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 abide cache conscious data layout principles, yet maintain the abstraction level of Object Oriented Programming

How compiler technology could possibly build a bridge between conflicting programming paradigms

By: Julian Müller, BSc.

Student Number: 1610585015

Supervisor: Stefan Reinalter, DI

Second Supervisor: Prof. Alexander Hofmann, DI

Wien, March 1, 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, March 1, 2019 Signature

Abstract

This work will prospect the possibility of using compiler technology as a mediator between the conflicting programming paradigms/philosophies *OOP* and *DoD*.
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 and influences 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.
To do so this thesis will investigate key concepts of both paradigms, as well as hardware specifics in modern computer architectures to explicate the reasons for their good/bad interaction with the hardware.
A prototypical implementation of a source-to-source transformation tool called *COOP* (**C**ache friendly **O**bject **O**riented **P**rogramming), developed in Clang's LibTooling environment, will determine wether or not compiler technology can be used to achieve a performance optimization on a classically OOP abidant target code base.

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

Acknowledgements

TODO dankscheeee

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 *Scala* for example 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 [17, 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 [29][36]. 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 [23]. 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)*" [33, 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 [18, 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

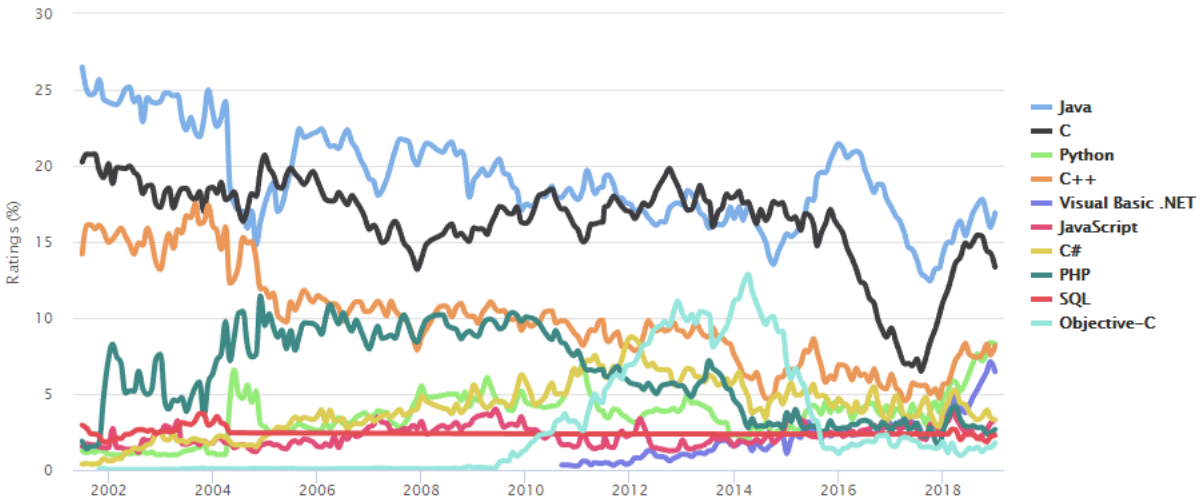

Figure 1: Most popular programming languages throughout the years (Source: [3]).

along, so to say. OOP offers 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 [6] - 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 [35, p. 272].

```
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

The abbreviation *AOS* stands for *Array Of Structures*[13] and it describes, what usually happens with Object Oriented Programming.
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 -

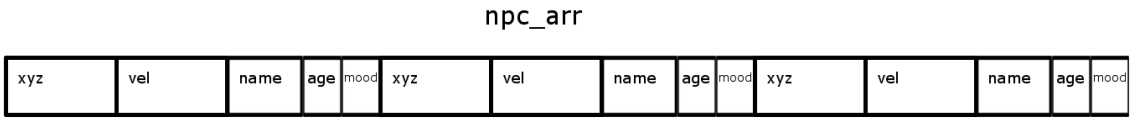

Figure 2: Visualization of how a *npc_arr* will exist in memory

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 NPCs - 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.* [31, p. 35]
>
> *Keep it simple and smart* (KISS principle). [34, p. 77]
>
> *[...] cohesion is an important object-oriented software quality attribute.*[6]

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.[31]
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 [26, 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, capacity and numerous technical properties. The intention behind this complex hierarchy of memory layers is of course 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[...].* [20, 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.* [19, 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).

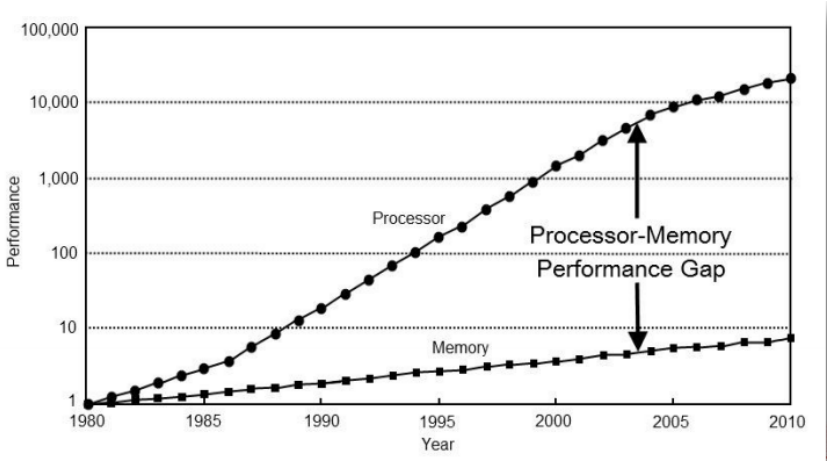

Figure 3: "***Starting with 1980 performance as a baseline, the gap in performance between memory and processors is plotted over time****. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with 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: [20, p. 289])

To solve the issue of ever diverging CPU/memory performances (commonly referred to as the *memory gap [19]*), 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 [12] in 1969. Originally cache technology was mentioned as *buffers*[15]. 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 [19, p. 191]. This principle originated in the work on *Virtual Memory* [15, 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

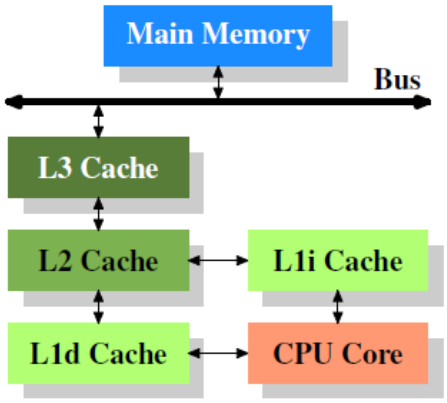

Figure 4: Exemplary, simplified model of a CPU core and its several cache modules (Source: [16, p. 15])

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) [28]. For reference access to main RAM "*can take on the order of 500 cycles[...]*" [19, 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 REF SEC).
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 [15, 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. [17, p. 268]
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 [19, 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 and levels 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 [21, p. 50]:

- 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: [19, 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 [19, 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 [15, 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*[15] as well as Jason Gregory's *Game Engine Architecture*[19] are highly recommended. 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: [15, 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 AU 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 data basically means:

> "*[...] bringing data in the data (or mixed instruction-data) cache before it is directly accessed by a memory instruction [...].*" [9, 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: [9, 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 [9]. 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 [32, 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.* [30, 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. DoD converts this information into a generic set of rules/best practices, a methodology for efficiency.

1.3 Data Oriented Design

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 [25, p. 5]. However disregarding intrinsic detail is predestined to result in poor resource utilization at that level. *Data oriented Design* wants us to be 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 [...].* [17, 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).

```
1 struct particle
2 {
3    float ms_alive, lifetime_in_ms;
4    float xyz[3];
5    Shader *shader;
6 };
7
8 struct particle_system
9 {
10   particle particles[1024];
11   unsigned particles_alive;
12   ...
13 };
```

Code 2: OOP typical, simplified 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 (Code 2 line 10). 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         [collision code]
12      }
13   }
14 }
```

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. It is 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.
These changes are starting to proliferate 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[17, 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 *better*?

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* [17, p. 25]. After all a database should handle data correctly.
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. Fabian says:

> *The structure of any data is a trade-off between performance, readability, mainte-*

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 a possible first normalization step for the particle system and how it indicates bad design

nance, future proofing, extendibility, and reuse. [17, p. 28]

Therefore we could attempt to treat our data 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 [11]. This is done because laying out your data in a linear fashion goes away from embedding meaning to its structure (Class/Struct) [17, p. 25] and 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 [17, 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 [17, 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 realize some things.

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

2. 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 member is a column/array

and each instance of a particle or particle system is now a mere index.

3. 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.

4. Especially after resolving redundancy, 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).

After *laying out our data* in a linear fashion, we can now implement a *data layout* that implements our new ideas.

1.3.3 Structure of Arrays / SOA

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 friendlier structure [17, p. 163]. We already know now, that the hardware, specifically the cache uses certain strategies to provide data out of main memory, that is likely to be used soon. This happens both when loading data in cache-line sized blocks and through prefetching mechanisms (see Section 1.2.4). Now when implementing the column-oriented approach that comes with data normalization we automatically almost perfectly conform to those auxiliary means 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.

AOS in the cache

Remembering Code 1, when examining the NPC class we know that after derivation (done by the compiler) it looks like Code 4. 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 multiplied with a delta-time to their current xyz. With a typical L1 D cache-line of 64 Byte and assuming that an NPC is 40 Byte (float = 4 Byte, pointer = 8 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 $2 \times 3 \times$ *sizeof(float)* $= 24$ Byte, we waste $40 - 24 = 16$ Byte of our cache per NPC. This waste scales linearly with our data.

```
struct NPC
{
//inherited from Obj
float xyz[3];
float vel[3];
//inherited from Human
char *name;
int age;
//NPC
int mood;
};
```

Code 4: NPC pod after derivation is done

As a quick approximation onto a damage report we could do the following: The smallest common multiple of our 40 Byte NPC and our 64 Byte cache-line is 320, so with eight NPCs we load five cache-lines (see Fig. 5). This equals five compulsory misses. After 320 Byte this layout repeats so we could extrapolate to greater numbers of NPCs Fig. 6. A cache miss oc-

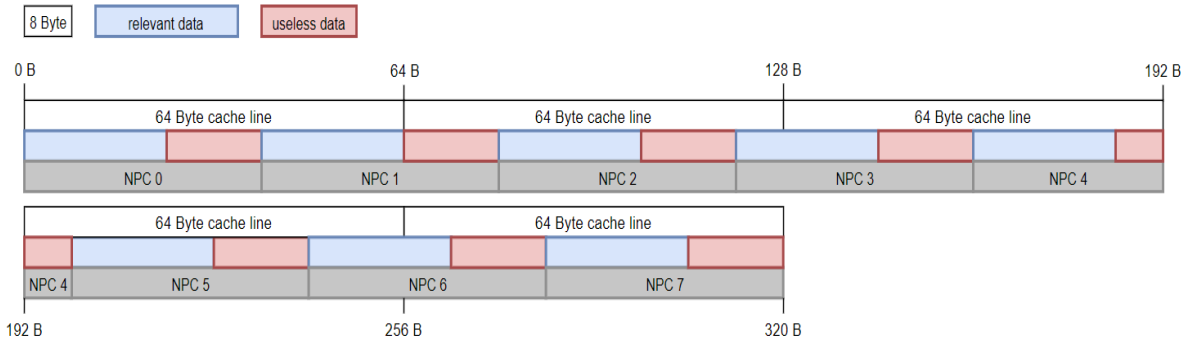

Figure 5: NPCs inside cache-lines, where blue is relevant data and red blocks represent unused data

curs whenever we do not find an AU inside a 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 miss! While accessing *NPC2*'s relevant data will partly load *NPC3*'s relevant data, we will only get the first eight Byte of it. The remnant will be loaded separately, 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. The statement that 1000 NPCs will count $1000npc = \frac{5*1000}{8}cms = 625cms$ cache misses for the position update routines (where *cms* are cache misses), is only a narrow assumption, since our cache-lines are defined in fixed quantums (see Fig. 6).

SOA in the cache

Implementing the *NPC* 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 ($5\frac{1}{3}$) positions or velocities. This doesn't prevent our data from overlapping in terms of cache-lines. The smallest common multiple of 12 and 64 is 192, so over $\frac{192}{64} = 3$ cache-lines we will fit $\frac{192}{12} = 16$ units. As can be seen in Fig. 6 the SOA attempt will develop better cache utilization for contiguous AU access on rising data sizes, disregarding any external eviction. This does not automatically equal the amount of cache misses, since repeated data access will find the data in the cache. But a SOA data model will reduce compulsory misses.

```
struct NPCs{
float xyz[3 * NUM_ENTITIES];
float vel[3 * NUM_ENTITIES];
char *name[NUM_ENTITIES];
int age[NUM_ENTITIES];
int mood[NUM_ENTITIES];
} npcs;
```

Code 5: SOA variant of the NPC

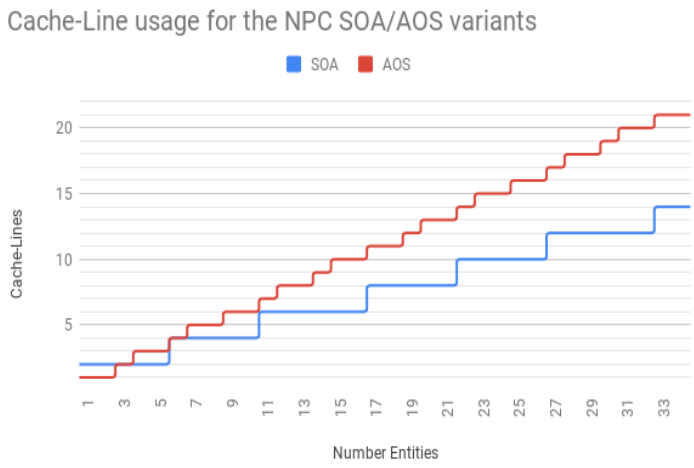

Figure 6: Cache-line efficiency comparing NPCs represented as SOA and AOS.

The unit we use for computations here is the size of three floats (12 Byte), so while a cache-line fits $\lfloor \frac{64}{12} \rfloor = 5$ complete units, the remnant of $64 - 12 \times 5 = 4$ Byte will belong to a unit, we will need another cache-line for. This overlap happens exactly $\frac{12}{64-60} = 3$ times until in the fourth cache-line this process repeats. Distributing logically dependent data over multiple cache-lines should be avoided, since access to it will result in more cache misses and is prone to eviction [13, p. 12-19].

It 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 SEC, 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 entities (see Fig. 7). 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*.

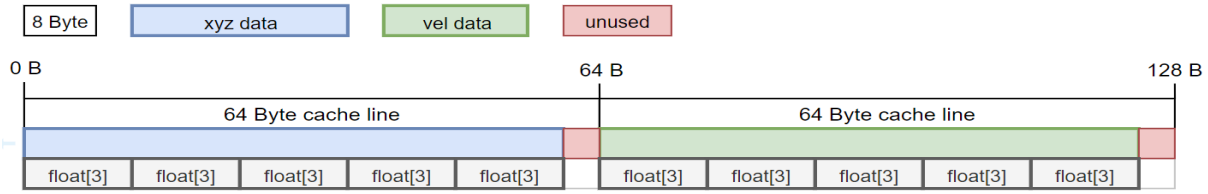

Figure 7: xyz and vel blocks inside cache-lines, where blue represents joint float[3] blocks of xyz data, green joint blocks of float[3] vel data and red is unused but intentional padding.

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 reduce the number of cache-lines the 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 computation 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* [37] 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 performs better than AOS, but it indicates, that there is still room for improvement.

1.3.4 Regarding temporal- and spatial locality

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

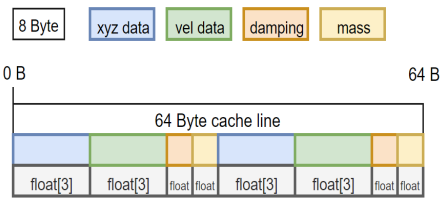

Figure 8: Unified/Grouped relevant data in a cache-line.

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* [5, 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. 8) - provided we have related data that fits the gap. It is however only possible to gain a performance boost out of this, when the unified data actually shares temporal locality.

Hot/Cold Splitting

A famous practical application of grouping a particular subset of data is called a *Hot/Cold Split* [35, 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 [10, 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.

```
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

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 (domain wise). If the original class hierarchy was designed well in terms of cohesion metrics, 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* [35, p. 213].
In the classical component pattern we will keep a container object that holds instances of each component [35, 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 members 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 [35, p. 215], which is a practice often despised even by OOP enthusiasts.
Components can be stored domain wise, while still being contextually linked individually on an object instance level. Their decoupling mechanism has proven great maintainability and even provides 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 [17, 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* or *Tiles*. 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.

```
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

Whenever data is grouped we might 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 play a scary sound when the players distance to an NPC falls below a certain threshold, 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.
An attempt to solve this, is to yet again separate the relevant members, however to a certain extend gain back the advantage of spatial locality. Applied 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 structures, that hold 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 subsets individually, to prevent loading unnecessary information. *The idea here is to get the benefit of locality at the outer-level and also unit-stride at the innermost-level* [37]. This attempt is compossible with grouping certain members, too. After all we are still able to access only specific sub arrays of buckets. A bucket should consist of logically related data, so we know that for a bucket's member array a_n and an element index npc_e each member array $a_{0...n-1}$ of a bucket contains data that is relevant to a distinct set of computations at position npc_e. For example $npc\_buckets[0].xyz[1]$ and $npc\_buckets[0].vel[1]$ will contribute to a distinct computation since they are used to describe a single abstract entity's state.
Finding the NPC nearest 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.

A crucial factor to the performance of an AOSOA data layout is it's subset size and the resulting *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 (so AOS only needlessly more confusing and incomplete since we grouped the members).
Concerning our *SUB_SET_SIZE*, independent of a cache's associativity, the *eviction* of elements inside contiguous blocks of memory will only ever occur when the data's size exceeds the cache's capacity. One first conclusion could be, that our bucket's size should not exceed our L1 D cache capacity (e.g. 32KiB), to guarantee, that the adressable unit at $a_n + npc_e$ will not be evicted by access on $a_{n+1} + npc_e$ nor by any access on $a_t + npc_e$ where $t > n$.
Additionally [13, p. 61] advises us to "*Optimize data structures [...] to fit in one-half of the first-level cache [...]*" (e.g. 16KiB) because the cache is hardware and therefore shared by all processes currently running. We usually can't expect exclusive access and full cache capacity exploitation. Demanding all resources might perform well in a situation where no other

processes demand frequent main memory access. It might also be prone to eviction when the systems workload is increased. Multi way associativity techniques accommodate us to great amounts in this case, but the moment we are trying to utilize a cache in its entirety we foster concurrency between processes, that will eventually affect the entire system.
Going even further [13, p. 3;66] warns about it's L2 hardware prefetching mechanisms to only work within page boundaries (e.g. 4KiB), so making sure a bucket fits this criteria we could further advance optimal prefetching behavior, because we ensure, that no $a_n + npc_e$ and $a_n + npc_{e+1}$ will exceed a page boundary. This complies to the principle of page locality [37].
We're not even done yet. AOSOA perfectly suffices parallelization. Dependent, on which cache level is shared among the CPUs/Cores we could further optimize against it by adapting our bucket's size to for example an L2 128 Byte cache-line, resulting in a *SUB_SET_SIZE* of $\frac{128}{8 \times sizeof(float)} = 4$, guaranteeing minimal cache-level synchronization overhead.
Another approach could be to determine the ratio between *single-entity-access-patterns* to *domain-related-access-patterns* in the source program. The more our application relies on entity-access patterns, meaning we want all or most of an entity's members (classic OOP access) the smaller our subsets could be. On the other hand, the more our application relies on domain-related access patterns, meaning we want all or many of our entities' specific members, the greater our subsets should be.

The AOSOA pattern is highly flexible making it a fit for a lot of use cases. Its limitations are defined by the minimal bucket size and the optimization goals. The Buckets' sizes can be modified at compile time or at run time. This way even when one can't reason about subset sizes, there is still the possibility to just try out different iterations and document changes in performance/cache utilization.

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 might not 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 [17, 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 / ISPC / JAI

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 [2]. 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 defined 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 game developer and 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 [8] 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* [38] 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.* [38]

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 records dependent on it (for example with macros), because due to missing *reflection* features, we can't iterate a record's members.
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 these kinds of tasks. 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* [5, p. 1].
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 'understanding' 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. 10). 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

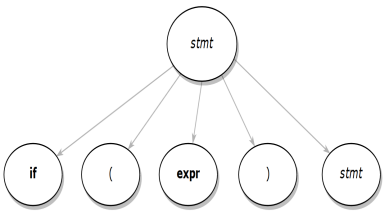

Figure 9: Parse tree for the if-stmt node.

A syntactical language definition can be done using the *context free grammar* notation or *BNF* (Backus-Naur Form) [5, 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. 9). 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 [5, p. 5-9].

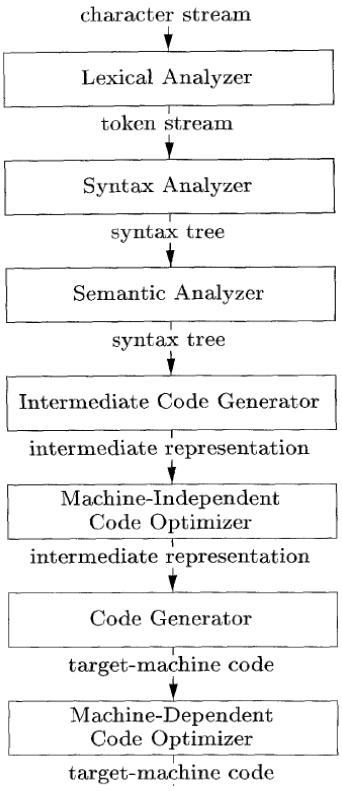

Figure 10: Phases of a compiler (Source: [5, p. 5]).

3.2 A useful interface / LibTooling

As can be seen in Fig. 10 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 intermediate representation (IR) it is forwarded to the back end which *synthesizes* the end product in the desired target language [5, 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 production-grade reusable libraries and interfaces [27] - 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.

```
1  struct Foo {
2      int bar;
3  };
4
5  int main(){
6      Foo f;
7      f.bar = 10;
8  }
```

Code 12: Example code in a Foo.cpp file

In Section 2.0.1 we saw, that languages supporting DoD natively rely on particular keywords to tag a record as SOA layouts for the compiler. We will try to carefully select the right ones using appropriate metrics (Not each record will qualify for our optimization). In case of finding any records at all it is easy enough, thanks to the functionality coming with Clang. The *AST*

```
|-CXXRecordDecl 0x8395d40 <p.cpp:1:1, line:3:1> line:1:8 referenced struct Foo definition
| |-DefinitionData pass_in_registers aggregate standard_layout trivially_copyable pod trivial literal
| | |-DefaultConstructor exists trivial
| | |-CopyConstructor simple trivial has_const_param implicit_has_const_param
| | |-MoveConstructor exists simple trivial
| | |-CopyAssignment trivial has_const_param needs_implicit implicit_has_const_param
| | |-MoveAssignment exists simple trivial needs_implicit
| | `-Destructor simple irrelevant trivial needs_implicit
| |-CXXRecordDecl 0x8395e68 <col:1, col:8> col:8 implicit struct Foo
| |-FieldDecl 0x8395f18 <line:2:2, col:6> col:6 referenced bar 'int'
| |-CXXConstructorDecl 0x8396178 <line:1:8> col:8 implicit used Foo 'void () noexcept' inline default trivial
| | `-CompoundStmt 0x8396670 <col:8>
| |-CXXConstructorDecl 0x83962c8 <col:8> col:8 implicit constexpr Foo 'void (const Foo &)' inline default trivial noexcept-unevaluated 0x83962c8
| | `-ParmVarDecl 0x8396400 <col:8> col:8 'const Foo &'
| `-CXXConstructorDecl 0x8396498 <col:8> col:8 implicit constexpr Foo 'void (Foo &&)' inline default trivial noexcept-unevaluated 0x8396498
|   `-ParmVarDecl 0x83965d0 <col:8> col:8 'Foo &&'
`-FunctionDecl 0x8395fd8 <line:5:1, line:9:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x8396830 <line:6:1, line:9:1>
    |-DeclStmt 0x83966b0 <line:7:2, col:7>
    | `-VarDecl 0x83960f8 <col:2, col:6> col:6 used f 'Foo' callinit
    |   `-CXXConstructExpr 0x8396680 <col:6> 'Foo' 'void () noexcept'
    `-BinaryOperator 0x8396748 <line:8:2, col:10> 'int' lvalue '='
      |-MemberExpr 0x83966f0 <col:2, col:4> 'int' lvalue .bar 0x8395f18
      | `-DeclRefExpr 0x83966c8 <col:2> 'Foo' lvalue Var 0x83960f8 'f' 'Foo'
      `-IntegerLiteral 0x8396728 <col:10> 'int' 10
```

Figure 11: AST dump of Code 12 generated with *clang -Xclang -ast-dump Foo.cpp*.

Matchers and the online *AST Matcher Reference* offer an excellent modular way of matching AST nodes against predefined patterns. Working with an AST is of course a lot more comfortable than working on plain text, but it also entails a new domain that one needs to familiarize with. Clang also provides proper functionality to do so. For example seeing what kind of AST nodes there are in a certain source code. When using *'clang -Xclang -ast-dump Foo.cpp'* on Code 12 among additional meta information we will see something like Fig. 11.

Figure 12: AST dump of Code 12 generated with some simple AST matchers in the easy to use *clang-query* environment.

In this textual representation of the AST we can see what AST nodes make up our record (*CXXRecordDecl, FieldDecl,* implicit *CXXConstructorDecl*s) and how it is used (*CXXConstructExpr, DeclRefExpr*). The *clang-query* tool (see Section 3.2) offers a platform for immediate testing, without setting up the rather complex structure a clang tool requires.

In Fig. 12 we explore some simple AST matchers like *cxxRecordDecl* and *functionDecl()*. Even though Code 12 only defines one function (main) with the matcher *functionDecl()* we have 4 matches. Also we don't even just match the main function but also our record's definition! This is due to implicit statements being handled the same way for matchers with low complexity. When looking up the AST matcher in the online AST Matcher Reference we will find the following documentation:

```
Matcher<Decl>  functionDecl   Matcher<FunctionDecl>...

Matches function declarations.

Example matches f
void f();
```

Code 13: AST Matcher Reference documentation for the matcher functionDecl()

In this case, when one is confronted with a problem the documentation won't explain, the '*cfe-dev – Clang Front End for LLVM Developers' Mailing List*' is the platform to receive help from active developers. The AST dumping mechanisms, the AST Matcher Reference, the clang-query environment and the clang mailing list as a last resort will be our sharpest swords in development.

4 A prototypical implementation for a source-to-source transformation tool generating cache friendly code / COOP

We have the tools, to programmatically strip down a program's records and reassemble it in a fashion that suits our needs (Clang). So it is time to consolidate our goals for a prototype. First of all, it should be fairly easy to integrate the tool into a working environment/existing toolchains. For reasons mentioned in Section 2 we can't ever expect our tool to be used otherwise. As simple as that sounds this leads to interesting design choices, we will briefly discuss in Section 4.1.
Even though the tool's scope will be limited due to being a one-man project it should demonstrate, that automated OOP to DoD data layout transformations are possible. To do so we will try to implement a Hot/Cold Split (see Section 1.3.4). Instead of completely changing the program's data layout this way we can implement a data driven optimization, that seems to be relatively easy to perform automatically. We will prove ourselves wrong later on in (TODO REF SEC) however.
Ideally we desire that the target program should provide zero additional information to our tool, so the process of transforming the target program into a cache friendly pendant won't interfere with the process of solving the problem. We will later see (TODO REF SEC) why this entails massive additional responsibility for our tool.
The tool needs to maintain the semantic integrity of the original source code. Even though changing the programs data layout will definitely affect the data access patterns (thus will actually change the programs data flow) the result must not be distinguishable from the original in other terms than performance. (TODO REF SEC) will show why this prerequisite will yet rely on additional effort.
We want the resulting program to be faster, measuring frame-times as well as cache-misses. While it is difficult to guarantee performance boosts for every possible source program we should rather aim for: Improves most programs. It definitely must not make the program slower though!

Summarized Goals

- Easily integrable in existing working environment
- Automated OOP to DoD data layout/access transformation
- Zero additional programming overhead for the user
- Improve most programs; mustn't worsen them.
- Maintain semantic integrity

From this point on we will talk about the specifics of the prototypical implementation called COOP (**C**ache friendly **O**bject **O**riented **P**rogramming) and will refer to the tool by this name.

4.1 Stand Alone Tool

Even though COOP is not aimed to be a commercially used tool, thinking of how such a technology could reach the industry it becomes clear, that nothing that requires major structural changes to the build setup or an engine's tool-chain could ever succeed. Hence even though we use the Clang front end infrastructure to implement our solution, we don't want potential users of COOP to depend on LLVM/Clang. So to start we first need to find a way to implement our solution utilizing LLVM/Clang in the right way.
There are various ways to use the framework LLVM/Clang provides. Since we are trying to improve a target programs performance by alternating parts of it, the classification of our tool fits is a *Code Optimization* [5, p. 583]. Compilers usually carry out optimization-passes either on the IR they provide or on the generated code in a machine specific way (see Fig. 10). While LLVM already comes with numerous optimization passes that are either *Analysis Passes*, *Transform Passes* or *Utility Passes*, it provides a framework to implement and register custom passes as well. However implementing an LLVM pass binds the user to the LLVM/Clang tool-chain [4]. Optimization passes are not interchangeable between independent compilers and we can't expect a working environment to change their build setup because of us.
The Clang front end functionality also provides infrastructure to access syntactic and semantic information about programs. A so called *Clang tool* can be created in three different ways. *LibClang* is a high level interface to clang. It already provides AST traversal yet won't give us full control over it. *Clang Plugins* provide full control over the AST as part of compilation. They are dynamically loaded by the compiler and can make or brake a build. However this again ties us to the LLVM tool-chain. Finally *LibTooling* is a C++ interface aimed at writing stand alone tools. It also provides full control over the AST is however subject to change and maintaining a tool based on it means continuous adaptation to new versions. [1]
When providing a stand alone tool, any build setup can adapt easily to it by invoking it manually. For example a *Makefile* could easily use COOP either before compilation or for a target of its

own (e.g. make coop).
The Clang front end functionality alone will limit us to source-to-source transformations, meaning in terms of compilers our target language equals our source language. This feels rather weird, since an optimization is usually realized as a pass and won't ever affect the source code we see in our IDEs. However an advantage of this approach is, that when the result of our tool is C++ source code, the whole bandwidth of optimizations provided by the compiler already can still be applied in a manner the compiler expects to do. We can't rely on the compiler to optimize our custom optimization pass. Also this way the optimized code remains relocatable.

But there is one major disadvantage in this approach. A pre-compile or 'source-to-source optimization pass' implies sudden structural changes to the code base. So the issue of 'losing the desired abstraction level' would just be postponed. This is irrelevant as long as the optimization is applied only before a shipping build is generated, but integrating COOP into an agile development process would only work with the help of version control systems, so it's changes can be undone easily and abstraction is only ever lost, when intended and reversible.
Speaking of agile development or any development model relying on short iterations a tool like COOP would only make sense when it's fast. As we will see later on, traversing numerous ASTs for a complete code base gets slow really (really) fast.

Since COOP will work on source files rather than on binaries we will need to present to it the files, that it is supposed to work on. There is of course always the option of manual forwarding per command line, but in favor of simplicity for example a compilation database can be generated automatically by some build tools and is therefore a convenient bearer of this information. For example when using CMAKE one can simply add '*set(CMAKE_EXPORT_COMPILE_COMMANDS ON)*' to the *CMakeLists.txt* file to create a *compile_commands.json* compilation database.
Another advantage of a compilation database is less manual overhead on COOPs integration. Files that include certain other files (like H/HPP-files) will not be processable if no information is given on where to find the included files. The tool instance, that is worked with to access Clang's functionality is in the end a proper compiler front-end that will go through each of the aforementioned steps of Lexing and Parsing and a complete set of symbols is imperative for a compiler to work.
In the end providing files manually will work, but will come with significantly more effort, so offering the option of providing a compilation database can accommodate the user to great amounts.

4.2 Automated OOP to DoD data layout/access transformation

Splitting a record's hot-/cold fields is in essence a trivial transformation when done manually and when given the set of hot/cold fields. Create a struct; move cold fields in it; create a pointer to a cold-struct instance in original record; Change all accesses to cold fields to accesses on cold-struct field pendants, respectively. This is why the Hot/Cold Split was deemed a fitting exemplary for a prototypical proof-of-concept implementation. To do this first of all we need the right AST nodes.
Comfortable access on the records and their fields is granted by appropriate AST matchers. After creating a source file's AST we can easily match against any record declaration in it and the moment we have a *CXXRecordDeclaration* node we have access to numerous helpful methods, that give us it's fields, constructors, methods, base classes etc. COOP defines a handful of matchers and callback-routines that filter wanted data (see Code 14 line 1 to 6).

```
1 auto file_match =
2    isExpansionInFileMatching(coop::match::get_file_regex());
3 DeclarationMatcher records =
4    cxxRecordDecl(file_match, unless(anyOf(isUnion(),
         isImplicit()))).bind("record_binding");
5 StatementMatcher members_used_in_functions =
6    memberExpr(file_match, hasAncestor(functionDecl(isDefinition())));
7
8 MatchFinder::MatchCallback *callback = new MemberRegistrationCallback();
9
10 MatchFinder data_aggregation;
11 data_aggregation.addMatcher(records, callback);
12
13 data_aggregation.matchAST(ASTs[0]->getASTContext());
```

Code 14: Some matchers used by COOP to filter relevant AST nodes and their utilization

The *file_match* matcher for example makes sure, we are not operating on - let alone transforming - files, that don't originally belong to our project, like system headers. The *records* matcher will give us all the records found in the compilation unit *unless* it is a union or an implicit match (see Section 3.2). By binding a matcher to a string we can retrieve the matcher's result in a callback routine. The callback needs to be implemented as a class definition extending Clang's own *MatchCallback* (see Code 15). Callbacks can then be added to a *MatchFinder* instance and finally be applied to an AST (see Code 14 line 8 to 13). The *MemberRegistrationCallback*'s overridden run method accesses the result's nodes through the string association we gave it earlier. COOP now registers the record's fields by remembering the pointers to their AST nodes. This way we will have access to the nodes' contexts at any time.

```
1 class MemberRegistrationCallback : public MatchFinder::MatchCallback {
2 public:
3    std::map<const CXXRecordDecl*, std::set<const FieldDecl*>> class_fields_map;
4 private:
5    void run(const MatchFinder::MatchResult &result)override{
6       const CXXRecordDecl *rd =
            result.Nodes.getNodeAs<CXXRecordDecl>("record_binding");
7       for(auto f : rd->fields()){
8          class_fields_map[rd].insert(f);
9       }
10   }
11 };
```

Code 15: Callback definition to register the records' members

Unfortunately even though collecting the relevant parts of our target program is fairly simple, the semantic understanding of the compiler about our fields is very limited. Even though Clang offers a vast set of methods to gather information about a record/field there is no such thing as a '*bool isFieldHot(const clang::FieldDecl* fd)*' function. The requirement for zero additional programming effort challenges COOP to endeavor in static analysis.

4.2.1 Data aggregation

One of the most challenging tasks for COOP is to identify a record's fields as hot or cold. Since we don't want to rely on the programmer to give that information to us (or even for him/her to figure it out) we need to check whether or not a field is used frequently. We need to know where, how often and together with which other fields of the same record it is used.
As a centralized point of reference COOP will construct a *record_info* instance for each record, that will hold references to all the AST nodes that are relevant to us. Besides a record's fields, the record_info instance will know about all the functions that use it's fields and all the loops that use it's fields. It is not enough to rely on the CXXRecordDecl AST node, since it will not be able to tell us where it's instances are used. For now it will function as a mere cache to our ASTs to quickly access a record's important nodes, that can be distributed all over our code base and thus be distributed among different ASTs.
To collect the data about how/where our records are utilized, we define a bunch of AST Matchers and callback routines, for example:

- MemberRegistrationCallback $\rightarrow$ registers records and their fields
- FunctionRegistrationCallback $\rightarrow$ registers functions and their member expressions
- LoopMemberUsageCallback $\rightarrow$ registers loops and their member expressions

After all the compilation units have been transformed into ASTs, the functions and loops need to crosscheck all the records, to see if they are relevant to any of them. If they work with a

records fields, they are remembered by the respective record_info. This way we can generate a matrix that encodes field-function/field-loop information in numerical values, an important step towards being able to evaluate and prioritize those relationships in a generic way. Ultimately

Figure 13: Excerpt of coops output on exemplary NPC and some arbitrary functions/loops

our evaluation faces a problem, that scales with our implementation details. Similarly to how we evaluated cache-line utilization in Section 1.3.3 for each function we could determine (estimate) its behavior for a certain Hot/Cold split in a brute force kind of way. So we can make statements about which split would be the best.

Through a function/member matrix we can determine cache utilization for each function individually. The number associated with a member expression can be interpreted as 'how much punishment would it mean to externalize me for this function'. For a simple case like Fig. 13 we could easily determine, that the function *calc* would like to have the *name, age, mood* field subset externalized. But externalizing either *pos* or *vel* would mean loading the respective cold struct instance as many times, as it's associated value. The *punishment* specifically would depend on how big the cold struct instance is, which also varies depending on whether or not each other field is hot or cold.

More formally there are $\sum_{k=0}^{n} \frac{n!}{k!(n-k)!}$ different combinations of how a record can be split, where n is the number of fields in the record and k is the number of fields to hive off. As an example we could imagine a record with 10 fields. Conclusively there are 1023 different combinations of possible splits. Checking each function lets say 50 would result in > 50.000 computations. Since we specifically regard the loops as well this number will become even higher - per record - and eventually with actual big code bases shoot through the roof.

So instead of cross checking each split scenario with each function/loop we would like to consolidate each fields 'importance' or it's 'weight' in a centralized spot, that will be checked against $n - 1$ other fields instead.

4.2.2 Metric for evaluation of field usages

We described fields to have relations to loops/functions. Expressing these relations numerically might get us into regarding existing metrics, hoping they provide information we can process.

The values assigned to a relation could be based on a lot of things, so at this point we should contemplate on what would be the most useful piece of information. Even though we will see, that the chosen metrics do not suffice our needs perfectly, they have aspects and methodology we can utilize for our purpose.

Cohesion metrics

An automated Hot/Cold Split will eventually externalize a subset of fields into another record. We do so by finding the hot data and conversely the cold data (see Section 1.3.4). When thinking about why the cold data was put together with the hot data in the first place we remember, that it might be due to our unfortunate abstraction (see Section 1.3.1). Talking about splitting records on account of their fields' relations sounds like what *cohesion metrics* try to solve.
Cohesion in a module describes to what extend, that module serves a single logical task [24, p. 172]. Its purpose is to indicate on how well a software architecture is defined. Modules with good cohesion have proven to be reusable and easy to maintain, whereas low cohesion indicates, that changes in the code will affect other parts of the code resulting in increased effort in development as well as in testing [24, p. 172].
There are several types of cohesion, that are used to classify a module, like *Coincidental Cohesion*, where elements are grouped with no logic concept for example in a utility or helper collection. This is considered to be low cohesion and should be avoided.
Logical Cohesion describes what we have found to be bad abstraction patterns coming with OOP. We group logically related elements, because they share a context. Consequently we will collect lots of fields, that belong to different domains. Cohesion metrics also recommend to avoid this kind of design.
Temporal/Procedural Cohesion both describe a grouping of fields, because they are processed at the same time/in a certain order. So basically when they share temporal locality. Even though we discussed earlier (see Section 1.2.4) that trying to design around principles of locality is one of our main goals in DoD, thinking in an OOP way this is rather bad, because it might promote monolithic class- and method definitions. Again cohesion metrics want our record definitions to follow a single task. This level of cohesion is considered to be acceptable but not ideal [24, p. 174]. But DoD doesn't argue here actually. Temporal/Procedural cohesion think in a bigger scale than what we meant earlier with temporal locality. For example grouping independent elements because they all are related to system startup/cleanup even though they don't interact. This is another example of where we find a big gap between OOP and DoD at first glance but they actually conform with each other for the most part, only differing in motivation.
Working our way up to the notion of an *ideally* cohesive model, we pass a few other levels until we reach *Functional Cohesion*. This level describes a module to group elements sharing a domain. The module will therefore serve a single purpose and changes to it won't affect code of other domains. Note that OOP very well allows for good abstraction, but again the inherent problem we face is that our intuitive abstractions don't accommodate neither software design nor our hardware. More importantly this definition of 'desirable cohesion' fits our needs to im-

plement a Hot/Cold split, so cohesion metrics might bear the right tools, to accommodate us.

There are a bunch of metrics like *LCC* (Loose Class Cohesion) and *TCC* (Tight Class Cohesion)[7, p. 3] or the famous *LCOM* (Lack of Cohesion in Methods) [6, p. 25]. Since cohesion metrics operate on object oriented code, they usually work with methods, as groups of field subsets. For example the LCOM defines a module's cohesiveness to be the "*number of pairs of methods operating on disjoint sets of instance variables, reduced by the number of method pairs acting on at least one shared instance variable*"[22, p. 8].
While cohesion metrics have a striking similarity in their procedure (splitting records according to their field usages), unfortunately they do differ in their intention and result. As for our Hot/-Cold split, we intend to be a performance optimization and are ready to group any fields, that share spatial/temporal locality. Ultimately our Hot/Cold Split will usually automatically divide a record into domain specific sub sets, its focus however is to minimize a record's stride for cache utilization. This means we could easily end up externalizing a domain related field for the sake of faster computation.

But all is not lost, because cohesion metrics provide us with proven methodology to identify and evaluate relations in a module (record) in numbers we can compare. We also learned, that to better fit our purpose, a target metric should regard our code's performance and/or size (memory stride).

Asymptotic Notations

An asymptotic notation or more famously O-notations describe an algorithm's complexity [14, p. 44]. It is not a measurement tool to actually evaluate the performance or memory size an algorithm uses since these depend on hardware/architecture/compilers. It describes how an algorithm scales depending on the problem size and defines upper-/lower borders for it's (asymptotic) growth depending on which notation is used. Hence the name because it deals with the *order* of an algorithm. There is for example the Θ-notation that expresses asymptotic upper- and lower bounds for a given procedure. Specifically the Big O-notation (or Landau's symbol) describes the worst-case for a procedure (asymptotic upper bound). Dealing with bounds makes sense because depending on the input (problem size), performance as well as needed memory space might vary drastically. Using the O-notation we can get estimations about a functions running time only by looking at its overall structure [14, p. 47]
An actual static performance analysis of code would break down the instructions to those we can find in the hardware's instruction set (depends also on the compiler), to get a grasp of how many cycles they need and ultimately how a cycle translates to (probably) nanoseconds.
The O-notation in its essence will also look at the instructions a procedure makes, when looking at the code. However it builds terms by evaluating control flow statements and eventually will omit constants and all but the highest order term. Expressed as a polynomial $g(n) = 5n^2 + 3n$ would translate to $g(n) = O(n^2)$. Since $g(n)$ is of order n^2 the equals notation is not perfectly

correct but commonly used. Leaving out constants and low order terms is due to their insignificance considering large n. After all it describes asymptotic behavior.
In the best case a procedure is of order $O(1)$, which means no matter how big the problem becomes it won't affect our performance further. This is the case for each procedure, that operates on a fixed amount of parameters (for example typical getters/setters). While even a setter can consist of several instructions, lets say for example 3, it would translate to $3 \times O(1)$. After omitting the constant 3 we are left with $O(1)$. It is referred to as *constant* growth.
Loops oftentimes iterate over dynamic ranges, so when a loop is operating instructions n times it is denoted as $O(n)$ or has *linear* growth.
Nested loops are often denoted as $O(N * M)$, where N and M represent the iterations for the outer- and inner loop. In cases where N and M are equal we refer to it as *exponential* growth or $O(n^2)$.

For our use case classifying our functions like this could be useful. After identifying the relevant functions (those that use our records' fields), we could evaluate them by determining their order. We could see which functions are the 'slowest' and thus reduce memory stride on the records they utilize. This could work by simply defining each field, that is used in those functions as hot. Functions that are known, to be slow, could now operate on hot data exclusively.

Unfortunately this bears some problems. A high ordered function might interact with our records very briefly (or not at all), yet would be considered a criteria for deciding which fields are hot/-cold. This alone illustrates why we can't blindly apply asymptotic bounds as a criteria. We have to adapt it to our motivation, by only considering or prioritizing instructions, that are (or imply) field usages, yet again only considering our fields might falsify a denotation of the function.
There often won't be the one slow function, the *single point of bottleneck*. Identifying a functions members as hot, in order to speed up that function might work, but might just as well worsen each other remaining function. Whether or not a field is to be considered hot has to be determined individually with their groupings (uses in functions) as a relevant indicator rather than a decisive factor.
Omitting constants and low order terms might provide fair estimations for large n, but not every program operates on huge amounts of data. Of course software, that doesn't depend on its data layout very much probably won't significantly benefit from a Hot/Cold split, but lower order parts of a procedure might affect the performance enough for them to deserve to have a say. Also they might be useful deciding factors in close calls.

Again all is not lost. Asymptotic notations provide us with useful policies to determine a procedure's complexity. We can adapt the way it reflects on constant; linear; quadratic etc. growth, to derive an evaluation for the fields it uses.

Quantitative cohesion alternation

```
1   void inc(Foo &foo){
2      foo.age += 1;
3   }
4   ...
5   void p2(Foo &foo){
6      foo.bar *= foo.bar;
7   }
8   ...
9   bool gt(Foo &foo1,
10             Foo &foo2)
11  {
12     return
13        foo1.bar > foo2.bar;
14  }
15  ...
16  for(Foo *foo : all_foos){
17     inc(*foo);
18  }
19  ...
20  for(int i = 0; i < N; ++i)
21  for(int o = i+1; o < N; ++o)
22  collision(foos[i], foos[o]);
```

Code 16: Exemplary pseudo–ish code

Now that we have found procedures, solving our problem partially we might be able to derive a fitting methodology. A first approach could be to try to evaluate field usages (non method member expressions) similarly like we would evaluate statements for an asymptotic notation. As a simple start we could count the amount of member expressions per function. Considering something like Code 16 the *inc* function in line 1 could be evaluated easily. We have one member expression *foo.age*.
When looking at the *p2* function in line five we notice that our scheme would now rate p2's relation to the field *Foo::bar* as a 2, since it occurs two times. Hence the field *Foo::bar* would be considered *hotter* than the field *Foo::age*.
But do we want this behavior? Cohesion metrics like LCOM don't count all field usages of a specific field for one method, but group them in sets for each method instead. This way it is not about which field is the most prominent one, but which fields share related context. Even though it is a true observation, that bar is used more often than age (at this point), considering the cache there is only one *Foo::bar* to work with. There is however an important difference to cohesion metrics we need to consider. Even though we are interested in domain relations (field groups) our cache utilization highly depends on which fields are loaded most frequently. The *gt* function (line 9) on the other hand uses two distinct *Foo::bar*s (assuming strict aliasing). *gt* will actually be responsible for loading two addressable units into the cache. A reduced stride between relevant data could be more efficient here (this situation should be prioritized over *inc/p2*). Depending on how big an actual Foo instance is and how many distinct *Foo::bar* fields are accessed in one function counting *distinct field usages* might be a more helpful evaluation of a relation. This is a cache conscious compromise between detecting domain relations, but prioritizing load frequency.

The access patterns that start to make things spicy for the cache however are usually loops. When looking at line 16 to 18 of Code 16 the for-loop iterating an arbitrary number of Foos might outclass any simple function, even if it is using dozens of distinct fields - hard coded. There is only one field usage to see, yet it could constitute numerous instances. That is why asymptotic notations evaluate loops like these with *linear growth*.
Even though a loop's amount of repetitions can sometimes be evaluated at compile time, in OOP where objects tend to be created on the heap and their containers are extended dynamically there is no real telling just how much distinct instances of field usages there will be. So we know, that loops might be game-changers for our evaluation, but as a matter of fact we

can't ever predict a perfect number of recurrence. So either we rely on helper indices/iterators, make a few test runs to log and memorize their peaks, or we try to find a reasonable estimation. Like the O-notation we might remember loops and associate them with an arbitrary factor *n* for now, but for an actual comparison of the fields, later we will need to decide how to rate them.
This estimated value we are going to associate with a field usage inside a loop could be based on experience but the best we are ever going to get out of experience is "*entirely depends on the use case*". A better way of evaluating a loops impact is in a way for it to not break our scale immediately, but allowing it to do so. Whats meant by that is the fact, that associating a field usage inside a loop with a number that is vastly higher than that of a function, will render our functions meaningless quickly. However the moment we start nesting loops inside each other (allowing for *quadratic/polynomial growth*), calling single functions impact-less might actually be true. Besides opposed to asymptotic notations we won't rigorously drop lower order terms, we will just make sure, that an expression is allowed to be much more significant.
Facing reality nested loops hold the greatest potential for bad cache utilization, since they can repeatedly load irrelevant data on several iterations. Our scale will definitely end up starting at a low level listing single function accesses on data then at some point rapidly going up where nested loops hang out together. Line 20 to 22 in Code 16 will quickly outperform other control flow statements and while a nested for loop like this is no rarity the *loop depth* of a statement, can be arbitrarily high, depending on the situation.

In terms of implementation, COOP remembers each function and each loop individually, as well as the member expressions they contain respectively. After data aggregation with the matchers and their callback routines we can create function/member and loop/member matrices like in Fig. 13 for each record.
Just like cohesion metrics we can use field subsets used in functions to declare contextual relation, assuming temporal locality through AST node familiarity. Meaning each row (function) embodies a relation between the fields it uses (see Fig. 14). Each value inside such matrix stands for the computational *weight* of that field for the function. A column represents a field's distribution among the functions. Totalizing a column determines the field's overall weight that will eventually be used to compare it to the others. Fields' relations to each other need to

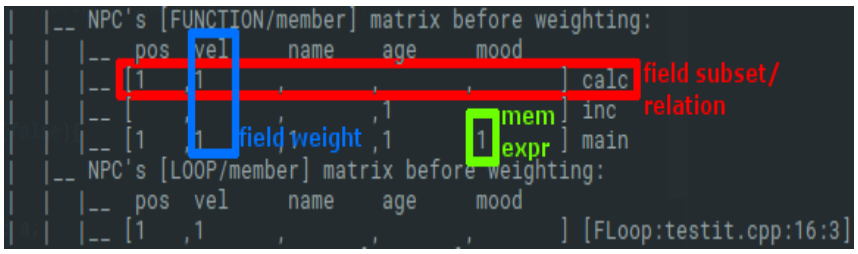

Figure 14: Relations depicted in function/member matrices

be valued, too. After all their correlation will work best, if they end up as hot data together. The aim is for their temporal locality to result in improved spatial locality. Each group of field

usages (function) needs to somehow weight its collective field impact, to heighten their chance to stay together as hot data. We prioritize small groups, since externalizing more fields (with low usage frequency) will result in less stride. Functions will therefore compete against each other, by adding to their field weights the overall number of fields minus the amount of fields they utilize. This scales with the number of fields a record has. As mentioned before instead of brute forcing our way through each possible split combination we now encode each field subset relation numerically so eventually it will influence a field's overall weight. It is important to form a decision over an overall weight because unlike an optimization for a specific algorithm, changing the programs data layout has the potential to affect ALL of its functions.

4.2.3 Field weight heuristic

When we are able to define a field's overall weight on a program, whats left to do is finding a delimiter, that actually divides the field set in two subsets, depending on those weights. This again is not a trivial operation and first of all we need to define what we intend to separate.
As can be seen in the Figures 15 to 20 we will face numerous different situations that result of arbitrary access patterns. While sometimes it is easy to rule out certain fields for others it is not. A generic set of rules to handle this needs to be able to process special cases while not losing credibility for common ones. It can do so by scaling with the problem.

Scaling delimiters

Scaling delimiters will adjust automatically as the problem changes. In a lot of cases a very easy heuristic will work comparably well. For example a (we will call it) *max/2* where we will just take the maximum field weight and divide it by two. Everything above the *max/2* will be hot and vice versa. Given our 'scope' is the maximum field weight this will yield very good results for a lot of cases, however the more significant the spikes are, the worse will be the affects on the resulting data layout. While the *max/2* regards quality it dismisses quantitative scaling.

A surprisingly easy yet effective candidate is the average, because it scales up when certain fields tend to be a lot more important than others and naturally divides our values according to their relative weight (see Fig. 15) unlike for example their median value. Unfortunately Fig. 16 shows, why sometimes a simple average determination might not be the best fit. If *field_a* or *field_c* were slightly less important, *field_b* would probably be considered hot, yet this way, even though it's rating is far from *impactless* it is considered to be cold.
On the other hand Fig. 17 shows why an average will also not scale well with the amount of fields in a record. The average narrows as the divisor grows, so on a record with a hand full of members an average might end up introducing fields to the hot subset that barely pass the average weight.

This leads to an interesting dilemma. Where to draw the line between hot and cold fields?

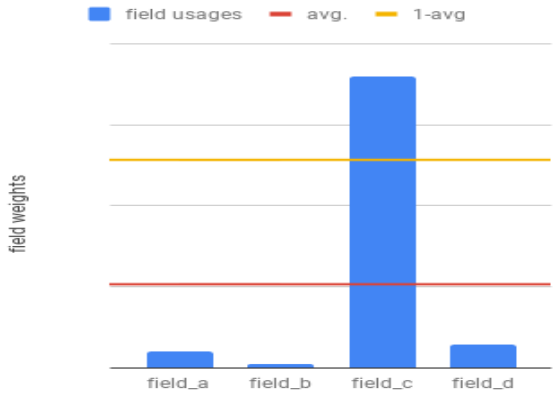

Figure 15: Good avg scaling.

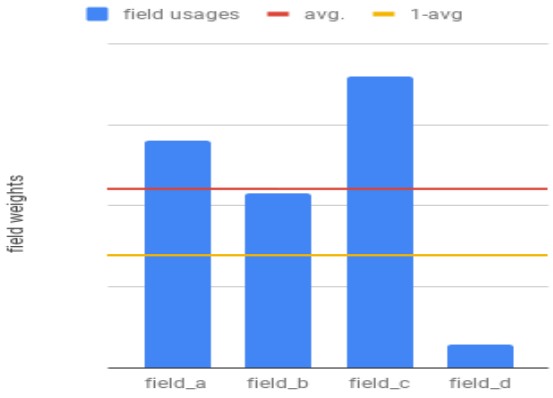

Figure 16: Difficult evaluation for avg scaling.

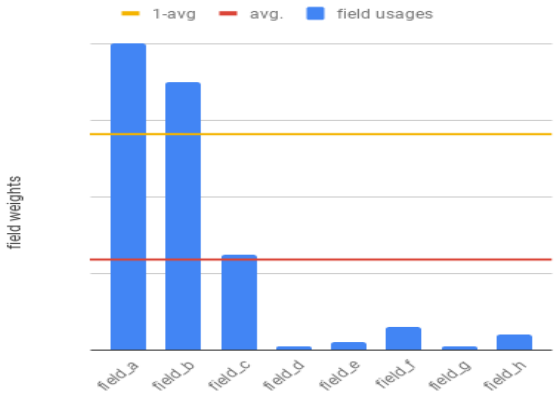

Figure 17: Bad avg scaling with more fields.

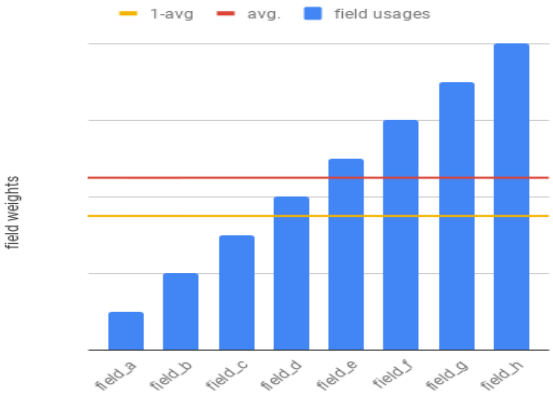

Figure 18: Problem of even distribution.

Should there be constant *magic numbers* defining the threshold of a hot field? Since different access patterns can produce arbitrary field weightings there is no good way of predicting a constant, that suffices our intention. But how can relative proportions work when they introduce false-positives? And can we get rid of them? Fig. 18 illustrates a difficult case that will in practice rarely occur but embodies our problem perfectly. An even distribution of field weights allows for no logical grouping of significance, at least in terms of 'drawing the line'. The average as a heuristic fails us in many situations. We referred to it because it provides a quick approximation of a good delimiter. The factors however that determine its scaling are contrary to the paradigms we follow. Number-of-fields as a divisor means decreasing averages with increasing amount of fields. This means the more fields our records have (consequently the more lack of cohesion), the higher the collective chance to be considered hot. Also the average behaves poorly towards well designed records. Consider Fig. 19 where *field_a*'s weight is just slightly higher than the others'. It will deem all fields but *field_a* cold and probably ruin the data layout.

The above diagrams propose another heuristic that we will call the *1-avg*, represented by the yellow lines. It introduces the reciprocal counterpart for the averages bad scalings. By taking the greatest field height minus the average, our tolerance for fields grows linearly as the average

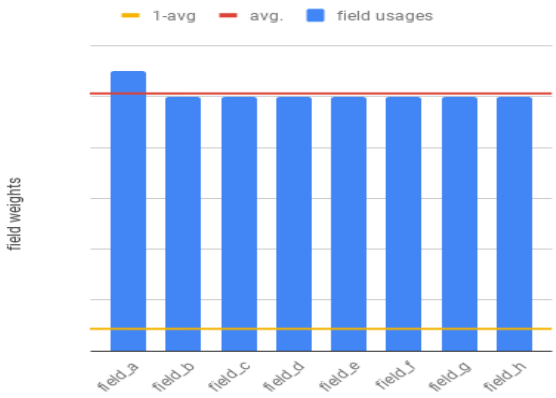

Figure 19: Bad avg homogeneous field weights.

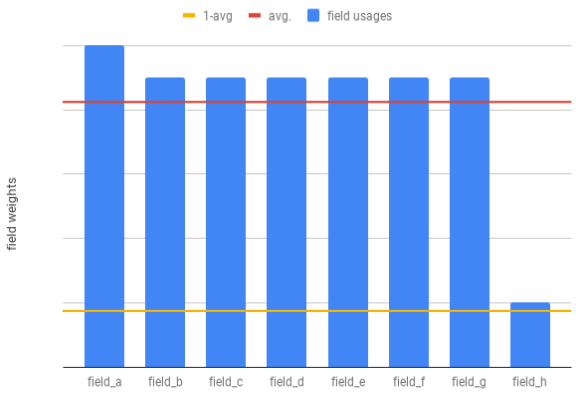

Figure 20: *1-avg* prone to false positives as well.

rises while regarding the overall scope of our field weights. In other words: The more quality we find in a record (the more the average trends towards the maximum field weight) the more fields we allow to be hot.
As can be seen in the above figures, this heuristic behaves much better for records that show great differences in their field weights. Of course it is very much possible for it to behave poorly in specific situations as well, again coming from spikes. Fig. 20 shows, that the reciprocal average quickly becomes too tolerant. We have seen, that *1-avg* is able to correct some mistakes the average makes but it introduces unwanted behavior on its own.

Combined scaling delimiters

An interesting take is on how to combine certain scaling delimiters. Depending on the case different heuristics will result in drastic deterioration of the data layout. We could try to get rid of errors by cherry-picking the strengths different heuristics provide.
One possibility could be to adapt the *max/2* heuristic. We can determine both the *avg* and the *1-avg* delimiters, take the greater one and divide it by 2 (we will call it *top/2*). The upper delimiter will always separate the most significant fields for us. As the *max/2* easily gave us a quick way of defining a tolerance towards lower significant fields, it did not in any way regard the fields' quantity as a defining factor. By possibly considering the *1-avg* (when it is greater) we adapt it given the right situation. This already yields improved results, is however still prone to special cases (see Fig. 21). This is because we don't consider the possibility of the cold remnant to be relevant in its entirety. The problem with scaling delimiters is, that no matter how much we narrow down the error by applying a certain heuristic, in another distribution the same approach might behave badly in terms of handling another error source. We need to evaluate subsets of fields ordered by significance to reliably rule them out or keep them collectively.

Regarding both the *avg* and the *1-avg* is interesting because combined they are able to cate-

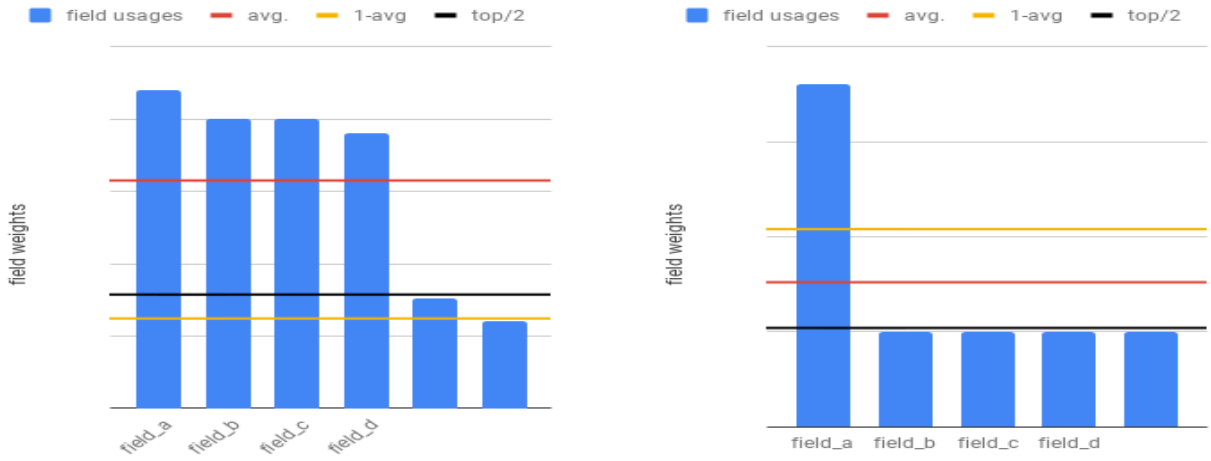

Figure 21: Improved *top/2* heuristic as it is able to rule out *1-avg* errors but still not well.

gorize the field weights to a certain extend. As mentioned before, there are two major scaling factors the fields' weights and their number. As the *1-avg* is the reciprocal of the *avg* we can derive information about a programs access patterns by looking whether *avg* or *1-avg* is greater than the other.

When the *avg* is greater, the fields' weights is proportionally more significant than the record's number of fields. When the *1-avg* is greater on the other hand, it means that proportionally there are more insignificant fields, than the subgroup of 'important/hot' fields. Well designed records will demonstrate $avg \approx f_{max}$. Anyhow we can interpret those two delimiters as an order of significance. They divide our scope into three spaces. One above the greater delimiter, one below the smaller delimiter and whatever is in between them (see Fig. 22). Fields in the upper space are certainly to be considered hot. Fields below the lower line are in a less significant order. Whatever is in between is what we can not categorize immediately. The idea is to now

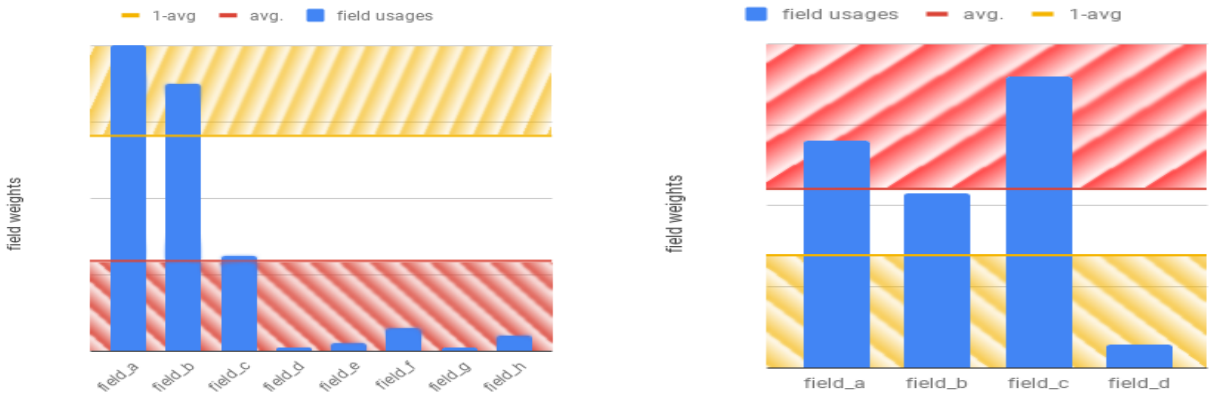

Figure 22: Field weight categorization by combined scaling delimiters. Top hatched space is of high significance.

recursively apply the ordering between the *avg* and the *1-avg* delimiters as long as we have

fields, that we can neither classify as high- nor little significance. This approach will work fine with data sets, that exhibit high significance varieties. On uniformly distributed field weights it will tend to include the whole field set for the hot data, yet since we encoded logical relations in the field weight a distribution like in Figure 18 bespeaks of defective access patterns rather than only bad data layout.
Regarding order of significance is a good approach to a generic solution, yet our recursive approach is not optimal since it is based on scaling delimiters, that are unaware of a field-group's collective impact. Also trying to break down a record's significance order into two (hot & cold) subsets immediately mitigates precision.

Order of significance / Significance groups

The problem is that until now we tried to divide a record in two subsets. This is actually what we want, but the truth is there can be an arbitrary amount of related subsets in the record's data fields and the more significance groups we have and the greater the difference in significance, the blurrier the delimiter becomes. Delimiters based on either quantitative or qualitative scaling introduce an error of a size we can hardly reason about when compared to the entirety of field weights (see Fig. 23). Percental tolerance factors will also always just move the error resulting in better behavior in some situations and worse in others.
Also one of the worst situations for us is to accidentally separate fields, that are logically related. With scaling delimiters it can easily happen, that two fields, that share an order of significance (due to our metric) are separated because the delimiter ever so slightly includes one of them but excludes the other. Whenever we split fields that are codependent we ensure worsened cache utilization, because the functions that use the hot field will most likely also use the cold field and will now have the additional overhead of the indirection to the cold struct instance. Our Recursive approach tried to solve this issue and will succeed in simple cases, yet ultimately it depends on the blurry scaling delimiters.

Scaling delimiters try to evaluate field weights individually, yet their sub-/optimal utilization strongly depends on their significance group. Determining whether or not a field should be considered hot should depend on the benefit/punishment of it's extraction. Externalizing a field always means that the remaining hot fields load less unnecessary stride into a cache-line, yet it also means whenever the extracted field is used unrelated cold data might be loaded as well. Cache-lines have concrete sizes (e.g. 64 Byte) so at this point we might be tempted to just multiply a field's size in byte to it's weight. This way we could see the fields impact on the cache capacity and evaluate whether or not determining a certain field as cold would imply great punishment. But this way the least frequently used field could suddenly be considered hot if it is just big enough. We may not confuse capacity with usage frequency, which is the actual criteria of a Hot/Cold Split Section 1.3.4. So it is actually enough to compare our field weights as a criteria for benefit and punishment. However in order to determine a significance group's impact on out program we are definitely interested in comparing its size to the cache-line size in order to

deduce possible stride (which we want to reduce after all)

But how can we determine significance groups? COOPs approach is to sort the fields according to their weights in a descending order and measure their weight differences. Normalized on the scope (f_{max}) we can compare them to the average deviation. Each value above the average will denote a new less significant group that spans until the next group is identified (see Fig. 24). The important thing is for the fields to be evaluated in terms of groups rather than individually.

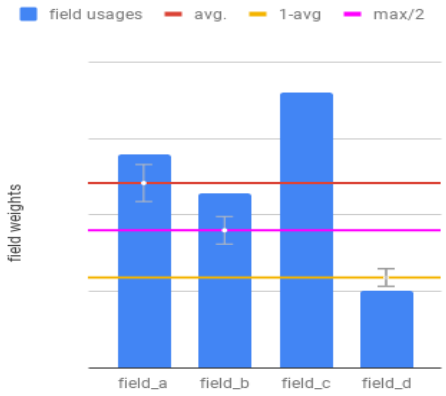

Figure 23: Scaling delimiters' errors can hardly be reasoned about and provide equally much punishment as benefit depending on the distribution.

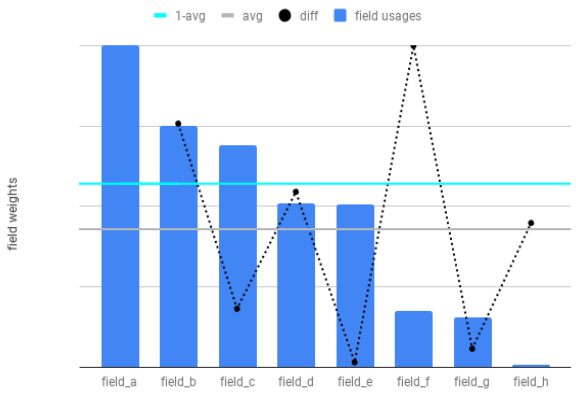

Figure 24: Determination of significance groups with field weight deltas. Normalized differences are projected on the field weights' scale for visualization.

How to evaluate a group though? First we need to determine the group's benefit/punishment in case of a split. Each significance group g_i has a type size s_i which is the summarized type sizes of it's fields, as well as a max_i which is it's highest field weight. With the cache-line size CLS, the general cold struct indirection overhead H and with n as the amount of significance groups, when a group g_i is extracted, it implies the following:
Extracting g_i means s_i less stride for the remaining hot data. We already ordered the fields' weights in a descending order, so now we can compare their type sizes with the additional space and as soon, as one or more of the hot fields fit in the additional space (minus H) we will effectively have reduced the number of cache-lines to fit 1 hot data instance by the factor $\frac{s_i}{CLS}$ (without splitting an addressable unit over two cache-lines). So the benefit will express through the re-utilization of the hot fields which is capped to the highest field weight of the hot fields (F_{max} or max_0). We can value it by multiplying the F_{max} with the determined benefit. This practically tells us how many cache-lines we will spare our important procedures.
But we can make our estimation a little better by interpreting each significance group as a representation for those procedures (loops/functions), that have contributed to it. So instead of capping our 'savings' estimation on the most significant value (F_{max}) we will take each significance groups maximal field weight, as this value represents the groups estimated loads. So

our savings $s(g)$ are:

$$s(g_i) = \sum_{k=0}^{i-1} max_k \times \frac{-H + \sum_{k=i}^{n} s_k}{CLS} \tag{1}$$

At this point our procedure will cut off everything until the last group of highest significance, because we have not yet determined the 'costs' of externalizing a significance group as a counter weight. As the stride is reduced for the hot subset it is increased in the cold subset. The payoff for a hot set is the additional indirection over the pointer to the cold struct instance. When applied correctly this is however comparably little as we plan to externalize greater means.
The cold data will always be accessed by going through the hot data (the pointer to the cold data is in the hot data). So whenever we are accessing cold data we automatically waste one cache-line per cold struct instance that is used to load and dereference the pointer to it. In terms of cache utilization this is really bad (not only in terms of capacity, but also because of possible thrashing since the hot and cold data lie at physically different locations (see Section 1.2.4)). Its only worth because we do it with data, that is accessed so rarely (relatively), that this cost is lower than the benefit we get by splitting it from the hot data. However, besides the additionally wasted cache-lines used solely for finding the actual data, we will increase the cold data's stride by s_i Byte, resulting in an estimated iteration overhead $o(g)$ of

$$o(g_i) = \sum_{k=i}^{n} max_k \times (1 + \frac{\sum_{k=i}^{n} s_k}{CLS}) \tag{2}$$

cache-lines for processing the cold data in its entirety. Ultimately for each g_i we say that a split is worth $w(g)$ when:

$$w(g_i) = s(g_i) > o(g_i) \tag{3}$$

Then we want to externalize it because the benefit is greater than the cost.

Unfortunately this will only work for tightly packed arrays of data, since we assume, that reduced stride immediately results in improved cache utilization. If we implemented an automatic SOA transformation this would be the way to go (see Section 1.3.3). Due to alignment issues (more on that in TODO REF SEC) COOP will allocate the hot and cold data in a way that is not exactly packed, but will consider alignments, so to a certain extend a reduced stride will have virtually no effect other than the additional indirection to the cold data - invalidating our progress so far.
To be more specific COOP will regard the target systems L1 D (configurable depending on optimization preferences) cache-line size and will try to pack as many elements into a line, yet align each entity group to the cache-lines, to prevent unnecessarily much entity splits over too many cache-lines (we briefly discussed this in Section 1.3.4). This means, that when our hot subset size is smaller than the target cache-line, reducing the stride will have effect, as soon as it frees enough space for the cache-line to encompass another instance (see Fig. 25).

On the other hand, if the hot subset size is greater than the cache-line, reduced stride will have effect, as soon as it results in reducing the number of cache-lines necessary to encompass an instance (see Fig. 26). So actually since we will introduce padding due to our alignment we might even want to consider keeping poorly ranked fields, as long as they only replace space, that otherwise would be used for padding. Again this needs proper evaluation, since at this

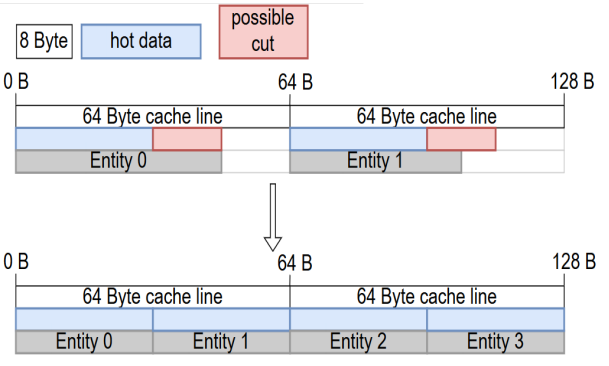

Figure 25: Aligned entities will be packed inside cache-lines. Reduced stride will be effective, as soon as it increases the amount of entities inside a cache-line.

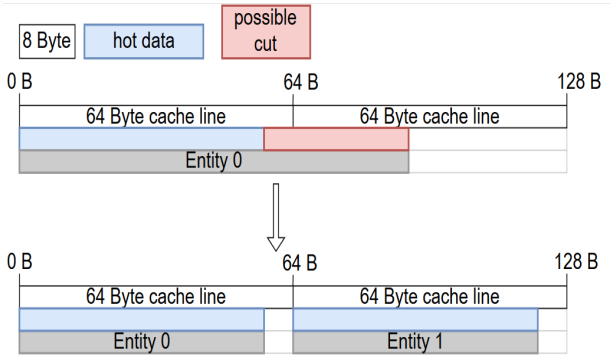

Figure 26: Reduced stride will be effective as soon as it reduces the number of cache-lines needed to encompass an entity. Entities are split upon minimum amount of lines.

point our procedure will blindly cut out fields because it can't reason about the splits impact to it's full extent. We again need to include our field weights into the equation, to be able to evaluate whether or not a split will result in more benefit, than punishment. More formally for a significance group g_i out heuristic $W(g)$ will consider a split worth when:

$$W(g_i) = \begin{cases} \left\lceil \frac{CLS}{\sum_{k=0}^{i-1} s_k} \right\rceil > \left\lceil \frac{CLS}{\sum_{k=0}^{i} s_k} \right\rceil \wedge w(g_i) & \text{for } \sum_{k=0}^{i-1} s_k < CLS \\ \left\lceil \frac{\sum_{k=0}^{i-1} s_k}{CLS} \right\rceil < \left\lceil \frac{\sum_{k=0}^{i} s_k}{CLS} \right\rceil \wedge w(g_i) & \text{for } \sum_{k=0}^{i-1} s_k > CLS \end{cases}$$

It is worth to note here, that this consideration of the groups type sizes is another kind of scaling that we haven't had considered before. This heuristic will yield different results depending on the actual type sizes and will therefore be more precise in its evaluation.

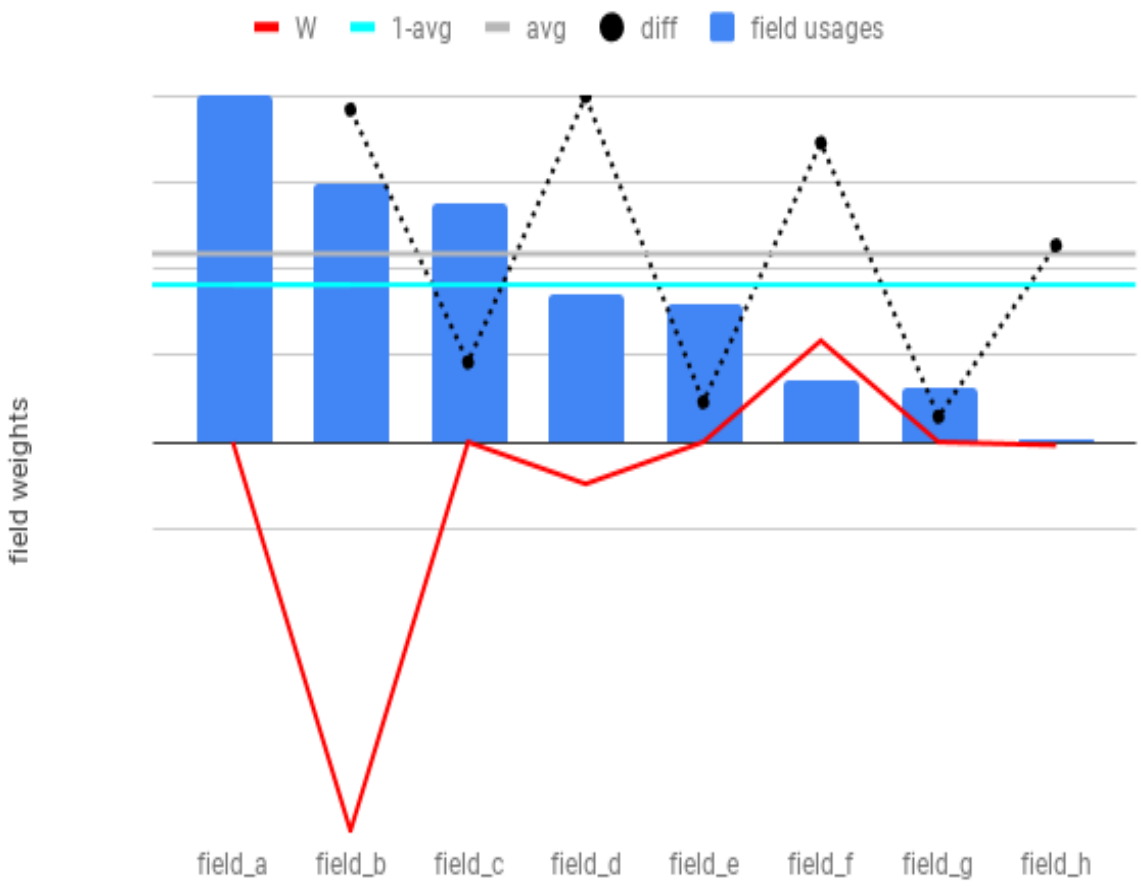

Figure 27: Exemplary field weights evaluated by our heuristic with the result, that its worth to make the split [*field_a*, ...*field_e*], [*field_d*, ...*field_h*]. $H = 8$. the fields' type sizes are all 8 and the *CLS* is 64.

Bibliography

[1] *Choosing the Right Interface for Your Application*.

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

[3] *TIOBE Programming Community Index*.

[4] *Writing an LLVM Pass*.

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

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

[7] BIEMAN, J. M. and B.-K. KANG: *Cohesion and reuse in an object-oriented system*. ACM SIGSOFT Software Engineering Notes, 20(SI):259–262, 1995.

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

[9] 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.

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

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

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

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

[14] CORMEN, T. H., C. E. LEISERSON, R. L. RIVEST and C. STEIN: *Introduction to algorithms*. MIT press, 2009.

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

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

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

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

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

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

[21] HILL, M. D.: *Aspects of cache memory and instruction buffer performance*. Techn. Rep., CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 1987.

[22] HITZ, M. and B. MONTAZERI: *Measuring coupling and cohesion in object-oriented systems*. 1995.

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

[24] INGENO, J.: *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd, 2018.

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

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

[27] LATTNER, C.: *LLVM and Clang: Next generation compiler technology*. In *The BSD conference*, pp. 1–2, 2008.

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

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

[30] 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.

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

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

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

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

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

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

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

[38] 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