Data Oriented Design

Organizing data for efficient processing

Germany, 2016

by

Jan Scheffczyk

Jana Deufel

Fachhochschule Bingen

Contents

1 Abstract

Within the last few decades CPU performance doubled almost annually, leaving the memory performance lacking behind. While new memory generations generally increase the total data throughput, the latency decrease is marginal at best. This disparity makes it difficult to reach a high CPU utilization, which can presently only be achieved by minimizing uncached memory reads, thus avoiding the latency-bottleneck.

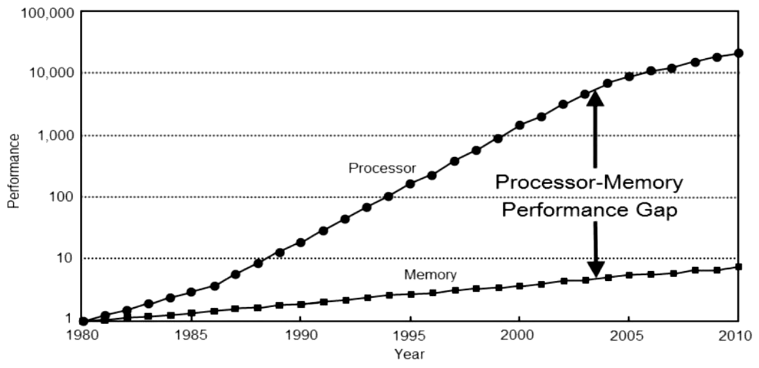
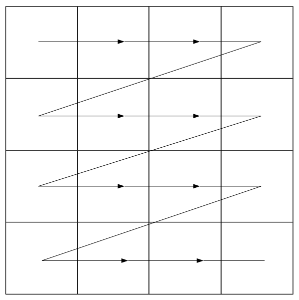


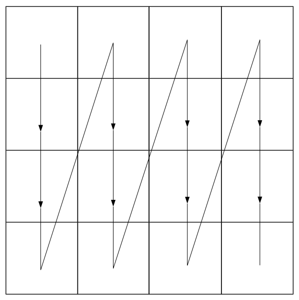
Figure 1: Processor-Memory Performance Gap [6]

This is the area Data Oriented design operates in. It focuses on the data, it's type, how it is laid in memory, and how it will be read and transformed. It bundles data and their transformation at the lowest possible level. This stands in direct contrast to more classical program-paradigms like Object-Oriented-Programming. Objects and their operations are bundled at a high abstraction level which often do not apply on hardware level, thus reducing the likelihood of an efficient implementation. Data Oriented design also promotes easier parallelization, high modularity, ease of testing and an excellent performance.

2 A simple example

The following example illustrates the fundamental idea of data oriented design. Suppose one wishes to access every element in a two-dimensional array. Two obvious possibilities come to mind:


(a) Row Major


(b) Column Major

The runtime complexity for both is $O(n^2)$, so equal performance is expected. The actual runtime-performance differs greatly however, as can be seen in figure 3. One should recognize that this difference can not be understood from a high level and abstract perspective.

"Understand the data to understand the problem" Mike Acton

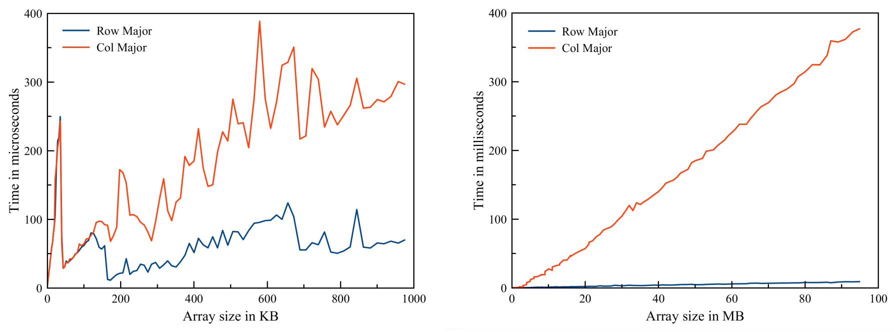


Figure 3: Row/Col Major with highlights for the cache line [1]

2.1 CPU Caches

The hardware component that causes the observed performance difference is the CPU Cache. Caches are special memory-modules which feature very low latency to reduce the initial problem of the processor-memory performance gap, at the cost of capacity. the cache is organized in a hierarchical manner. For each core there is a first level data cache (L1 DCache) and a second level cache(L2 Cache). The third level cache is usually shared across all cores which can results not only in coherency problems[2]. Most modern CPUs use 3 cache levels of which the first level is the smallest and fastest. Instead of loading data directly from the main-memory the CPU will load the data from the cache. If the needed data is currently stored in the cache (cache-hit) the data can be rapidly loaded into the CPU. If it is not (cache-miss), the data needs to be fetched from the next higher instance. The level 1 cache will acquire it's data from the level 2 cache and so on until cache hit or eventually main memory is reached. The data will be stored in a cache entry (write back) which consists of a section that stores the actual data called cacheline, a section that stores some of the physical memory address so future request can be identified and mapped to this entry called tag and some status bits. Since caches are fairly small it's content gets replaced quickly. So even if the instructions for a certain task have been loaded into the cache other tasks will replaces the original instructions. By the time the first task is called all it's instructions may already be replaced.

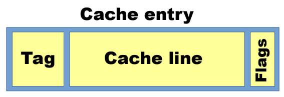


Figure 4: Cache entry

Notice that a cache entry always stores a full cache line i.e. even if only a single byte is needed a full cacheline is fetched from mem-

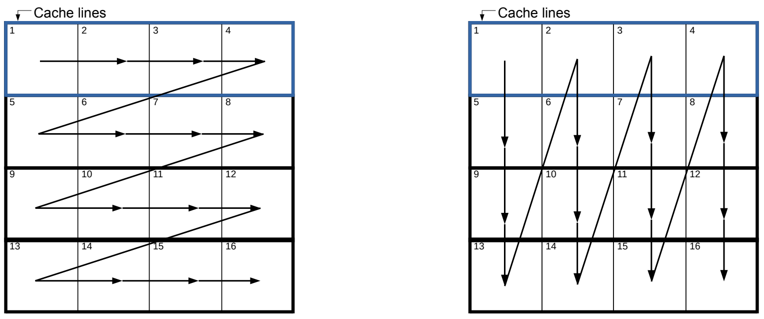
[1] Tests have been performed on a i7-4790K and 16GB DDR3 OS win7. The test program has been written in Java and is given in [1]

[2] The L3 Cache differentiates between 3 Cache hits, hit unshared, hi shared, hit modified

ory, which is typically 64Bytes[3]. Thus the data in close proximity of the requested memory-address is now also in the cache. Ideally all data that was fetched is actually needed for the current or an imminent instruction. A data-format that would enable such processing would be the ideal-data. This is realized by organizing the needed data in a continuous in section of memory. One very powerful tool to organize data in such a manner is an array.

"I don't know [data structure], but I know an array will beat it." Scott Mayers

Now the initial example can be understood. As seen in figure 5a the first four data-sets are stored within the first cacheline i.e. each access is routed to this cacheline, only causing a single cache miss. Therefore the whole Row-Major traversal only provokes further fetching of a new cachelines in step 5,9,13. Even better those cache misses can be eliminated by a process known as pre-fetching that will fetch data before it is needed. The compiler takes care of pre-fetching, however this is only possible if a simple access pattern, such as a linear array traversal, is used.



(a) Row Major (b) Col Major

Figure 5: Row/Col Major with highlights for the cachelines

[3] The cacheline size depends on the hardware and can differ from system to system. Mobile CPUs might have L1 cachelines of 32B and L2 cachelines of diffrent sizes [2]

In figure5b can be seen that the Col-Major provokes a cache miss on every step[4]. These cache misses cause the performance difference observed in figure 3. In fact memory and cache access is one of the most common and devastating bottlenecks in modern programming.

Cache	Size	Cachelines	Event	Cycles
L1	32KB	64 Bytes	cache hit	4
Instruction	32KB	N/A	N/A	N/A
L2	256KB	64 Bytes	cache hit	12
L3	varies	64 Bytes	cache hit unshared	26-31
			shared in other core	43
			modified in other core	60
			miss remote access	100-300

Table 1: Cache latencys for the 6th gen of i7 intel processors [5]

In summery large blocks of contiguous, homogeneous data that will be processed sequentially will keep the CPU busy. Where as fragmented data access or instructions will cause cache misses which in turn result in wasted cpu-times.

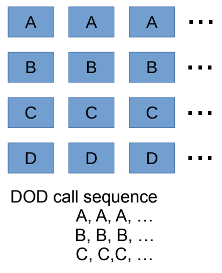
The implications for programmers can be condensed into three simple guidlines [7]:

- Small $\equiv$ fast
- Locality counts, stay in cache
- Predictable access patterns, to improve pre-fetching

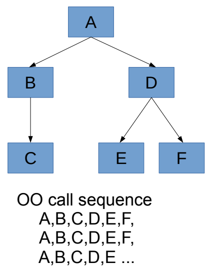
[4] This is only true for matrices that are so large that the first cacheline has been replaced before it is needed again.

2.2 Designing around the ideal data

The goal of data oriented design is to format input data in such a way that it can be efficiently processed. For current generations of CPUs the ideal data format consists of continuous and homogeneous memory layout as can be seen in figure 6a. Objects (OOP) on the other always form tree-structures because of their hierarchical nature (e.g. Inheritance-tress, containment trees, messaging trees). Tree generate a fragmented call structure, as can be seen in figure 6b, resulting in a frequent change of operations on different data i.e. both I-and-D-Cache data will be rapidly replaced causing cache misses.



(a) DOD Callsequence



(b) OO Callsequence

Figure 6: Call seq object oriented and data oriented [8]

To achieve the best possible data-layout it is often required to break down each object and isolate their components and then group components of the same type together.

2.3 Key-Value model

The principle of keys and values is often utilized in programming tasks. When searching for an entry the keys in the table and the searched key need to be compared until a match has been found. Remember that on a cache miss a full cache line will be fetched and stored so the rest of the cache line will be filled with the values attached to key. This data however is only needed when the current key is the one that is searched for, which is highly unlikely. The common case the next key is required for further comparisons.

One task that realizes this concept would be the modeling of car. A car object would contain attributes like id, weight, height, maxSpeed, etc. where the id represents the key and the other attribute represent the values. Classes store all their fields in the same memory location thus when the id is loaded the rest of the cache line is filled with the object's attributes. The search for a specific car object within a continuous data-structure is the exact same procedure as described above. This structure will scale towards the worst case regarding cache-misses and therefore performance.

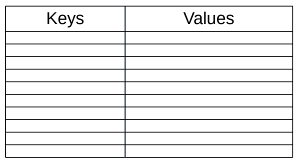


Figure 7: [3]

To form the ideal data structure the object needs to be separated into two groups, a key array that will store the ids and a value array that will store the rest of the object's-attributes. Each will have it's separate continuous piece of memory. To find the right index the key-array will be traversed and only data that is actually needed, the keys, will be stored in the D-cache. Once the correct key is found the index can be used to gather the corresponding data, from the value array, as is illustrated in figure 8.

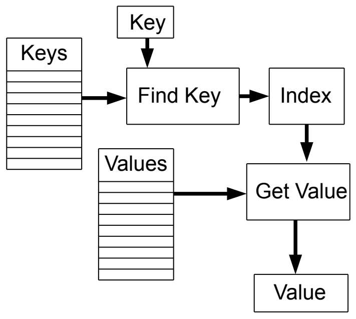


Figure 8: Improved key-value structure [3]

3 Data Oriented design put to the test

A linear search optimization is a rather specific example, a more general approach is considered next. An arbitrary method represented by the move() method of the car object, is considered and then a more hardware friendly version is discussed.

The object oriented car class can be examined in Code1. To simplify and condense the code for the example irrelevant attributes like id, weight, and so on, have been reduced into a single variable (unrelatedData) which will take up as much space as the individual attributes would have taken. In the object oriented approach all attributes for a specific car are stored within a car object as well as the method operate on this data.

```
class Car{
        vec3 position;
        vec3 speed;
        // 1 : data that is not relevant to the move() method
        char unrelatedData[256];
        void move(){
                position.x += speed.x;
                position.y += speed.y;
                position.z += speed.z;
        }

        // 2:  other methods
}
```

Example Code 1: Car class

To solve the same task in a data oriented manner the method and it's data need to separated from the class. When extracting a method the relevant input data needs to be identified first. In this case all unrelated data have already been masked. Position and speed data need to be separated from the car class and managed independently. In the next step the output data needs to be considered, in this case the updated position. There are different choices for the organization of input and output data.

One option is to create a wrapper for both input and output parameters. The input wrapper holds position and speed data and the output wrapper stores the updated position. The extracted method takes an array of input data, an array of output data and the size of the arrays. This is a common practice when the language of choice does not support classes but is just as valid otherwise.

```
struct InputMove{...}
struct OutputMove{...}

void moveAll(int size, InputMove inputs[], OutputMove outputs[]){...}
```

Example Code 2: The DOD approach

Another option is store the data in a manager object. The data itself is then stored directly in arrays, or another continuous data structure, as seen in Code2. Note that the use of the manager is not limited to the car class but rather to all objects that need to be moved in this manner. The data oriented Car object can add itself to the ManageMove to ensure that it is being processed, in this case moved. Through the returned index the object can access the current position and speed data.

```
class ManageMove{
        vec3 positions[];
        vec3 speeds[];

        unsigned int void add(vec3 pos,vec3 speed){...}
        void moveAll(){...}
}
```

Example Code 3: DOD approach with wrapper

3.1 Considering the expenses

To evaluate the expenses it also matters how the method is accessed. The object oriented approach iterates through all cars and calls the update() method(Code Example 4). This is a fairly common approach however it yields the worst performance. In a typical hierarchy the root class defines an update() method and each subclass overwrites it providing a unique implementation where various methods are called including move(). Therefore every call to an update() executes a different tasks which in turn requires different instructions. This will almost ensure that the I-cache has been completely replaced thus every call of move(), or any other method in fact, will cause an I-Cache miss. The instructions need to be loaded from memory which will cause 100-300 cycles of waiting.

```
for ( Car c : cars)
        // updates everything within the object that needs to be updated
        // among them is the move() method
        c.update();
}
```

Example Code 4: general update method

The data oriented approach simply calls moveAll() method on the manager object. Therefore the same instructions on all the data of the manager is used. Thereby only a single I-cache miss is generated for moving all objects. One might could further optimize this approach by explicitly using SIMD[5] operations.

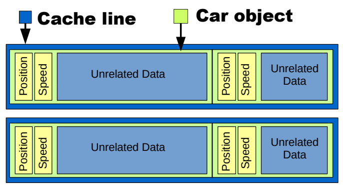


Figure 9: Car object

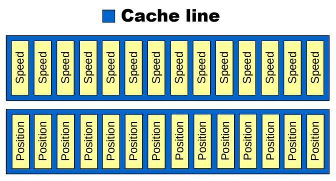


Figure 10: ManageMove Obj

One might argue that similar results can be achieved in object orientation by defining an interface Moveable. Creating a datastructure of the type Movable would allow to call the move() method for all objects no matter the actual type. However this would also neglect some of the object oriented notion that every object knows how to update itself. Further this would also break object encapsulation in a sense.

Next the D-cache misses need to be considered which depend on how the Car class is

[5] Single Instruction Multiple Data

allocated (as in code example1). If the allocation is scattered each access will provoke a D-cache miss. However even if the allocation is in a continuous manner and the method is accessed as in code example 4 the cache lines will store not only the needed but also the unrelated data ,as can be seen in figure 9. Thus if the object holds additional data the performance will degrade until the unrelated data will fill up the rest of the cache line. If the method needs data from other objects each access will yield another D-cache miss.

Again in the case of the ManageMove object only actually needed data is loaded thus all cache misses but the first are avoided (figure 10). Further more a simple access pattern was created that can easily pre-fetched. The unrelated data from the original class would be split up into other managers that perform operations on that data.

3.2 Test results

Because I-Cache misses are difficult to recreate in a closed test environment only D-Cache misses have been tested. The test have been performed on three different compilers which were set to highest optimization level respectively. As can be seen in figure 11 even in this very simple test the differences are substantial. The code for the test can be found in reference [1].

Large data fields are usually not stored directly in objects therefore a maximum size of 256Bytes for the unrelated data array has been chosen, figure 12b.A further increase of unrelated data results in further performance penalties however not nearly as steep as the first 256Bytes.

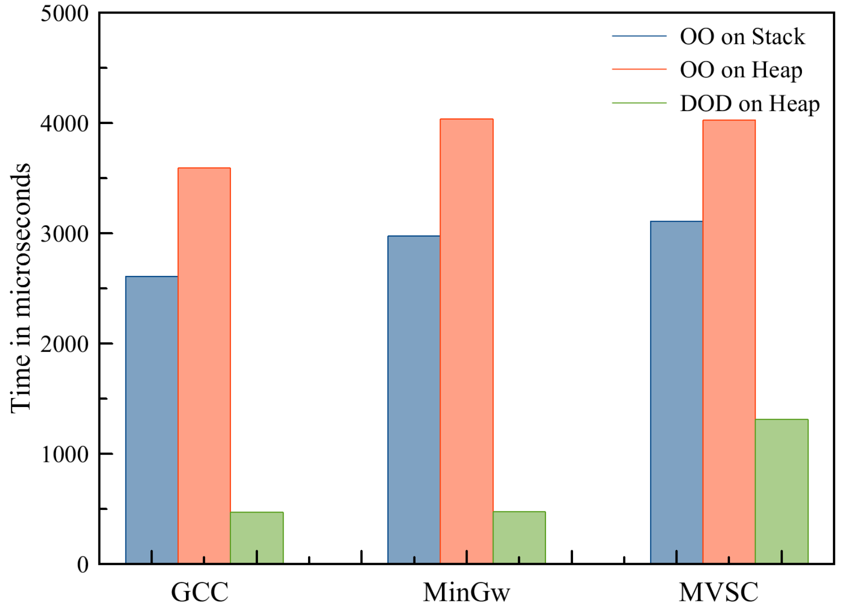
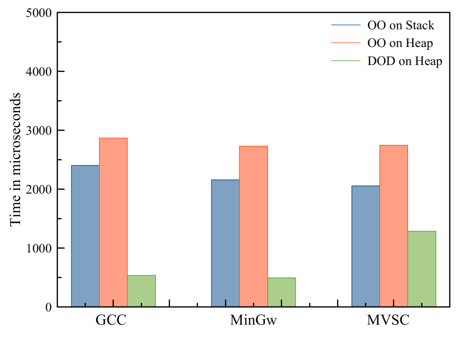
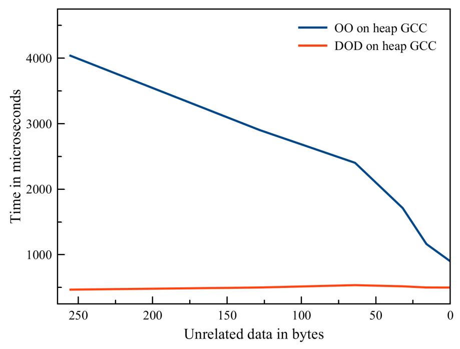


Figure 11: Test result with 256 Bytes of unrelated Data



(a) 64 Bytes of unrelated Data



(b) Progression from 256Bytes to 0

4 Final considerations

4.1 Maintenance

By separating methods and data from objects, highly independent structures are created. All the data needed to perform a task is stored directly in the manager object. These compact, single purpose manager object are easy to understand and modify because there are no further dependencies to other objects. An object oriented code base is highly hierarchical which introduces not only dependencies within each sub-tree but also dependencies between other objects. To understand a moderately complex application it is almost always required to consider data from a multitude of classes.

Also the context free approach allows for code reuse independent of application and context as long as the problem, thus the data transformation, is the same. Object orientation binds context to data in form of objects which can in turn hinder code reuse.

4.2 Parallelization

To take advantage of the higher core-count of modern CPUs multi-threading becomes increasingly important. Synchronizing object oriented code correctly without unnecessary blocking is very difficult and leads to locking of data to prevent concurrent access. This does not only add overhead it may also block threads leaving the CPU idling.

If every manager runs in a separate thread only the execution order needs to be synchronized. There is no need for any data synchronization since all data is independent. It is even easier to instantiate multiple threads within a single manager object. Then manager has a single task and a lot of data that all needs to be processed in the same manner. This is the perfect SIMD scenario and follows the same approach as GPU processing. In case of the ManageMove class each thread could run on it's own part of the array which guarantees that there will be no concurrent data access.

4.3 Testing

The independent entities also result in easy testing. Valid data can be generated and used as test input the output is checked to see if the transformation was correct. There are no hidden dependencies since all input data has been identified, this allows to test the complete algorithm.

4.4 Drawbacks

As discussed previously data oriented design differs heavily from the most common programing paradigm, object orientation. It is taught in most schools and universities. Thus most programs who read data oriented code for the first time will be confused as it differs from what they are used to.

It can also be challenging to interface with existing object oriented or procedural code. Data oriented design is often applied to a whole subsystem which will then be interfaced. This will still yield most of the data oriented benefits.

Furthermore data oriented design can lead to higher development costs due to the difficulty of writing perfectly isolated code. It also requires higher expertise from the developer especially if software for specific hardware is developed.

References

[1] Program code for the tests. https://gitlab.jan.m1234.de/jan/DataOrientedDesignTestsl.

[2] Qualcomm krait 300. http://www.7-cpu.com/cpu/Krait.html. cppcon [Online; accessed July 18, 2016].

[3] Mice Acton. Data-oriented design and c++. https://github.com/CppCon/CppCon2014/tree/master/Presentations/Data-Oriented%20Design%20and%20C%2B%2B, 2014. cppcon [Online; accessed July 18, 2016.

[4] Richard Fabian. Data-oriented design). http://www.dataorienteddesign.com/dodmain/l, 2009. [Online; accessed July 18, 2016].

[5] Agner Fog. *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical University of Denmark, 2016. http://www.agner.org/optimize/instruction_tables.pdfl [Online; accessed July 18, 2016].

[6] John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[7] Scott Meyers. Cpu caches and why you care. https://github.com/CppCon/CppCon2014/tree/master/Presentations/Data-Oriented%20Design%20and%20C%2B%2B, 2014. code::dive conference [Online; accessed July 18, 2016.

[8] Noel. Data-oriented design (or why you might be shooting yourself in the foot with oop). http://gamesfromwithin.com/data-oriented-designl, 2009. [Online; accessed July 18, 2016].