MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Game Engineering and Simulation Technology

Using LLVM/Clang to to maintain the abstraction level of Object Oriented Programming, yet abide SOA rules

How compiler technology could possibly make a link between conflicting programming paradigms

By: Julian Müller, BSc.

Student Number: 1610585015

Supervisor: Stefan Reinalter, DI

Second Supervisor: Prof. Alexander Hofmann, DI

Wien, January 28, 2019

FH University of Applied Sciences TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Wien, January 28, 2019 Signature

Abstract

This work will prospect the possibility of using compiler technology as a mediator between the conflicting programming paradigms *OOP* and *SOA*.
While Object oriented programming is often praised for its benefits on abstraction and maintainability, it encourages programmers to design inefficient datalayouts. Specifically in game engineering, where performance is a constitutive factor for a product's success, data oriented solutions are on the rise. While it is debated, wether or not performant data layouts inevitably entail challenging maintenance, surely the base concepts of *objects* are well observable in a game. Therefore this will be an attempt to make object oriented programming a valid option for ever rising demands on performance.
A prototypical implementation of a source-to-source transformation tool called *COOP* (**C**ache friendly **O**bject **O**riented **P**rogramming), developed in the Clang tooling environment, will determine wether or not compiler technology can be used to achieve a performance optimization on a classically OOP abidant target source code base.

Keywords: Object Oriented Programming, OOP, Structure Of Arrays, SOA, Data Oriented Design, Compiler, LLVM, Clang

Acknowledgements

TODO dankschee

Contents

1 Conflicting Paradigms

Numerous programming paradigms exist for even more general programming languages. Each of them come with different perks and offer different perspectives for a problem. Different programming paradigms, however don't necessarily exclude each other. Languages like *Lisp* or *Scala* for example both combine functional and object oriented programming.
This thesis will specifically concentrate on *Object Oriented Programming* (OOP) and *Data Oriented Design* (DoD). Whether or not data oriented design is even to be considered a programming paradigm is debatable [13, p. 1], however its fundamental ideas (specifically concerning data layout) conflict with those of existing paradigms like OOP, so for the sake of consistency in this manner of comparison, we will call it that way.
Depending on the domain, certain programmers will have different answers to the question which of both should be preferred. Each party will make compelling arguments to why their choice is mandatory. This is because those paradigms (partially) solve different problems and therefore offer dissenting perspectives on problems and their respective solutions for them. To understand why *Object Oriented Programming* and *Data Oriented Design* are rather cannibalistic to each other, we need to first have a look to what they are trying to solve, independently.

1.1 Object Oriented Programming / AOS

Starting with punch cards, each iteration of new programming generations provided new forms of abstraction for programmers, be it control flow statements, type systems, data structures like native arrays. When machine code started to simplify operations, FORTRAN partly introduced portability as early as 1957; LISP introduced symbolic processing and automated memory management until finally Simula/67 introduced objects in the 1960s [21][28]. Object Oriented Programming could not be called popular until the 1970s or 1980s, when Stroustrup created C++. Originally OOP was meant to be the way to go for creating graphics based applications [17]. This makes sense, since tree like data structures (GUIs), containing entities with shared behavior can easily be described through polymorphism.
OOP shines, whenever the problem to be modeled can be abstracted to one or more base classes, that define shared state and/or behavior. Even though some languages allow for a subclass to "*exclude variables and/or methods which would otherwise have been inherited from the superclass(es)*" [25, p. 4].
OOP quickly established and rooted itself in the industry without solely being used on graphics applications. This is due to the fact, that it represents a world model, we are taught since

elementary school. We are familiar with *is-a* relations ever since we learned that despite dissenting traits, both Labradors and Pugs are dogs. Arguably abstraction is one of - if not - the most important disciplines in programming [14, p. 5]. Since programs oftentimes try to model real life information, OOP delivers an easy to grasp, quick-to-learn approach to do so. That is also the reason, why there are so many OOP programmers around the world. Without trying to evaluate whether or not OOP's way of abstraction is superior to DoD, it is undoubtedly the more prominent one, especially for virtually any other profession than a programmer/computer scientist (see Fig. 1). This is however where OOP and modern computer architectures don't get

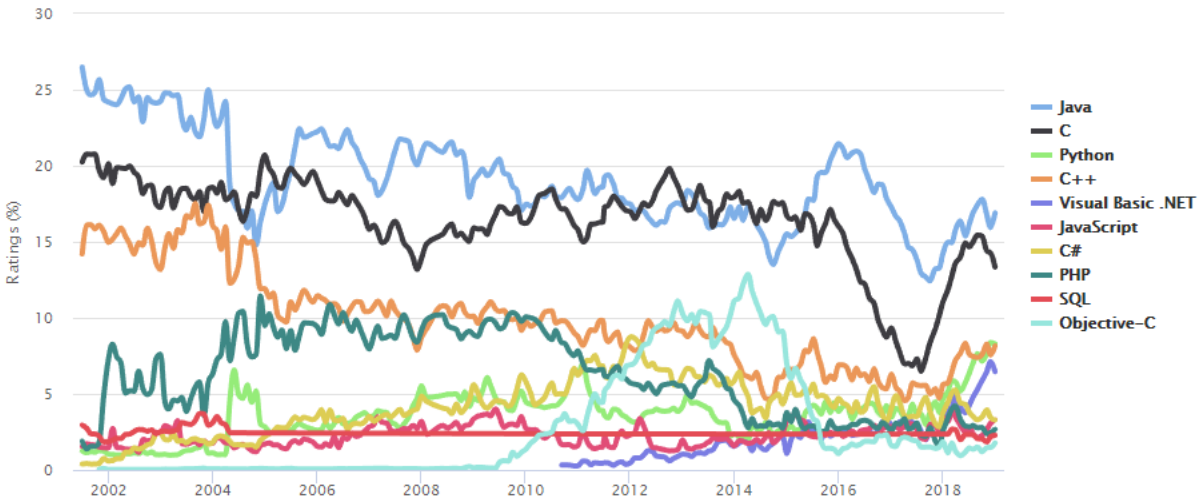

Figure 1: Most popular programming languages throughout the years (Source: [2]).

along, so to say. OOP ovvers an elegant way to intuitively model an issue into code, but doing so it encourages us to implement our data in an inefficient way. Inefficient because creating monolithic models of our data usually lack cohesion, which means, that the classes' members tend to not be related/dependent [4] - at least in terms of computational order.
In a home office application, juggling a few dozens of entities every other minute, this will not appear as a problem. And in this case it might be preferable to program such application in a strict object oriented way, since the development can be done fast and reliably even by a novice. On the other hand and especially in the game development industry OOP has proven to result in poorly performing software, due to inefficient data layouts. Because especially for games: data is performance [27, p. 272].
The abbreviation *AOS* stands for *Array Of Structures*[10] and it describes, what usually happens with Object Oriented Programming.

```
struct Obj {
   float xyz[3];
   float vel[3];
};

struct Human : Obj {
   char *name;
   int age;
};

struct NPC : Human {
   int mood;
};

NPC npc_arr[3];
```

Code 1: Example of some hierarchical POD class definitions

Considering the rather arbitrary C++ class definitions in Code 1 the *npc_arr* will occupy memory according to Fig. 2 (disregarding any *padding*). This is quite literally an array of structures -

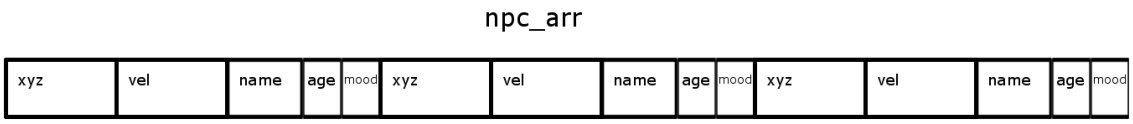

Figure 2: Visualization of how a *npc_arr* will exist in memory

hence the notation *AOS*. The following sections 1.2 and 1.3 will elaborate on *why* exactly this way of thinking/way of abstraction is inefficient.

1.2 CPU Caches and why they don't *fit* objects

The data layouts typically found when programming with objects and classes, are not inefficient because they lack logic. "*Each human - including students - will have positional traits*", is semantically correct. In fact it seems rather unfortunate, that modern computer architectures can't deal well with an abstraction that fits our perception of the world. But the hardware is certainly not to blame here. The problem with a monolithic class definition is much more that of common coding- or data access patterns.

1.2.1 Common data access patterns vs. Monolithic class definitions

Numerous coding best practices teach us to write simple, modular code.

> *Functions should do one thing. They should do it well. They should do it only.* [23, p. 35]
>
> *Keep it simple and smart* (KISS principle). [26, p. 77]
>
> *[...] cohesion is an important object-oriented software quality attribute.*[4]

Just like we want our class definitions to share a common responsibility or task, we want the set of instructions that iterate and probably transform a set of data to be as simple and modular as possible. So usually we try to not write monolithic *for-loops* handling every single aspect of a set of data.
For example in Fig. 2 we would not want a loop that handles each and every single member of an NPC. This would not only result in a big set of instructions, that hide the individual purposes of each expression, but also make it hard to maintain/change the code. Not to mention, that

different data often demands change at different times at runtime. Requirements can change quickly. Breaking up responsibilities that were coupled and forced to coexist change not so quickly.[23]
Exemplary if Code 1 was the model for a game, our game loop could at one point iterate over all the elements in the *npc_arr* to update their position and velocity for each frame. The *NPC's mood* could just as well be updated frequently in a separate function, that only encompasses the information relevant for the calculation of the updated mood. Their *Human::name*s however will most likely not change so frequently - if ever - so the instructions to modify that data will most likely depend on user input and exist in yet a whole other routine. This modularization of code is commonly referred to as *Separation of Concern* and has proven to improve the code's maintainability [19, p. 85]. *This* keeping the objects in some sort of set, then iterating over it for each routine, that manages a subset of the object's data, is a common access pattern that is applied on objects in OOP.
The interim conclusion here is, that even only for maintainability reasons, it is desirable for programmers to process logically related subsets of their data separately - but then why is the resulting software so slow compared to the same idea implemented with a *Data oriented Design*?
The Object Oriented Programming paradigm is exactly doing what it promises - providing a sort of abstraction, that programmers can intuitively apply to their problem definition. Consequently OOP programmers quickly adapt the habit of developing against their abstraction because it is intuitive. What is lost in the process is the concern of developing against the rationale and thinking about how it interacts with the hardware. *This* is probably the fundamental difference between OOP and DoD.
So whats our hardware's deal? Why do objects don't get along with it. Why can't we have super high speed machinery, that makes hardware concerns obsolete? Why can't we have anything nice?

1.2.2 A brief history of memory

To answer Section 1.2.1's concluding question: We do have high speed memory units at hand we just can't afford them. Modern computer systems rely on a variety of different memory units each differing in access latency and other properties. The intention behind this complex hierarchy of memory layers is obviously speed and is the result of an evolving cost-benefit calculation.

> *The task the computer designer faces is [...] design a computer to maximize performance while staying within cost[...].*[16, p. 8]

Originally

> *memory access latency and instruction execution latency were on roughly equal footing. [...] register-based instructions could execute in two to four cycles, and a main memory access also took roughly four cycles.*[15, p. 189]

This proportion changed significantly. While it is relatively cheap to produce high speed CPUs the same is not true for memory units. So whats happening, is that today's PCs/consoles are equipped with CPUs that are way faster than the greater parts of their available memory units. Due to increasing tick rates and *Instruction Pipelining* what used to be four cycle RAM reads are now several hundred cycles. Not Because RAM became slower - the opposite is the case - but because CPUs became that much faster in relation. This trend was thoroughly observed and documented by John L. Hennessy and David A. Patterson (see Fig. 3).

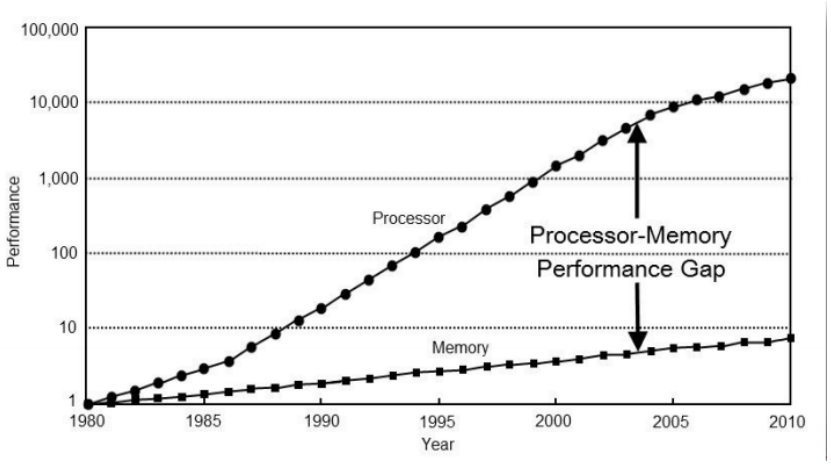

Figure 3: "***Starting with 1980 performance as a baseline, the gap in performance between memory and processors is plotted over time****. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 5.13 on page 313). The processor line assumes a 1.25 improvement per year until 1986, and a 1.52 improvement until 2004, and a 1.20 improvement thereafter*" (Source: [16, p. 289])

To solve the issue of ever diverging CPU/memory performances (commonly referred to as the *memory gap [15]*), specifically to reduce the latency of references to main memory, smaller but significantly faster (and more expensive) memory units are placed between the CPU and the main memory. These modules are called *Caches* - first named by C.J. Conti [9] in 1969. Originally cache technology was mentioned as *buffers*[11]. Not considering their complex, modern modalities and policies this is a fitting notation.

The basic idea behind a fast buffer interconnected between the CPU and the main memory is to create local copies of referenced data-chunks, in order to provide faster access on subsequent calls to the same *AU* (Adressable Unit) as well as the ones deemed likely to be accessed soon [15, p. 191]. This principle originated in the work on *Virtual Memory* [11, p. 15] and is today much more sophisticated.

So we actually do use high speed memory in our common computer architecture, we just don't have lots of it.

1.2.3 Cache modules and types

In today's PCs/consoles typically each CPU core has its own hierarchy of cache modules (see Fig. 4). Closest to the core (on-chip) is the *L1* (Level 1) cache. Accessing an AU in the Intel®

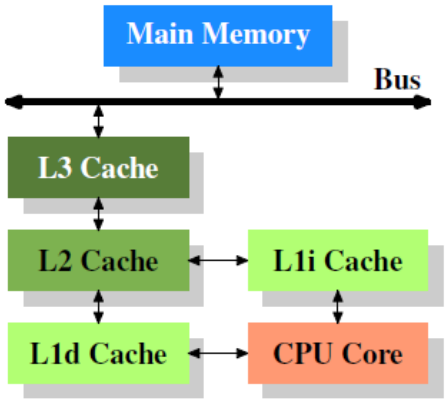

Figure 4: Exemplary, simplified model of a CPU core and its several cache modules (Source: [12, p. 15])

Core™ i7 Processor's L1 cache for example is almost as fast as accessing it in the very CPU's register - 4 cycles (2.1 to 1.2 ns) [20]. For reference access to main RAM "*can take on the order of 500 cycles[...]*" [15, p. 189]. Another cost unrelated reason, why we don't have lots of *L1 D* cache is that more memory means literally more physical space is occupied. Having more cache memory equals more *cache hits* (see Section 1.2.4) but as soon as the cache won't fit on-chip anymore, there is yet again additional latency. Thats why the L1 and sometimes L2 cache modules are kept comparably small but on-chip. Other L3 and L4 caches are each bigger and slower than their preceding counterparts respectively, but for the most part share the same ideas slowly converging latency times to the common main memory RAM.
CPU cores can share one or more cache modules (usually starting with L2 or L3), effectively accessing the same local copies of data. This entails synchronization issues, that will be mentioned in the context of this work later on (TODO).
Modern cache hierarchies include data caches as well as instruction caches, usually both in an L1 cache. However there are different takes on how to implement this. Harvey G. Cragon lists [11, p. 17]:

- instruction cache - holds only instructions
- data cache - holds only the data stream
- unified cache - holds both instructions and data
- split cache - two-cache system, one for instructions and one for data

The scope of this masters thesis will omit instruction-cache related subjects, because the upcoming attempts and techniques to achieve performance optimizations will focus on improving the data layout of a given target source code.

1.2.4 The CPUs cache utilization

A programmer will rarely ever directly interact with a cache module (though there are mechanisms for manual prefetching/clearing). The underlying idea for it was to be transparent to the programmer. However understanding the CPUs cache utilization enables one to tailor the data layout to it, resulting in faster access, less waiting and consequently higher throughput. Among other things, this is what *Data oriented Design* aims to do - developing against hardware concerns.
As mentioned before the basic idea of the cache is to provide local copies of data at a faster rate and prefetch data segments, that are likely to be used soon. This works by directing each main memory access to go through the cache. Main memory access means also going through virtual address translation, address and data buses and depending on the main memory, crossbar switching logic [15, p. 190]. Whenever the CPU requests access to a certain AU before saddling the horses and going on a journey to main memory, the cache will check whether or not the requested AU is currently present inside its buffer. If so it is referred to as a *cache hit*, otherwise a *cache miss*. For a modern L1 D cache this buffer, to be more specific, consists of several cache lines usually each of 64 bytes. Overall cache- and line sizes vary between different architectures but are standardized mostly due to Intel's designs.
Cache misses result in higher access latency and should be avoided if possible. It is however not always possible. *Mark Donald Hill* describes three classifications for cache misses:

- Compulsory Miss - Access to previously unreferenced data blocks. The very first data access will inevitably result in a main memory access.
- Conflict Miss - Due to data blocks mapping to the same cache-lines.
- Capacity Miss - Due to the cache's finite capacity. Lots of data access will eventually displace older/less-used entries in the cache (depending on Replacement policies).

The simplest and least efficient implementation to determine a cache *hit/miss* is to iterate each cache line comparing fitting criteria. To prevent this nowadays caches implement a certain associativity technique. This way each individual physical main memory address can be mapped to one or more specific cache lines. Doing so addresses are converted to and managed by metadata consisting of: [15, p. 193]

- Offset - Offset to the actual referenced Byte inside the cache line.
- Cache Line Index - Which cache line/s would hold the AU.
- Tag - Which cache sized block in main memory holds the original data.

In the case that each physical main memory address, has exactly one counterpart it is called a *direct-mapped* cache [15, p. 194]. In this case since the cache can by far not encompass the whole extend of the main memory, a lot of physical addresses will be mapped to the very same cache line, effectively extruding each other out of the cache when accessed. This is

called *eviction* and in unlucky cases will behave like all references are cache misses [11, p. 97]. To prevent this modern caches map a physical address to *n* cache lines. This is called *n-way associativity* - typically implemented as *8-way* or *16-way* caches depending on the level.
There is a lot more to cover about caching technologies and policies like: Replacement-/Write-/Coherency(MESI; MOESI; MESIF) policies and for further reading Harvey G. Cragon's *Memory Systems and Pipelined Processors*[11] as well as Jason Gregory's *Game Engine Architecture*[15] are highly recommended reads. However for the purpose of this work a few specifics are most interesting for us.
How can a data layout affect *hit ratios* and reduce calls to main memory? As mentioned before there are common data access patterns in software and the caches actually accommodate us by adapting their builtin prefetching mechanisms to it, following a set of *locality concepts*. Harvey G. Cragon counts three of those concepts: [11, p. 16]

- *Temporal locality* - Information recently referenced by a program is likely to be used again soon.
- *Spatial locality* - Portions of the address space near the current locus of reference are likely to be referenced in the near future.
- *Sequential locality* - A special case of spatial locality in which the address of the next will be the immediate successor of the present one.

At first glance these concepts are very straight forward, but their respective implementations for automatic hardware prefetching can be more complex than one would think. Prefetching basically means:

> "*[...] bringing data in the data (or mixed instruction-data) cache before it is directly accessed by a memory instruction [...].*" [6, p. 610]

There are however different strategies to decide which bytes should be faithfully loaded into the cache. Tien-Fu Chen and Jean-Loup Baer list two categories for prefetching strategies: [6, p. 610]

- *Spatial* - access to current block is basis for prefetching.
- *Temporal* - lookahead decoding of instruction stream is implied.

There are simple approaches like: whenever block *i* is accessed, prefetch block *i*+1, called the *One Block Lookahead*; stride based approaches storing previously referenced addresses in a table and calculating a stride based on current and previous addresses; combinations of both and many more [6]. Sometimes data is prefetched into the cache, sometimes into separate stream buffers. There are many different hardware prefetching methods to find and while data cache prefetching is considered to be more challanging [24, p. 4] it is still best practice to rely on spatial locality when modeling data to play into the cache's hand.
Compilers already make use of software prefetching (manual cache interaction instructions) in certain cases.

> *For array-based applications, the compiler can use locality analysis to predict which dynamic references to prefetch, and loop splitting and software pipelining to schedule prefetches.* [22, p. 223]

So we can already deduce that for an efficient data layout it would be beneficial to rely on arrays, or more generally, concepts compilers can 'comprehend' and optimize. Also even though the concept of *sequential locality* is only a special case, it is the one we can utilize best to derive adaptations for our data layout, since hardware prefetching has adapted best to it. How exactly can we convert this information into a generic set of rules/best practices, a methodology for efficiency? We don't it already exists.

1.3 Data Oriented Design / SOA

The whole purpose of adding abstraction layers is moving further away from the hardware mentally. This helps us to focus on constructing appropriate models for a problem [18, p. 5]. However disregarding intrinsic detail is predestined to result in poor resource utilization at that level. *Data oriented Design* wants us to be fully aware of what is beneath the source code and tailor the essential resources to it. Essential resources here being our data.

> *The data-oriented design approach doesn't build the real-world problem into the code. This could be seen as a failing [...] by veteran object-oriented developers [...]. [It] gives up some of the human readability [...], but stops the machine from having to handle human concepts [...].*[13, p. 7]

1.3.1 OOP and bad abstraction

DoD however deserves more credit than only being a performance optimization. (Wrong) abstraction not only moves us away further from the hardware, but also from the actual problem domain.
Even though *coupling* and its avoidance are a big deal in OOP it somehow seems natural to couple the data and the problem domain. This coupling has proven to lead to the exact same problems as coupling of unrelated classes. Its hard to make changes, especially when the reason for change is modification of fundamental design choices. But not only is contextual linkage rigid to conceptual change but its also hard to operate on it only in ways that haven't been thought of initially.
Imagine for example a simple particle system implemented in a typically OOP manner (see Code 2).

```
struct particle
{
   float ms_alive, lifetime_in_ms;
   float xyz[3];
```

```
    Shader *shader;
};

struct particle_system
{
    particle particles[1024];
    unsigned particles_alive;
    ...
};
```

Code 2: OOP typical particle system implementation

At first glance it appears to be a perfectly valid approach to make a particle system linked directly to the very particles it is supposed to manage. Each method will now operate on the particles array and just like that we can make numerous particle systems, each implicitly operating only on its own data. This is an easy to grasp concept that in terms of a particle system we could easily match with a real-world fireworks battery. While there are a few things, that can be nagged about, concerning only maintainability there is already a huge issue.
What about operations, that need to be applied to ALL particles? ... Why should we? When in the real world do two physically different particle systems ever interact with each other? Collision might come to mind. While it would be a lot easier to just iterate a big list of particles, we don't really care. It is still relatively easy to just check each particle system's particles against another particle system (see Code 3). And we will just do this for all the particle systems we have! They probably should exist somewhere in the same context anyway.

```
1 void particle_system::particle_system_collision(
2     particle_system *ps1,
3     particle_system *ps2)
4 {
5     for(unsigned i = 0; i < ps1->particles_alive; ++i)
6     {
7         particle *p1 = ps1->particles[i];
8         for(unsigned o = (ps1 == ps2 ? i+1 : 0); o < ps2->particles_alive; ++o)
9         {
10            particle *p2 = ps2->particles[o];
11            auto collision_data = sphere_collision(p1->m_xyz, p1->m_radius,
                  p2->m_xyz, p2->m_radius);
12            if(collision_data.colliding)
13            {
14                collide(*p1, *p2, collision_data);
15            }
16        }
17    }
18 }
```

Code 3: Example code how OOP could handle collision between different particle systems' particles

Even though Code 3 has some uncomfortable specifics (especially line 8), it is still doable. The moment we start to render this particle system however we might notice that our particles look weird. Our particles' sprites make use of transparency and even though we enabled transparency in our render back-end it doesn't look right. Well the particles need to be rendered in a certain order. We have not thought of that while designing our data model. Its easy enough to sort one particle system's particles according to their distance to camera respectively, but bringing EACH particle system's particles in order?
Well we could write an algorithm, that has a list of particle arrays; takes each particle system's particle array; unifies them to a big dynamically allocated array; then sorts it; then returns that array so it can be rendered. Each frame. While we are at it, we will need to change our render pass, since now we don't render each particle system, but one big array of particles.
This is starting to get ugly really fast. We never do this stuff for a real-world fireworks battery and we feel betrayed by the abstraction we made. Ultimately we come to a point where we understand, that a real-world projection of a problem domain into source code might not be the best choice. Its definitely possible, but just doesn't *fit* right. So instead of realizing overly complex solutions to make our code fit our abstraction we realize that our abstraction is obsolete and we should focus on a solution that fits our tasks. In the end data is not generic in the way it is used [13, p. 15]. Different jobs might require different solutions and OOP might be optimal for a subset of problems, but has evolved into a pattern, that is forced on domains as a go to.
Besides additional layers of abstraction do not equal superior usability/readability either. When using a third party library we hope the API designer provides functions that operate on data types we are already familiar with, like char pointers, instead of forcing us to learn how to use that developers self-made string class (KISS).

So arguably Object Oriented Programming is not automatically superior, neither in terms of maintainability nor performance. Actually it appears to be inferior to DoD. What does DoD do to be harder/better/faster/stronger?

1.3.2 Normalization of Data

When thinking in terms of the problem and designing against the data *Richard Fabian* references relational database models as a worthy category to look up to, in his book *Data-Oriented-Design: Software engineering for limited resources and short schedules*. After all a database should handle data correctly. Fabian says:

> *The structure of any data is a trade-off between performance, readability, maintenance, future proofing, extendibility, and reuse.* [13, p. 28]

We have already seen, that the object typical data-layout makes sense in terms of readability, but fails to be maintainable and in terms of cache utilization behaves poorly. In order to peel off of the OOP abstraction pattern we want our model to be a minimal, precise representation, that abides a format, that has proven to behave well. Therefore we attempt to treat our data

particle_id	**shader_id**	ms_alive	lifetime_in_ms	x	y	z
p0	s0	7034	10000	4.0	10.1	-12.9
p1	s0	122	10000	0.5	-8.1	-2.6
p2	s0	4310	10000	1.1	7.4	-3.0
...						
p1024	s1	86	100	7.0	0.0	0.5
p1025	s1	44	100	3.0	0.0	0.2
...						

system_id	particles_alive
ps0	462
ps1	99

system_id	**particle_id**
ps0	p0
ps0	p1
ps0	p2
...	
ps1	p1024
ps1	p1025
...	

Table 1: Example excerpt of how the simple particle system could be normalized

modeling process the way we would, when setting up a relational database. We normalize our data, according to the existing normalization stages developed by Edgar F. Codd and Raymond F. Boyce [8]. This is certainly not done because relational databases are famous for their excellent performance (they are not), but because it has proven to give advantageous insights to the data, its domain and the relations between its subsets. Dealing with the problem's domain and the underlying data eventually led to technologies like JPEG and MP3 [13, p. 47]. Understanding the data will lead to an understanding of the solution.
For our minimal example of a particle system, even just the first stages of normalization teaches us, that there should be no arrays of data in one element [13, p. 37] it also forces us to reconsider which data is accessed through which instance (key) and therefore whether or not there is a *relation* to find (see Table 1).

We will not implement a relational database for our application but generating this model can help us realizing some things.

1. particles are no longer represented block-wise inside a particle system, but are held in ONE unified table to begin with.

2. There is tons of redundancy. Not logical but contextual. Seeing the same *shader_id* and *lifetime_in_ms* over and over for each particle that will share a particle system certainly *smells* funny. It is very important to be able to link each particle to a specific shader, but contextually this is redundant. We can see, that some things might have been coupled due to our abstraction.

3. Instead of holding an array of particles we can just as well take its data subsets and link them against a unique key, which basically translates to each column is an array and each key is an index.

4. Rethinking which data belongs to which entity hints at *when* specific subsets are accessed and in which context (which key is used to get it).

1.3.3 Structure of Arrays

In contrast to the previously mentioned AOS (see Section 1.1) a *Structure of Arrays* is the *column-oriented database* pendant to a data layout that results in higher throughput by providing a cache friendly structure [13, p. 163]. We already know, that the hardware, specifically the cache uses certain strategies to prefetch data out of main memory (see Section 1.2.4). Now when implementing the column-oriented approach that comes with data normalization we automatically almost perfectly conform to those hardware prefetching mechanisms. We do so by implementing columns as arrays. The original concept of an *object* can now be simplified to an index, that retrieves the respective values in each column/array. Remembering Code 1, when examining the NPC class we know that after derivation (done by the compiler) it looks like Code 4.

```
1 struct NPC
2 {
3    //inherited from Obj
4    float xyz[3];
5    float vel[3];
6    //inherited from Human
7    char *name;
8    int age;
9    //NPC
10   int mood;
11 };
```

Code 4: NPC pod after derivation is done

Accessing subsets of an instance of NPC will always result in loading at least a whole cache line. Lets assume, that the game code will iterate over the NPCs to update their positions by adding their velocity and some delta-time to their current xyz. Modern architectures mostly rely on cache lines of 64 Byte and assuming that *sizeof(NPC)* is 40 Byte we can make the following assumptions. Considering the subset of data we actually want to work with on our update routine per NPC is (2 * 3 * *sizeof(float)*) 24 Byte, we waste 40 - 24 = 16Byte of our cache-line per NPC. If we had 1000 NPCs instead of only 3 to actually compute each NPCs updated position we would load 40.000 Byte. 16.000 Byte of that would be ignored in total. The smallest common multiple of our 40 Byte NPC and our 64 Byte cache-line is 320, so over eight NPCs worth of cache-line we can count the amount of cache misses produced and extrapolate to greater numbers of NPCs (see Fig. 5). For the purpose of simplicity and demonstration we will assume that the cache will only ever prefetch to the extend of loading the complete cache-line. In reality depending on the strategy any number of lines could be prefetched asynchronously. A cache miss occurs whenever we do not find an AU inside a cache line in terms of our visual representation in Fig. 5 this means, for each cache line we can count the first occurrence of a blue block that is at least intersecting the cache-line. While *NPC0* completely fits inside the first cache-line, *NPC1* does not. However *NPC1*'s relevant data also completely fits in the first cache-line, so accessing it's relevant data will not result in a cache

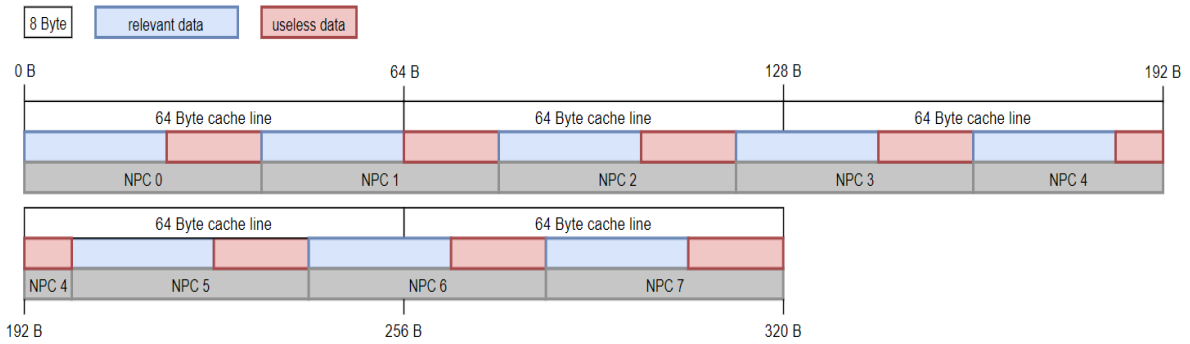

Figure 5: NPCs inside cache-lines, where blue is relevant data and red blocks represent unused data

miss! While accessing *NPC2*'s relevant data will partly load *NPC3*'s relevant data, we will only prefetch the first eight Byte of it.

```
1 struct NPCs{
2    float xyz[3 * NUM_ENTITIES];
3    float vel[3 * NUM_ENTITIES];
4    char *name[NUM_ENTITIES];
5    int age[NUM_ENTITIES];
6    int mood[NUM_ENTITIES];
7 };
8
9 NPCs npcs;
10
11 void update_npc_position(unsigned npc_id){
12    float local_xyz[3];
13    float local_vel[3];
14    memcpy(local_xyz, npcs.xyz + npc_id * 3, sizeof(float[3]));
15    memcpy(local_vel, npcs.vel + npc_id * 3, sizeof(float[3]));
16    local_xyz[0] += local_vel[0] * delta_t;
17    local_xyz[1] += local_vel[1] * delta_t;
18    local_xyz[2] += local_vel[2] * delta_t;
19    memcpy(npcs.xyz + npc_id * 3, local_xyz, sizeof(float[3]));
20 }
```

Code 5: SOA variant of the NPC

The remnant will be loaded separately from main memory, yet this will result in completely loading *NPC4*'s relevant data.
Continuing this we will count five cache misses and foregoing from 320 loaded Bytes this process will repeat. When eight NPCs result in five cache-misses for 1000 NPCs we will count $1000npc = \frac{5*1000}{8}cms = 625cms$ for the position update routines (where *cms* are cache misses).

After normalizing the *Obj*; *Human*; *NPC* thematic and implementing it in a SOA manner it could look like Code 5. What used to be individual class members are now columns/arrays, accessed by a key - the *npc_id*. From now on both reads to *xyz* and *vel* will fill a cacheline

worth of relevant data, respectively. A single cache-line now holds up to five positions or velocities. This doesn't prevent our data from overlapping in terms of cache-lines. This can be solved by adjusting the data's *alignment*. Since this will be used in the prototypical implementation it will be explained in TODO REF-SECTION, for now we could assume, that we align and pad our *float[3]* blocks of data to our cache-lines in a way that each cache-line holds exactly five of those relevant entities (see Fig. 6).

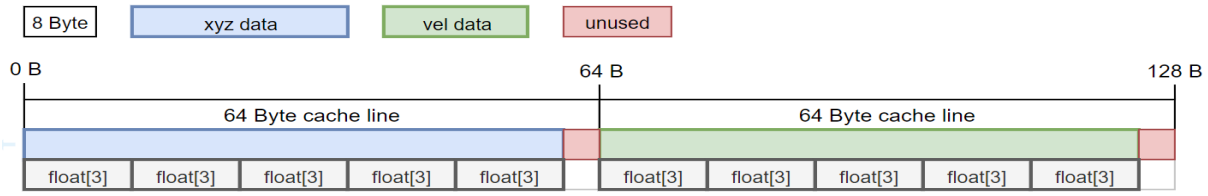

Figure 6: xyz and vel blocks inside cache-lines, where blue is joint float[3] blocks of xyz data, green is joint blocks of float[3] vel data and red is unused but intentional padding.

This leads to consistent throughput. In numbers we now have exactly two cache misses per five *NPCs* (We don't really have an NPC object anymore, but we are still allowed to *think* in objects). One for the positional data, one for the velocity data. Again for 1000 NPCs this would now result in 400 cache misses what translates to $225\times \sim 300$ clock cycles less latency than the AOS version, for position updates alone! Each frame!

This is starting to behave *optimal*. By not loading unneeded data into the cache we can store more relevant data. By Aligning and padding our data blocks correctly we attenuate the chance of *conflict misses* since we reduces the number of cache-lines related data depends on. We do however still have leftover space. The four Byte paddings we append to each $5 \times float[3]$ block has purpose, yet could theoretically hold information. Imagine our now theoretical NPC and thus our positional computation would involve a per NPC factor for maybe damping, as well as a mass. Still assuming that $sizeof(float) = 4$ this would be an additional eight Byte per NPC on each computation. Following our SOA approach we would define yet two new arrays for the damping factor and mass respectively. Accessing them would result in the utilization of another two cache-lines. Even though those cache-lines now suffice 16 NPCs each ($\frac{64}{sizeof(float)} = 16$) we now are dependent on four individual cache-lines to compute the *update_npc_position* for one NPC, so the amount of cache-lines scales linearly with the amount of parameters the function depends on (for SOA). In terms of eviction and consequently of conflict misses, this could yet again cultivate sub-optimal cache utilization. For the same reasons Intel's article on *Memory Layout Transformations*[29] also mentions increased pressure on the *TLB* (Translation Look-aside Buffer).
Even though we reduced the overall NPC per cache-line ratio, there is still unused information and scaling prone to eviction, all due to the individually related data blocks being physically separated. This doesn't countermand that SOA is arguably better than AOS, but it indicates, that there is still room for improvement.

1.3.4 Regarding temporal- and spatial locality / Components / AOSOA

The motivation behind our AOS to SOA conversion, was to maximize cache utilization when processing an NPC's position. We figured, that loading an object entails lots of unwanted data that does not share temporal locality with the information that is relevant to us. So we made sure, that instead of objects we loaded only wanted data. In order to do so we gave up something very important. When data is logically related it means, that it collaborates. Whenever we see, that certain subsets of data are frequently used together it is advisable not to separate them. So instead of rigorously converting each member into an array pendant, we group logically related data and create arrays of those groups instead (see Code 6).This way the relevant data concerning one NPC will not spread over different incoherent cache-lines, that could map to completely different segments in main memory.

```
struct npc_group{
   float xyz[3];
   float vel[3];
   float mass, damping;
};

npc_group
    npc_groups[NUM_ENTITIES];
```

Code 6: Consolidating related data

This technique bundles related data and makes sure it is successive in memory as well as in cache-lines, consequently we won't peril those cache-lines to extrude each other from the cache. The book *Compilers, Principles, Techniques and Tool* also describes a mechanism like this and refers to those groups as *blocks* [3, p. 786]. Also we get the chance to fully exploit our hardware's boundaries, as in now we can get rid of manually inserted padding Bytes (see Fig. 7) - provided we have related data that fits the gap. It is however only possible to gain a

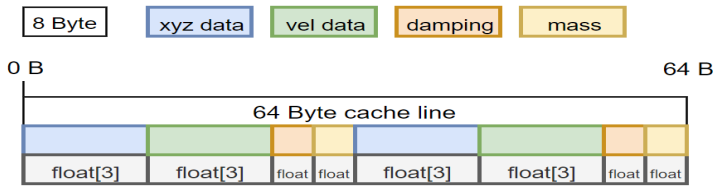

Figure 7: unified relevant data in a cache-line.

performance boost out of this, when the unified data is actually logically related. Whenever data is unified we basically have the same problem, we tried to get rid of in the first place - possibly loading unwanted data into the cache, increasing access latency. The moment we decide, that for example our game should provide the player with an indicator to which NPC is nearest to the player's avatar, we again would be doomed to load adherent information about the NPC's velocity, mass and stuff that was grouped to make position updates faster. Because for this we actually only want the NPCs' positions.

Hot/Cold Splitting

```
struct npc_cold_data{
   char *name;
   int age;
   int mood;
};

struct npc{
   float xyz[3];
   float vel[3];
   npc_cold_data
       *cold_data_ptr;
};
```

Code 7: The NPC class splitted into hot/cold data

A famous practical application of grouping a particular subset of data is called a *Hot/Cold Split* [27, p. 283]. It is also used to improve cache utilization, only it has a very specific definition of what members should be grouped.
The idea is to separate a record's member definitions into two subsets. One that contains all the hot-, and one that contains all the cold members, respectively. Data is hot when it is used frequently and cold when it is used rarely [7, p. 8]. By grouping together all the hot data we want to make sure, that data which is frequently used has a higher chance to exist in a cache line on access.
The cold data is externalized into a struct of its own. The original struct now contains only the hot data, as well as a pointer to a cold struct instance. Since access frequency does not necessarily resemble the original partitioning of the fields, this pattern emphasizes the preference of logical over contextual relation.
Especially for monolithic class definitions, there can be numerous logical subsets of data fields. For example one data subset of a classic OOP gameobject will mainly be used for physics calculations (velocity, acceleration, mass, colliders), another for rendering (vertice data, shaders, textures) and yet another that embeds the game object in the game's environment (health, strength, gold, stamina, etc.).
First of all it is not always apparent whether a field is hot/cold. An experienced programmer might feel confident enough for a reasonably small class definition to eyeball it. A better approach might be to wrap our fields with access mechanisms that let us count how often they are accessed at run time, yet again we could rely on static analysis. Since this work will specifically implement a hot/cold split in the prototype we will get back to this in TODO REF SEC.
Also we might end up picking individual fields of contextually differing data subsets. We could for example identify *velocity, vertice data, gold* as our hot fields, because they are frequently accessed. However they are utilized at different times of the game and individually they have no common logical relation that is relevant to our computations!
Even a split that divides contextual relation might result in a performance boost, if only the cold data is 'cold enough', but just as well a bad split might result in even worse cache utilization. In order to make a decision, that regards temporal- and spatial locality in a complex situation, we might need to find a way to evaluate, compare and eventually prioritize individual fields. An attempt to solve this will be made in the prototypical implementation, so more on that later on.

Components

After a grouping procedure the remnant members of the original NPC class could also be grouped by the same method we talked about before: take related members unify them in a struct and store those structs in an array so they are beyond equals. If the original class hierarchy was designed well the grouping of related data bits will start to resemble it, which might look like a step backwards at first, but remember, the related data groups are packed in seperate single purpose arrays and what counts is the access patterns to retrieve them. When we are done grouping all related data bits of the original NPC object, we will have recreated a so called *component pattern* [27, p. 213].
In the classical component pattern we will keep a container object that holds instances of each component [27, p. 214]. In favor of performance the container object should only hold pointers to the instances lying in their respective array. But actually and if we were able to group all domains of the original class, the object might be nothing else but an index, that can be used to retrieve a group out of its array.
Components are one widely used mechanism to decouple parts of a formerly shared entity. This is applied to classes and is therefore a statement to how OOP and DoD can work hand in hand. Not only is the component pattern useful for decoupling and performance interests, it also solves issues, that would normally be solved by applying multiple inheritance [27, p. 215], which is a practice even despised by OOP enthusiasts.
Components can be stored domain wise, while still being contextually linked individually on an object instance level. Their decoupling mechanism have proven great maintainability and even provide a comparably light weight interface for game designers. Accessing them can be done domain wise as well -> optimal cache utilization. This elegant arrangement between OOP and DoD makes it a favored pattern for modern game engines [13, p. 83].

Array Of Structure Of Arrays (AOSOA)

At first glance the idea of reintroducing the AOS concept seems confusing. In some cases depending on the original access patterns it might however be a good idea to separate the total amount of data into chunks that are often referred to as *Buckets*. We already went a step back before, when we decided to unify logically related bits and make arrays of groups. We figured, that this might be an optimal solution for a very specific computation, but might behave poorly in other situations. An attempt is to again separate the relevant members, however to a certain extend gain back the advantage of spatial locality. *The idea here is to get the benefit of locality at the outer-level and also unit-stride at the innermost-level*[29]. Applied

```
struct npc_bucket{
   float
   xyz[3] [SUB_SET_SIZE],
   vel[3] [SUB_SET_SIZE],
   mass   [SUB_SET_SIZE],
   damping[SUB_SET_SIZE];
};

npc_bucket
    npc_buckets[NUM_BUCKETS];
```

Code 8: AOSOA variant of grouped NPC traits

to our NPC it might look like Code 8. We merely define our former *columns*/arrays to hold only a subset of the total data respectively. The structure, that holds our member-arrays (the SOA) will however now be emplaced inside an array itself (the AO)! In Other words: We keep the data that will be used to transform each other close to prevent eviction. We enable the user to access specific members individually, to prevent loading unnecessary information.
This attempt is compossible with grouping certain members, too. After all we are still able to access only specific sub arrays of buckets. To retain the benefit of *page-locality* for the grouped member-arrays we need to find a fitting *bucket-size : number-buckets* ratio. In case of our Code 8 example, taken to the extreme *NUM_BUCKETS* $= 1$ would practically result in the classic SOA model, coming with all its advantages and disadvantages. On the other hand *SUB_SET_SIZE* $= 1$ would pretty much just be an object definition again (only needlessly more confusing and incomplete since we grouped the members).
If we want to optimize against our L1D cache, that performs an 8-way associativity mapping on 64 Byte cache-lines, we could argue that $\frac{8\times64}{(8\times sizeof(float)} = 16$ might be a fitting *SUB_SET_SIZE*, because all the elements in a bucket would then be guaranteed to map to different cache-lines. So no conflict misses when performing transformations on related data groups. Note that this statement is only 100% valid as long as our buckets are aligned correctly.
Finding the NPC next to the player would now mean iterating the *xyz* subsets of each bucket. Depending on the data bundled in a bucket (especially concerning alignment and padding) we still need to expect to load unwanted data subsequently to *xyz* but this will now happen on a *per-bucket* basis rather than on a *per-NPC* basis.

2 Motivation

We have now seen some of the fundamental differences between *Object Oriented Programming* and *Data oriented Design*. After recognizing the existence of a *memory gap* we got to know some of the memory units our modern computer architectures are composed of. This helped us understanding why the caching technologies we make use of today tolerate but do not thrive on the real world metaphors we use to design our data models.
We learned that DoD offers methods that comply to our modern hardware, by trading some of our beloved abstraction as well as readability in exchange for performance. So in terms of utilizing the memory hierarchy it is inarguable superior to OOP. But we have also identified abstraction to be one of the most important skills a programmer can have, since it is directly linked to a humans capability to solve a problem. We came to an understanding, that the abstraction model, that OOP inherits to us is widely accepted and applied in the industry, because it is intuitive and easy to learn.
Even when one is ready to abandon OOP it will persist. The industry is famous for its reluctance to change. Implementing a new programming language into a developer team might be beneficial in long term, but comes with less to zero temporary productivity and initial training. Also DoD requires an understanding of hardware concerns, that novices and fresh graduates usually don't have. Most projects and companies are not even that dependent on high performance code, instead rely on solutions, that are quickly developed and easy to maintain and for the same reasons even the games industry won't solely rely on DoD probably ever. We also depend on gameplay programmers or level designers who will interact with an engine heavily but shouldn't have to think about the underlying hardware all the time [13, p. 260]. The point is: No matter how much better other programming paradigms are on certain viewpoints, OOP is here to stay. Instead of trying to get rid of it we might just try to find a way to get the best out of both worlds.

The best out of both worlds

We already determined that DoD and OOP can get along to certain extends (see Section 1.3.4). The more we want to rely on DoD to obtain performance boosts however, the more we will dismantle our abstraction step by step. Ideally we could keep whats best about OOP and still have optimal performance as though we had implemented our idea using DoD.
As mentioned before in Section 1.3.1 DoD is not inferior to OOP in terms of maintainability per se. DoD's greatest disadvantage is, that it doesn't allow us to transfer a problem into code, the way we perceive it in the real world. The intuitive and thus advantageous abstraction model that

comes with OOP is lost. On the other hand better performance makes a strong case for DoD, especially for game developers.
In conclusion what we want is the real world metaphors coming with OOP as well as the performance benefits coming from a data layout, that facilitates optimal cache utilization. The question remains how, can we achieve both those things?

2.0.1 Native language support for DoD principles

There are languages in existence and under development, that aim to provide native support for SOA/AOSOA data structures!

Intel's ISPC

The *Intel SPMD Program Compiler* (ISPC) is specifically designed to support quick and easy development of *Single Program Multiple Data SPMD* applications, making use of implicit *Single Instruction Multiple Data* (SIMD) vector units [1]. Those instructions depend on SOA data layouts, thus the language provides a *soa* keyword to automatically transform an AOS defined struct into a SOA format. In Code 9 an AOS *npc_group* is defined in line 1. In line 6 the *pos_and_vel* is defines as a SOA holding 128 consecutive *x*, *y*, *z*, *v_x*, *v_y*, and *v_z* respectively. This also easily enables for AOSOA format as can be seen in line 7.

```
1 struct npc_group{
2    float x, y, z;
3    float v_x, v_y, v_z;
4 };
5
6 soa<128> npc_group
     pos_and_vel;
7 soa<16> npc_group
     aosoa_pos_and_vel[8];
```

Code 9: ISPC's native SOA support

Jonathan Blow and JAI

A prominent critic of the C++ language Jonathan Blow for example is working on the *JAI* programming language. There is currently no official documentation to it and it is unknown when the language will be released to public, but some of its features and its design goals are already well known. One of the highly anticipated features is automatic AOS to SOA conversion done by the compiler using nothing but a single keyword. There is no official documentation and information presented here originates only from the various online video talks Blow provides occasionally. In [5] the *SOA* keyword is introduced as a typespecifier when creating a struct, automatically informing the compiler to store the struct's members in a SOA fashion and granting correct access to them (see Code 10).

```
1 npc_group :: struct SOA{
2    xyz : [3] float;
3    vel : [3] float;
4 };
```

Code 10: JAI's native SOA support

2.0.2 High level abstraction hiding DoD

However we already know, that introducing new languages/technologies into a functioning industry is mostly viewed as a cost factor and since C++ is the most prominent language in the game development industry, we can't expect to see a lot of native language support for DoD principles like SOA in the near future (unless the ISO C++ committee decides in its favor).
One possible solution could be to provide high level abstraction containers, that internally work with data oriented concepts. In his online blog article *Implementing a semi-automatic structure-of-arrays data container* [30] Stefan Reinalter introduces a possible implementation for such mechanisms. Template meta programming is a way of interacting with the compiler and can to a certain extend overcome the inherent conflict between OOP and DoD, but Reinalter states that:

> *I really would like to have a fully automatic implementation, but I don't believe that's possible without compiler support.* [30]

Even when we can provide high level data containers, that implement a cache friendly data layout, it can't completely decouple the process of modeling the reality into code from reflecting about its data layout considering optimal hardware utilization. The high level containers in the end still need to be used correctly and based on implementation may require to define the relevant classes dependent on it (for example with macros).
If possible an ideal solution would be uncorrupted high abstraction code, that somehow translates to high performance code. The relevant keyword here is *translate*. Compilers normally do this kind of stuff. The question arises whether we could utilize compiler technology to accomplish our goal.

3 Compiler technology as a mediator between OOP and DoD

The inherent purpose of a compiler is to read a program defined in a *source language* and translate it to an equivalent pendant for a *target language* [3, p. 1]. They are yet another form of abstraction, that enable us to work more efficiently. Low-level languages interact 'less indirectly' with the hardware (more control) thus enable one to write better performing code. But they are usually harder to write as well as less portable, more prone to errors, and harder to maintain [3, p. 17].
Compilers provide some of the most important features a programmer needs, like syntactic and semantic analysis steps, which can automate the process of finding errors and even just smelly code. To do that they need some sort of 'understand' for the program.

3.1 A compiler's understanding of the program

Modern compilers implement several *phases* bundled in *passes* to provide an abstraction rich routine from reading mere character sequences until generating char sequences in the target language (see Fig. 9). Of course compilers are not thinking entities, but there are mechanisms to formally define a language as well as steps to generate semantic statements with it, ordering them in a way so that a computer can process them in a meaningful way.

```
stmt → expr ;
     | if ( expr ) stmt
```

Code 11: Exerpt of example context free grammar defining a (if)statement. Bold = terminal; italic = nonterminal

Syntax definition

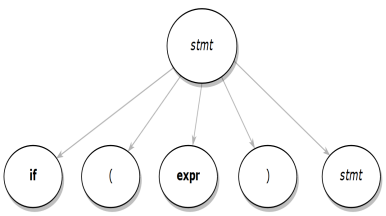

Figure 8: Parse tree for the if-stmt node.

A syntactical language definition can be done using the *context free grammar* notation or *BNF* (Backus-Naur Form) [3, p. 40]. Those grammars define a hierarchy of rules on how to form statements/expressions in the language. By defining a set of elementary symbols (*terminals*) for example keywords we can then define more complex *nonterminals*, like defining how a statement is formed. Ultimately we can make *production* rules that describe for example our control flow statements

(see Code 11). Just like this set of grammar rules basically constitutes a hierarchy we can deduce a *parse tree* for it as a concrete implementation, where beginning from the start symbol we can derive valid successors for each symbol by iterating its child nodes. Given a statement like: "***if true**) i++;*". After reading the first token *if* we can iterate our parse tree's production node for that statement and easily see, that the rule demands an opening bracket immediately following the if terminal, rendering the input as syntactically ill-formed (see Fig. 8). To be able to analyze a program like this we initially need to pass through a few *phases* transforming and collecting data until we have a representation, we can work with.

Lexical analysis

The compilers *Lexer* or *Scanner* takes the raw sequence of chars forming the source code and creates tokens out of char subsets it identifies as such. This information is used to fill the *Symbol table*, which holds information like types, relative positions of the values, scopes and more. The symbol table is used in several following phases and essential for correct linkage of different compilation units.

Syntax analysis

The syntax analyzer creates the first *Intermediate Representation* of the source code, the *Syntax Tree* or *Abstract Syntax Tree* (AST). It takes the token stream provided by the first phase and orders them in a tree like structure that already accounts for computational order and depicts the the grammar of the input.

Semantic analysis

Analyzing the AST from the previous phase is the *Semantic Analyzer*'s duty. It traverses the AST and constantly compares its nodes with the formal language definition and gathers information like type traits. Consequently *type checking* - which is important for statically typed languages - will take place in this phase [3, p. 5-9].

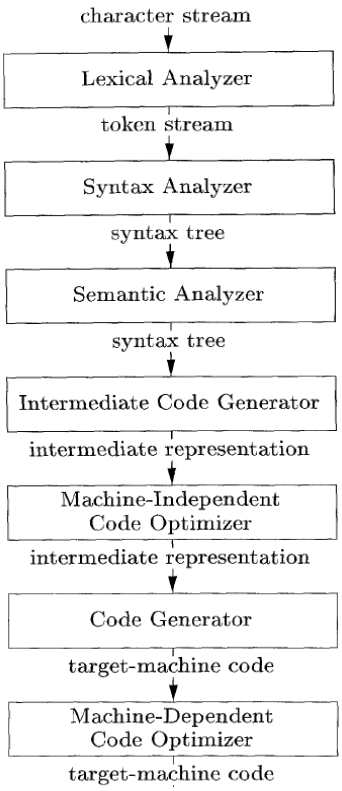

Figure 9: Phases of a compiler (Source: [3, p. 5]).

3.2 A useful interface

As can be seen in Fig. 9 there are several more phases left to describe, however Section 3.1 already provides information that we can utilize towards implementing a tool, that automatically translates OOP code into a cache friendly pendant.
Assuming we have access to our programs AST, we could start analyzing the code in an environment, that allows us to traverse the code in a tree like fashion. This means easy access to

the defined data layout, as well as the access patterns in use.
Luckily modern compilers are designed in a modular fashion and usually define *front ends* and *back ends* to facilitate a multiple language to machine mapping. The front end consists of the analysis phases as well as the intermediate code generation phases for a source language. After generating an immediate representation (IR) it is forwarded to the back end which *synthesizes* the end product in the desired target language [3, p. 4].
The front end is especially interesting to us since it provides us with the appropriate representations to thoroughly investigate a program.

LLVM/Clang

A rather prominent representative of such an assembler/compiler/debugger tool-chain is the open source LLVM project. The front end functionality for C++ is here implemented in the Clang compiler. All the functionality is accessible and furthermore served through diverse libraries and interfaces - for example the *LibTooling* library that brings functionality for parsing code, creating ASTs and running *FrontEndActions* over it. There are already mechanisms for recursive AST traversal like *Recursive AST Visitors* and AST matching functionality with the *AST Matchers*. Also tools like *clang-query* provide a *REPL* (Read-Eval-Print Loop) environment for quick testing.

4 Quotes

4.1 hennessy

4.1.1 principle of locality

The most important program property that we regularly exploit is the principle of locality: Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses. Two different types of locality have been observed. Temporal locality states that recently accessed items are likely to be accessed in the near future. Spatial locality says that items whose addresses are near one another tend to be referenced close together in time. [16, p. 38]

4.2 drepper

4.2.1 cache layout

4.2.2 Lc Ld

Even though most computers for the last several decades have used the von Neumann architecture, experience has shown that it is of advantage to separate the caches used for code and for data [12, p. 14]

4.3 compilers

4.3.1 what is a compiler

Simply stated, a compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language[...]. [3, p. 1]

4.3.2 evolution of programming languages

[3, p. 13]

4.3.3 requirements to optimizations

[3, p. 16]

4.3.4 optimizing compilers

Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code. Unfortunately , lower-level programs are harder to write and - worse still - less portable, more prone to errors, and harder to maintain. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions. [3, p. 17]

4.3.5 compiler phases

4.3.6 compiler front end

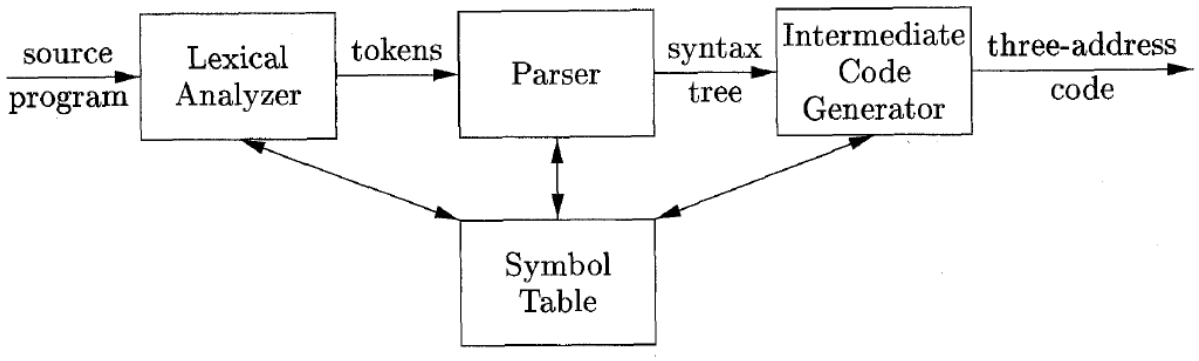

Figure 10: Model of compiler front end (Source: [3, p. 41]).

4.3.7 AST

In an *abstract syntax tree* for an expression, each interior node represents an operator; the children of the node represent the operands of the operator. More generally, any programming construct can be handled by making up an operator for the construct and treating as operands the semantically meaningful components of that construct. [3, p. 69]

Bibliography

[1] *Intel® SPMD Program Compiler User's Guide*.

[2] *TIOBE Programming Community Index*.

[3] AHO, A. V.: *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[4] AL DALLAL, J.: *A design-based cohesion metric for object-oriented classes*. International Journal of Computer Science and Engineering, 1(3):195–200, 2007.

[5] BLOW, J.: *Data-Oriented Demo: SOA, composition*.

[6] CHEN, T.-F. and J.-L. BAER: *Effective hardware-based data prefetching for high-performance processors*. IEEE transactions on computers, 44(5):609–623, 1995.

[7] CHILIMBI, T. M., M. D. HILL and J. R. LARUS: *Making pointer-based data structures cache conscious*. Computer, 33(12):67–74, 2000.

[8] CODD, E. F.: *A relational model of data for large shared data banks*. Communications of the ACM, 13(6):377–387, 1970.

[9] CONTI, C.: *Concepts for buffer storage*. IEEE Computer Group News, 2(8):9, 1969.

[10] COORPORATION, I.: *Intel 64 and IA-32 architectures optimization reference manual*, 2009.

[11] CRAGON, H. G.: *Memory systems and pipelined processors*. Jones & Bartlett Learning, 1996.

[12] DREPPER, U.: *What Every Programmer Should Know About Memory*, 2007.

[13] FABIAN, R.: *Data-oriented design: software engineering for limited resources and short schedules*. 2018.

[14] GHEZZI, C., M. JAZAYERI and D. MANDRIOLI: *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[15] GREGORY, J.: *Game engine architecture*. AK Peters/CRC Press, 2014.

[16] HENNESSY, J. L. and D. A. PATTERSON: *Computer architecture: a quantitative approach*. Elsevier, 2011.

[17] HUANG, P. J.: *A Brief History of Object-Oriented Programming*.

[18] KRAMER, J.: *Is abstraction the key to computing?*. Communications of the ACM, 50(4):36–42, 2007.

[19] LAPLANTE, P. A.: *What every engineer should know about software engineering*. CRC Press, 2007.

[20] LEVINTHAL, D.: *Performance analysis guide for intel core i7 processor and intel xeon 5500 processors*. Intel Performance Analysis Guide, 30:18, 2009.

[21] LOUDEN, K. C. and P. ADAMS: *Programming Languages: Principles and Practice, Wadsworth Publ*. Co., Belmont, CA, 1993.

[22] LUK, C.-K. and T. C. MOWRY: *Compiler-based prefetching for recursive data structures*. In *ACM SIGOPS Operating Systems Review*, vol. 30, pp. 222–233. ACM, 1996.

[23] MARTIN, R. C.: *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[24] MITTAL, S.: *A survey of recent prefetching techniques for processor caches*. ACM Computing Surveys (CSUR), 49(2):35, 2016.

[25] NELSON, M. L.: *An Introduction to Object-Oriented Programming*. Techn. Rep., NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 1990.

[26] NIEMANN, K. D.: *Von der Unternehmensarchitektur zur IT-Governance*. Springer, 2005.

[27] NYSTROM, R.: *Game programming patterns*. Genever Benning, 2014.

[28] REBELSKY, S. A.: *A Brief History of Programming Languages*, 1999.

[29] S., A.: *Memory Layout Transformations*, 2013.

[30] STEFAN REINALTER, D.: *Implementing a semi-automatic structure-of-arrays data container*, 2013.

List of Figures

List of Tables

List of Code

List of Abbreviations

ABC Alphabet

WWW world wide web

ROFL Rolling on floor laughing

A Anhang A

B Anhang B