Utrecht University

XVR simulation

ENCODE: From Object-Oriented to Data-Oriented, an Automatic ECS Conversion Designer and Education Tool

Anne van Ede
ICA-5844126

Master Thesis
Game and Media Technology

Department of Information and Computing Sciences
Utrecht University, The Netherlands
Examiners: Dr. Ing. Jacco Bikker and dr. Roland Geraerts
External Supervisor: Taco Petri (XVR)

July 25, 2021

Abstract

Data-Oriented Design (DOD) and its substrategy Entity Component System (ECS) are promising and more performant alternatives to Object-Oriented Design (OOD), especially for the game industry However, converting an existing project from OOD to ECS is challenging. Additionally, little scientific research is available on DOD and ECS.

This thesis catalogues information on ECS itself and its design strategies. The results show that an ECS design should be created around how the data is used and to what domain it belongs. We also formulate basic requirements for the visualization of those designs. Showing the connection within an ECS design and between the original OOD and the generated ECS design is essential.

This thesis proposes a technique to convert a design from OOD to DOD. This conversion technique uses *static program analysis* to extract the source design structure, which is then stored in an intermediate data model. From this intermediate model, a design is created. This design can be tailored to the project requirements with the use of planning profiles.

We demonstrate the effectiveness of this conversion technique by creating an OOD to ECS conversion tool named ENCODE. The created designs are compared to manually created designs. The results show that ENCODE is capable of creating designs with entities, components, and systems that correspond with the original OODs and the manual designs.

Contents

1 Introduction **4**
- 1.1 The Goal of this Thesis . . . 4
- 1.2 Structure . . . 4
- 1.3 Contributions . . . 5

2 Preliminaries **6**
- 2.1 CPU Cache and Latency . . . 6
- 2.2 OOD vs. DOD . . . 7
  - 2.2.1 Prefetching . . . 9
  - 2.2.2 Maintenance and Flexibility . . . 9
- 2.3 ECS . . . 10
  - 2.3.1 Entities and Components . . . 10
  - 2.3.2 Systems . . . 11
  - 2.3.3 Vectorization . . . 11
  - 2.3.4 Padding . . . 12
- 2.4 ECS Framework Implementations . . . 12
  - 2.4.1 Big Array-based . . . 12
  - 2.4.2 Sparse Sets-based . . . 13
  - 2.4.3 Archetype-based . . . 14
- 2.5 ECS Conclusion . . . 15

3 Related Work **16**
- 3.1 Manual OOD to DOD Conversion . . . 16
- 3.2 Language Conversion . . . 16
- 3.3 Compilers and Code Analysis . . . 17
- 3.4 Planning Profiles . . . 17
  - 3.4.1 Proposed Conversion Method . . . 17

4 ECS Insights **18**
- 4.1 Interview Setup . . . 18
  - 4.1.1 Selection . . . 18
  - 4.1.2 Question Topics . . . 19
- 4.2 Results . . . 20
  - 4.2.1 Advantages and Disadvantages of ECS . . . 20
  - 4.2.2 Visualization Guidelines . . . 21
  - 4.2.3 Splitting and Merging Components . . . 21
  - 4.2.4 Other Design Considerations . . . 23
- 4.3 Summary . . . 24
  - 4.3.1 ECS and ECS Design . . . 24
  - 4.3.2 Visualization . . . 25

5 Implementation Details **26**
- 5.1 Conversion Method . . . 26
- 5.2 Static Program Analysis . . . 26
- 5.3 Design Planning . . . 27
  - 5.3.1 Planning Parameters . . . 27
  - 5.3.2 Planning Profiles . . . 28
- 5.4 Plan Visualization . . . 29
  - 5.4.1 Tool Overview . . . 29
  - 5.4.2 OOD and ECS Code Comparison . . . 29

5.4.3 ECS Design Relationships 31
5.5 Concluding Results 31

6 Experimentation Results **32**
6.1 Setup 32
6.2 Parameter Influence 33
6.3 Sample Projects Comparison 33
6.3.1 Components and Entities 33
6.3.2 Systems 35
6.3.3 XVR's OS Performance 36
6.3.4 Concluding Results 36

7 Discussion **37**
7.1 ECS Knowledge Base 37
7.2 Conversion Method 37
7.3 ENCODE Evaluation 38
7.4 Concluding Remarks 38

8 Future Work **39**

9 Conclusion **40**

1 Introduction

Object-Oriented Design (OOD) is currently the dominant design strategy for games. However, it is not always the optimal choice when it comes to performance [17, 14, 8]. OOD yields a data structure based on real-world relationships in the data, ignoring hardware characteristics, most importantly caches. While CPU performance keeps improving, memory and caches lag behind. This gap results in the CPU not receiving data as fast as it can process it, leaving it idle more often than necessary [1]. The data throughput bottleneck can significantly impact application performance, especially with multi-core processors creating an ever-increasing demand for data. This problem will only grow more relevant in the future.

Data-Oriented Design (DOD) is becoming increasingly popular in the game development industry as an alternative to OOD [8]. DOD is a design approach that considers the characteristics of, among others, the cache for greater performance. Instead of structuring data based on real-world relationships, it arranges data based on how and when it is accessed. By storing data that is used simultaneously close together in memory, memory access is more efficient, and the overall performance of software increases [7, 14, 17, 1]. Besides performance, flexibility, maintainability and unit testing all benefit from choosing DOD over OOD [7, 14].

Entity Component System (ECS) is a sub-strategy of DOD that has been adopted by game companies and game engines alike [22, 4]. Where DOD is the general paradigm of considering the data and hardware first, ECS provides a practical structure for the software to follow to achieve a Data-Oriented Design. For all the benefits, DOD and in the same line ECS has not been the subject of much research. There is little literature on DOD, and research on ECS is almost non-existent. This leaves a large gap in research on ECS and ECS design.

In addition, applying ECS in a project is challenging. ECS is fundamentally different from OOD and requires considerable practice before it can be successfully applied [14, 8]. To our knowledge, no solution exists that can substantially aid in this conversion. Neither does a tool exist that helps bolster understanding of the technique and create an optimal design.

1.1 The Goal of this Thesis

This thesis has two main goals:

The first is to fill the gap in literature on ECS and ECS design. Because of the lack of research on DOD and more specifically ECS, knowledge is limited. This thesis aims to contribute general information on ECS. This knowledge includes information on characteristics of ECS, design considerations and visualization requirements.

The second objective of this thesis is to introduce a tool that analyzes the OOD input and automatically creates an ECS design. We aim to create the tool in such a way that it can aid in converting from OOD to ECS design while educating the user on the design choices.

1.2 Structure

We will first introduce the reader to DOD and ECS in Section 2. Next we give an overview of related research and methods in Section 3. We then discuss the insights on ECS, ECS design and visualization in Section 4. Section 5 introduces a novel conversion method for converting a design from OOD to DOD. It proves the methods validity by implementing it for conversion between OOD and ECS. We will then show the results of the tool in Section 6 and discuss them in Section 7. Finally, we conclude our findings and propose future work in Sections 8 and 9.

1.3 Contributions

This thesis has the following contributions:

1. We submit insights from experts on ECS characteristics and ECS design considerations to fill the current gap in literature surrounding ECS.
2. We introduce basic criteria for ECS design visualization that can become the basis for standardized visualization of such designs.
3. We introduce a novel conversion method that uses a combination of static program analysis, intermediate data models and planning profiles to analyze and convert a design from one design strategy to another.
4. We introduce ENCODE (ECS Conversion Designer and Education tool)[1], a tool that applies the proposed conversion method to convert an OOD to an ECS design. The gathered insights on ECS, ECS design and visualization are used to aid and educate users on conversion to ECS.

[1]ENCODE can be found at `https://github.com/AnneVanEde/encode`

2 Preliminaries

DOD aims to improve application performance by leveraging data access patterns. In this section, we will explain how these access patterns affect software performance. We do this by explaining the basics of memory latency and CPU caches. We will introduce the difference in how OOD and DOD data is stored in memory. Lastly, we will discuss ECS, a popular DOD approach and we will explain three different implementations.

2.1 CPU Cache and Latency

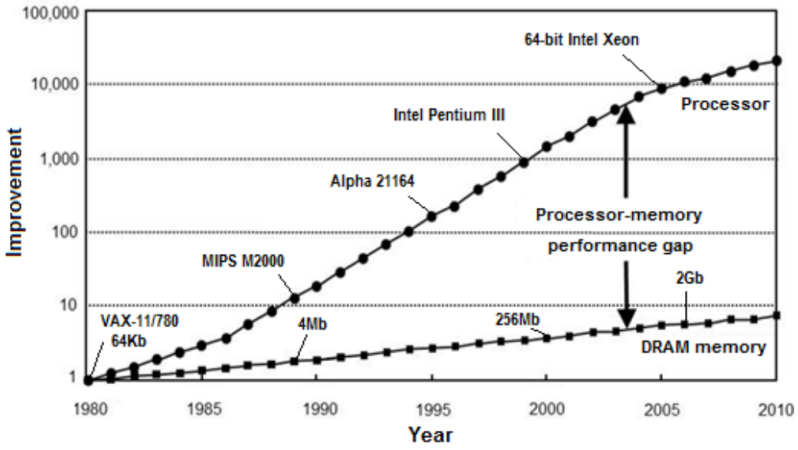


Figure 1: Yearly improvement of processor and DRAM memory speeds, for a time period of three decades. Processor speeds are gathered by execution time of standard SPECint benchmark programs. Memory speeds are memory access times. [6].

When computers were first developed, components like the CPU, memory and mass storage were developed together. Because of this, their performance was balanced. Memory, for example, could provide data almost as fast as the CPU could process it [5]. Nowadays, this is very different. CPU performance (in GFlops) has skyrocketed, and with multiple cores on a single die, the demand for data has only increased.

However, memory bandwidth and latency have not scaled in the same manner due to practical concerns [1]. The more complex motherboards and energy consumption required make the memory both impractical and prohibitively expensive. Because of this, memory bandwidth and latency have lagged behind, creating the gap that can be seen in Figure 1. Memory latency is more than 150 times greater than CPU clock speed, meaning that the CPU has to wait for 150 cycles before the requested data is fetched from RAM. Since most CPU operations take only a few cycles, this is a significant bottleneck for CPUs. This problem will increase in relevance in the future when more cores are put on a single CPU die if the total bandwidth does not improve.

Modern hardware tries to mitigate this problem by introducing *cache*. This is a small piece of memory that can be accessed much faster than RAM but only stores a small subset of the data in RAM. When data is requested from RAM, instead of just the byte the CPU needs, the memory subsystem pulls a chunk of contiguous data from memory, a *cacheline*. It stores the whole cacheline in the cache, and the CPU can process the requested data. If the CPU needs the next piece of data, it can first check if it is present in the cache. If it is, it can access it much faster than RAM. If it is not, it still has to wait for the data to be fetched from memory.

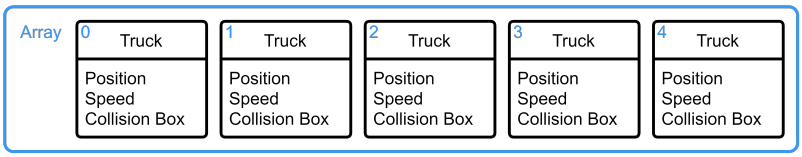


Figure 2: Five objects stored in an array.

Unfortunately, a cacheline cannot stay in the cache permanently, as when the CPU requests data that is not in the cache, a new cacheline will be requested from RAM. This new cacheline may overwrite the current cacheline. The cache can contain multiple cachelines, so it could still be there, but there is no guarantee. Therefore, it is essential to use all the data in the cacheline before it is swapped out with a new one.

Cache is built around the two concepts of *data locality*, *spatial locality* and *temporal locality*. Spatial locality is the concept that some data that is used together is also stored close together in memory. Temporal locality represents the notion of data that is reused close together in time. Temporal locality can be applied to data cache but also instruction cache, a cache for the instructions that are applied to the data. Just like data, instructions are evicted if they are not used for a long time.

2.2 OOD vs. DOD

Because of this memory gap and the long RAM latency, software performance is highly dependent on the number of RAM accesses. Minimizing cacheline transfers from RAM to cache improves performance, and so it is important to fully utilize a cacheline before it is evicted from the cache.

OOD does not take this into account. Data is grouped by 'objects' that correspond to real-world concepts. For example, in a military strategy game, a Truck and a Soldier are both objects. Each object might contain a dozen variables, like position, speed and health. Each object has all its data contiguous in memory; see Figure 2. If a Truck is precisely 64 bytes in size, it will fill an entire cacheline. So if all data from the Truck is needed, this is an optimal storage pattern.

However, in games, this is often not the case. Hardly ever is there only one of something. There are dozens of cars in a racing game and hundreds of people in a crowd simulation. OOD completely ignores this and handles every object on its own [14]. In addition, operations depend on the results of the previous operations. If two Trucks almost hit each other, checking for a collision before both vehicles' positions have been updated could lead to erroneous results. The second Truck might have slowed down or changed direction to avoid the collision. By first completely updating the first Truck, a collision might be detected that never happened. Instead, the positions of both Trucks need to be updated before checking for a collision. This means that the first Truck is pulled into the cache, and only part of its data is used before the second Truck is pulled in and only partially used. In an unfortunate situation, pulling the second Truck into cache pushes the first Truck out. The first Truck then has to be requested a second time for collision checking. Each time, the CPU has to wait for slow memory access.

Secondly, because there is no guarantee about the order in which objects are processed, OOD also makes poor use of the instruction cache. For example, if a Soldier is processed before the second Truck needs the collision instructions, those instructions could have been evicted from the instruction cache. Therefore, they need to be fetched as well as the data.

Only two steps of a game loop are described, but there are usually many steps involved in the loop. An object is pulled into cache at each stage, and only a fraction of its data is used. As the current object is only needed again when all other objects are processed, it is likely to

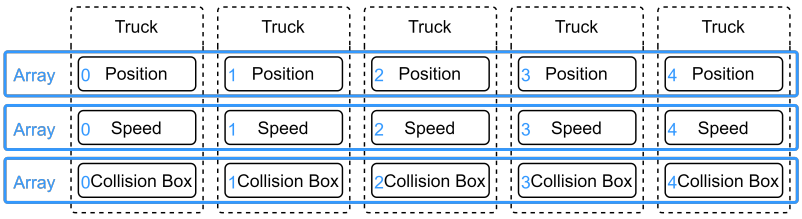


Figure 3: Five objects with their data stored in separate arrays.

be pushed out by the other objects. This means that the cache is used very inefficiently. For example, if a single float is processed from a 64-byte cacheline, only 1/16th of the cacheline is used. Moreover, if the cacheline is evicted each time, it incurs hundreds of wait cycles 16 times.

To use the cache more effectively, DOD stores the same data differently. Instead of based on real-world concepts, like OOD, it stores data based on how it is processed [14]. The application of this design approach is different for each situation. In games or other real-time software, a *Structure of Arrays (SOA)* is often used, where one property of a group of objects is stored continuously. As seen in Figure 3, instead of a list of Trucks in memory, there are now several lists for each piece of data—one array for the positions, one array for speed and one for the collision box. Now, if the positions are updated, a cacheline containing only positions is pulled into the cache. All data in the cacheline can be processed before the next cacheline will be fetched. Because all positions in the cacheline have been processed, this cacheline is not needed again. This means that the CPU only has to wait for the data to be pulled into cache once.

As the same transformation is done for all objects before moving on, the program instructions are not evicted from memory either. This means that the instructions do not have to be pulled into cache multiple times anymore.

In the DOD the Truck object does not really exist anymore; its data is scattered around memory. To find where the data is stored, there are two options. One is to create a dummy object that stores references to all data. It needs to follow a reference across memory, making the CPU wait for the new address before it can request the data itself. Therefore, while this is easier to reason about as it is very similar to objects, it loses almost all performance benefits. This technique is sometimes called *composition* [7, 17].

The other option is to store the object implicitly without a dummy object. Instead, all data from a single object is stored at the same index in their respective arrays. The first 'object' has all data stored at index 0 in each array. In some cases, the index does not have to be saved at all. For example, if the order in which the objects are stored is irrelevant. Finding a specific 'object' is more complex than with a dummy object. However, if this is not the primary use case, the indexing is more streamlined as no pointer references need to be followed across RAM.

Despite popular belief, DOD is not just SOAs or even ECSs. DOD centres around the understanding of what data we have and how, when and where it is used. Storing data sequentially, even if it is not in a SOA, may help with memory access, but as we saw before, it may depend on how the data relates to each other. In the same way, access patterns need to be considered. It does not improve anything to store data sequentially if it is used so slowly that the cacheline will be replaced before it is reaccessed [7].

2.2.1 Prefetching

Another hardware-related factor that contributes to the relevance of DOD for application performance is *prefetching*. If the memory access is predictable, a modern memory subsystem can predict what data the CPU will need next. It can then request that data before the CPU even requests it. Copying the data from RAM to the cache still takes the same amount of time, but as the request is made in advance, the memory can be there the moment the CPU needs it [1, 5]. A CPU die usually has multiple prefetching lines. If two arrays must be accessed simultaneously, both can be prefetched if their respective access patterns are predictable. The number of prefetch lines is dependent on the hardware, but there is a limit to the number of sequential accesses the prefetcher can handle . However, for prefetching to work, the memory accesses have to be predictable. With a less organized access pattern, the prefetcher cannot estimate the subsequent memory access.

As discussed, OOD exhibits a much less organized access pattern than DOD, making it an ill fit for prefetching. If all objects are in a single list, there is no guarantee that the objects are stored contiguously in memory. The array might store references to the objects, with the actual objects scattered through memory. Because of this, the access will also be scattered throughout memory. Even if all objects are physically stored in the array, it still does not guarantee predictable access. Take a Truck object of 64 bytes and a Soldier of 24 bytes that are stored mixed in a single array. Looping through the array, sometimes a 24-byte jump is made, and sometimes a 64-byte jump is made. The prefetcher will be unable to predict what is needed next. This only gets worse if more object types of different sizes are stored in a mixed array.

While there are ways to fix this, an SOA facilitates this naturally. All data is stored directly in arrays, contiguously. Every entry in the array is the same size, and the entire array is processed before a jump in memory is needed. Thus, the prefetcher can pull sequential cachelines for the CPU. Only if a new array is accessed does the CPU have to wait for RAM.

2.2.2 Maintenance and Flexibility

DOD considers hardware characteristics in its design. By storing the data conform those characteristics, DOD improves software performance compared to OOD.

Besides performance considerations, there is another reason to consider DOD; maintenance and flexibility. In OOD, hierarchical design is viewed as the best way to create runtime polymorphism. A developer should not need to revisit or change the base class to extend functionality. However, the base class must almost always be viewed or even altered if the game requirements change. These changes then have to be implemented in all child classes. Fabian even claims that "[...] having a base class that everything derives from has pretty much been a universal failure point in large C++ projects." [7].

DODs different data structure can alleviate this problem. Maintainability, reusability and testing also profit from the different software design. With all functions or systems written as simple data transformations, the result is small and independent pieces of code. These pieces are easy to understand, update or reuse [14]. Testing these small data transformations is also easy. Because they are already made to have simple input and output, there is no need to write complex tests. A simple database can be used with inputs that have to be matched to outputs [14, 7].

Because of these features, ECS can be an excellent choice to improve maintenance, flexibility, reusability and testing.

2.3 ECS

With the difference between OOD and DOD in data storage and its significance explained, we can explain a sub-strategy of DOD. ECS is similar to a SOA way of storing data, but there are a few notable differences.

DOD stores data in separate arrays as not all object data is always used together. Therefore it creates arrays of single data types, so the cacheline is used in its entirety before it is evicted from the cache.

While not all object data is always used together, certain parts of that data may be. This is the basis for ECS; data does not have to be stored separately if it is always used together. Instead, data that is used together can be stored together in a single array. For example, if the speed is always used together with the position, they are stored together, see Figure 4. Instead of querying two different arrays, a single array is accessed. This can improve readability and make it easier to reason about, especially if a lot of object data is needed in a transformation.

Because position and speed are always used together, a cacheline filled with both position and speed will still be used in its entirety. However, for the best performance, neither the speed nor the position must ever be used without the other. If that happens, there is once more unused data in the cache, and the two variables must be separated.

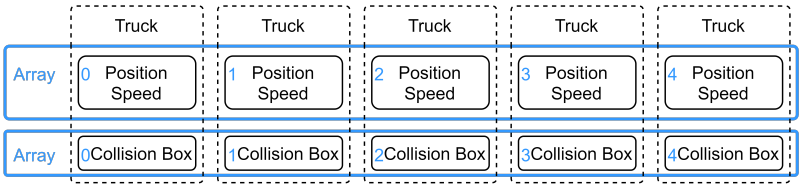


Figure 4: Five entities with their data stored in component arrays.

2.3.1 Entities and Components

In OOD, objects own their data. Either the data is stored directly in the object class, or they have a direct link to it. An object is identified by what object class it is, e.g. a Truck or a Soldier. The data it contains is present because of that object type. An object has Tire Friction because it is a Truck.

ECS has a reversed approach. ECS defines *entities* instead of objects. An entity is an all-purpose thing, everything in the game is an entity. They do not contain any data and usually only exist implicitly, as an integer ID.

The data itself is contained in *components*. Components are usually structs that only contain a few data members each. As explained before, a component only contains data that is constantly used together. An entity can have multiple components, each component representing a piece of the data of the entity. Components are stored contiguously in a separate array; each type of component has its own array, see Figure 4 [17]. Components are created, maintained and destroyed by a separate manager. Entities have no direct link to or influence on their components at all. Instead, an entity's ID can be used to find its components in their respective component arrays.

An entity is defined by its components, e.g. an entity is a Truck because it has a Tire Friction Component. Entities can be compared with a keyring, with the components being the keys. A keyring on its own is nothing; its value comes from the keys. So the only way to determine whose keyring it is is by looking at what keys are on the ring [2]. In the same way, the only way to determine what kind of entity it is is to look at its components.

As components are stored contiguously, and all data in the components is always used

together, it uses the caches optimally. Because most transformations run regardless of what entity a component belongs to, those transformations can run on a component array without consulting the entity or the entity ID. In most cases, it can run as linearly as a SOA. Components can still have only one variable (single-field components), for example, if that variable is not consistently used together with other data. However, they can also be created quite large if all data is always and only used together. Grouping data that belongs to the same domain, e.g. data concerning the rendering or the physics, can help with the comprehensibility of the code.

2.3.2 Systems

The third part of ECS is the *systems*. In OOD an object will typically consist of its data, as we discussed above, but also contain its transformations. When a game loop processes its game objects, it will instruct the objects to update themselves. OOD is fundamentally built around the concept of processing objects one at a time. For example, a Truck and a Soldier both need their position updated. The Soldier uses only the preferred speed to calculate the new position. The Truck takes the preferred speed and the friction of its tires into account and thus needs a different calculation, see Figure 6.

Systems are built around the idea that a transformation often happens for many entities. Systems only perform one kind of transformation on components. For example, a system would only update the speed or only update the position. Of course, a system can use multiple components to update the speed or the position, but, like an OOD method, it should have a single responsibility. Because a system only cares about components, it needs to specify which components it needs. The system does this by creating an *entity query*. Like a database query, an entity query specifies what components should be present and which should be absent in an entity. Only the entities that match this query get returned, and processed [7].

To only update the Trucks' position, a system would specify an entity query for the Preferred Speed Component and Tire Friction Component. The Trucks match this query and get returned. The Soldiers have the Preferred Speed component but not the Tire Friction Component and are not returned. The system for the Soldiers would have an entity query that returns the entities that have the Preferred Speed Component but do not have a Tire Friction Component. This time the Soldiers would be returned but not the Trucks. It is essential to specify that the Tire Friction Component should be absent. If the query only stated that a Preferred Speed Component should be present, both the Truck and the Soldier would be returned. An entity query can define what component should be present and define which components are not allowed to be present in an entity.

Because updating an entity in a system should only use the components from that entity, no race conditions occur when these systems are parallelized. This is true for SOA and ECS. Of course, if two threads read and write to the same cacheline, synchronization issues can still occur [17].

Where in OOD focuses its programs around the objects, DOD builds its programs around the data, how it is used and how it should be stored.

2.3.3 Vectorization

Outside of multi-threading, modern CPUs can do more than one operation at a time per core. The CPU can perform the same operation, like multiplying a float, on multiple pieces of data simultaneously. This is possible because the CPU has *vector* or *Single Instruction Multiple Data (SIMD)* instructions [21]. *Vectorization* is the process of rewriting a loop to process multiple elements of an array at the same time. Depending on the hardware, 4, 8, or even 16 floats can be processed simultaneously. Without vectorization, the CPU pulls a single float

to perform an operation four times. With vectorization, the CPU pulls all four floats at the same time, and with one instruction, performs all operations at the same time. The CPU returns four results in the same time it takes to do one multiplication.

For SIMD to work best, the data should be neighbouring in memory. If it is not, the data must be gathered before it can be processed with vectorization and scattered after it is. Because of this, single field components that only have a single type of data in an array are preferred for vectorization.

2.3.4 Padding

In OOD, depending on the used framework and compiler, padding might be added between structs to make them cache aligned. Padding might also be added inside structs when data types of varying lengths are mixed. For example, data types of one byte could be padded out to four bytes to align the next data member. This can make the struct much larger than expected.

Figure 5 shows the padding of a struct consisting of two floats and two chars. Each member is padded to be aligned on a four-byte boundary. Mixing the order in which the members are stored will lead to a larger struct because of this padding. Conversely, ordering the members based on lengths in bytes can create a smaller struct. These concepts apply to components as well.

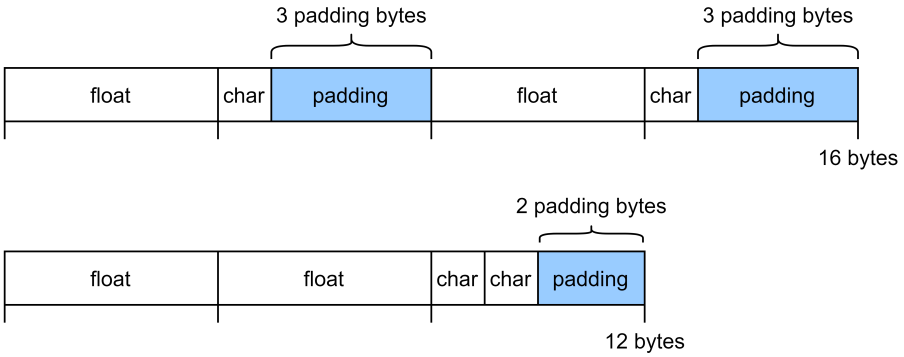


Figure 5: Padding of a struct.

2.4 ECS Framework Implementations

Because ECS relies on a data layout that, at a low level, matches the characteristics of the hardware, the implementation details of an ECS framework are relevant. The implementations differ in their solution to entities with a different set of components.

In the example used before, the Truck has components that the Soldier does not have. There are three ways to deal with this kind of situation.

2.4.1 Big Array-based

The Big Array approach handles this naively. If an entity does not have a specific component, the index slot in the component array is merely left empty. So, for example, if only two out of a hundred entities have a particular component, that components array will have two filled slots and 98 empty slots, see Figure 6.

The advantage of this approach is that there is no additional bookkeeping necessary for missing components or empty slots. The downside, however, is that these empty slots will also

translate into empty space in the cachelines. Because this re-introduces the original problem, this technique is not used in practice. Instead, alternatives are used that solve this problem while still using the cache as optimally as possible.

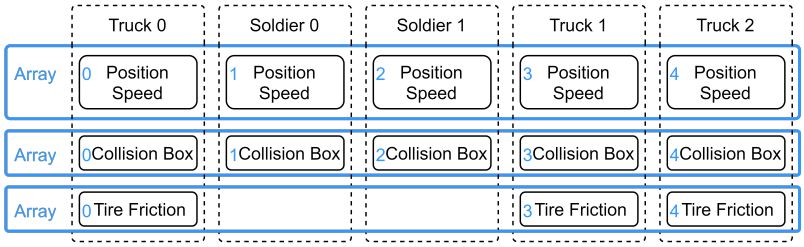


Figure 6: Five entities with their data stored in a Big Arrays-based ECS framework.

2.4.2 Sparse Sets-based

The second approach is called the *Sparse Sets-based* approach. Where Big Array has a single array for each component, Sparse Sets has two arrays per component, see Figure 7. One array is a *packed array* that stores the component data contiguously. The second array, a *sparse array*, stores a reference to the packed array for each entity. If an entity does not have that component, the index slot is empty. Because of this, the second array will have holes in it, but the actual component data is stored contiguously in memory.

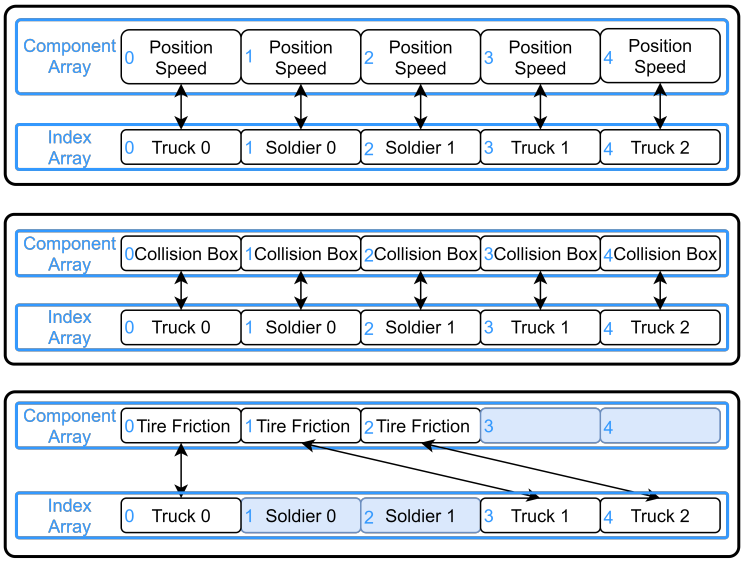


Figure 7: Five entities with different components stored in an Sparse Sets-based ECS framework.

Consider the case of the Trucks and the Soldiers. Every entity has the Position-Speed Component, so the index of the entity is also the index of the component. However, not every entity has a Tire Friction Component. Only the Trucks have that component, not the Soldiers. Figure 7 shows that the three Truck entities have corresponding Tire Friction Components. The index array has empty slots for the Soldiers that do not have this component. However, the component array does not have these holes. Instead, the component data is pushed up until it is contiguous. If a system only requires a single component, it can iterate over the entries of a component array. It can pull that data into memory contiguously without looking

at the index array. Furthermore, if a specific component for an entity is required, the index array is queried at the entity's index. Then the reference is followed to the needed component in the component array.

However, often a combination of components is needed instead of just one in an entity query. For each query, a manager will loop through the shortest component array, and for each component, follow the reference back to the entity index. With that index, the manager checks if that entity contains the other components. For example, consider an entity query of components Position-Speed and Tire Friction. The manager will first loop through the component array of Tire Friction, as it is shortest. For each component found there, it follows the reference back to the index array to get the entity index. Then the manager checks the index array of the Position-Speed Component to see if the entity has that component as well. If it does, it is confirmed that that entity has both required components, and the entity is returned to the system. The more components are required in a system, the slower the search gets.

However, a *structural change*, adding or removing components, can be done in constant time. The component only needs to be added to the end of the packed array (component array) and a reference to the sparse array (index array).

While many techniques are applied to lessen the impact, the two arrays and the indirection make the entity queries slower than with a single array. This technique, however, enables something few other techniques allow. Because structural changes are efficient, components can be added or removed to transfer data between systems. It can notify a system that a specific entity should be destroyed or simulate a user interface where keypresses are passed as components to an input entity.

2.4.3 Archetype-based

Unlike Sparse Sets, a *Archetype-based* approach does not store all components in a single array. Instead, for each unique combination of components, it creates an *archetype*. An archetype is defined by its combination of components. Therefore all entities with a specific set of components belong to the same archetype. For example, the Truck entity has a Position-Speed, a Collision Box and a Tire Friction Component. All Truck entities thus belong to the same archetype; the Truck Archetype. However, it also means that any other entity with those exact components belongs to the Truck Archetype as well. If one component is added or removed, the entity will belong to a different archetype.

Removing the Tire Friction Component for a Truck will change the archetype to the Soldier Archetype. The archetype does not care if the entity is a Truck or a Soldier; it will store every entity with the same archetype the same way. In practice, the entities are not named because of this. Giving archetypes names can insinuate similarities between archetypes and object classes. However, where objects have data because they are a specific class, archetypes are only defined by their components, like the keyring example. Thus, comparing archetypes with objects can hinder the mental change needed to design for ECS.

An Archetype-based ECS framework stores its components based on what archetype it belongs to, see Figure 8. A component array will only contain the components for one archetype. The more archetypes there are, the more scattered the components of a single type are stored. Adding or removing components will change an entity's archetype and force the entire entity with all component data to be copied to the new archetype. If all archetypes are properly filled, an entity query can check an entire archetype and skip it if it does not fulfil the requirements. This makes entity queries very fast.

Tag components If two entities have the same components, they have the same archetype and will be processed the same way. Therefore, if an entity needs to be processed differently,

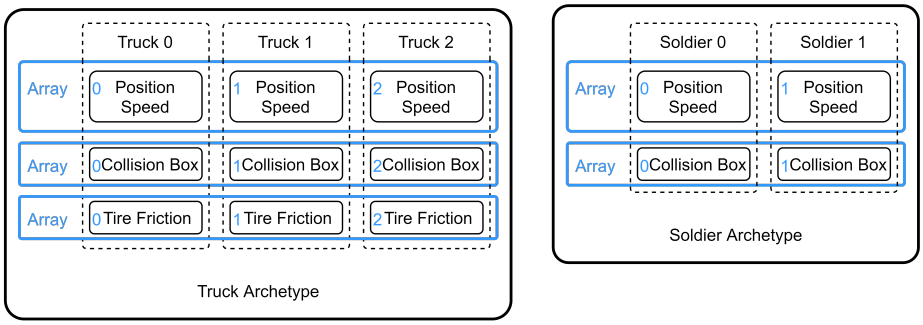


Figure 8: Five entities with different components stored in an Archetype-based ECS framework.

it is best to add a component to change its archetype. However, if there is no need for extra data, a *tag component* can be used. This component does not contain any data and only exists to change the archetype of an entity. Because this component has not data, adding and removing them can be done faster.

Unity DOTS and Archetypes Unity's Archetype-based implementation introduces another grouping, *chunks*. Chunks are 16KB large pieces of memory that only store component data for entities of a single archetype. These archetypes are used to dispatch work to different CPU cores. Because different cores process different chunks, different cores will not contest the same cacheline. If there are too few entities in an archetype, the chunk will only be half-filled, leaving a lot of empty space [22].

2.5 ECS Conclusion

ECS makes better use of caches than OOD, being designed around cache and memory characteristics. To achieve this, ECS stores its data contiguously in memory. This reduces the number of times a cacheline is accessed and enables effective prefetching. To make the technique more user friendly, it groups data that is consistently used together in components. Systems process those components, each performing small transformations on the component data.

Because ECS is designed around hardware characteristics, low-level implementation details can influence the performance in certain situations. The naive Big Array approach diminishes the performance when not all entities have the same components. Sparse Sets-based and Archetype-based frameworks are generally more performant, with the first favouring structural change performance and the second entity query performance. In addition, considering vectorization needs and minimizing padding in component design can improve overall performance.

Lastly, often the name 'ECS' is used for both the design principles and the implementation. However, as is explained, the ECS design principles can be implemented in different ways. 'ECS' will refer to the ECS design principles and for the framework 'ECS framework' will be used.

3 Related Work

DOD is fundamentally different from OOD, especially in how those two techniques store their data. This makes conversion between the two strategies difficult.

While ECS is underrepresented in literature, DOD, compilers, code analysis and language conversion are not. This section will discuss the most relevant research on these topics.

3.1 Manual OOD to DOD Conversion

The significant difference between OOD and DOD, from concept to implementation, can make the conversion from one design technique to the other a challenging procedure.

Literature generally suggests two methods for manual conversion. The first takes the whole object and splits small parts off one by one until the whole object is converted into parts. This technique is often used for composition-based DOD, which eliminates many of the performance benefits [14, 7, 17]

The other technique breaks the object down entirely in the first step. Recombining data is only done if these combinations are logical computation-wise. This method is used for component-based design such as ECS design. Fabian suggests that recombining the data into components can be simplified if relational database normalization is applied. The mentioned normalization rules ensure that data is only combined if it always belongs together. In addition, data should be grouped on how, how often and in what order the data is accessed. As some CPUs have a penalty for switching between reading and writes, separating the read and writes as much as possible can be beneficial for the performance [7].

Literature does not suggest how these criteria should be applied. Instead, it suggests that each situation requires a unique solution based on these principles. Especially in DOD, where the chosen data structure is heavily dependent on the use cases, every solution is unique. Despite many examples, there is not a single ruleset or tool that works for every project [14, 7].

This fact, combined with the steep learning curve and fundamental difference from OOD make conversion difficult, especially for developers new to ECS.

The following principles of DOD can be used in our work:

1. Data should only be grouped if it always belongs together.
2. Data should be grouped based on how and when it is accessed.
3. Read and write access should be separated for better performance.

3.2 Language Conversion

Language conversion is the practice of converting a codebase from one language to another. There are various methods to achieve this, from simple syntax replacing to compilers for this purpose. However, those methods struggle to maintain the structure of complex code.

A method that is capable of converting the design structure of the code uses an intermediate language. It analyzes the input code with a syntax analyzer and converts its logic to an intermediate language in several steps without changing the original program. From the intermediate language, the program can then be converted to the destination language. This method makes it possible to convert from and to any language [10].

The following insights can be applied to our research:

1. Syntax analyzers can gather information on program structure.
2. Using an intermediate language can help with converting between two programming languages while maintaining program structure.

3.3 Compilers and Code Analysis

The field of compilers and code analysis is prohibitively large to cover in its entirety. In this section, we will focus on the most relevant parts of our research.

Compilers use static program analysis, such as Abstract Syntax Trees, to reason about code behaviour without running it. This can be used for error detection and for producing efficient code. In addition, static program analysis can be used to reverse-engineer design patterns, for example, by inferring inter-class relationships [19, 16].

However, there is a limit to what static program analysis can do. In Object-Oriented languages, static analysis is not always capable of knowing which object is actually called because of dynamic types. More complex analysis is needed to extract this information [18].

The lessons from compilers' code analysis that can be applied to our work are the following:

1. Static program analysis can extract design patterns from source code.
2. Static program analysis cannot always infer which method call was made in Object-Oriented code.

3.4 Planning Profiles

While DOD optimization is a relatively new field of research, automatic optimization is the subject of quite a lot of research.

Many automatic parallelization and vectorization compilers and tools convert the input code to functioning output code. However, their result is usually limited and not comparable to manual conversion [3, 13, 11, 9, 21]. However, planning tools also exist that only create a plan and require the user to implement the suggested changes manually. Furthermore, they use a planning character or planning profile to adjust to different platform requirements without changing the analysis process for each platform [12, 9].

The following concepts can be used in our research:

1. There exist optimization tools that only create a plan. These leave the implementation of the changes to the user.
2. Planning profiles can be used to specify a plan to a specific platform and its requirements without changing the planning process itself.

3.4.1 Proposed Conversion Method

With the combination of the results from the presented research, the following conversion technique is proposed.

We will use static program analysis to deduce inter-class relationships, class members and method information from the source code.

Converting from one language can be done by analyzing syntax and parsing it to an intermediate language. We propose that changing from one data design model to another can be aided by using an intermediate data model. This model stores all relevant data in an independent format without changing the input code.

While there are some guidelines on how to create a DOD design, it is highly dependent on the situation and its requirements. Therefore, the created design should be adjustable by the user. To facilitate this, planning profiles will be used in addition to the guidelines to adapt a design to the requirements.

Static program analysis is used to extract information on the input source code. This information is then parsed into an intermediate data model to preserve program structure. Lastly, a design is generated based on guidelines and a planning profile determined by the user.

4 ECS Insights

CPU cache utilization affects the performance of an application. DOD data structures make better use of the cache than OOD data structures. It achieves this by storing its data based on how and when it is processed. A sub-design strategy of DOD, ECS, stores data in components and the transformations on that data in systems. The ECS design strategy can be implemented in three ways. Big Array-based is a naive version and not as performant as the other two, Sparse Sets-based and Archetype-based. Each of those has specific advantages and disadvantages. Unity's ECS is an Archetype-based framework.

Literature on ECS is limited and research on ECS design is non-existent. To gather knowledge on these topics, we held interviews with experts on ECS. This section will discuss the interviews and their results. We start by explaining the interview setup. Then we introduce the insights on ECS, ECS design and visualization. We summarize these results in Sections 4.3.1 and 4.3.2.

4.1 Interview Setup

We interviewed 10 experts with knowledge of the ECS domain. They were interviewed remotely with Zoom. After a short introduction, they were asked 20 open questions. This section describes how the interviews were conducted.

4.1.1 Selection

The interviewees were chosen based on their knowledge and experience with ECS. Most of the participants have created or contributed to an ECS framework that other developers actively use. Other participants have released relevant work or have extensive knowledge of the technique. Information on the experience of the participants can be found in Figure 9.

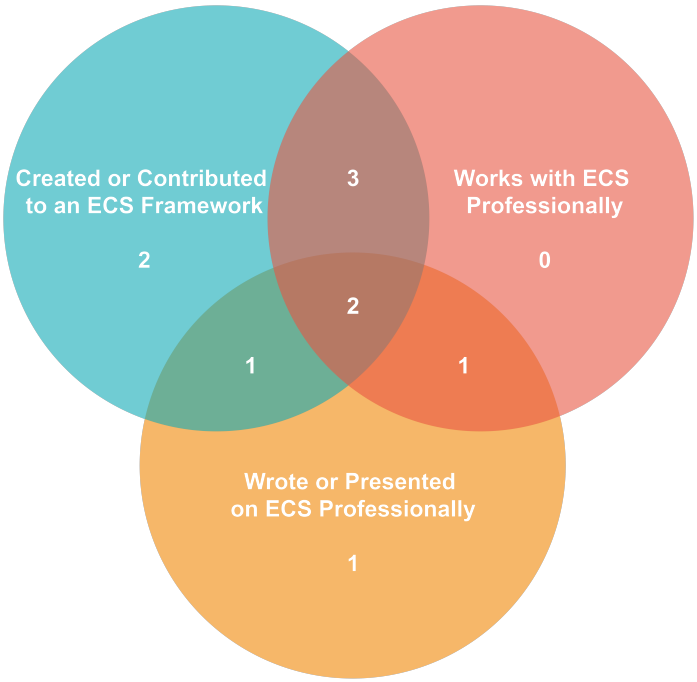


Figure 9: Experience of the interview participants. Note: Professional writing or presenting means, writing a (scientific) publication or presenting at a conference.

4.1.2 Question Topics

The 20 open questions were divided into three categories:

Personal Experience These questions collected information on the interviewees' experience with ECS. This information helps to validate the authority of the interviewee on the subject. In addition, familiarity with the different ECS implementations is gathered. This is important as each implementation has different advantages and disadvantages. Knowing which implementations the interviewee used can help categorize their answers.

1. Can you tell me about what got you first started with ECS?
2. Can you tell me how you currently use ECS?
3. What ECS implementations have you worked with?
4. Can you tell me in one or two sentences the biggest advantage that you see of ECS?
5. Can you tell me in one or two sentences the biggest disadvantage that you see of ECS?

ECS General Design The interviewee is asked about this process to get an overview of the steps taken when creating an ECS design. Information on the steps taken from start to finish gives a good overview of the design process. Also, it can create basic guidelines for ECS design visualization.

6. Can you walk me through the steps that you take while creating an ECS design?
7. What is the biggest misconception you had when starting to design for ECS?
8. Can you tell me some things to avoid when making an ECS design?

Component Design The questions in this category focus on the design of the components. Questions on what rules are used to create components give more detailed insight in ECS design.

9. When you are creating components, how do you decide how to split up the data?
10. What do you focus on when you put together components?
11. Do you take low-level factors like read/write separation and cacheline alignment and such into account making components?
12. If yes, can you tell me what you take into account and why?
13. When there are multiple options, how do you choose which component combination you take?
14. What do you do to keep the component design organized and legible for other programmers?
15. If you had to put it to a few rules, what rules would you create to guide the creation of components?
16. Are some of those rules more important than others? If yes, then what order?
17. Can you tell me some things to avoid when designing Components?
18. There are generally three implementations, Big Array, Archetypes and Sparse Sets. Is there a difference between how you would design Components for each type?

Closing Questions At the end of the interview, the interviewees were also asked these closing questions:

19. Is there anything that you have seen that I need to avoid?
20. Is there anything you consider important for ECS/Component design that I have not asked about?

4.2 Results

This section will summarize the relevant insights gained from the interviews. The results are grouped on topics. Each topic is introduced, the answers discussed, and the relevance explained. The complete transcripts can be found at `https://github.com/AnneVanEde/MasterThesis`.

4.2.1 Advantages and Disadvantages of ECS

Advantages "Performance" is usually the first thing mentioned when talking about ECS. The interviewees confirmed that performance is an important (and visible) feature of ECS. The memory layout can be improved by including only the relevant data for a transformation. This leads to better cacheline utilization and ultimately better performance.

However, it turned out that performance is not always the primary motivation for using ECS. Some said that performance was not a consideration at all. Instead, the most named advantage was flexibility. In many ways ECS is much more flexible than OOD. ECS separates the concerns of the program and keeps the data separate from the transformations. Because of this, designability, maintainability, testability, readability and reusability all improved.

For example, ECS solves the hierarchy's so-called 'diamond problem'. This is a problem where two objects in different 'subtrees' of a hierarchy have similar functionality. A Tank (subclass of Vehicle) and Soldier (subclass of Infantry) that both need the same grenade launcher can be a problem. In OOD, you would have to duplicate the functionality for both classes or bring it higher in the hierarchy, making it redundant for some other objects. In ECS, this feature can be defined as a component with a matching system. This component is then attached to the two different 'objects', and both have the functionality. This enables features to be defined separately and mixed and matched as needed. Because systems are simple transformations, with input and output, testing can be simplified as well. Systems can be run with different kinds of input, and the outputs are checked for correctness.

While these results coincide with the literature for DOD, the order of importance is different than the literature suggests. More than performance, flexibility and maintainability are an essential part of an excellent ECS design. The extent to which this will influence design decisions will depend on the situation and preference of the developers.

1. An ECS design should strike a balance between performance on the one hand and flexibility and maintainability on the other.
2. This balance is influenced by the circumstance and requirements of a situation.

Disadvantages Most of the disadvantages of ECS come down to one thing. It is an entirely different way of thinking to OOD, and therefore a shift in reasoning is needed. The learning curve of ECS is very steep, and a lot of knowledge is needed to get started. In addition, there is an unlearning curve for OOD. Many tools and functionality that are common in Object-Oriented Programming (OOP) are absent in ECS. Explanation on design choice can help to start with ECS.

Another minor disadvantage is that people have many misconceptions about ECS. Trying to apply ECS where it is not needed or even not suited can lead to failure. This only re-enforces their misconceptions of ECS. While ECS is very powerful, it is not applicable in all situations.

For all the advantages ECS brings, it is not easy to start with. However, helping people get started with ECS by educating them on design choices can help. They should also be taught when ECS is not applicable in a situation.

3. Explaining design choices to people new to ECS can help in learning the technique.

4. Newcomers to ECS should be educated to when ECS is applicable and when it is not.

4.2.2 Visualization Guidelines

To gather guidelines for design visualization, the interviewees described their design process. This highlighted the most important relationships within the design.

After determining that ECS is applicable for their situation, they create a mental overview of the game's or application's most important functionality. Finally, most wrote their design down in a kind of design document.

After they have created this design, they create the components and the systems. There are two approaches for this, first focusing on the components and creating the systems second or the other way around. Those who started with designing their systems focused on the transformations. They usually preferred smaller components or even single-field components. On the other hand, those who created their components first focused on the components in an entity type and the commonalities in data and components between different entity types. As a result, they usually created larger and fewer components.

The game functionality is the key to an ECS design. From this, both the components and the systems are created. This functionality should be visible and recognizable in the final plan. Despite the difference in focus for the two approaches for creating components and systems, both take two relationships into account. The first one is the relationship between systems and the components they use. The second relationship is the commonalities in components with other entity types. While both those relationships need to be incorporated into the design plan, the emphasis should be selectable.

5. The visualization should make the connection between the OOD code and the ECS design recognizable.

6. The visualization should show the relationship between systems and their components (system-component).

7. The visualization should show the commonalities in components between entity types (entity commonalities).

8. ECS design visualization should give the option to focus on either the system-component or the entity commonalities relationship.

4.2.3 Splitting and Merging Components

To gather more detail on the design process, the interviewees were asked about their considerations when creating components. The most important considerations were for splitting data into two components and merging data into a single component. In addition, component size was relevant to them. This information can establish requirements for the components of an ECS design.

Splitting Splitting components is usually done for performance reasons. The primary reason is that not all data is used in a system. This leads to unused data in the cache, reducing the performance of that system.

A similar but much more important reason is read-write contention. If system A writes to a component and system B wants to read from the same component, it will have to wait for system A to finish. This means that those systems can not run concurrently. The dependency between the two systems is also called a sync point. This is not always preventable, but if the read happens to the first half of the component and the write happens to the second half, the component can be split into two. The systems can now run in parallel. Of course, this is a variation on the problem that only half the component is used in a system, but it creates a much more significant performance bottleneck than unused cachelines.

There can also be framework dependent performance reasons to split components. For example, one interviewee mentioned Unity might not be able to vectorize multi-field components automatically. In addition, splitting blittable and non-blittable types can also be beneficial. Blittable data types have the same representation in managed and unmanaged code, like floats and integers. This means that it can be passed without conversion between sequential and concurrent code. Non-blittable data, however, cannot be passed to a concurrent job and has to be handled on the main thread. Therefore, mixing these types in a component means that the blittable data must be processed on the main thread.

There are some reasons beyond performance to split components. Components can be split on domain boundaries, as it can be confusing to put two concepts into a single component. Unless a good name is chosen to reflect all responsibilities of the component, the code becomes more complex and less maintainable. In addition, testing a component with a single responsibility is easier.

9. Data that is not used together should be put in separate components for better cache usage.

10. If one system writes to a component and another system needs to read from/write to it simultaneously, always split the component for fewer dependencies between systems.

11. Components should be split on domain boundaries for better understanding, maintainability and testability.

12. (**Unity**) Non-blittable data should be put in a separate component from blittable data.

13. (**Unity**) To allow vectorization, consider creating single-field components.

Merging Merging components is rarely done for performance reasons. Instead, it is done to make the codebase easier to reason about and work with. Besides grouping data together because they are used together, they can also be grouped because they belong to the same domain. This leads to fewer components, which is easier to understand, especially to less experienced developers.

14. Data that is processed together should be put in a single component to improve readability.

15. Data that belongs to the same domain should be put in a single component to improve readability.

Component Size Considerations Small components do not incur performance penalties, while large components with redundant data do. Consequently, there is no performance reason to merge components. However, having too many components can be challenging to work with, especially in a team. Deciding on component size is, therefore, a balancing act. For example, if a component is used in a hot loop, performance might be the most important thing, while a component that is used infrequently might be made more user-friendly at the cost of performance. Of course, these considerations also change based on the performance consequences of a choice.

16. Components that do not need to be performant could be merged to make the design more user-friendly.

4.2.4 Other Design Considerations

There are a few other considerations for designing components and systems. They are listed in order of importance to the developer.

Generic Components One of the pitfalls of developers who are new to ECS is that they make their components as specific as their classes used to be. Instead of objects, the data and their transformations are central. As a result, when thinking in gameplay terms, like a tank crossing a bridge, the fact that an NPC entering a building has the same transformations could be missed. These two concepts are very different along the axis of gameplay, but they may be similar along the transformation axis. Creating more general components helps with seeing the similarities between concepts.

17. Very specifically used or named components can hinder noticing similarities between concepts.

Archetype-based vs. Sparse Sets-based When making an ECS design, the used framework is a consideration. For example, a Sparse Sets-based framework can add and remove components in linear time, but these structural changes are costly and slow in an Archetype-based framework. However, in an entity query that contains many components, Archetype-based frameworks outperform Sparse Sets-based frameworks. This is because Archetype-based frameworks loop over all archetypes once to see if they match the query. Therefore, the time that it takes is not dependent on the number of components in the query. On the other hand, a Sparse Sets-based framework needs to loop over entities in the components array for every component mentioned in the query. While some tricks are applied, so it does not need to loop over every entity, the required time still goes up with every component added to the query.

18. (**Archetype-based**) Structural changes should minimized for better performance.
19. (**Sparse Sets-based**) Consider creating fewer, larger components, to improve entity query performance.

Prefetching and Super Systems Prefetching is one of the reasons ECS can be very performant. Because the access pattern of components is often linear, the next component can already be pulled into the cache before it is requested. However, there are a limited number of prefetching lines. This number differs for each CPU chip, but prefetching might not keep up if a system processes too many components. A solution could be to merge some components. However, it can also be an indication that a system has too much responsibility, a so-called Super System, which is an anti-pattern in ECS. If a system has too many responsibilities, it can become less maintainable and reusable as it increases complexity. In that case, the system should be split to reduce its responsibility.

20. If a system processes many components, consider splitting up the system, as it can be a super system, which is an ECS anti-pattern.

Cache Alignment and Padding Padding could be added to the components to make them cache aligned depending on the framework and compiler. Unity might even add padding at the end of chunks. While this could be a reason to split components or entities, it is essential to measure this. For example, merging two components might cause unused data to be pulled into the cache, which might be worse than the padding. Padding might also be added to align them to a 4-byte boundary when data types of varying lengths are mixed. This might make the component much larger than expected. Therefore, it is good practice to mind the order in which data is stored in the component.

21. Always order the data by their data type and size to avoid unnecessary padding.

4.3 Summary

The results of the interviews give insight into ECS and designing for ECS. In addition, it gave guidelines for the visualization of ECS designs.

4.3.1 ECS and ECS Design

The results showed that ECS is chosen more for its flexibility and maintainability than its performance possibilities. Learning ECS however is difficult because of the fundamental difference between OOD and ECS. Therefore, an ECS design should balance performance with flexibility and maintainability.

To make a design performant, it needs to ensure there is no unused data in the cache. Dependencies between systems also hinder performance as it might prevent running the systems concurrently. This means that all data in components should invariably be processed together, especially if systems have to read and write to it concurrently. The order in which the data types are stored inside a component should avoid unnecessary padding.

A plan becomes more maintainable if the developers working on the code can understand it. Creating larger components and therefore having fewer components can decrease the complexity of the code and the project. If a component and accompanying system do not have to be performant, performance can even be sacrificed for readability. In addition, it is beneficial to mind domain boundaries when creating components. If components contain a single real-world concept, like physics, and all its data is in a single component, the code becomes easier to comprehend.

Besides those two primary considerations, there are also a few good practice rules for an ECS design. For example, a pitfall for OOD programmers who start with ECS is that they create components that are very specific, similar to the original object. These very specifically used or even named components can prevent the developer from seeing the commonalities between entity types and components. Instead, the components should be named and used as general as possible.

Systems also should not have too many responsibilities. This is an anti-pattern of ECS that hinders reusability and maintainability. Such systems should be split up to reduce complexity.

Lastly, some considerations are framework dependent. For example, Archetype-based applications should consider minimizing structural changes as those can decrease performance. On the other hand, Sparse Sets-based applications do not have this problem. However, they should consider creating fewer but larger components to reduce the number of components in an entity query to get optimal performance.

There are also Unity specific factors. Because, unlike blittable data, non-blittable data has to be processed on the main thread, these two types of data should always be split into

separate components. Unity can also benefit from single-field components, as it allows for automatic vectorization.

All these considerations are dependent on the situation and personal preference. Therefore developers can prioritize some design aspects over others. For example, they could prioritize readability over performance or smaller over bigger components based on the framework and situation.

These results provide knowledge on ECS and ECS design. While there are many considerations, the balance between maintainability and performance is the most important one. A good design balances those according to the requirements of the situation and the framework.

4.3.2 Visualization

The results also introduce basic criteria for ECS design visualization. Visualization can aid in transitioning from OOD to ECS and can educate the developers on the design choices. To do this, it needs to communicate the most important aspects of the design.

One of those design aspects is the connection between the original OOD design and the new ECS design. Seeing how familiar concepts are handled in ECS can aid understanding of ECS.

It should also highlight some key relationships in the ECS design. The commonalities in components between different entity types should be clearly visualized. The relationship between the systems and the components they process is also important. The importance of each of those relationships can vary, so the focus should be adjustable in the visualization.

The guidelines introduced in this thesis can become the basis for a standard visualization of ECS designs. The key relationships in a design and the connection to the original OOD should be present in the design visualization.

5 Implementation Details

This section proposes a method to convert from OOD to DOD. Section 5.1 introduces this method and the steps it takes. In sections 5.2, 5.3 and 5.3.1 we introduce our tool, ECS Conversion Designer and Education tool (ENCODE). We explain how we combined this method with the information on ECS design from the previous section to convert from OOD to ECS. Section 5.4 explains how we implemented the guidelines on visualization in the tool.

5.1 Conversion Method

The conversion method this thesis proposes can convert from one design strategy to another by analyzing the code from the source strategy. A design for the destination strategy will be created. The steps used by the method are described below.

Static Program Analysis The first step extracts the structure from the source code. Static analysis is used to analyze the code without running it. It can extract information on inter-class relationships but also on variables and methods. To achieve this, it looks at the structure of the code.

Intermediate Data Model The proposed method uses an intermediate model to create a general conversion method that can be altered to any design strategy. This model encodes the data and program structure in a manner that is independent of the design strategy. The model should be general enough to be applicable in many design strategies but specific enough to accurately depict the intent and functionality of the source design strategy.

Design Planning The planning phase follows the best practice guidelines of the destination design strategy. However, these best practices might be dependent on the situation. Therefore, planning profiles are used to specify the design principles to the current situation and requirements. As the parameters for these profiles are incorporated in the planning stage, changing the profile will change the outcome without changing the design process.

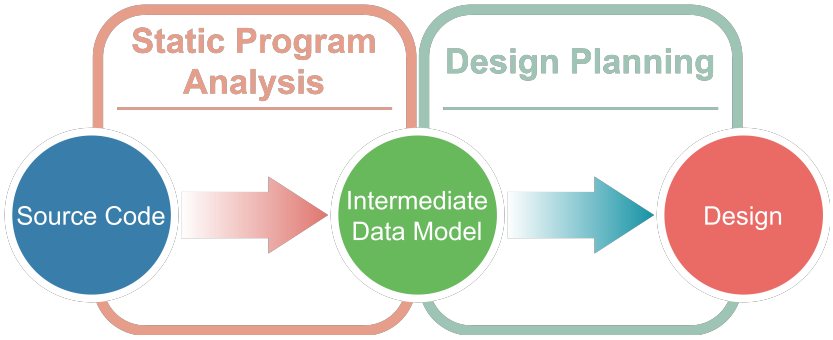


Figure 10: A diagram of the steps of the proposed design conversion method.

5.2 Static Program Analysis

The first phase of the tool uses static program analysis to extract the relevant information from the C# source code. It does this in two steps.

The first step uses the .NET Compiler Platform SDK, also known as the Roslyn APIs, to extract the raw data structure from the source code [15]. Compilers use *syntax trees* to check that the code is written with the correct structure and follows the language grammar. It also uses *semantic models* to check the meaning of all nodes of the syntax tree and check references

between code, such as method calls. The Roslyn APIs exposes these compiler analyses to the developer. By using the syntax trees and semantic models of the source code, the raw source code can be converted into a data structure that can be queried and processed.

However, the syntax trees and semantic models do not explicitly contain all the necessary information. Therefore, an extra analysis step is added to extract the implicit information as well. A intermediate OOD data model couples classes to their class members and their parent's class members. It also connects the class members to the methods they are used in. Reading a class member in a method is denoted differently from writing to a class member.

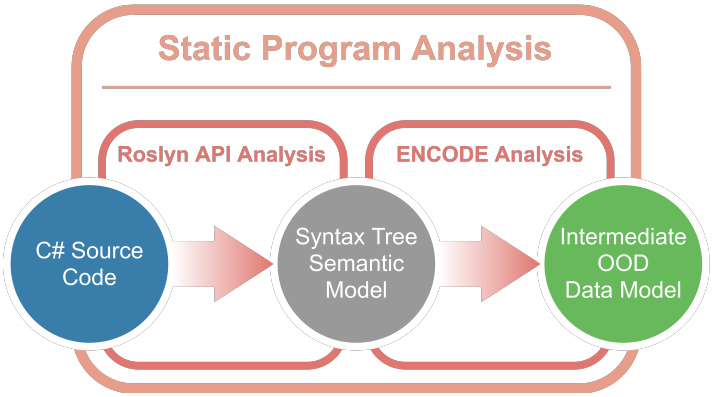


Figure 11: A simplified diagram of implementation of the static program analysis for C# Object-Oriented Code.

5.3 Design Planning

The planning phase uses the information in the intermediate OOD data model to create an ECS design. For this phase, another intermediate data model is created. This intermediate ECS data model is created to ensure that the ECS design can be re-planned without the code analysis phase having to run again. Therefore, the first step of this phase copies parts of the intermediate OOD data model to the ECS data model. All class members are converted to *component fields*, variables that can be put into components on their own or together with other component fields. They are the building blocks of components. Methods are converted to systems and are linked to the component fields they read or write. Classes themselves will be converted at a later stage.

After this preparatory step, the ECS design itself can be created. First, the component fields are combined into components. Next, the systems are updated to include the components they read and write respectively. Lastly, the *entity types* are created, a group of components that encapsulate a real-world concept, as the classes did.

After the planning phase the ECS design is finished. The entities, components and systems are automatically extracted from the OOD data model and stored in an ECS data model. With this ECS data model the ECS design is made.

5.3.1 Planning Parameters

ENCODE implements planning profiles to influence the generated ECS design. The planning profile parameters are based on the results of the interviews. These parameters can be altered by the user of the tool to fit the criteria of the situation. For each parameter, the insights gained from Section 4.2 pertaining ECS design that they are based on will be noted.

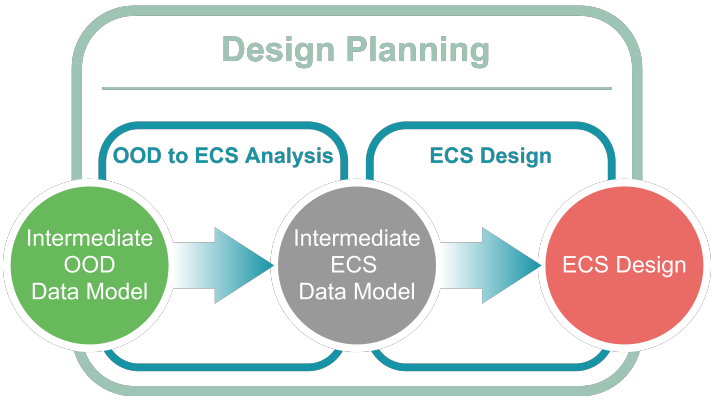


Figure 12: A simplified diagram of implementation of the design planning for an ECS design.

Merging based on Class With this parameter, the user can decide if the component fields should be combined into a component based on the similarity in class. If this option is set to false, each component member will have a separate component, i.e. it will create single-field components. If this option is enabled, component fields are merged if they have enough similarities in the classes in which they are found. How much similarity there needs to be can also be influenced. From only merging when the classes are an exact match to only needing 75% or 50% similarity. Lower is also possible but not recommended. This parameter is based on Rules 9, 11, 13 and 15, which are concerned with merging or splitting components based on domain boundaries.

Merging based on Systems This parameter works the same way as the previous. However, instead of considering similarities in classes, this concerns similarities in systems. If component fields are similar enough on in which systems they are used, they are combined in a component. This parameter is based on Rule 14, which is concerned with merging components based on how they are processed.

Split Read Access from Write Access The similarity between systems can be calculated in two ways. One is to treat read access to a variable and write access the same way. This means that the created components can be both read from and written to in a system. The other way splits the component fields on whether they are read or written to in a system. Components that are created this way are only read ór written in a system, never both. This parameter is based on Rule 10 that states read variables and write variables should be separated.

Split Blittable from Non-Blittable Variables If this is enabled, blittable types will be put in separate components from non-blittable. This parameter is based on Rule 12 that states that blittable data should be separated from non-blittable data in Unity's DOTS framework.

Order on Variable Type This feature orders the component fields in a component based on variable type. This ensures no mixing of variable types that can induce extra padding. This parameter is based on Rule 21 that explains that mixing variable types in a component can lead to unnecessary padding.

5.3.2 Planning Profiles

While the parameters can be altered independently, planning profiles can aid in tailoring the ECS design to the framework used. Pre-created profiles for specific frameworks can simplify

	General Profile			Unity Profile			Sparse Sets Profile		
	perf.	bal.	maint.	perf.	bal.	maint.	perf.	bal.	maint.
Class Merge	100%	100%	50%	false	100%	50%	50%	100%	100%
System Merge	100%	50%	50%	false	100%	50%	50%	50%	50%
Read/Write Split	true	true	false	true	true	false	true	true	false
Blittable Split	true	true	false	true	true	false	true	true	false
Order Variables	true	true	true	true	true	true	true	true	true

Table 1: The planning parameter values for each performance value. The design can be fully focused on performance, fully focused on maintainability or strike a balance between those two.

conversion. There are currently three planning profiles; the *General Profile*, the *Unity Profile* and the *Sparse Sets Profile*. The General profile uses the general knowledge gained from the interviews, while the two specific profiles use the knowledge gained for that specific framework.

In addition, the profiles can set the balance between performance and maintainability. The ECS design can be completely performance-oriented, completely maintenance-oriented or a balance of the two. Changing this value affects each planning profile differently, as where Unity's DOTS framework is more performant with single-field components, Sparse Sets-based frameworks are not. The values for each setting can be found in Table 1.

5.4 Plan Visualization

In addition to the requirements for the ECS design, the interviews also revealed requirements for the visualization of the design. By following these requirements, the transition between OOD and ECS can be aided. The two main categories are highlighting key relationships in the ECS design and showing the ties between the original OOD code and the generated ECS design.

5.4.1 Tool Overview

The tool screen is divided into two parts, see Figure 13. The left part lets the user set all parameters and choose a project on which to run the tool's analysis. This panel also shows the progress of the analysis. The right side provides the results of the parameters and analysis.

The right panel has three tabs. The first is for the advice that can help with choosing the correct parameters for the tool and manually specializing the design after it has been generated. The advice is linked to the profile chosen, displaying the specific advice for a specific framework. The second and third tab show the results of the analysis and designing phase of the tool.

5.4.2 OOD and ECS Code Comparison

The comparison tab in the tool enables the user to explore the links between the original OOD (on the left side) code and the ECS design (on the right side). Selecting an item on the left side will automatically display the matching item on the right side and vice versa. Classes can be compared to entities and methods to systems. These items can be further explored in the view tab.

The insights in Section 4.3.2 establish that showing the links between OOD and ECS design can help with transitioning between the two techniques. The compare tab implements this.

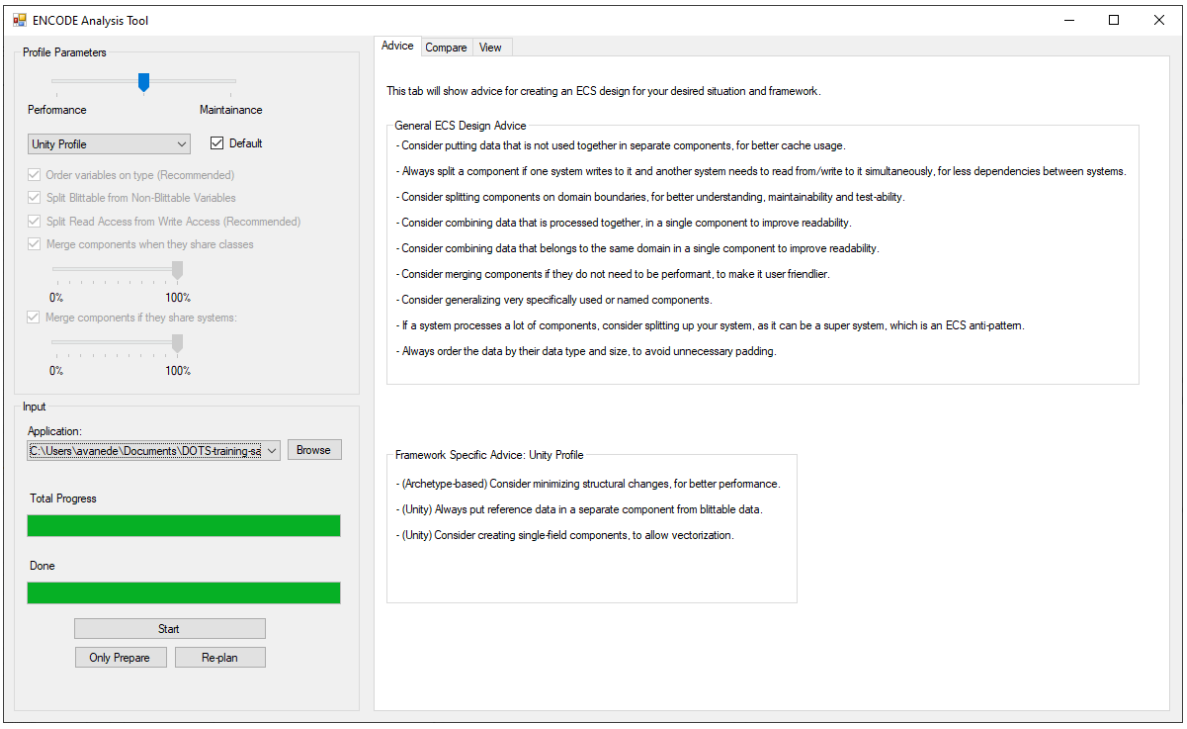


Figure 13: General overview of the tool interface with the general advice and the specific Unity-based advice.

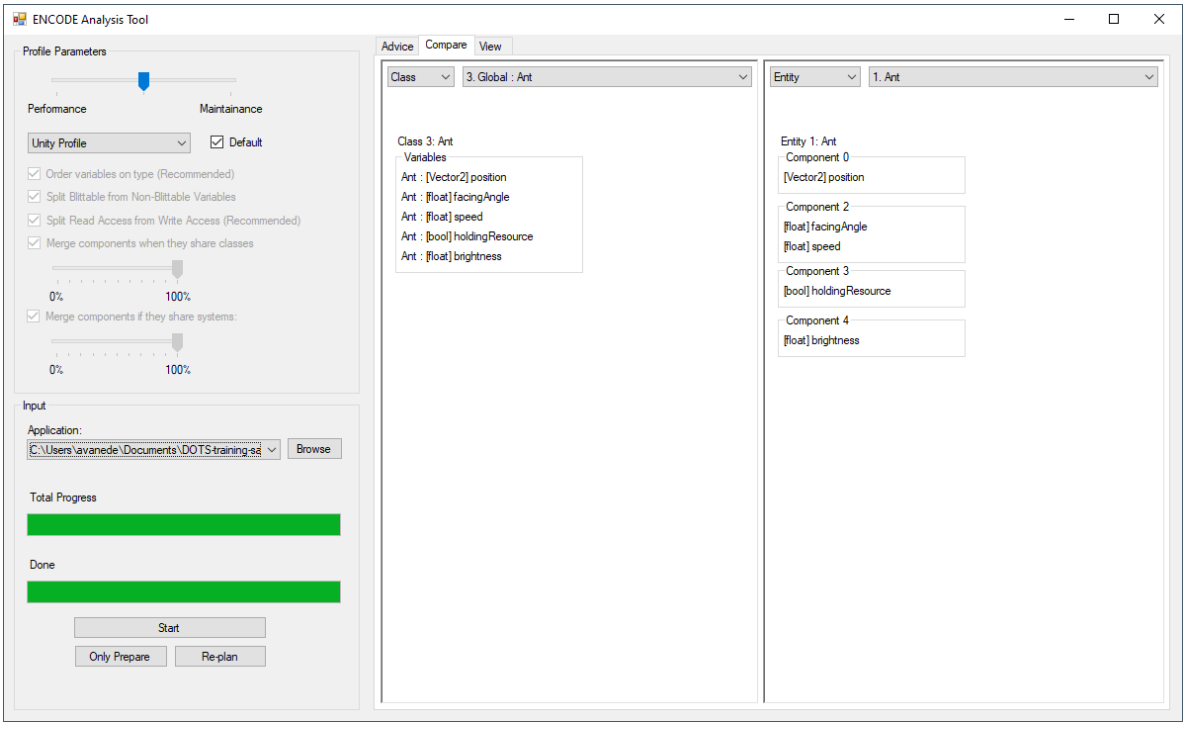


Figure 14: The visualization provided by the tool to compare the original OOD code with the generated ECS design.

5.4.3 ECS Design Relationships

An important relationship to visualize is the commonalities between different entity types. The relationship between systems and components is another important one. Figure 15 shows the visualization of the commonalities between entities. All components are shown in a grid. Components that are present in a specific entity type are drawn in black, creating showing the difference between the entity types. Components that are not present are drawn in grey. Component 0 is present in both the Obstacle and Ant entity types but not in the other two. The relationship between systems and components is shown similarly.

The user can explore the ECS design in detail in this visualization tab. Section 4.3.2 presents that showing the relationships within the design is an essential part of visualizing ECS design.

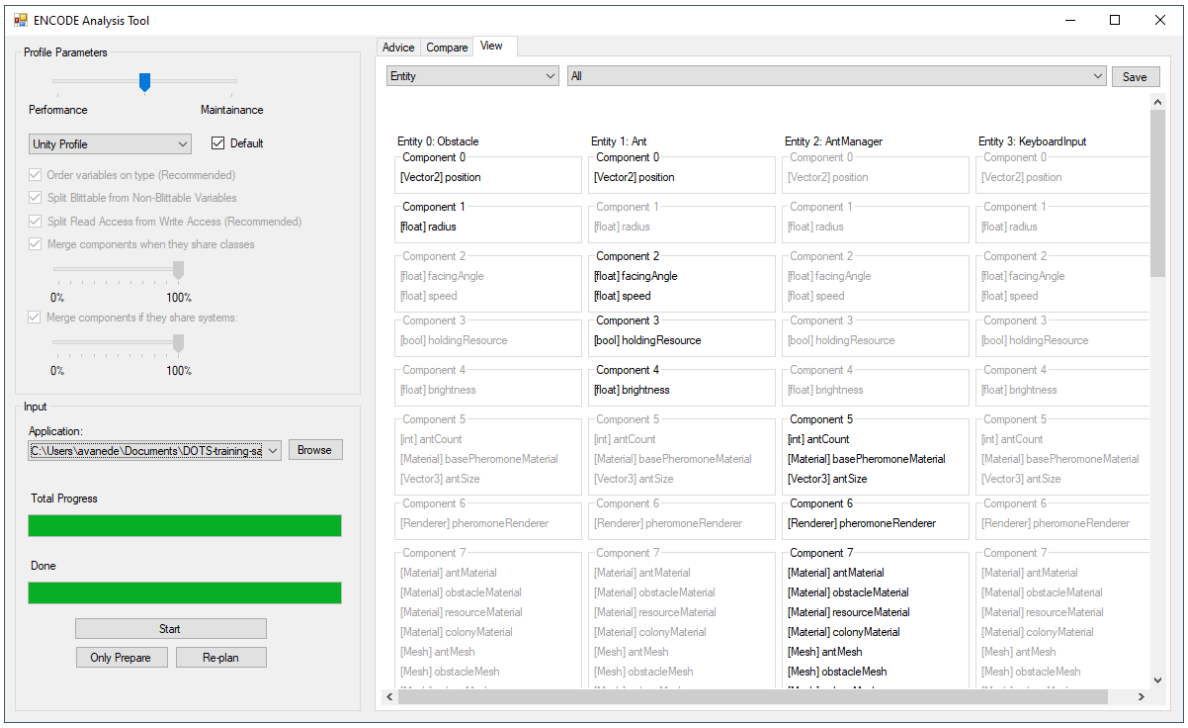


Figure 15: The visualization provided by the tool to view design relationships within the ECS design.

5.5 Concluding Results

This section introduced a method to convert from OOD to DOD. The three most important steps were explained; static program analysis, intermediate data model and design planning.

The method was then applied to create a tool that converted OOD code to an ECS design. It explained how each step was implemented and combined with the information on ECS design gained from the interviews. Lastly, it described how it implemented the visualization guidelines introduced in the interviews.

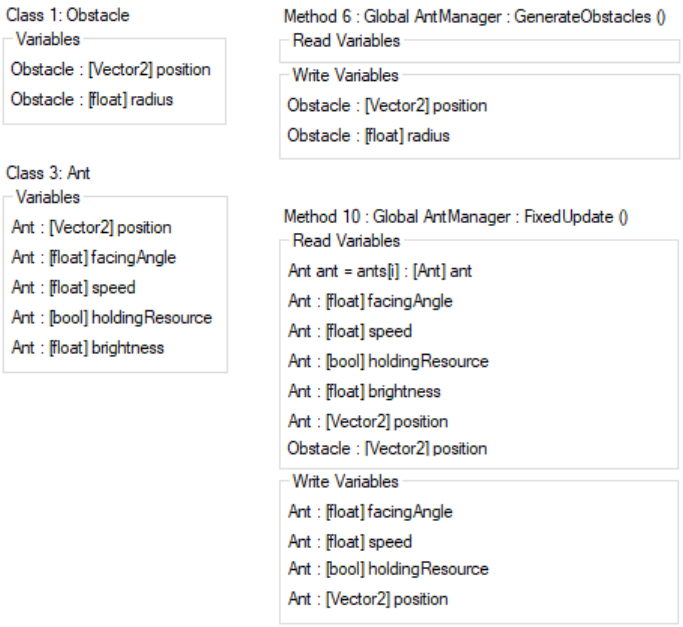


Figure 16: The base against which the parameters can be checked. The variables that are used in the methods but are not present in the classes have been removed for clarity.

6 Experimentation Results

To prove the conversion method introduced in Section 5.1 is valid, it is implemented for OOD to ECS conversion and tested on effectiveness by testing it on several sample projects.

Section 6.1 introduces the projects that the tools is tested on. Then, we discuss how the planning parameters influence the created design. Next, we compare the generated designs for the Unity sample projects to the manually converted design. Lastly, Section 6.3.3 explores how well the tool performs on a real-world application as XVR's On Scene (OS).

6.1 Setup

The tool is tested on five projects. All projects are C# software applications build in Unity 3D and Visual Studio.

The first four are sample projects that are used in Unity DOTS training. The participants of the training choose one of the 21 different available projects to convert from OOD to ECS by hand. The four projects that are chosen are the projects that are converted most often in the training. Because these projects are converted by hand in the training, we have access to the original OOD code and the manually converted ECS code. The projects that are chosen are AntPhermones (sic), BucketBrigade, CombatBees and HighwayRacers [2] .

The last project tested is XVR's On Scene (OS), a large and complex real-world application. This application is used to test how well the tool can convert large and complex software to ECS [20].

[2]The projects can be found at `https://github.com/Unity-Technologies/DOTS-training-samples/tree/master/Original`

6.2 Parameter Influence

This section will show the effectiveness of the planning profile parameters by showing their influence on the ECS design that the tool generates. For this purpose, we will take two classes from the AntPhermones project as an example. The two classes and the methods that use their class members can be found in Figure 16. The relevant parts of the generated ECS design can be found in Figure 17. Figures 17a through 17f show the same two entities created with six different parameter configurations.

Figure 17a shows an ECS design that does not split on blittable variables or read/write variables and variables only need 50% of classes and systems in common. The tool, therefore, put all data into a single component. The variables of this component are also not ordered by type to avoid padding.

Figure 17b has the same settings as the previous Figure, with the exception that the variables are now ordered based on variable type. This results in the boolean variable being moved from between the floats to the top of the component.

Figure 17c adds blittable variables splitting, splitting the non-blittable boolean *holdingResource* variable to a separate component.

Figure 17d adds read/write splitting and splits two more variables off. These two, *radius* and *brightness*, are only read in the systems, while the other variables are written.

Figure 17e was created with the same parameters as 17d, except for the class and system merge parameters. For the first design, the classes and systems needed to match 100%, ensuring that all components' variables were present in the classes they came from and are always used together in systems. Therefore *position*, which is present in both entities, is split off from $facingAngle$ and *speed* that are only present in the Ant entity.

Figure 17f has disabled the possibility for components to be merged based on class or system. Because of this, it creates single-field components, creating a separate component for each variable.

The results show that the planning parameters work as described in Section 5.3.1. Each parameter is capable of representing a choice in the ECS design process.

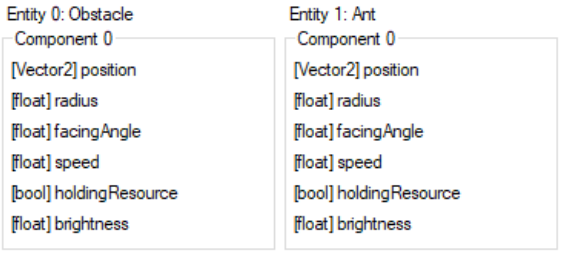
All created designs can be found at `https://github.com/AnneVanEde/MasterThesis`.

6.3 Sample Projects Comparison

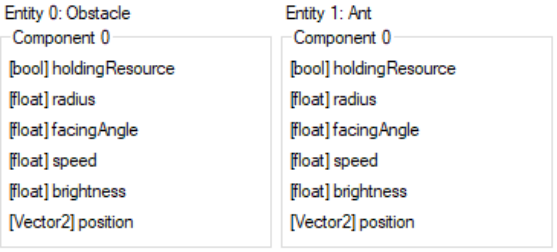
To show the overall effectiveness of the tool and the conversion method, we will compare the results from the tool to the manually created designs. We will compare the designs both on the components and the entity types and on the systems. Lastly we will look at the effectiveness of the tool on XVR's OS.

6.3.1 Components and Entities

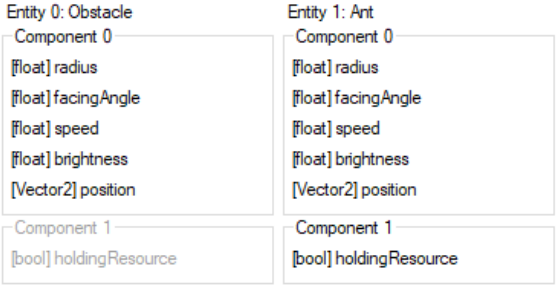
The results show a clear similarity between the ECS design created by the tool and the design that was manually created. Figure 18 shows an excerpt of the manual design that shows that the Ant entity is similar to the one found in Figure 17f. While there are some minor cosmetic changes to the design, the underlying structure is very similar. The minor differences are partially in naming, e.g. *brightness* is now described by $URPMaterialPropertyBaseColor$. Another change can be found in the boolean *holdingResource* that is interchanged for the tag



(a) Parameters: No ordering, no blittable splitting, no read/write splitting, 50% class, 50% system.

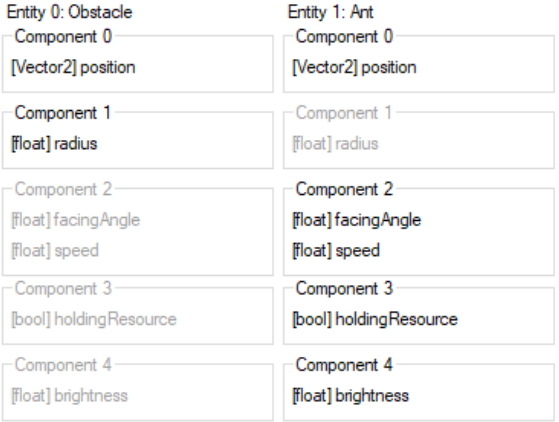


(b) Parameters: Ordering, no blittable splitting, no read/write splitting, 50% class, 50% system.

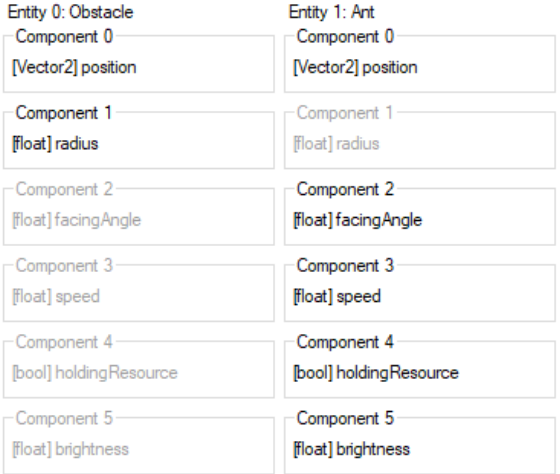


(c) Parameters: Ordering, blittable splitting, no read/write splitting, 50% class, 50% system.

(d) Parameters: Ordering, blittable splitting, read/write splitting, 50% class, 50% system.



(e) Parameters: Ordering, blittable splitting, read/write splitting, 100% class, 100% system.



(f) Parameters: Ordering, blittable splitting, read/write splitting, single-field components.

Figure 17: Two entities from the ECS design with different parameter configurations. Components that are drawn in black are present in the entity and components drawn in grey are not present.

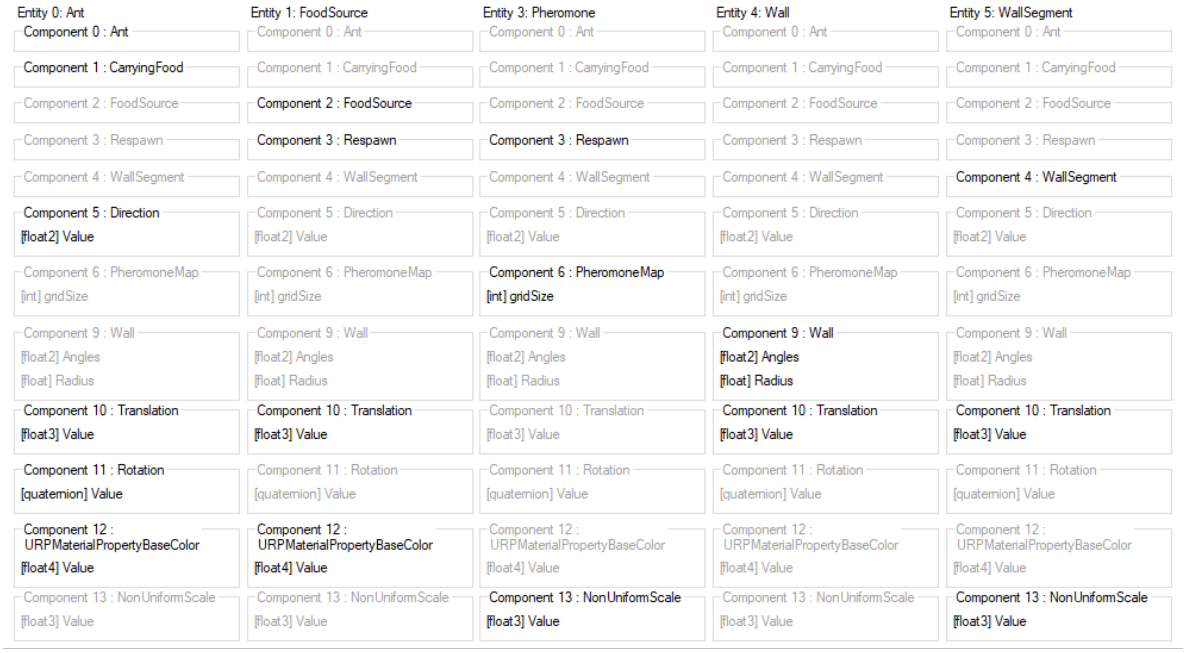


Figure 18: An excerpt of the manually created ECS design for the AntPhermones project.

component *CarryingFood* with the same purpose. The *speed* variable is the only one that is not present, as it has been moved to a different entity.

However, there are also considerable differences between the designs. This is most visible in classes that handle a lot of the game logic or represent more than one real-world concept. These classes handle all the processing in the code, and the tool created a single entity for them. The manually created design, however, created many different entities of these classes. All different aspects of these classes have been split up into separate entity types. Some of these entity types are singleton type entities and components, as they only concern application settings.

The other projects show the same patterns. The small classes that presented a single real-world concept were converted very similarly to the manual design. The classes that contained multiple concepts or were managers for the whole applications were less accurate. The manual design splits these into several parts, where the automatic design leaves it together.

6.3.2 Systems

The tool performs less than optimal when converting methods to systems. When compared to the manual ECS design, the automatically generated systems are not very similar. Some small methods are converted to systems that are similar to their manually created equivalent. However, most automatically generated systems are much bigger and handle many components. The manual systems are much smaller and split up to handle fewer responsibilities of the game logic. This is most likely since some methods contain the main loop of the OOD software. Therefore they handle almost all data in the program. The tool cannot split up methods, and therefore the two designs differ. This is an expected limitation of the design.

In addition, some systems process no components at all. This is because some methods are helper methods, only used to calculate a part of data for another method. The tool can not discriminate between these two with the current analysis method and creates separate systems for both.

Lastly, because the methods do not need to perform an entity query to gather their data, they sometimes use different entities. This is something that is discouraged in ECS design

as it introduces dependencies. The tool, however, converts the methods as they are, without avoiding these dependencies.

Despite limitations, the methods that are converted to systems are generally recognizable.

6.3.3 XVR's OS Performance

Finally, ENCODE was tested on a real-world application; XVR's OS. This code has a significantly more complex structure than the Unity sample projects used for the previous analysis. This project is used to measure the robustness and performance of the tool in a real-world setting.

In general, OS was handled satisfactory, and a design was created. However, besides taking significantly longer to analyze the code and generate the design (minutes instead of seconds), the more complex structure seemed to be the biggest hurdle. OS uses more complex patterns and more advanced features of both C# and Unity. With the known limitations of static program analysis, these results were expected. Fortunately, upon inspection, most of the variables not handled by the tool were irrelevant for the outcome. The remaining variables could easily be handled by more complex analysis.

The tools current visualization is also slightly problematic, as the design is so large that it cannot be displayed clearly. Details of the design are still comparable, but the similarities between enSamtities cannot be seen. A small change enabling comparing only part of the entities would solve this problem.

Besides some shortcomings, the complex real-world project was handled well and created an ECS design as expected.

6.3.4 Concluding Results

The conversion of entities and components showed promising results. Small classes that represented a single concept were converted similarly to the manual conversion. Some cosmetic differences are seen, but the underlying structure is the same. Larger classes that handled more than one concept were less accurately converted. This is because the manual design split them up, while the automatic design could not.

The systems design was of less quality. Small methods were converted well, but very big or very small methods did not perform as well. As ENCODE is incapable of splitting up methods, its design did not match the manual designs that did split up methods. Helper functions were also converted to separate systems, as the tool cannot distinguish between standard methods and helper functions.

Large real-world applications are generally handled well. However, the processing time is longer, and visualization less clear than the other sample projects. In addition, the complex software structure cannot be handled entirely by the tool because of the limitations of static program analysis.

Despite limitations, classes and methods are converted to entity types, components and systems. The original classes and methods are recognizable in the created ECS designs, and parallels can be seen with the manual designs.

7 Discussion

In the previous sections we tried to fill the gap in literature on ECS, ECS design and ECS design visualization. In addition, we created a tool that converts OOD code to an ECS design and educates the user on the design choices. In this final chapter, we discuss the findings of this research. We discuss the effectiveness of our method, the limitations of our work and the relevance of our research.

7.1 ECS Knowledge Base

Where other design strategies have literature on characteristics, best practices and standardized visualization, this knowledge is absent for ECS. To remedy the lack of research and literature on ECS, we performed qualitative interviews with experts in the field of ECS.

When using interviews to gather knowledge, a representative group of people should be chosen as participants. However, this research is not interested in the general view on ECS, but the best practices used by the most experienced users of the technique. Especially because newcomers to the technique are prone to suboptimal practices and designs, they are not a relevant group to interview. Instead, only people who have extensive knowledge on ECS were interviewed to create a more competent and practical view on ECS design.

The results as given in Chapter 4 was the consensus of all interviewees. The majority of the answers given by the participants matched with each other, and with the little research available on DOD. The few differences found were primarily due to experience with a different framework and are noted in the results.

As the view of these experts lined up with each other and the literature, despite the small number of participants, having a larger sample group is unlikely to alter the results drastically. Therefore, the gathered information can be viewed as a well-founded knowledge base on ECS.

7.2 Conversion Method

Besides aiming to fill the knowledge gap, this thesis also introduces a novel method to convert from OOD to DOD. It validates the method by implementing it to convert from OOD to ECS design with only OOD source code as input. This section will discuss the limitations of the introduced method by looking at the effectiveness of the three processing steps in the created tool, ENCODE.

Static Program Analysis The conversion method makes use of static program analysis, which is known to be able to reverse engineer design patterns [19]. It can identify classes, class members and methods as well as their relationships to each other. The method uses this information to record the structure of the OOD source code.

However, there are limitations to what static program analysis can do in Object-Oriented languages such as C#. For example, polymorphism method calls cannot always be analyzed at compile-time; it is unknown if they call the parent class or the derived class. The limitation of static program analysis might have lead to the more complex project, XVR's OS, being handled imperfectly.

In addition, the comparisons between ENCODE designs and manual designs in Section 6.3 show that smaller classes and methods are converted more accurately than the big ones. This is because the analysis is incapable of splitting up classes and methods. The method and the tool are therefore heavily dependent on the structure of the input. This decreases the accuracy of the overall design.

More complex analysis steps that could solve these problems was not attempted as it was out of the scope of this thesis. XVR's OS is also likely to be handled better with more

complex analysis. With the current analysis, the shown limitations are inevitable. Future research could extend the analysis technique to improve the generality of the code analysis.

Intermediate Data Model The results of the static program analysis are stored in an intermediate data model. It allows the conversion method to convert between any two design strategies and is applicable to different languages. ENCODE shows that the data model is effective in storing the data structure from the input code.

Planning Profiles To accommodate for different design requirements, planning profiles were used to alter the output design. ENCODE has shown they are effective at improving the quality of the designs without having to change the general design process.

7.3 ENCODE Evaluation

In this section we will discuss the visualization of the design in ENCODE and the projects used for evaluating the tool.

Visualization As shown in Section 5.4, the implementation of the visualization in the tool was generally effective. Like Section 4.3.2 suggests, the relationship between the two designs and the relationships within ECS are visible. Only when XVR's OS is visualized, some problems occur. The detail views and comparison between OOD and ECS work fine, but the design as a whole is too large to be visualized in its entirety. This could be fixed by letting the user choose to visualize only parts of the overview.

Sample Projects The four sample projects from Unity are simple. They are representative of beginners projects used to learn the technique as Unity uses them to teach their employees about ECS. Testing the effectiveness of the tool on such projects imitate the intended use of the tool; used at the start of learning ECS. Testing ENCODE on XVR's OS is a good indication of the tool's performance in a real-world setting.

7.4 Concluding Remarks

ENCODE extracts the data structure from the source code and with the help of the intermediate data model and the planning profiles an ECS design is created. The results showed the correctness of the analyses and the accuracy of the designs for both the small sample projects as the real-world sample project. Despite the limitations, ENCODE proves the effectiveness of the proposed conversion method.

8 Future Work

This thesis proposed a OOD to DOD conversion method and successfully implemented this technique in ENCODE. While the results of this thesis were satisfactory, future work could improve the results and usability of ENCODE.

Firstly, the code analysis could be improved. As Section 7.2 mentions, some analysis steps were not attempted as they were out of scope of this thesis. Adding more advanced analysis techniques could improve accuracy of the designs, especially for the real-world projects. Among other, dynamic program analysis (gathering information at runtime) could be included to solve the limitations of static program analysis in Object-Oriented languages.

In addition, the tool is dependent on the structure of the input code. A badly designed input will lead to a suboptimal ECS design. To improve the input of the tool, and with it the output, the tool could notify the user of large methods or classes that would negatively impact the accuracy of the design. ENCODE could advice the user to alter them before the the ECS design is created. This could happen at the end of the static program analysis.

Furthermore, the tool cannot automatically implement all design choices flawlessly. After the design is created, manual improvements should be made to make the design fit better to the project. To help the user with this, an interactive feature could be implemented. The tool could present warnings when a part of the design needs attention, and give advice on how to do these alterations. The user could also be enabled to change parts of the design and have the rest of the design re-planned based on the changes.

Beside the analysis and planning, the visualization could also be improved. Allowing the user to zoom in on a part of the design or even choose a subset of, for example, entities to show on the screen could give an even clearer view of the design relationships. Especially large designs could benefit from this.

Lastly, the proposed conversion method can be used to convert between OOD and other DOD strategies outside of ECS. For example, the method could be applied to converting an OOD to a SOA design.

9 Conclusion

The first goal of this thesis was to contribute knowledge on ECS to the gap in the literature. The results show that components should be created based on how they are used and to what concept domain they belong. In addition, the project requirements can influence the balance between performance and maintainability. This balance influences the design choices of the ECS design. The information gathered can be the basis for standardized design practices.. This research can function as a starting point for more research in Data-Oriented design patterns.

Next to design principles, this research gives some criteria on ECS designs visualization. Visualization should show the important relationships within and ECS design, such as the relationship between a system and the components it processes. Furthermore the connection between the design and the original OOD should be visible. Standardizing visualization could aid in understanding and converting to ECS. We argue that this thesis made a significant contribution on ECS and its design and visualization.

The second goal of this thesis was to introduce a novel method for design strategy conversion. We show that the combination of three techniques from unrelated domains could convert from an OOD to DOD. We show the validity of this conversion method by applying it to a design conversion from OOD to ECS. ENCODE takes OOD C# code and uses the described method to automatically generate an ECS design. The comparison with manual designs shows that the conversion and designing method is effective at generating an ECS design.

In conclusion, data and cache-based bottlenecks are only likely to increase in frequency and severity. Therefore, the necessity to adopt a Data-Oriented Design strategy will increase along with it. This thesis gathered information on ECS, proposed a OOD to DOD conversion method and presented an OOD to ECS conversion tool, ENCODE. Research on DOD and ECS and the conversion from OOD to DOD is likely to be increasingly relevant in the future. The findings of this research can be used to perform more in-depth research on these topics in the future.

Acknowledgment

This thesis would not have been possible without the contributions of numerous other individuals. In the first place Jacco Bikker, for tirelessly reading my work and advising on how best to present it. Secondly to Taco Petri and the other developers at XVR for listening to my programming plans and giving advice. We are also grateful for all the experts who gave their time to be interviewed and even answer questions and give feedback afterwards for clarification. Many thanks to my family and friends who gave feedback on the thesis, both on the process as the written work. Lastly, we also acknowledge XVR for supporting the first author during the research.

References

[1] F. Alted. Why modern CPUs are starving and what can be done about it. *Computing in Science & Engineering*, 12(2):68–71, 2010.

[2] E. Baumel. Understanding data-oriented design for entity component systems - Unity at GDC 2019. `https://www.youtube.com/watch?v=0_Byw9UMn9g&t=660s`, Apr 2019. (Accessed on 25-05-2021).

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and Thomas T. Lawrence. Parallel programming with Polaris. *Computer*, 29(12):78–82, 1996.

[4] M. Caini. Gaming meets modern C++ - a fast and reliable entity component system (ECS) and much more. `https://github.com/skypjack/entt`. (Accessed on 06/28/2021).

[5] U. Drepper. What every programmer should know about memory. *Red Hat Inc.*, 11:2007, 2007.

[6] D. Efnusheva, A. Cholakoska, and A. Tentov. A survey of different approaches for overcoming the processor-memory bottleneck. *International Journal of Computer Science and Information Technology*, 9(2):151–163, 2017.

[7] R. Fabian. *Data-oriented design: software engineering for limited resources and short schedules (book)*. R. Fabian (self-published), 2018.

[8] K. Fedoseev, N. Askarbekuly, A.E. Uzbekova, and M. Mazzara. Application of Data-Oriented Design in Game Development. In *Journal of Physics: Conference Series*, volume 1694, page 012035. IOP Publishing, 2020.

[9] S. Garcia, D. Jeon, C.M. Louie, and M.B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. *ACM SIGPLAN Notices*, 46(6):458–469, 2011.

[10] D. George, P. Girase, M. Gupta, P. Gupta, and A. Sharma. Programming language inter-conversion. *International Journal of Computer Applications*, 1(20):68–74, 2010.

[11] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S. Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996.

[12] D. Jeon, S. Garcia, C.M. Louie, and M.B. Taylor. Kismet: parallel speedup estimates for serial programs. In *Proceedings of the 2011 ACM international conference on Object Oriented programming systems languages and applications*, pages 519–536, 2011.

[13] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. *ACM SIGOPS Operating Systems Review*, 32(5):46–57, 1998.

[14] N. Llopis. Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). *Game Developer*, 16(8):43–45, 2009.

[15] Microsoft. The .NET Compiler Platform SDK. `https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/`, Oct 2017. (Accessed on 15-06-2021).

[16] A. Møller and M.I. Schwartzbach. Static program analysis. *Notes. Feb*, 2012.

[17] R. Nystrom. *Game Programming Patterns (book)*. Genever Benning, 2014. Online available at: http://gameprogrammingpatterns.com/contents.html.

[18] H. Seidl, R. Wilhelm, and S. Hack. *Compiler Design: Analysis and Transformation (book)*. Springer Science & Business Media, 2012.

[19] N. Shi and R.A. Olsson. Reverse engineering of design patterns from java source code. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 123–134. IEEE, 2006.

[20] XVR Simulation. Xvr simulation - incident command training tool for safety and security. `https://www.xvrsim.com/en/`. (Accessed on 07/15/2021).

[21] S. Sprenger, S. Zeuch, and U. Leser. Exploiting automatic vectorization to employ SPMD on SIMD registers. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 90–95. IEEE, 2018.

[22] Unity. Entity Component System - Unity Manual. `https://docs.unity3d.com/Packages/com.unity.entities@latest`, Oct 2020. (Accessed on 15-01-2021).