

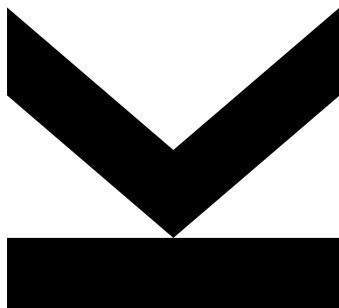
Author  
**Jakob Waldhäusl**

Submission  
**Institute for System  
Software**

Thesis Supervisor  
Dipl.-Ing. **Thomas Schrott**,  
BSc

September 2026

# **Performance Analysis of Data-oriented Programming in Rust**



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Computer Science

# Abstract

In the past decades, processor speeds have increased vastly, with memory speeds lagging behind. This causes memory layouts and access patterns to increasingly define performance instead of only the computational workload. To address this, data-oriented principles may be employed, the impact of which is seldom quantified in isolation.

This work attempts to measure said impact in Rust. For this, four architectures are implemented, namely Pointer Chasing (PC), Array of Structures (AoS), Structure of Arrays (SoA) and Bevy ECS. They all operate on an identical workload, isolating performance differences to the architecture at hand. These four architectures are benchmarked in three scenarios: Scenario 1 being compute-bound, Scenario 2 memory-bound and Scenario 3 inspecting allocation and deallocation. The results are supported by machine code inspection and profiling.

The SoA architecture outperforms the others in all three scenarios, while PC consistently places last. Bevy beats AoS for second place in Scenarios 1 and 2 but loses to it in Scenario 3. Notably, the reason for SoA's lead differs by scenario, being SIMD vectorization, cache locality and allocation strategy respectively.

Although the SoA architecture outperformed the others with regard to execution times, it suffers from other drawbacks such as potentially worse code ergonomics and increased susceptibility to human error, i.e., factors which were not investigated in this thesis. Therefore, blindly recommending the SoA architecture for every task would be short-sighted. The conclusion is that the choice of memory layout should not default to a data-oriented one, but to what best fits the task and bottleneck at hand.

# Kurzfassung

In den vergangenen Jahrzehnten sind Prozessorgeschwindigkeiten stark gestiegen, während Speichergeschwindigkeiten zurückblieben. Dadurch bestimmen zunehmend das Speicherlayout und die Zugriffsmuster die Performance und nicht mehr allein die Rechenlast. Um dem zu begegnen, können datenorientierte Prinzipien eingesetzt werden, deren Auswirkungen jedoch selten isoliert quantifiziert werden.

Diese Arbeit versucht, genannte Auswirkungen in Rust zu messen. Dazu werden vier Architekturen implementiert, nämlich Pointer Chasing (PC), Array of Structures (AoS), Structure of Arrays (SoA) und Bevy ECS. Sie alle operieren auf einer identischen Arbeitslast, sodass sich Performanceunterschiede allein auf die jeweilige Architektur zurückführen lassen. Diese vier Architekturen werden in drei Szenarien untersucht: Szenario 1 ist compute-bound, Szenario 2 memory-bound und Szenario 3 betrachtet Allokation und Deallokation. Die Ergebnisse werden durch Maschinencode-Inspektion und Profiling untermauert.

Die SoA-Architektur übertrifft die anderen in allen drei Szenarien, während PC durchgehend den letzten Platz belegt. Bevy schlägt AoS in den Szenarien 1 und 2 um den zweiten Platz, unterliegt AoS aber in Szenario 3. Bemerkenswert ist, dass der Grund für den Vorsprung von SoA je nach Szenario ein anderer ist, nämlich SIMD-Vektorisierung, Cache-Lokalität und Allokationsstrategie.

Obwohl die SoA-Architektur die anderen hinsichtlich der Ausführungszeiten übertraf, weist sie andere Nachteile auf, etwa eine potenziell schlechtere Code-Ergonomie und eine erhöhte Anfälligkeit für menschliche Fehler, also Faktoren, die in dieser Arbeit nicht untersucht wurden. Die SoA-Architektur daher blind für jede Aufgabe zu empfehlen, wäre kurzsichtig. Die Schlussfolgerung ist, dass die Wahl des Speicherlayouts nicht standardmäßig auf ein datenorientiertes fallen sollte, sondern auf jenes, das am besten zur Aufgabe und zum vorliegenden Bottleneck passt.

# Contents

|                                                       |            |
|-------------------------------------------------------|------------|
| <b>Abstract</b>                                       | <b>ii</b>  |
| <b>Kurzfassung</b>                                    | <b>iii</b> |
| <b>1 Introduction</b>                                 | <b>1</b>   |
| <b>2 Background</b>                                   | <b>3</b>   |
| 2.1 Data-oriented Programming . . . . .               | 3          |
| 2.2 CPU Caches . . . . .                              | 5          |
| 2.3 Data-Level Parallelism via SIMD . . . . .         | 5          |
| 2.4 Entity Component Systems . . . . .                | 6          |
| 2.5 Rust and Bevy engine . . . . .                    | 7          |
| <b>3 Benchmark Design and Implementation</b>          | <b>8</b>   |
| 3.1 Simulation . . . . .                              | 8          |
| 3.2 Evaluated Architectures . . . . .                 | 9          |
| 3.3 Scenario 1: Compute-Bound . . . . .               | 12         |
| 3.4 Scenario 2: Memory-Bound . . . . .                | 15         |
| 3.5 Scenario 3: Allocation and Deallocation . . . . . | 17         |
| 3.6 Execution and Configuration . . . . .             | 19         |
| <b>4 Evaluation</b>                                   | <b>21</b>  |
| 4.1 Methodology . . . . .                             | 21         |
| 4.2 Scenario 1: Compute-Bound . . . . .               | 23         |
| 4.3 Scenario 2: Memory-Bound . . . . .                | 26         |
| 4.4 Scenario 3: Allocation and Deallocation . . . . . | 31         |
| 4.5 Discussion . . . . .                              | 36         |
| 4.6 Limitations . . . . .                             | 37         |
| <b>5 Related Work</b>                                 | <b>38</b>  |
| <b>6 Future Work</b>                                  | <b>40</b>  |
| <b>7 Conclusion</b>                                   | <b>41</b>  |
| <b>Bibliography</b>                                   | <b>42</b>  |

|                         |           |
|-------------------------|-----------|
| Contents                | v         |
| <b>Listings</b>         | <b>45</b> |
| <b>List of Figures</b>  | <b>46</b> |
| <b>List of Tables</b>   | <b>47</b> |
| <b>List of Acronyms</b> | <b>48</b> |

# Chapter 1

## Introduction

**Motivation** In the past decades, processor speeds have vastly increased due to increased clock frequency, higher core counts, and higher instruction-level parallelism, i.e., more instructions executed per cycle, while DRAM has not been able to keep up [13]. This phenomenon is colloquially known as the “memory wall”. The resulting problem is that execution speed may no longer be limited by processing power, but by the processor having to wait for data from DRAM [8]. To address this gap, modern software development has to take the underlying hardware constraints into consideration.

**The Shift to DOP** *Object-oriented Programming (OOP)* has enjoyed wide adoption and is usually taught long before *Data-oriented Programming (DOP)* in most curricula. However, OOP poses challenges when it comes to cache-optimization. This is, for example, due to objects bundling data and logic, which leads to inefficient memory layouts and cache pollution [8]. More modern approaches, such as DOP, aim to mitigate these challenges (among others) by decoupling logic and data. However, there are more mechanisms that DOP employs to improve cache efficiency and execution time. Some of the more significant ones are *Single Instruction, Multiple Data (SIMD)* vectorization (Section 2.3) and use of the *Entity Component System (ECS)* software architecture design pattern (Section 2.4).

**Objectives and Methodology** The goal is to measure the difference in execution time between OOP and DOP approaches, as it is hardly ever quantified. The benchmarks lean towards the gaming industry, where there has been a significant push to adopt the ECS pattern. Four architectures are implemented, spanning a spectrum from maximum pointer indirection to maximum contiguity. They are ordered so that each architecture differs from the preceding one in primarily one property, isolating measured differences to it rather than to a combination of several.

These architectures are *Pointer Chasing (PC)*, *Array of Structures (AoS)*, *Structure of Arrays (SoA)* and Bevy ECS, where the former two represent OOP and the latter two DOP design. These architectures are measured in three scenarios, namely a compute-bound scenario, a memory-bound scenario, and one concerning allocation and deallocation. Therefore, each scenario isolates a different bottleneck. Because all architectures operate on an identical simulation, any measured difference can be attributed to the memory layout rather than to the workload. The results are further backed by machine-code inspection and profiling.

Rust acts as the programming language with the Bevy engine providing the ECS framework for the Bevy architecture. Rust provides memory safety without requiring a garbage collector [18]. The absence of garbage collection is especially desirable for benchmarking, as it eliminates non-deterministic collection pauses that would otherwise distort time measurements. Bevy builds directly on these properties. According to its documentation, Bevy ECS represents components as plain Rust structs and systems as ordinary functions, which the project contrasts with other Rust ECS implementations that often rely on more complex constructs such as lifetimes or builder patterns [4]. This keeps the framework-based implementation close in form to the hand-written architectures it is compared against, which is desirable when the comparison is meant to isolate memory layout rather than API design.

# Chapter 2

## Background

This section describes the relevant parts of *Data-oriented Programming* (DOP), which also include *Single Instruction, Multiple Data* (SIMD) vectorization and the *Entity Component System* (ECS) design pattern, as they are integral to the nature of the systems under inspection. Furthermore, a brief overview of Rust and the Bevy engine is presented.

### 2.1 Data-oriented Programming

The *Object-oriented Programming* (OOP) paradigm undoubtedly has many useful features such as encapsulation, polymorphism and inheritance. These features enable software developers to think of problems in terms of objects and their interactions, with objects ideally being self-contained. Due to this, programmers are able to easily reuse and maintain them without having to fear side effects [2]. Nevertheless, objects encapsulating the entirety of their data and logic is a double-edged sword.

For example, consider a program that solely updates the position field of a number of objects, where each object also holds further attributes, e.g., velocity or mass. Due to encapsulation, the CPU may have to load all the object's data into its cache, even though it should only have to load position for its computation. This phenomenon is often called “cache pollution” [8] which poses a problem due to another phenomenon called the “memory wall”, which describes the growing gap between processor and memory speeds. As processor speed increases faster than memory speed, an increasing number of programs becomes limited not by clock frequency but by the time spent waiting for data from memory. Over the past decades, processor performance has improved far more quickly than memory speed. Wulf and McKee [27] illustrated this disparity in 1995, reporting an annual increase of roughly 80 % in processor speed against

only 7 % for memory. While the exact rates have changed since, the underlying trend has persisted, and the resulting gap has become one of the biggest bottlenecks in high-performance computing, as the CPU often stays idle while waiting for data to be fetched from main memory [13].

One of the most promising solutions to these problems is DOP, as it aims to decouple data from logic and thus reduce cache pollution and the CPU idling [9]. DOP improves CPU utilization by fundamentally changing how data is handled by moving away from the classical OOP memory layout called *Array of Structures* (AoS) and instead using a more optimized *Structure of Arrays* (SoA) layout [9].

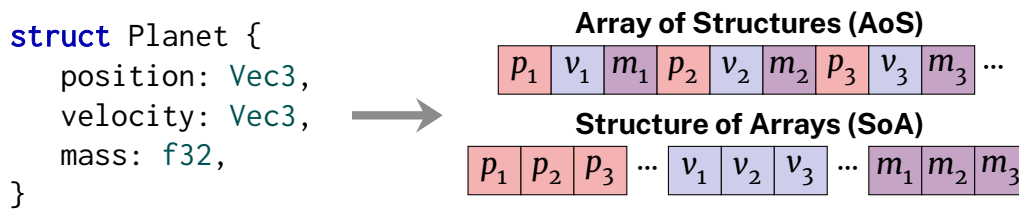


Figure 2.1: Conceptual Planet struct (left) and the corresponding physical memory layouts (right)

In Figure 2.1, both memory layouts are compared. On the conceptual level, there are simply structs with data, e.g., objects with fields. The more conventional AoS layout stores these structs sequentially within an array, thus the name. In contrast, the SoA layout creates multiple arrays which contain only one type of field. As an example, one could manage 1000 Planet objects that possess the fields position, velocity (each a Vec3) and mass (an f32). The AoS layout stores these as a single array of 1000 Planet entries, with each planet's position, velocity and mass kept adjacently in memory. The SoA layout instead uses one array per field: one holding the position of every planet, one for velocity and one for mass. This Planet struct loosely represents the structs used in the benchmarks. However, the mass field is replaced by a variably sized payload (see Section 3.1).

The reason this layout matters lies in a principle called spatial locality. Drepper argues [8] that since neighboring data is likely to be used together, it should be loaded into the cache together. This is what the entire idea of SoA is based upon. If one wants to perform a read or write operation on only the position and velocity of these 1000 planets, the CPU can load much more relevant data than with the AoS layout, as the cache is not clogged by the irrelevant mass data. Thus, while the SoA layout might appear confusing to programmers used to the AoS layout, it is necessary in order to properly achieve the performance gains granted by SIMD vectorization.

## 2.2 CPU Caches

As has been established, processor speeds have outgrown DRAM speeds. To reduce the time the CPU spends waiting on data from DRAM, CPU caches are used. The goal of a CPU cache is to hold data that has been recently used and is likely to be used again in the near future [15]. A typical CPU cache hierarchy is separated into three layers, called L1, L2 and L3, with the L1 cache being further split into a data cache and an instruction cache. From L1 to L3, each cache layer becomes slower but grows bigger in size [8].

A cache does not transfer data in single bytes but in fixed-size blocks called cache lines, which are usually 64 B in size [8]. Because a cache line is the smallest unit that can be transferred, no smaller piece of data, e.g., only the 24 B of a planet's position and velocity, can be loaded on its own. The surrounding bytes of the same line are always fetched as well. The data at hand is therefore ideally laid out contiguously, so that the cache lines are filled with relevant rather than irrelevant data, as it would otherwise result in cache pollution.

A cache is additionally organized as a number of sets, and every memory block maps to exactly one set, determined by its address. Within that set the block may occupy any one of  $n$  slots, called ways, which makes the cache  $n$ -way set associative [13]. When more blocks map to the same set than it has ways, they evict one another and must be reloaded on their next access, even though the cache as a whole is not full. Therefore, such a conflict miss [15] depends not on how much data is accessed, but on how the accessed addresses distribute across the sets, which becomes relevant in Section 4.3.

## 2.3 Data-Level Parallelism via SIMD

Traditionally, to perform an operation on data, the CPU loads the operands into a *General Purpose Register (GPR)* and computes a single result. Such registers hold a single scalar value. On modern processors they are typically 64-bit wide and can also be addressed at narrower widths (e.g., 32, 16 and 8-bit on x86), yet a scalar instruction still consumes two operands and produces exactly one result per operation. Therefore, throughput is bound by one result per instruction, regardless of how many data elements have to be processed. In order to overcome this limitation, modern CPUs contain specialized SIMD vector units.

These units use much wider registers that the CPU treats as a vector of several values, applying a single instruction to all of them at once. The AVX2 instruction set, widely available on current consumer CPUs, provides 256-bit registers, which hold eight single-precision or four double-precision floating-point values [16]. This doubles the width of the preceding 128-bit SSE extension, and is doubled again by AVX-512. Although the examples in this thesis concern floating-point data, the same principle applies to integer operations.

This effectively parallelizes the computation and significantly increases throughput, under the condition that the data is laid out in memory as explained in Section 2.1. In a SoA layout, the values of a single field are stored contiguously and can be loaded directly into a vector register. In an AoS layout, the same values are separated by the other fields of each struct, so the vector unit cannot be fed as intended and vectorization is inhibited. Thus, whether the compiler can vectorize a loop is a direct consequence of the memory layout. To achieve these hardware efficiencies without the programmer having to manually manage memory, the ECS design pattern may be employed.

## 2.4 Entity Component Systems

As the presentation by Bilas [6] at the Game Developers Conference in 2002 shows, there has been a desperate need for the ECS pattern since at least the early 2000s. He explains that with increasing complexity of game systems, the resulting hardcoded class tree must undergo repeated refactoring and will naturally stray further and further from clean OOP principles. He noticed that when re-examining the problem of complex object schemes or structures, he could also view it as a database and that in the end, only the data matters.

This data-centric perspective is the essence of the ECS pattern. It encourages a clean separation of data and logic by distributing these across three kinds of constructs: entities, components and systems. An *entity* is an identifier value to which components may be attributed. A *component* is a domain-specific datatype whose value is comparable to the field value of an object, and a single entity may possess multiple components. Lastly, a *system*, is a domain-specific piece of logic that performs computations on entities and their components. Entities and components represent a game state, with the systems performing the transitions between states [23].

As an example, we revisit the Planet objects from Section 2.1. We discussed that the SoA memory layout would greatly improve performance, but not how this layout would be achieved. Traditionally, one would define a Planet class with fields and methods and then instantiate objects. To transform this into a SoA memory layout, the ECS pattern is applied as follows:

- Entities are created as unique identifiers.
- Components such as `position`, `velocity` and `mass` are associated with these entities. In an archetype-based ECS like Bevy, this ensures that all `position` components are stored in a contiguous array, separate from `velocity` and `mass` [5].
- Systems are defined to perform transformations by querying for entities that possess a certain set of components. For example, a system may update the `position` component of all Planet instances that possess a `position` and a `velocity` component. This allows the system to iterate over a tightly packed array of only relevant data, enabling the use of SIMD instructions and filling the CPU cache with only relevant data.

## 2.5 Rust and Bevy engine

Rust [18] is a systems programming language which provides performance comparable to C++. Further, it achieves memory safety through its unique ownership system. The compiler enforces strict rules at compile-time on how data may be accessed and modified. By doing so, Rust eliminates entire classes of bugs such as data races, and does so without a garbage collector, whose collection pauses would otherwise be undesirable for benchmarking [18].

Bevy [3] is a data-driven game engine written in Rust, built around an archetype-based ECS as its core abstraction [4]. It expands upon Rust's foundation with a framework that prioritizes performance and modularity: archetypes group entities with identical combinations of components, storing each component in a contiguous, per-component array and thereby enforcing the SoA memory layout [5].

# Chapter 3

## Benchmark Design and Implementation

This chapter describes the benchmark program developed as part of this thesis, which is used to measure the performance of two object-oriented and two data-oriented architectures. First, it introduces the simulation on which all performance measurements are based. Then, the four architectures and the three scenarios in which they are measured are defined. The chapter concludes with the configuration and execution of an individual benchmark run.

### 3.1 Simulation

To properly analyze the performance difference between DOP and OOP, the used simulation must be conceptually and mathematically identical across all evaluated architectures within a given scenario. Otherwise, differences in performance cannot be reliably attributed to the effects of adhering to one paradigm or the other, as they could just as likely occur due to inhomogeneous workloads. Additionally, the simulation must provide a workload that is non-trivial enough to cause the property under inspection to dominate the runtime, while still exercising that property. For example, a complex calculation may dominate the run time in a compute-bound scenario, yet it does not exercise the memory hierarchy sufficiently to expose the differences between memory layouts.

The benchmark simulation performs gravitational calculations on a certain number of planets. These planets always have the same three conceptual attributes, a 3D position in space, a 3D velocity, and a payload that acts as a way to artificially inflate the size of the planet objects. The position and velocity of all planets are randomly generated during the setup phase with a seeded random number generator, while the payload is a

fixed size set at compile time. This design enables all three scenarios to operate on the same underlying simulation, with each of them varying a different parameter, e.g., planet count or payload size.

Following the laws of Newtonian gravitation [19], the simulation assumes that all planets are accelerated towards a central body of a large fixed mass (e.g., a black hole) at the origin. Due to the planets being accelerated towards this central mass, the planet's own mass cancels out, so the acceleration depends only on its position relative to that mass. The acceleration therefore simplifies to the following equation,

$$\vec{a} = \frac{G}{r^2} \cdot \hat{d}$$

where  $r$  is the distance between the planet and the center, and  $\hat{d}$  is the unit vector pointing from the planet towards the center. Here,  $G$  is not the physical gravitational constant but a fixed scaling factor that includes the constant central mass. Its value is derived from the gravitational constant  $6.674 \times 10^{-11} \text{ N m}^2 \text{ kg}^{-2}$  scaled by a factor of  $10^{14}$ , i.e.,  $G \approx 6674$ , so that the resulting accelerations remain in a numerically well-behaved range for 32-bit floating-point arithmetic. The planet's velocity and position are then integrated over a fixed time step  $\Delta t$ .

## 3.2 Evaluated Architectures

Chapter 2 contrasted OOP and DOP on the level of paradigms. However, a paradigm itself cannot be benchmarked. Concepts such as encapsulation or a data-oriented mindset leave no direct trace in hardware. What reaches the hardware is the resulting arrangement of data in memory and the pattern in which that data is accessed, which can then be measured. Thus, the comparison is reduced to representative implementations that differ precisely in these two properties. Since the simulated workload itself is identical across all of them (see Section 3.1), any measured difference can be attributed to the memory layout rather than to the workload. In total, four architectures are evaluated, two of which represent OOP and two of which represent DOP design principles. They are arranged such that each of them differs from the preceding one in primarily one property, namely memory layout, grouping of fields and usage of a higher-level ECS. Thus, this arrangement spans a range from maximum indirection to maximum contiguity.

**(1) *Pointer Chasing (PC)*** The PC architecture stores planets as `Vec<Box<PlanetAoS>>`, where `PlanetAoS` has three fields: a `Vec3` for the planet's velocity, another `Vec3` for its position, and an array `[u8; PAYLOAD_SIZE]` for the payload. Every planet is allocated individually on the heap, with the vector solely holding the pointers to said planets. In Rust, this indirection is provided by `Box<T>`, where the content of the box lives on the heap and the box itself is a pointer of known size referring to the content [18]. This architecture represents the reference semantics of common object-oriented languages such as Java or C#, where objects are allocated individually and accessed through references.

Since caches exploit spatial locality [8], this layout is the least favorable one under inspection, as traversing the collection means following a pointer per element instead of having the data be laid out in a contiguous manner. This traversal pattern is commonly referred to as pointer chasing with Drepper [8] using it to demonstrate the limits of hardware prefetching. However, when objects are created within a single uninterrupted loop, a general-purpose allocator may place them in near-contiguous memory, in which case the architecture would degenerate into the AoS architecture described below. To prevent this and ensure the layout causes pointer chasing, the vector is shuffled once after setup. This mirrors a long-living application under manual memory management, such as in C++, whose heap becomes fragmented as objects are allocated and freed over time. Managed languages with compacting garbage collectors, such as Java or C#, relocate live objects and thus reduce such fragmentation, so the shuffled layout represents the worst case for those runtimes rather than their typical state.

**(2) *Array of Structures*** The AoS architecture stores the identical `PlanetAoS` struct in a `Vec<PlanetAoS>`, i.e., without any indirection. The planets are laid out contiguously within the vector, whereas the fields of the planets remain interleaved, as illustrated in Figure 2.1. The position, velocity and payload of one planet are stored adjacently, followed by the fields of the next planet.

Within the object-oriented paradigm, this architecture therefore represents the most contiguous layout achievable. The indirection present in the PC architecture is removed while retaining the conceptual properties of an object, namely that all of its data is kept together. As the AoS and PC architectures share the exact same `PlanetAoS` struct and differ solely in the presence of a `Box<T>`, any measured difference can be directly attributed to the indirection alone.

**(3) Structure of Arrays** The SoA architecture, in contrast to the previous two, adheres to DOP principles. Instead of one vector holding whole planets, one vector is created per individual field. This results in three vectors for the x, y, and z coordinates of a planet's position, three vectors for the x, y, and z values of a planet's velocity and a vector holding the payload. Thus, a planet is no longer a singular entity with inhomogeneous data, but an index shared across seven homogeneous vectors.

This is the layout as described in Section 2.1 and represents the data-oriented ideal. It is implemented manually without any framework, so that it serves as the performance baseline against which any abstraction built on top of a SoA layout is measured. Any performance costs incurred by abstraction built on top of a SoA layout are measured relative to this baseline. However, manually managing and synchronizing seven parallel vectors is error-prone and cumbersome and does not reflect how data-oriented code is written in practice. There, such layouts are typically managed by a framework rather than by hand. This motivates the fourth architecture, which uses the Bevy ECS framework.

**(4) Bevy ECS** The Bevy architecture represents the same idea as the SoA architecture, but implemented with an ECS framework. Bevy groups entities, in this case planets, by archetypes, i.e., by their exact set of components [23], and stores the components of an archetype in contiguous, per-component storages [5]. Since all planets in this simulation possess an identical combination of components (Position, Velocity, Payload), they occupy the same archetype and the resulting storage pattern is thus conceptually equivalent to that of the SoA architecture.

Therefore, the difference to the SoA architecture is not how the data is stored, but the abstraction required to keep such a data-oriented layout maintainable. This includes entity identifiers, archetype metadata, and especially the query mechanism through which components are accessed. The idiomatic query iterator yields only one entity's components at a time, which could hinder vectorization. Thus, `contiguous_iter_mut` is used instead of `iter_mut`, as it returns slices of homogeneous component blocks on which the compiler may perform vectorization. A benchmark accompanying the feature confirms this. The idiomatic iteration shows no AVX2 speed up, whereas contiguous slices do [17]. As it is not yet part of a released engine version, the function is taken from an in-development build [17].

Therefore, Bevy represents DOP as it is realistically adopted and the comparison with the SoA architecture quantifies how much that convenience costs.

**Vector Types and Comparability** These four architectures do not only differ in memory layout, but also in their representation of position and velocity. While PC and AoS store them as `Vec3`, i.e., three consecutive `f32` values, the SoA architecture stores the individual coordinates in separate vectors of `f32`. The Bevy architecture uses `Vec3A`, which extends the three `f32` values by 4 B of padding and is aligned to a 16 B boundary [12]. A single position or velocity value therefore occupies 16 B instead of the 12 B occupied by the three coordinates in the remaining architectures. This is a deliberate decision, as each architecture uses the representation that is idiomatic for it. The measurements thus reflect the performance of code as it would be written in practice, instead of code that has been homogenized for the sake of uniformity.

Enforcing a single representation would distort the comparison in one direction or the other. Forcing Bevy to use `Vec3` would deny it the aligned accesses that vectorized instructions rely on, since 16 B memory accesses are expected to be 16 B aligned and the instruction variants handling unaligned accesses are slower [8]. Conversely, forcing `Vec3A` onto the PC and AoS structs would pad them and therefore inflate the memory footprint the comparison measures. This does not apply to SoA, which stores the coordinates as raw `f32` and thus has no vector type to align in the first place.

The consequence of this decision is that the amount of hot data per planet is not identical across all architectures. Except for Bevy, which carries 32 B of position and velocity per planet, all architectures carry 24 B. Bevy therefore trades a larger data footprint for the alignment its layout exploits. The payload, in contrast, is identical across all architectures and, when varied, is varied for all four simultaneously. This is relevant as it is one of the parameters varied in Scenarios 2 and 3.

### 3.3 Scenario 1: Compute-Bound

The first scenario aims to answer the question of how well the compiler can vectorize the per-planet update and how large the resulting performance difference is, depending on the architectures under inspection. Additionally, the results should show whether a SoA layout is a precondition for effective SIMD vectorization as has been previously stated in Section 2.3. In order to achieve this, this scenario features the entire calculation of gravitational pull (see Section 3.1) and sets the payload size deliberately to 0 B. Otherwise, the memory overhead could distort the results and would not allow proper conclusions from the measurements.

**Parametrization** In this scenario, the number of planets is the only parameter being varied. It is swept from 1 planet up to 30 000, while the payload remains fixed at 0 B and the number of warmup and measurement iterations as well as the number of samples, i.e., repeated measurements, remain constant. Varying solely the planet count enables one to isolate the performance impact of scaling the arithmetic workload linearly, as the memory layout and footprint of a single planet remain identical throughout the sweep. Any difference between the architectures can therefore be attributed to how efficiently each of them executes the same arithmetic. Lastly, the sweep is performed in fine-grained steps rather than in orders of magnitude. This is deliberate, as this approach causes threshold effects to remain visible instead of being averaged away between two distant measurements.

**Update Logic** The gravitational equation is translated directly into code. For every planet, the direction vector towards the center is obtained by negating the planet's position, from which the squared distance follows. The acceleration is then applied along the normalized direction. Lastly, velocity and position are updated over the fixed time step  $dt$ . Listing 3.1 shows the hot loop body being measured for the AoS and PC architectures. The sole difference between the two is the usage of `Box<T>` in the PC architecture, which enables the isolation of the effect of the usage of pointers versus direct memory access.

---

```

1  for planet in planets.iter_mut() {
2      // calculates acceleration
3      let direction = -planet.position;
4      let r_squared = direction.length_squared();
5      let a_scalar = G / r_squared;
6      let a = a_scalar * direction.normalize();
7
8      // updates velocity and position
9      planet.velocity += a * dt;
10     planet.position += planet.velocity * dt;
11 }

```

---

Listing 3.1: Update loop of the AoS and PC architectures

While there is a data dependency between the velocity and position of a planet, i.e., the velocity must be written before position can read it, it is not in the way of vectorization as it is contained within a planet. Whether SIMD vectorization occurs therefore depends on the respective memory layout.

---

```

1  for i in 0..planets.positions_x.len() {
2      let px = planets.positions_x[i];
3      let py = planets.positions_y[i];
4      let pz = planets.positions_z[i];
5      // calculates acceleration
6      let r_sq = px * px + py * py + pz * pz;
7      let r_recip = 1.0 / r_sq.sqrt();
8      let a_scalar = G / r_sq;
9      let nx = -px * r_recip;
10     let ny = -py * r_recip;
11     let nz = -pz * r_recip;
12     // updates velocity and position
13     planets.velocities_x[i] += nx * a_scalar * dt;
14     planets.velocities_y[i] += ny * a_scalar * dt;
15     planets.velocities_z[i] += nz * a_scalar * dt;
16     planets.positions_x[i] += planets.velocities_x[i] * dt;
17     planets.positions_y[i] += planets.velocities_y[i] * dt;
18     planets.positions_z[i] += planets.velocities_z[i] * dt;
19 }

```

---

Listing 3.2: Update loop of the SoA architecture

The SoA implementation performs the same computation, but on six of the seven vectors from Section 3.2, as the payload remains untouched. The difference to AoS and PC is therefore not what is computed, but which values lie adjacent in memory. There, the three coordinates of a single vector are adjacent and consecutive planets are separated by the stride of the entire struct. Here, the x, y, and z values of consecutive planets are adjacent and thus form homogeneous data streams that can be vectorized, at the cost of markedly higher code complexity. To keep the arithmetic identical across architectures, `normalize()` is mirrored through `r_recip`, as it is internally implemented as a multiplication by the reciprocal of the length rather than as three separate divisions.

---

```

1  for (mut pos_slice, mut vel_slice) in query.contiguous_iter_mut(&mut world).unwrap() {
2      for (pos, vel) in pos_slice.iter_mut().zip(vel_slice.iter_mut()) {
3          // calculates acceleration
4          let direction = -pos.0; // unwraps the Position component to its inner Vec3A
5          let r_squared = direction.length_squared();
6          let a_scalar = G / r_squared;
7          let a = a_scalar * direction.normalize();
8          // updates velocity and position
9          vel.0 += a * dt;
10         pos.0 += vel.0 * dt;
11     }
12 }

```

---

Listing 3.3: Update loop of the Bevy architecture with contiguous component slices

Finally, the Bevy architecture performs the same arithmetic and differs only in how the components are reached. As described in Section 3.2, `contiguous_iter_mut` yields slices rather than individual entities, over which the inner loop then iterates directly. Here, the trade-off again is between performance and code-complexity. Unlike the SoA architecture however, Bevy may occupy what could best be called a middle ground.

**Verification** While substantially better performance of the DOP-based architectures would suggest that vectorization has taken place, it would not by itself prove it. The generated x86-64 assembly is therefore inspected to determine whether the compiler emitted scalar instructions (`vaddss`, `vmulss`) or their packed SIMD counterparts (`vaddps`, `vmulps`), and whether 128-bit `xmm` or 256-bit `ymm` registers are used, with examples being present within the analysis in Section 4.2. The inspection uses `cargo-show-asm` [22], which interleaves assembly with the corresponding Rust source lines, and IDA Free [14], whose control flow representation makes the hot loop easier to identify. Vectorization can only occur if the compiler is permitted to emit the required instruction set extensions, enabled here by compiling with `-C target-cpu=native` (further build details in Section 4.1).

## 3.4 Scenario 2: Memory-Bound

As has been established in Section 2.1, cache pollution is the phenomenon when an object's data fills the CPU cache with data that is not relevant to the operation. In this scenario, this additional yet irrelevant data is simulated by the payload field of the planet structs, which is never read and serves solely to increase the size of a planet. The goal is to measure how strongly this unused data affects execution time, and from which object size onwards it begins to do so. As the intent is to have this scenario memory-bound, the calculation of gravitational pull is reduced to the update of a planet's position based on its current velocity, so as not to distort the results of benchmark runs with little payload. Conceptually, the planets do not rotate around a large mass anymore but wander through space in straight lines. Scenario 1 and Scenario 2 are thus complementary: the former minimizes the payload to measure the arithmetic, while the latter minimizes the arithmetic to measure the memory footprint's impact.

**Parametrization** In contrast to Scenario 1, the planet count remains fixed at 10 000, while the payload is the only parameter being varied, sweeping from 0 B up to 20 000 B. The number of warmup and measurement iterations as well as the samples per invocation again remain constant. The payload is deliberately declared as an array of fixed length rather than as a heap-allocated vector, so that it is stored within the struct itself and thus actually increases the size of a planet. In contrast, a vector would merely add a small, fixed-size handle to the structs size and thus leave the struct small. As the array's length is part of its type and thus must be known at compile-time, it requires a re-compilation for every payload size (see Section 3.6). The sweep is again performed in fine-grained steps, as the effects of cache conflict misses (see Section 2.2) are expected to appear at specific object sizes rather than gradually, and would be averaged away between two distant measurements.

**Update Logic** The core difference between architectures is now not directly visible in code anymore, as the differences are located in the underlying memory layout. For the PC and AoS architectures, the payload is interleaved with the position and velocity within each planet struct. Since the cache operates on fixed-size cache lines, every line fetched for the position and velocity also pulls in adjacent payload bytes, wasting precious cache bandwidth on irrelevant data. When performing an operation solely on the position and velocity vectors, only these relevant vectors may be loaded with the payload remaining untouched. Thus, the cache is filled only with relevant data, leading to potentially substantial performance improvements.

---

```

1  // PC and AoS
2  for planet in planets.iter_mut() {
3      planet.position += planet.velocity * dt;
4  }
5  // SoA
6  for i in 0..planets.positions_x.len() {
7      planets.positions_x[i] += planets.velocities_x[i] * dt;
8      planets.positions_y[i] += planets.velocities_y[i] * dt;
9      planets.positions_z[i] += planets.velocities_z[i] * dt;
10 }
11 // Bevy
12 for (mut pos_arr, mut vel_arr) in query.contiguous_iter_mut(&mut world).unwrap() {
13     for (pos, vel) in pos_arr.iter_mut().zip(vel_arr.iter_mut()) {
14         pos.0 += vel.0 * dt;
15     }
16 }

```

---

Listing 3.4: Update loop of all four architectures in Scenario 2

Listing 3.4 shows the reduced complexity of the update loop of all four architectures. The SoA architecture uses a `PlanetsSoA` struct which contains seven vectors, six of which represent the x, y and z values of position and velocity of all planets, with the seventh containing the payload of all planets. As the payload is not required for the update on position, the array is simply not accessed and the cache is thus kept full with the relevant coordinate data. The Bevy architecture follows the same principle, querying only for the position and velocity components and never accessing the payload component. Bevy's idiomatic iterator (`query.iter_mut`) closely resembles the OOP update loop and would offer comparable syntactic simplicity. Here, however, the `contiguous_iter_mut` variant is used instead, as only `contiguous_iter_mut` exposes it as `&mut [T]` slices. This slice-based access allows the compiler to vectorize the loop and thus increases comparability between the Bevy and SoA architectures, at the cost of a more complex and verbose nested loop.

**Verification** While again the assembly code is inspected to verify the existence of vectorization, the main focus now lies on the cache miss rate, which is gathered with AMD `uProf` [1]. A high cache miss rate would clearly indicate the cache being filled with irrelevant data, as a large payload size in relation to the relevant data leads to substantially more cache misses. As the effect of cache inefficiencies and cache conflict misses is highly dependent on the exact payload size used and different payload sizes necessitate separate binaries, it is not feasible to inspect every single binary created. Instead, the binaries at and around strong variations in performance are evaluated for SIMD instructions and cache miss rate to confirm the variations occur due to the aforementioned reasons.

### 3.5 Scenario 3: Allocation and Deallocation

With Scenario 1 measuring the cost of arithmetic and Scenario 2 the cost of a growing memory footprint, Scenario 3 measures the cost of repeatedly allocating and deallocating that data. This scenario simulates the repeated creation and destruction of planets. Practical implementations of this principle can be found with particle systems and projectiles in video games or short living entities in simulations. The core question is how expensive this repeated creation and destruction is, and how it scales with payload size and chunk size. Here, the chunk size declares how many planets are created and destroyed within a single iteration. As the focus is on memory management, the arithmetic in the update loop is left out

completely. Additionally, two subvariants of this scenario are inspected, namely by how much performance differs depending on whether the data of destroyed planets is used after destruction or whether it is discarded.

**Parametrization** This scenario, contrary to Scenarios 1 and 2, involves two parameters: payload size and chunk size. For the main sweep, the chunk size is fixed at 500 while the payload is swept from 0 B to 10 000 B in fine-grained 2 B steps, as the scaling behavior is the main property under inspection. To generate the heatmaps in Section 4.4, chunk size is swept from 1 to 1000 in steps of 20 and payload size is swept from 980 B to 1035 B. As with Scenario 2, every change in payload requires re-compilation.

**Update Logic** As was previously stated, a distinction is made between discarding the popped objects and using their data. The discarding of data best represents conventional particle systems where particles are created, worked with and then destroyed. The reuse of data better represents entities in more complex simulations, where the data may need to be read after destruction. Listing 3.5 shows this difference in code, where in the reuse case the `planets.pop()` function is surrounded by a `black_box`, which in this case simulates the usage of the planet's data. On the contrary, `let _ = planets.pop();` discards the data immediately, simulating a particle system. This distinction and the usage of `black_box` occurs analogously in the SoA and Bevy architectures. For the creation of the planets, the coordinates are filled with 0.

---

```

1 if reuse {
2     for _ in 0..chunk_size {
3         black_box(planets.pop());
4     }
5 } else {
6     for _ in 0..chunk_size {
7         let _ = planets.pop();
8     }
9 }

```

---

Listing 3.5: Deallocation of planets in PC and AoS architectures

In the reuse case, the PC and AoS architectures differ notably. Popping a planet from the AoS vector moves the whole struct out of storage, i.e., an implicit memcpy of the 24 B of coordinates plus the full payload, so its cost grows with the payload size. In the PC architecture, the same pop moves only the 8 B Box pointer while the data stays on the heap, making the cost payload-independent. This only shows in the reuse case, as

`black_box` forces the popped value to materialize; in the discard case the compiler is free to optimize the move away. This advantage is offset by allocation cost. In the PC architecture, every `push()` triggers a `malloc` and every `pop()` a `free`, for each element in every cycle. The AoS vector instead keeps its capacity across the cycles and thus barely allocates once it has grown. Therefore, PC trades a cheap pop against constant `malloc` and `free` overhead, and which effect dominates is expected to depend on payload and chunk size.

The difference between the AoS and SoA architectures lies in the AoS architecture requiring only a single `push()` and `pop()` per planet, whereas the SoA architecture requires seven such calls due to the split management of values across seven vectors. Bevy provides its own `spawn()` and `despawn()` functions; in the reuse case, however, explicit calls to `get::()`, `get::()` and `get::()` are necessary, as `despawn()` does not return the component data of the removed entity.

**Verification** To verify the results, the assembly code is inspected to confirm that allocations and deallocations have been carried out as intended and have not been optimized away by the compiler. For this, `cargo-show-asm` [22] is used. Additionally, the usage of the `memcpy` instruction is inspected in the AoS architecture, to confirm the hypothesis of it requiring expensive memory movements instead of simple pointer popping. A heatmap is used to visualize the runtimes and more easily discern if any threshold effects or performance anomalies occur.

## 3.6 Execution and Configuration

All architectures and benchmarks are compiled into a single executable. An individual benchmark run is configured almost entirely through *Command Line Interface (CLI)* arguments, with the parsing of the commands being handled by the `clap` crate [20]. Table 3.1 lists the available arguments.

The program itself does not contain the functionality of parameter sweeping, i.e., varying parameter values within defined ranges. Instead, PowerShell and bash scripts are provided which repeatedly execute the benchmark with varying values. This keeps the measured code free from any sweeping logic and makes every measurement reproducible from its arguments alone. In addition, the scripts recompile the benchmarks

| Short | Long         | Default  | Description                                   |
|-------|--------------|----------|-----------------------------------------------|
| -b    | --benchmark  | required | Scenario: s1, s2, s3 or reset                 |
| -p    | --planets    | 10 000   | Number of planets created during setup        |
| -w    | --warmups    | 2500     | Warmup iterations, results discarded          |
| -i    | --iterations | 5000     | Measured iterations per sample                |
| -s    | --samples    | 5        | Number of samples per invocation              |
| -c    | --chunksize  | 100      | Elements added and removed per iteration (S3) |
| -r    | --reuse      | false    | Enables the reuse variant (S3)                |
| -d    | --debug      | false    | Prints a runtime summary to the console       |

Table 3.1: CLI arguments for configuring the benchmark executable

with varying payload sizes for Scenarios 2 and 3, as the payload size is a compile-time constant and thus must be defined before the execution of the benchmarks.

A single invocation executes all four architectures in sequence. For every architecture and for every sample, the corresponding benchmark function is called twice: once with the number of warmup iterations and once with the number of benchmark iterations. Calling the same function for both warmup and benchmark iterations ensures that the cache is in a comparable state when the measurements begin. A typical invocation is shown in Listing 3.6.

```
cargo run --profile o3 -- -b s3 -w 1000 -i 5000 -s 5 -c 500 -r -d
```

Listing 3.6: A benchmark invocation for Scenario 3 via the command line

Measurement results are appended to a *Comma-Separated Values* (CSV) file per architecture and scenario. For example, the above invocation would append the measurements to the files `bevy.csv`, `soa.csv`, etc. located at `../Benchmarks/diagrams/s3/csv/`. Each line contains an entry for the respective architecture, planet count, number of iterations, chunk size, reuse of deallocated data in Scenario 3, payload size in bytes as well as the elapsed time in nanoseconds. In order to delete the generated CSV files, the `reset` value for the benchmark argument may be used. This acts as a way to create a clean slate and prevent appending measurements to those of previous benchmarks, should the CSV files not have been moved yet.

# Chapter 4

## Evaluation

Chapter 3 defined the benchmark design and introduced the four architectures under comparison, as well as the three scenarios. This chapter presents and interprets the measurements obtained from the benchmark runs. First, the setup and environment are documented. The subsequent sections then present the measurements of the three scenarios. A final discussion draws together the trends observed across all scenarios, followed by a discussion of the limitations of this evaluation.

### 4.1 Methodology

**Hardware** All benchmark runs were performed on a desktop machine equipped with an AMD Ryzen 7 5800X processor and 32 GB of DDR4-3200 memory. The processor provides eight physical cores, each exposing two hardware threads through *Simultaneous Multithreading (SMT)*. Its L1 data cache is 32 KiB in size, 8-way set associative, and organized in 64 B cache lines. The processor further supports the AVX2 instruction-set extensions, which the vectorized code paths rely on.

**Environment** The benchmarks were run on Ubuntu 26.04 LTS with the kernel 7.0.0-28-generic. Precision Boost Overdrive and C-States were disabled in the BIOS, the former to prevent frequency boosting, the latter to keep the benchmark core from entering low-power idle states. The remaining modifications were made at runtime by a setup script. Core Performance Boost was disabled and the frequency of all cores was locked to 3.6 GHz, thus eliminating another potential confounder. The SMT sibling sharing the physical core with logical CPU 2 (logical CPU 10) was disabled, so that no tasks running on logical CPU 10 would consume resources of core 2, to which the benchmark was pinned via taskset.

**Build/Compiler** The benchmarks were compiled with the rustc 1.95.0 compiler. Rust supports compilation profiles, which let the programmer declaratively define a set of compilation options. For the benchmark program, the profile o3 was defined, which inherits from the standard release profile, explicitly disables debug information, and sets the optimization level to 3. Two further compiler flags were set in the execution scripts: `-C target-cpu=native` and `-C link-arg=-fuse-ld=mold`. The former targets the CPU and enables AVX2 (supported by the used processor), the latter selects the mold linker, a drop-in replacement that only reduces linking time while producing an equivalent executable [25]. This choice therefore has no effect on the run time of the architectures.

**Measurement Protocol** To further diminish the impact of OS jitter, a setup with iterations and samples is used. In a given scenario, the four architectures perform a number of warmup runs, whose results are discarded, followed by a number of measured iterations. The warmup lets the caches reach a steady state before timing begins. The architectures are measured in the order of PC, AoS, SoA and Bevy. This whole procedure is executed samples times.

| Scenario | Warmups | Iterations | Samples | Measured Iterations |
|----------|---------|------------|---------|---------------------|
| S1       | 100     | 50         | 10      | 500                 |
| S2       | 200     | 200        | 5       | 1000                |
| S3       | 100     | 50         | 5       | 250                 |

Table 4.1: Per-scenario measurement parameters.

As an example, consider the measurement parameters for Scenario 2 in Table 4.1. Instead of each architecture being measured in one block of 1000 iterations, the architectures are measured in interleaved, smaller blocks, namely 5 samples of 200 iterations in the case of Scenario 2. Thus, a larger system interrupt would be more likely to affect multiple or all architectures instead of only one. Each sample stores the run time over its iterations update steps as a single `time_ns` value in the CSV. The division by the iteration count, resulting in the time per iteration, as well as the averaging over the samples are performed when the plots are generated, which display the mean per-iteration time across all samples.

## 4.2 Scenario 1: Compute-Bound

Scenario 1 measures performance in the compute-bound case, where the arithmetic of the physical update (see Section 3.1) dominates the run time. The only swept parameter is the planet count, and since all architectures perform the identical update, any measured difference stems solely from the memory layout and the resulting code.

**Scaling Behavior** Figure 4.1 plots the average per-iteration time in microseconds against the planet count in thousands, on linear axes and with one line per architecture. As is clearly visible, all four architectures scale linearly, yet with clear performance differences between them. The most performant one in calculating the positional update of 30 000 planets is, with a big margin, the SoA architecture, only taking 22.62  $\mu\text{s}$  per iteration. In second place is the other data-oriented approach, the Bevy architecture, requiring 106.34  $\mu\text{s}$  per iteration. Then follows the AoS architecture with 138.32  $\mu\text{s}$  and concluding with the PC architecture with 213.91  $\mu\text{s}$  per iteration.

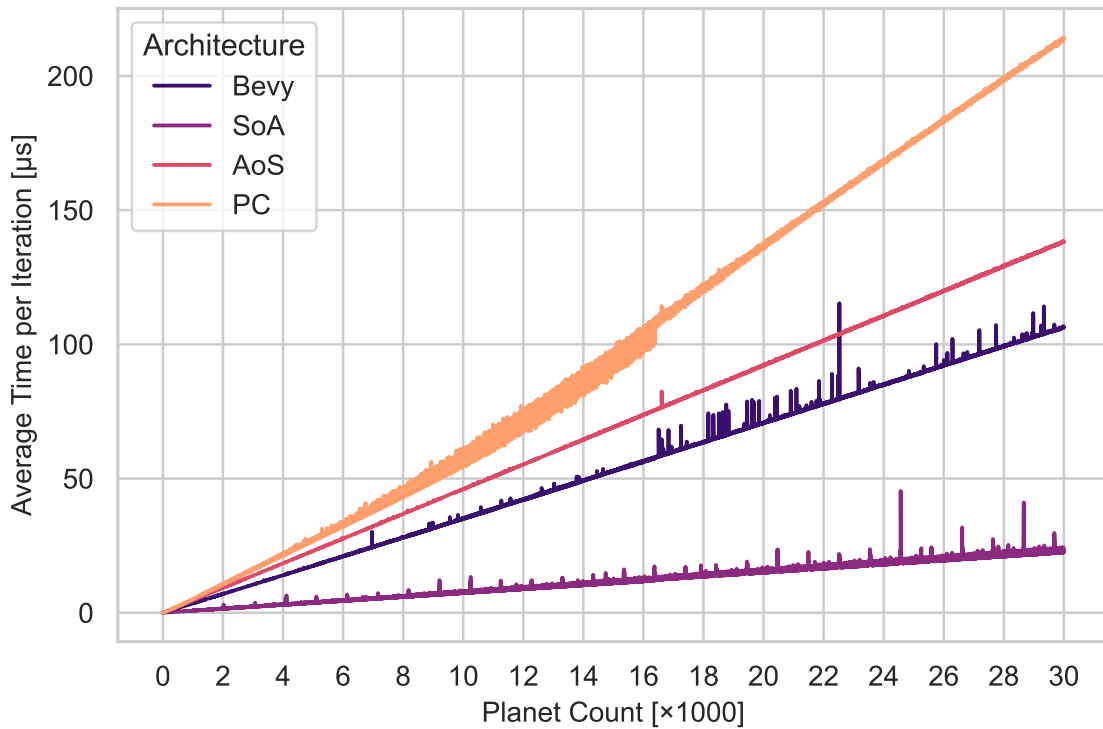


Figure 4.1: Average time per iteration versus planet count

Note the wide yet relatively constant variation in measurements regarding the PC architecture, which lessens after 16 384 planets. Additionally, a number of spikes in execution time are visible in the measurements of the Bevy and SoA architectures. However, these spikes do not affect the overall trend.

**Assembly Analysis** To verify the results in Figure 4.1, the machine code of the `bench_s1_pc`, `bench_s1_aos`, `bench_s1_soa`, and `bench_s1_bevy` functions is inspected for the wider ymm registers and packed instructions. As it can reach several hundred lines, Listing 4.1 focuses on the hot loop’s arithmetic, i.e., the calculation of acceleration and its application to velocity and position. The instructions are reordered to match the source order. For the SoA architecture, only the x-component of the update is shown for brevity.

| SoA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | AoS                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1 ; let px = planets.pos_x[i]; 2 vmovups ymm5, ymmword ptr [r10 + 4*rax] 3 ; let py = planets.pos_y[i]; 4 vmovups ymm6, ymmword ptr [r12 + 4*rax] 5 ; let pz = planets.pos_z[i]; 6 vmovups ymm7, ymmword ptr [r14 + 4*rax] 7 8 ; let r_sq = px * px + py * py + pz * pz; 9 vmulps ymm8, ymm5, ymm5 10 vmulps ymm9, ymm6, ymm6 11 vaddps ymm8, ymm8, ymm9 12 vmulps ymm9, ymm7, ymm7 13 vaddps ymm8, ymm8, ymm9 14 ; let r_recip = 1.0 / r_sq.sqrt(); 15 vsqrtss ymm9, ymm8 16 vdivps ymm9, ymm3, ymm9 17 18 ; let a_scalar = G / r_sq; 19 vdivps ymm8, ymm4, ymm8 20 ; let nx = -px * r_recip; 21 vmulps ymm5, ymm9, ymm5 22 23 ; planets.vel_x[i] += nx * a_scalar * dt; 24 vmulps ymm5, ymm8, ymm5 25 vmulps ymm5, ymm0, ymm5 26 vsubps ymm5, ymm10, ymm5 27 ; planets.pos_x[i] += planets.vel_x[i] * dt; 28 vmulps ymm5, ymm0, ymmword ptr [r11 + 4*rax] 29 vaddps ymm5, ymm5, ymmword ptr [r10 + 4*rax] 30 vmovups ymmword ptr [r10 + 4*rax], ymm5 31 add rax, 8 </pre> | <pre> 1 ; let dir = -planet.pos; 2 vmovsd xmm5, qword ptr [rax + rsi + 12] 3 vmovss xmm3, dword ptr [rax + rsi + 20] 4 5 ; let r_squared = dir.length_squared(); 6 vmulps xmm6, xmm5, xmm5 7 vmulss xmm4, xmm3, xmm3 8 vmovshdup xmm7, xmm6 9 vaddss xmm6, xmm6, xmm7 10 vaddss xmm4, xmm6, xmm4 11 12 ; let a_scalar = G / r_squared; 13 vdivss xmm6, xmm1, xmm4 14 ; let a = a_scalar * direction.normalize(); 15 sqrtss xmm4, xmm4 16 vdivss xmm4, xmm2, xmm4 17 vmulps xmm7, xmm7, xmm5 18 vmulps xmm6, xmm0, xmm7 19 20 ; planet.velocity += a * dt; 21 vmovsd xmm7, qword ptr [rax + rsi] 22 vsubps xmm6, xmm7, xmm6 23 vmovlps qword ptr [rax + rsi], xmm6 24 ; planet.pos += planet.vel * dt; 25 vmulps xmm6, xmm0, xmm6 26 vaddps xmm5, xmm5, xmm6 27 vmulss xmm4, xmm9, xmm4 28 vmovlps qword ptr [rax + rsi + 12], xmm5 29 vaddss xmm3, xmm3, xmm4 30 vmovss dword ptr [rax + rsi + 20], xmm3 31 add rsi, 24 </pre> |

Listing 4.1: ASM arithmetic, SoA versus AoS. In line 1, `r10` is the base address of the `pos_x` array and `rax` the element index. `4*rax` turns it into a byte offset, an f32 being 4 B.

Listing 4.1 shows that the SoA architecture is able to work with the wider 256-bit ymm registers, while the AoS architecture is forced to operate on the narrower 128-bit xmm registers. Additionally, the SoA architecture is able to vectorize, relying heavily on packed instructions such as `vmulps` and `vsqrtps`. Finally, the `add rax, 8` advances the index by eight elements, so the next eight planets are processed.

In contrast, the AoS architecture cannot vectorize fully, falling back to scalar instructions such as `vdivss` and `sqrtss`. Additionally, the stride `add rsi, 24` equals the size of the `PlanetAoS` struct in bytes, confirming that only one planet is processed per iteration rather than eight.

| AoS                                                                                                                                     | PC                                                                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1 ; planet.position (direct load) 2 vmovsd xmm5, qword ptr [rax + rsi + 12] 3 ; ... identical arithmetic ... 4 add rsi, 24 </pre> | <pre> 1 ; planet.position (pointer load + deref) 2 mov rdi, qword ptr [rcx + rsi] 3 vmovsd xmm5, qword ptr [rdi + 12] 4 ; ... identical arithmetic ... 5 add rsi, 8 </pre> |

Listing 4.2: ASM Load sequence: AoS versus PC

Regarding the low performance of the PC architecture, the machine code shows that the arithmetic between the AoS and PC architectures is identical. As visible in Listing 4.2, the main difference is how the planet's data is accessed, with AoS able to load data directly and PC having to fetch the data through the pointer address. This is further confirmed by the stride length of 24 for AoS, which is the `PlanetAoS` struct's size in bytes, whereas the stride length for PC is 8, a pointer's size in bytes.

The Bevy architecture does not compile to a single, isolated hot loop, as the chunked entity iteration produces several separate loops. Therefore, no representative excerpt is shown. Its arithmetic is partially vectorized, as some ymm registers are present in comparison to the object-oriented architectures, which use only xmm registers. However, the usage of these ymm registers is interwoven with shuffle instructions, e.g., `vshufpd`, that assemble the lanes, i.e., the individual value slots within a vector register. This reflects only partial vectorization compared to SoA, which in contrast can cleanly operate on eight planets at once. This is likely a consequence of the `Vec3A` layout, which stores each entity's x, y, and z coordinates together unlike the SoA architecture. However, the usage of the ECS still enables a considerable performance improvement compared to the object-oriented architectures, as it enables at least partial vectorization and usage of wider registers.

**Interpretation** To summarize, the memory layout dictates whether vectorization can occur or not. As was seen, the SoA architecture was able to fully exploit the usage of wide registers and packed instructions due to the component arrays, whereas the object-oriented approaches employing interleaved memory layouts were not able to vectorize. Bevy lies in the middle, achieving partial vectorization. Additionally, the effect of memory locality was isolated between the AoS and PC architectures, demonstrating that vectorization is not the only aspect that can improve performance. To conclude, in a compute-bound scenario, the benefit of a data-oriented approach lies not in performing each calculation faster (the arithmetic is identical) but in how many planets are processed per instruction. The performance of Bevy shows that a practical, ECS-based approach can outperform the object-oriented layouts in this compute-bound case, though the necessity to use structs such as Vec3A impedes the full speed-up that a pure SoA architecture would achieve. Thus, programs do not adhere exactly to either DOP or OOP, but lie on a spectrum spanned between the two.

### 4.3 Scenario 2: Memory-Bound

In contrast to Scenario 1, which measured the performance difference when compute-bound, Scenario 2 measures the impact when the computational workload is negligible, but large amounts of data are accessed. As defined in Section 3.4, the hot loop is reduced to a positional update based on existing velocity, with the payload size as the swept parameter. The decisive difference is that the object-oriented architectures carry the payload inline and must load it, whereas the SoA and Bevy architectures keep it separate and skip it during the update.

**Scaling Behavior** Figure 4.2 plots the average per-iteration time in microseconds against the payload size in KiB, on linear axes with one line per architecture. As was stated in Section 2.4, an ECS benefits greatly from the separation of relevant and irrelevant data, which is exactly what these results show. While the execution time of the PC and AoS architectures increases with payload size, the SoA and Bevy architectures remain constant. The order of performance is the same as in Scenario 1, with SoA being the fastest with an average time per iteration across the sweep of  $2.32\ \mu\text{s}$ . In second is the Bevy architecture with  $5.03\ \mu\text{s}$ , followed by the AoS architecture with  $26.69\ \mu\text{s}$  and the PC architecture with  $37.94\ \mu\text{s}$ .

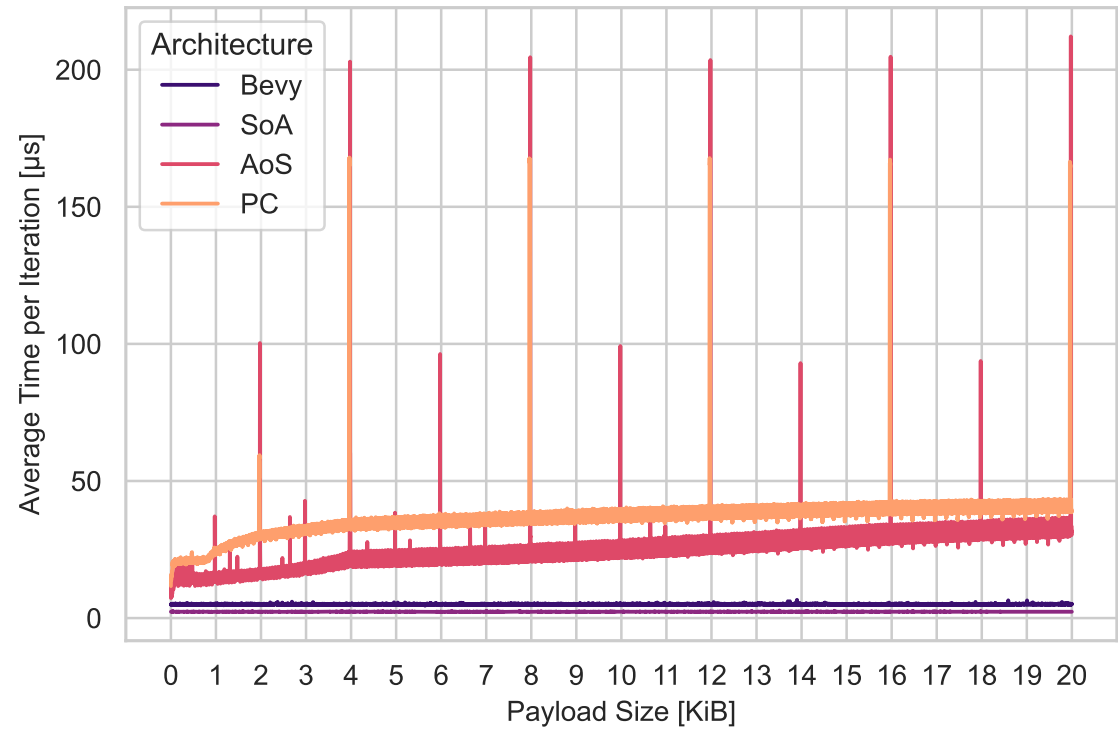


Figure 4.2: Average time per iteration versus payload size

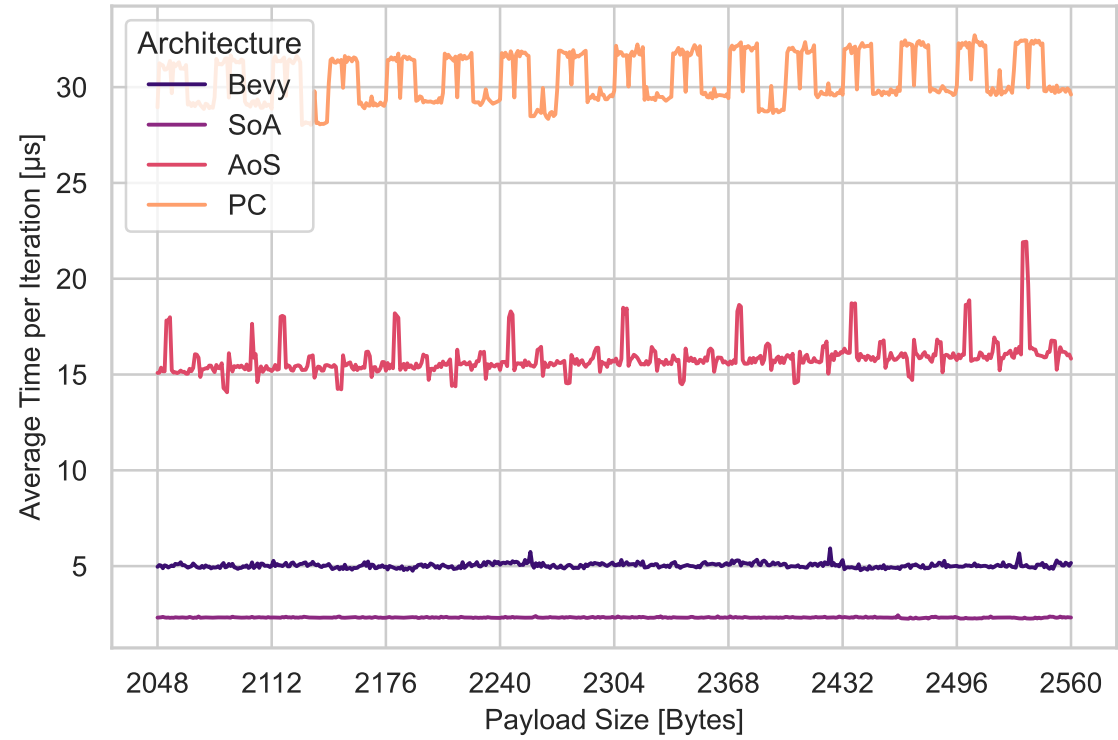


Figure 4.3: Pattern in execution time of PC and AoS architectures

While the data-oriented architectures feature smooth curves in Figure 4.2, the object-oriented architectures feature a distinct pattern of spikes. Both architectures possess large spikes in execution time at multiples of around 4 KiB, whereas the AoS architecture, in addition, features medium spikes at multiples of 2 KiB, as well as small spikes when the payload size is an odd multiple of around 1 KiB.

Additionally, what originally appeared as jitter in Figure 4.2 is another pattern in execution times, as Figure 4.3 clearly shows. Especially for the AoS architecture, a clear period of 64 B can be noted when looking at the taller spikes of the micro-pattern.

**Cache Analysis** The PlanetAoS struct possesses 24 B of relevant data, i.e., Vec3 velocity and Vec3 position. Thus, the size of a planet in the object-oriented architectures is exactly these 24 B plus the payload size, resulting in the aforementioned spikes being situated at *exactly* multiples of 1024 B. Additionally, zooming in reveals that these multiples form an upper bound, with each spike always being 4 B in width, as is visible in Figure 4.4. This occurs for all three spike categories: small, medium, and large. Within each category, lighter shades denote smaller and darker shades larger payload sizes.

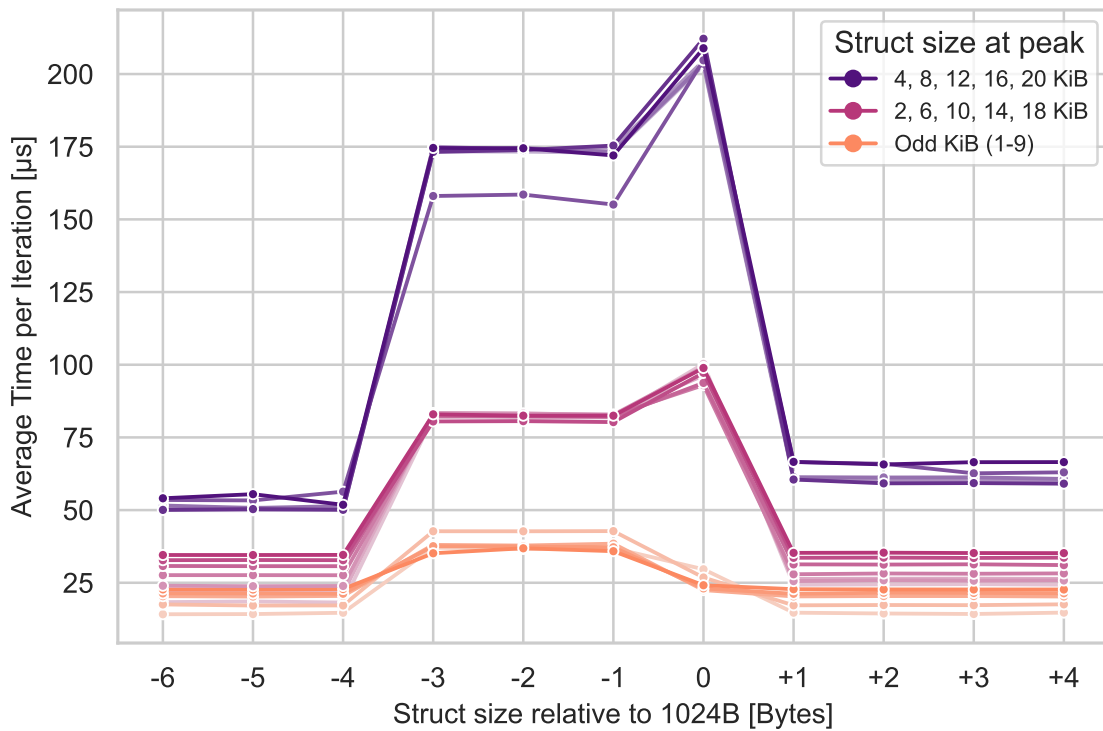


Figure 4.4: Spike position in relation to offset to struct size

This alignment to a power of two suggests the problem originating from a cache phenomenon called cache conflict misses, which Table 4.2 and Figure 4.3 in combination can prove. Table 4.2 presents the L1 cache miss rate as well as the *Cycles per Instruction (CPI)* of several binaries compiled for Scenario 2, the measurements being taken with AMD  $\mu$ Prof [1].

| Payload [B]       | PC   |        | AoS   |        | SoA  |        | Bevy |        |
|-------------------|------|--------|-------|--------|------|--------|------|--------|
|                   | CPI  | L1 [%] | CPI   | L1 [%] | CPI  | L1 [%] | CPI  | L1 [%] |
| 998 <sup>*</sup>  | 1.08 | 21.19  | 1.47  | 26.67  | 0.79 | 32.25  | 0.64 | 18.86  |
| 1010 <sup>+</sup> | 1.05 | 21.23  | 0.71  | 20.88  | 0.77 | 31.80  | 0.66 | 18.93  |
| 2022 <sup>*</sup> | 1.27 | 25.52  | 6.43  | 77.11  | 0.68 | 31.66  | 0.58 | 18.86  |
| 2034 <sup>+</sup> | 1.28 | 26.04  | 0.72  | 21.10  | 0.98 | 37.18  | 0.56 | 18.77  |
| 4070 <sup>*</sup> | 1.55 | 29.97  | 12.68 | 96.79  | 0.77 | 31.66  | 0.73 | 18.94  |
| 4082 <sup>+</sup> | 1.75 | 33.10  | 0.81  | 23.83  | 0.75 | 31.57  | 0.55 | 19.10  |

Table 4.2: CPI and L1 data cache miss ratio per architecture. Payload pairs mark the spikes at 1024 B, 2048 B and 4096 B: <sup>\*</sup> is the middle of the spike itself, <sup>+</sup> is just past it.

While the data-oriented architectures feature relatively constant values for CPI and L1 cache miss rate with no clear pattern between the on-spike (998 B, 2022 B and 4070 B) and off-spike (1010 B, 2034 B and 4082 B) values, the opposite is true for the AoS architecture. Here, the L1 cache miss rate clearly increases when execution time spikes, showing values of 26.67 %, 77.11 % and 96.79 % respectively. Additionally, the CPI increases significantly on spikes, with values of 1.474, 6.426 and 12.680 respectively. For PC, the miss rate and CPI rise only gradually and shows no on-/off-spike difference. Its spikes are therefore not explained by this measurement and are likely shifted by the additional pointer indirection and allocator offsets.

While the values in Table 4.2 indicate that the spikes occur due to a cache phenomenon, they do not fully explain *what* cache phenomenon is to blame. To find the cause, the cache architecture was inspected. For these benchmarks, an AMD Ryzen 7 5800X was used, which features an 8-way set associative cache with a L1 data cache 32 768 B in size and 64 B cache lines. The L1 cache is divided into 64 sets, following  $32\,768\text{ B} / (8\text{ ways} \times 64\text{ B})$  [13]. Each planet's data maps to one of these sets based on its address, the set being calculated by dividing the address by the 64 B line size and taking the result modulo 64.

Table 4.3 lists this progression. When the struct size is a power of two, the planets fall into only a few sets, e.g., a struct size of 4096 B maps every planet to the same set, 2048 B to two sets, 1024 B to four and so on. Each set holds only eight planets at once, since the cache is 8-way set associative. When more than eight planets map to the same set, each new one evicts an earlier planet, which must then be reloaded from memory on its next access. This repeated eviction and reloading is what raises the L1 miss rate seen in Table 4.2 and produces the spikes in execution time. It also explains their hierarchy, as at a struct size of 4096 B every planet lands in the same set, at 2048 B every second planet, and at 1024 B every fourth. The fewer sets involved, the more planets collide, which is why the 4096 B spikes are the largest and the 1024 B spikes the smallest.

| Struct size [B] | Set jump | Sets touched (cycle)   |
|-----------------|----------|------------------------|
| 128             | 2        | 2, 4, 6, 8, 10, ...    |
| 256             | 4        | 4, 8, 12, 16, 20, ...  |
| 512             | 8        | 8, 16, 24, 32, 40, ... |
| 1024            | 16       | 16, 32, 48, 0, 16, ... |
| 2048            | 32       | 32, 0, 32, 0, ...      |
| 4096            | 0        | 0, 0, 0, 0, ...        |

Table 4.3: Cache set touched by planets as a function of struct size

**Interpretation** In a memory-bound scenario the separation of relevant and irrelevant data decides the performance of an architecture. The SoA and Bevy architectures hold the irrelevant data in separate arrays and components, resulting in constant execution times. In contrast, the object-oriented architectures hold the irrelevant data in the struct itself, polluting the cache and increasing execution times. The conflict-miss spikes are a structural artifact of this interleaved layout, which the data-oriented architectures cannot exhibit at all. While in Scenario 1 DOP won by leveraging vectorization, it wins in this scenario through better cache efficiency. However, Bevy was able to close the gap to the SoA architecture far better than in Scenario 1, as the usage of the Query mechanism omits the irrelevant data just as the SoA architecture never touches its separate payload array. In conclusion, the performance is determined by the memory layout rather than by the computation itself.

## 4.4 Scenario 3: Allocation and Deallocation

This scenario investigates the differences in performance regarding allocation and deallocation of data. Per iteration, `chunk_size` planets are created and then removed, across a payload sweep, where the payload size influences the object size. To simulate both a particle system and a more complex simulation, two subscenarios are used, namely drop and reuse. In the drop case, the data is discarded once the planet is removed. In the reuse case, the popped value is passed to `black_box`, which forces it to be materialized and thereby triggers a memcopy of the payload.

The four architectures differ in how their data is allocated. The PC architecture stores each planet in a `Box<T>`, so every object requires an individual allocation and deallocation (`malloc / free`). The AoS and SoA architectures hold their planets directly in vectors and can therefore be allocated in bulk. The Bevy architecture uses its own ECS spawn / despawn logic.

Unlike the previous scenarios, no arithmetic is performed on the planets, as the timed region contains only the creation and removal of objects, isolating the cost of allocation and deallocation.

**Scaling Behavior (Reuse)** Figure 4.5 shows the performance in the reuse case on a logarithmic scale. As is visible, the Bevy architecture, in contrast to Scenarios 1 and 2, no longer beats the AoS architecture. Its execution times start high with an iteration time of  $59.17\ \mu\text{s}$  with a payload of 0 B, whereas the AoS architecture is roughly ten times faster with  $5.95\ \mu\text{s}$ , closely followed by the SoA architecture with  $6.06\ \mu\text{s}$ , which is followed by the PC architecture with  $13.05\ \mu\text{s}$ .

However, this order does not last beyond a payload of 1000 B, as the SoA architecture outperforms AoS with increasing payload and the relative gap to the Bevy architecture decreases. At a payload of 5 kB, Bevy takes, on average,  $194.61\ \mu\text{s}$  per iteration, AoS takes  $127.59\ \mu\text{s}$  and SoA takes  $94.81\ \mu\text{s}$ . The PC architecture's execution times rise steeply at a payload of roughly 1000 B but then follow the same scaling as the other three architectures, resulting in an average iteration time of  $2641.33\ \mu\text{s}$  at a payload of 5 kB.

Lastly, while there are some variations in execution time in the vicinity of 4 kB and 8 kB, they do not undermine the trend or the ordering of architectures. However, below a payload of 1000 B, the measurements for all architectures except Bevy overlap within the measurement noise and cannot be cleanly separated.

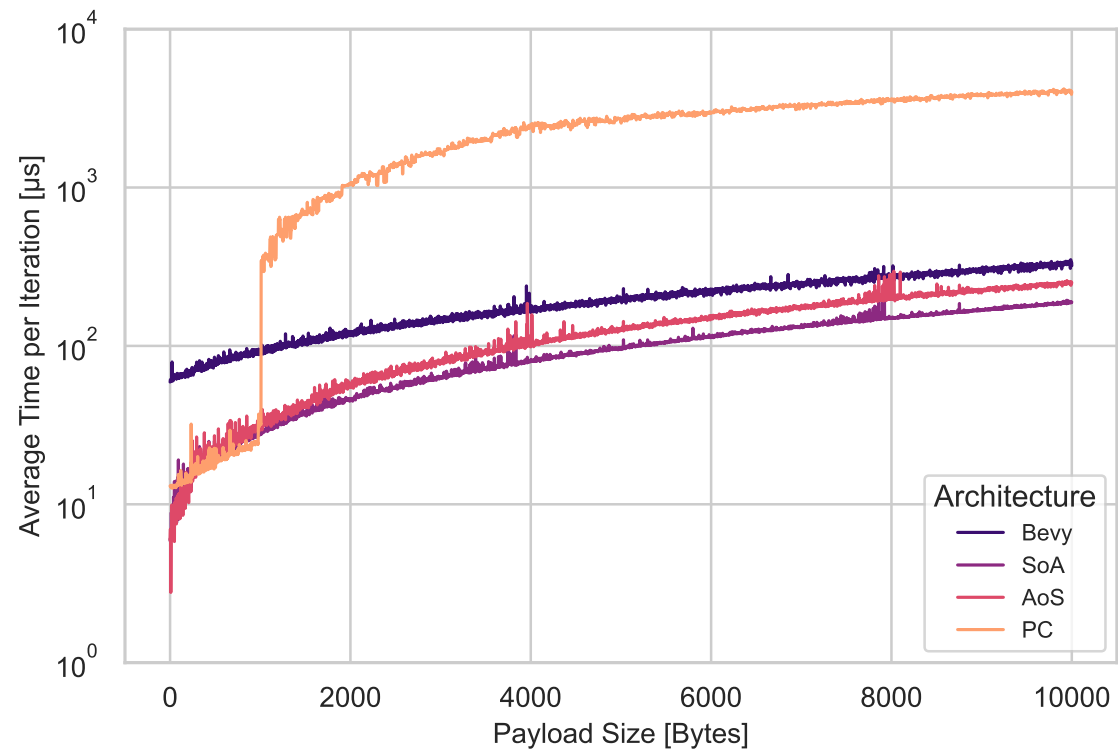


Figure 4.5: Reuse Case with a chunk size of 500

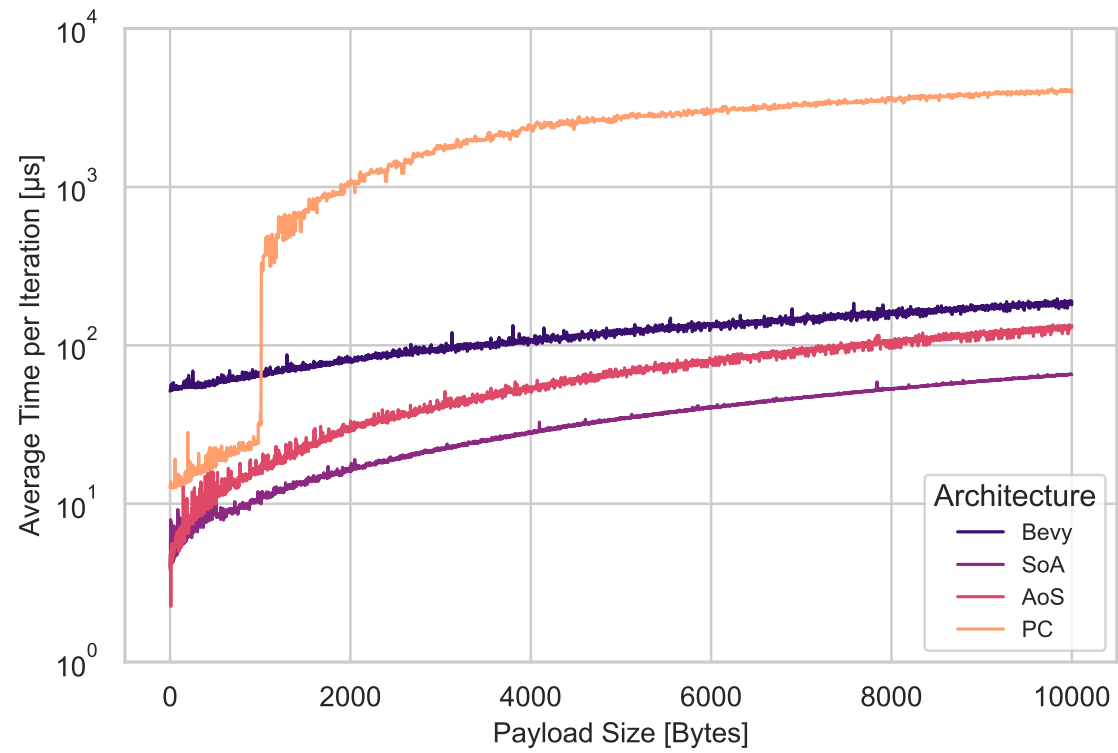


Figure 4.6: Drop Case with a chunk size of 500

**Scaling Behavior (Drop)** In contrast, the drop case does not feature the same variations in execution time at 4 kB and 8 kB. The SoA architecture leads in performance, followed by the AoS architecture and then Bevy. While the PC architecture does not experience a stark difference in performance, the other three architectures' average execution times drop significantly. At a payload of 5 kB, the values for SoA, AoS, Bevy and PC are 34.80  $\mu$ s, 69.21  $\mu$ s, 120.13  $\mu$ s and 2744.62  $\mu$ s respectively which are 36.7 %, 54.2 %, 61.7 % and 103.9 % of the execution times measured in the reuse case. Again, the PC architecture shows a steep rise in execution times at roughly 1 kB payload size.

**Analysis (Reuse versus Drop)** In the reuse case, the two bulk-storage architectures take substantially longer than in the drop case: the drop case completes in only 37 % to 62 % of the reuse time, while the PC architecture is unaffected and stays within measurement noise. The cause is that `black_box` forces the compiler to materialize the popped planet, which requires a `memcpy` the size of one planet (24 B + payload). In the drop case this copy is unnecessary, as the popped element is never read and the compiler only decrements the vector's length.

| AoS Reuse                                                                                                                                                                                                                 | AoS Drop                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <pre> 1 ; black_box(planets.pop()); 2 dec rbp 3 mov qword ptr [rax + 16], rbp 4 imul rsi, rbp, 2024 5 add rsi, r15 6 mov edx, 2024 7 mov rdi, r8 8 call r13 ; r13 = memcpy </pre>                                         | <pre> 1 ; let _ = planets.pop(); 2 dec rax 3 mov qword ptr [rdx + 16], rax 4 mov r14, rax 5 test rax, rax </pre> |
| PC Reuse                                                                                                                                                                                                                  |                                                                                                                  |
| <pre> 1 ; black_box(planets.pop()); 2 lea rcx, [r15 - 1] 3 mov qword ptr [rbx + 16], rcx 4 mov rax, qword ptr [r13 + 8*r15 - 8] 5 mov qword ptr [rsp + 80], rax 6 mov esi, 2024 7 mov edx, 4 8 call __rust_dealloc </pre> |                                                                                                                  |

Listing 4.3: ASM of `pop()`: AoS reuse versus drop (top), PC (bottom)

Listing 4.3 shows the AoS pop in both cases and the PC pop in the reuse case, all with a payload of 2 kB. In the AoS reuse case, the address of the popped element is first computed (lines 4–5) as its index times the object size of 2024 B. Line 8, `call r13`, is the decisive instruction: `r13` holds

the address of `memcpy`, so the element is copied out in full. This call is not present in the drop case, where the popped element is never read and only the length is decremented (`dec rax`).

In the PC reuse case, no payload `memcpy` occurs. Instead, only the 8 B `Box` pointer is read from the vector (line 4) and materialized for `black_box` (line 5). The `Box` is then freed via call `__rust_dealloc` (line 8), which is present in the drop case as well and dominates the cost in both. Thus AoS and SoA (not shown) differ between reuse and drop essentially by one payload-sized `memcpy`, whereas for PC only a pointer is copied and the allocation cost dominates.

**Analysis (PC anomaly)** Figure 4.5 and Figure 4.6 both showed a steep rise in execution time of the PC architecture at a payload of roughly 1 kB. Figure 4.7 shows that the cutoff is located at exactly 1008 B of payload and is independent of the chunk size, which only scales the height of the slow region. An identical cutoff exists in the drop case (not pictured). At this cutoff payload, the object size is 1008 B plus the 24 B of fixed fields, i.e., exactly 1032 B. As the cutoff is not gradual but sharp, it points to an allocator threshold that is exceeded above a struct size of 1032 B.

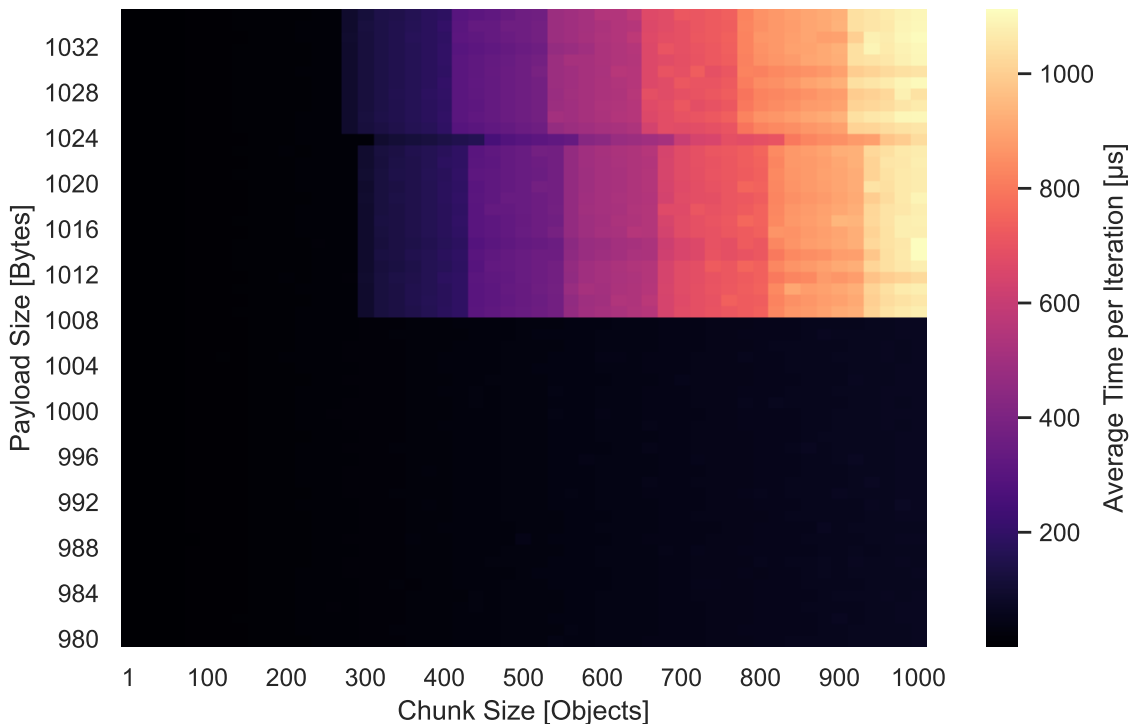


Figure 4.7: Finer sweep at 1 kB of payload

This exactly matches the default value of the glibc tunable `tcache_max`, which on 64-bit systems is 1032 B [24]. This value is the largest object size that glibc still keeps in a small per-thread cache of recently freed blocks. Reusing a block from this cache is considerably faster than a full allocation, as it needs no coordination between threads [7]. As long as an object is at most 1032 B in size, i.e., for payloads up to 1008 B, every creation and removal in the PC architecture is handled by this fast cache. Once the object grows beyond 1032 B, the cache no longer applies and each object must instead be requested from and returned to the general allocator, which is what produces the sharp rise in execution time.

To verify this, the fine sweep from Figure 4.7 was rerun with `GLIBC_TUNABLES=glibc.malloc.tcache_count=0`, which disables the per-thread cache [24]. Without the cache, the fast area below 1008 B should disappear and the execution times should rise to the level otherwise seen only above the cutoff. Figure 4.8 confirms this: the sharp edge is gone, and the times below 1008 B now match those measured above 1008 B in the previous run.

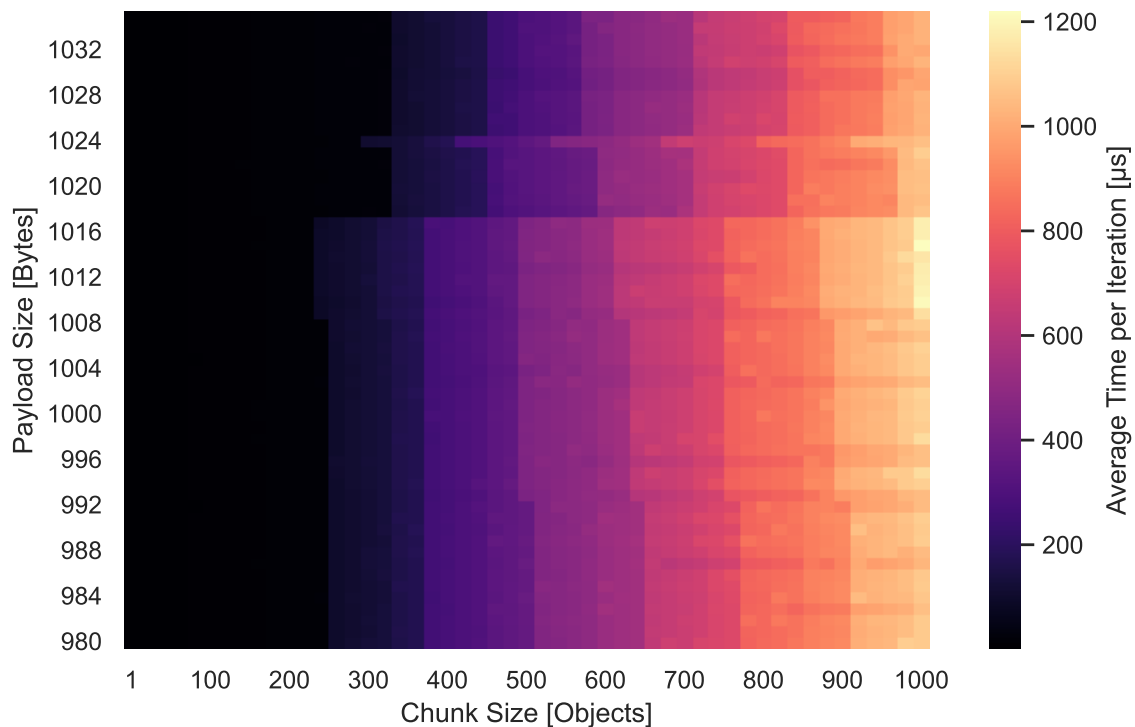


Figure 4.8: Finer sweep at 1 kB of payload, without per-thread cache

**Interpretation** The deciding factor in Scenario 3 is the allocation strategy, namely whether objects are allocated individually or in bulk. The PC architecture allocates and frees each planet separately and thus pays a per-object cost, whereas the AoS, SoA and Bevy architectures allocate in a contiguous buffer and avoid it. This is why PC is an order of magnitude slower, while the differences among the bulk-allocating architectures stay small. Contrary to Scenarios 1 and 2, Bevy no longer outperforms the AoS architecture, as without a compute or cache advantage the fixed overhead of its ECS spawn and despawn functions dominate. The sharp cutoff for PC follows from the per-object allocation, as only individual allocations hit the allocator's size threshold. To be noted is that the exact position at 1032 B is specific to the glibc allocator, whereas the disadvantage of per-object over bulk allocation is general. Running this benchmark with another operating system, and thus likely a different allocator, may yield different results and different threshold effects.

## 4.5 Discussion

The core message to be taken away from the results in the three Scenarios is that a memory layout outperforms the others due to different mechanisms, namely SIMD vectorization, cache locality and allocation strategy. The SoA architecture led throughout and PC trailed in every scenario, while Bevy beat AoS in Scenarios 1 and 2 through better vectorization and omitting irrelevant data, but lost in Scenario 3 once those advantages fell away and its ECS overhead dominated.

As the four architectures demonstrate, a program rarely fully adheres to DOP or OOP, but finds itself on a spectrum between the two. The SoA architecture, as the purest form of DOP, led in performance but requires manually managing several separate vectors, a trade-off between performance and convenience that is a recurring theme in the DOP literature [9]. The object-oriented approaches keep each planet's data together, which is often more convenient to work with, but could not match SoA in performance. Bevy may offer a middle ground, enabling some benefits of DOP while keeping the code closer to an object-like structure.

To conclude, choosing a memory layout should not be based solely on whether it is data-oriented or object-oriented, but on what best fits the requirements. Where performance is critical and the access pattern is regular, a SoA architecture may be preferable, whereas where convenience and maintainability weigh more heavily, an object-oriented layout may be an acceptable trade-off.

## 4.6 Limitations

The results of this evaluation are, to a certain extent, dependent on the environment in which they were measured. As cache sizes and associativity were instrumental for the spikes in Scenario 2 and the register width in Scenario 1, the scaling behavior may differ on processors different to the AMD Ryzen 7 5800X (Zen 3), which was used for all benchmark runs in this thesis. Likewise, the benchmarks were compiled with a single `rustc` version and ran on Linux with the `glibc` allocator. The allocation threshold observed in Scenario 3 is a property of that specific allocator, and both vectorization and allocation may differ under another compiler, allocator, or operating system.

Regarding the benchmark design itself, the results should not be generalized without care. The workload is a synthetic micro-benchmark that attempted to isolate one bottleneck per scenario. While this approach makes the observed effects clearly visible, it does not reflect the complete behavior of real applications where all three effects and more occur together and may interact. Additionally, the payload used to control object size is likewise artificial and the Bevy ECS used for the Bevy architecture represents only one framework rather than the ECS approach in general.

Finally, although measures were undertaken to reduce OS jitter via core pinning, a fixed clock and warmup runs, residual jitter remained visible occasionally, most noticeable in the results of the PC architecture. Thus, the results are best read as trends and orders of magnitude and less as exact values.

# Chapter 5

## Related Work

The impact of DOP principles on execution time has been investigated in prior studies. In 2021, Nyberg [21] tested four aspects of DOP for performance and cache utilization: removing polymorphism, switching from non-contiguous to contiguous structs, reducing struct size and contrasting a cache-friendly OOP layout to an ECS. All four improved performance and lowered miss rates for the data and unified caches, except for the removal of polymorphism, where the cache miss rate rose slightly. His study, however, was written in C++ and toggled these aspects individually, rather than isolating the layout spectrum from PC over AoS to SoA across separate scenarios, as done here.

While Nyberg isolated the individual effects, Faryabi [10] in 2018 examined whether they would materialize in practice. In one of three experiments, he converted the Unity game DwarfHeim to an ECS, though the conversion was not purely DOP. Performance increased in some cases, but Faryabi warned the gains could not be directly attributed to DOP, as too many confounders remained. This thesis, in contrast, removes such confounders by having all architectures run an identical workload, so any difference can be attributed to the memory layout alone.

In 2020, Fedoseev [11] likewise studied DOP in practice, comparing two nearly identical mobile games built with OOP and DOP across performance, maintainability and required skill level. The DOP build's frame rate was roughly 10% higher and maintainability was equal, though code reuse in follow-up projects was higher. On skill level, he noted DOP might require a solid background in computer science, as it demands comprehension of computer architecture and memory. Where Fedoseev weighed two complete games across several dimensions, this thesis instead isolates the execution-time impact of the memory layout itself.

In 2022, Wingqvist et al. [26] compared four builds of a small game: an OOP and a DOP build, each in single- and multi-threaded variants. Testing object counts of 5000 to 100 000 with cache miss rates simulated via Valgrind, the results favoured DOP, reaching a maximum speedup of 13.25 with the multi-threaded variant and a cache miss rate reduction of 5.57. The authors note, however, that a real game seldom features more than 5000 objects, so practical speedups would likely be smaller. Their cache numbers, however, come from simulation and the focus lies on threading, whereas this thesis measures on real hardware via profiling and isolates the memory layout across the four architectures.

While these all focus on the impact of DOP on video games through practical comparisons, none isolate the differences across the full memory layout spectrum of PC, AoS and SoA in Rust, as done in this thesis.

# Chapter 6

## Future Work

**Scenario 1** Bevy’s partial vectorization remains a subject of further inspection as it poses the question whether a SoA-like component layout [17] would allow for full vectorization and not be overshadowed by the increased overhead from component management. Additionally, the processor used for this thesis only allowed for the usage of 256-bit ymm registers. A processor featuring AVX-512 would allow for even broader registers and potentially increase the gap between the data-oriented and object-oriented architectures.

**Scenario 2** The source of the spikes in the execution time of the PC architecture (see Figure 4.2) remains open and could be investigated, as it is not explained by the cache conflict misses relevant for the AoS architecture. Additionally, this thesis only scaled the payload to 20 KiB which primarily surfaced L1 conflict-miss effects. Increasing the payload to higher values may reveal additional effects attributable to the L2 and L3 caches.

**Scenario 3** The tcache cutoff is only one of many thresholds. Thus, other limits such as the fastbin/smallbin boundary or the mmap threshold could be visualized through further scaling the payload. Further, different allocators could be investigated to reveal whether the PC architecture severely underperformed due to the specific allocator used in this thesis or whether it is a general problem with this architecture.

**General** While this thesis isolated effects, a combination of the three scenarios could reveal whether the measured effects interact with one another and, e.g., add up. Additionally, all benchmark runs were performed on a single core. ECS and SoA layouts usually scale differently when using multi-threading. Wingqvist et al. [26] investigated a multi-threaded build and could serve as a further basis.

# Chapter 7

## Conclusion

This work compared the performance of DOP and OOP memory layouts in Rust, featuring four architectures (Bevy, SoA, AoS, PC) benchmarked in three scenarios (Compute-bound, Memory-Bound, Allocation / Deallocation). The data-oriented SoA architecture led in all scenarios, with the PC architecture clearly landing in last place. However, the reason for differences were, depending on the scenario, different mechanisms.

In Scenario 1, vectorization primarily drove execution times, in Scenario 2, cache locality was of the most importance, and in Scenario 3, the allocation strategy was the deciding factor. Bevy was able to beat the AoS architecture in Scenario 1 and 2 but lost in Scenario 3, likely due to the ECS overhead without the advantages of vectorization and cache locality.

To conclude, the memory layout decides on performance. However, the bottleneck decides which memory layout is optimal. DOP and OOP are neither one-size-fits-all solutions, but span a spectrum. Depending on the task at hand and present bottlenecks, a memory layout best suited for these should be chosen.

# Bibliography

- [1] Advanced Micro Devices. 2026. *Amd  $\mu$ Prof*. Version 5.3. Advanced Micro Devices Inc. Retrieved 07/30/2026 from <https://www.amd.com/de/developer/uprof.html>.
- [2] Tony Albrecht. 2009. Pitfalls of Object Oriented Programming. Presented at Game Connect: Asia Pacific (GCAP) 2009. Sony Computer Entertainment Europe, Research & Development Division, (2009). Retrieved 07/27/2026 from <https://www.gamedevs.org/uploads/pitfalls-of-object-oriented-programming.pdf>.
- [3] Bevy Project Contributors. 2026. Bevy Engine. A data-driven game engine built in Rust. (2026). Retrieved 08/17/2026 from <https://bevy.org/>.
- [4] Bevy Project Contributors. 2026. *Bevy Quick Start Guide: ECS*. Bevy Engine. Retrieved 07/27/2026 from <https://bevy.org/learn/quick-start/getting-started/ecs/>.
- [5] Bevy Project Contributors. 2026. *bevy::ecs::storage::Table* — *Bevy API Documentation*. Bevy Engine. Retrieved 07/27/2026 from <https://docs.rs/bevy/0.19.0/bevy/ecs/storage/struct.Table.html>.
- [6] Scott Bilas. 2002. A Data-Driven Game Object System. Presented at the Game Developers Conference (GDC) 2002. Gas Powered Games, (2002). Retrieved 03/11/2026 from <https://www.gamedevs.org/uploads/data-driven-game-object-system.pdf>.
- [7] D. J. Delorie. 2017. Malloc Internals and You. Red Hat Developer. Retrieved 08/13/2026 from <https://developers.redhat.com/blog/2017/03/02/malloc-internals-and-you>.
- [8] Ulrich Drepper. 2007. What Every Programmer Should Know About Memory. Technical report. Red Hat, Inc., (November 2007). Retrieved 03/06/2026 from <https://people.freebsd.org/~lstewart/articles/cpumemory.pdf>.
- [9] Richard Fabian. 2018. *Data-oriented Design: Software Engineering for Limited Resources and Short Schedules*. Cited from the free on-line edition. Richard Fabian. ISBN: 978-1-916478-70-1. Retrieved 07/27/2026 from <https://dataorienteddesign.com/dodbook.pdf>.

- [10] Walid Faryabi. 2018. *Data-Oriented Design Approach for Processor Intensive Games*. Master's thesis. Norwegian University of Science and Technology (NTNU). <https://ntnuopen.ntnu.no/ntnu-xmloi/handle/11250/2575669>.
- [11] Kirill Fedoseev, Nursultan Askarbekuly, Ekaterina Uzbekova, and Manuel Mazzara. 2020. Application of Data-Oriented Design in Game Development. *Journal of Physics: Conference Series*. DOI: 10.1088/1742-6596/1694/1/012035.
- [12] Cameron Hart. 2026. *glam: Vec3A — API Documentation*. Retrieved 07/27/2026 from <https://docs.rs/glam/0.32.1/glam/f32/struct.Vec3A.html>.
- [13] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach*. (5th ed.). Morgan Kaufmann, Waltham, MA. ISBN: 978-0-12-383872-8.
- [14] Hex-Rays SA. 2026. *IDA Free. Version 9.4*. Hex-Rays. Retrieved 07/29/2026 from <https://hex-rays.com/ida-free>.
- [15] Mark Donald Hill. 1987. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph.D. thesis. University of California, Berkeley, (November 1987). Technical Report UCB/CSD 87/381.
- [16] Intel Corporation. 2026. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. Technical report 253665-U92US. Intel Corporation. Retrieved 08/17/2026 from <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [17] Jenya705. 2026. Contiguous access (Query::contiguous\_iter\_mut). Bevy Engine. Retrieved 08/07/2026 from <https://github.com/bevyengine/bevy/pull/21984>.
- [18] Steve Klabnik and Carol Nichols. 2023. *The Rust Programming Language*. (2nd ed.). Cited from the free online edition. No Starch Press, San Francisco, CA. ISBN: 978-1-7185-0310-6. Retrieved 03/07/2026 from <https://doc.rust-lang.org/book/>.
- [19] Daniel Kleppner and Robert J. Kolenkow. 2014. *An Introduction to Mechanics*. (2nd ed.). Cambridge University Press, Cambridge. ISBN: 978-0-521-19811-0. Retrieved 08/06/2026 from <https://archive.org/details/KleppnerD.KolenkowR.J.IntroductionToMechanics2014>.
- [20] Kevin Knapp and clap Contributors. 2026. *clap — Command Line Argument Parser for Rust*. Retrieved 07/22/2026 from <https://docs.rs/clap/4.6.0/clap/>.

- [21] Frank Nyberg. 2021. *Investigating the Effect of Implementing Data-Oriented Design Principles on Performance and Cache Utilization*. Bachelor's thesis. Umeå University.
- [22] cargo-show-asm Contributors pacac. 2026. *cargo-show-asm — Display Assembly Generated for Rust Source Code*. Version 0.2.57. Retrieved 07/29/2026 from <https://docs.rs/cargo-show-asm/0.2.57>.
- [23] Patrick Redmond, Jonathan Castello, José Manuel Calderón Trilla, and Lindsey Kuper. 2025. Exploring the Theory and Practice of Concurrency in the Entity-Component-System Pattern. *Proceedings of the ACM on Programming Languages*, 9, OOPSLA2, 1–28. DOI: 10.1145/3763050.
- [24] The GNU C Library. [n. d.] *Memory Allocation Tunables*. The GNU C Library Reference Manual. Free Software Foundation. Retrieved 08/13/2026 from [https://sourceware.org/glibc/manual/latest/html\\_mono/libc.html#Memory-Allocation-Tunables](https://sourceware.org/glibc/manual/latest/html_mono/libc.html#Memory-Allocation-Tunables).
- [25] Rui Ueyama. 2026. mold: A Modern Linker. Accessed: 2026-09-13. (2026). <https://github.com/rui314/mold>.
- [26] David Wingqvist, Filip Wickström, and Suejb Memeti. 2022. Evaluating the Performance of Object-Oriented and Data-Oriented Design with Multi-Threading in Game Development. In *2022 IEEE Games, Entertainment, Media Conference (GEM)*. IEEE. DOI: 10.1109/GEM56474.2022.10017610.
- [27] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23, 1, 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588.

# Listings

|     |                                                                                                                                                                               |    |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.1 | Update loop of the AoS and PC architectures . . . . .                                                                                                                         | 13 |
| 3.2 | Update loop of the SoA architecture . . . . .                                                                                                                                 | 14 |
| 3.3 | Update loop of the Bevy architecture with contiguous component slices . . . . .                                                                                               | 14 |
| 3.4 | Update loop of all four architectures in Scenario 2 . . . . .                                                                                                                 | 16 |
| 3.5 | Deallocation of planets in PC and AoS architectures . . . . .                                                                                                                 | 18 |
| 3.6 | A benchmark invocation for Scenario 3 via the command line                                                                                                                    | 20 |
|     |                                                                                                                                                                               |    |
| 4.1 | ASM arithmetic, SoA versus AoS. In line 1, r10 is the base address of the pos_x array and rax the element index. 4*rax turns it into a byte offset, an f32 being 4 B. . . . . | 24 |
| 4.2 | ASM Load sequence: AoS versus PC . . . . .                                                                                                                                    | 25 |
| 4.3 | ASM of pop(): AoS reuse versus drop (top), PC (bottom) . . .                                                                                                                  | 33 |

# List of Figures

|     |                                                                                                 |    |
|-----|-------------------------------------------------------------------------------------------------|----|
| 2.1 | Conceptual Planet struct (left) and the corresponding physical memory layouts (right) . . . . . | 4  |
| 4.1 | Average time per iteration versus planet count . . . . .                                        | 23 |
| 4.2 | Average time per iteration versus payload size . . . . .                                        | 27 |
| 4.3 | Pattern in execution time of PC and AoS architectures . . . . .                                 | 27 |
| 4.4 | Spike position in relation to offset to struct size . . . . .                                   | 28 |
| 4.5 | Reuse Case with a chunk size of 500 . . . . .                                                   | 32 |
| 4.6 | Drop Case with a chunk size of 500 . . . . .                                                    | 32 |
| 4.7 | Finer sweep at 1 kB of payload . . . . .                                                        | 34 |
| 4.8 | Finer sweep at 1 kB of payload, without per-thread cache . . . . .                              | 35 |

# List of Tables

- 3.1 CLI arguments for configuring the benchmark executable . . 20
- 4.1 Per-scenario measurement parameters. . . . . 22
- 4.2 CPI and L1 data cache miss ratio per architecture. Payload pairs mark the spikes at 1024 B, 2048 B and 4096 B: \* is the middle of the spike itself, + is just past it. . . . . 29
- 4.3 Cache set touched by planets as a function of struct size . . . 30

# List of Acronyms

**AoS** Array of Structures

**CLI** Command Line Interface

**CPI** Cycles per Instruction

**CSV** Comma-Separated Values

**DOP** Data-oriented Programming

**ECS** Entity Component System

**GPR** General Purpose Register

**OOP** Object-oriented Programming

**PC** Pointer Chasing

**SIMD** Single Instruction, Multiple Data

**SMT** Simultaneous Multithreading

**SoA** Structure of Arrays