 and L3 cache misses, which is exactly what is needed for this benchmark.

6.4.2.2 Rust PAPI

These hardware counters were accessed using a C library called Performance API (PAPI). The PAPI library was interfaced with Rust using external function calls through a Foreign Function Interface (FFI). An API is exposed this way allowing access to the hardware counters directly through Rust, making it very simple to count the cache misses before and after executing a function, where the value is the difference. The values were then plotted automatically through Gnuplot, a command utility to make graphs, which supports scripting. Although there is no Rust equivalent, a small hobby project called Rust-papi introduces Rust bindings, using the FFI to call the C functions in the library. The library, however, is approximately 5 years old at the time of writing and so required significant re-write to be usable. Using the newly rewritten Rust-papi library, data collection was possible by simply writing the tests in the main function and running as executable; this way compiler optimisations are performed.

6.4.2.3 Graphing Automation

To output graphs from the data collection, a library called RustGnuplot was used. RustGnuplot is a wrapper around the Gnuplot library. Gnuplot is a popular scientific command line plotting library that makes very easy to generate automated plots following data collection. This library is exactly what is needed to fully automate the cache benchmarking. By using the functionality provided by the wrapper library, only Rust code was needed to plot and output the graphs, which can be seen in the following results section.

6.4.2.4 C++ cache benchmark implementation

Unfortunately, there were no projects already available to collect the cache performance of ECS written in C++. Therefore, to compare the cache performance of the ECS written in Rust to a C++ implementation, the library called EnTT was used.

To collect L1 and L2 cache miss data from EnTT, a separate C++ project was created. This project made use of the PAPI C library to collect L1 and L2 data cache misses. This process was similar but some more template code was needed in order to setup and initialize PAPI and the EnTT libraries. Once setup, the data collected was written out to a ".dat" file; there was one file for each test. The ".dat" files were then parsed through a Gnuplot script generating the graphs. This whole process was automated using a bash script, to compile, run and graph the results.

6.4.3 Comparison with C++ ECS results

The following bar plots (Figure 45 - Figure 47 for the Rust ECS and Figure 48 - Figure 50 for the C++ ECS) will be used to highlight the difference in cache performance. There are three different scenarios, setting up 10000 empty entities, setting up 10000 entities with 3 components (R component, W1 component and W2 component) and finally an update where two systems W1 and W2 read from R and write/update the W1 and W2 components respectively.

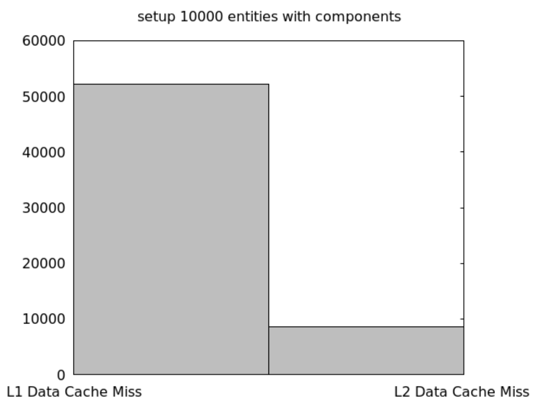


Figure 45: Rust - creating 10000 empty entities

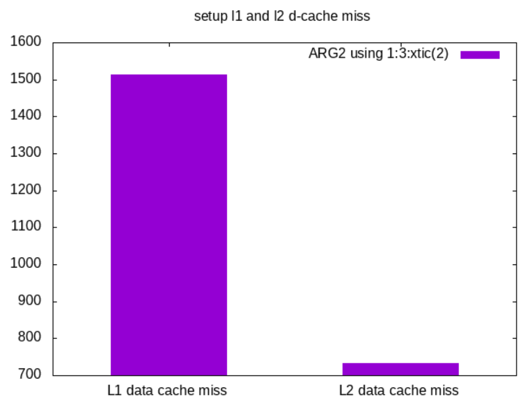


Figure 48: C++ - creating 10000 empty Entities

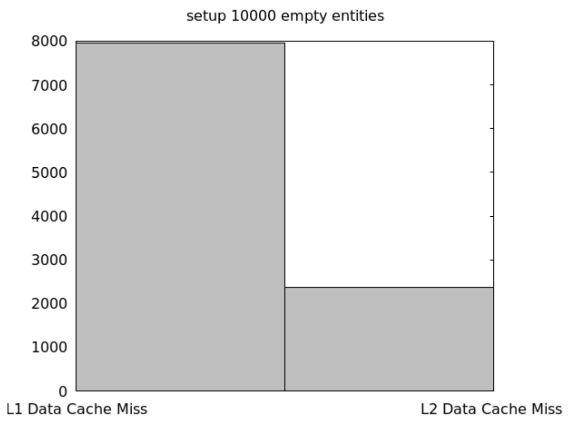


Figure 46: Rust - Setting up 10000 entities with components

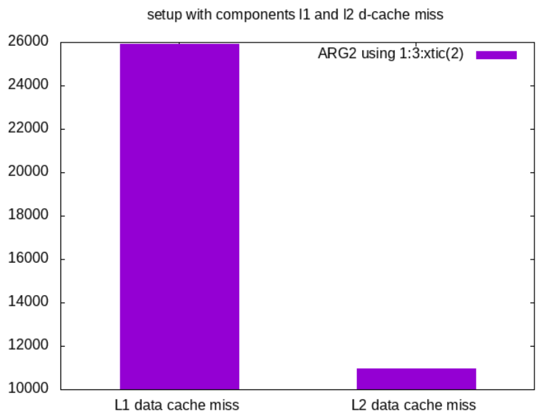


Figure 49: C++ - Setting up 10000 entities with components

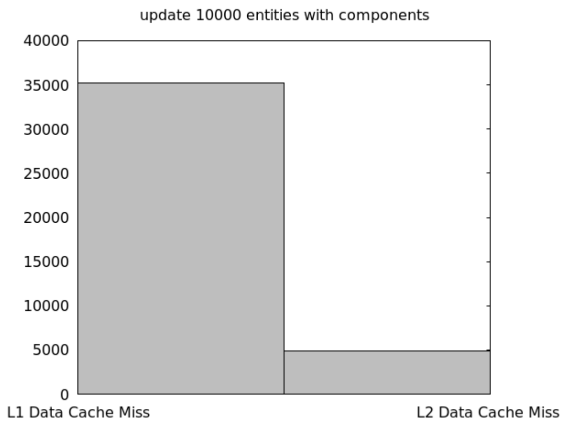


Figure 47: Rust - Updating 10000 Entities with components

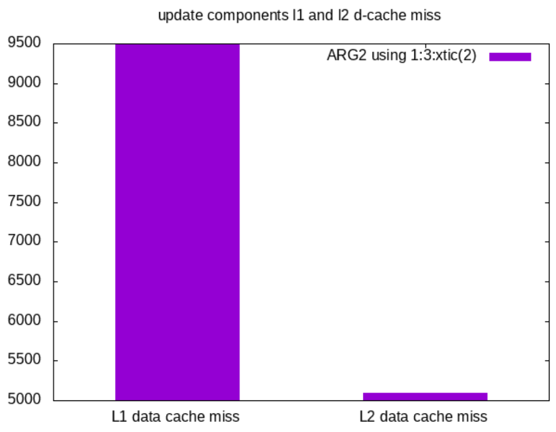


Figure 50: C++ - Updating 10000 Entities with components

It can be seen from the figures above that the cache performance of the ECS implemented in Rust is poor compared with the performance of the C++ ECS. The ECS used in C++ is named EnTT and is the fastest open source ECS framework.

The potential limitations of this framework could be attributed to a variety of factors related with differing levels of programming experience and maybe also levels of safety. One immediately identifiable limitation from the Rust ECS source code is the use of a custom Option type to keep reference to the component type rather than using the standard library Option type. A potential reason for poor cache performance could be the use of a self-implemented enum type to represent the presence of a component or lack thereof. The reason for this inclusion was to differentiate and not confuse with the Option type from the std library. In this case a compiler optimisation was not made use of. Take for example:

Component(Box<some component>)/Empty

Some(Box<some component>)/None

In the first case, a custom enum is used and in the second the standard library Option type is used. In the first case there are no optimisations performed whereas in the second case the Option of a reference is stored as efficiently as any other pointer type [43].

The following cache benchmarks show the reference CPU clock count (different from clock count since all processors perform at different speeds, this is more standardised) against a growing number of entities.

Three operations are used, create, read and write. The objective of these cache tests is to show the latencies in memory accesses. Since both libraries, EnTT and the Rust ECS use a Structure of Arrays (SOA, see appendix 9.3.2) architecture these latencies should be minimal. The graphs below show this performance for Rust ECS (Figure 51 - Figure 53) and EnTT (Figure 54 - Figure 56).

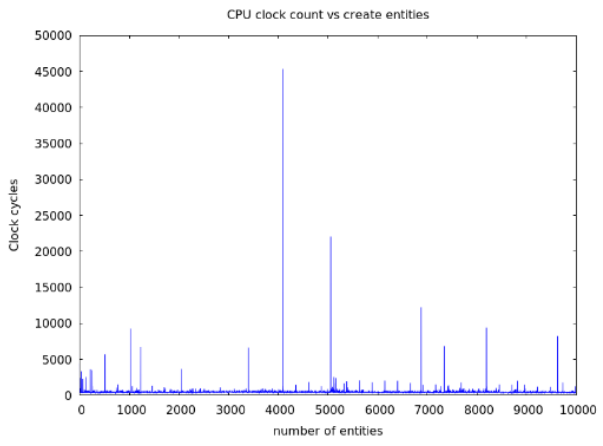


Figure 51: Rust - Adding entities to ECS

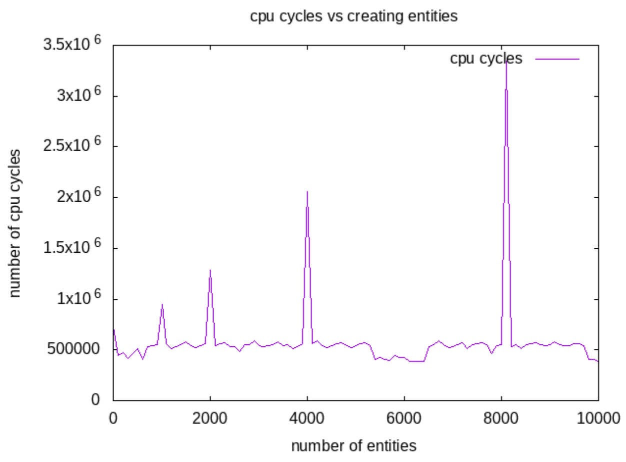


Figure 54: C++ - Adding entities to ECS

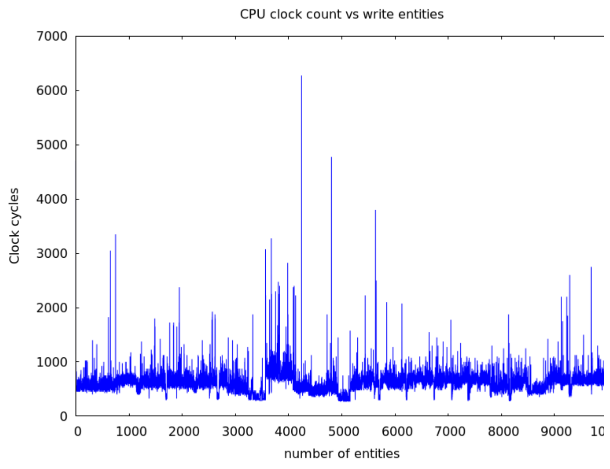


Figure 52: Rust - writing to entities in ECS

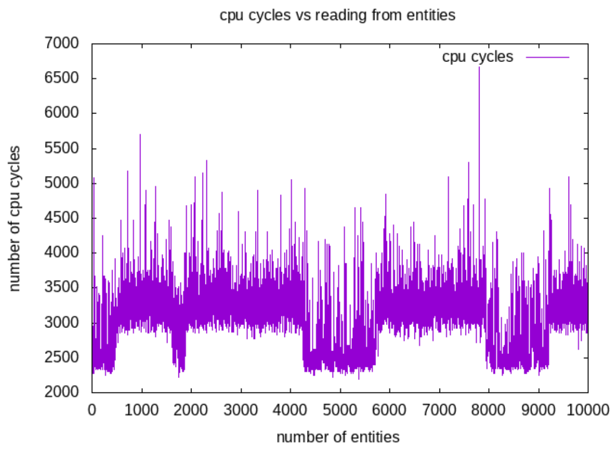


Figure 55: C++ - writing to entities in ECS

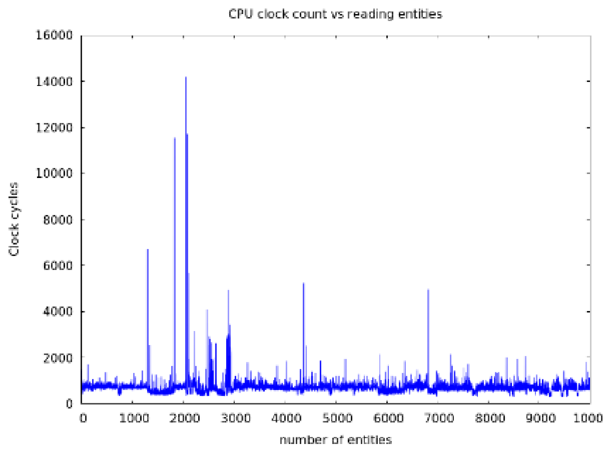


Figure 53: Rust - Reading from entities in ECS

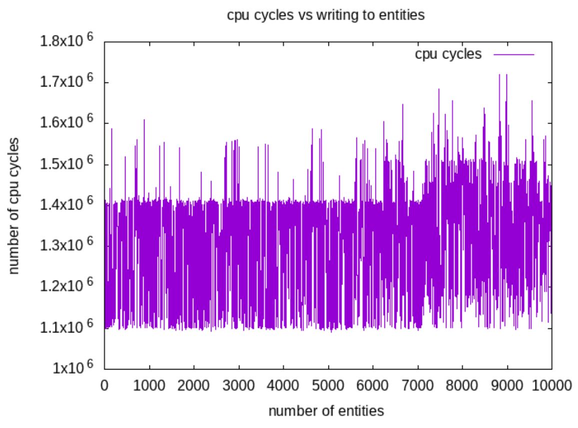


Figure 56: C++ - Reading from entities in ECS

The graphs show that the hypothesis of SOA performance is true, where the CPU clock cycles line is flat in shape. However, there is a surprising discovery where in the create operation, there are spikes in CPU clock cycles. The assumption is that the dynamically allocated array into which the entities are stored gets reallocated into a bigger chunk of memory. This causes both cache misses and a spike in the CPU clock cycles. Furthermore, these spikes occur in the same locations on each iteration of the cache performance benchmark.

It can be concluded from these benchmark tests that both approaches to creating an ECS succeed in minimising the latency from memory access by promoting data locality. Although it is not completely clear as to why there is a peak in clock cycles when creating entities, the performance in every other case is as expected.

7 Conclusion

7.1 Does Rust hold up to its claims?

Rust accomplishes what its Mozilla developers set out to achieve. This is further solidified through the creation of a performant, safe and concurrent ECS. The ECS benchmarks shown above indicate that the performance is in some cases much slower. Although this difference cannot be attributed to any specific cause, it is primarily due to a combination of the implementer's familiarity with the language and the language itself. Despite the disparity in performance, it is very possible to make efficient, well optimized games in Rust thanks to the low-level access and memory management model.

7.1.1 Friction

As for the experience of programming such a system in Rust, it cannot be described as friction-less. This is attributed to the strictness of the compiler and the tough learning curve (made slightly easier due to the extremely high quality of resources and community support). Despite this programming "friction", the trade-off being expressivity for safety, the advantages of Rust far outweigh the disadvantages in the games programming aspect. To elaborate further, the strict type system of Rust prevents common bugs found in C++ from occurring such as manipulating an uninitialized array, producing garbage data. Forcing the developer to initialize all variables to default values prevents this bug from occurring and may hinder expressing certain solutions but will prevent "undefined behaviour" from appearing.

7.1.2 Tooling

The Rust tool chain is much easier to use than the C++ tool chain. The issue with C++ is the fragmentation of the technologies used to compile and build C++ programs. There are far too many alternatives each with their own advantages and disadvantages. An example of this would be the variety in build tools where popular options seem to be CMake and Scons, which function completely differently from each other but have the same concepts. Rust tackles this by unifying and providing good tooling such as the single Rust compiler, built-in testing framework, package management and build tool. Furthermore, a tool called "rustup" manages the Rust installation, allowing one to install both stable and nightly releases and keep them up to date.

7.1.3 Libraries

The FFI interface functionality in Rust and no-overhead function calls to C++ have contributed to the use of C++ libraries (through wrappers) commonly used for games development such as the Simple Direct medial Layer (SDL) libraries. in addition, Rust has libraries such as Good Game EZ (ggez) for a Love2D-esque games programming framework, gfx-rs for platform independent low-level access to graphics APIs such as Vulkan and OpenGL which can be used to create custom graphics pipelines and various other libraries for maths, audio and physics, all of which can be used to create highly customized game engines. In addition, there are the Amethyst and Piston2D projects which are modern game engines written entirely in Rust.

These libraries compared to C++ are young excluding those that are wrappers to C++ libraries, much like the language Rust itself. C++ certainly does have the larger roster of libraries that can be used to create games as well as a fully-fledged, feature rich game engine Unreal Engine 4 (as of writing this paper) that neither Amethyst nor Piston can match (yet). Given time and community input, Rust's libraries and game engines might rival those of C++ one day.

7.1.4 Community

The Rust community is a major player in promoting the language and helping newcomers get to grips with the language's quirks and novelties. The community is lively and excited, with many forum boards, chats and in-person meetups around the world, where many great talks are given on the latest advancements and topics. It could be said that the ever-growing community of "rustaceans" rivals the participation in the C++ community.

7.1.5 Concluding

The ECS that was developed in this project meets the requirements described in section 4.4.4.1. The additional requirements listed in section 4.4.4.2 would have been implemented but for the time constraints.

Interestingly, in the process of writing the cache benchmarking tests in C++, the experience was generally frustrating but more so when forgetting to initialize the array in which results would be parsed into from PAPI. Despite the error, this project compiled and ran successfully producing results. However, the results were strange numbers, sometimes going negative as array access is unchecked.

The writer's experience with C++ is limited; this caused enough frustration to realise some of the merits of using Rust, where this sort of undefined behaviour would have never happened, saving the time spent debugging.

Finally, from the above it can be said that Rust is a serious competitor to C++ with great backing and community support. Taking the opening quote of this paper, Rust achieves what it set out to do and much more. Its great community and growing roster of libraries make it a vibrant and fun language to use regardless of how low-level one intends to get. Everyone should consider using Rust to create whatever type of application they want. As a games programming language, Rust is certainly a good choice for all levels of programmers, beginners included, to develop medium sized (indie) games. As for larger projects, such as "AAA" (big budget) games, Rust currently might not be the best fit. However, Rust's future is promising and given some time for its libraries to mature and with better profiling tools, it will have a place in the games industry.

8 References

[1] “Rust Case Study: Chucklefish Taps Rust to Bring Safe Concurrency to Video Games,” The Rust Project Developers, 2018.

[2] “sensAgent,” sensAgent, [Online]. Available: http://dictionary.sensagent.com/Undefined%20behavior/en-en/. [Accessed 26 March 2019].

[3] J. Blow, “Ideas about a new programming language for games.,” 19 September 2014. [Online]. Available: https://www.youtube.com/watch?v=TH9VCN6UkyQ. [Accessed 25 March 2019].

[4] “The Rust Programming Language,” [Online]. Available: https://www.rust-lang.org/en-US/. [Accessed 30 October 2018].

[5] “BCS Code of Conduct,” BCS - The Chartered Institute for IT, 3 June 2015. [Online]. Available: https://www.bcs.org/upload/pdf/conduct.pdf. [Accessed 3 November 2018].

[6] “Building a Rust Project,” Travis CI, [Online]. Available: https://docs.travis-ci.com/user/languages/rust/. [Accessed 31 October 2018].

[7] “GitHub Releases Uploading,” Travis CI, [Online]. Available: https://docs.travis-ci.com/user/deployment/releases/. [Accessed 31 October 2018].

[8] “Creating Releases,” Github, [Online]. Available: https://help.github.com/articles/creating-releases/. [Accessed 31 october 2018].

[9] “Benchmark tests,” rust-lang, [Online]. Available: https://doc.rust-lang.org/1.4.0/book/benchmark-tests.html. [Accessed 07 November 2018].

[10] M. Acton, “Data-Oriented Design and C++,” 29 September 2014. [Online]. Available: https://www.youtube.com/watch?v=rX0ItVEVjHc&t=403s. [Accessed 25 March 2019].

[11] J. Blow, “#Gamelab2018 - Jon Blow's Design decisions on creating Jai a new language for game programmers,” 13 July 2018. [Online]. Available: https://www.youtube.com/watch?v=uZgbKrDEzAs. [Accessed 25 March 2019].

[12] “Aliasing,” [Online]. Available: https://doc.rust-lang.org/nomicon/aliasing.html?highlight=Alias#why-aliasing-matters. [Accessed 29 March 2019].

[13] S. Klabnik and C. Nichols, “The match Control Flow Operator,” in *The Rust Programming Language*.

[14] S. Klabnic, “RefCell<T> and the Interior Mutability Pattern,” [Online]. Available: https://doc.rust-lang.org/book/ch15-05-interior-mutability.html#refcellt-and-the-interior-mutability-pattern. [Accessed 23 March 2019].

[15] “Struct std::collections::HashMap,” [Online]. Available: https://doc.rust-lang.org/std/collections/struct.HashMap.html. [Accessed 23 March 2019].

[16] s. klabnik and c. nichols, “What is ownership?,” in *The Rust Programming Language*, No Starch Press, 2018.

[17] S. Klabnik and C. Nichols, “What Is Ownership?,” in *The Rust Programming Language*, no starch press, 2018.

[18] “Rust vs. C++ Comparison,” 11 October 2018. [Online]. Available: https://www.codeproject.com/Articles/1263195/Rust-vs-Cplusplus-Comparison. [Accessed 23 March 2019].

[19] S. Klabnik and C. Nichols, “Using Modules to Reuse and Organize Code,” in *The Rust Programming Language*, no starch press, 2018.

[20] S. Klabnik and C. Nichols, “Cargo Workspaces,” in *The Rust Programming Language*, no starch press, 2018.

[21] K. Kuczmarski, “A tale of two Rusts,” 24 December 2016. [Online]. Available: http://xion.io/post/programming/rust-nightly-vs-stable.html. [Accessed 25 March 2019].

[22] T. C. S. Committee, “JTC1/SC22/WG21 - The C++ Standards Committee - ISOCPP,” The C++ Standards Committee, [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/.

[23] A. Turon, “Abstraction without overhead: traits in Rust,” 11 May 2015. [Online]. Available: https://blog.rust-lang.org/2015/05/11/traits.html. [Accessed 23 March 2019].

[24] “Null Dereference,” OWASP, [Online]. Available: https://www.owasp.org/index.php/Null_Dereference. [Accessed 29 March 2019].

[25] L. Tratt, “A Quick Look at Trait Objects in Rust,” 12 February 2019. [Online]. Available: https://tratt.net/laurie/blog/entries/a_quick_look_at_trait_objects_in_rust.html. [Accessed 23 March 2019].

[26] I. D. Scott, “Exploring Rust Fat Pointers,” 28 May 2018. [Online]. Available: https://iandouglasscott.com/2018/05/28/exploring-rust-fat-pointers/]. [Accessed 23 March 2019].

[27] S. Klabnik and C. Nichols, “Unsafe Rust,” in *The Rust Programming Language*, no starch press, 2018.

[28] S. Bilas, “A Data-Driven Game Object System,” Gas Powered Games, 2002.

[29] D. Hall and Z. Wood, “ECS Game Engine Design,” San Luis Obispo, 2014.

[30] J. Kulandai , “javapapers,” 09 24 2012. [Online]. Available: https://javapapers.com/core-java/why-multiple-inheritance-is-not-supported-in-java/. [Accessed 31 October 2018].

[31] A. Martin, “T-Machine,” T-Machine, 11 November 2007. [Online]. Available: http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/. [Accessed 30 October 2018].

[32] “The Specs Book,” slide-rs, [Online]. Available: https://slide-rs.github.io/specs/. [Accessed 31

October 2018].

[33] R. Nystrom, “Component,” in *Game Programming Patterns*, genever benning, 2014, pp. 226 - 228.

[34] “Jonathan Blow on using his own engine...,” Reddit, 2014. [Online]. Available: https://www.reddit.com/r/Games/comments/282l0k/jonathan_blow_on_using_his_own_engine_i_would/. [Accessed 3 November 2018].

[35] A. Myles, “downcast-rs,” [Online]. Available: https://github.com/marcianx/downcast-rs. [Accessed 25 March 2019].

[36] “Tokio,” tokio-rs, [Online]. Available: https://tokio.rs/. [Accessed 23 March 2019].

[37] “shred,” slide-rs, [Online]. Available: https://github.com/slide-rs/shred. [Accessed 23 March 2019].

[38] “Core Concepts for Beginners,” travis-ci, [Online]. Available: https://docs.travis-ci.com/user/for-beginners/. [Accessed 23 March 2019].

[39] “Understanding the data under a chart,” [Online]. Available: http://www.serpentine.com/criterion/tutorial.html#understanding-the-data-under-a-chart. [Accessed 23 March 2019].

[40] “lschmierer/ecs_bench,” [Online]. Available: https://github.com/lschmierer/ecs_bench. [Accessed 23 March 2019].

[41] abeimler, “ecs_benchmark,” [Online]. Available: https://github.com/abeimler/ecs_benchmark. [Accessed 25 March 2019].

[42] F. Zeitz, “Rust Cologne, March 2018: Florian Zeitz - Caches and You,” 16 March 2018. [Online]. Available: https://www.youtube.com/watch?time_continue=1588&v=GQBI-TcOTOk. [Accessed 23 March 2019].

[43] “Module std::option,” [Online]. Available: https://doc.rust-lang.org/std/option/. [Accessed 23 March 2019].

[44] R. Fabian, “Data-Oriented Design,” 08 October 2018. [Online]. Available: http://www.dataorienteddesign.com/dodbook/node2.html. [Accessed 5th November 2018].

9 Appendix

9.1 Project Proposal

Candidate No.: 146782

Supervisor: Dr Martin Berger

Rust for game engine development and games programming

Aims and objectives:

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. – Mozilla

The above statement taken from the homepage of the Rust Language highlights nicely the benefits of using Rust as a systems programming language. These benefits are extremely useful when programming video games (VG) and creating game engines (GE). Video game and game engine software are heavily reliant on performance and consequently are developed in languages like C and C++, which are industry standards. These languages, however, can be very “memory unsafe”. This causes a lot of undefined behaviour leading to long hours of debugging, which is a waste of time and money for developers. My dissertation project is to show that Rust is a better alternative for creating performant applications like game engines while not having segfaults and as a result avoiding the undefined behaviour problem. Rust, furthermore, promotes a data-oriented design which greatly simplifies the process by which one would architect a game engine and subsequently games. C++ has an object-oriented approach which does more harm than it seemingly should when architecting games.

The primary objectives of my project would be:

- To show whether Rust is a viable alternative from C and C++ for game engineering and game development.
    - I would go about proving this through experimentation, reading and papers on the subject matter.
    - Getting the opinions of those in industry, analysing them and providing as objective as possible an answer after consideration of all viewpoints.
- I would like to develop an Entity Component System (ECS) which can perform operations on many entities. I would code such a system in Rust, using best practices enforced by the language.
    - This would give me first-hand experience of developing in Rust, allowing me to add to the opinions of others and form my own to further validate the previous point.
- I would then lead on to benchmark the operations performed on the entities, validating or disproving my point about efficiency in Rust.
- Another objective would be to contrast my implementation with a text book example from C++ or C.

Extensions:

- Create as many of the following components of the game engine as time permits:
    - Physics engine
    - Sound engine
    - Networking
    - Rendering

Relevance:

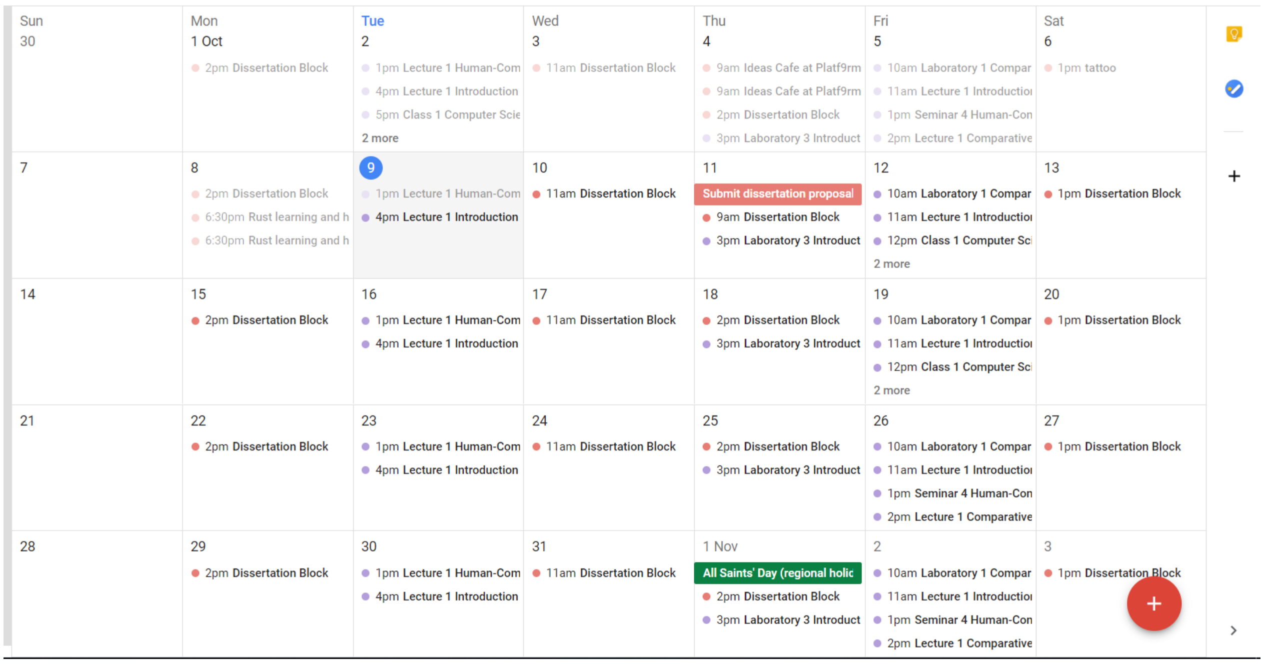
From my degree course, I will be using knowledge gained from my first, second and Industrial Placement years. The relevant knowledge listed briefly includes concepts covered in "Intro to programming", "Further programming", "Programming Concepts", "Mathematical Concepts", "Data structures and Algorithms", "Introduction to computer systems", "Compilers and Computer Architecture", "Program Analysis", "Operating systems", "Software Engineering" and finally knowledge gained from my Placement Year. My dissertation project further supports the following course objectives, which are as follows:

1. Demonstrate knowledge and understanding of the fundamental principles of computer science.
2. Demonstrate knowledge and understanding of how these principles, logical concepts, data structures, algorithms and design approaches such as object orientation can be used to develop software-based solutions.
3. Use appropriate criteria and tools for the planning, development, documentation, testing and evaluation of software systems.
4. Specify, design, construct, test and evaluate computer-based systems using appropriate tools. Recognise the capabilities and the limitations of computer-based solutions.

Resources:

The list of resources that I am currently using "The Book", which is an official Rust tutorial book produced by Mozilla and is freely available to read online. Other resources involved watching two videos, one at Rustconf 2018 by Catherine West and another, which was a response by Jonathon Blow. Catherine West and Jonathon Blow are both giants in the games industry as they have both put out commercial games and as a result are highly experienced in the subject area. Further resources that will be used are Rust User Group meetups in London where people convene to discuss Rust and work together on challenges/hacking problems. Another valuable resource is game programming patterns by Robert Nystrom. This book highlights common programming patterns used in the games industry and will be useful when trying to provide useful abstractions when programming my sub module. Furthermore, another useful resource, which covers in general all the concepts used in developing game engines is Game Engine Architecture by Jason Gregory. Open source code will be consulted such as the UE4 code repository and other open source game engines.

Timetable:



Sun	Mon	Tue	Wed	Thu	Fri	Sat
30	1 Oct ● 2pm Dissertation Block	2 ● 1pm Lecture 1 Human-Com ● 4pm Lecture 1 Introduction ● 5pm Class 1 Computer Scie 2 more	3 ● 11am Dissertation Block	4 ● 9am Ideas Cafe at Platf9rm ● 9am Ideas Cafe at Platf9rm ● 2pm Dissertation Block ● 3pm Laboratory 3 Introduct	5 ● 10am Laboratory 1 Compar ● 11am Lecture 1 Introductio ● 1pm Seminar 4 Human-Con ● 2pm Lecture 1 Comparative	6 ● 1pm tattoo
7	8 ● 2pm Dissertation Block ● 6:30pm Rust learning and h ● 6:30pm Rust learning and h	9 ● 1pm Lecture 1 Human-Com ● 4pm Lecture 1 Introduction	10 ● 11am Dissertation Block	11 Submit dissertation proposal ● 9am Dissertation Block ● 3pm Laboratory 3 Introduct	12 ● 10am Laboratory 1 Compar ● 11am Lecture 1 Introductio ● 12pm Class 1 Computer Sci 2 more	13 ● 1pm Dissertation Block
14	15 ● 2pm Dissertation Block	16 ● 1pm Lecture 1 Human-Com ● 4pm Lecture 1 Introduction	17 ● 11am Dissertation Block	18 ● 2pm Dissertation Block ● 3pm Laboratory 3 Introduct	19 ● 10am Laboratory 1 Compar ● 11am Lecture 1 Introductio ● 12pm Class 1 Computer Sci 2 more	20 ● 1pm Dissertation Block
21	22 ● 2pm Dissertation Block	23 ● 1pm Lecture 1 Human-Com ● 4pm Lecture 1 Introduction	24 ● 11am Dissertation Block	25 ● 2pm Dissertation Block ● 3pm Laboratory 3 Introduct	26 ● 10am Laboratory 1 Compar ● 11am Lecture 1 Introductio ● 1pm Seminar 4 Human-Con ● 2pm Lecture 1 Comparative	27 ● 1pm Dissertation Block
28	29 ● 2pm Dissertation Block	30 ● 1pm Lecture 1 Human-Com ● 4pm Lecture 1 Introduction	31 ● 11am Dissertation Block	1 Nov All Saints' Day (regional holic ● 2pm Dissertation Block ● 3pm Laboratory 3 Introduct	2 ● 10am Laboratory 1 Compar ● 11am Lecture 1 Introductio ● 1pm Seminar 4 Human-Con ● 2pm Lecture 1 Comparative	3 ● 1pm Dissertation Block

Shareable Link: https://calendar.google.com/calendar?cid=dmVua2F0ZXNoZ292aW5kMUBnbWFpbC5jb20

9.2 Project Plan

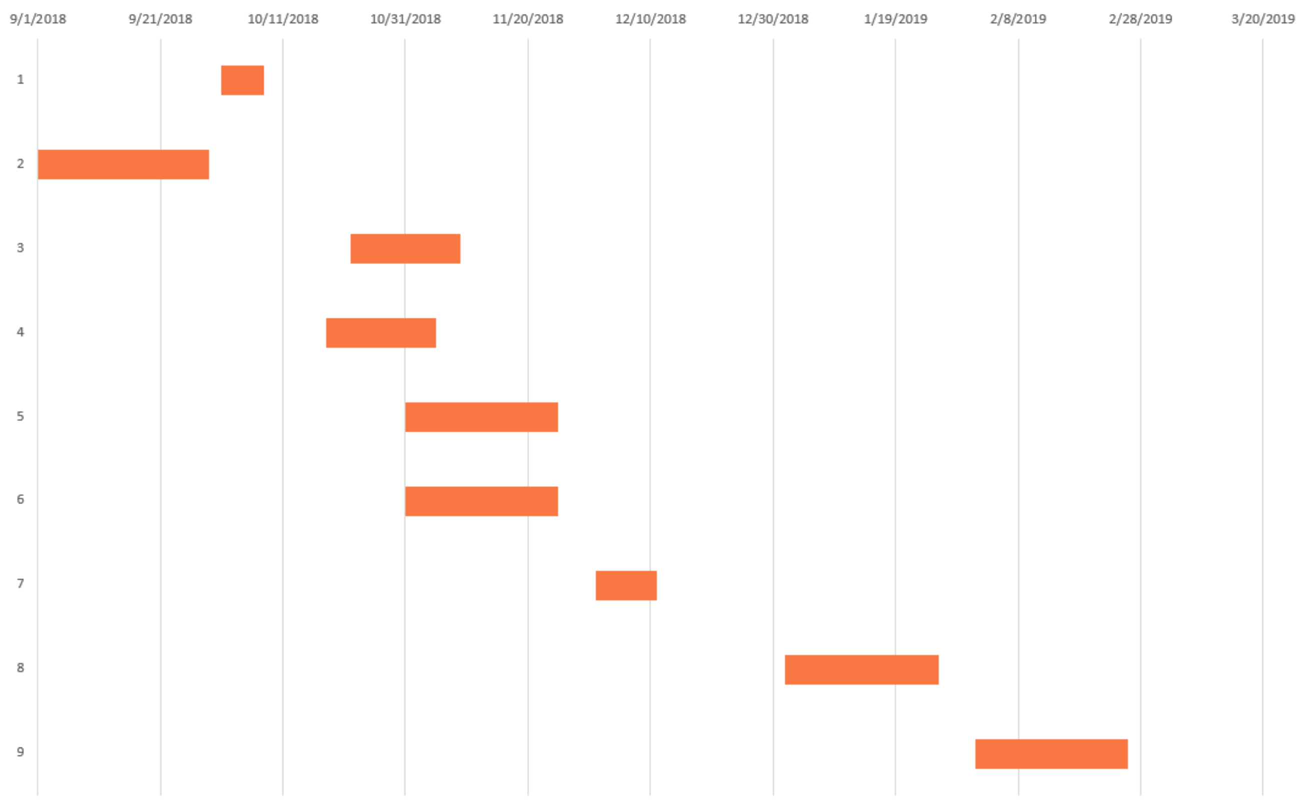


Figure 57 Gantt chart

	activity	start date	predecessor	time estimates (Days)		
				optimistic	normal	pessimistic
1	project proposal	10/1/2018		5	7	9
2	Rust Lang research	9/1/2018		21	28	30
3	game engine / ECS research	10/22/2018	2	15	18	25
4	Interim report	10/18/2018	2	15	18	21
5	ECS programming	10/31/2018	2	20	25	30
6	ECS testing	10/31/2018		20	25	30
7	ECS analysis	12/1/2018	5	7	10	15
8	ECS improvements	1/1/2019	6	20	25	30
9	Write up	2/1/2019	6	20	25	30

9.3 Data Oriented Design

9.3.1 Introduction

Data oriented design is an architectural pattern that places importance on the layout of data in memory. The OOP way would have a problem space represented in code as a set of abstractions, whereas in data-oriented design, the problem space is represented as data in memory, with transforms being applied to it. In this sense data-oriented design focuses on what the data looks like and how it’s laid out, and the transforms (logic) conducted on the data and their results. To note, transforms and data are clearly separated here whereas in OOP they are bundled together in a class. The assumption would be that the developer would know something about the target platform’s hardware, and so would understand better how to layout data in memory for optimal performance. This is an attempt to separate logic from data, improve performance and allow for better modularity. This design pattern is very useful for games, since these previously mentioned aspects can cater for platform specific efficiency and scalability/modularity [44].

9.3.2 SOA vs AOS

The Structure of Arrays (SOA) transformation is commonly used to benefit from fast memory accesses by taking advantage of data locality. Data locality is achieved by keeping all runtime data in contiguous memory, it is then possible to take advantage of cache prefetching, reducing the time taken to retrieve data from main memory [44].

```
typdef struct {

        float x;

        float y;

        float z;

} vec;


vec Array[100];
```

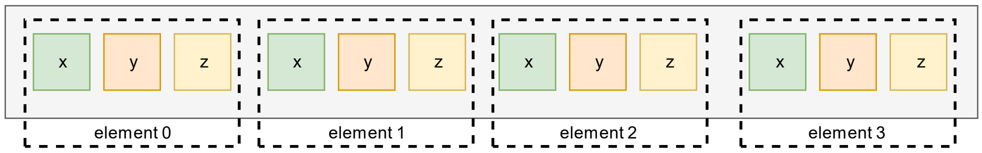


Figure 58: Array of Structures

```
typdef struct {

        float x[100];

        float y[100];

        float z[100];

} vec;


vec vector;
```

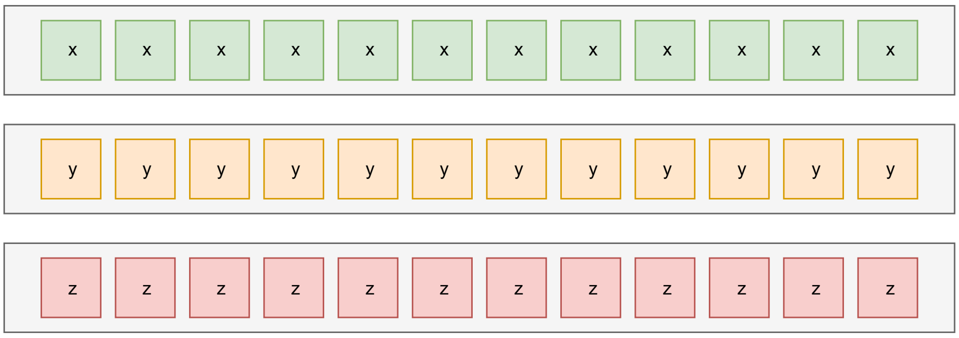


Figure 59: Structure of Arrays

In Figure 58 there is one array in which all 3 variables are stored, this pattern is Array of Structures (AoS). This is a valid approach to storing data but in some cases, this is less than adequate. Take for example a case in which only the x variable is processed, the y and z are not. Here, all three variables would have to be fetched and stored in cache. Should the data in each variable be large enough to fill cache before prefetching all elements there would be significant pauses as processing occurs.

To combat these slow pauses to fetch data from main memory, Figure 59 shows the SoA pattern. In this architecture each variable is stored separately that way when processing only the x variable, it is the only data that populates cache, reducing the need to pause for data retrieval from main memory.

9.4 Source code links

The following links contain the source code developed for the project. Build instructions exist within each project.

Non-concurrent ECS:
https://github.com/bonorumetmalorum/game_engine/releases/tag/submission_non_concurrent

Concurrent ECS :
https://github.com/bonorumetmalorum/game_engine/releases/tag/submission_concurrent

C++ EnTT cache benchmarks:
https://github.com/bonorumetmalorum/cpp_ecs_benchmarks/releases/tag/Submission

Unfortunately, due to the limitation of Travis-CI (continuous integration provider) only simplistic benchmarks can be found at:

https://s3-eu-west-1.amazonaws.com/enginebenchmark/18/18.1/report/index.html

These benchmarks reflect the up-to-date concurrent version of the ECS.

All these benchmarks can be re-run by downloading the projects and following the instructions provided in the respective ReadMe files.

It is recommended that the non-concurrent branch is used for benchmarking as this was used in the report. The names of the benchmarking tests have been updated for clarity.