DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Overview and Implementation of Modern Memory Management for Graphical Real-time Applications

Maximilian Kaltenbrunn

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Overview and Implementation of Modern Memory Management for Graphical Real-time Applications

Übersicht und Implementierung Moderner Speicherverwaltung für Grafische Echtzeitanwendungen

Author:	Maximilian Kaltenbrunn
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	Sven Liedtke, M. Sc.
Submission Date:	15.08.18

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, _____ _____

Maximilian Kaltenbrunn

Abstract

This thesis provides an overview of the low-level CPU architecture of modern computer systems with regard to the memory subsystem. The correct handling of memory can greatly improve the performance of graphical real-time applications. For this reason, the functionality of CPU caches is explained in detail. This explanation serves as a basis for practices that can improve performance. In addition, optimization strategies for memory allocations are presented in the form of various custom allocators. The importance of these efforts is continuously supported by implementations and performance tests. These performance tests show that optimizing a program's source code towards good memory management and efficient cache usage can yield performance increases of multiple orders of magnitude. To ease memory management, modern C++ features are introduced that can assist in object lifecycle handling. Finally, an outlook on future C++ language features and suggestions for future work is given.

Contents

1 Introductrion

The global market for graphical real-time applications continues to grow. In 2015, video game sales in the US alone reached 16.5 billion[1] U.S. dollars [Zat17]. But video games are not the only applications that feature real-time graphics. David Michael, an independent video game developer, defines serious games as "games that do not have entertainment, enjoyment, or fun as their primary purpose" [Mic06]. Michael states that this does not necessarily mean that serious games are not entertaining, instead "there is another purpose".

An example for non-entertainment focused video game technology usage can be seen at the National Aeronautics and Space Administration (NASA). In 2016, Epic Games provided insight into a collaboration with the NASA [Slo16]. According to the published Article, the NASA uses Unreal Engine and virtual reality hardware to build a system that can be used to train astronauts on earth. In combination with custom hardware built by the NASA, the system assists in the preparation for missions on the International Space Station (ISS). Other usage examples of game technology outside of gaming include scientific visualization [FHW08], movie production [Pim18] or self-driving vehicle research [Dos+17].

Real-time rendering of complex environments or huge data sets is a computationally heavy task. The geometry of a 3D scene can exceed several million triangles, each of which has to be processed every frame at 30+ frames per second. Textures, lighting and shadow calculations, physics simulations as well as animations demand additional computational power. For a long time these high performance requirements could only be satisfied by stationary computers, either in the form of game consoles or desktop computers. Virtual reality increases these requirements even further because scenes have to be rendered at 60 to 90 frames per second [GD18; OCD18]. Rendering at 90 frames per second means that a new frame needs to be drawn every 11.1ms. If the 11.1ms target is missed, the resulting inconsistency and latency can have discomforting effects on the user. Therefore not a single frame may exceed this limit [GD18].

In the attempt to make virtual reality approachable to the average consumer, companies push towards standalone VR headsets. These headsets typically run on mobile hardware that is very similar to the hardware that can be found in modern smartphones. This hardware got a lot faster in recent years [HZR16], but it is still far slower than a desktop computer. To compensate for that limitation, it is necessary to optimize every part of an application. In video games this often means decreasing texture resolution and geometric complexity. In addition to these artist driven optimizations the source

[1] short scale billion (10^9)

code needs to be optimized as well. Making unconstrained use of a modern CPU[2] can be complicated. The CPU is the core of a modern computer system and interfaces with every other component or subsystem. A difficulty with using the CPU to its full potential is the fact that it is typically faster than the subsystems it interfaces with. As a result it is required to instruct the processor in a way, that obviates waiting for other components. The CPU cannot continue to process data when it has to *wait* for the data to be delivered by the memory subsystem. The performance gap between the CPU and the memory subsystem grew for years and continues to grow. Today, it is more important than ever to focus on good memory usage.

[2]Central Processing Unit

Language Disclaimer

The topics that are covered in this thesis (especially those in chapter 2 and chapter 3) are not standardized. The technology explained exists in many different variations. In this thesis we want to cover what is *typically* found in mainstream, consumer oriented hardware. Therefore loose language like *typically* or *usually* is used throughout this thesis. Absolute statements are not possible in most cases which is why these qualifiers are needed.

2 Overview Memory

The data that is required by a computer program has to be stored somewhere in RAM (Random Access Memory). In the earlier days of computing, addressing memory was a limiting factor which lead to the so-called memory segmentation. In 1978 Intel introduced memory segmentation on the Intel 8086 as a way of addressing more than 64KB of memory [90]. The problem was, that the Intel 8086 is a 16-bit processor and therefore used 16 bit memory addresses. 2^{16} ($65,536_{10}$) maps directly to 64KB of byte-addressable memory. Memory segmentation splits the memory in different *segments*. Memory addresses then do not refer to single bytes but to a larger block of memory identified by its offset to the segments base location. The offset is measured in "number of blocks". This concept allowed the Intel 8086 to address one megabyte of memory [90]. While the x86-64 architecture dropped the concept of segmentation almost completely the concepts of different memory locations are still widely used by modern programming languages.

2.1 Stack and Heap

Today the memory for a program is split into five different segments: Text, BSS, Stack, Heap and Data. The Text segment is where the program's actual code (the instructions) is stored. The BSS segment (from **B**lock **S**tarted by **S**ymbol) holds global and static variables that are not explicitly initialized in the source code. The Data segment holds global and static variables which have explicit initialization. The Stack contains data variables as well as additional information relevant for code execution. The Heap is reserved for additional memory a program can request during runtime.

This chapter provides an introduction to the Stack and the Heap. The other segments are omitted because they do not provide huge optimization potential.

2.1.1 Stack

As already mentioned, the stack contains not only application data but also data relevant to program execution. The exact usage of the stack is defined by a so-called calling convention. The calling convention determines where *e.g.* function parameter values are stored. Calling conventions differ between programming languages and compilers which is why there is no single correct explanation. The following describes how the stack is used in 32-bit x86 C [Fer+06].

When a function is called, the parameters of the function are pushed onto the stack first. Then the return address is pushed onto the stack. This is needed to return to the

```
1  int foo(int a, int b, int c)
2  {
3      int x = a + b;
4      int y = b + c;
5
6      int z = x + y;
7
8      return z;
9  }
10
11 int main()
12 {
13     return foo(11, 22, 33);
14 }
```

Listing 2.1: C++ source code snippet.

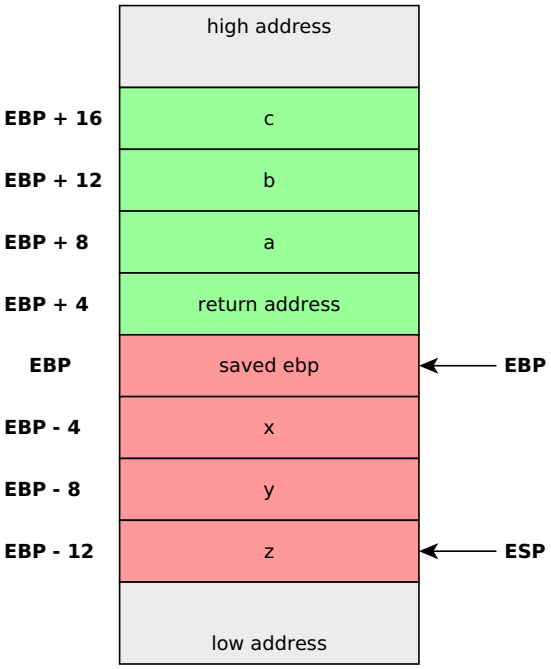


Figure 2.1: Stack right before the return statement in line 8 of Listing 2.1 is executed.

correct position in the program after executing the newly called function. The space after the return address is used to store local variables. Figure 2.1 illustrates this with the code sample from listing 2.1. Note that the stack shown in fig. 2.1 represents a stack of intels x86 architecture. Unlike the mental model that most people have in mind, this stack grows from high to low memory addresses.

EBP is a CPU register that is called frame pointer or base pointer. **ESP** is another CPU register that is used as the stack pointer. This pointer always points to the lowest stack element. The range between EBP and ESP is called the current *stack frame*. If an x86 program is compiled, the compiler defines the stack locations for function parameters and local variables relative to EBP. This enables access to these variables because the compiler can generate the required access instructions based on the assumption that EBP never changes. In this example the parameters `int a`, `int b` and `int c`, which are passed to the function `foo`, are placed on the stack in the range [EBP + 16] to [EBP + 8]. The return address is stored below that (at [EBP + 4]). At [EBP + 0] x86 stores an additional address. This is a saved EBP which is the EBP address that was valid before `foo` was called. The EBP gets restored to this address when `foo` returns. Everything below EBP is used for the local variables `int x`, `int y` and `int z` ([EBP - 4] to [EBP - 12]). Note that the stack addresses colored in green ([EPB + 16] to [EBP + 4]) are filled by the caller, while the red addresses ([EPB + 0] to [EBP - 12]) are filled by the callee. The return value of `foo` is not stored on the stack because it would be impossible to access after `foo` returns and EBP changes. Instead, the return value is stored in a special CPU register called EAX.

Some other values, that are not explained here can be are part of the stack as well. In x86, some CPU registers are defined as *caller-saved* and some are defined as *callee-saved*. If values that are stored in these registers are still required after a subroutine call, the caller or the callee is responsible for pushing these values onto the stack. The register values can then be restored when the subroutine returns.

The memory that is required to store the stack is provided by the operating system (OS) at program startup. If multiple threads are started during the execution of a program, the OS will allocate additional memory to allow each thread to have its own stack. Depending on the operating system the stack size varies. As an example: The default stack size of the test machine is 8MB[1]. If more stack space is required than the stack can provide, the program crashes due to a stack overflow. After a function finishes, the data is not freed. In C++ the destructors for local variables are called, but the memory itself remains untouched. Instead, ESP is set to EBP and EBP is set to the value it is pointing to. Adjusting ESP afterwards to revert the addition of function parameters is the callers responsibility.

The following limitations can be derived from these basic properties:

- The memory footprint / size of variables on the stack has to be known at compile time to generate the access instructions.

[1]The test machine is running x64 Ubuntu 16.04. The stack size was determined using the command `ulimit -a`.

- Data on the stack does not need to be allocated.
- Data on the stack is only valid in the current function and subroutines called from the current function.
  - Returning pointers or references to local variables is undefined behavior.
- The amount of memory the stack can provide is limited.

To circumvent these limitations programs can request additional memory. This additional memory comes from a memory region called Heap.

2.1.2 Heap

The Heap is a large memory pool that is managed by the operating system. If an application requires additional memory, this memory can be requested from the OS. If the application does not require the requested memory anymore, it can be returned. In C this is done using `void* malloc(size_t size)` to request memory and `void free(void* ptr)` to return it. In C++ both malloc and free are still available but usage is discouraged. Instead, the newer facilities `new T()` and `delete T` were used which correctly handle object oriented code by calling constructors and destructors. In modern C++, `new` and `delete` are not invoked directly anymore. They are now wrapped in more modern facilities using smart pointers [Sut14]. More details can be found in section 6.3.1 (Smart Pointers). `malloc` returns a `void*` that points to the start of a free memory region of the requested size. `new T()` returns a `T*` to the newly allocated object with its constructor already called. If the returned pointer is lost, the acquired memory cannot be returned to the operating system and is therefore unusable until the program exits. This is called a memory leak.

From a memory point of view all of these facilities are often the same because `new` and `delete` typically call a form of `malloc` or `free` internally. The GCC standard library libstdc++ calls `malloc` from `new` [Fre18] which is why we assume that `new` and `malloc` perform equal from a resource acquisition standpoint (*i.e.* ignoring the constructor call issued by `new`). Therefore, `new` and `malloc` are used interchangeably from now on.

2.2 Allocations

The process of obtaining memory from the operating systems Heap is called allocation. Allocations are executed by an *allocator*. In a C++ program the default allocator is implemented by the C++ standard library that is typically included as part of the compiler. This default allocator is used if the user does not override or overload it.

Allocation is not a simple task, especially with the performance requirements of modern software. Because of that, allocations are often a performance bottleneck without developers recognizing it. Since we assume that `new` and `malloc` perform equal, the following section highlights some problems with using `malloc`.

```
1 // Single allocation to store 5000 POINTERS to Particles
2 std::vector<Particle*> particles(5000);
3
4 for (int i = 0; i < 5000; i++)
5 {
6     // One additional allocation per Particle
7     particles.push_back(new Particle());
8 }
```

Listing 2.2: Example of malloc overuse

2.2.1 Problems with malloc

`malloc` is a system call as well as a general purpose facility. This means it has to be able to handle concurrently incoming requests of arbitrary size. It needs to be able to allocate memory blocks of a few bytes as well as memory blocks of several gigabytes. In fact, `malloc` takes a `size_t` as its parameter for the allocation size in bytes. On a 64-bit system this equals 2^{64} bytes ($\approx$ 18 Exabytes).

The required versatility does not come for free. `malloc` is fast but the constraints and requirements slow it down. High performance implementations of `malloc` can preallocate memory or handle differently sized requests with different allocation schemes to speed up the allocation, but this only helps to speed up some specific allocations. A general purpose allocator can never guarantee optimal performance for every possible request. Because of that calls to malloc should be kept as rare as possible.

Listing 2.2 shows an example of bad `malloc` use. In this snippet, `malloc` is called for every single one of the 5000 particles. In listing 2.3 a simple change reduces the malloc call count by 5000. This is achieved by not calling `new` for every Particle. Instead, a local variable is copied from the stack to the vector's pre-allocated memory. The memory required for all particles is allocated in line 1.

Comparing the execution time of listing 2.2 and listing 2.3 shows the overhead malloc introduces (see fig. 2.2). By simply changing the allocation pattern this programs execution time could be reduced by a factor of 2.66. Not all allocation problems can be solved by just altering a few lines of code. In section 6.1 another approach is shown using custom allocators.

```
1 // Single allocation to store 5000 actual PARTICLES
2 std::vector<Particle> particles(5000);
3
4 for (int i = 0; i < 5000; i++)
5 {
6     // No additional allocation required
7     particles.push_back(Particle());
8 }
```

Listing 2.3: One way to fix the massive amount of allocations is to move the allocation out of tight loops.

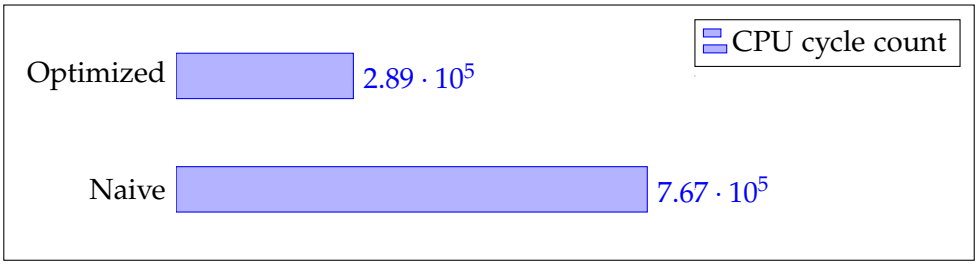


Figure 2.2: Execution times of listing 2.2 and listing 2.3

3 CPU Architecture

Modern CPUs are much faster compared to processors from 1985. In 1985 the CPU core frequency was similar to the frequency of the memory bus [Dre07]. This means that accessing main memory was only slightly slower than accessing a CPU register. Memory accesses therefore did not have to be optimized. In the years after, CPUs became faster. In particular, the clock frequency was increased rapidly by chip manufacturers. DRAM[1] manufacturers were unable to keep pace with this rapid development. Although it was possible to build very fast memory cells (known as SRAM [2]), the production of these was (and still is today) much more expensive than DRAM. The reason for that is the structure of SRAM compared to DRAM. An SRAM cell requires four to six transistors while a DRAM cell only requires a single one [Dre07]. Because of that it is uneconomical to replace the entire DRAM of a PC with SRAM cells. As a result memory accesses got slower compared to CPU cycles, which can have a negative effect on performance. If the data a process needs cannot be delivered fast enough to the CPU, the CPU has to wait and spend time idle. Since the problem could not be solved by memory manufacturers, the chip manufacturers introduced the concept of the memory hierarchy: Instead of replacing the entire main memory with fast SRAM, they include small SRAM memory regions directly into the CPU and try to handle as many memory requests as possible from those. These regions are called caches.

3.1 CPU Caches

A CPU cache is a hardware cache that aims to reduce the time it takes to access data from main memory. This is done by placing caches between the CPU and the main memory. If data is requested from main memory which is not already cached (a *cache miss*) it has to be loaded into the cache. If the data is required a second time it can be read from the fast cache instead of waiting for DRAM (a *cache hit*). Caches are not standardized and therefore differ between CPUs. Throughout this section the Intel Core i7-6700k is used as an example. The i7-6700k has three levels of cache. Details are listed in table 3.1

[1] Dynamic Random Access Memory - The type of memory that is used for main memory in modern computer systems.

[2] Static Random Access Memory

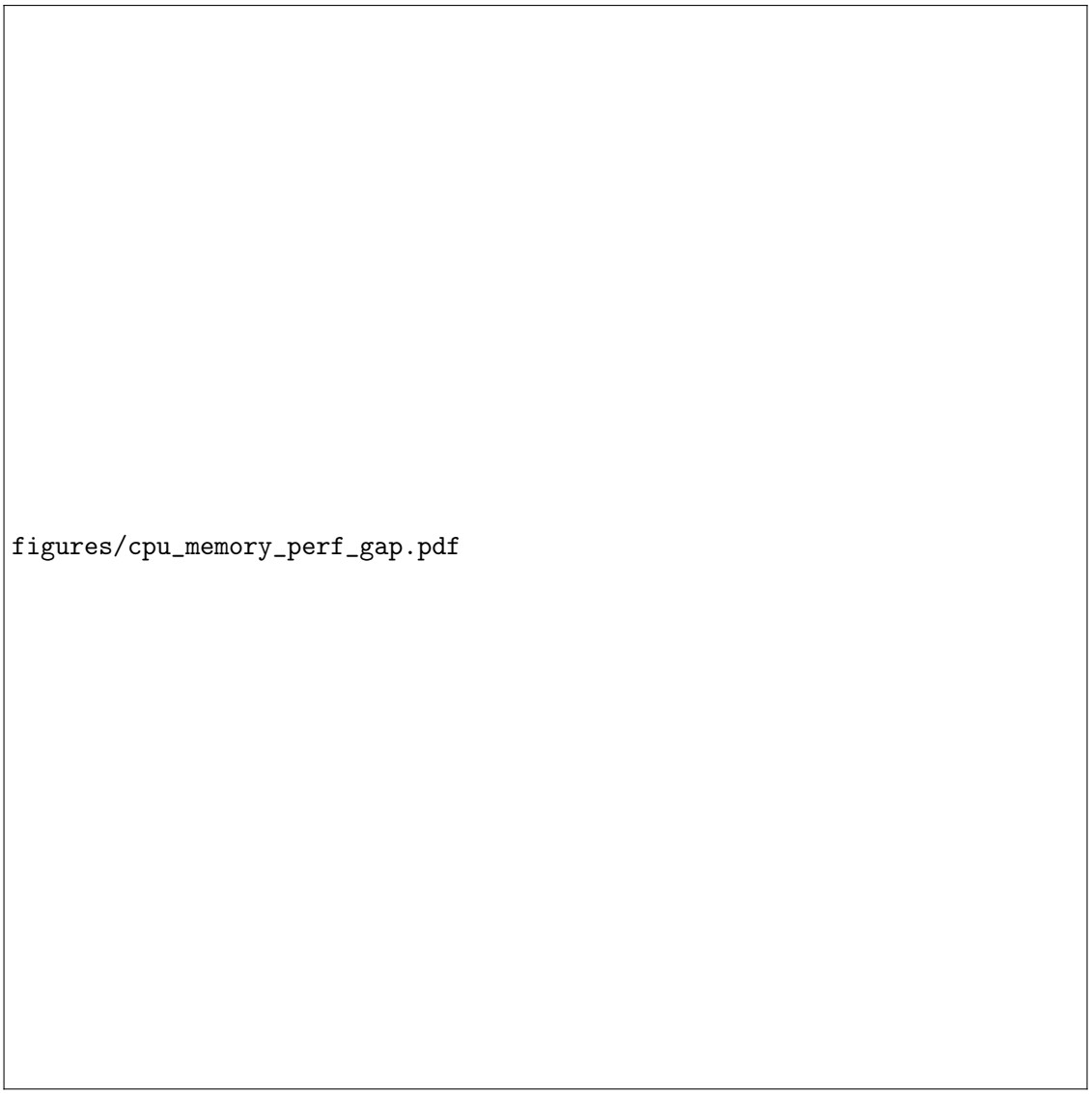


Figure 3.1: Performance comparison between CPU, DRAM, SRAM and Disk / SSD [BO10]. Notice that CPU cycle time matched DRAM and SRAM access latency in 1985.

Cache	L1 data	L1 instruction	L2	L3
Size	4 x 32 KB	4 x 32 KB	4 x 256 KB	8 MB
Associativity	8-way	4-way	4-way	16-way
Inclusivity	-		exclusive	inclusive
Latency	4 cycles		12 cycles	42 cycles

Table 3.1: Cache levels of the Intel Core i7-6700k

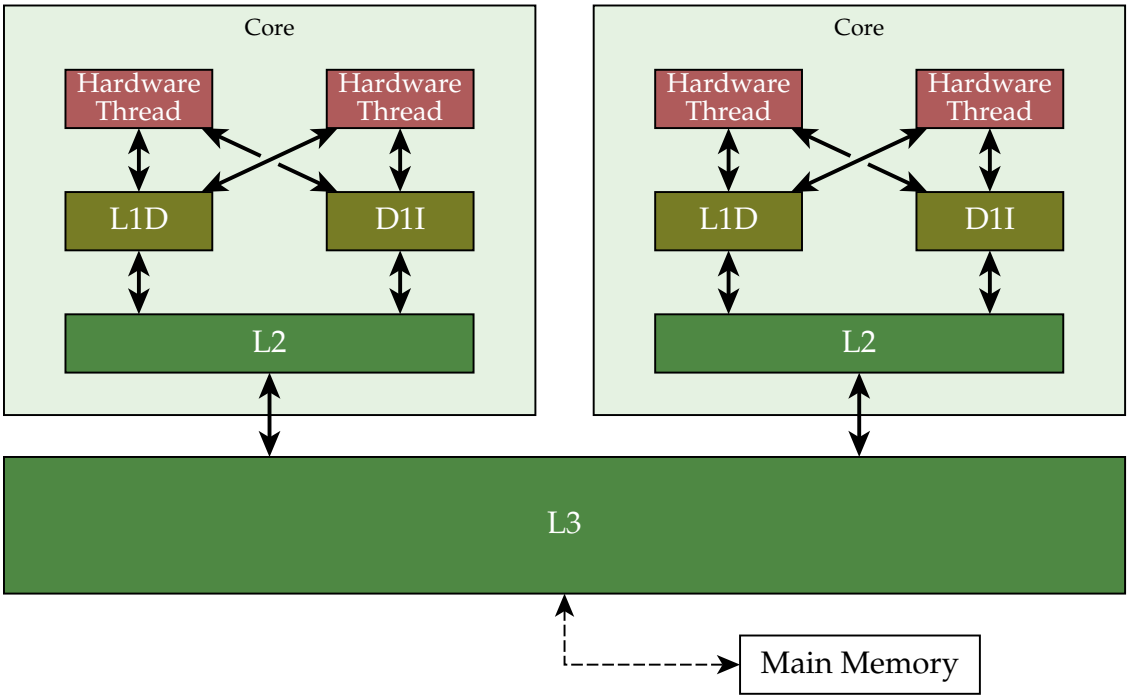


Figure 3.2: Cache hierarchy of the Intel Core i7-6700k. Note that only two of the four cores are shown here.

3.1.1 Data vs. Instructions

Table 3.1 shows two different level 1 caches: *L1 data* (short L1d) and *L1 instruction* (short L1i). L1i is a dedicated level 1 cache that stores a program's instructions. Since a program often flows linearly, a CPU can accurately predict which instructions (which in this case can be seen as plain data) will be required in the future. Because of that the instructions can be loaded into the fast instruction cache ahead of time to remove the DRAM access latency (in many cases) completely. If the CPU had to wait for instructions to be delivered from main memory it would stall until the instructions are available. This is known as *starvation*. L1i is not part of this thesis, it is only mentioned for the sake of completeness. Nonetheless, programmers can optimize instruction cache usage. The interested reader is encouraged to inform themself about branch (miss-)prediction.

L1d on the other hand is a level 1 cache dedicated to data. Data in this case means anything that is not an instruction, like a program's variables. Note that the distinction between data and instruction caches is typically made only on the lowest cache level. Higher cache levels drop that distinction and provide a common place for both data and instructions.

The rest of this thesis will focus on data caches since optimizing for data cache usage can yield huge increases in application performance.

3.1.2 Inclusive / Exclusive

Cache levels can be either inclusive or exclusive. An inclusive cache contains all the information stored in one of the lower level caches [Dre07]. In the case of the i7-6700k the level 3 cache is inclusive. It holds all the data available in L1d, L1i or L2. If data is requested from main memory, it is stored in the corresponding level 1 cache for direct use. In addition to this, the data is also loaded into the larger inclusive level 3 cache. If this data is evicted from L1 (because other data requires this space), DRAM memory access could still be avoided because the data may be present in the level 3 cache.

Exclusive caches behave the opposite way: The i7-6700k's level 2 cache is exclusive. Accordingly, it does not store the information that is in the level 1 caches. If memory is required from main memory, it is only stored in levels 1 and 3, as already mentioned. Only when the data is evicted from L1, it is written to L2. This is the only way of filling the level 2 cache in this case. When the data is needed in L1 again, it is *moved* from L2 to L1 [Dre07].

3.1.3 Associativity

Data loaded from main memory into the cache is identified by its main memory address. But there is a problem with this approach: The very large address space of the main memory must be projected to a few positions in the respective cache level. It is unavoidable that several addresses of the main memory are mapped to the same position in the cache. If that happens, a collision occurs, existing data is overwritten and thus evicted from the cache (level). To avoid the probability of a collision, modern CPUs can switch to alternative cache positions in the event of a collision. If the CPU can switch to any other position in the cache, the cache is called *fully associative*. The opposite (no other position in the cache can be used) is called *direct mapped*. If one of N different cache positions can be used for any given main memory address, the cache is *N-way associative* [Dre07].

Chip manufacturers have to decide how high the associativity can be. A higher associativity reduces collisions and therefore avoids that data gets evicted from cache erroneously. However, in case of a high associativity many positions in the cache must be checked to see if a certain memory address' data already exists in the cache. This increases cache access latency [Dre07].

3.1.4 Cache Line

Processors do not manage the cache at maximum granularity. If a 32 bit value is requested from main memory, not only the required 4 bytes are actually loaded into the cache. Instead, a full *cache line* is read. A cache line holds a continuous, fixed size memory range and every main memory address maps directly to one cache line. Since caches are not standardized the cache line size varies from architecture to architecture. In most consumer oriented CPU architectures the cache line size is 64 bytes (512 bit) [ZVN03]. From now on we assume a cache line size of 64 bytes.

Knowing of the existence of cache lines and how they behave is critical for optimizing code. A worst case example that arises from cache lines is only requiring a single byte from a memory location where the adjacent data is irrelevant. Even though a single byte of information would be sufficient, 64 bytes are loaded from main memory if the cache line that is needed for the requested memory address is not cached yet. 63 bytes of the loaded 64 bytes are not used. That means 98.4% of the loaded data is wasted and only clogs the memory bus. Figure 3.3 visualizes this problem.

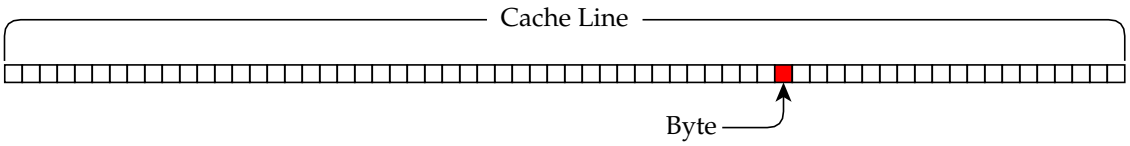


Figure 3.3: Visualization: Each square represents one byte of a 64 byte cache line. If only a single byte is required, 63 bytes are unnecessary load on the memory bus.

3.1.5 Cache Eviction

The term cache eviction was already mentioned in previous sections. Just like anything related to CPU caches, the eviction policies differ between CPU architectures. Generally speaking, cache eviction refers to the process of removing a cache line from a CPU cache or cache layer. As described in section 3.1.3 (Associativity) this is often necessary when new data is loaded into the cache. Which of the N cache lines of an N-way associative CPU cache gets selected for eviction is dependent on the cache eviction (or cache replacement) policy. Because the replacement policy directly influences cache efficiency, CPU manufacturers developed complex algorithms and heuristics to select the cache lines. Detailed knowledge of the cache eviction policies is typically not something that helps with optimizing code. Therefore they are not part of this thesis.

3.1.6 Coherency

Sorin *et al*. define cache coherency as:

> Coherence seeks to make the caches of a shared-memory system as functionally invisible as the caches in a single-core system. Correct coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores. [SHW11]

What this means is that caches operate fully invisible to programmers. Cache coherency therefore allows programmers to write code without having to know that caches even exist. While this might have performance implications, it is guaranteed that every

memory access behaves exactly as if the data was not cached in the first place. This includes the synchronization between multiple caches which might have copies of the data from the same main memory location.

Coherency in multi-core processors

The coherency definition provided above includes an important piece of information. Caches shall behave "as the caches in a single-core system" [SHW11] even if they operate as part of a shared-memory system. In fig. 3.2 it is clearly visible that in case of the Core i7-6700k multiple cores have distinct level 1 and level 2 caches. Therefore a single cache line can be stored in multiple caches which are physically located in different CPU cores. This is not a problem as long as the data in that cache line is only *read*. As soon as one of the cores *writes* data to that cache line, the CPU has to ensure that the other cores do not read from their version of the cache line since the data is now outdated. This is achieved by marking the cache line *dirty* in the CPU cache of the writing core and *invalid* in all other cores [Mey14]. The dirty flagged cache line then has to be written main memory, allow the other caches to load the correct data. CPUs can use use higher level shared caches or other dedicated implementations to speed up this process [SHW11].

A scenario highlighting this functionality is described in chapter 7 (Multithreading). For now the fact that coherency is guaranteed on hardware level is sufficient.

3.1.7 Prefetching

In section 3.1.1 (Data vs. Instructions) it was stated that CPUs can predictively load instructions into the instruction cache because instructions are typically laid out linearly in memory. Similar functionality is desirable for data caches as well, because it can shrink (or even remove) the effective DRAM access latency. To do this, the CPU tries to predict which data will be required in the near future and load this data into the data cache. This is called *prefetching* and is executed and instructed by the hardware itself. Neither the operating system nor application code instruct the CPU to prefetch a certain memory address. In some edge cases manual prefetching might be beneficial which is why CPU manufacturers provide libraries to do so.

Since predicting memory accesses is impossible in many cases, CPUs typically use a very simple approach for prefetching and rely on the programmer to access memory accordingly. If the CPU detects that accessed memory addresses follow a very simple, reoccurring pattern, it can assume that the pattern continues. This information can then be used as a basis for prediction. Details on how this knowledge can be transferred into cache efficient code are given throughout this thesis.

3.1.8 Performance measurements

Table 3.1 already showed the access latency for the Core i7-6700k. We therefore already know that a L1 cache hit is about 10.5 times as fast as a L3 cache hit. Mike Acton, princi-

```
template<size_t Pad>
struct Node
{
  Node* next;
  size_t pad[Pad];
}
```

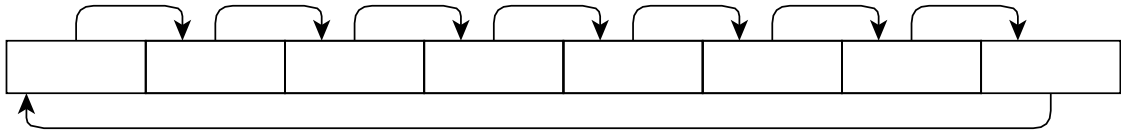
Listing 3.1: Nodes of a linked list used to measure cache effects.

ple programmer at Unity Technologies and former Engine Director at InsomniacGames, measures efficiency and game performance in "last level cache misses per frame" [Act14]. To complete the picture we need a full performance comparison including DRAM access latency.

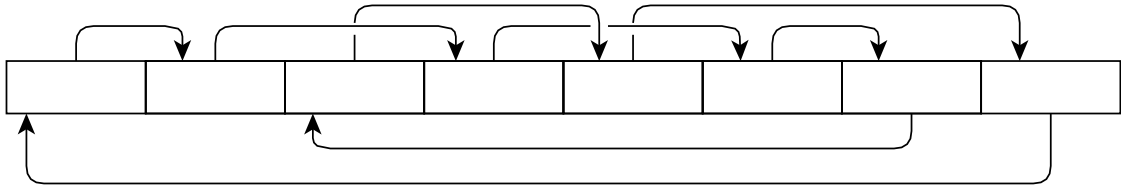
We repeated the measurement process described by Ulrich Drepper in "What Every Programmer Should Know About Memory" [Dre07]. These measurements were originally made in 2007. The new measurements therefore serve two purposes: First, to measure cache effects on a real system and validate the caches effects on performance. Second, to test if the caches behave similarly (in theory they should behave the same) as in 2007.

All measurements are performed by creating a singly linked list which is built using the Node data structure in listing 3.1. The list is then traversed multiple times using only the `Node* next` pointer even if the nodes are laid out linearly in memory. Because the lists are traversed multiple times we expect them to stay in the lowest cache they can physically fit in. This is not a real world scenario. Normally a list is iterated only once with other work happening before and after. What this test targets is the measurement and visualization of cache effects. The result we are looking for is the time it takes to access a single Node in this context. This equals the amount of CPU cycles per next pointer dereference. Since we want to compare the cache efficiency of linked lists with different Node paddings, we fix the size of the linked list. The size is called working set size. A linked list with a working set size of 2^N bytes contains $2^N / sizeof(Node)$ nodes. Each node's size, on a 64-bit operating system, equals $(8 * Pad) + 8$ bytes. A node with a padding of 7 (that is $(8 * 7) + 8 = 64$ byte) is exactly the size of a single cache line.

Two different list memory layouts are tested to isolate and visualize potential optimization by the hardware prefetcher. If the hardware prefetcher can speed up the list traversal, we should see that effect best if the list nodes are laid out linearly in memory. The next pointer dereference is a simple and therefore predictable memory access pattern in that case. To isolate this speedup, we compare these results with access times measured when the elements are aligned randomly by shuffling the list elements before the first traversal. The different layouts are illustrated in fig. 3.4. All test runs are executed on a system with an Intel Core i7-6700k running at 4.5Ghz and 32Gb of DDR4-2666 memory.



Linear / Sequential



Random

Figure 3.4: Memory layouts used for testing

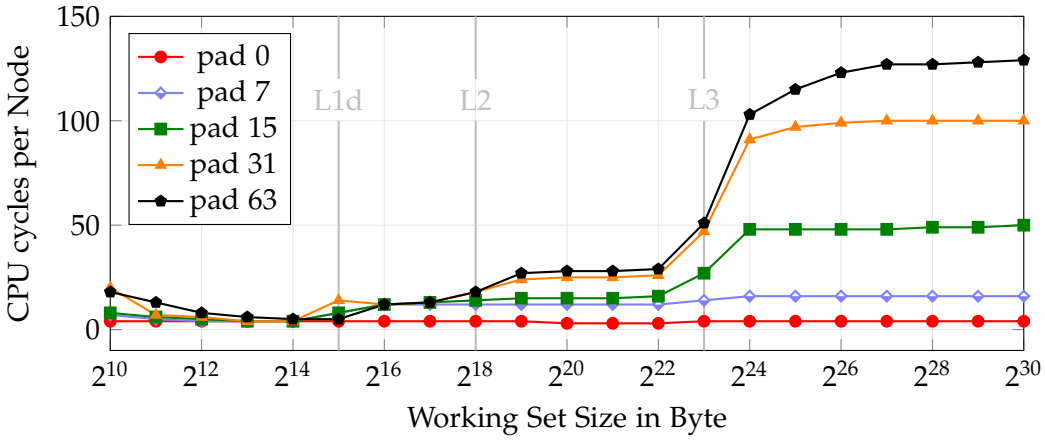


Figure 3.5: Traversing the linearly laid out linked list.

Linear Traversal

Figure 3.5 shows the performance test results of traversing the linear linked list for various different Node paddings. The results for a working set size of 2^{10} can be ignored. Since our test program has to share the CPU cache with the rest of the system, the working set is too small to filter out any noise introduced by system background processes. This is especially true when nodes with high padding values are tested because in that case, the total number of nodes in the list is very small.

What is immediately noticeable are the three steps at about 2^{15}, 2^{18} and 2^{23}. Those steps are easily explained: The cache sizes of L1d, L2 and L3 on the test system are 32KB (2^{15} byte), 256KB (2^{18} byte) and 8MB (2^{23} byte) respectively. We would expect to see a sharp edge in the graph but instead the edges are smoothed. In a perfect measurement

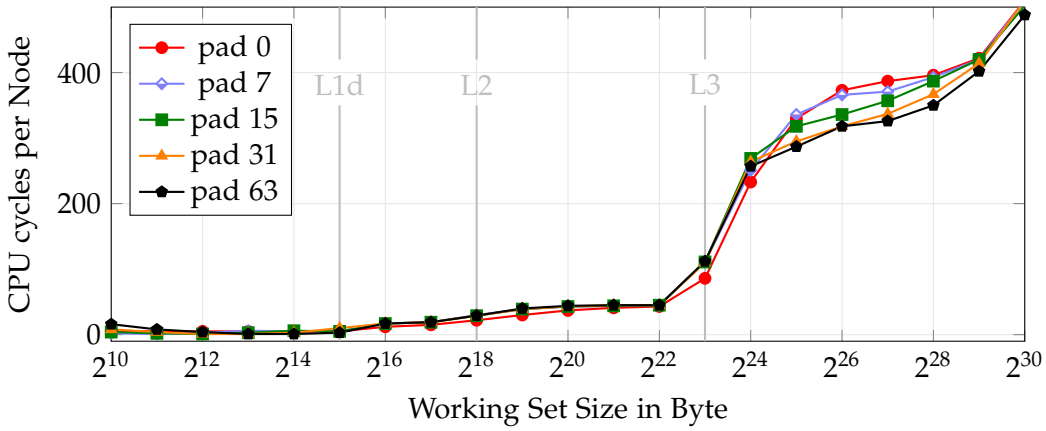


Figure 3.6: Traversing the randomly laid out linked list.

these edges would be sharper, but background processes of the operating system as well as the hardware prefetcher do not allow a perfect, isolated benchmark.

Random Traversal

Figure 3.6 shows the benchmark results of traversing the same linked list after shuffling the list elements physically in memory. It is immediately clear that this random access performs worse than the linear access. The differences between node paddings are gone completely, all test cases perform about equal.

Discussion

So why do we see these big performance differences? One might suspect that the performance differences are fully introduced by the prefetcher because of the predictability that comes with the linear memory accesses. To test this suspicion the same tests were repeated with the hardware prefetcher turned off.[3] Figure 3.7 and fig. 3.8 show the results of linear and random access respectively. While the prefetcher seemingly managed to decrease access latency (especially with working set sizes larger than the Level 3 cache) huge differences remain. In particular, lists with small node padding perform way better when traversed linearly even with the hardware prefetcher disabled. The reason for the far better performance when accessing the linearly laid out linked list is cache line usage. Section 3.1.4 already showed this effect briefly. In the case of zero padding (and therefore `sizeof(Node) == 8`)[4] a cache line can hold $64/8 = 8$ Nodes. That means for every cache miss, there are 7 cache hits immediately after each miss, because the following 7 nodes were already loaded on the same cache line. The same scenario when accessing nodes in the random list is different. If a cache miss occurs

[3] The hardware prefetcher on some Intel CPUs can be controlled using a Model-specific register (MSR) [Vis14].

[4] On the 64-bit test system

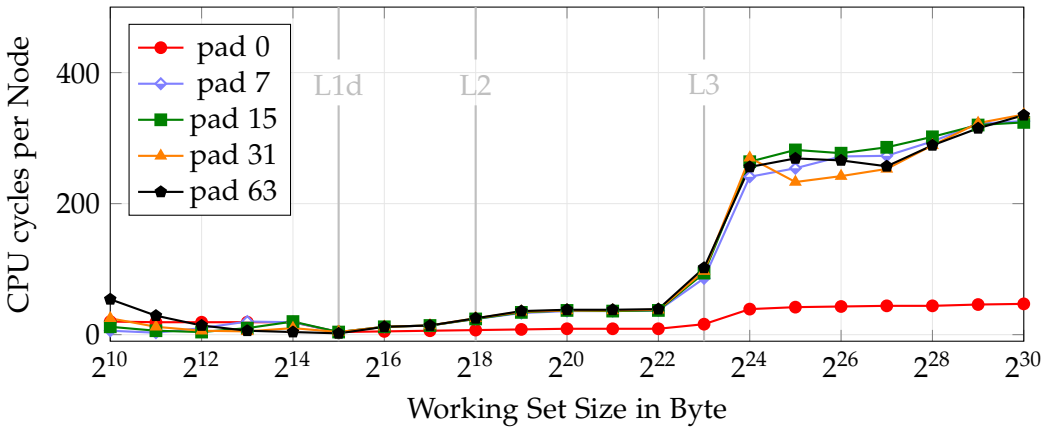


Figure 3.7: Traversing the linear laid out linked list with the hardware prefetcher disabled.

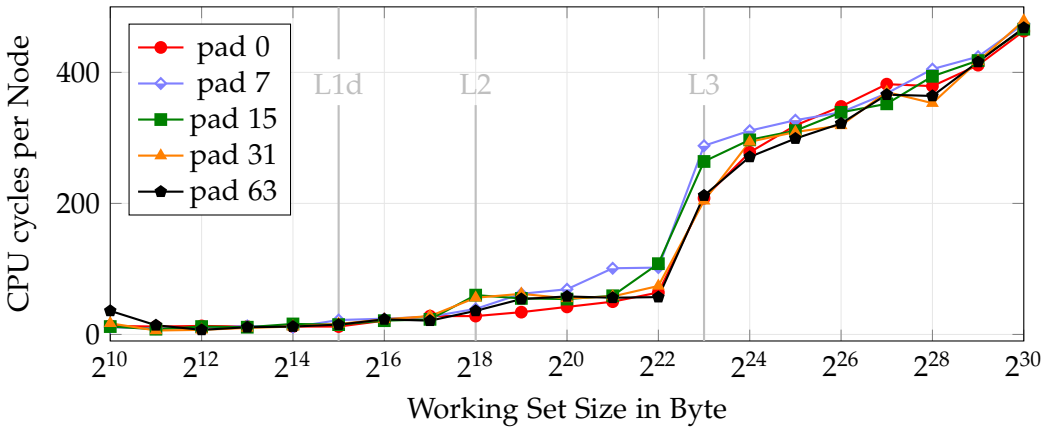


Figure 3.8: Traversing the randomly laid out linked list with the hardware prefetcher disabled.

and the cache line is loaded into the cache, the probability that the same cache line is accessed again right away is tiny. The larger the working set, the higher the probability that a cache line is either not yet in cache or the cache line is already evicted because other cache lines had to be loaded. The key point for the big performance differences is therefore the cache line usage, also called *locality*. This suspicion has been proven by logging cache hits and misses using Valgrind[5]. For details, see table 3.2. The linear linked list and its memory access pattern is called *cache friendly*. How the concept of cache friendliness and locality can be converted to actual code can be seen in section 5.1.1 (Cache Friendliness) and section 6.2.1 (AoS / SoA).

[5] "Valgrind is an instrumentation framework for building dynamic analysis tools. Cachegrind [one of callgrinds standard tools] is a cache and branch-prediction profiler."[VAL]

Type	Pad	Total reads	L1d miss	L2d miss	L1 miss rate	L2 miss rate
Linear	0	805 305 897	100 663 315	100 663 315	0.125	0.125
	7	100 663 273	100 663 253	100 663 253	0.999	0.999
	15	50 331 657	50 331 637	50 331 637	0.999	0.999
	31	25 165 829	25 165 829	25 165 829	1	1
	63	12 582 925	12 582 904	12 582 904	0.999	0.999
Random	0	805 305 877	805 089 322	780 475 513	0.999	0.969
	7	100 663 253	100 663 253	100 663 253	1	1
	15	50 331 637	50 331 637	50 331 637	1	1
	31	25 165 829	25 165 829	25 165 829	1	1
	63	12 582 925	12 582 925	12 582 925	1	1

Table 3.2: The recorded cache misses for the different test cases with disabled hardware prefetcher. We can clearly see that in the case of linear traversal with 0 Node padding there are exactly seven cache hits for every miss ($1/8 = 0.125$). From padding 7 upwards the node requires at least one full cache line. Without the prefetcher every access is a cache miss then.

3.1.9 Case Study: Matrix Multiplication

In this section the explanations and information given throughout this chapter shall be evaluated. Since the examples given so far were synthetic benchmarks, this section will focus on a real world problem: Matrix multiplication. The result c of the multiplication of two matrices a and b is defined as:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj} \tag{3.1}$$

The value of c_{ij} equals the sum of multiplying the entries of the ith row of matrix A and the jth column of matrix B. As a result matrix multiplications require N^3 multiplications for $N \times N$ matrices. The computational complexity of the naive approach therefore is $\mathcal{O}(n^3)$. The fastest known algorithm achieves $\mathcal{O}(n^{2.373})$ [Le 14].

Since the naive way for matrix multiplication involves traversing a matrix in column major order, the cache efficiency of this operation is low. Not only is a column major traversal required, it is required multiple times for the same columns. When multiplying two matrices A and B, every column of B needs to be traversed once for every row of matrix A. The distance in memory between the elements of a column equals the width of the matrix. A similar case could be seen in the previous performance test. A cache miss for every read during the traversal is expected if the matrix exceeds trivial dimensions. An implementation of this naive approach can be seen in listing 3.2.

Since cache hits are extremely fast compared to cache misses, an algorithm with a higher computational complexity but better cache efficiency could be faster than the naive approach. To check this, the following test was executed: When multiplying the two matrices A and B, B is transposed to get B^T before the multiplication. This allows

```
1  Matrix multiply(const Matrix& a, const Matrix& b)
2  {
3      Matrix result = Matrix(b.get_width(), a.get_height());
4
5      for (int x = 0; x < result.get_width(); x++)
6          for (int y = 0; y < result.get_height(); y++)
7              for (int i = 0; i < a.get_width(); i++)
8                  result.at(x, y) += a.at(i, y) * b.at(x, i);
9
10     return result;
11 }
```

Listing 3.2: Naive implementation of matrix multiplication.

row major traversal on B and therefore has a higher chance of cache hits and prefetcher optimizations. An implementation can be found in listing 3.3.

From a pure theoretical standpoint this solution should actually perform worse than the naive approach. In addition to the multiplication ($\mathcal{O}(n^3)$) a matrix needs to be transposed ($\mathcal{O}(n^2)$). The final complexity therefore is $\mathcal{O}(n^3 + n^2)$. Besides that, additional memory for the transposed matrix is required. In reality the cache efficiency prevails. Figure 3.9 shows the results of multiplying matrices of different sizes using both the naive and the improved algorithm.

From the test results it is immediately clear that optimizing for cache efficiency can lead to a faster algorithm, even though the computational complexity is higher. The cache efficiency optimization decreased the required CPU time significantly. Figure 3.10 shows the achieved speedup. The smallest speedup factor is twice as fast with a factor of 2.08 and a matrix of size 200×100. The highest speedup was achieved with a matrix of size 2000×1900. In this case the optimized version was 8 times as fast.

```
1  Matrix multiply_t(const Matrix& a, const Matrix& b)
2  {
3      Matrix result(b.get_width(), a.get_height());
4      Matrix bT = b.get_transposed(); //transpose b
5
6      for (int x = 0; x < result.get_width(); ++x) {
7          for (int y = 0; y < result.get_height(); ++y) {
8
9              int res = 0; // temporary counter on stack
10             for (int i = 0; i < a.get_width(); ++i) {
11                 // row major traversal of both matrices.
12                 res += a.at(i, y) * bT.at(i, x);
13             }
14             result.at(x, y) = res;
15         }
16     }
17     return result;
18 }
```

Listing 3.3: Matrix multiplication with improved cache efficiency by transposing one of the matrices.

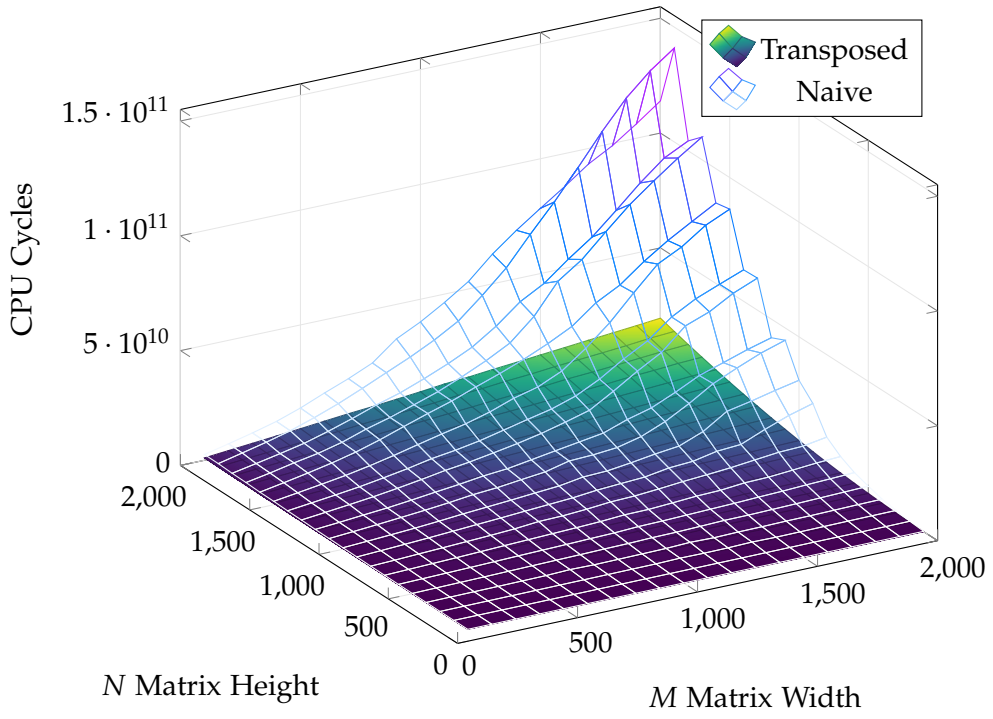


Figure 3.9: Performance results of multiplying different sized matrices. The result is the amount of CPU cycles required to multiply two matrices of size $N \times M$ and $M \times N$.

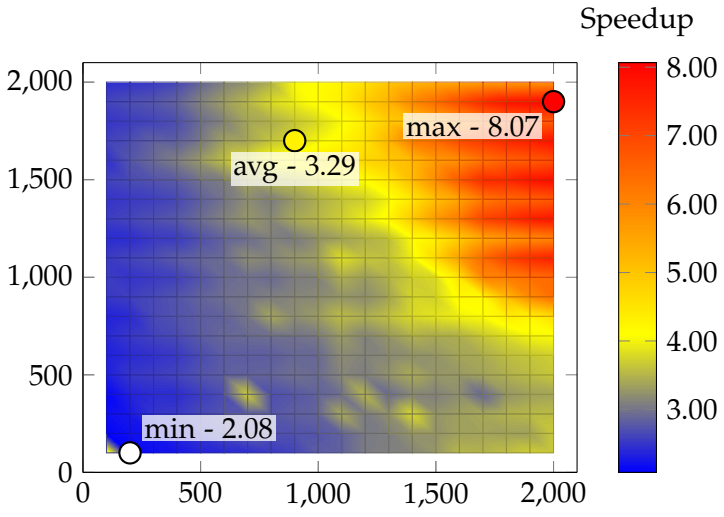


Figure 3.10: Achieved speedup by optimizing cache efficiency. The speedup is measured in $(cycles_{naive})$ / $(cycles_{transposed})$.

4 Specialties in Graphical Real-time Applications

Graphical real-time applications / video games have special reoccurring patterns or restrictions which are not typical for other types of programs. Throughout this chapter some of these specialities are highlighted. Chapter 5 (Goals) discusses how this is translated to goals we need to achieve as a result of these specialties and chapter 6 (Solutions) proposes actual solutions including C++ code samples.

4.1 Performance Requirements

As already discussed in the introduction of this thesis, graphical real-time applications or games have very high performance requirements. The trend towards virtual reality solutions based on mobile hardware strengthens these requirements even further. The recommended target framerate for VR applications is 90 frames per second (FPS) [GD18; OCD18]. When rendering at that framerate every 11.1 milliseconds a new frame has to be drawn. Based on the built-in display technology missing that limit often means that the last frame will be displayed for an additional 11.1ms leading to micro stutter because of inconsistent frame times [Gri+18]. It is therefore critical to ensure that performance is not an afterthought. Instead, performance has to be considered at all times when writing code for such systems.

4.2 Common Patterns

Iterating Large Amounts of Elements

Often it is necessary to iterate over many elements for a number of different reasons: updating game state for all entities, ticking the physics simulation on all physics actors, rendering all objects including complex effects like real-time shadows, reflections and lighting. All of those actions operate on many elements and require a different subset of each elements properties. Some of the mentioned actions even require multiple iterations themselves. This can be the case when *e.g.* multi-pass rendering techniques are used.

Rapid Creation and Destruction of Entities

In many cases it is required to create and destroy a lot of objects in a short amount of time. This pattern can be seen in *e.g.* particle systems where hundreds of particles get

spawned and removed continuously. Spawning projectiles in a shooter or visual effects as a result of gameplay events are additional temporarily created objects. Creation and destruction typically do not reduce the overall performance or frame rate. Instead, when many objects get created during a single frame, this frame misses the frame time target which causes stutter. If the creation of objects is optimized, this stuttering can be prevented.

4.3 Low / Fixed Memory Platforms

Modern computer systems have complex memory management including virtual memory. Virtual memory ensures that a larger memory pool (typically the computers hard drive or solid state drive) can be used as additional memory if the available main memory is exceeded. This is achieved by handling memory in pages of a certain size and map a virtual address space to physical memory addresses using page identifiers and page offsets. While it is not desirable to rely on *swapping* data onto a swap partition because it is magnitudes slower to access than DRAM, it ensures that applications do not crash in the event of running out of main memory. Another advantage of virtual memory is the ability to allocate multiple non-continuous memory regions which can then virtually be mapped to a single continuous memory region. If that is the case, an application can request a continuous memory region of size N even though a continuous region of that size is not physically available [Por17].

Game consoles often do not support virtual memory because accessing virtual memory requires a translation of the requested memory address to the real, physical memory address. This indirection introduces a performance overhead that is not desirable on game consoles which typically use slower hardware. By dropping virtual memory, it is mandatory to stay within the console's memory limits. Requiring a single byte more than the system can offer causes the application to crash. The difficulty with this is that memory should not be wasted. In many applications wasted memory comes in the form of memory fragmentation which can be very problematic on fixed memory platforms because a "virtually continuous memory region" cannot be created.

5 Goals

In this chapter we recall the information given in previous sections. This information is then used to translate the limitations and requirements into objectives to be pursued during the development of graphical real-time applications.

5.1 High Performance

5.1.1 Cache Friendliness

Section 3.1.8 introduced the concept of cache friendliness. The performance measurement from section 3.1.8 showed the impacts and the importance of an optimized memory layout when traversing lists. Because traversing lists is one of the reoccuring patterns in the development of games and other graphical real-time applications it is something we want to optimize.

Predictable Memory Access Patterns and High Cache Line Usage

The predictability of memory access patterns is very important because it can reduce DRAM access latency by allowing the CPU hardware prefetcher to load data before it is requested. An important limitation of that is that it not only requires a linear memory access pattern but these memory accesses have to be as close to each other as possible. This was shown by disabling the hardware prefetcher in section 3.1.8 (Performance measurements). The goal of cache friendliness is therefore not just "predictable memory access patterns" but "predictable memory access patterns with high cache line usage". In many cases this can be difficult to achieve. If only primitive data types like `float` or `int` are used, optimizing cache line usage is easy and automatically achieved when using an array or a `std::vector`. But often the data stored in such a container is more complex. Assumed the `Particle` structure shown in listing 5.1 is stored in a `std::vector`, and this vector is iterated to update all particle's velocities based on the linear force. The updated acceleration is then used to update the particles position. The `Particle` structure follows a pattern many programmers use when writing code. Member variables are grouped by logical connections the programmer made up during the creation of the struct. This method seems logical because it makes reading the struct definition easy. A look at the struct's internal memory layout reveals problems with this approach.

Listing 5.2 shows the result of pahole. Pahole is a linux command line utility that shows "data structure layouts encoded in debugging information formats" [Mel09] and therefore allows us to easily see how the data of structures and classes is laid out in

```
{
    float age = 0;
    float max_age = 0;
    bool pending_kill = false;

    Vector3 position{};
    Vector3 rotation{};
    Vector3 scale{};

    Vector3 linear_velocity{};
    Vector3 angular_velocity{};

    Vector3 linear_force{};
    Vector3 torque{};

    int color = 0;
    int texture_handle = 0;

    ParticleSystem* parent;
    void *user_ptr = nullptr;
    uint64_t flags = 0;
};
```

Listing 5.1: Example struct.

```
/*                                                  offset size */
struct Particle {
    float                       age;                /*     0     4 */
    float                       max_age;            /*     4     4 */
    bool                        pending_kill;       /*     8     1 */

    /* XXX 3 bytes hole, try to pack */

    struct Vector3              position;           /*    12    12 */
    struct Vector3              rotation;           /*    24    12 */
    struct Vector3              scale;              /*    36    12 */
    struct Vector3              linear_velocity;    /*    48    12 */
    struct Vector3              angular_velocity;   /*    60    12 */
    /* --- cacheline 1 boundary (64 bytes) was 8 bytes ago --- */
    struct Vector3              linear_force;       /*    72    12 */
    struct Vector3              torque;             /*    84    12 */
    int                         color;              /*    96     4 */
    int                         texture_handle;     /*   100     4 */
    class ParticleSystem *      parent;             /*   104     8 */
    void *                      user_ptr;           /*   112     8 */
    uint64_t                    flags;              /*   120     8 */
    void Particle(class Particle *);


    /* size: 128, cachelines: 2, members: 15 */
    /* sum members: 125, holes: 1, sum holes: 3 */
};
```

Listing 5.2: Output of pahole running on the example struct from listing 5.1.

memory. The result shows the problem: When we want to update a particle's velocity based on a particle's linear force we have to access data from two different cache lines. Another even bigger problem is `angular_velocity`. The cache line boundary information right after the `angular_velocity` member informs about the cache line boundary being 8 bytes before the message. This means that the cache line boundary is inside `angular_velocity`. To be precise the boundary is exactly between `angular_velocity.x` and `angular_velocity.y`. When only the angular velocity is needed for an algorithm, two cache lines are loaded of which only 12 bytes are used. The remaining 90.6% are wasted. Even though the struct gets aligned to the cache line size of 64 bytes to never occupy more than two cache lines, the cache line usage is suboptimal. The method of using arrays of structures (AoS) is what is problematic in this case. This is especially true if only a few data members need to be accessed at a time. A different approach is shown in section 6.2.1 (AoS / SoA).

Avoiding Virtual Function Calls

In C++ virtual functions are a way of propagating function calls from a base class to an instance of a derived class. Virtual functions allow the creation of flexible algorithms because knowing the real type of an object is not mandatory in order to call specialized functions on it. Driesen and Hölzle explain why virtual function calls can introduce performance problems: To allow calling a function that overrides a base type's function, the system has to know where to find the derived function in memory (dynamic dispatch). For that reason every class or struct that has virtual functions, has what is called a vtable. The vtable (short for virtual function table) keeps track of the memory addresses for virtual functions [DH95]. If a list of objects is iterated where each object might be of a different derived type that overrides the function that is called for each object, problems with cache usage arise. The fact that the vtable has to be loaded is not a big problem, after a few iterations every required type's vtable will be cached. Problematic is that the called function's instructions are scattered in memory without the program knowing which function will be called next. Therefore, the instructions for any given function cannot be prefetched [DH95]. Since this is an instruction cache phenomenon, it is not considered part of this thesis. It is nevertheless mentioned because the causes are based on the underlying memory model.

5.1.2 Memory Fragmentation

Multiple consecutive allocations and deallocations can lead to *memory fragmentation*. A heap of memory is called fragmented when chunks of allocated memory are divided by small unallocated blocks. Figure 5.1 illustrates this problem. Even though the final heap shown in fig. 5.1 has 16 byte of free memory available, requesting 16 byte would fail since allocations always have to be continuous. Allocating 8 byte twice would be possible, but storing a 16 byte datatype would not. This example is simplified just to show what memory fragmentation is and how it leads to wasted memory. The

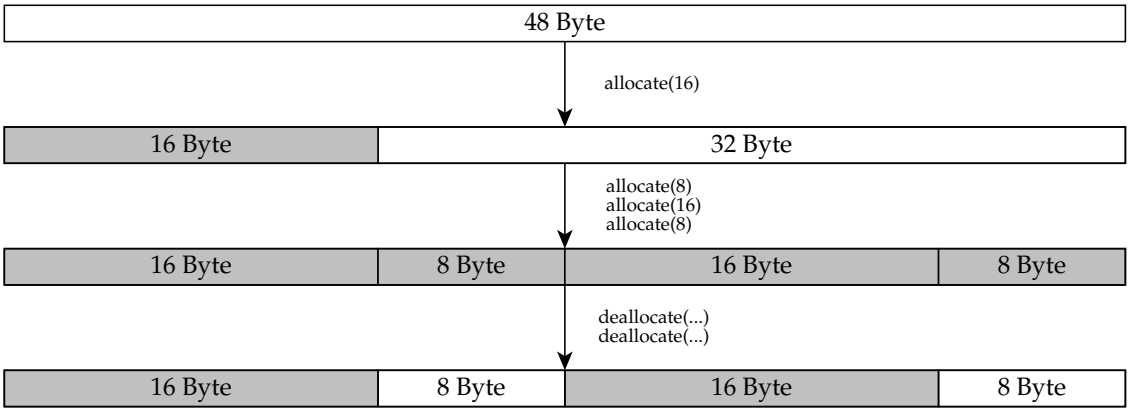


Figure 5.1: Consecutive allocations and deallocations can lead to memory fragmentation.

same problem occurs on large heaps of multiple gigabytes. On systems with limited memory (especially if the system is a fixed memory platform) the order of allocations and deallocations can decide whether an application can run or if it crashes with an out of memory exception. Rapid creation and destruction of objects in combination with fixed memory platforms leads directly into the importance of little to no memory fragmentation. Memory fragmentation can be solved by custom allocators that enforce certain criteria on allocation and deallocation patterns.

5.1.3 Malloc Call Reduction

In section 2.2.1 it was shown that calling malloc repeatedly can cause performance problems. These performance problems can only be solved by not calling malloc in the first place. The solution to this is often to allocate memory ahead of time and work with a buffer instead. This buffer is then managed by a custom allocator. Section 6.1 shows different allocators that can solve both memory fragmentation and malloc call reduction at the same time.

5.2 Ease of Use

Programmers have to keep many things in mind while developing applications. Keeping things simple should be targeted at all times. Complex code with hard to use APIs can lead to problems and memory management is no exception to that. Most facilities that try to counter the complexity of memory management target the avoidance of memory leaks. In section 6.3 these facilities are explained in detail.

5.3 Robustness to Code Changes

In 1974 Donald Knuth formalized the famous sentence "premature optimization is the root of all evil" [Knu74]. Programmers should not spend enormous amounts of time to optimize code that in most cases is not a performance critical section. In reality this means that code is optimized later in development once it is proven to be a performance bottleneck. As a result changes to the code have to be made. We define robustness to code changes as follows:

> Source code is robust to code changes if altering parts of the source code does not require a lot of additional changes, even if the initial change influences large portions of a code base.

In practice this means that a data structure or algorithm used for any given part of an application can easily be swapped with an alternative solution *without* changing access syntax or API calls everywhere in the code. If code is not robust to changes, programmers tend to not try different solutions and instead stick to the first working one which is often suboptimal when following Knuth's advice.

6 Solutions

In this chapter solutions to the mentioned problems are shown and the implementations that are part of this thesis are explained.

General Information

The sample implementations do not use the C++ standard STL (standard template library) because of its allocator model. Instead, the implementations are based on the EASTL (**E**lectronic **A**rts **S**tandard **T**emplate **L**ibrary) [Ped07]. The EASTL tries to solve some of the problems of the standard template library. Details why we chose EASTL instead of the official C++ standard template library can be found in section 8.1.

Another important distinction to what is often considered the correct usage of C++: In the provided implementations, allocation and construction of objects are not the same thing and therefore separated. Resource acquisition is initialization (RAII) is often considered the best way of handling memory and object lifetime. In game development RAII is in many cases not applied when dealing with memory [Ped07]. Therefore, the provided allocator implementations do not call constructors or destructors. They only handle raw memory allocations and deallocations. It is up to the user code to ensure correct handling of calling construction and destruction facilities manually or to provide a wrapper that handles these operations automatically. For the manual creation C++ provides placement new. Placement new allows to call `new` for a type on already allocated memory by passing a `void*` to `new`. A short code example is given in listing 6.1.

```
1 void* memory = my_allocator.allocate(sizeof(T));
2 T* my_T = new(memory) T();
3
4 //or combined
5 T* my_T2 = new(my_allocator.allocate(sizeof(T))) T();
```

Listing 6.1: Construction of an object on preallocated memory is done by using placement new.

6.1 Custom Allocators

Custom allocators can solve some of the difficulties in memory management, especially memory fragmentation and frequent malloc calls. In this section different custom allocators are introduced, each with pros and cons listed. All allocators have one thing in common: They all accept an allocator that is used as the *upstream allocator*. That means that the allocator itself uses another allocator to get the memory it requires. This allows chaining of different allocators and gives programmers the ability to build hierarchies of allocators. The root of every allocator chain as well as every allocator's default upstream allocator, is an allocator that forwards calls to the operating systems default allocation facilities. In the provided implementations this allocator is called `Malloc_Allocator`. The allocators implement their internal state in a way that allows sharing between multiple allocator instances. Because of that, different parts of an application can share a single allocator.

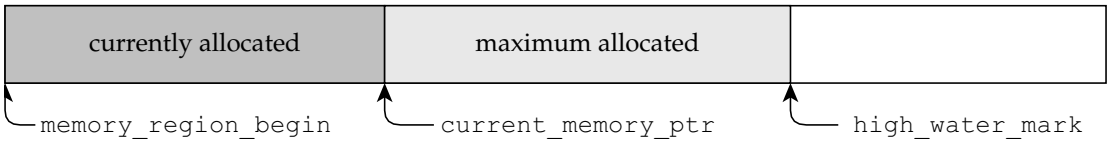


Figure 6.1: Illustration of a linear allocator.

6.1.1 Linear Allocator

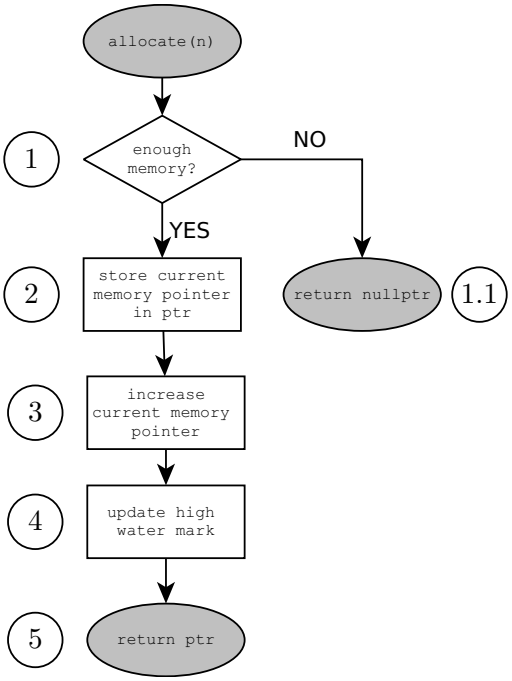
The linear allocator is the easiest allocator possible. On creation a large continuous memory region is allocated from the upstream allocator. Its members are:

- `size_t memory_region_begin`: A pointer that points to the first byte of the memory region that was allocated using the upstream allocator.
- `size_t capacity`: The full capacity in byte. `capacity` byte get allocated from the upstream allocator on creation.
- `size_t current_memory_ptr`: Initialized to match `memory_region_begin` on creation. Every allocation of N byte increases the `current_memory_ptr` by N.
- `size_t high_water_mark`: Used to keep track of the amount of memory allocated from the linear allocator during its entire lifetime.

Allocation

If an allocation of N bytes is performed using a linear allocator, the following happens:

1. Check if the new allocation exceeds the capacity.
   1.1. If it does, return `nullptr`.
2. Store `current_memory_ptr` in local variable `ptr`.
3. Increase `current_memory_ptr` by N.
4. Update `high_water_mark` if required.
5. Return `ptr`.



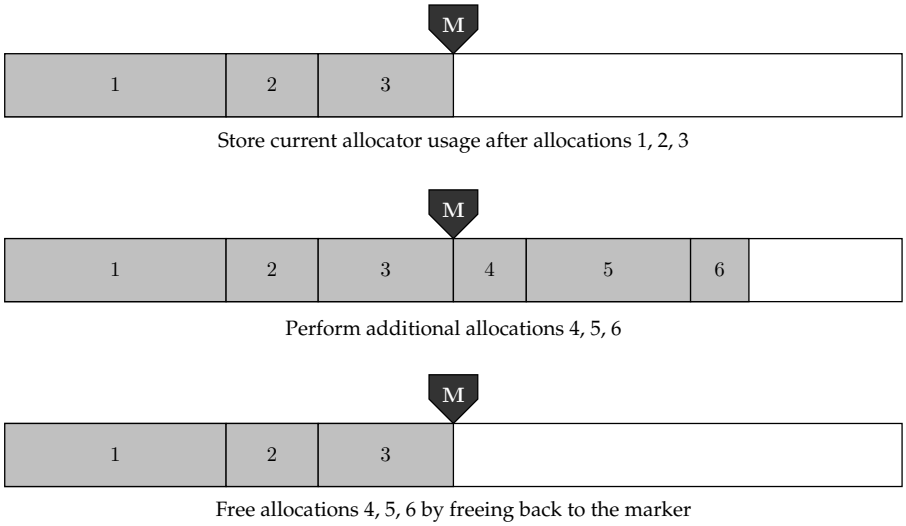


Figure 6.2: Visualization of the linear allocators marker functionality.

Deallocation

Since an allocation only increases a pointer value, linear allocators do not support individual deallocations. To free memory, the allocator provides a function `reset()` that resets all allocations by setting `current_memory_ptr` to `memory_region_begin`.

Special Functionality

To counter the lack of individual deallocations, a special struct named `Linear_Allocator_Marker` exists. At any time a `Linear_Allocator_Marker` can be requested from the linear allocator by calling `get_marker()`. This marker is used to store the current state of the allocator. Later in the code this marker can be restored by calling `free_to_marker(const Linear_Allocator_Marker& marker)` with the previously retrieved marker as parameter. Another helper struct exists which automatically calls `free_to_marker` in its destructor. It can be used with a macro `LINEAR_ALLOCATOR_LOCAL_SCOPE(free_allocator)`. Calling this macro immediatly creates an instance of the helper struct, retrieves the current marker and reapplies it in its destructor. This makes it easy to not waste memory if the allocations are only used within a scope.

Limitations

The limitations of this kind of allocator are clear: Deallocations are not possible which renders a linear allocator inapplicable in many cases.

Advantages

While this allocator is very limited, it has huge advantages. Since allocations are extremely simple and only require a single condition check, they are very fast.

The fact that the allocator does not support individual deallocations enforces simple allocation and deallocation patterns. Deallocations are only possible when everything that has been allocated after a marker gets deallocated as well. This ensures that there are no holes in between allocated memory blocks; Memory cannot be fragmented.

Performance

Allocations of arbitrary size are extremely fast. Since all allocation requests are essentially the same, the allocation size does not matter. Allocating hundreds of megabytes at once is exactly as fast as allocating just a few bytes. The results of a performance comparison between the linear allocator and malloc can be seen in table 6.1.

# Allocations	Size per alloc	Cycles Linear Allocator L	Cycles Malloc M	M/L
1000	8 Byte	12 629	73 824	5.85
1000	256 Byte	9395	189 199	20.14
1000	8 KB	9374	3 354 756	357.88
1000	1 MB	9966	4 927 997	494.48

Table 6.1: Performance results of different allocation patterns. The linear allocator outperforms malloc in all cases.

Possible Usage Scenarios

Often it is required to fill a container like an array or vector to perform a simple calculation on the elements. The data structure is destroyed immediately after the calculation. For use cases like this a linear allocator provides a simple facility to reduce allocations while maintaining the simplicity of a dynamically sized container. Since individual deallocations are not needed, the full memory region can be cleared in constant time.

Another use case can be found in `example_temporary_allocator.cpp`. This example uses a linear allocator as global temporary storage. Details on how this temporary usage can be used and the advantages it provides can be found in the comments of the implementation.

Figure 6.3: A double ended linear allocator shares a single memory region between two linear allocators.

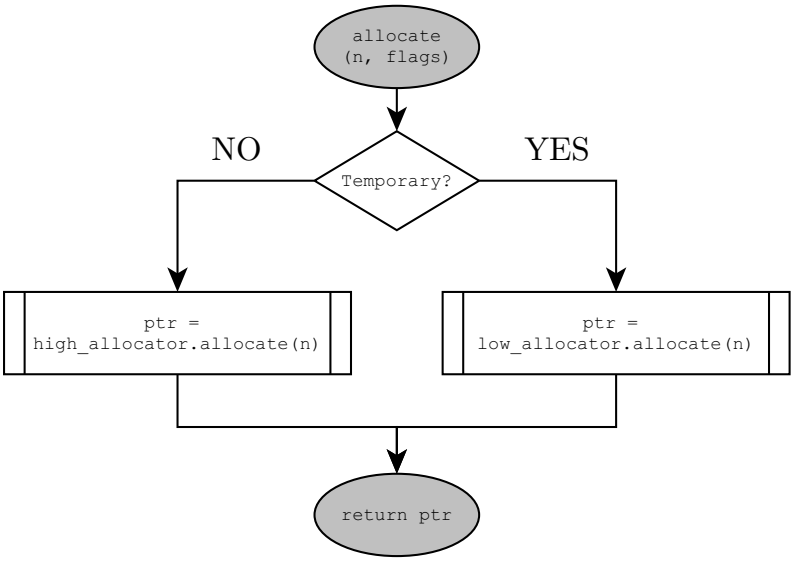


Figure 6.4: The double ended linear allocator forwards allocation requests to the respective linear allocator.

6.1.2 Double Ended Linear Allocator

The double ended linear allocator is basically the same as a plain linear allocator. The only difference is that in a double ended linear allocator two different linear allocators operate on the same memory region. One grows from the bottom upwards, the other grows from the top downwards. This allows the distinction between permanent and temporary allocations.

Allocation

EASTL has a parameter `flags` on all allocation functions. Part of these flags is the distinction between temporary allocations (`eastl::alloc_flags::MEM_TEMP`) and permanent allocations (`eastl::alloc_flags::MEM_PERM`). In the provided implementation, temporary allocations are redirected towards the allocator that operates on the low memory region, while permanent allocations are redirected to the allocator that manages the high memory regions.

Deallocation

Just like the simple linear allocator, the double ended linear allocator does not support individual deallocations.

Special Functionality

A reference to each of the underlying linear allocators can be obtained. This allows the creation of allocation markers and enables usage of the `LINEAR_ALLOCATOR_LOCAL_SCOPE` macro. Since the plain linear allocator internally manages the creation and setup of a double ended linear allocator, allocations do not have to happen on the double ended linear allocator. If `allocate(...)` is called on one of the suballocators directly, the second linear allocator is aware of that. Therefore, the suballocators can be passed to functions that expect a plain linear allocator without risking the loss of integrity.

Limitations

The limitations are the same as the limitations of a simple linear allocator.

Advantages

The double ended linear allocator inherits the advantages of a simple linear allocator, in that allocations are very fast. In addition to that, the distinction between temporary and permanent storage can be used to reduce memory fragmentation on a larger scale. It allows to keep data longer in a tightly packed linear allocator because rapid allocation / deallocation sequences can be performed on the temporary suballocator. Compared to two plain linear allocators, a double ended linear allocator has a flexible memory limit for both allocators.

Possible Usage Scenarios

A double ended linear allocator can be used as an allocator that is very low in the allocator hierarchy. The distinction between permanent and temporary memory allows to pack allocations better, especially when it is available very early in the lifecycle of an application. During application startup loading operations like decompressing textures or loading and parsing configuration files are performed. These operations require temporary memory and result in persistent data. The double ended linear allocator can pack this persistent data perfectly while leaving the rest of the memory unfragmented. If a more dynamic allocator is required once the application startup is finished, the double ended linear allocator can serve as an upstream allocator after the initial startup sequence.

6.1.3 Double Buffered Linear Allocator

The double buffered linear allocator (DBLA) is an allocator that wraps around two linear allocator instances. Contrary to the double ended linear allocator the linear allocators managed by the double buffered linear allocator do not operate on the same memory region. Instead, each of the linear allocators allocates its own memory region from the DBLA's upstream allocator. One allocator is active at a time with the possibility to retrieve direct access to the inactive one. The active and the inactive allocators can be swapped by calling `swap()`.

Allocation

Whenever an allocation is requested, the double buffered allocator selects the currently active linear allocator and forwards the allocation request to it.

Deallocation

Since this allocator is based on linear allocators, individual deallocations are not supported.

Special Functionality

In the provided implementation `operator linear_allocator&()` is implemented. This operator returns the currently active linear allocator and allows implicit usage of the DBLA in a linear allocator context. Besides that the DBLA has the member function `swap()`. `swap()` switches the active and the inactive allocator. An optional `bool` parameter that defaults to `true` indicates if the newly activated linear allocator should be reset. Since direct access to the linear allocators is possible, the usage of `Linear_Allocator_Markers` is supported.

Limitations

The double buffered linear allocator inherits the limitation of the linear allocator.

Advantages

The double buffered linear allocator inherits all the advantages of the linear allocator. Additionally it is possible to preserve memory without losing the ability of handling allocation requests.

Possible Usage Scenarios

A possible use case for a double buffered linear allocator is the creation of a render command queue. The game thread can fill a queue by allocating from the double buffered linear allocator. At the end of the frame the two allocators are swapped. Now

the render thread can work on the queued render commands while the game thread can continue to queue up new render commands for the next frame.

6.1.4 Free List Allocator

Contrary to all the other allocators shown so far the free list allocator can handle allocations and deallocations in arbitrary order. The free list allocator therefore is a general purpose allocator. To allow allocations and deallocations in random order the free list allocator allocates additional memory for all allocation requests to store a *header* that keeps track of allocation details. Free blocks of memory have a header that store how large the free block is as well as a pointer to the next free block. This pointer chains free memory regions to form a linked list of free memory blocks called the *free list*. Listing 6.2 shows these headers.

Allocation

For free list allocators two different allocation strategies are possible. The *First Fit* strategy traverses the free list until a free memory block that is large enough to handle the allocation request is encountered. The *Best Fit* strategy always traverses the full list and selects the smallest memory block possible. While first fit is faster in most cases, best fit can reduce memory fragmentation. Small allocation requests fill up small free blocks first, leaving large free memory regions available for future (possibly larger) requests. The strategies are variations of the same allocation algorithm:

First the block that is used for the allocation is selected depending on the strategy. While iterating the free list during block selection both the current free header and the previous free header are stored for later use. The required allocation size can be different for different free blocks. This is a result of a potentially requested alignment. Depending on the address of a free memory region more or less padding needs to be inserted before the header of the allocation. If no block is large enough for the allocation `nullptr` is returned. Otherwise it is checked if the selected block needs to be split. When the selected block is too small for another allocation after splitting, the current allocation gets changed to fill the entire memory block. If the selected block is large enough, a new free block gets inserted behind the memory needed for the allocation. At this point the free list nodes get updated by relinking changed nodes. Finally, an `allocated_header` gets inserted and the pointer to right after this header gets returned.

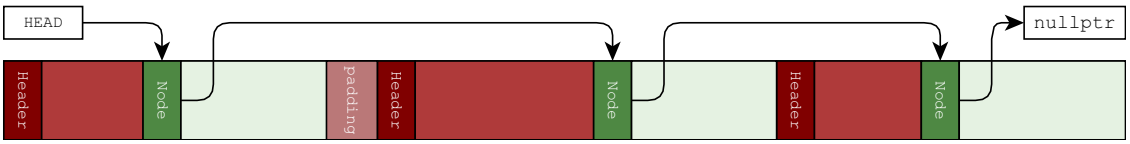


Figure 6.5: A free list allocator keeps a linked list of free memory blocks. The list is stored in the memory region managed by the allocator and therefore does not require additional memory.

```
1  struct allocated_header
2  {
3      size_t allocated_size_excluding_header = 0;
4      size_t padding = 0;
5  };
6
7  struct free_header
8  {
9      size_t free_memory_including_header = 0;
10     free_header* next_free = nullptr;
11 };
```

Listing 6.2: The headers used by the free list allocator. Constructors omitted for brevity.

Deallocation

On deallocation, the free list is iterated until the memory address of the current free header is larger than the value of the pointer to the memory that gets deallocated (`size_t address`). Current and previous free headers are stored for later use. The address of the header that precedes the memory that gets deallocated is calculated using:

```
size_t header_address = address - sizeof(allocated_header)
```

Now, the `allocated_header` can be accessed to get the padding information. `header_address - padding` is the address of the begin of the to be freed memory region and is the place where a new free_header has to be inserted. In the end adjacent free memory blocks are merged into a single big free block. This process is called coalescing.

Special Functionality

The free list allocator does not have any special functionality.

Limitations

As a general purpose allocator the free list allocator does not have significant limitations in functionality. A problem of the free list allocator is the usage of a linked list. Section 5.1.1 (Cache Friendliness) explained the importance of predictable memory access patterns. Traversing a linked list is not predictable and therefore has a high cache miss rate. Because of that a free list allocator can be slower than malloc.

Advantages

The huge advantage of a free list allocator is that it can be used as a general purpose allocator. By not introducing any limitations this allocator is versatile and usable in many cases.

Performance

While testing the performance of the free list allocator it became clear that the results are not as clear as the results of the linear allocator. The required linked list traversals alter the allocators performance based on allocation and deallocation patterns. To test the influence of the linked list the following scenarios were tested:

- Allocation only
- Allocation with ordered deallocation (First In First Out)
- Allocation with randomized deallocation order

All of these tests were repeated for 1000, 2000, 3000 and 4000 allocations. The results of the test with 2000 allocations are shown in fig. 6.6. The full test has shown a decreasing performance of the free list allocator when more allocations are handled. This is expected behavior because the internal linked list of allocated blocks grows proportional to the number of (not yet freed) allocations.

Possible Usage Scenarios

A general purpose allocator can naturally be used for everything. In practice the free list allocator is best used in cases where differently sized allocations are required but the total amount of allocations stays small. Additionally, allocations and deallocations should not be made in performance critical situations because the required time cannot be predicted accurately. These limitations make the free list allocator a convenient upstream allocator that is used as a base for specialized allocators.

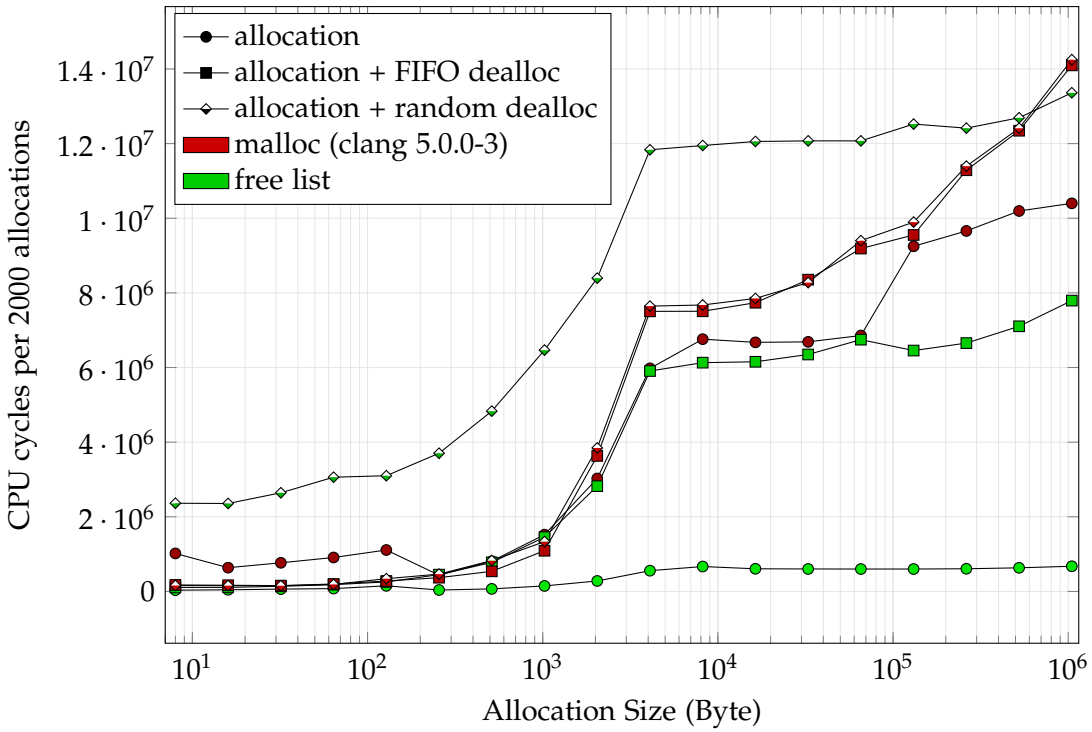


Figure 6.6: Performance comparison between malloc and the free list allocator. Different operations were tested to isolate potential worst case scenarios.

6.1.5 Pool Allocator

Whenever large amounts of similar sized objects are rapidly constructed and destructed a pool allocator can speed up that process. Similar to the free list allocator a pool allocator holds a linked list that internally connects free blocks of memory. What makes the pool allocator more performant than the free list allocator is the restriction to a single possible allocation size that cannot change once the allocator is initialized.

Internally the pool allocator only stores a single pointer to the first element of the linked list of free memory blocks (`HEAD`)

Allocation

Allocation requests are an easy operation. If `HEAD` is `nullptr` an allocation is not possible and `nullptr` is returned. If `HEAD` is non-null it is stored in a temporary variable `ptr`. `HEAD` is then replaced by the linked list node that follows `HEAD` (this can be `nullptr`). Finally `ptr` is returned.

Deallocation

Since the order of nodes does not matter a new node can be inserted anywhere. To make deallocations fast the node is simply added before the `HEAD` node. This is achieved by first creating a new node at the memory location that is passed to the deallocate function. Its next pointer is initialized to `HEAD`. Finally `HEAD` is replaced with the newly created node.

Special Functionality

The pool allocator does not have any special functionality.

Limitations

The limitation of a pool allocator is obvious. Every allocation request for the entire lifetime of a pool allocator can only be of a single size. If a smaller allocation is requested, memory is wasted. If a bigger allocation is requested, the allocation cannot be handled.

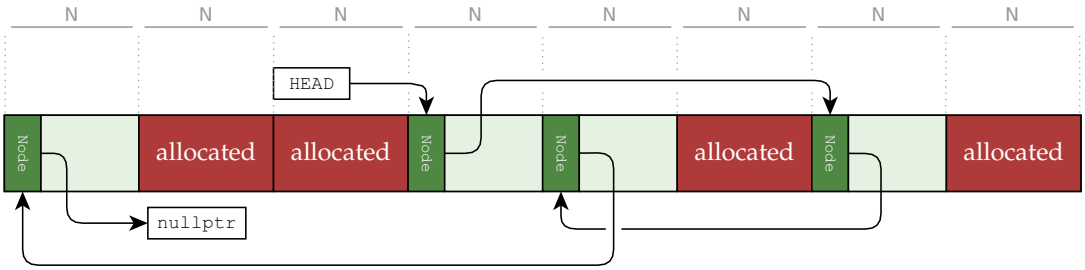


Figure 6.7: Pool allocator visualization.

Allocator	Cycles
Pool	187642
Malloc	942902

Table 6.2: Performance comparison between the pool allcoator and malloc.

Another limitation is that a pool allocator only manages raw memory. For objects it is required to construct these objects in the allocated memory using placement `new`. If objects have expensive construction or destruction requirements the fast allocation and deallocation of the required memory is insignificant. These requirements include *e.g.* (un-)registering objects at rendering, physics or entity management subsystems. To avoid the repeated execution of expensive construction and destruction code, an object pool can be used. Refer to section 6.2.2 for details.

Advantages

The pool allocator has multiple advantages. Firstly the allocator never suffers from memory fragmentation. By definition every unallocated block of memory will always be large enough to fit another allocation. Another advantage is performance: Since the allocator introduces limitations on allocation and deallocation patterns, it can take shortcuts in the allocation and deallocation algorithms. Therefore, a pool allocator has an extremely low overhead and is very fast.

Performance

To test the performance of the pool allocator, the following allocation / deallocation pattern was executed with an allocation size of 256 Byte:

1. 2000 allocations
2. 1000 deallocations (randomly selected)
3. 1000 allocations
4. 2000 deallocations (in random order)

The benchmark shows that the pool allocator is about 5 times as fast as malloc. The exact results are listed in table 6.2.

Possible Usage Scenarios

Chapter 4 (Specialties in Graphical Real-time Applications) included the rapid creation and destruction of similar objects in the common patterns of graphical real-time applications. Individual particles of particle systems or projectiles in shooter games can benefit from the performance advantages. The limitations of the pool allocator are not that severe in these cases, since the objects are typically of the exact same size.

6.2 Data Structures

The custom allocators introduced throughout this chapter so far can only solve fragmentation and `malloc` call problems. Section 5.1.1 (Cache Friendliness) introduced the importance of cache friendly code in the form of *locality*. The example that was part of the explanation of cache friendliness had the problem that a single object occupied multiple cache lines. If only a small amount of the data members of such an object are required to perform a given task cache usage is suboptimal. This issue can be solved by restructuring the data from the ground up.

6.2.1 AoS / SoA

The differentiation between AoS and SoA is relevant if multiple elements of the same type are stored together in a container that manages contiguous memory. Such containers are simple arrays or (in the case of C++) typically `std::vector`. Intel defines AoS and SoA in its architecture optimization reference manual [18a].

Array of structures (AoS) is a layout where instances of structs or classes are stored directly in the container. **Structure of arrays** (SoA) inverses this concept. Instead of storing instances of a struct in a container, the struct itself contains a container for every member. Listing 6.3 and listing 6.4 clarify this concept using a short code example. A visualization of the memory layout is shown in fig. 6.8.

AoS is the *object oriented* way of handling multiple elements of the same type while SoA is the *data oriented* option. Most programmers choose AoS for its simplicity and intuitive, logical encapsulation that is a direct result of the object oriented design philosophy. As shown in section 5.1.1 (Cache Friendliness) this can be problematic because of potentially inefficient cache line usage. The benchmarks in section 3.1.8 show that this problem gets worse when the structs are bigger than just a few data members. SoA ensures efficient cache line usage when iterating over a small amount of data members. In theory the iteration should be faster since close to all member accesses can be done with level 1 to level 2 cache speed.

Performance Measurement

To validate this suspicion a small test program is used. This program implements the Particle struct from listing 5.1. For the test, 5000 instances are kept in both SoA and AoS containers. Then each particles `linear_velocity` is updated based on its `linear_force`. The updated `linear_velocity` is then used to update its `position`. Performance numbers can be found in fig. 6.9.

SoA Container Implementation

For easy usage as well as the ability to switch between SoA and AoS a container that hides the underlying memory layout with a zero overhead abstraction is required. To be as robust to code changes as possible, the access syntax to the elements of

```
struct some_struct
{
    float a;
    int   b;
    bool  c;
}

// later in code
std::vector<some_struct> aos;

// increment each structs 'b'
for (int i = 0; i < aos.size; ++i)
{
    s[i].b++;
}
```

Listing 6.3: Simple array of structures (AoS) example.

```
struct some_struct
{
    std::vector<float> a;
    std::vector<int> b;
    std::vector<bool> c;
}

// later in code
some_struct soa;

// increment each structs 'b'
for (int i = 0; i < soa.b.size(); ++i)
{
    soa.b[i]++;
}
```

Listing 6.4: The same example as structure of arrays (SoA).

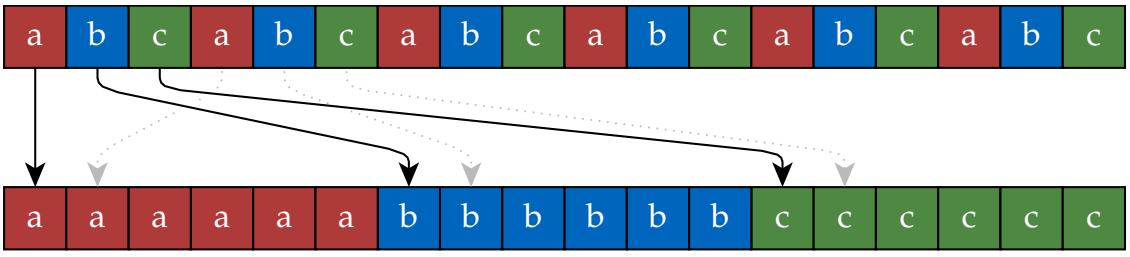


Figure 6.8: The different memory layout for SoA and AoS. Top is the AoS layout from listing 6.3. Bottom is the SoA layout from listing 6.4. The arrows indicate which memory locations are semantically equal. Note that the containers for a, b and c in the SoA layout are not necessarily adjacent.

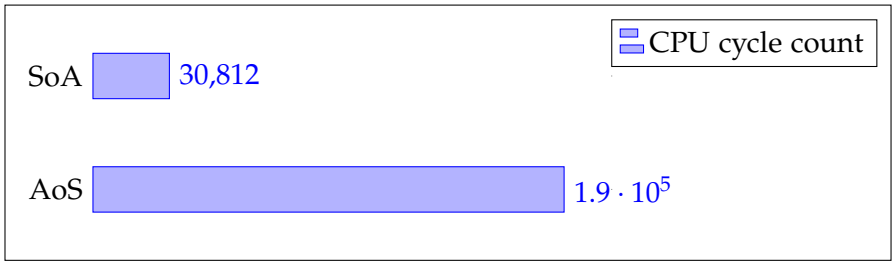


Figure 6.9: Performance of SoA and AoS memory layouts when only part of a structure is required during an iteration.

this container should not be different for the different memory layouts. At best, the interface to the data oriented container mimics object oriented practices. The library LibFlatArray provides such an implementation [And18]. Since the complexity of such an implementation is enormous, a perfect zero-overhead implementation is not included in the code samples of this thesis. Instead, the reader is encouraged to study the implementation of LibFlatArray.

6.2.2 Object Pools

As shown in section 6.1.5 (Pool Allocator) a pool allocator can speed up allocation and deallocation of equally sized memory blocks. If this memory is then used for objects with expensive construction and destruction requirements the performant resource allocation is insignificant. This can be solved by constructing an object once and reusing it afterwards without deconstruction in between uses. Instead of a de- and a following reconstruction, the object is notified when it enters or leaves the pool. It is expected to enter a *disabled* state while waiting in the pool for future use. Objects are considered to be *free* during that time. The implementation provided as part of this thesis targets highly efficient object pooling with as little overhead as possible. To achieve that without sacrificing ease of use the pool provides two different ways of notifying objects when they enter or exit the pool. Which notifiers are used can be specified by `Object_Notifier_Rules`. By default all notifiers are enabled. Objects that are acquired from a pool are wrapped in a `unique_ptr` with a deleter, that returns objects automatically when they are not used anymore.

The first of the two available notifier options is passing function objects for entering and exiting the pool. These functions have the signature `void (T*)` where `T` is the type of the Objects managed by the pool. By modifying the objects that are passed to these functions on entering and exiting the pool, user code can perform the required changes to the objects to transition to and from the disabled state. Since the functions are stored as `eastl::function<void(T*)>` they can be set using lambdas. The type that is managed by the object pool therefore does not have to be modified or prepared in any way.

The second available notifier option requires some preparation of the type

that is managed by the pool. The functions `void on_exit_free_list()` and `void on_enter_free_list()` need to be implemented by the type. Note that these functions are not virtual. Instead, the object pool uses a C++ feature called *Substitution Failure Is Not an Error* (in short *SFINAE*). SFINAE allows *i.a.* the compile-time detection of member functions. Using SFINAE, the object pool can check whether the managed type implements these functions and depending on the result conditionally include direct (*i.e.* statically dispatched) calls to these functions. Conditional compilation is enabled by `constexpr if`. The SFINAE implementation provides a zero overhead callback.

6.3 Ease of Use

6.3.1 Smart Pointers

When dealing with memory three important things have to be considered at all times:

1. If memory is allocated it has to be returned to the allocator.
2. If the pointer to the memory is lost the allocated memory can never be returned.
3. If two copies of the same pointer exist in different locations A and B and the memory is returned (deallocated) in A, the pointer in B:
   a) must not be dereferenced anymore.
   b) must never be returned (deallocated).

With Heap allocated memory it can be difficult to ensure these rules. The question that unifies all these difficulties is about ownership: Who *owns* a resource (in this case memory) at every given point in time? How can ownership be transferred? How can other parts of a program be informed that a certain resource is not available anymore, and how can these other parts prevent returning a resource if they still require it?

As a solution to these problems the C++ standard provides so called smart pointers. This section explains the different types of smart pointers and how they assist in answering the ownership question. This is only a brief overview. Smart pointers provide additional functionality that assits in creating and managing smart pointers. This additional functionality is not part of this thesis.

Unique Pointer

The C++ standard defines a unique pointer as:

> A unique pointer is an object that owns another object and manages that other object through a pointer. More precisely, a unique pointer is an object u that stores a pointer to a second object p and will dispose of p when u is itself destroyed [...]. In this context, u is said to *own* p. [ISO12]

This means that by using a unique pointer (`std::unique_ptr`) the lifetime of one resource can be bound to that of another resource. The standard further defines that "*u* can, upon request, transfer ownership to another unique pointer *u*2." What makes a `unique_ptr` interesting for ownership management, are the restrictions that apply to a `unique_ptr`:

> Each object of a type `U` [...] has the strict ownership semantics, specified above, of a unique pointer. In partial satisfaction of these semantics, each such `U` is `MoveConstructible` and `MoveAssignable`, but is not `CopyConstructible` nor `CopyAssignable`. [ISO12]

As a result a `unique_ptr` cannot be copied, it can only tranfer ownership using C++ move semantics. This prevents that two pointers to the same resource exist multiple times. Ownership is clearly defined. If a `unique_ptr` *u* is destroyed, the pointer that is owned by *u* is deleted using the *deleter* associated with *u*. This deleter can be changed which allows custom behavior when a `unique_ptr` is destroyed.

Constructing a `unique_ptr` is done by using the function `make_unique<T>(...)`. This function forwards all parameters to the constructor of `T` and ensures that the object is guaranteed to be lifecycle managed.

Obviously a `unique_ptr` is not always applicable. In many cases pointers to a resource are required in multiple locations. This shared ownership is managed by a shared pointer.

Shared Pointer

> `shared_ptr`implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. [ISO12]

A `shared_ptr` is a reference counted pointer. This is achieved by allocating additional memory for a *control block*. The control block of a `shared_ptr` stores the number of shared_pointers that refer to the same object. Whenever a `shared_ptr` is copied (either by copy construction or copy assignment) a reference count is incremented. If a `shared_ptr` is destroyed the reference count is decremented. All operations on the reference count are atomic [ISO12]. When a `shared_ptr` is destroyed and detects that the reference count will be 0 after decrementing it, the resource is deleted using the deleter.

The `shared_ptr` allows shared ownership of resources with clear rules on how the resource gets freed. At the same time `shared_ptr` forces the underlying resource to persist until the last instance of a `shared_ptr` gets destroyed. A `shared_ptr` is an *active* participant in a resources lifecycle. It is created using the function `make_shared<T>(...)`. EASTL allows to specify a special allocator that is used for the control block. A problem with `shared_ptr` is, that simple lifecycle observation is not possible. To observe a resources lifecycle another type of shared pointer can be used: The weak pointer.

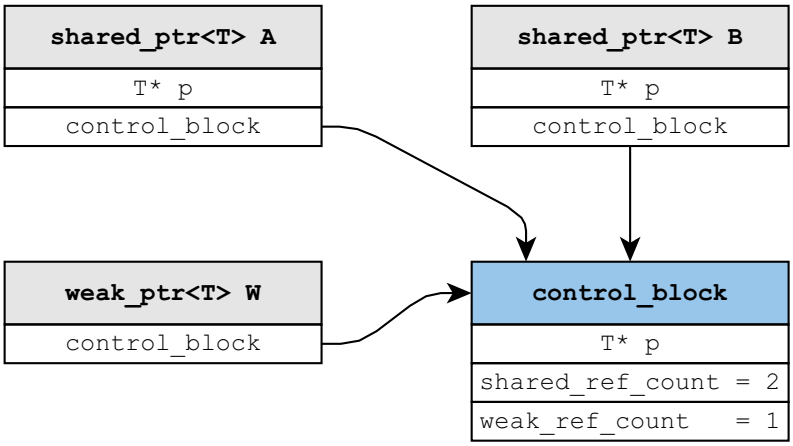


Figure 6.10: Diagram of the structure of shared and weak pointers. This is a simplified visualization. Implementation details differ because the C++ standard only defines behavior and API.

Weak Pointer

> The `weak_ptr` class template stores a weak reference to an object that is already managed by a `shared_ptr`. To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`. [ISO12]

A `weak_ptr` passively participates in a resources lifecycle that is managed using shared pointers. The `weak_ptr` can access the same control block as shared pointers without incrementing the shared pointer reference count. This means that shared pointers can operate normally, with weak pointers *observing* the resource lifecycle. A `weak_ptr` therefore can check if a resource is save to access before attempting an access operation. If the shared pointer reference count in the control block is 0 the resource is considered expired and access is unsafe. If the shared pointer reference count is non-zero accessing the resource is considered safe. To ensure that the resource is not deleted while accessing it through a `weak_ptr`, a `shared_ptr` has to be obtained beforehand using `weak_ptr.lock()`. This `shared_ptr` then ensures that access is safe until it is destroyed.

The control block stores an additional reference counter that counts weak pointer references. This is required since the memory that was allocated for the control block itself needs to be able to outlive every `shared_ptr` that points to it. Weak pointers might still access it after the last `shared_ptr` was destroyed. The last smart pointer (`shared_ptr` or `weak_ptr`) that gets destroyed is responsible for deleting the control block itself. Figure 6.10 visualizes the logical connections between smart pointers and the reference block as derived from the C++ standard. This is only meant as a logical reference. Implementation details may differ.

The Cost of Smart Pointers

Smart Pointers provide a clear solution to lifetime and ownership problems but there are some drawbacks:

1. `shared_ptr` points to a control block that is potentially allocated at a different location on the Heap (probably causing cache efficiency problems).
2. `unique_ptr` stores an additional pointer for the deleter, doubling the required memory for the `unique_ptr` itself.
3. Large chains of `unique_ptr` can exceed the stack because of recursing deleter calls [Sut16].

Evaluating these performance implications is not part of this thesis. However, since smart pointers are an important part of modern C++ programming they need to be evaluated in the context of graphical applications in the future.

6.3.2 Garbage Collection

Garbage collection is a form of automatic memory management. When programming in a garbage collected environment, a *garbage collector* tracks memory accesses / references and automatically frees memory regions that are no longer needed by the program. Prominent examples for garbage collected programming languages are Java and C#. In graphical applications garbage collection is often not a desired memory management solution because of its performance implications. For example: Java uses a "Stop the World" approach to garbage collection [ORC]. "Stop the World" means that all application threads are stopped as long as the garbage collector is running. If a target frame time of 16ms (or 11.1ms in VR) is required, unpredictable interruptions are not tolerable.

Another approach to garbage collection is implemented in Epics Unreal Engine 4 [EPI18]. In Unreal Engine a custom garbage collector is implemented on top of C++ using a custom C++ preprocessor. With implementing a custom solution the developers have better control over garbage collection events and can therefore minimize the performance impact.

Being able to rely on a garbage collector is convenient and offloads a huge amount of responsibility and complexity off of programmers. However the implementation of a garbage collector itself is very complex, and tweaking garbage collector settings for maximum performance can be difficult. Even with a garbage collector in place, memory allocations need to be reduced. While a memory leak is not possible the amount of “garbage” needs to be as small as possible to reduce the number of collection events. This means programmers do still have to keep memory in mind. The benefits of a garbage collector therefore are debatable.

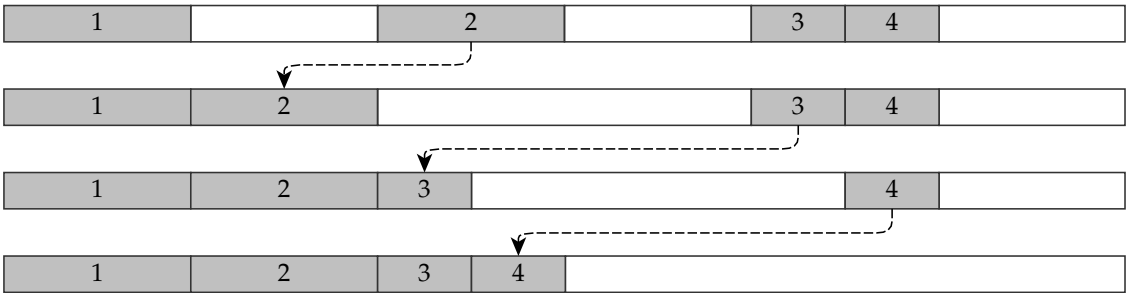


Figure 6.11: Defragmenting memory by relocating allocated blocks of memory.

6.3.3 Memory Defragmentation

Memory fragmentation can be a huge problem as shown insection 5.1.2. To solve fragmentation, the memory can be *defragmented*. Defragmentation describes shifting allocated memory blocks to lower memory regions. This in turn moves unallocated memory blocks to higher memory regions. If this process is repeated continuously, all allocated blocks will be in a continuous memory region eventually. As a result only a single, big memory region is free - the memory is not fragmented.

The advantages of degramenting memory are clear: If memory can never be fragmented the use of restricting allocators like a linear allocator is not required just because it reduces framentation. Instead, the allocator can be chosen depending on other requirements like simplicity.

But this advantage comes at a cost. Relocating allocated memory regions can be problematic. If a pointer points to an address in a memory region it points to something else if the region gets relocated. To circumvent this problem pointers that are aware of relocatable allocations need to be used. The relocations then have to be propagated to these special pointers. According to Jason Gregory, all engines developed by Naughtly Dog have support for memory defragmentation [Gre09]. With respect to the fact that some games developed by Electronic Arts have consumed system memory almost completely and would fail to run without fragmentation countermeasures [Ped07], memory defragmentation is an important consideration.

7 Multithreading

When it comes to high preformance applications one way of optimizing execution speed is multithreading. Multithreading allows the spreading of tasks across multiple physical CPU cores or hardware threads. This enables parallel execution and therefore significant performance improvements. A well known difficulty when dealing with multiple threads are so called *races* or *race conditions*. A race condition describes a problem that occurs when an algorithms result is bound to a specific thread execution order. Since the scheduling of threads is typically offloaded to the operating system, a specific execution order and execution time cannot be guaranteed. Data races can happen when multiple threads access the same data while one of the threads is writing to that data. Even incrementing an integer can be a problem because it consists of three distinct operations:

1. Load the value into a register
2. Add 1 to the value in the register
3. Write the value from the register back to memory

If a thread increments a variable, while another thread is incrementing the same variable and is between steps 1 and 3, the final result will be wrong. The second incrementation loads the value before the result of the first incrementation is written to memory. The variable effectively got incremented only once. If an algorithm does not protect such critical code paths by using thread-safe facilities like mutexes or semaphores the algorithm is subject to race conditions. How custom allocators or the provided data structures can be altered to be thread-safe is not part of this thesis since it does not differ from ordinary multithreaded programming.

Another problem that can be caused by multithreaded execution in combination with CPU caches is the so called *false sharing*. False sharing causes performance problems and severe thread scalability issues. The following section explains false sharing in detail and shows how it can be prevented. It is based on the article “Eliminate False Sharing” by Herb Sutter that was published in 2009.

7.1 False Sharing

First, recall the following paragraph from section 3.1.6 (Coherency in multi-core processors).

> A single cache line can be stored in multiple caches which are physically located in different CPU cores. As soon as one of the cores *writes* data to that

```
1  int result[P];
2  // Each of P parallel workers processes 1/P-th
3  // of the data; the p-th worker records its
4  // partial count in result[p]
5  for( int p = 0; p < P; ++p )
6      pool.run( [&,p] {
7          result[p] = 0;
8          int chunkSize = DIM/P + 1;
9          int myStart = p * chunkSize;
10         int myEnd = min( myStart+chunkSize, DIM );
11         for( int i = myStart; i < myEnd; ++i )
12             for( int j = 0; j < DIM; ++j )
13                 if( matrix[i*DIM + j] % 2 != 0 )
14                     ++result[p];
15     } );
16 // Wait for the parallel work to complete
17 pool.join();
18 // Finally, do the sequential "reduction" step
19 // to combine the results
20 odds = 0;
21 for( int p = 0; p < P; ++p )
22     odds += result[p];
```

Listing 7.1: Pseudocode example given by Herb Sutter in [Sut09]

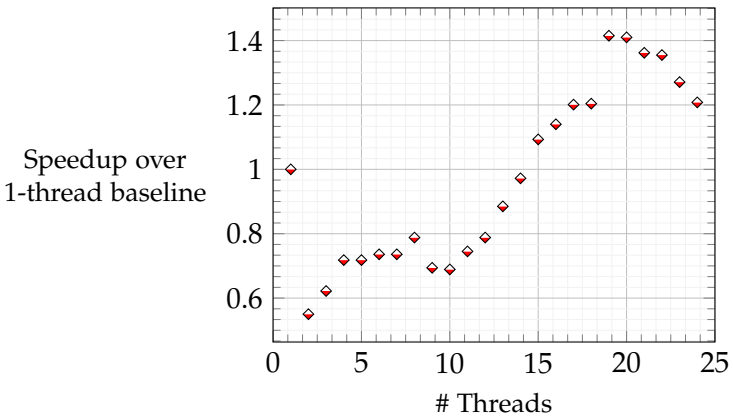


Figure 7.1: Speedup of the code from listing 7.1 with an increasing number of threads [Sut09]. The test system has a 24-core CPU.

> cache line the CPU has to ensure that the other cores do not read from their version of the cache line since the data is now outdated. This is achieved by marking the cache line *dirty* in the CPU cache of the writing core and *invalid* in all other cores [Mey14].

A simple example is used to illustrate what false sharing is and where it comes from: Suppose the amount of odd numbers in a matrix of size $DIM \times DIM$ need to be counted. Instead of traversing the full matrix single threaded, multiple worker threads are started to speed up the process. Each thread counts the elements of a predetermined part of the matrix. A potential implementation can be seen in listing 7.1.

Instead of having a single variable that gets incremented by all threads, each thread has its own slot in the array `int result[P]` (line 1) - no data is accessed by multiple threads simultaneously. To get the final result all threads have to finish their calculation before the interim results are summed up. Locking a mutex every time a shared counter is incremented is not required. Since mutex locking and unlocking has performance implications this algorithm is expected to be faster while maintaining the effective race condition prevention. In reality this algorithms has severe performance and scalability issues as can be seen in fig. 7.1. With 2 to 15 threads the multithreaded execution performs worse than a single thread. The maximum speedup with up to 25 threads is at about 1.4 times the single threaded performance. This unexpectedly bad performance is the result of false sharing.

The problematic part in the implementation example is where the temporary result of each executing thread is stored. In line 1 an array `int result[P]` is created that has a slot for every worker thread to store its counter. This counter is initialized in line 7 and incremented in line 14. Since this is done by every worker thread, multiple threads have a copy of one of the cache lines where `int result[P]` is stored. When a thread increments its counter in line 14 this cache line is marked invalid for all other threads. As soon as another worker wants to increment *its* counter, the cache line has to be loaded again because it is marked invalid. The now following increment invalidates the cache line once again for every other thread. This repeats until all worker threads are done. A single 64 byte cache line can fit $64/4 = 16$ slots of the result array. The probability that two workers have their counters on the same cache line (and therefore invalidate each others cached value) is very high. This behavior destroys the benefits of the CPU cache. The threads *share* a single cache line to ensure coherency even though it is not required in this case.

To prevent false sharing, it is required to spread out the counter variables in memory. Listing 7.2 shows how the example can be fixed. The required change is simple: Every thread now has a local variable (line 8) as a counter. These variables are stored on the threads stack. Because of that, multiple counter variables no longer happen to be on the same cache line. The result array still exists and still has to be accessed, in line 16 the local counter is written to the array. This can introduce false sharing, but since it only happens once per worker thread its performance impact is negligible. Figure 7.2 shows that the algorithm now scales perfectly linear with the number of threads.

```
1  int result[P];
2
3  // Each of P parallel workers processes 1/P-th
4  // of the data; the p-th worker records its
5  // partial count in result[p]
6  for( int p = 0; p < P; ++p )
7     pool.run( [&,p] {
8          int count = 0;
9          int chunkSize = DIM/P + 1;
10         int myStart = p * chunkSize;
11         int myEnd = min( myStart+chunkSize, DIM );
12         for( int i = myStart; i < myEnd; ++i )
13            for( int j = 0; j < DIM; ++j )
14              if( matrix[i*DIM + j] % 2 != 0 )
15                ++count;
16         result[p] = count;
17     } );
18 //  etc. as before
```

Listing 7.2: Pseudocode example solution given by Herb Sutter in [Sut09]

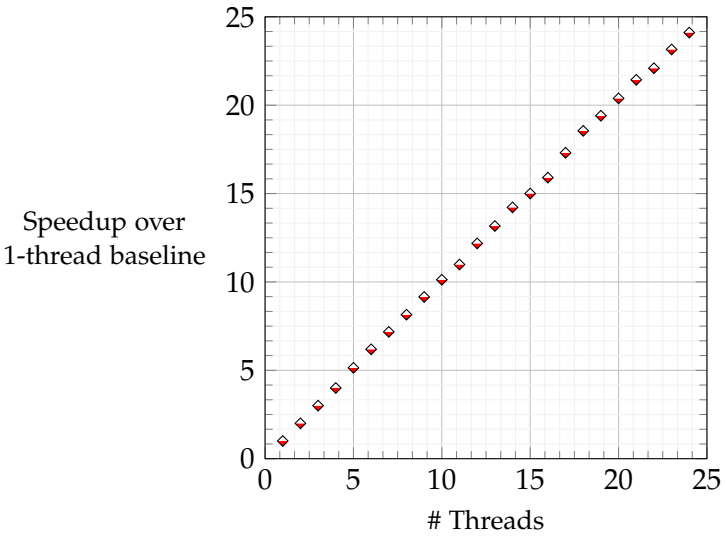


Figure 7.2: Speedup of the code from listing 7.2 with an increasing number of threads [Sut09]. The test system has a 24-core CPU. Performance scales perfectly with thread count.

8 Observations

8.1 Problems with C++

8.1.1 Template Overuse

The C++ template system is a powerful language feature the enables compile time programming. It allows compile time polymorphism using templated functions and types or type introspection using template specializations. In practice the existance of a template often requires other parts of a programs source code to be templated as well. For example:

Given a function that requires a vector of floats as parameter. The functions declaration could look like this:

```
void foo(const eastl::vector<float>& vec);
```

Somewhere in the program a float vector is created, but for performance reasons the vector is backed by a linear allocator. The vector's type therefore is `eastl::vector<float, Linear_Allocator>`. Since the types do not match, this vector cannot be passed to `foo`. To allow passing this vector to `foo`, its signature has to change:

```
template<typename Allocator_Type>
void foo(const eastl::vector<float, Allocator_type>& vec);
```

Now `foo` is a function template even though it can never change `vec` (since `vec` is declared `const`) and therefore never invokes any calls to the allocator. This is just a small example to illustrate the problem: If part of a system makes heavy use of templates that go beyond simple type information on containers, the rest of the system often is forced to propagate the use of templates outwards. This can only be circumvented by using run-time polymorphism in the form of virtual functions. The problem with virtual functions is that they are not as efficient as compile time polymorphism as shown in section 5.1.1 (Avoiding Virtual Function Calls). Templates do not have that run time overhead but slow down compilation speed, increase source code complexity and increase the size of the final executable.

8.1.2 Allocator Model

The allocator model that was (and still is) used in C++ is suboptimal for various reasons [Ped07]. Even though it was changed numerous times [18b], the suboptimal

allocator model remained in place for compatibility reasons. Up until C++17 the allocator model of the C++ STL (`std::allocator`) was fully *type based*. This means that it was impossible to pass an existing allocator to a container. Instead, only the allocator type was passed as a template argument and the container constructed its own instance of that type. Using type based allocators as if they were *instance based* requires usage of type tags to make the allocator itself a template. With type tags it is possible to treat different allocator template instantiations as one instance. This however leads to the problem of template overuse and complicates the usage of custom allocators tremendously.

The C++17 standard changes the allocator model again. Instead of an incremental change C++17 reworks everything that surrounds memory allocation by introducing the new namespace `std::pmr`. Pmr stands for "polymorphic memory resource" and is the new way of managing memory allocations. In `std::pmr` the role of the allocator is changed to only being a handle to a memory resource. The actual allocation is then performed in the so called `polymorphic_memory_resource`. As of writing none of the large compiler developers provide an implementation of the `std::pmr` namespace.

For that reason all implementations that are part of this thesis are based on the EASTL. In the EASTL allocators are instance based. Passing an allocator to *e.g.* a `vector` is supported and used throughout the examples given.

8.1.3 Limited Memory Layout Options

C++ is a powerful language that is capable of almost anything with enough patience and the willingness to implement complex, templated code. An example for that is the SoA / AoS container implementation that encapsulates and hides the underlying memory layout. This implementation was not provided as part of this thesis, instead it was referred to the existing LibFlatArray library [And18]. This library requires highly complex template specializations for every struct that is used in its context. To make usage easy these specializations are hidden behind simple macro invocations. The fact that the main macro that generates the code for the required template specializations emits more than 800 lines of code shows the complexity involved. For cache efficiency it would be beneficial if the C++ compiler would provide the ability to use primitive arrays and STL containers with an SoA memory layout. Jonathan Blow showcases such a feature as part of a new programming language in his talk "Data-Oriented Demo: SOA, composition" [Blo15] and shows that this abstraction can be done on a compiler level.

9 Conclusion and Future Work

This thesis provided an overview of the CPU architecture of modern computer systems and derived rules that assist in writing efficient code in response to that. Sample implementations that were used for performance tests and as illustrations of the concepts shown are provided. While implementing these samples and researching advanced C++ language features it became more and more clear that there is no universal solution to memory management. Providing custom allocators cannot solve all allocation related problems single handed. Providing specialized data structures cannot guarantee efficient cache usage. Making heavy use of smart pointers does not solve but *assist* in lifetime management of resources. The importance of optimizing towards good memory usage and cache friendliness was shown by various performance tests. These tests showed huge performance improvements.

Having developers dedicated to memory management on a team is not sufficient. These developers can only provide some generic facilities like common allocator types or object pools. To achieve the performance improvements that were shown throughout this thesis, everyone on the team has to optimize towards the presented metrics. Every developer needs to know about the memory subsystem and caches, and every developer needs to be aware of the implications of inefficient cache usage and Heap allocations. In the end, every developer needs to be able to evaluate memory management and cache efficiency for any given algorithm. Memory management is in many cases not something that can be optimized after an initial draft of a system. In most cases, memory management should be treated as a first class citizen when writing software. This is especially true if the targeted hardware is not as powerful as a top of the line, high end PC.

The history of DRAM shows that the issues of slow main memory will continue to exist. With this thesis as a basis it is therefore important to continue researching the discussed topics. The `std::pmr` namespace might be able to solve some of the issues by reducing the number of template functions and complexity. As soon as full implementations of this important C++17 feature are available they need to be evaluated with regard to performance as well as ease of use. Guidelines for smart pointer usage in the context of high performance applications are required and their performance implications need to be evaluated in detail.

Every update to the C++ standard continues to loosen the bond that forces every developer to know the details of CPU caches and the memory subsystem. In the future it might be possible to hide the underlying memory model behind an abstraction layer without sacrificing performance. Until then, it is important for every programmer to understand the details of memory management and the memory subsystem to ensure

optimal performance.

List of Figures

List of Tables

Bibliography

[18a] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. English. Version 248966-040. Intel. Mar. 8, 2018. April 2018.

[18b] *std::allocator*. cppreference.com, 2018.

[90] *8086 16-BIT HMOS MICROPROCESSOR 8086/8086-2/8086-1*. Order Number: 231455-005. Intel. Sept. 1990.

[Act14] M. Acton. *Data-Oriented Design and C++*. Talk at CppCon. 2014.

[And18] D. X. Andreas Schäfer Kurt Kanzenbach. *LibFlatArray*. Tech. rep. Friedrich-Alexander-Universität Erlangen-Nürnberg, Shanghai Jiao Tong University, 2018.

[Blo15] J. Blow. *Data-Oriented Demo: SOA, composition*. 2015.

[BO10] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 2nd. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0136108040.

[DH95] K. Driesen and U. Hölzle, eds. *The Direct Cost of Virtual Function Calls in C++*. 1995.

[Dos+17] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. " CARLA : An Open Urban Driving Simulator." In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[Dre07] U. Drepper. *What Every Programmer Should Know About Memory*. 2007.

[EPI18] *Garbage Collection*. Unreal Engine documentation. Epic.

[Fer+06] A. Ferrari, A. batson, M. Lack, and A. Jones. *x86 Assembly Guide*. 2006. URL: `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html` (visited on Aug. 13, 2018).

[FHW08] K.-I. Friese, M. Herrlich, and F.-E. Wolter. "Using Game Engines for Visualization in Scientific Applications." In: *ECS*. 2008.

[Fre18] I. Free Software Foundation. *[gcc] Contents of new$_o$pnt.cc*. 2018. URL: `https://gcc.gnu.org/viewcvs/gcc/trunk/libstdc++-v3/libsupc++/new_opnt.cc?view=markup` (visited on July 17, 2018).

[GD18] *VR Performance Best Practices*. Google LLC.

[Gre09] J. Gregory. *Game Engine Architecture*. Jeff Lander, 2009.

[Gri+18] Grinberg, K. Ono, N. Segal, and K. Netter. "VIRTUALIZATION METHOD OF VERTICAL-SYNCHRONIZATION IN GRAPHICS SYSTEMIS." US 8,754,904 B2. June 17, 2018.

[HZR16] M. Halpern, Y. Zhu, and V. J. Reddi. "Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction." In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Mar. 2016, pp. 64–76. DOI: 10.1109/HPCA.2016.7446054.

[ISO12] ISO. *Working Draft, Standard for Programming Language C++*. ISO N3337. Geneva, Switzerland: International Organization for Standardization, 2012.

[Knu74] D. Knuth. *Structured Programming with go to Statements*. Stanford University, 1974.

[Le 14] F. Le Gall. "Powers of Tensors and Fast Matrix Multiplication." In: *ArXiv e-prints* (Jan. 2014). arXiv: 1401.7714 [cs.DS].

[Mel09] A. C. de Melo. *pahole(1) - Linux man page*. Feb. 2009.

[Mey14] S. Meyers. *Cpu Caches and Why You Care*. code::dive conference. 2014.

[Mic06] D. Michael. *Serious games: games that educate, train and inform*. OCLC: ocm62780159. Boston, Mass: Thomson Course Technology, 2006. ISBN: 9781592006229.

[OCD18] *Guidelines for VR Performance Optimization*. Oculus VR.

[ORC] *Java Garbage Collection Basics*. Oracle.

[Ped07] P. Pedriana, ed. *EASTL – Electronic Arts Standard Template Library*. open-std.org, 2007.

[Pim18] K. Pimentel. *Animated Children's Series ZAFARI Springs to Life with Unreal Engine*. 2018. URL: https://www.unrealengine.com/en-US/blog/animated-children-s-series-zafari-springs-to-life-with-unreal-engine (visited on June 11, 2018).

[Por17] M. Porter. *Virtual memory and Linux*. Talk at Embedded Linux Conference Europe. 2017.

[SHW11] D. Sorin, M. Hill, and D. Wood. *A Primer on Memory Consistency and Cache Coherence*. Vol. 6. Nov. 2011.

[Slo16] K. Sloan. *How NASA Trains Astronauts with Unreal Engine*. 2016. URL: https://www.unrealengine.com/en-US/blog/how-nasa-trains-astronauts-with-unreal-engine (visited on June 11, 2018).

[Sut09] H. Sutter. "Eliminate False Sharing." In: *Dr.Dobb's* (2009).

[Sut14] H. Sutter. *Back to the Basics! Essentials of Modern C++ Style*. Talk at CppCon. 2014.

[Sut16] H. Sutter. *My CppCon talk video is online*. Sept. 27, 2016.

[VAL] *Valgrind User Manual*. URL: http://valgrind.org/docs/manual/manual-intro.html (visited on Aug. 10, 2018).

[Vis14] V. Viswanathan. *Disclosure of H/W prefetcher control on some Intel processors*. Sept. 24, 2014. URL: https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors (visited on Aug. 18, 2018).

[Zat17] G. Zatkin. "Awesome Video Game Data." Game Developers Conference. 2017.

[ZVN03] C. Zhang, F. Vahid, and W. Najjar. "Energy benefits of a configurable line size cache for embedded systems." In: *IEEE Computer Society Annual Symposium on VLSI, 2003. Proceedings.* Feb. 2003, pp. 87–91. DOI: 10.1109/ISVLSI.2003.1183357.