Hochschule Neubrandenburg
University of Applied Sciences

Bachelorarbeit zum Thema:

Laufzeitvergleiche für die Implementierung von Algorithmen in Java und C/C++

Zum Erlangen des akademischen Grades: "Bachelor of Engineering" (B.Eng.)

Bachelorarbeit an der Hochschule Neubrandenburg im Studiengang Geoinformatik
Erstellt und vorgelegt von Manuel Prager

Erstprüfer: Prof. Dr.-Ing. Andreas Wehrenpfennig
Zweitprüfer: Dipl.-Inform. Jörg Schäfer
Abgabetermin: 17. Mai 2010

urn:nbn:de:gbv:519-thesis2010-0125-2

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Neubrandenburg, den 17. Mai 2010

Manuel Prager

Danksagung

Für die Unterstützung hinsichtlich dieser Arbeit möchte ich mich bei einigen Personen bedanken. Zum einen bedanke ich mich bei Prof. Dr.-Ing. Andreas Wehrenpfennig und Dipl.-Inform. Jörg Schäfer dafür, dass Sie die Betreuung dieser Arbeit übernommen haben sowie für die Unterstützung während der Bearbeitungszeit.

Des Weiteren bin ich sehr dankbar, dass sich einige Korrektoren gefunden haben, die sich dieser Arbeit angenommen haben. Also in diesem Sinne, vielen Dank an: Andreas Gerull, Christina Möller (B.Eng.), Julia Koch, Markus Bradke (B.Eng.) und Tom Grimm.

An dieser Stelle möchte ich mich natürlich auch bei allen Kommilitonen bedanken, auf denen ich mich immer verlassen konnte (und hoffentlich auch umgekehrt). Ohne diese Unterstützung wäre ein so weites Vorankommen wohl nicht möglich gewesen.

Da die Unterstützung jeder einzelnen Person einen wichtigen Beitrag zu dieser Arbeit leistete, soll die vorangegangene Auflistungsreihenfolge der Personen von untergeordneter Bedeutung sein. Aufgrund dieser Tatsache erfolgt eine Danksagung an jede einzelne Person in Form des folgenden Java-Quelltexts. Dabei überlasse ich die Reihefolge der Auflistung dem Programm ;o)

```
import java.io.*;
import java.util.*;

public class Danksagung {

  public static void main(String[] args) {

    Object[] obj = {"Alle die ich vergessen habe ;o)", "Julia Koch",
     "Christina Möller", "Jörg Schäfer", "Markus Bradke", "Tom Grimm",
     "Andreas Gerull", "Andreas Wehrenpfennig"};
    int anz = obj.length;
    ArrayList personen =  new ArrayList(Arrays.asList(obj));

    for (int i=0; i<anz; i++) {
      int r=(int)(Math.random()*(anz-i));
      System.out.print("Vielen Dank an: " + obj[r] + ".\n");
      personen.remove(obj[r]);
      obj = personen.toArray();
    }
  }
}
```

Kurzfassung

Wie hinlänglich bekannt, hängt die Effizienz bzw. Performance von entwickelter Software im Hinblick auf verschiedene Bereiche und Problemgrößen maßgeblich von den Programmierkenntnissen und Wissen eines Softwareentwicklers ab. Dabei spielt sicherlich auch die zugrunde liegende Programmiersprache eine entscheidende Rolle. Genau in diesem Punkt soll diese Bachelorarbeit Klarheit darüber geben, inwiefern sich C/C++ und Java bezogen auf einige Fallbeispiele unterscheiden. Hierfür sollen Algorithmen mit Bezug auf verschiedene physikalische Bereiche eines Computers sowie differierenden Problemgrößen entwickelt und äquivalent implementiert werden. Auf der Basis von Laufzeitmessungen für verschiedene Anwendungen soll es möglich sein, Laufzeitvergleiche zu erstellen. Die hierdurch empirisch gewonnenen Informationen sollen analysiert, dargestellt und auftretende Abweichungen näher erläutert werden.

Da die Laufzeit von vielen unterschiedlichen Faktoren beeinflusst wird, ist eine theoretische Betrachtung der Grundlagen dabei von entscheidender Relevanz. Dabei sollen elementare Konzepte, Eigenschaften, Voraussetzungen sowie weitere Aspekte aufgeführt werden, um somit eine Hilfe für die Umsetzung und Bewertung späterer Fragestellungen zu erhalten.

Abstract

As sufficiently known, the efficiency and performance of developed software, concerning different fields and problem sizes, is significantly dependent on the programming skills and knowledge of the programmer. Thereby the underlying programming language plays a decisive role.
This particular point should be clarified by that bachelor thesis and it will show to what extent C/C++ and Java differ from each other. For this purpose, algorithms referring to different physical fields of a computer, as well as differentiating problem sizes, should be developed and implemented equivalently. Based on time measurements of various applications, duration comparisons are supposed to be possible. The resulting empirical information shall be analysed and presented, whereas upcoming variances are further explained.

Since the runtime is affected by several factors, a theoretical consideration of the basis appears to be of vital relevance. In the process, elementary concepts, features, requirements and further aspects will be mentioned, in order to get help for the realization and evaluation of eventual questions.

Inhaltsverzeichnis

Kapitel 1

1 Einführung

1.1 Ziel der Bachelorarbeit

Diese Bachelorarbeit zum Thema "Laufzeitvergleiche für die Implementierung von Algorithmen in Java und C/C++", wurde im Rahmen des Bachelor-Studium Geoinformatik und im Fachbereich Landschaftsarchitektur, Geoinformatik, Geodäsie und Bauingenieurwesen (LGGB) an der Hochschule Neubrandenburg erstellt.

Das Ziel dieser Bachelorarbeit ist die Erstellung von Laufzeitvergleichen für verschiedene Problemgrößen, zwischen Anwendungen der Programmiersprachen C/C++ und Java. Als Bereiche werden dabei die grundsätzlichen physikalischen Grenzen eines Testsystems herangezogen:

- Prozessorauslastung bzw. Rechenlast,
- Speicherverwaltung und
- Ein- und Ausgabeoperationen (I/O-Operationen) bzgl. der Dateiarbeit.

Des Weiteren sind die zu erstellenden Applikationen auf verschiedenen Betriebssystemen auszuführen. Der Vergleich von ermittelten Laufzeiten zwischen C/C++ und Java bezieht sich primär auf das jeweilige Betriebssystem. Bei Vergleichen zwischen den Betriebssystemen sei auf die unterschiedliche Software und Konfigurationen verwiesen.

Grundlage für die Laufzeitmessungen bilden dabei Algorithmen, die speziell auf die jeweiligen Bereiche angepasst sind. Nach der Entwicklung von entsprechenden Algorithmen muss eine weitestgehend äquivalente Implementierung in Applikationen der Programmiersprachen C/C++ und Java erfolgen. Durch das Ausführen der verschiedenen Programme sowie das Messen der Laufzeiten von relevanten Programmbereichen, soll es möglich sein Ergebnisse darzustellen und miteinander zu vergleichen. Auftretende Unterschiede bzgl. der Laufzeiten werden hierbei bestimmt und analysiert. Weiterhin sind Ursachen für Abweichungen zu benennen sowie ggf. näher zu erläutern.

Bei den zu erstellenden Anwendungen handelt es sich lediglich um Fallbeispiele. Die hierdurch resultierenden Laufzeitvergleiche bzw. Ergebnisse können nicht für alle Programme der zu vergleichenden Sprachen generalisiert werden.

1.2 Motivation

Einleitend könnte die Frage aufkommen, warum Laufzeitvergleiche zwischen unterschiedlichen Programmiersprachen von Relevanz sein könnten? Die Frage ist relativ leicht zu beantworten, denn ganz allgemein ausgedrückt ist Zeit immer mit finanziellen Ressourcen verbunden. So kann schon die Auswahl einer Programmiersprache für bestimmte Bereiche entscheidende Vorteile mit sich bringen. Darüber hinaus können Laufzeitmessungen auch weitere Engpässe identifizieren (z. B. Laufzeit von uneffizienten Funktionen oder Programmabschnitten) und ermöglichen so eine zielgerichtete Beseitigung.

Doch bei der Frage, welche Sprache nun effizienter sei, sind häufig Behauptungen zu finden, dass Java als Programmiersprache deutlich langsamer als C/C++ sei. So finden sich z. B. in Internet-Foren Aussagen, wie: "80% der Leute mit denen ich rede meinen, das Java so viel langsamer ist als C/C++. (...)" oder "Java ist etwa um Faktor 2 langsamer als C. (...)" [FOR 01]. Andererseits gibt es Auffassungen, dass solche Behauptungen alte Vorurteile sind und dass Java-Anwendungen aufgrund von Weiterentwicklungen wie neuen Sprachkonzepten (JIT) und zahlreichen Optimierungen gleich schnell oder gar schneller sind.

Performance- bzw. Effizienzvergleiche zwischen C/C++ und Java waren schon immer ein umstrittenes Thema. Doch wie viel kann solchen Bemerkungen wirklich beigemessen werden? Diese Arbeit soll im Hinblick auf wesentliche Bereiche eines Computers und bezogen auf einige Fallbeispiele, Aufschluss über das Laufzeitverhalten von Anwendungen der Sprachen C/C++ und Java geben.

1.3 Gliederung

Der Aufbau der Kapitel dieser Arbeit gliedert sich wie folgt:

Kapitel 2
Zunächst werden grundlegende Prinzipien, Konzepte und Begriffe erklärt, die zur Hinführung des Themas dienen sollen. Diese theoretische Betrachtung elementarer Eigenschaften soll außerdem eine Hilfe für die Analyse, Umsetzung und Bewertung späterer Fragestellungen darstellen.

Kapitel 3
In diesem Kapitel werden Algorithmen für die Bereiche: Prozessorauslastung bzw. Rechenlast, Speicherverwaltung, Ein- und Ausgabeoperationen (I/O-Operationen) bzgl. der Dateiarbeit dargestellt. Es werden außerdem die Herangehensweisen zur Erstellung dieser Algorithmen aufgezeigt und relevante Aspekte näher erläutert.

Kapitel 4
Die in diesem Kapitel aufgeführten Punkte erläutern die Vorgehensweise bei der Durchführung. Grundsätzliche Faktoren, die bei der Umsetzung eine Rolle spielen, wie z. B. der Implementierung der Algorithmen werden benannt und ggf. näher erläutert. Weiterhin sind Angaben zu dem verwendeten Testsystem, Compiler sowie weiteren Hilfsmitteln zu finden.

Kapitel 5
Dieses Kapitel dient der Darstellung der ermittelten Ergebnisse, hierbei werden Laufzeitvergleiche der verschiedenen Bereiche in entsprechenden Formen dargestellt. Resultierende Laufzeitunterschiede zwischen den Programmiersprachen werden benannt und ggf. analysiert.

Kapitel 6
Im letzten Kapitel wird zunächst eine Zusammenfassung über die ermittelten Laufzeiten gegeben. Anschließend sollen Ursachen für Laufzeitdifferenzen weitestgehend erläutert sowie eine Bewertung gegeben werden. Für den Abschluss dieser Arbeit werden weitere wesentliche Punkte aufgezeigt, die aufgrund des Umfangs dieser Arbeit nicht ausreichend berücksichtigt werden konnten.

Fachwörter bzw. Fremdwörter sind im Informationsglossar (unter Punkt 7) genauer erklärt und beschrieben. Darüber hinaus sind einige wichtige bzw. entscheidende Begriffe hervorgehoben. Im weiteren Verlauf dieser Arbeit werden einige Abkürzungen verwendet, die am häufigsten verwendeten werden im Folgenden erläutert.

Abkürzungen:

CPU	Central Processing Unit bzw. Bereich CPU-Auslastung / Rechenlast	
I/O	Input/Output bzw. Bereich I/O-Operationen hinsichtlich der Dateiarbeit	
RAM	Random Access Memory bzw. Bereich Speicherverwaltung	
E/Dim	Elemente pro Dimension einer Matrix, Liste oder Feldes (stellen die Problemgrößen dar)	
JAVAC	Java-Applikation, die mit dem javac-Compiler (SUN) übersetzt wurde	
JIKES	Java-Applikation, die mit dem Jikes-Compiler übersetzt wurde	
Win	Windows bzw. Betriebssystem "Microsoft Windows XP"	
Max	Maximalwert	(der größte Wert, der bei den Laufzeitmessungen je Problemgröße aufgetreten ist)
Min	Minimalwert	(kleinster ermittelter Wert je Problemgröße)
Mittel	Mittelwert	(Summe der ermittelten Werte durch Anzahl der Messungen je Problemgröße)
OPT	Optimiert	(bei der Sprachübersetzung wurde ein Optimierungsschalter angegeben)
UNO	Unoptimiert	(Übersetzung des Quellprogramms ohne Angabe einer Optimierungsoption)

Kapitel 2

2 Grundlagen

In diesem Kapitel wird zunächst auf grundlegende Begriffe, Paradigmen und Konzepte von Programmiersprachen eingegangen, verschiedene Sprachen werden benannt und klassifiziert. Es wird außerdem untersucht, welche elementaren Eigenschaften hinsichtlich der Problembereiche/-größen bei Konzepten eine entscheidende Rolle spielen können. Für die Bestimmung der Laufzeit werden einige Methoden benannt sowie weitere Voraussetzungen, die bei Laufzeitvergleichen wichtig sind, beschrieben. Ferner werden weitere allgemeine Aspekte, die von Relevanz hinsichtlich dieser Arbeit und Thema sind, aufgeführt und näher erläutert.

2.1 Was ist eine Programmiersprache?

Grundlegend lässt sich sagen, dass eine formale **Programmiersprache** ein "Hilfsmittel" ist, mit dem gesagt werden kann was ein Computer tun soll. Da ein Computer nur bestimmte Anweisungen ausführen kann, müssen Zeichen- bzw. Befehlsfolgen genau definiert werden. Diese Definition von Zeichen bzw. Befehlen wird als **Syntax** (Beschreibung) und was diese im Einzelnen auf einem Rechner bewirken als **Semantik** (Bedeutung) bezeichnet.

Durch eine abweichende Abhängigkeit von Sprachen bzgl. der Maschinen lassen sich Programmiersprachen in zwei Kategorien unterscheiden. Auf der einen Seite gibt es die so genannten »niedrigen Programmiersprachen«, welche auch als Assemblersprachen bezeichnet werden. Diese Sprachen sind maschinenorientiert, also abhängig von der Hardware und dienen der direkten Programmierung von Hardware. Zum anderen gibt es die »höheren Programmiersprachen«, welche maschinenunabhängiger sind und darüber hinaus ein höheres Abstraktionsniveau besitzen. Spezieller ausgedrückt lässt sich eine Programmiersprache wie folgt definieren:

> "Eine Programmiersprache ist ein notationelles System zur Beschreibung von Berechnungen in durch Maschinen und Menschen lesbarer Form." [Quelle: LOU 94]

[Punkt 2.1 nach Quellen: LOU 94, WIS 02]

2.2 Berechnungsparadigmen von Programmiersprachen

Mit dem Berechnungsparadigma einer Programmiersprache bzw. Programmiertechnik wird allgemein das grundsätzliche Prinzip (oder auch Denkmuster) verstanden.

Die folgenden Auflistungen sollen einen Auszug über existierende Programmierparadigmen geben. Aufgrund des Umfangs dieser Bachelorarbeit und Themenabgrenzung wird nicht auf alle aufgeführten Programmierparadigmen eingegangen. Es wird beispielhaft im Folgenden gezeigt, was diese beinhalten und welche grundlegenden Eigenschaften diese besitzen.

- Imperativ (prozedural)
- Funktional
- Logisch (deklarativ)
- Objektorientiert

Nach [LOU 94] ist derzeit die Mehrheit der Programmiersprachen imperativ, dazugehören beispielsweise **C/C++** und **Java**. Jedoch besitzen C++ und Java noch weitere sprachliche Prinzipien, um das **objektorientierte** Programmierparadigma (OOP) zu unterstützen.

Das wichtigste Merkmal einer **imperativen** Sprache ist eine Folge von Anweisungen (z. B. Befehle oder Kommandos). Diese Art einer Programmiersprache besitzt nach [LOU 94] drei grundlegende Eigenschaften:

1. Sequenzielle Ausführung von Instruktionen,
2. Verwendung von Variablen (stellen Speicherwerte dar) und
3. Verwendung von Zuweisungen um Werte von Variablen zu ändern.

Imperative Sprachen orientieren sich hinsichtlich der Berechnungen an dem »von-Neumann-Modell«. Es werden also Berechnungen als Folge von Anweisungen beschrieben. Bezieht sich eine Folge von Berechnungen auf ein einziges Datenelement, so spricht man vom »von-Neumann-Flaschenhals«. Dieser beschränkt die Möglichkeiten einer Sprache hinsichtlich paralleler und nichtdeterministischer Berechnungen sowie Berechnungen, die nicht notwendig von einer Ordnung abhängen. Es wird somit z. B. deutlich erschwert, Berechnungen auf viele unterschiedliche Datenelemente in Anspruch zu nehmen.

[Punkt 2.2 nach Quelle: LOU 94]

2.3 Konzepte - Sprachübersetzung

Mithilfe von Interpretern, Compilern bzw. Assemblern wird eine **Kompilierung** (Sprachübersetzung) vorgenommen. Dabei wird Quellcode, welcher in einer bestimmten Programmiersprache geschrieben wurde, in einen ausführbaren Zustand gebracht oder direkt ausgeführt.

Für die relevanten Sprachen ist in erster Linie das Interpreter- und Compiler-Konzept von entscheidender Bedeutung. Grundsätzlich ist ein **Interpreter** für die direkte Ausführung eines Programms zuständig. Ein **Compiler** hingegen bringt ein Programm in einem erforderlichen Zustand. Diese beiden relevanten Konzepte werden im Weiteren näher erläutert.

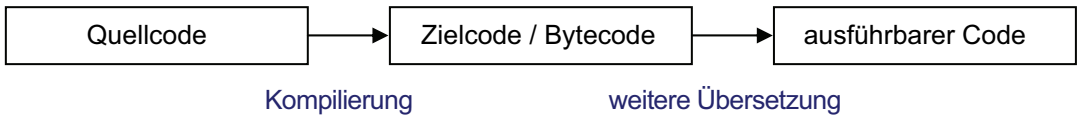
[Punkt 2.3 nach Quelle: LOU 94]

2.3.1 Compiler

Ein **Compiler** wandelt den Quellcode in ein Zielprogramm um. Dabei kann die Ausführung des Zielprogramms erst erfolgen, wenn es in einem geeigneten Zustand bzw. in einer für die Ausführung anwendbaren Form (→ Maschinensprache) verfügbar ist.

Folgendermaßen kann der zweistufige Kompilierungsprozess nach [LOU 94], stark vereinfacht veranschaulicht werden:

1. Stufe



2. Stufe

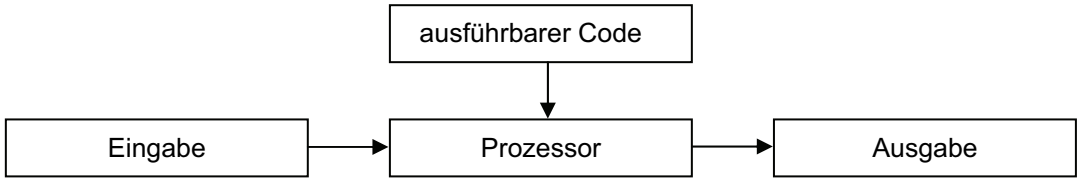


Abbildung 1: schematische Darstellung des Kompilierungsprozesses [nach Quelle: LOU 94]

Weitere Aufgaben eines Compilers sind u. a. das Prüfen auf syntaktische Korrektheit und (statische) Semantik des Quelltextes sowie das Erzeugen von Informationen für Binder und Debugger. Der Nachteil des Compilerkonzeptes ist, dass jede Änderung am Quelltext eine vollständige Neuübersetzung erfordert. Auf der anderen Seite wirkt sich dieses Sprachkonzept positiv auf die Performance bzw. Laufzeit von Programmen aus, da ein Programm nur einmal übersetzt werden muss und dann ausgeführt werden kann.

[Punkt 2.3.1 nach Quellen: LOU 94, SS 02]

2.3.2 Interpreter

Ein **Interpreter** für eine Maschine kann als Programm aufgefasst werden, dass Anweisung für Anweisung sukzessiv analysiert, übersetzt und direkt ausführt. Der Prozess der Programminterpretation ist einstufig, was bedeutet, dass ein Programm sowie eine Eingabe einem Interpreter übergeben wird. Anschließend erfolgt eine entsprechende Ausgabe.

Der Interpretationsprozess kann nach [LOU 94] wie folgt, vereinfacht dargestellt werden:

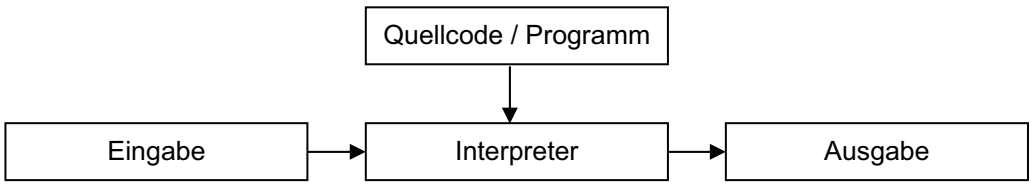


Abbildung 2: schematische Darstellung des Interpretationsprozesses [nach Quelle: LOU 94]

Im Vergleich zum Compiler erzeugt ein Interpreter kein Maschinenprogramm, dadurch fallen Zeiten für das Laden, Binden und Übersetzen weg, was wiederum die Zeit der Programmentwicklung verringert. Infolgedessen resultiert auch ein wichtiger Aspekt hinsichtlich der Laufzeitbestimmung, denn die schrittweise Analyse, Übersetzung und Ausführung wirkt sich nachteilig auf die Laufzeit bzw. Performance von Programmen aus. Es werden also im Gegensatz zum Compilerkonzept mehrere Vorgänge (mehrfach) ausgeführt, was schließlich zu einer Zunahme der Programmlaufzeit führt.

Im Gegensatz dazu besitzt dieses Konzept den Vorteil, dass bei einer Änderung am Quelltext keine komplette Neuübersetzung nötig ist. Des Weiteren ist ein so übersetztes Programm plattformunabhängig und kann auf beliebigen Maschinen ausgeführt werden.

[Punkt 2.3.2 nach Quellen: LOU 94, MSP 97, PR 97, SS 02]

2.3.3 Weitere Konzepte

Ein weiteres Konzept der Sprachübersetzung ist die **Just-in-Time-Kompilierung (JIT)**, wodurch eine erhebliche Performancesteigerung (u. a. bei paralleler Berechnung von rechenintensiven Anwendungen) erreicht werden kann. Im Allgemeinen lässt sich hinsichtlich der Funktionsweise sagen, dass ein Programm schon vor der eigentlichen Ausführung übersetzt wird. Was wiederum bedeutet, dass nur die am meisten gebrauchten Funktionalitäten übersetzt werden. So wird z. B. beim ersten Methodenaufruf der **Byte-Code** in Maschinensprache übersetzt und anschließend im Arbeitsspeicher gehalten. Wird die Methode dann wieder aufgerufen, so wird der übersetzte Code aus dem Arbeitsspeicher verwendet.

Mit **Ahead-of-Time Compilern (AOT)** (auch als **native Compiler** bezeichnet) lässt sich aus Byte-Code/Quellcode, direkt (nativer) Maschinencode für eine zugrunde liegende Architektur erzeugen. Dieses Konzept entspricht dem gängigen Compiler-Konzept, jedoch ist dieses Prinzip auch für Sprachen geeignet, die nicht ausschließlich Compiler für die Sprachübersetzung und Programmausführung verwenden (wie z. B. Java). Als Ergebnis liefert der AOT-Compiler eine Datei, die ohne eine Laufzeitumgebung direkt ausführbar ist.

Weitere Vorteile im Gegensatz zum JIT-Konzept sind, aufgrund der zuvor erfolgten Übersetzung, eine bedeutend schnellere Ausführung des Codes zur Laufzeit sowie prinzipiell geringere Startzeiten. Zudem ist ein so erzeugtes Programm erheblich schwieriger zu entschlüsseln und kann je nach Verwendung leistungsfähiger sein. Als Nachteil des AOT-Compiler-Konzeptes kann der Verlust der Plattformunabhängigkeit angesehen werden, denn der übersetzte Maschinencode kann nur auf der entsprechenden Architektur / Maschine, für die eine Übersetzung erfolgt ist, ausgeführt werden.

[Punkt 2.3.3 nach Quellen: STE 01, ULL 09, WIKI 07]

2.4 Programmiersprachen im Überblick

In diesem Punkt sollen wesentliche Eigenschaften der Sprachen C/C++/Java benannt und ein entsprechender Einfluss auf das Laufzeitverhalten näher erläutert werden.

Von der Entwicklung ist sowohl C++ als auch Java durch C beeinflusst worden. Diese Verwandtschaft spiegelt sich in der Verwendung von einigen äquivalenten Grundkonzepten und Merkmalen wider. Dabei sind die Ähnlichkeiten zwischen C und C++ größer als jeweils zu Java. Eine umfangreiche Abbildung nach [HPR 05] hinsichtlich der Entwicklung, Beeinflussung bzw. Abhängigkeit von Programmiersprachen (nach Jahren) kann dem »Anhang A« entnommen werden. Im weiteren Verlauf werden detaillierte Informationen zu den verschiedenen Sprachen gegeben, welche Gemeinsamkeiten und Unterschiede aufzeigen.

Die nachfolgend aufgeführte Tabelle gibt einen Überblick über grundlegende Eigenschaften der entscheidenden Programmiersprachen C/C++ und Java, die anhand von ausgewählten Algorithmen und hinsichtlich verschiedener Problemgrößen im Einzelnen verglichen werden.

Überblick - C/C++ und Java			
	C	**C++**	**Java**
Entwickler	Dennis Ritchie, (AT&T Bell Laboratories)	Bjarne Stroustrup, (AT&T Bell Laboratories)	Sun Microsystems (Tochter-firma der Oracle Corporation)
Erscheinung	1972	1983	1995
Paradigmen	imperativ, strukturiert	imperativ, strukturiert, objektorientiert, generisch	objektorientiert
Sprachübersetzung	Compiler	Compiler	kombinierter Ansatz aus Compiler und Interpreter, JIT / AOT-Compiler
Vorteile	- Leistungsfähigkeit und Flexibilität - bekannte Sprache (viele Compiler, Programme) - Standardisierung - Portabilität - geringer Wortschatz - Modularität - hardwarenah	- Abwärtskompatibilität zu C (inkl. Vorteilen) - unterstützt OOP	- kleiner, portabler, leichter anwendbar als C/C++ - dezentral, dynamisch - Speicherverwaltung - Konzept ist sicher - Robustheit, Stabilität - Plattformunabhängig - Performanz - Multithreading-Fähigkeit
Nachteile	- Sicherheit - Speichermanagement - Single-Threading	s. Nachteile "C"	- benötigt virtuelle Maschine (Installation, Start ist zeitaufwendig)

Tabelle 1: Überblick - C/C++ und Java [nach Quellen: AJ 00, EGG 05, LOU 94, MSP 01, SS 02, STE 01, WIKI 01/02]

Während die Unterschiede zwischen C und C++ verhältnismäßig gering sind, sind die Differenzen beider Sprachen zu Java erheblich größer. Besonders das für Java verwendete Interpreter-Konzept ist bei heutigen, weitverbreiteten Programmiersprachen einmalig. Eine Gemeinsamkeit verbindet jedoch alle drei Sprachen miteinander, sie gehören aktuell zu den am meist verwendeten (z. B. [TIO 01]).

Für weitere Besonderheiten hinsichtlich der zu vergleichenden Programmiersprachen sei auf die weiteren Punkte verwiesen. Es soll aufgezeigt werden, welche grundlegenden Eigenschaften der Sprachen, bestimmte Bereiche beeinflussen, um so evtl. resultierende Unterschiede besser bewerten zu können.

2.4.1 C/C++

Dass zwischen C und C++ eine gewisse Verwandtschaft besteht, lässt sich schon aus deren Namen erkennen. Infolgedessen handelt es sich bei C++ um den Nachfolger der Sprache C.

Die **Programmiersprache C** ist aufgrund der Leistungsfähigkeit, Freiheiten und nicht zuletzt wegen der hohen Verbreitung eine der Ältesten, die heutzutage noch Verwendung finden. C wurde mit dem Ziel entwickelt, eine Abstraktion für Assemblersprache zu schaffen. Einsatzgebiete wie System- und Betriebssystemprogrammierung sind daher besonders für C geeignet. Gründe für den Einsatz in solchen Gebieten sind Möglichkeiten der hardwarenahen Programmierung und der Effizienz, die C bietet. So ist es in C möglich Hardware direkt anzusprechen, um so eine bestmögliche Performance zu erzielen. Die daraus resultierende hohe Geschwindigkeit von C-Programmen ist außerdem häufig Anlass, Compiler, Programmbibliotheken oder Interpreter (z. B. die Java-VM) in C zu schreiben. Die Aspekte der hardwarenahen Programmierung und der damit verbundenen Freiheiten haben jedoch sicherheitskritische Gefahren zur Folge. So muss sich ein Softwareentwickler um viele Angelegenheiten wie z. B. der Speicherverwaltung selber kümmern. Dieser Mehraufwand erhöht u. a. die Komplexität sowie die Entwicklungszeit von C-Programmen. Bei falscher Implementierung sind Programm- oder gar Systemabstürze zu erwarten.

Bei **C++** handelt es sich hingegen um eine **objektorientierte** Erweiterung der Sprache C, was bedeutet das die in C enthaltenen Merkmale auch für C++ verfügbar sind, jedoch nicht umgekehrt. Aufgrund dieser Tatsache sind sämtliche Vorteile von C auch bei C++ zu finden. Dies gilt gleichermaßen auch für die damit verbundenen Nachteile. Anschließend sollen für C und C++ einige Merkmale zusammengefasst werden, die für verschiedene Laufzeitaspekte von Bedeutung sind.

- Nutzung von Compilern für die Sprachübersetzung
  → aus entsprechendem Quellcode wird direkt ausführbarer Code für eine Maschine erzeugt
- Möglichkeiten der hardwarenahen Programmierung bzw. Systemprogrammierung
  → direkte Speicherzugriffe
  → sehr hardwarenahe Konstrukte, usw.
- enge Verbindung zu UNIX-Betriebssystemen (relevant für Linux)
  → aufgrund etwaiger Anpassungen sind bessere Ergebnisse zu erwarten?

[Punkt 2.4.1 nach Quellen: AJ 00, KER 08, MSP 97, WIKI 13]

2.4.2 Java

Es mag vielleicht ein wenig verwundern, dass **Java** nach [AJ 00] mit dem Ziel entworfen wurde, Haushaltsgeräte zu steuern. Die Idee, die »Sun Microsystems« damit verfolgte, zielte darauf ab sämtliche Geräte im Haushalt von Kühlschrank über Videorecorder bis zur Heizung über einen einzigen Computer steuern zu können. Aus dieser Idee resultierte u. a. eine entscheidende Besonderheit, über die weder C noch C++ verfügt, die Plattformunabhängigkeit.

Dabei wird die Plattformunabhängigkeit bzw. Portabilität von Java-Programmen durch einen kombinierten Ansatz aus Compiler und Interpreter realisiert. Der Java-Compiler (z. B. »javac«) erzeugt hierbei aus entsprechenden Quelltext (*.java), plattformunabhängigen **Java-Byte-Code** (*.class). Für die Ausführung von Programmen ist ein Java-Interpreter (z. B. "java") zuständig - die sogenannte **virtuelle Java-Maschine (Java-VM)** - diese fungiert als Schnittstelle zwischen dem Java-Byte-Code und der Maschine (Hard- und Software).

Im Hinblick auf die Performance lässt sich sagen, dass durch das jeweilige Erfassen, Dekodieren und Ausführen der Direktiven eines Java-Interpreters, ein Performance-Verlust auftreten kann. Da der Bytecode aber relativ maschinennah ist, werden Nachteile bzgl. der Laufzeit bzw. Performance gering gehalten. Des Weiteren sind viele Java-Interpreter der Kategorie Just-in-Time-Kompilierung zuzuordnen, hierdurch können weitere Performance-Beeinträchtigungen minimiert werden. Im Vergleich zu einem Interpreter wird die Programmausführung nach [PR 97] um einen Faktor von 10 bis 20 beschleunigt und die Startzeit etwas verlängert.

Weiterhin können die im Folgenden aufgeführten Besonderheiten von Java nach [ULL 09, PR 97], einen negativen Einfluss auf die Performance haben.

- Interpreter-Konzept (entfällt durch Nutzung von AOT-Compilern)
- Abstraktionsgrad
- Objektorientierung
  → aufgrund der dynamischen Speicherverwaltung haben Objekte im Gegensatz zu herkömmlichen Daten, einen erhöhten Speicherbedarf
  → je nach Implementierung kann die dynamische Bindung, die Laufzeit eines Programms erhöhen, da sie eine Typprüfung zur Laufzeit benötigt und so den Quellcode vergrößert
- Aspekte der Laufzeitumgebung / Java-VM (→ s. auch Punkt "2.6.2 Software-Faktoren")
  → Initialisierung der Java-VM / Laufzeitumgebung
  → Einsatz des »**Java Garbage Collector**«
  → automatische Überprüfung von Array-Grenzen (Feldgrenzen)
  → Ausnahmebehandlungen

[Punkt 2.4.2 nach Quellen: AJ 00, KRÖ 06, LM 05, PR 97, SS 02, ULL 09, WIKI 05]

2.4.3 Compiler

Um aus den Quellprogrammen der verschiedenen Sprachen ausführbare Anwendungen zu erzeugen, stehen einige Compiler zur Verfügung. Für die Auswahl von Compilern sind ausschließlich freie Compiler in Betracht gezogen worden. Die nächste Tabelle soll dabei einen Auszug über einige gängige Compiler sowie einer Erläuterung geben.

Compiler für C/C++ und Java (Auszug)	
C/C++ - Compiler	**Beschreibung**
GCC	- GCC steht für "GNU Compiler Collection" und ist eine Compilersammlung für verschiedene Sprachen (z. B. C/C++, Java, u. a.) - die ursprüngliche Bedeutung von GCC war "GNU C Compiler", jedoch steht das Kommando »gcc« nach wie vor für den C-Compiler - bei GCC handelt es sich um freie Software nach GPL-Lizenz, der "Free Software Foundation"
MinGW / Mingw32	- MinGW ist die Abkürzung für "Minimalist GNU for Windows" und beinhaltet Portierungen der GNU-Entwicklerwerkzeuge, wie z. B. der GCC - diese Portierungen für Windows-Betriebssysteme ermöglichen die Entwicklung von Applikationen für eben diese Plattformen
Java-Compiler	**Beschreibung**
javac	- Compiler für Java und Bestandteil des JDK (Java Development Kit) der Firma Sun Microsystems - nach [WIKI 11] gehört das JDK zu den am meist verwendeten Java-SDKs (Software Development Kit) - der javac-Compiler hat die Aufgabe Java-Quellcode (*.java) in plattformunabhängigen Bytecode (*.class) zu übersetzen
Jikes	- Jikes ist ein in C++ geschriebener Compiler, welcher ursprünglich bei IBM entwickelt wurde - wie der javac ist auch der Jikes ein freier Compiler, um Bytecode aus Java-Dateien zu erzeugen - nach [JIK 01] verfügt der Compiler über folgende Vorteile: offene Quellen (open Source), strikte Kompatibilität zu Java, hohe Leistungsfähigkeit, Analyse von Abhängigkeiten, konstruktive Hilfe - ein weiterer Vorteil von Jikes ist die spürbar kürzere Übersetzungszeit
GJC	- GJC (Compiler for the Java Programming Language) ist ein portabler, optimierter Ahead-of-time-Compiler (sowie Bestandteil der GCC) - GJC kann Java-Quellcode zu Java-Bytecode und Java-Bytecode/-Quellcode direkt zu nativen Maschinen-Code kompilieren - für virtuelle Maschinen und Compiler werden die freien "GNU Classpath"-Klassenbibliotheken genutzt
MinGW Java Compiler	- s. auch MinGW / Mingw32 bei "C/C++ - Compiler" - beinhaltet die Portierung des GJC - Java-Compiler (gcc-java) für Windows-Betriebssysteme

Tabelle 2: Compiler für C/C++ und Java (Auszug) [nach Quellen: GCC 01, JIK 01, MIN 01, WIKI 06/11]

Im Verlauf dieser Arbeit werden weitere Angaben zu den verwendeten Compilern gegeben. Des Weiteren soll aufgezeigt werden, welche Möglichkeiten hinsichtlich der Optimierung für die ausgewählten Compiler zur Verfügung stehen.

2.5 Laufzeitvergleiche

Bei der **Laufzeit** handelt es sich allgemein ausgedrückt, um die Zeit die ein Programm oder ein bestimmter Programmabschnitt zur Ausführung benötigt. Dabei wird die Laufzeit durch die nach [WIKI 08] aufgeführten Punkte bestimmt:

- die Programmlogik (Folge von Anweisungen),
- die Eingabedaten (z. B. Zunahme der Komplexität von Berechnungen und der damit verbundenen Auswirkungen auf die Anzahl von Schleifendurchläufen oder Ähnlichem),
- der Compiler (Optimierungsstufe, z. B. Optimierung durch Angabe von Schaltern) und
- die Architektur und Taktfrequenz des Ausführungsrechners (Ausführungsgeschwindigkeit unter Berücksichtigung von Cache- und Pipelining-Effekten).

Werden die Ergebnisse einer Laufzeitmessung (auch als **Profiling** bezeichnet) von Programmen/ Programmabschnitten gegenübergestellt und ggf. in eine geeignete Form gebracht, so lässt sich von einem **Laufzeitvergleich** sprechen.

Hinsichtlich dieser Arbeit sollen Laufzeitvergleiche für relevante Programmabschnitte (C/C++/Java) durchgeführt werden. Um Laufzeitvergleiche zu erstellen ist es notwendig, sich zuerst mit der Frage zu befassen: "Wie kann die Laufzeit überhaupt bestimmt werden?". Diese Frage sowie substanzielle Anforderungen bzgl. Methoden/Funktionen (der Programmiersprachen), Herangehensweisen und deren Eigenschaften sollen in den nächsten Punkten geklärt werden.

2.5.1 Voraussetzungen für Laufzeitmessungen

Für die Laufzeitmessungen gilt es neben der Methode zur Bestimmung der Laufzeit, auch einige wichtige andere Punkte zu beachten, um präzise, aussagekräftige und vergleichbare Ergebnisse zu erhalten. Anschließende Aspekte nach [LM 05], sollten für die verschiedenen Laufzeitvergleiche berücksichtigt werden:

- Laufzeitmessungen müssen nur für relevante Bereiche/Programmabschnitte erfolgen
  - die Laufzeit von unnötigen Direktiven, Operationen, usw., sollte nicht in eine Messung einfließen
- Vermeidung von Zugriffen auf bestimmte Bereiche, die nicht primär in eine Messung mit einbezogen werden (z. B. Konsole, Internet, Dateisystem, usw.)
  - derartige Zugriffe sind meist mit hohem Zeitaufwand verbunden
- Vermeidung von Benutzerinteraktionen
  - bei Interaktionen durch einen Nutzer wird außerdem auch deren Reaktionszeit gemessen, dabei kann es zu verfälschten Laufzeitergebnissen kommen
- Durchführen von wiederholten Messungen
  - eine einzige Messung kann kein repräsentatives Ergebnis für einen Laufzeitvergleich liefern
  - z. B. kann ein Programm aufgrund von Initialisierungs- / Synchronisationsprozessen beim ersten Start, mehr Zeit benötigen als bei folgenden Ausführungen
  - Messungen könnten in Form einer Schleife wiederholt und anschließend ein entsprechender Mittel- oder Medianwert gebildet werden
- Verwendung von gleichen Ausgangsdaten / Testsystemen
  - Ausgangsdaten müssen für die verschiedenen Laufzeitmessungen äquivalent sein
  - die Programme der Sprachen C/C++ und Java sollten außerdem auf ein und demselben Testsystem(en) ausgeführt werden, um vergleichbare Ergebnisse zu erzielen
- Genauigkeit einer Methode berücksichtigen
  - wenn Methoden eine Genauigkeit z. B. im Millisekunden-Bereich erzielen, so sollte diese nur bedingt ausgenutzt werden, hier wären z. B. Operationen zweckmäßiger die einige Sekunden für die Ausführung in Anspruch nehmen, oder
  - Nutzung von genaueren Methoden

[Punkt 2.5.1 nach Quelle: LM 05]

2.5.2 Methoden der Laufzeitbestimmung

Es gibt unterschiedliche Möglichkeiten um die Laufzeit oder Effizienz von Programmen bestimmen zu können. Hauptfaktoren sind hierbei grundsätzlich die Anwendbarkeit und Genauigkeit einer jeweiligen Methode. Die folgende Tabelle nach [KUB 07], soll einige Möglichkeiten zur Bestimmung der Laufzeit sowie Vor- und Nachteile der aufgeführten Methoden benennen. Die dadurch resultierenden Ergebnisse sollen u. a. eine Begründung zur Auswahl einer geeigneten Bestimmungsmethode darlegen.

Methoden der Laufzeitbestimmung (Beispiele)			
	Beschreibung	**Vorteile**	**Nachteile**
Software (Profiler)	- separate Software (Profiler) kann zur Laufzeitbestimmung verwendet werden - dabei sind kommerzielle und freie Produkte zu unterscheiden - Bsp. C/C++: gprof, KProf - Bsp. Java: Profiler, JProbe, HPJMeter, Hyades (Eclipse Plug-in)	- kein Eingriff in das Programm bzw. Quelltextänderungen nötig - spart Zeit und Arbeit bzgl. der Implementierung von Methoden zur Bestimmung der Laufzeit	- Begrenzung der Software auf bestimmte Sprachen/Betriebssysteme - ungenügende Konfiguration (Laufzeitmessung von Programmabschnitten?) - zusätzliche Erzeugung von Overhead, welche somit die Messung verfälscht - ggf. ungenaue Ergebnisse - Methoden zur Auswertung/Darstellung von Ergebnissen? - Entgelt für kommerzielle Produkte
Betriebssystem	- Bsp.: für UNIX-Betriebssysteme steht ein "`time`"-Kommando zur Verfügung, dass eine Laufzeitmessung auf Kommandozeilenebene ermöglicht	- s. Vorteile "Software" - einfache Anwendung - Ausgabe der Ausführungszeit mit Verweildauer (*real*), Zeit im Benutzer- (*user*) und Systemmodus (*sys*)	- derartige Kommandos beziehen sich meist auf ein Betriebssystem, wie in diesem Fall "UNIX", für "Windows"-Systeme ist dem "`time`"-Kommando eine andere Funktion zugewiesen - geringe Genauigkeit (in Sekunden) - Zeitmessung von Programmteilen kaum oder nicht möglich
manuelle Implementierung	- durch Auslesen der Zeit des jeweiligen Betriebssystems bzw. nutzen von speziellen Zeit-Funktionen, kann eine angepasste Bestimmung der Laufzeit vorgenommen werden	- individuelles Messen von Programmteilen - hohe Genauigkeit und geringer Overhead (abhängig von der Implementierung)	- Aufwand für die Implementierung - zusätzlicher Eingriff in das Programm - bei falscher Implementierung sind nicht vergleichbare Ergebnisse die Folge - für komplexere Anwendungen ungeeignet

Tabelle 3: Möglichkeiten der Laufzeitbestimmung (Beispiele) [nach Quellen: FHF 01, IFI 01, KUB 07]

Bei der Auswahl der Methode zur Laufzeitbestimmung war es u. a. wichtig, genaue Ergebnisse zu erhalten. Des Weiteren sollten Laufzeiten nur für bestimmte Bereiche und nicht für die Gesamtheit eines Programms ermittelt werden, weshalb separate Software die nur eine schlechte oder keine Konfiguration zu lässt, nicht geeignet war. Weitere Software ist zudem meist auf bestimmte Sprachen begrenzt, welches die Laufzeitbestimmung für verschiedene Sprachen nicht ermöglicht. Außerdem handelt es sich bei einem Teil der professionellen Produkte um kommerzielle Software, die aufgrund unzureichender finanzieller Mittel nicht verwendet werden konnten.

Als Ergebnis der Recherche über Methoden der Laufzeitbestimmung war eine individuelle Implementierung, im Gegensatz zu den anderen Möglichkeiten von größerer Relevanz. Hierdurch kann eine nach den unterschiedlichen Anforderungen angepasste Messung der Laufzeit mit hoher Genauigkeit realisiert werden. Jedoch ist es im Weiteren noch nötig, genauere Informationen zur Verwendung von vorhandenen Zeit-Funktionen, der verschiedenen Sprachen zu erarbeiten und herauszufinden, wie diese implementiert werden müssen.

Der Ablauf einer manuellen Programmimplementierung kann dabei vereinfacht, wie folgt dargestellt werden:

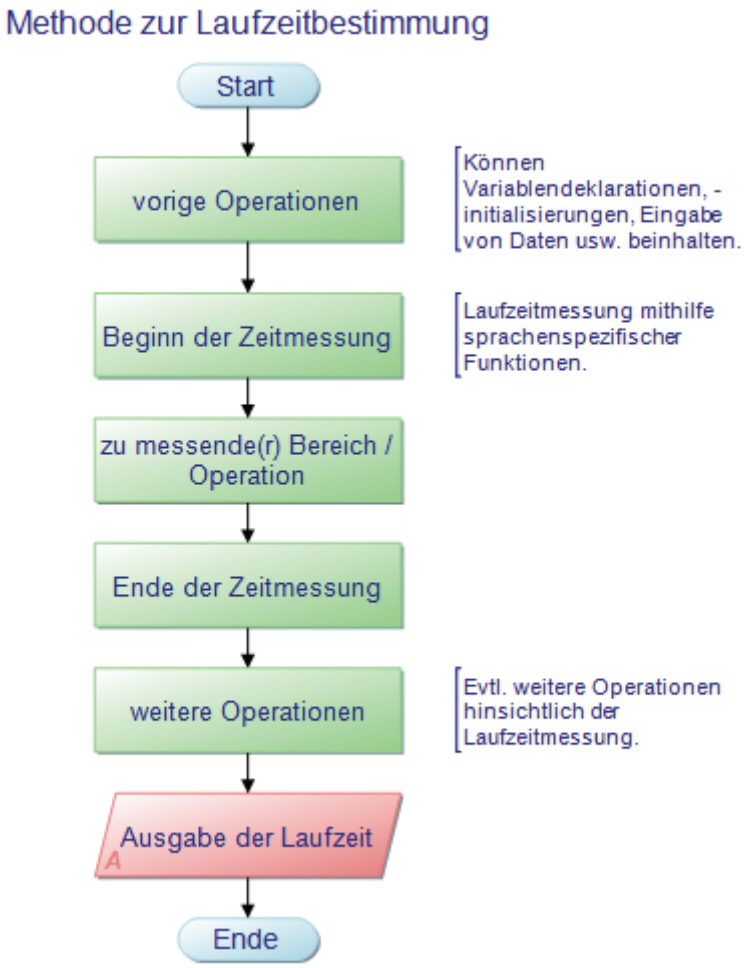


Abbildung 3: Programmablaufplan - Methode zur Laufzeitbestimmung

2.5.3 Instrumente für die Laufzeitmessung

Je nach Programmiersprache gibt es unterschiedliche Instrumente, die der manuellen Laufzeitbestimmung dienen können. Im weiteren Verlauf werden daher nur einige Instrumente dargestellt. Die ausgewählten Methoden sollen sich hierbei lediglich auf die Aufgabenstellung dieser Arbeit beschränken. Weiterhin sollen Grundlagen, wesentliche Funktionen sowie deren Eigenschaften hinsichtlich der Zeitmessung, der zu vergleichenden Sprachen aufgezeigt werden. Für die Messung in umfangreichen und vielschichtigen Programmen werden u. a. andere Herangehensweisen sowie weitere Methoden und Werkzeuge benötigt.

Hinweis: Das folgende in Teilen dargestellte Programm (Listing 1, Listing 2 und Listing 3), dient lediglich der Grundlagenforschung von elementaren Funktionen, die der Laufzeitbestimmung dienen. Es wurden ggf. wichtige Voraussetzungen und Aspekte nicht ausreichend berücksichtigt. Der in C geschriebene Quelltext soll außerdem äquivalent für die Herangehensweise von Programmen in C++ und Java fungieren, weshalb im Folgenden für C++/Java nur noch entscheidende Funktionen und Aspekte erläutert werden. Eine ausführliche Programmdarstellung wie in C wird somit nicht vorgenommen, da die Sprachen historisch betrachtet zudem auf C basieren und folglich korrespondierende Merkmale besitzen.

C

Für die Arbeit mit der Zeit (schließt nicht nur die Uhrzeit, sondern auch das Datum ein) stehen in der C-Bibliothek einige Funktionen bereit. Hierbei sind Definitionen und Prototypen in der Header-Datei `time.h` festgelegt. Zur Darstellung der Zeit verwenden die Funktionen in C zwei unterschiedliche Arten. Das erste Verfahren stellt die Zeit in Sekunden, seit 00:00 Uhr des 1. Januars 1970 (UNIX-Epoche) dar. Das zweite Verfahren verwendet hingegen einzelne Elemente, um die Zeit wiederzugeben. Die Elemente können hierbei z. B. Jahr, Monat, Tag, Stunde, usw. abbilden. Um die Laufzeit in C zu messen, gibt es nach [AJ 00] die in den weiteren Programmteilen dargestellten Möglichkeiten.

```
1.  #include <stdio.h>
2.  #include <time.h>

3.  /* Funktion um trivial Programmzeit zu verbrauchen */
4.  void waste_time () {
5.    int i;
6.    for (i=0;i<1000000000;i++);
7.  }
```

Listing 1: Teil 1 - Funktionen zur Laufzeitbestimmung in C [nach Quelle: AJ 00]

In Zeile 1 / Listing 1 wird zunächst die Header-Datei `time.h` eingebunden. Hierdurch wird die Ermittlung von zeitabhängigen Werten unter Verwendung vorhandener Funktionen unterstützt. Die in Zeile 4-7 deklarierte Funktion dient lediglich der Erzeugung bzw. dem Verbrauch von Programmzeit. Es handelt sich dabei um eine leere Schleife, die so oft wiederholt wird, bis Auswirkungen auf die Laufzeit festzustellen sind.

Die nächste Abbildung (Listing 2) zeigt den ersten Teil des Hauptprogramms. Hier werden in Zeile 9-11 Variablendeklarationen vorgenommen, die der Speicherung von Laufzeiten dienen sollen. In Zeile 12 wird die in Zeile 4-7 beschriebene Funktion aufgerufen, die einige Sekunden für die Ausführung (systemabhängig) in Anspruch nimmt. Die erste Möglichkeit zur Berechnung von Zeitunterschieden wird mithilfe von `time()` realisiert. Diese Funktion wird in C verwendet um die aktuelle Zeit zu ermitteln, `time()` liest hierbei die Zeit der internen Uhr aus. Der Rückgabewert, welcher der Variable `beginn_time` in Zeile 14 zugewiesen wird, enthält die Sekunden, die seit 00:00 Uhr des 1. Januars 1970 vergangen sind.

```
8.  int main (void) {
9.    time_t beginn_time, ende_time;
10.   long beginn_clock, ende_clock;
11.   double dauer_time, dauer_clock;

12.   waste_time();

13.   /* Die Zeit des Programmstarts festhalten. */
14.   beginn_time = time(0);
15.   beginn_clock = clock();

16.   waste_time();

17.   /* Liest die aktuelle Zeit ein und berechnet die Programmdauer. */
18.   ende_time = time(0);
19.   ende_clock = clock();

20.   dauer_time = difftime(ende_time, beginn_time);
21.   dauer_clock = ende_clock - beginn_clock;
```

Listing 2: Teil 2 - Funktionen zur Laufzeitbestimmung in C [nach Quelle: AJ 00]

Das zweite Verfahren arbeitet hingegen mir der Funktion `clock()`. Hierfür wird der Variable `beginn_clock` in Zeile 15, die seit Programmstart vergangene CPU-Zeit zugewiesen. Des Weiteren ist es für die Laufzeitbestimmung von großer Bedeutung, dass die Funktion `clock()` die Zeit nach [AJ 00] mit einer Auflösung von 1/100 Sekunde liefert. Sollte die CPU-Zeit allerdings nicht verfügbar sein, so liefert die Funktion den Wert "-1".

Um nach der Variablenzuweisung noch einige Zeit zu verbrauchen, wird in Zeile 16 (Listing 2) wieder die Funktion `waste_time()` aufgerufen. In Zeile 20 und 21 werden wiederum zwei Methoden dargestellt, um Zeitunterschiede zu berechnen. Die in Zeile 20 aufgeführte Funktion `difftime()`, subtrahiert die Zeit des ersten Argumentes mit der des Zweiten. Also in diesem Beispiel die Differenz von `ende_time` und `beginn_time`. Da der Aufruf dieser Funktion nicht zwingend notwendig ist, sondern auch mittels Subtraktions-Operator durchgeführt werden kann, wurde in Zeile 21 diese grundlegende arithmetische Operation verwendet.

```
22.   printf("CLOCKS_PER_SEC = %ld\n", CLOCKS_PER_SEC);
23.   printf("Ausfuehrungszeit mit time = %f Sekunden.\n", dauer_time);
24.   printf("Ausfuehrungszeit mit clock = %lf Sekunden.\n",
25.          dauer_clock/CLOCKS_PER_SEC);
26.   printf("Ausfuehrungszeit mit clock = %ld Hundertstel Sekunden.\n",
27.          (long)dauer_clock);
28.   printf("Ausfuehrungszeit mit clock seit Programmstart = %ld
29.          Hundertstel Sekunden.\n \n", clock());
30.   system("pause");
31. }
```

Listing 3: Teil 3 - Funktionen zur Laufzeitbestimmung in C [nach Quelle: AJ 00]

Die in Zeile 22-29 (Listing 3) beschriebenen Ausgaben sind für eine Bewertung der ermittelten Laufzeitunterschiede vorgesehen. Hierfür sei auf den nächsten Screenshot (Abbildung 4) verwiesen, dieser zeigt die Ausgaben des ausgeführten Programms auf Kommandozeilenebene. In Zeile 23 erfolgt die Ausgabe der ersten berechneten Zeit, die mithilfe der Funktion `time()` gemessen wurde. Die Ausgabe zeigt, dass die Laufzeit nur in ganze Sekunden ausgegeben wird, was die Genauigkeit somit erheblich begrenzt.

Um mit der Funktion `clock()` die gemessene Laufzeit in Sekunden auszugeben, muss das Ergebnis durch die Konstante **`CLOCKS_PER_SEC`** (alte Bezeichnung: `CLK_TCK`) dividiert werden (s. Zeile 25). Diese Konstante ist nach [TUC 01] wie folgt definiert: "Der Teiler `CLOCKS_PER_SEC` ist maschinenabhängig und spezifiziert, wie viel Uhr-Ticks auf der jeweiligen Maschine pro Sekunde vergehen.". Diese Konstante definiert demzufolge die `clock_t`-Einheiten pro Sekunde. Des Weiteren ist in der Abbildung 4 zu erkennen, dass die Ausführungszeit mit `clock()` eine Genauigkeit im Millisekundenbereich erzielt. Im Gegensatz zu `time()` wäre `clock()`, aufgrund der höheren Genauigkeit besser zur Bestimmung von Laufzeiten geeignet. Lediglich zur Grundlagen-Analyse und Übersicht wurde in Zeile 28 noch eine direkte Ausgabe von `clock()` umgesetzt. Diese zeigt die vergangene CPU-Zeit (im Hundertstel- bzw. Millisekundenbereich) seit Programmstart.

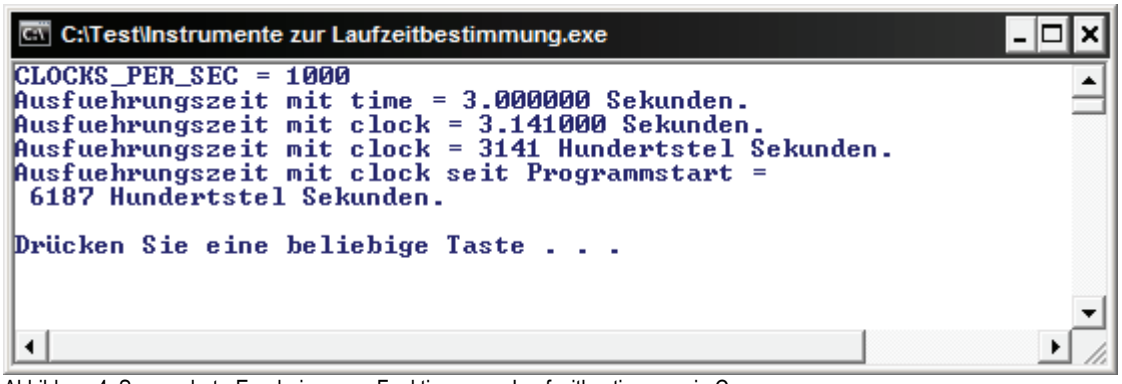


Abbildung 4: Screenshot - Ergebnisse von Funktionen zur Laufzeitbestimmung in C

Aus dieser Untersuchung von Funktionen zur Laufzeitmessung in C ergibt sich, dass eine individuelle Implementierung mithilfe von `clock()` und der maschinenabhängigen Konstante `CLOCKS_PER_SEC`, realisiert werden kann. Das aufgeführte Beispiel zeigt, dass es unter geringem Aufwand möglich ist, eine angepasste Bestimmung der Laufzeit vorzunehmen und außerdem eine ausreichend hohe Genauigkeit von 1/100 Sekunde zu erreichen.

C++

C++ stellt wie C eine `clock()`-Funktion zur Verfügung. Das Prinzip dieser Funktion ist wieder gleich dem von C. Als Rückgabewert liefert die Funktion, die seit dem Programmstart vergangene CPU-Zeit ("Clocks"). Um den Rückgabewert in Form von Sekunden zu erhalten, kann wieder auf die Konstante `CLOCKS_PER_SEC` zurückgegriffen werden, i.d.R. entspricht ein "Clock" genau einer Millisekunde (maschinenabhängig). Auf Basis gleicher Funktionen sollte eine im Vergleich zu C, sehr ähnliche Implementierung möglich und vergleichbare Ergebnisse zu erzielen sein.

Java

Für die einfache Zeitmessung in Java stehen nach [ULL 09] zwei Funktionen zur Verfügung. Zum einen gibt es die Funktion `static long currentTimeMillis()`, welche die vergangenen Millisekunden seit Mitternacht des 01.01.1970 zurückgibt. Zum anderen gibt es die Methode `static long nanoTime()`, diese liefert die Zeit in Nanosekunden vom genauesten System-Zeitgeber. Um die Funktionen nutzen zu können, ist es nötig die Klasse `final class java.lang.System` einzubinden. Diese enthält neben Funktionen zum Ermitteln der Zeit auch noch zahlreiche andere Funktionen. Da in C und C++ keine explizite Funktion zur Laufzeitbestimmung im Nanosekundenbereich vorhanden ist, soll auch in Java auf die Funktion `currentTimeMillis()` zurückgegriffen werden. Somit ist in allen drei Sprachen, eine gleich genaue Funktion zur Messung der Laufzeit im Millisekundenbereich vorhanden.

Zusammenfassung

Die abschließende Tabelle gibt einen Überblick von Methoden zur Laufzeitbestimmung sowie den wichtigsten Eigenschaften in C/C++ und Java. Für detailliertere Angaben zu den spezifischen Funktionen sei auf die jeweiligen Bibliotheken bzw. Spezifikationen verwiesen.

Zusammenfassung - Methoden für die Laufzeitbestimmung			
Funktionen - C/C++	**Rückgabewert**	**Genauigkeit**	**Besonderheit**
`time()`	- Sekunden die seit 00:00 Uhr des 1. Januars 1970 vergangen sind	- auf die Sekunde genau	- geringe Zeitauflösung
`clock()`	- die seit Programmstart vergangene CPU-Zeit in Form von "Uhr-Ticks" (auch als "CPU-Ticks" bzw. "Clocks" bezeichnet)	- im Hundertstel- bzw. Millisekundenbereich liefert die Funktion eine Auflösung von 1/100 Sekunde	- parallele Prozesse haben keinen Einfluss auf die Zeitmessung
`gettimeofday()`	- vergangene Sekunden seit 00:00 Uhr, 1. Januar 1970 in Form einer »`timeval`«-Struktur	- bis auf Mikrosekunden genau	- wie exakt eine Messung ist hängt von der jeweiligen Hardware ab
Funktionen - Java	**Rückgabewert**	**Genauigkeit**	**Besonderheit**
`currentTimeMillis()`	- vergangene Millisekunden seit Mitternacht des 01.01.1970	- Millisekundenbereich	-
`nanoTime()`	- Nanosekunden vom genauesten System-Zeitgeber	- Nanosekundenbereich	-

Tabelle 4: Zusammenfassung - Methoden für die Laufzeitbestimmung [nach Quellen: AJ 00, ULL 09, WIL 01]

[Punkt 2.5.3 nach Quellen: AJ 00, TUC 01, ULL 09, WIL 03, WOL 06]

2.6 Aspekte von Hard- und Software

Hard- und Software haben einen maßgeblichen Einfluss auf die Laufzeit von Anwendungen. Die nächsten Punkte zeigen einige Faktoren von Hard- und Software, die eine Auswirkung auf die Laufzeit bewirken. Dabei ist zu beachten, dass sich detaillierte Angaben zur Hard- bzw. Software auf den Stand[1] dieser Arbeit beziehen.

2.6.1 Einflussfaktoren der Hardware

Aufseiten der Hardware ist die Ausführungsgeschwindigkeit von Programmen mit Bezug auf die laufzeitrelevanten Bereiche grundsätzlich durch die folgenden physikalischen Komponenten begrenzt:

- Hauptprozessor,
- Arbeitsspeicher,
- Festplatte.

Neben den aufgeführten Komponenten gibt es eine Vielzahl weiterer Bestandteile (z. B. interne Bussysteme), die einen weiteren Einfluss bewirken. An diesen Komponenten sollen jedoch beispielhaft Eigenschaften und deren maßgebliche Einflussgrößen auf die Laufzeit benannt werden.

Hauptprozessor

Ein **Prozessor** (kurz CPU für Central Processing Unit) ist die zentrale Verarbeitungseinheit eines Systems. Er ist somit für die Abarbeitung der Befehle zuständig. Ein Prozessor besitzt unterschiedliche Komponenten und Merkmale, die dessen Leistungsfähigkeit bestimmen. Als wichtigstes Merkmal kann die **Wortbreite** (auch als Datenwort bzw. Wort bezeichnet) für Register, Datenbus, Adressbus und Steuerbus angesehen werden. Dieses gibt die Anzahl der Bits an, aus dem ein Maschinenwort besteht. Je breiter ein Maschinenwort, umso mehr verschiedene Zustände oder Werte können von einem Prozessor in einem Durchlauf verarbeitet werden. Beispielsweise bestimmt die Wortbreite von Registern die maximale Größe von Ganz- und die Genauigkeit von Fließkommazahlen. Die Datenbusbreite hingegen ist entscheidend für die Anzahl der Bits, die in dem Hauptspeicher geschrieben oder ausgelesen werden können.

Eine untergeordnete Rolle hinsichtlich der Arbeitsgeschwindigkeit eines Prozessors spielt die Taktfrequenz. So müssen hohe Taktraten aufgrund komplexer Befehle oder einer langsamen Ein- oder Ausgabe nicht mit einer steigenden Effizienz/Arbeitsgeschwindigkeit einhergehen. Die Taktfrequenz wird durch den Multiplikator und die Taktrate des Front Side Bus (FSB) bestimmt und beschreibt die Anzahl der Takte pro Sekunde, die verarbeitet werden können. Um Anhaltspunkte bezogen auf die tatsächliche Leistung einer CPU zu erhalten, sollte auf die Einheiten MIPS und FLOPS zurückgegriffen werden. **MIPS** (Million Instructions per Second) beschreibt die Anzahl von Maschinenbefehlen (in Millionen), die pro Sekunde ausgeführt werden können. Demgegenüber gibt die Einheit **FLOPS** (Floating Point Operations per Second) die pro Sekunde durchführbare Anzahl von Fließkommaoperationen an.

Der Einsatz von **Caches** ist ein weiterer wichtiger Aspekt um die Arbeitsweise weiter zu beschleunigen. Diese fungieren als schnelle Zwischenspeicher für Daten- und/oder Befehle und ermöglichen so einen sehr schnellen Zugriff. Bei diesen Speichern kann wiederum eine Unterscheidung zwischen Level-1- und Level-2-Cache getroffen werden. So ist der Level-1-Cache (L1-Cache) direkt im Prozessorkern integriert und hauptsächlich für die Ausführung sehr kurzer Schleifen aus wenigen Befehlen verantwortlich. Die direkte Implementierung im Prozessor hat den Vorteil, dass der L1-Cache mit der gleichen Taktrate wie der des Prozessors selbst, betrieben werden kann. Im Gegensatz dazu ist der L2-Cache nicht direkt im Kern implementiert. Der dadurch resultierende positive Aspekt ist, dass der L2-Cache erheblich größer als der L1-Cache sein kann. Als Nachteil kann die Nutzung einer niedrigeren Taktrate angesehen werden.

[1] Stand: Mai 2010

Aufgrund der geringen Größe (Kilo- bis Megabyte-Bereich) sowie weiteren Eigenschaften sind Caches nicht als Ersatz für den eigentlichen Arbeitsspeicher geeignet. Da Caches jedoch mit einer vielfachen Geschwindigkeit eines Arbeitsspeichers arbeiten, kommen besonders häufig genutzte Befehle bzw. Daten für eine Speicherung in Betracht. Im Hinblick auf das Laufzeitverhalten spielt die Frage: "Welche Daten werden in einem Cache gespeichert?" eine wichtige Rolle, denn ein Softwareentwickler hat keinen Einfluss auf diese Entscheidung. Vielmehr ist hierfür eine weitere Komponente verantwortlich, die sogenannte **Sprungvorhersage** (engl. Branch Prediction). Auf Basis dieser Komponente berechnet ein Prozessor während der Ausführungszeit, wohin der nächste Sprung einer Anwendung voraussichtlich führt. Anhand dieser Berechnung wird eine Entscheidung getroffen, ob Daten oder Programmteile im Cache abgelegt werden.

Caches besitzen i.d.R. eine Zugriffzeit von 1 bis 5 Takten, externe ein Vielfaches davon. Des Weiteren existieren neben Caches noch Spezialspeicher, diese werden z. B. als Befehlspuffer oder Sprungzielspeicher in Prozessorarchitekturen verwendet.

Neben Caches ist besonders für Server-PCs der Einsatz von mehreren Prozessoren nötig um die Laufzeit von rechenintensiven Operationen weiter zu reduzieren. Für Desktop PCs hat sich hingegen die Verwendung von **Mehrkernprozessoren** (Multicore-Prozessoren) durchgesetzt. Hierbei enthält ein Mehrkernprozessor mehrere einzelne Prozessoren auf einer Einheit. Was wiederum bedeutet, dass die verschiedenen Bestandteile eines Prozessors entsprechend der Anzahl an Kernen, mehrfach vorhanden sind. Diese Art von Prozessoren realisiert besonders für rechenlastige Anwendungen kürzere Laufzeiten. Dabei kann die Leistung dieser Prozessoren jedoch nur ausgenutzt werden, wenn die Aufgaben programmtechnisch entsprechend der Kernanzahl verteilt werden.

Arbeitsspeicher

Eine weitere Komponente, die neben dem Prozessor einen wesentlichen Einfluss auf die Leistung und folglich auf die Laufzeit hat, ist der **Arbeitsspeicher**, Hauptspeicher bzw. engl. RAM (Random Access Memory).

Zur Ausführungszeit werden Programme und Daten in diesem Speicher verarbeitet. Es handelt sich dabei um einen flüchtigen Speicher mit wahlfreiem Zugriff was bedeutet, dass auf jedes Byte bidirektional zugegriffen werden kann (Lesen und Ändern von Inhalten). Im Unterschied zu Massenspeichern erlauben Arbeitsspeicher einen deutlich schnelleren Zugriff auf die Inhalte. Somit hängt die Performance weniger von der Speichertechnologie/Bauform, als von der Größe des Arbeitsspeichers ab. Wird die maximale Größe des Speichers durch Anwendungen ausgenutzt, so werden weiteren Daten z. B. auf der Festplatte ausgelagert (auch als **Paging** / **Swapping** bezeichnet). Das Auslagern und Lesen der Inhalte von der Festplatte führt in der Folge zu einer enormen Zunahme der Laufzeit (vgl. Abschnitt "Festplatte").

Da der Arbeitsspeicher eine ständige Stromversorgung zur Speicherung von Daten benötigt und außerdem nur wenig Speicherplatz bietet, ist der Einsatz von Massenspeichern für eine längerfristige Speicherung von grundlegender Bedeutung.

Festplatte

Die gängigste Form eines Massenspeichermediums ist die **Festplatte** (HDD engl. für Hard Disk Drive), welche bis zu 2 TB an Speicherplatz bieten kann. Dieses Medium besteht aus mehreren rotierenden Metallscheiben, die mit einer ferromagnetischen Schicht versehen sind. Im staubdichten Gehäuse wird das Lesen und Schreiben von Daten durch spezielle Schreib-/Leseköpfe realisiert.

Die Geschwindigkeit mit denen Festplatten arbeiten ist u. a. abhängig von der Drehzahl/ **Umdrehungsgeschwindigkeit** der Metallplatten. Die meisten Platten arbeiten mit einer Drehzahl von 5.400 bis 10.000 U/min (Umdrehungen pro Minute). Die **Übertragungsgeschwindigkeit** von Daten für Lese- und Schreibvorgänge (kurz Übertragungsrate) liefert einen Wert für die Datenmenge, die pro Sekunde übertragen werden kann. In Abhängigkeit von der Ausführung und Bauform können Festplattenlaufwerke, Übertragungsraten in der Spanne von 30 bis 300 MB/s erreichen. Analog zu Prozessoren werden auch für Festplatten, Caches zum Zwischenspeichern von Daten verwendet. Diese sehr schnellen Speicher erhöhen mit zunehmender Größe, die Lese- sowie Schreibzugriffe.

Ein weiterer wichtiger Aspekt ist die **mittlere Zugriffszeit**. Dieser statistisch ermittelte Wert gibt die Zeit an, die gebraucht wird, um einen beliebigen Sektor auf der Festplatte zu erreichen. Die Zugriffszeit wird dabei durch die Spurwechselzeit, Latenzzeit und Kommando-Latenz beeinflusst.

- **Spurwechselzeit** (Zeit, die der Schreib-/Lesekopf braucht, um einen bestimmten Weg zurückzulegen. Zusätzlich wirkt sich die Antriebsstärke des Schreib-/Lesekopfes auf die Spurwechselzeit aus.)
- **Latenzzeit** (Von der Umdrehungsgeschwindigkeit abhängige Zeit, bis ein entsprechender Bereich/Sektor am Schreib-/Lesekopf vorbeigeht. Für den Mittelwert ergibt sich eine Dauer von einer halben Umdrehung.)
- **Kommando-Latenz** (Vom Festplattencontroller abhängige Zeit um ein Kommando zu verarbeiten. Diese Zeit kann mit einem Wert von ca. 0,2 Millisekunden bemessen werden.)

Für aktuelle Festplatten ergeben sich hieraus Werte, die nach [KER 08] in einem Bereich von ungefähr 10 ms liegen. Hinsichtlich der Performance zum Arbeitsspeicher sind 10 ms jedoch sehr viel, demgegenüber betragen die Zugriffszeiten für Arbeitsspeicher nach [WIKI 10] nur etwa 25 ns. Statistisch gesehen braucht eine Festplatte somit 400.000-mal länger für einen Zugriff als der Arbeitsspeicher. Es ist demzufolge wichtig zu erwähnen, dass bei einer Laufzeitmessung in der die Performance von der Festplatte abhängt, laufzeitkritische Effekte bzw. eine schlechtere Effizienz zu erwarten ist.

Für alle in diesem Punkt aufgeführten Speicherarten gilt grundsätzlich die Regel, dass die Arbeitsgeschwindigkeit von Speichermedien entgegengesetzt zur Speichergröße steht. Die folgende Abbildung soll dabei diesen Aspekt verdeutlichen.

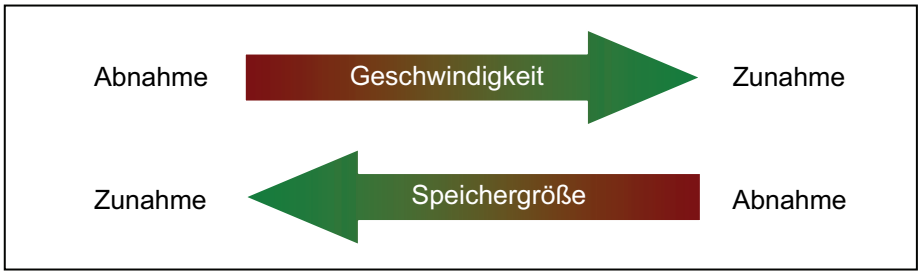


Abbildung 5: Bezug von Geschwindigkeit zur Speichergröße

[Punkt 2.6.1 nach Quellen: KER 08, PR 97, WIKI 09/10, WIS 04]

2.6.2 Software-Faktoren

Neben Einflussfaktoren der Hardware-Komponenten spielt auch die für die Ausführung der Programme erforderliche Software eine entscheidende Rolle.

Betriebssystem

In erster Linie wären Aspekte des zugrunde liegenden Betriebssystems zu benennen. Bei den verwendeten Systemen ist wiederum zwischen Windows- und Linux-Betriebssystem zu unterscheiden. Unabhängig vom konkreten Produkt sind nach [KER 08] die allgemeinen Aufgaben eines Betriebssystems:

- Prozessmanagement (Verteilung von Ressourcen, Ausführung von Aufgaben als Prozesse)
- Speichermanagement (Bereitstellung von ausreichend Speicher für Applikationen)
- Steuerung und Abstraktion der Hardware (Steuerung der Hardware durch Gerätetreiber)
- Ein- und Ausgabesteuerung (Steuerung von unterschiedlichen Ein- und Ausgabekanälen)
- Dateiverwaltung (Bereitstellung von Dateisystemen)
- Bereitstellen der Benutzeroberfläche (im Allgemeinen zwei Arten: GUI (Graphical User Interface) und die Konsole/Shell)

Um die Aufgaben ausführen zu können, besitzt ein Betriebssystem verschiedene Bestandteile, so ist z. B. der **Kernel** für die Steuerung aller Komponenten, Start und Ablauf von Applikationen, Laden und Koordinieren von Treibern für die Hardware zuständig. Auf der anderen Seite sind alle Bestandteile die nicht zum Kernel gehören **Systemprogramme** (z. B. Werkzeuge für die Steuerung und Analyse). Diese können unabhängige Anwendungen darstellen und zu einem beliebigen Zeitpunkt gestartet oder beendet werden.

Des Weiteren verfügt ein Betriebssystem über unterschiedliche Eigenschaften um Einfluss auf die Laufzeit von Anwenderprogrammen zu nehmen. Die im Weiteren aufgelistete Tabelle soll eine auszugsweise Übersicht über einige verschiedene Fähigkeiten, Eigenschaften bzw. Problemen eines Betriebssystems sowie deren Auswirkung auf die Laufzeit zeigen.

Einflusskriterien von Betriebssystemen (Beispiele)		
Eigenschaft	**Beschreibung**	**Auswirkung**
Deadlocks	- Problem, das auftritt, wenn Prozesse auf gleiche Ressourcen zugreifen und anschließend keine Entscheidung getroffen werden kann	- möglicher Absturz der Prozesse oder gar des Systems
Interrupt	- bezeichnet das (temporäre) Unterbrechen von Prozessen bzw. Programmen - Verwaltung erfolgt durch das Betriebssystem	- Anwendungen haben meistens keinen Einfluss auf Unterbrechungen, was wiederum laufzeitkritische Folgen hat
Kernelmode	- dieses Problem bezieht sich auf die Ausführung von Programmcode im Kernelmode	- durch Aufrufe von Prozessen niedriger Priorität, können alle anderen Prozesse blockiert werden
Malloc	- steht für die das Anfordern von neuem Speicher	- kann Paging bzw. Swapping auslösen - Verlängerung der Ausführungszeit
Multitasking	- ist die Fähigkeit scheinbar mehrere Prozesse gleichzeitig auszuführen - bei modernen Systemen wird präemptives Multitasking eingesetzt	- verschiedene Prozesse werden in kurzen Intervallen abwechselt ausgeführt - das System entscheidet wie lange Prozesse Rechenzeit in Anspruch nehmen dürfen
Paging / Swapping	- Aufgabe bzgl. der Speicherverwaltung, bei der Seiten aus dem Arbeitsspeicher auf der Festplatte in einer Auslagerungsdatei gespeichert werden	- bei zu wenig Arbeitsspeicher sind (deutlich) erhöhte Laufzeiten die Folge
Scheduling	- Aufgabe bzgl. der Verwaltung von Programmen	- starten, beenden und verwalten von parallelen Prozessen, Zuteilung von Ressourcen

Tabelle 5: Einflusskriterien von Betriebssystemen (Beispiele) [nach Quellen: FHW 01, KER 08, MSP 97]

Laufzeitumgebung

Um Java-Anwendungen ausführen zu können, wird neben dem Betriebssystem noch weitere Software benötigt, die sogenannte **Java Laufzeitumgebung** (JRE bzw. Java Runtime Environment). Grundsätzlich kann die JRE als Schnittstelle zwischen Betriebssystem und Java-Applikationen aufgefasst werden.

Dabei beinhaltet die Laufzeitumgebung u. a. die **Java Virtual Machine** (JVM / Java-VM), welche für die Ausführung und Überwachung der Anwendungen zuständig. Durch den Einsatz der JVM wird die Plattformunabhängigkeit der Java-Applikationen realisiert, da hierbei nur der kompilierte Java-Byte-Code interpretiert wird. Im Gegensatz dazu wird durch die Nutzung zusätzlicher Software für die Programmausführung, weiterer Overhead erzeugt. Dies äußert sich u. a. darin, dass sowohl Zeit für den Start als auch Ressourcen für den Betrieb der JRE benötigt werden. So kann beispielsweise die Überwachung der JVM auf mögliche Überschreitungen von Speicherbereichen, laufzeitrelevante Nachteile mit sich ziehen. Die Sicherheitsarchitektur, welche die JVM umfasst, kann als ein dreistufiges Modell aufgefasst werden. Diese besteht aus den aufgeführten Komponenten:

- "Class Loader" (ist für das Laden von benötigten Klassen zuständig)
- "Byte-Code Verifier" (Formatüberprüfung des Byte-Codes)
- "Security Manager" (überprüft Restriktionen von auszuführenden Klassen, anhand eines gültigen Sicherheitsmodells)

Die Folgenden nach [WOL 99] aufgeführten Punkte sind Merkmale bzw. Aufgaben der JVM, welche zudem Einfluss auf die Laufzeit bewirken können.

- Überprüfung der Indizes von Arrays/Feldern auf Überschreitungen bzgl. der Speicherbereiche
- Referenzüberprüfungen auf nicht zulässige Werte
- automatische Speicherverwaltung durch den "Garbage Collector" (Reservierung, Allokation und Freigabe von Speicher)
- Ausnahmebehandlung (Behandlung von Fehlern während der Programmausführung)
- Typsicherheit (Typumwandlung von Referenzen)
- Absicherung beim Speicherzugriff (ohne Nutzung von Zeigerdatentypen wie bei C/C++)

Aus den aufgeführten Punkten ergibt sich das Sicherheitsmechanismen (wie z. B. die Behandlung von Ausnahmen während der Programmlaufzeit), eine nachteilige Auswirkung auf die Leistung von Programmen haben. Zusätzliche Überprüfungen und Behandlungen von verschiedenen Direktiven legen Nahe, dass hierfür Ressourcen benötigt und so die Laufzeiten von Java-Anwendungen beeinträchtigt werden.

[Punkt 2.6.2 nach Quellen: JVM 01, KER 08, WOL 99]

Kapitel 3

3 Algorithmen

Dieses Kapitel dient der Darstellung von Algorithmen für die Bereiche: Prozessorauslastung bzw. Rechenlast, Speicherverwaltung, Ein- und Ausgabeoperationen (I/O-Operationen) bzgl. der Dateiarbeit. Zu Beginn soll zunächst eine Definition eines »Algorithmus« gegeben werden. Im Weiteren werden Vorgehensweisen für die Erstellung von entsprechenden Algorithmen beschrieben sowie eine Darstellung dieser in Form von Programmablaufplänen (PAP).

3.1 Definition

Programme, die in einer bestimmten Programmiersprache geschrieben sind, charakterisieren im Wesentlichen Algorithmen. Im Hinblick auf die Prozessorauslastung / Rechenlast, Speicherverwaltung sowie Ein- und Ausgabeoperationen bzgl. Dateiarbeit der Programmiersprachen C, C++ und Java sollen Algorithmen entwickelt werden, die speziell auf die jeweiligen Bereiche angepasst sind.

Ein Algorithmus lässt sich hierbei wie folgt definieren:

> "Ein Algorithmus ist eine finite (also endliche) Folge von Schritten, zur Lösung eines logischen oder mathematischen Problems." [aus Quelle: MSP 97]

In den anschließenden Punkten werden wesentliche Kriterien, die bei den Laufzeitvergleichen eine Rolle spielen benannt und es wird auf die jeweiligen Algorithmen näher eingegangen.

[Punkt 3 nach Quellen: MSP 97, PR 97]

3.2 CPU-Auslastung / Rechenlast

Um eine hohe Prozessorauslastung / Rechenlast zu erreichen, muss ein rechenintensiver Algorithmus ausgewählt werden. Es sollten über einen längeren Zeitraum, viele rechenintensive bzw. aufwendige Berechnungen durchgeführt werden, um ausreichende Ergebnisse zur Bewertung beobachten zu können.

Die im Folgenden aufgeführten Punkte zeigen die wesentlichen Kriterien, eines rechenintensiven Berechnungsalgorithmus.

1. Aufwendige bzw. anspruchsvolle Berechnungsoperationen
2. Berechnungen müssen sich über einen längeren Zeitraum erstrecken

Hierbei stellt sich die Frage, was ist eigentlich eine rechenintensive Berechnungsoperation? Die Auswahl der Operation fiel auf eine Matrizenmultiplikation. Da auch eine Matrizenmultiplikation für einen aktuellen Prozessor kein zeitliches Problem mehr darstellt, sollte es sich um Matrizen mit sehr vielen Elementen pro Dimensionen (E/Dim) handeln. Darüber hinaus sollte die Anzahl der Elemente je Dimension kontinuierlich erhöht und die Multiplikation der Matrizen entsprechend oft wiederholt werden, bis hinreichende Ergebnisse für Laufzeitvergleiche festgestellt werden können. Die Elemente pro Dimension einer Matrix stellen in diesem Bereich die Problemgrößen dar.

Der nachstehende Algorithmus beschreibt die Berechnung einer Matrizenmultiplikation von zwei gleichen 2D-Matrizen und somit den für die Laufzeitmessung relevanten Programmabschnitt. Deklarationen von Variablen, Matrizen und Ähnlichem sowie die Belegung dieser, werden nicht dargestellt.

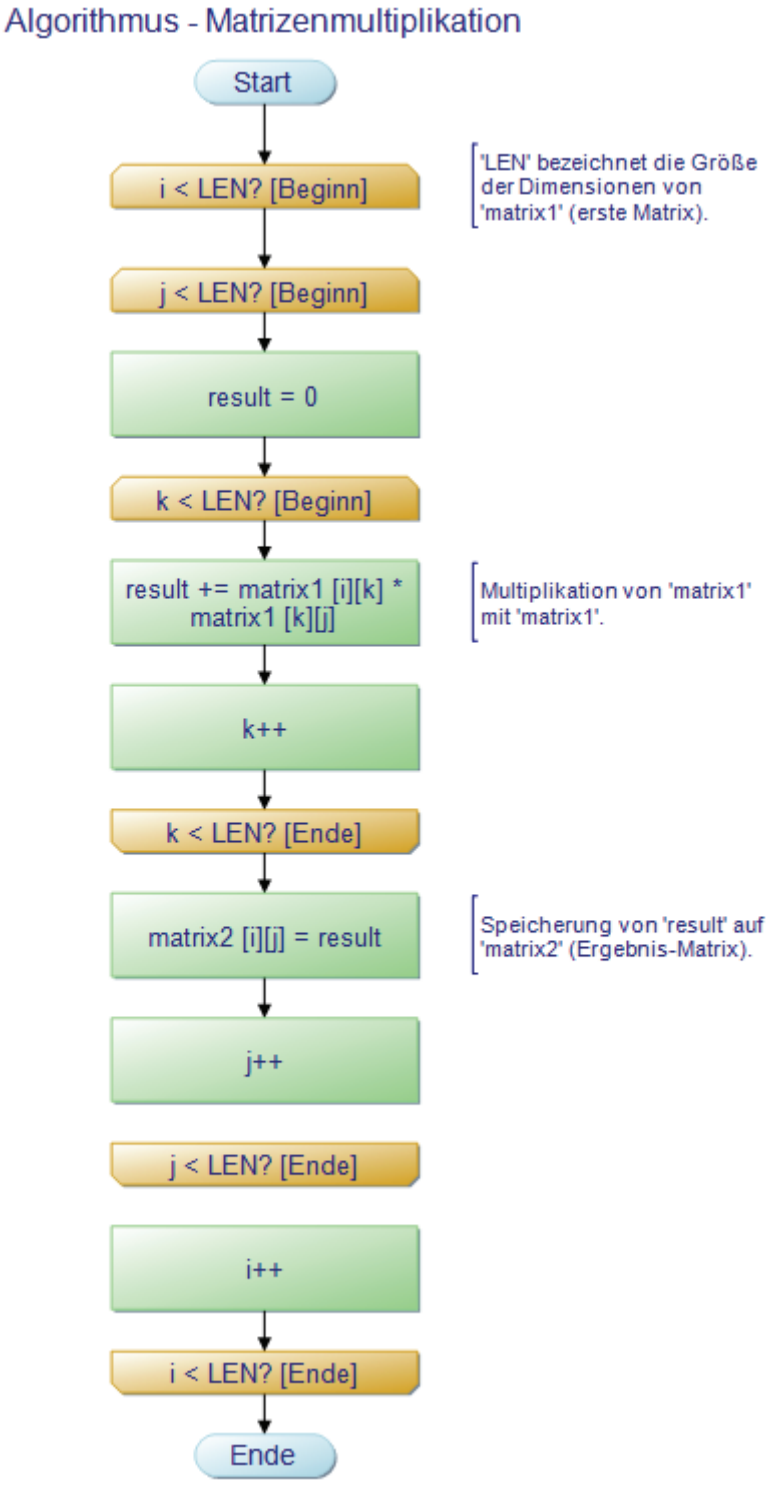


Abbildung 6: Programmablaufplan - Algorithmus Matrizenmultiplikation

3.3 Speicherverwaltung

Für den Bereich Speicherverwaltung wurde ein Vorgang gewählt, bei dem der Aspekt auf der Reservierung von Hauptspeicher liegt. Hierbei soll für steigende Problemgrößen dynamisch Speicher zur Verfügung gestellt werden. Dieser Vorgang steht im Weiteren unter den Begriffen "Speicher ausfassen" bzw. "**Allokation**".

Der folgende Algorithmus (Abbildung 7) beschreibt vereinfacht, dass Verketten von zwei Listen und das anschließende Zusammenfügen in einer Ergebnisliste.

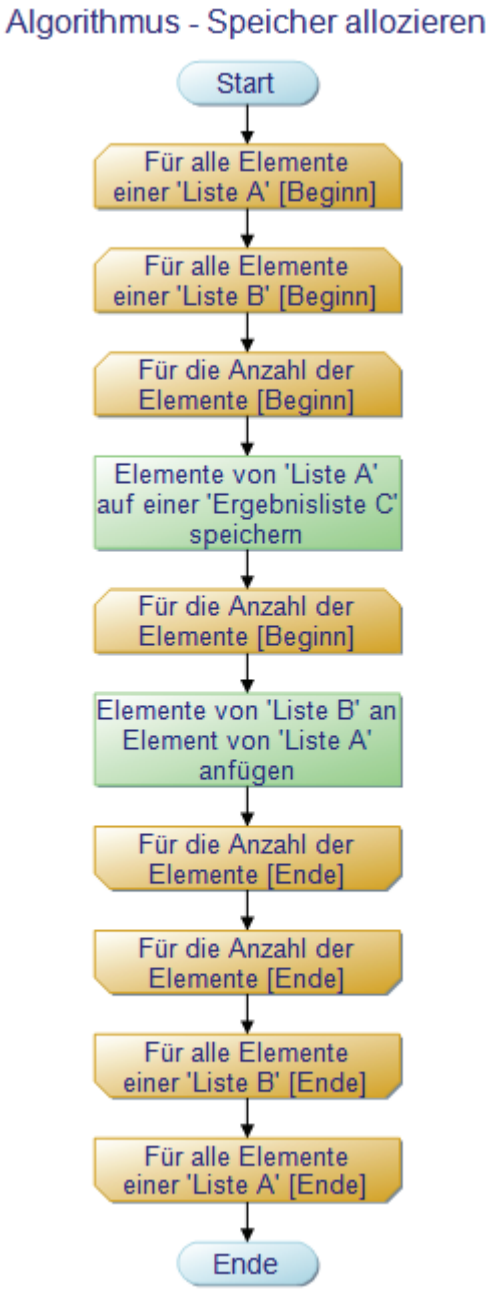


Abbildung 7: Programmablaufplan - Algorithmus Speicher allozieren

Zu Beginn werden jeweils die Dimensionen der Listen A, B für die Anzahl enthaltener Elemente durchlaufen. Dabei werden im weiteren Verlauf zwei weitere Schleifen benötigt, um jeweils ein Element aus der Liste A und B auf einer Ergebnisliste zu speichern. Die Ergebnisliste besteht aus der Verkettung aller Elemente der Liste A mit denen aus der Liste B.

3.4 I/O-Operationen bzgl. Dateiarbeit

Der Algorithmus für den Bereich "Ein- und Ausgabeoperationen bzgl. der Dateiarbeit" beschreibt grundlegend das Kopieren einer Datei. Dabei werden Daten einer Datei zeichenweise eingelesen und anschließend wieder zeichenweise in eine Zieldatei geschrieben. Zum Einlesen der Daten wären auch weitere Vorgehensweisen wie z. B. das byteweise einlesen denkbar, jedoch sollte eine möglichst hohe Auslastung im I/O-Bereich bzw. Verweildauer für die Inanspruchnahme der Operationen erreicht werden.

Der folgende Algorithmus stellt hierbei, den für die Messung der Laufzeit wichtigen Programmabschnitt dar:

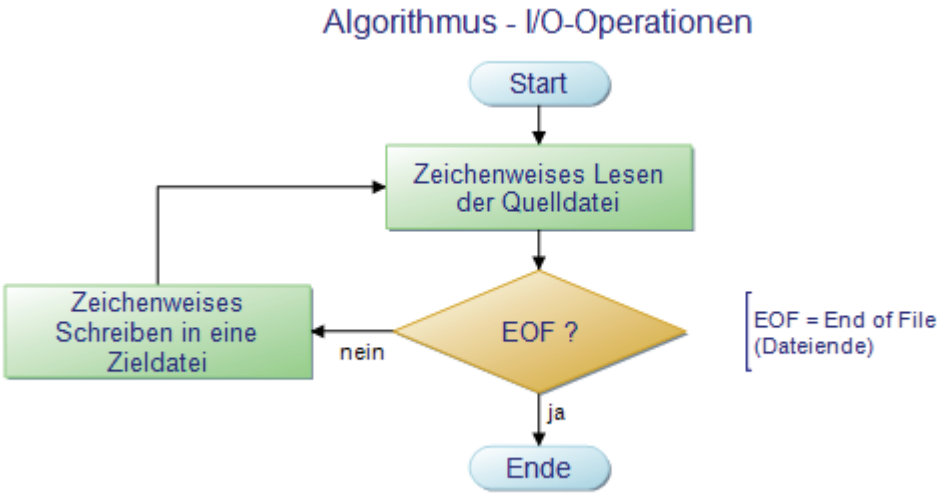


Abbildung 8: Programmablaufplan - Algorithmus I/O-Operationen

Der Programmablaufplan zeigt, dass im ersten Schritt ein Zeichen aus einer Quelldatei gelesen wird. Im Anschluss an diese Operation muss eine Prüfung auf das Dateiende (EOF) erfolgen. Ist das Dateiende noch nicht erreicht, kann das gelesene Zeichen in eine Zieldatei geschrieben werden. Dieser Vorgang wird im weiteren Verlauf solange wiederholt, bis ein entsprechendes Ende erreicht wird.

Kapitel 4

4 Durchführung

Dieses Kapitel beschreibt die Aspekte der Umsetzung sowie der damit verbundenen Vorgehensweise. Hierbei werden eingangs für die Durchführung verwendete Entwicklungsumgebungen und Compiler sowie das Testsystem aufgeführt. Weiterhin sollen einige Möglichkeiten zur Optimierung bzgl. des Laufzeitverhaltens aufgezeigt werden. Der weitere Verlauf dient der Beschreibung des Testaufbaus hinsichtlich Implementierung, Laufzeitbestimmung, Ausführung und Auswertung.

Die folgende Abbildung soll den prinzipiellen Ablauf verdeutlichen, dabei wird in den nächsten Punkten auf die im Programmablaufplan wesentlichen Schritte eingegangen.

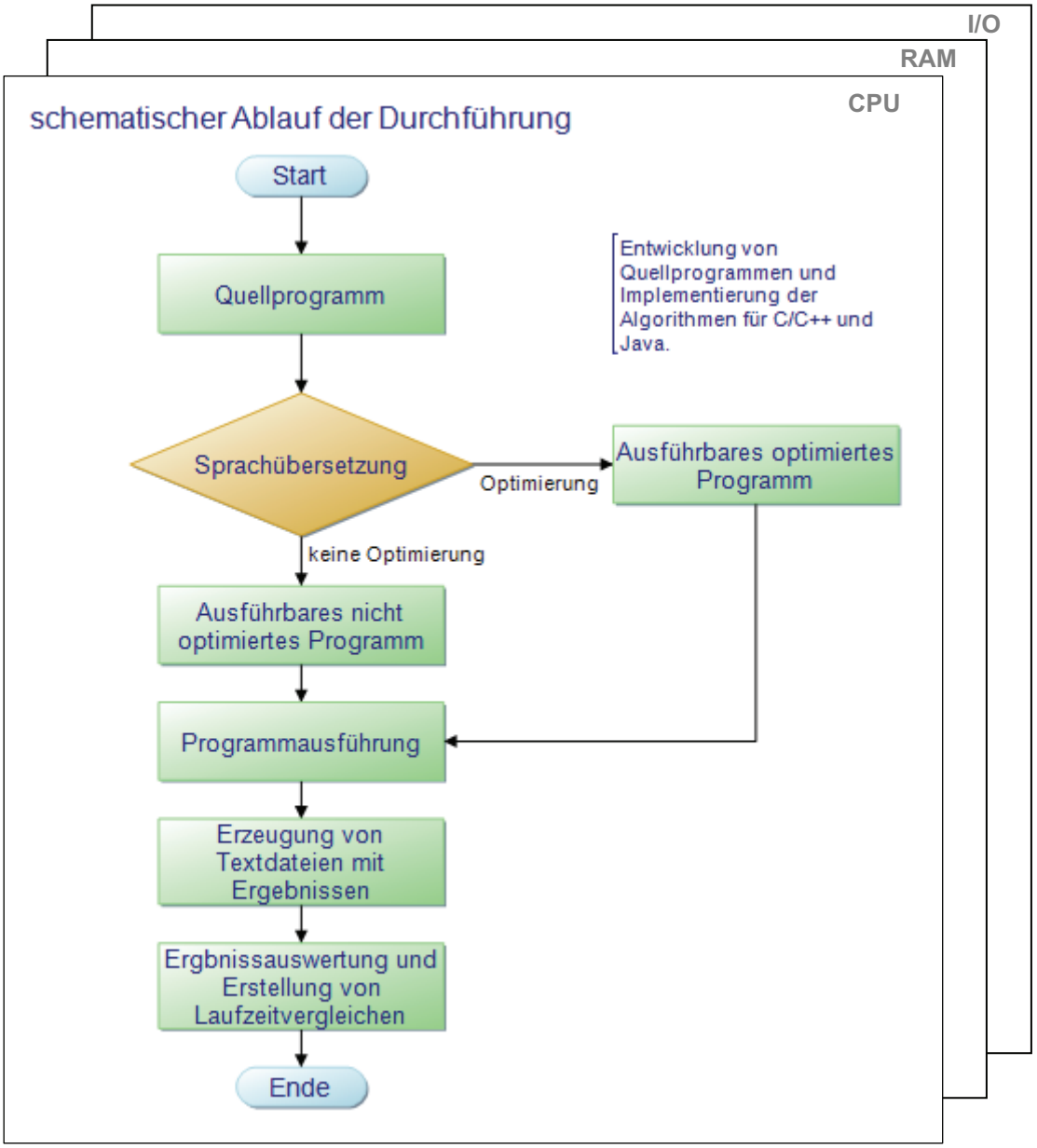


Abbildung 9: Programmablaufplan - schematischer Ablauf der Durchführung

Der dargestellte Ablauf für die Durchführung bezieht sich dabei auf Programme der Sprachen C/C++ und Java sowie für die Bereiche Prozessorauslastung / Rechenlast (CPU), Speicherverwaltung (RAM) sowie Ein- und Ausgabeoperationen bzgl. Dateiarbeit (I/O).

4.1 Testsystem

Die folgende Tabelle zeigt die detaillierte Hardware- und Software-Konfiguration des Testsystems, welches für die Laufzeitmessungen verwendet wurde. Die Informationen wurden dabei mithilfe der Anwendung *"System Information for Windows (SIW) 2010 Build 0310"* ermittelt.

Hardware-/Software-Konfiguration - Testsystem		
Komponente	**Produkt**	**Eigenschaften**
Prozessor	Intel Core 2 Quad Q8200 (Yorkfield)	- Technologie: 64 Bit / 45 nm CPU - Taktfrequenz: 4x 2,33 GHz (4 Kern-Prozessor) - L2-Cache: 2x 2048 KB - Front-Side-Bus: 1333.5 MHz
Arbeitsspeicher	Kingston DDR2 (PC2-6400)	- Speichergröße: 2x 2048 MB - Speed: 400 MHz (DDR2 800) - Latenzzeit: 5-5-5-31 (CL-RCD-RP-RAS)
Mainboard	ASUS P5N-D	- Sockel: LGA775 - BIOS: Phoenix (ASUS P5N-D ACPI BIOS Revision 1101)
Festplatte	Western Digital - WDC WD1600AAJS-00L7A0	- Baumform/Technologie: 3,5" Serial ATA (SATA) - Speichergröße: 160 GB - Cache: 8192 KB - Umdrehungsgeschwindigkeit: 7200 U/min - Transferrate: bis 300 MB/s - Zugriffszeit: 8,9 ms
Betriebssystem 1	Windows OS[2]	- Microsoft Windows XP Professional - Kernel Version: 5.1.2600.5857 (Version 2002) - Service Pack 3 - Java(TM) SE Runtime Environment (build 1.6.0_19-b04)
Betriebssystem 2	Linux OS	- Ubuntu 9.10 - *Karmic Koala* / Linux 2.6.31-20-generic - Gnome Desktop: Version 2.28.1 - Java(TM) SE Runtime Environment (build 1.6.0_19-b04)

Tabelle 6: Hardware-/Software-Konfiguration - Testsystem [ermittelt mit "SIW 2010 Build 0310"]

Das verwendete Testsystem war aufseiten der Hard- und Software bereits fertig installiert sowie vorkonfiguriert. Weitere benötigte Software (wie z. B. Compiler und Java Laufzeitumgebung) wurden entsprechend installiert (s. auch Punkt 4.3 Sprachübersetzung und Optimierung).

Damit die auszuführenden Anwendungen nicht von anderen Vorgängen beeinflusst werden, wurden einige Betriebssystemprozesse deaktiviert (vgl. 4.4 Ausführung). Diese weitere Konfiguration wurde (so äquivalent wie möglich) für beide Betriebssysteme vorgenommen.

[2] OS → Operating System (Betriebssystem)

4.2 Implementierung

Die Algorithmen, im Hinblick auf die Bereiche CPU-Auslastung/Rechenlast, Speicherverwaltung sowie Ein- und Ausgabeoperationen bzgl. Dateiarbeit wurden so äquivalent wie möglich in Applikationen der Programmiersprachen C, C++ und Java implementiert. Die weiteren Punkte zeigen die wesentlichen Aspekte, die für die jeweiligen Anwendungen von Bedeutung sind.

4.2.1 Bestimmung von Laufzeitunterschieden

Das Bestimmen der Laufzeit von relevanten Programmabschnitten hinsichtlich der verschiedenen Bereiche erfolgt grundsätzlich durch die Bildung, der Differenz aus der Zeit des Startpunktes sowie der Zeit eines Endpunktes.

Folgendes an C-Syntax gehaltenes Beispiel soll für alle erstellten Anwendungen, die allgemeine Bestimmung der Laufzeit darstellen:

```
double beginTime, endTime, diffTime;    // Variablen für die Zeitmessung
// weitere Operationen
            ...
beginTime = clock();                    // Beginn Zeitmessung

/* zu messende Operation */

endTime = clock();                      // Ende Zeitmessung
diffTime = endTime - beginTime          // Berechnung der Laufzeit
            ...
// weitere Operationen
```

Listing 4: Allgemeine Bestimmung der Laufzeit

Bei dieser Methode wird nur die Zeit der relevanten Operationen bzgl. der Problembereiche gemessen, d. h. der Startzeitpunkt (also Beginn der Zeitmessung) liegt unmittelbar vor der zu messenden Operation und der Endzeitpunkt direkt danach. Weitere Operationen (z. B. Bestimmung des Mittelwertes) die der Auswertung / dem Ergebnisvergleich dienen, verfälschen hierbei das Laufzeitergebnis nicht.

Darüber hinaus wurden für Auswertungszwecke in allen Programmen, die folgenden Operationen implementiert, Bildung von:

- Minimalwert,
- Maximalwert,
- Mittelwert und
- Median.

Hinweis: Für Java-Anwendungen wurden die Startzeit(en) der Java-Laufzeitumgebung bzw. Java-VM nicht mit gemessen.

4.2.2 CPU-Auslastung / Rechenlast

Im Weiteren werden Aspekte der Programme zur Berechnung einer 2D-Matrix-Multiplikation zwischen C/C++ und Java für den Problembereich CPU-Auslastung / Rechenlast aufgeführt.

Die 2D-Matrizen zwischen C/C++ und Java wurden aufgrund unterschiedlicher Speicherverwaltungskonzepte, programmtechnisch folgendermaßen abgebildet:

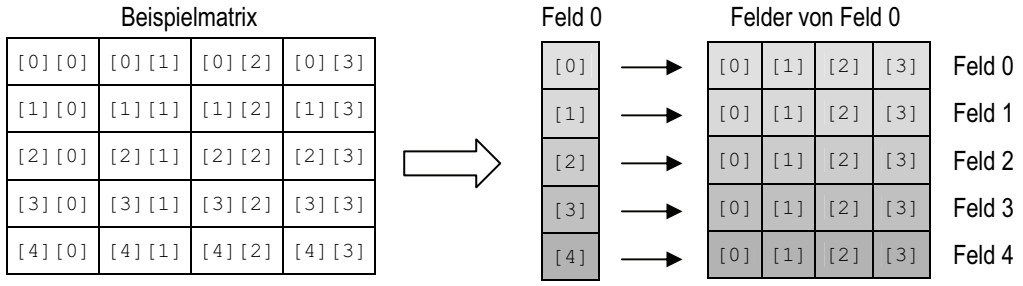


Abbildung 10: Programmtechnische Abbildung einer Matrix

> Diese Art der programmtechnischen Abbildung von Matrizen wurde auch in weiteren Bereichen (s. Punkt "4.2.3 Speicherverwaltung") verwendet und ist somit für sämtliche Matrizen bzw. mehrdimensionale Felder gültig.

Abbildung 10 zeigt eine zweidimensionale Matrix, die als ein Feld von Feldern abgebildet wird. Dieses Speicherkonzept ist in Java implementiert. Bei C/C++ hingegen erfolgt standardmäßig die Abbildung auf einen eindimensionalen Bereich im Speicher. Das aufgeführte Matrix-Konzept wurde aufgrund der besseren Flexibilität in den C/C++-Anwendungen verwendet, sodass es auch dem Java-Konzept entspricht.

Die Matrix-Multiplikation wurde für Problemgrößen im Bereich von 100 bis 1500 Elemente pro Dimension durchgeführt. Hierbei wurde das erste Element der Ausgangsmatrix mit dem Wert '1' beleget. Das nächste Element wurde jeweils durch das Inkrement des Vorgängers gebildet. Diese Belegung wurde entsprechend der Anzahl an Elementen und Dimensionen durchgeführt. Des Weiteren wurde für jede Problemgröße, die Multiplikation 100-mal wiederholt um aussagekräftige Daten zu erhalten. Die Problemgrößen wurden kontinuierlich um ein Intervall von 50 Elementen pro Dimension erhöht. Hierdurch nahm die Komplexität der Matrix-Multiplikation zu und es konnten besser vergleichbare Ergebnisse für die Laufzeitvergleiche erzielt werden.

Der folgende Programmablaufplan soll einen Überblick über die Funktionsweise des Programms zur Berechnung einer 2D-Matrix-Multiplikation (von zwei gleichen Matrizen) für C/C++ und Java geben. Weitere Operationen hinsichtlich der Laufzeitbestimmung sind nicht aufgeführt, da hier nur die grundlegende Funktionsweise dargestellt werden soll.

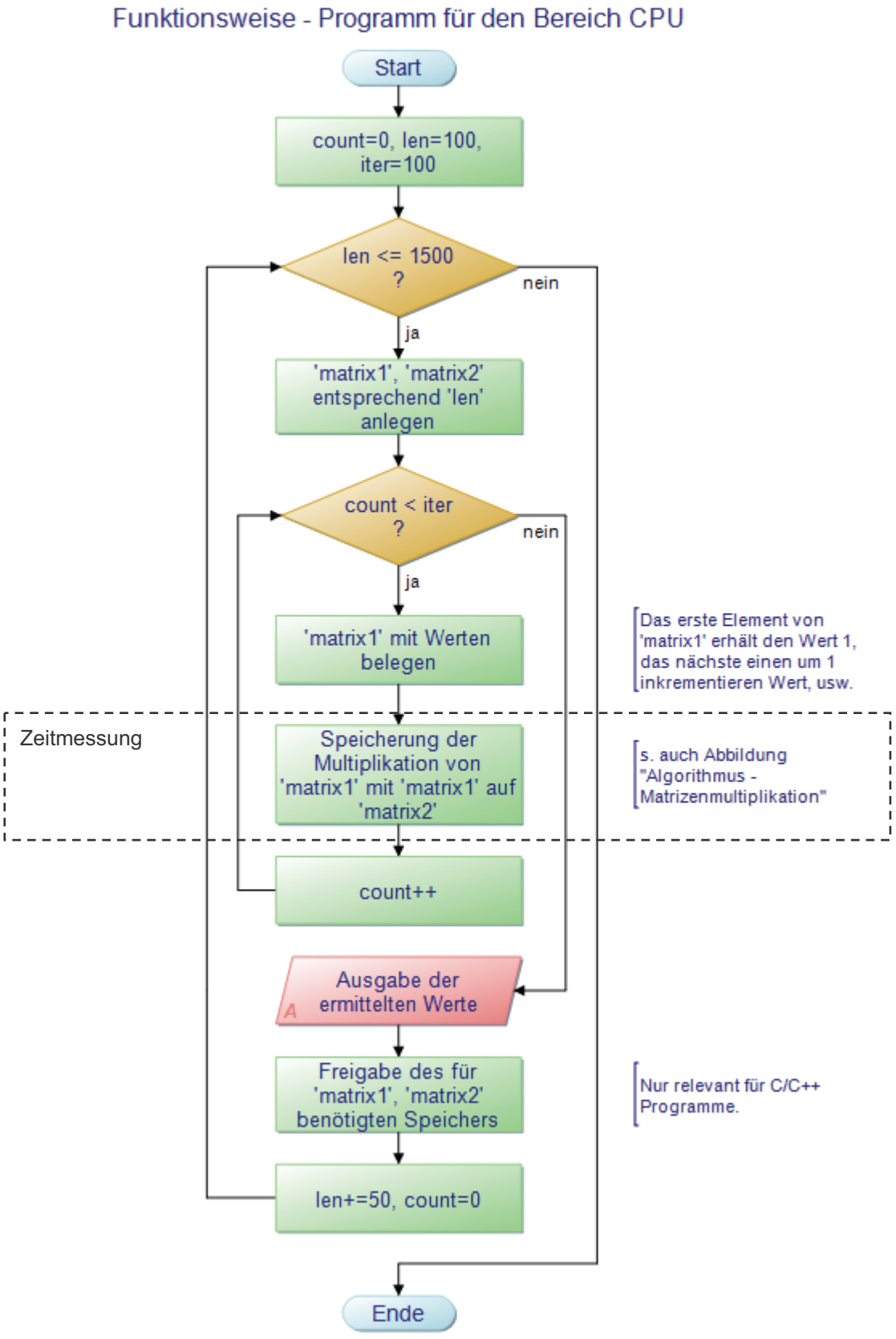


Abbildung 11: Programmablaufplan - Funktionsweise des Programms für den CPU-Bereich

4.2.3 Speicherverwaltung

Die Funktionsweise für die Programme zur dynamischen Allokation von Hauptspeicher besteht darin, zwei Ausgangslisten A, B anzulegen und diese entsprechend ihrer Größe mit Werten zu belegen. Dabei wird die Größe einer Liste nach der Anzahl enthaltener Elemente sowie die Größe der Elemente selbst bestimmt. Das Ergebnis soll eine Verkettung der beiden Listen beinhalten (Liste C), dabei sollen alle Werte der Liste A mit denen von Liste B verknüpft werden. Der laufzeitrelevante Abschnitt umfasst das Ausfassen, das für die Verknüpfung der beiden Listen benötigten Speichers.

Der im Weiteren aufgeführte Screenshot (Abbildung 12) stellt eine Verkettung von zwei Ausgangslisten mit jeweils 5 Elementen und einer Größe von 3 Zeichen pro Element dar.

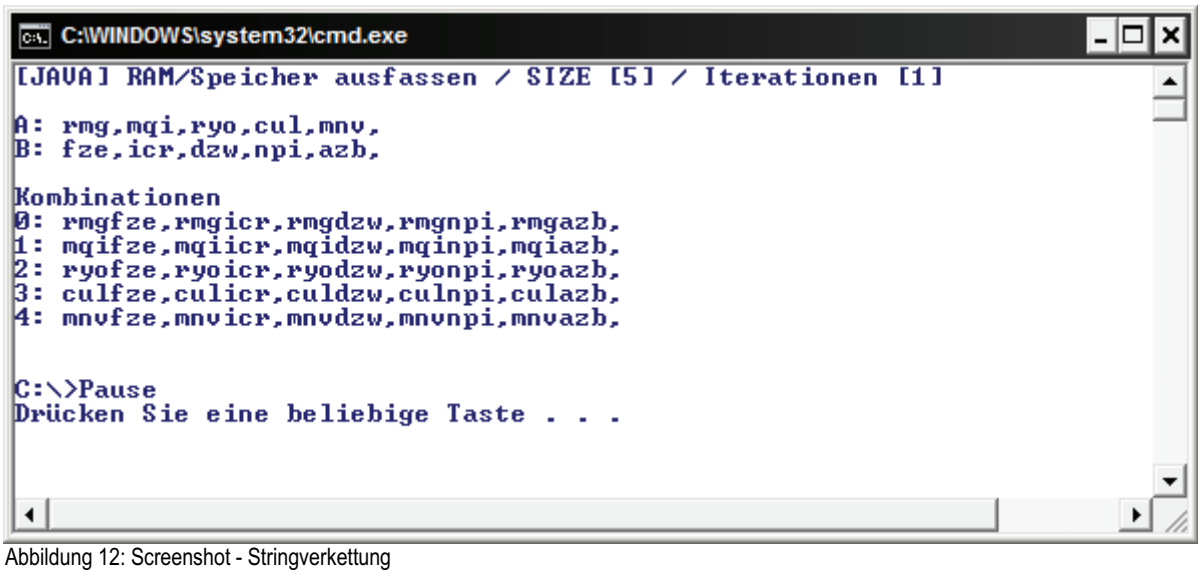


Abbildung 12: Screenshot - Stringverkettung

Für die Entwicklung von Applikationen zum Zweck der dynamischen Allokation stellten die Programmiersprachen C und C++ Funktionen bereit, bei Java hingegen übernahm der sogenannte **»Garbage Collector«** die dynamische Speicherverwaltung. So war es für C/C++-Programme nötig, die Größe des benötigten Speichers zu ermitteln, Speicher anzufordern und zu zuweisen sowie ihn wieder freizugeben. Der damit verbundene Aufwand war grundsätzlich mit laufzeitkritischen Problemen verbunden, da bei nicht korrekt freigegebenen Speicher der Prozess vom System beendet wurde oder gar ein Systemabsturz hervorgerufen werden konnte.

Ein weiteres Problem das bei der Umsetzung auftauchte, bestand darin, dass das Speicher allozieren in Abhängigkeit von der Größe nur sehr wenig Zeit in Anspruch nahm. Die hierfür verwendeten Methoden waren mit einer Genauigkeit im Millisekundenbereich nicht genau genug, um nur diesen Vorgang zu messen. Der folgende Quelltext (bezogen auf Java) war der zu Beginn der Einwicklung angedachte Vorgang um die Laufzeit zu messen.

```
for (i=0; i<size; i++) {
  for (j=0; j<size; j++) {

    // Beginn der Zeitmessung
    beginTime = System.currentTimeMillis();

    // An dieser Stelle erfolgt die Allokation von Speicher
    C[i][j] = new char[sLenMax * 2];

    // Ende der Zeitmessung
    endTime = System.currentTimeMillis();
    diffTime += endTime - beginTime;

                          ...
    /* String-Verkettung */
  }
}
```

Listing 5: Laufzeitmessung 1 - Speicher Allokation

Der Quellcode (Listing 5) zeigt, dass für jede Allokation eine Zeitmessung durchgeführt wird und die Laufzeiten für den gesamten Vorgang aufaddiert werden. Jedoch wird bei dieser Methode nur ein sehr geringer Zeitabschnitt gemessen. Dieser ist im Einzelnen mit Verlusten bzgl. der Genauigkeit verbunden. Aufgrund der schlechten Messbarkeit war es nötig, eine andere Vorgehensweise für die Laufzeitmessung zu implementieren.

Die zweite Methode (Listing 6) realisiert die Laufzeitbestimmung aus einer Simulation und der tatsächlichen Stringverkettung. Dabei werden zwei Laufzeitmessungen durchgeführt und aus der Differenz lässt sich die Zeit für die Allokation von Speicher ermitteln.

Im folgenden Codeabschnitt ist der Vorgang für die zweite Vorgehensweise vereinfacht dargestellt.

```
// 1. Zeitmessung für die Simulation
simBeginTime = System.currentTimeMillis();

      /* String-Verkettung und Simulation einer Zuweisung */

// Ende der 1. Zeitmessung für die Simulation
simEndTime = System.currentTimeMillis();
simDiffTime = simEndTime-simBeginTime;

                          ...
// 2. Zeitmessung
realBeginTime = System.currentTimeMillis();

      /* String-Verkettung und neuen Speicher ausfassen */

// Ende der 2. Zeitmessung
realEndTime = System.currentTimeMillis();
realDiffTime = realEndTime-realBeginTime;

// Zeit die für die Allokation von Speicher gebraucht wurde
allocTime = realDiffTime-simDiffTime;
```

Listing 6: Laufzeitmessung 2 - Speicher Allokation

Aufgrund der besseren Messbarkeit wurde die in Listing 6 aufgeführte Methode zur Laufzeitbestimmung verwendet und in allen Programmen äquivalent implementiert.

Der anschließende Programmablaufplan (Abbildung 13) soll die Vorgehensweisen der erstellten Programme in der Gesamtheit näher verdeutlichen. Aufgrund des Umfangs der Applikationen werden nur die wesentlichen Vorgänge aufgeführt.

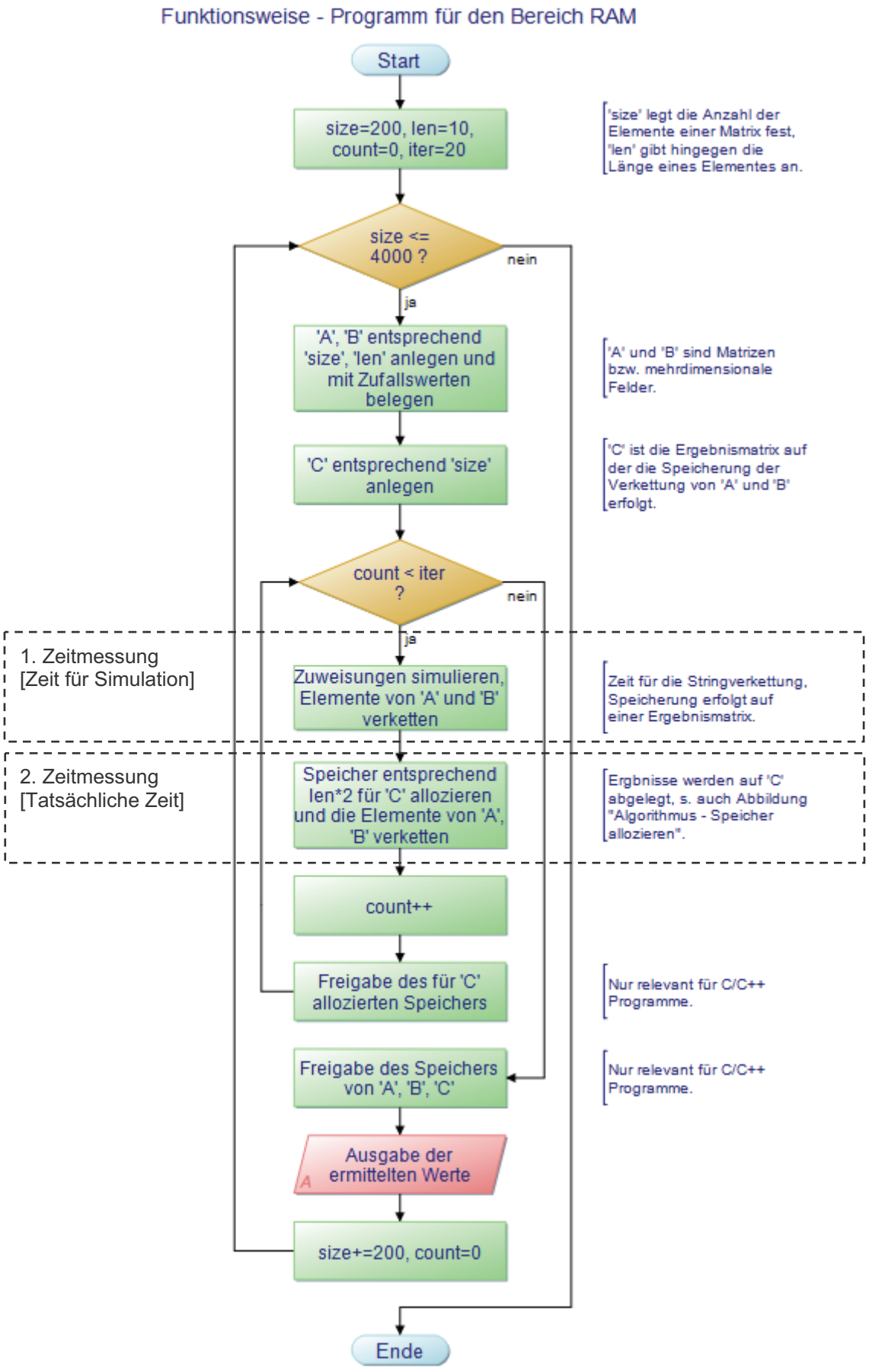


Abbildung 13: Programmablaufplan - Funktionsweise des Programms für den RAM-Bereich

4.2.4 I/O-Operationen bzgl. Dateiarbeit

Der Fokus zur Umsetzung lag bei diesem Bereich auf das zeichenweise Lesen und Schreiben von Daten. Hierbei stellten die jeweiligen Programmiersprachen Funktionen für die Realisierung bereit. Für C waren jedoch zwei äquivalente Funktionen (zum Lesen und Schreiben von Zeichen) vorhanden, weshalb auch beide separat in Programmen implementiert und dafür Laufzeitmessungen durchgeführt wurden. Die folgende Tabelle zeigt die verwendeten Methoden sowie eine weitere Erläuterung.

Funktionen zum zeichenweisen Lesen und Schreiben (C/C++/Java)		
Sprache	**Funktionen**	**Beschreibung**
C	`fgetc, fputc`	- die Funktion fgetc() liefert als Rückgabewert ein Zeichen aus einem Stream, fputc() hingegen ermöglicht das Schreiben eines Zeichens in einem Stream - die Streams beziehen sich in diesem Beispiel auf eine Ziel- und Quelldatei - beide Funktionen müssen nach der C89/99-Norm als Funktionen implementiert sein, was bedeutet das Argumente, die Adresse des Aufrufs auf dem Stack abgelegt werden und bei Bedarf wiedergeholt werden muss - in Abhängigkeit von den Funktionsaufrufen kann dies einiges an Zeit kosten
C	`getc, putc`	- getc() und putc() haben wie die obigen Funktionen die Aufgabe ein Zeichen aus einem Stream zu lesen bzw. in einem zu schreiben - jedoch sind diese Funktionen als Makros definiert, wodurch Sprünge entfallen und Werte von Parametern direkt eingesetzt werden können - der Einsatz von Makros kann kürzere Laufzeiten sowie eine Zunahme des Programmcodes zur Folge haben
C++	`get, put`	- für C++ liest die Funktion get() ein einzelnes Zeichen ein und mit put() lässt sich wiederum ein Zeichen schreiben
Java	`read, write`	- wie C und C++ arbeitet auch Java mit Streams, die äquivalenten Funktionen zum zeichenweisen lesen und schreiben sind hierfür read() und write()

Tabelle 7: Funktionen zum zeichenweisen Lesen und Schreiben (C/C++/Java) [nach Quellen: HER 04, WIN 01]

Die Programme wurden so umgesetzt, dass verschiedene Quelldateien und entsprechend viele Zieldateinamen als Parameter übergeben werden konnten. Daraufhin erfolgte die Abarbeitung mit der ersten angegebenen Datei. Die Zieldatei wurde dabei nach jedem Kopiervorgang wieder gelöscht. Die Anzahl der Kopiervorgänge je Datei wurde auf 20 Iterationen festgelegt.

Für die Problemgrößen wurden drei Dateien verwendet (im Folgenden auch als kleine, mittlere sowie große Datei bezeichnet).

- Datei (src1.7z) → Dateigröße = 150 MB
- Datei (src2.exe) → Dateigröße = 352 MB
- Datei (src3.mp4) → Dateigröße = 700 MB

Die Vorgehensweise der erstellten Programme soll durch den folgenden Programmablaufplan (Abbildung 14) näher verdeutlich werden.

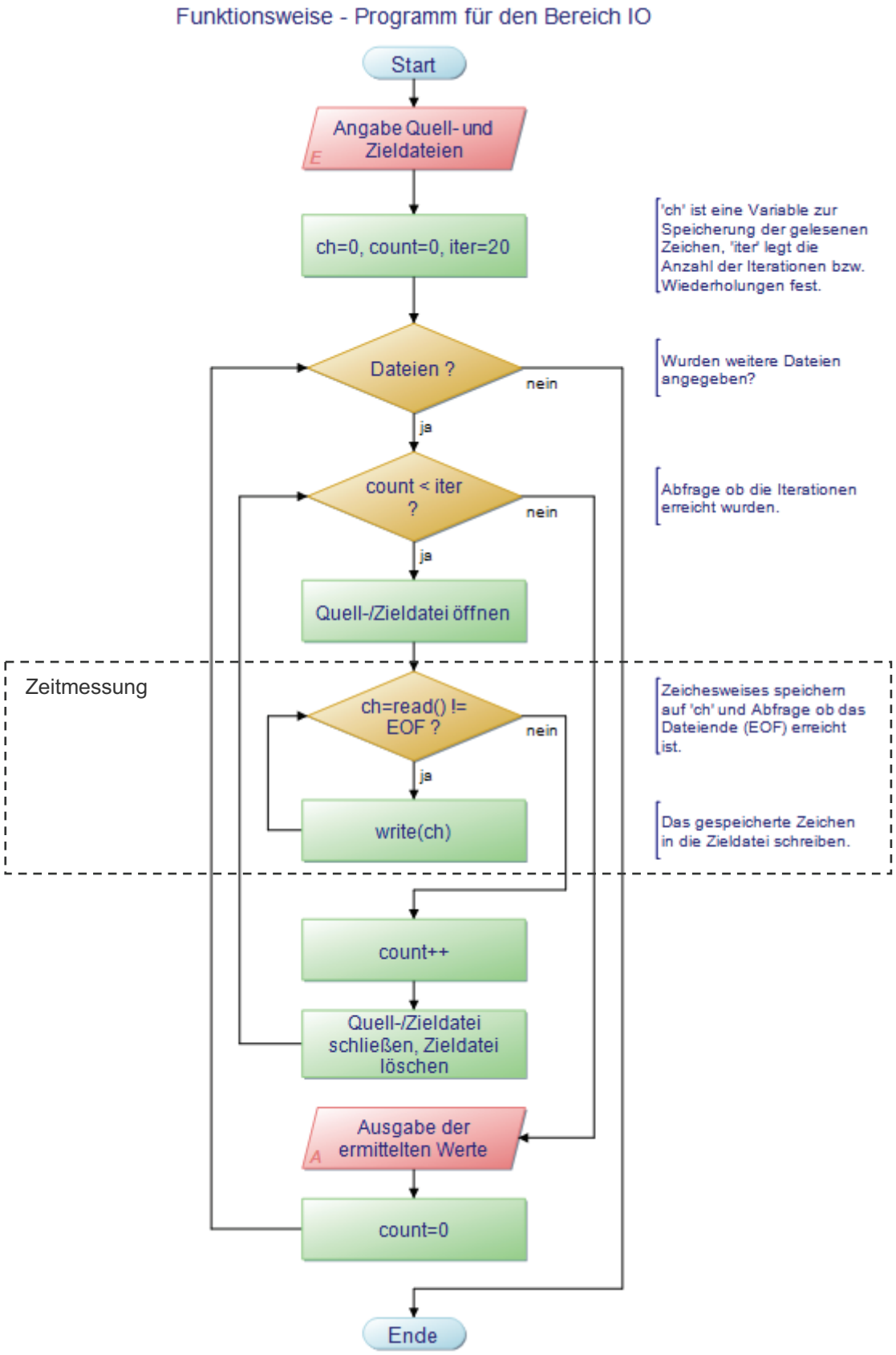


Abbildung 14: Programmablaufplan - Funktionsweise des Programms für den IO-Bereich

4.3 Sprachübersetzung und Optimierung

Für die Erstellung bzw. Entwicklung von C/C++-Anwendungen kam der »Dev-C++ Editor« in der Version 4.9.9.2 (Version 2, Juni 1991) zum Einsatz. Für Java-Anwendungen wurde der »Java-Editor« (Version 9.14k vom 14.03.2010) verwendet.

4.3.1 Compiler / Sprachübersetzung

Nachdem die Implementierung der Algorithmen und die Entwicklung der Programme abgeschlossen wurden, konnte die Sprachübersetzung erfolgen. Eine Übersetzung mithilfe der verwendeten Entwicklungsumgebungen wurde aufgrund von standardmäßig eingestellten Optimierungen und der mangelhaften Kontrolle des verwendeten Compilers nicht in Betracht gezogen.

Die Sprachübersetzung für C/C++-Quellprogramme wurde auf dem Windows-System durch den »MingW32-Compiler« (gcc version 4.4.0) und auf dem Linux-System (Ubuntu 4.4.1-4ubuntu9) durch den gcc-Compiler (version 4.4.1) realisiert. Für Java-Anwendungen wurde auf beiden Systemen der javac-Compiler in der Version 1.6.0_19 und für Windows zusätzlich der Jikes-Compiler (Version 1.22 - 3. Oktober 2004) eingesetzt. Dabei waren für beide Systeme der javac-Compiler sowie die benötigte Laufzeitumgebung in der Java(TM) SE Runtime Environment (build 1.6.0_19-b04) enthalten.

Die Quellprogramme wurden hierbei auf Kommandozeilenebene mithilfe des entsprechenden Compilers und Argumenten übersetzt. Der Aufruf ist dabei grundsätzlich wie folgt definiert:

```
[Compilerpfad]Compiler [-Optionen] Quellprogramm
```

Der folgende Screenshot zeigt auf der Kommandozeile für den definierten Aufruf ein Beispiel:

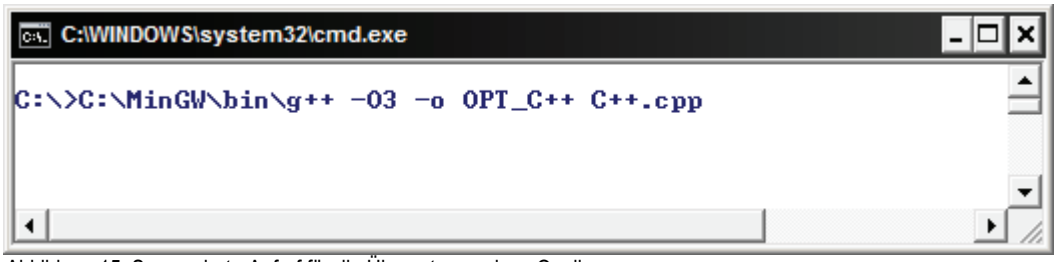


Abbildung 15: Screenshot - Aufruf für die Übersetzung eines Quellprogramms

Zur Erläuterung: `C:\MinGW\bin\g++` ist in diesem Beispiel der Aufruf des Compilers mit Pfadangabe. `-O3` definiert einen Optimierungsschalter des gcc-Compilers. `-o` und `OPT_C++` sind weitere Optionen, die signalisieren, dass eine Ziel-/Ausführungsdatei namens `OPT_C++` erzeugt werden soll. `C++.cpp` ist dabei das zu übersetzende Quellprogramm. Jedes Quellprogramm wurde dabei zweimal übersetzt, jeweils mit und ohne Angabe eines Optimierungsschalters für den entsprechenden Compiler.

Eine Auflistung von weiteren Informationen, Beschreibungen bzw. Erläuterungen hinsichtlich Optimierungsmöglichkeiten sowie Optionen/Schalter der verwendeten Compiler erfolgt im weiteren Verlauf.

Durch den Einsatz der Stapelverarbeitung bzw. Batchverarbeitung konnte der Übersetzungsprozess für die verschiedenen Quelldateien automatisiert werden. Eine ausführbare Datei enthält dabei mehrere Befehle, die sequenziell auf der Kommandozeile/Shell abgearbeitet werden (s. folgendes Beispiel, Listing 7).

```
/* Zu Beginn werden die Quelldateien unoptimiert kompiliert. */

gcc -o UNO_C C.c                 // gcc → Aufruf des C-Compilers
g++ -o UNO_C++ C++.cpp           // g++ → Skript das den C-Compiler mit
                                    nötigen Optionen aufruft
javac UNO_JAVAC.java             // Aufruf des ersten Java-Compilers
jikes UNO_JIKES.java             // zweiter Java-Compiler

/* Im zweiten Schritt erfolgt die Übersetzung mit Angabe eines vom
Compiler abhängigen Optimierungsschalters. */

gcc -O3 -o OPT_C C.c
                                  ...
```

Listing 7: Batch-File für die Sprachübersetzung [nach Quelle: LMU 01]

4.3.2 Möglichkeiten der Optimierung

»Optimierung« bezeichnet nach [PR 97]: "Den Versuch, das Laufzeit- oder Speicherverhalten eines Programms durch Codetransformationen zu verbessern." Allgemein ist zu bemerken, dass eigene Optimierungen essenziell für Performancesteigerungen sind. Die folgenden Punkte nach [LM 05, PR 97] sollen hierbei einige Beispiele aufzeigen.

1. Entfernung toten bzw. schleifeninvarianten Codes (nicht benötigte Direktiven/Anweisungen) / gemeinsamer Teilausdrücke
2. Befreien von Reservierungen für (lokale Variablen) bzw. die Erzeugung von temporären Objekten
3. Prüfen von zeitintensiven Algorithmen, Methoden bzw. Funktionen oder Ähnlichem (z. B. durch Laufzeitmessungen)
4. Prüfung von unnötig oft wiederholten Operationen
5. Befehlsanordnungen / Registerzuordnungen

Bei dem Einsatz von Abkürzungen für Variablen-, Methoden-, Funktionennamen oder Ähnlichem, wird kein gewinnbringender Effekt mit Bezug auf die Performance erreicht. Die meisten gängigen Compiler führen ohnehin vor diesem Hintergrund eine Optimierung durch. Weiterhin stellen die verwendeten Compiler einige Optionen (z. B. durch die Übergabe von Schaltern) für die Optimierung zur Verfügung. In den folgenden Punkten sollen wesentliche Schalter benannt und deren Funktionen erläutert werden.

[Punkt 4.3.2 nach Quellen: LM 05, PR 97, STE 01]

4.3.3 Optimierungsoptionen für Compiler

Ohne Angabe eines Schalters bzw. Option zum Zweck der Optimierung, versucht der gcc-Compiler die Zeit für die Kompilierung zu minimieren. Der Aufbau der Schalter ist dabei wie folgt definiert: »`-O[Level]`«. `-O` steht für *optimize*, `[Level]` ermöglicht die Angabe eines weiteren Parameters, um den Grad der Optimierung zu beeinflussen (s. Tabelle 8).

Optimierungsschalter für C/C++-Compiler (gcc)	
Option	**Beschreibung**
-O0	- es wird keine Optimierung vorgenommen, lediglich die Kompilierungszeit wird verringert - dieser Schalter entspricht der Standardeinstellung (bzw. keine Angabe einer Optimierungsoption)
-O/-O1	- "Optimierungsstufe 1", der Compiler versucht die Codegröße und Ausführungszeit zu minimieren, sodass sich die Zeit der Kompilierung nur geringfügig erhöht - entsprechend der Auswirkungen dieser Optimierung, ist (erheblich) mehr Speicher erforderlich
-O2	- "Optimierungsstufe 2", »gcc« führt fast alle unterstützten Optimierungen aus, die in keinen Kompromiss zu Geschwindigkeit und Speicherplatz stehen - im Vergleich zu "-O1" wird die Übersetzungszeit und die Leistung des erzeugten Codes weiter erhöht
-O3	- die 3. und höchste Stufe der Optimierung führt alle durch "-O2" angegebenen Optionen aus sowie weiterer - dieser Schalter bietet die effektivste Optimierung hinsichtlich der Performance von Programmen, dabei nimmt die Größe des Codes ggf. weiter zu
-Os	- dieser Schalter hat die Aufgabe die Codegröße zu reduzieren - es werden alle Optimierungen von "-O2", die keine Zunahme der Codegröße zur Folge haben, ausgeführt - weiterhin erfolgt eine Ausrichtung auf Optimierungen, die eine weitere Reduzierung der Codegröße mit sich bringen

Tabelle 8: Optimierungsschalter für C/C++-Compiler (gcc) [nach Quelle: GCC 02]

Die aufgeführten Optimierungsoptionen führen jeweils verschiedene Aufgaben durch, so werden für jeden Schalter eine Reihe von sogenannten **Flags** gesetzt. Beispielsweise besteht die Funktion von »`-finline-functions`« darin, alle "einfachen" Funktionen dort zu integrieren, wo ein entsprechender Aufruf der Funktionen erfolgt. Dabei entscheidet der Compiler heuristisch, welche Funktionen auf diese Weise implementiert werden. Der damit verbundene Vorteil ist die Einsparung von Laufzeit, die für Sprünge in und aus Funktionen benötigt wird. Dabei wird der Code welcher eine Funktion beinhaltet, an der Stelle des Aufrufs integriert. Im Gegensatz dazu steht der Nachteil, dass in Abhängigkeit der Anzahl von verwendeten Funktionen, die Speichergröße entsprechend zunimmt. »`-finline-functions`« wird u. a. bei der Angabe der "`-O3`"-Option ausgeführt. Eine detaillierte Auflistung aller Flags sowie Beschreibungen über deren Auswirkungen, der dargestellten Schalter bzw. Optionen kann der Quelle [GCC 02] entnommen werden.

Für die Auswahl eines Optimierungsschalters waren Funktionen hinsichtlich Geschwindigkeit von größerer Relevanz, als z. B. Optionen die eine Reduzierung der Codegröße zur Aufgabe haben. Um eine bestmögliche Optimierung zu erreichen, wurde für alle erstellten C- und C++-Anwendungen eine zusätzliche Sprachübersetzung mithilfe des "`-O3`"-Schalters vorgenommen. Hierdurch soll aufgezeigt werden, inwiefern der Compiler einen Einfluss auf die Laufzeit, im Gegensatz zu einer nicht optimierten Anwendung bewirken kann.

Die nachstehende Tabelle zeigt einige Optimierungsmöglichkeiten für den javac- bzw. Jikes-Compiler.

Optimierungsmöglichkeiten für Java-Compiler (javac, Jikes)	
Option	**Beschreibung**
-O	- bestmögliche Optimierung der Klassendateien - static-, final- und public-Methoden werden als inline-Methoden behandelt, somit wird die Ausführung des generierten Codes beschleunigt - Nachteil: der erzeugte Bytecode wird geringfügig größer
-g:none	- es werden keine Debuginformationen aufgenommen - relevant für die Größe des Byte-Code

Tabelle 9: Optimierungsmöglichkeiten für Java-Compiler (javac, Jikes) [nach Quellen: LM 05, STE 01]

Da die Java-Compiler nur die "-O" Option zur Optimierung der Ausführungsgeschwindigkeit anbieten, wurde für die zusätzliche Kompilierung der Java-Applikationen ausschließlich diese Option verwendet.

4.4 Ausführung

Zum Zeitpunkt der Laufzeitmessungen waren einige Prozesse deaktiviert. Hierzu zählen insbesondere sicherheitsrelevante Applikationen wie z. B. Firewall, Viren-Scanner, Update-Funktionen aber auch Indizierungsdienste oder Ähnliches. Neben diesen inaktiven Prozessen erfolgte während der Laufzeit keine Benutzerinteraktion, wodurch Einwirkungen auf die Laufzeiten hinsichtlich dieser Aspekte ausgeschlossen werden können.

Die Ausführung der erstellten Programme und Speicherung der ermittelten Laufzeiten und weiteren berechneten Werten wurde mithilfe eines Batch-Files (Listing 8) auf dem Testsystem realisiert. Diese ausführbare Datei enthält dabei u. a. folgende Anweisungen:

```
/* Zu Beginn werden einige Hilfsvariablen festgelegt. */

set AIM=CPU         // Angabe Bereich
set PARAM=src1.7z   // ggf. weitere Parameter (z. B. relevant für I/O)

/* Die folgenden Anweisungen stehen für den Aufruf der nicht optimierten
Programme. */

UNO_C.exe > "C - %AIM% - Ergebnisse [UNO].txt"
UNO_C++.exe > "C++ - %AIM% - Ergebnisse [UNO].txt"
java UNO_JAVAC > "Java - %AIM% - Ergebnisse [UNO].txt"
                          ...

/* Als Nächstes werden die optimierten Programme aufgerufen. */

OPT_C.exe > "C - %AIM% - Ergebnisse [OPT].txt"
                          ...
```

Listing 8: Batch-File zur Ausführung

Wie dargestellt, werden zu Beginn einige Hilfsvariablen festgelegt. Diese ermöglichen die Angabe des Bereichs, für den die Applikationen ausgeführt werden sollen sowie der Übergabe von weiteren optionalen Parametern. Im Anschluss werden die Programme aufgerufen, die ohne Angabe eines Optimierungsschalters übersetzt wurden. Danach erfolgt der Aufruf der optimierten Programme, alle Programme werden dabei sequenziell ausgeführt und grundsätzlich nach der folgenden Reihenfolge:

$$\underbrace{\text{C} \rightarrow \text{C++} \rightarrow \text{Java}}_{\text{unoptimiert}} \rightarrow \underbrace{\text{C} \rightarrow \text{C++} \rightarrow \text{Java}}_{\text{optimiert}}$$

Java (für Windows JAVAC → JIKES)

Für die Auswertung erzeugten die verschiedenen Programme gleich strukturierte Ausgaben, indem sämtliche Ergebnisse der Laufzeitmessungen und berechneten Werte (wie z. B. Mittelwert) enthalten waren. Die Programmausgaben wurden mithilfe des Umlenkoperators ">" in die im Batch-File angegebenen Dateien geschrieben.

4.5 Auswertung

Da die erstellten Dateien in Abhängigkeit von den Problemgrößen und Iterationen relativ groß bzw. unübersichtlich wurden und die Ergebnisse nicht ohne hohen Zeitaufwand grafisch darzustellen sind, war es nötig ein Programm zum Auslesen der relevanten Daten zu erstellen.

Das Auswerte-Programm wurde so konzipiert, dass ein Bereich (CPU, RAM, IO) und eine Zieldatei beim Programmaufruf angegeben werden mussten. Die im Weiteren aufgeführten Funktionen wurden dabei abgearbeitet:

- Auslesen von Problemgrößen, Minimal-, Maximal-, Median- und Mittelwert
- Ersetzen von Punkt "`.`" durch Komma "`,`" (Zahlenformat benötigt für »MS Excel«)
- Zieldatei (*.xls) entsprechend dem Namen der Quelldatei anlegen
- ausgelesene Werte in die Zieldatei schreiben, der Aufbau der Zieldatei ist der einer UNL-Datei (Dateiformat ähnlich einer CSV-Datei, die Werte sind hierbei durch ein Semikolon getrennt)

Um die aufgeführten Vorgänge für alle Ergebnisdateien durchzuführen, wurde mittels Batch-File das Auswerte-Programm entsprechend der Bereiche und zu lesenden Dateien mehrfach hintereinander aufgerufen (s. folgenden Ausschnitt "Listing 9").

```
/* Batch-File für die Auswertung */

set AIM=CPU         // Angabe Bereich

/* »programm« ist im Folgenden der Name des Auswerte-Programms */

java programm %AIM% "C - %AIM% - Ergebnisse [OPT].txt"
java programm %AIM% "C++ - %AIM% - Ergebnisse [UNO].txt"
java programm %AIM% "(JAVAC) Java - %AIM% - Ergebnisse [OPT].txt"
java programm %AIM% "(JIKES) Java - %AIM% - Ergebnisse [UNO].txt"
                                  ...
```

Listing 9: Batch-File für die Auswertung

Die durch das Auswerte-Programm erzeugte XLS-Datei beinhaltete nur die relevanten Daten für die Laufzeitvergleiche. Diese Datei konnte anschließend, z. B. mit »MS Excel« geöffnet und die Ergebnisse weiterverarbeitet sowie entsprechend grafisch dargestellt werden.

Die Vorgehensweise, die durch das Auswerte-Programm vorgenommen wird, kann dabei wie folgt abgebildet werden:

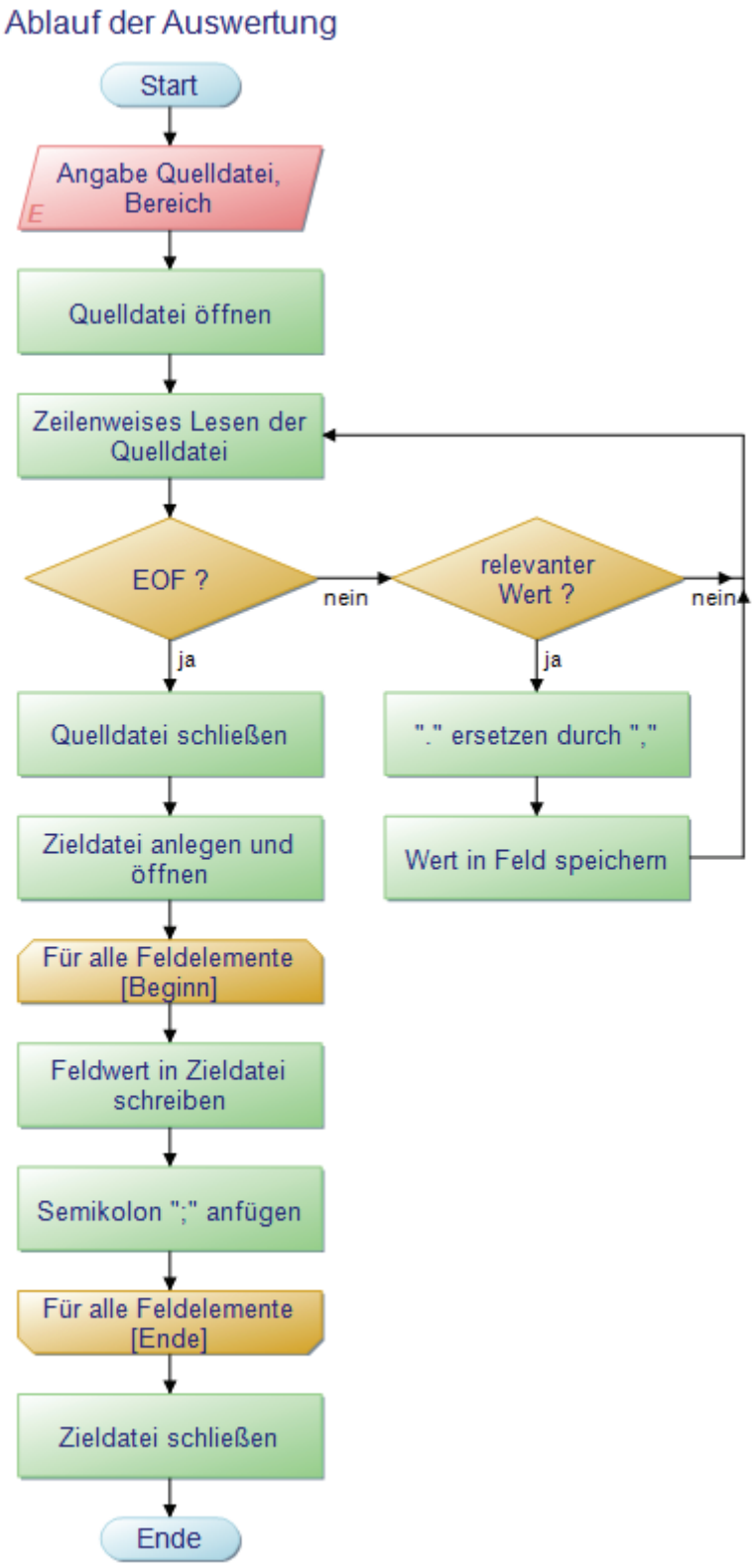


Abbildung 16: Programmablaufplan - Ablauf der Auswertung

Kapitel 5

5 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Laufzeitmessungen zwischen den unterschiedlichen Problembereichen und -größen dargestellt und bewertet. Dabei werden Laufzeitvergleiche von Applikationen der Bereiche CPU, RAM, I/O jeweils für Windows- und Linux-Betriebssystem aufgeführt. Für Vergleiche zwischen den Betriebssystemen ist zu beachten, dass die Systeme eine unterschiedliche Software-Konfiguration aufweisen und für die Sprachübersetzung von C/C++-Quellprogrammen verschiedene Compiler verwendet wurden (s. auch Kapitel "4 Durchführung"). Eine Analyse der Ursachen ist unter Punkt 6.2 zu finden.

Für die Darstellung in Diagrammen wurden die berechneten Mittelwerte verwendet. In weiteren Diagrammen sind darüber hinaus Werteabweichungen vom Mittelwert (Minimal-, Maximalwert und Medianwert) dargestellt. In Diagrammen dargestellte Symbole bzw. Markierungen (wie z. B. Kreise, Dreiecke, Kreuze, usw.) stellen Messungen bzw. Messpunkte für entsprechende Problemgrößen dar. Die Verbindungen zwischen diesen Markierungen wurden nicht gemessen und dienen lediglich der besseren Visualisierung.

Neben den Diagrammen sind auch Tabellen mit den entsprechenden Laufzeiten aufgeführt. Hierdurch können die in den Diagrammen abgebildeten Datenreihen detaillierter nachvollzogen werden. Aufgrund des Umfangs der Messungen und die Anzahl ermittelter Werte erfolgt in einigen Bereichen eine auszugsweise Darstellung von Laufzeiten. Ein weiterer Grund der nicht vollständigen Wertedarstellung besteht in der geringen Aussagekraft einiger Werte. Beispielsweise liegen verschiedene Laufzeiten bei geringen Problemgrößen sehr dicht beieinander bzw. konnten mit den implementierten Messmethoden nicht ausreichend genau gemessen werden. Weiterhin wurden für Darstellungszwecke mehrere Werte auf entsprechende Nachkommastellen gerundet (s. Fußnoten).

Für nicht aufgeführte Mittelwerte sei auf den Anhang sowie für auszugsweise dargestellte Werteabweichungen auf die beiliegende CD-ROM verwiesen.

Soweit nicht anders gekennzeichnet ist die **Einheit** der in den folgenden Diagrammen und Tabellen aufgeführten Laufzeiten bzw. Werte, in **Sekunden (s)** zu verstehen.

5.1 CPU-Auslastung / Rechenlast

Die im Folgenden aufgeführten Diagramme zeigen die Ergebnisse von Laufzeitmessungen der Programme zur Berechnung einer 2D-Matrix-Multiplikation zwischen C/C++ und Java. Die Applikationen wurden jeweils auf zwei unterschiedlichen Betriebssystemen ausgeführt (s. "5.1.1 Messung 1 - Windows OS" und "5.1.2 Messung 2 - Linux OS").

Angaben zur Messung:

- Problemgrößen: 100 bis 1500 E/Dim (je Matrix)
- Elementgröße: von 1 beginnend, das nächste Element ist jeweils das Inkrement um 1
- Intervall: 50 E/Dim
- Wiederholungen: 100 je Problemgröße

5.1.1 Messung 1 - Windows OS

Das nachfolgende Liniendiagramm zeigt einen Auszug von berechneten Mittelwerten der C/C++/Java-Programme (optimiert und unoptimiert), die auf dem Windows Betriebssystem ermittelt wurden.

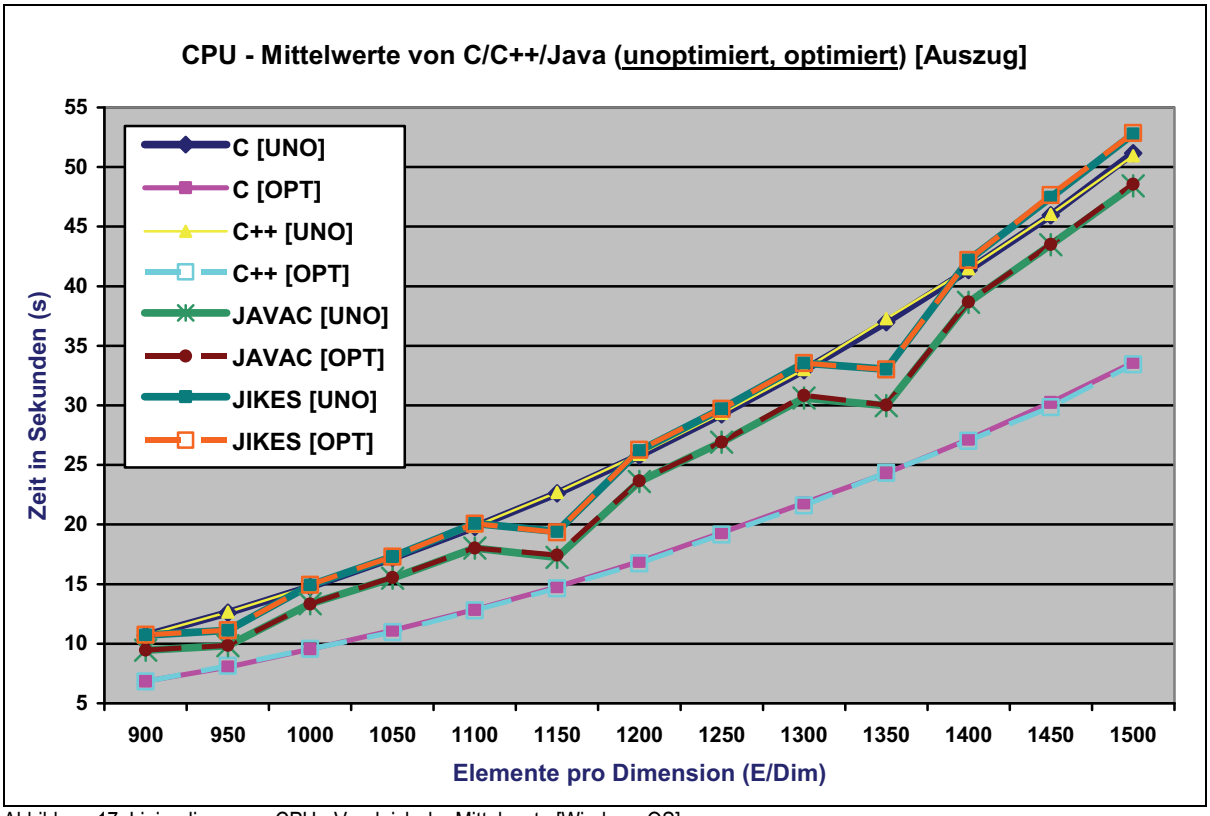


Abbildung 17: Liniendiagramm CPU - Vergleich der Mittelwerte [Windows OS]

Der Grafik ist zu entnehmen, dass die unoptimierten C/C++-Programme äquivalente Laufzeiten aufweisen. Bei den optimierten Varianten zeigt die C++-Anwendung bei steigender Komplexität marginal kürzere Laufzeiten und im gesamten Vergleich die beste Performance (s. auch "Tabelle 10: Vergleich CPU - Mittelwerte [Windows OS]").

Bei den Java-Anwendungen (JAVAC, JIKES) ist zwischen optimierter und unoptimierter Variante kein signifikanter Unterschied feststellbar. Jedoch sind bei den "Java-Datenreihen" temporäre Sprünge zu beobachten. Beispielsweise wird für die Multiplikation der Matrizen mit 1150 E/Dim, ca. eine halbe Sekunde weniger Zeit in Anspruch genommen als bei der Multiplikation von Matrizen mit 1100 E/Dim. Bezogen auf dieses Fallbeispiel kann dieser Effekt, gemessen ab einer bestimmten Komplexität, jeweils in einem Intervall von 200 E/Dim beobachtet werden.

Tabelle CPU (Auszug) - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden]								
E/Dim	C [UNO / OPT]		C++ [UNO / OPT]		JAVAC [UNO / OPT]		JIKES [UNO / OPT]	
1000	14,779	9,531	14,833	9,571	13,348	13,332	14,914	14,931
1050	17,182	11,118	17,211	10,968	15,510	15,562	17,298	17,289
1100	19,800	12,857	19,856	12,800	18,029	18,036	20,083	20,055
1150	22,605	14,766	22,707	14,631	17,234	17,442	19,423	19,341
1200	25,779	16,884	25,884	16,731	23,621	23,652	26,179	26,277
1250	29,238	19,306	29,318	19,165	26,923	26,912	29,704	29,711
1300	32,947	21,799	33,006	21,584	30,618	30,841	33,539	33,538
1350	36,994	24,312	37,274	24,326	29,984	30,037	33,042	33,011
1400	41,370	27,159	41,464	27,023	38,644	38,683	42,191	42,195
1450	45,955	30,229	46,037	29,852	43,485	43,497	47,456	47,645
1500	51,169	33,594	50,941	33,424	48,410	48,563	52,775	52,855

Tabelle 10: Vergleich CPU - Mittelwerte [Windows OS]

Die folgenden Liniendiagramme zeigen Werteabweichungen (in Sekunden) für die nicht optimierten (Abbildung 18) und darauf für die optimierten Anwendungen (Abbildung 19). Hierbei wurde jeweils der Minimal-, Maximal- und Medianwert durch den Mittelwert dividiert. Um die Abweichungen besser darzustellen, wurde der Mittelwert auf 0 normiert, als Differenzen sind hiervon Datenreihen für Minimal-, Maximal- und Medianwerte aufgeführt. Diese Darstellungsweise wird im weiteren Verlauf für alle Diagramme der Bereiche CPU bzw. RAM verwendet, in denen Werteabweichungen abgebildet werden. Ergänzend werden neben den Diagrammen Tabellen aufgeführt, diese enthalten jeweils einen Auszug der ermittelten Laufzeiten.

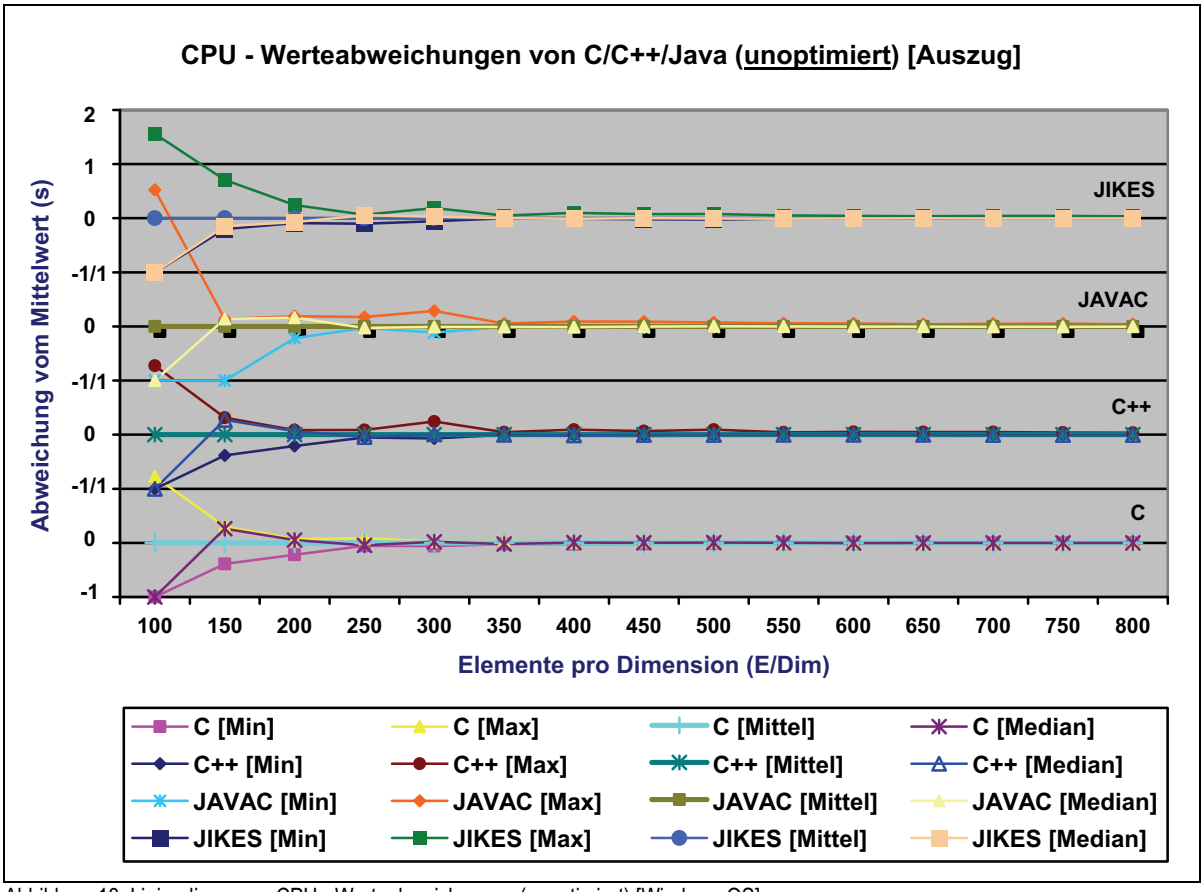


Abbildung 18: Liniendiagramm CPU - Werteabweichungen (unoptimiert) [Windows OS]

Tabelle CPU - Vergleich Werteabweichungen (C/C++/Java unoptimiert) [Zeit in Sekunden]										
Sprache	**Wert**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
C	Min	0	0,015	0,046	0,109	0,187	0,312	0,562	0,859	1,265
	Max	0,016	0,032	0,063	0,125	0,204	0,329	0,579	0,891	1,296
	Mittel	0,00719	0,0245	0,0589	0,1153	0,1991	0,3197	0,5753	0,8742	1,275
	Median	0	0,031	0,062	0,11	0,203	0,313	0,578	0,875	1,281
C++	Min	0	0,015	0,046	0,109	0,187	0,328	0,562	0,859	1,25
	Max	0,016	0,032	0,063	0,125	0,25	0,344	0,625	0,938	1,39
	Mittel	0,00703	0,0244	0,0581	0,1152	0,2013	0,3289	0,5716	0,8801	1,2719
	Median	0	0,031	0,062	0,11	0,203	0,328	0,563	0,875	1,266
JAVAC	Min	0	0	0,031	0,078	0,14	0,25	0,453	0,671	1,062
	Max	0,016	0,016	0,047	0,094	0,203	0,266	0,5	0,75	1,156
	Mittel	0,00454	0,0141	0,0395	0,0802	0,1577	0,2523	0,4581	0,6883	1,0767
	Median	0	0,016	0,046	0,078	0,156	0,25	0,453	0,687	1,078
JIKES	Min	0	0,015	0,046	0,093	0,187	0,312	0,578	0,828	1,297
	Max	0,016	0,032	0,063	0,11	0,235	0,328	0,641	0,907	1,438
	Mittel	0,00625	0,0188	0,0506	0,1034	0,1977	0,313	0,5836	0,8438	1,3325
	Median	0	0,016	0,047	0,109	0,203	0,313	0,578	0,844	1,328

Tabelle 11: Vergleich CPU - Werteabweichungen (unoptimiert) [Windows OS]

Bei den unoptimierten Programmen kommt es zu Beginn (bis 250 E/Dim) zu deutlichen Abweichungen. Diese sind bei allen Programmen relativ gleich stark ausgeprägt. Bei der JAVAC-Anwendung kann jedoch eine geringfügig höhere Abweichung bzgl. des Maximalwertes im Vergleich zu den anderen Applikationen festgestellt werden. Somit ist zu schlussfolgern, dass beim Programmstart (100 E/Dim) die größten Unterschiede auftreten. Im folgenden Programmverlauf nehmen die Abweichungen immer weiter ab, sodass diese gegen den Mittelwert gehen.

Die durch die Compiler optimierten Programme zeigen im Unterschied zu den nicht optimierten Applikationen, größere Abweichungen zum Beginn. Des Weiteren werden die Differenzen auch erst später kleiner, sodass eine Angleichung an den Mittelwert erfolgt. Die Maximalwerte der C/C++-Programme zeigen in diesem Fallbeispiel eine besonders große Abweichung. Diese kann bedingt durch die "niedrige" Komplexität der Operation auftreten. Was bedeutet, dass die Matrix-Multiplikation bei wenig E/Dim kaum Zeit in Anspruch nimmt, dies äußert sich in der schlechten Messbarkeit. So ist die Genauigkeit im Bereich von Millisekunden kaum ausreichend um präzise Vergleichsdaten für Matrizen mit wenigen E/Dim zu erhalten.

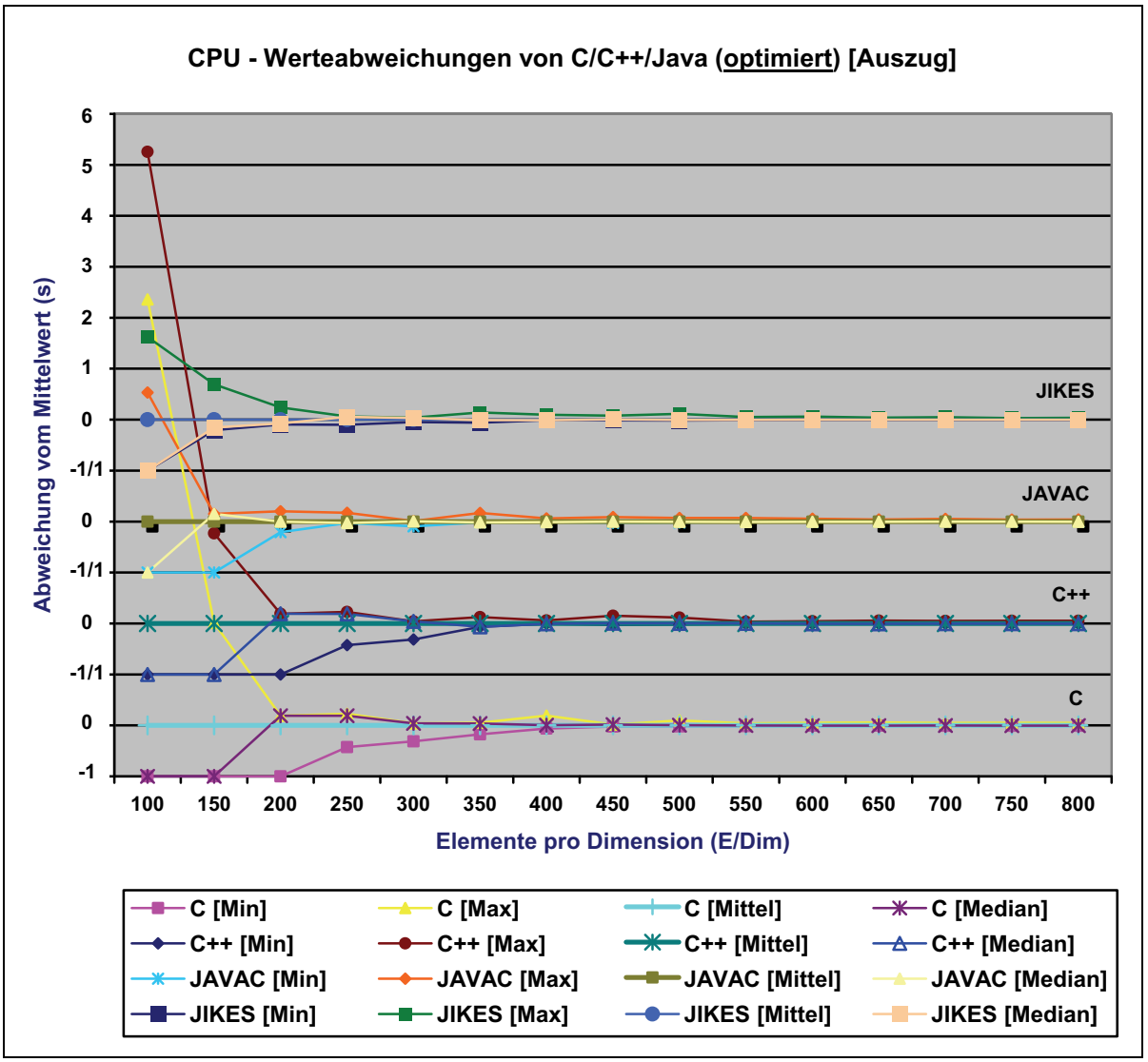


Abbildung 19: Liniendiagramm CPU - Werteabweichungen (optimiert) [Windows OS]

Tabelle CPU - Vergleich Werteabweichungen (C/C++/Java optimiert) [Zeit in Sekunden]

Sprache	Wert	100	150	200	250	300	350	400	450	500
C	Min	0	0	0	0,015	0,031	0,062	0,234	0,421	0,671
	Max	0,016	0,016	0,016	0,032	0,047	0,079	0,297	0,438	0,75
	Mittel	0,00171	0,0053	0,0134	0,0261	0,0452	0,0752	0,2494	0,4306	0,6839
	Median	0	0	0,016	0,031	0,047	0,078	0,25	0,437	0,687
C++	Min	0	0	0	0,015	0,031	0,078	0,234	0,421	0,671
	Max	0,016	0,016	0,016	0,032	0,047	0,094	0,25	0,5	0,765
	Mittel	0,00156	0,0058	0,0134	0,0261	0,0452	0,0836	0,2358	0,4344	0,6861
	Median	0	0	0,016	0,031	0,047	0,078	0,234	0,437	0,687
JAVAC	Min	0	0	0,031	0,078	0,141	0,25	0,453	0,687	1,062
	Max	0,016	0,016	0,047	0,094	0,157	0,297	0,485	0,766	1,156
	Mittel	0,00453	0,0139	0,0391	0,0803	0,1555	0,2544	0,4574	0,7044	1,0814
	Median	0	0,016	0,039	0,078	0,156	0,25	0,453	0,703	1,078
JIKES	Min	0	0,015	0,046	0,093	0,187	0,297	0,578	0,828	1,312
	Max	0,016	0,032	0,063	0,11	0,204	0,36	0,641	0,906	1,485
	Mittel	0,0061	0,0189	0,0508	0,1034	0,1964	0,3155	0,5842	0,8405	1,3344
	Median	0	0,016	0,047	0,109	0,203	0,313	0,578	0,843	1,328

Tabelle 12: Vergleich CPU - Werteabweichungen (optimiert) [Windows OS]

5.1.2 Messung 2 - Linux OS

Für die auf dem Linux Betriebssystem berechneten Mittelwerte, zeigt das nachfolgende Liniendiagramm einen Auszug der C/C++/Java-Programme (optimiert und unoptimiert).

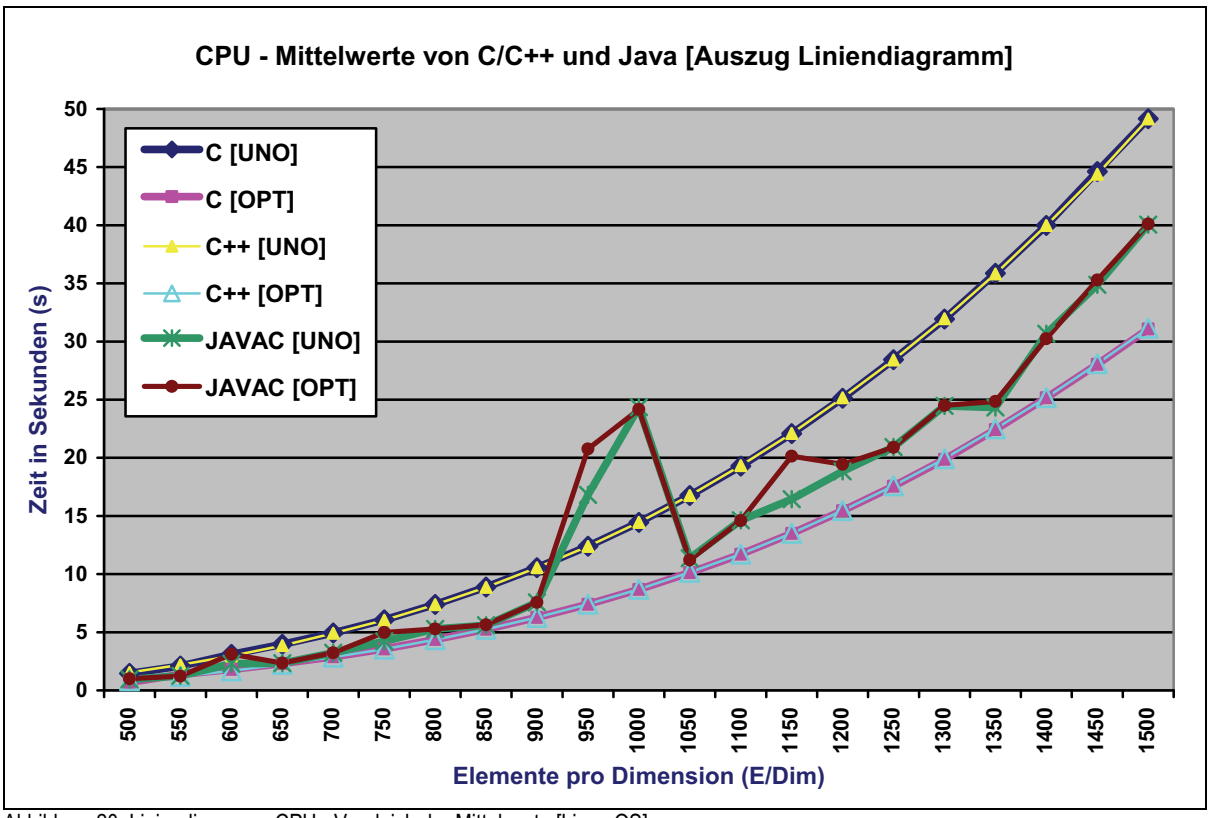


Abbildung 20: Liniendiagramm CPU - Vergleich der Mittelwerte [Linux OS]

Auf dem Linux-Betriebssystem weisen die unoptimierten C- bzw. C++-Programme äquivalente Laufzeiten auf, gleiches gilt für die optimierten Anwendungen. Im Gegensatz zu den ermittelten Windows-Ergebnissen kann die optimierte C-Anwendung bei steigender Komplexität geringfügig kürzere Laufzeiten vorweisen (s. "Tabelle 13: Vergleich CPU - Mittelwerte [Linux OS]").

Im Bereich von ca. 900 bis 1050 E/Dim weisen die Java-Programme verhältnismäßig hohe Laufzeiten auf. Ein Einfluss durch Betriebssystemprozesse oder Benutzerinteraktionen kann jedoch ausgeschlossen werden, da das unoptimierte bzw. optimierte Programm ein annähernd gleiches Laufzeitverhalten aufzeigt. Weiterhin wurden die Applikationen zu unterschiedlichen Zeiten ausgeführt.

Tabelle CPU (Auszug) - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden]						
E/Dim	C [UNO / OPT]		C++ [UNO / OPT]		JAVAC [UNO / OPT]	
800	7,3877	4,3716	7,3681	4,3586	5,29026	5,27899
850	8,8927	5,2586	8,8648	5,2563	5,60666	5,64469
900	10,5548	6,2733	10,5479	6,273	7,56386	7,57251
950	12,3938	7,4016	12,3843	7,4155	16,81887	20,76526
1000	14,4479	8,6806	14,4489	8,6964	24,35519	24,1646
1050	16,7586	10,1253	16,7652	10,149	11,36662	11,21599
1100	19,2917	11,7056	19,3343	11,7236	14,62598	14,59855
1150	22,0841	13,4941	22,0991	13,5179	16,44588	20,13759
1200	25,1245	15,4517	25,163	15,4705	18,83383	19,43799
1250	28,434	17,5588	28,4144	17,6001	20,92469	20,90188
1300	31,937	19,8685	31,9659	19,9285	24,46905	24,50398

Tabelle 13: Vergleich CPU - Mittelwerte [Linux OS]

Äquivalent zu den auf dem Windows-Betriebssystem ermittelten Ergebnissen, treten auch auf dem Linux-System die größten Abweichungen zu Beginn der Programmausführung auf. Die Maximalwerte der Java-Anwendung für 100 und 150 E/Dim, wurden aufgrund der sehr hohen Abweichung nicht im nachstehenden Diagramm (Abbildung 21) aufgeführt. Die ermittelten Werte können jedoch aus der darauf folgenden Tabelle ("Tabelle 14: Vergleich CPU - Werteabweichungen (unoptimiert) [Linux OS]") entnommen werden.

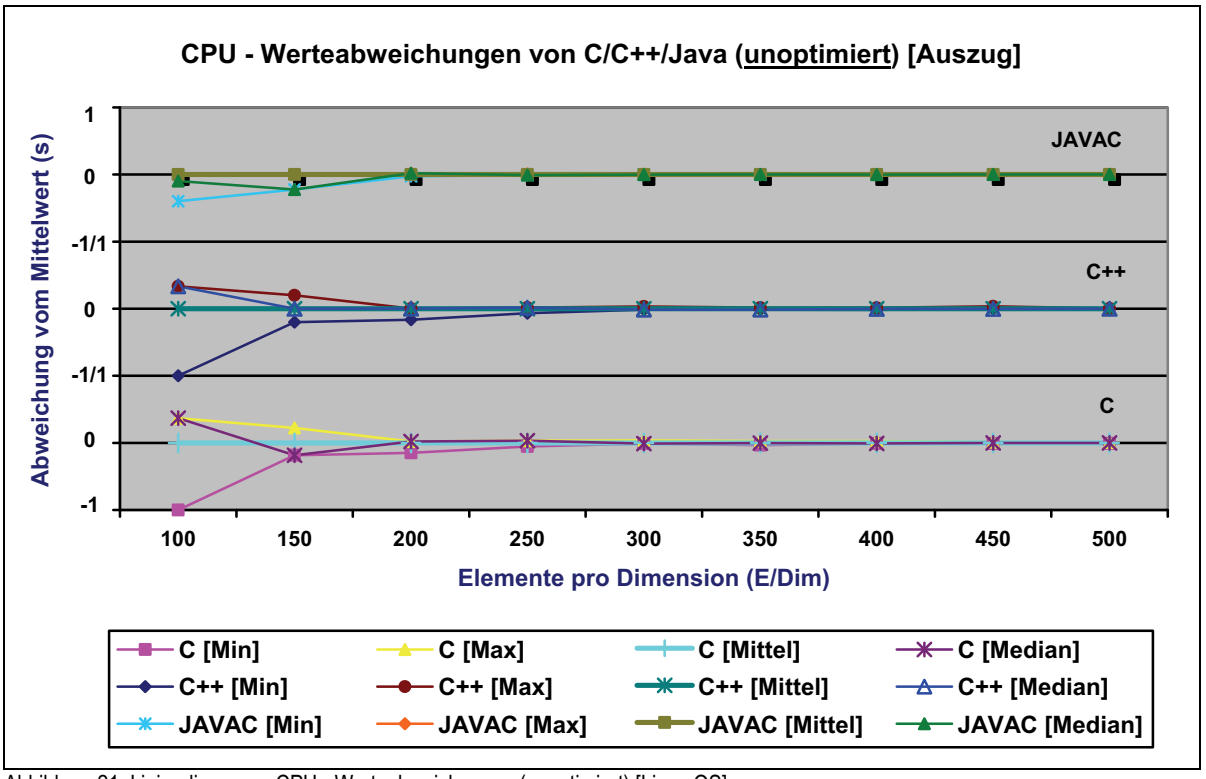


Abbildung 21: Liniendiagramm CPU - Werteabweichungen (unoptimiert) [Linux OS]

Tabelle CPU - Vergleich Werteabweichungen (C/C++/Java unoptimiert) [Zeit in Sekunden]										
Sprache	**Wert**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
C	Min	0	0,02	0,05	0,11	0,2	0,31	0,57	0,9	1,45
	Max	0,01	0,03	0,06	0,12	0,21	0,33	0,59	0,92	1,47
	Mittel	0,0073	0,0245	0,0588	0,1163	0,2023	0,3213	0,5741	0,9104	1,4618
	Median	0,01	0,02	0,06	0,12	0,2	0,32	0,57	0,91	1,46
C++	Min	0	0,02	0,05	0,11	0,2	0,32	0,6	1,02	1,53
	Max	0,01	0,03	0,06	0,12	0,21	0,33	0,62	1,06	1,55
	Mittel	0,0075	0,025	0,0598	0,1179	0,2031	0,3245	0,6108	1,0231	1,5418
	Median	0,01	0,025	0,06	0,12	0,2	0,32	0,61	1,02	1,54
JAVAC	Min	0,002	0,008	0,021	0,054	0,122	0,204	0,399	0,585	0,942
	Max	0,05	0,226	0,022	0,055	0,123	0,205	0,401	0,588	0,949
	Mittel	0,0033	0,0103	0,0216	0,0544	0,1222	0,2048	0,3994	0,5859	0,9443
	Median	0,003	0,008	0,022	0,054	0,122	0,205	0,399	0,586	0,944

Tabelle 14: Vergleich CPU - Werteabweichungen (unoptimiert) [Linux OS]

Bei den optimierten Applikationen (Abbildung 22, Tabelle 15) ist wiederum eine größere Werteabweichung festzustellen, welche durch die verwendeten Funktionen zur Laufzeitmessung bedingt sein kann. Funktionen die Laufzeiten im Bereich von Mikro- oder gar Nanosekundenbereich liefern, wären für Berechnungen die nur sehr wenig Zeit in Anspruch nehmen, zweckmäßiger.

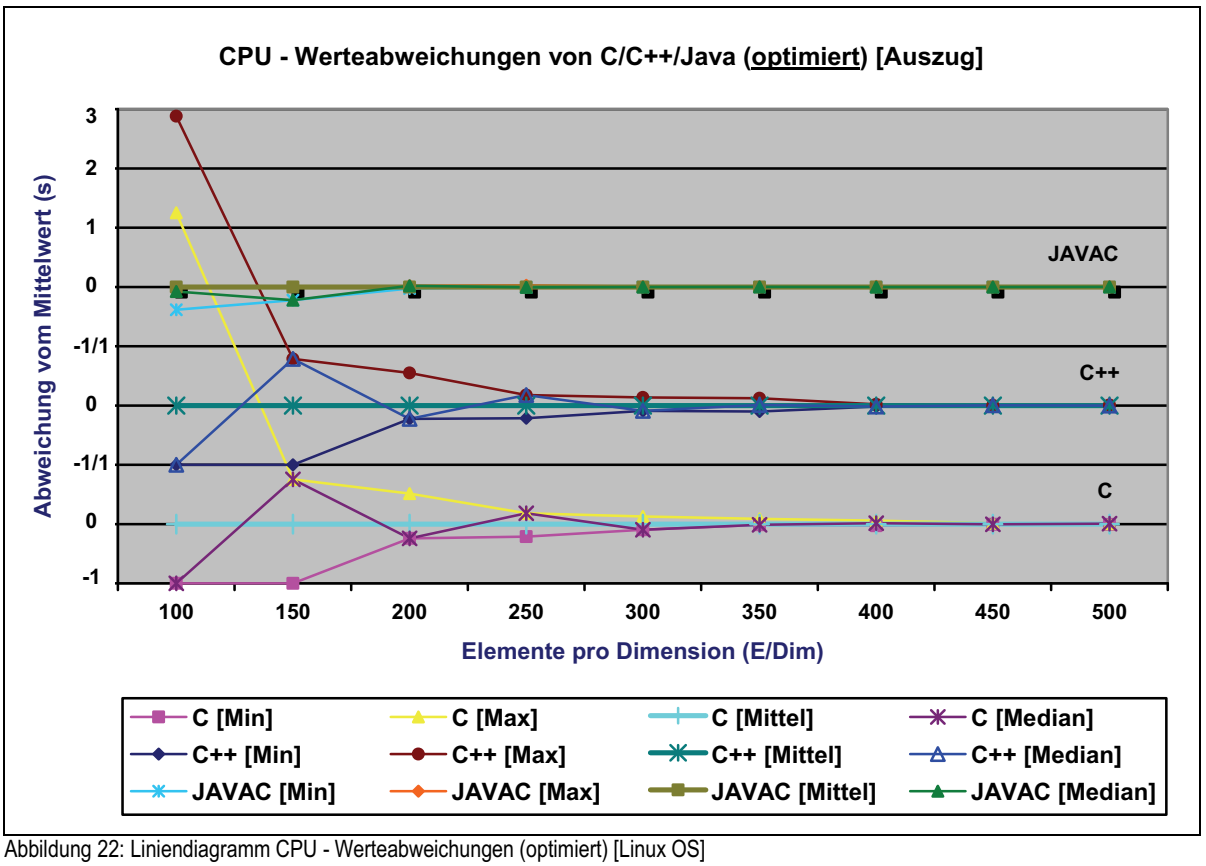


Abbildung 22: Liniendiagramm CPU - Werteabweichungen (optimiert) [Linux OS]

Wie bei dem Diagramm für unoptimierte Werte (Abbildung 21) sind auch in dieser Darstellung, die Maximalwerte der Java-Anwendung für 100 und 150 E/Dim aufgrund der sehr hohen Abweichung nicht aufgeführt. Die ermittelten Werte können der folgenden Tabelle entnommen werden.

Tabelle CPU - Werteabweichungen (C/C++/Java optimiert) [Zeit in Sekunden]

Sprache	Wert	100	150	200	250	300	350	400	450	500
C	Min	0	0	0,01	0,02	0,04	0,1	0,23	0,49	0,68
	Max	0,01	0,01	0,02	0,03	0,05	0,11	0,25	0,5	0,69
	Mittel	0,0016	0,0057	0,0132	0,0254	0,0443	0,1011	0,2364	0,4926	0,6854
	Median	0	0,01	0,01	0,03	0,04	0,1	0,24	0,49	0,69
C++	Min	0	0	0,01	0,02	0,04	0,08	0,26	0,49	0,82
	Max	0,01	0,01	0,02	0,03	0,05	0,1	0,27	0,5	0,84
	Mittel	0,0017	0,0056	0,0129	0,0255	0,044	0,0891	0,2648	0,498	0,8313
	Median	0	0,01	0,01	0,03	0,04	0,09	0,26	0,5	0,83
JAVAC	Min	0,002	0,008	0,021	0,054	0,122	0,205	0,403	0,584	0,997
	Max	0,043	0,226	0,022	0,056	0,123	0,206	0,408	0,586	1,003
	Mittel	0,0033	0,0103	0,0216	0,0544	0,1222	0,2055	0,4046	0,5843	1,0003
	Median	0,003	0,008	0,022	0,054	0,122	0,206	0,404	0,584	1,001

Tabelle 15: Vergleich CPU - Werteabweichungen (optimiert) [Linux OS]

5.1.3 Windows vs. Linux

Der im Weiteren aufgeführte Vergleich soll Unterschiede der optimierten Anwendungen mit Bezug auf die verwendeten Betriebssysteme aufzeigen. Wie bereits erwähnt, sei ausdrücklich auf die verschiedene Konfiguration der Systeme sowie Compiler hingewiesen.

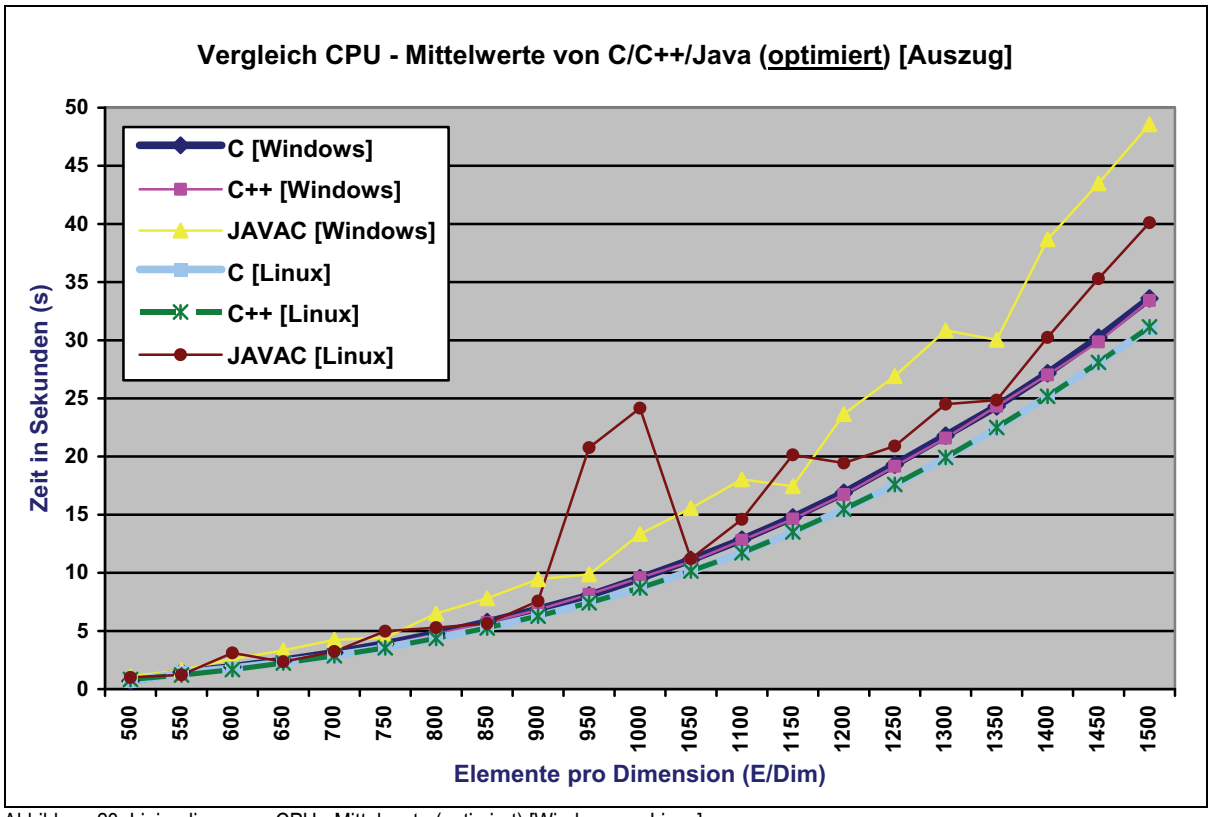


Abbildung 23: Liniendiagramm CPU - Mittelwerte (optimiert) [Windows vs. Linux]

Die C/C++-Anwendungen zeigen unabhängig vom zugrunde liegenden Betriebssystem ein gleiches Laufzeitverhalten, wobei die Anwendungen die auf dem Linux-System ausgeführt wurden, eine bessere Performance erzielen. Ab 1200 E/Dim kann auch bei den Java-Anwendungen ein annähernd gleiches Verhalten bzgl. der Laufzeit festgestellt werden, dabei sind kürzere Laufzeiten wieder bei der Linux-Variante zu finden.

Tabelle CPU (Auszug) - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden][3]

E/Dim	C [Windows / Linux]		C++ [Windows / Linux]		JAVAC [Windows / Linux]	
1000	9,531	8,681	9,571	8,696	13,332	24,165
1050	11,118	10,125	10,968	10,149	15,562	11,216
1100	12,857	11,706	12,800	11,724	18,036	14,599
1150	14,766	13,494	14,631	13,518	17,442	20,138
1200	16,884	15,452	16,731	15,471	23,652	19,438
1250	19,306	17,559	19,165	17,600	26,912	20,902
1300	21,799	19,869	21,584	19,929	30,841	24,504
1350	24,312	22,430	24,326	22,497	30,037	24,855
1400	27,159	25,159	27,023	25,178	38,683	30,249
1450	30,229	28,037	29,852	28,091	43,497	35,298
1500	33,594	31,080	33,424	31,144	48,563	40,110

Tabelle 16: Vergleich CPU - Mittelwerte (optimiert) [Windows vs. Linux]

[3] Werte gerundet auf die 3. Nachkommastelle

5.2 Speicherverwaltung

In diesem Punkt werden die ermittelten Laufzeiten der Programme für die Verkettung von Werten zweier Listen dargestellt. Hierbei lag der laufzeitrelevante Aspekt auf der Allokation von Speicher.

Angaben zur Messung:

- Problemgrößen: 200 bis 4000 E/Dim (je Liste bzw. Matrix)
- Elementgröße: 10 Zeichen (zufällige Buchstaben von 'a' bis 'z')
- Intervall: 200 E/Dim
- Wiederholungen: 20 je Problemgröße

5.2.1 Messung 1 - Windows OS

Im nächsten Diagramm (Abbildung 24) sind die Mittelwerte der verschiedenen Applikationen (C/C++/Java, optimiert und unoptimiert), die auf dem Windows Betriebssystem ermittelt wurden, dargestellt.

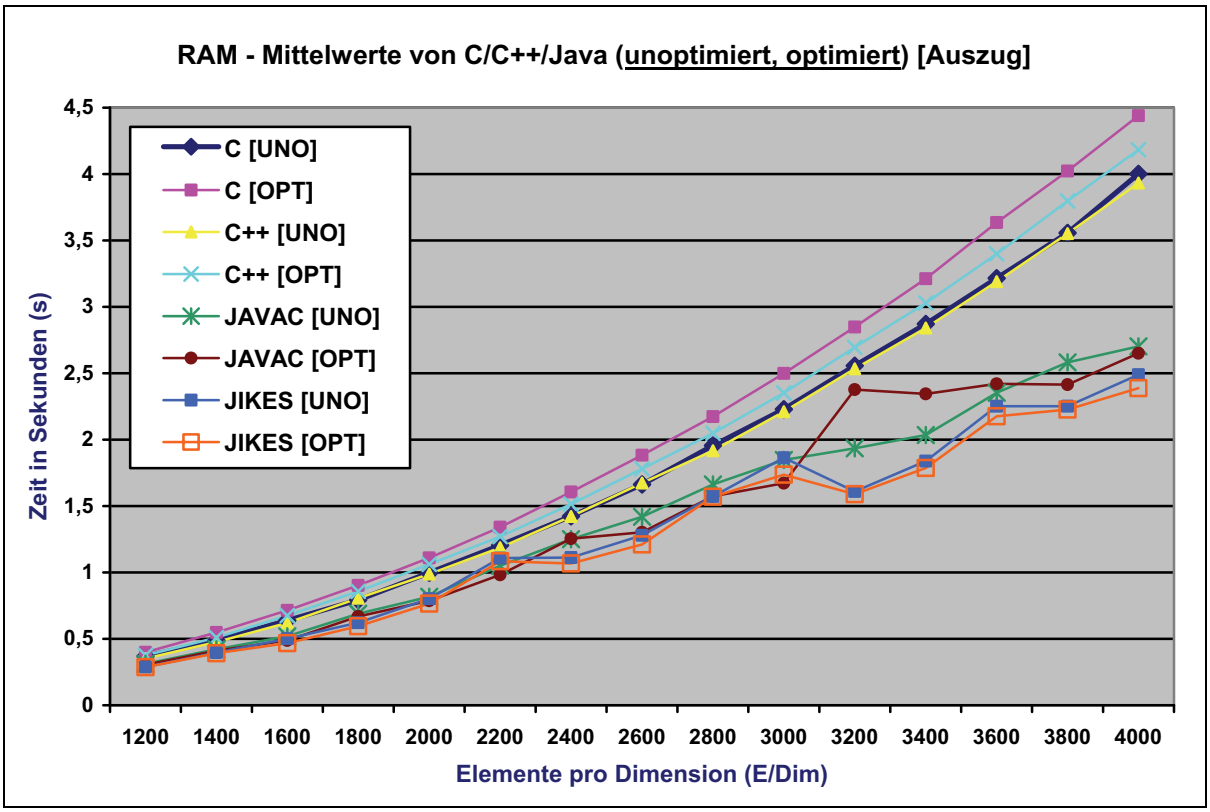


Abbildung 24: Liniendiagramm RAM - Vergleich der Mittelwerte [Windows OS]

Im Gegensatz zu Ergebnissen von anderen Bereichen fällt bei diesem Vergleich auf, dass die optimierten C/C++-Anwendungen eine signifikant schlechtere Performance als die unoptimierten Anwendungen aufweisen. Des Weiteren ist zwischen den optimierten C/C++-Varianten ein sichtbarer Unterschied vorhanden. Bei den unoptimierten ist die Differenz erheblich geringer, wobei die unoptimierte C++-Anwendung minimal kürzere Laufzeiten zeigt.

Weiterhin zeigen die Java-Anwendungen, trotz des nicht linearen Verlaufs zu jeder dargestellten Problemgröße die beste Performance. Betrachtet man den gesamten Verlauf, so sind die kürzesten Laufzeiten bei den JIKES-Anwendungen zu finden. Für die unoptimierte und optimierte Anwendung kann in diesem Fall ein Unterschied zugunsten der optimierten Variante festgestellt werden. Zwischen den JAVAC-Applikationen sind hingegen deutliche Abweichungen aufgetreten, die bessere Effizienz ist hierbei von der entsprechenden Problemgröße abhängig. In Relation zu den Laufzeiten für den Bereich CPU-Auslastung, so sind auch für diesen Bereich auffällige Sprünge bei den Java-Anwendungen zu verzeichnen.

Einen Auszug über gemessene Laufzeiten in Form von berechneten Mittelwerten wird in der nächsten Tabelle gegeben.

Tabelle RAM (Auszug) - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden][4]

E/Dim	C [UNO / OPT]		C++ [UNO / OPT]		JAVAC [UNO / OPT]		JIKES [UNO / OPT]	
2000	0,998	1,110	0,988	1,061	0,816	0,787	0,802	0,765
2200	1,205	1,341	1,187	1,270	1,050	0,982	1,109	1,084
2400	1,423	1,606	1,421	1,515	1,251	1,255	1,113	1,068
2600	1,663	1,883	1,674	1,780	1,420	1,303	1,280	1,211
2800	1,956	2,173	1,917	2,048	1,663	1,573	1,572	1,570
3000	2,230	2,498	2,211	2,352	1,848	1,672	1,864	1,736
3200	2,556	2,848	2,534	2,695	1,935	2,377	1,609	1,589
3400	2,871	3,213	2,841	3,029	2,035	2,344	1,839	1,787
3600	3,216	3,635	3,190	3,398	2,353	2,420	2,252	2,176
3800	3,556	4,023	3,552	3,797	2,581	2,414	2,252	2,227
4000	4,001	4,440	3,932	4,183	2,703	2,651	2,489	2,388

Tabelle 17: Vergleich RAM - Mittelwerte [Windows OS]

Im Folgenden werden die Werteabweichungen der unoptimierten (Abbildung 25, Tabelle 18) und optimierten Anwendungen (Abbildung 26, Tabelle 19) aufgeführt.

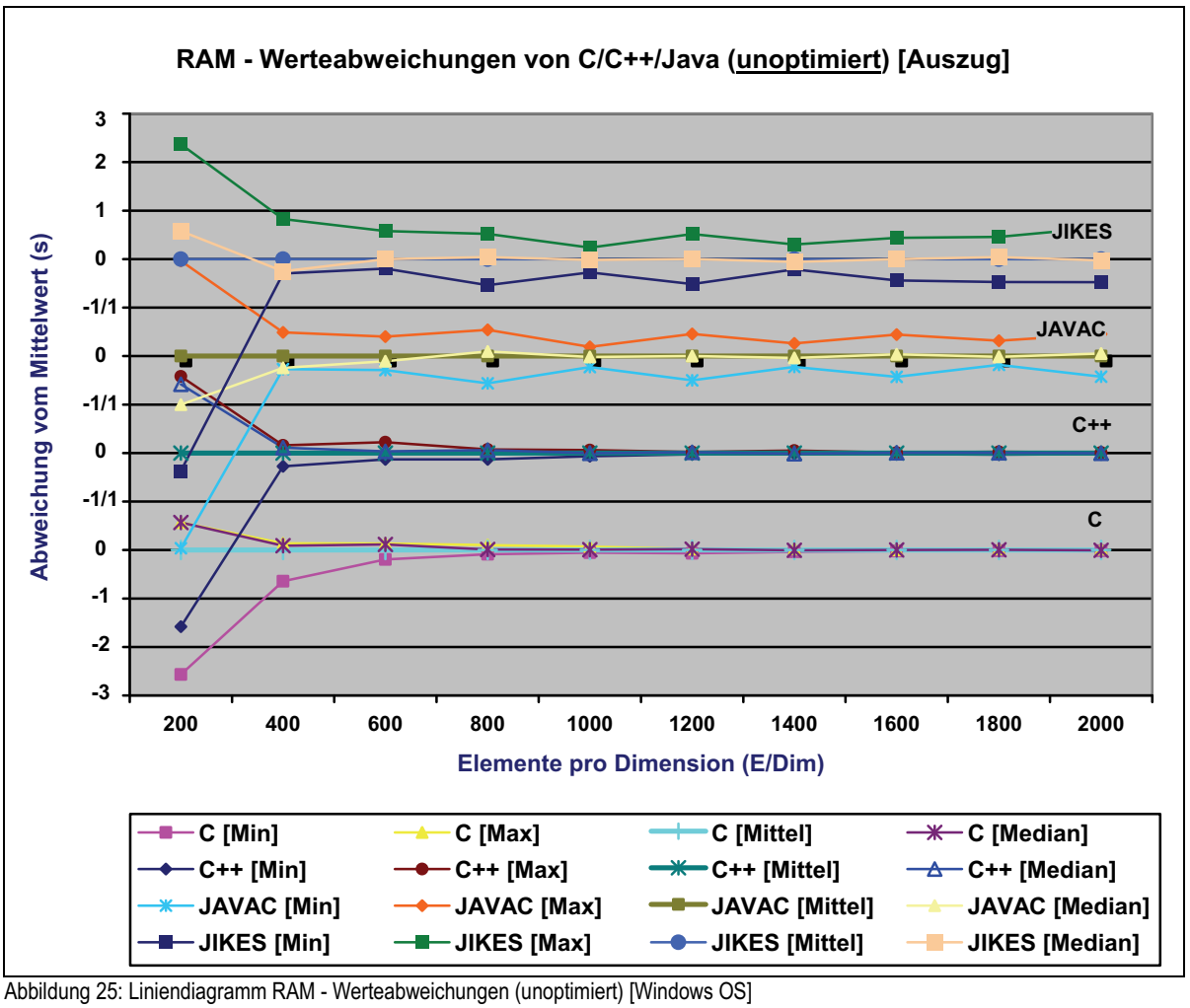


Abbildung 25: Liniendiagramm RAM - Werteabweichungen (unoptimiert) [Windows OS]

[4] Werte gerundet auf die 3. Nachkommastelle

Tabelle RAM - Vergleich Werteabweichungen (C/C++/Java unoptimiert) [Zeit in Sekunden]											
Sprache	**Wert**	**200**	**400**	**600**	**800**	**1000**	**1200**	**1400**	**1600**	**1800**	**2000**
C	Min	-0,016	0,015	0,078	0,156	0,249	0,344	0,469	0,624	0,78	0,983
	Max	0,016	0,048	0,11	0,188	0,281	0,376	0,501	0,642	0,813	1,017
	Mittel	0,0102	0,0422	0,0969	0,1706	0,2632	0,3688	0,4874	0,639	0,7931	0,9976
	Median	0,016	0,046	0,108	0,1725	0,265	0,3745	0,485	0,64	0,796	0,992
C++	Min	-0,016	0,03	0,078	0,14	0,235	0,343	0,468	0,609	0,78	0,969
	Max	0,016	0,048	0,11	0,173	0,267	0,36	0,5	0,626	0,828	1,001
	Mittel	0,0062	0,0414	0,0898	0,161	0,2518	0,3524	0,4757	0,621	0,8064	0,9882
	Median	0,015	0,046	0,093	0,171	0,2505	0,358	0,4695	0,624	0,812	0,9845
JAVAC	Min	-0,016	0,031	0,063	0,062	0,173	0,156	0,328	0,297	0,563	0,47
	Max	0,016	0,063	0,124	0,219	0,266	0,454	0,532	0,75	0,907	1,187
	Mittel	0,0054	0,0424	0,0884	0,1421	0,2234	0,3116	0,4218	0,5195	0,689	0,8165
	Median	0	0,032	0,079	0,1555	0,219	0,3115	0,405	0,539	0,679	0,859
JIKES	Min	-0,016	0,03	0,063	0,062	0,156	0,14	0,312	0,28	0,327	0,421
	Max	0,016	0,078	0,124	0,204	0,266	0,437	0,516	0,72	0,907	1,327
	Mittel	0,0048	0,0428	0,0785	0,1344	0,2151	0,2884	0,3968	0,5011	0,6216	0,8019
	Median	0,0075	0,032	0,0785	0,14	0,2115	0,2895	0,3745	0,501	0,648	0,78

Tabelle 18: Vergleich RAM - Werteabweichungen (unoptimiert) [Windows OS]

Bei den Werteabweichungen verhalten sich die C/C++-Applikationen ähnlich denen vorangegangener Messungen. So liegen die größten Abweichungen bei Beginn der Programmausführung bzw. bei den kleinsten Problemgrößen. Für die Java-Applikationen sind zu Beginn ähnliche Werteabweichungen feststellbar, jedoch gehen diese im Verlauf nicht gegen den Mittelwert, sondern schwanken von Problemgröße zu Problemgröße. Dieses Verhalten kann jeweils für die JAVAC- und JIKES-Anwendung beobachtet werden.

Für die optimierten Java-Anwendungen kann wieder eine kontinuierliche Divergenz im Programmverlauf verfolgt werden. Im Verhältnis zu den anderen Differenzen ist die Abweichung des Minimal- bzw. Maximalwertes der JIKES-Anwendung verhältnismäßig groß. Für die optimierten C/C++-Programme kann im Gegensatz zu den unoptimierten Varianten kein wesentlicher Unterschied festgestellt werden.

Interessanterweise dauert jede Wiederholung der optimierten C++-Anwendung bei 600 E/Dim exakt 0,093 Sekunden (s. "Tabelle 19: Vergleich RAM - Werteabweichungen (optimiert) [Windows OS]"). Diese Kontinuität konnte bei keiner weiteren Applikation und Problemgröße bereichsübergreifend beobachtet werden.

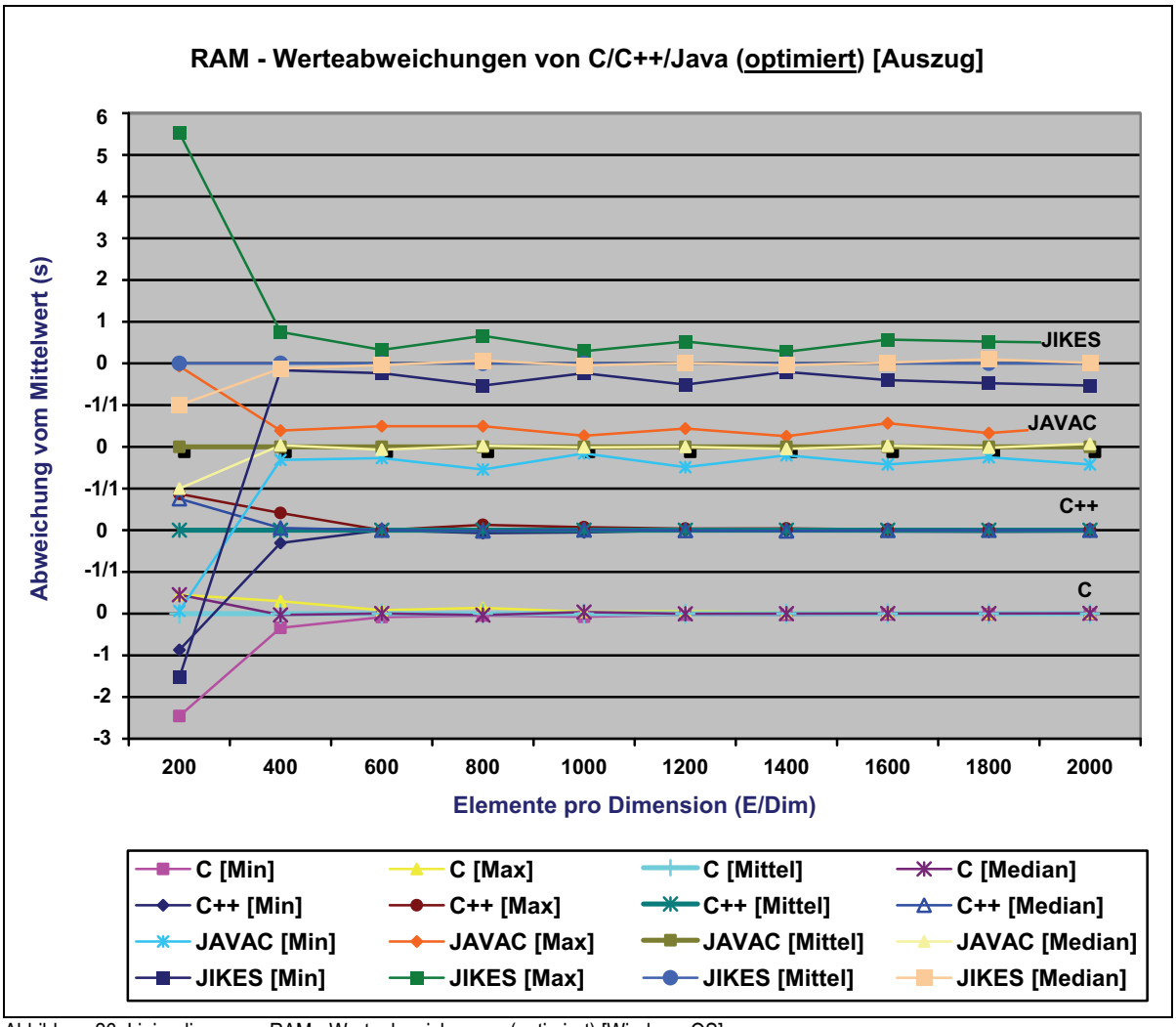


Abbildung 26: Liniendiagramm RAM - Werteabweichungen (optimiert) [Windows OS]

Tabelle RAM - Vergleich Werteabweichungen (C/C++/Java optimiert) [Zeit in Sekunden]

Sprache	Wert	200	400	600	800	1000	1200	1400	1600	1800	2000
C	Min	-0,016	0,032	0,093	0,171	0,264	0,389	0,531	0,702	0,889	1,093
	Max	0,016	0,063	0,11	0,204	0,298	0,421	0,563	0,734	0,907	1,126
	Mittel	0,011	0,0485	0,1016	0,1791	0,2853	0,3993	0,5483	0,7151	0,903	1,1101
	Median	0,016	0,047	0,102	0,173	0,296	0,398	0,547	0,7185	0,906	1,1165
C++	Min	-0,016	0,031	0,093	0,156	0,249	0,374	0,499	0,656	0,828	1,03
	Max	0,016	0,063	0,093	0,189	0,282	0,391	0,532	0,688	0,876	1,078
	Mittel	0,0086	0,0446	0,093	0,1677	0,2626	0,3758	0,5101	0,6743	0,8578	1,0611
	Median	0,015	0,047	0,093	0,1645	0,265	0,374	0,5005	0,6725	0,859	1,063
JAVAC	Min	-0,016	0,031	0,061	0,062	0,187	0,156	0,328	0,281	0,501	0,452
	Max	0,016	0,063	0,125	0,204	0,281	0,438	0,516	0,765	0,89	1,188
	Mittel	0,0055	0,0454	0,0836	0,1365	0,2219	0,3049	0,411	0,4882	0,6688	0,7865
	Median	0	0,047	0,078	0,14	0,2195	0,305	0,391	0,5005	0,6565	0,8435
JIKES	Min	-0,016	0,03	0,063	0,062	0,157	0,141	0,312	0,281	0,313	0,36
	Max	0,016	0,063	0,109	0,219	0,265	0,437	0,501	0,734	0,907	1,139
	Mittel	0,0025	0,0359	0,0824	0,1319	0,2047	0,2866	0,3914	0,4675	0,5956	0,7649
	Median	0	0,0315	0,079	0,141	0,195	0,289	0,3745	0,4775	0,6555	0,7735

Tabelle 19: Vergleich RAM - Werteabweichungen (optimiert) [Windows OS]

5.2.2 Messung 2 - Linux OS

Für die auf dem Linux-System ermittelten Laufzeiten zeigt das nachstehende Liniendiagramm (Abbildung 27) bzw. Tabelle (Tabelle 20) auszugsweise die berechneten Mittelwerte.

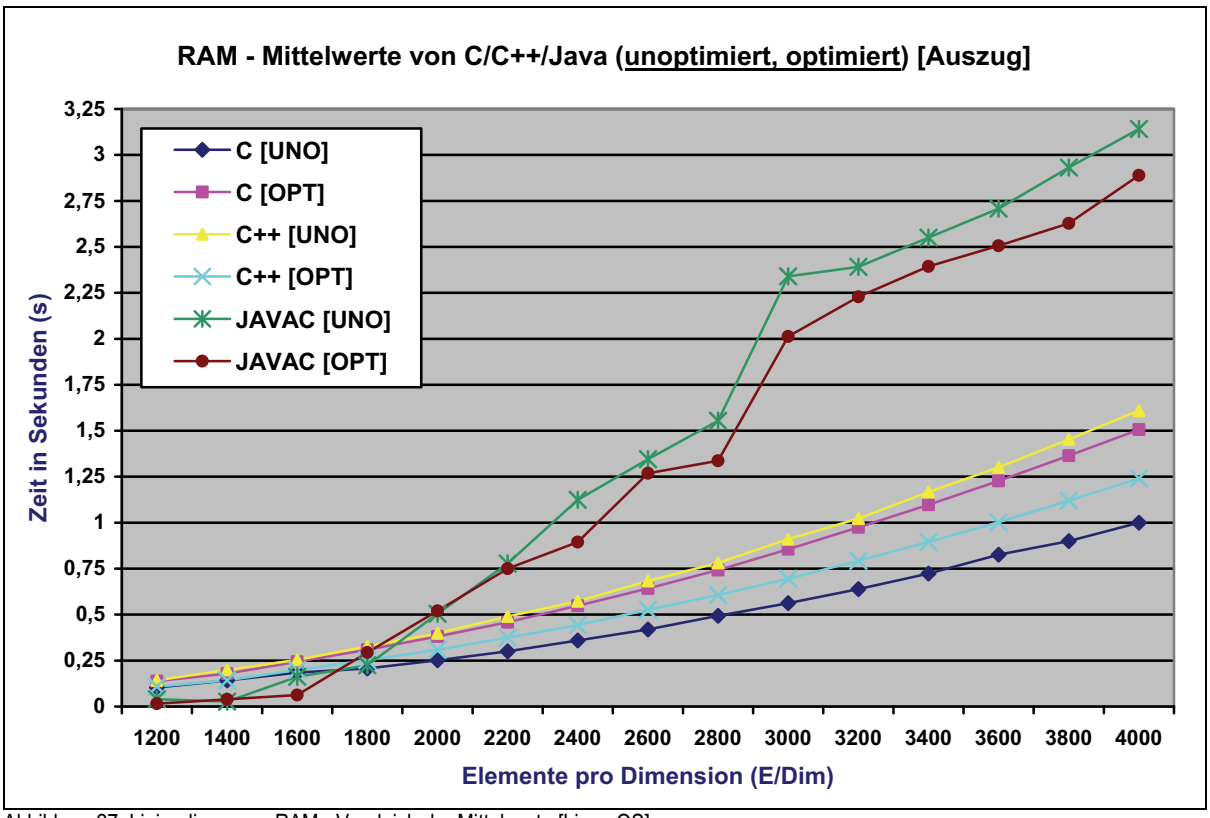


Abbildung 27: Liniendiagramm RAM - Vergleich der Mittelwerte [Linux OS]

Tabelle RAM - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden]						
E/Dim	C [UNO / OPT]		C++ [UNO / OPT]		JAVAC [UNO / OPT]	
2000	0,2515	0,381	0,3985	0,309	0,50485	0,51995
2200	0,3	0,4585	0,49	0,375	0,77825	0,7504
2400	0,359	0,548	0,5725	0,4425	1,12395	0,8941
2600	0,4195	0,6425	0,683	0,5255	1,3449	1,26855
2800	0,493	0,742	0,7815	0,6065	1,55375	1,33675
3000	0,562	0,855	0,91	0,695	2,34015	2,0128
3200	0,639	0,9745	1,0225	0,793	2,39075	2,22835
3400	0,723	1,0975	1,167	0,8955	2,5503	2,3942
3600	0,826	1,227	1,301	1,002	2,7078	2,5057
3800	0,8995	1,3645	1,453	1,12	2,93125	2,62755
4000	0,9995	1,506	1,61	1,241	3,14165	2,889

Tabelle 20: Vergleich RAM - Mittelwerte [Linux OS]

Bei den kleineren Problemgrößen weisen die JAVAC-Programme im Diagramm die kürzesten Laufzeiten auf, mit zunehmenden Verlauf bzw. steigenden Problemgrößen erhöhen sich die Zeiten jedoch deutlich. Wie auch die Messung auf dem Windows-System zeigt (Punkt 5.2.1), so ist auch bei diesem Vergleich ein klarer Unterschied zwischen optimierter und nicht optimierter Java-Anwendung festzustellen. Ähnlich sichtbare Differenzen zeigen auch die C- bzw. C++-Programme, wobei die optimierte C++- und für C die unoptimierte Anwendung eine bessere Effizienz aufweist. Letztere zeigt im Abschnitt von 1800 bis 4000 E/Dim die beste Performance.

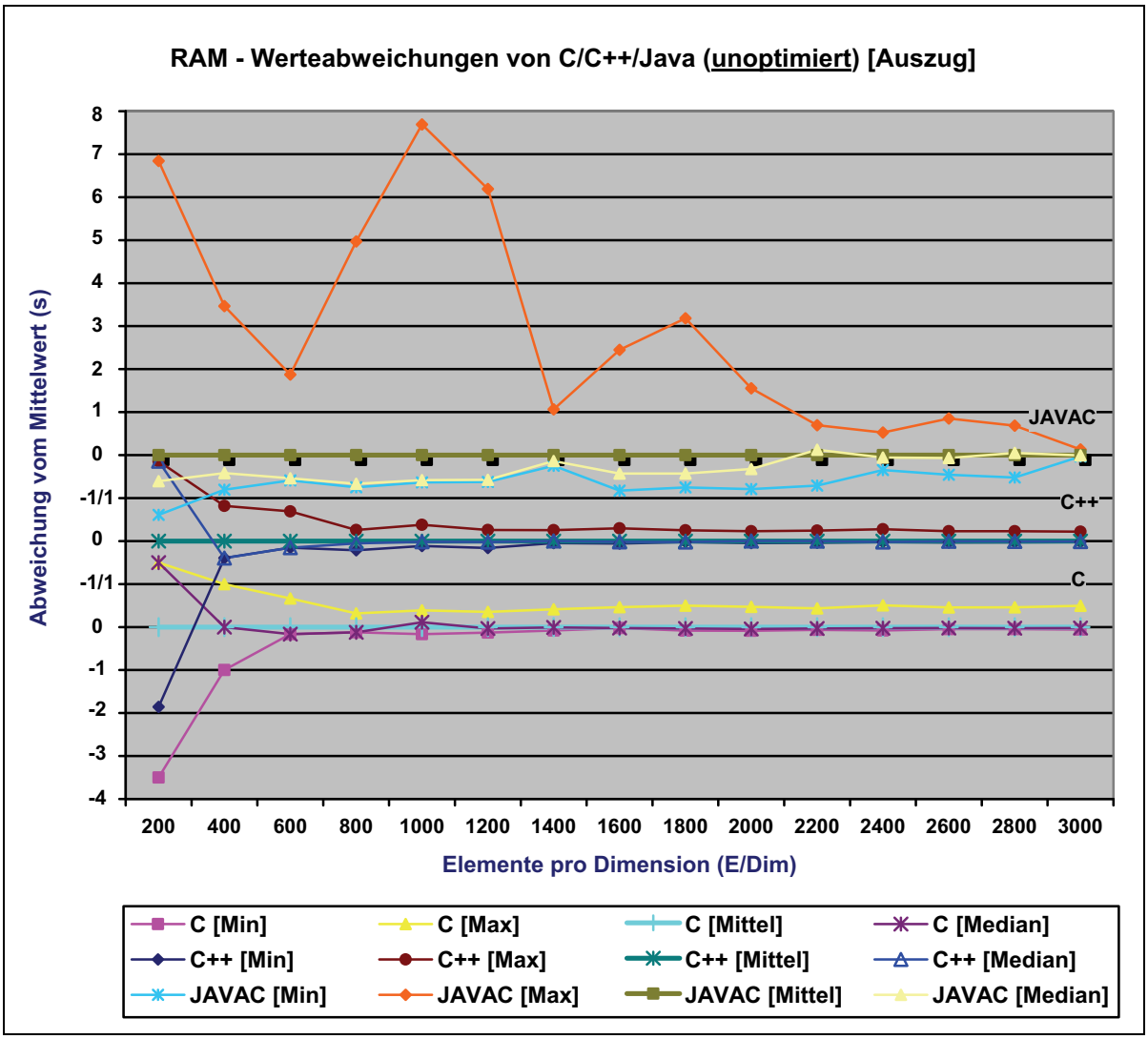


Abbildung 28: Liniendiagramm RAM - Werteabweichungen (unoptimiert) [Linux OS]

Tabelle - Vergleich Werteabweichungen (C/C++/Java unoptimiert) [Zeit in Sekunden]

Sprache	Wert	200	400	600	800	1000	1200	1400	1600	1800	2000
C	Min	-0,01	0	0,02	0,04	0,06	0,09	0,13	0,18	0,19	0,23
	Max	0,01	0,02	0,04	0,06	0,1	0,14	0,2	0,27	0,31	0,37
	Mittel	0,004	0,01	0,024	0,0455	0,072	0,1035	0,1415	0,1845	0,207	0,2515
	Median	0,01	0,01	0,02	0,04	0,08	0,1	0,14	0,18	0,2	0,24
C++	Min	-0,01	0,01	0,03	0,05	0,09	0,12	0,19	0,24	0,32	0,38
	Max	0,01	0,03	0,06	0,08	0,14	0,18	0,25	0,33	0,41	0,49
	Mittel	0,0035	0,0165	0,0355	0,0635	0,1015	0,143	0,1995	0,2545	0,3275	0,3985
	Median	0,01	0,01	0,03	0,06	0,1	0,14	0,2	0,25	0,32	0,4
JAVAC	Min	-0,001	0,001	0,004	0,006	0,01	0,015	0,02	0,028	0,056	0,106
	Max	0,02	0,023	0,028	0,143	0,239	0,288	0,055	0,559	0,939	1,29
	Mittel	0,00255	0,0052	0,0098	0,024	0,0275	0,0401	0,0267	0,1621	0,2245	0,5049
	Median	0,001	0,003	0,0045	0,008	0,0115	0,017	0,023	0,0925	0,128	0,342

Tabelle 21: Vergleich RAM - Werteabweichungen (unoptimiert) [Linux OS]

Bei den Differenzen (Abbildung 28, Tabelle 21) zeigen die Maximalwerte der JAVAC-Applikation eine besondere Auffälligkeit, so schwanken diese in einem Intervall von 400 E/Dim. Im Programmverlauf nehmen die Abweichungen schwankend ab, bis sie sich an den Mittelwert angleichen. Im Gegensatz dazu zeigen die Mittelwerte geringe Ungleichheiten, wobei die Differenzen der Minimal- und Maximalwerte für 600, 1400, 2400 und 3000 E/Dim im Verhältnis jeweils geringer werden.

Für die C- und C++-Anwendungen ist der Verlauf der Werte weitestgehend gleich, so treten die größten Differenzen bei 200 E/Dim auf, wobei die Minimalwerte eine größere Abweichung als die Maximalwerte aufzeigen. Weiterhin ist der Entwicklung der Werte zu entnehmen, dass eine Angleichung an den Mittelwert stattfindet, während die Maximalwerte jedoch in einem kontinuierlichen Abstand bleiben.

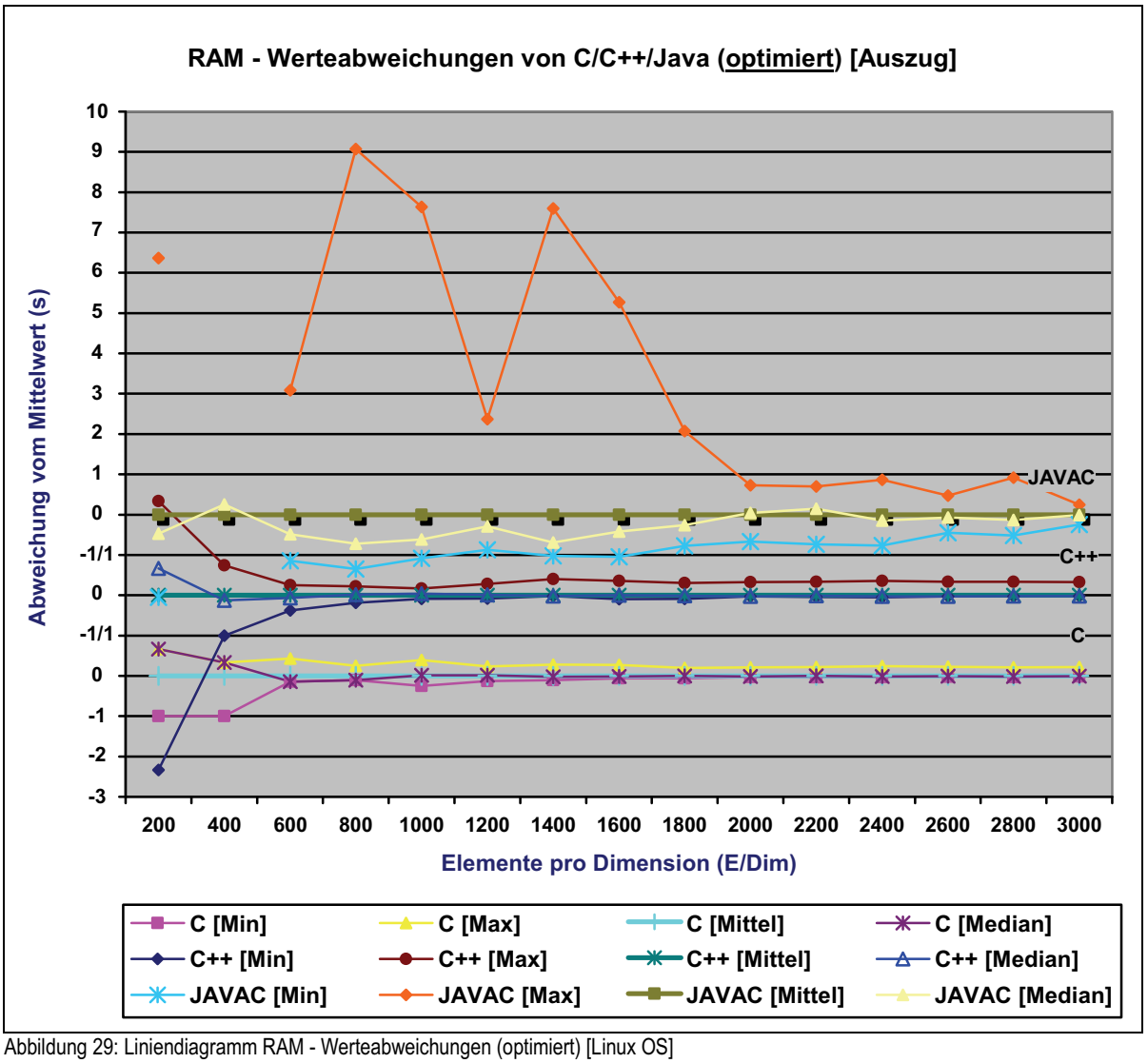


Abbildung 29: Liniendiagramm RAM - Werteabweichungen (optimiert) [Linux OS]

Eine wesentliche Äquivalenz hinsichtlich der Werteentwicklung zeigen auch die optimierten Programme (Abbildung 29), hierbei sind die Maximalwerte des Java-Programms weiter erhöht. Weniger große Abweichungen sind hingegen bei Programmbeginn der C-Anwendung festzustellen, während die Differenzen des C++-Programms an dieser Stelle weiter zugenommen haben.

Aufgrund hoher Abweichungen wurde Maximalwert der Java-Anwendung für 400 E/Dim sowie die Minimalwerte für 200, 400 E/Dim nicht dargestellt. Hierfür sei jedoch auf Tabelle 22 mit Werten für Problemgrößen im Bereich von 200 bis 2000 E/Dim verwiesen.

Tabelle RAM - Vergleich Werteabweichungen (C/C++/Java optimiert) [Zeit in Sekunden]											
Sprache	**Wert**	**200**	**400**	**600**	**800**	**1000**	**1200**	**1400**	**1600**	**1800**	**2000**
C	Min	0	0	0,03	0,05	0,07	0,12	0,16	0,23	0,29	0,37
	Max	0,01	0,02	0,05	0,07	0,13	0,17	0,23	0,31	0,37	0,46
	Mittel	0,006	0,015	0,035	0,056	0,0935	0,138	0,1795	0,244	0,309	0,381
	Median	0,01	0,02	0,03	0,05	0,095	0,14	0,175	0,24	0,31	0,375
C++	Min	-0,01	0	0,02	0,04	0,07	0,1	0,14	0,18	0,23	0,3
	Max	0,01	0,02	0,04	0,06	0,09	0,14	0,2	0,27	0,33	0,41
	Mittel	0,003	0,0115	0,032	0,049	0,077	0,109	0,143	0,199	0,2525	0,309
	Median	0,005	0,01	0,03	0,05	0,08	0,11	0,14	0,2	0,25	0,3
JAVAC	Min	-0,001	-0,062	-0,001	-0,005	-0,002	0,002	-0,001	-0,003	0,067	0,171
	Max	0,007	0,025	0,028	0,146	0,203	0,052	0,338	0,394	0,905	0,898
	Mittel	0,00095	0,0016	0,0069	0,0145	0,0235	0,0155	0,0393	0,0629	0,2943	0,52
	Median	0,0005	0,002	0,0035	0,004	0,009	0,011	0,012	0,0365	0,2175	0,5415

Tabelle 22: Vergleich RAM - Werteabweichungen (optimiert) [Linux OS]

5.2.3 Windows vs. Linux

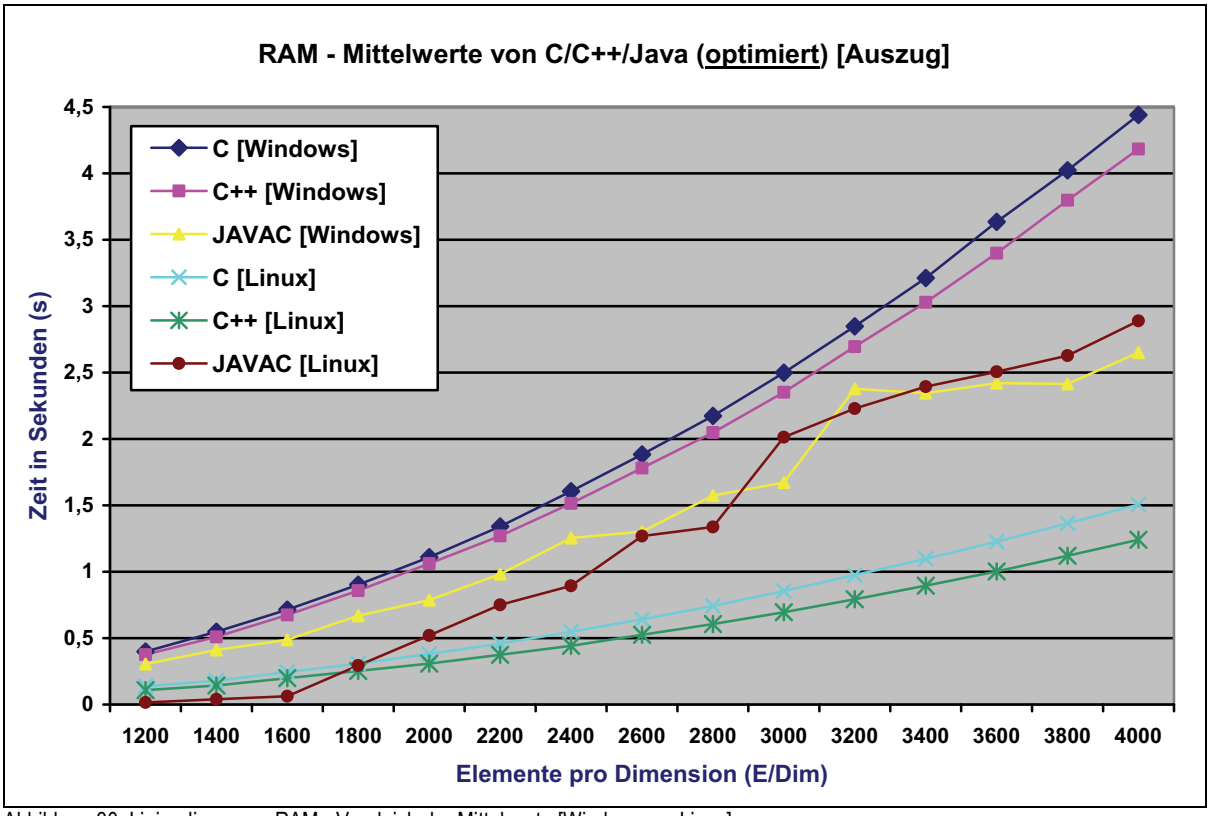


Abbildung 30: Liniendiagramm RAM - Vergleich der Mittelwerte [Windows vs. Linux]

Tabelle RAM (Auszug) - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden][5]

E/Dim	C [Windows / Linux]		C++ [Windows / Linux]		JAVAC [Windows / Linux]	
2000	1,110	0,381	1,061	0,309	0,787	0,520
2200	1,341	0,459	1,270	0,375	0,982	0,750
2400	1,606	0,548	1,515	0,443	1,255	0,894
2600	1,883	0,643	1,780	0,526	1,303	1,269
2800	2,173	0,742	2,048	0,607	1,573	1,337
3000	2,498	0,855	2,352	0,695	1,672	2,013
3200	2,848	0,975	2,695	0,793	2,377	2,228
3400	3,213	1,098	3,029	0,896	2,344	2,394
3600	3,635	1,227	3,398	1,002	2,420	2,506
3800	4,023	1,365	3,797	1,120	2,414	2,628
4000	4,440	1,506	4,183	1,241	2,651	2,889

Tabelle 23: Vergleich RAM - Mittelwerte (optimiert) [Windows vs. Linux]

Der Vergleich der Mittelwerte zwischen Windows- und Linux-Betriebssystem zeigt in diesem Fall für C/C++ erhebliche Laufzeitunterschiede. Für z. B. 4000 E/Dim ist die gleiche C-Anwendung auf dem Linux-System um den Faktor 2,9 schneller, für C++ beträgt der Faktor an dieser Stelle sogar 3,4. Für Java sind hingegen weniger große Unterschiede zu beobachten, was bedeutet, dass die Programme laufzeitbezogen dichter zusammenliegen und kürzere Laufzeiten von der jeweiligen Problemgröße abhängen.

[5] Werte gerundet auf die 3. Nachkommastelle

5.3 I/O-Operationen / Dateiarbeit

Die Ergebnisse der Programme zum zeichenweisen Kopieren werden in den folgenden Punkten aufgeführt. Die dargestellten Laufzeitergebnisse sind absteigend sortiert (im Diagramm von links nach rechts, in der Tabelle von oben nach unten).

Angaben zur Messung:

- Problemgrößen: 3 Dateien mit jeweils 150 MB, 352 MB, 700 MB
- Wiederholungen: 20 pro Datei

5.3.1 Messung 1 - Windows OS

Das nachstehende Balkendiagramm (Abbildung 31) zeigt die Laufzeiten der Applikationen, die auf dem Windows-Betriebsystem gemessen wurden. Hierbei ist als Besonderheit zu beachten, dass für die Sprache C, unterschiedliche Funktionen genutzt wurden. Zum einen sind die Funktionen `fgetc()`, `fputc()` und zum anderen die Makros `getc()`, `putc()` eingesetzt worden.

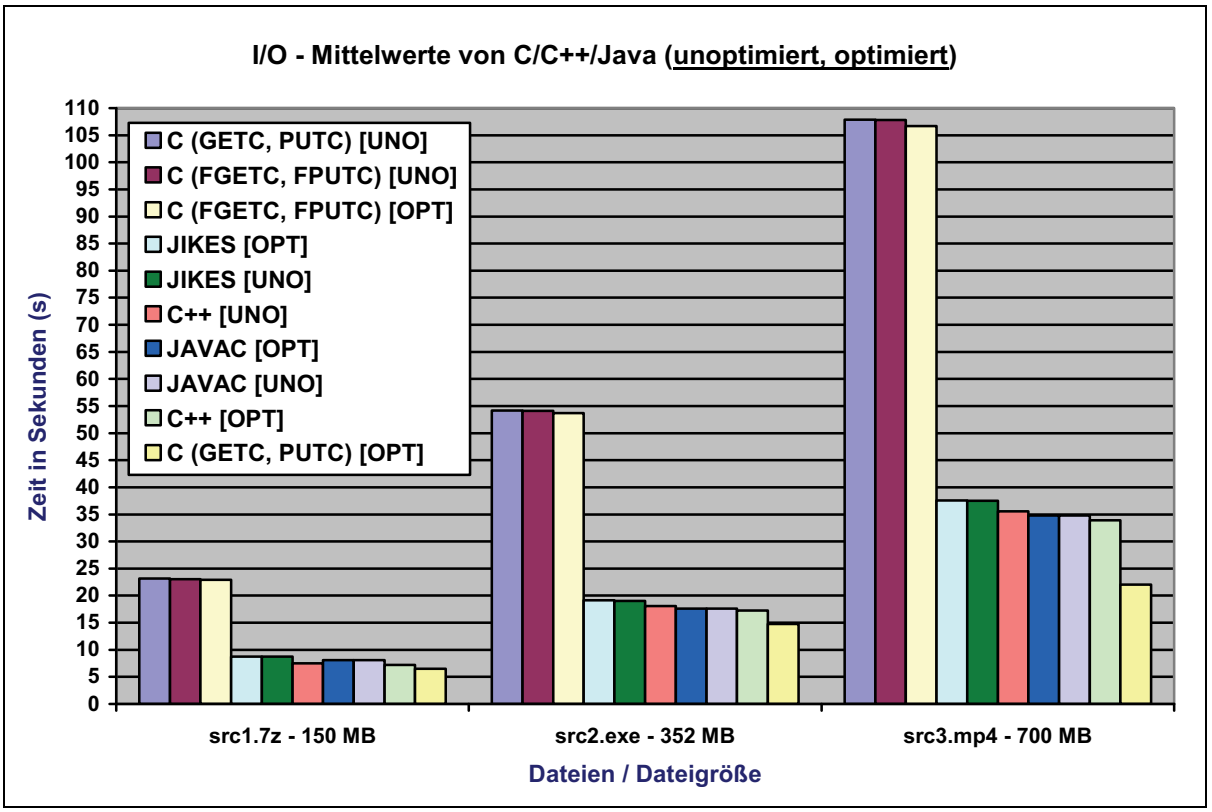


Abbildung 31: Balkendiagramm I/O - Vergleich der Mittelwerte [Windows OS]

Die kürzesten Laufzeiten konnten bei jeder Problemgröße mit der optimierten C-Variante gemessen werden, diese nutzte hierbei die Makro-Funktionen `getc()`, `putc()`. Ohne Optimierung sind die mit minimalem Abstand längsten Laufzeiten die Folge. Für die C++-Applikationen sind keine äquivalent großen Abweichungen bzw. Besonderheiten aufgetreten.

Bei den Java-Programmen liegen die mit dem javac-Compiler übersetzten Programme vor den JIKES-Anwendungen. Jedoch weisen die jeweils mit Optimierungsschalter kompilierten Quellprogramme minimal längere Laufzeiten auf. Im Weiteren soll ein Überblick mit allen berechneten Werten sowie Besonderheiten in Tabellenform (Tabelle 24) gegeben werden.

Tabelle I/O - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden]					
Sprache	**Besonderheit**	**Optimierung**	**src1.7z - 150 MB**	**src2.exe - 352 MB**	**src3.mp4 - 700 MB**
C	GETC,PUTC	nein	23,164	54,1837	107,8876
C	FGETC,FPUTC	nein	23,04295	54,14305	107,8346
C	FGETC,FPUTC	ja	22,91405	53,68285	106,6946
Java	JIKES	ja	8,7196	19,1414	37,57505
Java	JIKES	nein	8,7163	19,0258	37,5024
C++	-	nein	7,48675	18,0976	35,5773
Java	JAVAC	ja	8,09305	17,61735	34,82345
Java	JAVAC	nein	8,09065	17,6055	34,8087
C++	-	ja	7,2054	17,2219	33,903
C	GETC,PUTC	ja	6,4726	14,768	22,03895

Tabelle 24: Vergleich I/O - Mittelwerte [Windows OS]

Die dargestellten Werteabweichungen (Abbildung 32) beziehen sich jeweils auf die kleinste Datei (150 MB). Hierbei wird an einigen Anwendungen (unoptimiert und optimiert) beispielhaft gezeigt, wie die Werteentwicklung verläuft.

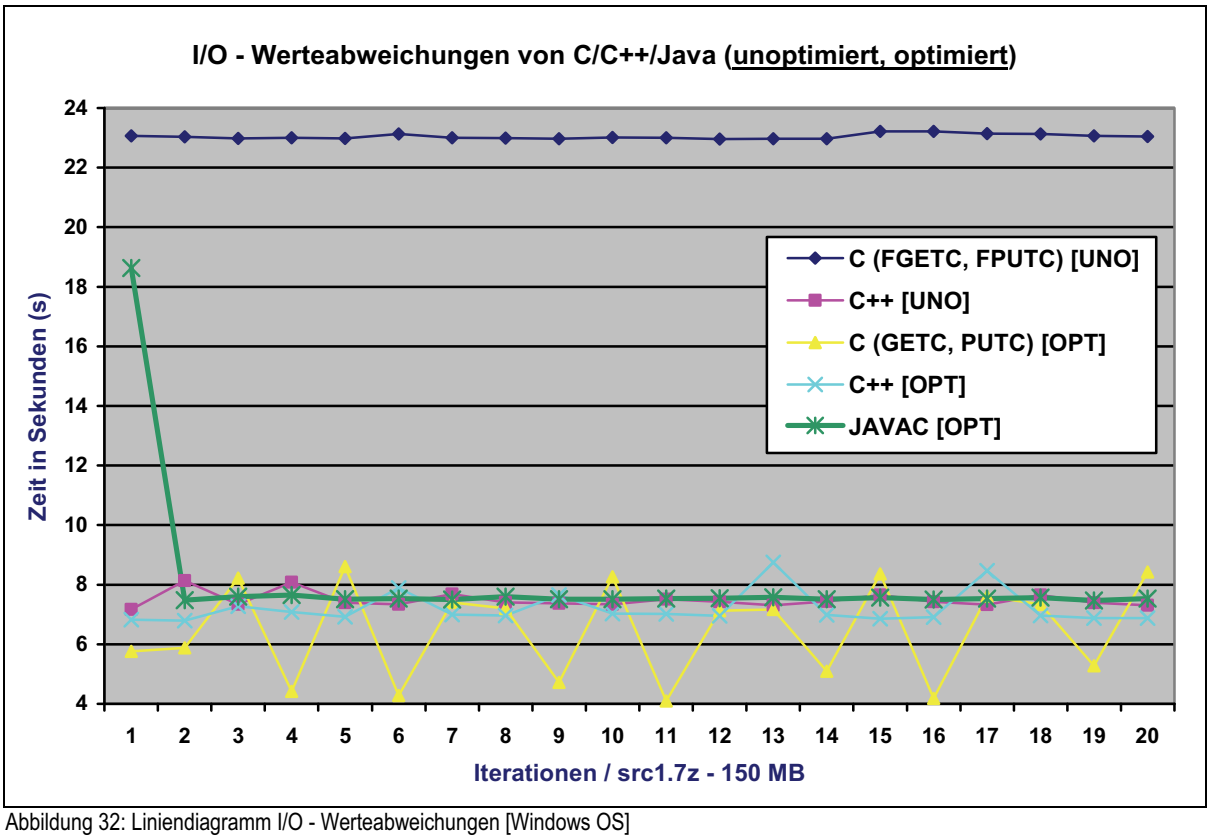


Abbildung 32: Liniendiagramm I/O - Werteabweichungen [Windows OS]

Einen überwiegenden linearen Verlauf kann bei der dargestellten unoptimierten C-Anwendung, welche die Funktionen `fgetc()`, `fputc()` nutzt, festgestellt werden. Völlig verschieden hingegen verhält sich die optimierte C-Applikation, in welcher die Makro-Funktionen `getc()`, `putc()` implementiert wurden. Bei dieser Variante sind, bis auf wenige Ausnahmen, bei jeder Iteration stark differierende Werte zu erkennen. Für die optimierte C-Anwendung sind weitaus weniger große Schwankungen festzustellen. Die aufgeführte Java-Datenreihe zeigt bei der ersten Ausführung eine verhältnismäßig hohe Abweichung, welche bei folgenden Wiederholungen nicht mehr auftritt. Diese zu Beginn hohe Differenz sowie ein folgendes annähernd lineares Verhalten ist auch für die nicht aufgeführten JAVAC- bzw. JIKES-Anwendungen zu beobachten. Die nicht dargestellten C-Programme weisen grundsätzlich ähnliche Werteabweichungen, wie bei der aufgeführten C-Datenreihe (unoptimiert) auf, lediglich zu Programmbeginn sind die Differenzen geringfügig größer.

5.3.2 Messung 2 - Linux OS

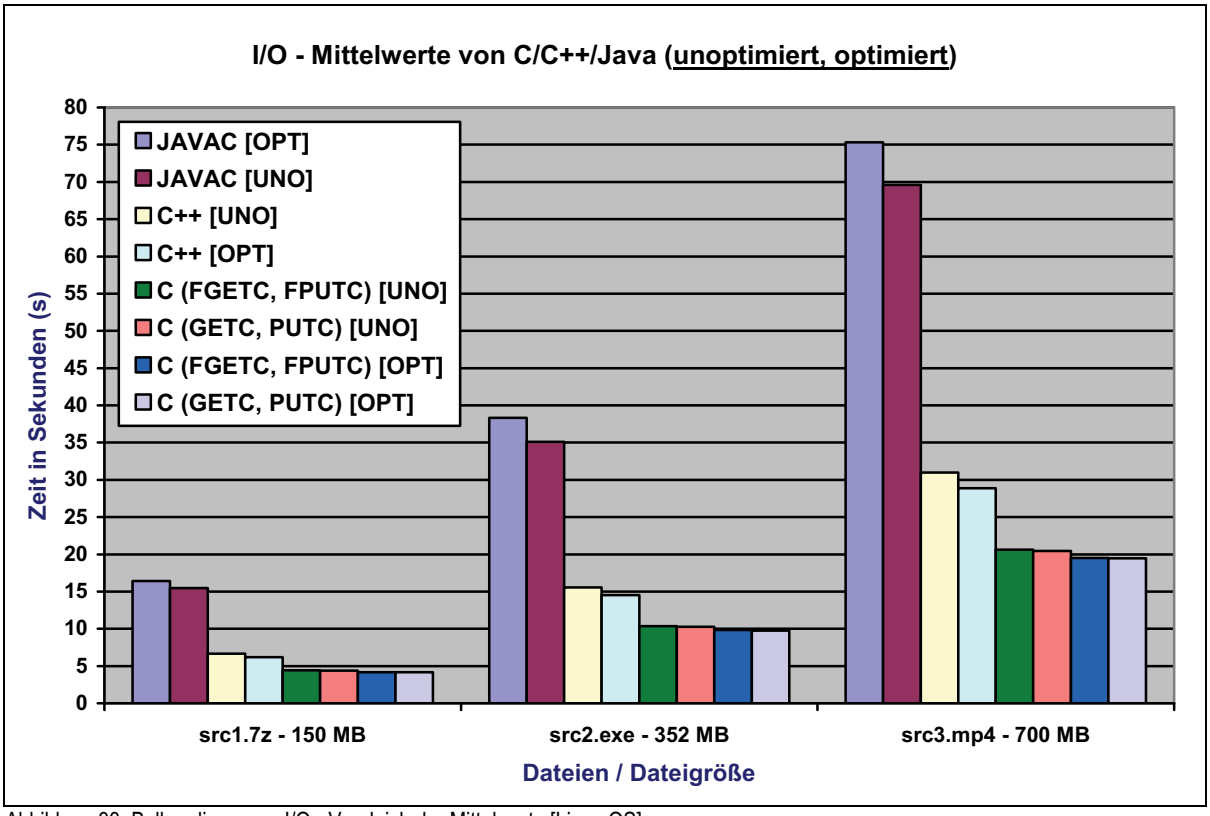


Abbildung 33: Balkendiagramm I/O - Vergleich der Mittelwerte [Linux OS]

Tabelle I/O - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden]

Sprache	Besonderheit	Optimierung	src1.7z - 150 MB	src2.exe - 352 MB	src3.mp4 - 700 MB
Java	JAVAC	ja	16,3934	38,3252	75,32675
Java	JAVAC	nein	15,4522	35,10975	69,6056
C++	-	nein	6,64	15,551	30,9735
C++	-	ja	6,174	14,506	28,875
C	FGETC,FPUTC	nein	4,416	10,3515	20,6285
C	GETC,PUTC	nein	4,3745	10,257	20,4315
C	FGETC,FPUTC	ja	4,1885	9,822	19,489
C	GETC,PUTC	ja	4,166	9,7665	19,4445

Tabelle 25: Vergleich I/O - Mittelwerte [Linux OS]

Die für Linux gemessenen Laufzeiten zeigen im Gegensatz zum Windows-System, eine weniger große Differenzierung. Die beste Performance erzielt (wenn auch mit minimalen Abstand) bei diesen Messungen wieder die optimierte C-Anwendung, welche Makros nutzt. Fast gleich auf liegt die optimierte C-Applikation, in der die Funktionen `fgetc()`, `fputc()` implementiert sind sowie folgend die unoptimierten Programme der Sprache C. Für C++ kann ein Unterschied zwischen unoptimierter bzw. optimierter Variante festgestellt werden. Dabei sind die C++-Anwendungen um einen Faktor von ca. 1,5 langsamer als das schnellste C-Programm.

Während die JAVAC-Anwendungen auf dem Windows-System Laufzeiten in Mittelfeld erzielen konnten, so weichen diese auf dem Linux-System in einem größeren Verhältnis ab. Die dabei erzielten Laufzeiten sind erheblich langsamer als die der C/C++-Applikationen. Der Unterschied zu den entsprechend kürzesten Laufzeiten der C-Anwendungen kann abhängig von der Problemgröße bzw. Optimierung, mit einem Faktor von ca. 3-4 bemessen werden. Eine weitere Auffälligkeit besteht zwischen der unoptimierten und optimierten JAVAC-Applikation. So sind für alle Dateien die geringeren Laufzeiten bei der unoptimierten Variante zu finden. Der Verlauf von gemessenen Werten erfolgt jeweils für C/C++ und Java (unoptimiert, optimiert) in den folgenden zwei Diagrammen. Die Problemgröße ist in diesen Darstellungen die kleinste Datei mit einer Größe von 150 MB.

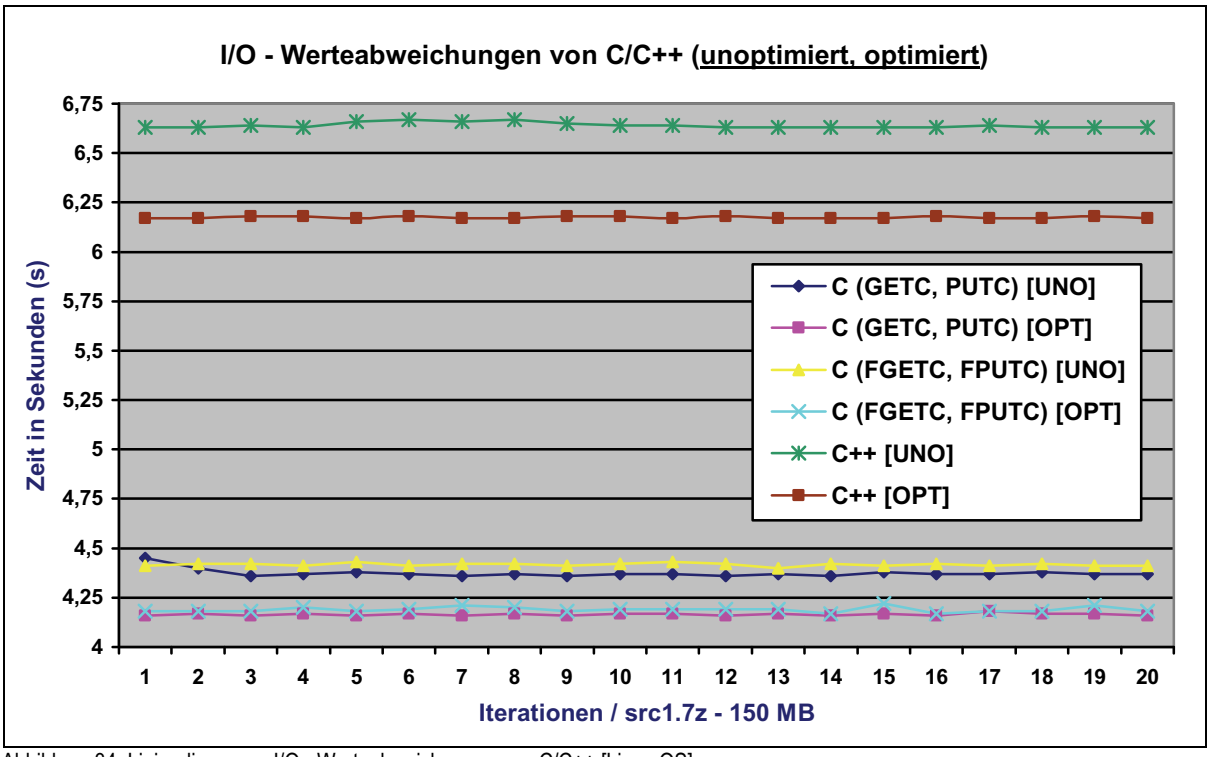


Abbildung 34: Liniendiagramm I/O - Werteabweichungen von C/C++ [Linux OS]

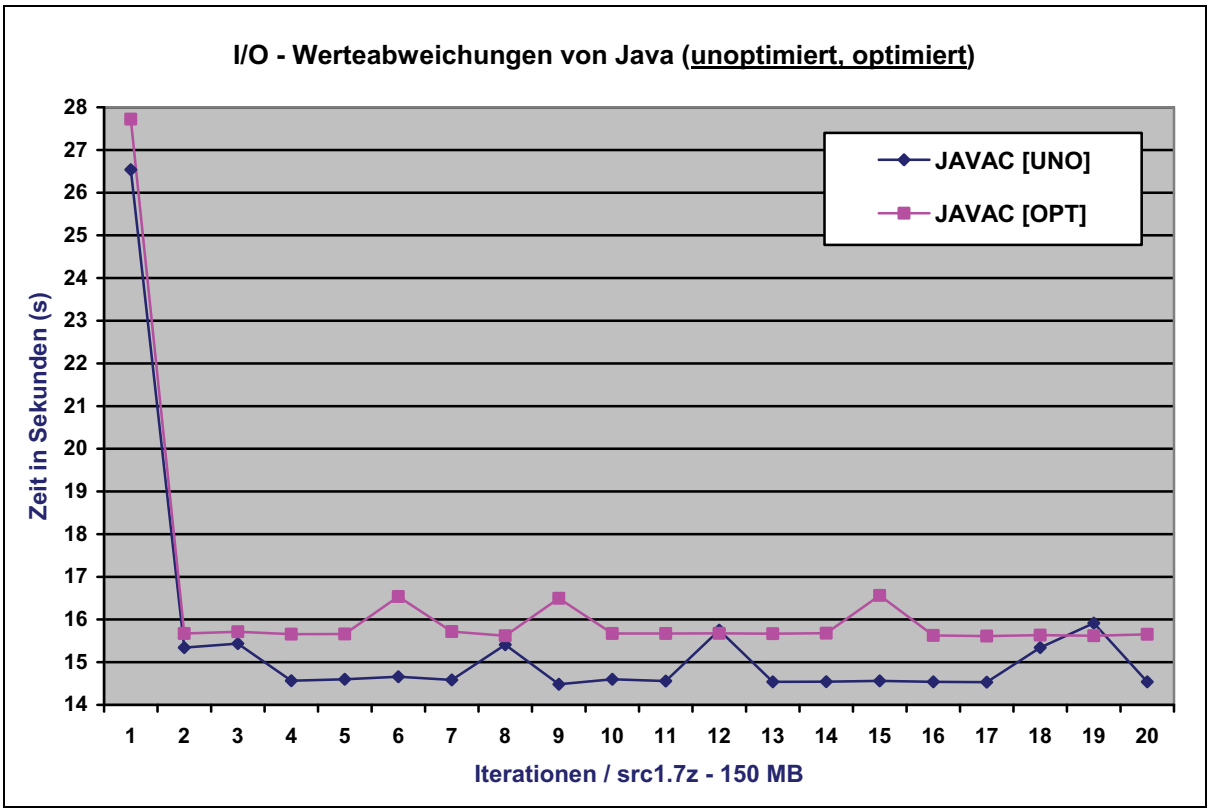


Abbildung 35: Liniendiagramm I/O - Werteabweichungen von Java [Linux OS]

Für alle C- sowie C++-Anwendungen kann im oberen Diagramm (Abbildung 34) ein annähernd linearer Verlauf der gemessenen Werte beobachtet werden. Im Gegensatz dazu sind für beide Java-Anwendungen (Abbildung 35) bei der ersten Ausführung erhebliche Abweichungen aufgetreten. Der Werteverlauf der Datenreihen zeigt, dass die Maximalwerte im Verlauf in einen Bereich von ca. einer Sekunde schwanken.

5.3.3 Windows vs. Linux

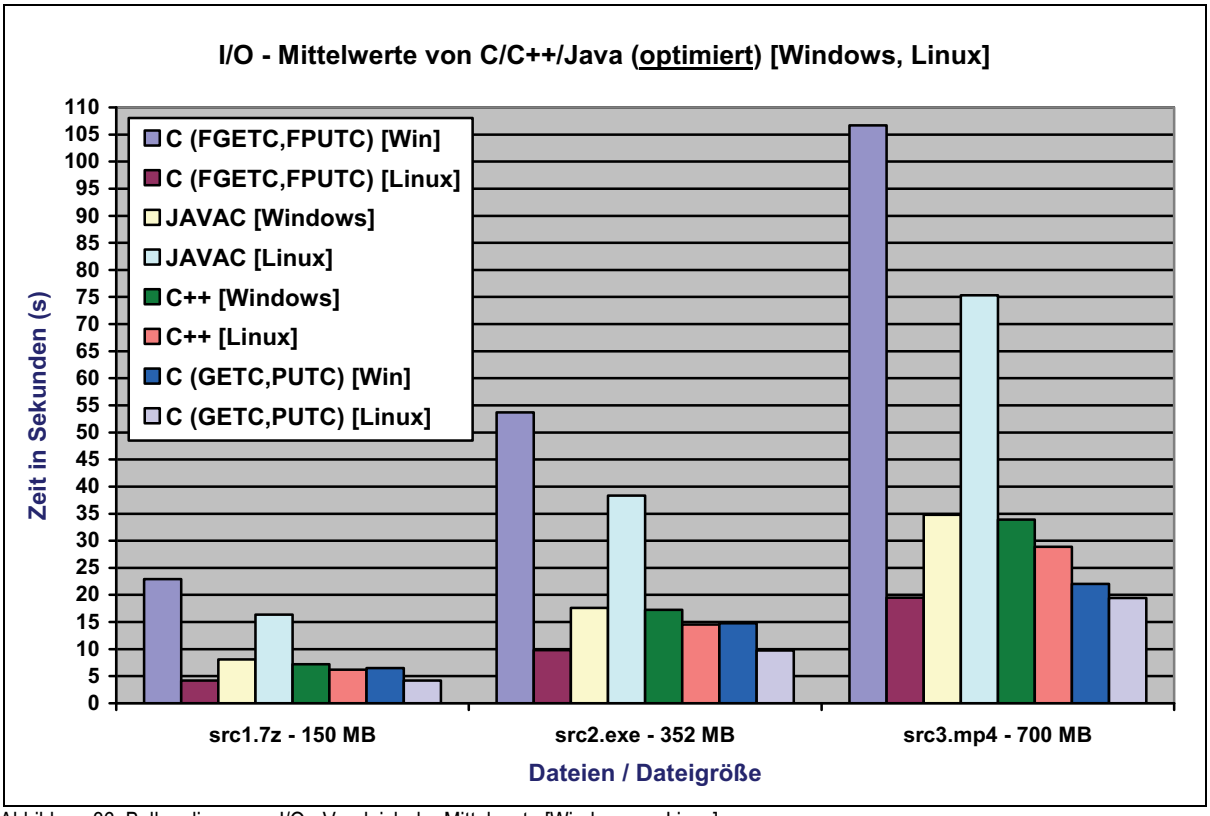


Abbildung 36: Balkendiagramm I/O - Vergleich der Mittelwerte [Windows vs. Linux]

Tabelle I/O - Vergleich der Mittelwerte von C/C++/Java (optimiert) [Zeit in Sekunden]

Sprache	Betriebssystem	Besonderheit	src1.7z - 150 MB	src2.exe - 352 MB	src3.exe - 700 MB
C	**Windows**	FGETC,FPUTC	22,91405	53,68285	106,6946
C	**Linux**	FGETC,FPUTC	4,1885	9,822	19,489
Java	**Windows**	JAVAC	8,09305	17,61735	34,82345
Java	**Linux**	JAVAC	16,3934	38,3252	75,32675
C++	**Windows**	-	7,2054	17,2219	33,903
C++	**Linux**	-	6,174	14,506	28,875
C	**Windows**	GETC,PUTC	6,4726	14,768	22,03895
C	**Linux**	GETC,PUTC	4,166	9,7665	19,4445

Tabelle 26: Vergleich I/O - Mittelwerte [Windows vs. Linux]

Der Vergleich der Laufzeiten für Windows und Linux zeigt, dass die C/C++-Anwendungen die beste Effizienz unter Linux erzielen. Abhängig von der Dateigröße beträgt die Differenz zu den Windows-Laufzeiten mehrere Sekunden. Die höchste Abweichung besteht für die C-Anwendung mit den Funktionen `fgetc()`, `fputc()`. Für die größte Datei (700 MB) kann die Linux-Variante eine um den Faktor 5,47 bessere Laufzeit als das Windows-Pendant aufweisen.

Konträr dazu verhalten sich die Laufzeiten der JAVAC-Applikationen. Hierbei sind die kürzeren Laufzeiten auf dem Windows-Betriebssytem gemessen worden. Ein Abstand der Laufzeiten bzgl. der größten Datei kann mit einem Faktor von 2,16 angegeben werden.

Kapitel 6

6 Zusammenfassung und Ausblick

Für eine abschließende Betrachtung finden sich in diesem Kapitel Zusammenfassungen von Laufzeitergebnissen der Bereiche Prozessorauslastung bzw. Rechenlast, Speicherverwaltung, Ein- und Ausgabeoperationen (I/O-Operationen) bzgl. der Dateiarbeit. Weitere Bewertungen, Erläuterungen zu Ursachen sowie ein Ausblick über weitere relevante Aspekte sollen einen thematischen Abschluss bilden.

6.1 Ergebnisübersicht

Zusammenfassend soll ein Überblick über einige ermittelte Werte gegeben werden. Die im Weiteren aufgeführten Tabellen zeigen je Betriebssystem, in der ersten Zeile die im Durchschnitt benötigte Zeit (in Sekunden) für die gesamte Programmausführung. Hierbei wurden alle berechneten Mittelwerte der Problemgrößen addiert. Die daraus resultierenden hohen Zeiten ermögliche eine bessere Darstellung hinsichtlich der Laufzeitdifferenzen.

In der zweiten Spalte wurde die Zeit der unoptimierten C-Anwendung als Referenz verwendet (auf "1" gesetzt), alle anderen Zeiten wurden durch diese dividiert. Die hierdurch berechneten Faktoren geben an, um wie viel die Anwendungen schneller bzw. langsamer, als die unoptimierte C-Applikation sind.

CPU-Auslastung/Rechenlast - Gesamtvergleich [Zeit in Sekunden][6]									
Wert	**System**	**C [UNO / OPT]**		**C++ [UNO / OPT]**		**JAVAC [UNO / OPT]**		**JIKES [UNO / OPT]**	
Summe	Windows	401,675	261,402	402,486	259,547	358,388	359,183	397,042	397,302
Faktor	Windows	1	1,537	0,998	1,548	1,121	1,118	1,012	1,011
Summe	Linux	390,786	240,457	390,570	240,634	311,921	322,035	-	-
Faktor	Linux	1	1,625	1,001	1,624	1,253	1,213	-	-

Tabelle 27: Gesamtvergleich CPU - Summe der Mittelwerte / Faktoren

Speicherverwaltung - Gesamtvergleich [Zeit in Sekunden][7]									
Wert	**System**	**C [UNO / OPT]**		**C++ [UNO / OPT]**		**JAVAC [UNO / OPT]**		**JIKES [UNO / OPT]**	
Summe	Windows	28,545	31,961	28,252	30,122	22,098	22,144	20,463	19,799
Faktor	Windows	1	0,893	1,010	0,948	1,292	1,289	1,395	1,442
Summe	Linux	7,264	10,873	11,534	8,881	21,890	19,887	-	-
Faktor	Linux	1	0,668	0,630	0,818	0,332	0,365	-	-

Tabelle 28: Gesamtvergleich RAM - Summe der Mittelwerte / Faktoren

I/O-Operationen / Dateiarbeit - Gesamtvergleich [Zeit in Sekunden][8]											
Wert	**System**	**C (GETC, PUTC) [UNO / OPT]**		**C (FGETC, FPUTC) [UNO / OPT]**		**C++ [UNO / OPT]**		**JAVAC [UNO / OPT]**		**JIKES [UNO / OPT]**	
Summe	Windows	185,24	43,28	185,02	183,29	61,16	58,33	60,51	60,53	65,25	65,44
Faktor	Windows	1	4,28	1	1,01	3,03	3,18	3,06	3,06	2,84	2,83
Summe	Linux	35,06	33,38	35,40	33,50	53,16	49,56	120,17	130,05	-	-
Faktor	Linux	1	1,05	0,99	1,05	0,66	0,71	0,29	0,27	-	-

Tabelle 29: Gesamtvergleich I/O - Summe der Mittelwerte / Faktoren

[6] Werte gerundet auf die 3. Nachkommastelle
[7] Werte gerundet auf die 3. Nachkommastelle
[8] Werte gerundet auf die 2. Nachkommastelle

Der Vergleich für den **Bereich CPU-Auslastung** zeigt, dass die verwendete Optimierungsoption des C/C++-Compilers unabhängig vom Betriebssystem, einen signifikanten Einfluss auf die Laufzeit bewirkt. In diesem Fallbeispiel liegt der Faktor für Windows-Anwendungen bei mehr als 1,5 und für Linux bei über 1,6. Für die Java-Applikationen hat der entsprechende Optimierungsschalter zu keiner Messung feststellbare Performanceverbesserungen gezeigt. Im Gegensatz zu den C-/C++-Applikationen sind minimal erhöhte Laufzeiten zu verzeichnen. Weiterhin weisen alle, auf dem Linux-Betriebssystem ausgeführten Anwendungen, in der Summe eine bessere Performance als die gleichen Windows-Anwendungen auf.

Für den **Bereich Speicherverwaltung** sind im Vergleich zum CPU-Bereich größere Abweichungen zwischen den Betriebssystemen vorhanden. Des Weiteren verhalten sich die optimierten RAM-Anwendungen verschieden zu denen aus dem Bereich CPU. So ist zwar für die optimierte C++-Anwendung auf dem Linux-System eine bessere Performance als für die unoptimierte erzielt worden, aber im Gegensatz zu den anderen optimierten C/C++-Programmen, sind signifikant schlechtere Laufzeiten gemessen worden. Ein weitestgehend konträres Verhalten kann auch für die Java-Applikationen beobachtet werden. So ist lediglich die optimierte JAVAC-Variante auf dem Windows-System geringfügig langsamer als die unoptimierte Anwendung. Bei den weiteren Java-Anwendungen sind die optimierten Programme, jeweils schneller als die entsprechenden unoptimierten.

Für den **Bereich I/O** kann wieder ein gleiches Verhalten wie für den Bereich CPU hinsichtlich der optimierten/unoptimierten Applikationen festgestellt werden. Die größten Differenzen traten für diesen Bereich bei der C-Anwendung mit den Makros `getc()`, `putc()` auf (vgl. "5.3.1 Messung 1 - Windows OS"). Für die sonstigen Anwendungen der Sprachen C/C++ sind dagegen geringere Abweichungen vorhanden. Durch die Optimierung der verschiedenen Java-Applikationen konnte für diesen Bereich kein gewinnbringender Effekt bzgl. der Laufzeiten erreicht werden. Die unter Linux ausgeführte optimierte JAVAC-Variante zeigte sogar eine deutlich schlechtere Performance als die unoptimierte.

6.2 Ursachen

Im weiteren Verlauf sollen Ursachen für die in Punkt 5 (Ergebnisse) dargestellten Laufzeitdifferenzen benannt und weitestgehend erläutert werden. Dabei ist u. a. eine Betrachtung der Grundlagen von entscheidender Relevanz (Punkt 2). Alle gemessenen Werte können der beigefügten CD-ROM bzw. ein Auszug aus entsprechenden Tabellen (Punkt 5 und 6.1) entnommen werden.

6.2.1 Implementierung / Konzepte

Die aufgetretenen Laufzeitunterschiede zwischen C/C++ und Java ergeben sich in erster Linie aus den jeweiligen Implementierungen der Algorithmen. Weiterhin gibt es verschiedene Gründe, wie eingangs erläutert, spielt das Konzept der Sprachübersetzung eine entscheidende Rolle. Dabei ist zum einen der durch den Compiler erzeugte Maschinencode für C/C++ sofort ausführbar. Im Gegensatz zu Java erzeugt ein Compiler lediglich Byte-Code, welcher erst durch einen Interpreter ausgeführt werden kann (vgl. "JIT- bzw. AOT-Konzept"). So kann der Interpreter-Aspekt einen entscheidenden Einfluss auf die Laufzeitunterschiede bewirkt haben.

Weiterhin ist die Verwendung von unterschiedlichen Compilern für ein Teil der Differenzen verantwortlich. Diese Annahme lässt sich aus dem Laufzeitvergleichen zwischen JAVAC und JIKES begründen. So ist jeweils die Performance der JAVAC-Anwendungen (für z. B. Punkt "5.1.1 Messung 1 - Windows OS") besser als die der JIKES-Anwendungen. Dass auch das Gegenteil der Fall sein kann, wird in Punkt "5.2.1 Messung 1 - Windows OS" dargestellt. Bei diesen Messungen konnten die JIKES-Programme sichtbar kürzere Laufzeiten vorweisen. Aus diesen Verhalten kann darauf geschlossen werden, dass die Reihenfolge der Ausführung keinen entscheidenden Einfluss auf die Laufzeiten der Anwendungen bewirkt hat. Für die Differenzen von Programmen der Sprachen C/C++ zwischen Windows und Linux könnten somit auch die unterschiedlichen Compiler verantwortlich sein.

Da für C/C++ im Gesamtvergleich meist kürzere Laufzeiten als für Java gemessen wurden, lässt darauf schließen, dass auch der Aspekt der hardwarenahen Programmierung eine Rolle spielt. So wurde wie bereits erwähnt, C speziell mit dem Ziel entwickelt eine Abstraktion für die Assemblersprache zu schaffen.

6.2.2 Betriebssystem und Laufzeitumgebung

Weiterhin können nach [PR 97] Leistungsverluste durch das zugrunde liegende **Betriebssystem** hervorgerufen werden. So ist es möglich, dass für z. B. Synchronisationsaufgaben, weitere Berechnungen und Aufrufe von Betriebssystemroutinen ausgelöst werden. Bei Betrachtung der Laufzeiten von C/C++-Anwendungen zwischen Windows und Linux fällt auf, dass unter Linux zum Teil erheblich kürzere Laufzeiten gemessen wurden als unter Windows. Diese Differenzen können durch die enge Anlehnung der Sprachen C/C++ an UNIX-artigen Systemen bedingt sein.

Bei Betrachtung der Mittelwerte der verschiedenen Sprachen und Bereiche fällt auf, dass C und C++ eine stetige Kontinuität im Verlauf vorweisen. Diese ist unabhängig vom verwendeten Betriebssystem sowie unterschiedlichen Problemgrößen zu beobachten. Das Ergebnis dieser Entwicklung lässt darauf schließen, dass die Betriebssysteme keinen signifikanten Einfluss auf das Laufzeitverhalten dieser Anwendungen bewirkt haben. Für Java hingegen kann keine solche Beständigkeit in der Werteentwicklung festgestellt werden. Da eine Einwirkung aufseiten des zugrunde liegenden Betriebssystems ausgeschlossen werden kann, sind hierfür anderen Ursachen denkbar.

Das bedeutet wiederum das diese Unstetigkeit durch die ausführende **Laufzeitumgebung** hervorgerufen wird. Beispielsweise sind beeinflussende Vorgänge der Java-VM für einen maßgeblichen Einfluss auf das Laufzeitverhalten die Ursache. Für die temporären Abweichungen im Bereich CPU, könnten so spezielle Sicherheitsmechanismen verantwortlich sein, die zu teilweise erheblich höheren Laufzeiten geführt haben (s. auch Punkt "2.6.2 Software-Faktoren" / Laufzeitumgebung). Weiterhin hat die Optimierung von Java-Anwendungen nur selten eine bessere Performance zur Folge gehabt. Hiefür können optimierende Vorgänge der Java-VM verantwortlich sein, was im Endeffekt bedeutet, dass eine Optimierung durch den Compiler nicht unbedingt nötig ist.

Durch den Einsatz der Laufzeitumgebung und der dadurch resultierenden Beeinflussung kann keine genaue Voraussage hinsichtlich des Laufzeitverhaltens differierender Problemgrößen und -bereiche getroffen werden. Vielmehr ist das Verhalten einiger Vorgänge (z. B. Berechnungen) von sicherheitsrelevanten Aspekten sowie Optimierungsvorgängen abhängig.

[Punkt 6.2.2 nach Quelle: PR 97]

6.2.3 Hardware

Ein Großteil der Laufzeitunterschiede kann durch die Verwendung von verschiedenen Implementierungen, Konzepten und zugrunde liegender Software erklärt werden. Neben diesen Ursachen ist auch die Hardware für einige Aspekte von Bedeutung. So ist besonders für den Bereich Dateiarbeit (Punkt 5.3 I/O-Operationen / Dateiarbeit) die Festplatte für Werteabweichungen verantwortlich. Gegenüber anderen Bereichen kam es im Verlauf immer wieder zu temporären Sprüngen, welche möglicherweise durch **Caching-Effekte** hervorgerufen wurden. Hiebei waren auf dem Windows-System mehr die optimierten als die unoptimierten C/C++-Programme betroffen. Für Java kam es lediglich bei der kleinsten Datei (150 MB) und beim ersten Durchlauf zu einer deutlichen Abweichung (Abbildung 32, Abbildung 35). Dadurch kann neben der Festplatte auch die Laufzeitumgebung als beeinflussender Faktor genannt werden. So kann beispielsweise ein Programm, aufgrund von Initialisierungs- / Synchronisationsprozessen beim ersten Start, mehr Zeit benötigen als bei folgenden Ausführungen.

Unter Linux sind für die C/C++-Anwendungen (unoptimiert, optimiert) jedoch keine Werteschwankungen aufgetreten. Für jede Datei und Iteration wurden weitestgehend äquivalente Laufzeiten gemessen. Verschieden dazu zeigen die Java-Anwendungen im Verlauf der Wiederholungen ein weniger lineares Verhalten. Neben diesen Differenzen wurden auch wieder für die kleinste Datei und bei der ersten Wiederholung erheblich höhere Laufzeiten gemessen. Die unterschiedlichen Differenzen im Verlauf zwischen den Betriebssystemen zeigen, dass neben den Hardware-Faktoren auch Effekte des Betriebssystems eine wesentliche Rolle spielen (s. auch Punkt "2.6.2 Software-Faktoren" / Betriebssystem).

6.2.4 Methoden zur Laufzeitbestimmung

Die Ursachen für **Werteabweichungen** bei Programmstart (Bereich CPU) sind bei den verwendeten Methoden zur Laufzeitmessung zu finden. Hierbei zeigen alle Programme die größten Unterschiede bei den kleinsten Problemgrößen auf und nehmen bei zunehmender Komplexität ab. Abhängig von der Dauer der Operationen gehen diese Abweichungen gegen den Mittelwert, sodass keine signifikanten Differenzen mehr festzustellen sind. Aufgrund der geringen Genauigkeit (1/100 ms) sind die sprachenspezifischen Methoden nicht für Operationen geeignet, die für die Ausführung nur einige Millisekunden oder weniger in Anspruch nehmen. Diese Annahme wird durch die gemessenen Werte begründet, welche z. B. für Windows bei 100 E/Dim entweder 0 oder 0,015 bzw. 0,016 Sekunden betragen. Zwischen diesen Werten existieren für alle Anwendungen der Sprachen C/C++ und Java keine anderen Laufzeiten. Jene Werte können auch bei folgenden Problemgrößen solange beobachtet werden, bis die Operationen einige 1/10 Millisekunden Zeit in Anspruch nehmen.

Diese Diskrepanz ist auch unter Linux nachzuvollziehen, jedoch betragen die Werte an dieser Stelle, entweder 0 oder 0,010 Sekunden. Dass die optimierten Anwendungen weit größere Abweichungen aufzeigen, liegt in der noch kürzeren Ausführungsdauer begründet. Besonders für C/C++ hat die durch den Compiler vorgenommene Optimierung eine meist bessere Performance vorgebracht, dies spiegelt sich in der noch schlechteren Messbarkeit wider. Bei Betrachtung der Messergebnisse fällt auf, dass für Windows (unoptimiert) bei 100 E/Dim relativ oft, die Laufzeiten 0,015 bzw. 0,016 Sekunden gemessen wurden. Bei den optimierten Anwendungen besitzt die Mehrheit der gemessenen Laufzeiten den Wert 0 (Sekunden), gleiches gilt auch wieder für Linux. Aus diesen Beispielen wird ersichtlich, dass Funktionen die Laufzeiten im Bereich von Mikro- oder Nanosekundenbereich liefern, zweckmäßiger für Berechnungen wären, die nur sehr wenig Programmzeit verbrauchen. Für die Werteabweichungen im Bereich Speicherverwaltung sind grundsätzlich die gleichen Ursachen, wie für den Bereich CPU verantwortlich. Jedoch sind die Abweichungen der Java-Anwendungen auf andere Ursachen (wie z. B. Einflussfaktoren der JRE/JVM) zurückzuführen.

6.3 Bewertung

Für eine abschließende Bewertung möchte ich auf die in der Einleitung (Punkt 1.2 Motivation) aufgeführten Zitate zurückkommen. So bezog sich die erste Aussage darauf, dass Java viel langsamer als C/C++ sei. Aufgrund der durchgeführten Laufzeitmessungen ergibt sich, dass diese Aussage nicht korrekt ist. Beispielsweise waren die Java-Anwendungen für den Bereich Speicherverwaltung unter Windows (s. Punkt 5.2.1), die mit der besten Effizienz.

Bei der Aussage das Java um den Faktor 2 langsamer als C sei, handelt es sich wieder um eine sehr allgemeine Behauptung, die für diese Fallbeispiele nicht gültig ist. So waren unter Windows unoptimierte C-, aber auch C++-Anwendungen, meist den entsprechenden Java-Anwendungen unterlegen. Andere Auffassungen, die Java anhand von Weiterentwicklungen wie JIT-Konzepten und zahlreichen Optimierungen gleich schnell oder gar schneller als C/C++-Applikationen sehen, kann nur bedingt zugestimmt werden. Zwar konnten die Java-Anwendungen (Bereich RAM) für Windows kürzere Laufzeiten vorweisen, unter Linux war jedoch genau das Gegenteil der Fall (besonders bei steigender Komplexität).

Zusammenfassend lässt sich sagen, dass keine allgemeingültigen Behauptungen zum Laufzeitverhalten von Applikationen unterschiedlicher Sprachen getroffen werden können. Vielmehr ist Effizienz von vielen unterschiedlichen Faktoren (vgl. Punkt "2 Grundlagen") abhängig. Für die umgesetzten Fallbeispiele konnte deshalb auch keine Sprache, durchgehend die Applikationen mit den kürzesten Laufzeiten vorweisen. In der Summe kann jedoch besonders für das Linux-Betriebsystem eine bessere Performance zugunsten der Programmiersprachen C und C++ festgestellt werden. Gleichermaßen haben die Laufzeitvergleiche auch gezeigt, dass Java bei Weitem nicht so langsam ist wie fälschlicherweise angenommen. Abhängig vom Bereich sowie weiteren Aspekten ist der Faktor (deutlich) geringer als 2. Nichtsdestotrotz sind Wissen und Kenntnisstand eines Softwareentwicklers entscheidend um Probleme so effizient wie möglich zu lösen. Denn uneffizienter Quellcode kann einen deutlich höheren Faktor als 2 aufweisen.

[Punkt 6.3 nach Quellen: FOR 01, PR 97]

6.4 Ausblick

Aufgrund des Umfangs dieser Arbeit konnten keine weitläufigen Aspekte hinsichtlich weiterer Laufzeitvergleiche umgesetzt werden. Die folgenden Punkte sollen einige wesentliche Kriterien aufzeigen, die nicht oder nicht ausreichend berücksichtigt werden konnten.

Ausblick über weitere relevante Aspekte für Laufzeitvergleiche (Beispiele)	
Bereich	**Beschreibung**
Betriebssystem	- bei der Verwendung von Betriebssystemen kam lediglich "Windows XP" und für Linux "Ubuntu" zum Einsatz, aufgrund der Verbreitung einer Vielzahl unterschiedlicher Betriebssysteme sowie Arten können somit nur eingeschränkte Aussagen getroffen werden - für weitere Durchführungen wären z. B. aktuelle, verteilte, Desktop-/Server-Systeme, unterschiedliche Distributionen sowie Live-CD/-DVD/-USB-Varianten von zusätzlicher Bedeutung
Dateiarbeit	- für den Bereich I/O wurden nur Funktionen zum zeichenweisen lesen und schreiben verwendet, die unterschiedlichen Sprachen bieten jedoch einen größeren Umfang an entsprechenden Funktionen - das Nutzen dieser Funktionen wäre für eine umfassende Betrachtung von entscheidender Relevanz
Felder/Arrays	- für C/C++-Programme wurde mehrdimensionale Matrizen als Felder von Feldern abgebildet, diese programmtechnische Abbildung entspricht dabei der von Java - für eine weitere Betrachtung können mehrdimensionale Felder in C und C++, anstatt der implementierten Abbildung, auch auf einen eindimensionalen Vektor abgebildet werden - neben der Abbildungsart von mehrdimensionalen Feldern wäre für C/C++ eine Unterscheidung zwischen dynamischen und statischen Feldern in Betracht zu ziehen
Netzwerk	- in dieser Arbeit beschränkten sich die I/O-Operationen auf die Dateiarbeit eines lokalen PCs, dabei wurde der Aspekt der Netzwerkprogrammierung nicht berücksichtigt - weitere Messungen hinsichtlich I/O-Durchsatz, Datenübertragung könnten im Netzwerk bzw. Internet durchgeführt werden
Parallelisierung	- aufgrund der stetigen Hardwareentwicklung wären für weitere Laufzeitvergleiche entsprechende Applikationen zu entwickeln, die das Potenzial dieser besser nutzen - so wäre z. B. für Multicore-Prozessoren eine Aufgabenverteilung der rechenintensiven Operation(en) für die Anzahl verschiedener Kerne zu entwickeln
Prozess-interaktion	- alle Messungen wurden auf einem Testsystem durchgeführt, bei dem störende Prozesse (z. B. Viren-Scanner, Firewall, Update-Funktionen, Index-Dienste und ähnliche) weitestgehend deaktiviert wurden - für die Fragestellung: "Wie Verhalten Applikationen auf einem normalen PC?" wären Laufzeitmessungen mit eben diesen Prozessen durchzuführen
Speicher	- der laufzeitrelevante Aspekt bezog sich für den Bereich Speicherverwaltung auf die Allokation von Speicher, weitere Messungen könnten z. B. für die Freigabe von Speicher sowie einer Kombination dieser Aufgaben durchgeführt werden
Sprachen	- das Thema dieser Arbeit beschränkte auf die Programmiersprachen auf C/C++ und Java - für weitläufige Untersuchungen wäre eine Verwendung von weiteren aktuellen bzw. weitverbreiteten Sprachen zweckmäßig (z. B. C#, Delphi, PHP, (Visual) Basic)
Sprachüber-setzung	- die verwendeten Compiler stellen lediglich einen kleinen Ausschnitt von verfügbaren Mitteln dar - so könnte die Sprachübersetzung mit weiteren Compilern erfolgen, wobei eine Unterscheidung zwischen freien und kommerziellen Compilern getroffen werden könnte - speziell für Java wären sogenannte »AOT-Compiler« von besonderem Interesse, da hierbei direkt Maschinencode erzeugt wird und somit eine bessere Performance zu erwarten ist

Tabelle 30: Ausblick über weitere relevante Aspekte für Laufzeitvergleiche (Beispiele)

Für die aufgeführten Bereiche sind aufgrund von z. B. vielschichtigeren und verteilten Anwendungen, unter Umständen komplexere Methoden zur Laufzeitbestimmung nötig und andere Ergebnisse zu erwarten. Grundsätzlich lässt sich sagen, dass eine Vielzahl von unterschiedlichen Faktoren berücksichtigt werden muss, um möglichst genaue Aussagen geben zu können.

Die Untersuchungen haben jedoch gezeigt, dass es sich lohnen kann, für Anwendungen auf bestimmte Sprachen zurückzugreifen. Hierdurch ist eine Einsparung von Ressourcen sowie eine Steigerung der Effizienz möglich, wobei grundsätzlich eine Abwägung zwischen den jeweiligen Vor- und Nachteilen einer Programmiersprache erfolgen sollte.

7 Informationsglossar

Abstraktion Das Ziel von **Abstraktion** besteht generell darin, komplexe / detaillierte Situationen, welche u. U. eine Verkomplizierung zur Folge haben, zu vereinfachen. Hierbei können beispielsweise komplexe Konzepte durch verständliche Modelle ersetzt werden, sodass eine Problemlösung ohne spezielles Wissen von Einzelheiten möglich ist. Mit Einzelheiten werden dabei Auswirkungen oder weitere Informationen bezeichnet, die nicht zwingend zur Problemlösung benötigt werden. [AU 96]

Allokation **Allokation** bzw. Allozierung bezeichnet allgemein den Vorgang bei dem Ressourcen für Anwenderprogramme reserviert werden. Im Zusammenhang dieser Arbeit wird der Begriff für die Reservierung von Hauptspeicher bzw. für das Ausfassen von Speicher verwendet. [MSP 97]

Array Als **Array** wird in der Programmierung eine Liste von Elementen gleichen Datentyps bezeichnet. Dabei können die in dieser Arbeit verwendeten Begriffe: Feld, Liste, Matrix als Synonym aufgefasst werden. [MSP 97]

Assembler Bei **Assemblersprache** handelt es sich um eine niedrige, Systemnahe Programmiersprache. Diese verwendet Abkürzungen bzw. Codes, wobei jeder Code einem Maschinenbefehl entspricht. Für die Umwandlung von Assemblercode in Maschinencode ist ein sogenannter **Assembler** bzw. Assemblierer zuständig. Dabei wird der Vorgang der Umwandlung als **assemblieren** bezeichnet. [MSP 97]

Ausnahmen "Formulierung von Alternativen der Programmausführung, um nach einem Fehler, z. B. Division durch 0, versuchter Zugriff auf eine nicht vorhandene Datei, usw., wieder einen Zustand zu erreichen, in dem die Programmausführung ordnungsgemäß fortgesetzt und zu Ende geführt werden kann." [PR 97]

Batchverarbeitung Die Batchverarbeitung (auch als Stapelverarbeitung bezeichnet) beschreibt grundsätzlich die Abarbeitung einer **Batch**- bzw. **Stapeldatei**. Eine solche Datei (Textdatei) enthält dabei eine Folge von Befehlen, die abhängig von der Sprache durch weitere Operatoren und Parameter ergänzt werden kann. Bei der Ausführung einer Batch-Datei erfolgt grundsätzlich eine sequenzielle Abarbeitung der darin enthaltenen Befehle. [MSP 97]

Binder → s. »Linker«

Byte-Code Ist ein codiertes Programm, welches der Compiler während der Sprachübersetzung aus Quellcode erzeugt. Dabei ist die Form der Codierung meist abstrakt und systemunabhängig, was wiederum zur Folge hat, das solche Programme nicht direkt auf einer Maschine ausgeführt werden können. Dieses Prinzip der Kompilierung wird durch die meisten Java-Compiler realisiert, für die Ausführung der Anwendungen wird ein darüber hinaus weitere Software benötigt, z. B. in Form eines Interpreters. [MSP 97]

Clocks Einheit, die für vergangene Prozessor-/CPU-Zeit verwendet wird. Die Anzahl der Clocks pro Sekunde ist dabei abhängig von der zugrunde liegenden Maschine. Der Begriff **Clocks** kann auch als "Uhr-Ticks" oder "CPU-Ticks" bezeichnet werden. [TUC 01]

Debugger Ein Programm das die Fehlersuche in einem anderen Programm ermöglicht. Der Programmierer kann mithilfe des **Debuggers** eine schrittweise Abarbeitung des Programms, das Überprüfen von Daten sowie das Testen von Bedingungen (z. B. Variablenwerte/-zuweisungen) durchführen. [MSP 97]

Determinismus	Beschreibt die Fähigkeit, Ausgaben zu prognostizieren oder vorher zu wissen, wie ein System Daten verarbeitet bzw. manipuliert. [MSP 97]
dezentral	Fähigkeit, Informationen und die Last für die Berechnung von Daten zu verteilen (wesentliches Charakteristikum von Client/Server-Anwendungen). [STE 01]
Direktive	Anweisung, Befehl, Operation
Distribution	**Distributionen** sind meist vorkonfigurierte Betriebssysteme, welche auf Linux basieren und weitere freie Software enthalten. [WIS 06]
Flag	Unter dem Begriff **Flag**, kann allgemein eine Markierung verstanden werden, die bei einer weiteren Verarbeitung oder Interpretation von Informationen benötigt wird. [MSP 97]
Funktion	In Programmiersprachen werden die Begriffe **Funktion**, Methode oder Routine für ein Unterprogramm verwendet, welches einen Wert zurückliefert. [MSP 97]
Garbage Collector	Der »**Garbage Collector**« ist ein Bestandteil des Java-Interpreters und hat die Aufgabe zur Laufzeit festzustellen, wie viel Speicher ein Objekt benötigt, diesen zu allozieren und ggf. nicht mehr benötigten Speicher freizugeben. [DPT 01]
generisch	Die **generische** Programmierung ist ein Verfahren zur Entwicklung wiederverwendbarer Software-Bibliotheken. Dabei werden Funktionen möglichst allgemein entworfen, um für unterschiedliche Datentypen und Datenstrukturen verwendet werden zu können. [WIKI 03]
heuristisch	Der Begriff heuristisch (von **Heuristik**, griech: "finden", "entdecken") kann Methoden bzw. Algorithmen beschreiben, die durch selbstlernende oder nichtdeduktive Techniken eine korrekte Lösung eines Problems liefern. [MSP 97]
imperativ	Das wichtigste Merkmal einer **imperativen** Sprache ist eine Folge von Anweisungen (z. B. Befehle oder Kommandos). Diese Art einer Programmiersprache besitzt drei grundlegende Eigenschaften: 1. Sequenzielle Ausführung von Instruktionen, 2. Verwendung von Variablen (stellen Speicherwerte dar) und 3. Verwendung von Zuweisungen um Werte von Variablen zu ändern. [LOU 94]
Iteration	Steht im Zusammenhang dieser Arbeit für die Wiederholungen von bestimmten Vorgängen, z. B. die Wiederholung einer Berechnungsoperation.
JRE	Abkürzung für **Java Runtime Environment** (Java Laufzeitumgebung) und bezeichnet Software, die zur Ausführung von Java-Anwendungen benötigt wird. (s. auch »JVM«)
JVM	Die JVM (**Java Virtual Machine** / Java-VM, virtuelle Java Maschine) ist ein Bestandteil der Java-Laufzeitumgebung (JRE) und für die Ausführung von Java-Byte-Code zuständig. Dabei kann die Java-VM als Schnittstelle zwischen dem Java-Byte-Code und der Maschine (Hard- und Software) aufgefasst werden. [KRÖ 06]
Kernel	Stellt den Kern eines Betriebssystems dar, der u. a. für das Starten von Applikationen, die Verwaltung von Systemressourcen, Zeit, Datum sowie der Behandlung von Speicher, Dateien und der Peripherie zuständig ist. [MSP 97]

Kompilierung Beschreibt den Prozess der Übersetzung eines Programms von Quellcode in ausführbaren Code (Objektcode bzw. Maschinencode). Abhängig vom zugrunde liegenden Quellcode existieren eine Reihe von verschiedenen Programmen, die für die Sprachübersetzung zuständig sind. Diese werden als **Compiler** (Kompilierer) bzw. Übersetzer bezeichnet. [MSP 97]

Kommandozeile → Konsole, Terminal, s. auch »Shell«

Linker Ein **Linker** (auch als **Binder** bezeichnet) ist ein Programm, dass Datendateien und kompilierte Module zusammenfügt. Ziel hierbei ist die Erzeugung eines ausführbaren Programms. Daneben kann ein Linker auch weitere Aufgaben übernehmen, wie z. B. das Erstellen von Bibliotheken. [MSP 97]

Makro In Programmen können **Makros** als eine Folge von Befehlen aufgefasst werden, welche durch ein Programm ausgeführt werden. Im Wesentlichen gleichen Makros Funktionen, während der **Kompilierung** werden jedoch Makros durch die Anweisungen bzw. Befehle ersetzt, die diese definieren. Im Gegensatz zu Funktionen, ermöglichen Makros eine Reduzierung der Laufzeit, da u. a. Sprünge in und aus Funktionen entfallen. [MSP 97]

Maschinensprache Ist das Ergebnis der **Kompilierung** von Quellcode in ausführbaren Code. Der dadurch resultierende ausführbare Code wird als **Maschinensprache** bzw. Maschinencode bezeichnet, da dieser Code der Einzige ist, welcher durch die Maschine / das System ausgeführt bzw. verarbeitet werden kann. [MSP 97]

nativ **Nativ** bezeichnet die ursprüngliche Form, z. B. bei nativem Code handelt es sich um Code, der eigens für eine bestimmte Maschine/Prozessor geschrieben wurde. [MSP 97]

Notation Für die Beschreibung von Elementen der Fachgebiete Programmierung und Mathematik verwendete Menge von Symbolen und Formaten. Grundsätzlich ist die Syntax einer Programmiersprache teilweise durch die **Notation** definiert. [MSP 97]

objektorientiert Eigenschaft von Systemen oder Programmiersprachen, die den Einsatz von **Objekten** unterstützten. [MSP 97]

Objektorientierung Beschreibt in der Programmierung einen Ansatz bzw. Paradigma, bei dem ein Programm als eine Sammlung diskreter Objekte (z. B. Datenstrukturen und Routinen) aufgefasst wird. [MSP 97]

Overhead Als **Overhead** werden für Rechnerprozesse unterstützende Arbeitsvorgänge, Informationen bezeichnet. Diese zusätzlichen Vorgänge und Informationen sind u. U. kein notwendiger Bestandteil der eigentlichen Operationen oder Daten, jedoch nehmen diese weitere Ressourcen wie z. B. Rechenzeit und Speicher in Anspruch. [MSP 97]

Paradigma Beispiel, Muster, Gleichnis [WIS 03]

Profiling **Profiling** bezeichnet grundsätzlich das Analysieren von Anwendungen bzw. des dadurch resultierenden Laufzeitverhaltens. Hierbei kann die Analyse in Form einer manuellen Implementierung oder durch Nutzen von Software erfolgen.

Programmiersprache "Eine **Programmiersprache** ist ein notationelles System zur Beschreibung von Berechnungen in durch Maschinen und Menschen lesbarer Form." [LOU 94]

Rekursion In der Programmierung ein Konzept, bei der sich eine **Funktion** (auch als Methode oder Routine bezeichnet) selbst aufruft. [MSP 97]

sequenziell in linearer Reihenfolge, vom Anfang schrittweise bis zum Ende [MSP 97]

Shell Ein Programm das als Schnittstelle für die direkte Kommunikation zwischen Betriebssystem und Anwender fungiert. Dabei sind für Betriebssysteme grundsätzlich zwei Arten zu unterscheiden, zum einen die GUI (Graphical User Interface, grafische Benutzeroberfläche) und zum anderen die CLI (Command Line Interface, Kommandozeile). [MSP 97, WIKI 12]

Stack **Stack** (Stapel) bezeichnet einen reservierten Speicherbereich, in dem Anwenderprogramme Daten über Zustände (z. B. Übergabeparameter, Rückkehradressen von Funktionen) zwischenspeichern. [MSP 97]

Stream Ein **Stream** (Datenstrom) ist durch eine vorwiegend kontinuierliche Folge von Daten, z. B. Bits, Bytes oder andere äquivalent strukturierte Einheiten gekennzeichnet. [MSP 97]

strukturiert Bei der **strukturierten** Programmierung, wird ein Programm in Module (oder auch Teilprogramme) als hierarchisch geordnete Bausteine zusammengestellt. Dabei erfolgt die Ausführung von Funktionen ausschließlich in jedem Baustein. [WIS 01]

sukzessiv nach und nach, aufeinander folgend [WIS 05]

UNIX Ein leistungsstarkes Mehrbenutzer-, Multitasking-Betriebssystem, welches im Vergleich zu anderen Systemen eine größere Architekturneutralität aufweist. Dabei ist die Systemsoftware hauptsächlich in der Programmiersprache C geschrieben. Auf **UNIX** basierend gibt es eine Vielzahl von weiteren Systemen, wie z. B. BSD, Linux, Mac OS. [MSP 97]

Quellenverzeichnis

Soweit nicht anders gekennzeichnet, stammen Begriffserklärungen, Textpassagen, Definitionen, Bilder bzw. Zitate aus den im Folgenden aufgelisteten Quellen. Dabei sind die Quellen in der Bachelorarbeit, immer durch eckige Klammern [ABC 01] bzw. [ABC 01/02] (bei mehreren Quellen) am Ende von Abschnitten, Absätzen, Abbildungen, Sätzen, Tabellen oder ganzen Texten gekennzeichnet.

Folgende Quellen wurden verwendet:

Bücher / Zeitschriftenartikel:

[AJ 00] **Aitken, Peter; Jones, Bradley L.:** C in 21 Tagen; Markt & Technik Verlag: München, 2000

[AU 96] **Aho, Alfred V.; Ullman Jeffrey D.:** Informatik - Datenstrukturen und Konzepte der Abstraktion; 1. Auflage (1996); International Thomson Publishing: Bonn, 1996

[EGG 05] **Eggink, Bernd:** C++ für C-Programmierer; 13. veränderte Auflage [November 2005]; Regionales Rechenzentrum für Niedersachsen (RRZN) / Universität Hannover, 2005

[HER 04] **Herold, Helmut:** Linux/Unix-Systemprogrammierung; 3., aktualisierte Auflage (2004); Addison-Wesley Verlag: München, 2004

[HPR 05] **Hübscher, Heinrich; Petersen, Hans-Joachim; Rathgeber, Carsten; Richter, Klaus, Dr. Scharf, Dirk:** IT-Handbuch - IT-Systemelektroniker/-in, Fachinformatiker/-in; 4. Auflage (2005); Westermann Schulbuchverlag GmbH: Braunschweig, 2005

[KER 08] **Kersken, Sacha**: IT-Handbuch für Fachinformatiker - Der Ausbildungsbegleiter; 3., aktualisierte Auflage (2008); Galileo Computing, 2008

[KRÖ 06] **Dr. Kröckertskothen, Thomas:** Java 2 - Grundlagen und Einführung; 5. unveränderte Auflage [April 2006]; Regionales Rechenzentrum für Niedersachsen (RRZN) / Universität Hannover, 2006

[KUB 07] **Kubiak, Sven:** Antwort- und Laufzeitmessungen: Prinzip, Implementierung und Experimente; Seminararbeit an der Universität Duisburg-Essen; 1. Auflage (2007); GRIN Verlag, 2007

[LM 05] **Louis, Dirk; Müller, Peter:** Das Java Codebook; Addison-Wesley Verlag: München, 2005

[LOU 94] **Louden, Kenneth C.:** Programmiersprachen - Grundlagen, Konzepte, Entwurf; Herausgegeben von: Prof. Dr. Bernd Mahr, Prof. Dr. Alexander Schill, Prof. Dr. Gottfried Vossen; 1. Auflage (1994); International Thomson Publishing, 1994

[MSP 97] **Microsoft Press:** Computer Fachlexikon, Fachwörterbuch; Deutsche Übersetzung von "Microsoft Press: Computer Dictionary, Third Edition"; Microsoft Press Deutschland, 1997

[PR 97] **Pomberger, Gustav; Rechenberg, Peter:** Informatik-Handbuch; Carl Hanser Verlag: München/Wien, 1997

[SS 02] **Saake, Gunter; Sattler Kai-Uwe:** Algorithmen & Datenstrukturen - Eine Einführung mit Java; 1. Auflage (2002); dpunkt.verlag: Heidelberg, 2002

[STE 01] **Steyer, Ralph:** Java 2 - Professionelle Programmierung mit J2SE Version 1.3 [Kompendium]; Markt & Technik Verlag: München, 2001

[ULL 09] **Ullenboom, Christian:** Java ist auch eine Insel - Programmieren mit der Java Standard Edition Version 6; 8., aktualisierte Auflage (2009); Galileo Computing, 2009

[WIL 03] **Willms, André:** Das C++ Codebook; 1. Auflage (2003); Addison-Wesley Verlag: München, 2003

[WOL 06] **Wolf, Jürgen:** C von A bis Z - Das umfassende Handbuch für Linux, Unix und Windows; 2., aktualisierte und erweiterte Auflage (2006); Galileo Computing, 2006

[WOL 99] **Wolff, Christian:** Einführung in Java - Objektorientiertes Programmieren mit der Java 2-Plattform; Teubner Verlag: Leipzig, 1999

Internetquellen:

[FHF 01] http://www.cn.informatik.fh-furtwangen.de/~mueller/ce45_swt/ws04/gr3/Dokumentation/Java_Profiler_R1.pdf
- Hochschule Furtwangen, Fachbereich Informatik / Zugriff am 17.03.2010

[FHW 01] http://www.fh-wedel.de/~si/seminare/ws01/Ausarbeitung/6.linuxrt/LinuxRT2.htm
- Fachhochschule Wedel / Zugriff am 14.05.2010

[FOR 01] http://forum.ubuntuusers.de/topic/java-ist-java-wirklich-so-viel-lngsamer-als-c/
- Zugriff am 22.03.2010

[DPT 01] http://www.dpunkt.de/java/Die_Sprache_Java/Die_Sprachelemente_von_Java/56.html
- dpunkt.verlag GmbH / Zugriff am 09.05.2010

[GCC 01] http://gcc.gnu.org/
- Free Software Foundation, Inc. / Zugriff am 08.03.2010

[GCC 02] http://gcc.gnu.org/onlinedocs/gcc-4.4.3/gcc/Optimize-Options.html#Optimize-Options
- Free Software Foundation, Inc. / Zugriff am 22.03.2010

[IFI 01] http://www.ifi.uzh.ch/study/Vorlesungen/unix/thema12/014_12.htm
- Institut für Informatik, Universität Zürich / Zugriff am 10.04.2010

[JIK 01] http://jikes.sourceforge.net/
- Zugriff am 18.04.2010

[JVM 01] http://www.statistik-portal.de/Intermaptiv/vm.asp
- Statistische Ämter des Bundes und der Länder / Zugriff am 07.05.2010

[LMU 01] http://www.cip.physik.uni-muenchen.de/cipdoc/node50.html
- Ludwig-Maximilians-Universität München / Zugriff am 24.04.2010

[MIN 01] http://www.mingw.org/
- Zugriff am 04.04.2010

[TIO 01] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
- TIOBE Software BV / Zugriff am 13.05.2010

[TUC 01] http://www.tu-chemnitz.de/urz/kurse/unterlagen/C/kap3/zeitmess.htm
- Technische Universität Chemnitz / Zugriff am 12.03.2010

[WIKI 01] http://de.wikipedia.org/wiki/C_(Programmiersprache)
- Wikimedia Foundation, Inc. / Zugriff am 04.03.2010

[WIKI 02] http://de.wikipedia.org/wiki/C%2B%2B
- Wikimedia Foundation, Inc. / Zugriff am 04.03.2010

[WIKI 03] http://de.wikipedia.org/wiki/Generische_Programmierung
- Wikimedia Foundation, Inc. / Zugriff am 04.03.2010

[WIKI 05] http://de.wikipedia.org/wiki/Dynamische_Bindung
- Wikimedia Foundation, Inc. / Zugriff am 06.03.2010

[WIKI 06] http://de.wikipedia.org/wiki/GNU_Compiler_Collection
- Wikimedia Foundation, Inc. / Zugriff am 08.03.2010

[WIKI 07] http://de.wikipedia.org/wiki/Ahead-of-time-Compiler
- Wikimedia Foundation, Inc. / Zugriff am 08.03.2010

[WIKI 08] http://de.wikipedia.org/wiki/Maximale_Laufzeit
- Wikimedia Foundation, Inc. / Zugriff am 16.03.2010

[WIKI 09] http://de.wikipedia.org/wiki/Festplatte
- Wikimedia Foundation, Inc. / Zugriff am 20.03.2010

[WIKI 10] http://de.wikipedia.org/wiki/Arbeitsspeicher
- Wikimedia Foundation, Inc. / Zugriff am 22.03.2010

[WIKI 11] http://de.wikipedia.org/wiki/Java_Development_Kit
- Wikimedia Foundation, Inc. / Zugriff am 18.04.2010

[WIKI 12] http://de.wikipedia.org/wiki/Betriebssystem-Shell
- Wikimedia Foundation, Inc. / Zugriff am 09.05.2010

[WIKI 13] http://de.wikipedia.org/wiki/C_%28Programmiersprache%29
- Wikimedia Foundation, Inc. / Zugriff am 09.05.2010

[WIL 01] http://www.willemer.de/informatik/cpp/timelib.htm
- Dipl.-Inform. Willemer, Arnold V. / Zugriff am 16.03.2010

[WIN 01] http://www.eckart-winkler.de/computer/c/makros.htm
- Winkler, Eckart / Zugriff am 24.04.2010

[WIS 01] http://www.wissen.de/wde/generator/wissen/ressorts/technik/computer/index,page=1250808.html
- wissenmedia GmbH, München / Zugriff am 04.03.2010

[WIS 02] http://www.wissen.de/wde/generator/wissen/ressorts/technik/computer/index,page=1218082.html
- wissenmedia GmbH, München / Zugriff am 04.03.2010

[WIS 03] http://www.wissen.de/wde/generator/wissen/ressorts/bildung/index,page=1208102.html
- wissenmedia GmbH, München / Zugriff am 04.03.2010

[WIS 04] http://www.wissen.de/wde/generator/wissen/ressorts/bildung/woerterbuecher/index,page=3842548.html
- wissenmedia GmbH, München / Zugriff am 19.03.2010

[WIS 05] http://www.wissen.de/wde/generator/wissen/ressorts/bildung/index,page=1251858.html
- wissenmedia GmbH, München / Zugriff am 09.05.2010

[WIS 06] http://www.wissen.de/wde/generator/wissen/ressorts/technik/computer/index,page=1309936,chunk=3.html
- wissenmedia GmbH, München / Zugriff am 12.05.2010

Abbildungsverzeichnis

Listing-Verzeichnis

Tabellenverzeichnis

zu Punkt 2.4
Programmiersprachen im Überblick

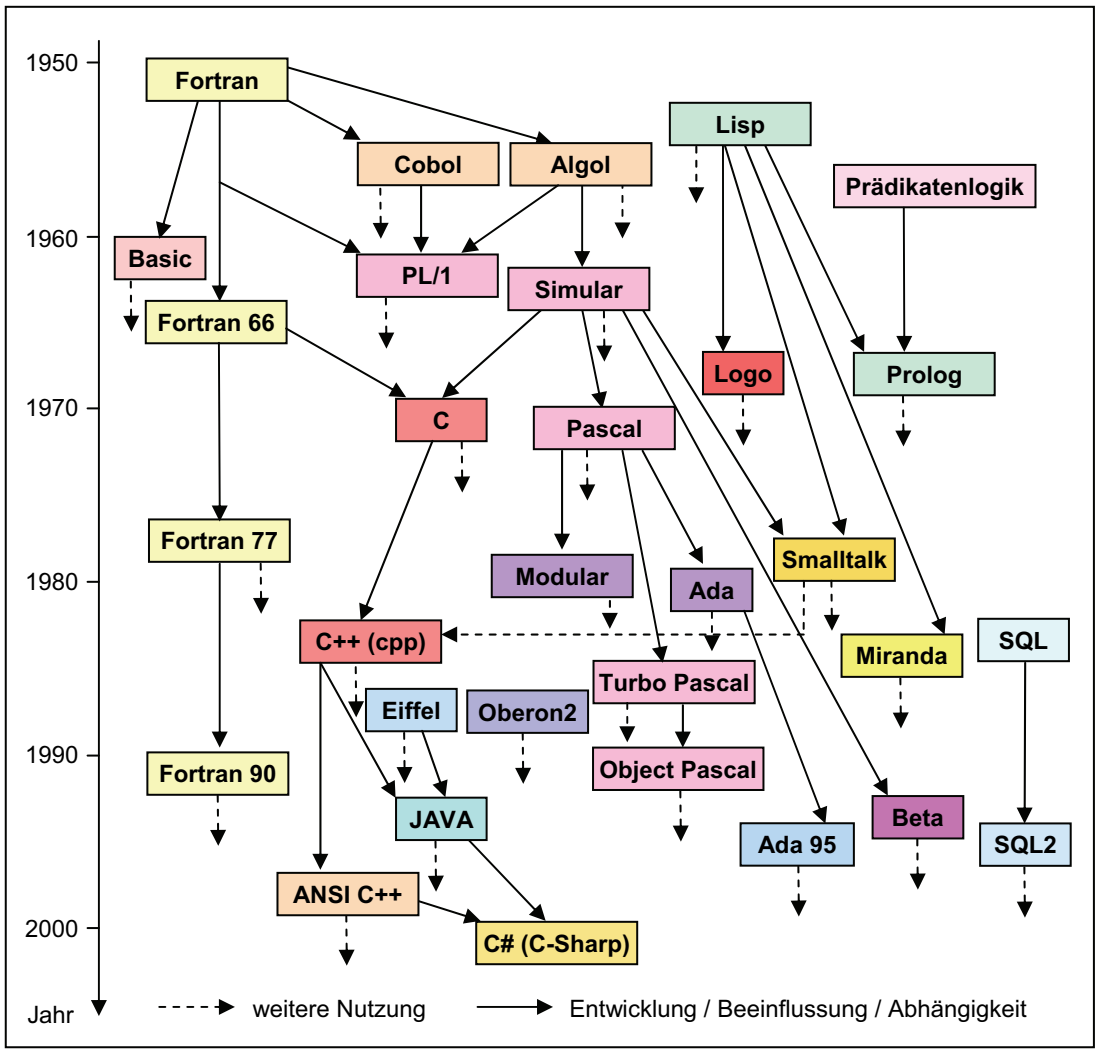


Abbildung 37: Stammbaum der Programmiersprachen [nach Quelle: HPR 05]

zu Punkt 5.1
Bereich CPU [Messung 1 - Windows OS]

Tabelle - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden][9]								
E/Dim	**C [UNO / OPT]**		**C++ [UNO / OPT]**		**JAVAC [UNO / OPT]**		**JIKES [UNO / OPT]**	
100	0,007	0,002	0,007	0,002	0,005	0,005	0,006	0,006
150	0,025	0,005	0,024	0,006	0,014	0,014	0,019	0,019
200	0,059	0,013	0,058	0,013	0,040	0,039	0,051	0,051
250	0,115	0,026	0,115	0,026	0,080	0,080	0,103	0,103
300	0,199	0,045	0,201	0,045	0,158	0,155	0,198	0,196
350	0,320	0,075	0,329	0,084	0,252	0,254	0,313	0,315
400	0,575	0,249	0,572	0,236	0,458	0,457	0,584	0,584
450	0,874	0,431	0,880	0,434	0,688	0,704	0,844	0,840
500	1,275	0,684	1,272	0,686	1,077	1,081	1,333	1,334
550	2,401	1,457	2,396	1,435	1,611	1,635	1,887	1,915
600	3,132	1,913	3,111	1,854	2,589	2,588	2,984	2,981
650	4,003	2,481	3,996	2,452	3,328	3,331	3,869	3,869
700	4,991	3,131	4,994	3,053	4,241	4,243	4,891	4,893
750	6,137	3,889	6,104	3,751	4,428	4,415	5,159	5,127
800	7,451	4,768	7,484	4,725	6,493	6,494	7,421	7,396
850	8,990	5,766	9,018	5,705	7,824	7,820	8,935	8,946
900	10,705	6,865	10,733	6,819	9,451	9,450	10,716	10,750
950	12,598	8,045	12,663	8,148	9,846	9,859	11,123	11,128
1000	14,779	9,531	14,833	9,571	13,348	13,332	14,914	14,931
1050	17,182	11,118	17,211	10,968	15,510	15,562	17,298	17,289
1100	19,800	12,857	19,856	12,800	18,029	18,036	20,083	20,055
1150	22,605	14,766	22,707	14,631	17,234	17,442	19,423	19,341
1200	25,779	16,884	25,884	16,731	23,621	23,652	26,179	26,277
1250	29,238	19,306	29,318	19,165	26,923	26,912	29,704	29,711
1300	32,947	21,799	33,006	21,584	30,618	30,841	33,539	33,538
1350	36,994	24,312	37,274	24,326	29,984	30,037	33,042	33,011
1400	41,370	27,159	41,464	27,023	38,644	38,683	42,191	42,195
1450	45,955	30,229	46,037	29,852	43,485	43,497	47,456	47,645
1500	51,169	33,594	50,941	33,424	48,410	48,563	52,775	52,855

Tabelle 31: Vergleich CPU - Mittelwerte [Windows OS]

[9] Werte gerundet auf die 3. Nachkommastelle

zu Punkt 5.1
Bereich CPU [Messung 2 - Linux OS]

Tabelle - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden]						
E/Dim	**C [UNO / OPT]**		**C++ [UNO / OPT]**		**JAVAC [UNO / OPT]**	
100	0,0073	0,0016	0,0075	0,0017	0,00332	0,00325
150	0,0245	0,0057	0,025	0,0056	0,01031	0,01028
200	0,0588	0,0132	0,0598	0,0129	0,02156	0,02158
250	0,1163	0,0254	0,1179	0,0255	0,05442	0,05443
300	0,2023	0,0443	0,2031	0,044	0,12219	0,12222
350	0,3213	0,1011	0,3245	0,0891	0,20484	0,20553
400	0,5741	0,2364	0,6108	0,2648	0,3994	0,40464
450	0,9104	0,4926	1,0231	0,498	0,58589	0,58425
500	1,4618	0,6854	1,5418	0,8313	0,94429	1,00034
550	2,0951	1,3769	2,2002	1,2107	1,2574	1,21773
600	3,0992	1,8066	2,9644	1,6848	2,29219	3,10696
650	3,9652	2,3102	3,8469	2,243	2,33694	2,3588
700	4,9341	2,8941	4,9188	2,8679	3,23808	3,22833
750	6,0718	3,5681	6,0404	3,5563	4,22957	4,98373
800	7,3877	4,3716	7,3681	4,3586	5,29026	5,27899
850	8,8927	5,2586	8,8648	5,2563	5,60666	5,64469
900	10,5548	6,2733	10,5479	6,273	7,56386	7,57251
950	12,3938	7,4016	12,3843	7,4155	16,81887	20,76526
1000	14,4479	8,6806	14,4489	8,6964	24,35519	24,1646
1050	16,7586	10,1253	16,7652	10,149	11,36662	11,21599
1100	19,2917	11,7056	19,3343	11,7236	14,62598	14,59855
1150	22,0841	13,4941	22,0991	13,5179	16,44588	20,13759
1200	25,1245	15,4517	25,163	15,4705	18,83383	19,43799
1250	28,434	17,5588	28,4144	17,6001	20,92469	20,90188
1300	31,937	19,8685	31,9659	19,9285	24,46905	24,50398
1350	35,8723	22,4297	35,8128	22,4965	24,31979	24,85491
1400	39,983	25,1585	39,9853	25,1784	30,66677	30,249
1450	44,6233	28,0373	44,3825	28,0905	34,87589	35,2977
1500	49,158	31,0802	49,1492	31,1437	40,05721	40,1097

Tabelle 32: Vergleich CPU - Mittelwerte [Linux OS]

zu Punkt 5.1
Bereich CPU [Windows vs. Linux]

Tabelle - Vergleich der Mittelwerte von C/C++/Java (optimiert) [Zeit in Sekunden][10]

E/Dim	C [Windows / Linux]		C++ [Windows / Linux]		JAVAC [Windows / Linux]	
100	0,002	0,002	0,002	0,002	0,005	0,003
150	0,005	0,006	0,006	0,006	0,014	0,010
200	0,013	0,013	0,013	0,013	0,039	0,022
250	0,026	0,025	0,026	0,026	0,080	0,054
300	0,045	0,044	0,045	0,044	0,155	0,122
350	0,075	0,101	0,084	0,089	0,254	0,206
400	0,249	0,236	0,236	0,265	0,457	0,405
450	0,431	0,493	0,434	0,498	0,704	0,584
500	0,684	0,685	0,686	0,831	1,081	1,000
550	1,457	1,377	1,435	1,211	1,635	1,218
600	1,913	1,807	1,854	1,685	2,588	3,107
650	2,481	2,310	2,452	2,243	3,331	2,359
700	3,131	2,894	3,053	2,868	4,243	3,228
750	3,889	3,568	3,751	3,556	4,415	4,984
800	4,768	4,372	4,725	4,359	6,494	5,279
850	5,766	5,259	5,705	5,256	7,820	5,645
900	6,865	6,273	6,819	6,273	9,450	7,573
950	8,045	7,402	8,148	7,416	9,859	20,765
1000	9,531	8,681	9,571	8,696	13,332	24,165
1050	11,118	10,125	10,968	10,149	15,562	11,216
1100	12,857	11,706	12,800	11,724	18,036	14,599
1150	14,766	13,494	14,631	13,518	17,442	20,138
1200	16,884	15,452	16,731	15,471	23,652	19,438
1250	19,306	17,559	19,165	17,600	26,912	20,902
1300	21,799	19,869	21,584	19,929	30,841	24,504
1350	24,312	22,430	24,326	22,497	30,037	24,855
1400	27,159	25,159	27,023	25,178	38,683	30,249
1450	30,229	28,037	29,852	28,091	43,497	35,298
1500	33,594	31,080	33,424	31,144	48,563	40,110

Tabelle 33: Vergleich CPU - Mittelwerte (optimiert) [Windows vs. Linux]

[10] Werte gerundet auf die 3. Nachkommastelle

zu Punkt 5.2
Bereich RAM [Messung 1 - Windows OS]

Tabelle - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden][11]								
E/Dim	C [UNO / OPT]		C++ [UNO / OPT]		JAVAC [UNO / OPT]		JIKES [UNO / OPT]	
200	0,010	0,011	0,006	0,009	0,005	0,005	0,005	0,002
400	0,042	0,049	0,041	0,045	0,042	0,045	0,043	0,036
600	0,097	0,102	0,090	0,093	0,088	0,084	0,079	0,082
800	0,171	0,179	0,161	0,168	0,142	0,137	0,134	0,132
1000	0,263	0,285	0,252	0,263	0,223	0,222	0,215	0,205
1200	0,369	0,399	0,352	0,376	0,312	0,305	0,288	0,287
1400	0,487	0,548	0,476	0,510	0,422	0,411	0,397	0,391
1600	0,639	0,715	0,621	0,674	0,519	0,488	0,501	0,468
1800	0,793	0,903	0,806	0,858	0,689	0,669	0,622	0,596
2000	0,998	1,110	0,988	1,061	0,816	0,787	0,802	0,765
2200	1,205	1,341	1,187	1,270	1,050	0,982	1,109	1,084
2400	1,423	1,606	1,421	1,515	1,251	1,255	1,113	1,068
2600	1,663	1,883	1,674	1,780	1,420	1,303	1,280	1,211
2800	1,956	2,173	1,917	2,048	1,663	1,573	1,572	1,570
3000	2,230	2,498	2,211	2,352	1,848	1,672	1,864	1,736
3200	2,556	2,848	2,534	2,695	1,935	2,377	1,609	1,589
3400	2,871	3,213	2,841	3,029	2,035	2,344	1,839	1,787
3600	3,216	3,635	3,190	3,398	2,353	2,420	2,252	2,176
3800	3,556	4,023	3,552	3,797	2,581	2,414	2,252	2,227
4000	4,001	4,440	3,932	4,183	2,703	2,651	2,489	2,388

Tabelle 34: Vergleich RAM - Mittelwerte [Windows OS]

[11] Werte gerundet auf die 3. Nachkommastelle

zu Punkt 5.2
Bereich RAM [Messung 2 - Linux OS]

Tabelle - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden]						
E/Dim	**C [UNO / OPT]**		**C++ [UNO / OPT]**		**JAVAC [UNO / OPT]**	
200	0,004	0,006	0,0035	0,003	0,00255	0,00095
400	0,01	0,015	0,0165	0,0115	0,00515	0,0016
600	0,024	0,035	0,0355	0,032	0,00975	0,00685
800	0,0455	0,056	0,0635	0,049	0,02395	0,0145
1000	0,072	0,0935	0,1015	0,077	0,0275	0,0235
1200	0,1035	0,138	0,143	0,109	0,04005	0,01545
1400	0,1415	0,1795	0,1995	0,143	0,02665	0,0393
1600	0,1845	0,244	0,2545	0,199	0,16205	0,06285
1800	0,207	0,309	0,3275	0,2525	0,22445	0,2943
2000	0,2515	0,381	0,3985	0,309	0,50485	0,51995
2200	0,3	0,4585	0,49	0,375	0,77825	0,7504
2400	0,359	0,548	0,5725	0,4425	1,12395	0,8941
2600	0,4195	0,6425	0,683	0,5255	1,3449	1,26855
2800	0,493	0,742	0,7815	0,6065	1,55375	1,33675
3000	0,562	0,855	0,91	0,695	2,34015	2,0128
3200	0,639	0,9745	1,0225	0,793	2,39075	2,22835
3400	0,723	1,0975	1,167	0,8955	2,5503	2,3942
3600	0,826	1,227	1,301	1,002	2,7078	2,5057
3800	0,8995	1,3645	1,453	1,12	2,93125	2,62755
4000	0,9995	1,506	1,61	1,241	3,14165	2,889

Tabelle 35: Vergleich RAM - Mittelwerte [Linux OS]

zu Punkt 5.2
Bereich RAM [Windows vs. Linux]

Tabelle - Vergleich der Mittelwerte von C/C++/Java [Zeit in Sekunden][12]						
E/Dim	**C [Windows / Linux]**		**C++ [Windows / Linux]**		**JAVAC [Windows / Linux]**	
200	0,011	0,006	0,009	0,003	0,005	0,001
400	0,049	0,015	0,045	0,012	0,045	0,002
600	0,102	0,035	0,093	0,032	0,084	0,007
800	0,179	0,056	0,168	0,049	0,137	0,015
1000	0,285	0,094	0,263	0,077	0,222	0,024
1200	0,399	0,138	0,376	0,109	0,305	0,015
1400	0,548	0,180	0,510	0,143	0,411	0,039
1600	0,715	0,244	0,674	0,199	0,488	0,063
1800	0,903	0,309	0,858	0,253	0,669	0,294
2000	1,110	0,381	1,061	0,309	0,787	0,520
2200	1,341	0,459	1,270	0,375	0,982	0,750
2400	1,606	0,548	1,515	0,443	1,255	0,894
2600	1,883	0,643	1,780	0,526	1,303	1,269
2800	2,173	0,742	2,048	0,607	1,573	1,337
3000	2,498	0,855	2,352	0,695	1,672	2,013
3200	2,848	0,975	2,695	0,793	2,377	2,228
3400	3,213	1,098	3,029	0,896	2,344	2,394
3600	3,635	1,227	3,398	1,002	2,420	2,506
3800	4,023	1,365	3,797	1,120	2,414	2,628
4000	4,440	1,506	4,183	1,241	2,651	2,889

Tabelle 36: Vergleich RAM - Mittelwerte (optimiert) [Windows vs. Linux]

[12] Werte gerundet auf die 3. Nachkommastelle