Benchmarking and Analysis of Entity Referencing Within Open-Source Entity Component Systems

---

Hugo Hansen & Oliver Öhrström
Date: 2020-06-17

Thesis:	Bachelor, 15 hp
Main field:	Computer Science
Program:	Game Development
Supervisor:	Olle Lindeberg
Examiner:	Steve Dahlskog

Abstract

Runtime performance is essential for real time games, the faster a game can run the more features designers can put into the game to accomplish their vision.

A popular architecture for video games is the Entity Component System architecture aimed to improve both object composition and performance. There are many tests for how this architecture performs under its optimal linear execution.

This thesis presents a performance comparison of how several popular open-source Entity Component System libraries perform when fetching data from other entities during iteration. An object-oriented test is also done to compare against and verify if the known drawbacks of object-orientation can still be seen within these test cases. Our results show that doing a random lookup during iteration can cause magnitudes worse performance for Entity Component Systems.

Table of contents

1. Introduction

Games are large software systems with complex logic that processes large amounts of data, all of this in real time while keeping to a strict time limit per frame. These requirements necessitates robust and fast software architectures that are easy to expand and modify during development [1].

An architecture that is rising in popularity within the games industry is the Entity Component System (ECS) pattern. This can be seen in talks at the Game Developers Conference by big companies such as Blizzard [2] and Unity [3]. The ECS pattern is even used in the new Minecraft: Bedrock Edition [4].

ECS is a software architecture built on the concept that objects should be created using composition rather than inheritance. Strict implementations of ECS will also follow data-oriented design (DOD) over object-oriented design (OOD) patterns. With these design principles ECS solves problems of complex hierarchies and with DOD maximises the usage of CPU cache [5, 6].

This thesis aims to contribute with benchmarks of some of the most popular open source ECS libraries on GitHub. The benchmarks focus on how the different libraries perform when systems that require accessing other entities are taken into account. The problem is that an entity might be moved, changed, or deleted between frames and any reference needs to take these events into consideration. Any reference made will also potentially break linear execution by requiring a memory lookup.

By writing test cases around this prospect we think that it will give a more fair comparison between the ECS libraries and an OOD solution. With this contribution there will be more available data for smaller developers looking to select a library to use in the development of their games and projects, at least for the chosen libraries. For libraries the five most popular C++ ECS repositories were looked at and three were selected based on predefined criterias. The libraries selected for the benchmark are the following: EnTT [7], EntityX [8], and ECS [9].

1.1 Problem

The problem that this thesis tries to solve is the lack of data when it comes to the benchmarking of open-source ECS libraries, specifically the area of entities referencing data outside of itself, like components from other entities. One problem often faced within game development is how to verify that an entity is still alive, this thesis will look at how this problem is solved by our chosen libraries and what effect this has on the performance.

With our thesis we seek to answer the two following questions:

1. What is the performance overhead of referencing other entities during iteration within the chosen selection of ECS libraries?

2. What do the libraries do to verify references and do they do anything to prevent mistakes caused by misuse?

1.2 Software Design Philosophies

In this section we describe and define the different software design philosophies we discuss in this study.

1.2.1 Object-oriented design

OOD is the design of software based on physical objects using inheritance. OOD builds programs around classes representing physical concepts (animal, shape, building) and deriving more complicated objects from basic versions of that object, a house is also a building. This pattern primarily focuses on the concepts of encapsulation, inheritance, and polymorphism together with being easy to structurally plan in advance.

1.2.2 Data-oriented design

DOD is the process of designing a program based on the data transformation that needs to occur for it to accomplish its goal. DOD works by thinking about what the result needs to be and then modeling the dataflow to reach the goal, logic is then created to transform this data in the correct way. Data-oriented design differs from object-oriented design in that the focus is on the data transformation being easy to understand while in OOD is the high level code structure that is the focus. This difference in philosophy has the potential to make DOD more efficient since both data and transformations can be optimized for specific architectures without changing the pattern of the program.

1.2.3 Entity Component System

ECS is a software architectural pattern commonly used in video games. The core concepts behind ECS is to build objects through shared components instead of using inheritance [10]. ECS is built around two core principles, composition over inheritance, and cache locality. ECS follows these principles by separating logic and data into different parts. The pattern contains three different architectural objects that give it its name: entities, components, and systems.

In the ECS pattern data is represented using components. A component is a small collection of plain data and contains no functions. Components describe one specific property, some examples would be position, weight, and colour.

An entity only contains an Id. This id is used to assign components to the entity. An entity can be seen as a collection of components. This layout closely corresponds to the structures of a database, where the columns represent components that are then referenced using an index/Id.

Systems are functions or objects that iterate over all entities and are responsible for logic. Systems use the available components of each entity to filter out what entities it should apply logic to. A system for moving entities would request all components of type position and velocity with the requirement being that the entity has both of those components.

Using ECS the programmer defines other objects using entities, components, and systems. This rules out the inheritance used in object-oriented-design (OOD) for a more data-oriented-design (DOD) approach.

2. Related Research

When searching for papers and theses that focus on the Entity Component System architecture you find that most works explain the architecture itself or an implementation of the architecture and compares it to an object-oriented implementation with the same functionality.

In their paper [10] Fontana et al. implements an ECS solution to tackle physical design tasks, the focus of the paper is the use of data-oriented-design as opposed to object-oriented-design in the area of electronic-design-automation. In the paper the researchers compare ECS to OOD by benchmarking the performance of ECS and OOD in physical design tasks. The results of their experiment shows that ECS is much more effective when data locality is high, while only being marginally slower in the case of low data locality.

In the paper [11] written by Lange et al. they implement wait free hashmaps for use with the ECS architecture. This implementation is benchmarked together with other implementations in order to gauge its usefulness. Something that this paper has in common with the work of Fontana et al. is that they describe the problems of OOD very well and how they are solvable using the ECS architecture.

But what we are the most interested in is comparisons between different open-source ECS implementations as this is more helpful for smaller developers that does not have the time or capacity to develop and test their own ECS solution, and what we have found when looking for just that are library benchmarking repositories on GitHub [12, 13, 14]. The three benchmarks do not cover the same libraries, with the exception being the ECS library. The test results can be seen on the GitHub pages and shows that all libraries are definitely not equal, with the difference between some being massive. There is just one problem with these benchmarks, they only do linear tests, what we mean by this is that these benchmarks do not contain test cases for when entities need to reference other entities. This is what we will focus on in this thesis in order to fill this gap.

We also want to see if the performance conclusion of the thesis written by Nilsson and Björkman [15] can be seen in our test results. In their thesis Nilsson and Björkman benchmarks an ECS and OOD solution in several systems where it was shown that for linear execution OOD can be faster than ECS when the total amount of data fits within cache while their data-oriented solution is faster when the data is larger than cache.

2.1 ECS Versus OOD

For game development the core problems with OOD (Object-Oriented-Design) is: compositional complexity, cache overhead, runtime changes to structures, and serialisation.

When creating game objects using OOD the structure usually looks something like shown in Figure 1. Here we have a class named Human that is used as a base class for two other classes, the Warrior and Monk classes.

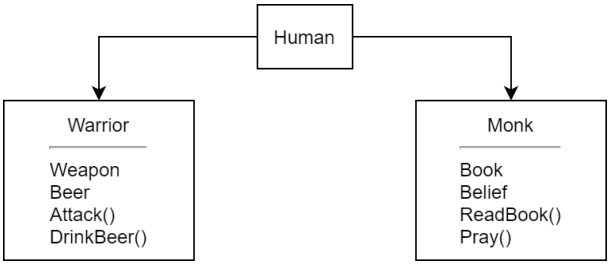

Figure 1: Simplified class diagram showing three classes: Human, Warrior, and Monk.

As more classes are added the complexity of the class hierarchy quickly becomes hard to manage and increasingly difficult to make new additions to, especially if the changes made do not perfectly conform to previous design considerations. Figure 2 shows a new class that is to be added to our existing structure.

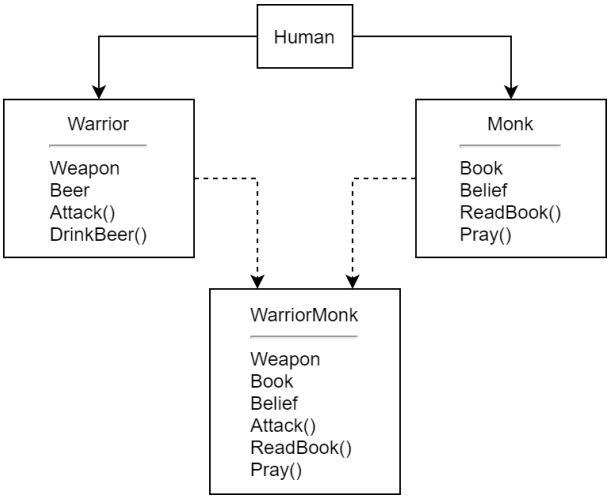

Figure 2: Simplified class diagram showing the attempted addition of a WarriorMonk class.

The dilemma of adding the new class doesn't have any current solution. Moving variables and methods from Warrior and Monk to human and then inheriting from human would work but bloating human and everything else that inherits from it is a bad idea. Another option would be to use multiple inheritance if we have access

to it. This is a better solution but doing this bloats WarriorMonk with methods it should not have access to. If we do not have access to multiple inheritance it is possible to inherit from Monk and then add the Weapon variable and Attack method to the class, but doing this we are forced to have duplicate Attack methods which makes this solution non sustainable. Similar situations and problems are described and discussed by Fontana et al. [10] and Lange et al. [11].

The problem of inheritance hierarchies also affects the program in runtime. How do you manage if a warrior needs to be changed into a monk without replacing the object with a new instance.

All these problems can be solved by the use of ECS. A possible solution using ECS could represent the traits weapon, beer, book, and belief as components. The three separate classifications of units are then constructed by giving them their corresponding components, shown in Figure 3. To implement the logic a system is created for each of the four abilities: attack, drink beer, read book, and pray, these systems will then query for entities that correspond to the requirement of their implementation, shown in Figure 3. By using this architecture the problems listed above simply do not exist.

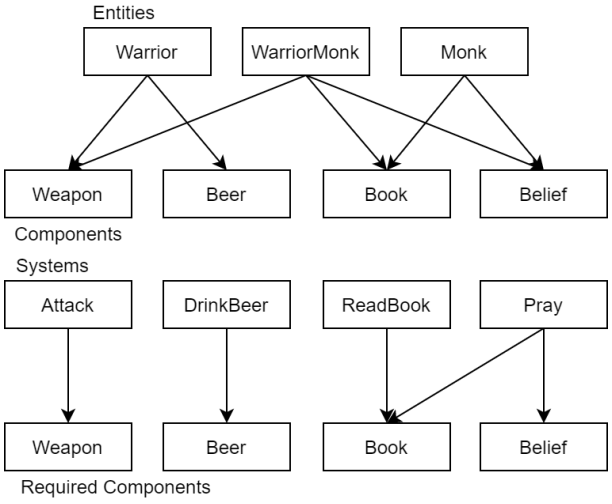

Figure 3: A solution of how to structure the objects in ECS.

By instead utilising ECS we have no need for the Human base class, we do not need to either bloat classes or have duplicate code. Another perk of ECS is that there are no structures to conform to, easing the addition of new content.

Since all data within ECS is separated from the logic serialisation can be as simple as writing all component arrays to a file, assuming that there are no reference types within the components. Networked synchronisation is also helped by ECS since each entity is just a number and each component is data related to a specific

entity id. A networked ECS packet can be as simple as saying that component C of entity E is changed to the new value V. The entity id:s can simply be synchronized when they are first created.

2.2 The Importance of Cache

With the ever increasing computational complexity of modern video games optimisations must be made to keep the game running at speed.

“Over the past decade, processor speed has increased. Memory access speed has increased at a slower pace. The resulting disparity has made it important to tune applications in one of two ways: either (a) a majority of data accesses are fulfilled from processor caches, or (b) effectively masking memory latency to utilize peak memory bandwidth as much as possible.” - Intel Corporation, 2019 [16]

ECS focuses on cache by storing the same type of data in large blocks and linearly iterating over it. Aligning memory in this way also allows the usage of efficient single instruction multiple data (SIMD) instructions and helps the processor with pre-fetching the data [16]. This approach differs from OOD in that OOD often requires objects to be stored as pointers since size might not be known at compile time and will not be the same for every object. In the worst case this means that every object might be in a separate cache line and thus each resulting in a cache miss. For operations that are CPU intensive this might not be a problem but if the operation is simple (like moving an object) this can lead to the CPU having to wait for data from RAM.

3. Method

This study aims to give smaller developers more data for when choosing an ECS library for their game or project. In this study we benchmark and analyse three different C++ ECS libraries. The benchmarks conducted focus on the performance of the different libraries when entity access is taken into account. Entities may move, change, or be deleted between ticks, due to this any reference made may break linear execution by requiring a memory lookup.

The study utilises experiment methodology, where three different ECS libraries and an OOD solution are benchmarked using test cases. The running time of each test was recorded and compiled into graphs, the results are compiled and presented in the Results section of the study. The source code of the libraries was also analysed in order to better analyse and discuss the test results.

3.1 Preparation

Windows 10 x64 was chosen as the test platform since it currently is the most popular platform for computer gaming [17, 18]. Since Windows was the chosen platform it was decided to also use the Microsoft Visual C++ compiler (MSVC), since it is a compiler native to Windows developed by Microsoft. The project was compiled in release mode with /O2x optimization enabled, since it is the highest speed optimization that MSVC supports. For project handling, CMake was used to easily link with the tested libraries. For the installation of the libraries vcpkg [19] was used where possible and direct linking with git used for those libraries not included in the package manager.

To find libraries a search was done on GitHub using the search term “entity component system OR ECS”. The results were then sorted by most stars first, this way we got the most popular Entity Component System libraries on GitHub [20].

Requirements for the libraries was the following:

- Documentation that includes installation instructions.
- Documentation that includes code for creating and updating entities.
- Needs to be written in C++.
- Needs to support Microsoft Visual C++ (MSVC) as compiler.

The top five search hits were: EnTT [7], EntityX [8], ecst [21], Kengine [22], and ECS [9]. Out of the five repositories, three were selected according to the criterias: EnTT, EntityX, and ECS. In addition to the chosen libraries a baseline solution using object-oriented design was also created and used.

For the test cases and more information on the considered libraries, see section 4. Evaluation.

3.2 Execution

While running the benchmarking application on our system, we made sure that there were as few applications running in the background as possible, so that the results would be minimally disturbed. The timing results were recorded with the use of std::chrono, to minimise measuring errors the timing was started just before the update function and stopped directly after. The results from each test were then saved to a file.

3.3 Data and Code Analysis

All test cases for all libraries (and the OOD solution) were run 50 times, the time results are saved to a CSV file between each test, each library (and the OOD solution) have seperate files for each test case.

When a test had run 50 times, the average time per entity, variance, and standard deviation was then calculated and saved for that particular test. The resulting data was put in a file used for every test case and library. This data was then put into graphs and analysed.

Once the data was produced, recorded and plotted, we compared the libraries in order to see if any conclusions could be made.

3.4 Method Discussion

The tests are currently all based on best case scenarios with no unnecessary data used anywhere, this was especially true for the OOD Implementation where additional data could potentially break the object into multiple cache lines. This does not line up with a realistic representation of entities that would be used in a game. Another thing that could be improved would be the test case scenarios. As of now they are very simple, more complex scenarios could be interesting. Because of this our experiment is not a good representation of the speed difference between OOD and ECS in a real scenario. However the tests should provide strong results for referencing, since the overhead is not based on the size of components but rather the logical structure they are stored within.

Within a real game there will be many more components making each entity much larger, this will in turn make the cache fill up on fewer entities.

To make timing as reliable as possible the tests were run multiple times and the difference between the runs were taken into consideration, due to the nature of games and that they are normally run with many tasks in the background this difference also gives us an understanding of the consistency of each library.

A further comparison that would be interesting would be cache misses, this was not done due to time constraints but would be a good continuation of the experiment.

4. Evaluation

4.1 Considered Libraries

In this section we will quickly go over the different ECS libraries that were looked into when choosing which to benchmark. We give some general information about the different libraries and how they are advertised.

4.1.1 EnTT

The EnTT library is a quite well known open source Entity Component System library that is used in both private and commercial projects, including Minecraft: Bedrock Edition. The library started off with the goal of becoming the fastest open source ECS library, when this was achieved the library was fast but featureless. After this EnTT saw big rewrites in order to become a better and more user friendly library. EnTT is a header-only library that runs C++17 [7].

4.1.2 EntityX

EntityX is an ECS library that uses C++11 features to provide type-safe component management, event delivery, etc. It was built during the creation of a 2D space shooter. The EntityX library is advertised as being “A fast, type-safe C++ Entity Component System” [8].

4.1.3 Ecst

The Ecst library is a natively multithreaded Entity Component System library written in C++14. It is a header-only library developed as a thesis. The goal of the thesis was to implement an Entity Component System library that focuses on performance and user friendly syntax [21]. Due to Ecst not currently having support for MSVC, as stated by the author in a comment on the project GitHub page [23], it is not included in the benchmarks of this thesis.

4.1.4 Kengine

Kengine or Koala engine is an ECS implementation that includes premade components and systems for features like collision, rendering, and scripting [22]. Since Kengine is an entire engine and not a library it is not included in the benchmarks of this thesis.

4.1.5 ECS

The ECS library is a lightweight Entity Component System library written in the C++11 and designed for fast prototyping. The library is quite small and is a header-only library. Although ECS is made using C++11 the author recommends the use of C++14 as it lets you use the auto type within lambda statements, easing the handling of entities. The library excels in its simplicity and easy to use structure. On the GitHub page the author mentions that there has not been any strides made in optimisation or speed, we take this into account when discussing the test results [9].

4.1.6 Object-oriented design baseline

In order to have a baseline to compare the libraries to we created a simple object-oriented approach that will also be tested. This implementation uses a base class that contains virtual functions for updating and keeps the position of the object, this class is then derived and a velocity and gameobject reference is added together with implementations for the update functions. The test case for OOD has a vector with pointers to all the gameobjects, using raw pointers for the linear and unsafe access tests, while using weak pointers from the C++ standard library for the safe access test. For each of the tests the vector is iterated over and the update is called on each gameobject in turn.

4.2 Test Setup

In this section we explain the different test cases that have been created and run. We also describe the hardware that was utilised to carry out the benchmarking. Tests focus on the impact component lookup has on the system performance of different ECS implementations. The tests are split into three cases: linear iteration, iteration with access, and iteration with access that checks if the target is still valid. The tests are run on each library and a basic OOD implementation as a baseline to compare the results with.

The test cases were run on a quite modern system, the following are the relevant hardware specifications of the system.

- CPU: 4.8GHz 8 core i9-9900K with hyperthreading enabled.
- Cache: L1, L2, and L3 are 512kB, 2MB, 16MB.
- RAM: 32GB 3200MHz.

All tests are run within the same program and use compile time type and component information and dynamic entity creation. The program is compiled in 64bit mode using Visual Studio on Windows 10 with the /O2x optimization flag set.

The program will start with creating all the entities for the test and then start a timer before executing the logic update for the test.

The components used for the tests are a position and velocity component each containing three floats. The final component contains an entity representation depending on the library, this is the target that is used for entity lookup. Each entity has a size of 32 bytes, represented by 12 bytes for position, 12 bytes for velocity and 8 bytes for the reference to the target entity.

The baseline OOD solution consists of a base class with a position and virtual update, this is then derived and velocity and logic is added. OOD referencing is made using a raw pointer for the unsafe access test and a weak pointer from the standard library is used to verify if the target is alive in the safe access test. A weak pointer does increase the size of the object to 40 bytes since it contains one extra pointer (8 bytes on x64).

- Each test is run with 1000, 10.000, 100.000, and 1.000.000 entities.
- In each test the entities have the same data layout.
- Each version of each test will be run 50 times.

The results will then be averaged and analysed. The data size of each test is 32kB, 320kB, 3,2MB, and 32MB. These sizes are without any data overhead within the storage solution of each library. For our system the tests with 1000 and 10.000 entities fit within the L1 cache while 100.000 entities will fully fit within the L3 cache, with most of it in the L2 cache. The tests with 1.000.000 entities have a data size twice the L3 cache size, see Table 1.

Table 1: The table shows how the component size of different entity amounts correlates to cache sizes.

Entity count	Data size*	Corresponding memory
1.000	32 kB	L1 cache (512 kB)
10.000	320 kB	L1 cache (512 kB)
100.000	3,2 MB	L3 cache (16 MB), (L2 can hold 2 MB)
1.000.000	32 MB	Main memory, (L3 can hold half)

*The data size does not include any extra data needed due to the library's specific data structure, this extra data is presented in Table 2, 3, and 4 in section: 6. Test Results.

4.3 Tests and Implementation

4.3.1 Test Loop

```
for (amount of test runs) do
      create N entities and give them the components
      save the entities to a temporary array

      for (each entity) do
            randomize a target entity using a fixed seed
            set that entity as target

      start timer
      run test dependent update loop
      stop timer and save result

add all results together and calculate average
```

4.3.2 Linear Test

Every entity/object will be moved to a new position based on current velocity.

```
for (each entity) do
      entity.position = entity.position + entity.velocity * delta time
```

4.3.3 Unsafe Access Test

Every entity/object will change its velocity to point towards a target entity/object without doing any verification on either the target or its data.

```
for (each entity) do
      get target from entity.target
      get position from target using library specific getter
      entity.velocity = target.position - entity.position
```

4.3.4 Safe Access Test

Every entity/object will check that its target is still alive and if it is then it will change its velocity to point towards the target entity/object position.

```
for (each entity) do
      get target from entity.target
      if (target is still alive) do
            get position from target using library specific getter
            entity.velocity = target.position - entity.position
      else
            continue
```

5. Code Analysis

In this section the following library specifics are analysed in order to be discussed in conjunction with the test results in an effort to answer the two research questions.

These parts of the libraries were analysed:

- Entity representation.
- Entity version control.
- Component storage.
- Entity referencing.
- Entity reference safety.

5.1 EnTT

EnTT uses integers to represent entities, where the integers contains both the index of the entity and the version. To check if an entity is still valid the version saved within the reference is compared to the one stored in the register, this comparison is very fast and equivalent to checking a pointer for null. Entity versions are stored in a continuous array making entity verifications an O(1) operation. For component storage EnTT uses a continuous array for each component type, to fetch a component the correct index for the specific array is retrieved from a sparse set where the entity index is used as key.

For reference safety EnTT provides debug assertion for both entities and components but has no checks on release builds, invalid entity references or components will thereby lead to undefined behaviour since the links within the sparse set will still remain active and can point to anything.

5.2 EntityX

EntityX uses an integer to represent both entity id and version. To verify that an entity reference is still valid, the version is checked against the version saved in the entitymanagers array of entities, making this operation O(1).

For component storage EntityX uses one array per component type, these arrays have one slot per entity no matter if the entity has that component or not, due to this the component arrays might not be continuous. Since there can be any number of component types EntityX uses one additional array that is used as a link to each component array. This extra array uses the index of each type as the key and a pointer to the component data as value.

To retrieve a component firstly the type index of the component is calculated, this index is used with the first array to get a pointer to the actual array of data, this data array is then indexed using the entity. Being a total of two array indexations this operation has O(1) complexity.

For runtime reference safety EntityX uses debug assertion for entity validity but this is removed for release builds making invalid references undefined behaviour since the attempts to fetch an entity will still map to a position relative to the array.

5.3 ECS

ECS differs from EnTT and EntityX in that the entity itself is responsible for its own component storage, creation, and deletion.

ECS uses unique identifiers as entity references that are then dereferenced by doing a linear search though the managers array of entities, making dereferencing O(N) complexity. For component storage ECS keeps a hashmap with component type index as key and a pointer to the component as value within each base entity, getting a component from an entity is thus O(1) complexity.

Since ECS uses searches for dereferencing it returns a null pointer if the entity does not exist, this is likely to cause the program to crash due to a segmentation fault but makes the reference validation into a null check. Beyond this ECS provides no extra safety for either debug or release builds.

The ECS library does not follow the strict definition of an Entity Component System due to it being built around object-oriented composition instead or data-orientation.

5.4 OOD

Our OOD implementation uses an array of gameobjects, stored as shared pointers, where each gameobject contains all its information within itself. For references either a raw pointer (unsafe access test) or a weak pointer (safe access test) was used. Dereferencing of both a raw pointer and a weak pointer is O(1) complexity. Since all data is stored within each object there is no operation needed to fetch that data once a reference has been acquired.

For validity checks there is no way to verify using the raw pointer while verification using the weak pointer is made using reference counting. The implementation of raw pointers has no safety checks and dereferencing a dead pointer will lead to undefined behaviour.

6. Test Results

In this section the results from the experiment are presented and analysed. Each library is represented with a graph and table showing the test results.

For each library there exists extra data that needs to be loaded into memory for the tests to be completed, the size of this data is determined in the code analysis. The data required in each library is then added to the data size of the components loaded during each test. See Table 2, 3, and 4 on the next page. This does not include the constant data required for the registers themself, only the data with linear size complexity.

The tables were made and put in this section as a way to make it easier to compare the results from the different tests with the cache level that the libraries fit within in said tests.

Table 2: Table shows the total memory needed per entity for the different tests.

Tests / Library	Linear Tests	Access Tests
EnTT	24 Bytes	64 Bytes
EntityX	24 Bytes	48 Bytes
ECS	81 Bytes*	97 Bytes
OOD	32 Bytes**	64 Bytes***

*ECS uses a hashmap with components that always needs to be loaded together with the entity id, world reference, and alive flag.
**OOD has to load the reference even in the linear test since it is part of the object.
***OOD only uses 32 bytes for unsafe referencing.

Table 3: Table shows the total data required to load during linear tests and what cache level it corresponds to for each library and each test.

Entities / Library	1.000	10.000	100.000	1.000.000
EnTT	24 kB -> L1	240 kB -> L1	2,4 MB -> L3	24 MB -> RAM
EntityX	24 kB -> L1	240 kB -> L1	2,4 MB -> L3	24 MB -> RAM
ECS	81 kB -> L1	810 kB -> L2	8,1 MB -> L3	81 MB -> RAM
OOD	32 kB -> L1	320 kB -> L1	3,2 MB -> L3	32 MB -> RAM

Table 4: Table shows the total data required to load during access tests and what cache level it corresponds to for each library and each test.

Entities / Library	1.000	10.000	100.000	1.000.000
EnTT	64 kB -> L1	640 kB -> L2	6,4 MB -> L3	64 MB -> RAM
EntityX	48 kB -> L1	480 kB -> L1	4,8 MB -> L3	48 MB -> RAM
ECS	97 kB -> L1	970 kB -> L2	9,7 MB -> L3	97 MB -> RAM
OOD*	64 kB -> L1	640 kB -> L2	6,4 MB -> L3	64 MB -> RAM

*Unsafe access is half the size for OOD making 10.000 fit within the L1 cache.

6.1 EnTT

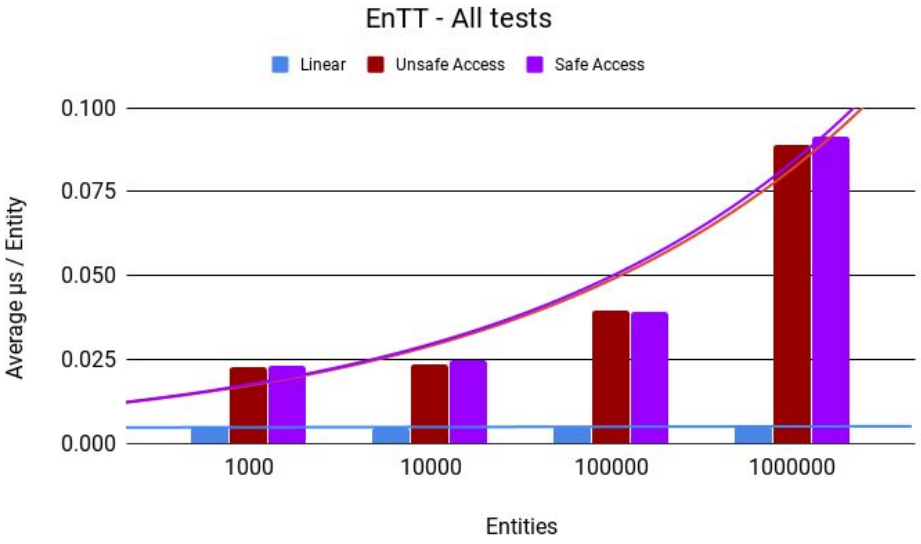

Figure 4: The graph shows average time per entity for each of the tests (Linear, Unsafe Access, and Safe Access). The lines are exponential lines that show the growth for the corresponding test.

Table 5: The table shows the data related to EnTT test runs.

Test	Entities	Average Time / Entity (µs)	σ in %
Linear	1000	0.004906	8.79
Linear	10000	0.0047992	3.01
Linear	100000	0.00500804	7.45
Linear	1000000	0.00512007	3.22
Unsafe Access	1000	0.022634	5.13
Unsafe Access	10000	0.0236504	1.62
Unsafe Access	100000	0.039511	5.76
Unsafe Access	1000000	0.08884	2.64
Safe Access	1000	0.022958	2.36
Safe Access	10000	0.024666	1.26
Safe Access	100000	0.039306	5.09
Safe Access	1000000	0.0913204	5.09

As Figure 4 and Table 5 shows, EnTT has an O(N) complexity on its linear test and this is supported by the code analysis. For the linear test EnTT also accesses memory linearly, this allows the CPU to prefetch data and prevents cache misses even for larger entity counts. Unsafe access and safe access should according to code analysis have a complexity of O(N), this is the case for 1.000 and 10.000 but it quickly increases after those tests. This can be explained by the data size since a reference in EnTT requires several jumps in memory and for 1.000.000 entities the data no longer fits in CPU cache, see Table 4. Safe and unsafe shows no significant

difference between each other, as it is within the standard deviance. EnTT had an average standard deviation of 4,28%.

6.2 EntityX

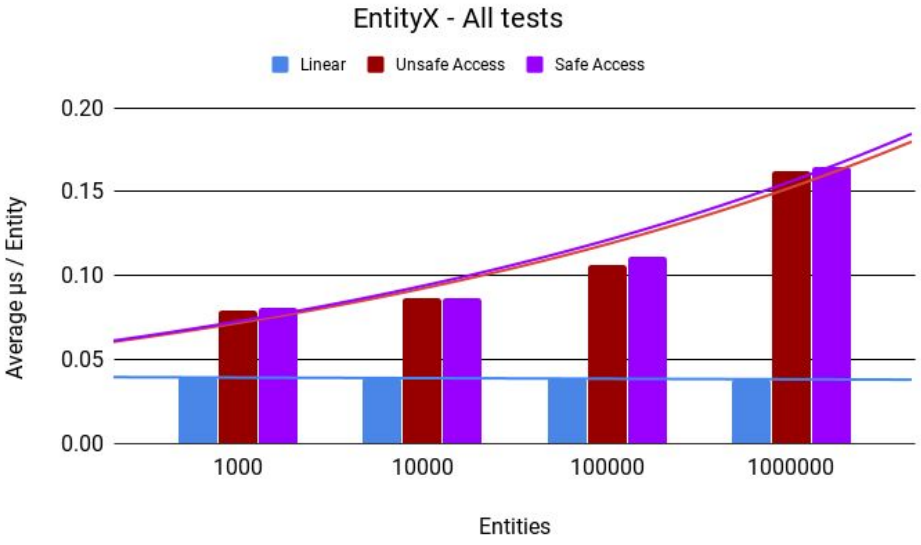

Figure 5: The graph shows average time per entity for each of the tests (Linear, Unsafe Access, and Safe Access). The lines are exponential lines that show the growth for the corresponding test.

Table 6: The table shows the data related to EntityX test runs.

Test	Entities	Average Time / Entity (µs)	σ in %
Linear	1000	0.03958	2.87
Linear	10000	0.038494	1.15
Linear	100000	0.038715	2.64
Linear	1000000	0.038351	0.95
Unsafe Access	1000	0.07923	2.73
Unsafe Access	10000	0.0864666	2.79
Unsafe Access	100000	0.106276	5.49
Unsafe Access	1000000	0.161994	1.63
Safe Access	1000	0.08049	1.18
Safe Access	10000	0.0864236	1.71
Safe Access	100000	0.111046	4.8
Safe Access	1000000	0.164747	1.86

As Figure 5 and Table 6 shows, EntityX looks to have O(N) complexity on the linear test, this is supported by the code analysis. EntityX stores components linearly within memory, this allows the CPU to prefetch the data during linear execution, this prevents cache misses. Unsafe access and safe access looks to have O(N) complexity for the 1.000 and 10.000 access tests, this is what is expected from the code analysis. 100.000 and 1.000.000 entities shows an increase over the expected

linear scale, this is explained by the data size being larger than cache, for a reference to be done firstly the correct component has to be found and then the correct index, this equates to two pointer dereferences and two indexations causing up to four cache misses. Safe and unsafe access shows no significant difference between each other, as it is within the standard deviance. EntityX had the best average standard deviation of all libraries (and OOD) at 2,48%.

6.3 ECS

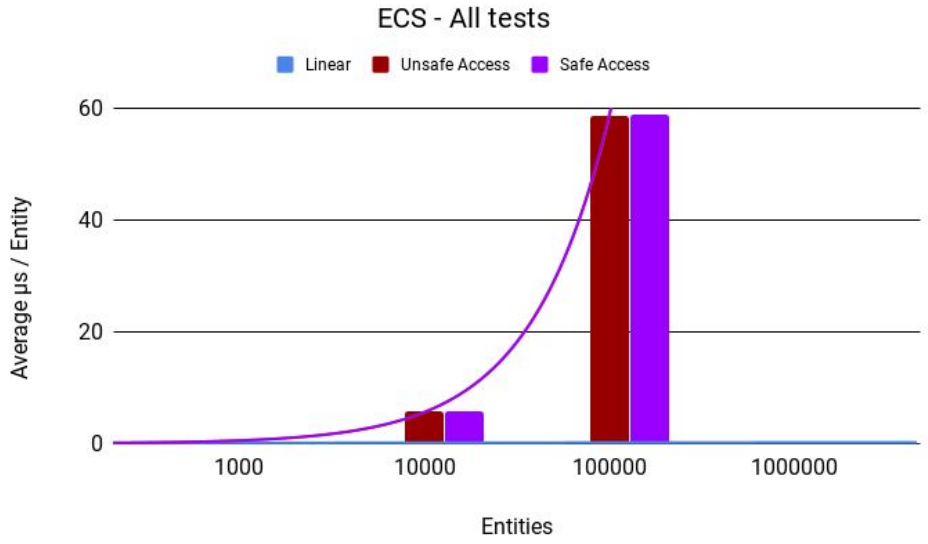

Figure 6: The graph shows average time per entity for each of the tests (Linear, Unsafe Access, and Safe Access). The lines are exponential lines that show the growth for the corresponding test.

Table 7: The table shows the data related to ECS test runs.

Test	Entities	Average Time / Entity (µs)	σ in %
Linear	1000	0.123272	18.93
Linear	10000	0.149326	8.84
Linear	100000	0.172149	2.24
Linear	1000000	N/A	N/A
Unsafe Access	1000	0.43055	5.15
Unsafe Access	10000	5.74433	3.3
Unsafe Access	100000	58.5479	1.91
Unsafe Access	1000000	N/A	N/A
Safe Access	1000	0.42998	3.58
Safe Access	10000	5.78322	2.35
Safe Access	100000	58.8688	3.78
Safe Access	1000000	N/A	N/A

As Figure 6 and Table 7 shows, ECS has an O(N) complexity on the linear test while both access tests have O(N^2) complexity, both of these are supported by code

analysis. Unsafe access and safe access has no measurable difference. ECS had an average standard deviation of 5,56%.

6.4 OOD

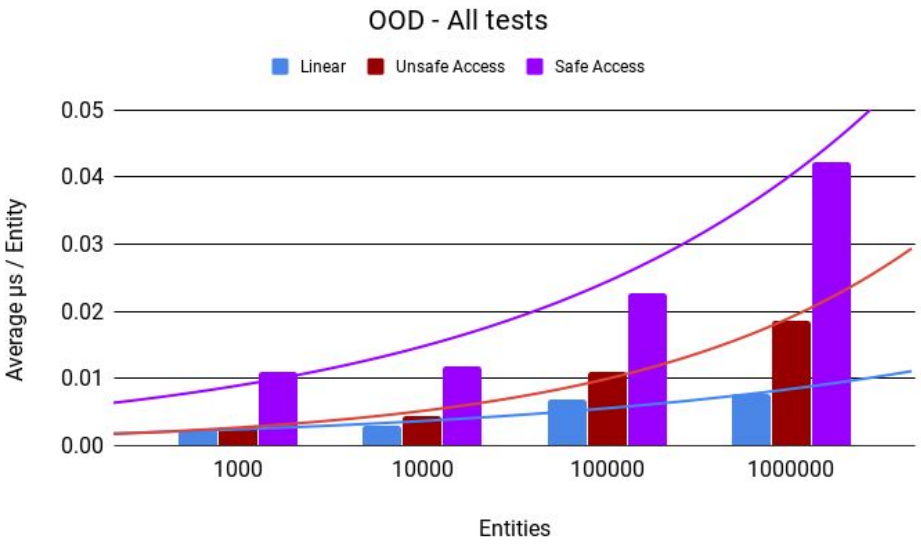

Figure 7: The graph shows average time per entity for each of the tests (Linear, Unsafe Access, and Safe Access). The lines are exponential lines that show the growth for the corresponding test.

Table 8: The table shows the data related to OOD test runs.

Test	Entities	Average Time / Entity (µs)	σ in %
Linear	1000	0.002358	28.91
Linear	10000	0.0029544	16.78
Linear	100000	0.00682028	10.66
Linear	1000000	0.00765473	4.76
Unsafe Access	1000	0.002744	16.93
Unsafe Access	10000	0.0044648	7.96
Unsafe Access	100000	0.0109687	7.87
Unsafe Access	1000000	0.0185698	2.06
Safe Access	1000	0.01103	6.3
Safe Access	10000	0.0117924	7.9
Safe Access	100000	0.0226739	31.03
Safe Access	1000000	0.0421883	1.33

As Figure 7 and Table 8 shows, OOD has O(N) complexity with 100.000 and 1.000.000 showing that memory layout of our OOD implementation makes it hard for the CPU to prefetch data leading to cache misses when the data size is larger than cache. Unsafe access has a marginal overhead over linear, explained by the small logical overhead of a pointer dereference and for the larger test sizes one additional cache miss. Safe access has a significant overhead over both of the

other tests, this can be explained by the weak pointer requiring several memory jumps, one for each of the two reference counters and a third for the actual data leading to a potensial three cache misses. Weak pointers also require one extra pointer for the control block that takes up 8 extra bytes and the control block itself, making the cache run out faster, see Table 4. The OOD solution had the worst average standard deviation at 11,87%.

6.5 Comparison

In this section we present graphs comparing the different solutions. Each test is presented in its own graph, ECS has been excluded from these graphs, due to being magnitudes slower, see Figure 6 and Table 7.

For all tests except the linear tests with 100.000 and 1.000.000 entities OOD ran the fastest with ECS (not shown in the graphs) being the slowest in all tests. EnTT was the second fastest after OOD in all tests except the high entity count linear tests (see Figure 8) where EnTT was the fastest. EntityX had the most consistent runtime amongst the different entity amounts but also ran consistently slower than both OOD and EnTT.

The reason for OOD being so fast within these tests is due to the small data size and may not necessarily be representative of how it would perform within a medium-large size game.

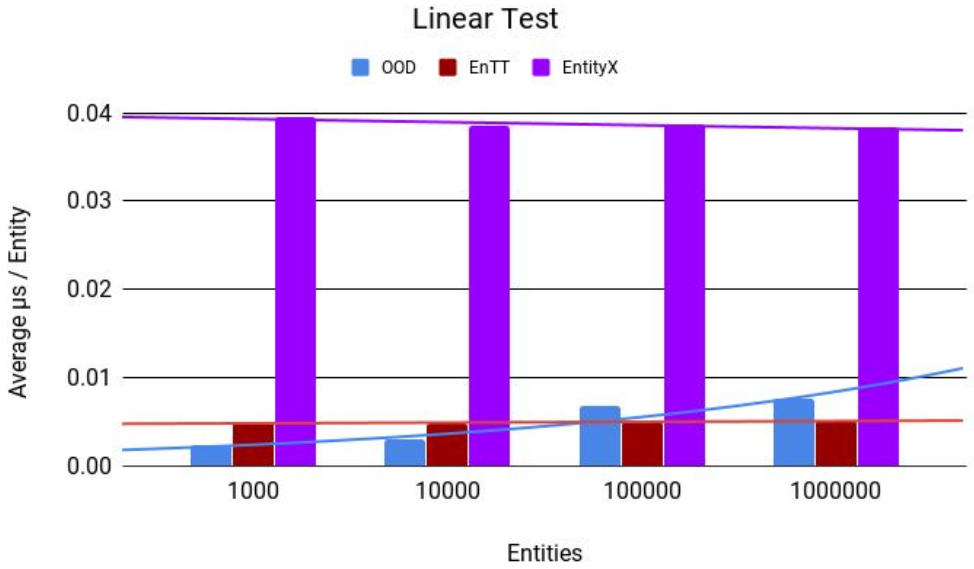

Figure 8: The graph shows average time per entity of all the libraries for the linear test (excluding ECS). The lines are exponential lines that show the growth for the corresponding library.

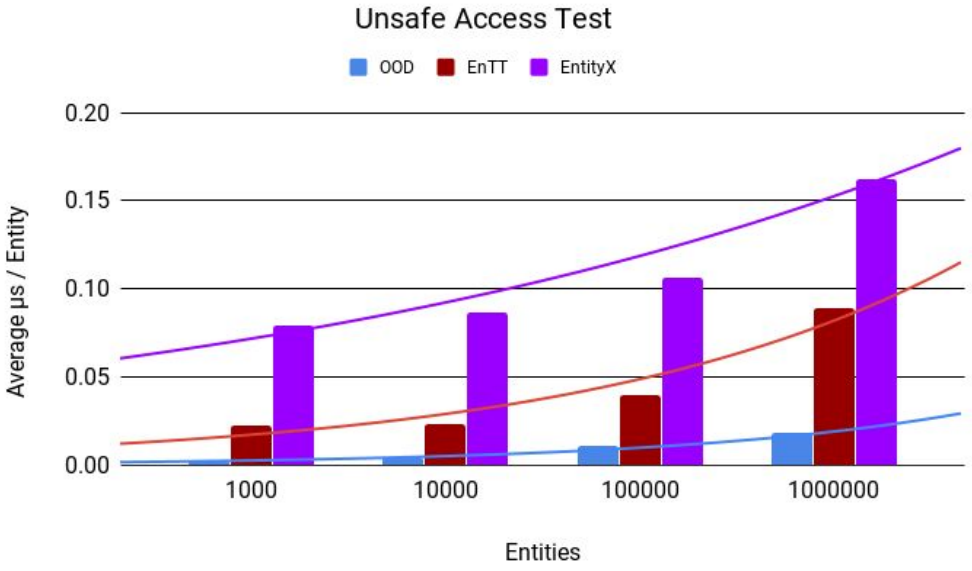

Figure 9: The graph shows average time per entity of all the libraries for the unsafe access test (excluding ECS). The lines are exponential lines that show the growth for the corresponding library.

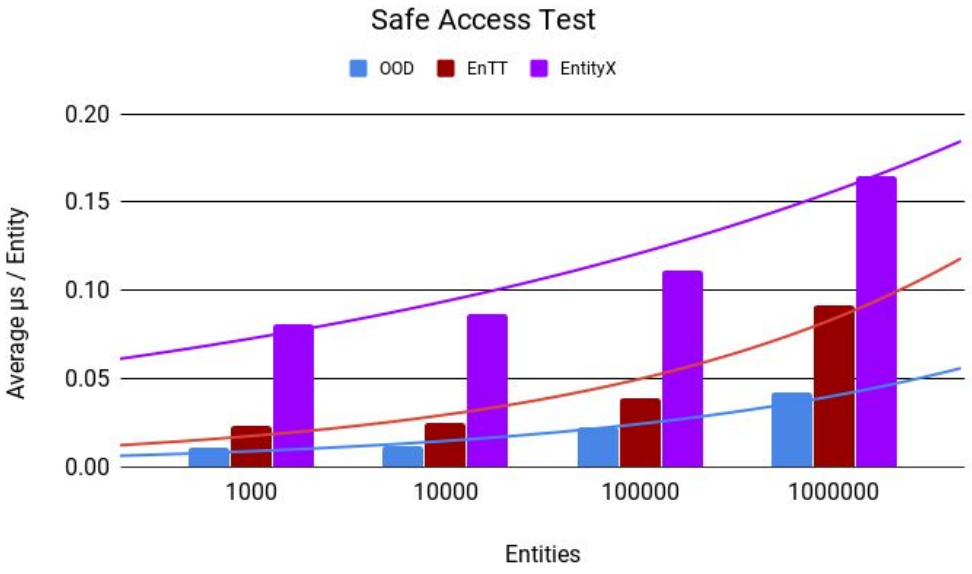

Figure 10: The graph shows average time per entity of all the libraries for the safe access test (excluding ECS). The lines are exponential lines that show the growth for the corresponding library.

7. Analysis

7.1 Library Analysis/Discussion

From our code analysis and results we can conclude that in all cases except for the ECS library the increase in time per entity is due to cache effects, ECS is due to having O(N^2) referencing complexity.

Object-oriented design is the classical programming paradigm, it is easy to use and can be made relatively efficient. OOD lacks is modularity and modifiability, and for a game that needs complicated objects, or objects that change during runtime OOD can quickly become complicated and lose efficiency. Therefore we start off with OOD and move towards EnTT which is closest related to DOD. (OOD may be perfectly fine for smaller projects, and larger projects that do not change much over time)

In our test OOD had the quickest linear runtime while the entity count was low but scaled worse than EnTT for large amounts of data. In our test the OOD objects were smaller than the L1 cache of our test system on 1000 and 10.000 entities and this shows in the results. For 100.000 entities it fit within the L3 cache, while 1.000.000 entities no longer fit within cache, making 1.000.000 the final linear time. This caching effect corresponds to the one shown in the tests by Nilsson, D. and Björkman, A. [15] and explains why OOD is faster while its data fits within cache.

OOD had the fastest references of all our tests, likely due to fewer operations needed to go from reference to data (being just a pointer dereference), though our test also shows that the basic stl weak pointer has a large overhead and is likely unfit for use as reference even though it is faster than the all the ECS libraries. For the references caching effects are also very noticeable with the 1.000.000 tests having a large overhead compared to the tests with less entities, see Figure 7. If OOD is chosen as a solution our recommendation is to go with a different reference design than week pointer to maximise performance.

The library ECS has taken one step away from OOD where it still keeps the core philosophies of objects and classes while changing the hierarchical architecture to a composition based one. This has the benefit of being more modular and modifiable while still being easy to use. The downside shown in this experiment is speed, due to ECS not storing data continuously within memory it does not utilize the cache as efficiently as the other libraries. ECS references are extremely slow due to being a linear search through its entity list, making the search in of itself worst case O(N^2) time complexity.

EntityX follows a more data-oriented approach with each component being grouped together in an array while wrapping it in an OOD shell, this makes it very easy to use for programmers that are used to OOD. EntityX also has an average speed and even with a two times overhead for referencing it is not a catastrophe if used sparingly. The overhead for entity lookup comes from having to jump to several locations within memory to fetch each required component, most likely leading to a cache miss per component, this is what we suspect makes the test results so consistent. For the tests with sizes larger than cache it can clearly be seen to affect the lookup time even more, see Figure 5 and Table 4.

The way EntityX stores components is to keep one array for each component type with a size equivalent to the capacity of the entities, this has the effect that fetching a component that an entity does not hold will still return a valid component though it will not be in a defined state, this can make debugging hard since no traceable error exists. This makes it very important to make sure that the entity is alive/exists before accessing its data. EntityX also has debug assertions for this specific case. EntityX was the implementation with the most consistent runtime amongst our tests with an average standard deviation of 2,48%.

EnTT is a pure data-oriented approach with no OOD objects anywhere, this can make it less user friendly and more error prone since it is a new way of thinking about programming. EnTT has the same referencing issues as EntityX where an unverified reference will likely still return valid data even if the entity or component is deleted, how to verify this data is more complicated though due to the data oriented approach. Like EntityX, EnTT has debug assertions for all references to try and catch errors in development but no runtime checks.

What EnTT loses in ease of use it gains in efficiency, being highly modular and memory efficient it also has an almost perfect usage of cache during linear execution, this due to EnTT not keeping empty slots within any component array and grouping data based on use cases. EnTT has the fastest linear execution of all of our tests when using large amounts of entities, though it has the second largest overhead for referencing amongst all our tests results. EnTT loses some of its linear efficiency on low amounts of data since its data management has overhead when starting iterations.

The reason for EnTT having large overhead in referencing is due to the amount of steps needed to go from an entity to the corresponding components. Since EnTT does not store every component in equally sized arrays a lookup in the component sparse set has to be made to get the correct index, this would make this the least efficient referencing scheme amongst the tested libraries if ECS had not done a linear search. For the tests of 100.000 and 1.000.000 entities it can also clearly be

seen that having to fetch an entity not currently loaded in the L1 cache has a large overhead for this library, see Figure 4 and Table 4.

Both EnTT and EntityX use version numbers to verify that an entity is still alive, this turns the check into a single integer comparison of O(1) complexity. ECS uses a search for a unique id turning the validity check into a linear search which is O(N), this is because of the integer comparison done per search step. Since all three of these solutions consist of just a single integer comparison the overhead for the validity check is minimal, and in the case of the ECS library nonexistent since it has to do the comparison anyways as part of the search.

Why EntityX is much slower than EnTT even though they are conceptually very similar is something we do not have an answer for, though it is consistent with the tests shown in the benchmarks made by Beimler [14]. Why this difference exists and why it is so large would make interesting future research.

7.2 Test Discussion

The four test sizes correspond to a total data amount of 32kB, 320kB, 3,2MB, and 32MB, see Table 1. These sizes fulfill all the different cache scenarios, having the cache sizes 512kB, 2MB, 16MB in the system used. This is shown in our results by most lookups taking more time after 100.000 entities since the complete data no longer would fit within the L1 cache, as can be seen in the graphs. The reason we stopped at 1.000.000 entities was because of time concerns.

If we would have used larger entities with more components these cache effects would have become apparent much earlier and would lead to more realistic test results, as entities usually would be much larger in the setting of games development.

8. Conclusion

All the tested Entity Component Systems in the experiment had a significant overhead for doing entity lookups during an update loop. Both EnTT and EntityX have similar overheads, if entity referencing is needed within these implementations it should be kept to a minimum, referencing entities that are currently located within cache is also shown to be much faster than if it has to be loaded from RAM.

The library ECS does not have any good solution to entity referencing due to using linear search to find the equivalent entity, this makes entity lookup a non acceptable action during runtime and if this is needed a pure OOD approach or a custom solution is likely a better alternative.

For reference safety both EnTT and EntityX have the same systems of entity versions and debug assertions while both ECS and our OOD approach requires pointer validations. For runtime reference errors only ECS is likely to crash while all others will cause undefined behaviour, since our tests show that validating the existence of data has a negligible overhead we conclude that this should always be done to negate this risk.

If speed and modularity are important, then EnTT seems likely to be the best option, if the project is more static and predefined, OOD is a tried and simple solution and likely the fastest. For projects in between these two extremes EntityX is likely the best option. With the ECS library having O(N) entity reference it does not look like a fitting solution to projects where that feature is needed and should at best be used for prototyping, unless the library is modified to make referencing O(1).

The tests also showed that none of our tested libraries had any noticeable penalty for verifying the validity of a referenced entity, this makes our conclusion that it should always be done to prevent data corruption or crashes.

8.1 Future work

It would be interesting to look at how adding and removing components and entities during runtime affects the results and to see how severely it affects the performance of an OOD solution. We suspect that doing runtime changes could be devastating for the performance of an OOD solution while not affecting the results of the ECS solutions very much.

Another thing that could make interesting future research would be to determine why EnTT and EntityX perform so differently while they are so conceptually similar.

It would also be interesting to have more complex test scenarios with more components and systems as this would likely have changed up the results of the experiment.

Testing on more compilers and operating systems would prove interesting when not looking from the perspective of games and game development.

Finally other kinds of devices could be interesting, such as phones, consoles, and microcontrollers, to see how different hardware architectures affect the result.

9. References

[1] Blow, J. (2004). Game Development: Harder Than You Think. *Queue* 1, (10), 28–37. DOI:https://doi.org/10.1145/971564.971590

[2] Ford, T. (2017). Overwatch' Gameplay Architecture and Netcode. Game Developers Conference 2017 (GDC). Retrieved 2020 April 17 from https://www.gdcvault.com/play/1024001/-Overwatch-Gameplay-Architecture-and

[3] Johansson, T. (2018) Job System & Entity Component System. Game Developers Conference 2018 (GDC). Retrieved 2020 April 17 from https://www.gdcvault.com/play/1024839/Job-System-Entity-Component-System

[4] Mojang. (2020) Minecraft Attributions. Retrieved 2020 April 17 from https://minecraft.net/en-us/attribution/

[5] Collin, D. (2010). Introduction to Data Oriented Design. Retrieved 2020 April 17 from https://www.slideshare.net/DICEStudio/introduction-to-data-oriented-design

[6] Acton, M. (2014). Data-Oriented Design and C++. The C++ Conference 2014 (CppCon). Retrieved 2020 April 17 from https://www.youtube.com/watch?v=rX0ItVEVjHc and https://github.com/CppCon/CppCon2014/tree/master/Presentations/Data-Oriented%20Design%20and%20C%2B%2B

[7] Caini, M. (2020). EnTT (6006917) [Open source ECS-library]. Retrieved 2020 April 19 from https://github.com/skypjack/entt

[8] Thomas, A. (2020). EntityX (13b01f9) [Open source ECS-library]. Retrieved 2020 April 19 from https://github.com/alecthomas/entityx

[9] Bloomberg, S. (2019). ECS (5f74e19) [Open source ECS-library]. Retrieved 2020 April 19 from https://github.com/redxdev/ECS

[10] Fontana, T., Netto, R., Livramento, V., Guth, C., Almeida, S., Pilla, L., & Güntzel, J. L. (2017, March). How Game Engines Can Inspire EDA Tools Development: A use case for an open-source physical design library. In *Proceedings of the 2017 ACM on International Symposium on Physical Design* (pp. 25-31).

[11] P. Lange, R. Weller and G. Zachmann, "Wait-free hash maps in the entity-component-system pattern for realtime interactive systems," 2016 IEEE 9th

Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), Greenville, SC, 2016, pp. 1-8.

[12] Schmierer, L. (2017) ecs_bench [Entity component system benchmarks]. Retrieved 2020 April 19 from https://github.com/lschmierer/ecs_bench

[13] Papari, A. (2016). entity-system-benchmarks [Entity component system benchmarks]. Retrieved 2020 April 19 from https://github.com/junkdog/entity-system-benchmarks

[14] Beimler, A. (2019). ecs_benchmark [Entity component system benchmarks]. Retrieved 2020 may 18 from https://github.com/abeimler/ecs_benchmark

[15] Nilsson, D. and Björkman, A. (2019). Profiling a Prototype Game Engine Based on an Entity Component System in C++. Degree project, Malmö University, Malmö. Retrieved 2020 May 18 from https://antonidag.github.io/essay.pdf

[16] Intel Corporation. (2019). *Intel® 64 and IA-32 Architectures Optimization Reference Manual(248966-042b)*. Retrieved 2020 April 15 from https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

[17] Valve Corporation. (2020). Steam Hardware & Software Survey. Retrieved 2020 May 13 from https://store.steampowered.com/hwsurvey

[18] Valve Corporation. (2020). Steam Hardware & Software Survey: April 2020. Retrieved 2020 May 13 from https://web.archive.org/web/20200131203537/https://store.steampowered.com/hwsurvey

[19] Microsoft. (2020). vcpkg: a C++ package manager for Windows, Linux, and MacOS. Retrieved 2020 May 13 from https://docs.microsoft.com/en-us/cpp/build/vcpkg?view=vs-2019

[20] GitHub, Search Query. Retrieved 2020 May 9 from https://github.com/search?l=C%2B%2B&o=desc&q=entity+component+system+OR+ECS&s=stars&type=Repositories

[21] Romeo, V. (2018). ecst (b3c42e2) [Open source ECS-library]. Retrieved 2020 April 19 from https://github.com/SuperV1234/ecst

[22] Phister, N. (2020). kengine(2874d15) [Open source ECS-implementation]. Retrieved 2020 April 19 from https://github.com/phisko/kengine

[23] Romeo, V. (2018). Setting up ecst - needed modules, compile issues [Issue Comment]. Retrieved 2020 April 19 from https://github.com/SuperV1234/ecst/issues/15#issuecomment-267791224