



Performance comparison between OOD and DOD with multithreading in games

David Wingqvist
Filip Wickström

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author(s):

David Wingqvist

E-mail: daag19@student.bth.se

Filip Wickström

E-mail: fai18@student.bth.se

University advisor:

Senior Lecturer Dr. Suejb Memeti

Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. The frame rate of a game is important for both the end-user and the developer. Maintaining at least 60 FPS in a PC game is the current standard, and demands for efficient game applications rise. Currently, the industry standard within programming is to use Object-Oriented Design (OOD). But with the trend of larger sized games, this frame rate might not be maintainable using OOD. A design pattern that mitigates this is the Data-Oriented Design (DOD) which focuses on utilizing the CPU and memory efficiently. These design patterns differ in how they handle the data associated with them.

Objectives. In this thesis, two games were created with two versions that used either OOD or DOD. The first game had multithreading included. New hardware utilizes several CPU cores, therefore, this thesis compares both singlethreaded and multithreaded versions of these design patterns.

Methods. Experiments were made to measure the execution time and cache misses on the CPU. Each experiment started with a baseline that was gradually increased to stress the systems under test.

Results. The results gathered from the experiments showed that the sections of the code that used DOD were significantly faster than OOD. DOD also had a better affinity with multithreading and was able to achieve at certain parts up to 13 times the speed of equivalent conditioned OOD. In the special case comparison DOD, even though it had larger objects, proved to be faster than OOD.

Conclusions. DOD has shown to be significantly faster in execution time with fewer cache misses compared to OOD. Using multithreading for DOD presented to be the most efficient.

Keywords: Game development, C++, Execution time, CPU cache, OpenMP

Acknowledgements

We would like to thank our supervisor Dr. Suejb Memeti and all the other supporters for the helpful feedback and discussions.

Contents

Abstract	i
Acknowledgements	iii
Glossary	vii
1 Introduction	1
1.1 Aim and objectives	2
1.2 Research questions	2
1.3 Thesis structure	2
2 Background	3
2.1 What is the memory hierarchy?	3
2.2 How does the CPU cache work?	4
2.3 What is OOD?	4
2.4 What is DOD?	5
2.5 Why use DOD?	6
2.6 What is multithreading?	7
3 Related Work	9
3.1 Our Work	10
3.1.1 Unique work	10
3.1.2 The game idea	11
3.1.3 Visualisation of the games	11
4 Method	13
4.1 Systems Under Test	14
4.2 Experiments	15
4.2.1 First experiment	15
4.2.2 Second experiment	15
4.2.3 Validating the tests	16
4.3 Implementation	16
5 Results	17
5.1 Execution time	17
5.2 L1 cache misses	20
5.3 Last-level cache misses	22
5.4 Comparing the design patterns	24

5.4.1	Execution time	24
5.4.2	Multithreading improvement	25
5.4.3	Cache utilization	25
5.5	Special case execution time comparison	26
6	Discussion	29
6.1	Implementation biases	29
6.2	Implications of the results	29
6.3	Analysis of special case	30
6.4	Cache misses and their relation to execution time	30
6.5	The access pattern of the two design patterns	31
6.6	The effect of multithreading	31
6.7	The complexity of both design patterns	32
7	Conclusion and Future Work	35
	References	37

Glossary

CPU Central Processing Unit - The component in the computer that executes instructions by programs. It handles arithmetic, controlling, logic and input/output operations depending on the program.

CPU cores A modern CPU has multiple physical cores within the chip. Each core can work with a separate task. This makes it possible to let the CPU do multiple tasks at the same time.

CPU threads Is simulated virtual versions of CPU cores. Each physical CPU core can have up to two threads running at the same time. Having two threads per core is called 'hyper-threading' on Intel CPUs and 'simultaneous multithreading' on AMD CPUs. This makes the CPU work as it had twice the amount of cores.

DOD Data-Oriented Design - A design pattern that puts the focus around orienting the code to be efficient for the data. This includes organizing the data in memory to efficiently utilize the CPU cache.

DRAM Dynamic Random-Access Memory - The computer's main memory that is used for every running application in the operating system. Often just called RAM. Data is lost when power is off.

ECS Entity Component System - Is a software architectural pattern that is used in game development for handling game objects.

FPS Frames Per Second - How many times per second a game can update and render to screen.

OOD Object-Oriented Design - An abstract design pattern that utilizes concepts such as objects and inheritance. Focuses on code re-usability and readability.

OS Operating System - System software that handles the computers hardware.

SRAM Static Random-Access Memory - Fast but expensive memory that is used in the CPU. Data is lost when power is off. Often has multiple cache levels where L1, L2 and L3 are the most common. L1 is closest to the CPU and the fastest and the smallest of the three.

VM Virtual Machine - An emulated computer system within a software.

Video games are applications that need to run in real-time. The standard in today's PC games is to be able to maintain a frame rate of at least 60 FPS. Anything below that can be seen as not responsive, slow and a bad experience for the user. Since modern games' scale gets larger every year, it is important to efficiently utilize each component of the PC to its full potential [1].

The current industry standard for writing code is to use the OOD pattern. This is since it is easy to understand, write and re-use code [2]. However, it is not the most optimal pattern to use in terms of CPU performance. The objects created in OOD are placed in memory unorganized which will impact the CPU cache usage negatively. It typically requires several fetch requests to complete its operations. This is detrimental to the efficiency of the application.

Over the past years, the gap between CPU speed and memory speed has been growing tremendously. This is further enhanced by how the two design patterns access their data. A solution to minimize this problem is by efficiently utilizing the CPUs cache [3] [4]. A design pattern within software development that can give more control over the cache is DOD, but it requires planning ahead of implementation. The location of the data in the memory is important for the cache. Namely, the spatial and the temporal locality principles. Therefore, it is of interest to know how the design patterns utilize these aspects. OOD might not guarantee the allocation of its member variables close to each other. DOD interests both how variables are allocated and accessed, which in turn takes care of the spatial and temporal locality problem [5] [6].

In this thesis, two games will be implemented using the two design patterns, OOD and DOD. One of the games will be using multithreading for both of the patterns. Performance metrics such as execution time and cache miss rate will be monitored to compare the CPU performance of OOD and DOD. As a result, the thesis aims to highlight the different aspects of performance that gets affected. The first game is entirely created by the authors with ideas gathered from other implementations. Thus, several bias factors will affect the performance of the tests conducted. As for the second game, the DOD implementation is produced with the EnTT¹ library. This library is generalized and professionally created. It is one of the most popular ECS libraries and is being used in the successful game Minecraft.

The experiments in the thesis aim to discover how the cache is utilized with both of these design methods. Multithreading is included as a deeper subject in how the cache would work with several CPU cores. This is interesting since some level of

¹Available at: <https://github.com/skypjack/entt>

cache is shared between the cores. The most generally shared level of cache is the last one which in this thesis is the third level, L3.

1.1 Aim and objectives

The aim of this thesis is to compare the CPU performance between DOD and OOD within the context of a small game. It also includes comparing the two design patterns but with multithreading. To achieve this aim, two simple games were developed with both of the patterns. Each author developed a game individually using both of the patterns and teamed up for the multithreaded version for one of the games. When the games had been developed, measurements were taken on the CPU with different amounts of game objects. These measurements included execution time and cache misses.

1.2 Research questions

For this thesis, two questions were created to divide and clearly state the main objectives of the thesis. First and foremost, the aim of this thesis is to compare the two design patterns against each other and the effect on the CPU. As a deep-dive and further investigation, multithreading was included. Computers from recent years have several CPU cores and CPU threads, where multithreading will likely be prevalent. Using multiple threads would affect the cache usage differently, as well as execution time. Since some levels of cache are shared between the cores, investigating how threads work together with cache is relevant. The data access pattern is bound to change depending on if singlethreading or multithreading is used. Listed below are the two research questions that will be answered in this thesis.

- RQ1: What are the performance differences on the CPU on aspects such as execution time and cache miss rate between Object-Oriented Design and Data-Oriented Design?
- RQ2: What are the performance differences on the CPU on aspects such as execution time and cache miss rate between the multithreaded versions of Object-Oriented Design and Data-Oriented Design?

1.3 Thesis structure

Chapter 2 will introduce the background to this thesis. The related work that conducts similar investigations to this thesis will be described in chapter 3. This chapter will also present what is special about this thesis and the developed games. The method that will be used for this thesis will be presented and discussed in chapter 4. Chapter 5 will display the results from the gathered data. The different aspects of the results and the general field will be discussed in chapter 6. Lastly, conclusions will be drawn from the results and potential future work will be suggested in chapter 7.

In this chapter, background information on DOD, OOD, the CPU cache, multi-threading, and memory hierarchy will be explained.

2.1 What is the memory hierarchy?

A computer has different layers of memory with different speeds and sizes. A visualisation of these layers can be seen in figure 2.1. At the top, closest to the CPU, is the register layer which is the smallest but also the fastest memory in the computer. After this comes the cache layer, which is larger but slower than the previously mentioned memory [3]. This layer itself holds different levels and is often called SRAM. Most modern CPUs nowadays have three levels of SRAM which are called L1, L2 and L3 [7]. L1 is the fastest and closest to the CPU but also the smallest. Every core in the CPU has an L1 and L2 while the L3 is shared between all cores. Following that is the main memory, DRAM which is often just called RAM. There are even more layers after this, but they are not relevant to this thesis and will not be explained.

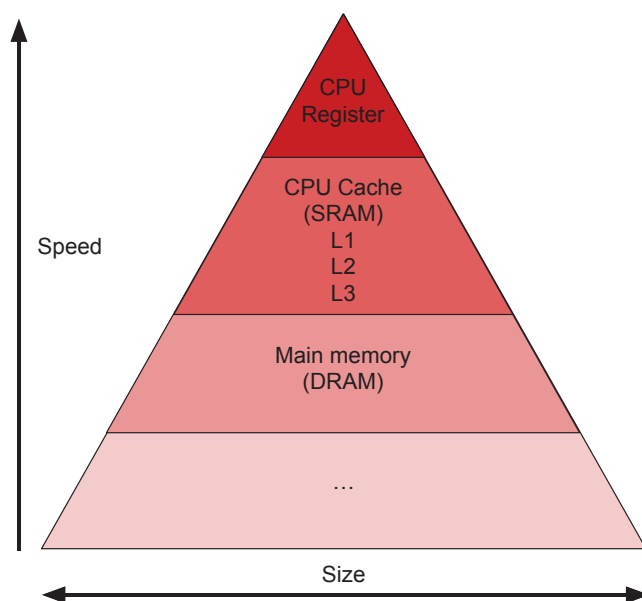


Figure 2.1: Illustration of the memory hierarchy.

2.2 How does the CPU cache work?

The cache within the CPU exists to speed up the computational work. Fetching data from RAM creates overhead, and the cache exists to mitigate this problem. Each time the CPU needs data it looks through the cache layer, starting at L1 until it reaches the last level, and if necessary, the RAM. When it finds the data, it copies it through the cache levels until it reaches L1. Since the data is now present in the L1 cache the CPU can quickly perform computations on it.

The cache is not only comprised of data but also the instructions that will be executed on said data. Instructions are fetched by the cache and stored for fast access in case of repetition and work similarly to previously mentioned data.

Terms such as temporal and spatial locality are of concern for the cache. For explanation, the temporal locality is the situation of utilizing a chunk of data again in the near future. Spatial locality on the other hand is the situation of using other data that is close to the accessed data [5] [6].

When data is fetched from the RAM, the CPU uses cache lines to fetch chunks of cache instead of individual data addresses. By keeping this in mind while implementing code, it is possible to speed up the application performance since the CPU is capable of utilizing the cache more efficiently.

If efficiency is of importance for the application, then utilizing the CPU to the highest extent is equally important. Fetching data from the cache or the RAM can affect the speed of the application tremendously. Lengyel [4] estimates that fetching data from the RAM takes around 500+ CPU cycles. Fetching from L2 takes 30-50 cycles, L1 takes 5-8 cycles and the register requires only one cycle. This means that fetching data from the cache should be done as often as possible for the application to be efficient.

2.3 What is OOD?

OOD is a design pattern within software development that focuses on objects and the re-usability of code. An object has a chunk of data and functions that can be used to change the data. Features specific for OOD are association and inheritance. Association creates a relationship between two classes that makes it possible for one of them to use the other. Inheritance is used when there is a lot of similarity between the two classes but still some differences. Figure 2.2 shows a base class called 'CircleObject' with common data and functions that can be used for child classes of the base class. The derived, or child classes such as 'Food' and 'Player', inherit the information from the base class and add individual information on top of it.

The advantage of OOD is how fast and effective it is to write code with the help of re-using code with inheritance. By encapsulating data, and isolating changes into classes, it is easier to debug and isolate problems in the code.

The disadvantage on the other hand is that OOD is not the most efficient pattern to use in terms of performance. The creation and destruction of objects are expensive. Many objects use complex associations, which leads to bad memory access times [2].

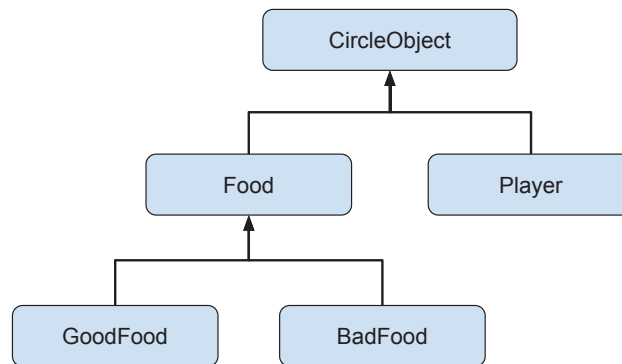


Figure 2.2: Inheritance between classes in OOD.

2.4 What is DOD?

DOD is a design pattern that aims to organize the data to achieve efficient utilization of the CPU cache. It differs from OOD by not utilizing abstract concepts such as objects and inheritance between classes. DOD's philosophy lies in the critical thinking of the data as one of the most important factors within the application. The pattern focuses on how to transform the data into actions the computer can perform in an effective manner [8].

As for understanding the code, DOD could be difficult for a beginner. This is since writing the code focuses intently on memory organization and the efficiency of the CPU. It presents a higher learning curve, but in return, it offers better performance when implemented correctly.

A common approach to DOD within the gaming industry is by using an ECS. It organizes the data while still maintaining code readability. Entities are simply integers that represent who they are. Components are the data that represent what these entities contain. They are not directly connected to each other, however, by using manager systems, components are linked to which entity they represent. Components are packed close to each other within the memory by using arrays. With these components being spatially close, they are guaranteed to be fetched in groups when a fetch request is issued. This in turn efficiently allows the CPU cache to continue its work with pre-fetched data in the next cycle.

In figure 2.3, two different methods of organizing data are displayed. An array of structures displays the OOD method of organizing data. On the other hand, structures of arrays are the DOD method of organizing data. This is organized to maximize the potential of the CPU. By using the DOD method, the chance of catching several components in the same fetch request is increased. This is performed since loops iterating over the components need to be fast.

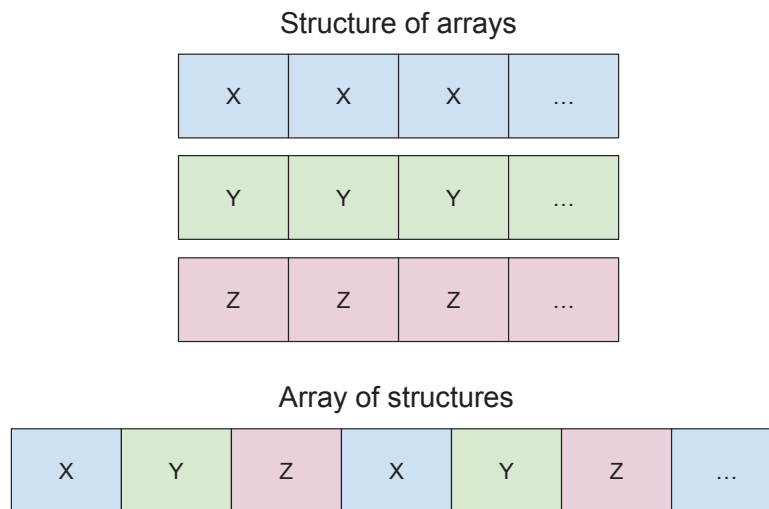


Figure 2.3: Different types of structuring of the data.

2.5 Why use DOD?

Games were not big in the early days of the game industry. Since then, game quality and world sizes have only increased. This presents a problem. How can a computer manage these sizes? As mentioned before, OOD is the current standard because of its simplicity to use. This however might not be scalable with the increasing world sizes. With the larger game worlds comes even more computational work such as artificial intelligence for non-player characters, moving projectiles, and more. All these aspects need to be checked and calculated accordingly.

To hold on to these aspects, another design pattern needs to be utilized since OOD most likely won't be able to keep up. With the problem of the gap between the memory and processor speeds, and how these design pattern utilize the RAM, DOD is the optimal design pattern to use. The demand for increasingly immersive games has been clear throughout history. Utilizing DOD would allow for more immersive games to exist while also increasing the size of the world inside the game [1] [3].

2.6 What is multithreading?

When an application utilizes several threads, it is called that the application is 'multithreaded'. Multithreading, as the name suggests, is the act of using several threads to divide the work between the cores of the CPU. A thread can be seen as a worker that can run separately from the main thread. Generally, the optimal method of dividing work is to allow one thread to run per available physical or virtual core.

There are different methods to handle multithreading. For this thesis, multicore and hyperthreading have been used. A visualisation of these two methods can be seen in figure 2.4. Multicore utilizes each of the CPU cores. The number of cores is often an even number where the most common nowadays is four, eight or 16. Multicore allows the CPU to divide work between these cores, each core can only handle one thread at a time. These cores share some cache levels with each other [9].

Hyperthreading on the other hand is only a little different. Same as multicore, it utilizes several cores. However, these cores are divided into virtual cores. These cores can then handle two threads at a time. This means that e.g. a four-core system can handle eight threads at a time. This doesn't double the power of the core, but it gives a boost of approximately 40% [10].

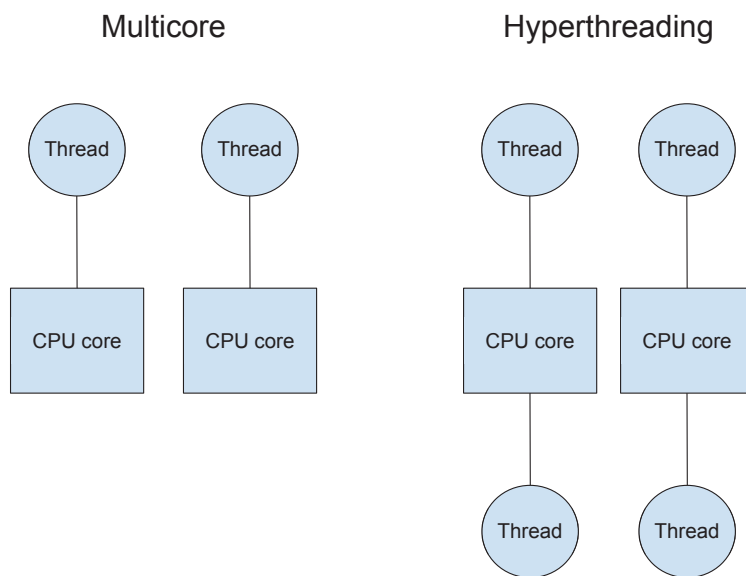


Figure 2.4: How the two multithreading methods handle threads.

Previous research articles about comparing DOD and OOD are very few and most other students' works were prevalent. As for the published research articles, some included focusing on game aspects, while others focused on general programming. In this section, several different related works within this specific field of programming will be presented and explained.

Fedoseev et al. [11] made a case study on OOD and DOD that executed similar tests as this thesis. Two games were developed, one using DOD and the other using OOD. Through the Unity game engine, the games were created, and the profiler was used to gather the performance difference between the design patterns. Using this game engine is a fast and easy way to develop games. However, it doesn't provide deeper access for better optimization. This game engine is on the other hand fairly optimized, enough to not cause any major performance discrepancies. The results produced showed that DOD was able to produce higher FPS. In addition, the CPU used fewer resources. Another important aspect that the paper discusses is the complexity of writing DOD code. It was briefly discussed that DOD was deemed more appropriate for advanced programmers.

Mironov et al. [12] present a data-oriented solution to accelerate the backtesting of trading strategies. This paper tried to implement DOD over historical trading events with the aim to achieve better performance outcomes. As a result, the DOD implementation was able to achieve a 33% speed-up compared to the OOD implementation.

Romeo and Fiumara [13] created their own ECS library and discussed the different aspects of OOD and DOD. These aspects included inheritance, composition, architecture, multithreading and more. This related work employs the same DOD pattern, ECS, as this thesis does. As for benchmarking the performance, a particle system was developed with the aim to analyze the underlying system. Lastly, the results, inner parallelism and no parallelism were compared to each other. Different parameters such as fixed or dynamic entity storage and entity scaling were also compared. For each test, the version with inner parallelism is shown as the faster implementation. In conclusion, around 65% performance boost was achieved with parallelism. Compared to this thesis, no concrete testing was done with either the cache or comparison between parallel OOD in mind.

Nyberg [14] investigated the data-oriented pattern and different design principles were analyzed with DOD. This related work is similar to this thesis with the main focus being on the CPU cache. The tests focused on comparing the performance differences between these principles against each other. Some of these include removing

run-time polymorphism and going from non-contiguous to contiguous allocation. As for the results, several different parts of the CPU cache were investigated. As for these tests, L1, L2 and L3 were included with L1 having both instructions and data misses measured.

Fontana et al. [15] investigated the two design patterns' impact on the cache, which is highly similar to this thesis work. Furthermore, it includes parallelization in its testings to investigate the impact of multithreading. This work has its focus area around Electronic Design Automation tools, and therefore the results can't be regarded as representative of a game application. The heavy focus was put on how the cache was wasted when OOD was used. It mentions organizing and iterating over consecutively accessed memory would potentially increase performance. Other aspects such as memory hierarchy were explained as well. The results showed that DOD was faster than OOD where the cache misses were fewer and the execution time was faster.

3.1 Our Work

This section will present what is special and unique about this thesis, and the kind of simple games that are going to be developed. Similar to some of the related work, this thesis is also going to investigate the performance difference of the CPU between the two design patterns.

3.1.1 Unique work

Only one of the related works included both parallel and non-parallel versions of DOD and OOD. These results were not representative of a game application, which will be further investigated in this thesis. This means exploring the effect of using several CPU threads and the compatibility of parallelization of each design pattern. Each pattern differs in how they access data. DOD in general should be able to be more consistent with its data access pattern. This thesis aims to further investigate the performance impact on these design patterns when both singlethreading and multithreading are utilized. It also aims to measure the difference in the data access pattern between the two. Generally, some levels of cache are shared between the CPU cores. One core fetching data could potentially catch a block of data that is needed by the next core. This in turn mitigates the overhead of using the memory speed which is slower than processor speeds [3].

With the games growing bigger, expecting customers to buy more expensive equipment is not a reliable business solution. Instead, packing the data to account for better CPU utilization, can achieve a larger customer base with lower-tiered equipment included. This thesis wants to provide information about how sacrificing a bit of simplicity can be beneficial for efficiency.

In summary, this thesis aims to fill the gap by comparing the CPU performance difference between both singlethreaded and multithreaded DOD and OOD. It also aims to discover how efficient the cache is when multiple CPU threads are fetching data within the scope of a small game. Further, it aims to compare the data access pattern of the two design patterns.

3.1.2 The game idea

The idea was to develop a simple game that stressed the CPU by having as many game objects as possible in the scene. It was inspired by the online web browser game called Slither.io, where a snake has to eat food in order to grow larger and defeat other players.

In this implementation, the player is a sphere that collects food in order to win. Good food adds to the score, while bad food should be avoided as it does the opposite. The size of the player depends on the score which makes it challenging. The game is over when the player reaches the maximum score. Some of the food, move in a random direction to make the game difficult. The collision was only between the player and the food for simplicity and not between every moving game object.

3.1.3 Visualisation of the games

The blue circle in figure 3.1 is the player that can move around, and the camera will follow. Green circles are good food and red circles are bad food. Good food increases the player's score, while bad food decreases it on collision. Bad food is moving in a set direction and good food is static. Player size is based on the number of points collected. The score is written in the top left corner of the window.

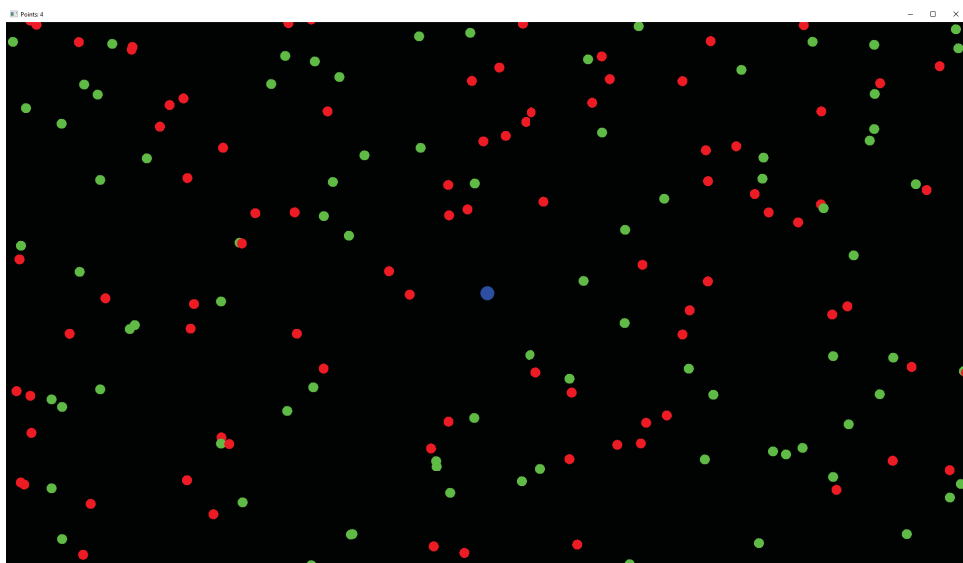


Figure 3.1: A final look at game 1.

The player and movement in figure 3.2 is the same in this version of the game. One of the differences is that all food is static and does not move. Instead, there are pink circles that are enemy players that can move. Food increases and decreases the score based on their size and color while the enemies reset the score down to zero on collision. The game is over when the player reaches a set amount of points.

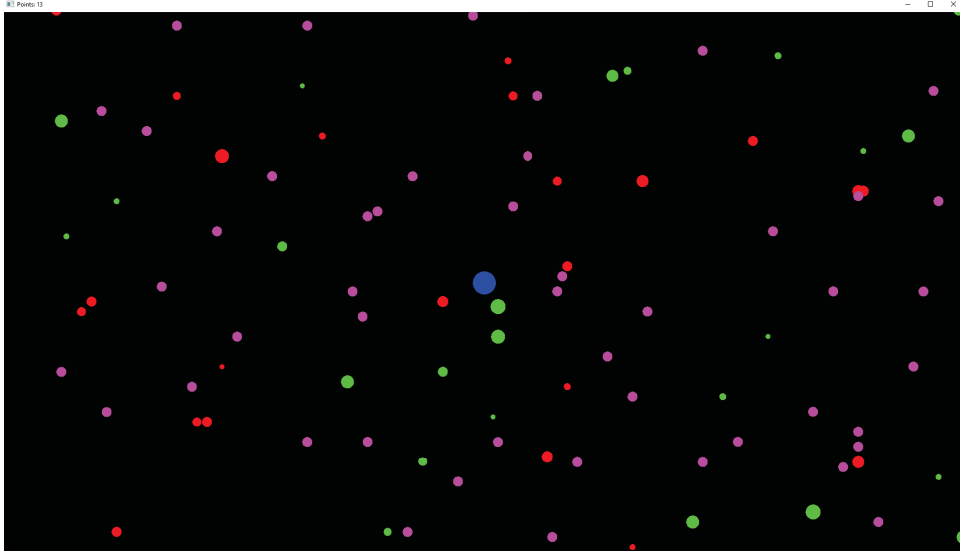


Figure 3.2: A final look at game 2.

To compare the two design patterns, two research questions were set up to investigate the primary performance factor (shown in section 1.2). The CPU is one of the most important factors that decide what speed and efficiency an application is able to maintain. To achieve the goals of this thesis, two similar games were developed and the design patterns were compared to each other. These games had similar game logic, code structure, and used both of the design patterns.

To answer this thesis' research questions, the experimental research method was chosen as the most appropriate method. This is since the thesis is experimental in nature. To get reliable results, setting up tests and measuring results is the most logical method of proceeding.

This research method does however present problems with the results collected. The problem of most significance is the bias from our own coding experience. These results will reflect how competently the code was written and how optimal the testing environment is set up.

Previously, none of the authors had any experience in making an ECS from scratch. But from following tutorials and researching the subject, more data was able to be gathered. In the end, an optimal performing ECS was set up. This was implemented with the experiments in mind and some usual features were not implemented. These features were, deleting components, and verifying if an entity had a component. This was since no component was deleted, and each entity always had each component in the game.

The experiments conducted were done in different testing environments such as Linux, Windows and Linux on a VM. Each test consisted of measuring DOD, OOD and multithreaded versions of these and then comparing them to each other with varying amounts of game objects.

For measuring how the cache was utilized and the number of cache misses, Valgrind¹ was used. Valgrind is a profiling tool in Linux that runs separately from the application and measures different aspects depending on the input. As for these tests, the 'cachegrind' tool was used. This tool displays statistics over cache hits and misses for data, as well as for instructions. It shows results over two levels of cache, the first level, and the last-level cache, which in this case was L3.

As for the multithreading, for the purpose of having a simple implementation, OpenMP² was used for threading. OpenMP is a high-level language that offers simple instructions to parallelize sections of code.

¹Available at: <https://valgrind.org>

²Available at: <https://www.openmp.org>

SFML's³ rendering API was used to create a window and draw all the game objects. It was primarily chosen for its simplicity and multi-platform compatibility. Using SFML affected the final results, this is further explained in the results chapter and the discussion chapters.

4.1 Systems Under Test

Multiple computers were used to conduct these experiments. Information about these System(s) Under Test (SUT) is listed in table 4.2. The left half of the table consists of computers used in the first experiment while the right half was for the second experiment. It is shown that experiment 1 used Windows and experiment 2 used Linux as the OS in the table.

SUT	<u>Experiment 1</u>			<u>Experiment 2</u>		
	1	2	3	4	5	6
CPU	i5-7600k	i7-8750h	i9-9900k	i5-7600k	i7-8750h	i9-9900k
Speed (GHz)	4,17	4,0	4,5	4,17	4,0	4,5
Cores	4	6	8	4	6	8
Threads	4	12	16	4	12	16
L1 size (KB)	256	384	512	256	384	512
L2 size (MB)	1	1,5	2	1	1,5	2
L3 size (MB)	6	9	16	6	9	16
RAM size (GB)	15,9	16	16	8,19	16	8
RAM speed (MHz)	2133	2667	3200	2133	2667	3200
OS	Windows 10 21H2	Windows 10 21H2	Windows 10 21H2	Linux Ubuntu 20.04.4	Linux Ubuntu 20.04.4	Linux Debian 11.3
Is a VM?	No	No	No	Yes	No	Yes

Table 4.2: Information about the systems used for the experiments.

³Available at: <https://www.sfml-dev.org>

4.2 Experiments

As mentioned above, two experiments were conducted to get the desired results. By conducting them with multiple computers, more reliable and widely applicable results were achieved. These experiments include both natively running them on the computer and also running them on VMs.

The first experiment measured how fast each design pattern could run on the CPU, while the second focused on the cache utilization for each pattern. Both of the experiments started with a baseline of game objects, which were gradually increased to test the limits of the implementations. Each test in every experiment ran for approximately 30 seconds to get a fair average value of the collected data. After running the experiments, the collected data were compared to each other.

The tests were divided since some problems were presented during the testing phase. Measuring the cache misses on Windows was difficult as there was no easy-to-use tool for it. Linux on the other hand was very simple and therefore chosen. As for the execution time, Windows got better values compared to Linux for both of the design patterns. This was clear when SUT 2 and SUT 5 were tested as they had the exact same specifications but with different OS.

4.2.1 First experiment

This part measured the speed each design pattern could manage to maintain. To get more representative results, these tests were conducted in a Windows environment. This is since both VMs can't achieve the same processing power as running the program natively, and it was necessary to use the same OS for this experiment.

Execution time was used to measure the overall efficiency of each design pattern. To get reliable results, the time it took to update a frame in the game was collected and stored within a fixed array. This array was later summed up to get the average of each collected frame.

4.2.2 Second experiment

As for the second part, this experiment focused on the cache misses for each design pattern. This was important to measure since it displays the underlying utilization of the CPU cache. Displaying higher amounts of cache misses is generally detrimental to the speed of the application. The other interesting part was how multithreading would affect cache usage when work is divided into multiple threads. As for the CPU utilization per thread, it was verified that each thread had a balanced workload.

4.2.3 Validating the tests

To get fair and average results, each of the tests in the two experiments ran for 30 seconds. During those test runs, the execution time for each frame was saved onto an array. This array could hold up to 100.000 frames at a maximum. These saved execution times were then averaged out for each run. This was done to remove as many outside factors as possible.

Some exceptions were made to the validation of the first experiment. Most of the tests only ran once due to time limitations. After the first run, the values were analyzed and tests with more than a 2% difference compared to the neighbouring tests needed further evaluations. But tests with only one run still had representative values as they could have had up to 100.000 samples.

The plan was to have both execution time and cache misses measure up to 100.000 game objects. However, this was not possible for the second experiment. The compilation time for cache misses took an exceedingly long time to perform. At around 50.000 game objects, it took around 20 minutes to compile the code. It was therefore decided to be the upper limit.

4.3 Implementation

Our own ECS was developed for the first game for maximum performance. An ECS needs to be able to gather all components for easy access. These components need to have a close locality, which requires an array within the code. This can be used with the standard library that C++ offers. For dynamic allocation, a vector is optimal. As for the implementation, the static allocation was chosen, and thus, a normal array was used.

As for looping through each component, checking if the entity possesses such a component is crucial to avoid unnecessary computation. This was not kept in mind since each entity would be guaranteed to have each component. This would in turn give the performance a slight boost since no verification was needed. As for other implementations, verification is highly recommended.

Avoiding unnecessary fetching from containers, pre-fetching and storing a reference to a component array is recommended. This eliminates the overhead of constant searching and fetching the component array from the main container. As for looping through the fetched component array, utilizing templates will maintain code simplicity.

When introducing multithreading to the system, either automatically detect parallel-able sections or give developers the choice to use multithreading with specific functions. Each has its pros and cons, the first game gives the choice of looping single or multithreaded with separate functions. This was with the testing as the main factor in mind.

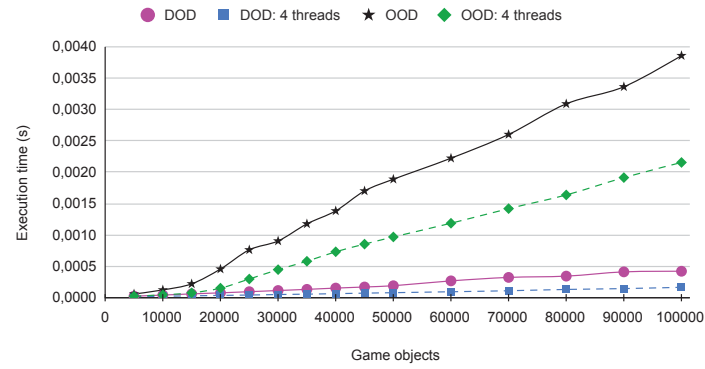
For the implementation of the first game, OpenMP was used to parallelize the for-loops within the code. This was used entirely for its simplicity with implementation. Since OpenMP is a higher levelled threading library, optimization between the threads was not investigated further.

In this chapter, the results will be displayed. Each section will be displaying the different performance metrics for each game created for this thesis. Grouped together are the corresponding DOD and OOD versions of the same game. The chapter will start off by presenting the experiments, then, later on, compare the design patterns' values and finally reveal a special case.

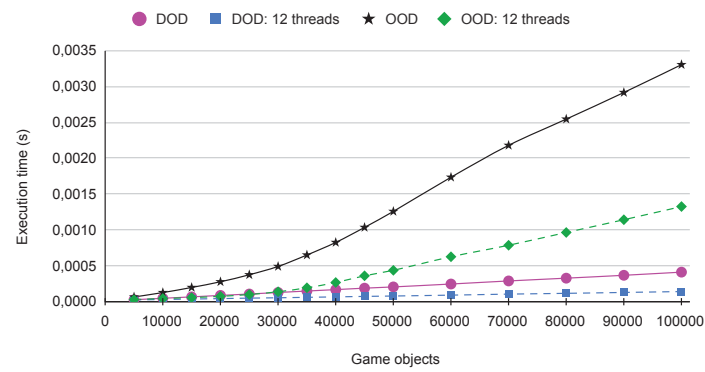
5.1 Execution time

This section will display the results of the first experiment where execution time was measured for both of the games. Execution time is the time it took for the CPU to produce one frame. Lower execution times are better and produce more FPS. In figure 5.1 dashed lines indicates multithreaded implementations.

Game 1 - SUT 1



Game 1 - SUT 2



Game 1 - SUT 3

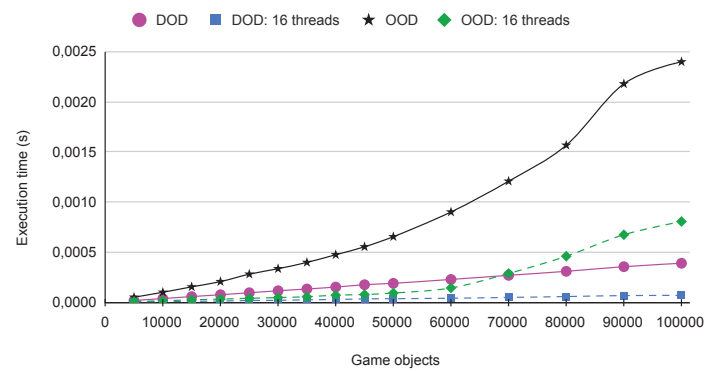
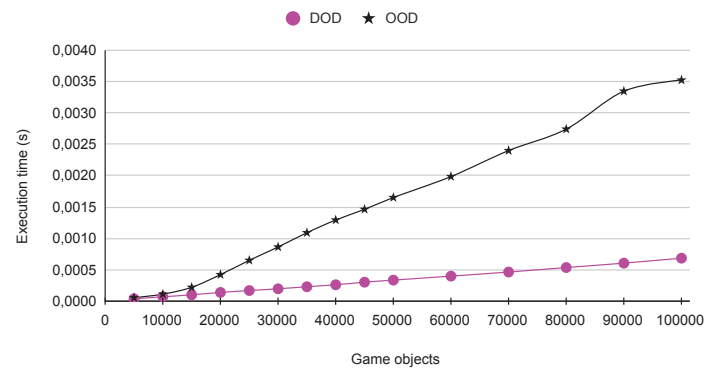
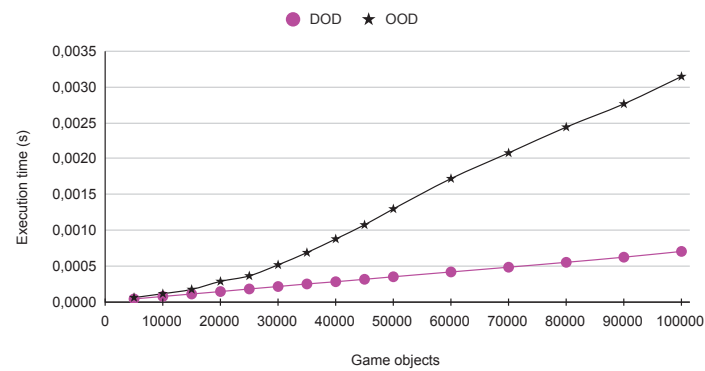


Figure 5.1: Execution time for SUT 1-3 in the first game.

Game 2 - SUT 1



Game 2 - SUT 2



Game 2 - SUT 3

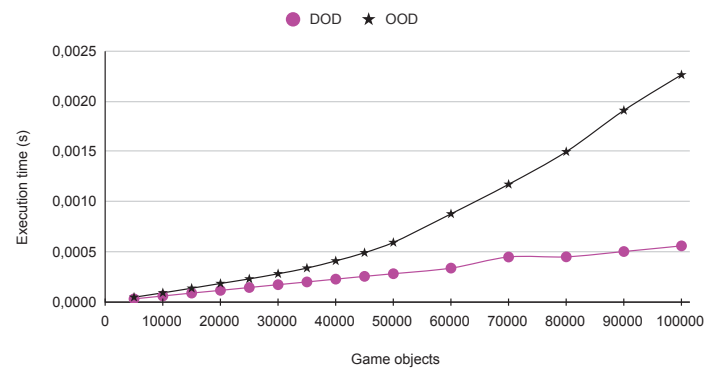


Figure 5.2: Execution time for SUT 1-3 in the second game.

5.2 L1 cache misses

This section will present the data cache misses from the first level for the two games. The figures show how many percent of the time the CPU can't find the data in the L1 cache and need to fetch the data from one of the slower levels in the memory hierarchy. Low values are better and mean less access to these layers.



Figure 5.3: L1 cache misses for SUT 4-6 in the first game.

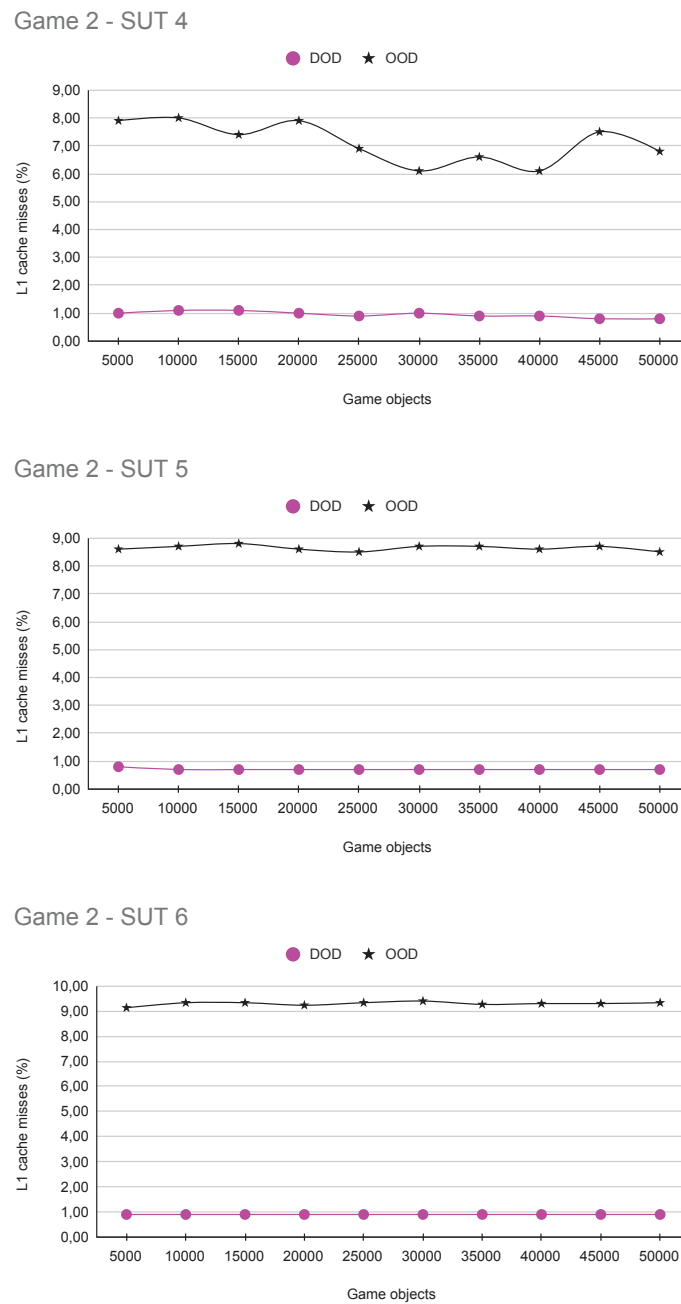


Figure 5.4: L1 cache misses for SUT 4-6 in the second game.

5.3 Last-level cache misses

This section will present the data cache misses in the last level for the two games. The figures show how many percent of the time the CPU can't find the data in the CPU cache and need to get the data from RAM. Low values are better and mean that less access to RAM is needed.

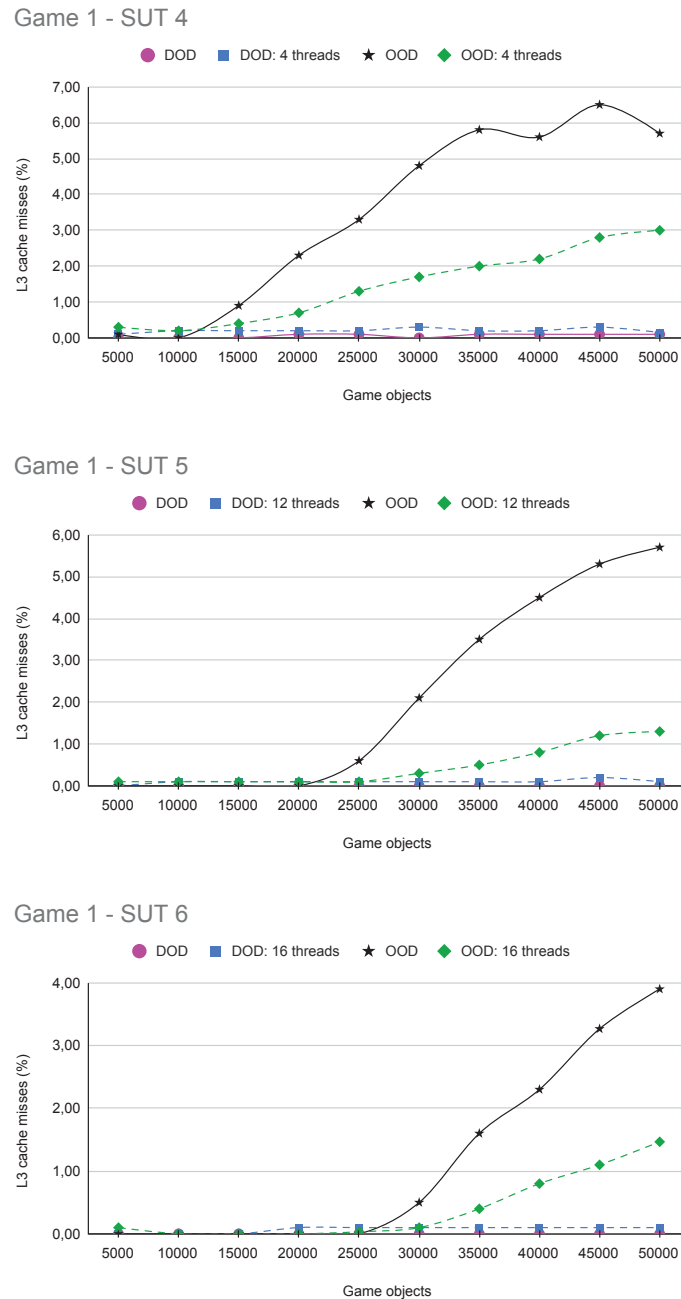


Figure 5.5: Last-level cache misses for SUT 4-6 in the first game.

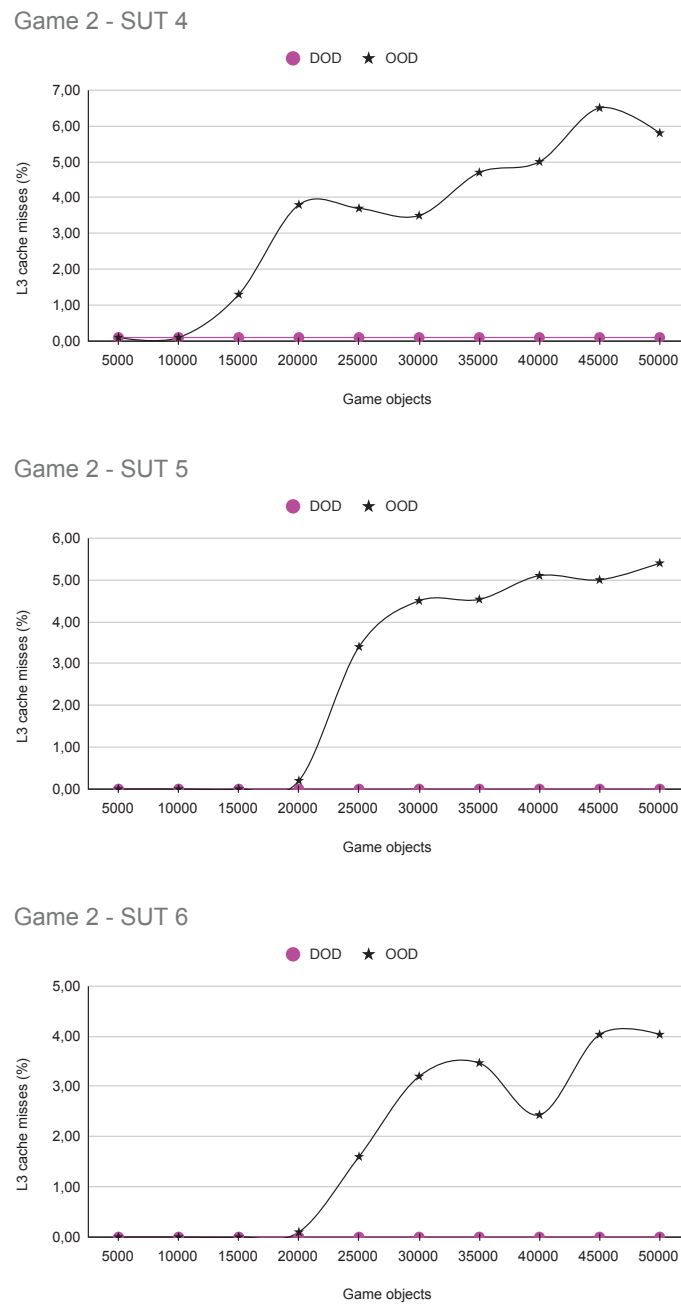


Figure 5.6: Last-level cache misses for SUT 4-6 in the second game.

5.4 Comparing the design patterns

In this section, the two design patterns are compared against each other and visualized by tables. Aspects such as cache misses, multithreading and execution time are compared and presented. The numbers in the tables represent the speed-up of the comparable methods.

5.4.1 Execution time

Each cell in table 5.2 tells the speed-up in execution time the DOD implementation got against the OOD implementation. Both singlethreading and multithreading are compared between the two design patterns in this table.

	SUT 1		SUT 2		SUT 3	
Game objects	1 thread	4 threads	1 thread	12 threads	1 thread	16 threads
10 000	3,0	2,0	2,7	1,5	2,5	1,7
20 000	5,9	4,3	3,2	1,7	2,7	2,2
30 000	7,9	8,9	3,8	2,6	2,9	2,2
40 000	9,1	11,5	4,9	4,0	3,1	2,4
50 000	9,9	12,2	6,1	5,6	3,4	2,4
60 000	8,3	12,6	7,1	6,9	3,9	3,3
70 000	8,1	12,9	7,5	7,5	4,4	5,6
80 000	9,0	12,5	7,8	8,3	5,0	7,6
90 000	8,2	13,2	7,9	8,9	6,1	9,6
100 000	9,2	13,0	8,0	9,4	6,1	11,2

Table 5.2: Improvement in execution time by using DOD over OOD in game 1.

5.4.2 Multithreading improvement

This section presents how the multithreading improved the execution time for the two design methods. Higher values are better.

	SUT 1		SUT 2		SUT 3	
Game objects	DOD	OOD	DOD	OOD	DOD	OOD
10 000	1,8	2,8	1,6	2,9	3,7	5,5
20 000	2,2	3,0	2,0	3,7	4,8	6,0
30 000	2,3	2,0	2,4	3,6	5,1	6,7
40 000	2,4	1,9	2,5	3,1	5,0	6,4
50 000	2,4	2,0	2,6	2,9	4,9	6,9
60 000	2,8	1,9	2,7	2,8	5,3	6,2
70 000	2,9	1,8	2,7	2,8	5,3	4,2
80 000	2,6	1,9	2,8	2,6	5,1	3,4
90 000	2,8	1,8	2,9	2,6	5,1	3,2
100 000	2,5	1,8	2,9	2,5	5,4	3,0

Table 5.4: Improvement in execution time by using multithreading in game 1.

5.4.3 Cache utilization

In this section, the results of how much of an improvement DOD gave over OOD in cache misses can be seen. Table 5.6 describes the first level in the cache, and table 5.8 shows the last-level cache. Some cells in the second table use infinity signs and this stands for an improvement by infinity. This is because of the zero percent misses that DOD returned for the last-level cache. More about this is explained in the discussion chapter.

	SUT 4		SUT 5		SUT 6	
Game objects	1 thread	4 threads	1 thread	12 threads	1 thread	16 threads
5 000	1,9	1,9	1,9	3,0	2,0	3,8
10 000	2,0	2,1	1,9	2,0	2,0	3,6
15 000	1,9	2,2	2,0	2,2	1,9	4,6
20 000	1,9	2,1	1,9	2,3	2,0	4,7
25 000	2,0	2,5	1,9	3,0	2,0	4,5
30 000	1,8	2,5	2,0	3,2	2,0	4,6
35 000	2,0	2,5	1,9	3,5	2,1	4,5
40 000	1,8	2,4	1,8	3,8	2,0	4,5
45 000	2,0	2,5	1,9	3,6	2,0	4,3
50 000	1,7	2,3	1,8	4,0	2,0	4,2

Table 5.6: First level cache improvement between DOD and OOD in game 1.

Game objects	SUT 4		SUT 5		SUT 6	
	1 thread	4 threads	1 thread	12 threads	1 thread	16 threads
5 000	∞	3,0	1,0	∞	1,0	∞
10 000	1,0	1,0	1,0	1,0	1,0	1,0
15 000	∞	2,0	1,0	1,0	1,0	1,0
20 000	23,0	3,5	1,0	1,0	1,0	0,0
25 000	33,0	6,5	∞	1,0	1,0	0,3
30 000	∞	5,7	∞	3,0	∞	1,0
35 000	58,0	10,0	∞	5,0	∞	4,0
40 000	56,0	11,0	∞	8,0	∞	8,0
45 000	65,0	9,3	∞	6,0	∞	11,0
50 000	57,0	20,0	∞	13,0	∞	14,7

Table 5.8: Last-level cache improvement between DOD and OOD in game 1.

5.5 Special case execution time comparison

As seen from the tests, DOD has a massive advantage over OOD. This is however with OOD having huge data put into smaller objects. These tests focused on their own specialized area. DOD did not have these objects included in the measurements. This in turn affects the performance of OOD massively. To remain transparent with the results, another test was added that had DOD also include these massive objects within its measurement. The results in figure 5.7 will give an overview of how the performance differentiates over the entire program with this special case.

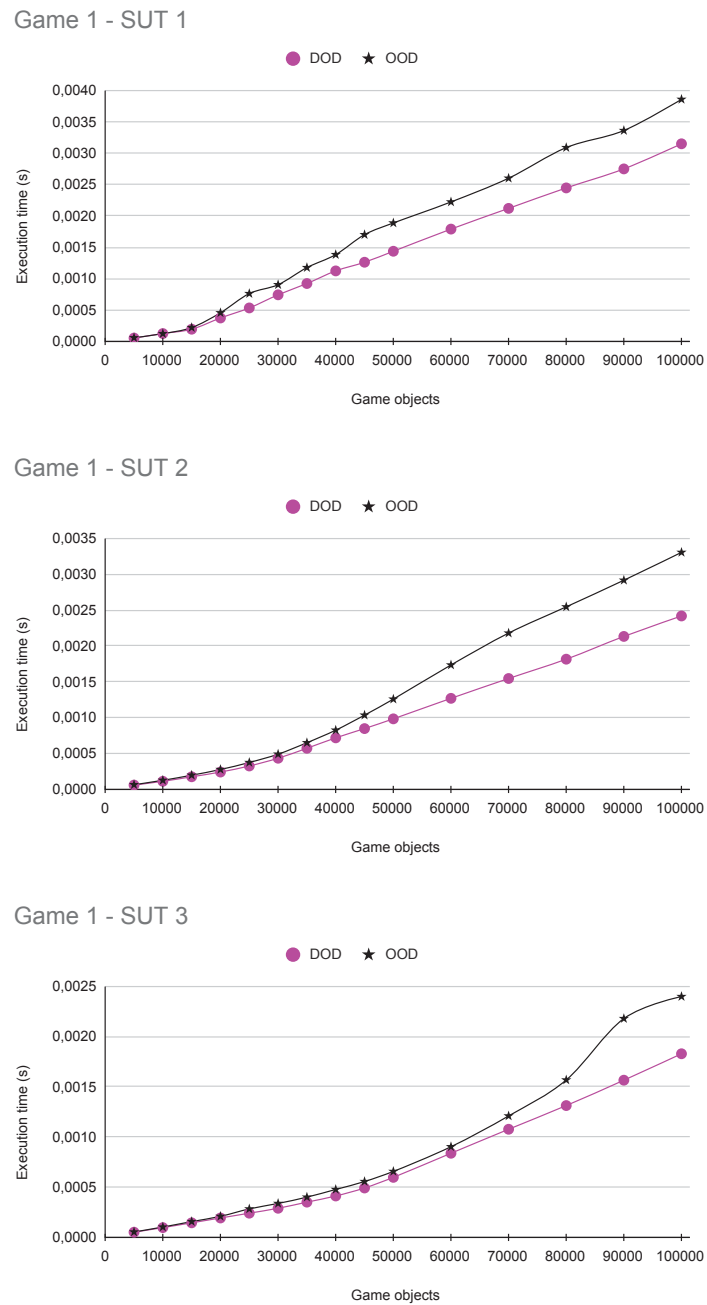


Figure 5.7: Execution time for SUT 1-3 in the first game when DOD is utilizing larger objects.

In this chapter, the results obtained, and the implementation will be discussed and analyzed further.

6.1 Implementation biases

The different implementations were created based on our view of standard implementations of the two design patterns. Since SFML rendering API was used, some objects necessary for rendering were out of control. Since OOD had these objects included within their own objects, they affected the cache usage by a large margin. DOD did not include these within the measured function to maintain its purpose. This in effect boosted the performance of certain sections of the code of the application for DOD compared to OOD. It did however not have the same effect on the entire application.

6.2 Implications of the results

As the results have shown and mentioned before, DOD had the advantage over OOD. Each case showed that DOD had lower cache misses and lower average execution time.

The multithreading proved to be beneficial for both the execution time and the cache hits. Using multiple threads gave fewer cache misses which in turn further increased the performance of the OOD section. If the application utilizes several cores, using DOD is the preferred design pattern. This is since it is the more adaptable choice of the two design patterns. Most tests showed that singlethreaded DOD was faster than multithreaded OOD. This in practice means, that neatly packed data is exceedingly faster than unorganized data. From these tests, it is clear that utilizing the CPU more than the RAM, is beneficial for performance.

OOD cache misses for L3 increased with the number of objects. For the other pattern DOD, no major increase was registered during the tests. This would most likely be because of the games being small enough to not require much cache space for DOD.

An important point to bring up is how Valgrind handles presenting the results. Throughout the tests, Valgrind appeared to skip any cache miss percentages below 0.1%. This means that any 0.0% displayed in the results still got cache misses but Valgrind deemed them insignificant.

In summary of the results, does this mean that a program can achieve 13 times the speed-up by using DOD? To answer that, while possible, it most likely will not happen. A regular game often does not include over 5.000 game objects. Reading these results, it can be summarized as: It is possible to double the performance, at most. But other factors such as how the data is accessed, organized and the volume of different data affect the performance. These experiments were made of simple games that had only a little functionality. Larger games might see a lesser performance boost compared to these tests.

6.3 Analysis of special case

The first results displayed were results from sections of the update function where DOD and OOD were utilized to their fullest extent. In practice, this means OOD were packed objects with a large amount of data, with DOD being smaller compact data structures. As for these tests, SFML rendering API was used and as such, the rendering objects within OOD were massive compared to DOD. DOD did also contain these objects but they were not accessed within the measured function. This was to maintain the principle of DOD and not include unorganised objects.

However, to remain transparent, a special case comparison was made in section 5.5. As displayed in those results, the performance difference was not the same as in the first results. This special case would be more representative of the entirety of the application. Even though it didn't reach the levels of 13 times the speed-up, it doesn't undermine the results obtained from the first experiment.

As displayed in the results, even though DOD handled unorganized data, it achieved a lower average execution time than OOD. This shows that DOD is exceptional to use even if large data is put into these structures.

6.4 Cache misses and their relation to execution time

Comparing the execution time between the design patterns, DOD clearly has the advantage. Using the EnTT library showed to be most efficient for both the first and last-level cache (LLC). Further investigating the cache misses, the LLC was constantly increasing for OOD.

Comparing the data misses with the execution time, it is clear that some kind of connection lies in between them. OOD that had higher cache miss rates, with an increasing number for LLC, and had higher average execution time. Within the implementation, OOD only consisted of one loop that iterated through each object, DOD had two loops. Each loop had to iterate through each game object, assuming that each loop would take a similar execution time, OOD still had a higher execution time which does not make much sense.

This is where data locality and utilizing the cache efficiently comes into effect. DOD had a higher chance of consecutively accessing data that could be operated on immediately from the cache, which excessively increased the performance the application was able to maintain.

These tests show that having smaller compact structures instead of large easy to read objects, had a tremendous impact on the cache and the speed of the application.

Keeping this in mind, larger games could certainly make use of the DOD pattern to increase their efficiency immensely in certain sections of code.

6.5 The access pattern of the two design patterns

Comparing the two design patterns, it is clear that there is a difference. While DOD usually stayed at a levelled pace, OOD often considerably increased its cache misses. This is entirely based on how the OS decide to place the data in memory. The developer often has low control over how the OS organize the data. This control can be improved by knowingly using items such as arrays. Arrays get their data initialized consecutively, which in turn gives a higher chance of close data locality.

Reading the results, it is clear to see that organizing the data gives fewer cache misses. Which in turn allows the application to run at a faster pace.

Noticing the multithreaded versions, there exists a difference compared to the singlethreaded versions. OOD on the first level of cache showed to be slightly irregular while the last level proved to be highly ineffective. It is the last level of cache that is the most detrimental to the performance. A cache miss on this level means that fetching from the RAM is necessary which is slow. The multithreading for OOD improved the cache miss rate for the last level of cache but proved to be more irregular at the first level of cache.

6.6 The effect of multithreading

The increase in speed was different depending on the number of game objects and the design pattern used. Generally, DOD gave a better boost when more entities were used. However, OOD had a higher boost for lower amounts of objects for each SUT. This however doesn't amount to much since DOD is already capable of running at twice the speed of OOD. These discrepancies could be the fault of either memory speeds, poor implementation or something else.

Looking at the results, DOD starts with a lower speed-up and steadily gets a higher speed-up with more entities added. As for the other design pattern, the speed-up steadily lowers. This means that DOD is better scalable with more entities added to the game.

Another interesting observation to make is the effect on the cache misses for multithreading. OOD gains fewer cache misses with multithreading added. This becomes more noticeable with more objects added. This was not further investigated in detail, but speculations can be provided. These might be the positives of using OOD. By having unorganized objects, the cache lines might pick up the necessary data by accident to help other cores.

6.7 The complexity of both design patterns

In this section, the experience working with DOD and OOD will be discussed for both of the games.

Game 1

In the beginning, an ECS tutorial was followed that allegedly had a low cache miss rate. After setting it up, it was clear that this implementation had a critical fault. It required a search for each component that it wanted to access. This presents the overhead of each search needed for each component which was clear in the debugging. With ideas inspired by the previous implementation, a new implementation was created. One single class called 'ECS' was made that took care of each component array and their initialization and more. To mitigate this constant search, a reference was stored within the calling function. With this reference, direct access was able to be achieved and in turn completely remove the need for a constant search for each component. In practice, this means going from $O(\log N)$ to $O(1)$ which is a major improvement.

Implementing this DOD certainly took time and it would be difficult to manage for a beginner programmer. However, an average programmer should be able to set this up without major problems.

As for the OOD, it was very simple to implement. It did however require several header files to set up. There is a clear trade-off between the two design patterns. OOD needs several header files to remain easy to navigate and DOD requires only one header file where all components are declared and defined.

The positives of using OOD are the simplicity in understandability and re-usability. This re-usability is however limited since re-using a class means re-using all of the components of the class. This does narrow it down over who and what has the need to re-use the said class. In this implementation, everything was simple enough to not have to worry about this.

OOD gave a more separate code in the form of functions. DOD did not have functions but could have been easily implemented. It is clear from comparing the two implementations that OOD is more easily understood than DOD. Separating the code into functions has its advantages, but DOD also has its advantages, it all depends on how well the implementation is done.

Game 2

The implementation of OOD was fairly easy to write with all the abstractions that the design pattern offered. Figuring out how the game would be played was the part that took most of the time. Implementing OOD required several files for every type of game object as most of them needed individual functions. Functionality such as inheritance was used to avoid re-writing code for the game objects. Hiding functions within objects were also utilized and made the code easy to read.

The DOD version of the game was easy to implement with the help of EnTT. Less time was spent on coding the game logic and some code could be re-used from OOD with minor adjustments. This implementation did not require as many files as OOD. Instead, one file with every type of component was defined with only the pure data without any functionality. For the first iteration of coding the game, OOD was

faster. This was because of the lack of knowledge about the most optimal functions to use in EnTT. The most significant changes that were made to fix this was:

- Swapping from 'view-function' to 'group-function' when multiple components were needed.
- Using the get-function from 'group' instead of from 'registry'.

These small changes made the implementation about two to three times faster than the previous version. It also made DOD faster than OOD. Using an already written ECS library helped and saved time. The implementation should be possible for a newer programmer that has understood the theory of ECS.

When comparing the readability between the two design patterns, OOD was a bit easier as some lines of code could be hidden away inside functions. On the other hand, DOD was easier to work with and was very flexible to use. If an entity needs to be changed, it is possible to just add or remove components without the need of changing much code.

Chapter 7

Conclusion and Future Work

For this thesis, two games were created using DOD and OOD with and without multithreading. The experiments compared the two design patterns' execution time and the cache misses. It also discussed the complexity of using the patterns and the effect that multithreading had. Through the tests that were conducted, results point toward a difference between the cache utilization of the two design patterns. These results also suggest a link between cache utilization and execution time. It has been clearly shown that the data access pattern is also different. OOD compared to DOD is further fluctuating in both the first and last level of cache. DOD proved to be the stable design pattern between the two. By utilizing DOD, it is most likely to increase the stability of the application, but also lower the execution time.

Further discussion has been made around the complexity of coding with the two design patterns in mind. OOD, which is the most commonly used design pattern, provides a fast and easy method of creating code. This is since it follows the pattern of being similar to the real world. However, this is not an optimal method of designing code. DOD, on the other hand, is a slightly more complex design pattern that provides a method of designing code to be friendly for the CPU.

As a short summary, utilizing the DOD is the preferred method if the application needs to run fast and be stable. This could be e.g. a larger game. In contrast to DOD, OOD provides a faster method of developing code. This design pattern would be preferable for smaller applications, but also for applications that are on a time crunch.

As for future work, it could be interesting to compare the two design patterns on a larger scale with a bigger game. This means larger game objects with more data and functionality where further representative results can be achieved.

It would also be fascinating to explore DOD with a lower level of multithreading. This could achieve even better CPU performance and more control.

Further investigating the significance of the data size for the components in DOD and the impact on multithreading could also be explored. Using larger data could possibly give more cache misses as less data can stay in the cache memory.

Finally, an interesting topic to do some further investigation on could be how faster RAM speeds can affect both OOD and DOD. The new DDR5 RAM has a higher speed and larger bandwidth, which could be interesting to measure [16]. Is the gap between CPU speed and memory still relevant or not?

References

- [1] A. Kuo, J. L. Hiler, and R. J. Lutz, “From super mario to skyrim: A framework for the evolution of video game consumption,” *Journal of Consumer Behaviour*, vol. 16, no. 2, pp. 101–120, 2017.
- [2] B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*. Springer, 2015, ch. 1,3, ISBN: 9783319242781.
- [3] C. Carvalho, “The gap between processor and memory speeds,” in *Proc. of IEEE International Conference on Control and Automation*, 2002.
- [4] E. Lengyel, *Game engine gems 2*. CRC Press, 2011, ch. 15, pp. 251–261, ISBN: 9781568814377.
- [5] B. Jacob, D. Wang, and S. Ng, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010, ch. 1.2, pp. 62–63, ISBN: 9780123797513.
- [6] J. Handy, *The cache memory book*. Morgan Kaufmann, 1998, ch. 1, pp. 5–8, ISBN: 9780123229809.
- [7] L. Null and J. Lobur, *Essentials of Computer Organization and Architecture*, 5th ed. Jones & Bartlett Publishers, 2018, ch. 6.4, ISBN: 9781284123036.
- [8] R. Fabian, “Data-oriented design,” 2013. [Online]. Available: <https://www.dataorienteddesign.com/dodmain.pdf>
- [9] N. Shavit and M. Herlihy, *The art of multiprocessor programming*. Morgan Kaufmann, 2012, ch. 1, ISBN: 9780123973375.
- [10] S. M. Thomas, *PostgreSQL High Availability Cookbook*, 2nd ed. Packt Publishing Ltd, 2017, ch. 9, ISBN: 9781787125537.
- [11] K. Fedoseev, N. Askarbekuly, E. Uzbekova, and M. Mazzara, “A case study on object-oriented and data-oriented design paradigms in game development,” 2020. [Online]. Available: <https://www.researchgate.net/publication/342735514>
- [12] T. Mironov, L. Motaylenko, D. Andreev, I. Antonov, and M. Aristov, “Comparison of object-oriented programming and data-oriented design for implementing trading strategies backtester,” in *ENVIRONMENT. TECHNOLOGIES. RESOURCES. Proceedings of the International Scientific and Practical Conference*, vol. 2, 2021, pp. 124–130. [Online]. Available: <https://doi.org/10.17770/etr2021vol2.6629>
- [13] V. Romeo and G. Fiumara, “Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time entity-component-system c++ 14 library,” Ph.D. dissertation, University of Messina, 2016. [Online]. Available: <https://www.researchgate.net/publication/305730566>

- [14] F. Nyberg, “Investigating the effect of implementing data-oriented design principles on performance and cache utilization,” 2021.
- [15] T. A. Fontana, S. Almeida, R. Netto, V. Livramento, C. Guth, L. Pilla, and J. L. Güntzel, “Exploiting cache locality to speedup register clustering,” in *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design: Chip on the Sands*, 2017, pp. 191–197.
- [16] “Kingston fury™ beast ddr5 memory - 16gb, 32gb 4800mt/s, 5200mt/s, 5600mt/s, 6000mt/s - kingston technology,” date accessed: 2022-05-09. [Online]. Available: <https://www.kingston.com/en/memory/gaming/kingston-fury-beast-ddr5-memory>

