3-26-2020

A Study of Execution Performance for Rust-Based Object vs Data Oriented Architectures

Joseph A. Vagedes

A STUDY OF EXECUTION PERFORMANCE FOR RUST-BASED OBJECT VS DATA ORIENTED ARCHITECTURES

THESIS

Joseph Vagedes, 2nd Lt, USAF

AFIT-ENG-MS-20-M-065

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-20-M-065

A STUDY OF EXECUTION PERFORMANCE FOR RUST-BASED OBJECT VS DATA ORIENTED ARCHITECTURES

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Joseph Vagedes, B.S.C.E.

2nd Lt, USAF

March 19, 2020

AFIT-ENG-MS-20-M-065

A STUDY OF EXECUTION PERFORMANCE FOR RUST-BASED OBJECT VS DATA ORIENTED ARCHITECTURES

THESIS

Joseph Vagedes, B.S.C.E.
2nd Lt, USAF

Committee Membership:

Douglas D Hodson, Ph.D
Chair

Scott L Nykl, Ph.D
Member

Ryan D Engle, Ph.D
Member

AFIT-ENG-MS-20-M-065

Abstract

In recent years, advances in the performance of computer hardware technology has begun to slow due the physical limitations of modern transistors including feature size and heat dissipation issues [1, 2]. Engineers have turned to adding additional cores to Central Processing Units (CPUs) to improve performance, however many modern day applications are not designed to take advantage of multi-core parallelism effectively. Software developers are no longer able to rely on improved hardware for increased performance in their software; they must turn to software optimizations.

The memory layout of a software system is one area that directly contributes to the performance of the software as it affects access time and throughput of data in cache [1]. This area of optimization tends to be overlooked by developers as they do not have the necessary technical background in computer architecture. In general, the commonly used programming paradigm, Object-Oriented Programming (OOP), does not store data in memory in such a way that it can be optimally used by the cache, resulting in slower performance.

This research investigates the Data-Oriented Design (DOD) paradigm, in particular, an architecture built off its principles: Entity-Component-System (ECS). ECS is commonly used by video game engines due to its ability to store data in a way that is optimal for the cache to access. Additionally, the structure of this paradigm produces a code-base that is simple to parallelize as the workload can be distributed across a thread-pool based on the data used with little to no need for data safety measures such as mutexes and locks. A final benefit, although not easily measured, is that the DOD paradigm produces a highly decoupled (i.e., a strong separation of concerns) code-base, resulting in more easily maintainable and extensible code.

DOD and ECS are not a catch-all replacement for OOP; they are most optimal for software systems with large amounts of data being used and operated on consistently [3]. This is why it is actively being developed and used in video game engines. This research argues that due to the similarities between video game engines and real-time simulators, this paradigm can replace the commonly used OOP paradigm in real-time simulators to improve performance.

To demonstrate the performance differences between the two paradigms for use in real-time simulators, two separate code-bases were developed, each using one of the paradigms. Multiple experiments were run on each code-base to determine how tolerant each was to changes in important aspects of simulations such as the amount of data used in a simulation. The DOD paradigm consistently outperformed the OOP paradigm in all experiments; it was seen that as more data is used in an experiment, the larger the difference in performance between the DOD and OOP paradigms.

This research closes by emphasizing that DOD is not a replacement to OOP in all use cases. DOD is most optimal in code-bases that contain and operate on a large amounts of data; this includes, but is not limited to, areas such as video games engines, real-time and non-real-time simulations, and high performance computing. This paradigm also has a strong use case for the development of multi-threaded applications as it reduces the complexity of developing the software system as methods for operating on data can be automatically distributed across a threadpool by a scheduler. This reduces, if not eliminates, the need for developers to think about data safety and mechanisms such as mutexes and locks.

Table of Contents

Page

Page

List of Figures

List of Tables

A STUDY OF EXECUTION PERFORMANCE FOR RUST-BASED OBJECT VS DATA ORIENTED ARCHITECTURES

I. Introduction

1.1 Background

Real-time simulators share many similarities with current games; they both manage a world composed of a set of interacting entities. The main differences between the two are how they are developed, how precisely they represent the real world, and their intended purpose. In real-time simulators, it is common for a developer to be responsible for handling all aspects of the software - from networking to memory-management and parallelization. These aspects must be meticulously developed due to their importance in the overall performance of the simulator [4, 5]; this makes developing these aspects difficult and slow. Maintainability also becomes an issue for developers; it may be necessary to refactor large portions of the code-base when upgrading or implementing new features due to tightly-coupled code [6, 7].

Games, on the other hand, are primarily developed using game engines. These engines handle the features that cause (or prevent) high performance such as the aforementioned networking, memory-management, and parallelization. Aspects such as these have been precisely developed and tweaked to be as efficient as possible. This allows a game developer to focus on developing features while not necessarily needing to worry about the overall organization of the architecture, which affects performance. An architecture that is becoming increasingly popular in the game development industry is the Entity-Component-System (ECS). This architecture is dual-hatted; it not only handles memory management and code parallelization, but it also forces the programmer to develop using the Data-Oriented Design (DOD) paradigm. A stand-out feature of DOD is that it produces highly decoupled code, effectively compartmentalizing pieces of the code-base. These features allow it to be highly extensible and easy to maintain due to the few dependencies between sections of the code-base. Code can be added and removed without necessarily effecting other parts of the code-base. This thesis claims that there are enough similarities between real-time simulators and game engines that implementing the ECS architecture for a real-time simulator could greatly improve the design and overall performance of real-time simulators.

1.2 Problem Statement

With the slowing advances in computer hardware technology, mainly the clock frequencies at which hardware runs, it is becoming increasingly important to optimize software systems for increased performance. Many optimizations have already been made in the realm of game development due to the intense computational requirements, however, the problem is that these same optimizations have not been made in other legacy software, including real-time simulators [8, 9]. These optimizations in games can be attributed to game engines. Game engines define how data is stored and accessed; they ensure how it is stored in an efficient way - the basis of the DOD paradigm. Older, or legacy, simulators on the other hand tend to focus on development through Object-Oriented Programming (OOP). The benefits of each, along with similarities and differences, are explained in chapter II.

This thesis hopes to determine if real-time simulators can implement software architectures originally developed and optimized for game engines to increase their performance and ease of development through the compartmentalized nature and memory layout of the DOD paradigm as explained in chapter II. If so, the DOD paradigm, and in particular, the ECS architecture are possible solutions to improving the performance legacy, or to-be developed modeling software, including real-time simulators.

1.3 Research Goals

The goal of this research is to investigate the performance differences between real-time simulators that use an OOP approach for computation and one that use a DOD approach and answer the question as to if DOD, in particular the ECS architecture, is a possible solution to developing high-performance real-time simulators.

1.4 Hypothesis

This research hypothesizes that games and real-time simulations share enough similarities that it is not only possible to implement a software architecture originally developed for games (the ECS architecture), but that it will also improve the performance considerably. It also theorizes that the code base produced due to this architecture will increase the maintainability of the software due to the compartmentalized nature of the DOD paradigm.

1.5 Approach

Two code-bases were created; one used the OOP paradigm and the other used the DOD paradigm (more specifically the ECS architecture). They were tasked with performing the same calculations on the same data-sets to simulate a real-time system. Comparisons in performance (how long it takes to execute all systems on all entities) are taken for each code-base and compared. During each set of tests, one of the below variables are changed:

- Size of each entity
- Number of entities
- Number of components used in a system
- Number of threads
- Number of systems

1.6 Assumptions/Limitations

The following assumptions/limitations are understood when designing and implementing the OOP and DOD code bases:

- Methods for operating on the state of entities are generalized to do resource-intensive, but meaningless calculations for simulating realistic computation.
- The optimizer settings for the compiler will be set to as low as possible to ensure data requests from memory are not optimized away. Additionally, this is to ensure compiler does not change the structure of the code base or how either the DOD or OOP architectures are executed.
- Both architectures will be ran on a 2018 Macbook Pro (MR9Q2LL/A) with a 2.3GHz Quad-Core Intel i5 processor (i5-8259U) with the following cache sizes:
  - L1: 4 x 32 KB
  - L2: 4 x 256 KB
  - L3: 6 MB
  - L4: 128 MB

1.7 Contributions

This thesis contributes to the fields of:

- **Modeling & Simulation:** DOD is a commonly used paradigm in games to greatly improve performance. As their are many similarities between games and real-time simulations, the same concepts, paradigms, and architectures are applicable to these simulations. By implementing these in real-time simulators, vast improvements to performance could be made.

- **High Performance Computing:** The ECS architecture can be scaled to as many threads as needed. Systems are run on separate threads where possible (no data races). If systems handle different sections of the data sufficiently, the computations may be paralleled to a high degree. If enough entities exist, multiple threads can handle the same calculations on different sets of entities, providing further parallelism.

1.8 Thesis Overview

This thesis is arranged into five chapters. Chapter II provides important background information on relevant computer hardware pieces that directly impact a computer's performance and different software paradigms used in common programming languages today. It also explains how each paradigm works with or against the computer hardware when used. Chapter III provides the experiment methodology, along with the respective code snippets for each run. Chapter IV shows the analysis of the data obtained from Chapter III and display the results in an efficient manner. Finally, Chapter V summarizes the research and impact of the research, while also providing future research opportunities based off this research.

II. Background

2.1 Overview

This chapter will provide an overview of the hardware components that directly affect the performance of code written using Object-Oriented Programming (OOP) and Data-Oriented Design (DOD) paradigms. It will then provide a technical summary of the aforementioned paradigms while characterizing the memory layout produced when used. It will explain how these paradigms work with or against the hardware while also identifying the advantages and disadvantages of each. It will end with a detailed explanation of the Entity-Component-System (ECS) architecture and one full implementation, Specs, used in the experimentation of this thesis.

2.2 Computer Hardware

It is common knowledge that the Central Processing Unit (CPU) is central to any computer or embedded system; it is responsible for fetch, decode, and execution of instructions. Modern processors rely heavily on multiple-core parallelism to increase performance as with current technologies, designers are no longer able to easily shrink feature sizes, or increase clock speeds, due to the physical limitations of transistors and issues with heat dissipation [1].

Each processor in a multi-core system has its own instruction pipeline and cache for which it operates on. When processors are working on different sets of data, there is no issue as each set of data is located only in the cache of that specific processor. However, when processors work on the same data simultaneously, performance issues and even data corruption issues (such as data races) can occur due to the same data being located in multiple caches that must be in sync when one writer exists. [1].

Modern cache follows a coherence protocol that is responsible for ensuring shared data among multiple caches remain consistent [1]. Many protocols exist, but the most commonly used is the Modified-Exclusive-Shared-Invalid (MESI) protocol which uses the four states (as seen in the name) to control what can and cannot be done to the cache. However, this does not prevent data races and memory corruption in all cases: the cache coherency protocol can only ensure data safety for primitive types depending on the processor. This means that a program can safely write an entire primitive before another thread will be able to read or overwrite that data. This prevents data races for primitive types. However, if a non-primitive data type is being used, such as a string, or a custom class, it cannot ensure data safety [10].

As an example, imagine a string is being written to by multiple threads. A string is not a primitive type, it is a wrapper around a list of primitive characters. When writing to a string (appending, overwriting, etc), each individual character is edited one at a time. The cache coherence protocol will ensure that each individual character is written to without issue, but determining which thread is writing to each character will be non-deterministic. To ensure issues like these do not occur, it is important to use a thread safety mechanism such as atomics or mutexes [11]. These mechanisms should even be used when operating on primitives as it will ensure the compiler does not reorder or remove operations when optimizing the code. Atomics and mutexes act as fences to ensure code is not reordered incorrectly.

Atomics exist at multiple levels: (i) Hardware, (ii) Operating system, and (iii) Programming language; however, true atomics only exist at the hardware level whereas mutexes are handled purely in software [12]. When a programmer signifies that a primitive type is atomic, the compiler uses special hardware locks and/or special operation codes to perform an operation in one instruction, or make it appear to other threads as if it was performed in one instruction. Atomics are only for single operations such as an add command [12, 11]. Mutexes, on the other hand, are data structures that allow you to lock sections of code and perform multiple operations before unlocking (known as the critical area). Mutexes prevent other threads from accessing the critical area while locked.

The cache is responsible for providing instructions and data to the processor. It is split into different levels that get further away (physical distance) from the processor. At each level, the size of the cache gets larger and the access time gets slower as seen in Figure 1. Without this cache model, the CPU would almost always be idle while it is waiting for instructions and data to come from memory as the performance of a CPU greatly outpaces that of the cache [1].

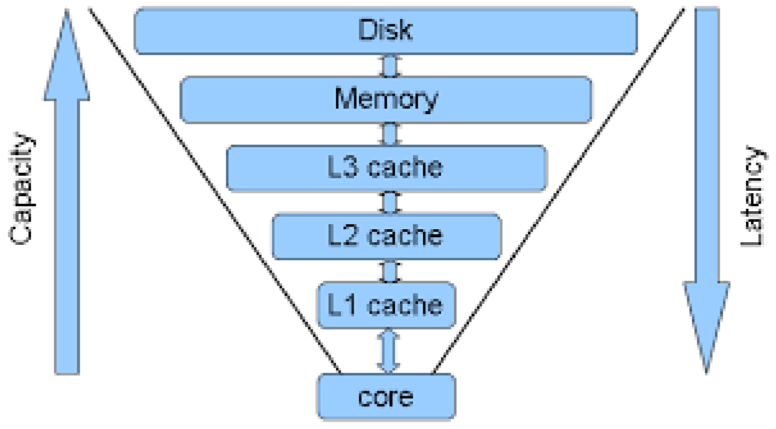

Figure 1: Generic Cache Hierarchy

With this model, the most recently used data and instructions are in the lowest levels of cache while the least recently used get progressively farther away. It is important to note that any data and instructions located in a particular level of cache is also located in all higher levels of cache. When a CPU requests instructions or data from memory, the cache model searches for it in the lowest level memory. If the requested data is found, it is considered a cache hit. If not found, it is a cache miss and it must be searched for in the next level of cache. This process continues until a cache hit occurs. Once the requested data or instructions are found, it not only provides that data to the CPU, but it also moves data/instructions down the cache hierarchy to the lowest level cache for future use. Cache misses are costly in terms of performance, changing the time it takes to get the requested data to the processor from a few clock cycles, to hundreds, thousands, or even hundreds of thousands of clock cycles (depending on where it is found in the cache hierarchy). During this time, the CPU either becomes idle, or more frequently, will context switch to another process while the requested data is being obtained [1].

Current cache models have hit rates of over ninety percent [1]; this is made possible due to the concepts of spatial and temporal locality. Spatial locality is the notion that commonly accessed data are close to each other in memory while temporal locality is the notion that if data is accessed, it will be accessed again. The cache model takes advantage of spatial locality by bringing in more than just the requested data into the lowest level cache. Data inside of a cache is split up into multiple blocks, varying in size depending on the make of the cache. If the CPU requests a specific piece of data, the entire block the piece of data resides in is brought into the CPU's cache. This improves spatial locality as it means much of the data around the requested data is ready to be used by the CPU immediately. The cache model takes advantage of temporal locality simply by the fact that it keeps the most recently used data in the lowest levels of cache and does not remove them unless replaced by another block. This is generally done via a least recently used decision, meaning the block that has not been used in the longest time is replaced with the incoming block. [1].

Of the two concepts, spatial locality is almost entirely dependent on the developer, and has the greatest effect on performance. The latter is also dependent on the developer, but occurs naturally in code as it is common and logical to work with the same data throughout a program. Spatial locality is important for a programmer to keep in mind when developing the code base as the way it stores and accesses data greatly affects performance [1].

If data is stored in an inefficient way, data contention issues between cores arise, degrading the overall performance of the program. To develop an efficient multi-threaded program, the goal should be to minimize memory bus traffic between cores. This can most easily be done by minimizing core interactions by minimizing shared data between each core [4]. The DOD and OOP paradigms have different memory models. The interactions between the core(s), cache, and these memory models will be explained in their respective sections, but it is important to note that they play an large role in the overhead performance of any programs developed using these paradigms.

2.3 Object-Oriented Programming

OOP is the most common paradigm used and is taught to nearly all students learning software development. OOP can be separated into four main areas: (i) encapsulation, (ii) abstraction, (iii) inheritance, and (iv) polymorphism. The focus of encapsulation is to wrap commonly accessed state (i.e. data inside a class) into a class (or object) that the programmer can use as if it was a single piece of data [13]. When these objects are instantiated during run-time execution, each piece of state in the object is located directly next to each other. Table 1 presents a simplified view of an OOP object in memory when instantiated, ignoring any overhead data.

Memory Address (Hexadecimal)					
0x00	0x04	0x08	0x0C	0x10	0x14
state1	state2	state3	state1	state2	state3
Object1			Object2		

Table 1: Memory Layout of Two OOP Objects

Encapsulation becomes a performance issue when developing a multi-threaded program. This is due to how each object is stored in memory as specified in Table 1. When an object is shared between threads, the entire object is loaded into the cache of both threads (as explained in section 2.2), regardless of what state is actually being used by each thread. This is an issue as valuable cache space is being used by unnecessary state, which means less of the necessary state can fit in the cache at any given time. This hurts the overall performance as it increases the amount of cache misses during execution as not all of the required state fits in the cache at any given time.

A benefit of encapsulation is that state can be made private, allowing the developer to better control how an object is manipulated [13]. This is because private state cannot be directly accessed (reading or writing) and the developer must choose to create methods that allow for interaction with the state of an object. Encapsulation helps to protect the object by preventing users from illegally or incorrectly updating the state of it by forcing state manipulation to be done through one of the public methods. An added benefit is that the class becomes easier to update: if the developer wants to change how a particular piece of state is manipulated, they can update the method accordingly. As long as the method interface (the inputs and outputs) do not change, none of the existing code-base will be affected. By creating methods for manipulating state, the second use-case of OOP becomes apparent - abstraction.

Abstraction is used to make understanding the code-base simple, along with reducing the complexity of the code from the view of the programmer [13, 14]. The goal is to present an interface to the programmer that is easy to use and works consistently. This allows the programmer to use the interface without necessarily understanding how it works. Consider any of the numerous sorting algorithms available in the standard library of most programming languages - these are perfect examples of abstraction. The programmer must only provide a data set and a pattern to sort on; all of the complex state manipulation that occurs on the data set is abstracted away and only a sorted data set is returned to the user. This is highly beneficial to the development cycle of any software product as the developer does not have to focus on the inner workings of every specific piece of code, but can abstract away many complexities and focus on the bigger picture, or the problem at hand.

Inheritance is one of the two major use-cases for OOP. It is the act of creating a new class that receives all the state and methods from a different class, while also allowing the user to add additional state and/or methods [14]. This promotes code reuse without the need of rewriting the code. Additionally, the programmer can choose to make state private that was not necessarily private in the parent class (though a programmer cannot make private data public without using a work-around such as a get() function). The programmer may also choose to override a method that was inherited from a parent class. This allows the programmer to ensure the method works correctly with any changes made in the child class.

As a class grows large, it becomes increasingly slow to move an instance of the class in memory as the entire object must be copied to a new location in memory. Once it grows too large, it becomes necessary to pass these objects around by reference (the memory location of the object). Passing these objects around by reference is still inherently slow due to jumping through pointers, but it is more efficient than copying large amounts of data in memory. Aside from performance loss, inheritance also tightly couples the two classes together; a change in a parent class will almost assuredly affect any child classes below (but not vise-versa). It can also be extremely difficult to refactor or delete a parent class as it will cause a ripple effect of changes down the class hierarchy. This will cause maintenance and development of the code-base to become slow and difficult.

The final area of OOP is polymorphism. Polymorphism is the ability for objects of different types to have the same interface and be treated as if they were the same object type [13]. As stated previously, inheritance guarantees that a child class has all of the state and methods as a parent class. Due to this, the child class can be treated as if it was the a parent class. The implementation of polymorphism is different in every language, but is typically done by instantiating the child class with the interface of the parent class.

There are two types of polymorphism: (i) static and (ii) dynamic. Static polymorphism is done at compile time and is typically used when overloading methods (though it differs per language) [15]. Method overloading is when a function has the same name, but different inputs or different functionality. This can be done for any functions, or methods associated with a class. They may have different return types, but that is not necessary to be considered overloaded. Functions can be overloaded in the same class or between parent and child classes. This is considered static as the compiler generates a separate function at compile time for each overloaded method that is used in the program. This may slow compilation, but does not affect performance.

Dynamic polymorphism occurs at run time and does incur a performance penalty due to pointer indirection for method calling. A common example of Dynamic polymorphism is the act of instantiating a child class as a parent class, however the performance losses occur when using virtual methods. A virtual method is a method that must be overwritten in a child class. Handling virtual methods is specific to the programming language, however it is commonly handled through the use of virtual method tables. A virtual method tables is a table the compiler adds to any class that contains virtual methods. The table contains the address of all virtual methods for that class. Figure 2 shows the difference between the indirection required for a virtual method and a non-virtual method. For a virtual method, the code must first follow the pointer to the virtual method table (i), then it must index the table (ii), finally it must follow the pointer to the actual method (iii). For a non virtual method, the code must only follow the pointer to the actual method (i). As it can be seen, this requires two additional points of indirection that causes virtual methods to be slower than their non-virtual counterparts.

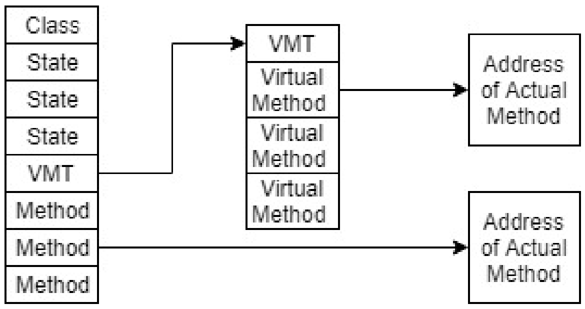

Figure 2: OOP Virtual Method Calling

Polymorphism is extremely important in the world of modeling and simulation as is what allows the developer to handle different levels of fidelity when performing calculations. As an example, consider a simulation that is modeling the Radio Frequency (RF) signatures of different objects for a radar system. Without polymorphism, it would be extremely difficult to represent multiple types of objects at different levels of accuracy, as multiple levels of fidelity present the need to perform different calculations depending on the object. An example of how this is handled can be seen in section 2.3.1.

2.3.1 C++

C++ is a system-level programming language released in 1985 and is one of the most common programming languages used today. C++ was created to add the OOP paradigm to C and is commonly used in projects ranging from embedded systems to graphical user interfaces. C++ is a compiled language which allows for very fast code execution compared to interpreted languages such as Python. Due to its age, there is a lot of infrastructure built around C++ and a lot of support can be found when programming C++.

Unfortunately, C++ has many issues that developers must overcome when programming in the language. C++ is unrestricted in what it allows you to do; this can result in unexpected behavior during execution. A prime example is that C++ does not enforce data safety when programming which can lead to many hard-to-diagnose issues when developing multi-threaded applications. It is up to the developer to determine where data safety mechanisms and memory management are required. Due to the complexities of ensuring safety in C++, it presents a barrier of entry that requires an enormous amount of experience to overcome, and even then, mistakes are still made.

As stated previously, C++ was developed around the idea of OOP, though the system for it is basic. It also allows for developers to develop and customize how they use the paradigm in their code-base, but this also presents issues for new developers with little experience. The main usage of OOP in C++ is inheritance and polymorphism allowing the developer to create large and complex inheritance hierarchies for advanced polymorphism. As stated in section 2.3, polymorphism is the ability to treat an object as if it was another object. An example on how this is done can be seen in Listing II.1. The child class inherits all state and methods from the base class by specifying it in the declaration of the child class. Polymorphism occurs when the child class is created, but it has an interface of the base class. The two children classes are inherently different, and may contain different state and methods from each other, but due to polymorphism, they are able to both be treated as the same type of object.

```
class base {/* No data yet */}
class child1: public base {/* No Data yet */}
class child2: public base {/* No Data yet */}

int main()
{
base *obj1 = new child1;
base *obj2 = new child2;
}
```

Listing II.1: Instantiating Child as Parent

When viewing code in C++, there is little difference between static and dynamic polymorphism. However, what happens behind the scenes, and the results, are considerably different. Listing II.2 shows static polymorphism by overloading the method print in the child class; it can be seen that the base object calls its method and the child object calls its method. When a child object is instantiated with the interface of the base object, it calls the base object's method as that is what the compiler sees it as.

```
class base {
public:
    void print() { cout << "base" << endl; }
}

class child: public base {
public:
    void print() { cout << "child" << endl; }
}

int main()
{
    base *b = new base;
    b->print(); //Prints "base" as expected

    child *c = new child;
    c->print(); //Prints "child" as the function was
       overloaded (static)

    base *bc = new child;
    bc-> print(); //Prints "base" as the compiler sees
       object as a Base and calls class method (static).
}
```

Listing II.2: Static Polymorphism

Per Listing II.3, dynamic polymorphism is occurring because the print() method now has the keyword virtual. Static polymorphism is still occurring in object b and c as they are instantiated as themselves. However, when a child class is instantiated as a parent class (object 'bc' in the example), the virtual keyword causes the code-base access the virtual method table to access child classes method rather than call the parent classes method. You can tell this is drastically different as Listing II.2, which uses static polymorphism as it produces the opposite outcome. It is important to note that this dynamic polymorphism is incurring a performance penalty as the virtual method table adds two additional points of indirection.

```
class base {
public:
    virtual void print() {cout << "child" << endl; }
}
class child: public base {
public:
    void print() { cout << "child" << endl; }
}

Int main()
{
    Base *b = new Base;
    b->print(); //Prints "base" as expected

    Child *c = new Child;
    c->print(); //Prints "child" as the function was
       overloaded (static)
    Base *bc = new Child;
    bc->print(); //Prints "child" as the compiler sees
       base object is virtual method, and accesses it via
       the VMT (dynamic)
}
```

Listing II.3: Dynamic Polymorphism

Consider the simulation found in Listing II.4 with two objects, one is a perfect sphere while the other is an aircraft. The sphere will always have the same RF signature but the aircraft has a different level of fidelity and perform more complex calculations to determine it's RF signature. With dynamic polymorphism, the solution is simple. Create a virtual method that is overloaded in the children classes. When the method is called on each object, it will call the child class function and properly calculate its RF signature rather than incorrectly calling the base classes method.

```
class radarObject
{
    //By setting this to zero, it enforces that the method
    //must be overrode in the child class, effectively
    //becoming an interface.
    virtual double calcRFSig() = 0;
}
class sphere: public radarObject
{
    double RFSig = 5;
    double calcRFSig()
    {
        return RFSig;
    }
}
class aircraft: public radarObject
{
    double state1 = 1;
    double state2 = 2;
    double state3 = 3;
    double calcRFSig()
    {
        //Complex calculation of internal state
        return state1*state2*state3;
    }
}
```

```
int main()
{
    radarObject *obj[2] = {new sphere, new aircraft};

    for(int i = 0; i < 2; i++) {
        std::cout << obj[i]->calcRFSig() << std::endl;
    }

    /* Output:
    5
    6
    */
}
```

Listing II.4: Handling Fidelity

2.4 Data-Oriented Design

DOD is a paradigm that focuses on efficient data storage and usage of the cache [16]. It does this by moving state out of classes and into separate data structures (commonly arrays) for each piece of state. The state data structure is indexed to access the data of objects. To be a pure DOD implementation, it must be guaranteed that each index in the data structure will contain data; this increases performance as it is no longer necessary to waste computational time verifying there is no null data. This causes the memory structure to change from object instances directly next to each other in memory (if in an array) to all instances of a piece of state next to each other in memory. Table 2 below represents the same two objects stored in memory using the DOD paradigm rather than the OOP paradigm found in Table 1. Each instance of an object is an index into each array of data. In this example, there exists three arrays (state1, state2, and state3) of length two. This programming pattern has two main benefits of use: (i) cache optimization using spatial locality and (ii) simplicity of parallelizing the code base.

Memory Address (Hexadecimal)					
0x00	0x04	0x08	0x0C	0x10	0x14
state1	state1	state2	state2	state3	state3
Object1	Object2	Object1	Object2	Object1	Object2

Table 2: Memory Layout of DOD Table

It is important to note that the greatest performance increases are only seen in code-bases that have a large amount of objects being operated on in a row. When this happens, the array length becomes larger, allowing it to better take advantage of spatial locality. The benefits of this paradigm is reliant on preventing cache misses when operating on large amounts of data. This design allows for more of the necessary data to be in the lowest level cache at once (and less, if not none, of the unnecessary data to be in cache). In OOP, entire objects are moved into the lowest level cache at once, even if the program is only operating on a few pieces of state inside of that object. This causes a lot of space in the cache to be filled by unused state inside of the object. Due to the structure that usage of the DOD paradigm produces, only the necessary state is moved into the CPU's cache. This ensures that more operations can be completed in a row before needing to bring additional data into the cache (and causing less cache misses).

Consider an OOP class that has three pieces of state and a function that only operates on two of the pieces of state at any given time. If that function is called on all instantiated objects (such as an update function in a real-time simulator), the cache line would look similar to OOP portion of Table 3. It can be seen that even though the function only works on two pieces of state, all three pieces of state are brought into the cache line. Due to this, one-third of the cache line is wasted. In the table, only four objects can fit in the cache line. Consider a DOD implementation of this where instead of a class, three arrays are initialized, each holding a particular piece of state. When an object is instantiated, its three pieces of state are added into the three arrays. When the same function is called across all objects, only the two necessary arrays are brought into the lowest level cache. This causes the cache line to look like the DOD portion of Table 3. It can be seen that there is no longer any wasted space and the cache can now fit six objects worth of state into the cache before it is full. This greatly reduces the amount of cache misses that can occur while a program is executing.

OOP Cache Line vs. DOD Cache Line												
OOP Cache Line	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3
DOD Cache Line	S1	S1	S1	S1	S1	S1	S2	S2	S2	S2	S2	S2

Table 3: OOP Cache Line vs. DOD Cache Line

This paradigm makes it easier to parallelize a code base. Since data is no longer wrapped in a class, it is easier to move only the data you need into different threads to be worked upon. As data is now stored in arrays of the same type, those specific arrays may be pushed into threads and operated on. For this type of execution, the programmer does not have to worry about any data races as the same data is never in multiple threads at once. An even more advanced system can take into account the access type (read and write) for each piece of data that a function needs. If two functions need only read access to a specific piece of data, that data could be copied into both threads without needing to worry about data races. This will be further expanded on in section 2.5.

The largest downside to this pattern is that data can no longer be hidden/ protected as can be done in the OOP paradigm. Since the data is no longer wrapped in a class that can limit how a piece of state is manipulated, there is less control on how a user interacts with the lowest level data in a program. Another issue is dealing with multiple levels of fidelity due to the fact that all data of the same type must be the same size and type. When dealing with multiple levels of fidelity, this tends not to be true as the amount of data needed to represent something can differ depending on how accurately something must be depicted.

2.4.1 Rust

The Rust programming language is a new programming language, officially released in 2015. The language is a system level, compiled language similar to C++. Rust's focus is on memory management, data safety, and concurrency. The stand out feature is that Rust's compiler checks that the developer is properly handling data-safety, error-handling, and concurrency. This checking is done at compile time, so there is no run-time performance loss for this additional safety. A major distinction of Rust is its use of lifetimes. Any piece of data instantiated in rust has an associated lifetime. Unlike other languages such as C++, if a piece of data is accessed after its lifetime, the compiler will see this and force the developer to fix the issue before the code will compile. Consider the Rust code in listing II.5. If this were programmed in C++, the code would compile, but it would be incorrect because 'x' would no longer exist when 'y' is set to 'x'. If you tried to use 'y' in anyway, you would get an error. However, Rust understands that the lifetime of 'x' ends at the bracket and it is cleaned up. If this code is compiled, the compilation would fail due to trying to access this data after its lifetime. Additionally, garbage collection is completely handled by Resource Acquisition is Initialization (RAII) due to the use of the aforementioned lifetimes.

```
fn main() {
    {
        let x: i32 = 5; //Start of lifetime
    } //end of lifetime
    let y: i32 = x; //error: Access after lifetime ended
}
```

Listing II.5: Lifetimes

Rust also uses an idea of ownership to prevent data issues. Ownership states that only one thing can "own" a piece of data. You can transfer ownership and you can borrow ownership, but their cannot be two owners without the use of smart pointers that control access. Listing listing II.5 shows how ownership occurs, similar to C++'s *unique_ptr* <> ().

```
fn main() {
    let x: i32 = 5;
    let y: i32 = x; //This is ok, it copies 5. x and y own
        two different versions of 5

    //Strings initalize on the heap, therefore it is not
       copied
    let str1: String = String::from("Hello");
    let str2: String = str1; //This is a change of
       ownership from str1 to str2.

    println!("{}", str1); //This errors as str1 is now
       invalid. It does not own "Hello" anymore
    let str3: String = String::from("Hello");
    {
        let str4: &String = &str3; //This borrows the
           value for the duration of str4's lifetime
        println!("{}", str4);
    } //str4's lifetime ends here
    println!("{}", str3); //str3 regained ownership so
       this does not error
}
```

Listing II.6: Ownership

The idea of ownership also comes into play for threads. Rust uses traits called send and sync to control data across threads. Send is a trait given to all data types that can be safely transferred across threads. This ensures the user does not attempt to transfer ownership of something that it cannot, such as a reference counted pointer. The sync trait is given to all data types that are safe to borrow ownership across threads. This once again ensures that the user does not attempt to borrow ownership of something that it cannot (anything that could cause data races or memory issues). Although this does not seem like many features for multi-threaded applications, these features, plus smart pointers, are able to ensure that data cannot be accessed unsafely from multiple threads.

Rust does have the concept of OOP, although it is handled differently than in C++. Encapsulation and abstraction is handled similar to C++. Rust uses the keyword *struct* to encapsulate data similar to C. However, Rust uses an additional keyword *impl* to specify implementing a method for a *struct*. This is different than in C++ where methods are implemented inside of the class. Rust implemented this design to better separate data from functionality as Rust focuses on "Composition over Inheritance", a main focus of DOD.The state inside of a *struct* is defaulted to private; any public state must use the keyword *pub*. This is opposite to C/C++ where data in a *struct* is defaulted to public, however the same as the C/C++ class where data is defaulted to private.

Another key difference is that Rust does not use inheritance as a way of polymorphism, it uses traits. A trait is similar to an interface in C++ (a class that contains a virtual method equal to zero). A trait defines a list of methods that must be implemented for a struct. If the struct intends on implementing a trait, it must define all of the methods in the trait. Once that is done, the struct can be used by anything that requires the trait. One weakness that Rust traits have is that traits cannot currently contain fields of data, which is why abstraction is not fully possible. listing II.7 shows an example of implementing a trait for two separate structs and using polymorphism to call the same function on two different structs, along with being able to have two different structs in the same array.

```
trait Pet {
    //This function must be overridden in any struct
       implementing the trait
    fn speak(&self)->String;
}


struct Dog {}
impl Pet for Dog {
    fn speak(&self)->String {
        return String::from("Woof!")
    }
```

```
}

struct Cat {}
impl Pet for Cat {
    fn speak(&self)->String {
        return String::from("Meow!")
    }
}

//This method accepts any object that has implemented the
   Pet class
fn makeSpeak<T: Pet>(pet: T) {
    println!(pet.speak());
}

fn main() {

let dog: Dog = Dog{};
let cat: Cat = Cat{};

//Dog and Cat structs both implement the Pet class, so
   they can be used in the function.
makeSpeak(dog);
makeSpeak(cat);

//Result:
```

```
//Woof!
//Meow!
}
```

Listing II.7: OOP in Rust

A final feature that Rust enforces is error handling. Many of the core features, and standard library features enforce the use of error handling by wrapping results from methods in an (Ok(T), Err(E)) enumerator. During the development phase, developers may simply “unwrap” this enumerator and access the result directly, but this could cause an unrecoverable error that will crash the program if an error occurred. However, if used properly, this Enumerator forces the developer to write error handling code to be able to retrieve the result of a method. Developers may also return results of methods this way in their own code to ensure they continue to use proper error handling. Listing listing II.8 provides examples of handling errors with unwrap() during developing, and how to properly handle errors with a finished product.

```
fn main() {
//If Test.txt is not found, this will cause the program to
    crash as the developer choice to unwrap() the result
   instead of error handling should only be used during
   development & testing, not in final product
let file: File = File::open("Test.txt").unwrap();

//The proper way to handle the possibility of getting an
   error
let file = match File::open("Test.txt") {
    Ok(f) => return f,
    Err(error) => {/*code for how to handle the error*/}
    }
}
```

Listing II.8: Rust Error Handling

2.5 Entity-Component-System

The ECS architecture is a software architecture commonly found in game development. It is currently being implemented in the well-known Unity game engine and has been actively developed and implemented by various game developers. This architecture focuses on the DOD paradigm, allowing for cache optimizations to reduce cache misses [17], automated parallelization [18], and a greatly decoupled code-base [17, 19]. These optimizations not only increase the computational performance (allowing for more work to be done per computational cycle), but also make implementing and maintaining the code-base simple and quick compared to other architectures. This is due to how the ECS architecture compartmentalizes Systems from the data, and Systems from themselves. The ECS can be broken up into three main parts (i) entity, (ii) component, and (iii) system along with a final important aspect, the execution pattern. The execution pattern is not a part of the ECS architecture, but is important to consider with this architecture. As this architecture is most prevalent in game engines, the below information will be discussed in reference to games and game engines.

The ECS architecture focuses on data-oriented programming rather than object-oriented programming. Instead of objects inheriting state and functionality from a parent object (object-oriented programming), the object will be composed of data

(data-oriented programming) [20]. This is considered the “Component” in the ECS architecture while the object is now the “Entity”. Finally, the methods for operating on state becomes the “System”. Each will be explained in-depth below.

2.5.1 Entity

The entity is any game-object spawned in the world. The world is merely the container of all entities. In the context of ECS, the world is a storage device that holds all state of entities as explained below. A few examples of entities would be the player, an enemy, or a bullet being fired from a weapon. An entity is analogous to an instanced class in OOP with an important exception: the entities do not contain data. An entity is only an index into a table containing the state (components) of all entities. As explained in Section 2.4, this table is structured in a manner that promotes cache optimization [21, 22]. The index value is generally equal to when it was instantiated. If it was the first object instantiated, it would have an id, or index, of zero.

2.5.2 Component

Components are generally individual pieces of state normally wrapped inside of a *struct*. This “wrapping” allows for each type of state to have a unique id even if they are both the same underlying data type such as a integer or double. As an example, a velocity and acceleration state could both be doubles, but they need to have different ids, so they are wrapped in a “velocity” and “acceleration” *struct* which provides unique ids for each type of state. This wrapping also allows for multiple pieces of state to be bundled as a single component, such as having a single position component rather than individual components for x, y, and z Cartesian coordinates.

Any state that would normally be inside of a class, is removed from the class and placed into individual components. Multiple pieces of state commonly used together (such as the aforementioned Cartesian coordinates) may continue to be wrapped together, though it is up to the developer [23]. This is a trade-off as some performance is lost as the *struct* must indexed before accessing the individual (x, y, z) values. As an example, if only one of the Cartesian Coordinates (x, y, or z) values are being accessed, then the other two coordinates are also being brought into the cache (as all three are wrapped in a *struct*) and wasting space.

As mentioned before, components are moved into arrays containing all instances of that component [21, 20]. A particular entity's location in that list is always equivalent to its unique identifier. It is important to note that the length of each array is always equal to the amount of entities, regardless of if every entity has that component or not. This is a drawback of the ECS architecture compared to a pure DOD paradigm as DOD only has a place in the array if the data exists. This design choice is necessary so that an entity's location in the list is the same across all components, thus simple to find a particular entity's component. This does hurt performance as null checks must be made before accessing the data. However with the advanced branch prediction that exists in most CPUs, this is mitigated [1].

2.5.3 System

Now that component storage is handled for each entity, the manipulation of these components must be considered. This is done through systems. There are two parts to this: (i) entities interacting with other entities and (ii) entities interacting with themselves. Interactions between entities tend to be complex due to the compartmentalized nature of the architecture. No entity knows about another entity, no component knows about another component, and no system knows about another system. This compartmentalization is a large benefit when it comes to the maintainability of the code-base as it allows for entities and systems to be added or removed without affecting anything else in the code-base. However, this does mean that a complex messaging system must be created to allow for communication between entities.

A system is a method that is automatically run on all entities that have the required components. Systems are defined by specifying the components that are needed as the inputs (anything being read or written), then defining the method that operates on the components [23]. It is important to remember that the system only has access to a single instance of an entity at any given time (e.g. entity zero does not know anything about entity one by default). When a system begins execution, it is provided an array for each of the requested components. This array is the same size as the amount of entities that exist in the world. The system will attempt to execute over each index in the array. If an index in one of the arrays is null (i.e. an entity doesn't have a required component), then it will move on to the next index. If all components exist, the system will execute on the components. This process continues until all entities with the necessary components have been operated on. As an example, if a system needs access to a "Position" and "Velocity" component, the iterator will provide, one entity at a time, all entities that contain the two components. The system is able to operate on all entities efficiently during run-time because only the necessary components are being brought into the cache (as seen in Table 3) resulting in fewer cache misses and more time operating on the components.

2.5.4 Execution Pattern

The execution pattern, i.e. system loop, is not part of the ECS architecture, but it important to consider for a real-time simulation. The system loop is responsible for responding to user input, executing all systems, and rendering the screen. However, since most modern-day military simulators are not responsible for rendering the screen, that can be replaced with sending User Datagram Protocol (UDP) packets to the secondary system responsible for rendering.

A common misconception is that the system loop is simply a while loop that waits for a boolean termination signal. In very simple implementations, this may be true, however in most cases, it is not. When using the loop stated above, the amount of time that passes per iteration is non-deterministic as it is dependent on how much processing is being done that loop. If any of the systems are dependent on time passing at a steady pace (e.g. sixty cycles/ frames a second), then time would speed up and slow down depending on how much processing was being done [24]. [A real-life example of this is Space Invaders: The enemies speeding up as more are eliminated is a byproduct of having to perform less calculations, and render less entities, on the screen. It was not intentional.]

The requirement of timing needing to pass at a steady pace makes the system loop more complicated. There are many ways to implement a system loop, and it is a decision that must be made early as it affects how the system logic is written. Common system loops focus on setting a max frame rate that should be obtainable on most systems, then each time step is an assumed value (1 second divided by the number of frames). A disadvantage to this is that if the frame rate is set too high, or the hardware is old, users may find that the in-simulation time slows down (i.e. the simulation seems to be in slow motion) as the CPU cannot process the amount of data fast enough such that the simulation appears to run in real-time [24]. Another implementation is that the system loop will actually calculate how much time has passed since the last frame and feed that time to the update functions (or the systems for an ECS architecture) [17]. The issue with this implementation is that it complicates programming time-dependent systems as code must be robust enough to calculate its result based on a certain time, instead of being able to assume that the next frame is a specific increase from the last frame.

As an example, implementing functionality such as “pressing the left arrow moves the character left” would be as simple as “If the left arrow is pressed, move left by one unit” in the first system loop implementation. This is because you know time will move the same every frame. If the system loop is implemented the second way, it becomes more complicated to create the functionality. This is because a new component that stores the previous loops current time must be created. Then in the system, that time is subtracted from the actual current time, then multiplied by some velocity.

There are many other implementations of system loops out there, each with its own pros and cons. However, one thing that should almost always be done on a separate thread is the rendering (or in this case, sending data packets to the renderer) as these actions are extremely slow[25]. Delegating it to a different thread will free up more computational time that can be devoted to user input and system execution.

2.6 Specs

Specs is an implementation of the ECS architecture continuously being developed for the programming language Rust. Specs main use is for game development, though the Application programming interface (API) has been developed such that it is an option for other use-cases (such as real-time simulations). Specs closely follows the architecture explained in section 2.5, but goes beyond what traditional ECS architectures by providing additional storage devices with different use-cases, simple -and safe- parallelism, and a with a high degree of flexibility.

Rather than only an array for storing components, Specs provides five different storage components each optimized for different use as seen in Table 4. Two of the most important storage types are “DenseVecStorage” and “HashmapStorage”. “DenseVecStorage" exists for components that exist in many entities where as "Hashmap-Storage" is for components that will exist in few entities. "HashmapStorage" is not inherently great for locality, but it provides quick access to components that are rarely used. Since not all Storage types are vectors now, components are provided to systems via an iterator. This iterator contains all requested state for an individual entity. The system then executes the method on the components provided by the iterator. The iterator automatically iterates to the next entity that has all necessary components. This process continues until all entities with the necessary components have been operated on. This iterating process can be slow if the system is requesting many components, so it is best to keep the amount of components needed in a system low.

Storage Type	**Description**	**Optimized for**
BTreeStorage	Works with a BTreeMap	no particular case
DenseVecStorage	Uses a redirection table	fairly often use components
HashMapStorage	Uses a HashMap	rare components
NullStorage	Can flag entities	doesn't depend on rarity
VecStorage	Uses a sparse Vec	commonly used components

Table 4: Different Storage Types for Specs

Additionally, Specs includes a special data type called resources which is used in systems. A resource is similar to a component, but instead of being specific to a certain entity, it is common to all entities [23]. There are many use cases for using a resources, one of which is to create a messaging system for handling events and communication between entities, though it is not necessary to be used for such. Resources are also highly useful for use in extremely large pieces of data so that they don't have to be passed around in memory. They can even be shared amongst multiple entities. This is common with much of the necessary data for graphics.

As stated before, Specs provides simple and safe parallelism. On top of defining what components will be used for a system, Specs also requires that the developer specify how each component will be used. This is defined by specifying either read or write access for each component (or resource). Specs then uses a scheduling algorithm to look at each system in the simulation and is able to determine what systems can be run in parallel based off of the read/write access of components for each system [18, 26]. Systems are then dispatched to a thread pool based off of the schedule created by the scheduling algorithm [23]. Specs also allows the developer to define any dependencies between systems. If it is vital that one system executes before another, it can be specified by the developer and the algorithm will adapt the schedule accordingly.

During Execution, when a system is being run, a query is made to the underlying storage for the requested data. In Specs, the underlying storage is a hashmap containing a storage type (specified in Table 4) for each component. The query returns the requested data in the form of an iterator. The querying process is costly in terms of performance due to the need of filtering out any entities that don't have the required pieces of state. Due to this, it is suggested that systems should contain the least amount of components possible. This is because the iterator checks each entity for the required components. If the system is requesting 100 components, it must verify all 100 components exist for the specific entity before providing the entity via the iterator. If the system contains that many components, it can probably be refactored into multiple smaller systems.

Consider the three systems defined in Table 5. It can be seen that system two and three both rely on reading component two, but write to different components. This means that they can be run in parallel. However both system two and three conflict with system one as system one is writing to component 2. The scheduling algorithm would schedule system one to execute on thread one; directly after finishing, system two and three would execute in parallel on two separate threads as there are no data races between the two.

	Component1	**Component2**	**Component3**	**Component4**
System1	READ	WRITE	WRITE	NULL
System2	NULL	READ	WRITE	NULL
System3	NULL	READ	NULL	WRITE

Table 5: System Scheduling Example

A final important benefit that Specs has in the area of real-time simulations is its high degree of flexibility with some OOP properties. An important difference between games and real-time simulators are multiple levels of fidelity as explained in section 2.3 and section 2.4. Specs allows for polymorphic *structs* to be used as components. Since these components may be of different sizes in memory depending on how they were instantiated, the component list actually holds references to the individual components instead of holding the actual components as normally done. Any system that uses this component is going to be inherently slower due to the indirection in accessing the *structs*, but is a necessary trade-off to be able to handle these multiple levels of fidelity. It is important to note that this feature should be used sparingly as it will degrade performance.

III. Methodology

3.1 System Under Study

The system under study for this experimentation is the software paradigm and architecture used to organize and execute a hypothetical real-time simulator. The experimentation considered the two software paradigms: (i) Object-Oriented Programming (OOP) and (ii) Data-Oriented Design (DOD). Two separate software architectures were developed, each using one of the two different paradigms. The DOD paradigm was developed using the Entity-Component-System (ECS) architecture while the OOP architecture consisted of a simple architecture focusing on the concepts of OOP. These two paradigms will be referred to as OOP / DOD architectures from here on out.

3.2 Variables

This experimentation focused on the performance impact that different variables had on the two separate architectures. This was done by measuring the only response variable, *completion_time*, of each architecture while changing a variables found in Table 6. *Completion_time* is measured as the time it takes to execute all systems on all entities once the world has been set up. It does not include the time it takes to set up the world.

The variables important to the experimentation can be seen in Table 6. For each experiment, each variable is in one of three categories. The first category, response variable, is reserved only for *completion_time*. This is the output variable that changes, and is measured, as the independent variable is iterated over. *Completion_time* values are used for determining the performance of each architecture.

The independent variable is the next category and is reserved for the variable(s) that the experiment is measuring the performance impact on. This variable has a set list of values that is iterated over during the experiment. At each iteration, the performance of each architecture is measured and used for data analysis in Chapter IV.

The final category are the factors that are held constant for the duration of each experiment. Any variable not in either of the first two categories are placed in this category to ensure they do not affect the experiment in any manner. Each experiment section below will provide a table of all variables and what category they are in.

Variable	Entity Count	Entity Size	Component Count	Thread Count	System Count
Unit	Entities	Bytes	Components	Threads	Systems

Table 6: Variables used in all experiments

3.3 Assumptions

As stated in Chapter I, there were three main assumptions: (i) the methods that operate on the data execute exactly the same for both architectures, (ii) the compiler optimization level is set to as low as possible, and (iii) both architectures and all experiments are ran on the same hardware.

The first assumption was that the methods that operate on the state of entities in the experimentation were designed to do the same operation for both architectures. Additionally, these operations were resource-intensive, but overall meaningless. The focus was for the methods to pull the necessary state into the cache and do enough calculations to prevent the compiler from optimizing the memory requests away. The method for these experiments merely request a specific amount of state from the entity and sums the requested state together. It then stores the value in a separate piece of state from the entity. The method: $comp_0 += comp_1 + ... + comp_N$

In an attempt to further enforce that memory requests were not optimized away, the next assumption is that the compiler optimization of the code was forcefully lowered to ensure methods were not optimized away. This also assumes that by lowering the optimization level, the compiler will not change the underlying structure of the code or how either the DOD or OOP architectures are executed.

The final assumption is that both architectures and all experiments run on the same hardware. This hardware is a 2018 Macbook Pro with a 2.3Ghz Quad-Core Intel i5 processor (MR9Q2LL/A). All results are based off of this hardware. Numerical results will vary on different hardware due to differences in the size and levels of cache in the Central Processing Unit (CPU) and Random Access Memory (RAM). Results based on *thread_count* may vary more greatly as a CPU with 8 cores and 16 threads will perform different than the aforementioned Macbook Pro with 4 cores and 8 threads.

3.4 Statistical Analysis

Each of the six experiments were designed to test whether a specific variable impacts the performance of the OOP or DOD architecture to a greater extent by measuring the completion time of each. A null and alternative hypothesis was made for each experiment with the goal of failing to reject the null hypothesis. A p-value was calculated using a Student's t-test, which is used to affirm or reject the alternative hypothesis. Affirming or rejecting the alternative hypothesis gives the ability to reject the null hypothesis (if the alternative was affirmed), or fail to reject the null hypothesis (if the alternative was rejected). Failing to reject the null hypothesis does not mean it is affirmed, only that there is strong statistical evidence that the null hypothesis may be true.

The paired Student's t-test was selected for use in this experimentation as it's purpose is to determine the probability that two sample sets are the same with respect to a single variable tested, which closely matches the goal of each experiment. The goal of each experiment was to determine if OOP and DOD (the two sample sets) completion times are affected similarly as you change a specific variable. To comply with the criteria of the Student's t-test, while still answering the hypothesis, the specific single variable used in the Student's t-test is the paired *completion_time* measurements between the two architectures. It was necessary to use the paired Student's t-test as there is a direct relationship between each specific data point between the two architectures (i.e. removing one of the points in the OOP measurement, leaves a dangling point in the corresponding DOD measurement).

The general steps to calculating a paired Student's t-test is as follows:

1. Calculate the difference ($d_i = y_i - x_i$) for each pair
2. Calculate the mean difference ($\bar{d}$)
3. Calculate the standard error ($SE(\bar{d}) = s_d/\sqrt{n}$) where s_d is standard deviation and n is the number of pairs
4. Calculate the t-statistic ($T = \bar{d}/SE(\bar{d})$)
5. Index the t-distribution table to get p-value

The p-value returned from the Student's t-test is a measurement of probability that the two data-sources are the same (i.e. if the p-value is .4, there is a 40% probability that the two data sources are similar). Typically a p-value less than or equal to 0.05 is considered statistically strong evidence. The experiments and hypotheses have been set up such that a p-value less than or equal to 0.05 indicates strong evidence to reject the null hypothesis while a p-value greater than 0.05 indicates weak evidence and fail to reject the null hypothesis.

3.5 Experimental Design

A detailed explanation of each experiment can be found in the below subsections, however, the purpose of each experiment can be seen in the list below:

To test the performance impact of:

- **Experiment 1:** Entity Size
- **Experiment 2:** Entity Count
- **Experiment 3:** Total Memory Size
- **Experiment 4:** Thread and System Count
- **Experiment 5:** Thread Count
- **Experiment 6:** Component Count

Each experiment consisted of multiple tests: each test executed the same code base, but with a different value for the independent variable as can be seen in tables provided in the subsection of each experiment. For each test, there was a warm-up period of 5 seconds to flush the cache and/or load the cache. This allowed the test to measure the performance during normal execution instead of measuring the start-up performance. It was arbitrarily selected that 100 samples would be taken for each data point. This decision was made to ensure each test to provide enough data to perform a reliable Student's t-test. Each sample measured the completion time (μs) to execute all systems on all entities. From there, outliers were removed and the remaining samples were averaged to produce a single data point used in each graph. These points were used to determine the change in completion time and calculate the p-value using the Student's t-test as explained in Section 3.4. The averaged data points can be found in the results of each experiment while the code for each experiment can be found in Appendix B through Appendix G.

Each Experiment was split into two distinct parts: (i) OOP architecture benchmark and (ii) DOD architecture benchmark. An important design choice for each experiment was to ensure the two benchmarks did not run in parallel. This was to ensure that one did not hog CPU resources and/or that there was no interference between the two. It was arbitrarily selected that the OOP architecture was run first for each experiment.

There are three phases in each experiment that occurs during the benchmark process as can be seen in Figure 3 and Figure 4. The first phase, which is similar in both architectures, is to configure the benchmark. This step sets important variables for benchmark execution and analysis. The variables important to this experiment include: warm-up time, sample size, re-sample size, and noise threshold. Warm-up time was used to set how long the benchmark should run before taking measurements. This allows the cache to be flushed of any old / stale values from previous experiments and gives a more realistic measurement of performance. Sample size determines how many measurements to take before performing statistical analysis on the data and providing a result. Re-sample size is set to determine how many re-samples are taken for each sample. Finally, the noise threshold is set to determine what, if any, values are noise. If the noise threshold is set to X%, then any value outside of the X% average is considered noise.

The second phase, setting up the data for experiment execution, is different for each of the two architectures and will be explained in Section 3.5.1 and Section 3.5.2. The final phase (phase three), also similar in both architectures, is the benchmark, analysis, and results display phase. During this phase, a lambda with the method to be measured (in this case, the execute function that runs all systems on all entities) is passed to Criterion. Criterion then executes this lambda without measurement for the duration of the warm-up time, after which it begins taking samples based off sample size and re-sample size. Once all samples have been taken, Criterion then performs statistical analysis as described in Appendix A and provides the results to the user. Phase two and three execute for every value of the independent value. As an example, should the independent variable hold values of $[128, 256, 512, 1024]$, then phase two and three will execute four times, once for each value.

3.5.1 OOP Benchmark Design

Phase two for the OOP benchmark consists of two steps as seen in Figure 3. The first step is to instantiate the objects (entities) that will be present in the architecture. The amount of objects instantiated varies depending on the experiment. As this is the OOP benchmark, the objects were created with all of the state inside of the object, along with the methods that are to be ran on the objects. The amount of state and methods inside of each object vary depending on the experiment.

The second step is initializing the storage which holds the objects instantiated in step one. This storage is responsible for executing the methods on each object. The storage holds entities in multiple groups depending on how many threads are available to the architecture. As an example, if there are four threads available to the system, then there are four groups, which have the instantiated objects evenly distributed amongst them. This allows for simple parallel execution of all methods as long as there are no data races, which is ensured by design. This storage has a method called execute() which executes all methods on all objects in the storage. This is what is passed to the lambda described in phase three for benchmarking and analysis.

3.5.2 DOD Benchmark Design

Phase two for the DOD benchmark consists of five separate steps as seen in Figure 4. The first step is initializing the storage. This is done first as entities no longer

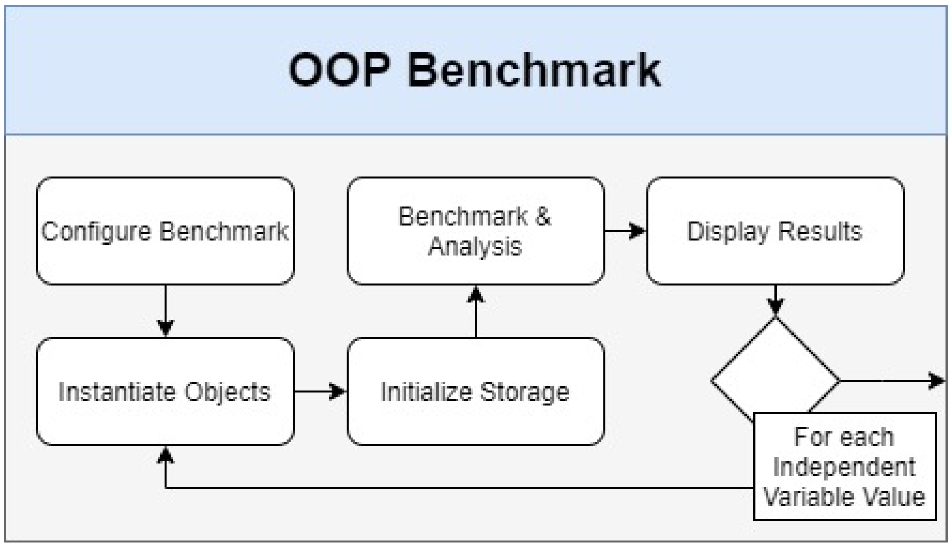

Figure 3: Flow Chart of The OOP Benchmark

hold state and methods. Therefore, to be able to instantiate entities, the storage for each piece of state must already exist. The storage is a hashmap of key-value pairs consisting of the *type_id* and vectors for each registered components. This leads to the second step, which is registering components. To be able to create an entity with components, the storage for the component must exist. This is done by registering the component with the storage. This causes the storage to create a key value pair with an empty vector for each component registered.

The next step is instantiating all entities with their required components. During this step, every entity that is instantiated increases the size of all component vectors by one, regardless of if the entity has the component. For any components that the entity does have, they are added to the respective component vectors at the entities index.

The final step is to initialize the dispatcher and register systems with the dispatcher. The dispatcher is responsible for executing any registered systems over the storage. Unlike the OOP design, the dispatcher is what executes systems (methods) in parallel rather than the storage itself. The dispatcher is initialized with no systems, however the final step is to register systems with the dispatcher. As described in Chapter II, a system consists of a method that will be executed across all entities that have a particular set of components. When systems are registered, the dispatcher is responsible for detecting any data conflicts based on the requested components, and schedule the systems in an optimal way such that no state is in conflict (i.e. no state is being written to simultaneously, or no state is being read while also being written).

Both the dispatcher and the storage are provided to the lambda described in phase three for benchmarking and analysis. The dispatcher has a method called execute which takes a reference to the storage. This method executes all systems over the data inside of the storage.

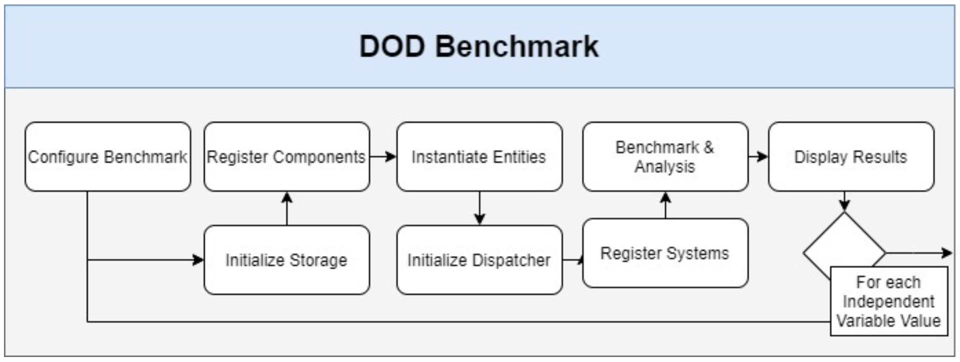

Figure 4: Flow Chart of The DOD Benchmark

3.5.3 Experiment One

The objective of this experiment was to compare the performance impact the size of entities has on the completion time of the OOP and DOD architectures. The following null and alternative hypotheses were made:

$H_0 : completion\_time(OOP) \leq completion\_time(DOD)$

$H_1 : completion\_time(OOP) > completion\_time(DOD)$

The two equations state that the null hypothesis is “completion time of the OOP architecture is less than or equal to that of the DOD architecture as entity size increases” i.e. the OOP architecture outperforms (or is equal to) the DOD architecture. The alternative hypothesis is “completion time of the OOP architecture is greater than that of the DOD architecture as entity size increases” i.e. the DOD architecture outperforms the OOP architecture.

This hypothesis was made due to the issues with cache performance described in section 2.2. For the OOP architecture, the entire entity is brought into the CPU’s cache regardless of what state is actually used where as for DOD, only the required state is brought into the CPU’s cache. Due to this, the cache should fill faster for the OOP architecture resulting in more cache misses compared to the DOD architecture.

Table 7 presents a list of all variables for this experiment. It should be noted that the independent variable, *entity_size*, iterates through the following values: [32, 64, 128, 256, 512, 1024, 2048] bytes. *Completion_time* was recorded as the average time it took to execute all systems on all entities once.

Response Variable		Factors held Constant			Independent Variable	
name	unit	name	count	unit	name	unit
completion_time	μs	*entity_count*	1000	entities	*entity_size*	bytes
		component_count	2	components		
		thread_count	1	threads		
		system_count	1	systems		

Table 7: Variables for Experiment One

3.5.4 Experiment Two

The objective of this experiment was to compare the performance impact the amount of entities has on the completion time of the OOP and DOD architectures. The following null and alternative Hypotheses were made:

$H_0 : completion\_time(OOP) \leq completion\_time(DOD)$

$H_1 : completion\_time(OOP) > completion\_time(DOD)$

The two equations state that the null hypothesis is "completion time of the OOP architecture is less than or equal to that of the DOD architecture as entity count increases" i.e. the OOP architecture outperforms (or is equal to) the DOD architecture. The alternative hypothesis is "completion time of the OOP architecture is greater than that of the DOD architecture as entity count increases" i.e. the DOD architecture outperforms the OOP architecture.

This hypothesis was made as it was theorized that as more entities are introduced into the simulation, more cache misses would occur due to the same issues presented in experiment 1. The cache misses would occur more frequently in the OOP architecture than the DOD architecture due to how the two architectures organize data in memory.

Table 8 presents a list of all variables for this experiment. It should be noted that the independent variable, *entity_count*, iterates through the following values: [10, 50, 100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 1000] entities. $Completion\_time$ was recorded as the average time it took to execute all systems on all entities once.

Response Variable		Factors held Constant			Independent Variable	
name	unit	name	count	unit	name	unit
completion_time	μs	*entity_size*	256	bytes	*entity_count*	entities
		component_count	2	components		
		thread_count	1	threads		
		system_count	1	systems		

Table 8: Variables for Experiment Two

3.5.5 Experiment Three

The objective of this experiment was to provide additional insight into the impact that *entity_size* and *entity_count* has on the OOP and DOD architectures by iterating both and viewing the effects based on the total amount of memory used in cache.

It was thought that both *entity_count* and *entity_size* would increase the amount of data brought into the CPU's cache for the OOP architecture. However, it was thought that only *entity_count* would increase the amount of data brought into the CPU's cache for the DOD architecture. This was expected to result in an increased cache miss rate once the total amount of data ($entity\_size * entity\_count$) exceeded the CPU's cache size. Since the OOP architecture would fill cache faster than the DOD architecture, it was theorized that the completion time would increase at different rates.

Table 9 presents a list of all variables for this experiment. It should be noted that the independent variable, *entity_size*, iterates through the following values: [32, 64, 128, 256, 512, 1024, 2048] bytes while *entity_count* iterates through the following values: [10, 50, 100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000] entities. *Completion_time* was recorded as the average time it took to execute all systems on all entities once.

Response Variable		Factors held Constant			Independent Variable	
name	unit	name	count	unit	name	unit
completion_time	μs	*thread_count*	1	threads	*entity_count*	entities
		system_count	1	systems	*entity_size*	bytes
		component_count	2	components		

Table 9: Variables for Experiment Three

3.5.6 Experiment Four

The objective of this experiment was to compare the performance impact the amount of threads and systems has on the completion time of the OOP and DOD architectures. The following null and alternative Hypotheses were made:

$H_0 : \mathit{completion\_time}(OOP) \leq \mathit{completion\_time}(DOD)$

$H_1 : \mathit{completion\_time}(OOP) > \mathit{completion\_time}(DOD)$

The two equations state that the null hypothesis is "completion time of the OOP architecture is less than or equal to that of the DOD architecture as thread and system count increases" i.e. the OOP architecture outperforms (or is equal to) the DOD architecture. The alternative hypothesis is "completion time of the OOP architecture is greater than that of the DOD architecture as thread and system count" i.e. the DOD architecture outperforms the OOP architecture. Opposite of all experiments before (as described in Section 3.5), the expected outcome was that the null hypothesis be affirmed.

This hypothesis was made due to the fact that the execution pattern for executing on multiple threads does not change between the OOP and DOD architectures. Both dispatch systems to threads to execute simultaneously. The only difference is the ease (for the user) in which it is to dispatch the systems to threads in the thread-pool.

Table 10 presents a list of all variables for this experiment. It should be noted that the independent variables, *system_count* and *thread_count*, iterate through the following values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] threads / systems. Both variables always equal each other. *Completion_time* was recorded as the average time it took to execute all systems on all entities once.

Response Variable		Factors held Constant			Independent Variable	
name	unit	name	count	unit	name	unit
completion_time	μs	*entity_count*	1000	entities	*thread_count*	threads
		entity_size	256	bytes	*system_count*	systems
		component_count	2	components		

Table 10: Variables for Experiment Four

3.5.7 Experiment Five

The objective of this experiment was to compare the performance impact the amount of threads has on the completion time of the OOP and DOD architectures. The following null and alternative Hypotheses were made:

$$H_0 : completion\_time(OOP) \leq completion\_time(DOD)$$
$$H_1 : completion\_time(OOP) > completion\_time(DOD)$$

The two equations state that the null hypothesis is "completion time of the OOP architecture is less than or equal to that of the DOD architecture as thread count increases" i.e. the OOP architecture outperforms (or is equal to) the DOD architecture. The alternative hypothesis is "completion time of the OOP architecture is greater than that of the DOD architecture as thread count increases" i.e. the DOD architecture outperforms the OOP architecture. Opposite of all experiments before Experiment four (as describe in Section 3.5), the expected outcome was that the null hypothesis be affirmed.

This hypothesis was made due to the fact that the execution pattern for executing on multiple threads does not change between the OOP and DOD architectures. Both dispatch systems to threads to execute simultaneously. The only difference is the ease (for the user) in which it is to dispatch the systems to threads in the thread-pool. It was expected however, that as you add available threads to the architectures, the completion time would decrease until the thread count surpasses the CPU thread count. This was expected as once it surpasses that point, no more benefit can be seen as anything over the thread count of the CPU is no longer truly running in parallel. You are only adding additional workload for managing the thread-pool.

Table 11 presents a list of all variables for this experiment. It should be noted that the independent variable, *thread_count*, iterate through the following values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] threads. *Completion_time* was recorded as the average time it took to execute all systems on all entities once.

Response Variable		Factors held Constant			Independent Variable	
name	unit	name	count	unit	name	unit
completion_time	μs	*entity_size*	256	bytes	*thread_count*	threads
		component_count	2	components		
		entity_count	1000	entities		
		system_count	15	systems		

Table 11: Variables for Experiment Five

3.5.8 Experiment Six

The objective of this experiment was to compare the performance impact the amount of components requested for a system being executed (i.e. a system needs four components to execute. Any entity that has all four requested components will have that system executed on it) has on the completion time of the OOP and DOD architectures. The following null and alternative Hypotheses:

$$H_0 : completion\_time(OOP) \leq completion\_time(DOD)$$

$$H_1 : completion\_time(OOP) > completion\_time(DOD)$$

The two equations state that the null hypothesis is "completion time of the OOP architecture is less than or equal to that of the DOD architecture as component count decreases" i.e. the OOP architecture outperforms (or is equal to) the DOD architecture. The alternative hypothesis is "completion time of the OOP architecture is greater than that of the DOD architecture as component count decreases" i.e. the DOD architecture outperforms the OOP architecture.

This hypothesis was made due to the fact that the DOD architecture has more granular control of what state is brought into the cache where as the OOP architecture brings the entire object into cache regardless of what state is used due to how state is organized in memory. As the method uses less state, less state will be brought into the cache for the DOD architecture resulting in less frequent cache misses and better performance (lower completion time).

Table 12 presents a list of all variables for this experiment. It should be noted that the independent variable, *component_count*, iterates through the following values: [16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2] components. *Completion_time* was recorded as the average time it took to execute all systems on all entities once.

Response Variable		Factors held Constant			Independent Variable	
name	unit	name	count	unit	name	unit
completion_time	μs	*entity_size*	256	bytes	*component_count*	components
		thread_count	1	threads		
		entity_count	1000	entities		
		system_count	1	systems		

Table 12: Variables for Experiment Six

3.6 Data Logging

Data logging was performed by the criterion crate (Rust's version of a user-developed library). Criterion is a statistics-driven micro-benching tool written in Rust. As a micro-bench, it is able to test small parts of a code base, i.e. single method calls. This is used to benchmark the method call to execute all systems on all entities.

It uses statistical analysis to determine the average time it takes to execute a piece of code over a predetermined amount of iterations. Additionally, it has the capability to detect any outliers and remove them from the results. Criterion provides the fastest, average, and slowest execution times for each benchmark, which will be used for providing results in chapter IV. An explanation of the process that Criterion uses can be found in Appendix A.

3.7 Summary

In summary, six separate experiments were run over five variables. The goal of these experiments was to determine the performance impact that different variations and combinations of the five variables had on the performance of the two software architectures. These experiments serve to provide an accurate answer as to which of the two paradigms provided better, more reliable performance in regards to real-time simulations by emulating the workflow of a real-time simulator. All data is calculated and recorded via the criterion crate, a statistics driven micro-benching tool which accurately finds the average run-time of a piece of code.

IV. Results and Analysis

4.1 Overview

This chapter describes the results obtained from the the experiments described in chapter III. Each section discusses a single experiment, first providing an overview of the experiment, then examining the performance of both Object-Oriented Programming (OOP) and Data-Oriented Design (DOD) architectures. The performance of the two software architectures are then compared to determine which is most tolerant to the independent variable(s). Finally, an explanation of the results is described to elaborate on the performance differences between the two software systems.

4.2 Experiment 1

The objective of this experiment was to determine if the size of entities has different effects on the performance of the OOP and DOD architectures. In this experiment, the independent variable, *entity_size*, was iterated through the following values: [32, 64, 128, 256, 512, 1024, 2048] bytes. Refer to Table 7 for all other variable values. The results for this experiment contain 100 data points per independent variable per software architecture. For Figure 5, of the 100 points, outliers were removed and the remaining were averaged to provide a single performance point. The averaged points are provided in Table 13 while the code used in this experiment can be found in Appendix B.

To determine if there is enough statistical evidence to show that entity size effects the performance of the DOD and OOP architectures differently, a paired Student's t-test comparing the completion time between the two architectures was performed. The data used for the Student's t-test can be found in the "Time" columns of Table 13. After performing the Student's t-test as described in Section 3.4, the resulting p-value was 0.015. A p-value of 0.015 is enough statistical evidence to reject the null hypothesis and fail to reject the alternative hypothesis. This means that there is enough evidence to suggest that entity count does effect the two architectures differently.

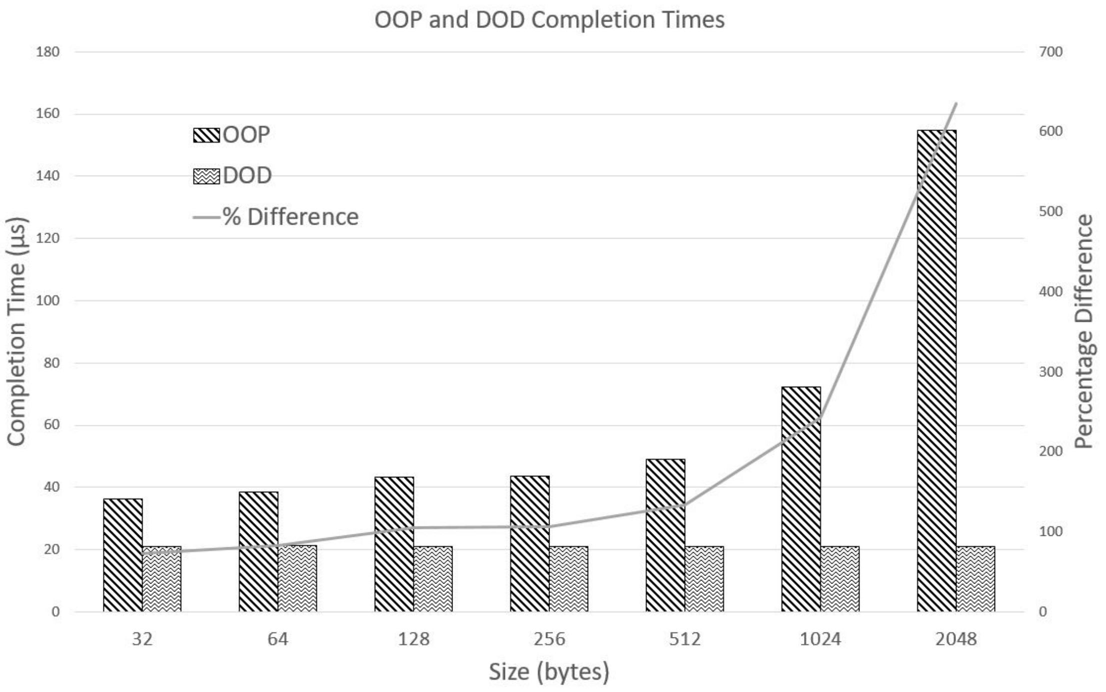

Figure 5: Graphical Results for Experiment One

Figure 5 shows the completion time (μs) of the OOP and DOD architectures on the left and the percent difference in completion time between the two architectures on the right. It can be seen that as *entity_size* increased, so did the completion time of the OOP architecture. Even though no additional work was being completed by the method, the completion time increased from $36.180\mu s$ to $154.77\mu s$, an increase of 327.78%.

The DOD architecture saw no noticeable increase in completion time while increasing the size of the entity. *completion_time* ranged from $21.017\mu s$ to $21.194\mu s$, an difference of 0.8%. This data not only supports the Student's t-test (on the interpolated data) in showing that the two architectures are effected differently, but that it effects the OOP architecture to a greater extent, greatly reducing performance compared to the DOD architecture.

The largest difference in performance between the two architectures was 635%, with a *completion_time* of 154.77μs for the OOP architecture and a *completion_time* of 27.49μs for the DOD architecture. It is important to note that the largest *completion_time*'s, at 8,192 and 16,384 bits, are uncharacteristically large for a real-time simulator and would not normally be seen in one. However, there is still a performance difference of over 100% between the two architectures at lower, more realistic, entity sizes.

From the data above, There is significant statistical evidence that *entity_size* has a role in determining the performance of the OOP architecture where as it does not for the DOD architecture. An expected reason for this result is that the OOP paradigm stores data per entity as shown in Table 1: all state in an entity is stored directly next to each other in memory. When the Central Processing Unit (CPU) requests a piece of state from the entity, the memory model loads the entire block of memory that the state is located in, into the CPU's cache. This generally means the entire entity, along with other data around it, is brought into the lowest level cache. If executing the same method over many entities, a lot of space in the CPU's cache is wasted on data that is never used by the processor. This results in many cache misses, causing the CPU to sit idle, or context switch, while it waits for the memory model to provide the required data.

The DOD paradigm stores data per state as shown in Table 2: all state of a specific kind are stored directly next to each other in memory. When the CPU requests a piece of state, the memory model will once again load the entire block of memory that the state is located in, into the CPU's cache. However, this means the particular state for all entities is now present in the CPU's cache. If executing the same method over many entities, no space is wasted as the state necessary for computation is already present for all entities. The CPU is able to execute the method over more entities before having a cache miss due to missing data and sitting idle while the memory model loads the required data. This properly explains why the performance of the OOP software system slows while the DOD software system remains nearly constant

	OOP		**DOD**		
size (bytes)	**% Change Between Measurements**	**Time (μs)**	**Time (μs)**	**% Change Between Measurements**	**% Difference**
32	N/A	36.392	21.096	N/A	72.51
64	106.24	38.664	21.194	100.46	82.43
128	111.84	43.242	21.052	99.33	105.41
256	100.58	43.494	21.086	100.16	106.27
512	112.72	49.027	21.017	99.67	133.27
1024	147.20	72.171	21.065	100.23	242.61
2048	214.45	154.77	21.053	99.94	635.14

Table 13: Numerical Results for Experiment One

4.3 Experiment 2

The objective of this experiment was to determine if the amount of entities has different effects on the performance of the OOP and DOD architectures. In this experiment, the independent variable, *entity_count*, was iterated through the following values: [10, 50, 100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000] entities. Refer to Table 8 for all other variable values. The results of this experiment contain 100 data points per independent variable per software architecture. For Figure 6, of the 100 points, outliers were removed and the remaining were averaged to provide a single performance point. The averaged points are provided in Table 14 while the code used in this experiment can be found in Appendix C.

To determine if there is enough statistical evidence to show that the entity count effects the performance of the DOD and OOP architectures differently, a paired Student's t-test comparing the completion time between the two architectures was performed. The data used for the Student's t-test can be found in Table 14. After performing the Student's t-test as described in Section 3.4, the resulting p-value was 0.00058. A p-value of 0.00058 is enough statistical evidence to reject the null hypothesis and fail to reject the alternative hypothesis. This means that there is enough evidence to suggest that entity count does effect the two architectures differently.

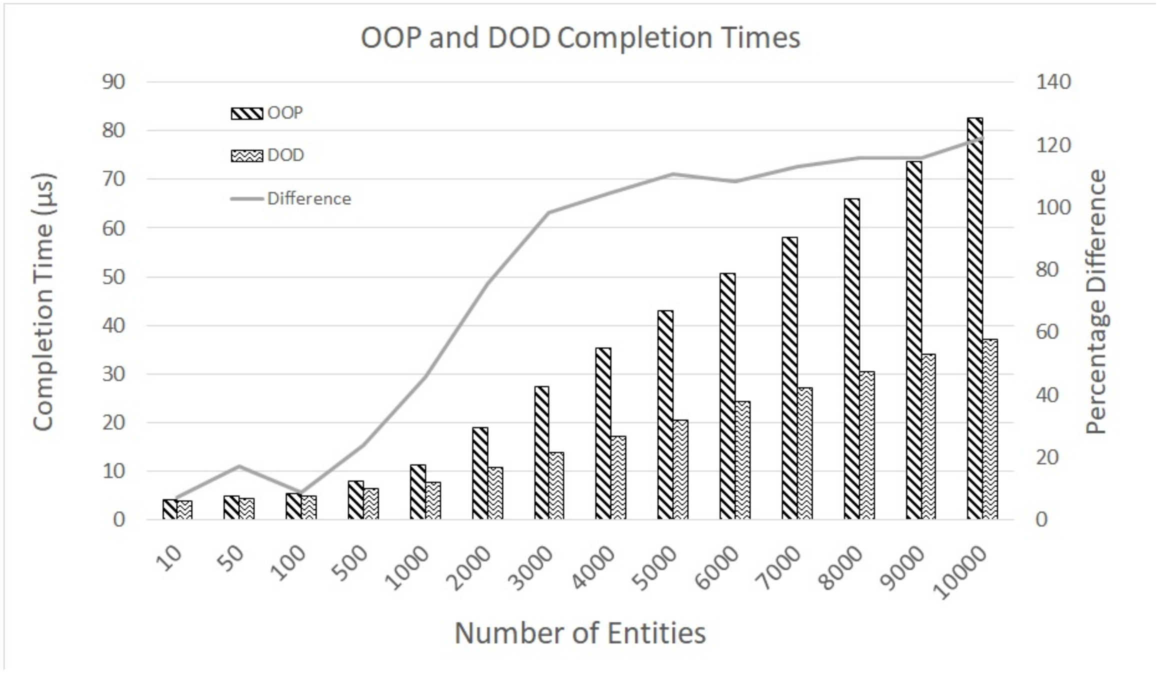

Figure 6: Graphical Results for Experiment Two

Figure 6 shows the completion time (μs) of the OOP and DOD software architectures on the left and the percent difference between the two architectures on the right. From Figure 6, it can be seen that for both OOP and DOD architectures, increasing the entity count did increase the completion time in a linear fashion as expected. Per the data, The OOP architecture had an increase in *completion_time* of 4.2μs for 10 entities to a *completion_time* of 72.7μs for 10,000 entities resulting in rate of roughly 7.86μs per 1,000 entities.

For the DOD architecture, there was an increase in *completion_time* of 3.9μs for 10 entities to a *completion_time* of 37.2μs for 10,000 entities resulting in a rate of roughly 3.33μs per 1,000 entities. This data not only supports the Student's t-test in showing that the two architectures are effected differently, but that it effects the OOP architecture to a greater extent, greatly reducing the performance compared to the DOD architecture. The average rate of increase differs by 135.9% between the two software architectures. At low entity count, the architectures performance is very similar: 4.2μs for the OOP architecture and 3.9μs for DOD architecture, a difference of only 7.34%, however it does not hold.

Due to the difference in the rate of increase in completion time, at 10,000 entities, that difference rose to 122.32%: 82.7μs for the OOP architecture and 37.2μs for the DOD architecture. The growth rate of the OOP architecture did not match the hypothesis, as it was expected to be greater than a linear rate of increase, however it did still hold true that the DOD architecture outperformed the OOP architecture as the number of entities increased. It is important to note that a low entity count is unusual for a simulation and a larger entity count would more accurately represent a simulation.

This difference in completion time rate can be explained with the same information provided in Section 4.2. As the amount of entities grow, there is a larger amount cache misses in the OOP software system due to the inefficient storage of data; the effect that $entity\_size$ has on the performance of the architecture is multiplied due to having many entities. The more entities, the greater the amount of cache misses; if a cache miss occurs every 100 entities, then 1,000 entities would result in 10 cache misses while 10,000 entities would result in 100 cache misses. Each cache miss affects the performance of the architecture. This is due to the unused data in each entity that is brought into the CPU's cache. It causes the cache to fill more often than it's DOD counterpart. DOD experiences an increase in cache misses also, as only so much data can fit in the CPU's cache at any given time. However, the rate of cache misses is much lower due to only the necessary state being brought into cache, wasting no space is cache.

	Entity_count	**10**	**50**	**100**	**500**	**1000**	**2000**	**3000**
OOP	**Time (μs)**	4.218	4.985	5.401	7.852	11.302	18.866	27.511
DOD	**Time (μs)**	3.930	4.260	4.962	6.332	7.757	10.755	13.878
	% Difference	7.34	17.03	8.85	24.01	45.70	75.42	98.23
	Entity_count	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**	**10000**
OOP	**Time (μs)**	35.278	43.029	50.604	58.026	65.918	73.62	82.703
DOD	**Time (μs)**	17.235	20.416	24.289	27.231	30.575	34.101	37.200
	% Difference	104.69	110.76	108.34	113.09	115.59	115.89	122.32

Table 14: Numerical Results for Experiment Two

4.4 Experiment 3

The objective of this experiment provide additional insight into the impact that *entity_size* and *entity_count* has on the OOP and DOD architectures. In this experiment, the independent variables, *entity_count* and *entity_size*, were iterated through the following values: [10, 50, 100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000] entities and [32, 64, 128, 256, 512, 1024, 2048] bytes respectively. Refer to Table 9 for all other variable values. The results for this experiment contain 100 data points per independent variable per software system. Of the 100 points, outliers are removed and the remaining are averaged to provide a single performance point for each. The averaged points are provided in Table 15 and Table 16 while, the code used for this experiment can be found in Appendix D.

Figure 7 and Figure 8 show two scatter plots of the different completion times for each total memory size value. the total memory size value was created by multiplying *entity_count* and *entity_size* together. That value is then divided by 8,000 bits, changing the unit from bits to kilobytes. Figure 7 shows the performance results for all points where as Figure 8 zooms into the first 5000KB. It can be seen that the DOD architecture consistently outperforms the OOP architecture when *total_memory_size* becomes greater than 5000KB. The peak difference between the

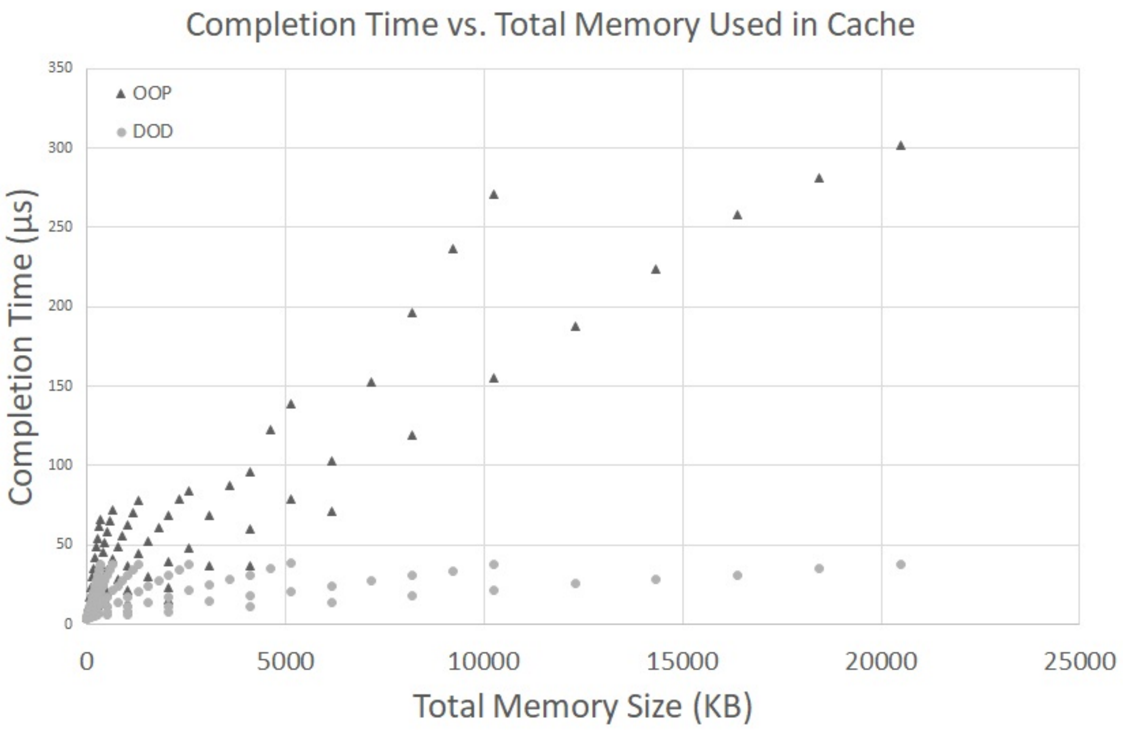

Figure 7: Graphical Results for Experiment Three: Full Scatter-plot of *total_memory_size*

two is when *total_memory_size* reaches 20.48MB: OOP's performance is 301.71μs where as DOD's performance is 38μs, a performance difference of 693.84%.

Figure 10, Figure 9, Figure 11 and Figure 12 break down the above results to show how each individual component of *total_memory_size* (*entity_count* and *entity_size*) effect the performance of the OOP and DOD architectures. Figure 10 and Figure 9 show that *entity_size* greatly effects the performance of the OOP system as *entity_count* increases, but does not effect the performance of the DOD system. Experiment 2 originally showed that increasing *entity_count* increased *completion_time* by roughly 3.3μs per 1,000 entities for the DOD system and by roughly 6.1μs per 1,000 entities for the OOP system.

The data from this experiment verifies this result for the DOD paradigm and expands the results to show that this is true regardless the size of the entity. This can

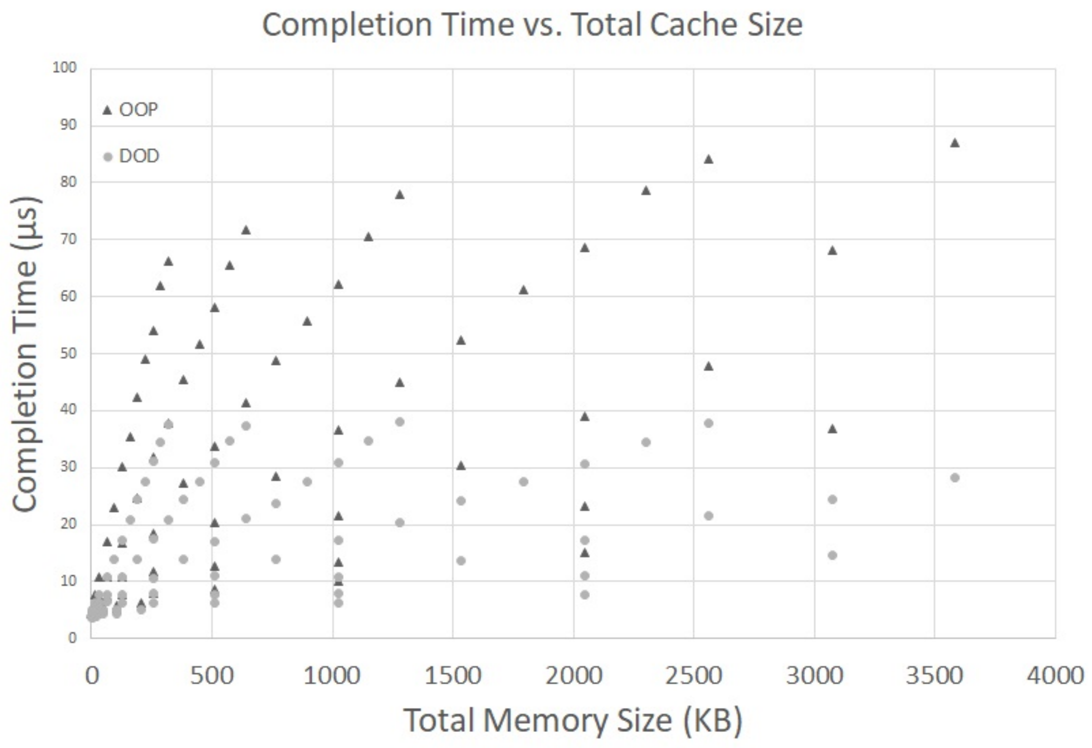

Figure 8: Graphical Results for Experiment Three: Partial Scatter-plot of *total_memory_size*

be seen by the fact that all entity sizes in Figure 10 have extremely similar slopes for the rate of increase in *completion_time*: roughly 3.3μs per 1,000 entities with little deviation. The same can not be said for the OOP paradigm. It can be seen from Figure 9 that as the size of the entity increases, the slope at which *completion_time* increases grows larger, and eventually becomes non linear at around 4096 bit sized entity.

These results can best be explained by the fact that the CPU's cache line is not being filled until after 2048 bits. This is also backed by the fact that the non-linear increase in *completion_time* occurs at different *entity_count* marks depending on the size of the entity. As an example, at 4096 bit size entities, the non-linear growth rate starts around 5000 entities. However, for the 8192 bit size entities, this non-linear growth rate starts between 2000 and 3000 entities. At the aforementioned points at

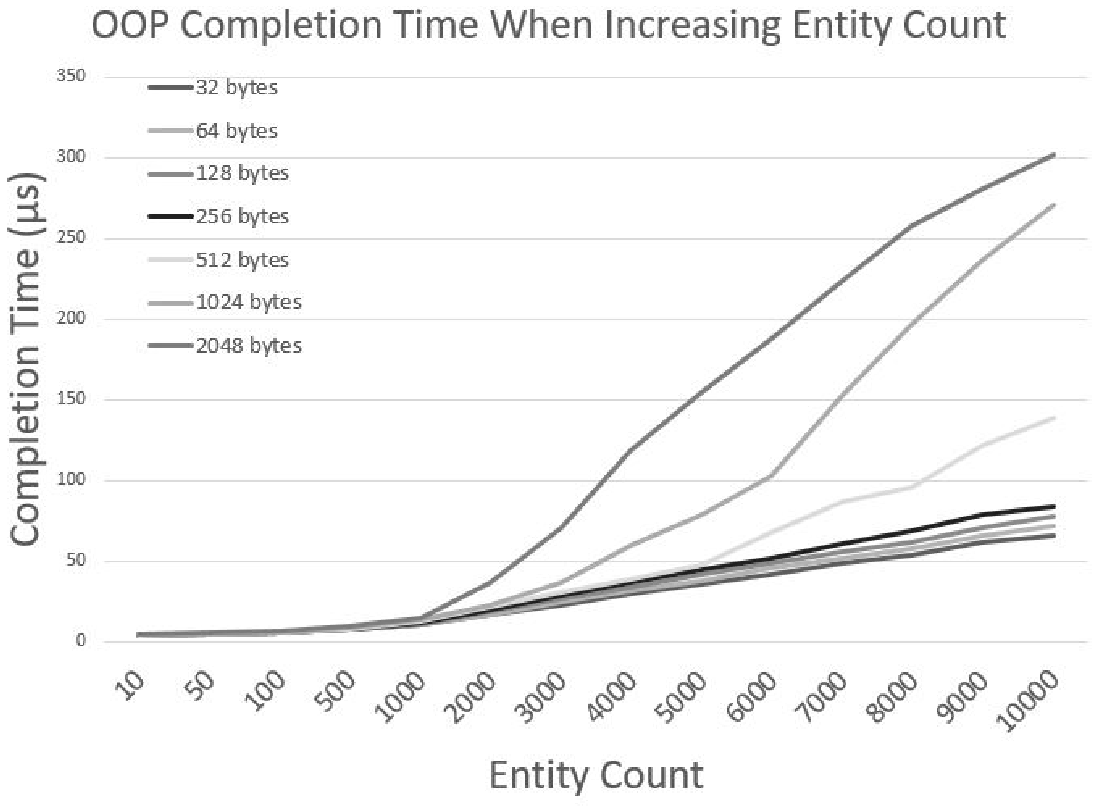

Figure 9: Graphical Results for Experiment Three: OOP *entity_size* effects

which the growth rate becomes non-linear, it is hypothesized that cache misses begin to occur at an increased rate, causing the *completion_time* to increase at a non-linear rate. It should be noted, even at it's lowest rate, the OOP architecture has a rate nearly 2x as large as the DOD architecture.

When comparing both the OOP and DOD architectures, the largest performance difference for the DOD architecture was only 7.1%; this was for the two sizes of 1,024 bits and 16,384 bit entity sizes, with corresponding values of 23.80μs and 25.51μs respectively. For the OOP architecture, the largest performance difference was 355% with corresponding values of 66.29μs and 301.71μs. These results show that the DOD architecture is more stable as *entity_count* increases, regardless of it's other variable values.

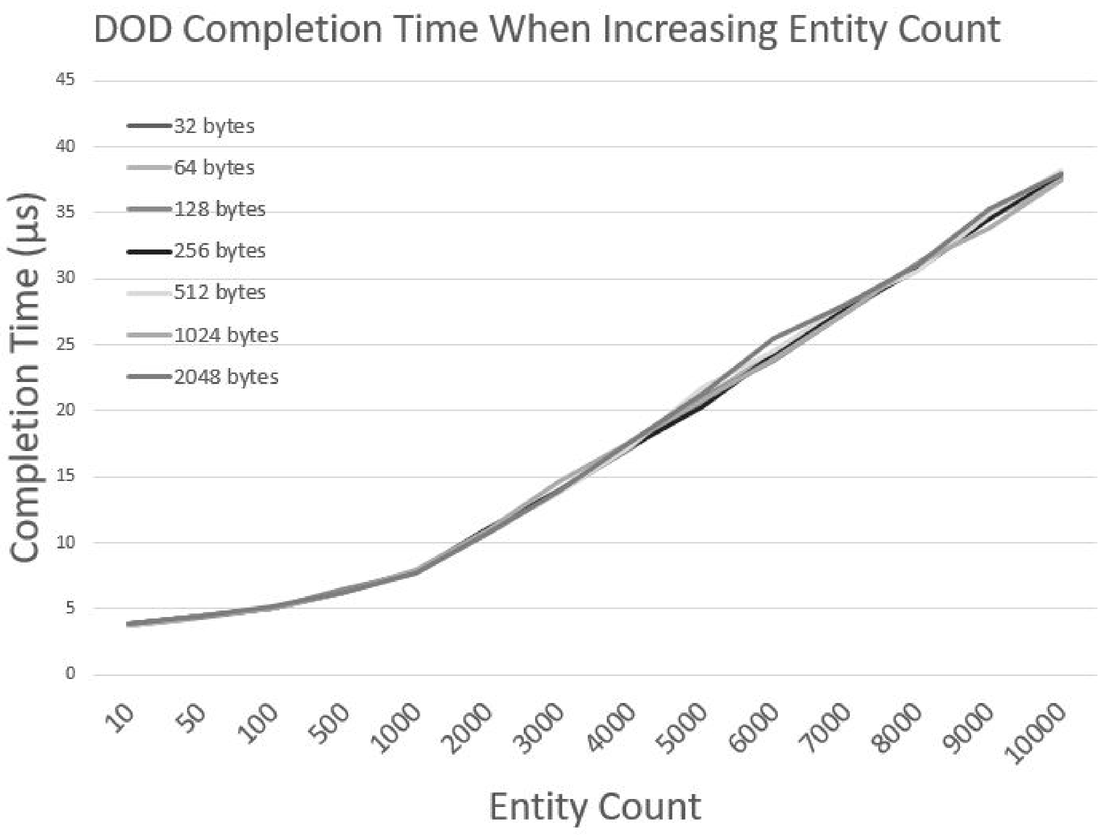

Figure 10: Graphical Results for Experiment Three: DOD *entity_size* effects

Figure 11 and Figure 12 confirm one of the observations made in Experiment 2: that *entity_count* effects the performance of the DOD architecture to a lesser extent than it effects the performance of the OOP architecture. This can be seen by the fact that as *entity_size* increases in Figure 12, the *completion_time* remains constant, regardless of the *entity_count*. Only *entity_count* affects the performance. However, Figure 11 shows that the *completion_time* does not remain constant as *entity_size* increases. The rate at which *completion_time* increases depends on the *entity_count*. As there is a larger *entity_count*, not only is the baseline *completion_time* higher, but the the rate at which *completion_time* increases is greater.

The largest performance difference in the OOP architecture can be seen when there are 10,000 entities in the simulation. The performance differed from 66.292μs to

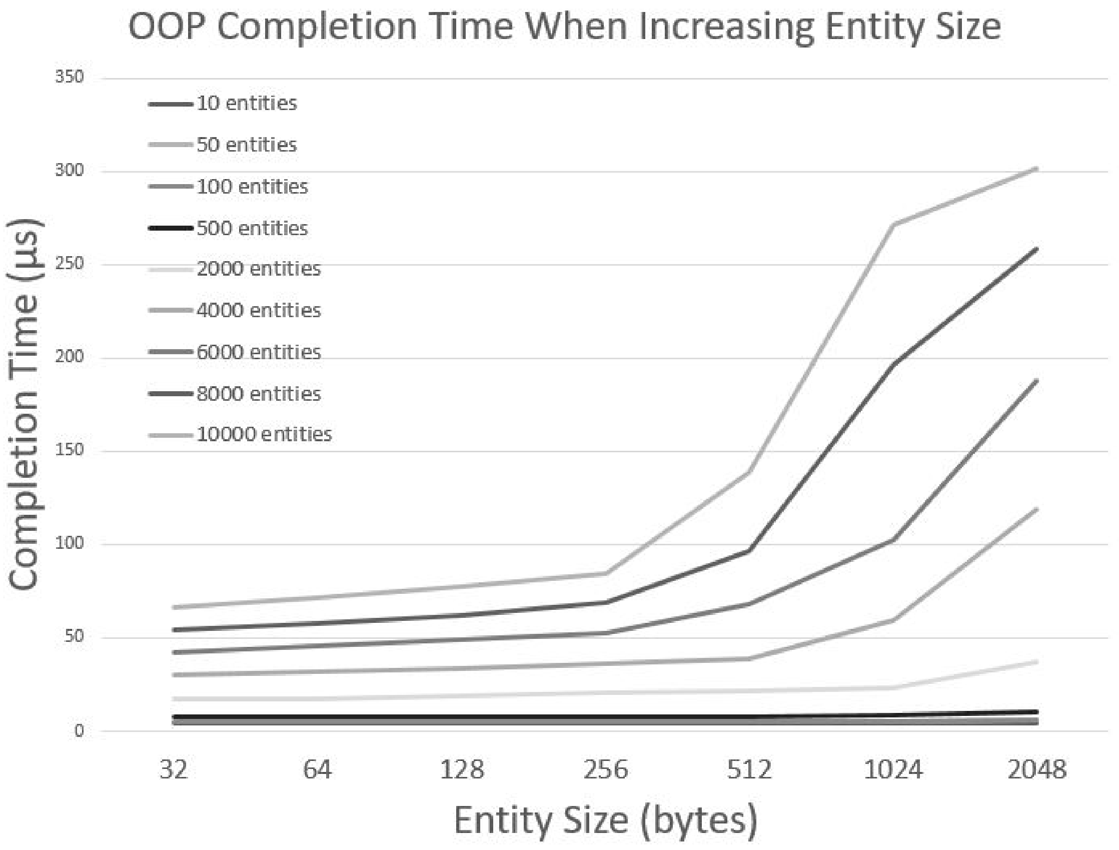

Figure 11: Graphical Results for Experiment Three: OOP *entity_count* effects

$301.71\mu s$ while *entity_size* was increased. That is a gain of 355% in *completion_time* by only increasing the amount of unused state in an entity. For the DOD architecture, the greatest performance difference was at 6,000 entities in the simulation. The performance differed from $23.799\mu s$ to $25.51\mu s$ while *entity_size* was increased. That is only a gain of 7%. As can be seen, there is a vast difference between the two architectures, 355% for the OOP architecture compared to 7% for the DOD architecture.

The four graphs above show that *entity_count* and *entity_size* work together to determine the performance of the OOP architecture due to the fact that the entire entity must be brought into the cache, wasting space and resulting in additional cache misses. However, the same does not hold true for the DOD architecture. The four graphs above show that only *entity_count* is used to determine the performance due

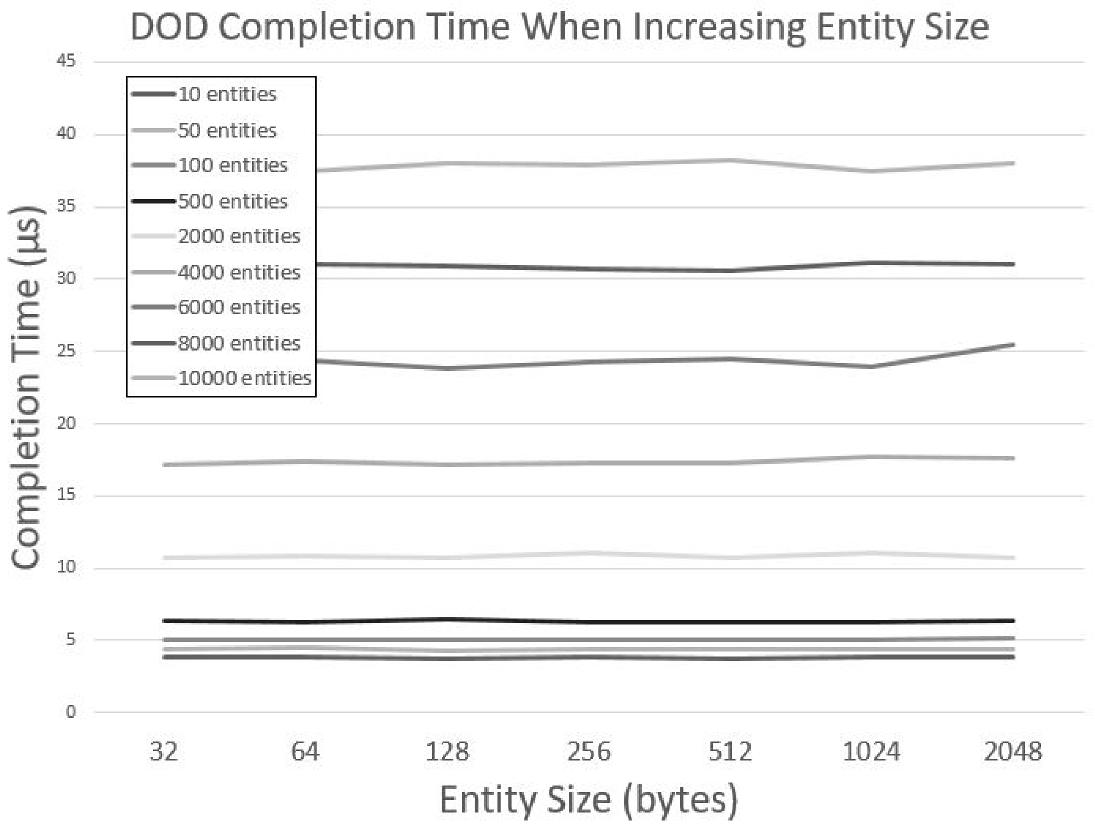

Figure 12: Graphical Results for Experiment Three: DOD *entity_count* effects

to the fact that only the necessary state is brought into the cache. This results in less cache misses.

The data above shows that *completion_time* would increase as *total_memory_size* increased, but that the OOP architecture's *completion_time* would increase at a rate much greater than the DOD architecture was confirmed. The three aforementioned figures show that the DOD architecture consistently outperformed the OOP architecture as *total_memory_size* increased.

Performance of OOP System (μs) while varying both the entity_count and entity_size														
entity_count	**10**	**50**	**100**	**500**	**1000**	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**	**10000**
32	4.309	5.026	5.370	7.802	10.809	16.987	23.112	30.285	35.459	42.319	49.082	54.139	61.898	66.292
64	4.285	5.003	5.369	7.556	10.857	16.900	24.787	31.900	37.925	45.43	51.661	58.137	65.533	71.787
128	4.408	5.060	5.478	7.567	10.897	18.54	27.275	33.856	41.434	48.866	55.844	62.232	70.541	77.957
256	4.263	5.104	5.504	7.745	11.698	20.295	28.484	36.563	44.917	52.320	61.126	68.640	78.719	84.131
512	4.276	5.147	5.509	7.996	12.660	21.587	30.327	39.002	47.819	68.275	87.130	96.089	122.30	138.78
1024	4.385	5.292	5.438	8.771	13.458	23.277	36.933	59.632	78.468	102.42	152.57	196.47	236.86	271.08
2048	4.321	5.835	6.339	10.095	15.178	37.169	71.056	119.02	154.77	187.56	224.10	258.29	280.90	301.71

Table 15: Numerical Results for Experiment Three OOP

Performance of DOD System (μs) while varying both the entity_count and entity_size														
entity_count	**10**	**50**	**100**	**500**	**1000**	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**	**10000**
32	3.822	4.390	5.016	6.331	7.801	10.768	13.930	17.171	20.819	24.398	27.443	31.194	34.507	37.549
64	3.823	4.471	5.047	6.236	7.780	10.813	14.035	17.431	20.876	24.352	27.646	30.986	34.615	37.443
128	3.739	4.337	5.004	6.471	7.701	10.706	13.914	17.135	20.988	23.799	27.454	30.879	34.623	37.993
256	3.812	4.396	5.091	6.227	7.848	11.068	13.893	17.297	20.289	24.289	27.637	30.687	34.596	37.889
512	3.727	4.406	5.052	6.206	7.724	10.739	13.768	17.237	21.691	24.513	28.207	30.583	34.952	38.189
1024	3.867	4.381	5.099	6.267	7.906	11.016	14.559	17.664	20.729	23.946	27.456	31.139	33.827	37.442
2048	3.840	4.419	5.163	6.309	7.700	10.731	13.927	17.596	21.253	25.510	28.072	31.004	35.336	38.006

Table 16: Numerical Results for Experiment Three DOD

4.5 Experiment 4

The objective of this experiment was to determine if the thread and system count has a different effect on the performance of the OOP and DOD architectures. In this experiment, the two independent variables, *thread_count* and *system_count* were iterated through the following values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Refer to Table 10 for all other variable values. The results for this experiment contain 100 data points per independent variable pair per software architecture. For Figure 13, of the 100 points, outliers were removed and the remaining were averaged to provide a single performance point. The averaged points are provided in Table 17 while the code used in this experiment can be found in Appendix E.

To determine if there is enough statistical evidence to show that the combination of *thread_count* and *system_count* does not effect the performance of the DOD and OOP architectures differently, a paired Student's t-test comparing the completion time between the two architectures was performed. The data used for the Student's t-test can be found in Table 17. After performing the Student's t-test as described in Section 3.4, the resulting p-value was 0.20. A p-value of 0.20 is not enough statistical evidence to reject the null hypothesis, thus it shows that the combination of system and thread count does not effect the two architectures differently, as expected.

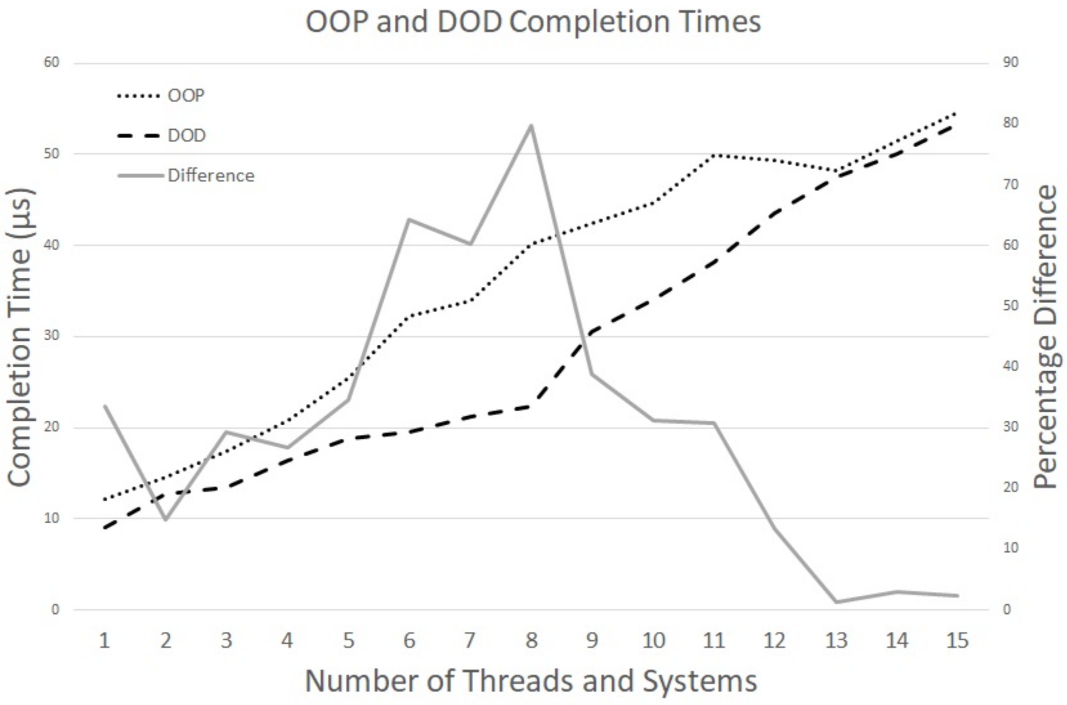

Figure 13: Graphical Results for Experiment Four

Interestingly, it can be seen in Figure 13 that performance does not remain the same as the amount of threads increase. It would be expected that since both threads and systems increase at the same rate, *completion_time* would remain constant as each thread does the same amount of work. The best explanation as to why this is not the case is the fact that as the amount of threads increases, so does the overhead of maintaining and pushing work to the thread. This is because the threads are managed by a thread-pool and the workload is pushed to the threads every cycle. However, Figure 13 confirms the results from the Student's t-test: that performance of both the DOD and OOP architectures remain similar. The performance difference is as little as 1.36% at 13 threads. The interesting results from this experiment is the performance difference of the systems while close the the thread count of the

CPU. The largest difference in performance occurs exactly at the thread count of the CPU, where the DOD system is outperforming the OOP system by 79.78%: the OOP architecture has a *completion_time* of 40.178μs and the DOD architecture has a *completion_time* of 22.344μs.

It was thought that this is due to the fact that the DOD is able to split the workload between threads more precisely than the OOP system. The OOP system can only split the workload by divvying up the entities between each thread and calling all systems on each entity; the DOD system, however, can divvy up the workload not only by the entity, but also by the system. Therefore one thread is not stuck calling all methods on a single entity: if a different thread finishes early, it can steal work from another thread and begin calling the methods of a different entity. This explains why at low threads, the performance is similar, but grows steadily apart. It is also thought that the performance begins to merge after the the CPU's thread count because after that point, only 8 of the possible threads are truly running in parallel, so overhead of managing many threads, but not actually running in parallel (past 8 threads) begins to dictate the performance of the software systems.

	System/Thread	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
OOP	**Time (μs)**	12.150	14.581	17.343	20.746	25.427	32.176	33.914	40.178
DOD	**Time (μs)**	9.103	12.699	13.427	16.377	18.879	19.575	21.150	22.344
	% Difference	33.47	14.82	29.17	26.68	34.68	64.37	60.35	79.78
	System/Thread	**9**	**10**	**11**	**12**	**13**	**14**	**15**	
OOP	**Time (μs)**	42.364	44.755	49.971	49.397	48.164	51.515	54.546	
DOD	**Time (μs)**	30.524	34.141	38.212	43.563	57.519	50.073	53.256	
	% Difference	38.79	31.09	30.77	13.39	1.36	2.88	2.42	

Table 17: Numerical Results for Experiment Four

4.6 Experiment 5

The objective of this experiment was to determine if the number of threads has a different effect on the performance of the OOP and DOD architectures. In this experiment, the independent variable, *thread_count*, was iterated through the following values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] threads. Refer to Table 11 for all other variable values. The results for this experiment contain 100 points, outliers were removed and the remaining were averaged to provide a single performance point. The averaged points are provided in Table 18 while the code used in this experiment can be found in Appendix F.

To determine if there is enough statistical evidence to show that the *thread_count* does not effect the performance of the DOD and OOP architectures differently, a paired Student's t-test comparing the completion time between the two architectures was performed. The data used for the Student's t-test can be found in Table 18. After performing the Student's t-test as described in Section 3.4, the resulting p-value was 0.40. A p-value of 0.40 is not enough statistical evidence to reject the null hypothesis, thus it shows that the amount of threads does not effect the two architectures differently, as expected.

Figure 14 shows the performance of the OOP and DOD architectures as more threads were provided to handle a workload. This experiment provides a workload of solving a recursive Fibonacci sequence 15 times for each entity. It was set up such that each calculation of the Fibonacci sequence could be calculated on different threads. Figure 14 supports the statistical evidence form the Student's t-test as it can be seen that the two architectures perform nearly identical, having only a 5.1% difference in *completion_time* with one thread and 1.1% difference in *completion_time* at 15 threads. As can be seen in figure, the OOP architecture is more unstable due to the decreased performance at 5 and 9 threads, however both architectures meet optimal performance at 4 threads. This makes sense as the tests were ran on a device with 4 cores and 8 threads, so results should be optimal at these points.

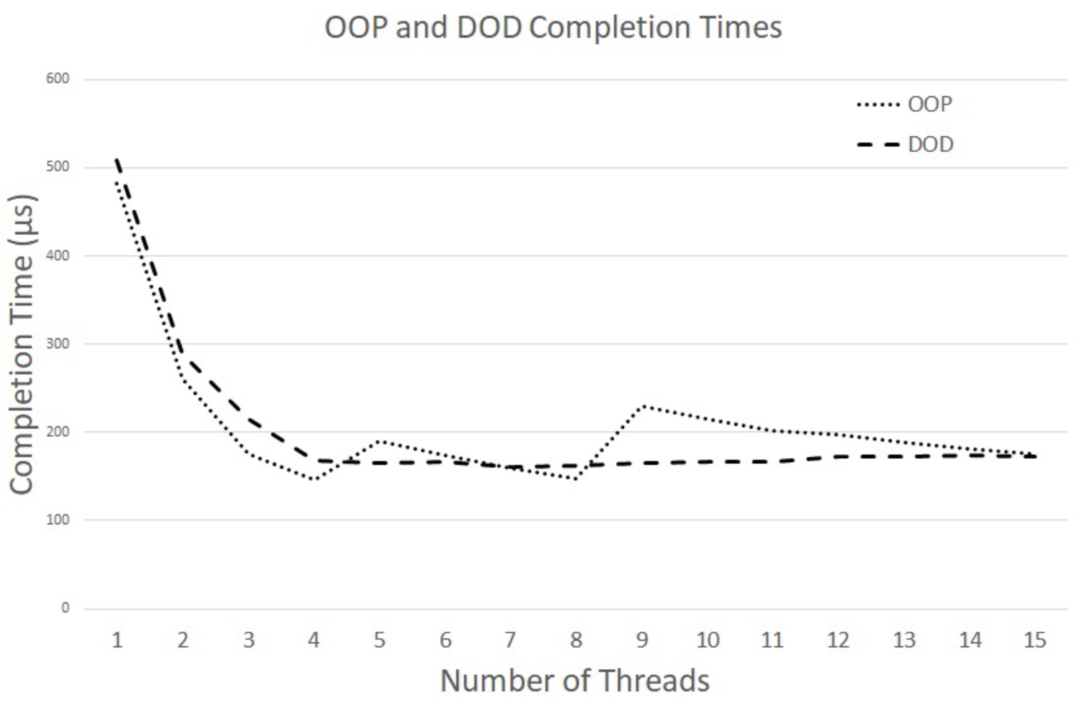

Figure 14: Graphical Results for Experiment Five

	Thread_count	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
OOP	**Time (μs)**	482	260	176	146	190	174	158	147	229	214	202	198	188	180	175
DOD	**Time (μs)**	508	289	215	168	165	166	160	162	164	167	167	172	173	174	173
	% Difference	5.1	9.7	18.0	12.5	15.4	5.3	0.9	9.1	39.2	28.5	21.0	14.8	8.6	3.8	1.1

Table 18: Numerical Results for Experiment Five

4.7 Experiment 6

The objective of this experiment was to determine if the amount components used in a system has different effects on the performance of the OOP and DOD architectures. In this experiment, the independent variable, *component_count*, was iterated through the following values: [16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2] components. Refer to Table 12 for all other variable values. The results of this experiment contain 100 data points per independent variable per software architecture. For Figure 15, of the 100 points, outliers were removed and the remaining were averaged to provide a single performance point. The averaged points are provided in Table 19 while the code used in this experiment can be found in Appendix G.

To determine if there is enough statistical evidence to show that the component count effects the performance of the DOD and OOP architectures differently, a paired Student's t-test comparing the completion time between the two architectures was performed. The data used for the Student's t-test can be found in Table 19. After performing the Student's t-test as described in section 3.4, the resulting p-value was 0.00956. A p-value of 0.00956 is enough statistical evidence to reject the null hypothesis and fail to reject the alternative hypothesis. This means that there is enough evidence to suggest that system size does effect the two architectures differently.

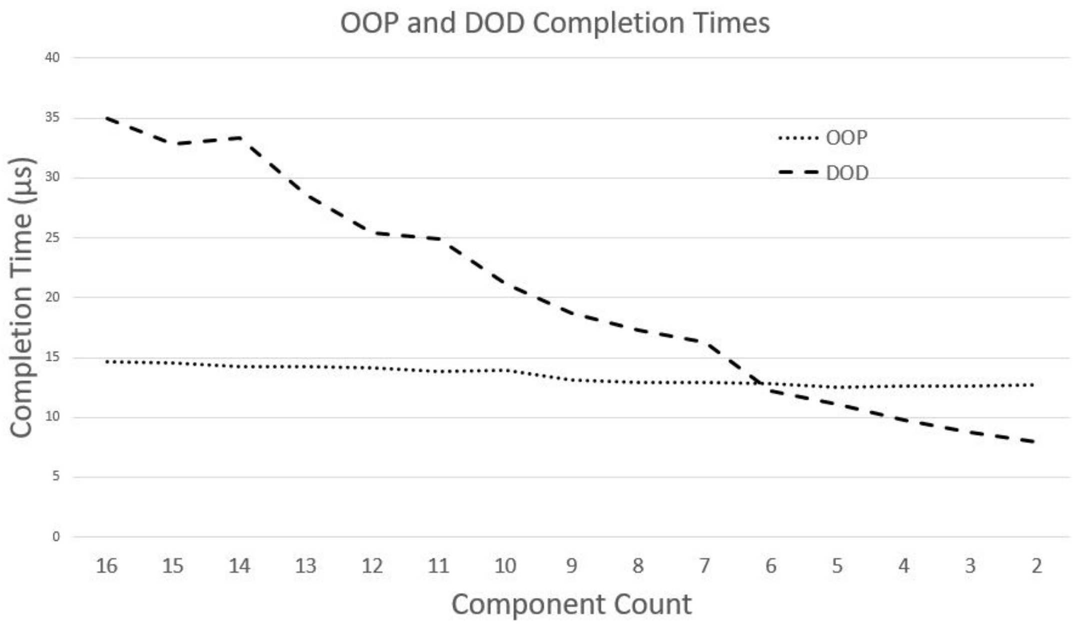

Figure 15: Graphical Results for Experiment Two: Comparing Completion Time

Figure 15 compares the *completion_time* of the DOD and OOP architectures while changing the amount of components being used by the systems. It shows that *completion_time* decreases at an extremely high rate for the DOD architecture where as it only decreases slightly for the OOP architecture, solidifying the statistical evidence provided by the Student's t-test that the amount of components the system uses effects the architectures differently. The DOD architecture's *completion_time* decreases by 339.69%, from 34.92μs to 7.94μs where as it only decreases by 15.7%, from 19.69μs to 14.68μs for the OOP architecture.

Interestingly, the performance of the DOD architecture rapidly increased as the amount of components used in each system decreased while performance of the OOP architecture remained almost constant. It was unexpected however, as to the performance difference between OOP and DOD when using a large amount of components in a system. At 16 components being used in a single system, the performance difference between OOP and DOD was 137.82% with DOD at 34.92μs and OOP at 14.68μs. This can best be explained by the DOD architecture needing to verify that an entity has all components required for the system, before executing the system.

The intersection in performance is when the method is using 6 components, making it only a benefit to use the DOD architecture when systems use less 7 components. It should be noted that 7 components is an extremely large amount components to be using in a single method however and can probably be refactored into multiple smaller systems.

	Component Count	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2
OOP	% Change from initial	1	1.23	2.93	3.23	3.63	5.82	5.75	12.31	13.71	13.32	14.82	17.31	16.35	16.51	15.70
OOP	Time (μs)	14.68	14.51	14.27	14.22	14.17	13.88	13.89	13.07	12.92	12.96	12.79	12.52	12.62	12.60	12.69
DOD	Time (μs)	34.92	32.86	33.37	28.61	25.57	24.92	21.16	18.67	17.33	16.23	12.24	11.06	9.79	8.78	7.94
DOD	% Change from initial	1	6.26	4.64	22.08	37.66	40.13	64.92	87.04	101.5	115.2	185.2	215.7	256.6	297.6	339.7

Table 19: Numerical Results for Experiment Six

V. Conclusion

5.1 Overview

This chapter summarizes the research and results obtained throughout the thesis. Section 5.2 restates important results found during experimentation and analysis. Section 5.3 states the impact that this research has on the field along with possibilities for future work and uses for the Entity-Component-System (ECS) architecture.

5.2 Research Conclusions

This research concludes that the Data-Oriented Design (DOD) paradigm, and in particular, the ECS architecture, does have a strong use case in the area of modeling & simulation, specifically in real-time simulations. It successfully tested many of the common attributes of a real-time simulator and their effects on the performance of the architecture compared to the Object-Oriented Programming (OOP) counterpart. Through six experiments, it is determined that this architecture greatly improves the overhead of retrieving and storing data for use in the application.

Through statistical analysis, experiment 1 shows that the ECS architecture is tolerant to the performance loss that OOP architectures have when operating on increasingly large objects. In normal use cases, a 100% improvement in the overhead of reading and writing data to memory is seen compared to the OOP counterpart. An improvement of over 600% to overhead is seen in uses cases where object size is extremely large, such as 16,384 bit size objects. Most importantly, this experiment shows that overhead performance remains constant regardless of the size of the object for ECS architectures, where as it plays an important role in determining the performance of an architecture using the OOP paradigm.

Experiment 2 shows that the ECS architecture is more tolerant to increased entity count in the simulation compared to a OOP architecture. In this experiment, it is seen that performance overhead of reading and writing data to memory increased at a rate of $3.33\mu s$ for every 1,000 entities for the ECS architecture where as it increased at a rate of $7.86\mu s$ every 1,000 entities for the OOP architecture. This is a difference of 135%. Experiment 3 reiterated and confirmed the findings of Experiments 1 and 2, providing further testing and results to solidify and verify the findings.

Experiments 4 and 5 shows that the performance overhead of parallelizing the code-base is similar for both architectures. The results show that there is a minimal improvement in overhead costs when operating on a threadpool with a count near the core count of the Central Processing Unit (CPU), though in any other amount of threads, the performance difference is negligible. The important aspect of this experiment is that performance remains similar between both architectures, however the ECS architecture handles the multi-threading automatically, and guarantees thread safety, where as the developer is responsible for both normally for the OOP paradigm.

Experiment 6 shows that the amount of components the system uses greatly effects the performance of the ECS architecture where as it has no affect on the OOP architecture. This experiment shows that the ECS architecture is only optimal when the amount of components used in a system is less than around 6 components. It is important to reiterate that these components are in regards to the components used in the system, not that exists in the entity. At its worst, the OOP architecture out performs the ECS architecture by 137%. At its best, the ECS architecture outperforms the OOP architecture by 60%. It is important to note that most systems only operate on small amounts of components at any given time, so the ECS architecture generally outperforms the DOD architecture.

Overall, ECS has shown to be an excellent candidate as a software architecture for for real-time simulators. This architecture reduces memory overhead consistently against all factors tested in the experiments. The only area of concern is if the systems that are executing on state require extremely large amounts of data from each entity. When this occurs, the ECS architecture is a poor choice for use in the software. It should once again be emphasized that DOD and ECS is not a replacement to OOP in all use cases. DOD is most optimal in code-bases that contain and operate on a large amounts of data; this includes, but is not limited to, areas such as video games engines, real-time simulations, simulations, and high performance computing.

5.3 Research Significance and Future Work

As improvements in the development of faster computer hardware continues to slow, the need for more performant and scalable software systems will increase. Hardware engineers have turned to increasing the core count in CPU's to improve the performance. However, developing safe, multi-threaded software is no small feat. Many developers are never taught the underlying aspects of hardware that their software runs on such as how data is stored and accessed, or how a CPU and/or operating system handles multiple threads. Due to this, they are unable to make many of the optimizations that can greatly increase the performance of their code-base.

This work has focused on researching, testing, and bench-marking a software architecture normally used in game development, for use in other areas. This software architecture abstracts away two of these software/hardware aspects that many developers are unaware of, or find difficult optimizing. The ECS architecture abstracts away the need for the developer determine how and where data is stored in memory, along with writing code that is thread-safe and concurrent. Additionally, the research done in this work has shown that the optimizations made to these two aspects consistently out performed that of the OOP architecture counterparts.

The ECS architecture and DOD paradigm as a whole have many areas of future research that could greatly benefit the field of modeling & simulation, high performance computing, and the Air Force. The following suggests five future work options based off this research:

- Research into additional ECS frameworks. As the ECS architecture does not have a standard implementation, other implementations may be better optimized for modeling and simulation use-cases. Possible frameworks to research include: (1) **legion** for its more precise ability to filter components, (2) **hecs** which builds on the successes of specs and legion; it is a library rather than an architecture, allowing the developer to more greatly customize the implementation for their needs, and (3) **froggy**, a prototype for the Component Graph System programming model. It aims to combine the convenience of composition-style OOP with the performance close to ECS.

- Development of a prototypical use case for modeling and simulation using existing ECS frameworks. Use an existing ECS architecture to design and develop a modeling and simulation program for a real-world problem such as a flight simulator. This will give the ability to test the ECS architecture's real-world application performance. Finally, compare this performance to the performance of an OOP designed framework.

- Development of a custom ECS architecture for use in future modern military simulators. This would allow for the development of specific functionality and optimizations for use cases in the field of modeling & simulation and the Air Force. This includes an optimized scheduling algorithm for providing system workloads to a threadpool, along with optimizing memory accesses for the needs of common modern military simulators.

- Research into the possible use of the ECS as a framework for high-performance computing. The ECS architecture is able to scale with threadcount, making it a perfect candidate for working on High-Performance Computers/Super Computers which provide the developer with an extremely large amount of threads. Additionally, its ability to schedule systems on different threads without data safety concerns allows the developer to easily develop highly parallelized code for high-performance computing.

- Research into the possible use of the ECS framework for Graphics Processing Unit (GPU) accelerated computing. The GPU specializes in executing the same piece of code (a shader) on a large amount of data points very efficiently. The ECS architecture does the same thing, executing a system on a large amount of entities. Also, the ECS architecture stores state in large arrays of contiguous memory, similar to how data is provided to the GPU from the CPU. Instead of ECS systems manipulating state via the CPU, systems could be replaced with calls to the GPU to do the same calculations on the same data, only with shaders. The memory is already in a structure that the GPU expects.

Appendix A. Benchmark Measurement and Analysis

The information provided below is directly taken from the user guide for the Criterion crate which can be found at `https://bheisler.github.io/criterion.rs/book/analysis.html`. The information is being directly transferred here to ensure available should the link above change. The information below is not the work of the author of this thesis, but the work of the creator of Criterion.

1.1 Measurement

The measurement phase is when Criterion.rs collects the performance data that will be analyzed and used in later stages. This phase is mainly controlled by the *measurement_time* value in the Criterion struct.

The measurements are done in a number of samples (see the *sample_size* parameter). Each sample consists of one or more (typically many) iterations of the routine. The elapsed time between the beginning and the end of the iterations, divided by the number of iterations, gives an estimate of the time taken by each iteration.

As measurement progresses, the sample iteration counts are increased. Suppose that the first sample contains 10 iterations. The second sample will contain 20, the third will contain 30 and so on. More formally, the iteration counts are calculated like so: $iterations = [d, 2d, 3d, ...Nd]$

Where N is the total number of samples and d is a factor, calculated from the rough estimate of iteration time measured during the warm-up period, which is used to scale the number of iterations to meet the configured measurement time. Note that d cannot be less than 1, and therefore the actual measurement time may exceed the configured measurement time if the iteration time is large or the configured measurement time is small.

Note that Criterion.rs does not measure each individual iteration, only the complete sample. The resulting samples are stored for use in later stages. The sample data is also written to the local disk so that it can be used in the comparison phase of future benchmark runs.

1.2 Analysis

During this phase Criterion.rs calculates useful statistics from the samples collected during the measurement phase.

1.2.1 Outlier Classification

The first step in analysis is outlier classification. Each sample is classified using a modified version of Tukey's Method, which will be summarized here. First, the interquartile range (IQR) is calculated from the difference between the 25th and 75th percentile. In Tukey's Method, values less than (25th percentile - 1.5 * IQR) or greater than (75th percentile + 1.5 * IQR) are considered outliers. Criterion.rs creates additional fences at (25pct - 3 * IQR) and (75pct + 3 * IQR); values outside that range are considered severe outliers.

Outlier classification is important because the analysis method used to estimate the average iteration time is sensitive to outliers. Thus, when Criterion.rs detects outliers, a warning is printed to inform the user that the benchmark may be less reliable. Additionally, a plot is generated showing which data points are considered outliers, where the fences are, etc.

Note, however, that outlier samples are not dropped from the data, and are used in the following analysis steps along with all other samples.

1.2.2 Linear Regression

The samples collected from a good benchmark should form a rough line when plotted on a chart showing the number of iterations and the time for each sample. The slope of that line gives an estimate of the time per iteration. A single estimate is difficult to interpret, however, since it contains no context. A confidence interval is generally more helpful. In order to generate a confidence interval, a large number of bootstrap samples are generated from the measured samples. A line is fitted to each of the bootstrap samples, and the result is a statistical distribution of slopes that gives a reliable confidence interval around the single estimate calculated from the measured samples.

This resampling process is repeated to generate the mean, standard deviation, median and median absolute deviation of the measured iteration times as well. All of this information is printed to the user and charts are generated. Finally, if there are saved statistics from a previous run, the two benchmark runs are compared.

1.2.3 Comparison

In the comparison phase, the statistics calculated from the current benchmark run are compared against those saved by the previous run to determine if the performance has changed in the meantime, and if so, by how much.

Once again, Criterion.rs generates many bootstrap samples, based on the measured samples from the two runs. The new and old bootstrap samples are compared and their T score is calculated using a T-test. The fraction of the bootstrapped T scores which are more extreme than the T score calculated by comparing the two measured samples gives the probability that the observed difference between the two sets of samples is merely by chance. Thus, if that probability is very low or zero, Criterion.rs can be confident that there is truly a difference in execution time between the two samples. In that case, the mean and median differences are bootstrapped and printed for the user, and the entire process begins again with the next benchmark.

This process can be extremely sensitive to changes, especially when combined with a small, highly deterministic benchmark routine. In these circumstances even very small changes (eg. differences in the load from background processes) can change the measurements enough that the comparison process detects an optimization or regression. Since these sorts of unpredictable fluctuations are rarely of interest while benchmarking, there is also a configurable noise threshold. Optimizations or regressions within (for example) +-1% are considered noise and ignored. It is best to benchmark on a quiet computer where possible to minimize this noise, but it is not always possible to eliminate it entirely.

Appendix B. Experiment One Code

2.1 Benchmark

```
use criterion::{criterion_group, criterion_main, Criterion};
extern crate thesis_experimentation;

use specs::prelude::*;
use thesis_experimentation::exp1::oop_obj::*;
use thesis_experimentation::exp1::oop::*;
use thesis_experimentation::exp1::dod::*;
use std::time::Duration;

#[inline]
fn dod_dispatch(d: &mut Dispatcher, mut w: &mut World) {
    d.dispatch_par(&mut w);
}

#[inline]
fn oop_dispatch<T: Exp1>(world: &mut OOPWorld<T>) { world.execute(); }

pub fn oop_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("oop_exp1");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);
    rayon::ThreadPoolBuilder::new().num_threads(1).build_global().unwrap();

    let o128 = obj_setup::<Obj128>();
    let o256 = obj_setup::<Obj256>();
    let o512 = obj_setup::<Obj512>();
    let o1024 = obj_setup::<Obj1024>();
    let o2048 = obj_setup::<Obj2048>();
    let o4196 = obj_setup::<Obj4096>();
    let o8192 = obj_setup::<Obj8192>();
    let o16384 = obj_setup::<Obj16384>();

    let mut world128 = OOPWorld::new(o128, 1);
    let mut world256 = OOPWorld::new(o256, 1);
    let mut world512 = OOPWorld::new(o512, 1);
    let mut world1024 = OOPWorld::new(o1024, 1);
    let mut world2048 = OOPWorld::new(o2048, 1);
    let mut world4196 = OOPWorld::new(o4196, 1);
    let mut world8192 = OOPWorld::new(o8192, 1);
    let mut world16384 = OOPWorld::new(o16384, 1);

    group.bench_function("oop_exp1_size_128", |b|
        b.iter(||oop_dispatch(&mut world128)));
    group.bench_function("oop_exp1_size_256", |b|
        b.iter(||oop_dispatch(&mut world256)));
    group.bench_function("oop_exp1_size_512", |b|
        b.iter(||oop_dispatch(&mut world512)));
    group.bench_function("oop_exp1_size_1024", |b|
        b.iter(||oop_dispatch(&mut world1024)));
    group.bench_function("oop_exp1_size_2048", |b|
        b.iter(||oop_dispatch(&mut world2048)));
    group.bench_function("oop_exp1_size_4196", |b|
        b.iter(||oop_dispatch(&mut world4196)));
    group.bench_function("oop_exp1_size_8192", |b|
        b.iter(||oop_dispatch(&mut world8192)));
    group.bench_function("oop_exp1_size_16384", |b|
        b.iter(||oop_dispatch(&mut world16384)));
}

pub fn dod_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("dod_exp1");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);

    let entity_size: Vec<i32> = vec![128,256,512,1024,2048,4096,8192,16384];

    entity_size.iter().for_each(|size| {
        let mut world = World::new();
        setup_component(&mut world).unwrap();
        setup_entity(*size, &mut world).unwrap();
        let mut dispatcher = setup_dispatcher(*size);

        let mut bench_name = String::from("dod_exp1_size_");
        let i = size.to_string();
        bench_name.push_str(&i);

        group.bench_function(bench_name, |b|
            b.iter( || dod_dispatch(&mut dispatcher, &mut world)));
    });
```

```
}
criterion_group!(oop, oop_criterion_benchmark);
criterion_group!(dod, dod_criterion_benchmark);
criterion_main!(oop,dod);
```

Listing B.1: Experiment 1: Benchmark

2.2 Modules

```
pub mod oop;
pub mod dod;
pub mod oop_obj;
pub mod dod_component;
pub mod dod_system;
```

Listing B.2: Experiment 1: Modules

2.3 DOD

```
use specs::prelude::*;
use std::io;
use super::dod_component::*;
use super::dod_system::*;
use std::sync::Arc;

//Add components to the world
pub fn setup_component(world: &mut World)-> io::Result<()> {
    world.register::<Comp_i64_0>();
    world.register::<Comp_i64_1>();
    world.register::<Comp_i128_0>();
    world.register::<Comp_i128_1>();
    world.register::<Comp_i128_2>();
    world.register::<Comp_i128_3>();
    world.register::<Comp_i128_4>();
    world.register::<Comp_i128_5>();
    world.register::<Comp_i128_6>();
    world.register::<Comp_i128_7>();
    world.register::<Comp_i128_8>();
    world.register::<Comp_i128_9>();
    world.register::<Comp_i128_10>();
    world.register::<Comp_i128_11>();
    world.register::<Comp_i128_12>();
    world.register::<Comp_i128_13>();
    world.register::<Comp_i128_14>();
    world.register::<Comp_i128_15>();
    world.register::<Comp_i128_16>();
    world.register::<Comp_i128_17>();
    world.register::<Comp_i128_18>();
    world.register::<Comp_i128_19>();
    world.register::<Comp_i128_20>();
    world.register::<Comp_i128_21>();
    world.register::<Comp_i128_22>();
    world.register::<Comp_i128_23>();
    world.register::<Comp_i128_24>();
    world.register::<Comp_i128_25>();
    world.register::<Comp_i128_26>();
    world.register::<Comp_i128_27>();
    world.register::<Comp_i128_28>();
    world.register::<Comp_i128_29>();
    world.register::<Comp_i128_30>();
    world.register::<Comp_i128_31>();
    world.register::<Comp_i128_32>();
    world.register::<Comp_i128_33>();
    world.register::<Comp_i128_34>();
    world.register::<Comp_i128_35>();
    world.register::<Comp_i128_36>();
    world.register::<Comp_i128_37>();
    world.register::<Comp_i128_38>();
    world.register::<Comp_i128_39>();
    world.register::<Comp_i128_40>();
    world.register::<Comp_i128_41>();
    world.register::<Comp_i128_42>();
    world.register::<Comp_i128_43>();
    world.register::<Comp_i128_44>();
    world.register::<Comp_i128_45>();
    world.register::<Comp_i128_46>();
    world.register::<Comp_i128_47>();
    world.register::<Comp_i128_48>();
    world.register::<Comp_i128_49>();
    world.register::<Comp_i128_50>();
    world.register::<Comp_i128_51>();
    world.register::<Comp_i128_52>();
    world.register::<Comp_i128_53>();
    world.register::<Comp_i128_54>();
    world.register::<Comp_i128_55>();
    world.register::<Comp_i128_56>();
    world.register::<Comp_i128_57>();
    world.register::<Comp_i128_58>();
    world.register::<Comp_i128_59>();
    world.register::<Comp_i128_60>();
    world.register::<Comp_i128_61>();
    world.register::<Comp_i128_62>();
    world.register::<Comp_i128_63>();
    world.register::<Comp_i128_64>();
    world.register::<Comp_i128_65>();
    world.register::<Comp_i128_66>();
    world.register::<Comp_i128_67>();
    world.register::<Comp_i128_68>();
    world.register::<Comp_i128_69>();
    world.register::<Comp_i128_70>();
    world.register::<Comp_i128_71>();
    world.register::<Comp_i128_72>();
    world.register::<Comp_i128_73>();
    world.register::<Comp_i128_74>();
    world.register::<Comp_i128_75>();
    world.register::<Comp_i128_76>();
    world.register::<Comp_i128_77>();
    world.register::<Comp_i128_78>();
    world.register::<Comp_i128_79>();
    world.register::<Comp_i128_80>();
    world.register::<Comp_i128_81>();
    world.register::<Comp_i128_82>();
    world.register::<Comp_i128_83>();
    world.register::<Comp_i128_84>();
    world.register::<Comp_i128_85>();
    world.register::<Comp_i128_86>();
    world.register::<Comp_i128_87>();
    world.register::<Comp_i128_88>();
    world.register::<Comp_i128_89>();
    world.register::<Comp_i128_90>();
    world.register::<Comp_i128_91>();
    world.register::<Comp_i128_92>();
    world.register::<Comp_i128_93>();
    world.register::<Comp_i128_94>();
    world.register::<Comp_i128_95>();
    world.register::<Comp_i128_96>();
    world.register::<Comp_i128_97>();
    world.register::<Comp_i128_98>();
    world.register::<Comp_i128_99>();
    world.register::<Comp_i128_100>();
    world.register::<Comp_i128_101>();
    world.register::<Comp_i128_102>();
    world.register::<Comp_i128_103>();
    world.register::<Comp_i128_104>();
    world.register::<Comp_i128_105>();
    world.register::<Comp_i128_106>();
    world.register::<Comp_i128_107>();
    world.register::<Comp_i128_108>();
    world.register::<Comp_i128_109>();
    world.register::<Comp_i128_110>();
    world.register::<Comp_i128_111>();
    world.register::<Comp_i128_112>();
    world.register::<Comp_i128_113>();
    world.register::<Comp_i128_114>();
    world.register::<Comp_i128_115>();
    world.register::<Comp_i128_116>();
    world.register::<Comp_i128_117>();
    world.register::<Comp_i128_118>();
    world.register::<Comp_i128_119>();
    world.register::<Comp_i128_120>();
    world.register::<Comp_i128_121>();
    world.register::<Comp_i128_122>();
    world.register::<Comp_i128_123>();
    world.register::<Comp_i128_124>();
    world.register::<Comp_i128_125>();
    world.register::<Comp_i128_126>();
    world.register::<Comp_i128_127>();

    return Ok(())
}

//Add entities to the world
pub fn setup_entity(entity_size: i32, world: &mut World)->io::Result<()> {
    match entity_size {
        128 => {
            for _ in 0..5000 {
                world.create_entity()
                    .with(Comp_i64_0(criterion::black_box(5)))
                    .with(Comp_i64_1(criterion::black_box(5)))
                    .build();
            }
        }
        256 => {
            for _ in 0..5000 {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .build();
            }
        }

        512 => {
            for _ in 0..5000 {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .build();
            }
        }

        1024 => {
            for _ in 0..5000 {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .build();
            }
        }

        2048 => {
            for _ in 0..5000 {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .with(Comp_i128_8(criterion::black_box(5)))
                    .with(Comp_i128_9(criterion::black_box(5)))
                    .with(Comp_i128_10(criterion::black_box(5)))
                    .with(Comp_i128_11(criterion::black_box(5)))
                    .with(Comp_i128_12(criterion::black_box(5)))
                    .with(Comp_i128_13(criterion::black_box(5)))
                    .with(Comp_i128_14(criterion::black_box(5)))
                    .with(Comp_i128_15(criterion::black_box(5)))
                    .build();
            }
        }

        4096 => {
            for _ in 0..5000 {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .with(Comp_i128_8(criterion::black_box(5)))
                    .with(Comp_i128_9(criterion::black_box(5)))
                    .with(Comp_i128_10(criterion::black_box(5)))
                    .with(Comp_i128_11(criterion::black_box(5)))
                    .with(Comp_i128_12(criterion::black_box(5)))
                    .with(Comp_i128_13(criterion::black_box(5)))
                    .with(Comp_i128_14(criterion::black_box(5)))
                    .with(Comp_i128_15(criterion::black_box(5)))
                    .with(Comp_i128_16(criterion::black_box(5)))
                    .with(Comp_i128_17(criterion::black_box(5)))
                    .with(Comp_i128_18(criterion::black_box(5)))
                    .with(Comp_i128_19(criterion::black_box(5)))
                    .with(Comp_i128_20(criterion::black_box(5)))
                    .with(Comp_i128_21(criterion::black_box(5)))
                    .with(Comp_i128_22(criterion::black_box(5)))
                    .with(Comp_i128_23(criterion::black_box(5)))
                    .with(Comp_i128_24(criterion::black_box(5)))
                    .with(Comp_i128_25(criterion::black_box(5)))
                    .with(Comp_i128_26(criterion::black_box(5)))
                    .with(Comp_i128_27(criterion::black_box(5)))
                    .with(Comp_i128_28(criterion::black_box(5)))
                    .with(Comp_i128_29(criterion::black_box(5)))
                    .with(Comp_i128_30(criterion::black_box(5)))
                    .with(Comp_i128_31(criterion::black_box(5)))
                    .build();
            }

        }

        8192 => {
            for _ in 0..5000 {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .with(Comp_i128_8(criterion::black_box(5)))
                    .with(Comp_i128_9(criterion::black_box(5)))
```

```
                        .with(Comp_i128_10(criterion::black_box(5)))
                        .with(Comp_i128_11(criterion::black_box(5)))
                        .with(Comp_i128_12(criterion::black_box(5)))
                        .with(Comp_i128_13(criterion::black_box(5)))
                        .with(Comp_i128_14(criterion::black_box(5)))
                        .with(Comp_i128_15(criterion::black_box(5)))
                        .with(Comp_i128_16(criterion::black_box(5)))
                        .with(Comp_i128_17(criterion::black_box(5)))
                        .with(Comp_i128_18(criterion::black_box(5)))
                        .with(Comp_i128_19(criterion::black_box(5)))
                        .with(Comp_i128_20(criterion::black_box(5)))
                        .with(Comp_i128_21(criterion::black_box(5)))
                        .with(Comp_i128_22(criterion::black_box(5)))
                        .with(Comp_i128_23(criterion::black_box(5)))
                        .with(Comp_i128_24(criterion::black_box(5)))
                        .with(Comp_i128_25(criterion::black_box(5)))
                        .with(Comp_i128_26(criterion::black_box(5)))
                        .with(Comp_i128_27(criterion::black_box(5)))
                        .with(Comp_i128_28(criterion::black_box(5)))
                        .with(Comp_i128_29(criterion::black_box(5)))
                        .with(Comp_i128_30(criterion::black_box(5)))
                        .with(Comp_i128_31(criterion::black_box(5)))
                        .with(Comp_i128_32(criterion::black_box(5)))
                        .with(Comp_i128_33(criterion::black_box(5)))
                        .with(Comp_i128_34(criterion::black_box(5)))
                        .with(Comp_i128_35(criterion::black_box(5)))
                        .with(Comp_i128_36(criterion::black_box(5)))
                        .with(Comp_i128_37(criterion::black_box(5)))
```

```
                    .with(Comp_i128_38(criterion::black_box(5)))
                    .with(Comp_i128_39(criterion::black_box(5)))
                    .with(Comp_i128_40(criterion::black_box(5)))
                    .with(Comp_i128_41(criterion::black_box(5)))
                    .with(Comp_i128_42(criterion::black_box(5)))
                    .with(Comp_i128_43(criterion::black_box(5)))
                    .with(Comp_i128_44(criterion::black_box(5)))
                    .with(Comp_i128_45(criterion::black_box(5)))
                    .with(Comp_i128_46(criterion::black_box(5)))
                    .with(Comp_i128_47(criterion::black_box(5)))
                    .with(Comp_i128_48(criterion::black_box(5)))
                    .with(Comp_i128_49(criterion::black_box(5)))
                    .with(Comp_i128_50(criterion::black_box(5)))
                    .with(Comp_i128_51(criterion::black_box(5)))
                    .with(Comp_i128_52(criterion::black_box(5)))
                    .with(Comp_i128_53(criterion::black_box(5)))
                    .with(Comp_i128_54(criterion::black_box(5)))
                    .with(Comp_i128_55(criterion::black_box(5)))
                    .with(Comp_i128_56(criterion::black_box(5)))
                    .with(Comp_i128_57(criterion::black_box(5)))
                    .with(Comp_i128_58(criterion::black_box(5)))
                    .with(Comp_i128_59(criterion::black_box(5)))
                    .with(Comp_i128_60(criterion::black_box(5)))
                    .with(Comp_i128_61(criterion::black_box(5)))
                    .with(Comp_i128_62(criterion::black_box(5)))
                    .with(Comp_i128_63(criterion::black_box(5)))
                    .build();
            }
```

```
        }

        16384 => {
            for _ in 0..5000 {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .with(Comp_i128_8(criterion::black_box(5)))
                    .with(Comp_i128_9(criterion::black_box(5)))
                    .with(Comp_i128_10(criterion::black_box(5)))
                    .with(Comp_i128_11(criterion::black_box(5)))
                    .with(Comp_i128_12(criterion::black_box(5)))
                    .with(Comp_i128_13(criterion::black_box(5)))
                    .with(Comp_i128_14(criterion::black_box(5)))
                    .with(Comp_i128_15(criterion::black_box(5)))
                    .with(Comp_i128_16(criterion::black_box(5)))
                    .with(Comp_i128_17(criterion::black_box(5)))
                    .with(Comp_i128_18(criterion::black_box(5)))
                    .with(Comp_i128_19(criterion::black_box(5)))
                    .with(Comp_i128_20(criterion::black_box(5)))
                    .with(Comp_i128_21(criterion::black_box(5)))
                    .with(Comp_i128_22(criterion::black_box(5)))
```

```
                        .with(Comp_i128_23(criterion::black_box(5)))
                        .with(Comp_i128_24(criterion::black_box(5)))
                        .with(Comp_i128_25(criterion::black_box(5)))
                        .with(Comp_i128_26(criterion::black_box(5)))
                        .with(Comp_i128_27(criterion::black_box(5)))
                        .with(Comp_i128_28(criterion::black_box(5)))
                        .with(Comp_i128_29(criterion::black_box(5)))
                        .with(Comp_i128_30(criterion::black_box(5)))
                        .with(Comp_i128_31(criterion::black_box(5)))
                        .with(Comp_i128_32(criterion::black_box(5)))
                        .with(Comp_i128_33(criterion::black_box(5)))
                        .with(Comp_i128_34(criterion::black_box(5)))
                        .with(Comp_i128_35(criterion::black_box(5)))
                        .with(Comp_i128_36(criterion::black_box(5)))
                        .with(Comp_i128_37(criterion::black_box(5)))
                        .with(Comp_i128_38(criterion::black_box(5)))
                        .with(Comp_i128_39(criterion::black_box(5)))
                        .with(Comp_i128_40(criterion::black_box(5)))
                        .with(Comp_i128_41(criterion::black_box(5)))
                        .with(Comp_i128_42(criterion::black_box(5)))
                        .with(Comp_i128_43(criterion::black_box(5)))
                        .with(Comp_i128_44(criterion::black_box(5)))
                        .with(Comp_i128_45(criterion::black_box(5)))
                        .with(Comp_i128_46(criterion::black_box(5)))
                        .with(Comp_i128_47(criterion::black_box(5)))
                        .with(Comp_i128_48(criterion::black_box(5)))
                        .with(Comp_i128_49(criterion::black_box(5)))
                        .with(Comp_i128_50(criterion::black_box(5)))
```

```
                    .with(Comp_i128_51(criterion::black_box(5)))
                    .with(Comp_i128_52(criterion::black_box(5)))
                    .with(Comp_i128_53(criterion::black_box(5)))
                    .with(Comp_i128_54(criterion::black_box(5)))
                    .with(Comp_i128_55(criterion::black_box(5)))
                    .with(Comp_i128_56(criterion::black_box(5)))
                    .with(Comp_i128_57(criterion::black_box(5)))
                    .with(Comp_i128_58(criterion::black_box(5)))
                    .with(Comp_i128_59(criterion::black_box(5)))
                    .with(Comp_i128_60(criterion::black_box(5)))
                    .with(Comp_i128_61(criterion::black_box(5)))
                    .with(Comp_i128_62(criterion::black_box(5)))
                    .with(Comp_i128_63(criterion::black_box(5)))
                    .with(Comp_i128_64(criterion::black_box(5)))
                    .with(Comp_i128_65(criterion::black_box(5)))
                    .with(Comp_i128_66(criterion::black_box(5)))
                    .with(Comp_i128_67(criterion::black_box(5)))
                    .with(Comp_i128_68(criterion::black_box(5)))
                    .with(Comp_i128_69(criterion::black_box(5)))
                    .with(Comp_i128_70(criterion::black_box(5)))
                    .with(Comp_i128_71(criterion::black_box(5)))
                    .with(Comp_i128_72(criterion::black_box(5)))
                    .with(Comp_i128_73(criterion::black_box(5)))
                    .with(Comp_i128_74(criterion::black_box(5)))
                    .with(Comp_i128_75(criterion::black_box(5)))
                    .with(Comp_i128_76(criterion::black_box(5)))
                    .with(Comp_i128_77(criterion::black_box(5)))
                    .with(Comp_i128_78(criterion::black_box(5)))
```

```
                        .with(Comp_i128_79(criterion::black_box(5)))
                        .with(Comp_i128_80(criterion::black_box(5)))
                        .with(Comp_i128_81(criterion::black_box(5)))
                        .with(Comp_i128_82(criterion::black_box(5)))
                        .with(Comp_i128_83(criterion::black_box(5)))
                        .with(Comp_i128_84(criterion::black_box(5)))
                        .with(Comp_i128_85(criterion::black_box(5)))
                        .with(Comp_i128_86(criterion::black_box(5)))
                        .with(Comp_i128_87(criterion::black_box(5)))
                        .with(Comp_i128_88(criterion::black_box(5)))
                        .with(Comp_i128_89(criterion::black_box(5)))
                        .with(Comp_i128_90(criterion::black_box(5)))
                        .with(Comp_i128_91(criterion::black_box(5)))
                        .with(Comp_i128_92(criterion::black_box(5)))
                        .with(Comp_i128_93(criterion::black_box(5)))
                        .with(Comp_i128_94(criterion::black_box(5)))
                        .with(Comp_i128_95(criterion::black_box(5)))
                        .with(Comp_i128_96(criterion::black_box(5)))
                        .with(Comp_i128_97(criterion::black_box(5)))
                        .with(Comp_i128_98(criterion::black_box(5)))
                        .with(Comp_i128_99(criterion::black_box(5)))
                        .with(Comp_i128_100(criterion::black_box(5)))
                        .with(Comp_i128_101(criterion::black_box(5)))
                        .with(Comp_i128_102(criterion::black_box(5)))
                        .with(Comp_i128_103(criterion::black_box(5)))
                        .with(Comp_i128_104(criterion::black_box(5)))
                        .with(Comp_i128_105(criterion::black_box(5)))
                        .with(Comp_i128_106(criterion::black_box(5)))
```

```
                    .with(Comp_i128_107(criterion::black_box(5)))
                    .with(Comp_i128_108(criterion::black_box(5)))
                    .with(Comp_i128_109(criterion::black_box(5)))
                    .with(Comp_i128_110(criterion::black_box(5)))
                    .with(Comp_i128_111(criterion::black_box(5)))
                    .with(Comp_i128_112(criterion::black_box(5)))
                    .with(Comp_i128_113(criterion::black_box(5)))
                    .with(Comp_i128_114(criterion::black_box(5)))
                    .with(Comp_i128_115(criterion::black_box(5)))
                    .with(Comp_i128_116(criterion::black_box(5)))
                    .with(Comp_i128_117(criterion::black_box(5)))
                    .with(Comp_i128_118(criterion::black_box(5)))
                    .with(Comp_i128_119(criterion::black_box(5)))
                    .with(Comp_i128_120(criterion::black_box(5)))
                    .with(Comp_i128_121(criterion::black_box(5)))
                    .with(Comp_i128_122(criterion::black_box(5)))
                    .with(Comp_i128_123(criterion::black_box(5)))
                    .with(Comp_i128_124(criterion::black_box(5)))
                    .with(Comp_i128_125(criterion::black_box(5)))
                    .with(Comp_i128_126(criterion::black_box(5)))
                    .with(Comp_i128_127(criterion::black_box(5)))
                    .build();
            }
        }
        _ => {}
    }
    return Ok(())
}
```

```

//Add systems to the dispatcher, set up threadcount
pub fn setup_dispatcher<'a, 'b>(size: i32)->Dispatcher<'a, 'b> {

    let pool = Arc::from(rayon::ThreadPoolBuilder::new().num_threads(1).build().
    unwrap());

    match size {
        128 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_128bit_0, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        _ => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_256bit_0, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }
    }
}
```

Listing B.3: Experiment 1: DOD

2.4 DOD Components

```rust
use specs::prelude::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i64_0(pub i64);
impl Component for Comp_i64_0 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i64_1(pub i64);
impl Component for Comp_i64_1 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_0(pub i128);
impl Component for Comp_i128_0 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_1(pub i128);
impl Component for Comp_i128_1 {
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_2(pub i128);
impl Component for Comp_i128_2 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_3(pub i128);
impl Component for Comp_i128_3 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_4(pub i128);
impl Component for Comp_i128_4 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_5(pub i128);
impl Component for Comp_i128_5 {
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_6(pub i128);
impl Component for Comp_i128_6 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_7(pub i128);
impl Component for Comp_i128_7 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_8(pub i128);
impl Component for Comp_i128_8 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_9(pub i128);
impl Component for Comp_i128_9 {
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_10(pub i128);
impl Component for Comp_i128_10 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_11(pub i128);
impl Component for Comp_i128_11 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_12(pub i128);
impl Component for Comp_i128_12 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_13(pub i128);
impl Component for Comp_i128_13 {
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_14(pub i128);
impl Component for Comp_i128_14 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_15(pub i128);
impl Component for Comp_i128_15 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_16(pub i128);
impl Component for Comp_i128_16 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_17(pub i128);
impl Component for Comp_i128_17 {
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_18(pub i128);
impl Component for Comp_i128_18 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_19(pub i128);
impl Component for Comp_i128_19 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_20(pub i128);
impl Component for Comp_i128_20 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_21(pub i128);
impl Component for Comp_i128_21 {
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_22(pub i128);
impl Component for Comp_i128_22 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_23(pub i128);
impl Component for Comp_i128_23 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_24(pub i128);
impl Component for Comp_i128_24 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_25(pub i128);
impl Component for Comp_i128_25 {
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_26(pub i128);
impl Component for Comp_i128_26 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_27(pub i128);
impl Component for Comp_i128_27 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_28(pub i128);
impl Component for Comp_i128_28 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_29(pub i128);
impl Component for Comp_i128_29 {
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_30(pub i128);
impl Component for Comp_i128_30 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_31(pub i128);
impl Component for Comp_i128_31 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_32(pub i128);
impl Component for Comp_i128_32 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_33(pub i128);
impl Component for Comp_i128_33 {
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_34(pub i128);
impl Component for Comp_i128_34 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_35(pub i128);
impl Component for Comp_i128_35 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_36(pub i128);
impl Component for Comp_i128_36 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_37(pub i128);
impl Component for Comp_i128_37 {
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_38(pub i128);
impl Component for Comp_i128_38 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_39(pub i128);
impl Component for Comp_i128_39 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_40(pub i128);
impl Component for Comp_i128_40 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_41(pub i128);
impl Component for Comp_i128_41 {
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_42(pub i128);
impl Component for Comp_i128_42 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_43(pub i128);
impl Component for Comp_i128_43 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_44(pub i128);
impl Component for Comp_i128_44 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_45(pub i128);
impl Component for Comp_i128_45 {
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_46(pub i128);
impl Component for Comp_i128_46 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_47(pub i128);
impl Component for Comp_i128_47 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_48(pub i128);
impl Component for Comp_i128_48 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_49(pub i128);
impl Component for Comp_i128_49 {
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_50(pub i128);
impl Component for Comp_i128_50 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_51(pub i128);
impl Component for Comp_i128_51 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_52(pub i128);
impl Component for Comp_i128_52 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_53(pub i128);
impl Component for Comp_i128_53 {
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_54(pub i128);
impl Component for Comp_i128_54 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_55(pub i128);
impl Component for Comp_i128_55 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_56(pub i128);
impl Component for Comp_i128_56 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_57(pub i128);
impl Component for Comp_i128_57 {
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_58(pub i128);
impl Component for Comp_i128_58 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_59(pub i128);
impl Component for Comp_i128_59 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_60(pub i128);
impl Component for Comp_i128_60 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_61(pub i128);
impl Component for Comp_i128_61 {
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_62(pub i128);
impl Component for Comp_i128_62 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_63(pub i128);
impl Component for Comp_i128_63 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_64(pub i128);
impl Component for Comp_i128_64{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_65(pub i128);
impl Component for Comp_i128_65{
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_66(pub i128);
impl Component for Comp_i128_66{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_67(pub i128);
impl Component for Comp_i128_67{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_68(pub i128);
impl Component for Comp_i128_68{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_69(pub i128);
impl Component for Comp_i128_69{
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_70(pub i128);
impl Component for Comp_i128_70{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_71(pub i128);
impl Component for Comp_i128_71{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_72(pub i128);
impl Component for Comp_i128_72{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_73(pub i128);
impl Component for Comp_i128_73{
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_74(pub i128);
impl Component for Comp_i128_74{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_75(pub i128);
impl Component for Comp_i128_75{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_76(pub i128);
impl Component for Comp_i128_76{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_77(pub i128);
impl Component for Comp_i128_77{
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_78(pub i128);
impl Component for Comp_i128_78{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_79(pub i128);
impl Component for Comp_i128_79{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_80(pub i128);
impl Component for Comp_i128_80{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_81(pub i128);
impl Component for Comp_i128_81{
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_82(pub i128);
impl Component for Comp_i128_82{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_83(pub i128);
impl Component for Comp_i128_83{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_84(pub i128);
impl Component for Comp_i128_84{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_85(pub i128);
impl Component for Comp_i128_85{
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_86(pub i128);
impl Component for Comp_i128_86{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_87(pub i128);
impl Component for Comp_i128_87{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_88(pub i128);
impl Component for Comp_i128_88{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_89(pub i128);
impl Component for Comp_i128_89{
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_90(pub i128);
impl Component for Comp_i128_90{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_91(pub i128);
impl Component for Comp_i128_91{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_92(pub i128);
impl Component for Comp_i128_92{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_93(pub i128);
impl Component for Comp_i128_93{
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_94(pub i128);
impl Component for Comp_i128_94{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_95(pub i128);
impl Component for Comp_i128_95{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_96(pub i128);
impl Component for Comp_i128_96{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_97(pub i128);
impl Component for Comp_i128_97{
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_98(pub i128);
impl Component for Comp_i128_98{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_99(pub i128);
impl Component for Comp_i128_99{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_100(pub i128);
impl Component for Comp_i128_100{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_101(pub i128);
impl Component for Comp_i128_101{
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_102(pub i128);
impl Component for Comp_i128_102{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_103(pub i128);
impl Component for Comp_i128_103{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_104(pub i128);
impl Component for Comp_i128_104{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_105(pub i128);
impl Component for Comp_i128_105{
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_106(pub i128);
impl Component for Comp_i128_106{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_107(pub i128);
impl Component for Comp_i128_107{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_108(pub i128);
impl Component for Comp_i128_108{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_109(pub i128);
impl Component for Comp_i128_109{
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_110(pub i128);
impl Component for Comp_i128_110{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_111(pub i128);
impl Component for Comp_i128_111{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_112(pub i128);
impl Component for Comp_i128_112{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_113(pub i128);
impl Component for Comp_i128_113{
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_114(pub i128);
impl Component for Comp_i128_114{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_115(pub i128);
impl Component for Comp_i128_115{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_116(pub i128);
impl Component for Comp_i128_116{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_117(pub i128);
impl Component for Comp_i128_117{
    type Storage = DenseVecStorage<Self>;
```

```rust
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_118(pub i128);
impl Component for Comp_i128_118{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_119(pub i128);
impl Component for Comp_i128_119{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_120(pub i128);
impl Component for Comp_i128_120{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_121(pub i128);
impl Component for Comp_i128_121{
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_122(pub i128);
impl Component for Comp_i128_122{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_123(pub i128);
impl Component for Comp_i128_123{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_124(pub i128);
impl Component for Comp_i128_124{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_125(pub i128);
impl Component for Comp_i128_125{
    type Storage = DenseVecStorage<Self>;
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_126(pub i128);
impl Component for Comp_i128_126{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_127(pub i128);
impl Component for Comp_i128_127{
    type Storage = DenseVecStorage<Self>;
}
```

Listing B.4: Experiment 1: DOD Components

2.5 DOD Systems

```
use specs::prelude::*;
use specs::Join;
use super::dod_component::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_128bit_0;
impl<'a> System<'a> for Sys_128bit_0 {

    type SystemData = (WriteStorage<'a, Comp_i64_0>, ReadStorage<'a, Comp_i64_1
    >);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_0;
impl<'a> System<'a> for Sys_256bit_0 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
    Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}
```

Listing B.5: Experiment 1: DOD Systems

2.6 OOP

```
use rayon::iter::IntoParallelRefMutIterator;
use rayon::*;
```

```rust
use std::sync::{Arc, RwLock};
use super::oop_obj::*;
pub type ThreadPoolWrapper = Option<::std::sync::Arc<::rayon::ThreadPool>>;

//Responsible for spawning the objects for the experiment
pub fn obj_setup<T: Exp1>()-> Vec<T> {

    let mut vec: Vec<T> = Vec::new();
    for _ in 0..5000 {
        let tmp = T::new(criterion::black_box(5));
        vec.push(tmp);
    }

    return vec;
}

//Struct to imitate the World in ECS Architecture
//Stages are set up to match that of the ECS World
pub struct OOPWorld<T: Exp1> {
    stages: Vec<Stage<T>>,
    pool: Arc<RwLock<ThreadPoolWrapper>>
}
impl <T: Exp1> OOPWorld <T> {
    pub fn new(vec: Vec<T>, thread_count: usize)->OOPWorld<T>{
        let pool: ThreadPoolWrapper = Some(Arc::from(
            ThreadPoolBuilder::new().num_threads(thread_count).build().unwrap())
    );
        let pool: Arc<RwLock<ThreadPoolWrapper>> = Arc::from(RwLock::from(pool))
    ;

        let stage: Stage<T> = Stage::new(vec);
        let mut stages: Vec<Stage<T>> = Vec::new();
        stages.push(stage);

        return OOPWorld{
            stages,
            pool
        };
    }

    //Executes all methods in the same manner as the ECS Architecture
    pub fn execute(&mut self){
        let stages = &mut self.stages;
        self.pool
            .read()
            .unwrap()
            .as_ref()
            .unwrap()
            .install(move || {
                for stage in stages {
                    stage.execute();
                }
            });
    }
}

```

```
//Struct to imitate the Stage in ECS Architecture
struct Stage<T: Exp1> {
    groups: Vec<Vec<T>>
}
impl <T: Exp1> Stage <T> {
    fn new(vec: Vec<T>)-> Stage<T> {

        let mut groups: Vec<Vec<T>> = Vec::new();
        groups.push(vec);

        return Stage {
            groups
        };
    }
    fn execute(&mut self) {
        use rayon::iter::ParallelIterator;
        self.groups.par_iter_mut().for_each(|group| {
            for obj in group {
                obj.run();
            }
        })
    }
}
```

Listing B.6: Experiment 1: OOP

2.7 OOP Objects

```
pub trait Exp1: Send {
    fn run(&mut self);
    fn new(val: i128)->Self;
}

pub struct Obj128(pub i64, pub i64);
impl Exp1 for Obj128 {
    fn run(&mut self) {
        self.0 += self.1;
    }
    fn new(val: i128)->Self {
        let val= val as i64;
        return Obj128(val,val);
    }
}

pub struct Obj256(pub i128, pub i128);
impl Exp1 for Obj256 {
    fn run(&mut self) {
        self.0 += self.1;
    }
    fn new(val: i128)->Self {
        return Obj256(val,val);
    }
}

pub struct Obj512(pub i128, pub i128, pub i128, pub i128);
impl Exp1 for Obj512 {
    fn run(&mut self) {
        self.0 += self.3;
```

```rust
    }
    fn new(val: i128)->Self {
        return Obj512(val,val,val,val);
    }
}

pub struct Obj1024(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);
impl Exp1 for Obj1024 {
    fn run(&mut self) {
        self.0 += self.7;
    }
    fn new(val: i128)->Self{
        return Obj1024(val,val,val,val,val,val,val,val);
    }
}

pub struct Obj2048(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);
impl Exp1 for Obj2048 {
    fn run(&mut self) {self.0 += self.15; }
    fn new(val: i128)->Self {
        return Obj2048(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }
}
```

```

pub struct Obj4096(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128);
impl Exp1 for Obj4096 {
    fn run(&mut self) {self.0 += self.31; }
    fn new(val:i128)-> Self {
        return Obj4096(val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,
                       val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,val);
    }
}

pub struct Obj8192(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);
impl Exp1 for Obj8192 {
    fn run(&mut self) {self.0 += self.63; }
    fn new(val:i128)-> Self {
        return Obj8192(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }
}

pub struct Obj16384(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                     pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                     pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                     pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                        pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);
impl Exp1 for Obj16384 {
    fn run(&mut self) {self.0 += self.127; }
    fn new(val:i128)-> Self {
        return Obj16384(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                        val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }
}
```

Listing B.7: Experiment 1: OOP Objects

Appendix C. Experiment Two Code

3.1 Benchmark

```rust
use criterion::{criterion_group, criterion_main, Criterion};
use std::time::Duration;
use specs::prelude::*;
use thesis_experimentation::exp2::dod::*;
use thesis_experimentation::exp2::oop::*;
use thesis_experimentation::exp2::oop_obj::*;

#[inline]
fn dod_dispatch(d: &mut Dispatcher, mut w: &mut World) {
    d.dispatch_par(&mut w);
}

#[inline]
fn oop_dispatch<T: Exp2>(world: &mut OOPWorld<T>) { world.execute(); }

pub fn dod_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("dod_exp2");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);

    let entity_count = vec![10, 50, 100, 500, 1000, 2000, 3000, 4000, 5000,
    6000, 7000, 8000, 9000, 10000];

    entity_count.iter().for_each(|count|{
        let mut world = World::new();
        setup_component(&mut world).unwrap();
        setup_entity(*count, &mut world).unwrap();
        let mut dispatcher = setup_dispatcher();

        dispatcher.setup(&mut world);

        let mut bench_name = String::from("dod_exp2_entity_count_");
        let i = count.to_string();
        bench_name.push_str(&i);

        group.bench_function(bench_name.as_str(), |b| b.iter(|| dod_dispatch(&
    mut dispatcher, &mut world)));
    });
}

fn oop_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("oop_exp2");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);

    let entity_count = vec![10, 50, 100, 500, 1000, 2000, 3000, 4000, 5000,
    6000, 7000, 8000, 9000, 10000];

    entity_count.iter().for_each(|count|{
        let vec = obj_setup::<Obj1024>(*count);
        let mut world = OOPWorld::new(vec, 1);

        let mut bench_name = String::from("oop_exp2_entity_count_");
        let i = count.to_string();
        bench_name.push_str(&i);

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
     world)));
    });
}

criterion_group!(dod_exp2, dod_criterion_benchmark);
criterion_group!(oop_exp2, oop_criterion_benchmark);
criterion_main!(dod_exp2, oop_exp2);
```

Listing C.1: Experiment 2: Benchmark

3.2 Modules

```
pub mod dod;
pub mod oop;
pub mod oop_obj;
pub mod dod_system;
pub mod dod_component;
```

Listing C.2: Experiment 2: Modules

3.3 DOD

```
use specs::{World, WorldExt, Builder, Dispatcher, DispatcherBuilder};
use std::io;
use super::dod_component::*;
```

```rust
use super::dod_system::*;
use std::sync::Arc;

pub fn setup_component(world: &mut World) -> io::Result<()> {
    world.register::<Comp_i128_0>();
    world.register::<Comp_i128_1>();
    world.register::<Comp_i128_2>();
    world.register::<Comp_i128_3>();
    world.register::<Comp_i128_4>();
    world.register::<Comp_i128_5>();
    world.register::<Comp_i128_6>();
    world.register::<Comp_i128_7>();
    return Ok(());
}

pub fn setup_entity(entity_count: i32, world: &mut World) -> io::Result<()> {
    for _ in 0..entity_count {
        world.create_entity()
            .with(Comp_i128_0(criterion::black_box(5)))
            .with(Comp_i128_1(criterion::black_box(5)))
            .with(Comp_i128_2(criterion::black_box(5)))
            .with(Comp_i128_3(criterion::black_box(5)))
            .with(Comp_i128_4(criterion::black_box(5)))
            .with(Comp_i128_5(criterion::black_box(5)))
            .with(Comp_i128_6(criterion::black_box(5)))
            .with(Comp_i128_7(criterion::black_box(5)))
            .build();
    }

    return Ok(());
}

pub fn setup_dispatcher<'a, 'b>()->Dispatcher<'a, 'b> {

    let pool = Arc::from(rayon::ThreadPoolBuilder::new().num_threads(1).build().
    unwrap());
    let dispatcher = DispatcherBuilder::new()
        .with_pool(pool)
        .with(Sys_256bit_0, "sys", &[])
        .build();

    return dispatcher
}
```

Listing C.3: Experiment 2: DOD

3.4 DOD Components

```
use specs::prelude::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_0(pub i128);
impl Component for Comp_i128_0 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```
#[allow(non_camel_case_types)]
pub struct Comp_i128_1(pub i128);
impl Component for Comp_i128_1 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_2(pub i128);
impl Component for Comp_i128_2 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_3(pub i128);
impl Component for Comp_i128_3 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_4(pub i128);
impl Component for Comp_i128_4 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```rust
#[allow(non_camel_case_types)]
pub struct Comp_i128_5(pub i128);
impl Component for Comp_i128_5 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_6(pub i128);
impl Component for Comp_i128_6 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_7(pub i128);
impl Component for Comp_i128_7 {
    type Storage = DenseVecStorage<Self>;
}
```

Listing C.4: Experiment 2: DOD Components

3.5 DOD Systems

```rust
use specs::prelude::*;
use super::dod_component::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_0;
```

```
impl<'a> System<'a> for Sys_256bit_0 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}
```

Listing C.5: Experiment 2: DOD Systems

3.6 OOP

```
use std::sync::{Arc, RwLock};
use rayon::*;
use rayon::iter::IntoParallelRefMutIterator;
use crate::exp2::oop_obj::*;

type ThreadPoolWrapper = Option<::std::sync::Arc<::rayon::ThreadPool>>;

pub fn obj_setup<T: Exp2>(entity_count: i32)-> Vec<T> {

    let mut vec: Vec<T> = Vec::new();
    for _ in 0..entity_count {
        let tmp = T::new(criterion::black_box(5));
        vec.push(tmp);
    }

    return vec;
}

//----------------------------------------------------------
pub struct OOPWorld<T: Exp2> {
    stages: Vec<Stage<T>>,
    pool: Arc<RwLock<ThreadPoolWrapper>>
}

impl <T: Exp2> OOPWorld <T> {
    pub fn new(vec: Vec<T>, thread_count: usize)->OOPWorld<T>{
        let pool: ThreadPoolWrapper = Some(Arc::from(ThreadPoolBuilder::new().
    num_threads(thread_count).build().unwrap()));
        let pool: Arc<RwLock<ThreadPoolWrapper>> = Arc::from(RwLock::from(pool))
    ;

        let stage: Stage<T> = Stage::new(vec);
        let mut stages: Vec<Stage<T>> = Vec::new();
        stages.push(stage);

        return OOPWorld{
            stages,
            pool
        };
    }

    pub fn execute(&mut self){
        let stages = &mut self.stages;
        self.pool
            .read()
            .unwrap()
            .as_ref()
            .unwrap()
            .install(move || {
                for stage in stages {
                    stage.execute();
                }
            });
    }
}

//----------------------------------------------------------

struct Stage<T: Exp2> {
    groups: Vec<Vec<T>>
}

impl <T: Exp2> Stage <T> {
    fn new(vec: Vec<T>)-> Stage<T> {

        let mut groups: Vec<Vec<T>> = Vec::new();
        groups.push(vec);

        return Stage {
            groups
        };
    }

    fn execute(&mut self) {
        use rayon::iter::ParallelIterator;
        self.groups.par_iter_mut().for_each(|group| {
            for obj in group {
                obj.run();
            }
        })
    }
}
```

Listing C.6: Experiment 2: OOP

3.7 OOP Objects

```
pub trait Exp2: Send {
    fn run(&mut self);
    fn new(val: i128)->Self;
}

pub struct Obj1024(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);
impl Exp2 for Obj1024 {
    fn run(&mut self) {
        self.0 += self.7;
    }
    fn new(val: i128)->Self{
        return Obj1024(val,val,val,val,val,val,val,val);
    }
}
```

Listing C.7: Experiment 2: OOP Objects

Appendix D. Experiment Three Code

4.1 Benchmark

```rust
use criterion::{criterion_group, criterion_main, Criterion};
use std::time::Duration;
use thesis_experimentation::exp3::oop::*;
use thesis_experimentation::exp3::dod::*;
use thesis_experimentation::exp3::oop_obj::*;
use specs::prelude::*;

#[inline]
fn oop_dispatch<T: Exp3>(world: &mut OOPWorld<T>) { world.execute(); }

#[inline]
fn dod_dispatch(d: &mut Dispatcher, mut w: &mut World) {
    d.dispatch_par(&mut w);
}

fn oop_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("oop_exp3");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);
    rayon::ThreadPoolBuilder::new().num_threads(1).build_global().unwrap();

    let entity_count = vec![10, 50, 100, 500, 1000, 2000, 3000, 4000, 5000,
   6000, 7000, 8000, 9000, 10000];

```

```
    entity_count.iter().for_each(|count|{
        let vec = obj_setup::<Obj128>(*count);
        let mut world = OOPWorld::new(vec, 1);

        let mut bench_name = String::from("oop_exp3_count_");
        let i = count.to_string();
        bench_name.push_str(&i);
        bench_name.push_str("_size_128");

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
 world)));
    });

    entity_count.iter().for_each(|count|{
        let vec = obj_setup::<Obj256>(*count);
        let mut world = OOPWorld::new(vec, 1);

        let mut bench_name = String::from("oop_exp3_count_");
        let i = count.to_string();
        bench_name.push_str(&i);
        bench_name.push_str("_size_256");

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
 world)));
    });

    entity_count.iter().for_each(|count|{
        let vec = obj_setup::<Obj512>(*count);
        let mut world = OOPWorld::new(vec, 1);

        let mut bench_name = String::from("oop_exp3_count_");
        let i = count.to_string();
        bench_name.push_str(&i);
        bench_name.push_str("_size_512");

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
     world)));
    });

    entity_count.iter().for_each(|count|{
        let vec = obj_setup::<Obj1024>(*count);
        let mut world = OOPWorld::new(vec, 1);

        let mut bench_name = String::from("oop_exp3_count_");
        let i = count.to_string();
        bench_name.push_str(&i);
        bench_name.push_str("_size_1024");

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
     world)));
    });

    entity_count.iter().for_each(|count|{
        let vec = obj_setup::<Obj2048>(*count);
        let mut world = OOPWorld::new(vec, 1);

        let mut bench_name = String::from("oop_exp3_count_");
        let i = count.to_string();
        bench_name.push_str(&i);
        bench_name.push_str("_size_2048");

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
       world)));
    });

    entity_count.iter().for_each(|count|{
        let vec = obj_setup::<Obj4096>(*count);
        let mut world = OOPWorld::new(vec, 1);

        let mut bench_name = String::from("oop_exp3_count_");
        let i = count.to_string();
        bench_name.push_str(&i);
        bench_name.push_str("_size_4096");

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
       world)));
    });

    entity_count.iter().for_each(|count|{
        let vec = obj_setup::<Obj8192>(*count);
        let mut world = OOPWorld::new(vec, 1);

        let mut bench_name = String::from("oop_exp3_count_");
        let i = count.to_string();
        bench_name.push_str(&i);
        bench_name.push_str("_size_8192");

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
     world)));
    });

    entity_count.iter().for_each(|count|{
        let vec = obj_setup::<Obj16384>(*count);
        let mut world = OOPWorld::new(vec, 1);

        let mut bench_name = String::from("oop_exp3_count_");
        let i = count.to_string();
        bench_name.push_str(&i);
        bench_name.push_str("_size_16384");

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
     world)));
    });
}

pub fn dod_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("dod_exp3");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);

    let entity_size: Vec<i32> = vec![128,256,512,1024,2048,4096,8192,16384];
    let entity_count: Vec<i32> = vec![10, 50, 100, 500, 1000, 2000, 3000, 4000,
   5000, 6000, 7000, 8000, 9000, 10000];

    entity_size.iter().for_each(|size| {
        entity_count.iter().for_each(|count| {
            let mut world = World::new();
            setup_component(&mut world).unwrap();
            setup_entity(*size, *count, &mut world).unwrap();
            let mut dispatcher = setup_dispatcher(*size);
            let mut bench_name = String::from("dod_exp3_count_");
            let i = count.to_string();
            bench_name.push_str(&i);
            bench_name.push_str("_size_");
            let i = size.to_string();
            bench_name.push_str(&i);

            group.bench_function(bench_name, |b| b.iter( || dod_dispatch(&mut
   dispatcher, &mut world)));
        })
    });
}

criterion_group!(oop_exp3, oop_criterion_benchmark);
criterion_group!(dod_exp3, dod_criterion_benchmark);
criterion_main!(oop_exp3, dod_exp3);
```

Listing D.1: Experiment 3: Benchmark

4.2 Modules

```
pub mod oop;
pub mod dod;
pub mod oop_obj;
pub mod dod_component;
pub mod dod_system;
```

Listing D.2: Experiment 3: Modules

4.3 DOD

```
use specs::prelude::*;
use std::io;
use super::dod_component::*;
use super::dod_system::*;
use std::sync::Arc;

//Add components to the world
pub fn setup_component(world: &mut World)-> io::Result<()> {
    world.register::<Comp_i64_0>();
    world.register::<Comp_i64_1>();
    world.register::<Comp_i128_0>();
    world.register::<Comp_i128_1>();
    world.register::<Comp_i128_2>();
    world.register::<Comp_i128_3>();
    world.register::<Comp_i128_4>();
    world.register::<Comp_i128_5>();
    world.register::<Comp_i128_6>();
    world.register::<Comp_i128_7>();
    world.register::<Comp_i128_8>();
    world.register::<Comp_i128_9>();
    world.register::<Comp_i128_10>();
    world.register::<Comp_i128_11>();
    world.register::<Comp_i128_12>();
    world.register::<Comp_i128_13>();
    world.register::<Comp_i128_14>();
    world.register::<Comp_i128_15>();
    world.register::<Comp_i128_16>();
    world.register::<Comp_i128_17>();
    world.register::<Comp_i128_18>();
    world.register::<Comp_i128_19>();
    world.register::<Comp_i128_20>();
    world.register::<Comp_i128_21>();
    world.register::<Comp_i128_22>();
    world.register::<Comp_i128_23>();
    world.register::<Comp_i128_24>();
    world.register::<Comp_i128_25>();
    world.register::<Comp_i128_26>();
    world.register::<Comp_i128_27>();
    world.register::<Comp_i128_28>();
    world.register::<Comp_i128_29>();
    world.register::<Comp_i128_30>();
    world.register::<Comp_i128_31>();
    world.register::<Comp_i128_32>();
    world.register::<Comp_i128_33>();
    world.register::<Comp_i128_34>();
    world.register::<Comp_i128_35>();
    world.register::<Comp_i128_36>();
    world.register::<Comp_i128_37>();
    world.register::<Comp_i128_38>();
    world.register::<Comp_i128_39>();
    world.register::<Comp_i128_40>();
    world.register::<Comp_i128_41>();
    world.register::<Comp_i128_42>();
    world.register::<Comp_i128_43>();
    world.register::<Comp_i128_44>();
    world.register::<Comp_i128_45>();
    world.register::<Comp_i128_46>();
    world.register::<Comp_i128_47>();
    world.register::<Comp_i128_48>();
    world.register::<Comp_i128_49>();
    world.register::<Comp_i128_50>();
    world.register::<Comp_i128_51>();
    world.register::<Comp_i128_52>();
    world.register::<Comp_i128_53>();
    world.register::<Comp_i128_54>();
    world.register::<Comp_i128_55>();
    world.register::<Comp_i128_56>();
    world.register::<Comp_i128_57>();
    world.register::<Comp_i128_58>();
    world.register::<Comp_i128_59>();
    world.register::<Comp_i128_60>();
    world.register::<Comp_i128_61>();
    world.register::<Comp_i128_62>();
    world.register::<Comp_i128_63>();
    world.register::<Comp_i128_64>();
    world.register::<Comp_i128_65>();
    world.register::<Comp_i128_66>();
    world.register::<Comp_i128_67>();
    world.register::<Comp_i128_68>();
    world.register::<Comp_i128_69>();
    world.register::<Comp_i128_70>();
    world.register::<Comp_i128_71>();
    world.register::<Comp_i128_72>();
    world.register::<Comp_i128_73>();
    world.register::<Comp_i128_74>();
    world.register::<Comp_i128_75>();
    world.register::<Comp_i128_76>();
    world.register::<Comp_i128_77>();
    world.register::<Comp_i128_78>();
    world.register::<Comp_i128_79>();
    world.register::<Comp_i128_80>();
    world.register::<Comp_i128_81>();
    world.register::<Comp_i128_82>();
    world.register::<Comp_i128_83>();
    world.register::<Comp_i128_84>();
    world.register::<Comp_i128_85>();
    world.register::<Comp_i128_86>();
    world.register::<Comp_i128_87>();
    world.register::<Comp_i128_88>();
    world.register::<Comp_i128_89>();
    world.register::<Comp_i128_90>();
    world.register::<Comp_i128_91>();
    world.register::<Comp_i128_92>();
    world.register::<Comp_i128_93>();
    world.register::<Comp_i128_94>();
    world.register::<Comp_i128_95>();
    world.register::<Comp_i128_96>();
    world.register::<Comp_i128_97>();
    world.register::<Comp_i128_98>();
    world.register::<Comp_i128_99>();
    world.register::<Comp_i128_100>();
    world.register::<Comp_i128_101>();
    world.register::<Comp_i128_102>();
    world.register::<Comp_i128_103>();
    world.register::<Comp_i128_104>();
    world.register::<Comp_i128_105>();
    world.register::<Comp_i128_106>();
    world.register::<Comp_i128_107>();
    world.register::<Comp_i128_108>();
    world.register::<Comp_i128_109>();
    world.register::<Comp_i128_110>();
    world.register::<Comp_i128_111>();
    world.register::<Comp_i128_112>();
    world.register::<Comp_i128_113>();
    world.register::<Comp_i128_114>();
    world.register::<Comp_i128_115>();
    world.register::<Comp_i128_116>();
    world.register::<Comp_i128_117>();
    world.register::<Comp_i128_118>();
    world.register::<Comp_i128_119>();
    world.register::<Comp_i128_120>();
    world.register::<Comp_i128_121>();
```

```rust
    world.register::<Comp_i128_122>();
    world.register::<Comp_i128_123>();
    world.register::<Comp_i128_124>();
    world.register::<Comp_i128_125>();
    world.register::<Comp_i128_126>();
    world.register::<Comp_i128_127>();

    return Ok(())
}

//Add entities to the world
pub fn setup_entity(entity_size: i32, entity_count: i32, world: &mut World)->io
    ::Result<()> {

    match entity_size {
        128 => {
            for _ in 0..entity_count {
                world.create_entity()
                    .with(Comp_i64_0(criterion::black_box(5)))
                    .with(Comp_i64_1(criterion::black_box(5)))
                    .build();
            }
        }
        256 => {
            for _ in 0..entity_count {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .build();
            }
        }

        512 => {
            for _ in 0..entity_count {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .build();
            }
        }

        1024 => {
            for _ in 0..entity_count {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .build();
            }
        }

        2048 => {
            for _ in 0..entity_count {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .with(Comp_i128_8(criterion::black_box(5)))
                    .with(Comp_i128_9(criterion::black_box(5)))
                    .with(Comp_i128_10(criterion::black_box(5)))
                    .with(Comp_i128_11(criterion::black_box(5)))
                    .with(Comp_i128_12(criterion::black_box(5)))
                    .with(Comp_i128_13(criterion::black_box(5)))
                    .with(Comp_i128_14(criterion::black_box(5)))
                    .with(Comp_i128_15(criterion::black_box(5)))
                    .build();
            }
        }

        4096 => {
            for _ in 0..entity_count {
                world.create_entity()
```

```
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .with(Comp_i128_8(criterion::black_box(5)))
                    .with(Comp_i128_9(criterion::black_box(5)))
                    .with(Comp_i128_10(criterion::black_box(5)))
                    .with(Comp_i128_11(criterion::black_box(5)))
                    .with(Comp_i128_12(criterion::black_box(5)))
                    .with(Comp_i128_13(criterion::black_box(5)))
                    .with(Comp_i128_14(criterion::black_box(5)))
                    .with(Comp_i128_15(criterion::black_box(5)))
                    .with(Comp_i128_16(criterion::black_box(5)))
                    .with(Comp_i128_17(criterion::black_box(5)))
                    .with(Comp_i128_18(criterion::black_box(5)))
                    .with(Comp_i128_19(criterion::black_box(5)))
                    .with(Comp_i128_20(criterion::black_box(5)))
                    .with(Comp_i128_21(criterion::black_box(5)))
                    .with(Comp_i128_22(criterion::black_box(5)))
                    .with(Comp_i128_23(criterion::black_box(5)))
                    .with(Comp_i128_24(criterion::black_box(5)))
                    .with(Comp_i128_25(criterion::black_box(5)))
                    .with(Comp_i128_26(criterion::black_box(5)))
                    .with(Comp_i128_27(criterion::black_box(5)))
                    .with(Comp_i128_28(criterion::black_box(5)))
                    .with(Comp_i128_29(criterion::black_box(5)))
                    .with(Comp_i128_30(criterion::black_box(5)))
                    .with(Comp_i128_31(criterion::black_box(5)))
                    .build();
            }

        }

        8192 => {
            for _ in 0..entity_count {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .with(Comp_i128_8(criterion::black_box(5)))
                    .with(Comp_i128_9(criterion::black_box(5)))
                    .with(Comp_i128_10(criterion::black_box(5)))
                    .with(Comp_i128_11(criterion::black_box(5)))
                    .with(Comp_i128_12(criterion::black_box(5)))
                    .with(Comp_i128_13(criterion::black_box(5)))
                    .with(Comp_i128_14(criterion::black_box(5)))
                    .with(Comp_i128_15(criterion::black_box(5)))
```

```
                        .with(Comp_i128_16(criterion::black_box(5)))
                        .with(Comp_i128_17(criterion::black_box(5)))
                        .with(Comp_i128_18(criterion::black_box(5)))
                        .with(Comp_i128_19(criterion::black_box(5)))
                        .with(Comp_i128_20(criterion::black_box(5)))
                        .with(Comp_i128_21(criterion::black_box(5)))
                        .with(Comp_i128_22(criterion::black_box(5)))
                        .with(Comp_i128_23(criterion::black_box(5)))
                        .with(Comp_i128_24(criterion::black_box(5)))
                        .with(Comp_i128_25(criterion::black_box(5)))
                        .with(Comp_i128_26(criterion::black_box(5)))
                        .with(Comp_i128_27(criterion::black_box(5)))
                        .with(Comp_i128_28(criterion::black_box(5)))
                        .with(Comp_i128_29(criterion::black_box(5)))
                        .with(Comp_i128_30(criterion::black_box(5)))
                        .with(Comp_i128_31(criterion::black_box(5)))
                        .with(Comp_i128_32(criterion::black_box(5)))
                        .with(Comp_i128_33(criterion::black_box(5)))
                        .with(Comp_i128_34(criterion::black_box(5)))
                        .with(Comp_i128_35(criterion::black_box(5)))
                        .with(Comp_i128_36(criterion::black_box(5)))
                        .with(Comp_i128_37(criterion::black_box(5)))
                        .with(Comp_i128_38(criterion::black_box(5)))
                        .with(Comp_i128_39(criterion::black_box(5)))
                        .with(Comp_i128_40(criterion::black_box(5)))
                        .with(Comp_i128_41(criterion::black_box(5)))
                        .with(Comp_i128_42(criterion::black_box(5)))
                        .with(Comp_i128_43(criterion::black_box(5)))
```

```
                    .with(Comp_i128_44(criterion::black_box(5)))
                    .with(Comp_i128_45(criterion::black_box(5)))
                    .with(Comp_i128_46(criterion::black_box(5)))
                    .with(Comp_i128_47(criterion::black_box(5)))
                    .with(Comp_i128_48(criterion::black_box(5)))
                    .with(Comp_i128_49(criterion::black_box(5)))
                    .with(Comp_i128_50(criterion::black_box(5)))
                    .with(Comp_i128_51(criterion::black_box(5)))
                    .with(Comp_i128_52(criterion::black_box(5)))
                    .with(Comp_i128_53(criterion::black_box(5)))
                    .with(Comp_i128_54(criterion::black_box(5)))
                    .with(Comp_i128_55(criterion::black_box(5)))
                    .with(Comp_i128_56(criterion::black_box(5)))
                    .with(Comp_i128_57(criterion::black_box(5)))
                    .with(Comp_i128_58(criterion::black_box(5)))
                    .with(Comp_i128_59(criterion::black_box(5)))
                    .with(Comp_i128_60(criterion::black_box(5)))
                    .with(Comp_i128_61(criterion::black_box(5)))
                    .with(Comp_i128_62(criterion::black_box(5)))
                    .with(Comp_i128_63(criterion::black_box(5)))
                    .build();
            }
        }

        16384 => {
            for _ in 0..entity_count {
                world.create_entity()
                    .with(Comp_i128_0(criterion::black_box(5)))
                    .with(Comp_i128_1(criterion::black_box(5)))
                    .with(Comp_i128_2(criterion::black_box(5)))
                    .with(Comp_i128_3(criterion::black_box(5)))
                    .with(Comp_i128_4(criterion::black_box(5)))
                    .with(Comp_i128_5(criterion::black_box(5)))
                    .with(Comp_i128_6(criterion::black_box(5)))
                    .with(Comp_i128_7(criterion::black_box(5)))
                    .with(Comp_i128_8(criterion::black_box(5)))
                    .with(Comp_i128_9(criterion::black_box(5)))
                    .with(Comp_i128_10(criterion::black_box(5)))
                    .with(Comp_i128_11(criterion::black_box(5)))
                    .with(Comp_i128_12(criterion::black_box(5)))
                    .with(Comp_i128_13(criterion::black_box(5)))
                    .with(Comp_i128_14(criterion::black_box(5)))
                    .with(Comp_i128_15(criterion::black_box(5)))
                    .with(Comp_i128_16(criterion::black_box(5)))
                    .with(Comp_i128_17(criterion::black_box(5)))
                    .with(Comp_i128_18(criterion::black_box(5)))
                    .with(Comp_i128_19(criterion::black_box(5)))
                    .with(Comp_i128_20(criterion::black_box(5)))
                    .with(Comp_i128_21(criterion::black_box(5)))
                    .with(Comp_i128_22(criterion::black_box(5)))
                    .with(Comp_i128_23(criterion::black_box(5)))
                    .with(Comp_i128_24(criterion::black_box(5)))
                    .with(Comp_i128_25(criterion::black_box(5)))
                    .with(Comp_i128_26(criterion::black_box(5)))
                    .with(Comp_i128_27(criterion::black_box(5)))
                    .with(Comp_i128_28(criterion::black_box(5)))
```

```
                        .with(Comp_i128_29(criterion::black_box(5)))
                        .with(Comp_i128_30(criterion::black_box(5)))
                        .with(Comp_i128_31(criterion::black_box(5)))
                        .with(Comp_i128_32(criterion::black_box(5)))
                        .with(Comp_i128_33(criterion::black_box(5)))
                        .with(Comp_i128_34(criterion::black_box(5)))
                        .with(Comp_i128_35(criterion::black_box(5)))
                        .with(Comp_i128_36(criterion::black_box(5)))
                        .with(Comp_i128_37(criterion::black_box(5)))
                        .with(Comp_i128_38(criterion::black_box(5)))
                        .with(Comp_i128_39(criterion::black_box(5)))
                        .with(Comp_i128_40(criterion::black_box(5)))
                        .with(Comp_i128_41(criterion::black_box(5)))
                        .with(Comp_i128_42(criterion::black_box(5)))
                        .with(Comp_i128_43(criterion::black_box(5)))
                        .with(Comp_i128_44(criterion::black_box(5)))
                        .with(Comp_i128_45(criterion::black_box(5)))
                        .with(Comp_i128_46(criterion::black_box(5)))
                        .with(Comp_i128_47(criterion::black_box(5)))
                        .with(Comp_i128_48(criterion::black_box(5)))
                        .with(Comp_i128_49(criterion::black_box(5)))
                        .with(Comp_i128_50(criterion::black_box(5)))
                        .with(Comp_i128_51(criterion::black_box(5)))
                        .with(Comp_i128_52(criterion::black_box(5)))
                        .with(Comp_i128_53(criterion::black_box(5)))
                        .with(Comp_i128_54(criterion::black_box(5)))
                        .with(Comp_i128_55(criterion::black_box(5)))
                        .with(Comp_i128_56(criterion::black_box(5)))
                        .with(Comp_i128_57(criterion::black_box(5)))
                        .with(Comp_i128_58(criterion::black_box(5)))
                        .with(Comp_i128_59(criterion::black_box(5)))
                        .with(Comp_i128_60(criterion::black_box(5)))
                        .with(Comp_i128_61(criterion::black_box(5)))
                        .with(Comp_i128_62(criterion::black_box(5)))
                        .with(Comp_i128_63(criterion::black_box(5)))
                        .with(Comp_i128_64(criterion::black_box(5)))
                        .with(Comp_i128_65(criterion::black_box(5)))
                        .with(Comp_i128_66(criterion::black_box(5)))
                        .with(Comp_i128_67(criterion::black_box(5)))
                        .with(Comp_i128_68(criterion::black_box(5)))
                        .with(Comp_i128_69(criterion::black_box(5)))
                        .with(Comp_i128_70(criterion::black_box(5)))
                        .with(Comp_i128_71(criterion::black_box(5)))
                        .with(Comp_i128_72(criterion::black_box(5)))
                        .with(Comp_i128_73(criterion::black_box(5)))
                        .with(Comp_i128_74(criterion::black_box(5)))
                        .with(Comp_i128_75(criterion::black_box(5)))
                        .with(Comp_i128_76(criterion::black_box(5)))
                        .with(Comp_i128_77(criterion::black_box(5)))
                        .with(Comp_i128_78(criterion::black_box(5)))
                        .with(Comp_i128_79(criterion::black_box(5)))
                        .with(Comp_i128_80(criterion::black_box(5)))
                        .with(Comp_i128_81(criterion::black_box(5)))
                        .with(Comp_i128_82(criterion::black_box(5)))
                        .with(Comp_i128_83(criterion::black_box(5)))
                        .with(Comp_i128_84(criterion::black_box(5)))
```

```
                    .with(Comp_i128_85(criterion::black_box(5)))
                    .with(Comp_i128_86(criterion::black_box(5)))
                    .with(Comp_i128_87(criterion::black_box(5)))
                    .with(Comp_i128_88(criterion::black_box(5)))
                    .with(Comp_i128_89(criterion::black_box(5)))
                    .with(Comp_i128_90(criterion::black_box(5)))
                    .with(Comp_i128_91(criterion::black_box(5)))
                    .with(Comp_i128_92(criterion::black_box(5)))
                    .with(Comp_i128_93(criterion::black_box(5)))
                    .with(Comp_i128_94(criterion::black_box(5)))
                    .with(Comp_i128_95(criterion::black_box(5)))
                    .with(Comp_i128_96(criterion::black_box(5)))
                    .with(Comp_i128_97(criterion::black_box(5)))
                    .with(Comp_i128_98(criterion::black_box(5)))
                    .with(Comp_i128_99(criterion::black_box(5)))
                    .with(Comp_i128_100(criterion::black_box(5)))
                    .with(Comp_i128_101(criterion::black_box(5)))
                    .with(Comp_i128_102(criterion::black_box(5)))
                    .with(Comp_i128_103(criterion::black_box(5)))
                    .with(Comp_i128_104(criterion::black_box(5)))
                    .with(Comp_i128_105(criterion::black_box(5)))
                    .with(Comp_i128_106(criterion::black_box(5)))
                    .with(Comp_i128_107(criterion::black_box(5)))
                    .with(Comp_i128_108(criterion::black_box(5)))
                    .with(Comp_i128_109(criterion::black_box(5)))
                    .with(Comp_i128_110(criterion::black_box(5)))
                    .with(Comp_i128_111(criterion::black_box(5)))
                    .with(Comp_i128_112(criterion::black_box(5)))
```

```rust
                    .with(Comp_i128_113(criterion::black_box(5)))
                    .with(Comp_i128_114(criterion::black_box(5)))
                    .with(Comp_i128_115(criterion::black_box(5)))
                    .with(Comp_i128_116(criterion::black_box(5)))
                    .with(Comp_i128_117(criterion::black_box(5)))
                    .with(Comp_i128_118(criterion::black_box(5)))
                    .with(Comp_i128_119(criterion::black_box(5)))
                    .with(Comp_i128_120(criterion::black_box(5)))
                    .with(Comp_i128_121(criterion::black_box(5)))
                    .with(Comp_i128_122(criterion::black_box(5)))
                    .with(Comp_i128_123(criterion::black_box(5)))
                    .with(Comp_i128_124(criterion::black_box(5)))
                    .with(Comp_i128_125(criterion::black_box(5)))
                    .with(Comp_i128_126(criterion::black_box(5)))
                    .with(Comp_i128_127(criterion::black_box(5)))
                    .build();
            }
        }
        _ => {}
    }
    return Ok(())
}

//Add systems to the dispatcher, set up threadcount
pub fn setup_dispatcher<'a, 'b>(size: i32)->Dispatcher<'a, 'b> {

    let pool = Arc::from(rayon::ThreadPoolBuilder::new().num_threads(1).build().
   unwrap());
```

```

    match size {
        128 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_128bit_0, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        _ => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_256bit_0, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }
    }
}
```

Listing D.3: Experiment 3: DOD

4.4 DOD Components

```
use specs::prelude::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i64_0(pub i64);
impl Component for Comp_i64_0 {
```

```rust
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i64_1(pub i64);
impl Component for Comp_i64_1 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_0(pub i128);
impl Component for Comp_i128_0 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_1(pub i128);
impl Component for Comp_i128_1 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_2(pub i128);
impl Component for Comp_i128_2 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_3(pub i128);
impl Component for Comp_i128_3 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_4(pub i128);
impl Component for Comp_i128_4 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_5(pub i128);
impl Component for Comp_i128_5 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_6(pub i128);
impl Component for Comp_i128_6 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_7(pub i128);
impl Component for Comp_i128_7 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_8(pub i128);
impl Component for Comp_i128_8 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_9(pub i128);
impl Component for Comp_i128_9 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_10(pub i128);
impl Component for Comp_i128_10 {
```

```rust
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_11(pub i128);
impl Component for Comp_i128_11 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_12(pub i128);
impl Component for Comp_i128_12 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_13(pub i128);
impl Component for Comp_i128_13 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_14(pub i128);
impl Component for Comp_i128_14 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_15(pub i128);
impl Component for Comp_i128_15 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_16(pub i128);
impl Component for Comp_i128_16 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_17(pub i128);
impl Component for Comp_i128_17 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_18(pub i128);
impl Component for Comp_i128_18 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_19(pub i128);
impl Component for Comp_i128_19 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_20(pub i128);
impl Component for Comp_i128_20 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_21(pub i128);
impl Component for Comp_i128_21 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_22(pub i128);
impl Component for Comp_i128_22 {
```

```rust
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_23(pub i128);
impl Component for Comp_i128_23 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_24(pub i128);
impl Component for Comp_i128_24 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_25(pub i128);
impl Component for Comp_i128_25 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_26(pub i128);
impl Component for Comp_i128_26 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_27(pub i128);
impl Component for Comp_i128_27 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_28(pub i128);
impl Component for Comp_i128_28 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_29(pub i128);
impl Component for Comp_i128_29 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_30(pub i128);
impl Component for Comp_i128_30 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_31(pub i128);
impl Component for Comp_i128_31 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_32(pub i128);
impl Component for Comp_i128_32 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_33(pub i128);
impl Component for Comp_i128_33 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_34(pub i128);
impl Component for Comp_i128_34 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_35(pub i128);
impl Component for Comp_i128_35 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_36(pub i128);
impl Component for Comp_i128_36 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_37(pub i128);
impl Component for Comp_i128_37 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_38(pub i128);
impl Component for Comp_i128_38 {
```

```rust
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_39(pub i128);
impl Component for Comp_i128_39 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_40(pub i128);
impl Component for Comp_i128_40 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_41(pub i128);
impl Component for Comp_i128_41 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_42(pub i128);
impl Component for Comp_i128_42 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_43(pub i128);
impl Component for Comp_i128_43 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_44(pub i128);
impl Component for Comp_i128_44 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_45(pub i128);
impl Component for Comp_i128_45 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_46(pub i128);
impl Component for Comp_i128_46 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_47(pub i128);
impl Component for Comp_i128_47 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_48(pub i128);
impl Component for Comp_i128_48 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_49(pub i128);
impl Component for Comp_i128_49 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_50(pub i128);
impl Component for Comp_i128_50 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_51(pub i128);
impl Component for Comp_i128_51 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_52(pub i128);
impl Component for Comp_i128_52 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_53(pub i128);
impl Component for Comp_i128_53 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_54(pub i128);
impl Component for Comp_i128_54 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_55(pub i128);
impl Component for Comp_i128_55 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_56(pub i128);
impl Component for Comp_i128_56 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_57(pub i128);
impl Component for Comp_i128_57 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_58(pub i128);
impl Component for Comp_i128_58 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_59(pub i128);
impl Component for Comp_i128_59 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_60(pub i128);
impl Component for Comp_i128_60 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_61(pub i128);
impl Component for Comp_i128_61 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_62(pub i128);
impl Component for Comp_i128_62 {
```

```rust
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_63(pub i128);
impl Component for Comp_i128_63 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_64(pub i128);
impl Component for Comp_i128_64{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_65(pub i128);
impl Component for Comp_i128_65{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_66(pub i128);
impl Component for Comp_i128_66{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_67(pub i128);
impl Component for Comp_i128_67{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_68(pub i128);
impl Component for Comp_i128_68{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_69(pub i128);
impl Component for Comp_i128_69{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_70(pub i128);
impl Component for Comp_i128_70{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_71(pub i128);
impl Component for Comp_i128_71{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_72(pub i128);
impl Component for Comp_i128_72{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_73(pub i128);
impl Component for Comp_i128_73{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_74(pub i128);
impl Component for Comp_i128_74{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_75(pub i128);
impl Component for Comp_i128_75{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_76(pub i128);
impl Component for Comp_i128_76{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_77(pub i128);
impl Component for Comp_i128_77{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_78(pub i128);
impl Component for Comp_i128_78{
```

```rust
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_79(pub i128);
impl Component for Comp_i128_79{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_80(pub i128);
impl Component for Comp_i128_80{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_81(pub i128);
impl Component for Comp_i128_81{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_82(pub i128);
impl Component for Comp_i128_82{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_83(pub i128);
impl Component for Comp_i128_83{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_84(pub i128);
impl Component for Comp_i128_84{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_85(pub i128);
impl Component for Comp_i128_85{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_86(pub i128);
impl Component for Comp_i128_86{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_87(pub i128);
impl Component for Comp_i128_87{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_88(pub i128);
impl Component for Comp_i128_88{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_89(pub i128);
impl Component for Comp_i128_89{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_90(pub i128);
impl Component for Comp_i128_90{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_91(pub i128);
impl Component for Comp_i128_91{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_92(pub i128);
impl Component for Comp_i128_92{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_93(pub i128);
impl Component for Comp_i128_93{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_94(pub i128);
impl Component for Comp_i128_94{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_95(pub i128);
impl Component for Comp_i128_95{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_96(pub i128);
impl Component for Comp_i128_96{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_97(pub i128);
impl Component for Comp_i128_97{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_98(pub i128);
impl Component for Comp_i128_98{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_99(pub i128);
impl Component for Comp_i128_99{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_100(pub i128);
impl Component for Comp_i128_100{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_101(pub i128);
impl Component for Comp_i128_101{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_102(pub i128);
impl Component for Comp_i128_102{
```

```rust
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_103(pub i128);
impl Component for Comp_i128_103{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_104(pub i128);
impl Component for Comp_i128_104{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_105(pub i128);
impl Component for Comp_i128_105{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_106(pub i128);
impl Component for Comp_i128_106{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_107(pub i128);
impl Component for Comp_i128_107{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_108(pub i128);
impl Component for Comp_i128_108{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_109(pub i128);
impl Component for Comp_i128_109{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_110(pub i128);
impl Component for Comp_i128_110{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_111(pub i128);
impl Component for Comp_i128_111{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_112(pub i128);
impl Component for Comp_i128_112{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_113(pub i128);
impl Component for Comp_i128_113{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_114(pub i128);
impl Component for Comp_i128_114{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_115(pub i128);
impl Component for Comp_i128_115{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_116(pub i128);
impl Component for Comp_i128_116{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_117(pub i128);
impl Component for Comp_i128_117{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_118(pub i128);
impl Component for Comp_i128_118{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_119(pub i128);
impl Component for Comp_i128_119{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_120(pub i128);
impl Component for Comp_i128_120{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_121(pub i128);
impl Component for Comp_i128_121{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_122(pub i128);
impl Component for Comp_i128_122{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_123(pub i128);
impl Component for Comp_i128_123{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_124(pub i128);
impl Component for Comp_i128_124{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_125(pub i128);
impl Component for Comp_i128_125{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_126(pub i128);
impl Component for Comp_i128_126{
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_127(pub i128);
impl Component for Comp_i128_127{
    type Storage = DenseVecStorage<Self>;
}
```

Listing D.4: Experiment 3: DOD Components

4.5 DOD Systems

```
use specs::prelude::*;
use specs::Join;
use super::dod_component::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_128bit_0;
impl<'a> System<'a> for Sys_128bit_0 {

    type SystemData = (WriteStorage<'a, Comp_i64_0>, ReadStorage<'a, Comp_i64_1
    >);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_0;
impl<'a> System<'a> for Sys_256bit_0 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}
```

Listing D.5: Experiment 3: DOD Systems

4.6 OOP

```
use super::oop_obj::*;
use std::sync::{Arc, RwLock};
use rayon::*;
use rayon::iter::IntoParallelRefMutIterator;

type ThreadPoolWrapper = Option<::std::sync::Arc<::rayon::ThreadPool>>;

pub fn obj_setup<T: Exp3>(entity_count: i32)-> Vec<T> {

    let mut vec: Vec<T> = Vec::new();
    for _ in 0..entity_count {
        let tmp = T::new(criterion::black_box(5));
        vec.push(tmp);
    }

    return vec;
}

//----------------------------------------------------------
pub struct OOPWorld<T: Exp3> {
    stages: Vec<Stage<T>>,
    pool: Arc<RwLock<ThreadPoolWrapper>>
}

impl <T: Exp3> OOPWorld <T> {
    pub fn new(vec: Vec<T>, thread_count: usize)->OOPWorld<T>{
        let pool: ThreadPoolWrapper = Some(Arc::from(ThreadPoolBuilder::new().
   num_threads(thread_count).build().unwrap()));
        let pool: Arc<RwLock<ThreadPoolWrapper>> = Arc::from(RwLock::from(pool))
   ;

        let stage: Stage<T> = Stage::new(vec);
        let mut stages: Vec<Stage<T>> = Vec::new();
        stages.push(stage);

        return OOPWorld{
            stages,
            pool
        };
    }

    pub fn execute(&mut self){
        let stages = &mut self.stages;
        self.pool
            .read()
            .unwrap()
            .as_ref()
            .unwrap()
            .install(move || {
                for stage in stages {
                    stage.execute();
                }
            });
    }
}

//-----------------------------------------------------------

struct Stage<T: Exp3> {
    groups: Vec<Vec<T>>
}

impl <T: Exp3> Stage <T> {
    fn new(vec: Vec<T>)-> Stage<T> {

        let mut groups: Vec<Vec<T>> = Vec::new();
        groups.push(vec);

        return Stage {
            groups
        };
    }

    fn execute(&mut self) {
        use rayon::iter::ParallelIterator;
        self.groups.par_iter_mut().for_each(|group| {
            for obj in group {
                obj.run();
            }
        })
    }
}
```

Listing D.6: Experiment 3: OOP

4.7 OOP Objects

```
pub trait Exp3: Send {
    fn run(&mut self);
    fn new(val: i128)->Self;
}

pub struct Obj128(pub i64, pub i64);
impl Exp3 for Obj128 {
```

```rust
    fn run(&mut self) {
        self.0 += self.1;
    }
    fn new(val: i128)->Self {
        let val= val as i64;
        return Obj128(val,val);
    }
}

pub struct Obj256(pub i128, pub i128);
impl Exp3 for Obj256 {
    fn run(&mut self) {
        self.0 += self.1;
    }
    fn new(val: i128)->Self {
        return Obj256(val,val);
    }
}

pub struct Obj512(pub i128, pub i128, pub i128, pub i128);
impl Exp3 for Obj512 {
    fn run(&mut self) {
        self.0 += self.3;
    }
    fn new(val: i128)->Self {
        return Obj512(val,val,val,val);
    }
}
```

```rust

pub struct Obj1024(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);
impl Exp3 for Obj1024 {
    fn run(&mut self) {
        self.0 += self.7;
    }
    fn new(val: i128)->Self{
        return Obj1024(val,val,val,val,val,val,val,val);
    }
}

pub struct Obj2048(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);
impl Exp3 for Obj2048 {
    fn run(&mut self) {self.0 += self.15; }
    fn new(val: i128)->Self {
        return Obj2048(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }
}

pub struct Obj4096(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);
impl Exp3 for Obj4096 {
    fn run(&mut self) {self.0 += self.31; }
    fn new(val:i128)-> Self {
        return Obj4096(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val,
                       val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }
}

pub struct Obj8192(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128);
impl Exp3 for Obj8192 {
    fn run(&mut self) {self.0 += self.63; }
    fn new(val:i128)-> Self {
        return Obj8192(val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,
                       val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,
                       val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,
                       val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,val);
    }
}

pub struct Obj16384(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                    pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);
impl Exp3 for Obj16384 {
    fn run(&mut self) { self.0 += self.127; }
    fn new(val: i128) -> Self {
        return Obj16384(val, val, val, val, val, val, val, val, val, val, val,
    val, val, val, val, val,
                        val, val, val, val, val, val, val, val, val, val, val,
    val, val, val, val, val,
                        val, val, val, val, val, val, val, val, val, val, val,
    val, val, val, val, val,
                        val, val, val, val, val, val, val, val, val, val, val,
    val, val, val, val, val,
                        val, val, val, val, val, val, val, val, val, val, val,
    val, val, val, val, val,
                        val, val, val, val, val, val, val, val, val, val, val,
    val, val, val, val, val,
                        val, val, val, val, val, val, val, val, val, val, val,
    val, val, val, val, val,
                        val, val, val, val, val, val, val, val, val, val, val,
    val, val, val, val, val);
    }
}
```

Listing D.7: Experiment 3: OOP Objects

Appendix E. Experiment Four Code

5.1 Benchmark

```rust
use criterion::{criterion_group, criterion_main, Criterion};
use std::time::Duration;
use specs::prelude::*;
use thesis_experimentation::exp4::dod::*;
use thesis_experimentation::exp4::oop::*;
use thesis_experimentation::exp4::oop_obj::*;

#[inline]
fn dod_dispatch(d: &mut Dispatcher, mut w: &mut World) {
    d.dispatch_par(&mut w);
}

#[inline]
fn oop_dispatch<T: Exp4>(world: &mut OOPWorld<T>) { world.execute(); }

pub fn dod_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("dod_exp4");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);

    let thread_count = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];

    thread_count.iter().for_each(|count|{

        let mut world = World::new();
        setup_component(&mut world).unwrap();
        setup_entity(&mut world).unwrap();
        let mut dispatcher = setup_dispatcher(*count);

        dispatcher.setup(&mut world);

        let mut bench_name = String::from("dod_exp4_thread_count_");
        let i = count.to_string();
        bench_name.push_str(&i);

        group.bench_function(bench_name.as_str(), |b| b.iter(|| dod_dispatch(&
    mut dispatcher, &mut world)));

    });
}

fn oop_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("oop_exp4");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);

    let thread_count = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];

    thread_count.iter().for_each(|count| {
        let vec = obj_setup::<Obj2048>(1000);
        let mut world = OOPWorld::new(vec, *count);

        let mut bench_name = String::from("oop_exp4_thread_count_");
        let i = count.to_string();
        bench_name.push_str(&i);

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
    world)));
    });
}

criterion_group!(dod_exp4, dod_criterion_benchmark);
criterion_group!(oop_exp4, oop_criterion_benchmark);
criterion_main!(dod_exp4, oop_exp4);
```

Listing E.1: Experiment 4: Benchmark

5.2 Modules

```
pub mod dod;
pub mod oop;
pub mod oop_obj;
pub mod dod_component;
pub mod dod_system;
```

Listing E.2: Experiment 4: Modules

5.3 DOD

```
use specs::prelude::*;
use std::io;
use super::dod_component::*;
use super::dod_system::*;
use std::sync::Arc;

pub fn setup_component(world: &mut World) -> io::Result<()> {
    world.register::<Comp_i128_0>();
    world.register::<Comp_i128_1>();
    world.register::<Comp_i128_2>();
    world.register::<Comp_i128_3>();
    world.register::<Comp_i128_4>();
    world.register::<Comp_i128_5>();
    world.register::<Comp_i128_6>();
    world.register::<Comp_i128_7>();
    world.register::<Comp_i128_8>();
    world.register::<Comp_i128_9>();
    world.register::<Comp_i128_10>();
    world.register::<Comp_i128_11>();
    world.register::<Comp_i128_12>();
    world.register::<Comp_i128_13>();
    world.register::<Comp_i128_14>();
    world.register::<Comp_i128_15>();
    return Ok(());
}

pub fn setup_entity(world: &mut World) -> io::Result<()> {
    for _ in 0..1000 {
        world.create_entity()
            .with(Comp_i128_0(criterion::black_box(5)))
            .with(Comp_i128_1(criterion::black_box(5)))
            .with(Comp_i128_2(criterion::black_box(5)))
            .with(Comp_i128_3(criterion::black_box(5)))
            .with(Comp_i128_4(criterion::black_box(5)))
            .with(Comp_i128_5(criterion::black_box(5)))
            .with(Comp_i128_6(criterion::black_box(5)))
            .with(Comp_i128_7(criterion::black_box(5)))
            .with(Comp_i128_8(criterion::black_box(5)))
            .with(Comp_i128_9(criterion::black_box(5)))
            .with(Comp_i128_10(criterion::black_box(5)))
            .with(Comp_i128_11(criterion::black_box(5)))
            .with(Comp_i128_12(criterion::black_box(5)))
            .with(Comp_i128_13(criterion::black_box(5)))
            .with(Comp_i128_14(criterion::black_box(5)))
            .with(Comp_i128_15(criterion::black_box(5)))
            .build();
    }
    return Ok(())
}

pub fn setup_dispatcher<'a, 'b>(thread_count: usize)->Dispatcher<'a, 'b> {
    let pool = Arc::from(rayon::ThreadPoolBuilder::new().num_threads(
    thread_count).build().unwrap());
    match thread_count {
        1 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .build();
            return dispatcher;
        },
        2 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .build();
            return dispatcher;
        },
        3 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .build();
            return dispatcher;
        },
        4 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .build();
            return dispatcher;
        },
        5 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .build();
            return dispatcher;
        },
        6 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .build();
            return dispatcher;
        },
        7 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .with(Sys_256bit_6, "sys6", &[])
                .build();
            return dispatcher;
        },
        8 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .with(Sys_256bit_6, "sys6", &[])
                .with(Sys_256bit_7, "sys7", &[])
                .build();
            return dispatcher;
        },

        9 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .with(Sys_256bit_6, "sys6", &[])
                .with(Sys_256bit_7, "sys7", &[])
                .with(Sys_256bit_8, "sys8", &[])
                .build();
            return dispatcher;
        }
        10 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .with(Sys_256bit_6, "sys6", &[])
                .with(Sys_256bit_7, "sys7", &[])
                .with(Sys_256bit_8, "sys8", &[])
                .with(Sys_256bit_9, "sys9", &[])
                .build();
            return dispatcher;
        }
        11 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .with(Sys_256bit_6, "sys6", &[])
                .with(Sys_256bit_7, "sys7", &[])
                .with(Sys_256bit_8, "sys8", &[])
                .with(Sys_256bit_9, "sys9", &[])
                .with(Sys_256bit_10, "sys10", &[])
                .build();
            return dispatcher;
        }
        12 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .with(Sys_256bit_6, "sys6", &[])
                .with(Sys_256bit_7, "sys7", &[])
                .with(Sys_256bit_8, "sys8", &[])
                .with(Sys_256bit_9, "sys9", &[])
                .with(Sys_256bit_10, "sys10", &[])
                .with(Sys_256bit_11, "sys11", &[])
                .build();
            return dispatcher;
        }
        13 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .with(Sys_256bit_6, "sys6", &[])
                .with(Sys_256bit_7, "sys7", &[])
                .with(Sys_256bit_8, "sys8", &[])
                .with(Sys_256bit_9, "sys9", &[])
                .with(Sys_256bit_10, "sys10", &[])
                .with(Sys_256bit_11, "sys11", &[])
                .with(Sys_256bit_12, "sys12", &[])
                .build();
            return dispatcher;
        }
        14 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .with(Sys_256bit_6, "sys6", &[])
                .with(Sys_256bit_7, "sys7", &[])
                .with(Sys_256bit_8, "sys8", &[])
                .with(Sys_256bit_9, "sys9", &[])
                .with(Sys_256bit_10, "sys10", &[])
                .with(Sys_256bit_11, "sys11", &[])
                .with(Sys_256bit_12, "sys12", &[])
                .with(Sys_256bit_13, "sys13", &[])
                .build();
            return dispatcher;
        }
        15 => {
            let dispatcher = DispatcherBuilder::new()
                .with_pool(pool)
                .with(Sys_256bit_0, "sys0", &[])
                .with(Sys_256bit_1, "sys1", &[])
                .with(Sys_256bit_2, "sys2", &[])
                .with(Sys_256bit_3, "sys3", &[])
                .with(Sys_256bit_4, "sys4", &[])
                .with(Sys_256bit_5, "sys5", &[])
                .with(Sys_256bit_6, "sys6", &[])
                .with(Sys_256bit_7, "sys7", &[])
                .with(Sys_256bit_8, "sys8", &[])
                .with(Sys_256bit_9, "sys9", &[])
                .with(Sys_256bit_10, "sys10", &[])
                .with(Sys_256bit_11, "sys11", &[])
                .with(Sys_256bit_12, "sys12", &[])
                .with(Sys_256bit_13, "sys13", &[])
                .with(Sys_256bit_14, "sys14", &[])
                .build();
            return dispatcher;
        }
        _ => {panic!("Unexpected␣thread␣count");}
    };
}
```

Listing E.3: Experiment 4: DOD

5.4 DOD Components

```
use specs::prelude::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_0(pub i128);
impl Component for Comp_i128_0 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_1(pub i128);
```

```rust
impl Component for Comp_i128_1 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_2(pub i128);
impl Component for Comp_i128_2 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_3(pub i128);
impl Component for Comp_i128_3 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_4(pub i128);
impl Component for Comp_i128_4 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_5(pub i128);
```

```rust
impl Component for Comp_i128_5 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_6(pub i128);
impl Component for Comp_i128_6 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_7(pub i128);
impl Component for Comp_i128_7 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_8(pub i128);
impl Component for Comp_i128_8 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_9(pub i128);
```

```rust
impl Component for Comp_i128_9 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_10(pub i128);
impl Component for Comp_i128_10 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_11(pub i128);
impl Component for Comp_i128_11 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_12(pub i128);
impl Component for Comp_i128_12 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_13(pub i128);
```

```
impl Component for Comp_i128_13 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_14(pub i128);
impl Component for Comp_i128_14 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_15(pub i128);
impl Component for Comp_i128_15 {
    type Storage = DenseVecStorage<Self>;
}
```

Listing E.4: Experiment 4: DOD Components

5.5 DOD Systems

```
use specs::prelude::*;
use super::dod_component::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_128bit_0;
impl<'a> System<'a> for Sys_128bit_0 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>);

    fn run(&mut self, mut x: Self::SystemData) {
        for x in (&mut x).join() {
            x.0 += x.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_0;
impl<'a> System<'a> for Sys_256bit_0 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_1;
impl<'a> System<'a> for Sys_256bit_1 {

    type SystemData = (WriteStorage<'a, Comp_i128_2>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_2;
impl<'a> System<'a> for Sys_256bit_2 {

    type SystemData = (WriteStorage<'a, Comp_i128_3>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
```

```
pub struct Sys_256bit_3;
impl<'a> System<'a> for Sys_256bit_3 {

    type SystemData = (WriteStorage<'a, Comp_i128_4>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_4;
impl<'a> System<'a> for Sys_256bit_4 {

    type SystemData = (WriteStorage<'a, Comp_i128_5>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

```

```
#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_5;
impl<'a> System<'a> for Sys_256bit_5 {

    type SystemData = (WriteStorage<'a, Comp_i128_6>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_6;
impl<'a> System<'a> for Sys_256bit_6 {

    type SystemData = (WriteStorage<'a, Comp_i128_7>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_7;
impl<'a> System<'a> for Sys_256bit_7 {

    type SystemData = (WriteStorage<'a, Comp_i128_8>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_8;
impl<'a> System<'a> for Sys_256bit_8 {

    type SystemData = (WriteStorage<'a, Comp_i128_9>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_9;
impl<'a> System<'a> for Sys_256bit_9 {

    type SystemData = (WriteStorage<'a, Comp_i128_10>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_10;
impl<'a> System<'a> for Sys_256bit_10 {

    type SystemData = (WriteStorage<'a, Comp_i128_11>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
```

```rust
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_11;
impl<'a> System<'a> for Sys_256bit_11 {

    type SystemData = (WriteStorage<'a, Comp_i128_12>, ReadStorage<'a,
    Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_12;
impl<'a> System<'a> for Sys_256bit_12 {

    type SystemData = (WriteStorage<'a, Comp_i128_13>, ReadStorage<'a,
    Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_13;
impl<'a> System<'a> for Sys_256bit_13 {

    type SystemData = (WriteStorage<'a, Comp_i128_14>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_14;
impl<'a> System<'a> for Sys_256bit_14 {

    type SystemData = (WriteStorage<'a, Comp_i128_15>, ReadStorage<'a,
    Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += y.0;
        }
    }
}
```

Listing E.5: Experiment 4: DOD Systems

5.6 OOP

```
use super::oop_obj::*;
use std::sync::{Arc, RwLock};
use rayon::ThreadPoolBuilder;
use rayon::iter::IntoParallelRefMutIterator;

type ThreadPoolWrapper = Option<::std::sync::Arc<::rayon::ThreadPool>>;

pub fn obj_setup<T: Exp4>(entity_count: i32) -> Vec<T> {

    let mut vec: Vec<T> = Vec::new();
    for _ in 0..entity_count {
        let tmp = T::new(criterion::black_box(5));
        vec.push(tmp);
    }

    return vec;
```

```
}

pub struct OOPWorld<T: Exp4> {
    stages: Vec<Stage<T>>,
    pool: Arc<RwLock<ThreadPoolWrapper>>,
    count: usize
}

impl <T: Exp4> OOPWorld <T> {
    pub fn new(vec: Vec<T>, thread_count: usize)->OOPWorld<T>{
        let pool: ThreadPoolWrapper = Some(Arc::from(ThreadPoolBuilder::new().
   num_threads(thread_count).build().unwrap()));
        let pool: Arc<RwLock<ThreadPoolWrapper>> = Arc::from(RwLock::from(pool))
   ;

        let stage: Stage<T> = Stage::new(vec);
        let mut stages: Vec<Stage<T>> = Vec::new();
        stages.push(stage);

        return OOPWorld{
            stages,
            pool,
            count: thread_count
        };
    }

    pub fn execute(&mut self){
        let stages = &mut self.stages;
        let count = self.count.clone();
        self.pool
            .read()
            .unwrap()
            .as_ref()
            .unwrap()
            .install(move || {
                for stage in stages {
                    stage.execute(count);
                }
            });
    }
}

struct Stage<T: Exp4> {
    groups: Vec<Vec<T>>
}

impl <T: Exp4> Stage <T> {
    fn new(vec: Vec<T>)-> Stage<T> {

        let mut groups: Vec<Vec<T>> = Vec::new();
        groups.push(vec);

        return Stage {
            groups
        };
    }

    fn execute(&mut self, count: usize) {
        use rayon::iter::ParallelIterator;
        self.groups.par_iter_mut().for_each(|group| {
            for obj in group {
                match count {
                    1 => {
                        obj.run0();
                    },
                    2 => {
                        obj.run0();
                        obj.run1();
                    },
                    3 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                    },
                    4 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                    },
                    5 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                    },
                    6 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                    },
                    7 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                        obj.run6();
                    },
                    8 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                        obj.run6();
                        obj.run7();
                    },
                    9 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                        obj.run6();
                        obj.run7();
                        obj.run8();
                    },
                    10 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                        obj.run6();
                        obj.run7();
                        obj.run8();
                        obj.run9();
                    },
                    11 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                        obj.run6();
                        obj.run7();
                        obj.run8();
                        obj.run9();
                        obj.run10();
                    },
                    12 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                        obj.run6();
                        obj.run7();
                        obj.run8();
                        obj.run9();
                        obj.run10();
                        obj.run11();
                    },
                    13 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                        obj.run6();
                        obj.run7();
                        obj.run8();
                        obj.run9();
                        obj.run10();
                        obj.run11();
                        obj.run12();
                    },
                    14 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                        obj.run6();
                        obj.run7();
                        obj.run8();
                        obj.run9();
                        obj.run10();
                        obj.run11();
                        obj.run12();
                        obj.run13();
                    },
                    15 => {
                        obj.run0();
                        obj.run1();
                        obj.run2();
                        obj.run3();
                        obj.run4();
                        obj.run5();
                        obj.run6();
                        obj.run7();
                        obj.run8();
                        obj.run9();
                        obj.run10();
                        obj.run11();
                        obj.run12();
                        obj.run13();
                        obj.run14();
                    }
                    _ => {panic!("unexpected␣thread_count");}
                }
            }
        })
    }
}
```

Listing E.6: Experiment 4: OOP

5.7 OOP Objects

```
pub trait Exp4: Send {
    fn run0(&mut self);
    fn run1(&mut self);
    fn run2(&mut self);
    fn run3(&mut self);
    fn run4(&mut self);
    fn run5(&mut self);
    fn run6(&mut self);
    fn run7(&mut self);
    fn run8(&mut self);
    fn run9(&mut self);
    fn run10(&mut self);
    fn run11(&mut self);
    fn run12(&mut self);
    fn run13(&mut self);
    fn run14(&mut self);
    fn new(val: i128)->Self;
}

pub struct Obj2048(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128, pub i128);

impl Exp4 for Obj2048 {
    fn run0(&mut self) {self.15 += self.0; }
    fn run1(&mut self) {self.14 += self.0; }
    fn run2(&mut self) {self.13 += self.0; }
    fn run3(&mut self) {self.12 += self.0; }
    fn run4(&mut self) {self.11 += self.0; }
    fn run5(&mut self) {self.10 += self.0; }
    fn run6(&mut self) {self.9 += self.0; }
    fn run7(&mut self) {self.8 += self.0; }
    fn run8(&mut self) {self.7 += self.0; }
    fn run9(&mut self) {self.6 += self.0; }
    fn run10(&mut self) {self.5 += self.0; }
    fn run11(&mut self) {self.4 += self.0; }
    fn run12(&mut self) {self.3 += self.0; }
    fn run13(&mut self) {self.2 += self.0; }
    fn run14(&mut self) {self.1 += self.0; }
fn new(val: i128)->Self {
    return Obj2048(val,val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val);
}
}
```

Listing E.7: Experiment 4: OOP Objects

Appendix F. Experiment Five Code

6.1 Benchmark

```rust
use criterion::{criterion_group, criterion_main, Criterion};
use std::time::Duration;
use specs::prelude::*;
use thesis_experimentation::exp5::dod::*;
use thesis_experimentation::exp5::oop::*;
use thesis_experimentation::exp5::oop_obj::*;


#[inline]
fn dod_dispatch(d: &mut Dispatcher, mut w: &mut World) {
    d.dispatch_par(&mut w);
}

fn oop_dispatch<T: Exp5>(world: &mut OOPWorld<T>) { world.execute(); }

pub fn dod_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("dod_exp5");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(10);

    let thread_count = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];

    thread_count.iter().for_each(|count|{

        let mut world = World::new();
        setup_component(&mut world).unwrap();
        setup_entity(&mut world).unwrap();
        let mut dispatcher = setup_dispatcher(*count);

        dispatcher.setup(&mut world);

        let mut bench_name = String::from("dod_exp5_thread_count_");
        let i = count.to_string();
        bench_name.push_str(&i);

        group.bench_function(bench_name.as_str(), |b| b.iter(|| dod_dispatch(&
    mut dispatcher, &mut world)));

    });
}

fn oop_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("oop_exp5");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(10);

    let thread_count = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];

    thread_count.iter().for_each(|count| {
        let vec = obj_setup::<Obj2048>(1000, *count);
        let mut world = OOPWorld::new(vec, *count);
```

```rust

        let mut bench_name = String::from("oop_exp5_thread_count_");
        let i = count.to_string();
        bench_name.push_str(&i);

        group.bench_function(bench_name.as_str(), |b| b.iter(||oop_dispatch(&mut
     world)));
    });
}

criterion_group!(dod_exp5, dod_criterion_benchmark);
criterion_group!(oop_exp5, oop_criterion_benchmark);
criterion_main!(dod_exp5, oop_exp5);
```

Listing F.1: Experiment 5: Benchmark

6.2 Modules

```rust
pub mod dod;
pub mod oop;
pub mod dod_component;
pub mod dod_system;
pub mod oop_obj;
```

Listing F.2: Experiment 5: Modules

6.3 DOD

```rust
use specs::prelude::*;
use std::io;
use super::dod_system::*;
use super::dod_component::*;
use std::sync::Arc;

pub fn setup_component(world: &mut World) -> io::Result<()> {
    world.register::<Comp_i128_0>();
    world.register::<Comp_i128_1>();
    world.register::<Comp_i128_2>();
    world.register::<Comp_i128_3>();
    world.register::<Comp_i128_4>();
    world.register::<Comp_i128_5>();
    world.register::<Comp_i128_6>();
    world.register::<Comp_i128_7>();
    world.register::<Comp_i128_8>();
    world.register::<Comp_i128_9>();
    world.register::<Comp_i128_10>();
    world.register::<Comp_i128_11>();
    world.register::<Comp_i128_12>();
    world.register::<Comp_i128_13>();
    world.register::<Comp_i128_14>();
    world.register::<Comp_i128_15>();
    return Ok(());
}

pub fn setup_entity(world: &mut World) -> io::Result<()> {
    for _ in 0..1000 {
        world.create_entity()
            .with(Comp_i128_0(criterion::black_box(5)))
            .with(Comp_i128_1(criterion::black_box(5)))
            .with(Comp_i128_2(criterion::black_box(5)))
            .with(Comp_i128_3(criterion::black_box(5)))
            .with(Comp_i128_4(criterion::black_box(5)))
            .with(Comp_i128_5(criterion::black_box(5)))
            .with(Comp_i128_6(criterion::black_box(5)))
            .with(Comp_i128_7(criterion::black_box(5)))
            .with(Comp_i128_8(criterion::black_box(5)))
            .with(Comp_i128_9(criterion::black_box(5)))
            .with(Comp_i128_10(criterion::black_box(5)))
            .with(Comp_i128_11(criterion::black_box(5)))
            .with(Comp_i128_12(criterion::black_box(5)))
            .with(Comp_i128_13(criterion::black_box(5)))
            .with(Comp_i128_14(criterion::black_box(5)))
            .with(Comp_i128_15(criterion::black_box(5)))
            .build();
    }
    return Ok(())
}

pub fn setup_dispatcher<'a, 'b>(thread_count: usize)->Dispatcher<'a, 'b> {
    let pool = Arc::from(rayon::ThreadPoolBuilder::new().num_threads(
    thread_count).build().unwrap());

    let dispatcher = DispatcherBuilder::new()
        .with_pool(pool)
        .with(Sys_256bit_0, "sys0", &[])
        .with(Sys_256bit_1, "sys1", &[])
        .with(Sys_256bit_2, "sys2", &[])
        .with(Sys_256bit_3, "sys3", &[])
        .with(Sys_256bit_4, "sys4", &[])
        .with(Sys_256bit_5, "sys5", &[])
        .with(Sys_256bit_6, "sys6", &[])
        .with(Sys_256bit_7, "sys7", &[])
        .with(Sys_256bit_8, "sys8", &[])
        .with(Sys_256bit_9, "sys9", &[])
        .with(Sys_256bit_10, "sys10", &[])
        .with(Sys_256bit_11, "sys11", &[])
        .with(Sys_256bit_12, "sys12", &[])
        .with(Sys_256bit_13, "sys13", &[])
        .with(Sys_256bit_14, "sys14", &[])
        .build();
    return dispatcher;
}
```

Listing F.3: Experiment 5: DOD

6.4 DOD Components

```
use specs::prelude::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_0(pub i128);
impl Component for Comp_i128_0 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```rust
#[allow(non_camel_case_types)]
pub struct Comp_i128_1(pub i128);
impl Component for Comp_i128_1 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_2(pub i128);
impl Component for Comp_i128_2 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_3(pub i128);
impl Component for Comp_i128_3 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_4(pub i128);
impl Component for Comp_i128_4 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```rust
#[allow(non_camel_case_types)]
pub struct Comp_i128_5(pub i128);
impl Component for Comp_i128_5 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_6(pub i128);
impl Component for Comp_i128_6 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_7(pub i128);
impl Component for Comp_i128_7 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_8(pub i128);
impl Component for Comp_i128_8 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```rust
#[allow(non_camel_case_types)]
pub struct Comp_i128_9(pub i128);
impl Component for Comp_i128_9 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_10(pub i128);
impl Component for Comp_i128_10 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_11(pub i128);
impl Component for Comp_i128_11 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_12(pub i128);
impl Component for Comp_i128_12 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```rust
#[allow(non_camel_case_types)]
pub struct Comp_i128_13(pub i128);
impl Component for Comp_i128_13 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_14(pub i128);
impl Component for Comp_i128_14 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_15(pub i128);
impl Component for Comp_i128_15 {
    type Storage = DenseVecStorage<Self>;
}
```

Listing F.4: Experiment 5: DOD Components

6.5 DOD Systems

```rust
use specs::prelude::*;
use super::dod_component::*;

fn fib(n: i128)->i128
{
    if n <= 1 {

        return n;
    }
    return fib(n-1) + fib(n-2);
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_128bit_0;
impl<'a> System<'a> for Sys_128bit_0 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>);

    fn run(&mut self, mut x: Self::SystemData) {
        for x in (&mut x).join() {
            x.0 += fib(x.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_0;
impl<'a> System<'a> for Sys_256bit_0 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
   Comp_i128_1>);

```

```
    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_1;
impl<'a> System<'a> for Sys_256bit_1 {

    type SystemData = (WriteStorage<'a, Comp_i128_2>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_2;
impl<'a> System<'a> for Sys_256bit_2 {

    type SystemData = (WriteStorage<'a, Comp_i128_3>, ReadStorage<'a,
    Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_3;
impl<'a> System<'a> for Sys_256bit_3 {

    type SystemData = (WriteStorage<'a, Comp_i128_4>, ReadStorage<'a,
    Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_4;
impl<'a> System<'a> for Sys_256bit_4 {

    type SystemData = (WriteStorage<'a, Comp_i128_5>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_5;
impl<'a> System<'a> for Sys_256bit_5 {

    type SystemData = (WriteStorage<'a, Comp_i128_6>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
```

```rust
pub struct Sys_256bit_6;
impl<'a> System<'a> for Sys_256bit_6 {

    type SystemData = (WriteStorage<'a, Comp_i128_7>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_7;
impl<'a> System<'a> for Sys_256bit_7 {

    type SystemData = (WriteStorage<'a, Comp_i128_8>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

```

```rust
#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_8;
impl<'a> System<'a> for Sys_256bit_8 {

    type SystemData = (WriteStorage<'a, Comp_i128_9>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_9;
impl<'a> System<'a> for Sys_256bit_9 {

    type SystemData = (WriteStorage<'a, Comp_i128_10>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
```

```
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_10;
impl<'a> System<'a> for Sys_256bit_10 {

    type SystemData = (WriteStorage<'a, Comp_i128_11>, ReadStorage<'a,
    Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_11;
impl<'a> System<'a> for Sys_256bit_11 {

    type SystemData = (WriteStorage<'a, Comp_i128_12>, ReadStorage<'a,
    Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_12;
impl<'a> System<'a> for Sys_256bit_12 {

    type SystemData = (WriteStorage<'a, Comp_i128_13>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_13;
impl<'a> System<'a> for Sys_256bit_13 {

    type SystemData = (WriteStorage<'a, Comp_i128_14>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
```

```rust
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256bit_14;
impl<'a> System<'a> for Sys_256bit_14 {

    type SystemData = (WriteStorage<'a, Comp_i128_15>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut x, y): Self::SystemData) {
        for (x, y) in (&mut x, &y).join() {
            x.0 += fib(y.0);
        }
    }
}
```

Listing F.5: Experiment 5: DOD Systems

6.6 OOP

```rust
use super::oop_obj::*;
use std::sync::{Arc, RwLock};
use rayon::ThreadPoolBuilder;
use rayon::iter::IntoParallelRefMutIterator;

```

```rust
type ThreadPoolWrapper = Option<::std::sync::Arc<::rayon::ThreadPool>>;

pub fn obj_setup<T: Exp5>(entity_count: i32, thread_count: usize) -> Vec<Vec<T>>
     {

    let mut vec: Vec<Vec<T>> = Vec::new();
    for i in 0..thread_count {
        vec.push(Vec::new());
        for _ in 0..entity_count/(thread_count as i32) {
            vec.get_mut(i).unwrap().push(T::new(criterion::black_box(5)));
        }
    }
    return vec;
}

//----------------------------------------------------------
pub struct OOPWorld<T: Exp5> {
    stages: Vec<Stage<T>>,
    pool: Arc<RwLock<ThreadPoolWrapper>>,
}

impl <T: Exp5> OOPWorld <T> {
    pub fn new(vec: Vec<Vec<T>>, thread_count: usize)->OOPWorld<T>{
        let pool: ThreadPoolWrapper = Some(Arc::from(ThreadPoolBuilder::new().
    num_threads(thread_count).build().unwrap()));
        let pool: Arc<RwLock<ThreadPoolWrapper>> = Arc::from(RwLock::from(pool))
    ;

```

```
        let stage: Stage<T> = Stage::new(vec);
        let mut stages: Vec<Stage<T>> = Vec::new();
        stages.push(stage);

        return OOPWorld{
            stages,
            pool,
        };
    }

    pub fn execute(&mut self){
        let stages = &mut self.stages;

        self.pool
            .read()
            .unwrap()
            .as_ref()
            .unwrap()
            .install(move || {
                for stage in stages {
                    stage.execute();
                }
            });
    }
}

//----------------------------------------------------------

```

```
struct Stage<T: Exp5> {
    groups: Vec<Vec<T>>
}

impl <T: Exp5> Stage <T> {
    fn new(vec: Vec<Vec<T>>)-> Stage<T> {

        let groups = vec;

        return Stage {
            groups
        };
    }

    fn execute(&mut self) {
        use rayon::iter::ParallelIterator;
        self.groups.par_iter_mut().for_each(|group| {
            for obj in group {
                obj.run0();
                obj.run1();
                obj.run2();
                obj.run3();
                obj.run4();
                obj.run5();
                obj.run6();
                obj.run7();
                obj.run8();
                obj.run9();
                obj.run10();
                obj.run11();
                obj.run12();
                obj.run13();
                obj.run14();
            }
        })
    }
}
```

Listing F.6: Experiment 5: OOP

6.7 OOP Objects

```
fn fib(n: i128)->i128
{
    if n <= 1 {

        return n;
    }
    return fib(n-1) + fib(n-2);
}

pub trait Exp5: Send {
    fn run0(&mut self);
    fn run1(&mut self);
    fn run2(&mut self);
    fn run3(&mut self);
    fn run4(&mut self);
    fn run5(&mut self);
    fn run6(&mut self);
    fn run7(&mut self);
    fn run8(&mut self);
    fn run9(&mut self);
    fn run10(&mut self);
    fn run11(&mut self);
    fn run12(&mut self);
    fn run13(&mut self);
    fn run14(&mut self);
    fn new(val: i128)->Self;
}
pub struct Obj2048(pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128,
                   pub i128, pub i128, pub i128, pub i128, pub i128, pub i128,
    pub i128, pub i128);

impl Exp5 for Obj2048 {
    fn run0(&mut self) {self.15 += fib(self.0); }
    fn run1(&mut self) {self.14 += fib(self.0); }
    fn run2(&mut self) {self.13 += fib(self.0); }
    fn run3(&mut self) {self.12 += fib(self.0); }
    fn run4(&mut self) {self.11 += fib(self.0); }
    fn run5(&mut self) {self.10 += fib(self.0); }
    fn run6(&mut self) {self.9 += fib(self.0); }
    fn run7(&mut self) {self.8 += fib(self.0); }
    fn run8(&mut self) {self.7 += fib(self.0); }
    fn run9(&mut self) {self.6 += fib(self.0); }
    fn run10(&mut self) {self.5 += fib(self.0); }
    fn run11(&mut self) {self.4 += fib(self.0); }
    fn run12(&mut self) {self.3 += fib(self.0); }
    fn run13(&mut self) {self.2 += fib(self.0); }
    fn run14(&mut self) {self.1 += fib(self.0); }
    fn new(val: i128)->Self {
        return Obj2048(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }
}
```

Listing F.7: Experiment 5: OOP Objects

Appendix G. Experiment Six Code

7.1 Benchmark

```
use criterion::{criterion_group, criterion_main, Criterion};
use std::time::Duration;
use thesis_experimentation::exp6::oop::*;
use thesis_experimentation::exp6::oop_obj::*;
use thesis_experimentation::exp6::dod::*;
use specs::prelude::*;

#[inline]
fn oop_dispatch<T: Exp6>(world: &mut OOPWorld<T>) { world.execute(); }

#[inline]
fn dod_dispatch(d: &mut Dispatcher, mut w: &mut World) {
    d.dispatch_par(&mut w);
}

fn oop_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("oop_exp6");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);
    rayon::ThreadPoolBuilder::new().num_threads(1).build_global().unwrap();

    let o2048 = obj_setup::<Exp2048>();
    let o1912 = obj_setup::<Exp1912>();
    let o1792 = obj_setup::<Exp1792>();
    let o1664 = obj_setup::<Exp1664>();
    let o1536 = obj_setup::<Exp1536>();
    let o1408 = obj_setup::<Exp1408>();
    let o1280 = obj_setup::<Exp1280>();
    let o1152 = obj_setup::<Exp1152>();
    let o1024 = obj_setup::<Exp1024>();
    let o896 = obj_setup::<Exp896>();
    let o768 = obj_setup::<Exp768>();
    let o640 = obj_setup::<Exp640>();
    let o512 = obj_setup::<Exp512>();
    let o384 = obj_setup::<Exp384>();
    let o256 = obj_setup::<Exp256>();

    let mut world2048 = OOPWorld::new(o2048);
    let mut world1912 = OOPWorld::new(o1912);
    let mut world1792 = OOPWorld::new(o1792);
    let mut world1664 = OOPWorld::new(o1664);
    let mut world1536 = OOPWorld::new(o1536);
    let mut world1408 = OOPWorld::new(o1408);
    let mut world1280 = OOPWorld::new(o1280);
    let mut world1152 = OOPWorld::new(o1152);
    let mut world1024 = OOPWorld::new(o1024);
    let mut world896 = OOPWorld::new(o896);
    let mut world768 = OOPWorld::new(o768);
    let mut world640 = OOPWorld::new(o640);
    let mut world512 = OOPWorld::new(o512);
    let mut world384 = OOPWorld::new(o384);
    let mut world256 = OOPWorld::new(o256);

    group.bench_function("oop_exp6_2048", |b| b.iter(||oop_dispatch(&mut
   world2048)));
    group.bench_function("oop_exp6_1912", |b| b.iter(||oop_dispatch(&mut
   world1912)));
    group.bench_function("oop_exp6_1792", |b| b.iter(||oop_dispatch(&mut
   world1792)));
    group.bench_function("oop_exp6_1664", |b| b.iter(||oop_dispatch(&mut
   world1664)));
    group.bench_function("oop_exp6_1536", |b| b.iter(||oop_dispatch(&mut
   world1536)));
    group.bench_function("oop_exp6_1408", |b| b.iter(||oop_dispatch(&mut
   world1408)));
    group.bench_function("oop_exp6_1280", |b| b.iter(||oop_dispatch(&mut
   world1280)));
    group.bench_function("oop_exp6_1152", |b| b.iter(||oop_dispatch(&mut
   world1152)));
    group.bench_function("oop_exp6_1024", |b| b.iter(||oop_dispatch(&mut
   world1024)));
    group.bench_function("oop_exp6_896", |b| b.iter(||oop_dispatch(&mut world896
   )));
    group.bench_function("oop_exp6_768", |b| b.iter(||oop_dispatch(&mut world768
   )));
    group.bench_function("oop_exp6_640", |b| b.iter(||oop_dispatch(&mut world640
   )));
    group.bench_function("oop_exp6_512", |b| b.iter(||oop_dispatch(&mut world512
   )));
    group.bench_function("oop_exp6_384", |b| b.iter(||oop_dispatch(&mut world384
    )));
    group.bench_function("oop_exp6_256", |b| b.iter(||oop_dispatch(&mut world256
    )));
}

pub fn dod_criterion_benchmark(c: &mut Criterion) {
    let mut group = c.benchmark_group("dod_exp6");
    group.warm_up_time(Duration::from_secs(5));
    group.sample_size(100);
    group.nresamples(100);

    let entity_state_count = vec![2048, 1912, 1792, 1664, 1536, 1408, 1280,
    1152, 1024, 896, 768, 640, 512, 384, 256];

    entity_state_count.iter().for_each(|count| {
        let mut world = World::new();
        setup_component(&mut world).unwrap();
        setup_entity( &mut world).unwrap();
        let mut dispatcher = setup_dispatcher(*count);
        let mut bench_name = String::from("dod_exp6_");
        let i = count.to_string();
        bench_name.push_str(&i);
        group.bench_function(bench_name, |b| b.iter( || dod_dispatch(&mut
    dispatcher, &mut world)));
    });
}

criterion_group!(oop_exp6, oop_criterion_benchmark);
```

```rust
criterion_group!(dod_exp6, dod_criterion_benchmark);
criterion_main!(oop_exp6, dod_exp6);
```

Listing G.1: Experiment 6: Benchmark

7.2 Modules

```rust
pub mod oop;
pub mod dod;
pub mod oop_obj;
pub mod dod_obj;
```

Listing G.2: Experiment 6: Modules

7.3 DOD

```rust
use specs::prelude::*;
use std::io;
use super::dod_obj::*;
use std::sync::Arc;

//All Entities use 2048 bits, which is 16 i128's
pub fn setup_component(world: &mut World)-> io::Result<()> {
    world.register::<Comp_i128_0>();
    world.register::<Comp_i128_1>();
    world.register::<Comp_i128_2>();
    world.register::<Comp_i128_3>();
    world.register::<Comp_i128_4>();
    world.register::<Comp_i128_5>();
    world.register::<Comp_i128_6>();
    world.register::<Comp_i128_7>();
    world.register::<Comp_i128_8>();
    world.register::<Comp_i128_9>();
    world.register::<Comp_i128_10>();
    world.register::<Comp_i128_11>();
    world.register::<Comp_i128_12>();
    world.register::<Comp_i128_13>();
    world.register::<Comp_i128_14>();
    world.register::<Comp_i128_15>();

    return Ok(())
}

//All Entities use 2048 bits, which is 16 i128's
pub fn setup_entity(world: &mut World)->io::Result<()> {

    for _ in 0..5000 {
        world.create_entity()
            .with(Comp_i128_0(criterion::black_box(5)))
            .with(Comp_i128_1(criterion::black_box(5)))
            .with(Comp_i128_2(criterion::black_box(5)))
            .with(Comp_i128_3(criterion::black_box(5)))
            .with(Comp_i128_4(criterion::black_box(5)))
            .with(Comp_i128_5(criterion::black_box(5)))
            .with(Comp_i128_6(criterion::black_box(5)))
            .with(Comp_i128_7(criterion::black_box(5)))
            .with(Comp_i128_8(criterion::black_box(5)))
            .with(Comp_i128_9(criterion::black_box(5)))
            .with(Comp_i128_10(criterion::black_box(5)))
            .with(Comp_i128_11(criterion::black_box(5)))
            .with(Comp_i128_12(criterion::black_box(5)))
            .with(Comp_i128_13(criterion::black_box(5)))
            .with(Comp_i128_14(criterion::black_box(5)))
            .with(Comp_i128_15(criterion::black_box(5)))
            .build();
    }
    return Ok(())
}

//This differs based on which experiment is going on
pub fn setup_dispatcher<'a, 'b>(size: i32)->Dispatcher<'a, 'b> {

    let pool = Arc::from(rayon::ThreadPoolBuilder::new().num_threads(1).build().
   unwrap());

    match size {
        2048 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_2048, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        1912 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_1912, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        1792 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_1792, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        1664 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_1664, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        1536 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_1536, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        1408 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_1408, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        1280 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_1280, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        1152 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_1152, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        1024 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_1024, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        896 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_896, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        768 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_768, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        640 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_640, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        512 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_512, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        384 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_384, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        256 => {
            let dispatcher = DispatcherBuilder::new()
                .with(Sys_256, "sys", &[])
                .with_pool(pool)
                .build();
            return dispatcher;
        }

        _ => {panic!("unknown␣data␣size");}
    }
```

```
}
```

Listing G.3: Experiment 6: DOD

7.4 DOD Components

```
use specs::prelude::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_0(pub i128);
impl Component for Comp_i128_0 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_1(pub i128);
impl Component for Comp_i128_1 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_2(pub i128);
impl Component for Comp_i128_2 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```rust
#[allow(non_camel_case_types)]
pub struct Comp_i128_3(pub i128);
impl Component for Comp_i128_3 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_4(pub i128);
impl Component for Comp_i128_4 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_5(pub i128);
impl Component for Comp_i128_5 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_6(pub i128);
impl Component for Comp_i128_6 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```
#[allow(non_camel_case_types)]
pub struct Comp_i128_7(pub i128);
impl Component for Comp_i128_7 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_8(pub i128);
impl Component for Comp_i128_8 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_9(pub i128);
impl Component for Comp_i128_9 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_10(pub i128);
impl Component for Comp_i128_10 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```
#[allow(non_camel_case_types)]
pub struct Comp_i128_11(pub i128);
impl Component for Comp_i128_11 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_12(pub i128);
impl Component for Comp_i128_12 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_13(pub i128);
impl Component for Comp_i128_13 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Comp_i128_14(pub i128);
impl Component for Comp_i128_14 {
    type Storage = DenseVecStorage<Self>;
}

#[derive(Debug)]
```

```
#[allow(non_camel_case_types)]
pub struct Comp_i128_15(pub i128);
impl Component for Comp_i128_15 {
    type Storage = DenseVecStorage<Self>;
}
```

Listing G.4: Experiment 6: DOD Components

7.5 DOD Systems

```
use specs::prelude::*;
use super::dod_component::*;

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_2048;
impl<'a> System<'a> for Sys_2048 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
    Comp_i128_1>,
                        ReadStorage<'a, Comp_i128_2>, ReadStorage<'a, Comp_i128_3
    >,
                        ReadStorage<'a, Comp_i128_4>, ReadStorage<'a, Comp_i128_5
    >,
                        ReadStorage<'a, Comp_i128_6>, ReadStorage<'a, Comp_i128_7
    >,
                        ReadStorage<'a, Comp_i128_8>, ReadStorage<'a, Comp_i128_9
    >,
                        ReadStorage<'a, Comp_i128_10>, ReadStorage<'a,
    Comp_i128_11>,
                            ReadStorage<'a, Comp_i128_12>, ReadStorage<'a, Comp_i128_13>,
                            ReadStorage<'a, Comp_i128_14>, ReadStorage<'a, Comp_i128_15>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12, o13, o14, o15): Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12, o13, o14, o15) in (&mut o0, &o1, &o2, &o3, &o4, &o5, &o6, &o7, &o8, &o9, &o10, &o11, &o12, &o13, &o14, &o15).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0 + o7.0 + o8.0 + o9.0 + o10.0 + o11.0 + o12.0 + o13.0 + o14.0 + o15.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_1912;
impl<'a> System<'a> for Sys_1912 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a, Comp_i128_1>,
                            ReadStorage<'a, Comp_i128_2>, ReadStorage<'a, Comp_i128_3>,
                            ReadStorage<'a, Comp_i128_4>, ReadStorage<'a, Comp_i128_5>,
                            ReadStorage<'a, Comp_i128_6>, ReadStorage<'a, Comp_i128_7
```

```
                        >,
                        ReadStorage<'a, Comp_i128_8>, ReadStorage<'a, Comp_i128_9>,
                        ReadStorage<'a, Comp_i128_10>, ReadStorage<'a, Comp_i128_11>,
                        ReadStorage<'a, Comp_i128_12>, ReadStorage<'a, Comp_i128_13>,
                        ReadStorage<'a, Comp_i128_14>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12, o13, o14): Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12, o13, o14) in (&mut o0, &o1, &o2, &o3, &o4, &o5, &o6, &o7, &o8, &o9, &o10, &o11, &o12, &o13, &o14).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0 + o7.0 + o8.0 + o9.0 + o10.0 + o11.0 + o12.0 + o13.0 + o14.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_1792;
impl<'a> System<'a> for Sys_1792 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a, Comp_i128_1>,
                        ReadStorage<'a, Comp_i128_2>, ReadStorage<'a, Comp_i128_3
    >,
                            ReadStorage<'a, Comp_i128_4>, ReadStorage<'a, Comp_i128_5
    >,
                            ReadStorage<'a, Comp_i128_6>, ReadStorage<'a, Comp_i128_7
    >,
                            ReadStorage<'a, Comp_i128_8>, ReadStorage<'a, Comp_i128_9
    >,
                            ReadStorage<'a, Comp_i128_10>, ReadStorage<'a,
    Comp_i128_11>,
                            ReadStorage<'a, Comp_i128_12>, ReadStorage<'a,
    Comp_i128_13>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12
    , o13): Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12, o13) in (&
    mut o0, &o1, &o2, &o3, &o4, &o5, &o6, &o7, &o8, &o9, &o10, &o11, &o12, &o13)
    .join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0 + o7.0 + o8.0 + o9.0
     + o10.0 + o11.0 + o12.0 + o13.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_1664;
impl<'a> System<'a> for Sys_1664 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
    Comp_i128_1>,
                        ReadStorage<'a, Comp_i128_2>, ReadStorage<'a, Comp_i128_3
    >,
                        ReadStorage<'a, Comp_i128_4>, ReadStorage<'a, Comp_i128_5
    >,
                        ReadStorage<'a, Comp_i128_6>, ReadStorage<'a, Comp_i128_7
    >,
                        ReadStorage<'a, Comp_i128_8>, ReadStorage<'a, Comp_i128_9
    >,
                        ReadStorage<'a, Comp_i128_10>, ReadStorage<'a,
    Comp_i128_11>,
                        ReadStorage<'a, Comp_i128_12>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12
    ): Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12) in (&mut o0,
     &o1, &o2, &o3, &o4, &o5, &o6, &o7, &o8, &o9, &o10, &o11, &o12).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0 + o7.0 + o8.0 + o9.0
     + o10.0 + o11.0 + o12.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_1536;
impl<'a> System<'a> for Sys_1536 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
    Comp_i128_1>,
                        ReadStorage<'a, Comp_i128_2>, ReadStorage<'a, Comp_i128_3
    >,
                        ReadStorage<'a, Comp_i128_4>, ReadStorage<'a, Comp_i128_5
    >,
                        ReadStorage<'a, Comp_i128_6>, ReadStorage<'a, Comp_i128_7
    >,
                        ReadStorage<'a, Comp_i128_8>, ReadStorage<'a, Comp_i128_9
    >,
                        ReadStorage<'a, Comp_i128_10>, ReadStorage<'a,
    Comp_i128_11>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11):
    Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11) in (&mut o0, &o1,
     &o2, &o3, &o4, &o5, &o6, &o7, &o8, &o9, &o10, &o11).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0 + o7.0 + o8.0 + o9.0
     + o10.0 + o11.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_1408;
impl<'a> System<'a> for Sys_1408 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a, Comp_i128_1>,
                       ReadStorage<'a, Comp_i128_2>, ReadStorage<'a, Comp_i128_3>,
                       ReadStorage<'a, Comp_i128_4>, ReadStorage<'a, Comp_i128_5>,
                       ReadStorage<'a, Comp_i128_6>, ReadStorage<'a, Comp_i128_7>,
                       ReadStorage<'a, Comp_i128_8>, ReadStorage<'a, Comp_i128_9>,
                       ReadStorage<'a, Comp_i128_10>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10): Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5, o6, o7, o8, o9, o10) in (&mut o0, &o1, &o2, &o3, &o4, &o5, &o6, &o7, &o8, &o9, &o10).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0 + o7.0 + o8.0 + o9.0 + o10.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_1280;
impl<'a> System<'a> for Sys_1280 {

    type SystemData = (WriteStorage<’a, Comp_i128_0>, ReadStorage<’a, Comp_i128_1>,
                       ReadStorage<’a, Comp_i128_2>, ReadStorage<’a, Comp_i128_3>,
                       ReadStorage<’a, Comp_i128_4>, ReadStorage<’a, Comp_i128_5>,
                       ReadStorage<’a, Comp_i128_6>, ReadStorage<’a, Comp_i128_7>,
                       ReadStorage<’a, Comp_i128_8>, ReadStorage<’a, Comp_i128_9>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6, o7, o8, o9): Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5, o6, o7, o8, o9) in (&mut o0, &o1, &o2, &o3, &o4, &o5, &o6, &o7, &o8, &o9).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0 + o7.0 + o8.0 + o9.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_1152;
impl<’a> System<’a> for Sys_1152 {

    type SystemData = (WriteStorage<’a, Comp_i128_0>, ReadStorage<’a, Comp_i128_1>,
                        ReadStorage<'a, Comp_i128_2>, ReadStorage<'a, Comp_i128_3
   >,
                        ReadStorage<'a, Comp_i128_4>, ReadStorage<'a, Comp_i128_5
   >,
                        ReadStorage<'a, Comp_i128_6>, ReadStorage<'a, Comp_i128_7
   >,
                        ReadStorage<'a, Comp_i128_8>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6, o7, o8): Self::SystemData
   ) {
        for (o0, o1, o2, o3, o4, o5, o6, o7, o8) in (&mut o0, &o1, &o2, &o3, &o4
   , &o5, &o6, &o7, &o8).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0 + o7.0 + o8.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_1024;
impl<'a> System<'a> for Sys_1024 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
   Comp_i128_1>,
                        ReadStorage<'a, Comp_i128_2>, ReadStorage<'a, Comp_i128_3
   >,
                        ReadStorage<'a, Comp_i128_4>, ReadStorage<'a, Comp_i128_5
   >,
                        ReadStorage<'a, Comp_i128_6>, ReadStorage<'a, Comp_i128_7>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6, o7): Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5, o6, o7) in (&mut o0, &o1, &o2, &o3, &o4, &o5, &o6, &o7).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0 + o7.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_896;
impl<'a> System<'a> for Sys_896 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a, Comp_i128_1>,
                        ReadStorage<'a, Comp_i128_2>, ReadStorage<'a, Comp_i128_3>,
                        ReadStorage<'a, Comp_i128_4>, ReadStorage<'a, Comp_i128_5>,
                        ReadStorage<'a, Comp_i128_6>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5, o6): Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5, o6) in (&mut o0, &o1, &o2, &o3, &o4, &o5, &o6).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0 + o6.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_768;
impl<’a> System<’a> for Sys_768 {

    type SystemData = (WriteStorage<’a, Comp_i128_0>, ReadStorage<’a, Comp_i128_1>,
                       ReadStorage<’a, Comp_i128_2>, ReadStorage<’a, Comp_i128_3>,
                       ReadStorage<’a, Comp_i128_4>, ReadStorage<’a, Comp_i128_5>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4, o5): Self::SystemData) {
        for (o0, o1, o2, o3, o4, o5) in (&mut o0, &o1, &o2, &o3, &o4, &o5).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0 + o5.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_640;
impl<’a> System<’a> for Sys_640 {

    type SystemData = (WriteStorage<’a, Comp_i128_0>, ReadStorage<’a, Comp_i128_1>,
                       ReadStorage<’a, Comp_i128_2>, ReadStorage<’a, Comp_i128_3>,
                       ReadStorage<’a, Comp_i128_4>);

    fn run(&mut self, (mut o0, o1, o2, o3, o4): Self::SystemData) {
        for (o0, o1, o2, o3, o4) in (&mut o0, &o1, &o2, &o3, &o4).join() {
            o0.0 += o1.0 + o2.0 + o3.0 + o4.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_512;
impl<’a> System<’a> for Sys_512 {

    type SystemData = (WriteStorage<’a, Comp_i128_0>, ReadStorage<’a, Comp_i128_1>,
                       ReadStorage<’a, Comp_i128_2>, ReadStorage<’a, Comp_i128_3>);

    fn run(&mut self, (mut o0, o1, o2, o3): Self::SystemData) {
        for (o0, o1, o2, o3) in (&mut o0, &o1, &o2, &o3).join() {
            o0.0 += o1.0 + o2.0 + o3.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_384;
impl<'a> System<'a> for Sys_384 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
   Comp_i128_1>,
                       ReadStorage<'a, Comp_i128_2>);

    fn run(&mut self, (mut o0, o1, o2): Self::SystemData) {
        for (o0, o1, o2) in (&mut o0, &o1, &o2).join() {
            o0.0 += o1.0 + o2.0;
        }
    }
}

#[derive(Debug)]
#[allow(non_camel_case_types)]
pub struct Sys_256;
impl<'a> System<'a> for Sys_256 {

    type SystemData = (WriteStorage<'a, Comp_i128_0>, ReadStorage<'a,
   Comp_i128_1>);

    fn run(&mut self, (mut o0, o1): Self::SystemData) {
        for (o0, o1) in (&mut o0, &o1).join() {
            o0.0 += o1.0;
        }
    }
}
```

Listing G.5: Experiment 6: DOD Systems

7.6 OOP

```
use super::oop_obj::*;
use std::sync::{Arc, RwLock};
use rayon::*;
use rayon::iter::IntoParallelRefMutIterator;

type ThreadPoolWrapper = Option<::std::sync::Arc<::rayon::ThreadPool>>;

pub fn obj_setup<T: Exp6>()-> Vec<T> {

    let mut vec: Vec<T> = Vec::new();
    for _ in 0..5000 {
        let tmp = T::new(criterion::black_box(5));
        vec.push(tmp);
    }

    return vec;
}

//----------------------------------------------------------
pub struct OOPWorld<T: Exp6> {
    stages: Vec<Stage<T>>,
    pool: Arc<RwLock<ThreadPoolWrapper>>
}

impl <T: Exp6> OOPWorld <T> {
    pub fn new(vec: Vec<T>, )->OOPWorld<T>{
        let pool: ThreadPoolWrapper = Some(Arc::from(ThreadPoolBuilder::new().
   num_threads(1).build().unwrap()));
        let pool: Arc<RwLock<ThreadPoolWrapper>> = Arc::from(RwLock::from(pool))
   ;

        let stage: Stage<T> = Stage::new(vec);
        let mut stages: Vec<Stage<T>> = Vec::new();
        stages.push(stage);

        return OOPWorld{
            stages,
            pool
        };
    }

    pub fn execute(&mut self){
        let stages = &mut self.stages;

        self.pool
            .read()
            .unwrap()
            .as_ref()
            .unwrap()
            .install(move || {
                for stage in stages {
                    stage.execute();
                }
            });
    }
}

//----------------------------------------------------------

struct Stage<T: Exp6> {
    groups: Vec<Vec<T>>
}

impl <T: Exp6> Stage <T> {
    fn new(vec: Vec<T>)-> Stage<T> {

        let mut groups: Vec<Vec<T>> = Vec::new();
        groups.push(vec);

        return Stage {
            groups
        };
    }

    fn execute(&mut self) {
        use rayon::iter::ParallelIterator;
        self.groups.par_iter_mut().for_each(|group| {
            for obj in group {
                obj.run();
            }
        })
    }
}
```

Listing G.6: Experiment 6: OOP

7.7 OOP Objects

```
pub trait Exp6: Send {
    fn new(val: i128)->Self;
    fn run(&mut self);
}

pub struct Exp2048(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp2048 {
    fn new(val: i128)->Self {
        return Exp2048(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.1 + self.2 + self.3 + self.4 + self.5 + self.6 + self.7 +
     self.8 + self.9 + self.10 + self.11 + self.12 + self.13 + self.14 + self
    .15;
    }
}

pub struct Exp1912(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp1912 {
    fn new(val: i128)->Self {
        return Exp1912(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.2 + self.3 + self.4 + self.5 + self.6 + self.7 + self.8 +
     self.9 + self.10 + self.11 + self.12 + self.13 + self.14 + self.15;
    }
}

pub struct Exp1792(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp1792 {
    fn new(val: i128)->Self {
        return Exp1792(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.3 + self.4 + self.5 + self.6 + self.7 + self.8 + self.9 +
     self.10 + self.11 + self.12 + self.13 + self.14 + self.15;
    }
}

pub struct Exp1664(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp1664 {
    fn new(val: i128)->Self {
        return Exp1664(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
   val,val);
    }

    fn run(&mut self) {
        self.0 += self.4 + self.5 + self.6 + self.7 + self.8 + self.9 + self.10
   + self.11 + self.12 + self.13 + self.14 + self.15;
    }
}

pub struct Exp1536(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp1536 {
    fn new(val: i128)->Self {
        return Exp1536(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
   val,val);
    }

    fn run(&mut self) {
        self.0 += self.5 + self.6 + self.7 + self.8 + self.9 + self.10 + self.11
     + self.12 + self.13 + self.14 + self.15;
    }
}

pub struct Exp1408(pub i128,
                   pub i128,
```

```rust
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128,
                    pub i128
);

impl Exp6 for Exp1408 {
    fn new(val: i128)->Self {
        return Exp1408(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.6 + self.7 + self.8 + self.9 + self.10 + self.11 + self
    .12 + self.13 + self.14 + self.15;
    }
}

```

```
pub struct Exp1280(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp1280 {
    fn new(val: i128)->Self {
        return Exp1280(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.7 + self.8 + self.9 + self.10 + self.11 + self.12 + self
    .13 + self.14 + self.15;
    }
```

```
}

pub struct Exp1152(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp1152 {
    fn new(val: i128)->Self {
        return Exp1152(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.8 + self.9 + self.10 + self.11 + self.12 + self.13 + self
    .14 + self.15;
    }
}

pub struct Exp1024(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp1024 {
    fn new(val: i128)->Self {
        return Exp1024(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.9 + self.10 + self.11 + self.12 + self.13 + self.14 +
    self.15;
    }
}

pub struct Exp896(pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128,
                  pub i128
);

impl Exp6 for Exp896 {
    fn new(val: i128)->Self {
        return Exp896(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.10 + self.11 + self.12 + self.13 + self.14 + self.15;
    }
}

pub struct Exp768(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp768 {
    fn new(val: i128)->Self {
        return Exp768(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.11 + self.12 + self.13 + self.14 + self.15;
    }
}

pub struct Exp640(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp640 {
    fn new(val: i128)->Self {
        return Exp640(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.12 + self.13 + self.14 + self.15;
    }
}

pub struct Exp512(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

impl Exp6 for Exp512 {
    fn new(val: i128)->Self {
        return Exp512(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.13 + self.14 + self.15;
    }
}

pub struct Exp384(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);

```

```
impl Exp6 for Exp384 {
    fn new(val: i128)->Self {
        return Exp384(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.14 + self.15;
    }
}

pub struct Exp256(pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128,
                   pub i128
);
```

```

impl Exp6 for Exp256 {
    fn new(val: i128)->Self {
        return Exp256(val,val,val,val,val,val,val,val,val,val,val,val,val,val,
    val,val);
    }

    fn run(&mut self) {
        self.0 += self.15;
    }
}
```

Listing G.7: Experiment 6: OOP Objects

Bibliography

1. David Patterson John Hennessy. *Computer Architecture : A Quantitative Approach.* Elsevier, 5th edition, 2007.

2. Ghavam Shahidi. Slow-down in power scaling and the end of moore's law? In *International Symposium on VLSI Technology, Systems and Application.* IEEE, 2019.

3. Jonathan Mines. Data-oriented vs object-oriented design, March 2018. [Online; accessed 12/01/2019].

4. Jason Roberts Shameem Akhter. *Multi-Core Programming: Increasing Performance through Software Multi-threading.* Richard Bowles, 1st edition, 2006.

5. Raj Jain Mahbub Hassan. *High Performance TCP/IP Networking: Conecepts, Issues, and Solutions.* Person Education Inc., 1st edition, 2004.

6. William B. McNatt and James M. Bieman. Coupling of design patterns: Common practices and their benifits. *Computer Software and Applications Conference*, 2001.

7. Charles Scalfani. Goodbye, object oriented programming, July 2016. [Online; accessed 01/10/2020.

8. Seon Wook Kim Matthew DeVuyst, Dean Tullsen. Runtime parallelization of legacy code on a transactional memory system. In *High-Performance and Embedded Architectures and Compilers.* Association for Computing Machinery, 2011.

9. Pedro Cajaraville Diego Rodrigues Fernando Tinetti, Monica Lopez. Fortran legacy code performance optimization: Sequential and parallel processing with open mp. In *World Congress on Computer Science and Information Engineering.* IEEE, 2009.

10. Luis Ceze. Atomic operations in hardware, 2006. [Online; accessed 11/12/2020.

11. Dmitry Namiot. On lock-free programming patterns. *World Scientific and Engineering Academy and Society*, 2016.

12. Federica Rinaldi Triangles. Lock-free multithreading with atomic operations, 2019.

13. Ashwin Urdhwareshe. Object-oriented programming and its concepts. *International Journal of Innovation and Scientific Research*, 2016.

14. Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *Object-oriented programming systems, languages and applications*, 1986.

15. Tamara Munzner. Dynamic control flow polymorphism and switch statements, 2012. [Online; accessed 01/15/2020.

16. Rajive Joshi. Data-oriented architecture: A loosely-coupled real-time soa. *Real Time Innovations, Inc*, 2007.

17. Robert Nystorm. *Game Programming Patterns*. Genever Benning, 1st edition, 2014.

18. Ferdinand Majerech. Component-based entity architecture for game development. Master's thesis, Silesian University, 2015.

19. Dennis Wiebusch. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. *IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 2015.

20. Daniel Hall. Ecs game engine design. Master's thesis, California Polytechnic State University, 2014.

21. Douglas D. Hodson and Jeremy Millar. Application of ECS game patterns in military simulators. In *Int'l Conf. Scientific Computing CSC'18*, pages 14–17. CSREA Press, 2018.

22. Douglas D. Hodson and Raymond R. Hill. The art and science of live, virtual, and constructive simulation for test and analysis. *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 11(2):77–89, 2014.

23. Thomas Schaller. *The Specs Book*. Crates.io, 1st edition, 2018.

24. Bruno Feijo Marcelo Zamith, Luis Valente and Esteban Clua. Game loop model properties and characteristics on multi-core cpu and gpu games. *SBGames 2016*, 2016.

25. Aura Conci Louis Valente and Bruno Feijo. Real time game loop models for single player computer games. *IV Brazilian Symposium on Computer Games and Digital Entertainment*, 2016.

26. Dave England Abdennour El Rhalibi and Steven Costa. Game engineering for multiprocessor architecture. *Conference DiGRA 2005: Changing views-worlds in play*, 2005.

Acronyms

API Application programming interface. 35

CPU Central Processing Unit. iv, 6, 8, 9, 22, 32, 41, 43, 48, 50, 52, 58, 61, 64, 70, 71, 77, 78, 80

DOD Data-Oriented Design. iv, v, 1, 2, 3, 4, 5, 6, 9, 21, 22, 26, 30, 32, 39, 40, 41, 43, 45, 47, 48, 49, 50, 51, 52, 53, 54, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78

ECS Entity-Component-System. iv, 1, 2, 3, 5, 6, 30, 31, 32, 33, 35, 39, 76, 77, 78, 79, 80

GPU Graphics Processing Unit. 80

MESI Modified-Exclusive-Shared-Invalid. 6

OOP Object-Oriented Programming. iv, v, 2, 3, 4, 6, 9, 10, 11, 12, 14, 15, 22, 23, 26, 31, 38, 39, 40, 41, 43, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79

RAII Resource Acquisition is Initialization. 24

RAM Random Access Memory. 41

RF Radio Frequency. 14, 19

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 26–03–2020	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2018 — Mar 2020

4. TITLE AND SUBTITLE

A Study of Execution Performance
for Rust-Based Object vs Data Oriented Architectures

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Vagedes, Joseph A, 2nd Lt

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-20-M-065

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

To investigate the Data-Oriented Design (DOD) paradigm, in particular, an architecture built off its principles: the Entity-Component-System (ECS). ECS is commonly used by video game engines due to its ability to store data in a way that is optimal for the cache to access. Additionally, the structure of this paradigm produces a code-base that is simple to parallelize as the workload can be distributed across a thread-pool based on the data used with little to no need for data safety measures such as mutexes and locks. A final benefit, although not easily measured, is that the DOD paradigm produces a highly decoupled code-base, resulting in more easily maintainable and extensible code.

15. SUBJECT TERMS

Real-Time Simulations, DOD, ECS, Cache Optimization

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Douglas D. Hodson, AFIT/ENG
U	U	U	UU	352	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636 x4719 Douglas.Hodson@afit.edu