WILHELM BÜCHNER HOCHSCHULE
Mobile University of Technology

Fachbereich Informatik

Bachelorarbeit

im Studiengang Game Development (B.Sc.)

zur Erlangung des akademischen Grades
Bachelor of Science

Thema:	Softwarearchitektur, Programmier- und Optimierungstechniken im Kontext von Spiele-Engines
Autor:	Michael Gerhold
Betreut durch:	Dr.-Ing. Thomas Kalbe
Abgabetermin:	02.11.2021

Abstract

Bei Videospielen handelt es sich um Software mit Echtzeitanforderungen. Um diese Anforderungen erfüllen zu können, ist eine möglichst effiziente Ausnutzung moderner Hardware von essenzieller Bedeutung. Durch die Art und Weise, *wie* Software programmiert wird, können Optimierungsmechanismen moderner Hardware sowohl ausgehebelt als auch verstärkt werden. Das Ziel der vorliegenden Arbeit ist die Vorstellung von Architekturmustern und Programmiertechniken, welche die Eigenschaften moderner Hardware berücksichtigen und eine möglichst effiziente Programmausführung ermöglichen. Die Arbeit soll dabei mehr als nur einen Leitfaden darstellen, sondern auch tieferes Verständnis vermitteln, indem jeweils die theoretischen Grundlagen für die gefundenen Schlussfolgerungen erläutert werden.
Begleitend zu dieser Arbeit wurde außerdem eine rudimentäre Spiele-Engine implementiert, bei welcher die dargestellten Konzepte umgesetzt wurden. Der jeweils aktuelle Stand der Implementierung ist zu finden unter `https://github.com/mgerhold/engine2d`.

Inhaltsverzeichnis

Abstract i

1 Einleitung 1

2 Programmierung unter Berücksichtigung der Optimierungstechniken moderner Hardware 3

2.1 Optimieren von Code . . . 3
2.1.1 Definition und Fokus im Kontext dieser Arbeit . . . 3
2.1.2 Wann optimieren? . . . 5
2.1.3 Relevanz von O-Notation . . . 5
2.1.4 Hot Code versus Cold Code . . . 6
2.1.5 Zeitmessungen . . . 6
2.2 Caching . . . 7
2.2.1 Funktionsweise . . . 7
2.2.2 Zeitmessungen . . . 9
2.3 Cache-Assoziativität . . . 12
2.4 Pre-Fetching . . . 14
2.4.1 Funktionsweise . . . 15
2.4.2 Zeitmessungen . . . 16
2.5 Instruction-Caching . . . 19
2.6 Branch-Prediction . . . 21
2.6.1 Funktionsweise . . . 22
2.6.2 Zeitmessungen . . . 23
2.6.3 Branchless programming . . . 24
2.7 Parallelisierung . . . 25
2.7.1 Motivation . . . 26
2.7.2 Amdahlsches Gesetz . . . 27
2.7.3 Cache-Ping-Pong . . . 29
2.7.4 False-Sharing . . . 31
2.8 Softwareentwicklung im Kontext des Betriebssystems . . . 33
2.8.1 Dynamische Speicherallokationen . . . 35
2.8.2 Zeitmessungen . . . 36
2.9 Einfluss des verwendeten Programmierparadigmas . . . 37
2.10 Rendering-Optimierungen . . . 40

3 Repräsentation von Objekten in der virtuellen Spielwelt 42

3.1 Prozedurale Programmierung . . . 42
3.2 Objektorientierter Entwurf . . . 43
3.3 Composition over Inheritance . . . 47

3.4 Entity-Component-System-Architektur 48
3.4.1 Verbesserungen der Komponenten-basierten Architektur 49
3.4.2 Data-oriented Design 51
3.5 Implementierungen in etablierten Produkten bzw. Projekten . . 51

4 Fallstudie: Implementierung einer rudimentären 2D-Engine 53
4.1 ECS-Architektur 53
4.1.1 Verwaltung wiederverwendbarer Entities 55
4.1.2 Speicherung der Komponenten 56
4.1.3 Verwaltung verschiedener Sparse Sets 58
4.1.4 Funktionalität durch Systeme 60
4.2 Rendering 61
4.3 Input 62
4.4 Scripting 63
4.5 Weitere Systeme 65
4.6 Demonstration 66
4.6.1 Technische Demonstration 66
4.6.2 Implementierung eines Flappy Bird-Klons 67

5 Fazit 72

Anhang iv

Abbildungsverzeichnis xvi

Listingverzeichnis xvii

Literaturverzeichnis xviii

1 Einleitung

Videospiele sind Software. Allerdings handelt es sich bei ihnen um eine sehr spezialisierte Form von Software:

- Spiele unterstützen verschiedenste Eingabegeräte, z. B. Mäuse, Tastaturen, Gamepads sowie speziell angepasste Controller wie Gitarren oder Tanzmatten,
- ihre Ausgabe erfolgt multimedial, z. B. visuell, auditiv und haptisch (Vibration, Druckwiderstand),
- manche Spiele – im Regelfall besonders teure Produktionen – nutzen die neueste zur Verfügung stehende Technologie, werden dort zu Innovationstreibern und liefern sogar branchenübergreifende Beiträge.

Aus diesen Anforderungen ergibt sich der folgende Versuch einer Definition: »Most [...] video games are examples of what computer scientists would call *soft real-time interactive agent-based computer simulations.*«[1] Jeder Bestandteil dieser Definition[2] stellt besondere Anforderungen an die Implementierung. Die vorliegende Arbeit fokussiert sich dabei auf Techniken zur Einhaltung der genannten Echtzeitanforderungen, denen ein Videospiel unterliegt: Durch die zwingend benötigte Bildwiederholrate – 24 Bilder pro Sekunde, um Bewegungen als zusammenhängend wahrzunehmen; 30, 60 oder mehr Bilder pro Sekunde, um der Wiederholrate von Bildschirmen zu entsprechen – ist unumstößlich die maximale Zeitspanne festgelegt, innerhalb welcher der jeweils nachfolgende Frame dargestellt werden muss. Im Gegensatz zu manch anderen Echtzeitsystemen, z. B. in der Automobilbranche, resultiert aus einer Nichteinhaltung dieser Echtzeitbedingung zwar kein Personen-, jedoch u. U. ein enormer wirtschaftlicher Schaden.

Spiele-Engines

Die Technologie, die Spielen zugrunde liegt, hat i. d. R. große wiederverwendbare Teile. Diese Softwaremodule werden oft als »Spiele-Framework« oder »Spiele-Engine« bezeichnet. Die Grenzen zwischen Spiel, Framework und Engine sind dabei fließend und lassen sich daher nicht genau definieren. Um den Inhalt der vorliegenden Arbeit praxisnah und nutzbar zu halten, werden daher Themen und Techniken behandelt, die für die meisten Spiele von Relevanz sind.
Um die praktische Nutzbarkeit der dargestellten Inhalte zu belegen, wurden einerseits begleitend zu den eher theoretischen Betrachtungen jeweils Zeitmessungen durchgeführt, welche verschiedene mögliche Ansätze miteinander

[1] Gregory, 2019, S. 9
[2] Für eine ausführliche Beschreibung der Bestandteile siehe Gregory, 2019, S. 9 f.

vergleichen, sowie andererseits eine rudimentäre Spiele-Engine implementiert, in der die gewonnenen Erkenntnisse in die Praxis umgesetzt wurden.

2 Programmierung unter Berücksichtigung der Optimierungstechniken moderner Hardware

Da es sich bei Spielen um Echtzeit- und somit um zeitkritische Systeme handelt, liegt bei der Spiele-Entwicklung ein starker Fokus auf der Performance. Aufgrund einer asymmetrischen Entwicklung von Prozessoren und Speicher (siehe Abschnitt 2.2) wurde es nötig, bereits auf Hardware-Ebene Optimierungen vorzunehmen. Obwohl potenziell jede Art von Software von diesen Optimierungen profitieren kann, so kann die konkrete Implementierung der Software jedoch maßgeblichen Einfluss auf die Effektivität dieser Optimierungen haben. Die nachfolgenden Abschnitte erläutern den Einfluss, den Programmiererinnen und Programmierer beim Schreiben von Programmcode auf die Effizienz von Hardware-Optimierungen nehmen können, und belegen die daraus entstehenden Schlussfolgerungen sodann mit Zeitmessungen.

2.1 Optimieren von Code

Zunächst soll definiert werden, was im Rahmen der vorliegenden Arbeit unter dem Optimieren von Code verstanden werden soll. Anschließend werden verschiedene Hardware-Optimierungstechniken vorgestellt sowie deren bestmögliche Ausnutzung mit Code-Beispielen illustriert und durch Zeitmessungen belegt.

2.1.1 Definition und Fokus im Kontext dieser Arbeit

Programmcode, der seine Aufgaben bereits korrekt erfüllt, lässt sich in Bezug auf verschiedene Dinge optimieren, z. B.

- Lesbarkeit,
- Wartbarkeit,
- Testbarkeit,
- Geschwindigkeit und
- Energieverbrauch.

Während Geschwindigkeit und Energieverbrauch objektiv messbare Größen darstellen, lässt sich beispielsweise die Lesbarkeit des Codes nicht gänzlich objektiv bewerten, da eine solche Bewertung stark von den Erfahrungen der bewertenden Personen abhängt. Oft bringt außerdem eine Verbesserung in einem Bereich eine Verschlechterung in einem anderen Bereich mit sich. Als

```
1  float Q_rsqrt( float number ) {
2    long i;
3    float x2, y;
4    const float threehalfs = 1.5F;
5    x2 = number * 0.5F;
6    y  = number;
7    i  = * ( long * ) &y; // evil floating point bit level hacking
8    i  = 0x5f3759df - ( i >> 1 ); // what the fuck?
9    y  = * ( float * ) &i;
10   y  = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
11 //y  = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration,
12                                              // this can be removed
13   return y;
14 }
```

Listing 2.1: »fast inverse square root«-Algorithmus aus Quake III Arena[4]

recht bekanntes Beispiel sei an dieser Stelle der »fast inverse square root«-Algorithmus zu nennen, der zur Verbesserung der Geschwindigkeit im Spiel »Quake III Arena«[3] diente (siehe Listing 2.1).
Mit Hilfe dieses Algorithmus ist es möglich, näherungsweise den Kehrwert der Quadratwurzel einer Zahl zu berechnen. Die Berechnungsschritte, insbesondere in Zeile 8, sind nicht ohne weiteres nachzuvollziehen[5]. Diese Berechnung kann dazu genutzt werden, Vektoren zu normalisieren. In Computerspielen wird dieser Berechnungsschritt i. d. R. viele Male pro Sekunde ausgeführt, da er insbesondere für die Beleuchtungsberechnung von essenzieller Bedeutung ist. Deshalb entschied man sich hier bewusst *für* die bessere Performance und *gegen* gute Lesbarkeit. Oftmals beschränken sich die Ziele von Optimierungen auf objektiv messbare Größen:

> »The goal of optimization is to improve the behavior of a correct program so that it also meets customer needs for speed, throughput, memory footprint, power consumption, and so on. [...] Unacceptably poor performance is the same kind of problem for users as bugs and missing features.«[6]

Auch im Kontext dieser Arbeit sollen Messungen die Grundlage für Schlussfolgerungen darstellen. Der Fokus liegt dabei auf Zeitmessungen: Da es sich bei Videospielen um Software handelt, die strengen Echtzeit-Anforderungen unterliegt – z. B. dem Erreichen einer konstanten Bildrate von 30 oder 60 Bildern pro Sekunde –, stellt dies die wichtigste Metrik dar. Auf manchen

[3] id Software, 1999
[4] Quelle: `https://github.com/id-Software/Quake-III-Arena/blob/master/code/game/q_math.c#L552`, gekürzt und Formatierung geändert, aufgerufen am 10.08.2021
[5] Es werden Details der Darstellung von Fließkommazahlen auf Maschinenebene ausgenutzt, um diese Berechnung zu implementieren.
[6] Guntheroth, 2016

Plattformen wie z. B. Smartphones kann jedoch auch der Energieverbrauch eine wichtige Größe sein, die im Desktopbereich praktisch irrelevant ist.

2.1.2 Wann optimieren?

In Kontext der Frage, *wann* Code optimiert werden sollte, erlangte das folgende Zitat von Donald E. Knuth Berühmtheit:

> »We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.«[7]

Natürlich existieren jedoch auch dazu gegensätzliche Meinungen:

> »It is OK to optimize. It is OK to learn efficient programming idioms, and to apply them all the time, even if you don't know what code is performance-critical.«[8]

Es existiert auch der Ansatz, dass erst *nach* der Durchführung von Messungen optimiert werden sollte:

> »So, how do you know *which* [...] of your code to optimize? For that, you need a *profiler*. [...] It can tell you how much time is spent in each function. You can then direct your optimizations toward only those functions that account for the lion's share of the execution time.«[9]

Dieser Standpunkt wird unter anderem auch in McShaffry und Graham, 2013 vertreten[10]. Die vorliegende Arbeit gibt eine Übersicht über eine Auswahl relevanter Themen, die die Umsetzung einer Kombination aus den beiden letztgenannten Standpunkten ermöglicht. Durch Zeitmessungen sollen Architekturansätze und Programmiertechniken in Bezug auf Effizienz bewertet werden, sodass diese bereits bei der Planung von Software berücksichtigt werden können. Dies widerspricht nicht dem Ansatz, nach der Implementierung weitere Messungen durchzuführen, um eventuelle »Flaschenhälse« ausfindig zu machen.

2.1.3 Relevanz von O-Notation

Mit Hilfe der O-Notation lässt sich abschätzen, wie die Laufzeit oder der Speicherverbrauch eines Algorithmus mit größer werdenden Eingabedaten skalieren. Da es sich dabei um das asymptotische Verhalten handelt, wird eine Abschätzung mit Hilfe der O-Notation vor allem bei sehr großen Datensätzen

[7] Knuth, 1974
[8] Guntheroth, 2016
[9] Gregory, 2019, S. 99
[10] vgl. McShaffry und Graham, 2013, S. 846 f.

relevant. Die tatsächlich gemessene Laufzeit eines Algorithmus wird bei der O-Notation nicht berücksichtigt. Auch die Eigenschaften moderner Hardware finden hierbei keine Berücksichtigung. Das bedeutet, dass im Rahmen der O-Notation bei beispielsweise einer linearen Suche nicht von Interesse ist, ob die Suche auf einem Array oder einer einfach verketteten Liste als Datenstruktur stattfindet – in beiden Fällen wird die Laufzeit mit $O(n)$ abgeschätzt. In den späteren Abschnitten wird jedoch gezeigt, dass der Austausch der Datenstruktur deutliche Performanceunterschiede mit sich bringen kann. Im Kontext des Zählens von Instruktionen zur Messung von Performance konstatiert auch Jason Turner: »[...] a lot of what I learned in computer science about efficiency of algorithms [...] has just kind of been thrown away lately because it didn't take into account modern caching architectures and design of CPUs«[11]. Da im Rahmen dieser Arbeit Algorithmen nicht *theoretisch* betrachtet bzw. analysiert, sondern praxisrelevante Messungen durchgeführt werden sollen, wird auf die O-Notation nicht näher eingegangen.

2.1.4 Hot Code versus Cold Code

Beim Optimieren von Code sollte nicht jedem Programmteil die gleiche Aufmerksamkeit zukommen:

> »There is a well-known, albeit rather unscientific rule of thumb known as the *Pareto principle* [...]. It is also known as the *80/20 rule* [...]. In computer science, this principle has been applied to [...] software optimization, where as a rule of thumb 80% (or more) of the wall clock time spent running any piece of software is accounted for by only 20% (or less) of the code.«[12]

Man spricht von »Hot Code«, wenn ein Programmteil besonders häufig ausgeführt wird und damit für die Gesamtperformance der Software deutlich relevanter als anderer Programmcode ist. Auf diesem Code sollte beim Optimieren daher das Augenmerk liegen. Bei weniger häufig ausgeführtem Code spricht man dagegen von »Cold Code«. Mit Hilfe von Zeitmessungen können die entsprechenden Codeteile identifiziert werden.

2.1.5 Zeitmessungen

Sämtliche im Rahmen dieser Arbeit angefertigten Zeitmessungen basieren auf C++-Programmen. Die Zeitmessungen wurden in fast allen Fällen mit Hilfe des »Google Benchmark«-Frameworks angefertigt[13]. Dieses stellt sicher, dass

[11] transkribiert aus Irving et al., 2021
[12] Gregory, 2019, S. 99
[13] siehe `https://github.com/google/benchmark`, aufgerufen am 10.08.2021

die Tests ausreichend oft wiederholt werden, sodass sie zu aussagekräftigen Messergebnissen führen. Um Rückschlüsse auf die Praxis ziehen zu können, waren die Compiler-Optimierungen auf der höchsten Stufe (O3) aktiviert. Getestet wurde unter Windows 10 mit dem MSVC-Compiler von Microsoft sowie unter Ubuntu mit dem GCC-Compiler. Die Quelltexte der zur Zeitmessung geschriebenen Programme, die nicht bereits zu Teilen im Hauptteil der Arbeit zu sehen sind, befinden sich im Anhang dieser Arbeit.

2.2 Caching

Während bis in die 1980er Jahre die Leistungsfähigkeit der verschiedenen Hardwarekomponenten von Computern (Hauptprozessor, Speicher etc.) in etwa miteinander vergleichbar war, hat sich dies ab der darauf folgenden Dekade entscheidend verändert: Die Taktraten der Prozessoren stiegen deutlich schneller an als die des angebundenen Speichers, weshalb Speicherzugriffe seitdem oft den die Performance des Gesamtsystems begrenzenden Faktor darstellen[14].

2.2.1 Funktionsweise

Eine Technik, um diesen Effekt abzuschwächen, stellt das sog. »Caching« dar. Dabei verfügt jeder Prozessor bzw. Prozessorkern (bei Mehrkernprozessoren) jeweils über einen zusätzlichen Speicher – den Cache.

> »Cache memories are small, high-speed buffer memories used in modern computer systems to hold temporarily those portions of the contents of main memory which are (believed to be) currently in use. Information located in cache memory may be accessed in much less time than that located in main memory [...]. Thus, a central processing unit (CPU) with a cache memory needs to spend far less time waiting for instructions and operands to be fetched and/or stored.«[15]

Hardwareseitig sind dabei die Caches als sog. »static random access memory« (SRAM) und der Hauptspeicher als sog. »dynamic random access memory« (DRAM) implementiert. Diese beiden unterschiedlichen Speicherarten stellen Kompromisse zwischen Geschwindigkeit und Kosten dar: Während SRAM deutlich schneller als DRAM ist, ist er auch wesentlich kostspieliger. Daher beträgt die Speicherkapazität des verbauten SRAM üblicherweise nur einen Bruchteil der des DRAM[16].

[14] vgl. Nystrom, 2014, S. 269 f.
[15] Smith, 1982, S. 1
[16] vgl. Drepper, 2007, S. 5

Da es im Regelfall keinen Lokalitätszusammenhang zwischen Programminstruktionen und Daten gibt, besitzen die meisten modernen Prozessoren getrennte Caches für die Instruktionen und die Arbeitsdaten.
Die Geschwindigkeit, mit der auf Speicher zugegriffen werden kann, wird im Allgemeinen durch physikalische Gesetze begrenzt (man spricht von »wire delay«). Soll also eine bestimmte Geschwindigkeit erreicht werden, setzt das eine räumliche Nähe zum Prozessor sowie eine eingeschränkte Größe der Speicherkomponente voraus. Aus diesem Grund können Caches nicht beliebig groß werden, ohne Zugriffsgeschwindigkeit einzubüßen. Um dennoch größere Caches zu ermöglichen, kommt heutzutage i. d. R. eine Hierarchie aus mehreren Caches zum Einsatz. Man spricht von Level-1-, Level-2- sowie ggf. Level-3-Cache (je nach Prozessormodell)[17]. Mit aufsteigendem Level steigt auch die Größe des Caches, während die Zugriffsgeschwindigkeit sinkt. Die oben erwähnte Aufteilung in Instruction- und Data-Cache existiert dabei üblicherweise lediglich beim Level-1-Cache.
Caches sollen im Allgemeinen dazu dienen, Speicherlokalität auszunutzen. Das bedeutet, dass Speicherbereiche, die neben Speicherbereichen liegen, auf die zuletzt zugegriffen wurde, mit höherer Wahrscheinlichkeit ebenfalls kurz danach benötigt werden. Ein Beispiel, das Speicherlokalität illustriert, sind lokale Variablen einer Funktion: Diese liegen innerhalb des Stack-Frames der Funktion nebeneinander im Speicher. Wenn diese Variablen nun gemeinsam im Cache liegen, bringt das einen Geschwindigkeitsvorteil, da innerhalb der Funktion mit hoher Wahrscheinlichkeit hauptsächlich auf diesen Variablen operiert wird. Damit diese positive Eigenschaft überhaupt zum Tragen kommen kann, ist es jedoch eine Grundvoraussetzung, dass beim Zugriff auf eine Speicherstelle nicht lediglich diese Stelle in den Cache geladen wird, sondern ein größerer Bereich. Diesen Bereich nennt man »Cache-Line«. Auf aktueller Hardware hat eine Cache-Line üblicherweise eine Größe von 64 Byte. Sei also einmal angenommen, dass ein Programm auf eine 4 Byte große Integer-Variable zugreifen möchte und sich der entsprechende Speicher bisher noch nicht im Cache befindet, dann werden insgesamt 64 Bytes in den Cache geladen. Das notwendige Nachladen von Daten in den Cache bezeichnet man als »Cache-Miss«. Die ursprünglich »angefragte« Speicheradresse muss dabei nicht am Beginn dieser 64 Bytes liegen. Die Position der Daten innerhalb der Cache-Line ist von deren Adresse abhängig: Die Adressen von Cache-Lines sind immer Vielfache ihrer Länge, also im Regelfall Vielfache von 64 Bytes. Wird weiterhin das Beispiel von lokalen Variablen einer Funktion betrachtet, so zeigt diese Vorgehensweise deutlich, dass die Wahrscheinlichkeit hoch ist, dass sich nach dem Zugriff auf eine lokale

[17] Ggf. existiert daneben noch ein Level-4-Cache, der sich jedoch nicht auf dem eigentlichen Prozessor-Chip befindet, sondern extern angebunden ist. Vgl. dazu Solihin, 2015, S. 134 f.

```
1 struct Node {
2     uint64_t data;
3     Node* next;
4 };
```

Listing 2.2: Struktur der Knoten der einfach verketteten Liste

Variable nicht nur diese, sondern auch die restlichen lokalen Variablen im Cache befinden. Nachfolgende Zugriffe auf diese Variablen sind dadurch deutlich schneller, da sie sich bereits »näher am Prozessor« befinden.
Eine Intel i7-6700-CPU (Skylake-Architektur) benötigt für einen Zugriff auf den Level-1-Data-Cache 4 bis 5 Takte. Beim Level-2-Cache sind es 12 Takte und beim Level-3-Cache ca. 40 Takte. Zugriff auf den Hauptspeicher schlägt mit 42 Takten und zusätzlichen 51 ns zu Buche[18]. Die Zugriffszeiten können sich also zwischen einem Cache-Hit (wenn die benötigten Daten bereits im Cache vorhanden sind) und einem Cache-Miss um mehr als eine Größenordnung unterscheiden[19].
Auch in der Praxis sind aufgrund des Caches deutliche Performance-Unterschiede zu erwarten:

> »I wrote two programs that did the *exact same* computation. The only difference was how many cache misses they caused. The slow one was *fifty times* slower than the other.«[20]

Zusammenfassend lässt sich also sagen, dass das Speicherlayout einen großen Einfluss auf die zu erwartende Performance haben kann, denn »[...] if data is spread out, more cache lines must be loaded from memory onto the processor cache, which can increase memory access latency and reduce performance compared to data that's located close together.«[21]

2.2.2 Zeitmessungen

Die Auswirkungen des Caches werden nun anhand praktischer Experimente untersucht.

Speicherlokalität

Als erstes Beispiel dient eine einfach verkettete Liste, bestehend aus vier Knoten. Die Struktur der Knoten ist in Listing 2.2 zu sehen. Ein Knoten hat auf dem Testsystem eine Größe von 16 Bytes.

[18] Quelle: `https://7-cpu.com/cpu/Skylake.html`, Zugriff am 04.08.2021
[19] vgl. Drepper, 2007, S. 13 ff.
[20] Nystrom, 2014, S. 273
[21] Williams, 2019, S. 265

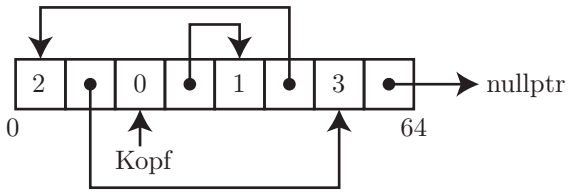

Abbildung 2.1: Alle Elemente der einfach verketteten Liste befinden sich in derselben Cache-Line.

Quelle: Abb. selbst erstellt

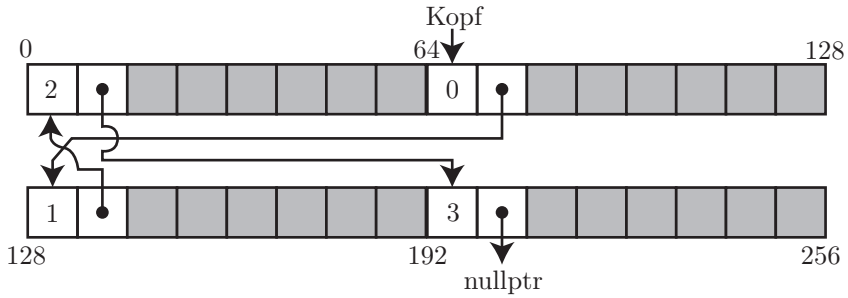

Abbildung 2.2: Die Elemente der verketteten Liste sind über mehrere Cache-Lines verteilt.

Quelle: Abb. selbst erstellt

Es soll untersucht werden, welche Auswirkungen die »Platzierung« der Knoten im Speicher hat. Bei einem ersten Test werden 64 Bytes Speicherplatz mit einem sog. »Alignment« von ebenfalls 64 Bytes allokiert. Das bedeutet, dass die Adresse des angeforderten Speicherplatzes an einem 64-Byte-Raster ausgerichtet (*aligned*) wird. Konkret bedeutet das, dass sich die Adresse ganzzahlig ohne Rest durch 64 dividieren lässt. Dadurch wird sichergestellt, dass der allokierte Speicher exakt einer Cache-Line entspricht[22]. Innerhalb dieses Speicherplatzes wird sodann eine einfach verkettete Liste in zufälliger Reihenfolge aufgebaut. Ein mögliches Speicherlayout ist in Abb. 2.1 gezeigt.
Als Vergleichsaufbau wird eine weitere, ebenfalls aus vier Knoten bestehende Liste aufgebaut; diesmal werden die Knoten der Liste jedoch *jeweils* auf der Startadresse einer Cache-Line platziert. Jedem Knoten steht sozusagen eine komplette Cache-Line zur Verfügung, wobei jeweils 48 der 64 Bytes ungenutzt bleiben. Ein Speicherlayout, das dabei entstehen kann, ist in Abb. 2.2 dargestellt. Bei der Zeitmessung wurde sodann jeweils die verkette Liste durchlaufen und die enthaltenen Daten summiert. Der Test ergab, dass die zweite Variante auf dem Testsystem ca. 35 % mehr Zeit in Anspruch nahm als die erste Variante. Dieses Ergebnis demonstriert, dass eine erhöhte Speicherlokalität bereits bei sehr kleinen Datenstrukturen positiven Einfluss auf das Laufzeitverhalten des Programms haben kann.

[22] Auf dem Testsystem haben die Cache-Lines eine Größe von 64 Byte.

Zugriffsmuster und Datenmenge

Ein weiteres Experiment soll herausfinden, wie sich die Zugriffszeiten bei größeren Datenstrukturen verhalten und welche Auswirkungen das jeweilige Zugriffsmuster dabei hat. Zu diesem Zweck wird ein Array im Heap-Speicher allokiert und danach werden alle Array-Elemente addiert. In die Zeitmessung fließen dabei lediglich die Lese- und Additionsoperationen ein[23]. Aufgrund der in Abschnitt 2.2 beschriebenen Arbeitsweise des Caches ist zu erwarten, dass Zugriffe auf benachbarte Daten deutlich schneller erfolgen als auf weit voneinander entfernte Daten. Um diese Annahme zu überprüfen, erfolgt das Iterieren über das Array mit verschiedenen Zugriffsmustern. Seien N die Anzahl der Arrayelemente, von denen alle jeweils eine Größe von einem Byte haben, und C die Größe einer Cache-Line in Bytes, wobei $N \gg C$. Es werden die folgenden Zugriffsmuster getestet:

1. Schrittweite 1 Byte
   - Folge der Indizes, auf die zugegriffen wird:
     $(a_i)_{i=0,...,N-1}$ mit $a_i = i$, also $(0, 1, 2, 3, ..., N-1)$
   - Dieses Zugriffsmuster soll die Tatsache ausnutzen, dass nach dem Laden einer Cache-Line die nachfolgenden Elemente ebenfalls bereits geladen wurden. Im Rahmen des verfügbaren Caches ist bei diesem Muster optimale Performance zu erwarten.

2. Schrittweite 64 Bytes
   - Folge der Indizes, auf die zugegriffen wird:
     $(b_i)_{i=0,...,N-1}$ mit $b_i = (iC) \bmod N + \left\lfloor \frac{iC}{N} \right\rfloor$,
     also $(0, C, 2C, ..., N-C, 1, C+1, 2C+1, ..., N-C+1, ..., N-1)$
   - Bei diesem Zugriffsmuster greifen zeitlich direkt aufeinander folgende Speicherzugriffe immer auf Adressen zu, die einen Mindestabstand von C Bytes haben. Das soll dazu führen, dass für jeden einzelnen Speicherzugriff jeweils eine neue Cache-Line geladen werden muss. Für dieses Zugriffsmuster wäre also die schlechtestmögliche Performance zu erwarten.

3. zufälliges Zugriffsmuster
   - Die Folge der Indizes, auf die zugegriffen wird, ist eine zufällig gewählte Permutation der unter 1. bzw. 2. gezeigten Folgen.
   - Bei diesem Zugriffsmuster sind mehr Cache-Misses als unter 1., aber weniger als unter 2. zu erwarten. Die erwartete Performance liegt also zwischen der der anderen beiden.

[23] Schreibvorgänge flossen in diesen Test nicht ein, führten jedoch zu ähnlichen Resultaten.

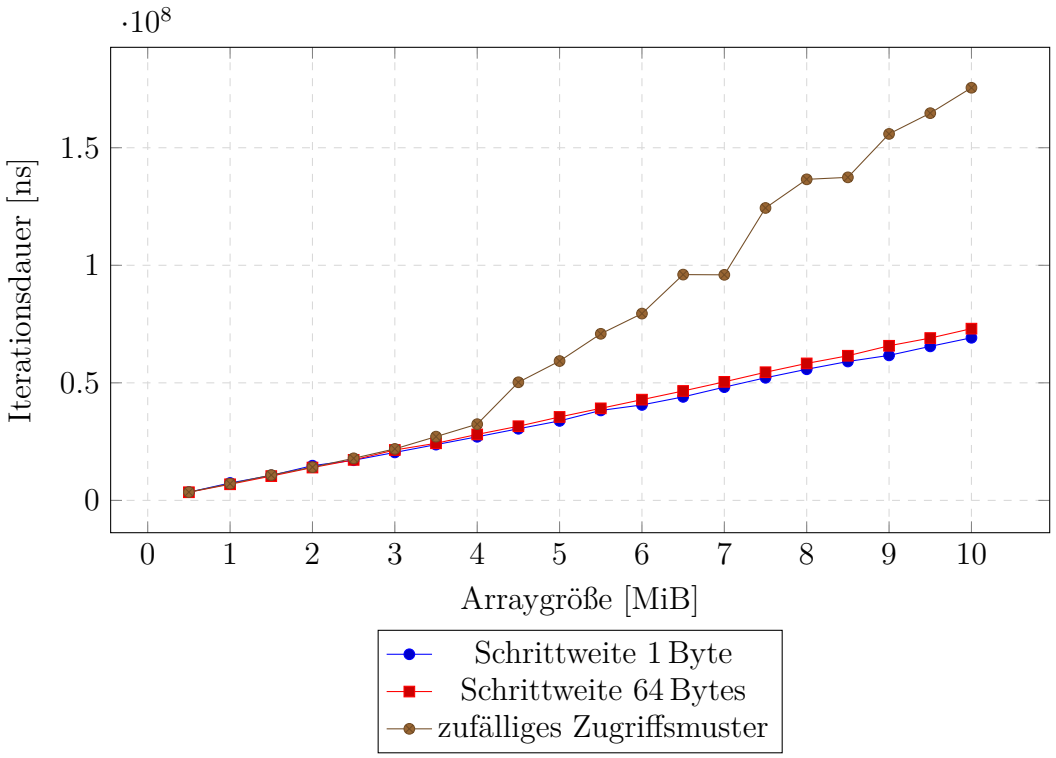

Abbildung 2.3: Iterationsdauer in Abhängigkeit der Datenmenge und des Zugriffsmusters

Quelle: Abb. selbst erstellt

Die Messergebnisse sind in Abb. 2.3 zu sehen. Zu erkennen ist, dass die Zugriffsmuster auf dem Testsystem bis zu einer Arraygröße von ca. 3 MiB praktisch keine Rolle spielen. Danach jedoch verschlechtert sich das zufällige Zugriffsmuster gegenüber den anderen beiden Mustern drastisch. Eine Verschlechterung gegenüber des erstgenannten Zugriffsmusters ist zu erwarten, da es ab einer gewissen Arraygröße nicht mehr möglich ist, das gesamte Array dauerhaft im Cache zu halten[24].

Dass sich das zweitgenannte Zugriffsmuster nahezu identisch zum erstgenannten verhält, lässt sich durch die bisher beschriebenen Eigenschaften des Caches nicht erklären. Auf diesen Sachverhalt wird jedoch in Abschnitt 2.4 ausführlich eingegangen.

2.3 Cache-Assoziativität

Eine wichtige Zusatzeigenschaft üblicher Cache-Implementierungen ist die sog. »Cache-Assoziativität«. Damit ist gemeint, dass ein Zusammenhang zwischen

[24] Die Auswirkungen des Caches kommen hier zum Tragen, obwohl der Cache des Testsystems deutlich größer als 3 MiB ist (der Level-3-Cache hat hier eine Größe von 12 MiB), jedoch wird dieser für alle laufenden Anwendungen sowie für das Betriebssystem gleichzeitig genutzt.

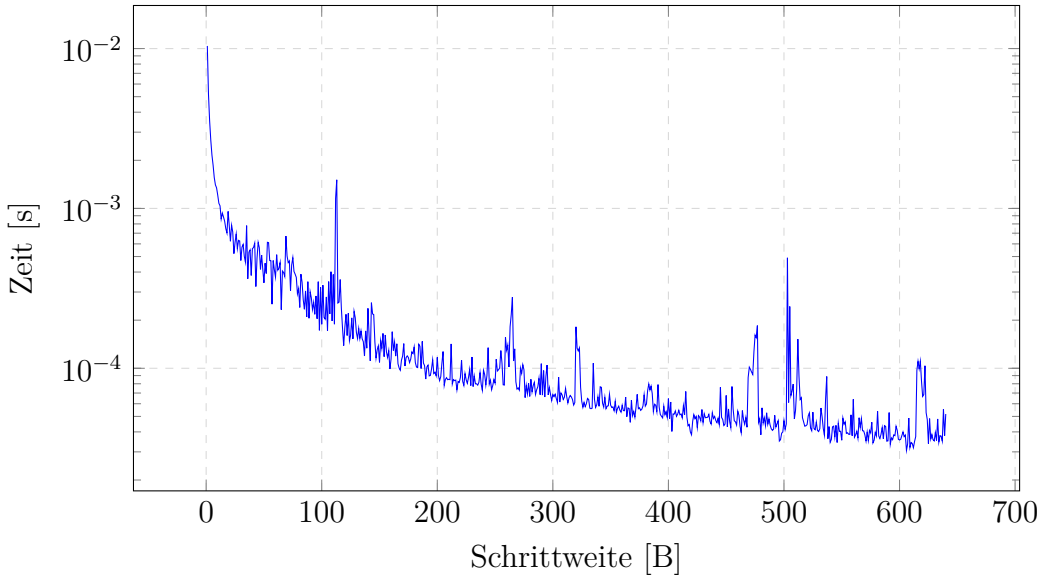

Abbildung 2.4: Einzel-Testreihe zu den Auswirkungen von Cache-Assoziativität (Arraygröße 8 MiB)
Quelle: Abb. selbst erstellt

der Adresse einer Cache-Line im Arbeitsspeicher (vor dem Laden) und im Cache (nach dem Laden) besteht: Wird eine Cache-Line geladen, kann diese nicht an einer beliebigen Stelle im Cache positioniert werden, sondern die mögliche(n) Platzierung(en) hängen von der zu ladenden Adresse ab. Eine übliche Implementierung, die einen Kompromiss zwischen Effizienz und Einfachheit darstellt, ist ein sog. »N-way set associative cache«. Bei einer solchen Implementierung stehen jeder Cache-Line N verschiedene Positionen innerhalb des Caches zur Verfügung, an denen sie platziert werden kann. Ein typischer Wert ist $N = 16$, was bedeutet, dass bis zu 16 verschiedene Cache-Lines, welche auf dieselben Positionen des Caches abgebildet werden, gleichzeitig im Cache gehalten werden können. In Abhängigkeit vom Zugriffsmuster auf den Speicher kann das im schlimmstem Fall dazu führen, dass die effektiv nutzbare Größe des Caches auf $16 \cdot 64$ Bytes sinkt – unabhängig von der tatsächlichen Größe des Caches[25].
Da zur Bestimmung möglicher Positionen innerhalb des Caches üblicherweise Teile der Basisadresse der jeweiligen Cache-Line genutzt werden, sind vor allem bei Zugriffsmustern, deren Schrittweite Zweierpotenzen betragen, Performanceverschlechterungen zu erwarten. Abb. 2.4 zeigt die Auswirkungen der Schrittweite beim Iterieren über einen 8 MiB großen Puffer auf die Laufzeit. Zu sehen sind deutliche Peaks vor allem bei den Schrittweiten 128, 256 sowie 512 MiB. Die positiven Auswirkungen des Caches sind in diesen Fällen also drastisch reduziert.

[25] vgl. Ostrovsky, 2010

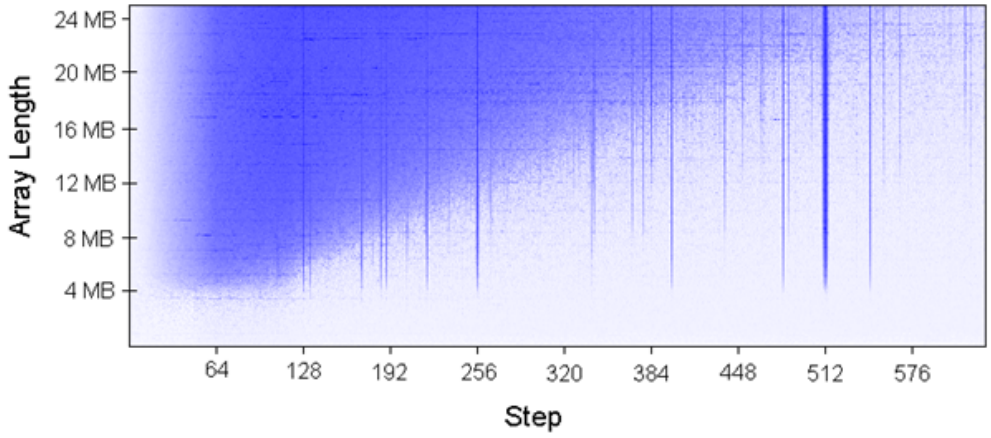

Abbildung 2.5: Auswirkungen von Cache-Assoziativität
Quelle: Ostrovsky, 2010

Abb. 2.5 zeigt die Ergebnisse einer noch größeren Anzahl von Messungen. Dunklere Bereiche im Diagramm entsprechen einer längeren Laufzeit. Bis zu einer gewissen Puffergröße – in der Abbildung 4 MiB – treten noch keine negativen Auswirkungen auf. Das liegt darin begründet, dass ausreichend viele relevante Cache-Lines gleichzeitig im Cache gehalten werden können (in Abhängigkeit der oben eingeführten Variablen N). Oberhalb dieser Größe sind die negativen Folgen jedoch in allen Messreihen ausgeprägt.
In Bezug auf die Programmierpraxis können diese Erkenntnisse dazu genutzt werden, unerwartete Performanceprobleme begründen und ggf. sogar beseitigen zu können: Wird z. B. über einen (hinreichend großen) Puffer gleichartiger Objekte iteriert, die alle eine Größe haben, die einer Zweierpotenz entspricht, kann durch eine absichtliche (und eigentlich unnötige) Vergrößerung dieser Objekte u. U. eine Verbesserung erreicht werden.

2.4 Pre-Fetching

Wie in Abschnitt 2.2 erläutert spricht man von Cache-Hits oder Cache-Misses, je nachdem, ob sich der Speicher, auf den zugegriffen werden soll, bereits im Cache befindet (*Hit*) oder nicht (*Miss*). Dabei sollen Cache-Misses so weit wie möglich reduziert werden, damit der Prozessor möglichst wenig Zeit damit verbringen muss, auf Daten zu warten. Man spricht in einem solchen Fall von einem »stall« (engl. für stehenbleiben, hinauszögern). »[...] one of the two aims of cache design is to minimize the miss ratio. Part of the approach to this goal is to select a cache fetch algorithm that is very likely to fetch the right

information, if possible, before it is needed.«[26]

2.4.1 Funktionsweise

Die Arbeitsweisen der Algorithmen, welche den in den Cache zu ladenden Speicher bestimmen, werden unterschieden in

- »Demand-Fetching«, bei dem eine Cache-Line immer genau dann in den Cache geladen wird, wenn sie benötigt wird, und
- »Pre-Fetching«, wobei versucht wird, die benötigten Speicherbereiche vorherzusagen und in den Cache zu laden, *bevor* Sie benötigt werden.

Demand-Fetching lässt sich aus offensichtlichen Gründen nicht gänzlich verhindern – schließlich lassen sich nicht alle Speicherzugriffe zuverlässig voraussagen. Können Algorithmen jedoch möglichst häufig von Pre-Fetching Gebrauch machen, so kann das Laden von Cache-Lines bereits parallel zur Ausführung von Instruktionen geschehen, welche auf bereits vorher geladenem Speicher operieren. Somit können die Wartezeiten des Prozessors minimiert werden.
Eine große Rolle bei Pre-Fetching-Algorithmen spielt die Auswahl der Cache-Lines, die im Voraus geladen werden sollen. Smith konstatiert dazu:

> »We believe that in cache memories, because of the need for fast hardware implementation, the only possible line to prefetch is the immediately sequential one [...]. That is, if line i is referenced, only line $i + 1$ is considered for prefetching. Other possibilities, which sometimes may result in a lower miss ratio, are not feasible for hardware implementation in a cache at cache speeds.«[27]

Diese Aussage ist aus heutiger Sicht jedoch als veraltet anzusehen: Neben komplexeren hardwarebasierten Mustererkennungsalgorithmen kommen heutzutage auch Software-Lösungen zum Einsatz, um Cache-Misses vorherzusagen oder zu minimieren. Letztgenannte Lösungen werden dabei üblicherweise von Compilern umgesetzt, können alternativ aber auch direkt in den Programmcode eingebracht werden[28]. Ggf. wird dieser Code aus dem Hauptprogramm ausgelagert und läuft in einem separaten Thread, der das laufende Programm analysiert und das Pre-Fetching anstößt.[29] Im Folgenden wird Hardware-Pre-Fetching betrachtet, da dieses für jegliche Software transparent und präsent ist, während Software-Pre-Fetching i. d. R. lediglich von Compilern eingesetzt wird.

[26] Smith, 1982, S. 481
[27] Smith, 1982, S. 482 f.
[28] i. d. R. mithilfe sog. »Compiler-Intrinsics«, also Compiler-spezifischen Befehlen, die nicht zum Umfang der Programmiersprache gehören und vom Compiler in entsprechenden Maschinencode übersetzt werden
[29] vgl. Byna et al., 2009, S. 405

Im Idealfall werden Cache-Lines derart im Voraus geladen, sodass jeder Zugriff auf den Speicher zu einem Cache-Hit führt und dass auf alle Cache-Lines, die geladen werden, auch tatsächlich zugegriffen wird. Um Verzögerungen zu verhindern, müssen Cache-Lines dabei ausreichend früh geladen werden. In der Praxis ist es unmöglich, diesen Idealfall zu erreichen, da Cache-Misses durch »aggressives« Pre-Fetching verringert werden, wodurch jedoch die Anzahl »ungenutzter«, im Voraus geladener Cache-Lines ansteigt. Auch der Zeitpunkt des Pre-Fetchings muss wohl überlegt sein: Zu früh geladene Cache-Lines belegen unnötig viel Speicher im Cache, zu spät geladene führen zu Wartezeiten[30].
Zwei Pre-Fetching-Techniken, die in modernen Prozessoren Verwendung finden, sind

- sequentielles Pre-Fetching sowie
- Stride-Prefetching.

Beim Stride-Prefetching wird analysiert, in welchem Abständen bzw. mit welcher Schrittweite (engl. *stride* für Schritt) aufeinander folgende Speicherzugriffe stattfinden. Es werden dann Cache-Lines mit derselben Schrittweite im Voraus geladen. Das sequentielle Pre-Fetching stellt dabei einen Spezialfall des Stride-Prefetchings dar, nämlich den Fall, dass die Schrittweite höchstens der Länge einer Cache-Line entspricht[31]. Auf die Implementierung des Pre-Fetchings in der Hardware wird an dieser Stelle nicht näher eingegangen, da sie keinen direkten Einfluss auf die Programmierung hat und aus Sicht der Software transparent ist.
Pre-Fetching-Techniken können die Ergebnisse der Zeitmessungen im Unterabschnitt »Zugriffsmuster und Datenmenge« des Abschnitts 2.2.2 (S. 11) erklären: Dort wurde mit einer festen Schrittweite von 64 Byte auf den Speicher zugegriffen. Die Zugriffe waren nur geringfügig langsamer als bei einer Schrittweite von einem Byte. Die leicht schlechtere Performance lässt sich in den »Rücksprüngen« erklären, die immer dann auftreten, wenn der ganze Speicher in 64-Byte-Schritten durchlaufen wurde und danach die nächste Iteration mit einer Verschiebung von einem Byte von vorne beginnt.

2.4.2 Zeitmessungen

Im Folgenden werden verschiedene Zugriffsmuster miteinander verglichen.

[30] vgl. Solihin, 2015, S. 162 f.
[31] vgl. ebd.

Schrittweite und Arraygröße

Es sollen die Auswirkungen der Schrittweite beim Iterieren eines Arrays gemessen und dabei dessen Größe berücksichtigt werden. Der Hardware-Prefetcher des Testsystems sollte dabei die jeweilige Schrittweite erkennen und Cache-Lines bereits laden, bevor sie gebraucht werden. Es ist dabei jedoch zu erwarten, dass die durch die Speicherzugriffe verursachte Latenz nicht komplett ausgeglichen werden kann: Die Schrittweite hat direkten Einfluss darauf, nach jeweils wie vielen Arrayzugriffen die nachfolgende Cache-Line zur Verfügung stehen muss. Seien die Größe jeder Cache-Line in Bytes mit C sowie die Schrittweite in Bytes mit S bezeichnet. Dann beschreibt der Quotient $\frac{C}{S}$, wie viele Zugriffe durchschnittlich pro Cache-Line möglich sind, bevor eine weitere geladen werden muss. Je größer die Schrittweite S dabei wird, umso häufiger werden neue Cache-Lines benötigt. Dies muss nicht unmittelbar ein Problem darstellen, da Techniken existieren, z. B. sog. »Stream Buffers«, welche bei einem Cache-Miss nicht nur die geforderte, sondern auch direkt mehrere darauf folgende Cache-Lines nachladen[32]. Erst wenn die Schrittweite einen gewissen Wert übersteigt, lässt sich die Latenz somit nicht mehr ausgleichen.
Die durch eine Vergrößerung der Schrittweite erwartete Performanceverschlechterung hängt zusätzlich auch von der Gesamtdatenmenge ab: Ist die betrachtete Datenstruktur hinreichend klein, kann sie ggf. dauerhaft im Cache gehalten werden, wodurch Cache-Misses – unabhängig vom Zugriffsmuster – nicht mehr auftreten, nachdem alle relevanten Cache-Lines erstmalig geladen wurden. Dies wurde bereits in Abschnitt 2.2 erläutert und dort durch Messergebnisse belegt. In Abb. 2.6 ist zu sehen, wie sich die Iterationsdauer (pro MiB) in Abhängigkeit der Gesamtdatengröße und der verwendeten Schrittweite beim Iterieren verhält. Die Messergebnisse bestätigen die Erwartungen: Ist die Gesamtdatenmenge so groß, dass nicht davon ausgegangen werden kann, dass diese dauerhaft im Cache gehalten werden kann, führen größere Schrittweiten zu einer deutlichen Verschlechterung.
Dennoch sind die positiven Auswirkungen des Hardware-Prefetchings zu messen: Abb. 2.7 zeigt den Vergleich zwischen dem Zugriff mit fester Schrittweite (1 Byte bzw. 512 Byte) und dem Zugriff mit zufälligem Muster. Dabei ist zu beobachten, dass sich die Laufzeit bei zufälligem Zugriff gegenüber der festen Schrittweite um bis zu ca. 600 % verschlechtern kann, was darauf zurückzuführen ist, dass das Pre-Fetching in diesem Fall praktisch wirkungslos ist.

[32] vgl. Solihin, 2015, S. 164

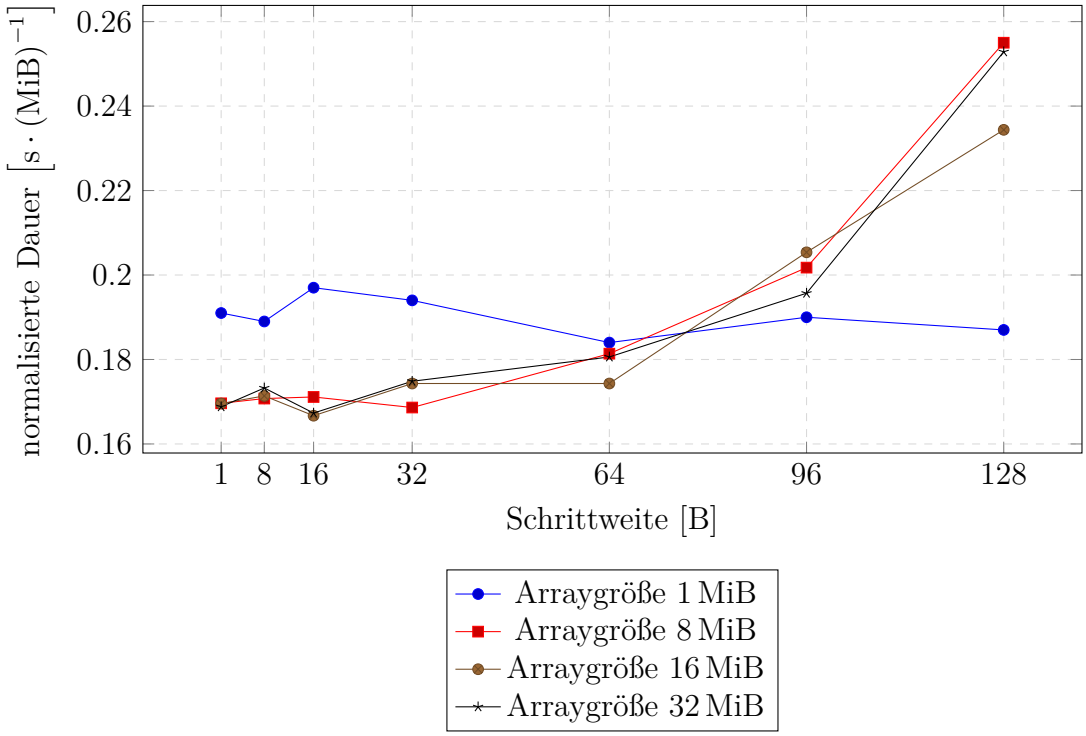

Abbildung 2.6: Iterationsdauer pro MiB in Abhängigkeit von Schrittweite und Arraygröße

Quelle: Abb. selbst erstellt

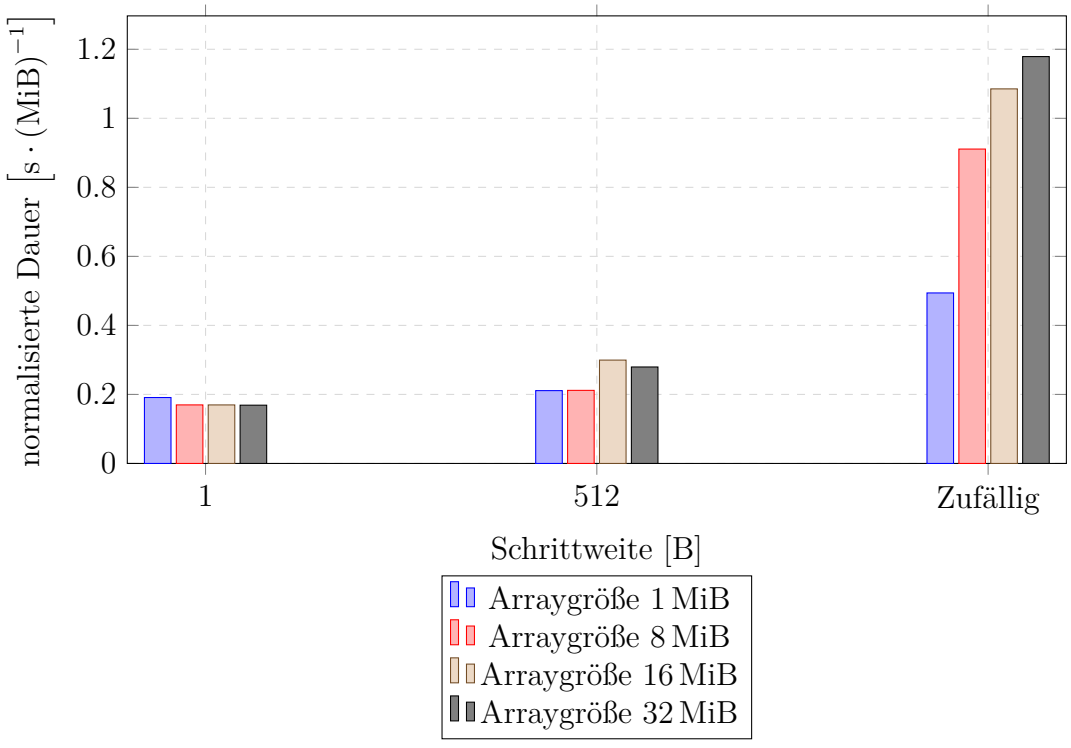

Abbildung 2.7: Vergleich zwischen Iteration mit fester Schrittweite und zufälligem Zugriffsmuster

Quelle: Abb. selbst erstellt

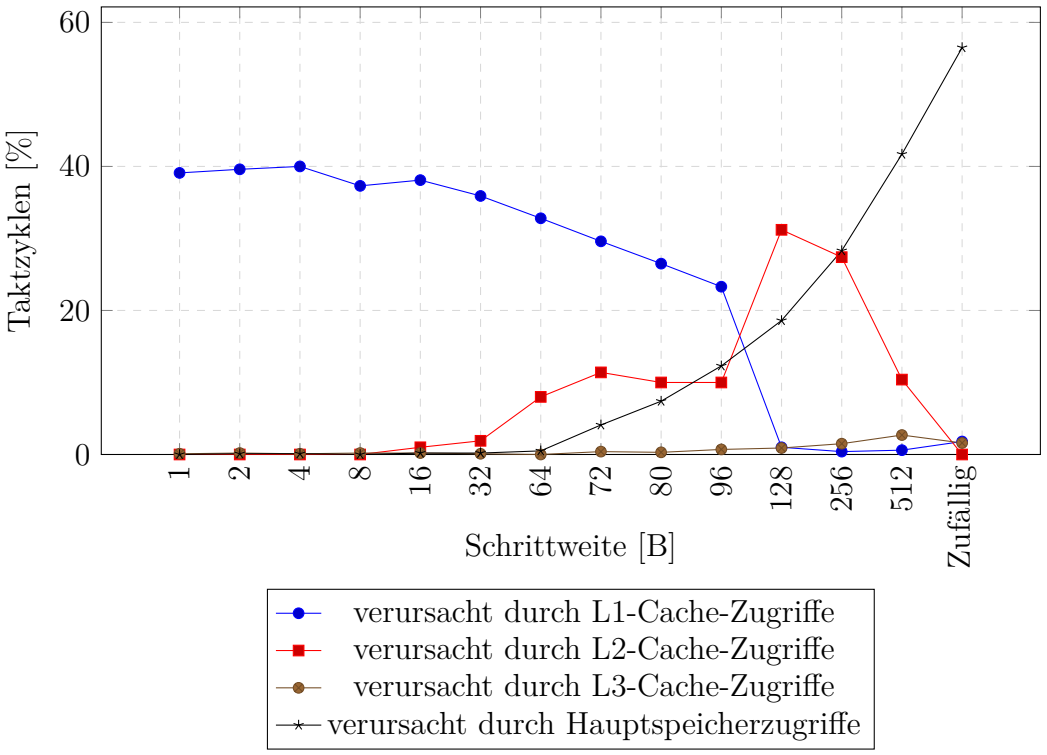

Abbildung 2.8: Auswirkungen der verschiedenen Cache-Hierarchiestufen auf die Speicherlatenz

Quelle: Abb. selbst erstellt

Anteil der verschiedenen Cache-Hierarchiestufen am Gesamtprozess

Mit spezieller Profiler-Software wie z. B. Intel VTune[33] kann u. a. gemessen werden, welchen Anteil die jeweiligen Hierarchiestufen des Caches sowie der Arbeitsspeicher an der Latenz haben. Die in Abb. 2.8 gezeigten Messwerte entstanden durch die mehrfache Iteration über insgesamt 32 MiB Daten mit verschiedenen Schrittweiten sowie mit zufälligem Zugriffsmuster. Zu sehen ist, wie viel Prozent der Taktzyklen durch Speicherzugriffe der jeweiligen Cache-Hierarchiestufe beansprucht wurden. Zu beachten ist dabei, dass die Zugriffe mit steigender Hierarchiestufe langsamer werden[34]. Auch hier zeigt sich erneut, dass der Cache praktisch keinerlei Performancegewinn bringt, wenn die Speicherzugriffe in zufälliger Reihenfolge erfolgen.

2.5 Instruction-Caching

Wie in Abschnitt 2.2 erläutert findet die Technik des Cachings nicht nur bei den vom Programm verwendeten Daten Verwendung, sondern auch bei den Instruktionen, die das Programm selbst ausmachen. Konkret handelt es sich

[33] siehe `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html#gs.7eq5an`, aufgerufen am 04.08.2021

[34] Das belegt die in Abb. 2.6 zu sehenden Verschlechterungen der gemessenen Gesamtzeiten.

```
uint32_t digits10(uint64_t v) {
    uint32_t result = 0;
    do {
        ++result;
        v /= 10;
    } while (v);
    return result;
}
```

Listing 2.3: Funktion zur Berechnung der Anzahl von Dezimalziffern einer Zahl[36]

```
uint32_t digits10_4(uint32_t v) {
    auto result = uint32_t{ 1 };
    for (;;) {
        if (v < 10U) return result;
        if (v < 100U) return result + 1U;
        if (v < 1000U) return result + 2U;
        if (v < 10000U) return result + 3U;
        v /= 10000U;
        result += 4U;
    }
}
```

Listing 2.4: verbesserte Funktion zur Berechnung der Anzahl von Dezimalziffern einer Zahl[37]

dabei um den Maschinencode, der vom Compiler erzeugt wurde. Da das Caching der Instruktionen grundlegend genauso erfolgt wie das der Daten, soll an dieser Stelle nicht erneut auf die Details eingegangen werden, sondern stattdessen ein konkretes Beispiel gezeigt werden, welches die Auswirkungen des Instruction-Caches demonstriert. Das folgende Beispiel und insbesondere der dazugehörige Code sind dabei nah angelehnt an ein von Andrei Alexandrescu im Jahr 2015 gezeigtes Beispiel[35]. Untermauert wird dieses zusätzlich durch Zeitmessungen. Als Ausgangspunkt wählt Alexandrescu eine Funktion zur Berechnung der Anzahl der Dezimalziffern einer Zahl. Der entsprechende Code ist in Listing 2.3 zu sehen. Alexandrescu möchte demonstrieren, dass durch Vermeidung von Divisionen vorzeichenloser Ganzzahlen ein Performancegewinn möglich ist. Dazu präsentiert er den in Listing 2.4 gezeigten Ansatz.

Die Tatsache, dass Alexandrescu genau vier `if`-Anweisungen pro Iteration kodiert hat, begründet er in Zeitmessungen. Diese Zeitmessungen sollen nun nachvollzogen werden: Dazu werden verschiedene Varianten der in Listing 2.4 gezeigten Funktion erstellt, bei denen zwei bis neun `if`-Anweisungen pro Iteration verwendet werden. Es werden sowohl die Größe der vom Compiler erzeugten

[35] vgl. Alexandrescu, 2015
[36] aus Alexandrescu, 2015
[37] nach Alexandrescu, 2015

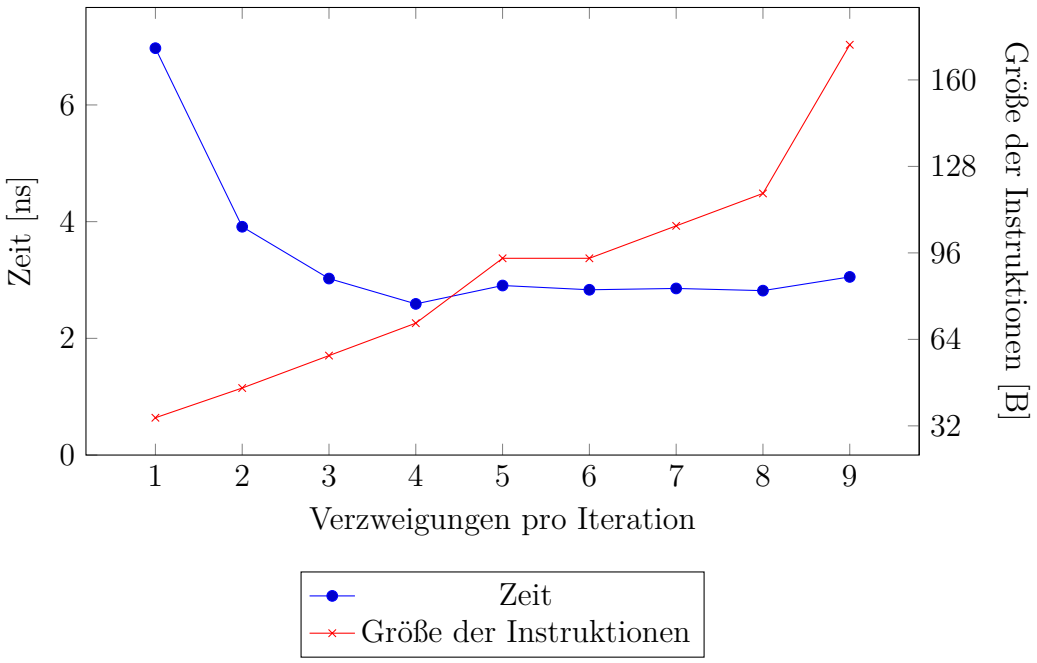

Abbildung 2.9: Zusammenhang zwischen Größe der Instruktionen und der benötigten Zeit

Quelle: Abb. selbst erstellt

Instruktionen in Byte ermittelt[38] als auch Zeitmessungen durchgeführt. Die Ergebnisse der Messungen sind in Abb. 2.9 dargestellt. Zu sehen ist, dass vier `if`-Anweisungen pro Iteration zu einer Größe des Maschinencodes führen, der in einer schnellstmöglichen Ausführung resultiert. Dies stützt die von Alexandrescu aufgestellte These, dass die Anzahl der Cache-Misses bei größerem Programmcode zunimmt und sich die Performance darüber hinaus – wenn auch nur wenig – verschlechtert. Anzumerken ist hierbei, dass die Ergebnisse stark vom verwendeten Compiler abhängig sind: Die durchgeführten Messungen erfolgten mit dem MSVC-Compiler von Microsoft. Mit dem GCC-Compiler sowie dem Clang-Compiler ließen sich die Ergebnisse nicht reproduzieren, was sich durch die unterschiedlichen Optimierungsstrategien der verschiedenen Compiler begründen lässt.

2.6 Branch-Prediction

Wie in Abschnitt 2.5 beschrieben unterliegt der Instruction-Cache prinzipiell denselben Mechanismen wie der Data-Cache[39]. Insbesondere schließt das auch

[38] Dazu wurde vom Compiler Maschinencode erzeugt, der mit den Assembler-Befehlen sowie dem tatsächlichen Programmcode annotiert in einer Textdatei abgelegt wurde.

[39] Es kann jedoch Unterschiede in den Details geben, z. B. dass auf manchen Prozessoren die kleinstmögliche Speichereinheit, die geladen werden kann, bei den Instruktionen *nicht* der Größe einer Cache-Line entspricht, sondern kleiner als diese ist (vgl. dazu Drepper, 2007, S. 58).

die Anwendung von Pre-Fetching ein. Im Gegensatz zu den Daten kann jedoch bei den Instruktionen nicht im Detail deren Speicherlayout bestimmt werden, da dieses vom Compiler festgelegt wird[40]. Zwar ist es vorteilhaft, dass Instruktionen zwischen Sprung-Anweisungen linear im Speicher liegen, jedoch sind es genau die Sprünge, die das Pre-Fetching erschweren, denn »the jump target might not be statically determined; and even if it is static the memory fetch might take a long time if it misses all caches.«[41]

2.6.1 Funktionsweise

Um sich diesem Problem entgegenzustellen, versuchen heutige Prozessoren, die Ergebnisse der Auswertungen von Bedingungen, die zu Verzweigungen (engl. *branches*) im Programmfluss führen können, vorherzusagen. Man spricht in diesem Zusammenhang von »Branch-Prediction«[42].

> »Once a prediction is made, the processor can speculatively execute instructions depending on the predicted outcome of the branch. If branch prediction rates are high enough to offset misprediction penalties, the processor will likely have a better overall performance. Without branch prediction, such a processor must stall whenever there are unresolved branch instructions.«[43]

Es existieren verschiedene Ansätze für die Branch-Prediction. An dieser Stelle soll beispielhaft die sog. »One-Level-Branch-Prediction« beschrieben werden. Trifft der Prozessor bei der Ausführung auf eine Verzweigung (*conditional jump instruction*), so werden die k niederwertigsten Bits der Speicheradresse der Sprunginstruktion als Index in eine Tabelle mit 2^k Einträgen genutzt. Jeder Eintrag der Tabelle hat eine Länge von n Bits. Führt die Auswertung der Sprungbedingung dazu, dass der Sprung durchgeführt wird, wird der entsprechende Tabelleneintrag inkrementiert. Ist das Gegenteil der Fall, wird er dekrementiert. Ein Überlauf wird dabei ausgeschlossen, indem eine Dekrementierung nur erfolgt, wenn der Tabelleneintrag größer als Null ist; eine Inkrementierung erfolgt respektive nur dann, wenn der Eintrag kleiner als $2^n - 1$ ist. Um das Eintreten

[40] Eine Ausnahme hiervon stellt die Programmierung in Assembler-Sprachen dar, bei der das Speicherlayout der Instruktionen sehr wohl exakt bestimmt werden kann.
[41] Drepper, 2007, S. 55 f.
[42] Gemeint ist hier »dynamic branch prediction« im Unterschied zu »static branch prediction«. Letztere wird vom Compiler durchgeführt, der durch Analyse des Programmcodes das Ergebnis von Sprüngen vorherzusagen versucht, um die erzeugten Instruktionen daraufhin zu optimieren, indem z. B. Anweisungsblöcke in eine andere als die ursprünglich kodierte Reihenfolge gebracht werden. Ersteres hingegen findet ausschließlich zur Laufzeit und gänzlich auf Ebene der Hardware, also des Prozessors statt.
[43] Lee et al., 2001

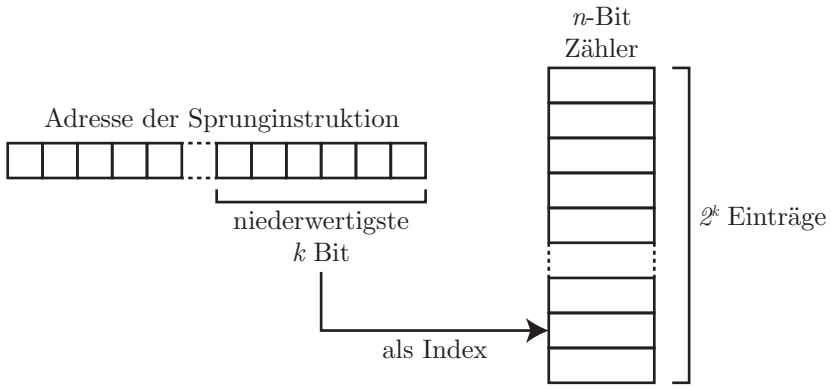

Abbildung 2.10: schematischer Aufbau eines One-Level-Branch-Predictors
Quelle: Abb. selbst erstellt, angelehnt an Lee et al., 2001

oder Nicht-Eintreten des Spungs vorherzusagen, wenn die entsprechende Anweisung *erneut* erreicht wird, wird der Tabelleneintrag ausgelesen: Ist dieser kleiner als 2^{n-1}, also kleiner oder gleich der Hälfte des maximal darstellbaren Werts, so wird vorausgesagt, dass der Sprung *nicht* erfolgt. Die entsprechenden Anweisungen werden ausgeführt, während die Sprungbedingung ausgewertet wird. In einem solchen Fall spricht man von »instruction-level parallelism«[44]. Stellt sich später heraus, dass der Sprung *doch* hätte stattfinden sollen, werden die Ergebnisse der spekulativ ausgeführten Berechnungen verworfen und stattdessen der korrekte Code-Pfad ausgeführt[45]. Abb. 2.10 verdeutlicht den schematischen Aufbau eines One-Level-Branch-Predictors.
Da es sich bei einem Branch-Predictor im Wesentlichen um einen assoziativen Container handelt, können sich diese auch in ihren Implementierungsarten ähneln. Beispielsweise kann die Zuordnung von Sprungadresse zu Tabelleneintrag auch durch eine Hash-Funktion umgesetzt werden. Durch komplexere Techniken wie z. B. Two-Level-Branch-Prediction können auch Korrelationen zwischen verschiedenen Sprunginstruktionen bzw. deren Auswertung berücksichtigt werden, um bessere Vorhersagen zu ermöglichen. Auch das Erkennen von Mustern ist möglich[46].

2.6.2 Zeitmessungen

Um die Auswirkungen der Vorhersagbarkeit von Sprüngen während der Programmausführung zu messen, wird ein Programm genutzt, welches eine der beiden Lösungen einer (lösbaren) quadratischen Gleichung bestimmt. *Welche* der beiden Lösungen gefragt ist, soll dabei vom Zufall abhängen. Durch das Generieren von Zufallszahlen wird zur Laufzeit entschieden, welchem Pfad die

[44] siehe auch Abschnitt 2.7
[45] vgl. Lee et al., 2001
[46] Diesbezüglich sei an dieser Stelle auf Lee et al., 2001 verwiesen.

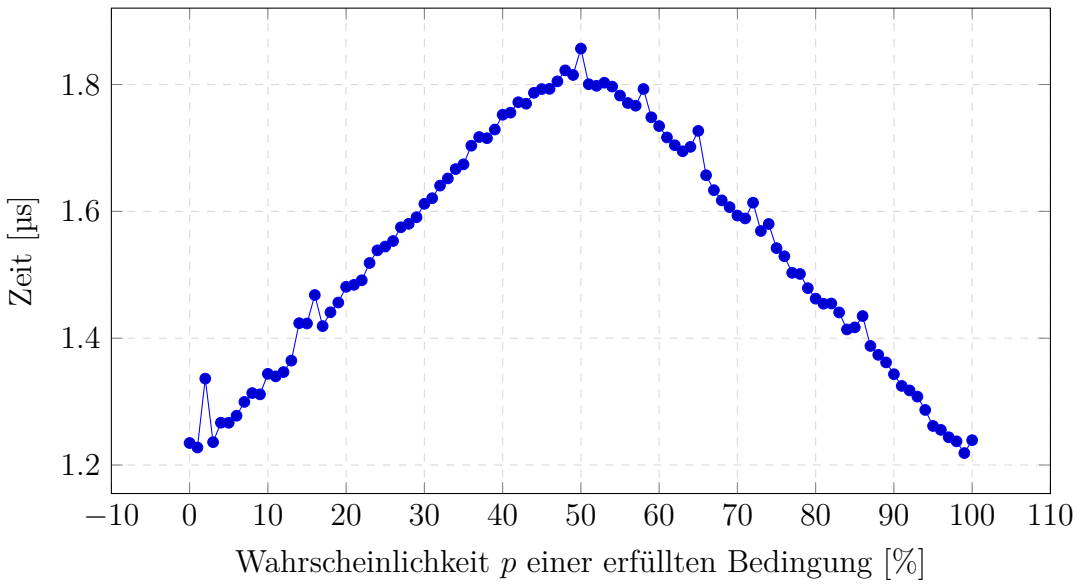

Abbildung 2.11: Zusammenhang zwischen der Vorhersagbarkeit von Sprüngen und der Laufzeit
Quelle: Abb. selbst erstellt

Programmausführung folgen soll. Gekoppelt wird die Entscheidung jeweils an eine vorgegebene Wahrscheinlichkeit. Damit ist gemeint, dass die Wahrscheinlichkeit, dass das Programm während eines Testlaufs dem ersten Pfad folgt, p beträgt (und der Wert von p vorgegeben werden kann). Respektive beträgt die Wahrscheinlichkeit q, dass dem zweiten Pfad zu folgen ist, $q = 1 - p$. Je kleiner die Differenz zwischen p und q ist, umso schlechter kann der Prozessor im Voraus abschätzen, welcher Code ausgeführt werden muss. Für $p = q = 0.5$ ist also die schlechtmöglichste Performance zu erwarten. Die entsprechenden Messergebnisse sind in Abb. 2.11 zu sehen und bestätigen diese Vermutung. Zu erkennen ist außerdem, dass die Unvorhersagbarkeit des Programmflusses im Vergleich zu sicherer Vorhersagbarkeit zu einer Laufzeitverschlechterung um knapp 50 % führen kann.

2.6.3 Branchless programming

Eine Möglichkeit, die negativen Effekte fehlgeschlagener Branch-Prediction zu verhindern, ist die Programmierung ohne konditionale Sprünge (engl. *branchless programming* oder *branch-free programming*). Oft lässt sich das z. B. realisieren, indem Wahrheitswerte als Zahlen interpretiert werden. Das folgende Beispiel stammt aus Johnson, 2021. Listing 2.5 zeigt drei verschiedene Implementierungen eines Algorithmus, der das Maximum zweier Zahlen bestimmt. Die Variante maxNaive ist dabei zwar leicht verständlich, verfügt jedoch über eine konditionale Verzweigung. Die beiden anderen Varianten verzichten darauf. Bei der Funktion

```
int maxNaive(int a, int b) {
    if (a > b) {
        return a;
    }
    return b;
}

int maxBranchless(int a, int b) {
    return (a > b) * a + (a <= b) * b;
}

int fastMax(int a, int b) {
    int diff = a - b;
    int dsgn = diff >> 31;
    return a - (diff & dsgn);
}
```

Listing 2.5: Vermeidung konditionaler Sprünge[47]

fastMax ist der Code ohne Verständnis der Zweierkomplementdarstellung von Ganzzahlen nicht zu verstehen. Ähnlich wie bei Listing 2.1 in Abschnitt 2.1.1 wird hier ein potenzieller Geschwindigkeitsvorteil mit schlechterer Lesbarkeit »erkauft«.

Um festzustellen, welche der drei Varianten die beste Performance bietet, werden Zeitmessungen durchgeführt. Bei den Messungen zeigt sich, dass hierbei große Unterschiede zwischen verschiedenen Compilern auftreten: Beim MSVC-Compiler ist die Funktion maxBranchless am schnellsten, beim Clang-Compiler sind maxNaive und maxBranchless gleich schnell, während beim GCC-Compiler maxNaive am besten abschneidet. Die Messergebnisse sind in Abb. 2.12 zu sehen. Hier zeigt sich deutlich das Potenzial, das Compiler-Optimierungen mit sich bringen, und dass nur durch Messungen sicher festgestellt werden kann, welche Optimierungen auf der Zielplattform tatsächlich Verbesserungen bringen. Zwar ist bei komplexeren Beispielen anzunehmen, dass die Compiler das Problem nicht in diesem Maße analysieren können, jedoch kann das nur durch Messungen gezeigt werden.

2.7 Parallelisierung

Prozessoren, welche die parallele Ausführung mehrerer Befehlsstränge erlauben, sind heute in den verschiedensten Gerätearten zum Standard geworden, u. a. sogar in Smartphones. Der vorliegende Abschnitt legt dar, wie diese Hardware ausgenutzt werden kann und welche Probleme sowie Limitierungen es dabei gibt.

[47] angelehnt an Johnson, 2021

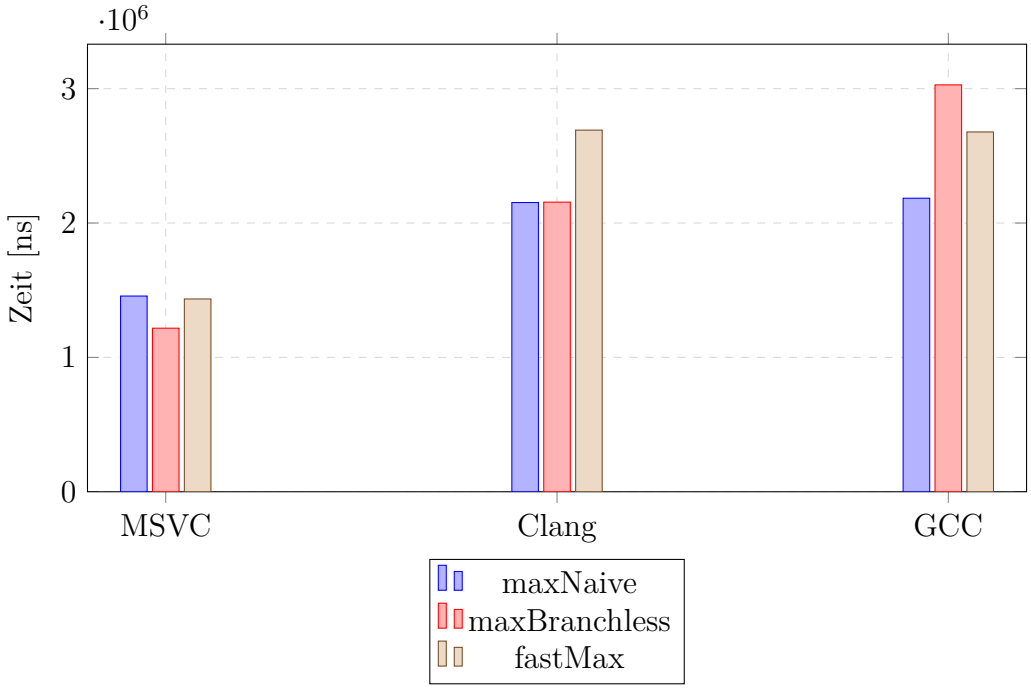

Abbildung 2.12: Vergleich verschiedener Algorithmen zum Branchless-Porgramming

Quelle: Abb. selbst erstellt

2.7.1 Motivation

Bereits 1965 sagte Gordon E. Moore voraus, dass sich die Anzahl der Transistoren pro Mikroprozessor ca. alle zwei Jahre verdoppeln würde[48]. Mit dieser Aussage sollte er bis heute Recht behalten. Abb. 2.13 unterstützt diese Aussage – zu beachten ist die logarithmische Skalierung. In derselben Abbildung ist jedoch auch zu erkennen, dass dagegen die Taktfrequenz von Mikroprozessoren diesem Trend mittlerweile nicht mehr folgt. Aufgrund physikalischer Begrenzungen äußert sich die erhöhte Leistungsfähigkeit von Prozessoren heutzutage immer mehr in der Erhöhung der Anzahl von Prozessorkernen anstatt in einer höheren Taktfrequenz.

> »[...] chip manufacturers have increasingly been favoring multicore designs with 2, 4, 16, or more processors on a single chip over better performance with a single core. Consequently, multicore desktop computers, and even multicore embedded devices, are now increasingly prevalent. The increased computing power of these machines comes not from running a single task faster but from running multiple tasks in parallel.«[49]

[48] vgl. Moore, 1965
[49] Williams, 2019, S. 8

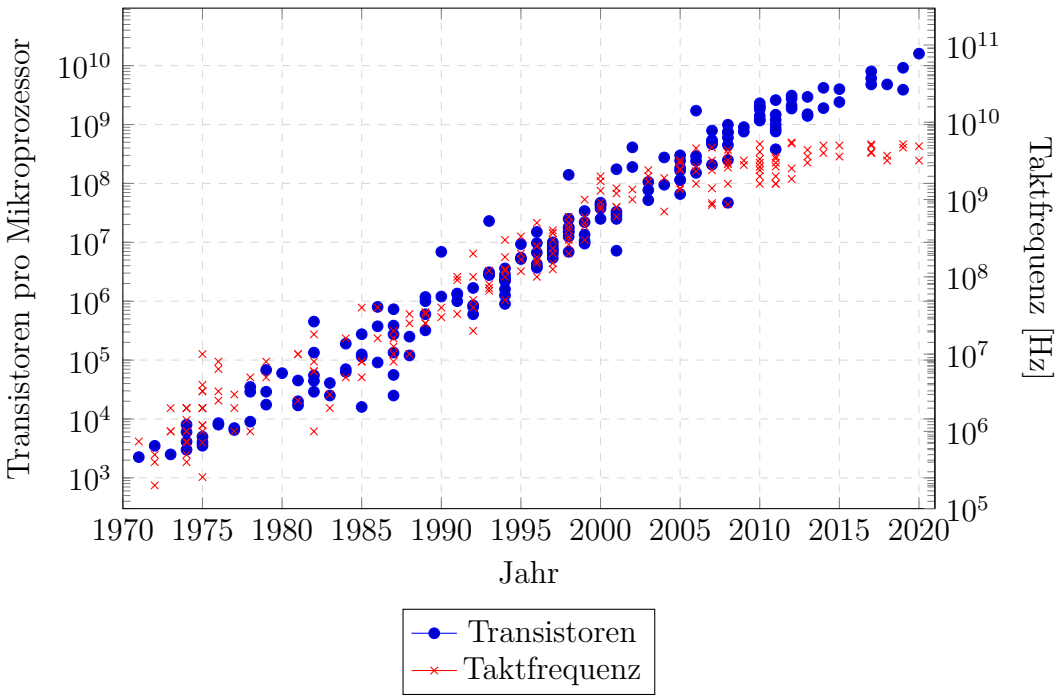

Abbildung 2.13: Anzahl der Transistoren und Taktfrequenzen von Mikroprozessoren in den Jahren 1970 bis 2020

Quelle: Abb. selbst erstellt, Daten von `https://en.wikipedia.org/wiki/Microprocessor_chronology`, aufgerufen am 06.08.2021

Während jegliche Software von einer gesteigerten Taktfrequenz unmittelbar profitieren kann, müssen Programme speziell angepasst werden, um von der vergrößerten Anzahl von Prozessorkernen Gebrauch machen zu können. Aufgrund der oben geschilderten Entwicklung wird diese Art der Anpassung bei Performance-kritischer Software immer wichtiger. Da eine umfassende Beschreibung von Techniken zur Parallelisierung von Programmcode den Rahmen dieser Arbeit sprengen würde, soll an dieser Stelle im Wesentlichen auf eine Auswahl von Fallstricken eingegangen werden, die eine effiziente parallele Ausführung verhindern können, aber nicht offensichtlich sind. Grundlegende Begriffe wie Threads und Mutexes werden an dieser Stelle als bekannt vorausgesetzt.

2.7.2 Amdahlsches Gesetz

Ein vereinfachtes Modell, welches die Skalierbarkeit von Software beschreiben soll, stellt das nach Gene Amdahl benannte »Amdahlsche Gesetz« dar. Bei diesem wird angenommen, dass ein Programm sowohl aus *seriellen* als auch aus *parallelen* Programmteilen besteht. In den parallelen Abschnitten kann Gebrauch von mehreren Prozessorkernen gemacht werden, während die seriellen Abschnitte nicht von diesen profitieren. Als Beispiel wäre hier ein Programm zu nennen, welches mehrere Threads erzeugt, die sich die Verarbeitung einer

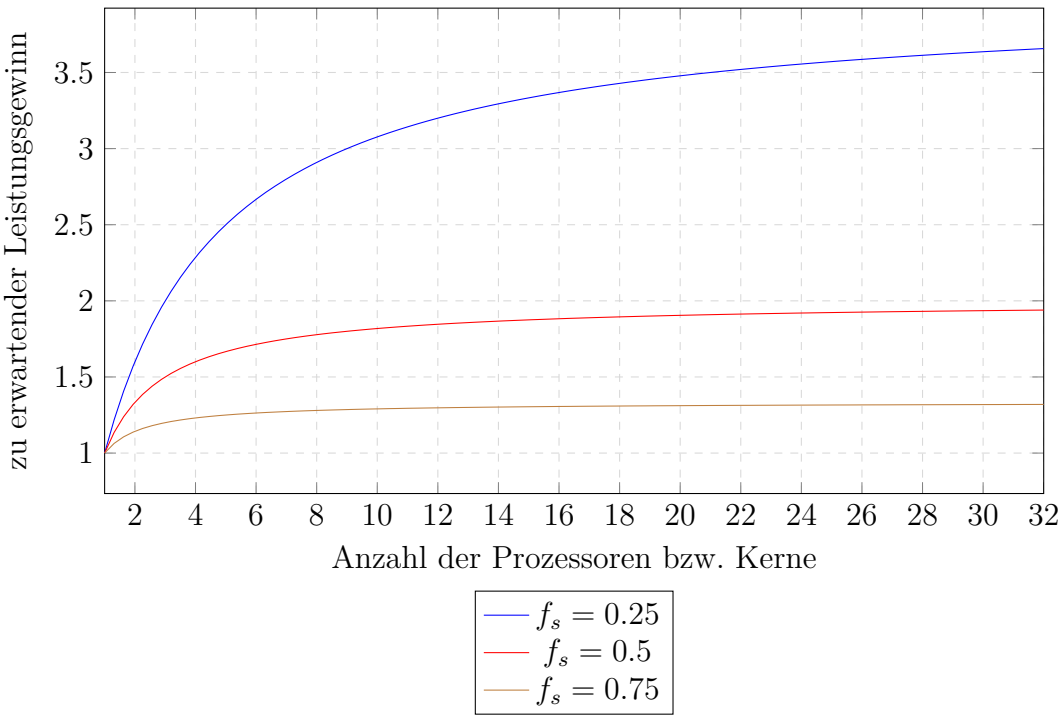

Abbildung 2.14: das Amdahlsche Gesetz
Quelle: Abb. selbst erstellt

großen Datenmenge gleichmäßig teilen. Haben alle Threads ihre jeweiligen Berechnungen abgeschlossen, können deren Ergebnisse kumuliert werden, um ein Gesamtergebnis zu erhalten. Das Erzeugen der Threads sowie das Akumulieren der Teilergebnisse stellen hierbei die seriellen Programmteile dar, während die eigentlichen Berechnungen parallel ablaufen können. Seien der relative Anteil der seriellen Programmteile am Gesamtprogramm bezeichnet mit f_s und die Anzahl der Prozessoren bzw. Prozessorkerne mit N. Dann besagt das Amdahlsche Gesetz, dass der zu erwartende Leistungsgewinn abgeschätzt werden kann durch[50]

$$P = \frac{1}{f_s + \frac{1-f_s}{N}}.$$

In Abb. 2.14 sind drei Plots dieser Gleichung zu sehen. Von großem Interesse ist vor allem die maximal erreichbare Leistungssteigerung, also der Grenzwert für $f_s \to 0$. Es ist

$$\lim_{f_s \to 0} \frac{1}{f_s + \frac{1-f_s}{N}} = N.$$

[50] vgl. Williams, 2019, S. 277 f.

Verfügt ein Programm also über keinerlei serielle Bestandteile, sondern lediglich über parallele, so skaliert es unmittelbar mit der Anzahl der verfügbaren Prozessoren. Jedoch kann ein Programm nicht ausschließlich aus parallelen Programmabschnitten bestehen, da die Verwaltung von Threads[51] ein nicht gänzlich parallelisierbarer Vorgang ist. Angenommen, der Anteil der seriellen Programmteile f_s sei konstant, aber es stünden unbegrenzt viele Prozessoren bzw. Prozessorkerne zur Verfügung ($N \to \infty$), dann gilt

$$\lim_{N\to\infty} \frac{1}{f_s + \frac{1-f_s}{N}} = \frac{1}{f_s}.$$

Daraus folgt, dass die Leistungsverbesserung durch die seriellen Programmteile begrenzt wird. Besteht beispielsweise nur ein Prozent des Programms aus seriellen Anteilen, ist die Leistungssteigerung dennoch begrenzt auf Faktor $\frac{1}{0.01} = 100$, selbst bei einer unbegrenzten Anzahl von Prozessoren.
Aus dem Amdahlschen Gesetz folgt in der Praxis also, dass

- die Menge der seriellen Programmteile möglichst gering gehalten werden sollte und
- ein Programm nicht beliebig durch die Erhöhung der Anzahl der Prozessoren beschleunigt werden kann.

2.7.3 Cache-Ping-Pong

Beschreibung

Wie in Abschnitt 2.2 beschrieben laufen Speicherzugriffe bei heutigen Prozessoren i. d. R. immer über den Cache[52]. Üblicherweise verfügt dabei jeder Prozessorkern jeweils über eigene Level-1- und Level-2-Caches. Wenn mehrere Threads auf denselben Speicher zugreifen und diesen modifizieren, müssen die Kopien dieses Speichers, die in den verschiedenen Caches liegen, synchron gehalten werden. Solche Speicherzugriffe sind oft nicht unbedingt sofort ersichtlich: Beispielsweise handelt es sich auch bei üblichen Synchronisationsprimitiven wie atomaren Variablen (engl. *atomics*) oder Mutexes um sog. »shared memory«, also Speicher, auf den mehrere Threads zugreifen können. Beispielsweise kann ein einfacher Mutex mithilfe eines sog. »Flags«, also eines boolschen Werts implementiert werden, auf den atomar zugegriffen wird. Ein Thread, der den Mutex sperren möchte, liest in einer Schleife den aktuellen Wert aus und schreibt als neuen Wahrheitswert »wahr«. Dabei handelt es sich um eine

[51] und/oder Prozessen
[52] bzw. über die Cache-Hierarchie

```
1  using Iterator = std::vector<int>::iterator;
2  static std::atomic<std::size_t> gSum{ 0 };
3
4  static void accumulate1(Iterator begin, Iterator end) {
5      for (auto it = begin; it != end; ++it)
6          gSum += *it;
7  }
8
9  static void accumulate2(Iterator begin, Iterator end) {
10     std::size_t localAccumulator{ 0 };
11     for (auto it = begin; it != end; ++it)
12         localAccumulator += *it;
13     gSum += localAccumulator;
14 }
```

Listing 2.6: Demonstration von Cache-Ping-Pong

Operation, die von aktuellen Prozessoren atomar unterstützt wird[53]. Wenn der vorherige Wert »falsch« war, ist klar, dass der aktuelle Thread den Wert geändert hat. Die Schleife kann beendet werden. War er jedoch vorher bereits »wahr«, ist der Mutex bereits von einem anderen Thread gesperrt worden und die Schleife muss fortgesetzt werden.
Wenn mehrere Threads immer wieder im Wechsel modifizierend auf denselben Speicher zugreifen und daher fortlaufend die Caches invalidiert werden, spricht man von sog. »Cache-Ping-Pong«.

Zeitmessungen

Um die Effekte von Cache-Ping-Pong zu messen, wird ein Programm eingesetzt, welches die Summe von Zahlen berechnet, die sich in einem Array befinden. Die Ergebnisse werden in einer globalen Variable gSum abgelegt. Auf diese Variable wird atomar zugegriffen. Listing 2.6 zeigt den relevanten Programmteil. Während Funktion accumulate1() in jedem Iterationsschritt die aktuelle Zahl zur Gesamtsumme addiert, verfügt Funktion accumulate2() über eine lokale Hilfsvariable (Zeile 10) für das Akkumulieren. Erst am Ende der gesamten Berechnung wird das Ergebnis zur globalen Variable addiert.
Um die beiden Funktionen miteinander zu vergleichen, werden zwei Benchmarks erstellt, die jeweils ein Array mit Zufallszahlen sowie zwei Threads erzeugen. Die Threads summieren jeweils eine Hälfte der Zufallszahlen auf. Führen beide Threads die Funktion accumulate1() aus, entsteht eine ausgeprägte »Konkurrenzsituation« (engl. *contention*) in Bezug auf die globale Variable gSum, da beide Threads fortlaufend auf diese zugreifen und sie modifizieren. Die jeweiligen Caches müssen dabei immer wieder synchronisiert werden. Führen beide

[53] Im C++-Standard existiert beispielsweise dafür die Funktion test_and_set() in der Klasse std::atomic_flag, welche auf die entsprechenden Maschinensprachen-Instruktionen abgebildet wird.

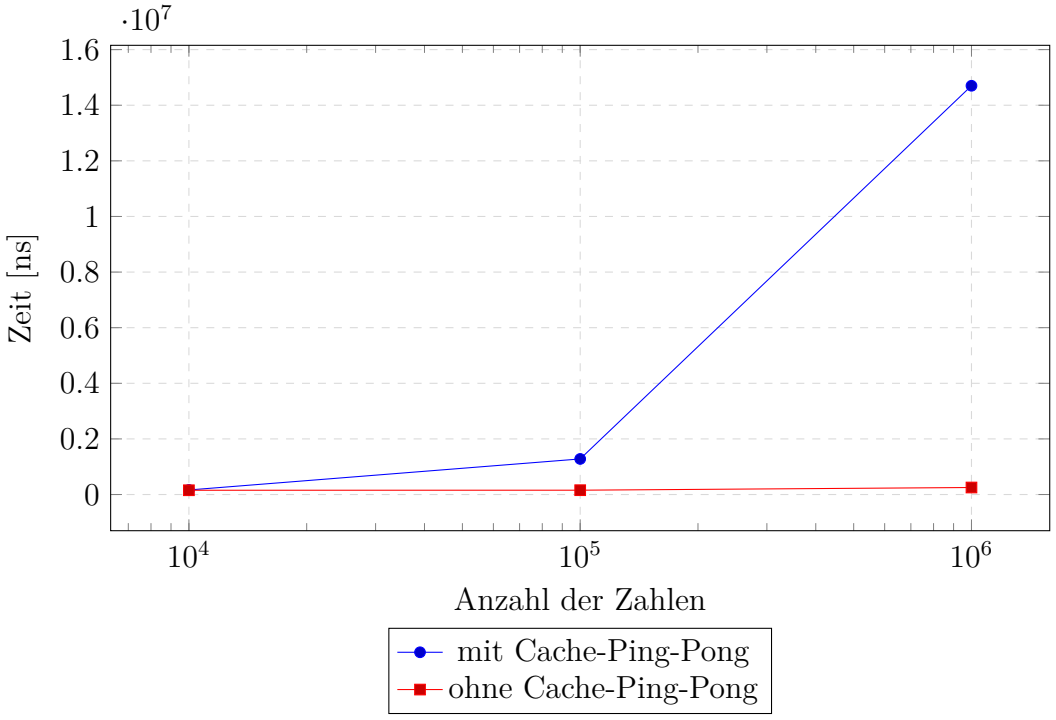

Abbildung 2.15: Auswirkungen von Cache-Ping-Pong
Quelle: Abb. selbst erstellt

Threads jedoch die Funktion accumulate2() aus, so gibt es insgesamt lediglich zwei Zugriffe auf die globale Variable, nämlich am Ende der Berechnungen. Abb. 2.15 zeigt die drastischen Auswirkungen dieser Änderung. Umfasst das Array eine Million Elemente, so verschlechtert sich das Laufzeitverhalten um Faktor 60.

2.7.4 False-Sharing

Beim sog. »False-Sharing« handelt es sich um ein Problem, welches dem Cache-Ping-Pong stark ähnelt: Von False-Sharing spricht man, wenn mehrere Threads modifizierend auf paarweise verschiedene Speicherbereiche zugreifen, die sich jedoch in derselben Cache-Line befinden. Obwohl es sich also *nicht* um shared memory handelt, können dennoch Performanceverschlechterungen auftreten, die sich auf den Mechanismus des Cachings zurückführen lassen. Ob es zu False-Sharing kommt, hängt also unmittelbar vom Speicherlayout der Variablen ab, auf welche durch die existierenden Threads zugegriffen wird[54]. Listing 2.7 enthält Ausschnitte aus einem Programm, das die Auswirkungen von False-Sharing demonstrieren soll.
Erneut sollen hierbei Elemente eines Arrays aufsummiert werden. Es werden mehrere Threads erstellt, die jeweils die Funktion accumulate ausführen. Diese

[54] vgl. Williams, 2019, S. 264 f.

```
1  struct Accumulator {
2      std::uint64_t value{ 0 };
3  };
4
5  struct alignas(64) AlignedAccumulator {
6      std::uint64_t value{ 0 };
7  };
8
9  void accumulate(const std::vector<int>& numbers,
10                  std::uint64_t& accumulator) {
11     for (const auto number : numbers)
12         accumulator += number;
13 }
14 // ...
15 template<typename AccumulatorType>
16 void accumulateMultithreaded(const std::vector<int>& numbers) {
17     AccumulatorType accumulators[numThreads];
18     std::thread threads[numThreads];
19     for (unsigned i{ 0U }; i < numThreads; ++i)
20         threads[i] = std::thread{ accumulate,
21                                   std::cref(numbers),
22                                   std::ref(accumulators[i].value)
23                                 };
24     for (auto& thread : threads)
25         thread.join();
26 }
27
28 int main() {
29     const auto numbers = createRandomNumbers(size);
30     accumulateMultithreaded<Accumulator>(numbers);
31     accumulateMultithreaded<AlignedAccumulator>(numbers);
32 }
```

Listing 2.7: Demonstration von False-Sharing

Funktion nutzt als Akkumulationsvariable eine Variable, die ihr in Form einer Referenz als Parameter übergeben wird (Zeilen 20 bis 23 bzw. Zeilen 9 bis 13). Diese Akkumulationsvariablen befinden sich in einem Array aus einem Wrapper-Objekt (Zeile 17). Durch den Typparameter AccumulatorType (Zeile 15) der generischen Funktion accumulateMultithreaded lässt sich dieser Wrapper-Typ austauschen: Das Funktions-Template kann wahlweise mit dem Typ Accumulator oder AlignedAccumulator instanziiert werden. Diese beiden Typen unterscheiden sich lediglich in ihrer »Speicherausrichtung« (engl. *alignment*):

- Objekte des Typs Accumulator sind standardmäßig, also nach dem größten enthaltenen Element ausgerichtet. Da das größte (und einzige) Element des Objekts eine Größe von 64 Bit bzw. 8 Byte hat, wird das Gesamtobjekt im Speicher so ausgerichtet, dass seine Startadresse ohne Rest durch 8 teilbar ist. Das bedeutet, dass z. B. ein Array, das genau acht Objekte dieses Typs enthält, eine Gesamtgröße von $8 \cdot 8\,\text{Byte} = 64\,\text{Byte}$ hat.

- Im Falle des Typs AlignedAccumulator wird dagegen mit Hilfe des alignas-

Schlüsselworts ein Alignment von 64 Byte vorgegeben. Das hat zur Folge, dass der Compiler das Objekt künstlich vergrößert, indem dieses »aufgefüllt« wird, bis es eine Gesamtgröße von 64 Byte hat. Man spricht in diesem Zusammenhang von »Padding«. Ein Array von acht Objekten dieses Typs hätte damit eine Größe von $8 \cdot 64\,\text{Byte} = 512\,\text{Byte}$. Es ist dabei jedoch sichergestellt, dass jedes Objekt dieses Typs in einer eigenen Cache-Line liegt.

In den Zeilen 32 und 33 werden die beiden Varianten der Funktion sodann aufgerufen.

Zeitmessungen

Als Grundlage für die Zeitmessungen dient das in Listing 2.7 gezeigte Programmfragment. Nicht zu sehen ist dort der für die verwendete Benchmark-Bibliothek relevante Quelltext. Für den Gesamtquelltext sei auf den Anhang verwiesen.
Zu erwarten ist, dass unter Verwendung des AlignedAccumulator-Datentyps eine Performanceverbesserung möglich ist. Diese wird durch einen erhöhten Speicherverbrauch »erkauft«. Abb. 2.16 zeigt die aufgezeichneten Messergebnisse für acht Threads. Die Ergebnisse bestätigen die Erwartung. Die Unterschiede sind hierbei zwar nicht so drastisch wie bei der Zeitmessung zum Cache-Ping-Pong, da geteilter Speicher nochmals höhere Anforderungen an die Synchronisation stellt, aber dennoch ist bei größer werdender Elementanzahl ein deutlicher Unterschied festzustellen.
Für die Praxis ist zu schlussfolgern, dass

- beim Multithreading auch dann Leistungsverschlechterungen aufgrund von impliziter Synchronisation auftreten können, obwohl die verschiedenen Ausführungsstränge *nicht* modifizierend auf gemeinsamen Speicher zugreifen und dass
- derartige Probleme durch Veränderung des verwendeten Speicherlayouts bzw. des Alignments vermieden werden können.

2.8 Softwareentwicklung im Kontext des Betriebssystems

Allen aktuellen Spieleplattformen ist gemein, dass auf ihnen ein Betriebssystem vorhanden ist. Spiele werden also nicht »für eine Plattform« entwickelt, sondern für ein Betriebssystem, welches auf der entsprechenden Plattform zur Verfügung steht. Dieses stellt eine Vermittlungsschicht zwischen der Soft- und Hardware dar. Es kontrolliert die Speicherverwaltung und abstrahiert die zu Grunde liegende Hardware, indem es eine einheitliche Programmierschnittstelle (engl.

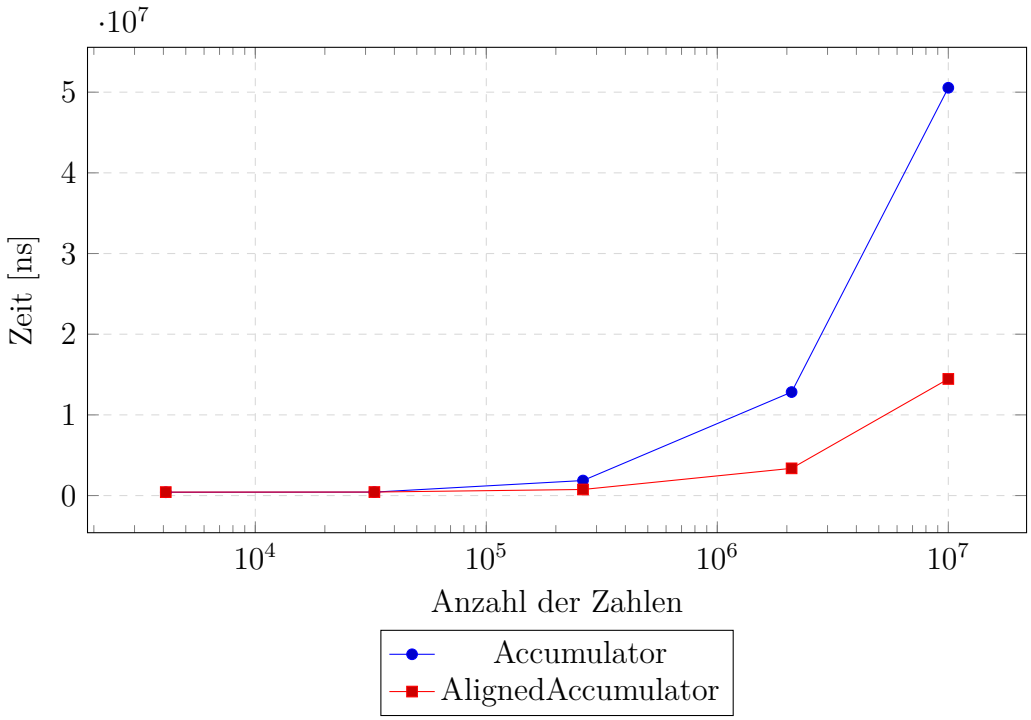

Abbildung 2.16: Auswirkungen von Speicheranordnung und False-Sharing
Quelle: Abb. selbst erstellt

application programming interface, kurz API) zur Verfügung stellt. Das hat zur Folge, dass manche Hardware-Funktionalitäten, welche von der Software in Anspruch genommen werden wollen, gewissen Restriktionen unterliegen, die vom Betriebssystem forciert werden. Unter anderem betrifft das Funktionen

- zum Ansprechen von Hardwaregeräten, insbesondere Geräte zur Ein- und Ausgabe,
- zum Zugriff auf das Dateisystem,
- zur Verwaltung von Prozessen sowie
- zur Speicherverwaltung.

Betriebssysteme stellen i. d. R. eine API zur Verfügung, um Benutzerprogrammen diese Funktionalitäten zur Verfügung zu stellen. Aufrufe solcher Funktionen werden »Systemaufrufe« (engl. *system call* oder *kernel call*) genannt. Ein Systemaufruf »involves a context switch into protected mode. Such context switches can cost upwards of 1000 clock cycles.«[55] Systemaufrufe sollten daher, sofern möglich, vermieden werden. Auf keinen Fall sollten sie in Hot Code[56] Verwendung finden, um die Performance nicht zu sehr zu beeinträchtigen.

[55] Gregory, 2019, S. 267
[56] siehe Abschnitt 2.1.4

2.8.1 Dynamische Speicherallokationen

Während sich Dateisystemzugriffe relativ leicht zu Beginn eines Videospiels unterbringen lassen – z. B. während eines Ladebildschirms –, stellen vor allem Systemaufrufe bezüglich der Speicherverwaltung ein schwierigeres Problem dar, denn »a program's memory requirements are often not fully known at compile time. A program usually needs to allocate additional memory *dynamically*.«[57] Der Speicher, der für diesen Zweck vom Betriebssystem zur Verfügung gestellt wird, wird »Heap-Speicher« (engl. *heap memory* oder *free store*) genannt. Im Gegensatz zum »Stack-Speicher« (engl. *stack memory*), dessen Größe bereits während der Kompilierung und dem Linken des Programms feststeht, kann die Größe des genutzten Heap-Speichers während der Laufzeit variieren. Als Grundregel formuliert Gregory:

> »Keep heap allocations to a minimum, and never allocate from the heap within a tight loop.«[58]

Es existieren verschiedene Techniken, um diese Regel zu erfüllen. Praktisch alle davon basieren auf der Idee, *im Voraus* bereits eine feste und ausreichend große Speichermenge zu allokieren, die dann innerhalb des Hot Codes dynamisch genutzt werden kann. Die Techniken unterscheiden sich im Wesentlichen darin, *wie* der allokierte Speicher verwaltet wird. Gregory nennt die folgende, für Spiele-Engines typische Auswahl von Ansätzen[59]:

- Stack-Based Allocators: Der allokierte Speicher wird wie ein Stack verwaltet. Das Freigeben des Speichers entspricht dem Zurücksetzen des Stack-Zeigers. Es werden hierbei keine einzelnen Speicherbereiche freigegeben, sondern allenfalls der gesamte Speicher. Dadurch kann keine Fragmentierung auftreten, obwohl Speicherbereiche unterschiedlicher Größe angefordert werden können.

- Double-Ended Stack Allocators: Innerhalb des allokierten Speichers werden zwei Stacks verwaltet, von denen einer »von unten nach oben« und der andere »von oben nach unten« wächst. So kann der Speicher nach Bedarf dynamisch auf zwei verschiedene Systeme aufgeteilt werden.

- Pool Allocators: Speicherbereiche identischer Größe werden verwaltet und können auch wieder freigegeben werden. Aufgrund der identischen Größe kann zwar keine Fragmentierung auftreten, jedoch kann es vorkommen, dass mehr Speicher angefordert werden muss als tatsächlich genutzt wird,

[57] Gregory, 2019, S. 155
[58] Gregory, 2019, S. 427
[59] vgl. Gregory, 2019, S. 427 ff.

da die Größe der verwalteten Speichereinheiten nicht zwangsläufig den Anforderungen des Programms entspricht. Um diesem Problem entgegen zu wirken, können mehrere Pool Allocators eingesetzt werden, die Einheiten verschiedener Größe verwalten.

2.8.2 Zeitmessungen

Um die Auswirkungen von Heap-Allocations auf die Performance zu messen, werden zwei Container der C++-Standardbibliothek miteinander verglichen: `std::vector`, ein dynamisch wachsendes Array, und `std::list`, eine doppelt verkettete Liste. Standardmäßig verfolgt die vorhandene Implementierung von `std::vector` eine exponentielle Wachstumsstrategie. Das bedeutet, dass immer dann, wenn der vorhandene Speicherplatz nicht ausreicht, ein Vielfaches – z. B. das Doppelte – des bisher vorhandenen Speicherplatzes allokiert wird und die vorhandenen Elemente in den neuen Speicher kopiert werden[60]. Mit wachsender Containergröße nimmt also die Häufigkeit von Allokationen ab. Es ist jedoch auch möglich, bereits im Vorfeld Speicherplatz einer gewünschten Größe zu reservieren, sodass nur einmalig eine Allokation stattfinden muss. Die Implementierung der verketteten Liste dagegen allokiert für jedes einzufügende Element neuen Heap-Speicher. Als Vorteil der verketteten Liste ist jedoch zu nennen, dass bei einer Vergrößerung der Datenstruktur keine Kopien notwendig sind. Es wurde gemessen, wie viele Iterationen in einer vorgegebenen Zeit durchgeführt werden können, wobei in jeder Iteration eine zufällige Anzahl von Elementen, maximal jedoch 2000, in einen `std::vector` bzw. in eine `std::list` eingefügt werden. Dies soll ein typisches Verhalten von Spiele-Engines simulieren, denn:

> »Virtually all game engines allocate at least some temporary data during the game loop. This data is either discarded at the end of each iteration of the loop or used on the next frame and then discarded.«[61]

Die Ergebnisse der Messungen sind in Abb. 2.17 zu sehen. Sie zeigen deutlich, dass die Auswirkungen der Speicherallokationen wesentlich schwerwiegender sind als die der zusätzlichen Kopien im Fall von `std::vector`. Es sollte dabei jedoch auch beachtet werden, dass in beiden Tests zu `std::vector` jeweils nicht der gesamte allokierte Speicher auch genutzt wird. Der tatsächlich genutzte Anteil hängt stark vom konkreten Anwendungsfall und der gewählten Allokationsstrategie ab.

[60] Unterstützt der im Container gespeicherte Datentyp »move semantics« (ggf. ab C++ 11), kann stattdessen ein *Move* stattfinden.

[61] Gregory, 2019, S. 434

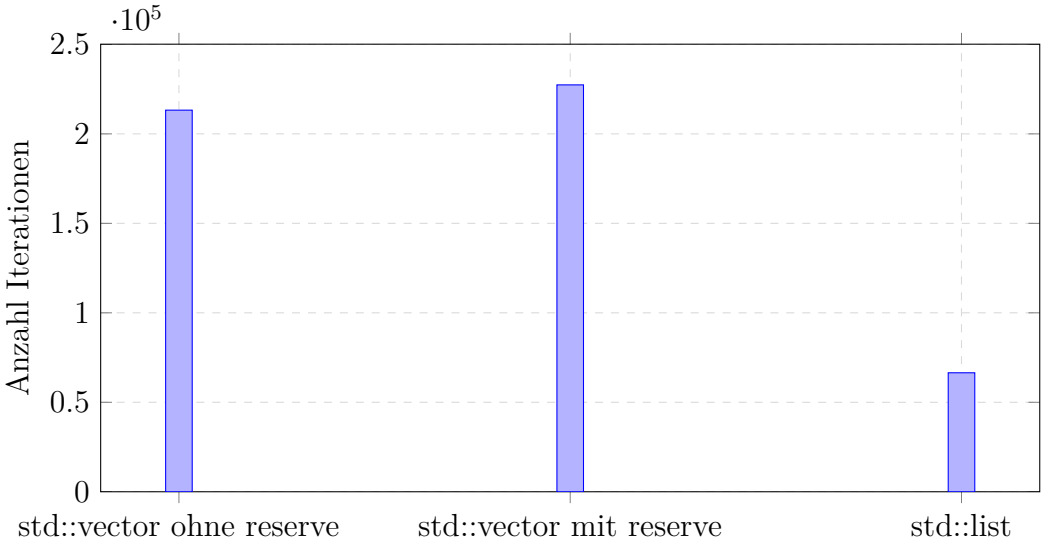

Abbildung 2.17: Auswirkungen von Heap-Allocations auf die Laufzeit
Quelle: Abb. selbst erstellt

2.9 Einfluss des verwendeten Programmierparadigmas

Das verwendete Programmierparadigma kann einen direkten Einfluss darauf haben, ob die bisher vorgestellten Optimierungstechniken angewendet werden können oder nicht. Viele Folgerungen aus den durchgeführten Zeitmessungen und Problemanalysen lassen sich beispielsweise nur dann konsequent umsetzen, wenn es möglich ist, das Speicherlayout der vom Programm verwendeten Daten genau festzulegen. Manche Paradigmen und vor allem die Programmiersprachen, in denen sie üblicherweise Anwendung finden, lassen das jedoch nicht zu. Die meisten Videospiele werden mit Hilfe imperativer Programmiersprachen realisiert. Innerhalb dieser Gruppe findet die Implementierung meistens mit einem der folgenden Programmierparadigmen statt:

- prozedurale Programmierung (z. B. in C, Rust, Python)
- objektorientierte Programmierung (z. B. in Java, C#, C++)
- funktionale Programmierung (z. B. in Haskell)

Es können auch Programmiersprachen verwendet werden, die mehrere Paradigmen unterstützen, z. B. C++ oder Rust. In ursprünglich rein objektorientierten Sprachen stehen zunehmend funktionale Konzepte zur Verfügung (z. B. LINQ in C# und Streams in Java). Vor allem rein objektorientierte Sprachen bringen mehrere Probleme mit sich:

- Die meisten dieser Sprachen arbeiten im Wesentlichen mit Referenzen, was zur Folge hat, dass das Speicherlayout von Daten nur bedingt beeinflusst werden kann. Legt man beispielsweise in Java ein Array eines

selbst definierten Datentyps an, so liegen die Daten nachfolgend nicht hintereinander im Speicher, sondern das Array selbst beinhaltet lediglich Referenzen auf die eigentlichen Daten, welche potenziell »verstreut« im Speicher sein können. Das hat zur Folge, dass sich sämtliche Optimierungen bezüglich des Daten-Caches, die in den vorherigen Abschnitten beschrieben wurden, hier nicht umsetzen lassen.

- Objektorientierte Sprachen umfassen i. d. R. einen Garbage Collector. *Wann* dieser aktiv wird, lässt sich ggf. zwar beeinflussen, jedoch nicht gänzlich kontrollieren. Bei Performance-kritischem Code ist es jedoch von großer Wichtigkeit, genau beeinflussen zu können, wann und in welchem Maße das Speichermanagement erfolgt.

Als Sprache der Wahl hat sich hauptsächlich C++ durchgesetzt, »the language of choice among most modern game developers«[62]. C++ ist eine Programmiersprache, die mehrere Programmierparadigmen unterstützt, allen voran die drei oben genannten. Darüber hinaus lassen sich in C++ Details wie das Speicher-Alignment festlegen, sodass das Speicherlayout exakt definiert werden kann. Ein weiterer wichtiger Punkt ist, dass es sich bei C++ um eine in Maschinencode übersetzte Programmiersprache handelt. Diese Eigenschaft ist zwar weitestgehend unabhängig vom Programmierparadigma, jedoch handelt es sich um ein wichtiges Sprachmerkmal in Bezug auf die Performance: Sprachen, die interpretiert, JIT-kompiliert[63] oder in einer virtuellen Maschine ausgeführt werden, weisen prinzipbedingt schlechtere Laufzeiteigenschaften als in Maschinencode kompilierte Sprachen auf.
Wie beschrieben kann bei der Verwendung von C++ auf verschiedene Programmierparadigmen zurückgegriffen werden. Manche Funktionalitäten der Sprache werden jedoch bewusst vermieden. Vererbung unter Verwendung virtueller Funktionen[64] erzeugt beispielsweise implizit eine sog. »V-Table«, also eine Tabelle, die einem konkreten Datentyp die entsprechenden Funktionszeiger der virtuellen Funktionen zuordnet. Die Größe selbst definierter Datentypen ändert sich somit, was in einem unerwünschten Speicherlayout resultieren kann. Hinzu kommt die zusätzliche Indirektion, die beim Aufrufen der virtuellen Funktionen notwendig ist.
Dennoch ist die Verwendung des `class`-Schlüsselworts zur Definition eigener Datentypen üblich. Beispielsweise werden Klassen oft genutzt, um das sog.

[62] Gregory, 2019, S. 4
[63] engl. *just-in-time compilation* beschreibt die abschnittsweise Kompilierung von Programmen zur Laufzeit
[64] Im Rahmen von C++ ist der Begriff »Methode« unüblich, da der standardisierte Name hier *member function* lautet, weshalb auch hier von Funktionen statt Methoden die Rede ist.

»RAII-Idiom« (von engl. »resource acquisition is initialization«) umzusetzen.

> »C++'s language-enforced constructor/destructor symmetry mirrors the symmetry inherent in resource acquire/release function pairs such as [...] *new*/*delete*. This makes a stack-based (or reference-counted) object with a resource-acquiring constructor and a resource-releasing destructor an excellent tool for automating resource management and cleanup.«[65]

Ein weiteres C++-Sprachfeature, auf das i. d. R. im Kontext von Spiele-Engines verzichtet wird, sind *Exceptions*:

> »The ability to separate error detection from error handling in such a clean way is certainly attractive, and exception handling is an excellent choice for some software projects. However, exception handling does add some overhead to the program. The stack frame of any function that contains a try-catch block must be augmented to contain additional information required by the stack unwinding process. Also, if even one function in your program (or a library that your program links with) uses exception handling, your entire program must use exception handling – the compiler can't know which functions might be above you on the call stack when you throw an exception.«[66]

Neben der potenziellen Verschlechterung der Leistung führt Gregory auch weitere Argumente gegen die Verwendung von Exceptions an, nämlich einerseits die Unübersichtlichkeit im Code, die dadurch entstehen kann, dass sogar Funktionen, die keinerlei für Exceptions relevanten Code enthalten, im Fehlerfall Teil des *Unwinding*-Prozesses[67] werden können. Andererseits geht er auf Performanceverschlechterungen ein, die entstehen können, obwohl die benutzte Implementierung der Exception-Behandlung keinen Overhead zur Laufzeit erzeugt, solange kein Fehler auftritt: Die zusätzlich nötigen Instruktionen, die zur Exception-Behandlung eingefügt werden müssen, können die Performance des Instruction-Caches verschlechtern oder das *Inlining*[68] einer Funktion verhindern[69].

[65] Sutter und Alexandrescu, 2004, S. 24
[66] Gregory, 2019, S. 123
[67] Von *Unwinding* spricht man, wenn man den Vorgang beschreibt, der im Falle einer geworfenen Exception ausgeführt wird. Dabei muss der Aufrufsstack von innen nach außen »aufgelöst« werden, was bedeutet, dass die Destruktoren sämtlicher Objekte ausgeführt werden müssen und zum jeweils übergeordneten Stack-Rahmen zurückgesprungen werden muss. Der Vorgang wird fortgesetzt, bis die Exception behandelt wird, spätestens in der verwendeten Laufzeitbibliothek, also außerhalb des eigenen Programms.
[68] Von *Inlining* spricht man, wenn der Compiler die Instruktionen einer Funktion an den Ort kopiert, von dem sie aufgerufen wird, anstatt einen Sprung zu generieren.
[69] Gregory, 2019, S. 124

2.10 Rendering-Optimierungen

Da das Thema des (Echtzeit-)Renderings sehr umfassend ist und alleine die Behandlung einiger weniger Optimierungstechniken bei weitem den Rahmen dieser Arbeit sprengen würde, wird auf eine ausführliche Darstellung an dieser Stelle verzichtet. Erwähnt werden soll jedoch, dass es sich bei heutigen Grafikkarten um dedizierte Hardware handelt, die für die parallele Verarbeitung ausgelegt ist. Derartige Hardware verfügt über Hunderte oder sogar Tausende Kerne zur parallelen Verarbeitung. Beispielsweise verfügt eine NVIDIA RTX 2070-Grafikkarte über 2304 Kerne[70]. Auch neben dem Rendering lässt sich derart spezialisierte Hardware als Ergänzung zur CPU einsetzen: Mit Hilfe von sog. »Compute-Shadern« lassen sich praktisch beliebige Berechnungen vom Grafikprozessor durchführen, insofern sich diese entsprechend parallelisieren lassen.

Um ein Beispiel für mögliche Optimierungen beim Rendering zu nennen, soll kurz auf eine Optimierungstechnik eingegangen werden, welche auch im praktischen Teil dieser Arbeit zum Einsatz kommt: Möchte man Geometrie auf dem Bildschirm darstellen lassen, so müssen die entsprechenden Geometriedaten[71] an die Grafikkarte übertragen werden. Danach wird der für das Zeichnen der Geometrie nötige Zustand hergestellt (relevante Texturen werden »aktiviert«[72], das benötigte Shaderprogramm[73] wird ausgewählt etc.) und anschließend wird über einen sog. »Draw-Call« der Vorgang des Renderns angestoßen.

Da die Kommunikation zwischen Spiele-Engine und Grafikkarte immer über den Treiber abläuft und jeder Schritt dieser Kommunikation recht aufwendig ist, sollte die Anzahl der Aufrufe von Funktionen der Grafikschnittstelle[74] auf ein Minimum begrenzt werden. Zu diesem Zweck kann man die Technik des sog. »Batch-Renderings« einsetzen. Dabei werden möglichst viele API-Aufrufe zusammengefasst. Dafür müssen die darzustellenden Geometriedaten i. d. R. CPU-seitig sortiert werden, bevor sie der Grafikkarte übergeben werden können. Sollen z. B. 2000 Dreiecke dargestellt werden, denen insgesamt zwei verschiedene Shaderprogramme sowie 16 verschiedene Texturen zugeordnet sind, lassen sich durch Sortieren die Daten so anordnen, dass zuerst sämtliche Geometrie, die

[70] Quelle: `https://www.pcgamesn.com/nvidia-rtx-2070-review-benchmarks-complete`, aufgerufen am 13.08.2021

[71] Damit sind die Vertex-Daten gemeint. Diese stellen jeweils eine Zusammenfassung aller für einen Eckpunkt der Geometrie relevanten Informationen dar. Dazu gehören üblicherweise die Position des Eckpunkts, seine Texturkoordinaten und seine Farbe, aber auch beliebige weitere, anwendungsspezifische Attribute.

[72] Man spricht hier von *texture binding*.

[73] Ein Shaderprogramm ist eine Sammlung mehrerer Programmroutinen (mindestens ein *vertex shader* sowie ein *fragment shader*), über die es möglich ist, Teile der Render-Pipeline anzupassen und so die Bildschirmausgabe zu steuern.

[74] z. B. OpenGL, Vulkan, DirectX, Metal etc.

das erste der beiden Shaderprogramme benötigt, zusammengefasst gezeichnet werden kann und danach die verbleibende Geometrie. Da es auf allen aktuellen Grafikkarten möglich ist, 16 Texturen gleichzeitig zu binden, können die 2000 Dreiecke somit mit nur zwei Draw-Calls gezeichnet werden. Ein in einer Spiele-Engine verankerter Automatismus für diese Optimierungstechnik verhindert Ineffizienzen bei der Darstellung.

3 Repräsentation von Objekten in der virtuellen Spielwelt

Fast jedes Videospiel verfügt über eine »virtuelle Spielwelt«, in der sich Objekte verschiedenster Art befinden. Manche dieser Objekte sind dabei statisch, andere wiederum bewegen sich dynamisch, können mit den Spielenden interagieren oder werden direkt oder indirekt von diesen gesteuert. Die Art und Weise, wie diese Objekte in der jeweils verwendeten Programmiersprache repräsentiert werden, hat sich im Laufe der Geschichte der Videospiele deutlich geändert. Die folgenden Abschnitte geben einen Überblick über verschiedene Herangehensweisen. Auf Beispiele aus der Zeit vor 1990 wird bei der Betrachtung verzichtet.

3.1 Prozedurale Programmierung

Um in der Anfangszeit der Videospielprogrammierung effizienten Gebrauch von der zur Verfügung stehenden Hardware machen zu können, war deren genaue Kenntnis notwendig. Spiele mussten gezielt auf die jeweilige Plattform angepasst werden, um die benötigte Leistung abfragen zu können. Daher wurde anfänglich vorwiegend in Assemblersprachen programmiert. Erst in den 1990er Jahren hielten Hochsprachen wie C Einzug und erlaubten somit den Sprung von der rein imperativen zur strukturierten bzw. prozeduralen Programmierung. Für leistungskritische Programmabschnitte wurde jedoch noch weit darüber hinaus mit Assemblersprachen gearbeitet[75].

Anhand des Ego-Shooters Doom[76] soll beispielhaft eine Möglichkeit gezeigt werden, wie unter Verwendung der prozeduralen Programmierung Objekte der Spielwelt repräsentiert werden können. Der in Listing 3.1 gezeigte Code-Ausschnitt aus der Datei `p_mobj.c` zeigt die Funktion P_SpawnMobj(), die einen neuen Gegner in der Spielwelt instanziiert. Die Zeilen 9 und 10 sorgen dabei dafür, dass ein Funktionszeiger auf die (nicht im Listing enthaltene) Funktion P_MobjThinker() in eine doppelt verkettete Liste eingetragen wird. In dieser Liste befinden sich Funktionszeiger auf Funktionen, die für verschiedenste Objekte zuständig sind. Dazu gehören neben den Gegnern z. B. auch interaktive Gegenstände wie Türen. Im Main-Loop des Spiels wird über diese Liste iteriert und die registrierten Funktionszeiger werden genutzt, um die zu den in der Spielwelt vorhandenen Spielobjekten (im Quelltext als »Actor« bezeichnet) gehörenden Funktionen aufzurufen.

[75] Madhav, 2014, S. 2 f.

[76] id Software, 1993, Quellcode online verfügbar unter `https://github.com/id-Software/DOOM`, aufgerufen am 02.08.2021

```
1  mobj_t* P_SpawnMobj(fixed_t x, fixed_t y, fixed_t z,
2                      mobjtype_t type) {
3      // ...
4      mobj->type = type;
5      mobj->info = info;
6      mobj->x = x;
7      mobj->y = y;
8      // ...
9      mobj->thinker.function.acp1 = (actionf_p1)P_MobjThinker;
10     P_AddThinker (&mobj->thinker);
11     return mobj;
12 }
```

Listing 3.1: Funktion P_SpawnMobj() zum Instanziieren von Gegnern in Doom

Die eigentlichen Daten der verschiedenen Objekte werden in einer sich selbst verwaltenden Datenstruktur abgelegt, welche vor Spielstart einmalig 6 MiB Speicher anfordert und so Speicher-Allokationen während des Spielens verhindert. Durch die verwendete Architektur ist es auch ohne Vererbung möglich, die verschiedenen Actors gleichermaßen zu behandeln, obwohl es sich dabei um verschiedene Datentypen handelt. Die Typsicherheit geht dabei jedoch verloren, da ausschließlich mit untypisierten Zeigern (void*) gearbeitet wird.

3.2 Objektorientierter Entwurf

Als C++ mehr und mehr Verbreitung in der Spielebranche fand, etablierten sich auch die damit zur Verfügung stehenden Werkzeuge der objektorientierten Programmierung. Um Objekte in der Spielwelt zu repräsentieren, wurde in diesem Zusammenhang oft auf Vererbung gesetzt: Eine Basisklasse stellt dabei die Wurzel der gesamten Vererbungshierarchie dar; Unterklassen erben von dieser Klasse und spezialisieren sich immer weiter. Listing 3.2 soll diesen Hierarchieaufbau illustrieren.
Da alle Objekte somit über die Schnittstelle der identischen Basisklasse angesprochen werden können, können diese in einem gemeinsamen Datencontainer gehalten werden. Bei der Iteration über diesen Container werden sodann virtuelle Funktionen der Basisklasse aufgerufen, die durch »virtual dispatch«[77] an die jeweilige Unterklasse »weitergereicht« werden. In Listing 3.3 ist dieses Vorgehen dargestellt: In den Zeilen 1 bis 3 werden ein Container angelegt und zwei Objekte verschiedener Typen in diesen eingefügt. Im Game-Loop kann sodann über sämtliche Elemente iteriert werden, um deren jeweilige update()- bzw. render()-Funktion aufzurufen.
Neben den in Abschnitt 2.9 erwähnten Performanceverschlechterungen, die

[77] siehe Abschnitt 2.9

```
struct GameObject {
    virtual void update() = 0;
    virtual void render() const = 0;
};

struct Player : public GameObject {
    Player(const string& name, int health)
        : name{ name }, health{ health } { }
    void update() override { /* ... */ }
    void render() const override { /* ... */ }
    string name;
    int health;
    // ...
};

struct Enemy : public GameObject {
    Enemy(int health, int colorIndex)
        : health(health), colorIndex(colorIndex) { }
    void update() override { /* ... */ }
    void render() const override { /* ... */ }
    int health;
    int colorIndex;
    // ...
};
```

Listing 3.2: Repräsentation von Objekten der Spielwelt durch eine Vererbungshierarchie

```
vector<unique_ptr<GameObject>> gameObjects;
gameObjects.push_back(make_unique<Player>("Gordon", 100));
gameObjects.push_back(make_unique<Enemy>(35, 2));
// ...
bool gameIsRunning = true;
while (gameIsRunning) {
    for (auto& gameObject : gameObjects)
        gameObject->update();
    for (const auto& gameObject : gameObjects)
        gameObject->render();
    // ...
}
```

Listing 3.3: Game-Loop über einen Container von GameObject-Instanzen

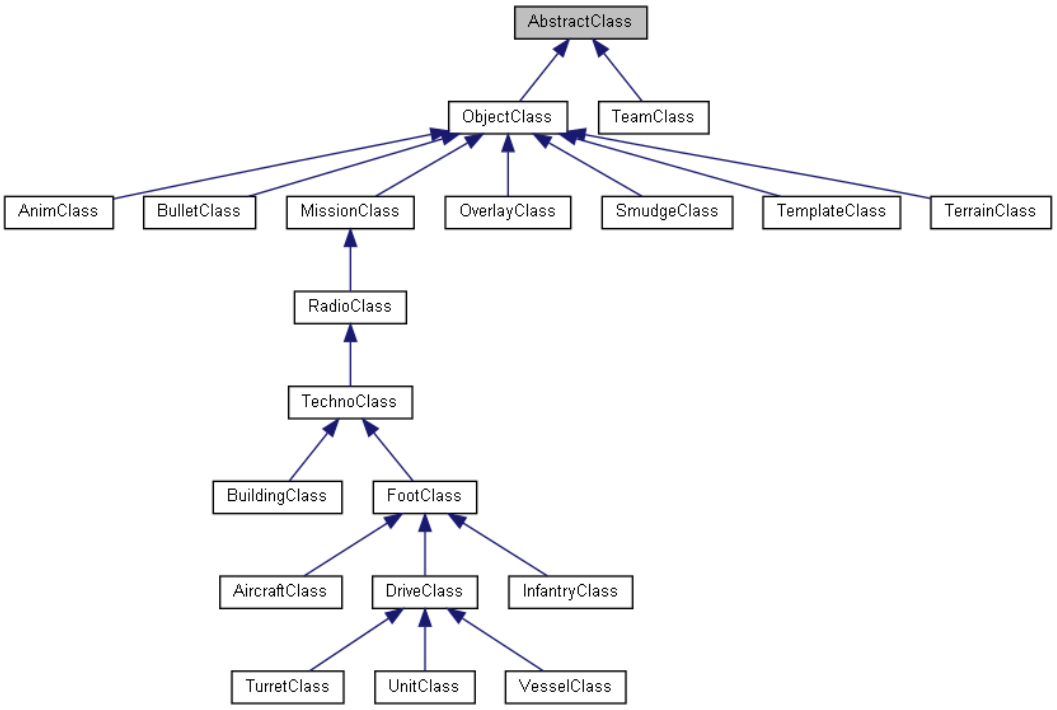

Abbildung 3.1: Vererbungshierarchie von Command & Conquer: Red Alert
Quelle: Abb. mit doxygen erstellt aus dem Original-Quellcode

durch die Verwendung virtueller Funktionen auftreten können, kann diese Herangehensweise noch zu weiteren Problemen führen. Diese sollen als konkretes Beispiel am Quelltext von »Command & Conquer: Red Alert«[78] gezeigt werden. Dort ist das soeben erläuterte Konzept einer Vererbungshierarchie umgesetzt worden. Als Basisklasse dient dort die Klasse AbstractClass. In der entsprechenden Quelltext-Datei heißt es: »This class is the base class for all game objects that have an existence on the battlefield.«[79] Die von dieser Klasse ausgehende Vererbungshierarchie ist in Abb. 3.1 zu sehen. Zu erkennen ist, dass die Grundidee der objektorientierten Programmierung, nämlich die Modellierung von (echten oder – wie in diesem Fall – virtuellen) Objekten, bei diesem Entwurf verletzt wird: Viele der Klassen modellieren nicht *wirklich* Objekte, die im Spiel auftauchen, sondern fassen lediglich die gemeinsame Funktionalität der Unterklassen zusammen, um Code-Doppelungen zu verhindern. Dabei entstehen »künstliche Zwischenklassen« wie z. B. TechnoClass, in deren Quelltextdatei es heißt: »This is the common data between building [sic] and units.«[80]
Bei der Vermittlung der Grundlagen objektorientierter Programmierung wird

[78] Westwood Studios, 1996, Quellcode der »Remastered Collection« online verfügbar unter `https://github.com/electronicarts/CnC_Remastered_Collection`, aufgerufen am 13.08.2021
[79] Quelle: `https://raw.githubusercontent.com/electronicarts/CnC_Remastered_Collection/master/REDALERT/ABSTRACT.H`, aufgerufen am 13.08.2021
[80] Quelle: `https://raw.githubusercontent.com/electronicarts/CnC_Remastered_Collection/master/REDALERT/TECHNO.H`, aufgerufen am 13.08.2021

das Konzept der Vererbung i. d. R. als eine »ist ein«-Beziehung dargestellt. Als Beispiele werden oft Analogien aus dem Alltag bemüht: Eine Katze »ist ein« Säugetier, ein Rechteck »ist ein« Viereck. Diese Denkweise bricht in diesem Praxisbeispiel jedoch zusammen: Die Aussage

> »Ein Vehikel (UnitClass) *ist ein* bewegliches Objekt (DriveClass) *ist ein* bewegliches Objekt (FootClass) *ist ein* Oberobjekt von Gebäuden und Einheiten (TechnoClass) *ist ein* Objekt, das Nachrichten empfangen kann (RadioClass), *ist ein* Objekt, das die Zuweisung von Kommandos behandelt (MissionClass), *ist ein* Objekt (ObjectClass) *ist ein* abstraktes Objekt (AbstractClass)«[81]

wirkt nicht nur sehr gekünstelt und übermäßig komplex, sondern umfasst eine Reihe von Beschreibungen, denen keine Analogie zugeordnet werden kann. Hier werden also nicht *wirklich* Objekte modelliert, sondern die gewünschten Objekte werden zur Reduzierung von Code-Doppelungen in ein abstraktes Konstrukt gezwungen.
Auch, wenn es sich hierbei in erster Linie um ein ästhetisches »Problem« handelt und man diesen Aspekt daher ignorieren könnte, kann beim objektorientierten Entwurf ein weiteres Problem entstehen. Beispielsweise existiert in der Klassenhierarchie von Command & Conquer: Red Alert – wie in Abb. 3.1 zu sehen – die Klasse AircraftClass. Was die Abbildung nicht zeigt, ist, dass diese Klasse nicht nur von FootClass, sondern auch von der Klasse FlyClass erbt. Letztere behandelt zwar alle fliegenden Objekte (und damit explizit auch Flugzeuge), ist jedoch selbst keine Unterklasse von AbstractClass. Dieser »Logikfehler« musste offensichtlich in Kauf genommen werden, um einem bekannten Problem bei Mehrfachvererbung aus dem Weg zu gehen:

> »[...] multiple inheritance in C++ poses a number of practical problems. For example, multiple inheritance can lead to an object that contains multiple copies of its base class' members – a condition known as the "deadly diamond" or "diamond of death."«[82]

Abhängig von der Komplexität des zu entwickelnden Spiels und des Spielgenres lässt sich das letztgenannte Problem vermeiden: Obwohl im Spiel »Half-Life«[83] z. B. eine quantitativ deutlich umfangreichere Klassenhierarchie vorliegt, weist diese jedoch hauptsächlich eine eher »breite« Topologie ohne besonders tiefe Vererbung und insbesondere ohne Mehrfachvererbung auf. Das (nicht technische)

[81] Die hier gewählten Umschreibungen sind sinngemäße Übersetzungen, die direkt aus den Kommentaren innerhalb der Quelltextdateien stammen.
[82] Gregory, 2019, S. 1049
[83] Valve, 1998, Quellcode online verfügbar unter `https://github.com/ValveSoftware/halflife`, aufgerufen am 13.08.2021

»Problem«, dass nicht alle Klassen Objekte der Spielwelt modellieren, ist jedoch auch hier präsent.
Beide gezeigten Projekte weisen darüber hinaus die Eigenschaft auf, dass die jeweiligen Klassendefinitionen sehr umfangreich werden: Header-Dateien mit Hunderten oder Tausenden von Zeilen sind keine Seltenheit in diesen Projekten. Die Klassen werden zu monolithischen Gebilden, die schwer zu überblicken sind.

3.3 Composition over Inheritance

Mit der Zeit fanden Bestrebungen statt, die den im vorherigen Abschnitt demonstrierten Problemen entgegen wirken sollten.

> »When object-oriented programming first hit the scene, inheritance was the shiniest tool in the toolbox. It was considered the ultimate code-reuse hammer, and coders swung it often. Since then, we've learned the hard way that it's a heavy hammer indeed. Inheritance has its uses, but it's often too cumbersome for simple code reuse.
>
> Instead, the growing trend in software design is to use composition instead of inheritance when possible. Instead of sharing code between two classes by having them *inherit* from the same class, we do so by having them both *own an instance* of the same class.«[84]

Bei einem solchen Ansatz lässt sich jedes Objekt der Spielwelt derart abstrahieren, dass lediglich die Kombination aus Komponenten, die ihm zugewiesen werden, das Objekt selbst ausmachen. Es ist dabei also nicht mehr nötig, für jede Objektart eine eigene Klasse zu implementieren. Stattdessen existiert eine einzige Klasse (z. B. GameObject, Actor oder Entity genannt), die eine beliebige Kombination aus Komponenten aggregieren kann. Durch die spezifische Kombination werden der Typ und das Verhalten des Gesamtobjekts definiert. Die Implementierungsdetails können hierbei stark variieren. Ein möglicher Ansatz ist in Nystrom, 2014 gezeigt. Im Folgenden wird jedoch eine davon abweichende Implementierung umrissen. Listing 3.4 zeigt eine mögliche Implementierung der *Komponenten*. Es wird eine abstrakte Oberklasse (Component) genutzt, von der die tatsächlichen Komponenten dann erben. Letztere enthalten für sie relevante Daten sowie Funktionalität. Beispielhaft sind die beiden Komponenten Position und Health zu sehen.
Da die Objekte der Spielwelt nunmehr als reine Container von Komponenten dienen sollen, wird für sie nur eine einzige Klasse benötigt. Diese ist in Listing

[84] Nystrom, 2014, S. 215

```cpp
#include <memory>
#include <vector>

using std::vector, std::unique_ptr, std::move, std::make_unique;

struct Component {
    virtual void update(float deltaTime) = 0;
    virtual void render() const = 0;
};

using ComponentPtr = std::unique_ptr<Component>;

struct Position : public Component {
    float x, y;
    Position(float x, float y) : x{ x }, y{ y } { }
    void update(float deltaTime) override {
        // apply gravity
        y -= 0.5f * 9.81f * deltaTime * deltaTime;
    }
    void render() const override { }
};

struct Health : public Component {
    int health;
    Health(int health) : health{ health } { }
    void update(float) override { }
    void render() const override { }
};
```

Listing 3.4: Komponenten in einer Kompositions-basierten Architektur

3.5 zu sehen. Sie enthält eine Funktion zum Hinzufügen neuer Komponenten sowie zwei Funktionen, um die update()- bzw. render()-Funktionen der ihr zugewiesenen Komponenten aufzurufen. Die Komponenten selbst werden über Pointer referenziert, die in der Member-Variable mComponents gehalten werden. Der gezeigte Code ist nur illustrierend zu verstehen, da wesentliche Konzepte dort nicht implementiert sind (z. B. das Entfernen von Komponenten und die Kommunikation der verschiedenen Komponenten untereinander).
Durch den Einsatz von C++-Templates (in Listing 3.5) können einem GameObject neue Komponenten sehr elegant hinzugefügt werden. Die Benutzung der implementierten Schnittstellen ist in Listing 3.6 gezeigt[85]. Diese Architektur zeigt, wie die Probleme des objektorientierten Entwurfs verhindert werden können und eine bessere Skalierbarkeit erreicht werden kann.

3.4 Entity-Component-System-Architektur

Es ist festzuhalten, dass mit allen bisher dargestellten Herangehensweisen sehr wohl weltweit kommerziell erfolgreiche und bekannte Spiele entwickelt wurden.

[85] Um die Lesbarkeit des gezeigten Codes zu erhöhen, wurden Teilaspekte der C++-Move-Semantics außer Acht gelassen.

```
30 class GameObject {
31 public:
32     template<typename T, typename... Args>
33     void addComponent(Args... args) {
34         mComponents.push_back(make_unique<T>(args...));
35     }
36
37     void update(float deltaTime) {
38         for (auto& component : mComponents)
39             component->update(deltaTime);
40     }
41
42     void render() {
43         for (const auto& component : mComponents)
44             component->render();
45     }
46 private:
47     vector<ComponentPtr> mComponents;
48 };
```

Listing 3.5: Game-Objects in einer Kompositions-basierten Architektur

Eine Ausrichtung der Software-Architektur nach den in Abschnitt 2 gezeigten Optimierungstechniken hat bei diesen Herangehensweisen in der Praxis jedoch allenfalls in Details stattgefunden.

3.4.1 Verbesserungen der Komponenten-basierten Architektur

Ausgehend von der in Abschnitt 3.3 gezeigten, Komponenten-basierten Architektur, können weitere Verbesserungen vorgenommen werden:

> »Some component systems take this even further. Instead of a GameObject that contains its components, the game entity is just an ID, a number. Then, you maintain separate collections of components where each one knows the ID of the entity its attached to.
>
> These *entity component systems* take decoupling components to the extreme and let you add new components to an entity without the entity even knowing.«[86]

Es ist dabei jedoch festzuhalten, dass die Bezeichnung »Entity-Component-System« (*ECS*) zwar üblich, aber nicht korrekt ist. Stattdessen spricht man korrekterweise von dem »entity component system architectural pattern«[87], da eine solche Architektur aus den drei namensgebenden Bestandteilen besteht:

- *Entities* repräsentieren Objekte in der Spielwelt und entsprechen den in vorherigen Abschnitten auch als Game-Object oder Actor bezeichneten

[86] Nystrom, 2014, S. 225
[87] Caini, 2019

```
50 int main() {
51     vector<GameObject> gameObjects;
52     GameObject player;
53     player.addComponent<Position>(0.0f, 0.0f );
54     player.addComponent<Health>(100);
55     gameObjects.push_back(move(player));
56
57     GameObject enemy;
58     enemy.addComponent<Position>(10.0f, 0.0f);
59     enemy.addComponent<Health>(70);
60     gameObjects.push_back(move(enemy));
61
62     bool gameRunning = true;
63     while (gameRunning) {
64         for (auto& gameObject : gameObjects)
65             gameObject.update(1.0f / 60.0f);
66         for (const auto& gameObject : gameObjects)
67             gameObject.render();
68         // ...
69     }
70 }
```

Listing 3.6: Verwendung einer Kompositions-basierten Architektur

Objekten. Das wesentliche Merkmal bei einer ECS-Architektur ist die Tatsache, dass ein Entity hier lediglich einer Identifizierungsnummer (*ID*) entspricht.

- *Components* stellen einfache Datencontainer dar, welche die für einen Aspekt eines Entitys verantwortlichen Informationen kapseln. Üblicherweise stellen Komponenten keinerlei Funktionalität zur Verfügung. Ein Beispiel für eine solche Komponente stellt die Transformation (Translation, Rotation, Skalierung) eines Objekts dar. Diese Transformation kann beispielsweise als 4×4-Matrix repräsentiert werden und somit eine Komponente darstellen.

- *Systems* iterieren – i. d. R. einmal pro Wiederholung des Game-Loops – über eine oder eine Kombination mehrerer Komponenten genau derjenigen Entitys, welchen (mindestens) die entsprechenden Komponenten zugewiesen wurden. Beispielsweise kann ein System über alle Entities iterieren, denen jeweils sowohl eine Transformations- als auch eine Sprite-Komponente zugewiesen wurden, um die entsprechenden Sprites unter Berücksichtigung der jeweiligen Transformation am Bildschirm darzustellen.

Bei einer ECS-Architektur werden Daten und Funktionalitäten also strikt voneinander getrennt. Dies steht im deutlichen Gegensatz zur objektorientierten Programmierung, bei der Objekte ihre Daten sowie die dazugehörige Funktionalität *gemeinsam* kapseln.

3.4.2 Data-oriented Design

Der ECS-Architektur liegt die Idee des »Data-oriented Design« (*DOD*) zugrunde.

> »In essence, data-oriented design is the practice of designing software by developing transformations for well-formed data where the criteria for well-formed is guided by the target hardware and the patterns and types of transforms that need to operate on it.«[88]

Es geht bei diesem Ansatz also darum, die Zugriffsmuster auf benötigte Daten – einschließlich der Instruktionen – zu kennen und die Daten daraufhin so anzuordnen, dass die Zugriffe für die Zielplattform optimiert werden. Grundsätzlich wird also versucht, die in Abschnitt 2 präsentierten Eigenschaften moderner Hardware so weit wie möglich beim Entwurf der Softwarearchitektur zu berücksichtigen.
Zu beachten ist, dass die durch eine ECS-Architektur zu erwartende Performanceverbesserung erst bei sehr großen Softwareprojekten, insbesondere bei sog. »Triple-A«-Spielen eine tragende Rolle spielt; dennoch ist – aus bereits genannten Gründen – eine Komponenten-basierte Architektur vorteilhaft: »[...] component based programming is an incredibly powerful tool to make code flexible and to speed up iterations during development.«[89]
Wie in Abschnitt 3.4 beschrieben, iterieren die Systeme der ECS-Architektur über (Kombinationen von) Komponenten. In Abschnitt 2 wurde durch die Betrachtung von Hardwareeigenschaften sowie Messergebnisse belegt, dass das Iterieren über fortlaufenden bzw. kontinuierlichen Speicher mit fester Schrittweite die vorhandenen Hardwareoptimierungen wie z. B. das Caching bestmöglich ausnutzt. Die grundlegende Schlussfolgerung aus dieser Tatsache ist nun, dass die Daten der Komponenten der ECS-Architektur möglichst kontinuierlich und ohne Lücken im Speicher liegen sollten. In Caini, 2019 wird eine mögliche Implementierung einer ECS-Architektur vorgestellt, die auch als Grundlage für den die vorliegende Arbeit begleitenden praktischen Teil dient. Details zur umgesetzten Implementierung sind in Abschnitt 4.1 zu finden.

3.5 Implementierungen in etablierten Produkten bzw. Projekten

Die in den vorherigen Abschnitten vorgestellten Architekturen fanden in verschiedensten Spielen bzw. Spiele-Engines Verwendung. Beispiele für prozedurale

[88] Fabian, 2018, S. 4
[89] Caini, 2019

sowie objektorientierte Programmierung wurden bereits in den Abschnitten 3.1 und 3.2 genannt. Als prominentes Beispiel für den Komponenten-basierten Entwurf ist die Unity-Engine zu nennen: »The Unity framework's core GameObject class is designed entirely around components.«[90] Der vermutlich größte Konkurrent der Unity-Engine, die Unreal-Engine, setzt dagegen auf einen objektorientierten Ansatz: »In Unity, GameObject is [a] C# class which you cannot directly extend. In UE4, Actor is a C++ class which you can extend and customize using inheritance.«[91]
Zu erwähnen ist, dass für die Unity-Engine auch ein zweiter Ansatz verfügbar ist: Unter dem Namen »Data-oriented Technology-Stack« (DOTS) existiert eine ECS-Architektur für die Engine[92]. Auch die »Artemis Engine«[93] implementiert eine ECS-Architektur[94]. Natürlich ist die hier gezeigte Auswahl keinesfalls erschöpfend, sie soll jedoch verdeutlichen, dass sämtliche verschiedenen Architekturen in der Vergangenheit erfolgreich zum Einsatz kamen, dass aber vor allem in der Gegenwart eine Fokussierung auf Data-oriented Design stattfindet, um den steigenden Anforderungen gerecht zu werden.

[90] Nystrom, 2014, S. 231
[91] Quelle: `https://docs.unrealengine.com/4.26/en-US/Basics/UnrealEngineForUnityDevs/`, aufgerufen am 15.08.2021
[92] vgl. `https://unity.com/de/dots`, aufgerufen am 15.08.2021
[93] vgl. `https://github.com/ArtemisEngine/Artemis-Engine`
[94] vgl. Nystrom, 2014, S. 290

4 Fallstudie: Implementierung einer rudimentären 2D-Engine

Die in Kapitel 2 vorgestellten und durch Messergebnisse untermauerten Erkenntnisse führten unter anderem zu den in Kapitel 3 gezeigten Verbesserungen in der Softwarearchitektur von Spielen bzw. Spiele-Engines. Um eine praktische Umsetzung der dargestellten Konzepte zu zeigen, entsteht begleitend zur vorliegenden Arbeit eine rudimentäre Spiele-Engine. In den folgenden Abschnitten soll auf einige Details dieser Implementierung eingegangen und dabei aufgetretene Fallstricke beschrieben werden. Der Fokus liegt bei diesem Projekt nicht auf der Erstellung eines Spiels, sondern auf der Schaffung eines Software-Pakets, welches die Erstellung von Spielen derart ermöglicht, sodass die Erkenntnisse aus den vorherigen Kapiteln implizit berücksichtigt werden.
Bei der entstandenen Software handelt es sich aus Gründen des Umfangs *nicht* um ein für die Praxis geeignetes Paket, sondern eher um eine Machbarkeitsstudie, welche als Grundlage für weitere Entwicklung dienen kann. Der Schwerpunkt bei der Entwicklung liegt dabei auf der Erstellung einer eigenen Implementierung einer Entity-Component-System- bzw. ECS-Architektur[95]. Zwar existieren hierfür frei verfügbare Lösungen, jedoch ermöglicht eine eigene Implementierung ein deutlich tiefer gehendes Verständnis der während des Entwicklungsprozesses auftretenden Problemstellungen.
Um den Rahmen der vorliegenden Arbeit nicht zu sprengen, fokussieren sich auch die nachfolgenden Betrachtungen auf die ECS-Architektur. Die restlichen Systeme der entwickelten Engine werden lediglich angerissen.
Der jeweils aktuelle Stand der Implementierung – auch über den Kontext dieser Arbeit hinaus – ist unter `https://github.com/mgerhold/engine2d` zu finden.

4.1 ECS-Architektur

Die umgesetzte Implementierung der ECS-Architektur basiert auf der Blog-Serie von Michele Caini[96]. Entities werden hierbei durch einfache Identifikationsnummern repräsentiert[97]. Eine zentrale Klasse namens Registry verwaltet eine Liste aller aktiven Entities. Da es im Kontext von Spielen üblich ist, dass oft neue Entities erzeugt und aktive Entities zerstört werden müssen, wird ein Mechanismus eingesetzt, der die Wiederverwendung von Entity-Identifikationsnummern

[95] siehe Abschnitt 3.4
[96] siehe Caini, 2019
[97] Der verwendete Datentyp ist dabei austauschbar. Standardmäßig werden vorzeichenlose 32-Bit-Zahlen verwendet.

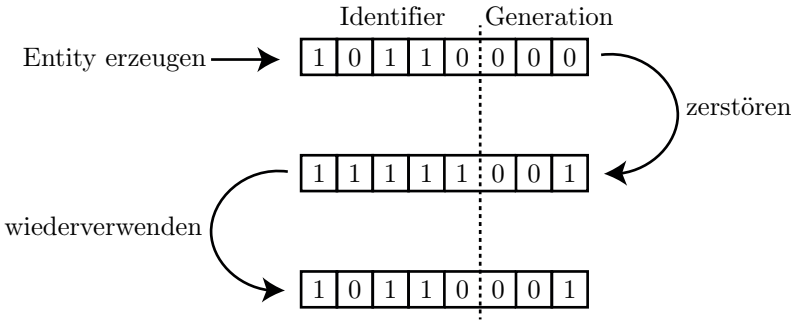

Abbildung 4.1: Wiederverwendung von Entity-IDs
Quelle: Abb. selbst erstellt

(Entity-IDs) ermöglicht. Hierzu wird die Ganzzahl, welche das Entity repräsentiert, in zwei Teile aufgeteilt:

- Die »Identifier-Bits« stellen die eigentliche ID des Objekts dar.
- Die »Generation-Bits« dienen zur Differenzierung unterschiedlicher Objekte, welche dieselbe ID benutzen. Die in den Generation-Bits gespeicherte Zahl wird erhöht, wenn eine ID wiederverwendet wird.

Standardmäßig werden $\frac{5}{8}$ der verfügbaren Bits für den ID-Anteil verwendet, während die restlichen Bits der Kodierung der Generation dienen. Abb. 4.1 stellt die Veränderungen dar, die bei der Zerstörung eines Entitys und anschließender Wiederverwendung der Identifier-Bits zur Erzeugung eines neuen Entitys auftreten[98]. Diese Schritte stellen sich wie folgt dar:

1. Es wird ein neues Entity erzeugt, indem den Identifier-Bits eine aufsteigende Nummer zugewiesen wird (im Beispiel $22_{10} = 10110_2$). Die Generation-Bits werden auf Null initialisiert. Somit ergibt sich als Gesamt-ID $10110000_2 = 176_{10}$.
2. Wird das Entity zerstört, werden dessen Identifier-Bits auf einen ungültigen Wert gesetzt. In der vorliegenden Implementierung entspricht dies dem maximal darstellbaren Wert, hier also $11111_2 = 31_{10}$, was in einer Gesamt-ID von $11111000_2 = 248_{10}$ entspricht. Außerdem wird die in den Generation-Bits kodierte Zahl um Eins erhöht, sodass bei Wiederverwendung der Identifier-Bits eine davon verschiedene Gesamt-ID entsteht.
3. Soll ein neues Entity erzeugt werden, werden die zuvor genutzten Identifier-Bits wiederverwendet und die (bereits erhöhte) Generation übernommen. Damit ergibt sich als neue Gesamt-ID $10110001_2 = 177_{10}$.

[98] Beispielhaft werden 8-Bit-Große Entity-IDs angenommen (5 Identifier-Bits, 3 Generation-Bits).

```
void destroyEntity(Entity entity) noexcept {
    const auto index = getIndexFromEntity(entity);
    increaseGeneration(mEntities[index]);
    swapIdentifiers(mNextRecyclableEntity, mEntities[index]);
    ++mNumRecyclableEntities;
}
```

Listing 4.1: Zerstören eines Entitys

Obwohl die Identifier-Bits beider Entities identisch sind, können diese also durch einen einfachen Ganzzahlvergleich unterschieden werden. Etwaige Referenzen auf das nicht mehr existierende Entity können somit als ungültig erkannt werden. Dieses System kann zu Fehlern führen, wenn eine Referenz auf ein nicht mehr existierendes Entity derart lange gehalten wird, bis die Identifier-Bits des Entitys so oft wiederverwendet wurden, dass eine Erhöhung der Generation-Bits zu einem Überlauf führt. Dies bringt zwei voneinander verschiedene Entitys hervor, die jedoch über exakt identische Gesamt-IDs verfügen. Da als Datentyp in der Praxis mindestens eine 32-Bit-Ganzzahl anzunehmen ist und ungültige Referenzen i. d. R. nicht derart lange gehalten werden, ist dieser Umstand in der Praxis vernachlässigbar.

4.1.1 Verwaltung wiederverwendbarer Entities

Ein wichtiger Teil dieses Mechanismus ist die Speicherung bzw. Verwaltung der wiederverwendbaren Identifier-Bits. Hierfür könnte man beispielsweise eine einfach verkettete Liste im Sinne einer Warteschlange verwenden. Diese würde jedoch zusätzlichen Speicher in Anspruch nehmen und darüber hinaus zu dynamischen Speicherallokationen sowie zusätzlichen Indirektionen führen, was einer optimalen Speicher- bzw. Cache-Ausnutzung entgegen spricht. Stattdessen kann die bereits bestehende Datenstruktur genutzt werden, um eine »implizite Liste« von wiederverwendbaren IDs zu führen. Dazu werden – neben der Entity-Liste – zwei zusätzliche Variablen genutzt:

- In einer Variable numRecyclableEntities (anfänglich Null) wird die Anzahl der derzeit wiederverwendbaren IDs und
- in einer zweiten Variable nextRecyclableEntity (anfänglich die ungültige ID) die als nächstes wiederzuverwendende ID gespeichert.

Wenn ein Entity zerstört wird, werden dessen Identifier-Bits mit denen in der Variable nextRecyclableEntity getauscht sowie der Wert von numRecyclableEntities um Eins erhöht. Listing 4.1 zeigt den relevanten Code-Ausschnitt. In Abb. 4.2 ist der Prozess veranschaulicht. Die gestrichelten Pfeile illustrieren, wie durch das beschriebene Verfahren eine verkettete Liste *innerhalb der bestehenden*

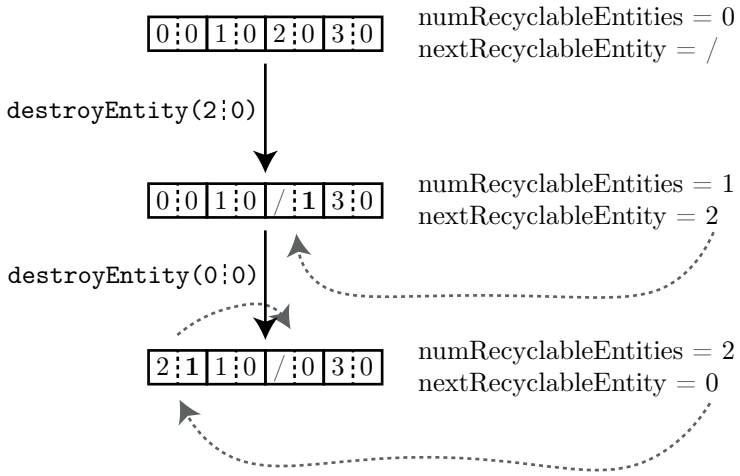

Abbildung 4.2: Zerstören von Entities
Quelle: Abb. selbst erstellt

```
[[nodiscard]] bool isEntityAlive(Entity entity) const noexcept {
    const auto index = getIndexFromEntity(entity);
    return mEntities[index] == entity;
}
```

Listing 4.2: Überprüfung auf Existenz eines Entitys

Datenstruktur entsteht. Die Variable nextRecyclableEntity verweist dabei auf den Listenkopf. Die sukzessive gelesenen Identifier-Bits verweisen auf den Index ihrer jeweiligen Nachfolger. Die fettgedruckten Ziffern in der Abbildung heben die inkrementierten Generation-Bits hervor.
Wenn ein neues Entity erzeugt werden soll, wird zunächst mit Hilfe der Variable numRecyclableEntities überprüft, ob Entity-IDs wiederverwendet werden können. Falls ja, werden die Identifier-Bits in nextRecyclableEntity mit denen des Elements, auf das diese indizieren, getauscht. Die Zahl der noch verbleibenden wiederverwendbaren Entities wird verringert und der sich nun wieder innerhalb der Liste befindende Identifier wird zurückgegeben.
Durch diesen Aufbau lässt sich sehr leicht herausfinden, ob ein Entity mit gegebener ID derzeit existiert oder nicht: Es muss lediglich überprüft werden, ob dasjenige Element des Entity-Arrays, welches durch die Identifier-Bits der gegebenen ID indiziert wird, mit dieser übereinstimmt. Listing 4.2 zeigt den dafür relevanten Code.

4.1.2 Speicherung der Komponenten

Wie in Abschnitt 3.4.2 erwähnt sollten die einzelnen Komponenten bei einer ECS-Architektur hintereinander und lückenlos im Speicher liegen. Eine Herausforderung dabei besteht darin, dass alle Entities über beliebige Kombinationen

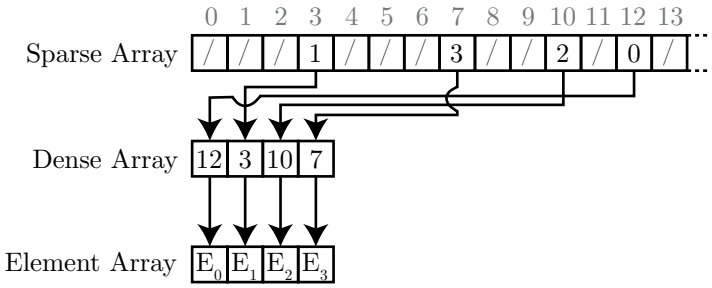

Abbildung 4.3: Aufbau der »Sparse Set«-Datenstruktur
Quelle: Abb. selbst erstellt

von Komponenten verfügen können[99]. Die Assoziation zwischen einem Entity und einer Komponente erfolgt über die Identifier-Bits der Entity-ID. Diese dienen als Index in den Datencontainer, welcher die jeweilige Komponenten-Art aufnimmt.
Da nicht jedes Entity über alle Komponenten verfügt, würden im Datencontainer Lücken entstehen. Dieses Problem wird über eine zusätzliche Indirektion gelöst. Die als »Sparse Set« bezeichnete Datenstruktur verfügt dabei über drei dynamische Arrays:

- Das »Sparse Array« ist mindestens so groß wie die aktuelle Anzahl *aller* Entities der gesamten Spielszene. Jedes Element dieses Arrays ist entweder ungültig oder enthält einen Index, der auf das »Dense Array« verweist.
- Im »Dense Array« ist eine Liste aller Entities gespeichert, die über die Komponenten-Art verfügen, die in diesem Sparse Set gespeichert werden soll.
- Das »Element Array« enthält die eigentlichen Nutzdaten, also die tatsächlichen Komponenten. Jede Komponente wird dabei demjenigen Entity zugeordnet, auf das durch das Element des Dense Arrays, welches denselben Index wie die Komponente besitzt, referenziert wird.

Dieser Aufbau ist (unter Zuhilfenahme von Beispiel-Werten) in Abb. 4.3 dargestellt. Die Sparse-Set-Datenstruktur ermöglicht somit die folgenden Vorgänge:

- Sollen alle Entities gefunden werden, die über eine gegebene Komponente verfügen, muss lediglich über das Dense Array des zu dieser Komponente gehörenden Sparse Sets iteriert werden. Die Informationen liegen lückenlos hintereinander im Speicher, was für optimale Cache-Effizienz sorgt.

[99] Damit ist keine Aggregation oder Komposition im Sinne der objektorientierten Programmierung gemeint, sondern eine »Kopplungs-freie« Zuordnung: Weder Entities noch Komponenten haben »Kenntnis« voneinander, aber sie werden dennoch miteinander assoziiert.

- Sollen alle Komponenten eines Typs gelesen und/oder verändert werden, genügt die Iteration über das Element Array des entsprechenden Sparse Sets. Auch hier liegen die Daten kontinuierlich vor.
- Soll geprüft werden, ob ein gegebenes Entity über eine Komponente einer bestimmten Art verfügt, so werden die Identifier-Bits des Entitys genutzt, um in das Sparse Array des zur Komponente gehörenden Sparse Sets zu indizieren. Der dort gespeicherte Wert dient als Index in das zugehörige Dense Array. Wird dort der Wert gelesen, der den Identifier-Bits des Entitys entspricht, dann verfügt das Entity über die entsprechende Komponente.

Die Datenstruktur ist also für die häufigsten Anwendungsfälle, nämlich das Iterieren über relevante Entities und/oder Komponenten in Bezug auf die Cache-Nutzung optimiert. Im in der Praxis seltener auftretenden Fall, dass überprüft werden soll, ob ein Entity über eine bestimmte Komponente verfügt, wird der eben genannte Performance-Vorteil durch eine zusätzliche Indirektion »erkauft«. Ebenfalls nimmt diese Datenstruktur durch das Sparse Array deutlich mehr Speicherplatz in Anspruch als eigentlich nötig. Auch dies wird in Kauf genommen, um die Iterationsvorgänge so effizient wie möglich zu machen.

4.1.3 Verwaltung verschiedener Sparse Sets

Über eine Verwaltungsklasse namens ComponentHolder erfolgen Zugriffe auf die verschiedenen Sparse Sets. Dazu verwaltet diese Klasse eine Liste von Zeigern auf die entsprechenden Container. Damit dies möglich ist, wird mit sog. »Type-Erasure« gearbeitet. Das bedeutet, dass die strenge Typisierung von C++ umgangen wird, damit die verschiedenen Sparse Sets demselben Datentyp entsprechen, obwohl sie Container für unterschiedliche Datentypen darstellen. Die Problematik lässt sich verdeutlichen, wenn versucht wird, Zeiger auf mehrere std::vector<T>-Instanzen in einem Container abzulegen, wobei der Typparameter T jedoch bei jedem std::vector<T> verschieden sein kann: Unter C++ sind beispielsweise std::vector<int> und std::vector<std::string> verschiedene Datentypen und es gibt keinen Zeiger-Datentyp, der in der Lage ist, sowohl auf Container des einen als auch des anderen Typs zu zeigen. Eine Ausnahme stellt der Zeigertyp void* dar, der als »Universalzeiger« verstanden werden kann. Nutzt man diesen Zeigertyp, um das Typensystem von C++ zu umgehen, verzichtet man automatisch auf eine große Menge von Überprüfungsmechanismen, die der Compiler für gewöhnlich einsetzt, um die Korrektheit des Programms sicherzustellen. Für die korrekte Verwendung der »von Datentypen befreiten« Daten ist also der Programmierer bzw. die Programmiererin selbst verantwortlich. Hierbei auftretende Fehler führen zu sog. »undefined behavior«, wodurch

```
1  template<typename First, typename... Rest>
2  [[nodiscard]] auto get() const noexcept {
3    using ranges::views::filter,
4          ranges::views::transform,
5          ranges::views::zip;
6
7    return zip(getComponent<First>().indices(),
8               getComponent<First>().template elements<First>())
9      | filter([this](auto&& tuple) {
10         return (has<Rest>(std::get<0>(tuple)) && ...);
11     })
12     | transform([this](auto&& tuple) {
13         return std::forward_as_tuple(
14           std::get<0>(tuple), std::get<1>(tuple),
15           getComponent<Rest>()
16             .template get<Rest>(std::get<0>(tuple))...);
17     });
18 }
```

Listing 4.3: Abrufen einer Kombination von Komponenten

der gesamte Programmablauf unvorhersagbar bzw. unbestimmt wird, da der Compiler in solch einem Fall keinerlei Garantien mehr gibt.

Damit über die Klasse ComponentHolder auf die von dieser verwalteten Sparse Sets zugegriffen werden kann, wird auf Mittel der generischen – in Form von C++-Templates – sowie der funktionalen Programmierung zurückgegriffen. Beispielsweise zeigt Listing 4.3, wie ein »View«[100] über eine beliebige Kombination von Komponenten erzeugt werden kann. Die gezeigte Funktion get ist ein sog. »variadic template«, also ein Funktions-Template mit einer variablen Anzahl von Typparametern. Der Typparameter First sowie das Parameter-Pack Rest stellen dabei eine Liste von Datentypen von Komponenten dar (Zeile 1). Die Hilfsfunktion getComponent<T> gibt eine Referenz auf das Sparse Set zurück, welches die Komponenten vom Typ T enthält. Über diese Referenz kann auf das dazugehörige Dense Array (über die Member-Funktion indices) sowie das Element-Array (über die Member-Funktion elements<T>[101]) zugegriffen werden. In den Zeilen 7 und 8 werden das Dense-Array und das Element-Array der Komponententyps First mittels eines Zip-Views miteinander verbunden. Mit Hilfe eines Lambda-Ausdrucks werden in den Zeilen 9 bis 11 diejenigen Einträge des Views verworfen, welche Entities repräsentieren, die zwar über eine Komponente vom Typ First, jedoch nicht auch über alle Komponenten, deren Typen in Rest spezifiziert sind, verfügen. In den Zeilen 12 bis 17 werden sodann alle Referenzen auf die geforderten Komponenten »gesammelt« und als View

[100] Ein View ist ein iterierbares Hilfsobjekt, über welches Zugriff auf die referenzierten Objekte ermöglicht wird.

[101] Diese Funktion erfordert die nochmalige explizite Nennung des Datentyps. Dies ist notwendig aufgrund von Type-Erasure (s. o.) – der Container hat keine Kenntnis über den Datentyp der in ihm gespeicherten Daten.

```
const auto cameraTransformMatrix = // ...
mRenderer.clear(true, true);
mRenderer.beginFrame(cameraTransformMatrix);
for (auto&& [entity, dynamicSprite, transform] :
     mRegistry.components<DynamicSpriteComponent,
                          TransformComponent>()) {
  mRenderer.drawQuad(transform.position,
                     transform.rotation,
                     transform.scale,
                     *(dynamicSprite.shaderProgram),
                     *(dynamicSprite.sprite.texture),
                     dynamicSprite.sprite.textureRect,
                     dynamicSprite.color);
}
mRenderer.endFrame();
```

Listing 4.4: System zum Rendern dynamischer Sprites

über entsprechende Tupel zurückgegeben.
Das gezeigte Beispiel zeigt den Grad an Komplexität, den die Implementierung einer ECS-Architektur mit sich bringt – verschachtelte C++-Templates ziehen sich durch die gesamte Implementierung. Um die Code-Korrektheit so gut wie möglich sicherzustellen, wurden Unit-Tests sowie ein Werkzeug zur Überprüfung auf Speicher-Leaks sowie ungültige Speicherzugriffe verwendet[102]. Diese innere Komplexität wird in Kauf genommen, da sie nach außen hin eine komfortabel nutzbare Programmierschnittstelle bietet.

4.1.4 Funktionalität durch Systeme

Die bisher vorgestellten Bestandteile der ECS-Architektur – die Entities sowie die Komponenten – sind für die Verwaltung und Speicherung von Daten zuständig. Um Funktionalität implementieren zu können, kommen die in Abschnitt 3.4 vorgestellten Systeme zum Einsatz. Bei Systemen handelt es sich um Code, der über alle Entities, die über eine spezifizierte Kombination von Komponenten verfügen, iteriert. Die Komponenten können dabei ausgelesen und/oder verändert werden. Listing 4.4 zeigt beispielsweise eine vereinfachte Version des Systems, welches das Rendering dynamischer Sprites implementiert. Im Kern besteht der Code aus einer Schleife über die Entities, welche sowohl über eine DynamicSpriteComponent als auch eine TransformComponent verfügen. Zu beachten ist, dass die Daten gleichartiger Komponenten aufgrund der verwendeten Sparse-Set-Datenstruktur kontinuierlich im Speicher liegen. Damit soll die Cache-Effizienz beim Iterieren verbessert werden.

[102] Verwendet wurde das Werkzeug »memcheck« des Programms »valgrind«, siehe `https://valgrind.org/`, aufgerufen am 28.08.2021

```
Renderer renderer{ window };

// ...

// main loop:
while (/* ... */) {
    renderer.beginFrame();
    renderer.drawQuad(transform.matrix(),
                      shader,
                      texture,
                      textureRect,
                      color);
    // more calls to drawQuad()
    renderer.endFrame();
    renderer.swap();
}
```

Listing 4.5: API des Batch-Renderers

4.2 Rendering

Um die grafische Darstellung von Spielelementen zu ermöglichen, wird auf die OpenGL-Schnittstelle zurückgegriffen. Die Erzeugung eines Fensters sowie eines OpenGL-Kontexts wird dabei von der Bibliothek *glfw*[103] übernommen. Die Bibliothek *glad*[104] lädt zur Laufzeit Funktionszeiger zu den OpenGL-Funktionen, sodass diese für die Software zur Verfügung stehen.

Die implementierte Engine verfügt über einen Batch-Renderer, also über ein Software-Modul, welches die darzustellenden Elemente so gruppiert, dass möglichst wenig Kommunikation mit der Grafikkarte bzw. dem Grafikkartentreiber stattfinden muss. Der Renderer bietet dabei eine Schnittstelle an, die zunächst an »immediate mode OpenGL« (auch »legacy OpenGL« genannt) erinnert. Listing 4.5 verdeutlicht dies. Die API ähnelt dem aus älteren OpenGL-Anwendungen bekannten Muster von Aufrufen der Funktionen glBegin(), glColor3f(), glVertex2f(), glEnd() (und weiteren). Im Gegensatz zu diesen OpenGL-Befehlen bewirken jedoch die in Listing 4.5 gezeigten Aufrufe *nicht* die sofortige Kommunikation mit dem Grafiktreiber. Stattdessen führen Aufrufe der drawQuad()-Funktion dazu, dass die übergebenen Argumente in eine Liste gespeichert werden. Dabei handelt es sich um einen sog. »Command Buffer«. Der Command Buffer verfügt über eine konfigurierbare Maximalgröße. Sobald er komplett gefüllt ist, werden die gespeicherten Kommandos zunächst nach dem zu verwendenden Shader und – bei identischem Shader – nach der darzustellenden Textur sortiert. Der sortierte Command Buffer wird sodann in möglichst große »Bündel« (engl. *batches*) zusammengefasst, deren zugehörige Geometriedaten vorbereitet und jeweils als Ganzes an die Grafikkarte übertra-

[103] siehe `https://www.glfw.org/`, aufgerufen am 31.08.2021
[104] siehe `https://github.com/Dav1dde/glad`, aufgerufen am 31.08.2021

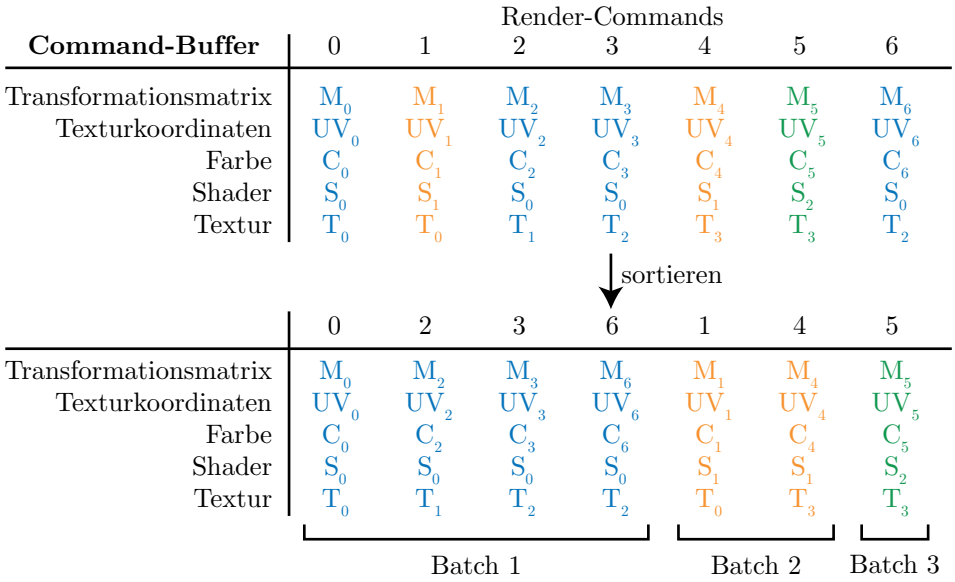

Abbildung 4.4: Arbeitsweise des Command-Buffers
Quelle: Abb. selbst erstellt

gen werden. So wird es möglich, eine große Zahl von Elementen, welche mit Hilfe verschiedener Shader und/oder Texturen dargestellt werden sollen, mit verhältnismäßig wenigen OpenGL-API-Aufrufen und vor allem möglichst wenigen Draw-Calls[105] anzuzeigen, was der Verbesserung der Performance dient. Abb. 4.4 illustriert, wie dadurch Draw-Calls eingespart werden können. Nach dem Sortieren der Render-Commands müssen lediglich zwischen den verschiedenen Batches die jeweils korrekten Shader und Texturen gebunden werden.

4.3 Input

Die Bibliothek glfw stellt einen plattformübergreifenden Zugriff auf übliche Eingabegeräte (Maus, Tastatur, Gamepad) zur Verfügung. Die so zur Verfügung gestellte Schnittstelle – welche in der Sprache C geschrieben ist – wird darüber hinaus weiter abstrahiert, um den Zugriff auf spielrelevante Informationen zu vereinfachen. In Listing 4.6 ist die öffentliche Schnittstelle für den Zugriff auf die Tastatur sowie die Maus gezeigt. Die Schnittstelle ermöglicht beispielsweise sowohl die Abfrage, ob eine bestimmte Taste zum Zeitpunkt der Abfrage gedrückt gehalten wird (Funktion `keyDown()`), als auch, ob sie während des aktuellen Frames herunter gedrückt wurde (Funktion `keyPressed()`).
In dieser ersten Implementierung wird vorerst auf die Unterstützung von Gamepads verzichtet. Diese ähnelt prinzipiell jedoch stark der Tastaturunterstützung, weshalb sie später leicht hinzugefügt werden kann.

[105] vgl. Abschnitt 2.10

```cpp
enum class Key : std::size_t {
  A = GLFW_KEY_A,
  B = GLFW_KEY_B,
  // ...
};

enum class MouseButton : std::size_t {
  Left = 0,
  Right = 1,
  // ...
};

class Input final {
public:
  Input();
  [[nodiscard]] bool keyDown(Key key) const;
  [[nodiscard]] bool keyPressed(Key key) const;
  [[nodiscard]] bool keyRepeated(Key key) const;
  [[nodiscard]] bool keyReleased(Key key) const;
  [[nodiscard]] glm::vec2 mousePosition() const;
  [[nodiscard]] bool mouseInsideWindow() const;
  [[nodiscard]] bool mouseDown(MouseButton button) const;
  [[nodiscard]] bool mousePressed(MouseButton button) const;
  [[nodiscard]] bool mouseReleased(MouseButton button) const;
private:
  // ...
};
```

Listing 4.6: Schnittstelle für den Zugriff auf Tastatur und Maus

4.4 Scripting

Viele Spiele(-Engines) ermöglichen die Ausführung von »externem« Programmcode während der Laufzeit. Im Regelfall ist dabei das zur Laufzeit stattfindende Interpretieren von Code gemeint – im Gegensatz zur Ausführung von Code, der bereits vor der Laufzeit in nativen Maschinencode übersetzt wurde. Die für diese Programmbestandteile verwendeten Sprachen werden üblicherweise als Scriptsprachen bezeichnet.
Die Definition des Begriffs »Scriptsprache« ist – vor allem im Kontext von Spiele-Engines – nicht eindeutig. Während Gregory Scriptsprachen definiert als »programming [languages] whose primary purpose is to permit users to control and customize the behavior of a software application«[106], definieren McShaffry und Graham Scriptsprachen als »anything higher level than C++ that is embedded into the core game engine«[107]. Als Vorteile beim Einsatz einer Scriptsprache nennen Sie[108]:

- »Rapid prototyping«: Komplexe Systeme lassen sich oft in einer Scriptsprache einfacher und kürzer implementieren als beispielsweise unter C++.

[106] Gregory, 2019, S. 1134
[107] McShaffry und Graham, 2013, S. 334
[108] vgl. McShaffry und Graham, 2013, S. 334 f.

Spiele oder Spiele-Engines, die sog. »hot reloading«, also das (erneute) Laden von Scripts zur Laufzeit unterstützen, verkürzen die Iterationszeit noch weiter. Ein neues Kompilieren des Spiel- oder Engine-Codes ist bei Änderungen an den Scripts nicht notwendig.

- Designfokus: Scriptsprachen sind i. d. R. leichter zu erlernen und können deshalb eher von Personen, die am Game-Design arbeiten, benutzt werden, um ihre Ideen umzusetzen. Auch Möglichkeiten des Moddings werden durch Scriptsprachen ermöglicht.

Neben diesen Vorteilen bringen Scriptsprachen jedoch auch Nachteile mit sich:

> »Scripts are generally very slow when compared to C++, and they take up more memory, which is one of the main disadvantages to using a scripting language. [...] Crossing the boundary between C++ and script is also very slow, especially if you're marshalling a large amount of data.«[109]

Für die im Rahmen dieser Arbeit implementierte Engine wurde Lua als Scriptsprache gewählt[110]. Im Gegensatz zu anderen üblicherweise verwendeten Sprachen wie z. B. Python oder C# beansprucht der Lua-Interpreter weniger Systemleistung und deutlich weniger Speicher. Die Sprache selbst bringt jedoch auch weniger Funktionalität mit. Da der Scriptcode allerdings hauptsächlich Engine-spezifischen Code ausführt, werden externe Bibliotheken, beispielsweise für den Zugriff auf das Dateisystem, nicht benötigt.
Um die verschiedenen Scripts voneinander zu trennen und unerwünschte Seiteneffekte zwischen diesen zu verhindern[111], wird für jedes geladene Script ein separater Lua-Interpreter gestartet. Scripts können sodann an Entities[112] »angehängt« (engl. *attached*) werden. Dafür existiert eine Komponente, welche einen Zeiger auf das auszuführende Script enthält. In Anlehnung an die C#-Schnittstelle, die von der Unity-Engine bereitgestellt wird, ruft die Engine automatisiert Funktionen mit vordefinierten Namen auf. Beispielsweise wird die Lua-Funktion `onAttach()` aufgerufen, sobald ein Script an ein Entity angehängt wird. Die Funktion `update()` wird dagegen einmal pro Frame für jedes Entity aufgerufen, welches über das jeweilige Script verfügt. Verfügen mehrere Entities über dasselbe Script, so »teilen« sich diese – wie beschrieben – einen gemeinsamen Lua-Interpreter. Das hat zur Folge, dass sie gemeinsam auf globale Variablen innerhalb des Lua-Scripts zugreifen können. Durch diesen

[109] McShaffry und Graham, 2013, S. 336
[110] siehe `https://www.lua.org/`, aufgerufen am 16.09.2021
[111] Man spricht hierbei auch von »Sandboxing«.
[112] siehe Abschnitt 3.4

```lua
spriteSheetGUID = "22171ab7-6b8b-474f-bf24-366724625641"
shaderGUID = "b520f0eb-1756-41e0-ac07-66c3338bc594"

-- onAttach is called once for every entity with this script
function onAttach(entity)
    --[[ this function attaches all required components
         to this entity to make it display the first sprite
         of a sprite sheet ]]--

    spriteSheet = c2k.assets.spriteSheet(spriteSheetGUID)
    local transform = entity:attachTransform()
    transform.scale.x = 60
    transform.scale.y = 60

    local shader = c2k.assets.shaderProgram(shaderGUID)
    local sprite = entity:attachDynamicSprite()
    sprite.shaderProgram = shader
    sprite.texture = spriteSheet.texture
    currentIndex = 1
    sprite.textureRect = spriteSheet.frames[currentIndex].rect

    entity:attachRoot()
    input = c2k.getInput()
end

-- update is called every frame for this entity
function update(entity)
    if input:keyPressed(Key.Enter) then
        -- display the next sprite of the sprite sheet
        currentIndex = currentIndex + 1
        if currentIndex == #spriteSheet.frames + 1 then
            currentIndex = 1
        end
        sprite = entity:getDynamicSprite()
        sprite.textureRect = spriteSheet.frames[currentIndex].rect
    end
end
```

Listing 4.7: Lua-Beispielscript

Mechanismus ist eine Art »shared memory« zwischen gleichartigen Scripts möglich.

Aus Gründen des Umfangs wird an dieser Stelle auf eine detaillierte Beschreibung der Lua-Schnittstelle verzichtet. In Listing 4.7 ist ein Beispiel-Script zu sehen, welches einen Teil der von der Engine zur Verfügung gestellten Schnittstelle demonstriert.

4.5 Weitere Systeme

Eine Spiele-Engine umfasst i. d. R. noch deutlich mehr Komponenten bzw. Subsysteme als die hier vorgestellten:

- Physik-Simulation

- Netzwerk
- (De-)Serialisierung
- Audio (Musik und Sound-Effekte)
- Font-Rendering
- Lokalisation
- Post-Processing/Render-Targets
- Video-Unterstützung (z. B. für Zwischensequenzen)
- Partikelsysteme[113] und andere Grafik-Effekte
- HDR-Unterstützung
- und viele mehr

Hinzu kommt noch der Bereich der »Tools«: Mit Hilfe diverser Werkzeuge und Editoren können Assets sowie Levels für Spiele erstellt werden. Auch diese gehören im weiteren Sinne zu einer Spiele-Engine. Aufgrund der gewählten Fokussierung der vorliegenden Arbeit sowie aus Zeit- sowie Umfangsgründen wird auf eine detailliertere Beschreibung dieser Bestandteile verzichtet und an dieser Stelle auf Gregory, 2019 verwiesen.

4.6 Demonstration

Die folgenden Abschnitte demonstrieren die Fähigkeiten der implementierten Engine. Zunächst wird gezeigt, dass diese in der Lage ist, mit Hilfe der eingesetzten ECS-Architektur eine große Anzahl von Objekten zu verwalten. Danach belegt die Realisierung eines Spiels ihre Praxistauglichkeit.

4.6.1 Technische Demonstration

Abb. 4.5 zeigt eine Beispielanwendung, die mit Hilfe der implementierten Engine erstellt wurde. Zu sehen ist ein Partikelsystem mit ca. 100.000 Partikeln sowie mehreren, größeren Sprites. Das Verhalten dieser Sprites, z. B. deren Drehung, wird durch Lua-Scripts definiert. Durch einen Tastendruck kann das in der Bildschirmmitte zu sehende Sprite ausgetauscht werden. Dabei werden verschiedene Ausschnitte desselben Sprite-Sheets[114] gezeigt. Auch dies erfolgt über ein Lua-Script, welches in Listing 4.7 zu sehen ist.

[113] Einfache Partikelsysteme stehen in der Engine bereits zur Verfügung (siehe Abschnitt 4.6).
[114] auch Texture-Atlas genannt

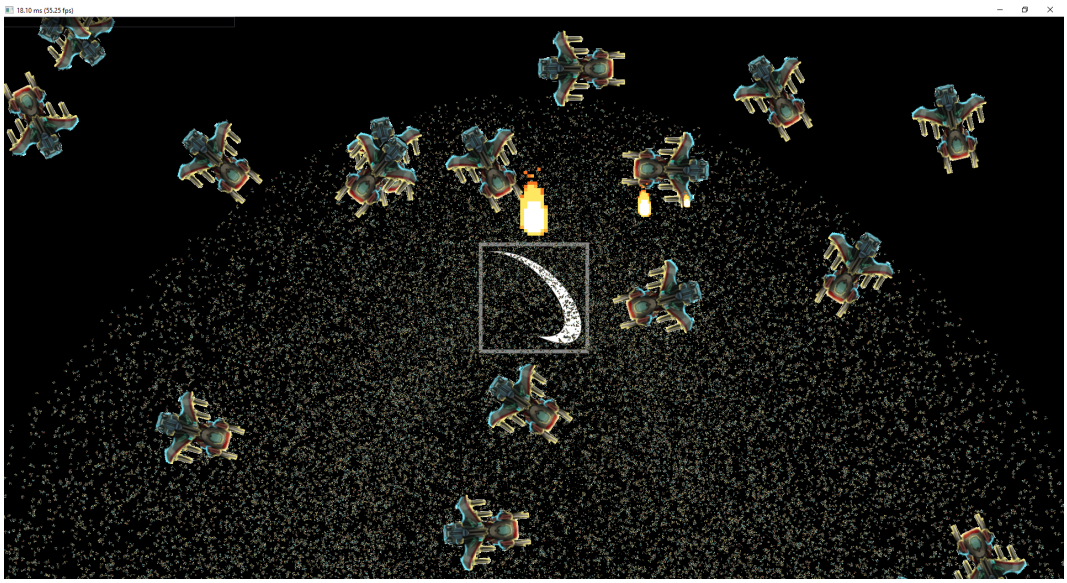

Abbildung 4.5: Demonstration der implementierten Engine
Quelle: Abb. selbst erstellt

Die gezeigte Demonstrations-Szene läuft auf dem Testsystem mit ca. 60 Bildern pro Sekunde. Dies ist darin begründet, dass sämtliche Geometriedaten in jedem Frame erneut an die Grafikkarte übermittelt werden. Vor allem Partikelsysteme profitieren jedoch stark davon, einen Großteil der nötigen Berechnungen auf der Grafikkarte durchzuführen.
Durch die verwendete ECS-Architektur[115] muss für Partikelsysteme kein separates Pooling[116] eingesetzt werden: Jedes Partikel ist ein eigenständiges Entity. Durch die automatisierte und effiziente Verwaltung der zugrunde liegenden Datenstrukturen findet für alle Entities ein Pooling-ähnliches Verfahren statt, ohne dass dieses explizit programmiert werden muss. Die Implementierung eines solchen Partikelsystems ist daher verhältnismäßig einfach.

4.6.2 Implementierung eines Flappy Bird-Klons

Obwohl die gezeigte Anwendung demonstriert, dass die implementierte Engine in der Lage ist, eine große Anzahl von Objekten gleichzeitig zu verwalten und darzustellen, so ist damit jedoch nicht gezeigt, dass es mit ihr möglich ist, ein Spiel zu realisieren. Aus diesem Grund wurde eine weitere Anwendung erstellt, die genau dies demonstrieren soll. Bei dieser Anwendung handelt es sich um eine Implementierung des Smartphone-Spiels »Flappy Bird«[117]. Beim aktuellen Entwicklungsstand der Engine lassen sich Anwendungen mit minimalem

[115] siehe Abschnitt 3.4
[116] Unter Pooling versteht man die Wiederverwendung von Objekten aus einem anfangs erzeugten Objekt-Pool statt deren häufiger Zerstörung und Neuerstellung. Dies wird zur Effizienzsteigerung genutzt.
[117] siehe `https://en.wikipedia.org/wiki/Flappy_Bird`, aufgerufen am 31.10.2021

```
1  #include <Application.hpp>
2  #include <AssetDatabase.hpp>
3
4  using namespace c2k;
5
6  class FlappyBird : public Application {
7    void setup() noexcept override {
8      const auto scriptGUID =
9        GUID::fromString("1b0b54bf-d36a-4067-bd60-cc69900bb9bc");
10     mAssetDatabase.loadFromList(
11       AssetDatabase::assetPath() / "assets.json");
12     mRegistry.createEntity(ScriptComponent{
13         &mAssetDatabase.scriptMutable(scriptGUID) });
14     mRenderer.setClearColor(Color{ 0.5f, 0.8f, 0.8f, 1.0f });
15   }
16
17   void update() noexcept override { }
18 };
19
20 int main() {
21   FlappyBird game;
22   game.run();
23 }
```

Listing 4.8: Der C++-Code der Flappy Bird-Implementierung

Einsatz von C++ erstellen, sodass der Großteil der Logik in Lua implementiert werden kann. Darüber hinaus nutzt die Engine Konfigurationsdateien im JSON-Format. Langfristig soll das Ziel sein, dass bei der Entwicklung von Spielen mit der vorliegenden Engine gänzlich auf C++ verzichtet werden kann, um die Zugänglichkeit zu verbessern.

C++-Code der Flappy Bird-Implementierung

Listing 4.8 zeigt den benötigten C++-Code der Flappy Bird-Implementierung. Die Engine bietet eine Klasse namens Application an, von der die FlappyBird-Klasse erbt und deren virtuelle setup()- sowie update()-Funktionen überschreibt. Innerhalb der setup()-Funktion wird zunächst die Konstante scriptGUID definiert. Dabei handelt es sich um eine Identifikationsnummer (GUID von engl. »globally unique identifier«[118]), welche das Haupt-Lua-Script eindeutig identifiziert, welches die Spiellogik enthält. Die explizite Angabe solcher Identifikationsnummern im Code ist dabei jedoch als Zwischenlösung anzusehen: Ein langfristiges Ziel ist es, dass diese Nummern von der Engine transparent generiert sowie verwaltet werden.
In Zeile 10 des gezeigten Quelltextes wird dann zunächst die Liste aller benötigten Assets (Texturen, Sprite-Sheets etc.) geladen und daraufhin auch die

[118] siehe `https://en.wikipedia.org/wiki/Universally_unique_identifier`, aufgerufen am 31.10.2021

Assets selbst. Nach diesem Schritt befinden sich alle Assets jeweils (abhängig von ihrem Typ) entweder im CPU- oder Video-RAM und können – identifiziert über ihre GUIDs – über die Instanz der AssetDatabase-Klasse genutzt werden.
In Zeile 12 wird der Spielwelt ein Entity hinzugefügt, welches über eine ScriptComponent-Instanz verfügt. Durch das Vorhandensein dieser Komponente auf einem Entity wird das entsprechende Script von der Engine automatisiert ausgeführt. In Zeile 14 wird die Hintergrundfarbe des Spiels definiert.
Zu bemerken ist, dass der Funktionskörper der update()-Funktion leer bleibt: Da die gesamte Spiellogik Lua-seitig implementiert ist und die Engine das entsprechende Lua-Script automatisiert ausführt, ist C++-seitig kein zusätzlicher Code notwendig.
In der main()-Funktion wird eine Instanz der FlappyBird-Klasse erzeugt und die Anwendung durch den Aufruf der Member-Funktion run() zur Ausführung gebracht.

Lua-Code der Flappy Bird-Implementierung

Lua unterstützt keine Objektorientierung. Aufgrund der in der Sprache enthaltenen Datenstruktur der *Tabelle* lässt sich jedoch ein objektorientierter Programmierstil »simulieren«. Bei diesen Tabellen handelt es sich um assoziative Container, vergleichbar mit *Dictionary*- bzw. *Map*-Datenstrukturen anderer Programmiersprachen. Durch den Einsatz sog. »Meta-Tabellen«[119] können Tabelleninstanzen als Klassen dienen. Für detailliertere Informationen diesbezüglich sei an dieser Stelle auf die Lua-Dokumentation verwiesen[120].
Durch diesen Ansatz lassen sich Spiele, die mit Hilfe der vorliegenden Engine geschrieben werden, ähnlich strukturieren wie beispielsweise Spiele, die in C# mit der Unity-Engine erstellt werden. Listing 4.9 zeigt das Haupt-Script des implementierten Spiels. Die onAttach()-Funktion wird dabei von der Engine aufgerufen, sobald das Script an ein Entity angehängt wird. In dieser Funktion werden in globalen Variablen Referenzen auf das Input- und das Zeit-System der Engine gespeichert, um diese leichter zugänglich zu machen. Außerdem wird eine Instanz der Klasse[121] Game erzeugt.
Einzelne Spiel-Elemente werden durch Klassen modelliert. Dazu gehört z. B. der steuerbare Vogel oder die verschiedenen Hintergrund-Ebenen, welche unterschiedlich schnell bewegt werden, um einen Parallax-Effekt zu erzeugen. Aus Gründen des Umfangs wird an dieser Stelle auf weitere Code-Beispiele

[119] siehe `https://www.lua.org/pil/13.html`, aufgerufen am 31.10.2021
[120] siehe »Programming in Lua«, `https://www.lua.org/pil/contents.html`, aufgerufen am 31.10.2021
[121] Wie beschrieben unterstützt Lua keine objektorientierte Programmierung und somit auch keine Klassen. Da die Tabellen und Meta-Tabellen hier jedoch wie Klassen eingesetzt werden, wird auf eine strikte Unterscheidung im Text verzichtet.

```lua
require("Game")

function onAttach(_)
    input = c2k.getInput()
    time = c2k.getTime()
    game = Game.new(Vec2.new(800, 600))
end

function update(_)
    game:update()
end
```

Listing 4.9: Haupt-Script der Flappy Bird-Implementierung

verzichtet, jedoch kann der diesem Abschnitt zugrunde liegende Stand des Lua-Quellcodes online abgerufen werden unter `https://bit.ly/3vYBFPO`.
In Abb. 4.6 ist ein Screenshot des vorläufigen Endergebnisses zu sehen. Der Vogel kann mittels Mausklick oder Leertaste gesteuert werden. Wird ein Röhren-Paar erfolgreich passiert, wird die Punktzahl erhöht (Bildmitte oben) und aus der oberen der passierten Röhren fallen goldene Sterne. Diese sind – analog zum vorherigen Abschnitt – mithilfe eines Partikelsystems implementiert. Berührt der Vogel den Boden oder eine der Röhren, stürzt er ab. Durch einen Mausklick kann danach das Spiel neu gestartet werden. Eine Video-Demonstration des Spiels ist online verfügbar unter `https://youtu.be/uHyaiQdBEQ0`.

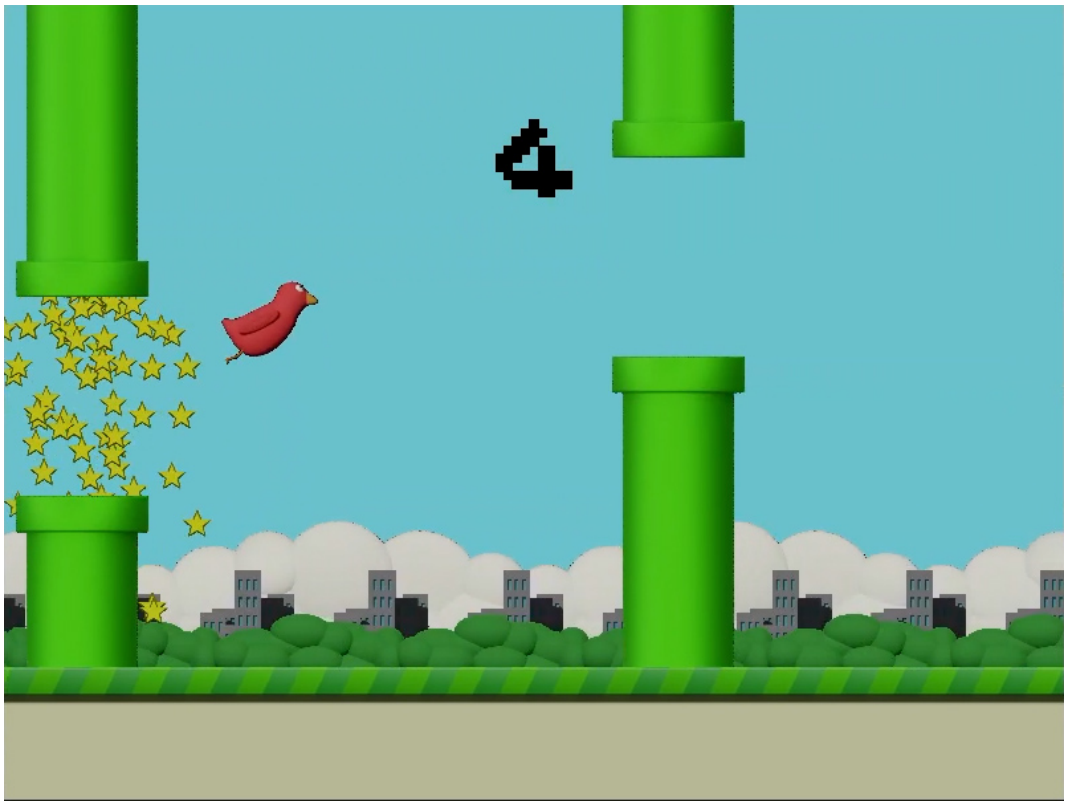

Abbildung 4.6: Screenshot der Flappy Bird-Implementierung
Quelle: Abb. selbst erstellt

5 Fazit

Es lässt sich zusammenfassend sagen, dass die in Abschnitt 2 vorgestellten Programmiertechniken deutliche Auswirkungen auf die Performance von Software haben. In zeitkritischen Anwendungen sollten diese daher berücksichtigt werden. Die Erfahrungen, die bei der in Abschnitt 4 vorgestellten Implementierung gemacht wurden, belegen dabei jedoch auch, dass eine konsequente Berücksichtigung dieser Techniken die »innere Komplexität« der Software drastisch erhöht: Um nach außen eine leicht verwendbare Programmierschnittstelle anbieten zu können, ohne auf die Effizienz zu verzichten, sind fortgeschrittene Programmiertechniken wie z. B. C++-Template-(Meta-)Programming und Type-Erasure erforderlich. Programmkorrektheit lässt sich nur durch systematisches Testen und die Verwendung von zusätzlichen Analysewerkzeugen validieren.
Da bei Spiele-Projekten eines geringen Umfangs derartige Optimierungen im Regelfall unnötig sind, kann in diesem Kontext auf sie verzichtet werden. Ist eine effiziente Ausnutzung der Hardware jedoch notwendig, um das gewünschte Spielerlebnis erzeugen zu können, sind diese Techniken unverzichtbar. Vor allem für Firmen, welche Spiele mit Hilfe eigens entwickelter Engines realisieren oder die selbst eine Engine kommerziell vertrieben, sind die vorgestellten Techniken daher essenziell. Obwohl die Entwicklung von Spielen immer zugänglicher wird, so sind Personen, welche die zugrunde liegende Technologie entwickeln, auch weiterhin auf genaue Kenntnis der Hardware angewiesen. Auch bei der Verwendung weiter abstrahierter Programmiersprachen können die vorgestellten Inhalte hilfreich sein und sollten deshalb für alle Entwicklerinnen und Entwickler von Interesse sein.
In Bezug auf die Implementierung einer eigenen Engine ist zu erwähnen, dass das C++-Ökosystem sehr umfangreich ist, und daher zu empfehlen ist, auf bereits existierende und erprobte Implementierungen von Submodulen zurückzugreifen. Im Rahmen dieser Arbeit wurde beispielsweise eine eigene ECS-Architektur implementiert, obwohl dafür bereits mehrere frei verwendbare Bibliotheken existieren, welche auch in kommerziell erfolgreichen Spielen Verwendung finden. Liegt der eigene Fokus also *nicht* auf dem Erlernen der Architektur, so lässt sich durch die Verwendung einer solchen Bibliothek potenziell die Programmkorrektheit verbessern sowie die Gesamtdauer der Entwicklung reduzieren.

Ausblick

Auch über den Hochschulkontext hinaus soll an der begonnenen Engine-Implementierung weiter gearbeitet werden. Die Zielsetzung ist dabei allerdings nicht die Veröffentlichung kommerzieller Spiele: Eine Spiele-Engine berührt zahlreiche Teilgebiete aus der Software-Entwicklung und stellt ein Projekt mit

potenziell endlosem Umfang dar. Primär soll der Fokus bei der Weiterentwicklung daher auf der persönlichen Weiterbildung sowie der Weitergabe des dabei gewonnenen Wissens liegen. Die Kenntnisse und Fertigkeiten, die dabei erlangt werden können, lassen sich universell auf andere Software-Projekte übertragen und sind deshalb von unschätzbarem Wert.

Anhang

Erste Zeitmessung zum Caching:

```
#include <range/v3/all.hpp>
#include <benchmark/benchmark.h>
#include <cstdint>
#include <new>
#include <algorithm>
#include <vector>
#include <iostream>

struct Node {
    uint64_t data;
    Node* next;
};

constexpr auto cacheLineSize =
    std::hardware_constructive_interference_size;

static void BM_OneCacheLine(benchmark::State &state) {
    using namespace ranges;
    using namespace ranges::views;
    constexpr auto alignment =
        static_cast<std::align_val_t>(cacheLineSize);
    constexpr auto numNodes = cacheLineSize / sizeof(Node);
    auto sequence =
        ints(std::size_t{ 0 }, numNodes) | ranges::to_vector;
    shuffle(sequence);

    Node* nodes =
        static_cast<Node*>(operator new[](sizeof(Node) * numNodes,
                           alignment));
    Node* lastNode = nullptr;
    for (std::size_t i = 0; i < sequence.size(); ++i) {
        Node* const currentNode = &nodes[sequence[i]];
        *currentNode = Node{ .data = i, .next = nullptr };
        if (i > 0) {
            lastNode->next = currentNode;
        }
        lastNode = currentNode;
    }
    for (auto _ : state) {
        uint64_t sum = 0ULL;
        Node* current = &nodes[sequence[0]];
        while (current != nullptr) {
            sum += current->data;
            current = current->next;
        }
```

```
46          benchmark::DoNotOptimize(sum);
47      }
48      operator delete[](nodes, alignment);
49  }
50  BENCHMARK(BM_OneCacheLine);
51
52  static void BM_MultipleCacheLines(benchmark::State &state) {
53      using namespace ranges;
54      using namespace ranges::views;
55      constexpr auto alignment =
56          static_cast<std::align_val_t>(cacheLineSize);
57      constexpr auto numNodes = cacheLineSize / sizeof(Node);
58      auto sequence =
59          ints(std::size_t{ 0 }, numNodes) | ranges::to_vector;
60      shuffle(sequence);
61
62      Node* nodes =
63          static_cast<Node*>(operator new[](cacheLineSize * numNodes,
64                                            alignment));
65      Node* lastNode = nullptr;
66      for (std::size_t i = 0; i < sequence.size(); ++i) {
67          Node* const currentNode = &nodes[sequence[i] * numNodes];
68          *currentNode = Node{ .data = i, .next = nullptr };
69          if (i > 0) {
70              lastNode->next = currentNode;
71          }
72          lastNode = currentNode;
73      }
74      for (auto _ : state) {
75          uint64_t sum = 0ULL;
76          Node* current = &nodes[sequence[0] * numNodes];
77          while (current != nullptr) {
78              sum += current->data;
79              current = current->next;
80          }
81          benchmark::DoNotOptimize(sum);
82      }
83      operator delete[](nodes, alignment);
84  }
85  BENCHMARK(BM_MultipleCacheLines);
86
87  BENCHMARK_MAIN();
```

Zweite Zeitmessung zum Caching:

```cpp
#include <range/v3/all.hpp>
#include <benchmark/benchmark.h>
#include <vector>
#include <limits>
#include <iostream>

static constexpr auto cacheLineSize =
    std::hardware_constructive_interference_size;

static auto createData(std::size_t bytes) {
    std::vector<char> buffer(bytes);
    for (std::size_t i{ 0 }; i < buffer.size(); ++i) {
        buffer[i] = static_cast<char>(
            i % std::numeric_limits<char>::max());
    }
    auto sequence =
        ranges::views::ints(std::size_t{ 0 },
                            buffer.size()) | ranges::to_vector;
    ranges::shuffle(sequence);
    return std::make_pair(buffer, sequence);
}

static void BM_StepSizeOne(benchmark::State &state) {
    const auto bytesToAllocate{
        static_cast<size_t>(state.range(0)) };
    const auto [buffer, sequence] = createData(bytesToAllocate);
    for (auto _ : state) {
        std::size_t sum{ 0 };
        for (std::size_t i{ 0 }; i < buffer.size(); ++i) {
            const std::size_t index{
                i * cacheLineSize % buffer.size()
                + i * cacheLineSize / buffer.size() };
            benchmark::DoNotOptimize(index);
            sum += buffer[i];
        }
        benchmark::DoNotOptimize(sum);
    }
    benchmark::DoNotOptimize(sequence);
}
BENCHMARK(BM_StepSizeOne)->DenseRange(
    512 * 1024,
    32 * 1024 * 1024,
    512 * 1024);

static void BM_StepSizeSixtyFour(benchmark::State &state) {
    const auto bytesToAllocate{
        static_cast<size_t>(state.range(0)) };
```

```
48      const auto [buffer, sequence] = createData(bytesToAllocate);
49      for (auto _ : state) {
50          std::size_t sum{ 0 };
51          for (std::size_t i{ 0 }; i < buffer.size(); ++i) {
52              const std::size_t index{
53                  i * cacheLineSize % buffer.size()
54                  + i * cacheLineSize / buffer.size() };
55              benchmark::DoNotOptimize(index);
56              sum += buffer[index];
57          }
58          benchmark::DoNotOptimize(sum);
59      }
60      benchmark::DoNotOptimize(sequence);
61  }
62  BENCHMARK(BM_StepSizeSixtyFour)->DenseRange(
63      512 * 1024,
64      32 * 1024 * 1024,
65      512 * 1024);
66
67  static void BM_RandomAccessPattern(benchmark::State &state) {
68      const auto bytesToAllocate{
69          static_cast<size_t>(state.range(0)) };
70      const auto [buffer, sequence] = createData(bytesToAllocate);
71      for (auto _ : state) {
72          std::size_t sum{ 0 };
73          for (std::size_t i{ 0 }; i < buffer.size(); ++i) {
74              const std::size_t index{
75                  i * cacheLineSize % buffer.size()
76                  + i * cacheLineSize / buffer.size() };
77              benchmark::DoNotOptimize(index);
78              sum += buffer[sequence[i]];
79          }
80          benchmark::DoNotOptimize(sum);
81      }
82      benchmark::DoNotOptimize(sequence);
83  }
84  BENCHMARK(BM_RandomAccessPattern)->DenseRange(
85      512 * 1024,
86      32 * 1024 * 1024,
87      512 * 1024);
88
89  BENCHMARK_MAIN();
```

Messungen zum Pre-Fetching:

```cpp
#include <vector>
#include <iostream>
#include <string_view>
#include <random>
#include <cstdlib>
#include <cstdint>

enum class Mode {
    Sequential,
    Random,
    Invalid,
};

static Mode evaluateMode(std::string_view modeString) {
    if (modeString == "-seq") {
        std::cout << "Sequential mode\n";
        return Mode::Sequential;
    }
    if (modeString == "-rand") {
        std::cout << "Random mode\n";
        return Mode::Random;
    }
    std::cout << "Invalid mode: " << modeString << '\n';
    return Mode::Invalid;
}

int main(int argc, char** argv) {
    if (argc < 3) {
        std::cout << "Usage: " << argv[0]
                  << " (-seq|-rand) dataSize [stride]\n";
        return EXIT_FAILURE;
    }
    Mode mode = evaluateMode(argv[1]);
    const size_t dataSize = static_cast<size_t>(atoi(argv[2]))
                                                * 1024 * 1024;
    const size_t stride = argc > 3 ? atoi(argv[3]) : 1;
    constexpr size_t numIterations = 20;
    std::cout << "Iterations: " << numIterations << "\nSize: "
              << dataSize << " Bytes\n";
    if (mode == Mode::Sequential) {
        std::cout << "Stride: " << stride << '\n';
    }
    std::vector<uint8_t> buffer(dataSize, 1);
    std::random_device randomDevice;
    std::mt19937_64 randomGenerator{ randomDevice() };
    std::uniform_int_distribution<size_t> distribution{
        static_cast<size_t>(0), dataSize - 1 };
```

```
48      size_t sum{ 0 };
49      for (size_t iter{ 0 }; iter < numIterations; ++iter) {
50          size_t index{ 0 };
51          for (size_t i{ 0 }; i < dataSize; ++i) {
52              sum += buffer[index];
53              if (mode == Mode::Sequential) {
54                  index = (index + stride) % dataSize;
55              } else if (mode == Mode::Random) {
56                  index = distribution(randomGenerator);
57              }
58          }
59      }
60      std::cout << sum << '\n';
61      return EXIT_SUCCESS;
62  }
```

Messungen zur Branch-Prediction:

```cpp
#include <benchmark/benchmark.h>
#include <vector>
#include <utility>
#include <random>
#include <cmath>

static constexpr size_t numCalculations = 100;

static void BM_BranchPrediction(benchmark::State& state) {
    std::random_device randomDevice;
    std::mt19937 randomEngine{ randomDevice() };
    std::uniform_int_distribution distribution{ -100, 100 };
    std::uniform_int_distribution positiveDistribution{ 1, 99 };
    std::vector<std::pair<int, int>> calculations;
    calculations.reserve(numCalculations);
    for (size_t i{ 0 }; i < numCalculations; ++i) {
        auto x1 = distribution(randomEngine);
        auto x2 = distribution(randomEngine);
        calculations.emplace_back(-x1 - x2, x1 * x2);
    }
    for (auto _ : state) {
        const auto percentage = static_cast<int>(state.range(0));
        for (const auto& pair : calculations) {
            const auto roll = positiveDistribution(randomEngine);
            const double p = pair.first;
            const double q = pair.second;
            int result;
            if (roll < percentage) {
                result = static_cast<int>(-p / 2.0 +
                    std::sqrt((p / 2.0) * (p / 2.0) - q));
            } else {
                result = static_cast<int>(-p / 2.0 -
                    std::sqrt((p / 2.0) * (p / 2.0) - q));
            }
            benchmark::DoNotOptimize(result);
        }
    }
}
BENCHMARK(BM_BranchPrediction)->DenseRange(0, 100, 1);

BENCHMARK_MAIN();
```

Messungen zum False-Sharing:

```
#include <benchmark/benchmark.h>
#include <thread>
#include <atomic>
#include <vector>
#include <algorithm>
#include <numeric>
#include <random>

static constexpr unsigned numThreads{ 8U };

static auto createRandomNumbers(std::size_t count) {
    std::vector<int> result;
    result.resize(count);
    std::iota(result.begin(), result.end(), 0);
    std::random_device randomDevice;
    std::mt19937 randomEngine{ randomDevice() };
    std::shuffle(result.begin(), result.end(), randomEngine);
    return result;
}

struct Accumulator {
    std::size_t value{ 0 };
};

struct alignas(64) AlignedAccumulator {
    std::size_t value{ 0 };
};

static void accumulate(const std::vector<int>& numbers,
                       std::size_t& accumulator) {
    for (const auto number : numbers) {
        accumulator += number;
    }
}

template<typename AccumulatorType>
static void accumulateMultithreaded(
    const std::vector<int>& numbers) {
    AccumulatorType accumulators[numThreads];
    std::thread threads[numThreads];
    for (unsigned i{ 0U }; i < numThreads; ++i) {
        threads[i] = std::thread{ accumulate,
                                  std::cref(numbers),
                                  std::ref(accumulators[i].value)
                                };
    }
    for (auto& thread : threads) {
```

```
48          thread.join();
49      }
50      for (auto& accumulator : accumulators) {
51          benchmark::DoNotOptimize(accumulator.value);
52      }
53  }
54
55  static void BM_AccumulateFalseSharing(benchmark::State& state) {
56      std::size_t size = state.range(0);
57      const auto numbers = createRandomNumbers(size);
58      for (auto _ : state) {
59          accumulateMultithreaded<Accumulator>(numbers);
60      }
61  }
62  BENCHMARK(BM_AccumulateFalseSharing)->Range(1, 10000000);
63
64  static void BM_AccumulateNoFalseSharing(benchmark::State& state) {
65      std::size_t size = state.range(0);
66      const auto numbers = createRandomNumbers(size);
67      for (auto _ : state) {
68          accumulateMultithreaded<AlignedAccumulator>(numbers);
69      }
70  }
71  BENCHMARK(BM_AccumulateNoFalseSharing)->Range(1, 10000000);
72
73  BENCHMARK_MAIN();
```

Messungen zu den Auswirkungen von Heap-Allocations:

```cpp
#include <vector>
#include <list>
#include <random>
#include <string_view>
#include <chrono>
#include <format>
#include <iostream>
#include <limits>
#include <cstdint>

struct Statistics {
    std::size_t numLoops;
    std::size_t numAllocated;
    std::size_t numUsed;
    double elapsedTime;
};

static constexpr std::size_t maxCount{ 2000 };
static std::random_device gRandomDevice{};
static std::mt19937_64 gRandomEngine{ gRandomDevice() };

static std::size_t getRandomNumber(std::size_t max) {
    std::uniform_int_distribution dist{ std::size_t{ 0 }, max };
    return dist(gRandomEngine);
}

template<typename InnerLoopFunc>
static Statistics benchmark(InnerLoopFunc innerLoopFunc,
                            double benchmarkDuration) {
    using Clock = std::chrono::high_resolution_clock;
    using Duration = std::chrono::duration<double>;
    auto statistics = Statistics{
        .numLoops{ 0 },
        .numAllocated{ 0 },
        .numUsed{ 0 },
        .elapsedTime{ 0.0 },
    };
    auto start = Clock::now();
    double elapsed;
    while ((elapsed = (Duration(Clock::now() - start)).count())
           < benchmarkDuration) {
        innerLoopFunc(statistics);
        ++statistics.numLoops;
    }
    statistics.elapsedTime = elapsed;
    return statistics;
}
```

```
48
49 static void printResults(std::string_view benchmarkName,
50                           const Statistics& statistics) {
51     std::cout << std::format(
52             "------{}------\n"
53             "num loops: {}\n"
54             "num allocated: {}\n"
55             "num used: {}\n"
56             "% used: {}\n"
57             "frame time: {} s\n"
58             "fps: {}\n"
59             "elapsed time: {} s\n"
60             "--------------------------\n\n",
61             benchmarkName, statistics.numLoops,
62             statistics.numAllocated, statistics.numUsed,
63             100.0 * static_cast<double>(statistics.numUsed)
64             / static_cast<double>(statistics.numAllocated),
65             statistics.elapsedTime / statistics.numLoops,
66             statistics.numLoops / statistics.elapsedTime,
67             statistics.elapsedTime);
68 }
69
70 template<template<typename> typename Container,
71          bool doReserve,
72          bool hasCapacity>
73 static void innerLoop(Statistics& statistics) {
74     using ValueType = std::uint16_t;
75     Container<ValueType> buffer;
76     if constexpr (doReserve)
77         buffer.reserve(maxCount);
78     const auto numElements = getRandomNumber(maxCount);
79     for (std::size_t i{ 0 }; i < numElements; ++i) {
80         buffer.push_back(
81             static_cast<ValueType>(
82                 getRandomNumber(
83                     std::numeric_limits<ValueType>::max())));
84     }
85     if constexpr (hasCapacity)
86         statistics.numAllocated += buffer.capacity();
87     else
88         statistics.numAllocated += buffer.size();
89     statistics.numUsed += buffer.size();
90 }
91
92 int main() {
93     constexpr double duration{ 5.0 };
94     const auto vector = innerLoop<std::vector, false, true>;
95     const auto vectorReserve = innerLoop<std::vector, true, true>;
```

```
96      const auto list = innerLoop<std::list, false, false>;
97      printResults("std::vector w/o reserve",
98                   benchmark(vector, duration));
99      printResults("std::vector w reserve",
100                  benchmark(vectorReserve, duration));
101     printResults("std::list",
102                  benchmark(list, duration));
103 }
```

Abbildungsverzeichnis

2.1 Einfach verkettete Liste in einer einzigen Cache-Line 10
2.2 Einfach verkettete Liste in mehreren Cache-Lines 10
2.3 Iterationsdauer in Abhängigkeit der Datenmenge und des Zugriffsmusters 12
2.4 Einzel-Testreihe zu den Auswirkungen von Cache-Assoziativität 13
2.5 Auswirkungen von Cache-Assoziativität 14
2.6 Iterationsdauer pro MiB in Abhängigkeit von Schrittweite und Arraygröße 18
2.7 Vergleich zwischen Iteration mit fester Schrittweite und zufälligem Zugriffsmuster 18
2.8 Auswirkungen der verschiedenen Cache-Hierarchiestufen auf die Speicherlatenz 19
2.9 Zusammenhang zwischen Größe der Instruktionen und der benötigten Zeit 21
2.10 schematischer Aufbau eines One-Level-Branch-Predictors 23
2.11 Zusammenhang zwischen der Vorhersagbarkeit von Sprüngen und der Laufzeit 24
2.12 Vergleich verschiedener Algorithmen zum Branchless-Porgramming 26
2.13 Anzahl der Transistoren und Taktfrequenzen von Mikroprozessoren in den Jahren 1971 bis 2020 27
2.14 das Amdahlsche Gesetz 28
2.15 Auswirkungen von Cache-Ping-Pong 31
2.16 Auswirkungen von Speicheranordnung und False-Sharing 34
2.17 Auswirkungen von Heap-Allocations auf die Laufzeit 37
3.1 Vererbungshierarchie von Command & Conquer: Red Alert . . . 45
4.1 Wiederverwendung von Entity-IDs 54
4.2 Zerstören von Entities 56
4.3 Aufbau der »Sparse Set«-Datenstruktur 57
4.4 Arbeitsweise des Command-Buffers 62
4.5 Demonstration der implementierten Engine 67
4.6 Screenshot der Flappy Bird-Implementierung 71

Listingverzeichnis

2.1 »fast inverse square root«-Algorithmus aus Quake III Arena . . 4
2.2 Struktur der Knoten der einfach verketteten Liste 9
2.3 Funktion zur Berechnung der Anzahl von Dezimalziffern einer Zahl 20
2.4 verbesserte Funktion zur Berechnung der Anzahl von Dezimalziffern einer Zahl 20
2.5 Vermeidung konditionaler Sprünge 25
2.6 Demonstration von Cache-Ping-Pong 30
2.7 Demonstration von False-Sharing 32
3.1 Funktion P_SpawnMobj() zum Instanziieren von Gegnern in Doom 43
3.2 Repräsentation von Objekten der Spielwelt durch eine Vererbungshierarchie 44
3.3 Game-Loop über einen Container von GameObject-Instanzen . . 44
3.4 Komponenten in einer Kompositions-basierten Architektur . . . 48
3.5 Game-Objects in einer Kompositions-basierten Architektur . . . 49
3.6 Verwendung einer Kompositions-basierten Architektur 50
4.1 Zerstören eines Entitys 55
4.2 Überprüfung auf Existenz eines Entitys 56
4.3 Abrufen einer Kombination von Komponenten 59
4.4 System zum Rendern dynamischer Sprites 60
4.5 API des Batch-Renderers 61
4.6 Schnittstelle für den Zugriff auf Tastatur und Maus 63
4.7 Lua-Beispielscript 65
4.8 Der C++-Code der Flappy Bird-Implementierung 68
4.9 Haupt-Script der Flappy Bird-Implementierung 70

Literaturverzeichnis

Alexandrescu, Andrei (2015). *Writing Fast Code*. URL: https://www.youtube.com/watch?v=vrfYLlR8X8k (besucht am 04. 08. 2021).

Byna, Surendra, Yong Chen und Xian-He Sun (2009). »Taxonomy of Data Prefetching for Multicore Processors«. In: *J. Comput. Sci. Technol.* 24, S. 405–417. DOI: 10.1007/s11390-009-9233-4.

Caini, Michele (2019). *ECS back and forth*. URL: https://skypjack.github.io/2019-02-14-ecs-baf-part-1/ (besucht am 15. 08. 2021).

Drepper, Ulrich (2007). *What Every Programmer Should Know About Memory*. Hrsg. von Ulrich Drepper. URL: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf (besucht am 02. 08. 2021).

Fabian, Richard (2018). *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules*. ISBN: 9781916478701.

Gregory, Jason (2019). *Game engine architecture*. Third edition. Boca Raton: CRC Press. ISBN: 9781138035454.

Guntheroth, Kurt (2016). *Optimized C++*. First edition. Beijing: O'Reilly. ISBN: 9781491922064.

Irving, Rob, Jason Turner und Justin Meiners (2021). *Efficient Programming With Components*. Hrsg. von CppCast. URL: https://cppcast.com/efficient-programming-components/ (besucht am 21. 08. 2021).

Johnson, Jobin (2021). *Branchless programming. Does it really matter?* URL: https://dev.to/jobinrjohnson/branchless-programming-does-it-really-matter-20j4 (besucht am 10. 08. 2021).

Knuth, Donald (1974). *Structured programming with go to statements*. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.6084&rep=rep1&type=pdf (besucht am 29. 08. 2021).

Lee, Ben, Alexey Malishevsky, Douglas Beck, Andreas Schmid und Eric Landry (2001). *Dynamic Branch Prediction*. URL: https://web.archive.org/web/20190717130447/http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/ (besucht am 05. 08. 2021).

Madhav, Sanjay (2014). *Game programming algorithms and techniques: A platform-agnostic approach*. Always learning. Upper Saddle River NJ u.a.: Addison Wesley. ISBN: 9780321940155. URL: https://www.informit.com/articles/article.aspx?p=2167437.

McShaffry, Mike und David Rez Graham (2013). *Game coding complete*. 4. ed. Boston, Mass.: Course Technology. ISBN: 9781133776574. URL: http://www.loc.gov/catdir/enhancements/fy1212/2012930785-d.html.

Moore, Gordon E. (1965). *Cramming more components onto integrated circuits*. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf (besucht am 06. 08. 2021).

Nystrom, Robert (2014). *Game programming patterns*. s.l.: genever benning. ISBN: 9780990582908.

Ostrovsky, Igor (2010). *Gallery of Processor Cache Effects*. URL: http://igoro.com/archive/gallery-of-processor-cache-effects/ (besucht am 09. 08. 2021).

Smith, Alan Jay (1982). »Cache Memories«. In: *ACM Comput. Surv.* 14.3, S. 473–530. ISSN: 0360-0300. DOI: 10.1145/356887.356892.

Solihin, Yan (2015). *Fundamentals of Parallel Multicore Architecture*. Chapman and Hall/CRC Computational Science Ser. Boca Raton: Chapman and Hall/CRC. ISBN: 9781482211184. URL: https://ebookcentral.proquest.com/lib/gbv/detail.action?docID=5320099.

Sutter, Herb und Andrei Alexandrescu (2004). *C++ coding standards: 101 rules, guidelines, and best practices*. 2. print. Boston: Addison-Wesley. ISBN: 0321113586.

Williams, Anthony (2019). *C++ Concurrency in Action*. 2. Aufl. s.l.: Manning Publications. ISBN: 9781617294693.