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 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, April 11, 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, April 11, 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 [19, 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 [33][42]. 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 [26]. This makes sense, since tree like data structures (e.g. GUIs) containing entities with shared behavior can easily be described with 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)*" [37, p. 4] the general concept of objects usually includes derivation and extending base behavior.
OOP quickly established and rooted itself in the industry without solely being used on graph-

ics 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 [20, 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

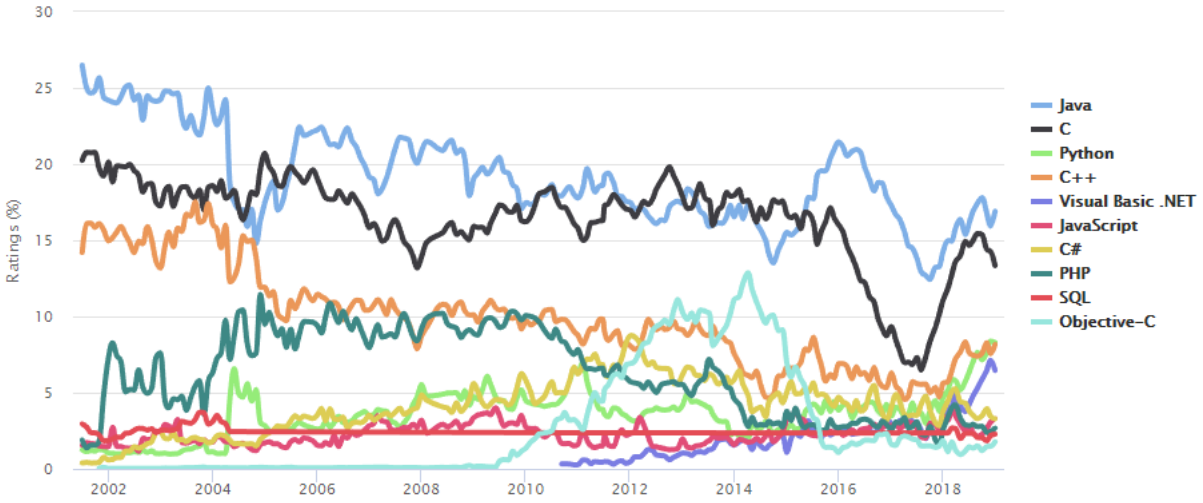

Figure 1: Most popular programming languages throughout the years (Source: [3]).

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 or domain affiliation. 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 where oftentimes lots of game entities are being processed several times per second: "*data is performance*" [39, p. 272].

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

NPC npc_arr[3];
```

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

```
struct NPC : Human {
   int mood;
};
```

Code 1: Example of some hierarchical POD class definitions

The abbreviation *AOS* stands for *Array Of Structures*[15] 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 -

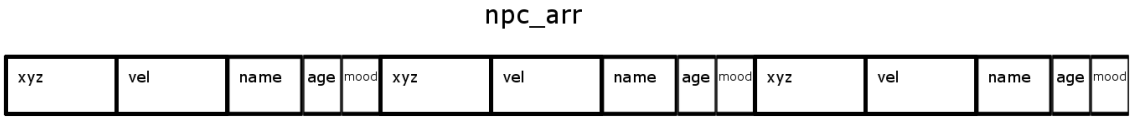

Figure 2: Visualization of how *npc_arr* will exist in memory

hence the notation *AOS*. The following sections 1.2 and 1.3 will elaborate on *why* exactly this way of thinking/way of abstraction and its respective layout in memory 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.* [35, p. 35]
>
> *Keep it simple and smart* (KISS principle). [38, 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 hides 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.[35]
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 [30, 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[...].* [22, 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.* [21, 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).

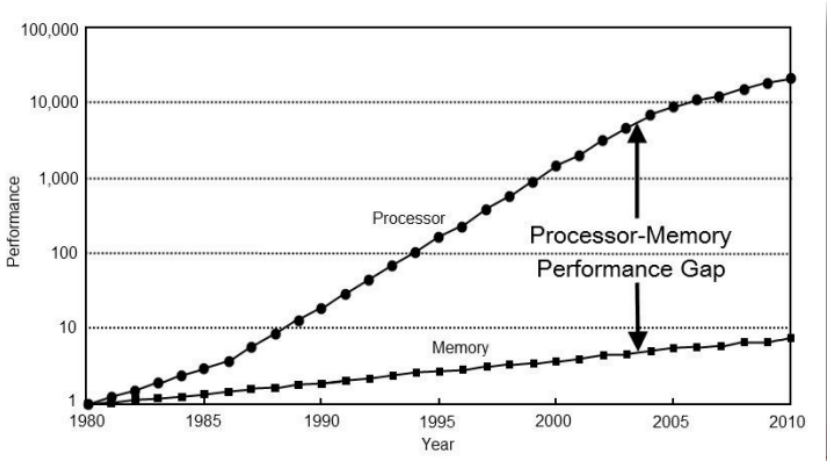

Figure 3: "***Starting with 1980 performance as a baseline, the gap in performance between memory and processors is plotted over time****. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with 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: [22, p. 289])

To solve the issue of ever diverging CPU/memory performances (commonly referred to as the *memory gap [21]*), 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 [14] in 1969. Originally cache technology was mentioned as *buffers*[17]. 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 [21, p. 191]. This principle originated in the work on *Virtual Memory* [17, 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 i7

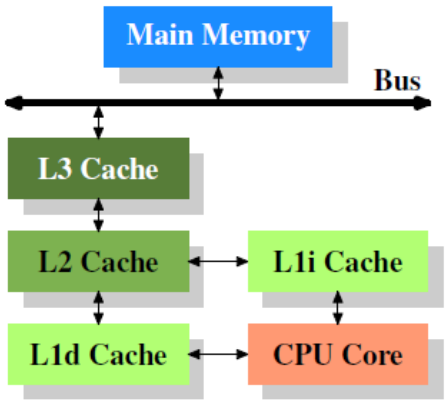

Figure 4: Exemplary, simplified model of a CPU core and its several cache modules (Source: [18, p. 15])

Processor's L1 cache for example is almost as fast as accessing it in the very CPU's registers - 4 cycles (2.1 to 1.2 ns) [32]. For reference access to main RAM "*can take on the order of 500 cycles[...]*" [21, 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 as the overlying smaller and faster cache-modules may not work on diverging information.
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 [17, 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

While optimizing against better instruction cache utilization can result in significant performance boosts (see section 8.3), 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. [19, 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 [21, 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 [24, p. 50]:

- Compulsory Miss - Access to previously unreferenced data blocks. E.g. 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 accesses 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: [21, 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 [21, 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 [17, 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*[17] as well as Jason Gregory's *Game Engine Architecture*[21] 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: [17, 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 [...].*" [11, 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: [11, 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 [11]. 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 [36, 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.* [34, 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 [29, 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 [...].* [19, 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, xyz[3];
4    Shader *shader;
5 };
6
7 struct particle_system
8 {
9    particle particles[1024];
10   unsigned particles_alive;
11   ...
12 };
```

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. 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[19, 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* [19, 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

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

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, maintenance, future proofing, extendibility, and reuse.* [19, 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 [13]. This is done because laying out your data in a linear fashion goes away from embedding meaning to its structure (Class/Struct) [19, 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 [19, 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 [19, 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) encouraging us to implement domain wise processing of an entities attributes.

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* (SOA) is the *column-oriented database* pendant to a data layout that results in higher throughput by providing a cache friendlier structure [19, 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

```
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

$2 \times 3 \times$ *sizeof(float)* $=$ 24 Byte, we waste $40 - 24 =$ 16 Byte of our cache per NPC. Due to our algorithms complexity (see section 4.2.2) this waste scales linearly with our data.
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-

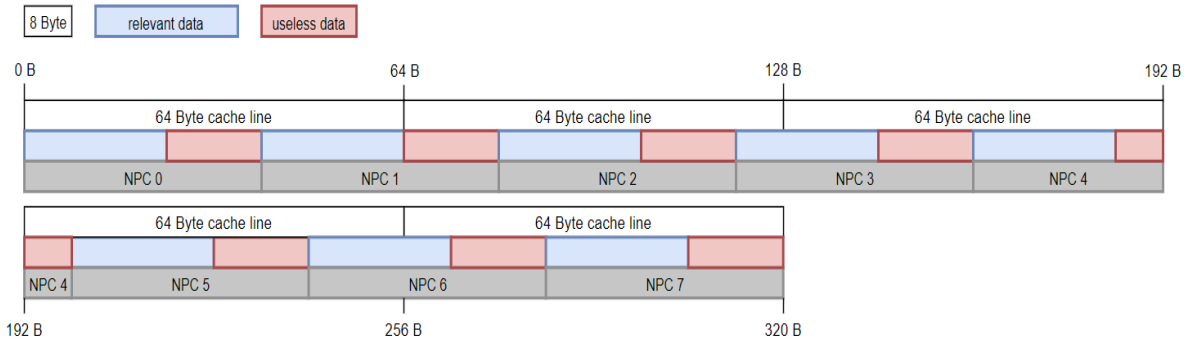

Figure 5: NPCs inside cache-lines, where blue is relevant data and red blocks represent unused data

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

```
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

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.

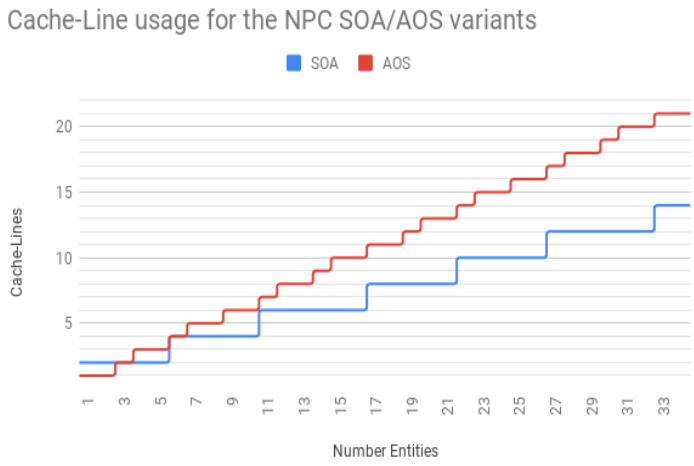

Figure 6: Cache-line efficiency comparing NPCs represented as SOA and AOS.

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 [15, 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 section 4.3.2, 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

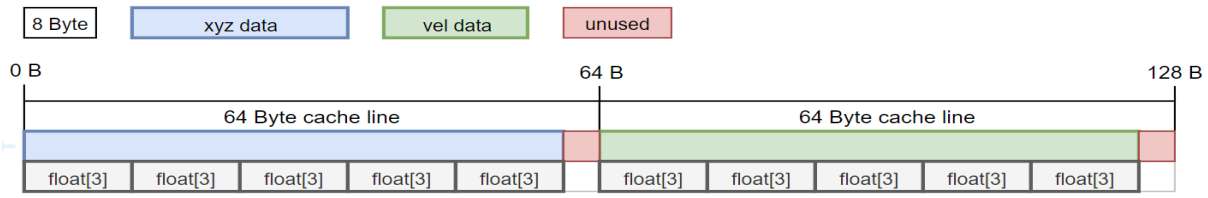

Figure 7: xyz and vel blocks inside cache-lines, where blue represents joint float[3] blocks of xyz data, green joint blocks of float[3] vel data and red is unused but intentional padding.

optimal. By not loading unneeded data into the cache we can store more relevant data. By Aligning and padding our data blocks correctly we attenuate the chance of *conflict misses* since we 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* [43] 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 it might only be optimal for certain types of computations (e.g. SIMD) there is still room for improvement.

Note that while SOA greatly fits SIMD it is not considered to be an ideal solution to encompass objects [21, p. 1060]. For that we need solutions, that regard both temporal and spatial locality.

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/is involved in the same data transformations. Whenever we see, that certain subsets of data are frequently used together it is advisable not to separate them. So instead of rigorously converting each member into an array pendant, we group logically related data and create arrays of those groups instead (see Code 6).This way the relevant data concerning one NPC will not spread over different incoherent cache-lines, that could map to completely different segments in main memory.

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

npc_group
    npc_groups[NUM_ENTITIES];
```

Code 6: Consolidating related data

This technique bundles related data and makes sure it is successive in memory as well as in cache-lines, consequently we won't peril those cache-lines to extrude each other from the

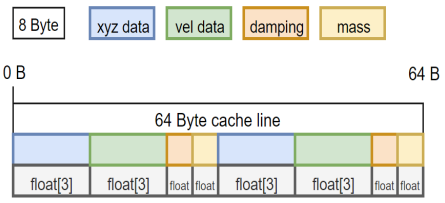

Figure 8: Unified/Grouped relevant data in a cache-line.

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 shares temporal locality.

Hot/Cold Splitting

A famous practical application of grouping a particular subset of data is called a *Hot/Cold Split* [39, 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 [12, 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.

```
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

The cold data is externalized into a struct of its own. The original struct now contains only the hot data, as well as a pointer to a cold struct instance. Since access frequency does not necessarily resemble the original partitioning of the fields, this pattern emphasizes the preference of logical over contextual relation.

Especially for monolithic class definitions, there can be numerous logical subsets of data fields. For example one data subset of a classic OOP gameobject will mainly be used for physics calculations (velocity, acceleration, mass, colliders), another for rendering (vertice data, shaders, textures) and yet another that embeds the game object in the game's environment (health, strength, gold, stamina, etc.).

First of all it is not always apparent whether a field is hot/cold. An experienced programmer might feel confident enough for a reasonably small class definition to eyeball it. A better approach might be to wrap our fields with access mechanisms that let us count how often they are accessed at run time, yet again we could rely on static analysis. Since this work will specifically implement a prototypical tool that performs automated hot/cold splits we will discuss this further in section 4.

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! Access frequency can't be the sole deciding factor for a split so this pattern should only be applied when a certain knowledge about underlying hardware components is given.
Whenever we split a record we reduce both the size of the hot data and the stride to get to subsequent instances resulting in better cache utilization (see Fig. 9). So in terms of a heuristic (that we will later describe) we want to cutoff as much fields as possible. Even a split that divides

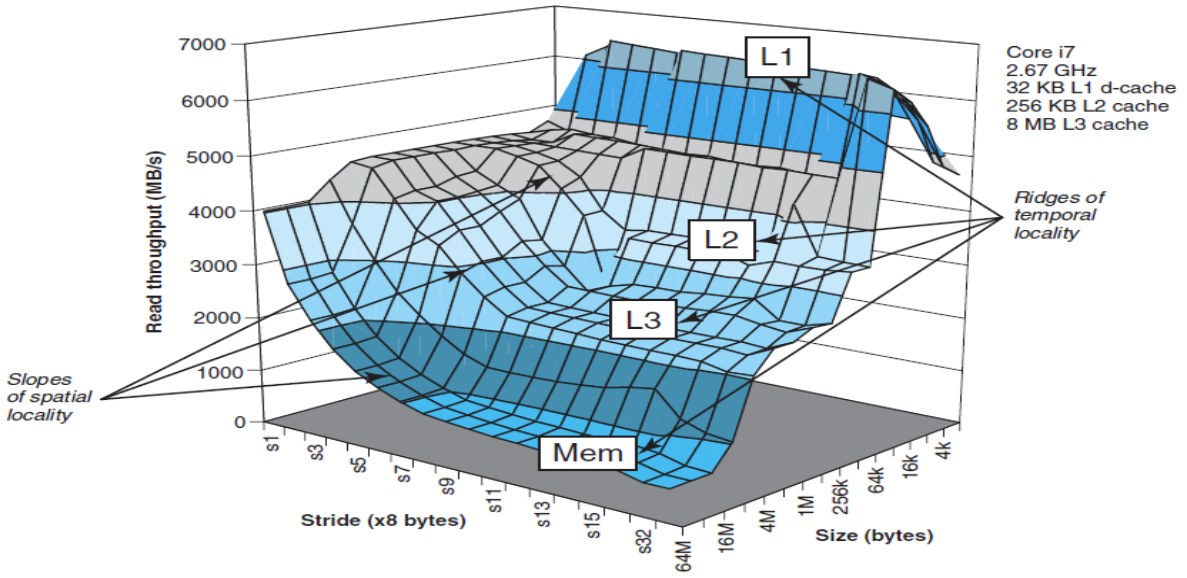

Figure 9: Relation of a record's stride and size to the read throughput, characterized as the memory mountain. (Source: [10, p. 623])

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 in section 4.

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 in order to create spatial locality for domain related data. 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* [39, p. 213].
In the classical component pattern we will keep a container object that holds instances of each

component [39, 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 [39, 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 [19, 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/appease 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 [43]. 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 may have 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 (not considering other processes). 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 [15, 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 [15, 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 [43].
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 one or a few 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 as one of the most important concepts in programming [10, p. 24] and 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 [19, 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 [9] 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* [44] 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.* [44]

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. 11). 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

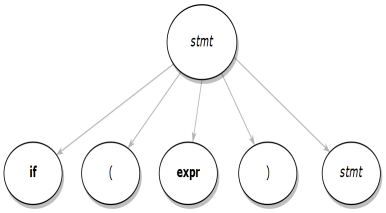

Figure 10: Parse tree for the if-stmt node.

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. 10). 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].

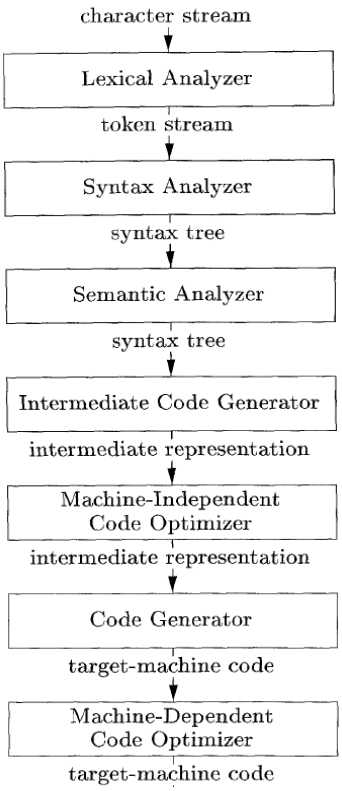

Figure 11: Phases of a compiler (Source: [5, p. 5]).

3.2 A useful interface / LibTooling

As can be seen in Fig. 11 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 [31] - 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 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

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

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. 12.

Figure 13: 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. 13 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 see that while hot/cold splits are conceptually trivial, performing a particular split is only beneficial given the meta information about the data is correct. Automating the retrieval of such meta data will be a fundamental mile-stone in development and set the base for all further source transformations (see section 4.2.1).
Ideally we desire that the target program should provide zero additional programming overhead 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 why this entails massive additional responsibility for our tool (see section 4.2).
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. Section 5.0.3 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. However optimizing the data layout of a program will only really affect software, that heavily relies on it. For example transformations on multidimensional arrays are perfect candidates for locality optimizations [10, p. 617]. 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 the less COOP implicates structural changes to the build setup or an engine's tool-chain the higher are its chances of being used. 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. 11). 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 if possible we should avoid expecting users to change their build setup for us.

The Clang front end functionality 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 split-candidate evaluation through static analysis

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 14: 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. 14 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 [27, 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 [27, 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 [27, 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)[8, 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*"[25, 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 [16, 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 [16, 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 $3 \cdot O(1)$. 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 $O(N \cdot M)$, 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

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.

```
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

The access patterns that start to make things spicy for the cache however are usually loops.

> *Loops have good temporal and spatial locality [...] the greater the number of loop iterations, the better the locality.* [10, p. 589].

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 that function 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 reside 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/deep, depending on the situation.
In order to actually account for an expression's loop depth we will need measures that will be discussed in section 6. For now we imagine having instant access on them.

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. 14 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. 15). 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 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

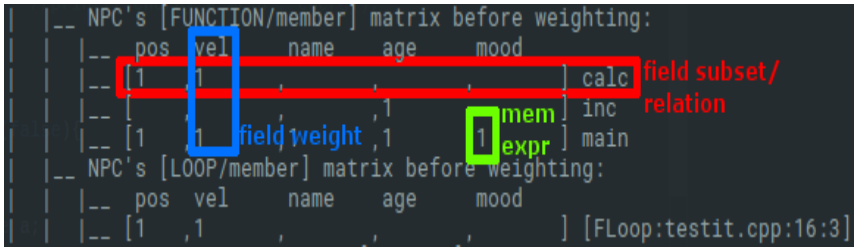

Figure 15: Relations depicted in function/member matrices

(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 heuristics

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 16 to 21 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. 16) unlike for example their median value. Unfortunately Fig.

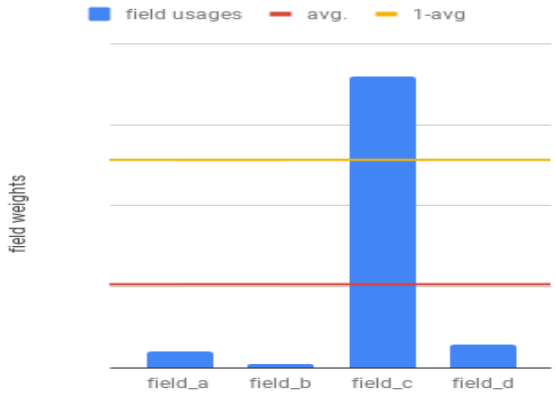

Figure 16: Good avg scaling.

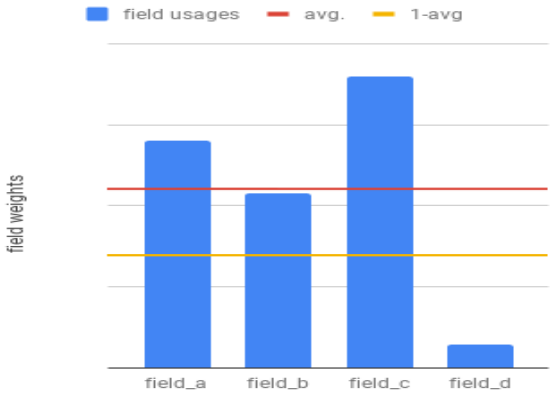

Figure 17: Difficult evaluation for avg scaling.

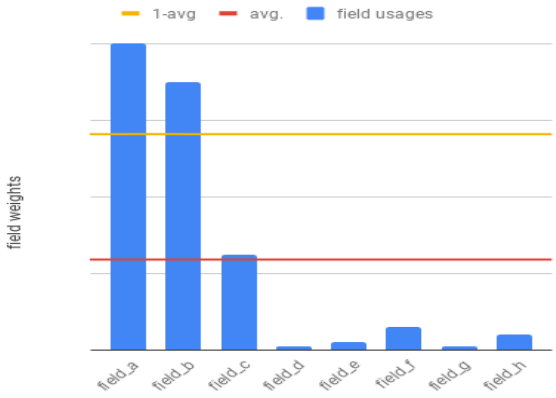

Figure 18: Bad avg scaling with more fields.

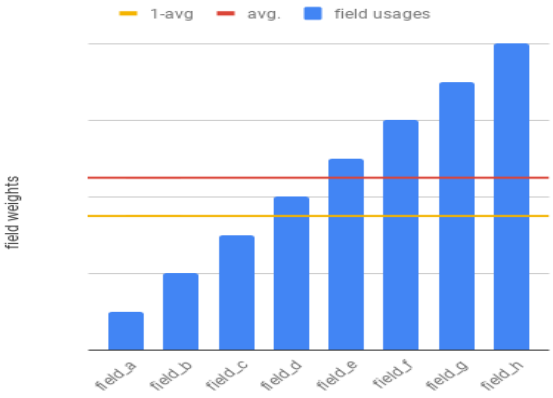

Figure 19: Problem of even distribution.

17 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. 18 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? 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. 19 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

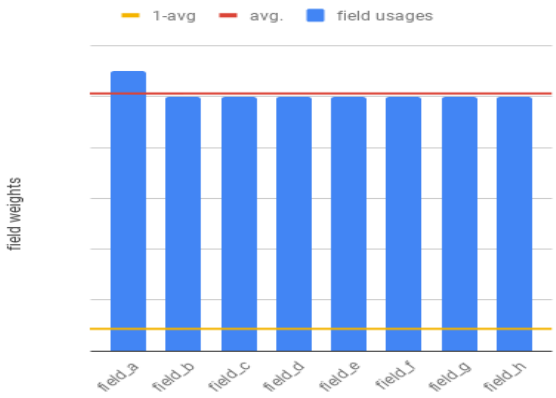

Figure 20: Bad avg homogeneous field weights.

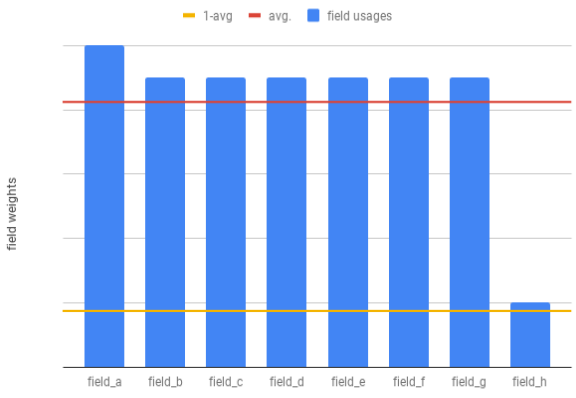

Figure 21: *1-avg* prone to false positives as well.

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. 20 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 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. 21 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. 22). 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

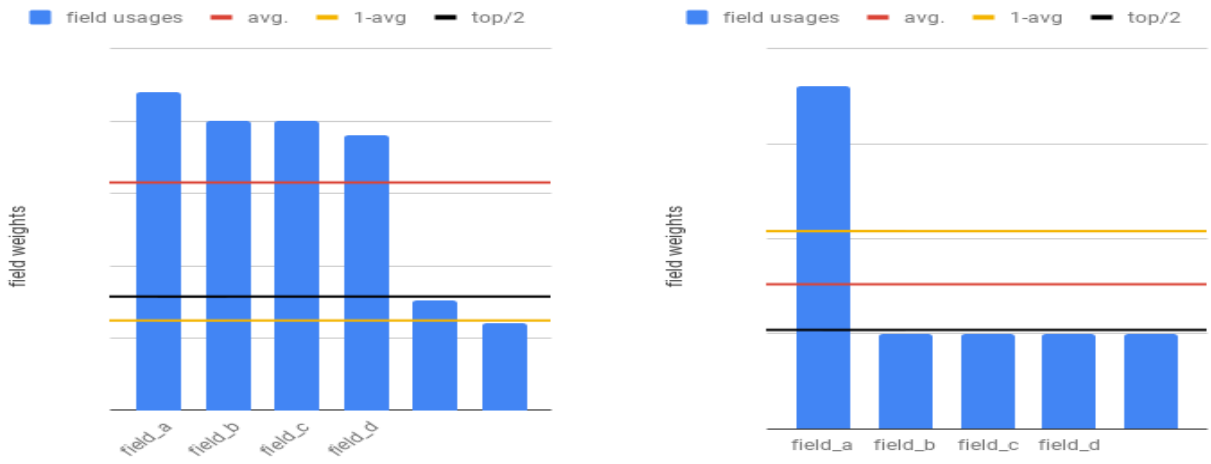

Figure 22: Improved *top/2* heuristic as it is able to rule out *1-avg* errors but still not well.

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 categorize 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. 23). 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 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 19 bespeaks of defective access patterns rather

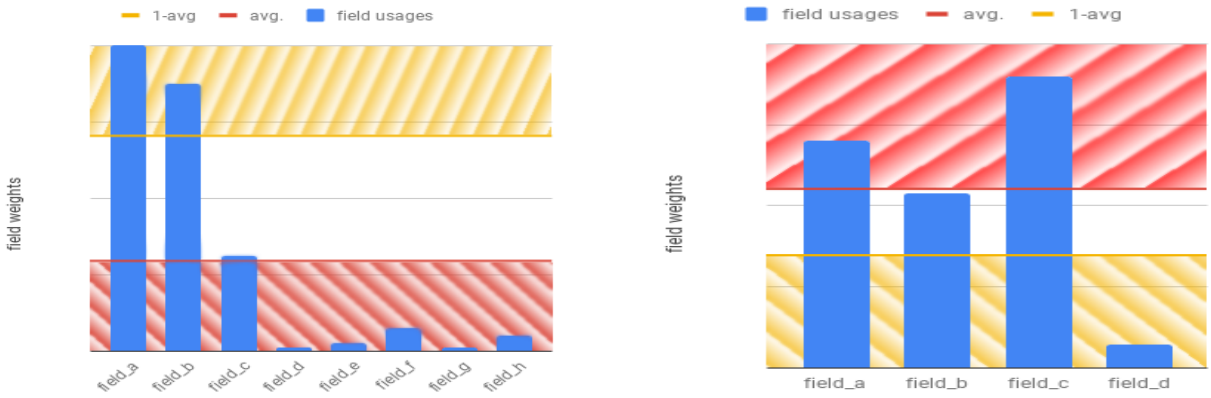

Figure 23: Field weight categorization by combined scaling delimiters. Top hatched space is of high significance.

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. 24). 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 our 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. 25). The important thing is for the fields to be evaluated in terms of groups rather than individually.

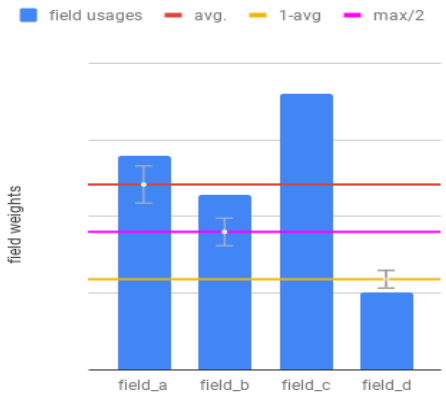

Figure 24: Scaling delimiters' errors can hardly be reasoned about and provide equally much punishment as benefit depending on the distribution.

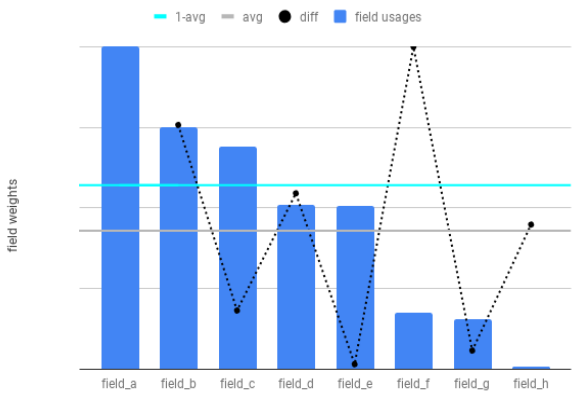

Figure 25: Determination of significance groups with field weight deltas. Normalized differences are projected on the field weights' scale for visualization.

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:

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

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

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

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 estimated benefit is greater than the estimated 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 (see section 4.3.2) 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. 26).
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. 27). 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

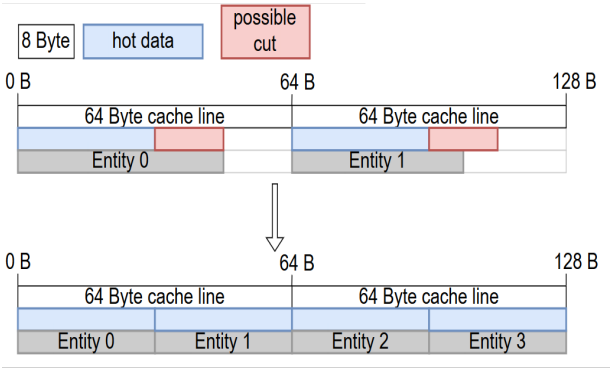

Figure 26: Aligned entities will be packed inside cache-lines. Reduced stride will be effective, as soon as it increases the amount of entities inside a cache-line.

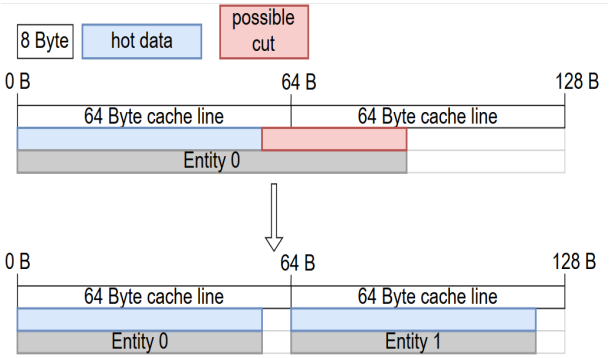

Figure 27: Reduced stride will be effective as soon as it reduces the number of cache-lines needed to encompass an entity. Entities are split upon minimum amount of lines.

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 our heuristic $W(g)$ will consider a split beneficial when:

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

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. Scaling delimiters

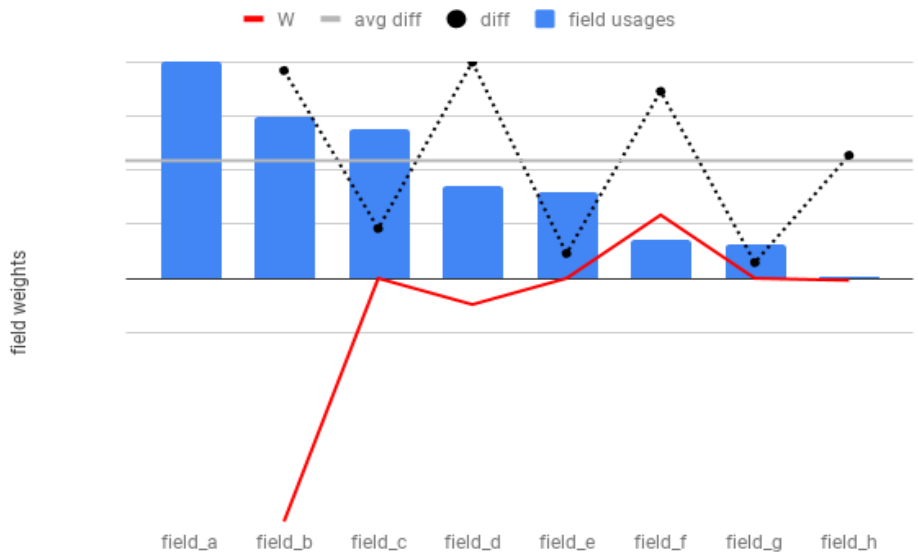

Figure 28: Exemplary field weights evaluated by our heuristic with the result, that its worth to make the split hot:[*field_a*, ...*field_e*], cold:[*field_d*, ...*field_h*].

solely relying on comparing the field weights will tend to split over aggressively, because they don't consider that the benefit of a split depends also on how the resulted record's will interact with a specific target cache. In fact to actually accomplish an improvement a cut will now need to extend a certain threshold. So in many cases COOP will now rather do nothing, than trying to force a split, that will actually worsen the cache utilization.

Unfortunately this heuristic might be problematic, as soon as we deal with significant spikes. Imagine a field, that outweighs the others. Since we only consider significance groups as split candidates (opposed to fields individually) we will now compare $n - 1$ fields to be split. This might actually make sense, because the highly rated single field might provide well enough cache utilization to outweigh the externalization of the rest, however we will strip ourselves from optimization potential as soon, as deviation significance leads to false positives on field to group assignment. When the significance group that field is assigned to, is not considered to be worthy of externalizing, but individually the falsely assigned field would be, we produced an error. Assigning a field to a significance group, that in reality should be in a lower order group, will prevent this field from being examined distinctively. So again we are limited by scaling delimiters, only this time when evaluating delta deviation instead of quality. Even though this is a special case, it is not unlikely enough to be disregarded.

One possible solution is to again follow a recursive approach. Whenever significance groups are identified we can recursively apply the procedure inside the distinct groups to identify more groups, but without a fitting recursion break, this will eventually lead to creating n groups for n fields.

It is hard to reason about when to stop the recursion, because forth going each group will have its own significance scope and therefore will evaluate its internal discrepancies differently - relatively more significant. To a certain extend this is exactly what we want, but at the same time it will eventually lead to over-grouping. What we need is an atomic unit that denotes a step of significance. This way instead of losing the scope in recursion, we will have a recursion-depth independent measure.

A brief discourse to data analysis in statistics

The methodology defined in statistics can yield very good results for our intention, however statistics are usually applied to large sets of data. A record with a number of fields that validates it to be examined statistically will rarely (if ever) be found in practice, however the methodology can be applied to our problem partially.

We could improve the significance groups' granularity and their robustness by applying measurements, that are known to behave better with spikes in their *population* or that can help us detect such spikes, so we can adapt our routines. Norbert Henze describes different *measures of dispersion* [23, p. 32], that we can consider. Statistically significant spikes can for example be identified by determining the *empiric standard deviation* for a population. The standard deviation (usually denoted as σ) is defined as:

$$\sigma := +\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{4}$$

It can be used to quantize the order of discrepancy to the norm. If a value for example is 2σ it is more likely to be significant, 3σ even more so. Consequently we could define that as soon as a field weight delta $f_i - f_{i-1} > T\sigma$ (where f is a field weight and T is an arbitrary constant) we would need a distinct significance grouping starting from f_i, since including field weights prior to i could compromise the significance group assignment for low ordered field weights.

Another measure of classification is the *interquartile range* (IQR). It describes the difference between the upper- and the lower most quartiles and is mostly resistent to spikes, because it operates on the median instead of the arithmetic mean. Significant discrepancies can be determined by orders of IQR units. For a set of elements $x = (x_1, x_2, ..., x_n)$ that is ordered, the IQR is defined by the difference of the upper and lower quartiles $x_{0.25}$ and $x_{0,75}$ which are the

respective medians of the elements to the left and right of the median $\tilde{x}$.

$$x_{0,25} = \begin{cases} \frac{1}{2}(x_{n\cdot 0,25} + x_{(n\cdot 0,25)+1}), & \text{if } n \text{ is even} \\ x_{\lfloor n\cdot 0,25+1 \rfloor}, & \text{if } n \text{ is odd} \end{cases} \tag{5}$$

$$x_{0,75} = \begin{cases} \frac{1}{2}(x_{n\cdot 0,75} + x_{(n\cdot 0,75)+1}), & \text{if } n \text{ is even} \\ x_{\lfloor n\cdot 0,75+1 \rfloor}, & \text{if } n \text{ is odd} \end{cases} \tag{6}$$

The IQR which is then determined by subtracting $x_{0,75} - x_{0,25}$ describes for us the range of the 50% of the elements around the median $\tilde{x}$. However this doesn't mean that each value above $\tilde{x} + \frac{1}{2} \cdot IQR$ is considered a significant spike.
A famous practical application of the IQR are *Box-Plot*s [47, p. 916] which are a popular utensil for quickly describing data distributions and possible spikes. They define spikes to be $> T \cdot IQR$, where T is a constant that is often between four and seven. However pin-pointing those 'spike thresholds' usually depends on the populations environment.
We could apply this methodology to identify spikes in the field weight deltas so we could determine which ranges (expressed in quartiles) need their own significance grouping. Also identifying fields with weightings below the defined low spike level could immediately associated as cold fields, as we can guarantee, that at least 75% of the other fields are more significant.

The methods above rely on constants, that are determined by experience and plausibility. Our algorithm can not yet rely on statistically relevant data to define those constants, yet we are able to make a plausible assumption. The higher we define the threshold for a spike to be the more tolerant we are for inaccuracies on the other end respectively. The closer we define our thresholds to $x_{0,25}$ and $x_{0,74}$ the higher the chance, that we identify local maxima/minima as spikes, which will lead to similar problems as blurry scaling delimiters (separating fields, that belong together).
Since the IQR defines the 50% of elements around the median, this range also defines a bound of significance. In theory $0,5 \cdot IQR + IQR + 0,5 \cdot IQR$ should define the whole range of elements if they share an order of significance. In order to prevent local maxima/minima to be identified as spikes we will tweak this assumption to $3 \cdot IQR$. This will identify spikes relatively quickly and induce a more fine granular significance ordering.

4.3 COOP's affect on the data layout

Until now we have seen how COOP is able to make an educated guess about the logic relations (expressed in temporal locality) and the computational impact of a field. Even though static analysis will rarely be able to predict a programs data flow precisely (because of run-time dependent variables) we have developed a way, that combines traits coming from different metrics to hint at what fields might be the most relevant to us. The numbers (for example access

frequencies) will in reality diverge from those we have assumed, however we have succeeded when our educated guess resembles the proportions of run-time data.

Even though we have set up the basis for a successful split at this point COOP's results would be surprisingly bad/mediocre, depending on the tested target code. Until now we have identified fields to be hot or cold and are able to separate them accordingly. We are affecting the fields' layouts inside their record, but have not yet considered how the records are layed out in memory.
The whole point of the AOS to SOA/HotColdSplit/Component/AOSOA transformations we discussed in section 1.3 is to improve spatial locality of our data. The actual benefit is the result of reduced stride and a better instance-size to cache-line-size ratio. By identifying temporal locality of our data, we can infer, that spatial locality on those data will result in better cache utilization, but at this point the instances of the record we are examining might be allocated on the heap completely scattered. With scattered heap segments like that, reduced stride in our records will have no effect, since subsequent segments of the memory (expected by the common access patterns we discussed in section 1.2.1) will most likely be unrelated to our 'hot' data. Cache prefetching mechanisms will not be able to accommodate us. We can never expect to find a successor instance inside an already loaded cache-line.
In this scenario we will even have separated the 'cold' data and therefore taken what little correlation the hot and the cold data had. So depending on how the data is distributed in memory until now our optimization might behave good, but just as well might deteriorate the cache utilization.

Another important part is the allocation of the cold data. By our definition fields are separated when the remaining hot data benefits from the split more, than the cold data suffers from it. By this definition it might very well happen that we extrude fields, that are used comparably little, but this does not mean, that those fields are used infrequently in their routines. Only because a *field_a* was assigned a field weight of e.g. $720,000$ the extracted cold field *field_d* with a field weight of $35,000$ can't be regarded impact-less. We deemed it beneficial to split *field_d*, but that doesn't mean we shall ignore it.
Cold fields might just as well participate in routines that abide the access patterns, we can optimize around. For example a particles position and velocity might very likely be considered hot, the particle's field *radius* will also be used frequently, but our heuristic recognized it to be little enough to warrant a split. The routines that use radius will most likely do so for a bunch of particles in a row. Making sure the cold data is packed in contiguous blocks of memory will improve those 'cold' routines.
Consequently COOP will not only parse the target program for instance allocations, that tend to behave badly, but also care about the cold structs to be allocated in a cache friendly manner.

4.3.1 Hot data allocations we want to adapt

There are different ways to obtain memory to hold data. Specifically when executing a C++ program the compiler will have made precautions to reserve memory for e.g. the actual code, variables known at compile time, a block of memory to hold local variables [45, p. 587].
We are especially interested in the data segments *stack*, *heap*, as well as *.bss .data*, because this is where the target program will reserve memory of the records we aim to split. Static and global variables exist in the *.data* segment (when initialized) and the *.bss* (if uninitialized). Local variables are placed on the stack. An instance that is allocated with e.g. *malloc, calloc* or *new* is placed on the *heap*. Note for later: the *.bss* won't actually affect the object file's size. The loader will merely zero initialize it (also depends on the OS) on program start [10, p. 659].

We want our data to be in contiguous memory blocks. With techniques like *placement new* we can easily transform the target code to place data where we want it but we should also clarify what specific allocations we want and can change without hurting cache utilization or corrupting semantic integrity. So whenever we find a statement, that allocates memory for a record that we want to split, we can change it in a way that will hopefully leave the succeeding code unaffected (see Code 17).

```
1 Foo all_foos[N];
2
3 //Foo f_0;   becomes:
4 all_foos[i] = Foo();
5 Foo &f_0 = all_foos[i]; //NOT OK: semantic integrity corrupted
6 ...
7 //Foo f_1[10];   becomes:
8 Foo *f_1 = new (all_foos+i) Foo[10]; //NOT OK: semantic integrity corrupted
9 ...
10 //Foo *f_2 = new Foo();   becomes:
11 Foo *f_2 = new (all_foos+i) Foo();  //OK: conserved semantic integrity
12 ...
13 //Foo *f_3 = new Foo[10];   becomes:
14 Foo *f_3 = new (all_foos+i) Foo[10];  //OK: conserved semantic integrity
```

Code 17: Examples of how we can change allocations using placement new to emplace the data where it is among related data.

However these changes imply using some index *i* that will determine the data's new location in the Foo collection *all_foos*. More importantly while a local variables lifetime is bound to it's scope, now it 'lives' in the *all_foos* and we will manually need to clean it up properly when we reached the variables original deadline, as well as maintaining the index *i* so it will again provide *all_foos+i* as allocatable memory. For the placement new variant there should exist a respective *delete* call that we will need to change as well. Instead of the delete call we will want to invoke that instance's destructor manually and manage the memory address on our own.
However local variables are intended to live and work inside the scope they are defined in. In terms of cache utilization we should assume, that their creation implies temporal locality with all

the other local variables of that same scope so at this point we probably should not try to force it to live among data, that it will not share contextual relation with.
Also we can't maintain semantic integrity when changing local variables. We have effectively changed the data types of *f_0* and *f_1* which will break successive statements easily or just cause subtle bugs (consider how *sizeof* will still work, but yield different results for *f_0* and *f_1*). Maintaining semantic integrity can only work, when working with pointer types.
In case of *f_3* the target program already does handle its data in data structures, that suffice the intention of contiguous memory blocks. Without further static analysis we can't predict which data allocation might have been intended regarding principles of locality. If an array allocation like this happens more than once in a program, these distinct blocks of memory might still correlate, but they might just as well be 'bundled' because they constitute one logic unit of computation. Forcing them to exist among other instances of their kind will most likely not provide additional benefit and as stated before without further static analysis we can't properly reason about them. To avoid breaking the code unnecessarily or to disturb a data flow that we deem optimal anyway, we only want to 'correct' the instance allocations, that lead to scattered related data. In other words: single instance heap allocations of hot data.
There is a specific initialization pattern, we target and that is high frequent heap accesses. Whenever we want to set up an array of pointers for a record we will probably engage a loop that iterates the pointers, calling new a lot. For each time the new operator is invoked we will undergo a context switch into kernel mode search a memory block of appropriate size in the heap context switch back into user mode [21, p. 427]. Depending on the heaps fragmentation this will scatter the instances all over our free memory. What we want instead is for those instances to be emplaced in some predefined contiguous block of memory.

4.3.2 Customized memory management for splitted records / COOP's free list

A contiguous block of memory for our hot data instances will prevent them from laying across the programs free memory. This way minimized stride will result in improved cache utilization. However the moment we initialize instances in our own predefined scope of memory we need to manage that space on our own. This is what custom allocators are used for. However assuming that block-wise initialization of data in the target source code is done intentionally, we do not want to interfere with it. Instead we will focus on providing management utensils to prevent high frequent heap access, as discussed in the previous section (see section 4.3.1). There are many different implementations of custom allocators, but usually they are used to either allocate fixed sized blocks of memory or provide a demanded capacity dynamically. Since we will only handle single instance allocation COOP's custom memory management system will somewhat be a special case of a *fixed block allocator* where the block size is the records size. In other words what we want is a so called *pool allocator* [21, p. 430].
COOP will inject a data structure into the target project, that is able to find and free available

space without adding significant run-time overhead. A rather simple implementation for our pool allocator is a *Free List*.

But first of all we need to inject code into the target source code, that will reserve us some memory. The data we will want to manage with our free list would have originally lived in the free memory. The heap can make use of the entire addressable space which exceeds the stack. So we don't want to move all those data to the stack, because this would potentially overflow quickly, depending on how extensively the target source code allocates heap memory. We could also rely on dynamic allocations, however in order to minimize manual memory management obligations we will instead target a static data segment. COOP will ultimately inject a bunch of (templated!) code into the target project, so in order to keep the resulting object files' sizes minimal we will use the *.bss* segment. The *.bss* section occupies no actual space, the object file merely needs to remember how much space it will have to demand on program load [10, p. 659]. On Unix systems the *size* command line tool is a handy utensil for investigating this. For example when creating a simple *bss.cpp* file with an uninitialized global array of 3 *int*s and another *data.cpp* file with an initialized array of 3 *int*s we can investigate their object files' segment sizes, as well as their file sizes like this: (see Fig. 29)

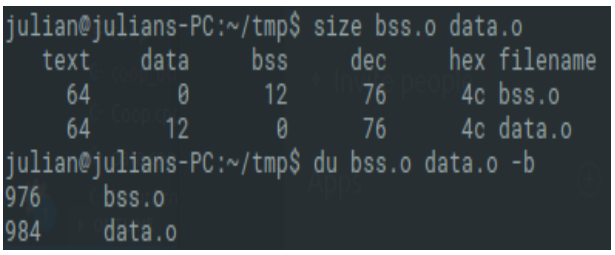

Figure 29: Comparison of .bss and .data sizes for different initializations.

So for a splitted record *Foo*, COOP will generate the following code injection in order to reserve the appropriate amount of contiguous memory space:

```
1 constexpr size_t hot_size_plus_ali_NPC =
2    coop::size_plus_alignments(N, alignment, element_size);
3
4 constexpr size_t cold_size_plus_ali_NPC =
5    coop::size_plus_alignments(M, alignment, element_size);
6
7 char byte_data_Foo[hot_size_plus_ali_Foo + cold_size_plus_ali_Foo];
8
9 coop_free_list hot_free_list_instance_Foo(
10   byte_data_Foo,
11   byte_data_Foo+hot_size_plus_ali_Foo,
12   alignment, sizeof(Foo));
13
14 coop_free_list cold_free_list_instance_Foo(
15   byte_data_Foo+hot_size_plus_ali_Foo,
16   byte_data_Foo+hot_size_plus_ali_Foo+cold_size_plus_ali_Foo,
```

```
17    alignment, sizeof(coop_cold_fields_Foo));
```

Code 18: Code injection to generate contiguous block of memory for a record Foo's instances.

Line 1 to 5 in Code 18 determine how much memory in Byte we need to encompass all instances. In line seven we reserve the actual memory (uninitialized *.bss*) which will be the combined sizes of the hot and the cold data for the record Foo. In line 9 to 17 we create the free list instances. Code 18 also shows that we rely on some additional variables and computations here. The sizes of the memory block, that our free lists will manage, need to be adjusted to the way the free list will operate. As mentioned before in section 4.2.3 to maximize access efficiency we will align our data. On the one hand we will pack as much instances together in a cache-line, on the other hand we will align those instance groups to the cache-lines alignment. This way we will possibly generate some padding between those instance groups, but we also guarantee, that the minimum amount of cache-lines is needed to load an instance. Disadvantageous strides will not lead to instances being separated among more cache-lines than necessary.
But now whenever this results in additional bytes for padding, we also effectively increased the necessary memory space to encompass all our entities. This is what the *size_plus_alignments(size_t number_elements, size_t alignment, size_t element_size)* function does. It takes the number of entities *N* we expect during runtime (line 2), the *alignment* that we will need to adjust on, and the *element_size* which will for example be the size of our record Foo.
The padding per instance group is dependent on whether or not the record's size is equal to, greater than or smaller than the cache-line size and defined like this (where *S* is the record's size in Byte and *CLS* the cache-line size):

$$pad\_per\_group = \begin{cases} 0 \text{ for } S == CLS \\ (\lfloor \frac{S}{CLS} \rfloor + 1) \cdot CLS - S \text{ for } S > CLS \\ CLS - (\lfloor \frac{CLS}{S} \rfloor \cdot S) \text{ for } S < CLS \end{cases} \tag{7}$$

The element size is also not just *sizeof(Foo)*, because we need to consider the possibility for a very small record to be split. For example if record Foo consisted of only an int (4 Byte) and a long int (8 Byte) and we wanted to split the long int away, leaving us with a single 4 Byte. As we will see when we explore the free list, it depends on its entities to be at least the size of a pointer, so it can properly resemble a linked list. That's why the *element_size* will be the maximum of the size in Byte of Foo and a Foo pointer type, to guarantee the free list to work as intended.
When we have the padding per instance group we can calculate how much capacity we need to encompass all our elements including their groups' padding respectively (where *m* is the

number of instances per group, *n* is the total amount of instances):

$$capacity = (m \cdot S + pad\_per\_group) \cdot \left\lceil \frac{n}{m} \right\rceil + CLS \tag{8}$$

The alignment is the target cache's alignment, which COOP will retrieve from the system on program start. On Unix systems information on the caches can be found in *sys/devices/system/cpu/cpu*<CPU ID>*/cache/index*<cache ID>. For example the file *sys/devices/.../type* contains whether or not this cache is a data cache or not. The file *coherency_line_size* will contain the cache-line size in Byte.
The same is done for the record Foo's new cold field struct. But the number of elements ratio between the record and its cold struct pendant will (should) not be 1:1 for reasons we will discuss in section 5.0.3. For now its left to say, that *N* and *M* will be determined by the user, as we can not predict how many instances will actually be made.

Finally the free list instances are created (Code 18 line 9 to 14) which will be given the byte range they will administer, as well as the target cache's line size and the 'chunk' size (which is just our record's size).

```
1   class coop_free_list {
2   ...
3   coop_fre_list(char *data_start, char *data_end, size_t alignment, size_t
        block_size){
4      ...
5      for(size_t Ts = 0; free_ptr < end; free_ptr = *next)
6      {
7         if(++Ts > Ts_per_chunk)
8         Ts = 1;
9
10        *next =
11           free_ptr+block_size+(Ts == Ts_per_chunk ? padding_to_next : 0);
12
13        if(*next >= end){*next = nullptr; break;}
14     }
15     free_ptr = begin;
16  }
17
18  template<typename T> T * get()
19  {
20     T *ret = union_cast<T*>(free_ptr);
21     free_ptr = *next;
22     return ret;
23  }
24
25  void free(void *p)
26  {
27     char *tmp_ptr = free_ptr;
28     free_ptr = union_cast<char*>(p);
29     *next = tmp_ptr;
```

```
30    }
31 private:
32    union {
33       char *free_ptr;
34       char **next;
35    };
36    char * begin = nullptr;
37    char * end = nullptr;
38    };
```

Code 19: Shortened excerpt of COOP's freelist without asserts and some initialization code.

The *coop_free_list* itself will only have three fields. Two char pointers (*begin* and *end*) which will mainly be used for bounds checking and a union containing a char pointer *free_ptr* that can also be used as a pointer to a char pointer.
In their essence free lists are only linked lists. Linked lists usually contain several nodes, that are linked to their successors, sometimes their predecessors and depending on specific implementations even more pointers to relevant nodes like the list head and end. I classic introductions to linked lists, nodes are allocated dynamically and a linked list can therefore grow (and spread) over the free memory. A linked list's nodes usually contain a data element, that in our case would hold the concrete instance and one or more pointers to other nodes,so the list can be iterated by following the trail the pointers constitute. In the case of a free list we want to work with contiguous memory blocks instead. Also to further work towards beneficial cache behavior it utilizes the data's memory space to also contain the pointer information to other 'nodes'. Each respective memory range, that will encompass an instance, will actually contain the pointer to the next element, whenever it does not at the moment represent an existing instance (see Fig. 30).
Since we especially want to handle single instance allocations and pack those into cache-line groups we will need further effort to ensure, that those groups are aligned optimally. Code 19's lines 3 to 15 contain an important part of the free lists constructor. Here we initially iterate over our entire range to set up the adjacency relations of our 'nodes'. Until this point we already determined the number of instances per cache-line group (or chunk). The free list instance will work with any data type, but will almost entirely work with char pointers to avoid unnecessary much templated code. However since 'generic' data types are often represented as *T*, the number of instances per chunk will here be *Ts_per_chunk*.
Starting at an address, that will initially be aligned to a cache-line we will treat it as a pointer to a char pointer instead of a char pointer, so we can emplace the information of where to find the next node in it (line 7 to 13). This is a simple operation, because we know the size of an instance (our *block_size*). Since the block size encodes Byte units we can just add it to the address of this iteration to get the address of the upcoming element. Doing so we will count towards the *Ts_per_chunk*. When we exceed the number of elements in a cache-line group, instead of only adding the block_size to the current address we will also add the necessary padding we determined earlier to jump towards the next aligned adress and start a new cache-

line group (see Fig. 30).
Treating the current address as a pointer to a char pointer is possible by accessing it through the union's *char **next*. To obtain a pointer to an element the free list provides the *get* method

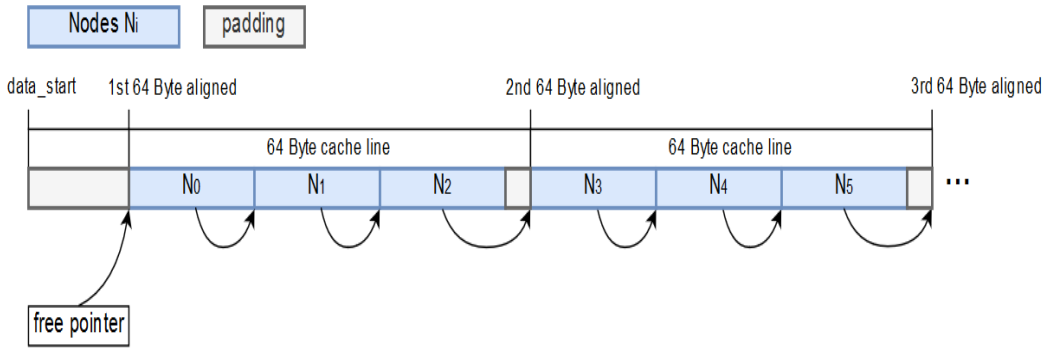

Figure 30: The free list's layout in memory. Starting with a (possible) initial padding, the instances here represented by nodes will be grouped in a way, that a maximum amount of nodes fits a cache-line. Each group is again aligned to the cache-lines (here 64 Byte) to guarantee minimal loads.

which will simply return the actual free pointer and declare the 'next' node to be the currently free element (Code 19 line 18 to 23). It is equally easy to declare an instance's node as free space again, by caching the current free pointer in a temporary variable, setting the actual free pointer to the newly free address and declaring it's 'next' element as the cached variable (Code 19 line 25 to 30).

Problematic fragmentation

Just like the free memory ultimately our free list is prone to fragmentation, as soon, as the target program's instances 'die' more frequently as well as randomly. On the one hand we do not have to worry about a for loop trying to iterate over all the elements in the free list, by merely iterating an index variable over the original array, because our free list is injected into the code and therefore the target code will have no interaction with our free list other than the ones we intend to. But fragmentation still hurts us, as it will deteriorate the number of instances to cache-lines ratio. For example when we have three instances in a cache-line group and 'destroy' every first and second element our instance : cache-line ratio will effectively have gone from 3:1 to 1:1.

However as new elements are created the 'holes' in between the actual data blocks will refill. Pool allocators are inherently robust when it comes to fragmentation. So In certain situations, where our target program requires a bunch of instances, deletes a random subset of them regressing our instance : cache-line ratio to 1:1, we might encounter non optimal cache utilization.

This is a special case and might be neglected but a possible solution would be for our free list's to shift blocks of memory together. More specifically whenever an address is freed, we

could take the last element and place it there. This way no matter how the *get*/*free* ratio of the target program would be, our data instances would be contiguous in memory. But its not that easy. Since we actually shifted the data a pointer to that data is now invalid.
To sustain the pointer it could either be a smart pointer, that observes the free list, reacting to changes, or our pool allocator could provide a second array solely consisting of pointers, to the original data array. The *get* method would now return an index to the pointer array and the corresponding pointer would lead to the actual data. Whenever data is shifted the pointer array is traversed and updated accordingly. Whoever is holding an index to the pointer array, will now access the actual data through an additional indirection [21, p. 440].
This could solve the fragmentation for special data lifetime patterns, but as mentioned above pool allocators are themselves quite robust to fragmentation so at this point its worth mentioning in case the pool allocator proves to be unqualified provide improved cache-utilization on its own.

Beneficial alignments of quasi continuous entity groups in memory can strongly affect a target programs cache utilization. With our heuristic considering the field's type sizes in Byte and the target cache's line-size we are moving away from making an educated guess towards calculating actual relations between a split and it's cache utilization. Measurements will show, that this approach is improving decision making on splits, however we will notice, that sometimes but not always our formulas are off a bit. Not meaning there is some sort of race condition going on. It rather depends on the project we are testing. To be more specific, it depends on the record declarations we are examining and the record sizes of the resulting split records. It might happen, that a target programs hot-record size is different from what we calculated when we tried to make a decision whether or not a split at a certain point is beneficial or not. How so?

4.3.3 Structure Padding and field reordering

Whenever we work with fields of data, in C++ we do so by associating a type declaration to it. A field's type on first glance is nothing but a piece of information on how much actual space we need to reserve in memory to suffice our data's capacity. For Code 20's char definition *c* and int *i* we could assume, that their layout in memory resembles their definitions in the source code, meaning there is one Byte for the char *c* and the following next four Bytes are reserved for our int *i*. Now sometimes and arguably most of the times this assumption will be valid, however depending on the particular address of *c* in memory *c*'s immediate succeeding Bytes will not belong to *i*.

```
char c;//+3B (possibly)
int i;

struct Foo{
   float f1;//+4B padding
   int *i_ptr;
   float f2;//+4B padding
};
```

Code 20: Example field declarations to elaborate on structure padding

For our modern processors compilers lay out memory abiding alignment constraints in order to make accessing them faster [41]. We already talked about how COOP will align its data managed by the free list, so that an instance will be split upon a minimal amount of cache-lines. In the best case this means finding an instance's data on one cache-line entirely. This is done, so we can guarantee a minimal amount of main memory loads, when accessing our data. As we learned those main memory loads can be extremely expensive so aligning data in a way that reduces load operations can have a significant impact on run time performance.
This is also what compilers do, when generating machine code for our field declarations. Accordingly, any data type has a defined alignment requirement, that is the information on how to align this piece of memory to guarantee optimal access performance. While chars encompass only a single Byte, they can be placed anywhere without further alignment requirements. Four Byte floats, or ints for example must be placed at an address divisible by four; Eight Byte longs or doubles at addresses divisible by eight and so on. This is sometimes referred to as them being *self-aligned*. One can manually arrange for the compiler to ignore alignment requirements (e.g. using pragma pack, or invoking the compiler with certain command line parameters), however this virtually always comes with punishment in terms of performance, so it is advised not to do so, unless its unavoidable [41].
Not aligning a data type not only implicates unnecessary many loads, but also results in additional effort for the memory controller, since in order to operate on the data it must now mask, shift and logically OR the two together into a CPU register, so the imminent operations can treat it normally [21, p. 160].

Lets come back to our example Code 20. In order to abide alignment requirements, depending on where char *c* lays in memory, our char *i* might follow a three Byte long hole of padding. The same goes for field declarations inside record definitions. While the float *Foo::f1* is guaranteed to abide its alignment requirement (we will shortly see why) the int poitner *Foo::i_ptr* will need to follow a four Byte padding in order to be aligned correctly. Float *f2* follows a higher alignment requirement, which suffices its own requirements (All alignment requirements are powers of two).
The compiler will add one last padding to this record definition, because it considers the possibility of struct *Foo* to be used in continuous blocks of memory. In order not to invalidate the field paddings we need to ensure, that they are correct for each successional instance. This can be done by aligning the record to its highest field alignment requirement. For our struct *Foo* this means, that in order to suffice the eight Byte alignment requirement we need trailing padding of an additional four Byte. Consequently the size of our struct *Foo* is not 16, but 24 Byte. Defining *Foo*'s fields in a different order could get rid of (in this case) unnecessary padding but essentially this explains how on certain test files COOP's methodology to determine a favorable split won't match the actual situation.
When considering a split for a certain subset of fields, there is not yet an actual compiled

version of it. We do not have the means to call *sizeof(non-existent-record)* on it, so either we implement a JIT compiler to momentarily compile a version of our record, or we consider structure padding our selfes. Merely summarizing the field's type sizes won't be enough, as it may disregard possible structure padding.
In section 4.2.3 we defined our formula on how to determine whether or not a split at a certain point is favorable or not. In order for this formula to work properly it needs to consider a theoretical split-record's structure padding.
We have seen, that records can be defined in suboptimal ways. We have seen before in section 1.3.4 that padding just like unused data is effectively lost potential in terms of cache utilization and we also have the means to 'improve' the target definitions layout. COOP will in fact rearrange a record's field definitions in a descending order (descending alignment requirements) to avoid unnecessary structure padding. With this COOP is able to get rid of most padding Bytes.

In section 8.2.3 we will discuss some major problems that this approach implicates. Actually this topic really hints at why an optimization like this can **never** be considered 'legal' or safe in terms of C++ language compliance and we will see, why COOP will work fine in some situations but in principal can't work.

5 Source transformations and pitfalls of semantic integrity

With the information and methodology we have gathered so far, we can now apply actual changes to the target source code. One could think that the 'hard' part is over. And from this on it is mere cause of 'find and replace'. In this section we will deliberately get lost in a chain of changes and surplus changes, before we discover a rather elegant solution. This is to show, how complex this supposedly trivial topic is and how seemingly small changes can imply tremendous impact on the target source code. Without a clean concept of what needs, should and mustn't change the rat tail of code changes will grow.

First of all we need a second data aggregation step, because without going too far now, there is lots of things that need change and lots of information we need, to do so. For now lets start with something we can change, without any additional information: The new struct for the cold fields.
For a record *Foo* definition like the one in line 1 to 7 of Code 21 we could create a struct containing the cold fields and call it *coop_cold_fields_Foo*. The name is quite arbitrary but includes *'coop'* and is expressive so further investigation and troubleshooting is not complicated unnecessarily. Also this attenuates the chance of accidentally running into namespace conflicts. This convention is applied to most changes, however to reduce the excerpts' sizes we will use shortened, yet expressive abbreviations.
The newly created *cold_struct* holds the cold fields. The original record keeps the hot fields. It is also given an additional public section for upcoming additions, like a pointer to an instance of the cold struct. To ensure that resources are cleaned up properly we will also create a destructor in the original record (Code 21 line 26) that on the one hand destroys the cold_struct instance and also notices the free list, that the instance's space is now reusable.

```
1 struct Foo {
2    int hot_0;
3    int hot_1;
4    int cold_0 = 10;
5    int cold_1;
6    int cold_2
7 };
8
9 void bar(Foo *f){f->cold_1 = 100;}
10
11 //becomes:
```

```
12 struct cold_struct {
13    int cold_0 = 10;
14    int cold_1;
15    int cold_2
16 };
17 struct Foo {
18    int hot_0;
19    int hot_1;
20
21    Foo(){
22       cold_data_ptr =
23          new (cold_free_list.get<cold_struct>()) cold_struct();
24    }
25
26    ~Foo(){
27       cold_data_ptr->~cold_struct();
28       cold_free_list.free(cold_data_ptr);
29    }
30 public:
31    cold_struct *cold_data_ptr = nullptr;
32
33 };
34
35 void bar(Foo *f){f->Foo::cold_data_ptr->cold_1 = 100;}
```

Code 21: Example of how a record might be split. However this split implies problematic aftereffects.

Extracting the cold fields implies changes to several parts of the target program so to further adapt it to the new data layout we will need to gather the AST nodes for several constructors and operators of the original record definition, the destructor, member expressions associating cold fields, calls of the new operator as well ad delete calls associating cold field and access spec declarations (to ensure code injections are placed with proper encapsulation).
Some AST nodes like constructors and operators can usually be extracted out of the record definition's AST node, however this does not always work properly, especially when the actual definition of the wanted e.g. destructor is in another file. So we need to again traverse the ASTs searching for new information. Luckily this is again rather easy, as we can use the AST Matchers (see Code 22).

```
1 //find access specs:
2 cxxRecordDecl(hasDescendant(accessSpecDecl().bind(coop_access_s)))
3 //find delete calls
4 cxxDestructorDecl(isDefinition(), hasName(regex))
5 //find constructor definitions
6 cxxConstructorDecl(isDefinition(), unless(isImplicit()))
7 //find copy assignment operator
8 cxxMethodDecl(isDefinition(), isCopyAssignmentOperator(), unless(isImplicit()))
9 //find move assignment operator
10 cxxMethodDecl(isDefinition(), isMoveAssignmentOperator(), unless(isImplicit()))
```

Code 22: Shortened excerpts for some AST Matchers to retreive information we need to implement changes –second data aggregation.

5.0.1 Redirecting cold field access to the externalized subset

In line 9 of Code 21 we see a function, that uses a member expression *f->cold_1* to assign the field *cold_1* a new value 100. The Expression is now broken after the externalization and needs to be fixed. With the prior AST Matchers we will find every member expression that associates cold data. COOP iterates them and redirects them appropriately. This is rather simple, since all there is to do is injecting access on the cold data pointer first. After retrieving the respective AST node we will assemble the code fragments, we need to constitute the correct access to the cold field.

```
1 std::stringstream access_redirection;
2 access_redirection << field->getParent()->getQualifiedNameAsString() << "::";
3 access_redirection << "->" << field_decl_name;
4 replace_text(mem_expr->getMemberLoc(), access_redirection.str());
```

Code 23: Assembling the proper access ot a cold field through the additional indirection.

One might notice, that Code 23 will produce qualified accesses. Usually COOP will use the record's qualifiers in the target code, to ensure proper namespace associations, but not in this case. Records might happen to be in a sub/super class relation. Whenever a super class and its extension are both splitted we need to ensure, that the right cold field struct is accessed. This could either be solved by making the splitted record's name part of it's cold data pointer's identifier (e.g. *cold_data_ptr_Foo* instead of *cold_data_ptr*) or by making sure the cold fields original record's affiliation is expressed through qualified access (e.g. *Foo::cold_data_ptr* instead of *cold_data_ptr*).
Performing automated splits on records involved in inheritance is a complex topic of its own we will further discuss in section 8.1.4.

Another small but important detail here is, that the original code is replaced from (including) the first character of the fields identifier (Code 23 line 4). This way we don't need to even be aware whether or not the original access comes from an instance or a pointer. The access on the cold field will always be through the cold data pointer so we can safely rely on the arrow operator -> to retrieve it.

5.0.2 Guaranteed existence of cold fields

Access on the cold fields can be redirected to the *cold_data_ptr* so the proper field can be dereferenced. But right now we can not guarantee, that whenever the target program expects

a cold field to exist, that it actually does.
During development redirecting access to a cold field was tested through an inlined access method. This seems like a good practice, since prior to the field access we can check the address for validity and this effectively enables lazy evaluation for the cold fields instantiation. This also meant, that there needed to be several versions of access methods, because different access situations based on const qualifiers would produce compile time errors otherwise. Clang enables us to adjust to those situations. For example a Member Expression's qualified type can easily be accessed through the AST node like this:

```
bool is_const = mem_expr->getType().isConstQualified();
```

Code 24: Retreiving type information from a clang::MemExpr pointer.

So depending on the qualified types of the member expressions we could decide which access method would not break the code. However as we discussed before, cold fields are not automatically used rarely, they just happen to be used relatively low frequent in comparison to the hot fields. Lazy evaluation and constant additional validation checks could in those cases hurt the runtime instead of initial loads since the original code might access the fields one by one several times. So instead the simple solution is: making the cold data pointer public and letting the target code 'directly' access the pointer to the cold fields. However this also implies that we need another mechanism to ensure, that cold fields are existent/initialized correctly whenever the target program assumed they would be.

The previous attempt of always accessing cold fields would create an instance of the cold data struct on the spot. Now that we have decided against it and in favor of initial loads we will turn to the original record's constructors. So instead of generating a cold struct instance on demand, we will make sure to create it, when the original record's instance is created. A convenient way to intercept the original record's creation is altering its constructors.
That is why we defined AST Matchers to find them. Again retrieving them through the record definition's AST node should work in most cases, but prove to sometimes refer to the constructor's declaration instead of it's definition. There is a few things to be aware of. First of all there can be numerous constructors for a record. In order to guarantee a cold struct instance to be present (and initialized) on creation, we should probably inject the respective line into each constructor. This would certainly work, however is wasteful since there might be delegating constructors. Whenever a constructor merely extends another constructor both of them are executed.

Fortunately Clang accommodates us with this issue as well. A record's constructors are defined in their own type of AST node called *clang::CXXConstructorDecl*. These nodes know about delegating constructors and let us traverse their call hierarchy easily. So in order to never repeatedly and unnecessarily create a cold struct instance we will follow the constructor's target constructor until we find a 'root constructor' (see Code 25).

```
1   clang::CXXConstructorDecl *ctor = ...;
2   while(ctor->isDelegatingConstructor())
3      ctor = ctor->getTargetConstructor();
```

Code 25: Traversing the constructors' delegating constructors until we find a 'root' node.

Several constructors might eventually lead to the same root constructors, so besides we also need to refrain from injecting the creation code multiple times to the same constructor now.

However what do we do if there is no constructor definition at all? Well there will be a constructor but it might be an implicitly generated default constructor. In this case COOP would now actually have to create a 'user-defined' constructor and inject it into the record definition. The actual greatest benefit of the previously mentioned access method is, that we do not rely on altering constructors but can always expect the target code to be in possession of a cold struct instance when needed. Without it we are forced to inject additional constructors. And its getting worse.

5.0.3 Semantic integrity and deep copy emulation

The clear upside of a getter access method that creates cold struct instances on the spot is, that it will never try to dereference a nullpointer due to nonexistent cold data. That is as long as we have enough space reserved in our pool allocator. Same goes for constructor adaptations/creations.
At this point COOP faces a major problem. Both of the above attempts do not consider yet, that externalizing the data layout will very directly affect the data flow of the program even after redirecting cold field accesses correctly. The problem is that externalizing a subset of fields will not only affect the code, that is written by the original programmer, but also all of the implicitly generated code, that is yet to come from the compiler.
For example for the purpose of testing COOP a program was written that implemented a simple particle system in a classic object oriented manner. Fig. 31 shows two screenshots of that application. On the left there are some particles flying around, in their midst burns a cozy particle fire. On the right we see the same application after an early version of coop tried to improve it. We can see, that the particle fire now has another color.
To be specific, the particle fire has now another shader. How so? The particle systems defined in the application are initialized with a single particle as their prototype. The fire and the particle gun shooting orange balls share a prototype. The prototype is only changed before it is given to the other particle system as a reference. Since our particles are defined as classic objects, they (amongst others) define their shader as a direct property of theirs. COOP recognized the shader as field that is used rarely compared to the positional traits, as well as the particles velocity, acceleration and mass, that are all used in physics calculations. Consequently the shader field was externalized into a cold struct. When the prototype particle was created for the

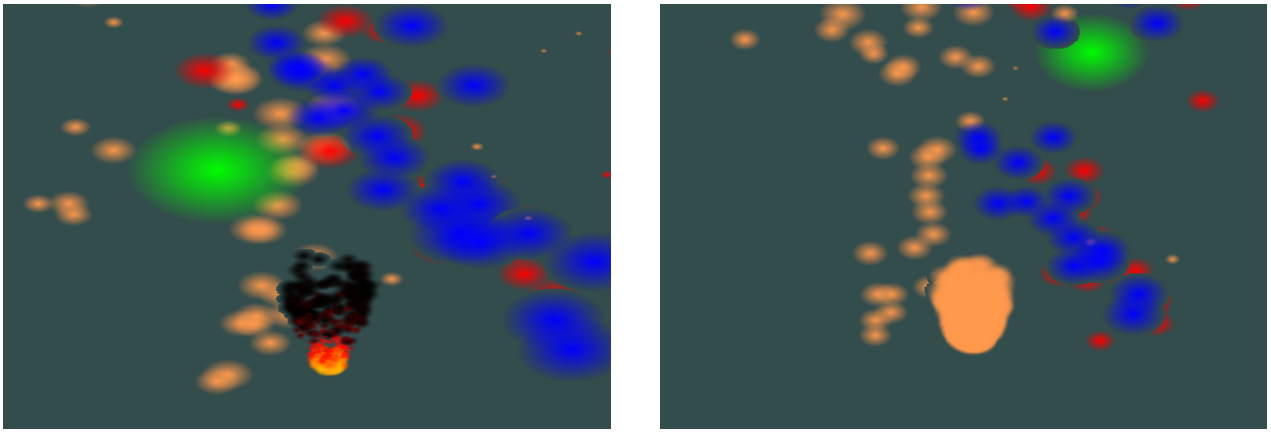

Figure 31: Screenshots of simple particle application. Left - original. Right - faulty shaders on the fire because of semantic corruption.

cozy fire its shader field was initialized with the 'fire shader'. Forth going we created another prototype for the orange particle gun by copying the fire's prototype. We then altered the new prototype copy to make it fit for the particle system shooting orange balls and gave it the 'orange shader'. This is the part where we accidentally messed up the fire particle system's prototype. By externalizing the cold subset from the particle record, we not only affected the fields direct associations, but also the programs interactions concerning the whole record. Copying the particle record used to copy every single field inside the particle. Where once was (amongst others) the shader field we now copied the pointer to the first prototypes cold data. We have effectively disabled copying of cold data and created shared state among copied instances of hot data. This is not a semantic equivalent to the original program.

Actually this outcome can be considered lucky. The consequences of losing semantic integrity can easily break the code. If we were to create temporary copies of our particle, as soon as the copy left the scope it would now destroy the cold struct instance, while the original still depends on it. A quick and dirty fix to this is to make the cold data pointer a shared pointer. This will on the one hand prevent temporary copies to prematurely destroy the shared data, but on the other hand this is still not better. Temporary copies need to be able to alter the copied data, without affecting the original data. The substantial problem here is, that what used to be distinct data is now shared state.

To fix this we now need to make sure, that the copy constructors and/or copy assignment operators work properly after a split, too. Thats why we previously searched for their AST nodes to be able to access and alter their definitions. Also from this point on the house of cards will start to totter. There are several things, we need to consider. Are there any user defined copy constructors/copy assignment operators, because the user will not have included our cold data pointer in the self made copy assignment operator and field access redirection will do the rest. On the other hand when there is no user defined copy assignment operator but an implicit one,

we now need to actually make sure, that this copy assignment operator will copy anything but our cold data pointer. On top of that it will need to manually copy all the cold fields. Since it is an assignment both our above methods (access method, constructor initializing) will already have provided us with an instance of the cold data struct, so no need to create a new one.
This means iterating all the fields and copying them manually. In case we are dealing with static array types, we will resort to *memcpy*, by determining its size like in Code 26.

```
1   siez_t size = field->getASTContext().
2       getTypeSizeInChars(dyn_cast_or_null<ConstantArrayType>(type.getTypePtr())).
3       getQuantity();
```

Code 26: Retrieving the size of a static array type through its AST node.

When there is no user defined copy constructor but an implicit defined one, we also need to do the same for the copy constructor, with the exception, that in this case we also need to generate a new cold struct instance. At least the copy constructor may just use the copy assignment operator. However we have now already bloated the original record definition.
Note also, that this is the reason why we can't have the access getter method for that creates cold struct instances on the spot without validation and redundancy checks. While this would work for copy constructors, each time we made use of the copy assignment operator (where we access cold fields) we would create a new instance, even though the assignment determines, that there already is an instance to work with.
On top of that since we might have now manually defined a copy constructor/assignment operator, we gave up implicitly defined move semantics for our target program. E.g. a move constructor will be defined implicitly if the respective record does not have a user defined copy constructor/assignment operator [46, p. 5] (among other criteria). We initially had no problem with move assignment operators and move constructors, since they could just take the pointer to the cold struct instance without a problem. Note that even creating a user made destructor will prevent the compiler from generating implicit copy and move semantics.
At this point several algorithms might no longer profit from move semantics and instead have to rely on copying which means unnecessary much object creations/deletions. We might have just deteriorated the target programs performance instead of optimizing it. While manually assembling the copy assignment operator was quite possible we can't do the same for the move semantic pendants. We don't know how to destroy the fields on our own and can't just blindly call delete on fields.
Luckily if move assignment operators and move constructors were originally implicitly defined and we prevented them with 'user defined' copy constructors/assignment operators, we can bring them back by defining them as *default*.

At this point we have a working implementation, however there is a solution that requires much less injected code, AST traversal for additional information like constructors/destructors/operators and almost works out of the box.
By directly initializing the cold data pointer *in-class* we also affect each constructor. As soon

as an instance is initialized it is ensured to point to an instance of a cold data struct. However even though this drastically reduces the required code injections, it also is not yet a semantic equivalent to the original code, because it still suffers from faulty instance copying.
Before we mentioned the shared pointer as a quick and dirty 'fix'. Indeed a smart pointer could solve our problems. COOP's solution to maintain semantic integrity with the original program is to emulate a deep copy by defining the behavior in a container for our cold data pointer (see Code 27). The important lines are line 8 and 12. In line 12 we make sure, that an instance is created as soon as we create an instance of the hot data (Accordingly the hot data holds an instance of a *deep_cpy_ptr*). Line 8 ensures, that when there is a copy, we will copy the actual cold data, rather than only the pointer.

```
1  struct deep_cpy_ptr{
2     deep_cpy_ptr(){}
3     deep_cpy_ptr(const deep_cpy_ptr &other){
4        *this=other;
5     }
6     deep_cpy_ptr & operator=(const deep_cpy_ptr &other){
7        if(this==&other)return *this;
8        *ptr = *other.ptr;
9        return *this;
10    }
11    ~deep_cpy_ptr(){
12       ptr->~cold_struct();
13       cold_free_list.free(ptr);
14    }
15
16    cold_struct *ptr = new (cold_free_list.get<cold_struct>()) cold_struct();
17 };
```

Code 27: Shortened version of COOP's container for the pointer to the cold data struct intance. It only defines a single field (the pointer) so no additional memory space is spend.

This works fine as the in-class initialization guarantees, that copy construction will create a new cold struct instance and copy assignment will not instantiate redundant cold struct instances.
The only thing worth mentioning is, that no matter where a hot data instance is created (be it on the stack or heap), from now on they will each have their own cold struct instance. Our pool allocator for the hot data will provide space for single instance allocations of hot data, however temporary copies of hot data instances on the stack or array allocations on the heap of hot data instances will always create cold struct instances on the pool allocator for cold data. When assigning char arrays to our allocators for them to administrate, we will want to make sure, that there is enough space for the cold data.

5.0.4 Constructor initializers associating externalized fields and const qualified cold fields

Unfortunately under a certain circumstance we can't avoid interfering with constructor definitions. Redirecting cold field associations will implicitly care about cold field initializations happening inside a hot record's constructor. The initialization of cold fields might however also happen through constructor initializers in a member initialization list (see Code 28).

```
1   struct Foo {
2      Foo(int h, float c):hot_field(h), cold_field(c){}
3
4      int hot_field;
5      float cold_field;
6   };
7
8   //becomes:
9   srtuct cold_data {
10     float cold_field;
11  };
12  struct Foo {
13     Foo(int h, float c):hot_field(h){
14        cold_data_ptr.ptr->cold_field = c;
15     }
16
17     int hot_field;
18     deep_cpy_ptr cold_data_ptr = ...;
19     ...
20  };
```

Code 28: Simplified source transformations for problematic cold field initializations in initialization list.

In this case the cold fields initialization can't remain in the initializer list, because after the split the constructor can't access them the same way. Unfortunately this implies possibly worsened performance, as usually member initialization list can e.g. resort to copy construction directly instead of cunstruction, assignment operator and destruction steps for constructor arguments that are *passed by value*.
In order to make the changed source code even compile, COOP will move the cold field initializations into the constructor's body (line 14).
However this implies further changes. Whenever a record's constructor initializes non-static cosnt qualified data members, moving their initialization into the constructors body will also result in compilation errors, as they may only be initialized in-class or in member initialization lists. On solution in this case would be to extend the cold struct by an additional constructor, that resemble the target constructor of the hot data record. This way the 'hot constructor' could internally construct the cold struct instance with the appropriate 'cold constructor'. This not only requires us to again bloat the target source code, but also won't work properly considering, that we want to in-class initialize the cold fields instance. Calling an additional 'cold constructor'

inside the ’hot constructor’ would either lead to multiple cold struct instances per hot instance, or imply a possible costly copy assignment, when all we want to do is modify distinct cold fields.

Since COOP is meant to be a finalization step optimization rather than being an iteration based development tool it will simply get rid of the cold field’s const qualifier at all.
As the const qualifier is merely a developer’s guide at the point of COOP’s invokation we will want to ignore it in favor of possible performance hazards. So whenever a cold field possesses local const qualifiers, its image in the cold struct will lose local const qualifiers at all.

6 The Project Context

Up until this point COOP is already conceptually able to process simple test software consisting of a single source file (e.g. c/cpp) and whatever files it may include. When using the LibTooling suite we will actually create a *Tool* instance, that comprises elements of the Clang compiler front-end. To obtain the ASTs we depend on through out our entire process we feed the source files to that Tool instance. It includes a preprocessor that will resolve an preprocessor directives. This means, that any include files will be correctly treated as parts of the compilation unit. This way we can think of the resulting ASTs as representations of the to-be object files.
Tings get tricky however as soon as we actually start to try operating on multiple files. The *Tool* instance is perfectly able to sequentially process different source files and will produce ASTs for each of them. So while we are not necessarily facing any exceptions we will at this point at least run into strange and faulty behavior. We want and need to operate on project scope. Our entire optimization strategy depends on knowing what records and fields there are, as well as every single usage of them throughout our source files respectively.
The LibTooling suite is intended to perform modular source-to-source transformations, for example automated style checks/improvements. This works because while those tools can be given multiple source files, ther transformation criteria will never depend on shared state between the created ASTs. In terms of optimizations we effectively want to perform a link time optimization, so we can operate on the entirety of our target project's code.

Creating an AST for each translation unit will hurt us in terms of multiply defined AST nodes. For example when our project defines a *Foo.hpp* file, that contains the class definition of class *Foo* and we have two source files *Foo.cpp* and *main.cpp* both including *Foo.hpp* we will already have multiple AST nodes for our Foo's class definition in our context, as well as all the AST nodes representing method and field declarations. We wouldn't care if our transformation criteria was modular, however searching the ASTs for record definitions will now yield two class definitions and without further clarification our routine will treat them separately as they are distinct AST nodes. This applies to virtually every type of AST node.
Also e.g. functions and methods might associate AST nodes that will be declared in their AST context, however their respective definitions could (and will often times) exist in an entirely other translation unit.

Source transformations are also affected, since Clang's *Rewriter* utility class will not be able to behave correctly when operating on distinct AST nodes pointing to the same physical file (The Rewriter is the source transformation tool, provided by Clang). It wont necessarily break but

if changes coming from different Rewriter instances affect overlapping segments of the source file strange things will happen (or no changes might be committed at all). Not to mention applying the same (redundant) changes coming from seemingly different AST nodes to the same physical file segments. So first of all we will need a proper 'file to Rewriter instance' mapping, which can easily be done.

AST nodes are able to be dereferenced into their *Source Locations* which are Clang's mechanism to associate an AST node to a specific physical location in a specific file. All of the relevant information concerning the AST are represented in it's respective *AST Context* e.g. which source files it represents (*SourceManager*) or what language options need to be considered (*LangOptions*). So whenever we want to change a specific code segment, instead of creating a Rewriter instance on the spot, we will first check, whether or not there already exists a rewriter instance that was assigned to a specific AST Context.
Ongoing we will need more sophisticated mechanisms to prevent redundant processes as well as correctly associate AST nodes, that depend on each other, exceeding their definition's AST contexts. To do so we will in certain terms logically link several AST's resembling the work, that would otherwise be done by the linker using the symbol table.

6.0.1 Global AST Node Representations

As mentioned before when working with multiple source files we will most likely encounter multiply defined AST nodes which resemble the same source code for different translation units. For us these are redundant nodes and whenever we happen to use redundant AST nodes we will produce blurred or plainly wrong results. For example consider Fig. 32. When creating the function/member matrix discussed in section 4.2 we could end up counting all member expressions of one source file *Foo.cpp* and account them to the class definition we found in *Foo.hpp* for the *AST Context 1*. The member expressions we find in *main.cpp* we might account to the class definition node we found in *Foo.hpp* but this time we refer to the AST node coming from *AST Context 0*. To ensure consistent association of possibly redundant AST nodes and logically unique entities, we need to somehow check whether or not AST nodes refer to the same source. Clang's AST node type *clang::Decl* is the base for all declaration based AST nodes. It does provide functionality to retrieve a unique ID for a declaration, however that ID is only valid inside a single AST.
To distinguish different AST nodes and identify 'project scope duplicates' we need to somehow generate a unique ID for an AST node, that will reproduce the same result for each AST context. The entities name is not enough since logically different entities can share a name in different in different namespaces and even scopes. COOP will therefore use the nodes source locations. The one thing that each duplicate AST node will have in common is its reference to the physical source file, as well as its position inside that file. No other distinct Definition/Declaration, can share this trait. The appropriate information can be retrieved by consulting the AST context's

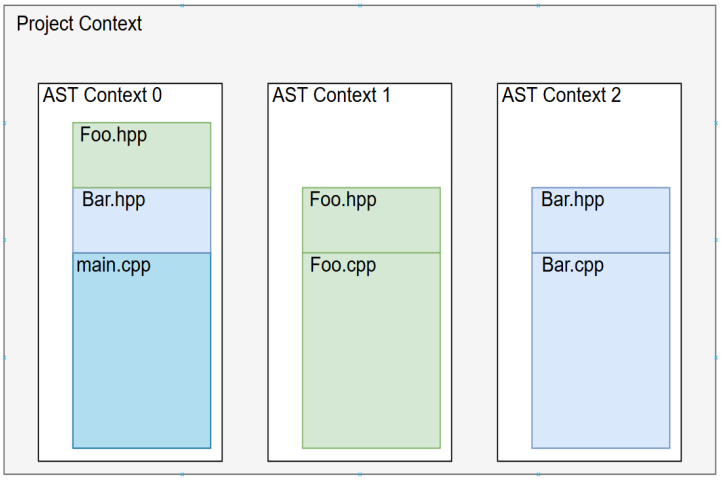

Figure 32: Visualization of how different AST contexts might include identical/redundant information when regarded in an abstract project scope.

source manager instance.
Even though an AST context will include the declarations that come with resolving preprocessor directives, the definitions of other source files won't be present in it. After all this is something the linker would do, in the following steps of actual compilation. So even though we can now identify duplicate declarations inside our project context, the 'link' of a declaration to its definition still has to be done manually by us.
This will actually happen in the *Data Aggregation* step discussed in section 4.2.1 however only now we have the background to explain some interesting details. The matchers defined in section 4.2 will search for declarations or definitions depending on what we need of them. Whenever a matcher yields a possible candidate and invokes our callback routine, COOP will then create a wrapper object for virtually every node we want to remember.
These *coop::ptr_ID* wrappers contain a generic pointer to any kind of AST node, a unique identifier and a pointer to the relevant AST context. For each AST node type we need to remember, there will be a global container of *ptr_ID*s. The data aggregation's callback routines that are invoked when a matcher finds a relevant AST node, will now always generate an ID for the node they examine and register it in the global container, when no other ptr_ID with the same unique ID is yet present.
From now on whenever an AST node is retrieved from the AST before doing anything else we will always first ask a *coop::global<T>* container whether or not there is already an AST node, that represents this particular node. In most cases this already provides enough security for our demands, because most of the times we only ever care about counting entity occurrences. Theoretically it is possible now to globally register for example two distinct field declarations from the same record coming from different AST contexts, depending on the sequence in that they were found. And in most cases this will not even be a problem, as long as we are only interested in counting occurences and need to associate a e.g. member expression to a field

declaration. However the ASTs are traversed and matched sequentially, so for example field and record declarations will be matched together and therefore point to a single AST context already as a result of their traversal sequence.
But whenever we need to depend on the AST nodes definition, because the definition contains valuable information, there is a problem. Functiondeclarations as well as record declarations might be forwarded/prototyped at one place but defined somewhere else (even in different files).

In this case we need to manually ensure, that we never process an AST node declaration, that can't find the appropriate definition. For example lets consider Fig. 32 again. Imagine *Foo.hpp* declares a relevant function, that is then defined in *Foo.cpp*. In case AST context 0 was created first, the moment we found the function declaration, we created the appropriate ID and remembered its AST context (AST context 0). After that, when traversing the AST 1 we would again find the function declaration, only this time we are able to associate it to a definition, because the AST context 1 encompasses the *Foo.cpp* file. In this case we need to make sure that our global registry remembers the AST node from AST context 1. So we 'bend' the existing *coop::ptr_ID* node pointer and AST context pointer accordingly. Same goes for record definitions, as they are subject to forwarding. Code 29 partly shows the callback routine, that is invoked by the AST matchers when they find an AST node of type *clang::FunctionDecl* (a fucntion declaration).

```
1 void coop::PrototypeBending::run(const MatchFinder::MatchResult &result){
2    const FunctionDecl *proto = ...;
3    const FunctionDecl *func = proto->getDefinition();
4
5    if(!func){return;}
6
7    std::string id = coop::naming::get_decl_id<FunctionDecl>(proto);
8
9    auto global_f = coop::global<FunctionDecl>::get_global(id);
10   if(global_f){
11      global_f->ptr = func;
12      global_f->ast_context = result.Context;
13   }else{
14      coop::global<FunctionDecl>::set_global(func, id, result.Context);
15   }
16 }
```

Code 29: Shortened excerpt of the callback routine, that registers function declarations for COOP in the data aggregation step.

In line 3 we determine whether this declaration is able to associate its definition. Next in line 7 we generate a unique ID for the record declaration, that we will use to determine whether or not there is already a node, that associates this declaration. If so this will be a declaration node coming from another AST, that will therefore not be able to associate this definition. In line 9 we ask the global registry whether it can return us a node that is registered with the id we made. If

so we now have to bend the existing nodes pointer to the AST node that provides fill information and also associate the right AST context to the registry for that node. We can't just delete the existing ptr_ID and create the correct one, as we must assume that at this point pointers to this ptr_ID might have been created.

6.0.2 Project contextual AST abbreviation

All of the above methods ensured, that our processes can interact on the global scale of the target project. The inherent problem we had, was that our optimization requires project contextual information, like what fields are used when and where and how often. We need to be able to link certain AST nodes from one AST to other nodes of other AST contexts. Now we can and its exactly what COOP needs to do next.

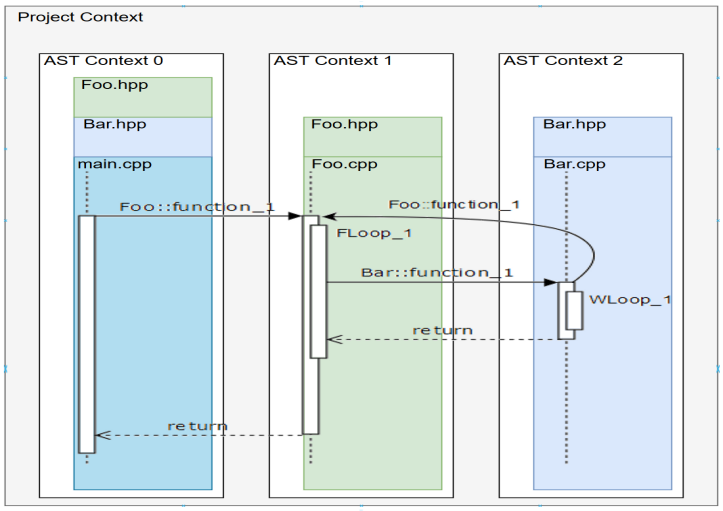

Figure 33: Function calls and loops can interact in all sorts of forms in a program and recursion must be considered as it implicates similar behavior on access patterns as loops do.

In order to create the function/loop member matrices we need to correctly depict the loop-depth of a field usage. Meaning how many scopes encompass this particular member expression that belong to a loop. This problem is not exactly trivial, since loops might be nested, functions might contain loops, loops might call functions that contain other loops (see Fig. 33). Theoretically this nesting of functions and loops is endless, it also might express recursion.

COOP's solution to this is to create a directed graph managing several AST nodes of functions and loops, that point to each other (given they assosiate each other in the original code). This graph would resemble the ASTs we created from the target project's translation units so why do it? Because in our own abbreviation of those ASTs we can 'merge' the different ASTs . Thanks to global registration of AST nodes we can now associate definitions to other definitions where we could only declare conceptual relation before. This resembles the steps the linker would normally do using a symbol table, although in this particular case we only do it for function calls and loops.

A node class *fl_node* (function/loop node) is used to hold either an AST *clang::FunctionDecl* node or a *clang::Statement* node if its a do/while or for loop. It also holds a list of child nodes, a count variable, that will later hold this particular nodes loop depth and indicators to whether it

is involved in recursion.

After establishing the relevant links between the ASTs embodied in our AST abbreviation graph first of all we want to detect recursion. We search for recursion by traversing the graph and remembering the already visited nodes. When we follow a branch and end up on a node, we already have visited then we know it is the start of a recursive call. It is marked as such. Without further, more ambitious static analysis, there is no telling to what extend this recursive call resembles for example linear-, logarithmic- or even polynomial growth, so at this point any evaluation of its impact on the program is highly speculative. In section 8.1 we will see that in order to make COOP viable much more static analysis would be needed.
In order to finalize the prototype we will merely treat recursive calls as another linear loop, so their 'depth' is equally important to a field's weight. With an evaluation of our programs data flow like this, that not necessarily resembles reality, but will hopefully mirror the proportions, we are now able to reason about a field's weight in terms of asymptotic notation (see section 4.2.2).

However to correctly evaluate a field's relation to other fields of the same record, we still need to check whether fields are contextually related. In order to do this, we can again use our project context AST abbreviation. Whenever a function *A* accesses a record *Foo*'s fields *m_a* and *m_b* we already can establish, that *m_a* and *m_b* have contextual relation. But for each function/method *B_i*, that is called by *A* we can't immediately decide whether it contains yet other fields of record *Foo*, that should be regarded as contextually linked.
To establish correct field relations, we will traverse our AST abbreviation from bottom up. In this context this means, we start at functions that invoke no other functions. For each node we make sure that it's field usages are accounted to it's parent function calls. Assuming function *A* uses *Foo::m_a* and it calls another function *B* that itself uses *Foo::m_b* we will now treat *A* as if it associates *Foo::_a* and *Foo::m_b*. This way when applying measurements, we derive from cohesion metrics (see section 4.2.2) we can correctly infer field relations. Remember, that cohesion metrics like *LCOM4* only regard a record's methods. We operate project context wide, and therefore can't rely solely on comparing methods.
This is not due to cohesion metrics being sloppy, they are originally just used to make a statement on the records composition. We determine our own derivation of a cohesion metric, that reasons about a field's weight. For that we need not it's records cohesion index, but the fields' individual evaluation.

This sets the basis for our function/member matrices as we discussed in section 4.2.1. As of now COOP is able to operate on a project that depends on linking several object files together and creating function/member and loop/member matrices, that encapsulate information for a record's fields to determine domain relations and usage frequency.

7 Measurements

In section 4 we consolidated our goals for a tool like COOP. One of those goals was for it to improve the performance of a target program and never worsen it. In this chapter we will discuss COOP's results and see that the outcomes of our optimization varies greatly and depend on the target program. We will also see that COOP (while working as intended on some) is by nature not a generic fit for any kind of program, as its optimization potential derives from the target program's structure and access patterns.
Further there are a few different traits we want to measure about COOP. The optimization effects; Different heuristics on field weight evaluations; COOP's run times.

7.1 COOP's impact on performance

In order to see the affects of our optimization on a target program, we will measure its run-time and its frames per second (if something like a 'game-loop' is given).
Cache utilization will be tested with *Valgrind*'s dedicated tool *Cachegrind* created by Nicholas Nethercote. Valgrind runs programs on a simulated CPU. When specifically regarding improvements in cache friendliness this saves us from external factors like other processes thrashing our cache-lines. This way we can run isolated tests to compare the un-/optimized versions of our target program. In case of Cachegrind it will explicitly simulate a cache hierarchy and observe e.g. accesses and misses for us. Cachegrind is an amazing tool, since it is easy to use, works out of the box and is completely free.

7.1.1 Low complexity test scenario

To quickly see where the strengths and weaknesses of COOP are we will first observe the results of a very simple test case. This is the quickest way to see possible improvements/deteriorations and the easiest way to reason about how certain aspects impact certain numbers. For our first simple test we will define an arbitrary struct, its fields and some access patterns, that should enable COOP to refine the cache utilization. Lets just go with our NPC from the previous chapters.

```
1 struct NPC {
2     float pos[3];
3     char * const name; //unfavorable structure padding!
4     float vel[3];
5     int age, mood;
```

```
6     [some constructors]
7 };
8
9 int main(){
10    //classic OOP initializations
11    NPC **npcs = new NPC*[N];
12    for(unsigned i = 0; i < N; ++i)
13    {
14       npcs[i] = new NPC([ini arguments]);
15    }
16
17    //some positional-domain calculations
18    for(unsigned i = 0; i < N; ++i)
19    {
20       some_calculation(npcs[i].pos, npcs[i].vel, delta_time);
21    }
22    [event based data accesses to other/cold fields]
23    [cleanup code]
24 }
```

Code 30: Simple test code in order to make first tests. It will heavily use an NPC's positional properties (pos, vel) and basically disregard the fields name, age and mood, as they would be accessed based on user generated events.

After running COOP on this code we will identify the fields *name, age* and *mood* as cold fields. We build both the optimized and the unoptimized versions several times gradually increasing *N* in order to demonstrate how COOP behaves on growing data capacities. In the following graphs we will oppose the optimized- and the unoptimized versions where *(ac)* marks the values ***after coop*** has been applied, so the 'optimized' values. When looking at the number of cache misses

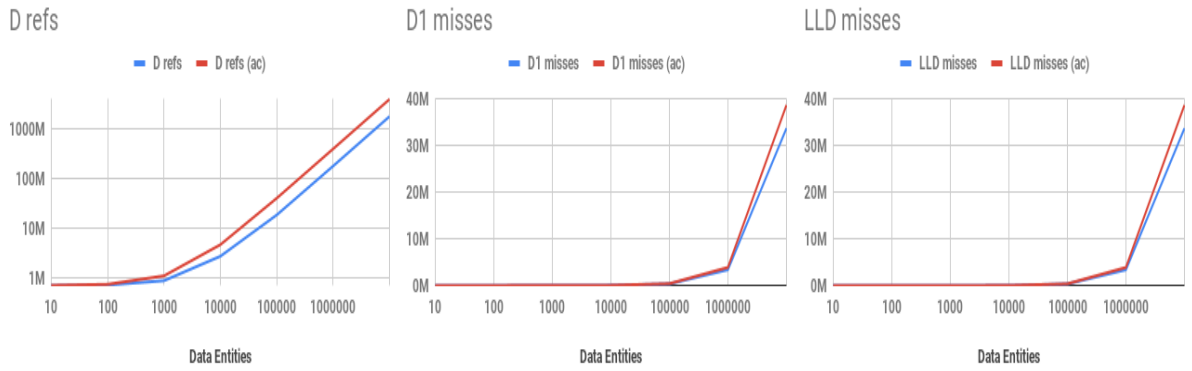

Figure 34: Measurements for *N* data entities and the resulting cache utilizations. The red line marks the 'post-optimization' values. Note that the *D refs* graph uses a logarithmic scale so the increase in D refs as N grows actually becomes very large!

Fig. 34 reveals alarming numbers. Despite our optimization, that specifically is designed to reduce cache misses we not only somehow managed to increase the amount of cache misses, but also the number of total data accesses! At this point COOP looks bad and further looking at

the execution times we will see, that there is actually an observable deterioration in run time as well (see Fig. 35). While moving time spent away from the operating system by interchanging

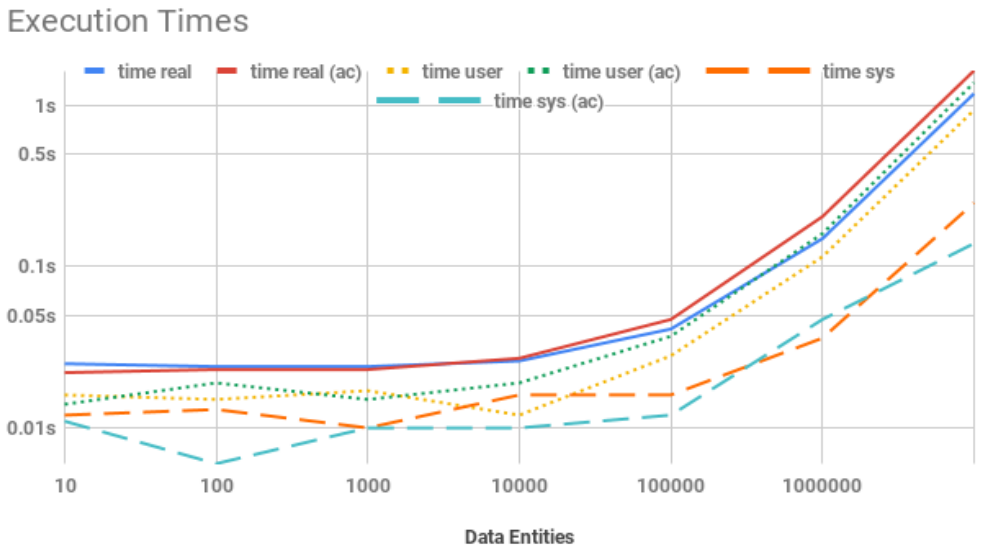

Figure 35: Measured runtimes for un-/optimized versions of our simple test code with rising N. These measurements have been made by using the Unix command *time*.

heap allocations with placement news to our bss data works well in the beginning, the overhead of initializing our free list grows linearly with the amount of data. While the operating system's free list that will handle our heap allocations already exists on a user program start, our custom allocator introduces additional overhead to the program, that will ultimately pitch towards inferior initialization overhead that might outscale the benefits of its fast allocation times.
Our specialized memory solution also is responsible for the additional data accesses, as well as the additional cache misses. Cachegrind offers post processing tools as well, like *cg_annotate*, that provides a per function statistic on cache misses etc. As we initialize the two free lists (one for the hot and one for the cold data) we will also produce cache misses scaling linearly with our *N*.

In our particular case not only our custom memory management adds initialization overhead, but it also won't improve the data layouts situation. The whole point of our pool allocator was for the single heap allocations to be contiguous in memory. In this particular test case, the data entities are sequentially allocated on the heap. Those allocations are made on a fresh heap, so even though this initialization pattern is prone to heap fragmentation in this case there is no prior fragmentation. Also since the entities will only be deallocated on program end, hence no new allocations during computation and possible fragmentation overhead, our pool allocator will effectively have no positive impact on the program. What about cache utilization? Even though increased initialization overhead adds yet more cache misses, shouldn't we at least have improved the access on the data during computation?

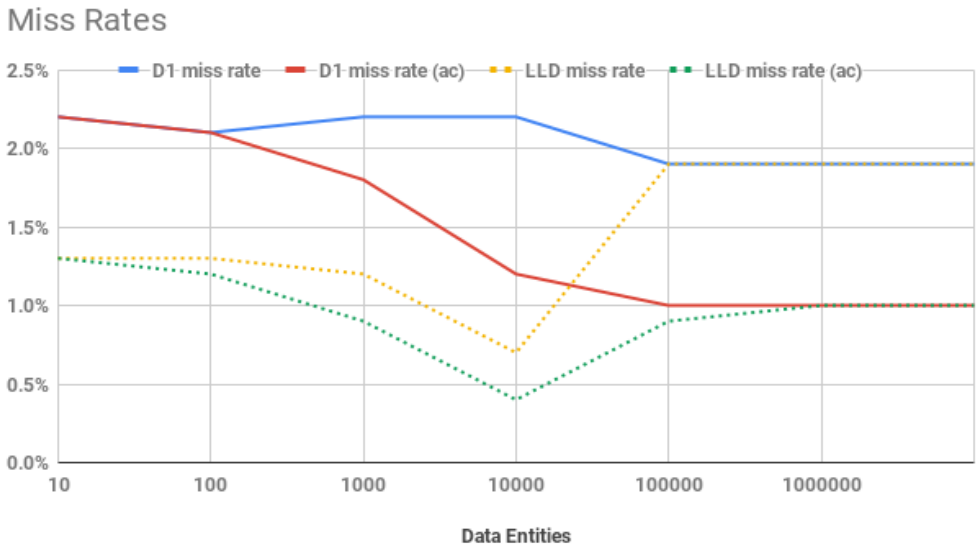

Figure 36: The small test programs miss rates show improvement, as the cold data no longer resides unused in our cache. Note how the last level cache miss rates rise between 10,000 and 100,000 entities due to the total amount of our data exceeding the cache size.

Fig. 36 illustrates the miss rates of our test programs with growing *N*. We can observe, that the unoptimized versions of our test program have higher miss rates even though there are more data accesses in general. As the data in both the optimized and unoptimized versions is contiguous in memory, this is solely attributable to the difference in the data layout produced by our optimization.

While our affect on the data layout was beneficial, the assumption, that single heap allocation initializations are inherently bad depends on the target programs complexity and data flow. Further static analysis could potentially identify bad initialization patterns more precisely but more on that in section 8.1.

7.1.2 Testing an OOP particle system

Our next test will work with a more complex target project, a particle system. While it will be deliberately programmed in a rather OOP manner and completely managed by the CPU, it will also provide us with an example of a program with undetermined runtime as there will be a 'game loop' and the user will trigger the program stop. In this case we would measure the targets delta times or FPS(frames per second), to see whether or not COOP can induce an improvement.
Specifically we will create a scene with four particle systems each shooting a bunch of particles at a center point, so the particles from the different systems collide and bounce off each other. The systems load will increase with several parameters: The particles life time; The particle

systems fire rate; the size of the bulks that are released at once. All those factors affect the total amount of particles that will be in the scene at one discrete time step.

Besides the rendering there will be one other significant part to the program. The physics. The computation of all the collisions and resulting changes to velocities, accelerations and ultimately positions will demand most of the CPUs resources and will happen once per frame. When increasing the total amount of particles in the scene, we will experience higher delta times, as processing their individual state changes will consume more and more time. Besides observing our optimization's impact on the cache utilization we will compare the programs' delta times over a fixed amount of frames (see Fig. 38).
The particles' setup is a rather simple POD (plain old data) type with all the data we need, for example a particles position, acceleration, velocity, mass, lifetime, radius, texture and shader. COOP's static analysis will yield field weightings as can be seen in Fig. 37. The shader and

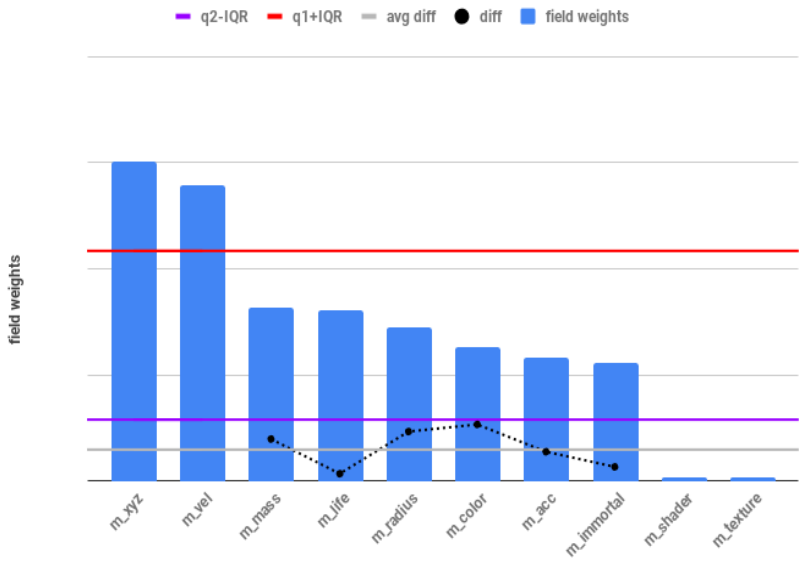

Figure 37: The particle's field weighst according to COOP. The q1/2 are the quartiles, that together with the interquartile range define our spike points. Inside the mid fragment we will induce further significance grouping as discussed in section 4.2.3.

texture fields were determined to be relatively cold enough to favor a split. This seems strange, as they will be used each frame, however our heuristic assumes, that hiving them off will benefit the other fields' computations enough to achieve improved cache utilization. In fact in the original version of the test program COOP would state, that it could not detect a favorable split, so it was deliberately worsened. Originally the shader and texture fields were merely pointers to their respective objects. To provoke bad cache utilization we instead let the particle directly hold their instances increasing the overall particles size to a point, where COOP would see optimization potential.

Now with the shader and texture field externalized to a cold struct the following differences in run time behavior were found Fig. 38. Comparing the two builds through cachegrind we find,

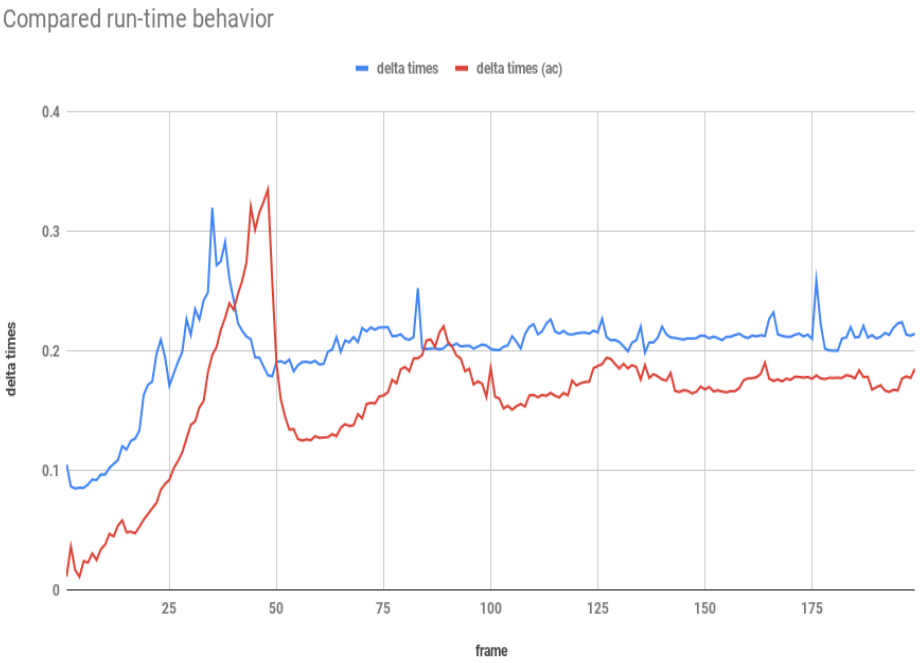

Figure 38: Observed delta times for both the optimized and unoptimized versions of our particle demo. The red line marks the delta times after the optimization was applied and shows that with our less frequently used fields externalized we can accomplish improvement. These measurements were made with approximately 96,000 particles.

that COOP successfully reduced the level one data cache's miss rate from 9.5% to 9.1% and the last level data cache's miss rate from 0.9% to 0.7%.

How is it, that in this case we could measure improvement when our simple test showed that initial initialization overhead might outscale improved cache utilization? The particles cold data is indeed managed by our pool allocator that is initialized on program start, however we are now measuring frame times inside a game loop. The additional initialization overhead still exists only the applications purpose differs. In this case the trade off is worth, as the initial overhead is comparably insignificant to constant improvement over an undetermined amount of frames.

Right now this looks promising, given that the target application profits more from an improved data layout than it suffers from program load overhead. However our particle system can easily be used in a way that resembles its actual usage in a real application and the results will be completely different. Lets now introduce a new instance of a particle system to our scene. Additionally to the four particle cannons from before we now initialize a system that will be a nice little fire. Its particles will have a short lifetime starting white gradually turning red and finally

turn to black smoke particles. The relevant part however is that those particles won't be colliding with other ones. While COOP right now won't notice that in any way its actual implications for our application are huge.
Right now (as said before) the physics steps is a major part of our little demo and those particles won't even bother to run through all those computations. With a colliding/non-colliding particles ratio of only 1:60 we have lost all our previous advantage (see Fig. 39). We introduced

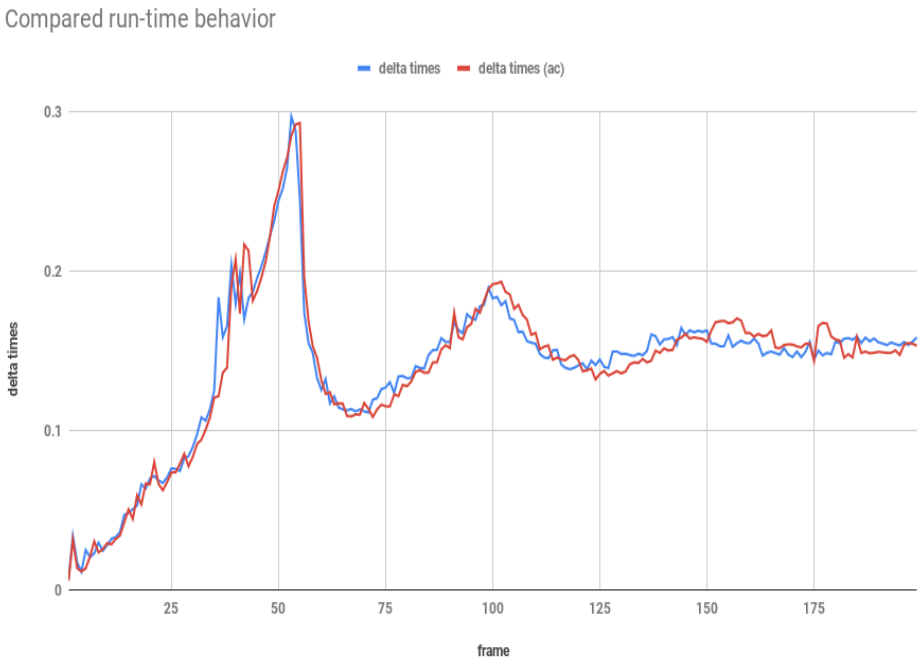

Figure 39: Compared delta times of an optimized and an unoptimized build showing that COOP's metrics have major issues with variant behavior of splitted instances.

a mere 1000 fire particles to a scene handling up to 60,000 colliding particles. While a thousand particles not having to calculate collision would usually not have a great impact on our system, now it does. Taking away the 'field weights' of other fields like position, velocity, radius and acceleration that are all highly important to physics calculations the externalized fields shader and texture become relatively more significant.
While the split is beneficial for one subset of the particles, it is now hazardous for another. With a data capacity that exceeds our cache size this becomes increasingly deteriorating, as more and more cache misses on frequently used data will continuously make slow loads from the lower hierarchy memory units. This is a major issue for COOP and will also be discussed further in section 8.1.3.

8 Conclusions, major problems and future work

This chapter will conclude the thesis, recapitulate the most important implications coming with a tool like COOP, discuss it's greatest weaknesses and shortcomings and will evaluate whether or not the author thinks that automated OOP to DoD transformation tools are a viable approach in general.

8.1 Future Work

There are numerous problems coming with a performance optimization approach like COOP. Some of them were already discussed in the previous sections as we solved them or found approximations that suffice a solution. In this section we will introduce some complications, that will dismiss COOP as a viable option as it is but could conceptually be solved in future work.

8.1.1 Language feature coverage and situational variety

First of all making a tool like COOP understand all kinds of language features there are, is conceptually not necessarily difficult but a lot of work. During development most of the work was adapting code to yet another language feature that came up in a new iteration of a test source-file.
For example right now templates are not supported, also even just multi-variable declarations are unsupported. Not because multi-variable declarations would be hard to integrate but at some point it became clear, that focusing on language compliance is a time consuming rat hole. Especially in the beginning practically each new test file will reveal another feature or situation, that needs to be handled individually. Also while the LibTooling API is tremendously helpful it also still bears some bugs that require creative workarounds.

8.1.2 Hidden field usages

```
1 MyRecord mr;
2 some_function(&mr.int_field);
```

Code 31: Simple example of how to hide a field usage from the LibTooling API

The foundation of our static analysis was on the one hand to identify the records in a project as well as their respective fields. On the other hand the most important information considering our Hot/Cold split were the field usages. The LibTooling environment and especially the AST matchers provide excellent methods to filter those *clang::MemberExpr*

member expressions. This way we can assemble detailed charts on what is used where and by what. When given this information we can pass it to our metrics and heuristics to reason about a fields weighting. This whole process is corrupted when there are field usages, that we can't recognize as such.
In Code 31 we invoke *some_function* that takes an int pointer as an argument with the address of our custom record's *int_field* field. At first glance this might be harmless, as the *mr.int_field* member usage is correctly registered by our AST matchers, however that function could invoke numerous other functions running an arbitrary amount of loops and recursion. Each consecutive use of that particular int would by our means represent a field usage however Clang won't recognize it as one.

To fix this we could rely on further static analysis to trace member usages of this kind. Whenever we find a function, that is invoked with a field for a parameter we would need to treat this function similarly to how we treat functions, that directly use instances of our records and more specifically treat that variable as a field usage for the functions scope, as well as for all functions, that are in turn invoked as well.

8.1.3 False Positives

COOP analyzes all records that are found in the given source files and provide a definition. While some records are utilized with access patterns, that allow for optimization potential using a Hot/Cold split, others are not. As of now COOP is not smart enough to recognize for example a singleton pattern or really any kind of factory pattern. So while the single instance of such a record might appear on several loops it won't have the effect on the cache COOP anticipates.
Right now COOP would ask the user before attempting an automated split including a user provided estimation of how many instances should be anticipated so externalizing the singleton's cold fields will not harm the instance or the systems performance, however it is an indicator, that false positives exist.
As long as our heuristic provides true statements about a record's 'splitability' we don't need to fear splitting it. The obtained improvements will just be so insignificant that it won't really matter. What we do need to be careful about are further changes and implementations, as those would need to consider the possibility that we are handling false positives.

On the other hand in our measurements we clearly determined that we are perfectly able to confuse COOP due to its inability of branch prediction. Section 7.1.2 showed that COOP would blindly identify an exemplary particle record as a split candidate because it provided code that would results in domain heavy profiles yet those very code segments were positioned in control flow statements.
The resulting 'optimization' proved to be beneficial as long as (and only if) all existing particles actually performed those domain specific computations.

Further static analysis could reveal code segments as situational and weight them differently but ultimately, as e.g. particle amounts can vary entirely dependent on an in game situation there is no way of actually making a safe assumption about those code segments.
Profile guided optimizations (PGO) might come closest to providing accurate meta data for those entities but whether or not PGO is an appropriate measure also depends on the target programs purpose. Performing automated splits on records can to a certain extend predictably influence a class in a positive way, but this shows, that even with given theoretically sophisticated static analysis or PGO we can't make reliable predictions as soon as we are dealing with purely run-time dependent properties. This conceptually disqualifies automated layout transformations to keep responsibility about split decisions.

Right now the final decision and thus partially the responsibility are passed to the user that is however far from optimal. One of our initial goals for COOP was for the user to provide zero additional programming overhead. There is indeed no additional programming required however we demand information about a records anticipated instances.

8.1.4 Inheritance

Inheritance is one of the core features that comes with an object oriented programming language. It also is one of those core features that enable the programmer for those intuitive abstraction concepts we talked about in section 2.
Unfortunately it is also very complex in terms of making an automated split. On the one hand "*Software engineer culture is drifting away from heavy use of inheritance anyway*" [39, p. 285] but as OOP in general it is still used and may not be ignored.
In Code 4 we have seen, that the compiler will inject certain traits to a class so that it constitutes a *is a* relation to its base classes. Since that injection is done at compile time, in terms of static analysis those inherited traits are only found in the base classes.
Member expressions retrieved with the Clang API correctly associate inherited members to the respective base classes. Accordingly the process of analyzing and optimizing a record can not be done only for a subclass, as gathered data and transformations might actually happen in the base class. This does however greatly affect how we need to analyze fields. Since COOP assembles function/member and loop/member matrices per record inherited state will never appear among the inherent state of a sub class.
If the new state variables heavily interact with inherited fields those correlations would not be considered as a split factor right now. For example imagine a struct *Base* with a field *a* and a struct *Sub* with field *b* that extends *Base*. *Sub* also defines a method *Foo* that returns the accumulated values of *a* and *b*. The two funciton/member matrices for both *Base* and *Sub* would show no correlation between *a* and *b*.

```
1 struct Base {                  struct Sub : public A {
2     int a;                         int b;
3     int Bar(){return a;}           int Foo(){return a + b;}
```

```
4 };                              };
5 Base fnc/mem matrix:           Sub fnc/memmatrix:
6 a                               b
7 [x]Bar                          [y]Foo
```

Code 32: Function/Member matrices for classes related by inheritance.

It might very well be, that splitting either of the fields *a* or *b* might result in great deteriorated performance (if a correlation exists). In terms of correct split decisions COOP needs to temporarily attribute base class state to its subclasses resembling the behavior the compiler would normally do for us.

Hot/Cold splits inside a class hierarchy affect the subclasses of a splitted base class. As long as class individual traits are considered (in terms of possible correlation) are subclasses affected negatively by a split in its upper hierarchy? In most cases the opposite is the case. A split in a base class will effectively reduce the sub-class' size as well but it does not imply an automatic performance boost for the sub class. On the other hand, the split envisioned in Fig. 40 would

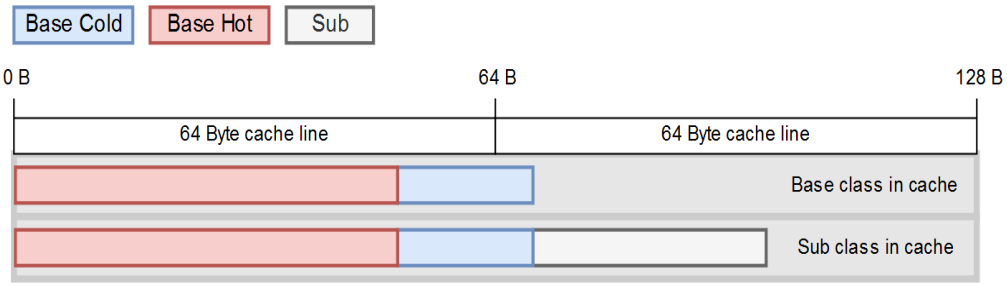

Figure 40: Example of how a Base class and a Sub class could exist in the cache. In this case externalizing the Base's fields will effectively not reduce the Sub class' size enough to reduce the necessary amount of cache-lines to encompass it.

even separate the Sub class' fields over two cache-lines.

Not only there is the possibility to worsen the data layout for a sub-class by splitting its base, we might also end up splitting a sub-class yet after injecting the base classes' fields to it the changes are irrelevant as those new fields completely change the data layout once more (see Fig. 41)! Attributing members to sub-classes solves those issues relatively easily but splitting a record that is involved in inheritance becomes complex quite fast considering that a hierarchy might be arbitrarily large and especially when the language allows for multiple inheritance. When fields are attributed correctly we can reason about why certain fields should not be split (they have correlation). The next step would be for a Sub class to deem a base class' fields cold. This is a bold step, as the base class might actually rely on it quite frequently. Again this could be expressed by a cost-benefit estimation as we did before for the individual fields, when deciding whether to split it or not.

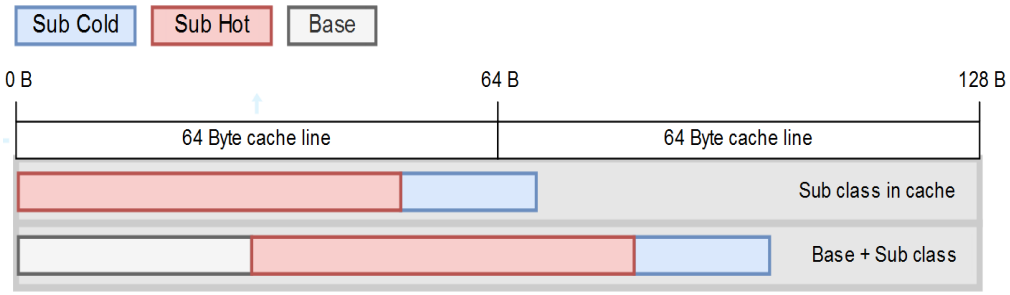

Figure 41: Example of how a split in a sub class might be ineffective as soon, as its base's fields are brought into the picture.

In case of a large inheritance hierarchy, instead of creating *N* cold structs, we could instead merge them to one cold struct. This way when a record *Sub* extends *C*, which extends *B*, which extends *A*, The *Sub* class would not end up having three cold struct pointers but one. Otherwise it would inherit it's bases' cold struct pointers as well

It is possible for automated Hot/Cold splits to comply with inheritance, however several factors need to be considered. A record's layout changes, as its bases change also structure padding and alignments will change as soon as a base changes. In order to correctly anticipate the data layout the inheritance hierarchy needs to be extracted and splits need to be executed in the correct order starting with the base classes so no changes to a record could result in consequent reevaluation of one of its sub classes.

8.2 Conceptual limitations

Until now we discussed issues, that for the most part can be solved applying further static analysis of the target source code. Unfortunately there are problems, that either partially invalidate our progress until now or plainly disallow COOP completely. This section will list several reasons to why COOP and this kind of optimization attempt is at least to be considered quite limited and actually to be precise not completely possible.

8.2.1 Templates

A major drawback in source-to-source transformations is its inability (or limitation) to reason about and also change records, that include templated code. Templates are a useful feature, that allow for a program to interact with the compiler, that will transform it to machine code. Whenever a record contains a field of a templated/generic type our heuristic can't work as intended as it considers a fields type size to be relevant. A generic type does not have a defined type size in our static analysis.
This does not dismiss templated types altogether. As templates are resolved at compile time

we could again do some of the compiler's job ourselves. While the Clang API will only find the generic templated version of a class definition for us, we could search for invocations of that class and retrieve template parameters to then construct the resulting class layout ourselves.
Our static analysis would then analyze all of the versions we can assemble for our templated record definition. However even though there will be different versions of that class definition in the end, source file transformations would affect all of the resulting versions. While our heuristics could find a good split in one version of it, another version might not benefit from that split.
As different versions of a templated record are determined at compile time we can retrieve all of the versions. There won't be unexpected 'new' versions of it being assembled at run-time. Only if each version of our templated definition profit from a split we could actually apply the respective changes.

While our source file changes limit us to consider all versions of a record an optimization performing on machine code or intermediate representations would actually have more optimization potential, as it could treat different versions differently.

8.2.2 Data layout VS access patterns

With the measurements we have seen, that an automatically improved data layout can result in better cache utilization and also performance, however it is bound to certain factors and conditions. The reasons for this we discussed in section 1.3. Its all about data locality effective cache-line utilization and ultimately data throughput. Data is performance [39, p. 272]. In section 4.3 we elaborated on how our automated hot-cold split might be null and void the moment the target program allocates it's instances scattered on the heap and how COOP would therefore allocate contiguous blocks of memory to then redefine the allocations in the target code so our theoretically reduced stride would actually translate to real optimization.
That was one example of how theoretical improvement on the data layout might have been ineffective and luckily in this case there was a solution. However there is another dependency to our optimization, that can prevent COOP from actually accomplishing improvement. If the access patterns of the target program don't abide the best practices we talked about in section 1.2.1 or if the program's task just does not depend on processing (iterating) its instances' members domain wise we again can't make use of reduced stride.
An improved data layout is a mighty tool only as long as the respective access patterns to it's instances allows for a minimized stride to result in improved cache-line utilization. It is therefore fairly easy to write a program, that will not profit from an 'improved' data layout and consequently not profit from COOP. In those cases the additional initialization overhead coming with our pool allocator will in most cases even result in deterioration of performance.

8.2.3 Structural changes in record layouts

In section 4.3.3 we explained why calculating a record's size in memory is not as easy as accumulating it's fields type sizes. Structure padding is done by the compiler to ensure optimal data loads. In order to determine, whether or not a split is beneficial we would calculate the original record's size including padding, to further reduce stride we would minimize padding by reordering the record's fields in a descending order.
This is a relatively trivial optimization, that depending on the original definition can have huge impacts on cache utilization for great amounts of data. In fact it is trivial enough to wonder why existing compilers don't do it.
Eric S. Raymond explains, that

> "*Automatic reordering would interfere with a systems programmer's ability to lay out structures that exactly match the byte and bit-level layout of memory-mapped device control blocks*" [41].

Other than that since we are altering an existing code base reordering fields has another implication. In section 8.1.2 we explained how field usages can stay undetected due to C/C++'s ability to manually create pointers. When given a source code there is always the possibility that for whatever reasons fields are accessed through pointers or even pointer arithmetic.

```
1 struct A{
2    int a = 1;
3    int b = 2;
4 } foo;
5 printf("%d\n", *(reinterpret_cast<int*>(&foo)+1)); //out: 2
```

Code 33: Simple example of why automated field rearrangements can't be considered legal.

Code 33 shows a simple example of how a record's field can be accessed without ever actually using it's identifier. Reordering fields inside a record would not break the target code, but induce bugs that are very hard to find and ultimately corrupt the semantic integrity of our 'optimized version'.
The follow up question is now: Why does COOP grant itself the liberty of reordering fields? The reason for this is that it already breaks code like Code 33 anyways, even without field rearrangements. COOP not only reorders fields it extracts them completely. Any pointer gymnastics, that would originally lead to a certain field can now no longer trust their record's data layout anyway. This is a conceptual contradiction to an optimization like ours and consequently invalidates COOP in its entirety! At least in terms of language compliance.

In conclusion an optimization process like COOP can only ever be executed in a reliable way when the user (or again further static analysis) can guarantee, that at no point in the target code base fields are accessed without invoking their identifiers.
Comparably harmless pointers hiding a field will corrupt the results of our metrics/heuristics

and might lead to hazardous splits and deteriorated performance but pointer arithmetic and type punning might actually result in corrupted semantics.

Implementation dependent layout specifics

In addition to the previously mentioned concerns about structural changes in a record's layout it is also worth mentioning, that determining a record's size in Byte actually depends on the underlying language implementation. In C the standard would not dictate the underlying type of an enum other than for it to be an integral type [41]. C++11 introduced typed enums. Untyped enums are guaranteed to be at least the size of an int. Also the position of virtual table pointers for dynamic dispatch varies depending on the compiler.

> "*Traditionally, it was placed after the class's user-declared datamembers. However, some compilers have moved it to the beginning of the class for performance reasons.*" [28, p. 258]

In fact the standard does not even demand dynamic dispatch to be implemented through virtual function tables and respective pointers.
Since our optimization operates in an environment where we can't use *sizeof*, nor invoke the target compilers behavior right now it entirely depends on assumptions and common cases, for example dynamic dispatch being implemented through an external table of function pointers and the virtual table pointer being at the beginning of the class.
Since virtual table pointers are inherently 'hidden' members (added by the compiler like padding Bytes) we can't unintentionally rearrange them in terms of descending type size, however when determining a record's size they need to be considered, as they occupy memory on their own and might as well induce additional structure padding.
To properly adjust to these circumstances the user would need to be able to share information about the target compiler with COOP so it can invoke compiler dependent behavior and adjust certain routines.

8.3 Conclusion

In the evolution of modern hardware the memory units the memory components of our today's systems pose a bottleneck in terms of throughput. Caching mechanisms implemented in complex memory hierarchies provide measures to counter this performance discrepancy but their correct utilization is the programmer's responsibility.
Object Oriented Programming is a powerful paradigm that allows for a programmer to model a problem in concepts he or she is familiar with. It offers a mechanism of abstraction that is intuitively adapted to a variety of problems. While a programmer benefits from abstraction in terms of maintainability it usually translates to a data layout, that inefficiently utilizes our modern hardware.

Data oriented design practices focus on the problems data and pragmatically rationalizes abstraction in favor of locality principles. This may lead to code that is inferior in terms of maintainability but can and often will result in a performance boost as the resulting data layouts accommodate caching mechanisms better.
Automated support for DoD principles can be implemented in a languages compiler and seamlessly convert a record definition into a structure that abides locality principles. In C++ however there is no intrinsic support for such data structures. Template meta programming has the ability to interact with the compiler and can be used to introduce DoD support, it is however limited and leaves optimization responsibility to the programmer.
In an attempt to create a generic optimization pattern our exemplary implementation COOP parses a target set of source files and attempts to determine a record's fieldproperties. This enables us to execute an automated Hot/Cold split based on metrics and heuristics we tailored to identify cold fields. Changes in the target source code can in fact produce a semantic equivalent that bears the potential to utilize the cache better than the original.

Automated data layout optimizations bear high potential to move time consuming effort that requires a certain amount of knowledge and expertise from the programmer to an optimization tool. The programmer's responsibility to efficiently utilize the cache can effectively be transferred to a particular step in a build process. This saves time and reduces error potential. Most importantly with automated optimizations of this kind a programmer can maintain certain levels of abstraction which especially for novices will result in a more comfortable mindset while still profiting from superior cache utilization.

These optimizations are however highly dependent on a variety of factors. A multitude of programs and records disqualify as a subject to our optimization merely by not relying on common access patterns. Also programs that don't work with great sets of data will effectively not profit from an optimization that changes the data layout. There is a range of conceptual contradictions that either make a point against static analysis to determine split opportunities or source-to-source transformations as they rely on a target compiler's implementation specific behavior.

After having implemented and worked with an exemplary attempt to an automated optimization tool like COOP I am now positive, that static analysis to identify split opportunities is possible, however very limited in its precision and depends on a variety of factors. Future work on such analysis steps are possible however their effort is conceptually limited (see the halting problem) and compared to PGO probably not worth it. Complex and project contextual analyzes on an ASTs are very time consuming for big projects. In comparison I subjectively can't rate them superior to profile guided optimizations. On the one hand it is automated and therefore can be run over night, doesn't require supervision of a quality assurance team, but on the other hand since it tries to make educated guesses about run time specific behavior based on static data it will ultimately always include an error of unknown significance. In cases of low complexity

and basic programming patterns that error might be small however as there are theoretically infinite scenarios a program can be constituted there is just no way of guaranteeing that a tool like COOP will always yield better results, than a split based on PGO data.
While I personally still support the idea of removing the programmer's responsibility to execute an optimization like a Hot/Cold split or transforming a record into AOSOA buckets, I can no longer endorse moving the responsibility from identifying split opportunities from the programmer to a tool that makes educated guesses.
It is perfectly possible to execute automated layout optimizations however the underlying information on which record/fields to split/externalize/turn into AOSOA should ideally be provided by the programmer. In this case for example a much better attempt at implementing COOP would be to either introduce an annotation that can mark a record/certain fields for a split or pass this information as command line parameters.
In order to comply with language features that resolve during compilation (like templated code) and to enable a tool like COOP to be integrated in an agile process it would also be advisable to move away from source-to-source transformations and instead execute LTOs (Link Time Optimization) or manually transform object files/executables. In case of LTOs we would need to develop compiler specific implementations. A more complex solution would only depend on the platform (*exe, ELF*) and manipulate object files/executables directly. Denis Bakhvalov explained in a 2018 articel on *Machine code layout optimizations* [7] how he automates function splitting, where he would extract cold code blocks from a function to new functions. Here code would be cold due to residing in a control flow statement that through profiling data cold be identified as rarely used. A similar project is Facebook's BOLT tool, that also operates on profiling data and modifies instruction placements in binaries. It is therefore independent of the compiler used to produce it [40].

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] BAKHVALOV, D.: *Machine code layout optimizations*, 2018.

[8] 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.

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

[10] BRYANT, R. E., O. DAVID RICHARD and O. DAVID RICHARD: *Computer systems: a programmer's perspective*, vol. 281. Prentice Hall Upper Saddle River, 2003.

[11] 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.

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

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

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

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

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

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

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

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

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

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

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

[23] HENZE, N.: *Stochastik für Einsteiger: eine Einführung in die faszinierende Welt des Zufalls*. Springer-Verlag, 2011.

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

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

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

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

[28] KALEV, D.: *ANSI/ISO C++ professional programmer's handbook*. Macmillan Computer Publishing, 1999.

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

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

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

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

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

[34] 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.

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

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

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

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

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

[40] PANCHENKO, M.: *Accelerate large-scale applications with BOLT*, 2018.

[41] RAYMOND, E. S.: *The Lost Art of Structure Packing*.

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

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

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

[45] STROUSTRUP, B.: *Einführung in die Programmierung mit C++*. Pearson Deutschland GmbH, 2010.

[46] STROUSTRUP, B.: *Moving right along*. 2010.

[47] WILLIAMSON, D. F., R. A. PARKER and J. S. KENDRICK: *The box plot: a simple visual method to interpret data*. Annals of internal medicine, 110(11):916–921, 1989.

List of Figures

List of Tables

List of Code

List of Abbreviations

OOP Object Oriented Programming

DoD Data oriented Design

AOS Array Of Structures

SOA Structure Of Arrays

AU Adressable Unit

AST Abstract Syntax Tree

PGO Profile Guided Optimization

LTO Link Time Optimization

A Anhang A

B Anhang B