HAW HAMBURG

Bachelorarbeit

Aaron Sielaff

Java vs. Kotlin: Fallstudie zur Energieeffizienz und Performance

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Aaron Sielaff

Java vs. Kotlin: Fallstudie zur Energieeffizienz und Performance

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Julia Padberg
Zweitgutachter: Prof. Dr. Christian Lins

Eingereicht am: 26. Mai 2024

Aaron Sielaff

Thema der Arbeit

Java vs. Kotlin: Fallstudie zur Energieeffizienz und Performance

Stichworte

Java, Kotlin, Energieeffizienz, Performance, Programmiersprachen, Vergleich

Kurzzusammenfassung

Die Performance und die Energieeffizienz sind häufig keine großen Faktoren bei der Wahl einer Programmiersprache. In dieser Bachelorarbeit wird untersucht, ob eine Sprachempfehlung auf Grundlage der Energieeffizienz oder der Performance für bestimmte Operationsdomänen getroffen werden kann. Dies geschieht durch einen Vergleich der Programmiersprachen Java und Kotlin auf der JVM. Dabei werden die Bereiche Algorithmen, Dateien, Listen und (De-)Serialisierung untersucht. Diese Programmiersprachen wurden aufgrund ihrer Popularität, ihrer Ähnlichkeiten zueinander und der breiten Verwendung im mobilen Bereich gewählt. Es werden zuerst einzelne Operationen mit festem Input und danach mit unterschiedlichem Input getestet, wobei beim unterschiedlichen Input alle Operationen eines Bereiches mit Ausnahme der Algorithmen kombiniert werden. Danach werden die Kosten der einzelnen Operationen betrachtet und mit statistischen Tests untersucht. Dabei wurde festgestellt, dass die Last der Operationen häufig zu gering war, um eine genaue Aussage zu treffen. Für den Bereich Dateien konnten kaum Unterschiede zwischen den Sprachen gefunden werden. Eine Sprachempfehlung kann nur für die (De-)Serialisierung für Kotlin ausgesprochen werden, sofern die Laufzeit ein wichtiges Entscheidungskriterium ist.

Aaron Sielaff

Title of Thesis

Java vs. Kotlin: Case study on energy efficiency and performance

Keywords

Java, Kotlin, Energy Efficiency, Performance, Programming Language, Comparison

Abstract

Performance and energy efficiency are often not major factors when choosing a programming language. This bachelor thesis examines whether a language recommendation can be made on the basis of energy efficiency or performance for certain operation domains. This is done by comparing the programming languages Java and Kotlin on the JVM in the areas of algorithms, files, lists and (de-)serialization. Java and Kotlin were chosen based on their popularity, similarity, and widespread use in mobile application development. First, individual operations with a fixed input are tested. After that, the operations of a domain are combined and tested with different inputs. The algorithms are still individually tested. The costs of the individual operations are then considered and examined using statistical tests. It was found that the load of the operations was often too low to make an accurate statement. Hardly any differences were found between the languages in the file domain. A recommendation can only be made for the (de-)serialization domain, where Kotlin is superior if runtime is an important decision criterion.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Ablaufsverzeichnis

Abkürzungsverzeichnis

API Programmierschnittstelle

CLBG Computer Language Benchmarks Game

GFX Graphics

I/O Eingabe/Ausgabe

JSON JavaScript Object Notation

JVM Java Virtual Machine

RAM Random Access Memory (Arbeitsspeicher)

Glossar

Compiler Flag

Optionen, die dem Compiler übergeben werden, um die Kompilierung zu beeinflussen. 40

Datenquelle

Eine Ansammlung von Implementierungen. 5

Einzeltest

Test einer Grundoperation mit einem festen Input. 28, 43, 58

Garbage Collector

Der Garbage Collector bereinigt den Arbeitsspeicher von nicht mehr benötigten Reservierungen. 65

Interoperabilität

Die Möglichkeit mit einem anderen System oder einer anderen Programmiersprache zusammenzuarbeiten. 1, 8

Kernelmode

Befindet sich ein Prozess im Kernelmode, genießt er höhere Rechte, wodurch zum Beispiel direkter Zugriff auf Hardwareressourcen möglich ist. 26

Kolmogorow-Smirnow-Test

Es wird nach der größten Differenz zwischen zwei kumulativen Verteilungen gesucht. Aus dieser Differenz wird eine Teststatistik berechnet, mit der die Wahrscheinlichkeit einer gleichen Verteilung bestimmt und die Ähnlichkeit bewertet werden kann. Die Nullhypothese ist eine gleiche Verteilung [6]. 35, 61

Kombinationstest

Test in dem alle Grundoperationen einer Operationdomäne verbunden werden. Jede Wiederholung hat einen neuen Input. 29, 50, 58

Memory Allocation

Zuteilung und Reservierung des Arbeitsspeicher für einzelne Prozesse. 65

Operationsdomäne

Arbeitspakete, wie zum Beispiel Algorithmen, Dateien und Listen. 3, 5

Overhead

Der Ressourcenverbrauch, der durch zusätzliche Aktionen entsteht. 24, 27, 44

Package

Die Hardwareeinheit, die umgangssprachlich als "CPU" bekannt ist und weitere Komponenten wie Grafikchips enthalten kann. 26

Performance

Performance beschreibt, wie wenig Hardwareressourcen verwendet werden. Darunter fallen zum Beispiel Arbeitsspeicher, CPU und Laufzeit [13]. 2, 10

REST-Server

Ein REST-Server ist ein Server, auf dem eine REST-API läuft. REST sind flexible Richtlinie für eine Client/Server-Architektur mit Kommunikation über HTTP [39]. 17

Seed

Ein Wert mit dem ein Zufallsgenerator initialisiert wird. Derselbe Seed bewirkt immer dieselbe Ausgabe. 31

Shapiro-Wilk-Test

Es wird eine gewichtete Summe der Messwerte und die Varianz verwendet, um eine Teststatistik zu berechnen. Diese Teststatistik wird mit einer Tabelle verglichen, um die Wahrscheinlichkeit einer Normalverteilung zu bekommen. Die Nullhypothese geht von einer Normalverteilung aus [50]. 21, 35, 38, 61

Signifikanzniveau

Das Signifikanzniveau legt eine Grenze fest, ab der ein Ergebnis statistisch signifikant ist. 58

Spearman'scher Rangkorrelationskoeffizient

Es wird jedem Messwert ein Rang zugeordnet und dann aus den Differenzen zwischen den einzelnen Rängen und dem Durchschnittsrang eine Teststatistik berechnet. Anhand dieser Teststatistik kann bewertet werden, ob eine monotone Korrelation existiert. Die Nullhypothese ist, dass beide Datensätze unabhängig sind [25]. 36, 63

System Overhead

Der Ressourcenverbauch, der durch die Aktionen des Systems entstehen, die für das Experiment nicht relevant sind. 16

Usermode

Der Standardmodus, in dem sich Prozesse befinden. Der Usermode hat weniger Rechte. 27

Variationstest

Test für einen Algorithmus. Jede Wiederholung hat einen neuen Input. 30, 43, 58

Welch's t-Test

Es wird mithilfe der Varianz die Anzahl der Freiheitsgrade und eine Teststatistik berechnet. Mit diesen beiden Werten kann ein p-Wert aus einer Tabelle entnommen werden, der bei der Entscheidung hilft, ob für eine Softwaremetrik beide Sprachen denselben Mittelwert haben. Der Test benötigt eine gleiche Verteilung beider Messungen. Die Nullhypothese ist ein gleicher Mittelwert für beide Datensätze [25]. 35, 61

1 Einleitung

1.1 Motivation

Java ist eine weitverbreitete Programmiersprache, die auf verschiedenen Plattformen genutzt werden kann. Einige Programmierer sehen in Kotlin eine Alternative zu Java, während andere Kotlin eher als Nachfolgersprache betrachten. Dieser Gedanke wird durch Kotlins beidseitiger Interoperabilität zu Java gefestigt. Durch diese können bestehende Java-Projekte ohne Probleme mit Kotlin erweitert werden, wobei eine Kotlin-Funktion aus einer Java-Funktion und umgekehrt aufgerufen werden kann. Dazu kommt, dass Kotlin die von Google bevorzugte Sprache für die Entwicklung von Mobile-Apps für Android-Systeme ist [3].

Java belegt in der jährlichen Stackoverflow Developer Survey [51], in welcher verschiedene Fragen an die Benutzer gestellt werden, den siebten Platz in den benutzten Programmiersprachen. In weiteren Ranglisten liegt Java in den oberen 5 Plätzen, während Kotlins Position unterschiedlich bewertet wird. Hier schwanken die Plätze zwischen dem 25. und dem 12. Platz [8, 53, 27, 16].
Von den Programmierern, die in 2023 Java benutzt haben, wollen 24,85 % der Programmierer Kotlin in 2024 mindestens einmal verwenden [51].

Die Wahl einer Programmiersprache basiert dabei oft auf verschiedenen Faktoren. Dabei spielen unter anderem Faktoren wie Projektdauer, Fähigkeiten der Teammitglieder, das Budget, die Voraussetzungen des Kunden oder organisationsbezogene Auflagen eine Rolle. Für bestimmte Aufgaben könnte beispielsweise eine spezialisierte oder domainspezifische Programmiersprache besser geeignet sein, während für eine Standarddesktopanwendung eine vielseitige Programmiersprache hilfreich sein kann [57].
Bei geeigneter Wahl der Programmiersprache könnten große Energieeinsparungen möglich sein. In einigen Benchmarks verbraucht Java 1,98-mal so viel Energie wie C.

In den gleichen Benchmarks verbraucht die Skriptsprache Python bis zu 75,88-mal so viel Energie wie C [34].

Aufgrund der Verbreitung von Java, der Ähnlichkeit zwischen Java und Kotlin und dem Wechsel von Entwicklern zu Kotlin könnte es bei vielen Anwendungen einen Einfluss auf den Energieverbrauch und die Performance geben. Für eine Einschätzung müsste jedoch bekannt sein, wie groß die Unterschiede zwischen den Programmiersprachen sind. Das Ziel dieser Thesis ist es, die Größe dieser Unterschiede zu finden.

1.2 Ziel der Arbeit

Bei der Wahl einer Programmiersprache spielen, wie in Unterkapitel 1.1 erwähnt, viele Faktoren eine Rolle. Unter solchen Faktoren fallen auch die Energieeffizienz und die Performance einer Programmiersprache. Dabei haben diese Faktoren jedoch nicht selten kleinere Rollen in der Entscheidung oder werden nicht berücksichtigt, da der Fokus auf anderen Faktoren liegt. Bei einem großen Projektumfang oder bei einer großen Verbreitung der Anwendung kann es jedoch große Unterschiede bei der Performance oder dem Energieverbrauch geben. Dadurch könnte ein Wechsel sich lohnen.
Verbraucht eine Anwendung am Tag beispielsweise 100 Wattstunden, kann eine Differenz in der Energieeffizienz große Auswirkungen haben. Eine Differenz von 1 %, also 1 Wattstunde, kann, wenn diese Anwendung auf 1 000 Geräten ausgeführt wird, schon zu einer Einsparung oder einem Mehrverbrauch von 1 Kilowattstunde führen. Da Java und Kotlin im mobilen App-Bereich verwendet werden, wo Apps hunderttausendfach oder sogar millionenfach heruntergeladen werden, kann eine kleine Differenz schon große Auswirkungen auf den Gesamtstromverbrauch haben.
Aus diesem Grund ist es wichtig diese Faktoren zumindest zu betrachten.

Zu dieser Thematik existieren schon einige Studien, welche mehrere Programmiersprachen auf die Energieeffizienz und Performance untersuchen, jedoch ist Kotlin hier nicht vertreten [10, 34, 35]. In weiteren Studien werden Java und Kotlin zwar verglichen, die Ergebnisse zwischen den Studien sind aber nicht eindeutig. In einigen Studien wird die Laufzeit, Energie- und Ressourceneffizienz von Java- und Kotlin-Apps auf Android-Geräten untersucht [36, 55]. Dabei stellt [36] keinen statistisch signifikanten Unterschied fest, während [55] Unterschiede in der CPU-Leistung und benötigten Arbeitsspeicher feststellt. In den von [13] untersuchten Benchmarks ist die Laufzeit der Java-Lösung meist geringer und benötigt weniger Arbeitsspeicher.

Das Ziel dieser Arbeit ist es, die Energieeffizienz und die Performance von Java und Kotlin näher zu untersuchen, um ein breiteres Bild dieser Metriken zu erhalten. Aus diesem Grund soll eine Unabhängigkeit von der Zielplattform gewahrt werden. Während der Untersuchung wird auch überprüft, ob auf Grundlage der Energieeffizienz oder der Performance eine Sprachempfehlung für bestimmte Operationsdomänen getroffen werden kann.

1.3 Vorgehensweise

Um eine Vergleichbarkeit der Programmiersprachen zu erreichen, sollen verschiedene Operationsdomänen in Java als auch in Kotlin umgesetzt werden. Operationsdomänen sind Arbeitspakete, welche verschiedene Operationen beinhalten. In dieser Arbeit werden folgende Operationsdomänen untersucht: *Algorithmen*, *Dateien*, *Listen*, *Serialisierung*. Die Operationsdomäne *Listen* umfasst beispielsweise alle Operationen, die mit Listen zu tun haben, wie das Lesen, Löschen und Suchen von Elementen. Dabei sind Grundoperationen die grundlegenden Implementierungen, die benutzt werden können, um andere Operationen einer Operationsdomäne umzusetzen. Das Suchen von Elementen benötigt zum Beispiel das Lesen von Elementen, um zu funktionieren. Die Operationsdomäne *Algorithmen* ist besonders, da hier keine wirklichen Grundoperationen existieren, sondern einzelne Algorithmen durch Grundoperationen anderer Operationsdomänen zusammengesetzt werden. Als Beispiel dient auch hier das Durchsuchen von Listen, denn es kann mithilfe von verschiedenen Algorithmen umgesetzt werden, um die Effizienz zu steigern. Zuerst wird der Energie- und Ressourcenverbrauch aller Grundoperationen und Algorithmen gemessen. Danach werden für die einzelnen Operationsdomänen in größeren Testdurchläufen alle Grundoperationen in unterschiedlicher Ausführungsreihenfolge durchgeführt. Dadurch können die Operationsdomänen auf verschiedene Art und Weise betrachtet werden. Bei den Messungen fallen unter Ressourcenverbrauch die maximale CPU-Leistung, der maximale RAM-Verbrauch sowie die Laufzeit. Anhand dieser Messungen kann dann auch die Performance beider Sprachen verglichen werden.

Die Implementierungen der Algorithmen werden, wenn möglich, aus Rosetta Code [44] übernommen und wenn nötig angepasst. Rosetta Code ist eine Sammlung von Programmierproblemen und Lösungen in verschiedenen Programmiersprachen [44]. Bei den Implementierungen von Rosetta Code muss auf ihre Vergleichbarkeit geachtet werden. So wird nicht immer dieselbe Arbeit ausgeführt. Es wurde unter anderem festgestellt, dass einige Implementierungen Unterschiede in der Eingabe/Ausgabe (I/O), wie Dateien oder einprogrammierten Konstanten, haben. Auch wurden in einigen Implementierungen Listen sortiert, in anderen jedoch nicht [35]. Wie mit diesen Problemen umgegangen wird, wird in Unterkapitel 2.3.2 besprochen.

Alle Messergebnisse werden im Anschluss auf statistische Signifikanz geprüft und analysiert, damit anschließend die Performance und die Energieeffizienz für bestimmte Operationsdomänen bewertet und eine Sprachempfehlung untersucht werden kann.

1.4 Aufbau der Bachelorarbeit

Kapitel 2 stellt den jetzigen Stand der Forschung dar und baut durch Erläuterungen sowie Erklärungen das Fundament für diese Ausarbeitung und die verwendeten Tests. Danach wird sich in Kapitel 3 mit der Methodik befasst. Es erläutert, wie die Implementierung und Durchführung geplant ist sowie welche Softwaremetriken und Analysetechniken verwendet werden. Die Umsetzung des Experiments wird in Kapitel 4 behandelt. Hier werden ebenfalls Besonderheiten in der Umsetzung oder Auffälligkeiten besprochen. Kapitel 5 stellt die Ergebnisse vor, welche dann in Kapitel 6 diskutiert werden. Den Abschluss dieser Ausarbeitung bildet Kapitel 7 mit einem Fazit, in welchem die Ergebnisse abschließend bewertet werden und eine Einordnung zu anderen Studien stattfindet. Dort wird ebenfalls auf die initiale Frage eingegangen, ob eine operationsdomänenabhängige Sprachempfehlung gegeben werden kann und ein Ausblick auf mögliche zukünftige Forschungen gegeben.

2 Grundlagen

In diesem Kapitel wird der Stand der Forschung zusammengefasst und Grundlagenwissen für die folgenden Kapitel geschaffen. Dabei werden zuerst die zu untersuchenden Programmiersprachen vorgestellt. Danach wird definiert, was Operationsdomänen sind und aus welchen Grundoperationen diese bestehen. Zum Schluss wird der Stand der Forschung betrachtet. Dabei werden Datenquellen und Methodiken untersucht, um die Methodik dieser Untersuchung darauf aufzubauen. Die Ergebnisse der Studien werden auch betrachtet, um einen Vergleich zu den Ergebnissen dieser Untersuchung zu ermöglichen.

2.1 Programmiersprachen

In diesem Unterkapitel wird sich mit den beiden ausgewählten Programmiersprachen auseinandergesetzt. Dabei werden die Verbreitung, die Einsatzgebiete und die Unterschiede betrachtet. Zum Schluss wird darauf eingegangen, warum diese beiden Programmiersprachen ausgewählt wurden.

Java [30] ist eine statisch typisierte objektorientierte Programmiersprache, deren Programme in einen Bytecode kompiliert werden. Dieser Bytecode wird in einer Jar-Datei gespeichert. Die Jar-Datei beinhaltet alle benötigten Dateien und wird mithilfe der Java Virtual Machine (JVM) interpretiert, wodurch eine plattformunabhängige Ausführung möglich wird. Java wurde zuerst 1995 veröffentlicht und wird heutzutage in vielen Bereichen verwendet. Beispielsweise wird es zur Entwicklung von Android-Apps, Web-Applikationen, Desktop-GUI-Anwendungen, aber auch für den Einsatz in der Cloud, sowie für Big Data verwendet [29, 18].

Kotlin [19] ist eine moderne, statisch typisierte Programmiersprache, die objektorientierte und funktionale Ansätze kombiniert. Dabei bietet Kotlin eine Alternative zu Java auf der JVM und ist beidseitig interoperabel. Das ermöglicht beiden Sprachen das Teilen und Verwenden der gleichen Codebasis in einem Projekt. Die erste stabile Veröffentlichung erschien 2016. Da Kotlin nicht das komplette „Java Collections Framework“ neu implementiert, sorgt es für eine Kompatibilität zu den „Interfaces“, wodurch es zu keinen Problemen mit bestehenden Implementierungen kommt. Kotlin findet vor allem bei Android-Softwareentwicklern Anwendung, was auch daran liegt, dass Kotlin die von Google bevorzugte Sprache für die Entwicklung von Mobile-Apps für Android-Systeme ist [3].

2.1.1 Popularität

Laut der jährlichen Stackoverflow Developer Survey [51] belegt Java den siebten Platz in den benutzten Programmiersprachen, mit einer Nutzeranzahl von 26 757 bei 87 585 gegebenen Antworten, was 30,55 % entspricht. Dabei tendieren Programmieranfänger mit 35,17 % etwas stärker zu Java als professionelle Programmierer, bei denen 30,49 % der Befragten Java nutzen.
Kotlin wird dabei bisher von 7 935 Programmierern benutzt, was 9,06 % entspricht. Hier tendieren eher professionelle Programmierer mit 9,70 % zu Kotlin im Gegensatz zu Programmieranfängern, von denen 6,67 % Kotlin verwenden.
Von den Programmierern, die in 2023 Java benutzt haben, wollen 6 648 Programmierer Kotlin in 2024 mindestens einmal verwenden [51].

Der TIOBE Index [53], welcher in [13] verwendet wird, ist eine Rangliste von Programmiersprachen und wird mithilfe von 25 Suchmaschinen berechnet.
Dabei wird „+ “‘<language> programming‘“ als Suchbegriff verwendet und die Treffer gezählt. Danach werden die Werte für jede Suchmaschine und jede Programmiersprache normalisiert. Das heißt, alle Programmiersprachen zusammen ergeben 100 % [52].
Im TIOBE Index befindet sich Java auf Platz vier aller Programmiersprachen mit einer Bewertung von 7,87 % und hält damit denselben Platz wie im letzten Jahr. Dabei hat Java jedoch 4,34 % im Vergleich zum Vorjahr verloren [53].

Im November 2023 befand sich Kotlin im TIOBE Index mit 0,92 % auf Platz 15 und ist damit seit dem November 2022, wo es einen Wert von 0,58 % hatte, um acht Plätze aufgestiegen. Im Januar 2024 befindet sich Kotlin hingegen auf Platz 25 und ist damit seit dem Januar 2023 um acht Plätze gefallen. Dabei ist Kotlin von 0,48 % auf 0,85 % gestiegen [53]. Diese Variation zeigt, dass der TIOBE Index nicht als einzige Statistik genommen werden darf.
Da bei der Berechnung nur der eine Suchbefehl verwendet wird, kann der Index die Popularität eventuell falsch darstellen. Dazu kommt, dass einige der verwendeten Suchmaschinen, wie zum Beispiel die eingebauten Suchen in Amazon [1], Ebay [9] oder Walmart [54], nicht unbedingt die besten Quellen sind, um die Popularität und wahre Verwendung von Programmiersprachen zu messen. Der TIOBE Index wird dennoch als ein Datenpunkt verwendet, da zumindest unter den Top zehn sieben Programmiersprachen enthalten sind, die auch im Stackoverflow Developer Survey unter den Top zehn sind.

„The State of Developer Ecosystem 2023“, dessen Inhalt auf 26 348 Entwicklern basiert, liefert auch Erkenntnisse zur Popularität von Sprachen [16]. Java belegt hier den fünften Platz mit 49 %, wobei 3 % der Entwickler zukünftig Java benutzen oder zu Java migrieren wollen. Kotlin belegt hingegen den dreizehnten Platz mit 15 % und 6 % der Entwickler wollen zukünftig Kotlin benutzen oder zu Kotlin migrieren.
33 % der Entwickler geben Java und 8 % geben Kotlin als eine ihrer primären Programmiersprachen an. Dabei wollen 58 % der Java-Entwickler bei Java bleiben, während 9 % zu Kotlin wechseln wollen. Bei den Kotlin-Entwicklern wollen 65 % bei Kotlin bleiben und nur 1 % wollen zu Java wechseln [17].

Im Octoverse-Report von GitHub [12], welcher anhand der Repositories zusammengestellt wird, belegt Java den vierten und Kotlin den zwölften Platz [8].

Das „RedMonk Programming Language Rankings“, welches von [13] verwendet wird, berechnet die Plätze mithilfe von GitHub und Stackoverflow. Hier belegt Java den dritten und Kotlin den siebzehnten Platz. Das Wachstum von Kotlin stagniert, da es bereits eine gewisse Sättigung erreicht hat und andere populäre Programmiersprachen verdrängen muss, um weiterzuwachsen [27].

2.1.2 Unterschiede

Die beiden Programmiersprachen haben viele Ähnlichkeiten, aber auch einige Unterschiede, welche in diesem Abschnitt vorgestellt werden.

Compiler

Jeder Kotlin-Release hat zusätzlich zum JVM-Compiler auch Varianten für nativen Code und für Javascript [21]. Der native Compiler Kotlin/Native kompiliert den Code zu nativem Binärcode, der ohne eine virtuelle Maschine direkt auf einer Plattform ausgeführt werden kann. Dabei werden Android, Linux, diverse Apple-Betriebssysteme und Windows unterstützt. Durch Kotlin/Native wird eine beidseitige Interoperabilität zu weiteren Programmiersprachen, wie C/C++, Swift oder Objective-C ermöglicht [22].

Ein Java-Release beinhaltet hingegen nur den JVM-Compiler. Oracle [32] entwickelt aber auch GraalVM [28], welches Java zu einer „Native Executable“ kompilieren kann. Dabei werden Linux, macOS und Windows unterstützt. Die Executable beinhaltet nur den benötigten Code und kann benutzt werden, um leichtgewichtige Container Images zu erstellen. Der Ressourcenverbrauch wird im Gegensatz zur JVM-Version reduziert, die Anwendung startet schneller und ist direkt mit voller Leistung verwendbar [33].

Features

Die Tabelle 2.1 listet spracheigene Features auf und die folgende Auflistung enthält Änderungen, die in Kotlin umgesetzt wurden, um Probleme, die es in Java gibt, zu beheben [20].

Kotlins Änderungen nach [20]

- Das Type-System kontrolliert die Null References
- Keine Raw Types wie zum Beispiel List
- Arrays in Kotlin sind invariant
- Kotlin hat echte Funktionstypen im Gegensatz zu Javas SAM-conversions
- Use-site Varianz ohne Wildcards
- Checked Exceptions wurden entfernt

Einige Änderungen und Features von Kotlin dienen der Fehler- und Exceptionsvermeidung vom Compiler, sodass diese nicht in einer Produktionsumgebung auftauchen. Darunter fallen Änderungen wie zum Beispiel die „use-site Varianz ohne Wildcard“, die „null-safety“, das „Umgehen von Raw Types“ oder „Smart Casts“ [20]. Auf die genaue Funktionsweise der in diesem Unterkapitel erwähnten Features wird nicht eingegangen, da sie für diese Ausarbeitung nicht relevant sind.

Den Gedanken der Fehler- und Exceptionsvermeidung findet man auch unter Entwicklern von Android-Apps. So geben in einer Umfrage 42 von 98 Entwicklern an, ihre App zu Kotlin migriert zu haben, um Fehler zu reduzieren [24]. Des Weiteren scheinen Kotlin-Programme weniger Source Code zu benötigen, zumindest geben 34 Entwickler an, nach der Migration weniger Source Code zu haben. Eine Mischung aus leichter zu lesen, leichter zu verstehen und Wartbarkeit wurde von 23 Entwicklern erwähnt.

In Tabelle 2.1 werden die spracheigenen Features der beiden Programmiersprachen nach [20] aufgelistet. Einige dieser spracheigenen Features waren für die befragten Entwickler auch Gründe, weshalb sie Kotlin vorziehen. So geben in der Umfrage 42 Entwickler an, dass die „null-safety“ ein Grund für den Wechsel zu Kotlin war. Weiterhin werden die „Extension Functions“ von 23 und die „Coroutinen“ von 21 Entwicklern als einen ihrer Gründe angegeben. Kotlins „data classes“ werden noch von 16 und generell mehr Features von 7 Entwicklern als Grund erwähnt [24].

Tabelle 2.1: Spracheigene Features nach [20]

Java	Kotlin
Checked exceptions	Null-safety
Primitive Datentypen	Extension functions
Statische Klassen und Methoden	Coroutines
Generische Wildcard Types	Data classes
Ternärer Operator	String templates
	Properties
	Primary constructors
	First-class delegation
	Type inference for variable and property types
	Singletons
	Declaration-site variance und Type projections
	Range expressions
	Operator overloading
	Companion objects
	Smart casts
	Separate Interfaces für read-only und mutable Collections
	Lambda Ausdrücke + Inline Funktionen = Performante benutzerdefinierte Kontrollstrukturen

Aufgrund der Verbreitung von Java, der Ähnlichkeit und Interoperabilität zwischen Java und Kotlin und dem Wechsel von Entwicklern zu Kotlin wurden diese Sprachen für den Vergleich ausgewählt. Dabei kann untersucht werden, ob die Änderungen, die in Kotlin vorgenommen wurden, eine Auswirkung auf die Energieeffizienz und die Performance haben. Während die Performance oft nur von den Benutzern subjektiv bewertet wird, kann gerade für die Energieeffizienz eine objektive und globale Betrachtung hilfreich sein. Gerade im mobilen App-Bereich, in dem Java und Kotlin vertreten sind, wo Apps hunderttausendfach oder sogar millionenfach heruntergeladen werden, kann eine kleine Differenz schon große Auswirkungen auf den Gesamtstromverbrauch haben.

Damit eine systematische Untersuchung stattfinden kann, werden die zu untersuchenden Bereiche in Operationsdomänen aufgeteilt. Was Operationsdomänen genau sind, wird im nächsten Abschnitt erläutert.

2.2 Operationsdomäne

Operationsdomänen sind Arbeitspakete, welche verschiedene Operationen zu einem Bereich beinhalten. Dabei setzt sich eine Operationsdomäne aus mehreren Grundoperationen zusammen. Diese Grundoperationen sind atomare Bestandteile, die die Implementierung einer Operationsdomäne umsetzen und für andere Operationen benötigt werden. Die folgenden Operationsdomänen können definiert werden und sollten alle grundlegenden Arbeitspakete umfassen: *Algorithmen*, *Berechnungen*, *Collections*, *Dateien*, *Datenbank*, *(De)-Serialisierung*, *Grafik*, *Kommunikation*.
Die *Collections* beinhaltet verschiedene Arten von Collections, wie beispielsweise Listen, Sets oder Maps. In dieser Ausarbeitung wird die Operationsdomäne anhand von Listen getestet, weshalb sie als Operationsdomäne *Listen* erwähnt wird.
Die Operationsdomäne *Algorithmen* ist besonders, da einzelne Algorithmen durch Grundoperationen anderer Operationsdomänen zusammengesetzt werden und keine eigenen Grundoperationen existieren. Verschiedene Algorithmen, wie Berechnungen, Sortier- oder Wegfindungsalgorithmen, haben beispielsweise oft wenig Gemeinsamkeiten. Ein Sortieralgorithmus verwendet zum Beispiel Grundoperationen von der Operationsdomäne *Listen*, während diese Grundoperationen bei einer komplexen Berechnung eher nicht benötigt werden.

Von den aufgelisteten Operationsdomänen werden die *Datenbank*-, *Grafik*- und *Kommunikationsdomäne* nicht untersucht, da bei diesen Operationen externe Faktoren eine große Rolle spielen. Beispielweise kann die gewählte Datenbank-Sprache oder auch die verwendete Bibliothek eine große Rolle spielen. Dadurch würde es unter anderem zu einem Vergleich von Bibliotheken und nicht direkt von den Programmiersprachen kommen. Dies könnte umgangen werden, indem dieselbe Java-Bibliothek verwendet wird, was aber eventuell nicht repräsentativ für Kotlin wäre. Außerdem würde hier dann nur eine der verschiedenen Datenbank-Anbindungen getestet werden. Die Operationsdomäne *Grafik* hat das gleiche Bibliotheksproblem. Dasselbe gilt auch für den Testserver, der für die Kommunikationsdomäne benötigt wird, wobei hier auch noch ein möglicher Einfluss des Netzwerkes betrachtet werden muss.

Des Weiteren wird die Operationsdomäne *Berechnungen* nicht getestet, da eine Messung der Grundoperation durch die geringen Kosten schwierig ist. Die Namen der Operationsdomänen werden in dieser Ausarbeitung in kursiv dargestellt, um zu differenzieren, wann über eine Operationsdomäne geredet wird.

2.2.1 Grundoperationen

Tabelle 2.2 listet die Grundoperationen für die Operationsdomänen *Dateien*, *(De-)Serialisierung* und *Listen* auf, dabei wurde sich an [55] orientiert. Die Vorgehensweise der Studie wird in Unterkapitel 2.4.1 besprochen. Die CRUD-Operationen, also das Erstellen, Lesen, Updaten und Löschen von Elementen, wurden als Grundoperation für die Operationsdomänen *Dateien* und *Listen* definiert. Bei der Operationsdomäne *Listen* wird zwischen sequenziellen und zufälligen Operationen unterschieden. Für die Operationsdomäne *(De-)Serialisierung* wird einen Objekt in ein String und umgekehrt umgewandelt.

Tabelle 2.2: Auflistung der Grundoperationen nach Operationsdomäne

Operationsdomäne	**Grundoperationen**
Dateien	Erstellen, Lesen, Schreiben und Löschen
(De-)Serialisierung	Serialisieren und Deserialisieren eines Objekts/Strings
Listen	Erstellen einer leeren Liste, sequentielles und zufälliges Lesen von Elementen, Einfügen von Elementen am Anfang, am Ende der Liste und an zufälligen Positionen, Verändern von zufälligen Einträgen, sowie sequentielles und zufälliges Löschen von Elementen

2.2.2 Algorithmen

Die Operationsdomäne *Algorithmen* ist wie erwähnt ein Außenseiter, da Algorithmen keine Grundoperationen sind. Diese Operationsdomäne wird dennoch betrachtet, da Algorithmen häufig Anwendung in größeren Systemen finden. Die Operationsdomäne umfasst dabei verschiedene Bereiche. Im folgenden Abschnitt werden die verwendeten Algorithmen mit einer Beschreibung aufgelistet. Zur besseren Darstellung werden die Algorithmen, die in dieser Ausarbeitung untersucht werden, fett hervorgehoben.

Berechnungen

Matrix-Kettenmultiplikation Bei der Matrix-Kettenmultiplikation werden mehrere Matrizen miteinander multipliziert. Die Aufgabe des Algorithmus ist es, die optimale Lösung zur Berechnung und ihre Kosten zu liefern [43].

Kompression

Huffman Encoding Die Huffman-Kodierung ordnet eine Binärcodierung zu Zeichen zu, sodass die verwendeten Bits reduziert werden. Der Algorithmus hat die Aufgabe, eine Tabelle zu generieren, die die Huffman-Kodierung eines Satzes darstellt [42].

Sortieren

Heapsort Es wird eine binäre Heap-Struktur erstellt, was die effiziente Entnahme des größten Elements erlaubt. Das Array wird von hinten nach vorne aufgebaut [45].

Quicksort Der Quicksort-Algorithmus wählt ein Element aus, anhand dessen Unterlisten erstellt werden, die kleiner sind. Wenn diese Unterliste fertig ist, wird diese auf dieselbe Art sortiert. Danach wird ein neues Element gewählt [46].

Stringmanipulation

Tokenization Ein String wird an jeder Stelle geteilt, wo ein nicht-escaptes Separatorzeichen vorkommt [47].

Suche

A* Der Algorithmus sucht den Pfad mit den geringsten Kosten zwischen zwei Knoten. Dabei ist es egal, wie viele Knoten zwischen dem Start- und Endknoten liegen [40].

Binäre Suche Die binäre Suche halbiert den Suchbereich und verkleinert so den Suchraum, bis das Ziel gefunden wurde [41].

Durch diese breite Verteilung der Bereiche sollte eine gute Repräsentation von verschiedenen Algorithmen möglich sein. Im nächsten Abschnitt werden existierende Datenquellen, die für eine Untersuchung benutzt werden können, besprochen.

2.3 Datenquellen

Wenn die Energieeffizienz von verschiedenen Programmiersprachen miteinander verglichen werden soll, ist es sinnvoll, verschiedene, vergleichbare Implementierungen zu verschiedenen Problemen zu haben [34]. Da Programmiersprachen sehr unterschiedlich sein können, kann es sinnvoll sein, vorhandene Datenquellen zu verwenden, statt auf Eigenimplementierungen zurückzugreifen. Dadurch kann Zeit gespart und Fehler in den Implementierungen verhindert werden. Datenquellen sind in diesem Zusammenhang eine Ansammlung von Implementierungen. Dabei muss darauf geachtet werden, dass eine Datenquelle auch zur Untersuchung passt. Sollen zum Beispiel Grundoperationen miteinander verglichen werden, ist eine Ansammlung von grafischen Oberflächen keine sinnvolle Datenquelle. Mit dem Computer Language Benchmarks Game (CLBG) [4] und Rosetta Code [44] gibt es zwei große Datenquellen, die in mehreren Studien verwendet werden. Es können aber auch andere Datenquellen zum Einsatz kommen. Die *AndroidTimeMachine* [2] ist ein Datensatz, der Informationen über verschiedene Open-Source-Apps beinhaltet. Sie kann genutzt werden, um verschiedene Open-Source-Apps zu sammeln [36].

2.3.1 Computer Language Benchmarks Game

Das CLBG wird von [13] und [34] verwendet. Dabei werden Benchmarks mit Lösungen in verschiedenen Programmiersprachen angeboten. Benchmarks müssen objektiv, zuverlässig und reproduzierbar sein [13]. Das Ziel ist das Vergleichen von Implementierungen innerhalb und zwischen Programmiersprachen. Es sind 26 Programmiersprachen im CLBG zu finden, eine Auflistung befindet sich in Tabelle A.1 im Anhang.
Das CLBG beinhaltet ein Framework für das Ausführen, Testen und Vergleichen von Problemlösungen, sowie zu erwartende Werte. Dabei ist ein vielseitiger Satz von bekannten Problemstellungen vorgegeben [34]. Eine Auflistung der 13 Benchmarks ist im Anhang in der Tabelle A.2 zu finden. Auf die Benchmarks wird hier nicht genauer eingegangen, da diese in dieser Ausarbeitung nicht verwendet werden.
Einziges Ziel ist es, die schnellste Lösung für das Problem zu haben, solange die Implementierung die Vorgaben erfüllt. Dadurch werden eventuell nicht die üblichen Programmierpraktiken einer Sprache verwendet. So wird zum Beispiel Lazy Evaluation nicht betrachtet, da diese nicht von jeder Sprache unterstützt wird [35]. Lazy Evaluation ist eine Compiler-Technik, bei der mit dem Auswerten von Ausdrücken gewartet wird, bis dieser Ausdruck auch wirklich verwendet wird [5].

Da das CLBG keine Kotlin-Implementierungen beinhaltet, müssen eigene oder externe Implementierungen verwendet werden. Beispielsweise nutzt [13] das CLBG gemischt mit eigenen Kotlin-Implementierungen.

2.3.2 Rosetta Code

Die zweite Datenquelle ist Rosetta Code, welche von [10] und [35] verwendet wird. Rosetta Code setzt keine Grundstruktur für Lösungen voraus, wodurch es zu einer größeren Varietät kommen kann. Durch die gegebenen Freiheiten lassen sich hier eher übliche Programmierpraktiken wiederfinden. Dabei wurde aber auch festgestellt, dass nicht immer dieselben Arbeitsschritte durchgeführt werden. So haben einige Implementierungen unterschiedliche Ein- und Ausgabemechanismen, wie zum Beispiel Dateien oder einprogrammierte Konstanten. Zusätzlich werden in einigen Implementierungen Listen sortiert, in anderen jedoch nicht [35].
In Rosetta Code gibt es 1 261 Aufgaben, 399 Aufgabenprototypen und Lösungen von 925 Programmiersprachen, wobei nicht jede Aufgabe Lösungen in allen Programmiersprachen beinhaltet [44]. Während das CLBG Tests und die erwarteten Werte beinhaltet, gibt es diese bei Rosetta Code nicht. Die Lösungen eines Problems sind aus verschiedenen Gründen nicht immer nutzbar. So sind einige Lösungen nicht ausführbar, andere verwenden veraltete oder nicht verfügbare Bibliotheken und wieder andere liefern das falsche Ergebnis [35].

Rosetta Code liefert im Vergleich zum CLBG eine größere Auswahl an Problemen und Programmiersprachen auf Kosten von Arbeit, um die Vergleichbarkeit der Lösungen sicherzustellen.

2.4 Untersuchung der Methodiken anderer Studien

In diesem Unterkapitel werden die von verschiedenen Studien verwendeten Methodiken betrachtet.

2.4.1 Messmethodik

Es gibt zwei Methoden, um den Energieverbrauch zu messen. Die indirekte Methode ist feingranular und hat eine hohe Abtastrate, während die direkte Methode grobgranular ist und eine niedrige Abtastrate besitzt. Bei der indirekten Methode wird Software verwendet, um den Energieverbrauch zu schätzen, wobei hier weiterer System Overhead, ungenaue Schätzungen und geringe Interoperabilität Probleme verursachen können [10]. Im Gegensatz dazu misst die direkte Methode den Energieverbrauch der Hardware direkt. Dadurch ist das Messergebnis zwar genauer, es kann aber nur der Energieverbrauch des gesamten Systems gemessen werden [10].

Da Energiemessungen anfällig für den Einfluss von Betriebstemperaturen sind, sollte zwischen Messungen gewartet werden [35]. Dabei wurde die Wartezeit von [35] und [36] auf zwei Minuten gesetzt, während [10] eine Wartezeit von drei Minuten verwendet hat.

Indirekte Methode

In [34] wird mithilfe der system-Funktion in C die Benchmarklösungen ausgeführt, während der system-Aufruf gemessen wird. Die system-Funktion führt die gegebenen Argumente aus, was hier der Befehl zum Ausführen der Benchmarks ist. Die Energie- und Laufzeitmessungen finden gleichzeitig statt, da das gleichzeitige Messen keine statistisch signifikante Auswirkung auf die Messergebnisse hat. Im Gegensatz dazu findet die Arbeitsspeichermessung getrennt statt. Der Energieverbrauch der CPU und des DRAMs wird in Joule gemessen und addiert.

[35] bildet die Basis für [34] und erweitert die Untersuchung auf Rosetta Code. Dazu wird in Rosetta Code nach Algorithmusimplementierungen gesucht, um diese zu vergleichen. Dabei wird darauf geachtet, dass die Implementierungen gewählt werden, die am ähnlichsten zu den Implementierungen der anderen Programmiersprachen sind. Hierbei werden jedoch nicht die Kriterien der Ähnlichkeit erwähnt. Von den ausgewählten Implementierungen wird die I/O normalisiert, da zum Beispiel einige Implementierungen aus Dateien lesen, während andere Konstanten im Source Code haben. Die Tests müssen so gestaltet werden, dass sie einen für Messungen signifikanten Umfang enthalten.

In [36] werden zwei verschiedene Android-Geräte verwendet. Um Java- und Kotlin-Apps zu vergleichen, werden Apps gewählt, die von Java zu Kotlin migriert wurden. Es wird die letzte Java- und die funktional äquivalente Kotlin-Version ausgewählt. Die Nutzung der Apps findet automatisiert statt. Während dieser automatisierten Nutzung werden Messungen durchgeführt. Vor der Testdurchführung werden Hintergrundprozesse geschlossen und die App neu installiert.

In [55] wird von eigens hierfür entwickelten Apps die Grundoperationen von Java, Kotlin und Flutter auf Android getestet. Zur Kompilierung wird Kotlin/Native und der JVM-Compiler verwendet. Vor jeder Testausführung wird eine komplette Garbage Collection forciert. Die Tests werden auf einem Windows-System durchgeführt. Zusätzlich zu den Tests wird auch ein REST-Server für die Kommunikationstests und ein emuliertes Android-Gerät ausgeführt. In Tabelle 2.3 sind die verwendeten Tests und die Durchführung aufgelistet. In der Spalte Laufzeit wird aufgelistet, was für die durchschnittliche Laufzeitberechnung verwendet wird. Verwendete Variablen werden nicht zufällig generiert, es wird aber auch nicht erwähnt, nach welchem Schema. In allen Testfällen außer dem „Start-Up Test“ wird der maximale Arbeitsspeicherverbrauch und die maximale CPU-Nutzung der gesamten Durchführung gemessen.

In [13] werden Benchmarks genutzt, um Java und Kotlin zu vergleichen, dabei werden diese auf einer Virtuellen Maschine ausgeführt und für die JVM kompiliert.

Direkte Methode

[10] verwendet ein externes Messgerät, welches über einen auf einem Raspberry Pi laufenden Server gesteuert wird. Das Messgerät hat dabei eine Genauigkeit von 1,5 % und eine minimale Abtastrate von einer Sekunde. Um die Energiemessung nicht zu beeinflussen, wird darauf geachtet, dass das System neu gestartet wird und Hintergrundprozesse geschlossen werden. Es gibt jedoch trotzdem Probleme mit den Messergebnissen. Einige Aufgaben werden in weniger als einer Sekunde beendet, wodurch es keine Messdaten gibt, da das externe Messgerät eine zu geringe Abtastrate hat. Eine Erläuterung, weshalb die Laufzeit der Tests nicht auf über eine Sekunde verlängert wird, fehlt.

Tabelle 2.3: Testdurchführungen nach [55]

Durchführung	Laufzeit
Start-p RAM wurde 30 s nach Start gemessen	10 Durchläufe
Collection Operations Die einfachste Listenumsetzungen ArrayList verwenden; Das Element ist ein Objekt, was einen Double und einen String beinhaltet	
Sequential Adding 10 000 Elemente nacheinander ans Ende einer ursprüglich leeren Liste anhängen	Alle Operationen
Sequential Reading Alle Elemente einer Liste mit 10 000 Elementen nacheinander lesen	Alle Operationen
Random Reading 10 000 Elemente zufällig aus einer Liste mit 100 000 Elementen nacheinander lesen	Alle Operationen
Random Removal 10 000 Elemente zufällig aus einer Liste mit 100 000 Elementen nacheinander entfernen	Alle Operationen
Filter and Sort Filtern/Sortieren einer Liste mit 10 000 Elementen	10 000 Durchläufe
REST-GET and POST REST-Server in Spring umgesetzt und lokal bereitgestellt; Request Body beinhaltet ein JSON-Objekt mit einem Long, einem Double und drei Strings	Antwortszeit von 10 000 Requests
Database Operations SQLite Datenbank verwendet und bestand aus einen Table mit zwei Integer-type Columns und drei Text-type Columns INSERT Ursprünglich leere Tabelle SELECT ALL Tabelle mit 10 000 Einträgen SELECT ONE Tabelle mit 10 000 Einträgen; WHERE clause UPDATE ONE Tabelle mit 10 000 Einträgen DELETE ONE Ursprüngliche Tabelle mit 10 000 Einträgen	Jeweils 1 000 Befehle
De-/Serialization Objekt mit einem Long, einem Double und drei Strings; Umwandlung von JSON und Objekt	10 000 Durchläufe
File Operations Lesen, Speichern und Löschen; 100 MB Dateien; Arbeitsspeicher als Ziel	100 Durchläufe

Softwaremetriken

Eine Softwaremetrik ist objektiv, wenn die Messung von einem automatisierten Gerät genommen werden kann. Softwaremetriken, wie Laufzeit und Arbeitsspeicherverbrauch, sind für die Untersuchung der Performance sowie der Energieeffizienz interessant [23]. Die Performance ist abhängig von der Implementierung und der Programmiersprache. Performance beschreibt, wie wenig Hardwareressourcen, wie zum Beispiel Arbeitsspeicher oder CPU, verwendet werden. Somit ist auch die Laufzeit eines Programmes Teil der Performance [13].

Tabelle 2.4: Verwendete Softwaremetriken

Messung	**Quellen**
Arbeitsspeicher	
Maximaler Arbeitsspeicher	[13, 34, 35, 36, 55]
Gesamter Arbeitsspeicher	[35]
CPU	
Maximale CPU-Auslastung (%)	[13, 36, 55]
CPU-Zeit	[13]
Energie	[10, 34, 35, 36]
Laufzeit	[10, 34, 35, 36, 55]
Sonstiges	
App-Größe	[36]
Delayed Frames	[36]
Frame Time	[36]
Garbage Collector Calls	[36]
Größe des Source Codes	[13]

Tabelle 2.4 zeigt, dass am häufigsten der maximal verwendete Arbeitsspeicher, die Laufzeit, der Energieverbrauch sowie die maximale CPU-Auslastung gemessen werden.

2.4.2 Werkzeuge

In Tabelle 2.5 werden mögliche externe Messgeräte aufgelistet.

Tabelle 2.5: Hardwarewerkzeuge

Werkzeug	Beschreibung	Quelle
Intel Power Gadget	Misst Energieverbrauch für Intel Prozessoren, Windows und macOS unterstützt	[23]
Watts Up Pro	Misst Energieverbrauch, Keine Softwareeinschränkungen	[10]

Die verwendeten Softwarewerkzeuge der hier vorgestellten Quellen werden in Tabelle 2.6 aufgelistet. Dabei wurde die jRAPL-Programmierschnittstelle (API) und das *time*-Werkzeug von zwei Quellen verwendet, wobei [35] eine Weiterführung der Untersuchung in [34] ist.

Tabelle 2.6: Softwarewerkzeuge

Werkzeug	Beschreibung	Quelle
Android Debugging Bridge	Verschiedene Werkzeuge für Android	[36]
Android Profiler Tool	Teil der integrierten Entwicklungsumgebung *Android Studio*	[55]
jRAPL	API von Intel's Running Average Power Limit (RAPL), misst den Energieverbrauch	[34], [35]
memory_profiler	Python Bibliothek, misst den gesamten Arbeitsspeicherverbrauch	[35]
Popen	Teil des Python Moduls subprocess, misst verwendete Systemressourcen	[13]
time	Kommando für auf Unix basierenden Systemen, misst verwendete Systemressourcen	[34], [35]

Zum Schluss zeigt Tabelle 2.7 eine Auflistung von Software zur Energie- oder Leistungsmessung nach [11], wobei hier mehr Informationen, wie die Zielplattform, die Abtastrate und der Median der Fehlerrate, mit angegeben werden.

Tabelle 2.7: Software zur Leistungs-/Energiemessung nach [11]

Werkzeug	Zielplattform	Abstastrate (ms)	Fehlerrate (%)
Jalen	Linux	500	-
PowerAPI	Linux	500	0,5 - 3
jRAPL	Linux	1	1,13
Jolinar	Linux	500	3
RAPL	Linux	1	3
SoCWatch	Windows, Linux & Android	1 - 1 000	-
PowerGadget	Windows & Linux	1 - 1 000	-
JouleMeter	VMs & Windows	1 000	5
VMeter	VMs	-	6
BitWatts	VMs	500	2
PowerBooter	Android	-	0,8
GreenOracle	Android	-	10
AEP	Android	-	-
PETrA	Android	1 000	0,04

2.4.3 Statistische Analyse-Methoden

Um die statistische Signifikanz der Ergebnisse zu stärken, wird der Durchschnitt der Ergebnisse aus mehreren Durchläufen berechnet [13, 34, 35, 36, 55]. Dabei verwendet [13] das arithmetische Mittel für die rohen Daten und das harmonische Mittel zur Durchschnittsberechnung. Meistens werden zehn Durchläufe verwendet [13, 34, 35]. Von [36] werden 20 Durchläufe verwendet, während die Anzahl der Durchläufe bei [55] je nach Test variiert.

Drei Quellen verwenden neben dem Mittelwert noch weitere statistische Analyse-Methoden. Die Auflistung dieser Methoden ist in Tabelle 2.8 zu finden. Die meisten Tests haben die Untersuchung der Verteilung als Ziel. Einer dieser Tests ist der Shapiro-Wilk-Test, der auf Normalverteilung testet und von allen drei Quellen verwendet wird.

Tabelle 2.8: Verwendete statistische Analyse-Methoden

Test	Anwendungsgebiet	Quelle
Anderson-Darling-Test	Abweichung der Häufigkeitsverteilung von der Hypothese	[35]
Benjamini-Hochberg-Prozedur	Interpretation von p-Werten	[36]
Cliff's δ	Unterschiede auf statistische Signifikanz prüfen	[36]
Mann-Whitney-U-Test	Verteilung von zwei Ergebnissen gleich oder unterschiedlich	[36]
Rea-Parker-Methode	Nominale Klassifikation des Korrelationskoeffizients	[35]
Shapiro-Wilk-Test	Test auf Normalverteilungen	[34, 35, 36]
Spearman'scher Rangkorrelationskoeffizient	Korrelationsart feststellen	[34, 35]

Nur [36] visualisiert die Messwertverteilung. Dabei werden Boxplots und Q-Q-Diagramme benutzt.

In [13] werden die Implementierungen mit den erwarteten Werten des CLBGs verglichen. Dabei wird zuerst der Durchlauf mit dem geringsten Eingabewert verglichen und erst, wenn hier eine Übereinstimmung vorhanden ist, wird die Messung gestartet.
Hierbei muss angemerkt werden, dass eigene Kotlin-Implementierungen verwendet wurden, wodurch das CLBG keine Werte vorgeben konnte. Zusätzlich wurde von [13] nicht erwähnt, aus welchen Gründen die Werte abweichen könnten und nach welchen Kriterien das Herausfiltern erlaubt ist. Die gemessenen CPU-Werte wurden in den Ergebnissen auch nicht vorgestellt. Diese Quelle wird trotzdem verwendet, um das Gesamtbild der Forschung bezüglich eines Vergleiches von Java und Kotlin aufzuzeigen.

2.5 Ergebnisse der Studien

Die von [35] erstellten Ranglisten weisen Unterschiede zwischen CLBG und Rosetta Code auf. Diese Unterschiede sind aber erklärbar und die Ergebnisse sind vergleichbar. Beispielsweise nutzen zum Beispiel einige Implementierungen in Rosetta Code schlecht umgesetzte Lösungen. Die Voraussetzungen einer Aufgabe sind auch nicht immer komplett definiert, wodurch, wie in Unterkapitel 2.3.2 erwähnt, nicht benötigte Arbeitsschritte durchgeführt werden. Es gibt auch Implementierungen mit zusätzlichen oder ineffizienten Datenstrukturen [35].

Tabelle 2.9: Messdaten nach [13]

	Java			**Kotlin**		
Benchmark	Time (s)	Mem. (Kb)	Size (B)	Time (s)	Mem. (Kb)	Size (B)
Fannkuch-redux	0,675	32 924	1 309	1,195	34 528	1 298
Fasta	5,555	44 552	2 473	8,778	35 888	1 591
N-body	12,694	34 080	1 504	12,323	35 696	1 452
Reverse-complement	2,155	620 420	2 183	2,472	642 658	2 136

In Tabelle 2.9 werden die Messwerte von [13] in Bezug auf die CPU-Zeit in Sekunden, den maximalen Arbeitsspeicherverbrauch in Kilobyte und die Größe des Sourcecodes in Byte aufgelistet.

Tabelle 2.10: Delta (%) der Messdaten

Benchmark	**Time (%)**	**Memory (%)**	**Size (%)**
Fannkuch-redux	43,5	4,6	-0,8
Fasta	36,7	-24,1	-55,4
N-body	-3,0	4,5	-3,6
Reverse-complement	12,8	3,4	-2,2

In Tabelle 2.10 ist das Delta der Messwerte aus Tabelle 2.9 zwischen Kotlin und Java in Bezug auf Kotlin in Prozent angegeben:

$$\frac{x_{Kotlin} - x_{Java}}{x_{Kotlin}}$$

In [13] wird auch ein Delta berechnet, jedoch nicht mit festem Bezugspunkt. Aus diesem Grund wird das Delta hier neu berechnet. Betrachtet man das Delta, sieht man, dass der Sourcecode von Java für alle Benchmarks größer ist. Im Fasta-Benchmark fällt dieser Unterschied am größten aus. Java verbraucht nur im Fasta Benchmark mehr Arbeitsspeicher. Die Laufzeit von Java ist nur im N-body Benchmark höher als Kotlins Laufzeit.

Im Folgenden werden die Erkenntnisse von [13] zusammengefasst.
Im Fasta-Benchmark kann beobachtet werden, dass öfters Type-Casting auftritt. Dies zeigt, dass Kotlin bei dieser Arbeit einen bestimmten Overhead hat, was durch die Abhängigkeit zur Java-I/O-API erklärbar wäre.
Beim Fannkuch-Redux-Benchmark manipuliert die Java-Implementierung das int-Array direkt, während die Kotlin-Implementierung Funktionen benutzt, welche Teile des Arrays auf dem Heap duplizieren. Dadurch gibt es auch hier einen Overhead, der sich auf den Arbeitsspeicherverbrauch auswirkt.
Werden, wie bei dem N-body-Benchmark, echte Datentypen verwendet, so arbeiten die Implementierungen auf dieselbe Art.
Die Kotlin-Implementierung des Reverse-complement-Benchmarks nutzt
Java-Bibliotheken, wodurch Kotlin wieder eine Abhängigkeit zur Java-API hat, was einen Einfluss auf die Performance hat.

[55] untersucht die Grundoperationen der Operationsdomänen *Datenbank*, *Listen*, *Dateien*, *Kommunikation* und *(De-)Serialisierung*. Bei diesem Vergleich treten einige Unterschiede auf. So benötigt Kotlin meist weniger CPU-Auslastung, besitzt dafür aber einen höheren RAM-Verbrauch.

Mit Ausnahme der Appgröße unterscheidet sich der Ressourcen- und Energieverbrauch von Java- und Kotlin-Apps nicht statistisch signifikant [36].

3 Methodik

In diesem Kapitel wird die Methodik beschrieben, die verwendet wird, um eine Antwort auf die Forschungsfrage zu finden. Dabei wird der Zusammenhang mit den in Kapitel 2 besprochenen Grundlagen hergestellt. Zuerst wird erklärt, welche Datenquellen verwendet werden. Danach wird der Aufbau des Experiments beschrieben. Zum Schluss wird auf verschiedene Analysemethoden eingegangen, um die Messergebnisse zu untersuchen.

3.1 Datenquellen

Für die Operationsdomänen *Dateien*, *(De-)Serialisierung* und *Listen* werden die in Unterkapitel 2.2.1 erwähnten Grundoperationen mit eigenen Implementierungen umgesetzt.

Für die Operationsdomäne *Algorithmen* werden hingegen Implementierungen aus Rosetta Code genutzt. Hierdurch sollten Fehler in den Implementierungen verringert werden. Zusätzlich sollte es auch eine Zeitersparnis geben, da nicht alle Implementierungen selbst umgesetzt werden müssen, was es erlaubt, mehr Algorithmen zu untersuchen. Das CLBG wird aufgrund der fehlenden Kotlin-Implementierungen nicht verwendet.

Bei der Verwendung von Rosetta Code muss, wie in Unterkapitel 2.3.2 erwähnt, auf die Vergleichbarkeit der Implementierungen geachtet werden. Sollten Implementierungen zu unterschiedlich sein, werden diese angepasst. Dadurch sollten die möglichen Auswirkungen von unterschiedlichen Arbeitsschritten minimiert werden. Dabei kann jedoch nicht garantiert werden, dass jeder Funktionsaufruf von Sprachfeatures gleich umgesetzt ist. Es ist zu erwähnen, dass Neu- oder Eigenimplementierungen fehleranfälliger sind, wenn die umsetzende Person noch keine oder nur wenig Erfahrung mit der jeweiligen Programmiersprache hat.

3.2 Aufbau

In diesem Unterkapitel werden zuerst die verwendeten Softwaremetriken und Messwerkzeuge vorgestellt. Danach wird beschrieben, wie die Messungen durchgeführt werden.

3.2.1 Softwaremetriken

Es werden von den in Tabelle 2.4 erwähnten Softwaremetriken die meistverwendeten Softwaremetriken gemessen. Darunter fallen die benötigte Energie, der maximale Arbeitsspeicher, die maximale CPU-Leistung und die Laufzeit.

Die Energie wird mithilfe der indirekten Methode gemessen. Diese wurde aufgrund der hohen Abtastrate und der breiten Verwendung in anderen Studien gewählt. In Unterkapitel 2.4.2 wurde die RAPL-API und der Wrapper jRAPL für Java erwähnt. Diese haben eine hohe Abtastrate und keine zu große Fehlerrate, wie in Tabelle 2.7 zu sehen ist. Ein weiterer Wrapper von RAPL ist *turbostat* [7]. Dieses Werkzeug sollte eine ähnliche Abtastrate und Fehlerrate besitzen. Zusätzlich ist *turbostat* sprachunabhängig, da die Energie eines Kommandos gemessen wird. Aus diesen Gründen wird *turbostat* verwendet.

Die Leistung wird in Watt gemessen, wobei mehrere Messwerte zur Verfügung stehen. Es wird einmal die gesamte Leistung des Packages, die Leistung des CPU-Anteils, die Leistung des Graphics (GFX)-Anteils und die Leistung des Random Access Memory (Arbeitsspeicher)s (RAMs) gemessen. Das Package ist die komplette Hardwareeinheit, die häufig umgangssprachlich als „CPU“ bezeichnet wird. Darin können verschiedene Hardwarekomponenten enthalten sein, so kann je nach Modell neben der CPU auch ein Grafikchip Teil des Packages sein. Die verbrauchte Energie des Grafikchips wird durch den GFX-Anteil beschrieben. Da keine grafischen Anwendungen getestet werden, ist der GFX-Anteil jedoch vernachlässigbar.

Alle anderen Softwaremetriken werden mit dem *time*-Werkzeug [14] gemessen.
Das Kommandozeilen-Werkzeug *time* wurde von [34, 35] verwendet. Dabei kann die Zeit in Echtzeit, Kernelzeit und Userzeit gemessen werden.
Die Kernelzeit ist die Zeit, die der Prozess im Kernelmode verbracht hat. Mit dem Kernelmode gehen erhöhte Berechtigungen und somit Sicherheitsimplikationen einher. Ein Prozess kann beispielsweise nur direkt auf Hardwareressourcen zugreifen, wenn er sich im Kernelmode befindet.

Aufgrund der Sicherheitsimplikationen wird der Kernelmode für den Prozess nur für die Ausführung der entsprechenden Funktionen aktiviert. Die Betrachtung des Sicherheitsaspekts ist nicht Teil dieser Ausarbeitung. Der Usermode hat weniger Rechte und ist der Standardmodus. Analog zur Kernzelzeit ist daher die Userzeit die Zeit, in der der Prozess im Usermode ausgeführt wird [56].
Die CPU-Auslastung wird mit folgender Formel aus diesen drei Zeitmessungen berechnet, wobei $t_{Userzeit} + t_{Kernelzeit}$ die Gesamtzeit im User- und Kernelmode beschreibt und $t_{Echtzeit}$ die gemessene Echtzeit ist [14]:

$$CPU_{\%} = \frac{t_{Userzeit} + t_{Kernelzeit}}{t_{Echtzeit}}$$

Wenn die CPU nur einen Thread besitzt, liegt der Wert dieser Formel zwischen 0 % und 100 %. Besitzt die CPU mehrere Threads, ist eine maximale CPU-Auslastung von *100 % * Anzahl der Kerne* möglich. Ein Wert über 100 % bedeutet also eine garantierte Beteiligung von mehreren Threads.

3.2.2 Durchführung

Da in dieser Ausarbeitung eine plattformunabhängige Untersuchung stattfinden soll, wird von den in Unterkapitel 2.1.2 vorgestellten Compilern der JVM-Compiler verwendet.
Damit die Unabhängigkeit der Messungen garantiert werden kann, wird jedes Experiment in eine eigene Jar-Datei kompiliert und mithilfe der Kommandozeile ausgeführt und gemessen. Jedes Experiment erhält dabei Inputs über die Kommandozeile, damit sowohl die Java als auch die Kotlin-Implementierung denselben extern gespeicherten Datensatz verwenden.
Da es nicht möglich ist, nur die Grundoperation zu messen, beinhaltet jedes Experiment einen Vorbereitungsmodus, damit der Overhead der Experimente gemessen werden kann. Durch diesen Modus werden alle Aktionen außer der zu messenden Grundoperation durchgeführt. Mit diesem Modus können die Direktkosten berechnet werden. Für die Direktkosten werden aus den Mittelwerten der normalen Messung und der Messung des Vorbereitungsmodus ein relatives Delta, welches in Unterkapitel 3.3 definiert wird, berechnet.

Damit die Experimente automatisiert ausgeführt werden, wird die Subprocess-Bibliothek [37] von Python [38] verwendet. Teil dieser Bibliothek ist auch die von [13] verwendete *Popen*-Funktion. Diese Bibliothek wird benutzt, um über Python eine Kommandozeile aufzurufen und einen Befehl auszuführen.

Bei den Randbedingungen der Messungen wird sich an [55] orientiert. Es werden Listen mit einer Größe von 10 000 Elementen eingesetzt und die Listenumsetzung „ArrayList“ verwendet. Bei den Grundoperationen Lesen, Schreiben und Löschen der Operationsdomäne *Dateien* wird mit 100 MB Dateien gearbeitet. Dabei werden die Dateien jedoch auf die Festplatte geschrieben und haben nicht wie in Tabelle 2.3 den Arbeitsspeicher als Ziel. Diese Änderung soll zu einem reelleren Szenario führen.

In der Operationsdomäne *(De-)Serialisierung* wird ein Objekt benutzt, dessen Attribute aus zwei Strings, drei Long und einem Double bestehen. Das Objekt wird in einen JavaScript Object Notation (JSON)-String umgewandelt. JSON ist ein leicht lesbares Datenformat, das weitverbreitete Verwendung findet.
Die Listen bei der Operationsdomäne *Listen* bestehen aus Objekten, in denen sich ein String, ein Long und ein Double befindet.

Zuerst wird eine Testmessung durchgeführt, um zu bestimmen, ob die Energiemessung und die Ressourcenmessung zum gleichen Zeitpunkt durchgeführt werden können. Dabei wird darauf geachtet, ob die Messungen einander beeinflussen.

Einzeltest
Danach werden die Grundoperationen einzeln ausgeführt und gemessen. Es kommt derselbe Datensatz für alle Wiederholungen zum Einsatz. Dieser Test wird in dieser Ausarbeitung als Einzeltest bezeichnet.

Die Messung der Operationsdomänen *(De-)Serialisierung* und *Listen* werden pro Grundoperation 1 000-mal wiederholt. Die einzelnen Algorithmen werden ebenfalls jeweils 1 000-mal wiederholt. Bei der Operationsdomäne *Dateien* werden die Grundoperationen 100-mal wiederholt.

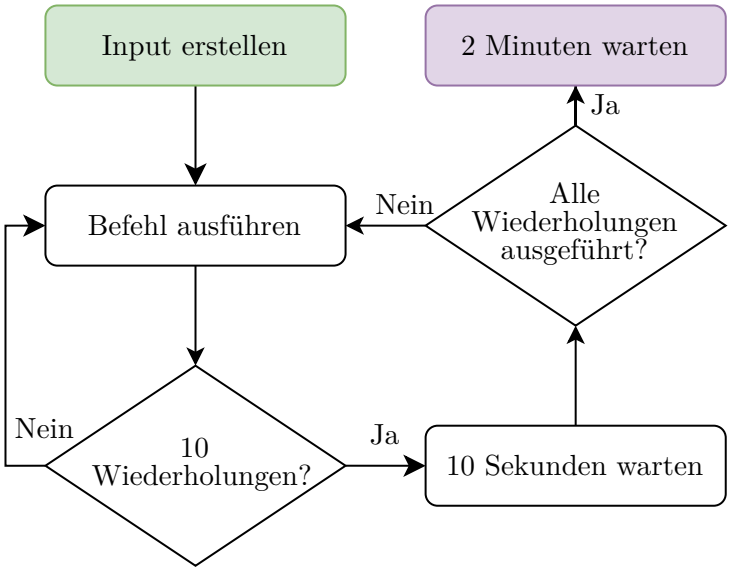


Abbildung 3.1: Ablauf des Einzeltests

Abbildung 3.1 visualisiert den Testablauf des Einzeltests. Es wird zu Beginn ein Datensatz erstellt und für alle Tests verwendet. Es werden immer zehn Wiederholungen in einem Satz zusammengefasst. Nach jedem Satz wird für zehn Sekunden gewartet. Zwischen jedem Experiment werden, wie bei [35] und [36], zwei Minuten gewartet. Durch diese Wartezeiten sollte ein Einfluss durch vorherige Experimente in Bezug zur Temperatur verhindert werden. Warum dies wichtig ist, wurde in Unterkapitel 2.4.1 erläutert. Dabei kann jedoch nicht jeder Einfluss verhindert werden. Es ist beispielsweise nur sehr schwer möglich, eine neutrale und konstante Testumgebung in Bezug zur Außentemperatur zu schaffen, da die Tests zu unterschiedlichen Zeitpunkten an unterschiedlichen Tagen ausgeführt werden.

Kombinationstest

Im nächsten Schritt werden alle Grundoperationen einer Operationsdomäne in einem Test verbunden. Dabei wird dieser Test Kombinationstest genannt. Die Visualisierung befindet sich in Abbildung 3.2. Der Test wird 100-mal wiederholt. Die Anzahl der einzelnen Grundoperationen bleibt dabei gleich. Für jede Wiederholung wird ein anderer Datensatz verwendet. Die Ausführungsreihenfolge, also die Permutation, ist jedoch unterschiedlich. Eine Durchführung könnte zum Beispiel bei der Operationsdomäne *Listen* jeweils eine Grundoperation ausführen und dann wieder von vorne beginnen. Die nächste Durchführung könnte hingegen einige Operationen zu Beginn häufiger ausführen und andere erst zum Schluss.

Diese Herangehensweise wurde gewählt, um eine reelle Anwendung einer Operationsdomäne zu simulieren.
Die für die Messung verwendeten Permutationen werden zufällig bestimmt. Dabei kann jede Grundoperation 0- bis 10-mal wiederholt werden, bevor die nächste Grundoperation an der Reihe ist. Es werden dieselben 100 Permutationen für beide Sprachen verwendet. Zwischen den Experimenten wird wie bei den Einzeltests zwei Minuten gewartet. Im Gegensatz zu den Einzeltests wird beim Kombinationstest zwischen den einzelnen Wiederholungen fünf Sekunden gewartet.

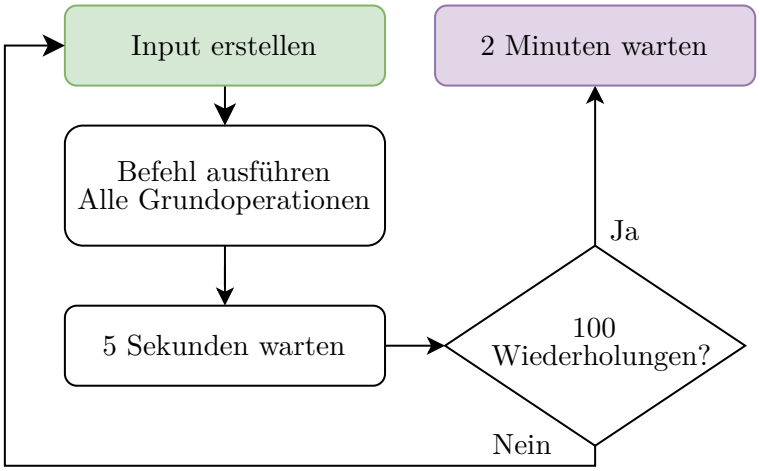


Abbildung 3.2: Ablauf des Kombinationstests

Variationstest

Da die Operationsdomäne *Algorithmen*, wie in Unterkapitel 2.2.1 erwähnt, keine Grundoperationen beinhaltet, werden die Algorithmen noch einmal einzeln getestet und 100-mal mit unterschiedlichen Datensätzen wiederholt. Der restliche Testablauf ist wie beim Kombinationstest und wird in Abbildung 3.3 visualisiert. Die Bezeichnung für diesen Test ist Variationstest.

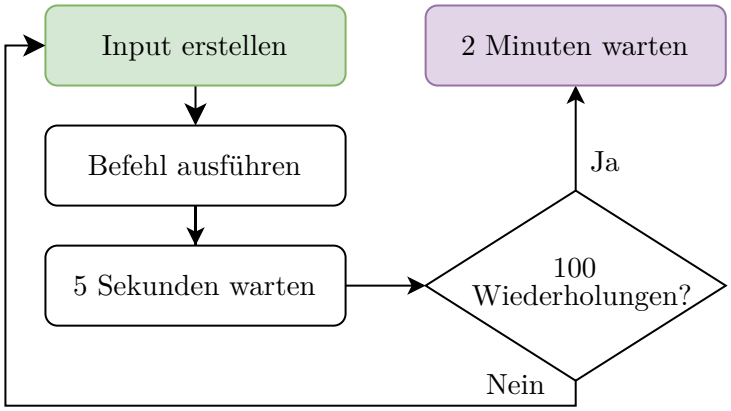


Abbildung 3.3: Ablauf des Variationstests

Variablen

Alle Variablen werden schematisch generiert. Es wird derselbe Datensatz für beide Programmiersprachen verwendet.

Dateien Es werden 100 MB an Daten erzeugt. Diese Daten werden dann auf die Anzahl der Zeilen aufgeteilt und mit Kleinbuchstaben aufgefüllt. Die Anzahl der Zeilen beträgt im Einzeltest 100 und wird im Kombinationstest für jede Wiederholung zufällig bestimmt. Dabei wird ein Wert zwischen 10 und 10 000 gewählt. Die Zeilenanzahl wird als Seed für den Zufallsgenerator genommen, um die Dateien für den Lese- und Löschtest zu erzeugen.

Listen Für die Erstellung der Listen-Objekte wird eine Zahl verwendet. Dabei startet die Zahl pro Datensatz bei null und wird fortlaufend um eins erhöht. Der String besteht aus der Nummer als Text, sowohl die long- als auch die double-Variable wird gleich der Zahl gesetzt.

(De-)Serialisierung Das Serialisierungsobjekt wird auf zwei Arten generiert. Bei der ersten Art, die bei der Serialisierung verwendet wird, wird anhand einer Zahl die Variablen generiert. Im Einzeltest ist diese Zahl zehn, im Kombinationstest startet die Zahl bei null und wird schrittweise um eins erhöht. Dabei haben die Variablen folgendes Format:

- **String 1** "Erster String " + Zahl als String
- **String 2** "Zweiter String " + Zahl als String
- **long 1** Zahl
- **long 2** 2 * long 1
- **long 3** 3 * long 2
- **double** long 3 / 7

Bei der Deserialisierung werden die Zahlen auf dieselbe Art erzeugt, die Strings sind aber zufällige Zeichen und haben eine Länge von zehn. Für den Einzeltest wird auch die Zahl zehn verwendet. Beim Kombinationstest werden zehn Objekte mit Zahlen von 0 bis 9 erstellt.

Binäre Suche Es wird eine sortierte Liste generiert. Die Werte starten bei eins und jeder nachfolgende Wert ist um eine zufällige Zahl zwischen 0 und 5 größer als der vorherige. Der gesuchte Wert ist beim Einzeltest der Wert am Index 150. Für den Kombinationstest befindet sich der gesuchte Wert jeweils 25-mal in der unteren Hälfte, in der oberen Hälfte, außerhalb der Liste mit einem geringeren Wert und außerhalb der Liste mit einem höheren Wert.

Heapsort und Quicksort Es wird eine nicht sortierte Liste mit zufälligen Werten zwischen -10 000 und 10 000 erstellt.

Huffman Encoding Es wird ein String der Länge 30 mit zufälligen Zeichen generiert.

Matrix-Ketten-Manipulation Es wird eine Liste mit 100 Dimensionen mit Werten zwischen 1 und 100 generiert.

Tokenization Beim Einzeltest werden fünf Strings der Länge fünf mit fünf Separatorzeichen und einem escapten Separatorzeichen verbunden. Dadurch werden verschiedene Fälle betrachtet. Das wäre einmal das Trennen von Strings, dann zwei Separatorzeichen direkt hintereinander und der dritte Fall wäre das Verhindern der Trennung durch das Escape-Zeichen. Der String hat dabei folgendes Format: "*String | Separator | String | Separator | String | Separator | String | Escapter Separator | String | Separator | Separator*"
Im Kombinationstest wird ein String der Länge 30 mit zufälligen Zeichen generiert. Die Zeichen umfassen die Kleinbuchstaben *a - n*, das Separatorzeichen *;* und das Escapezeichen^.

Für die **Deserialisierung**, das **Huffman Encoding** und den Einzeltest bei der **Tokenization** werden dieselben Zeichen verwendet. Die Zeichen umfassen das Alphabet mit Groß- und Kleinbuchstaben, das Leerzeichen und folgende Satzzeichen [.,*!?*].

3.3 Analysemethoden

Aus den Messergebnissen des Einzeltests werden verschiedene statistische Werte berechnet. Das arithmetische Mittel und der Median werden benutzt, um die zentrale Tendenz der Messergebnisse darzustellen. Dabei ist der Median hilfreich, wenn die Messwerte wenige, besonders starke Ausläufer besitzen. Der Median ist gegenüber solchen Fluktuationen resistenter. Die Varianz und Standardabweichung sind ein Maß, um die Streuung der Daten zu beschreiben, während der Standardfehler des Mittels genutzt werden kann, um die Genauigkeit der Mittelwerte abschätzen zu können [25]. In Tabelle 3.1 werden die statistischen Werte noch einmal mit ihren Formeln aufgelistet.
Um die Differenz zwischen zwei Werten zu betrachten, kann das absolute und das relative Delta verwendet werden [26].

Deltas nach [26]:

$$\delta_{Absolut} = x - y$$

$$\delta_{Relativ} = 100 * \frac{\delta_{Absolut}}{x}$$

Tabelle 3.1: Statistische Werte nach [25]

Statistischer Wert	**Anwendungsgebiet**	**Berechnung**
Arithmetisches Mittel	zentrale Tendenz der Messergebnisse darstellen	$\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$
Median	zentrale Tendenz der Messergebnisse darstellen	Wert in der Mitte einer sortierten Liste auswählen. Wenn gerade Anzahl: Arithmetisches Mittel der beiden mittleren Werte
Varianz	Beschreibung der Streuung	$\sigma^2 = \frac{1}{n-1}\sum_{k=1}^{n}(x_k - \bar{x})^2$
Standardabweichung	Beschreibung der Streuung	$\sigma = \sqrt{\sigma^2}$
Standardfehler des Mittels	Beschreibung der Genauigkeit der Schätzung des Mittels	$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$

Für die Kombinationstests werden weitere statistische Tests hinzugezogen, da durch die gesteigerte Komplexität eine höhere Variation erwartbar ist.

Um die Verteilung der Messergebnisse zu untersuchen, wird der Shapiro-Wilk-Test angewandt. Dieser Test stellt fest, ob die Messergebnisse normalverteilt sind. Dies ist insofern relevant, da die Wahl der darauffolgenden Tests davon abhängt, ob die Messergebnisse normalverteilt sind oder nicht. Der genaue Ablauf des Shapiro-Wilk-Tests wird im Anhang unter Ablauf A.1 dargestellt. Es wird eine gewichtete Summe der Messwerte und die Varianz verwendet, um eine Teststatistik zu berechnen. Diese Teststatistik wird mit einer Tabelle verglichen, um die Wahrscheinlichkeit einer Normalverteilung zu bekommen [50]. Hierbei muss angemerkt werden, dass die Tabellen in [50], die für die Vergleiche verwendet werden, nur für bis zu 50 Messwerte reichen. Implementierungen, wie zum Beispiel die Umsetzung von *SciPy* [48] bieten den Shapiro-Wilk-Test auch für viel größere Mengen von Messergebnissen an. Die *SciPy*-Implementierung bietet auch einen p-Wert an, der als alternatives Ergebnis Entscheidungshilfe bieten kann. Dabei bleibt die Teststatistik auch für $n > 5000$ genau, der p-Wert verliert allerdings an Genauigkeit [49].

Sind die beiden Messergebnisse nicht normalverteilt, wird überprüft, ob dieselbe Wahrscheinlichkeitsverteilung vorliegt. Dies kann mit dem Kolmogorow-Smirnow-Test geschehen. Dieser wird in Ablauf A.3 im Anhang beschrieben und sucht nach der größten Differenz zwischen zwei kumulativen Verteilungen. Aus dieser Differenz wird eine Teststatistik berechnet, mit der die Wahrscheinlichkeit einer gleichen Verteilung bestimmt und die Ähnlichkeit bewertet werden kann [6].

Wenn die Messergebnisse beider Sprachen dieselbe Verteilung haben, kann mithilfe des Welch's t-Tests, dessen Ablauf im Anhang unter Ablauf A.3 genauer beschrieben wird, die einzelnen Softwaremetriken untersucht werden. Dabei mithilfe der Varianz die Anzahl der Freiheitsgrade und eine Teststatistik berechnet. Mit diesen beiden Werten kann ein p-Wert aus einer Tabelle entnommen werden, der bei der Entscheidung hilft, ob für eine Softwaremetrik beide Sprachen denselben Mittelwert haben [25].

Mithilfe des Spearman'schen Rangkorrelationskoeffizienten kann die Korrelation zwischen den Java- und Kotlin-Messwerten festgestellt werden. Der Spearman'schen Rangkorrelationskoeffizient, welcher in [34, 35] Verwendung findet, wird benutzt, um eine monotone Korrelation zwischen zwei geordneten Variablen festzustellen. Das bedeutet, steigt die eine Variable, steigt oder fällt die andere. Ist dies nicht konsistent, so ist die Beziehung nicht monoton und diese Methode kann nicht verwendet werden. Die Nullhypothese ist, dass es keine Korrelation gibt.
Um den Rangkorrelationskoeffizienten zu berechnen, wird jedem Messwert ein Rang zugeordnet und dann aus den Differenzen zwischen den einzelnen Rängen und dem Durchschnittsrang eine Teststatistik berechnet. Anhand dieser Teststatistik kann bewertet werden, ob eine monotone Korrelation existiert [25]. Der genaue Ablauf ist im Anhang in Ablauf A.4 dargestellt.

4 Umsetzung

In diesem Kapitel werden die wichtigsten Punkte der Umsetzung der einzelnen Experimente sowie aufgetretene Besonderheiten besprochen. Zuerst werden die Implementierungen betrachtet. Danach wird die verwendete Hard- und Software aufgezeigt. Zum Schluss wird sich mit der Testmessung auseinandergesetzt, um zu entscheiden, ob eine kombinierte Messung der Experimente möglich ist.

4.1 Implementierung

Zuerst werden die Algorithmen und die Besonderheiten in der Implementierung besprochen. Danach wird die Implementierung der anderen Operationsdomänen besprochen.

4.1.1 Algorithmen

Die Implementierungen von Rosetta Code lassen sich in verschiedene Gruppen einteilen. Tabelle 4.1 liefert eine Übersicht über die Gruppen und die beinhalteten Algorithmen.

Gruppe	Algorithmen
Vergleichbare Implementierungen	Heapsort, Matrix-Ketten-Multiplikation
Leichte Anpassungen	Binäre Suche, Huffman Encoding
Neuimplementierung	A*-Algorithmus, Quicksort, Tokenization

Tabelle 4.1: Ähnlichkeit der Algorithmus-Implementierungen zwischen Java und Kotlin

Die erste Gruppe besteht aus den **vergleichbaren Implementierungen**, die einfach so übernommen werden konnten. Darunter fallen der **Heapsort-Algorithmus** und die **Matrix-Ketten-Multiplikation**.

Dann gibt es in der zweiten Gruppe die Implementierungen, die **leichte Anpassungen** benötigen, um die richtigen Ergebnisse zu liefern oder eine bessere Vergleichbarkeit durch die Ausführung der gleichen Arbeitsschritte zu schaffen. Die Vergleichbarkeit in Bezug zu Rosetta Code wird in Unterkapitel 2.3.2 besprochen. Durch eine Anpassung sollten gemessene Unterschiede aufgrund der Programmiersprachen auftreten. In dieser Gruppe befinden sich die **binäre Suche** und das **Huffman Encoding**.

In der **binären Suche** weisen sowohl die Java als auch die Kotlin-Implementierung einen Fehler in der rekursiven binären Suche auf. Dieser tritt auf, wenn eine Zahl gesucht wird, die größer als die letzte Zahl im Suchbereich ist.
In der iterativen Kotlin-Implementierung ist eine Berechnung anders als in der entsprechenden Java-Implementierung, weshalb sie angepasst wird. Da Javas Implementierung nur auf einem Datentyp funktioniert, wurde Kotlins generische Lösung eingeschränkt.

Beim **Huffman Encoding** wird in der Kotlin-Implementierung eine benötigte Variable dynamisch berechnet und ist nicht wie in Java statisch.

In der letzten Gruppe sind die Implementierungen so unterschiedlich, dass eine **Neuimplementierung** zum besseren Vergleich sinnvoll ist. Hier befinden sich der **A*-Algorithmus**, der **Quicksort-Algorithmus** und die **Tokenization**.
Der **A*-Algorithmus** hat noch zusätzlich das Problem, dass die Java-Implementierung fehlerhaft ist.

Die Implementierungen des **A*-Algorithmus** nutzen unterschiedliche Karten und Mechaniken. Hier wird die Kotlin-Implementierung an die Java-Implementierung angepasst und neu programmiert.
Die Java-Implementierung ist auch problematisch, da die verwendete Methode „findNeighborInList“ nicht existiert und die compareTo-Methode fehlerhaft ist.
Bei den Messungen hat der **A*-Algorithmus** eine hohe Variation in den Messwerten aufgezeigt. Dabei variiert unter anderem die Laufzeit stark.
Es wurde ein 5x5 Feld mit zufälligen Werten von 0 bis 20 initialisiert. Ein Durchlauf benötigt mal weniger als eine Sekunde, ein anderer Durchlauf kann über zwei Stunden benötigen. Diese Variation ist für die Auswertung und die Analyse problematisch. Es können zwar direkte Vergleiche der einzelnen Messwerte gezogen werden, aber einige statistische Tests wie der Shapiro-Wilk-Test könnten je nach Wiederholungsanzahl weniger aussagekräftig sein. Außerdem ist diese Variation atypisch für den **A*-Algorithmus**.

Der Algorithmus wurde, um eine Ursache zu finden, auf einer anderen Art von Feld getestet. Dazu wurde ein Labyrinth erzeugt, in dem die Wege geringe Kosten und die Wände hohe Kosten haben. In diesen Labyrinthen ist die Frequenz der langen Durchläufe reduziert, tritt aber trotzdem auf.
Die Ursache für die langen Laufzeiten scheint ein häufiges Betreten einzelner Zellen zu sein, was gerade in Sackgassen in der Nähe zum Ziel auftritt.
Ein möglicher Grund für die hohe Laufzeit könnte eine fehlerhafte oder wenig optimierte Umsetzung in Rosetta Code sein. Es ist ebenfalls nicht auszuschließen, dass die Neuimplementierung der findNeighborInList-Methode die Ursache für diese starke Varianz ist. Die findNeighborInList-Methode wurde danach so angepasst, dass nur die Koordinaten verglichen werden. Dadurch wurde die Laufzeit verkürzt und die großen Variationen eliminiert. Durch diese Änderung wird eventuell nicht der kürzeste Pfad gefunden, wenn die Wegkosten nicht statisch zu einer Zelle sind, sondern von dynamischen Änderungen beeinflusst werden. Dies ist für den hier verwendeten Test aber nicht relevant, da dieser Fall nicht auftritt und beide Implementierungen auf dieselbe Art funktionieren. Ist die Implementierung nicht korrekt, so befindet sich in beiden Programmiersprachen der gleiche Fehler.

Beim **Quicksort-Algorithmus** sind die Implementierungen sehr unterschiedlich. Die Kotlin-Implementierung wird an die Java-Implementierung angepasst und neu implementiert.

Beide Implementierungen des **Tokenization** haben andere Ansätze. In diesem Experiment wird der manuelle Ansatz von Java verwendet.

4.1.2 Operationsdomäne

Für die **Operationsdomäne *Dateien*** werden die Daten zum Beschreiben der Dateien in der Jar-Datei erzeugt. Beim Kombinationsexperiment wird darauf geachtet, unterschiedlich viele Zeilen in den Dokumenten zu haben, damit die Programmiersprachen sowohl auf wenige Leseoperationen als auch auf viele Leseoperationen getestet werden. Dadurch kann überprüft werden, ob es zu einem Unterschied im Ressourcen- oder Energieverbrauch kommt.

Bei der **Operationsdomäne** ***(De-)Serialisierung*** wird keine Bibliothek verwendet, damit nicht unterschiedliche Bibliotheken verglichen werden. Selbst wenn dieselbe Bibliothek verwendet wird, könnte es zu einer Benachteiligung einer Programmiersprache kommen, wenn erst eine Übersetzung in eine andere Sprache geschehen muss oder ein Zugriff über eine API erfolgt, die die andere Programmiersprache nicht hat.
Es wird kein allgemeingültiger Parser geschrieben, da der Fokus hier erst mal nur auf der direkten Umwandlung „Objekt zu Text“ und „Text zu Objekt“ liegt.

4.2 Verwendete Hard- und Software

In diesem Unterkapitel wird die verwendete Hard- und Software sowie die verwendeten Compiler aufgezeigt.

CPU i5-6200U | 2.30GHz, 2 Kerne, 4 Threads

RAM 8 GB

Festplatte 256 GB SSD

Betriebssystem Linux Mint 21.3 | Kernel: 5.15.0-102-generic

JDK OpenJDK 22 [31]

Kotlin-Compiler Version 1.9.23-release-779

Mess-Werkzeuge *time* und *turbostat*

Es werden keine Compiler Flags geändert, damit das Ergebnis nicht durch bestimmte Einstellungen verfälscht wird. Außerdem kann erwartet werden, dass die Standardeinstellungen ein ausgeglichenes Ergebnis liefern, was von vielen Programmierern verwendet wird.

Die CPU hat durch ihre zwei Kerne vier Threads, wodurch bei der Messung der CPU-Auslastung maximal 400 % erreicht werden können. Für die Betrachtung der Ergebnisse wird die CPU-Auslastung normalisiert, sodass nur 100 % möglich sind.

Die automatisierte Experimentausführung und die Datenverarbeitung werden mit Python umgesetzt, wobei die Dateierstellung für die Grundoperationen Lesen und Löschen der Operationsdomäne *Dateien* in C [15] implementiert wurde.
Die statistischen Tests werden mithilfe von SciPy [48] ausgeführt.

4.3 Testmessung

Die Testmessung wurde anhand des (De-)Serialisierungsexperiments durchgeführt. Dabei wurde sowohl die Serialisierung als auch die Deserialisierung in der Java-Implementierung 10 000-mal mit denselben Inputs ausgeführt und gemessen. Bei der ersten Messung wurde mit dem *turbostat*-Werkzeug das *time*-Werkzeug gemessen, was wiederum das Experiment ausgeführt und gemessen hat. Danach wurde das *turbostat*-Werkzeug getrennt ausgeführt. Da das *time*-Werkzeug die Messung direkt auf der Jar-Datei ausführt, sollte es hier keinen Unterschied geben.

Zuerst wurde das arithmetische Mittel, der Median und die Standardabweichung der Messergebnisse bestimmt. Der Mittelwert und die Standardabweichung beider Messungen sind in Abbildung 4.1 zu finden. Das Symbol im Mittelpunkt stellt den Mittelwert dar und die Linien zeigen das Intervall $[\bar{x} - \sigma, \bar{x} + \sigma]$. Die kombinierte Messung wird durch einen Kreis dargestellt, während ein Kreuz für die getrennte Messung verwendet wird.

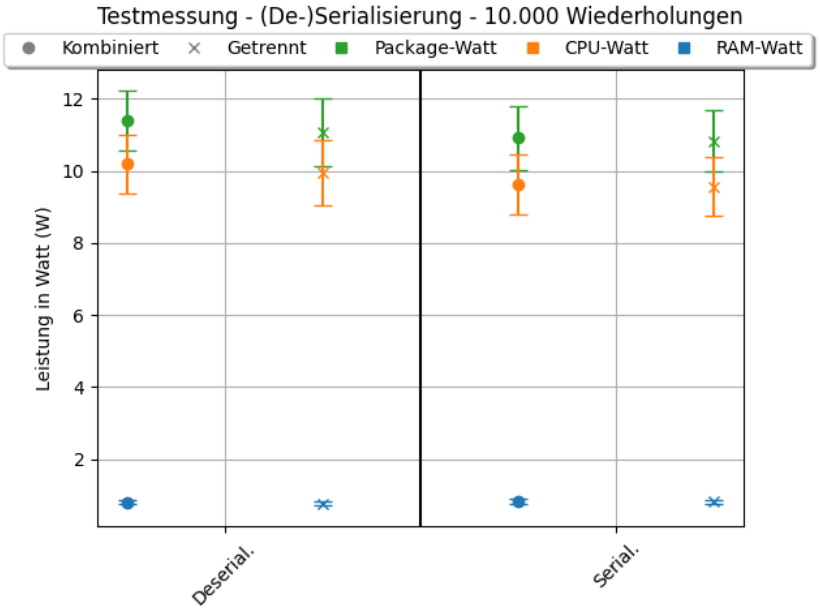


Abbildung 4.1: Testmessung für den Vergleich einer kombinierten und einer getrennten Messung

Die Unterschiede zwischen den Mittelwerten fallen relativ klein aus. Die relativen Deltas der Mittelwerte liegen bei der Deserialisierung zwischen 2,0 % und 2,6 % und bei der Serialisierung zwischen 0,4 % und 1,2 %. Eine Verwendung der Mediane zur Berechnung ergibt keinen nennenswerten Unterschied. Um die Ergebnisse deuten zu können, muss das *turbostat*-Werkzeug genauer betrachtet werden.
Die Fehlerrate von *RAPL* hat laut Tabelle 2.7 einen Median von 3 %, während der Wrapper *jRAPL* eine Fehlerrate von 1,13 % besitzt. Für das *turbostat*-Werkzeug ist die Fehlerrate nicht bekannt. Da es aber auch ein Wrapper von *RAPL* ist, kann davon ausgegangen werden, dass es eine ähnliche Fehlerrate besitzt.

Alle relativen Deltas liegen unter der Fehlerrate von *RAPL* und, mit Ausnahme der RAM-Watt-Messung, liegen alle Werte bei der Serialisierung unter einem Prozent, also unter der Fehlerrate von *jRAPL*. Dabei muss aber angemerkt werden, dass es unwahrscheinlich ist, dass die relativen Deltas nur durch Messfehler zustande kommen. Eine Beeinflussung der Messung durch das *time*-Werkzeug ist aber auch nicht eindeutig, da es einen bemerkbaren Unterschied zwischen den relativen Deltas der Deserialisierung und Serialisierung gibt. Bei der Deserialisierung würde das *time*-Werkzeug relativ mehr Leistung als in der Serialisierung benötigen. Eine weitere Möglichkeit wäre eine Variation zwischen den Tests durch andere externe Faktoren. Die Ursachenforschung für diese Unterschiede zwischen der Deserialisierung und der Serialisierung würde über diese Bachelorarbeit hinausgehen.
Sollte es eine Beeinflussung der Messung durch das kombinierte Messen geben, so ist diese in der Testmessung so gering, dass sie nicht eindeutig festgestellt werden kann. Aus diesem Grund wird für die weiteren Messungen die kombinierte Messmethodik verwendet.

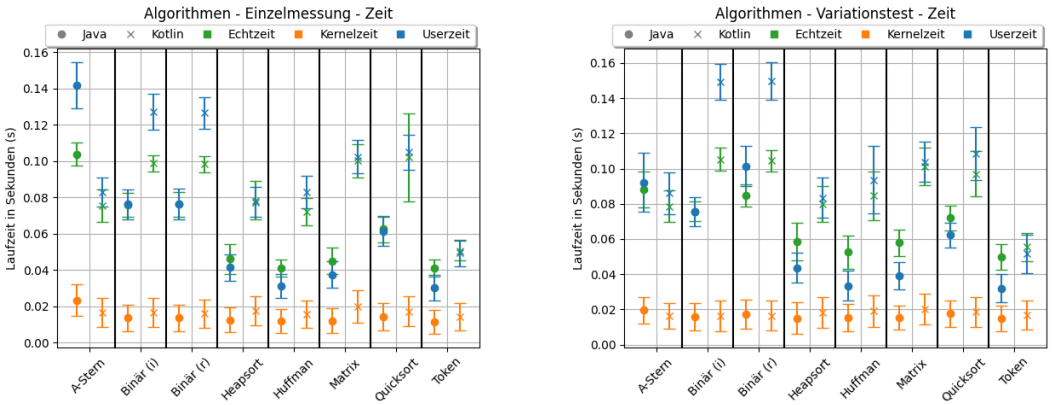
5 Ergebnisse

In diesem Kapitel werden die Ergebnisse der einzelnen Experimente vorgestellt. Dabei werden die Ergebnisse in die Operationsdomänen aufgeteilt und nacheinander betrachtet.
Die Ergebnisse der Messung werden wieder mithilfe des Mittelwerts und der Standardabweichung dargestellt. Für Java wird als Symbol ein Kreis und für Kotlin ein Kreuz verwendet.

5.1 Algorithmen

In Abbildung 5.1 sind die Algorithmus-Laufzeiten dargestellt. Dabei sind in Abbildung 5.1a die Laufzeiten des Einzeltests zu finden, während in Abbildung 5.1b die Laufzeiten des Variationstests dargestellt sind. Bei den Laufzeiten wird, wie in Unterkapitel 3.2.1 erläutert, zwischen Echtzeit, Kernelzeit und Userzeit unterschieden.
Wenn man sich die Messungen des Einzeltests ansieht, erkennt man, dass bis auf die **A*-Algorithmus**-Implementierung Kotlin höhere Laufzeiten aufweist. Tendenziell verbringen Kotlin-Prozesse mehr Zeit im Usermode, als in Echtzeit, während bei Java die umgekehrte Tendenz beobachtet werden kann. Die verbrachte Zeit ist im Kernelmode hingegen gleich stark ausgeprägt und stabil, was durch die niedrige Varianz angedeutet wird.
Der Vergleich mit den Messungen zu dem Variationstest, dessen Ergebnisse in Abbildung 5.1b zu finden sind, zeigt, dass sich vor allem die Echt- und Userzeitmesswerte geändert haben.
Beim **A*-Algorithmus** ist eine dramatische Verbesserung von der Java-Implementierung in Bezug zur Echt- und Userzeit bemerkbar. Bei der rekursiven **binären Suche**, dem **Heapsort** und dem **Huffman Encoding** weist Java eine Verschlechterung in der Echt- und Userzeit auf.

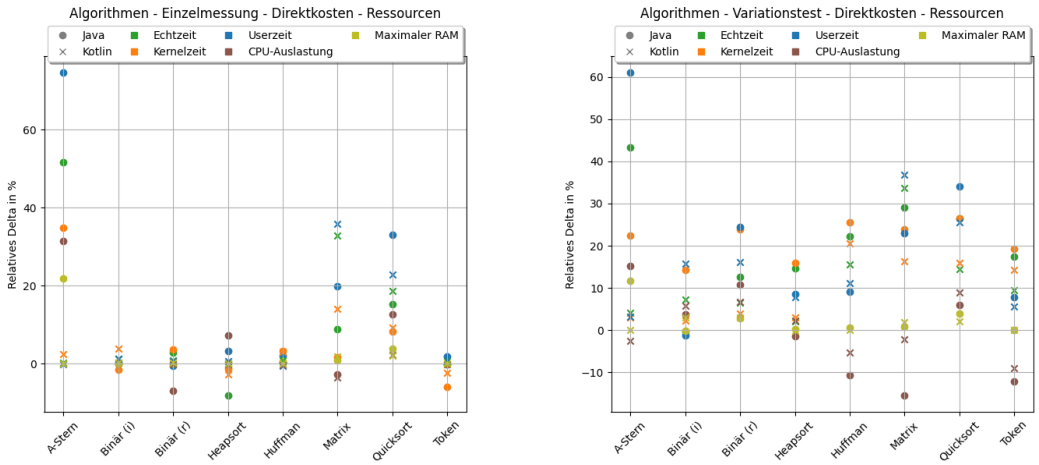
Die rekursive **binäre Suche** hat bei Java zusätzlich noch eine weitaus höhere Userzeit als Echtzeit. Diese Ergebnisse sind weder im Einzeltest noch in der iterativen **binären Suche** widergespiegelt.
Es ist auch zu erkennen, dass Java beim Variationstest eine höhere Echtzeit bei der **Matrix-Ketten-Multiplikation** aufweist, während dies bei der **binären Suche** und dem **Huffman Encoding** für Kotlin bei der Echt- und Userzeit der Fall ist.
Die **Quicksort**-Mittelwerte der Echt- und Userzeiten liegen weiter auseinander und die Echtzeitmessung bei Kotlin hat eine geringere Standardabweichung.
Bei der **Tokenization** hat Java eine geringere Echtzeit und Kotlin eine geringere Standardabweichung bei der Echt- und Userzeit.



(a) Gleicher Input (b) Variabler Input

Abbildung 5.1: Laufzeitmessungen der Algorithmen

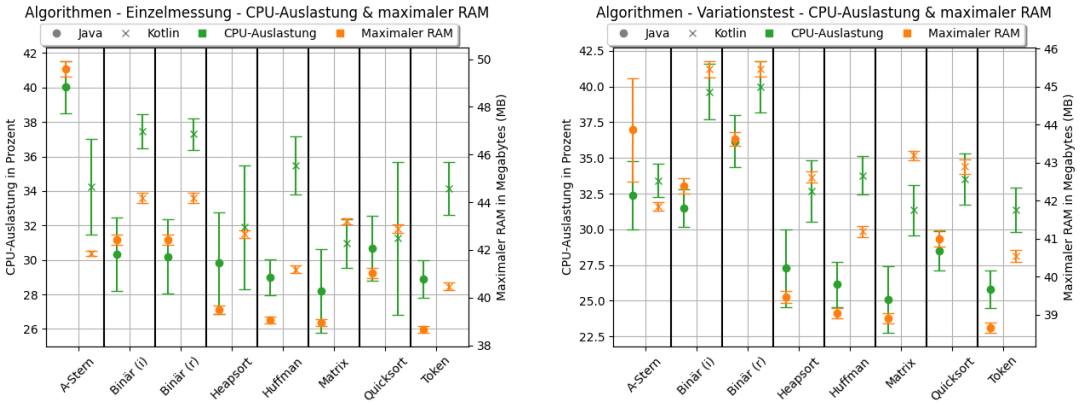
Zur besseren Verdeutlichung der Unterschiede werden in Abbildung 5.2 die Ressourcenmessungen als relative Deltas der Direktkosten für alle Algorithmen dargestellt. Die Berechnung des relativen Deltas wurde in Unterkapitel 3.3 definiert und gibt in diesem Fall an, welchen Anteil die gemessenen Algorithmen an den Gesamtkosten haben. Beim **A*-Algorithmus** hat Java für alle Laufzeiten einen hohen Anteil. Bei der **Matrix-Ketten-Multiplikation** gibt es höhere relative Deltas, wobei diese bei Kotlin stärker ausfallen. Ansonsten fallen beim **Quicksort** die höheren Direktkostenanteile auf.
Diese Beobachtungen sind auch im Variationstest vertreten und die Direktkostenanteile sind meistens gestiegen. Die Echt- und Userzeit beim **A*-Algorithmus** ist hingegen gesunken. Dies könnte ein Hinweis auf den Einfluss des Inputs auf die Direktkosten sein.
Im Einzeltest gibt es bei dem **Heapsort** und der **Tokenization** einige negative relative Deltas in Bezug zur Laufzeit, was bedeutet, dass der gemessene Overhead in diesen Fällen größer ist als die Messung mit Overhead und Algorithmusausführung.



(a) Gleicher Input (b) Variabler Input

Abbildung 5.2: Ressourcendirektkosten der Algorithmen

In Abbildung 5.3 werden die CPU-Auslastung und der maximale RAM der Algorithmen dargestellt. Außer beim **A*-Algorithmus** haben die Kotlin-Implementierungen im Einzeltest einen höheren Bedarf an RAM und CPU-Auslastung. Dieser höhere Bedarf fällt bei beiden Implementierungen der **Binären Suche** am größten aus.
Die CPU-Auslastung ist bei dem **A*-Algorithmus**, dem **Huffman Encoding** und der **Tokenization** für beide Programmiersprachen im Variationstest gesunken. Gerade beim **A*-Algorithmus** ist Java stark gefallen, sodass der Mittelwert kleiner als der Kotlin-Mittelwert ist.
Bei dem **Heapsort**, der **Matrix-Ketten-Multiplikation** und dem **Quicksort** ist die CPU-Auslastung nur bei Java gesunken.
Bei Kotlin hingegen ist die CPU-Auslastung beim **Quicksort** gestiegen.
Für beide Programmiersprachen ist die CPU-Auslastung bei der **binären Suche**, vor allem der rekursiven, gestiegen.
Für den Einzeltest sind die Direktkosten, wie in Abbildung 5.2a zu sehen ist, sehr gering oder sogar negativ. Dieser Trend tritt beim Variationstest verstärkt auf, was bedeutet, dass eine Auswertung der Direktkosten in Bezug zu den Algorithmen nicht möglich ist. Nur die CPU-Auslastung hat im Variationstest größere Unterschiede zum Einzeltest.

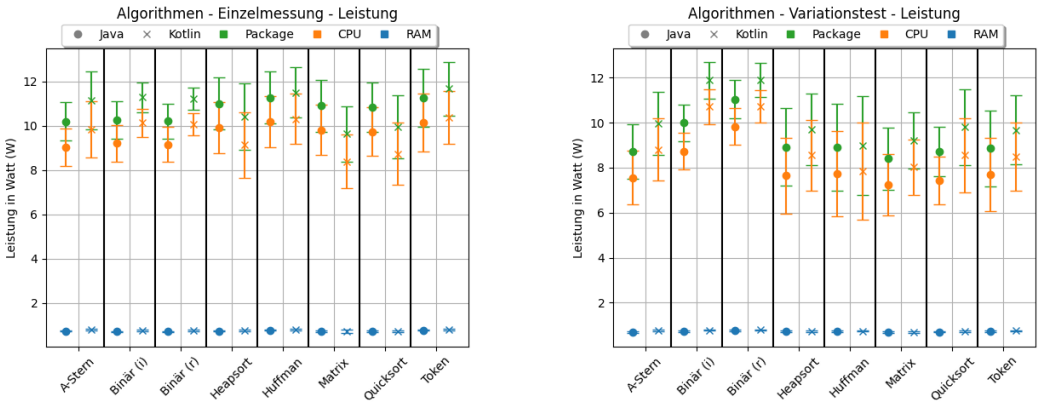


(a) Gleicher Input (b) Variabler Input

Abbildung 5.3: CPU-Auslastung und maximaler RAM der Algorithmen

In Abbildung 5.4 werden die Leistungsmessungen der Algorithmen in Watt dargestellt. Die benötigte RAM-Watt-Leistung ist im Vergleich zur CPU-Leistung geringer und stabiler. Im Einzeltest benötigen die Java-Prozesse meistens weniger Leistung im Vergleich zu Kotlin, wobei der **Heapsort**, der **Quicksort** und die **Tokenization** die Ausnahmen bilden.

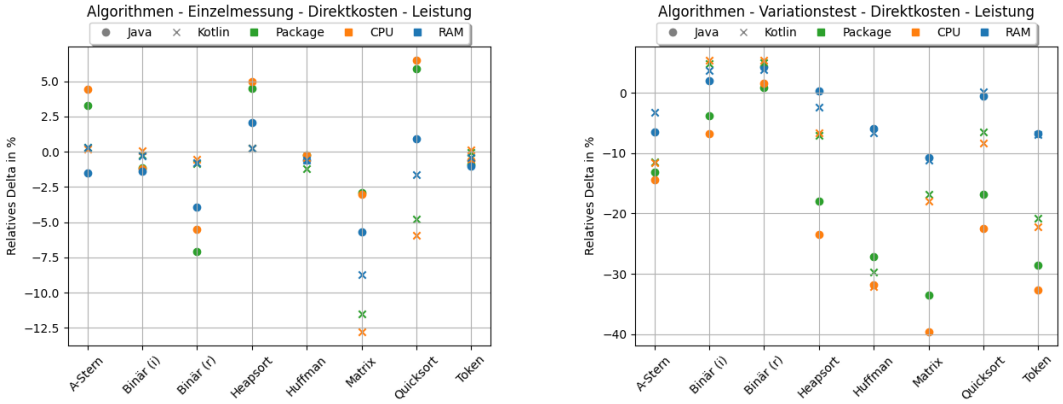
Beim Variationstest ist die benötigte Package- und CPU-Leistung, mit Ausnahme der **binären Suche**, gesunken. Bei dem **Heapsort**, der **Matrix-Ketten-Multiplikation** und dem **Quicksort** hat sich das Verhältnis Java-Kotlin umgedreht.



(a) Gleicher Input (b) Variabler Input

Abbildung 5.4: Leistungsmessungen der Algorithmen

In Abbildung 5.5 werden die Leistungsmessungen als relative Deltas der Direktkosten für alle Algorithmen dargestellt. Die meisten Werte haben ein negatives relatives Delta, wobei diese im Variationstest größer ausfallen. Die Package- und CPU-Messungen des **A*-Algorithmus**, des **Heapsort** und des **Quicksort** haben im Einzeltest bei Java ein höheres relatives Delta im Vergleich zu den anderen Messungen.
Im Variationstest sind Werte, die positiv waren, wie beispielsweise beim **A*-Algorithmus** negativ, während negative Werte wie bei der rekursiven **binären Suche** jetzt positiv sind.

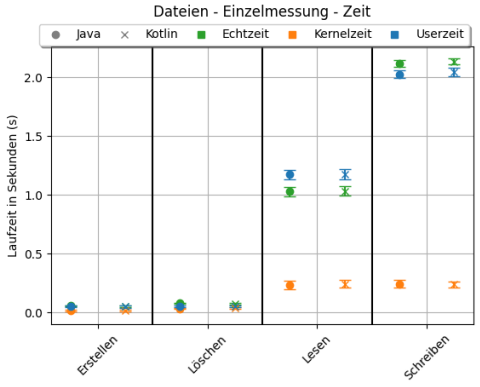


(a) Gleicher Input (b) Variabler Input

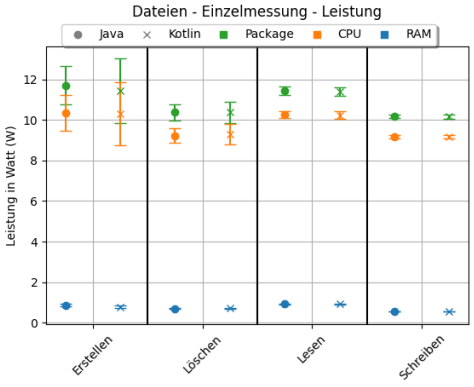
Abbildung 5.5: Leistungsdirektkosten der Algorithmen

5.2 Dateien

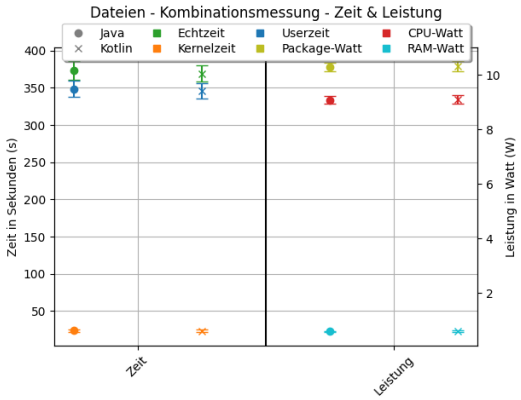
Die Ergebnisse der Laufzeit- und Leistungsmessungen der Operationsdomäne *Dateien* sind in Abbildung 5.6 zusammen dargestellt.
Die Laufzeiten in Abbildung 5.6a sind sehr ähnlich, wobei sie beim Erstellen und Löschen am geringsten sind und beim Schreiben am größten.
Wie in Abbildung 5.6c zu sehen ist, sind die Mittelwerte der Laufzeitmessungen zwischen den Sprachen sehr ähnlich. Die Laufzeiten haben sich um ein Vielfaches erhöht.



(a) Laufzeitmessungen des Einzeltests



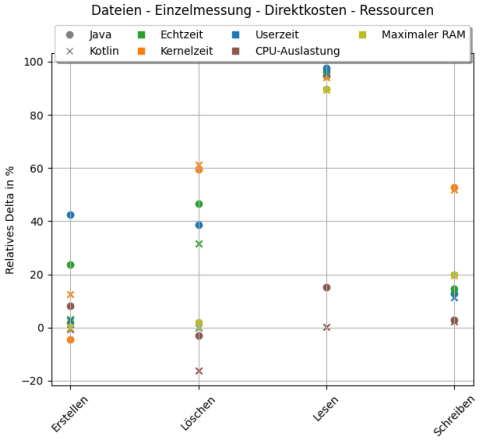
(b) Leistungsmessungen des Einzeltests



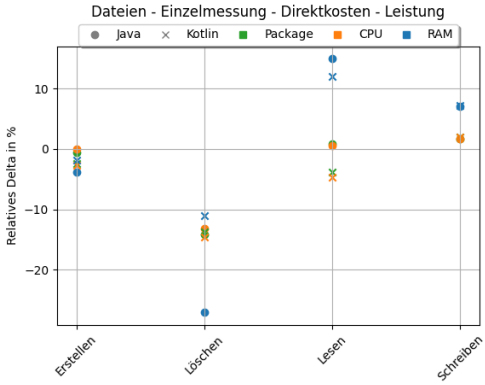
(c) Laufzeit- und Leistungsmessungen des Kombinationstests

Abbildung 5.6: Laufzeit- und Leistungsmessungen der Operationsdomäne Dateien

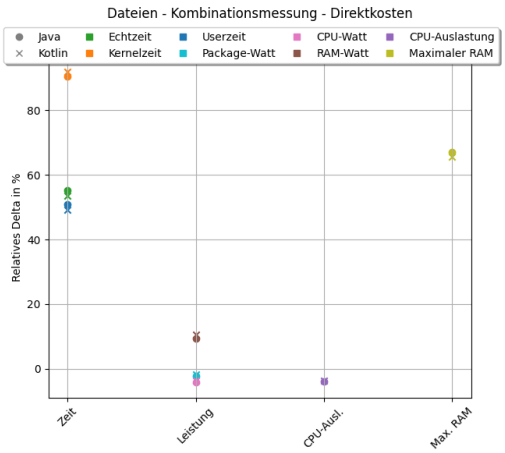
Die Direktkosten, die in Abbildung 5.7 als relative Deltas dargestellt werden, zeigen das die Grundoperation Erstellen im Einzeltest einen höheren Anteil an den User- und Echtzeitmesswerten bei der Java-Implementierung gibt. Beim Löschen von Dateien besitzen alle Laufzeitmessungen bis auf die Userzeit bei Kotlin ein hohes relatives Delta. Das Lesen hat den größten Direktkostenanteil bei allen Laufzeitmessungen. Zum Schluss fallen die Direktkosten bei der Kernelzeit für das Schreiben größer aus. Die relativen Deltas der Laufzeiten haben sich im Vergleich zum Einzeltest gemittelt.



(a) Laufzeitdirektkosten des Einzeltests



(b) Leistungsdirektkosten des Einzeltests

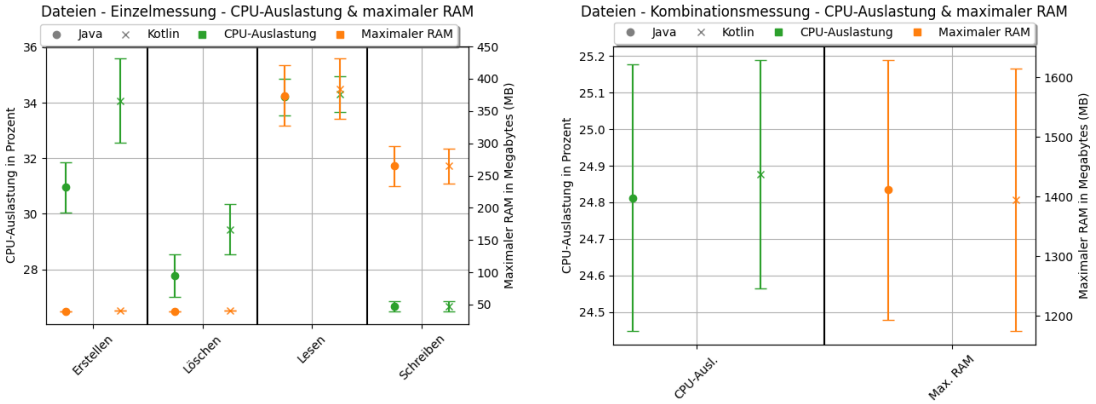


(c) Direktkosten des Kombinationstests

Abbildung 5.7: Direktkosten der Operationsdomäne Dateien

Die benötigten Leistungen, die in Abbildung 5.6b dargestellt werden, sind auch hier sehr ähnlich, wobei die Kotlin-Messungen eine höhere Standardabweichung haben. Ansonsten sind die Werte für alle Grundoperationen vergleichbar. Der Wert der Leistungsmessung hat sich beim Kombinationstest nicht erhöht und ist im Vergleich zu einigen Grundoperationen mit der Standardabweichung zurückgegangen.
Der Großteil der Leistungsdirektkosten ist negativ. Im Einzeltest sind nur beim Lesen und Schreiben höhere relative Deltas bei der RAM-Watt-Messung. Der Kombinationstest hat auch nur bei der RAM-Watt-Messung ein höheres relatives Delta.

Die CPU-Auslastung, welche in Abbildung 5.8 mit dem maximalen RAM dargestellt wird, ist im Einzeltest bei Kotlin für das Erstellen und Löschen im Vergleich viel höher als beim Lesen und Schreiben. Der maximale RAM befindet sich bei allen Grundoperationen auf demselben Niveau.
Beim Kombinationstest fällt die CPU-Auslastung geringer als in der Einzelmessung aus und ist stabiler. Kotlin hat dabei eine minimal höhere CPU-Auslastung. Der maximale RAM ist stark angestiegen und hat eine hohe Standardabweichung, wobei Java einen etwas höheren RAM-Verbrauch hat.
Für den maximalen RAM ist der Anteil der Direktkosten beim Lesen und Schreiben erhöht. Dabei ist der Anteil beim Lesen mit mehr als 80 % besonders hoch. Die Direktkosten der CPU-Auslastung sind im Kombinationstest negativ und beim maximalen RAM ist das relative Delta im Vergleich zum Lesen im Einzeltest gesunken, aber höher als beim Schreiben.

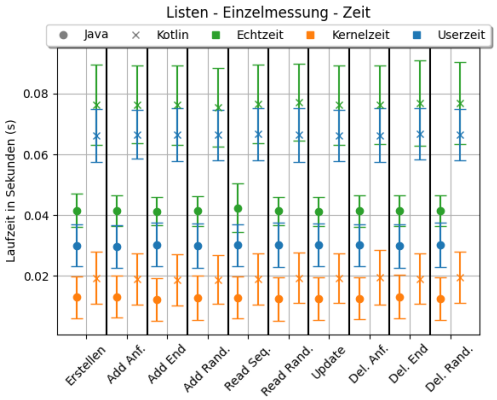


(a) Einzeltest (b) Kombinationstest

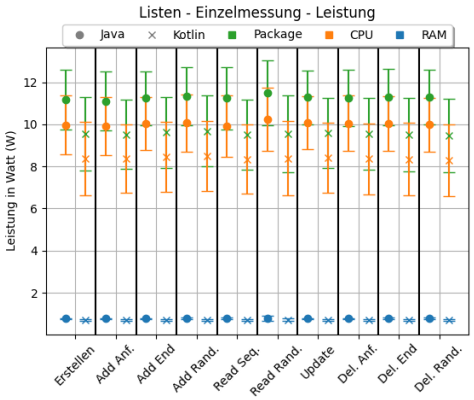
Abbildung 5.8: CPU-Auslastung und maximaler RAM der Operationsdomäne Dateien

5.3 Listen

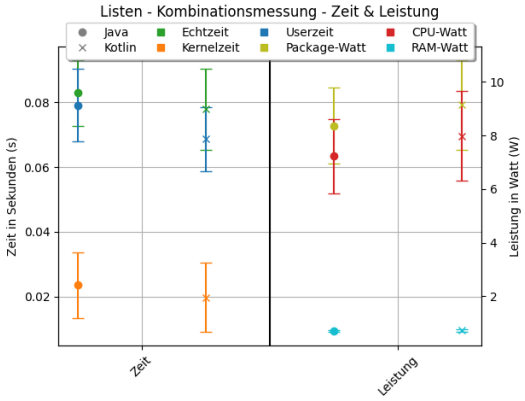
Die Abbildung 5.9 zeigt, dass die Ergebnisse für die Laufzeit- und Leistungsmessungen denselben Trend für alle Grundoperationen folgen. Die Kotlin-Implementierung hat eine höhere Laufzeit, wobei diese Distanz bei der Kernelzeit kleiner ausfällt. Die Performance ist zwischen den Grundoperationen auch vergleichbar. Bei der Kombinationsmessung hat Kotlin die niedrigeren Laufzeiten. Die Laufzeit hat sich bei Java in etwa verdoppelt.
Bei den Direktkosten, die in Abbildung 5.10 dargestellt werden, hat die Kernelzeit der Java-Implementierung die größten relativen Deltas, während die Userzeit-Direktkosten negativ sind. Die Kotlin-Implementierung hat meistens negative Direktkosten. Bei dem Kombinationstest sind die Direktkosten der Laufzeit bei Java positiv und besitzen für die User- und Kernelzeit die größten relativen Deltas, während Kotlin negative Werte hat.



(a) Laufzeitmessungen des Einzeltests



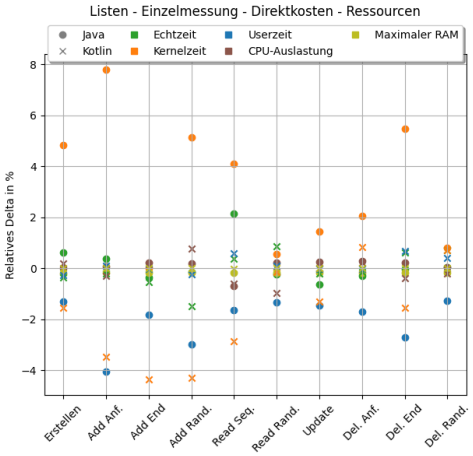
(b) Leistungsmessungen des Einzeltests



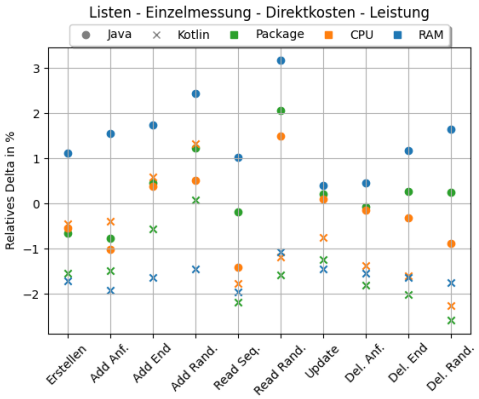
(c) Laufzeit- und Leistungsmessungen des Kombinationstests

Abbildung 5.9: Laufzeit- und Leistungsmessungen der Operationsdomäne Listen

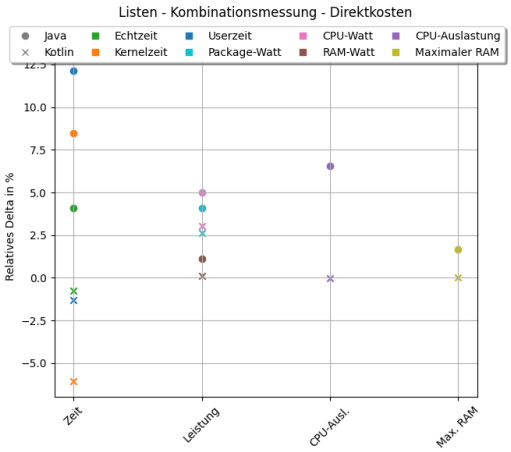
Die benötigte Leistung ist im Einzeltest für Kotlin geringer. Im Kombinationstest ist die benötigte Leistung für Java stark gesunken, wodurch sie geringer ausfällt. Bei Kotlin sind die Direktkosten oft negativ, während die Package- und RAM-Watt-Werte für Java meistens positiv sind. Die Direktkosten des Kombinationstests sind alle positiv, während die Package- und CPU-Watt im Vergleich zum Einzeltest gestiegen sind.



(a) Laufzeitdirektkosten des Einzeltests

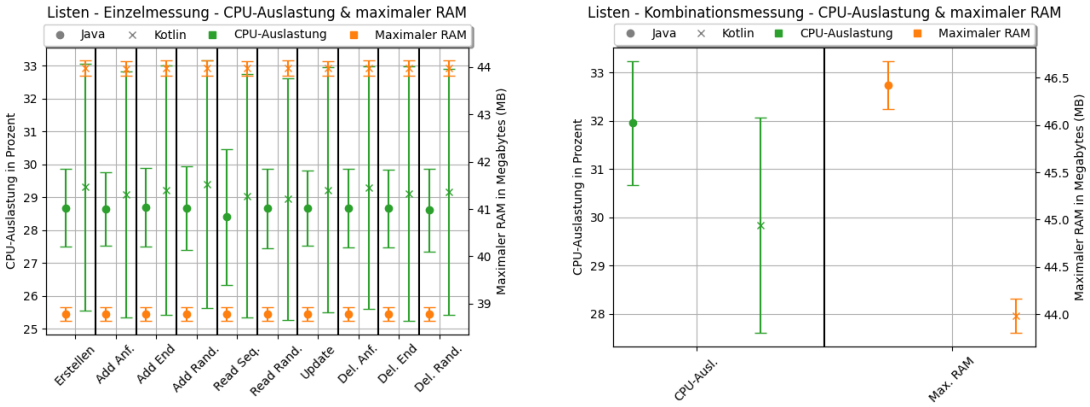


(b) Leistungsdirektkosten des Einzeltests



(c) Direktkosten des Kombinationstests

Abbildung 5.10: Direktkosten der Operationsdomäne Listen



(a) Einzeltest

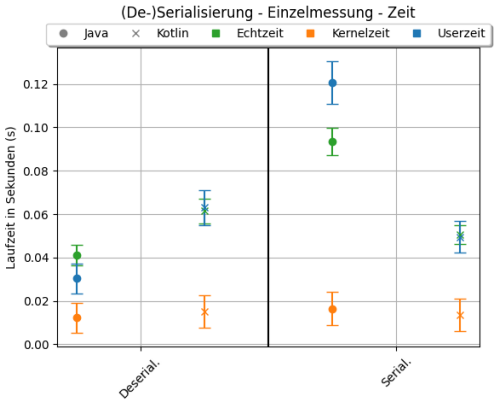
(b) Kombinationstest

Abbildung 5.11: CPU-Auslastung und maximaler RAM der Operationsdomäne Listen

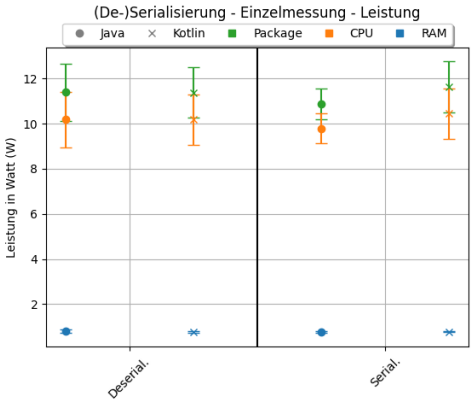
In Abbildung 5.11 werden die CPU-Auslastung und der maximale RAM der Operationsdomäne *Listen* dargestellt. Wie auch schon bei den vorherigen Abbildungen zu sehen ist, folgen alle Listenoperationen demselben Trend. Kotlin benötigt durchgehend mehr RAM, während bei der CPU-Auslastung Kotlin einen leicht höheren Mittelwert hat. Bei der CPU-Auslastung fällt auch auf, dass die Standardabweichungen bei Kotlin sehr groß ausfallen. Die Direktkosten im Einzeltest weisen geringe relative Deltas auf.
Im Kombinationstest ist die CPU-Auslastung leicht angestiegen. Der Anstieg ist bei Java größer als bei Kotlin, wodurch die CPU-Auslastung bei Java höher ist. Der maximale RAM hat sich bei Kotlin kaum geändert, bei Java gab es einen größeren Anstieg, wodurch Java nun mehr maximalen RAM benötigt. Für den maximalen RAM fällt der Direktkostenanteil gering aus, während bei der CPU-Auslastung Java ein größeres relatives Delta besitzt.

5.4 (De-)Serialisierung

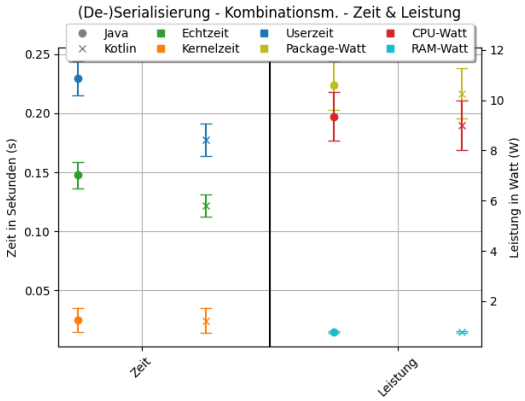
Wie in Abbildung 5.12a zu erkennen ist, weisen die Laufzeiten größere Unterschiede zwischen den Grundoperationen bei der Java-Implementierung auf. Bei der Deserialisierung hat Kotlin eine höhere Laufzeit. Bei der Serialisierung hingegen ist die Laufzeit bei Java höher. Für die Kernelzeit sind die Mittelwerte ähnlich. Im Kombinationstest gab es einen Anstieg und Java hat eine höhere Echt- und Userzeit.
In Abbildung 5.13 sind die Direktkosten der Messungen als relative Deltas zu finden. Die Direktkostenanteile fallen im Einzeltest bei der Deserialisierung für Kotlin größer aus, während sie bei der Serialisierung geringer sind. Dies ist auch in dem Kombinationstest zu sehen, wo Java höhere relative Deltas besitzt.



(a) Laufzeitmessungen des Einzeltests



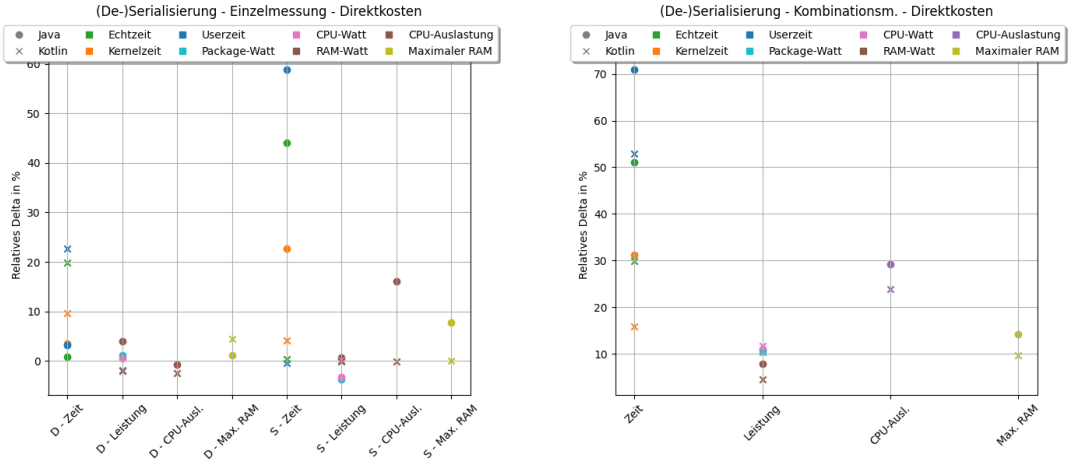
(b) Leistungsmessungen des Einzeltests



(c) Laufzeit- und Leistungsmessungen des Kombinationstests

Abbildung 5.12: Laufzeit- und Leistungsmessungen der Operationsdomäne (De-)Serialisierung

Abbildung 5.12 zeigt, dass, mit Ausnahme der Package- und CPU-Leistung bei der Serialisierung, die Mittelwerte der Leistungsmessungen im Einzeltest sehr ähnlich sind. Die Leistung ist im Kombinationstest gesunken und Kotlin benötigt etwas weniger Leistung als Java. Die Direktkosten der Leistungsmessungen fallen im Einzeltest gering aus, während sie beim Kombinationstest größere relative Deltas besitzen.



(a) Einzeltest

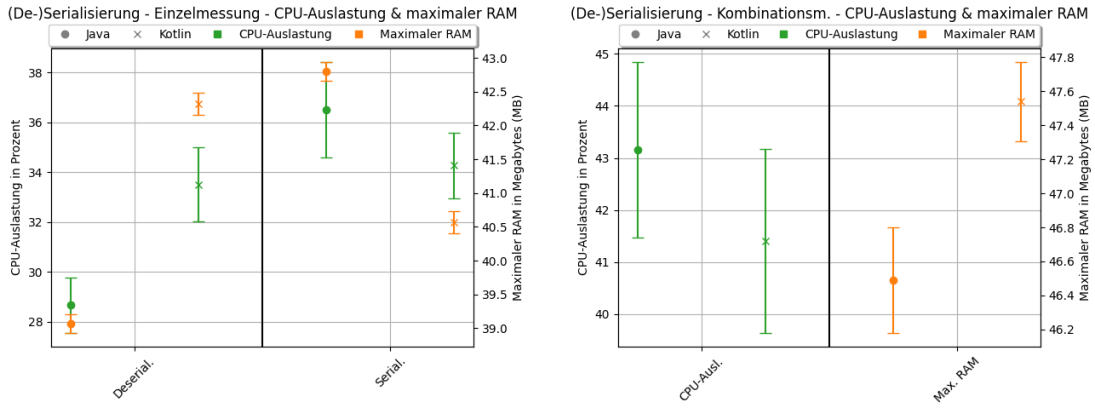
(b) Kombinationstest

Abbildung 5.13: Direktkosten der Operationsdomäne (De-)Serialisierung

Abbildung 5.14 zeigt denselben Trend wie bei den Laufzeiten. Bei der Deserialisierung sind die Java-Messwerte geringer und bei der Serialisierung höher. In der Kombinationsmessung ist der Mittelwert von Java größer als von Kotlin. Eine Ausnahme des Trends ist der maximale RAM, wo in der Kombinationsmessung der Java-Mittelwert geringer ist.

Die Direktkosten der CPU-Auslastung bei der Deserialisierung sind negativ, während die CPU-Auslastung von Java bei der Serialisierung ein großes relatives Delta aufweist. Bei der Kombinationsmessung ist die CPU-Auslastung angestiegen.

Der Direktkostenanteil fällt für Kotlin beim maximalen RAM nur bei der Deserialisierung größer als Java aus. Die relativen Deltas sind im Kombinationstest angestiegen.



(a) Einzeltest

(b) Kombinationstest

Abbildung 5.14: CPU-Auslastung und maximaler RAM der Operationsdomäne Dateien

6 Analyse

In diesem Kapitel werden die in Kapitel 5 vorgestellten Ergebnisse besprochen, analysiert und statistischen Tests unterzogen. Dabei wird zuerst die Operationsdomäne *Algorithmen* betrachtet. Anschließend werden die Einzeltests der anderen Operationsdomänen untersucht. Darauffolgend werden dann die Kombinationstests betrachtet. Ein zusammenfassendes Fazit bildet den Abschluss des Kapitels.

Es wird ein Signifikanzniveau von 1 % angenommen. Das bedeutet, dass ein relatives Delta kleiner als 1 % keine statistische Signifikanz hat. Es muss angemerkt werden, dass die Fehlerrate vom *time*-Werkzeug nicht bekannt ist.

6.1 Algorithmen

In diesem Unterkapitel wird sich mit der Operationsdomäne *Algorithmen* auseinandergesetzt. Zuerst wird der Einzeltest betrachtet und danach der Variationstest.

6.1.1 Einzeltest

Die relativen Deltas der Mittelwerte zwischen Java und Kotlin werden in Tabelle 6.1 aufgelistet und statistisch signifikanten Deltas werden hervorgehoben.
Alle in diesem Unterkapitel angesprochenen Verteilungen sind im Anhang als Abbildungen A.1 bis A.3 zu finden.
Bei dem **A*-Algorithmus** und der **Tokenization** gibt es eine Anomalie mit der Kernelzeit. Die Mittelwerte zeigen ein signifikantes Delta, was 29,64 % beziehungsweise -23,46 % entspricht. Der Median zeigt hingegen bei beiden Werten ein Delta von 0,00 %. Da die Verteilungen der Messung jedoch keine signifikanten Ausläufer oder Cluster aufweisen, wird der Mittelwert als ausschlaggebende Metrik verwendet.

Beim *Heapsort* gibt es eine Anomalie mit der CPU-Auslastung. Der Mittelwert beträgt bei Java 29,83 %, der Median hingegen 31,75 %. Dadurch kommt es zwischen den Mittelwerten zu einem relativen Delta von -6,94 %, während dieses bei den Medianen nur -0,79 % beträgt. Dies liegt daran, dass die Verteilung der Java-Messwerte zum Großteil geringer ist, wodurch Ausreißer sich stärker auf den Mittelwert auswirken. Aus diesem Grund wird hier das relative Delta der Mediane genommen, da diese, wie in Unterkapitel 3.3 besprochen, resistenter gegen Fluktuationen sind.

Tabelle 6.1: Relatives Delta (%) der Mittelwerte aller Algorithmen

Softwaremetrik	**A-Stern**	**Iter. Binär**	**Rek. Binär**	**Heapsort**
Echtzeit	**27,38**	**-30,27**	**-28,99**	**-70,16**
Kernelzeit	**29,64**	**-22,06**	**-17,37**	**-39,47**
Userzeit	**41,51**	**-67,06**	**-66,15**	**-87,63**
CPU-Auslastung	**14,45**	**-23,47**	**-23,48**	-0,79*
Max. RAM	**15,61**	**-4,15**	**-4,16**	**-8,03**
Package-Watt	**-9,25**	**-10,11**	**-9,98**	**5,37**
CPU-Watt	**-8,93**	**-10,12**	**-9,99**	**7,95**
RAM-Watt	**-8,17**	**-6,35**	**-5,88**	**-2,64**
	Huffman	**Matrix**	**Quicksort**	**Token**
Echtzeit	**-75,84**	**-122,60**	**-63,33**	**-23,13**
Kernelzeit	**-32,46**	**-65,47**	**-22,29**	**-23,46**
Userzeit	**-166,50**	**-173,46**	**-71,43**	**-64,27**
CPU-Auslastung	**-22,31**	**-9,72**	**-1,82**	**-18,20**
Max. RAM	**-5,48**	**-10,92**	**-4,55**	**-4,70**
Package-Watt	**-2,03**	**11,59**	**8,05**	**-3,45**
CPU-Watt	**-1,46**	**14,55**	**10,28**	**-2,25**
RAM-Watt	**-2,97**	**2,42**	0,34	**-3,92**

Hervorgehobene Werte markieren ein signifikantes Delta. Das * markiert Werte, wo der Median verwendet wird. Ein negativer Wert entspricht einem größeren Messwert bei Kotlin.

Die Werte in Tabelle 6.1 zeigen, dass Kotlin in den meisten Fällen mehr Leistung und Ressourcen benötigt. Nur beim *A*-Algorithmus* benötigt Java mehr Ressourcen. Bei der Leistung hingegen gibt es mehrere Fälle, in denen Java mehr Leistung verbraucht. Dies ist bei dem *Heapsort*, der *Matrix-Ketten-Multiplikation* und dem *Quicksort* für die Package-Watt und CPU-Watt der Fall. Bei der *Matrix-Ketten-Multiplikation* kommt noch der RAM-Watt hinzu. Die CPU-Auslastung des *Heapsorts* und die RAM-Watt-Leistung des *Quicksort* bilden hier Ausnahmen, wo es zu keinem signifikanten Unterschied kommt.

Bei den Direktkosten fallen die negativen Werte auf. Diese bedeuten, dass der Overhead einen höheren Ressourcenverbrauch hat als der Overhead in Kombination mit dem Algorithmus. In diesen Fällen sind die Kosten des reinen Algorithmus so gering, dass sie durch Variationen überschattet werden. Bei dem *A*-Algorithmus*, der *Matrix-Ketten-Multiplikation* und dem *Quicksort* gibt es wahrscheinlich statistisch signifikante Kosten in der Algorithmus-Implementierung, die Höhe ist aber unbekannt. Der höhere Ressourcenverbrauch des *A*-Algorithmus* bei Java scheint durch den Algorithmus und nicht dem Overhead zu kommen, zumindest sind hier die Direktkosten entsprechend hoch. Bei der *Matrix-Ketten-Multiplikation* gibt es sowohl bei der Echtzeit als auch bei der Userzeit eine Differenz von 0,03 Sekunden zwischen Java und Kotlin. Diese Differenz macht aber nicht den kompletten Unterschied zwischen Java und Kotlin aus. Die Differenz bei der Userzeit des *Quicksorts* ist $< 0{,}01$ Sekunden und die Differenz bei der Echtzeit ist 0,01 Sekunden. Hierdurch ist der Unterschied zwischen Java und Kotlin also nicht erklärt.

Die Tabelle 6.1 zeigt viele statistisch signifikante Unterschiede, die aber nicht durch die Direktkosten erklärbar sind. Das bedeutet, dass entweder die Unterschiede durch den Overhead oder die Variation in den Messungen und dem System kommen. Eine höhere Last auf den Algorithmus wäre nötig, um Genaueres sagen zu können.

6.1.2 Variationstest

In Tabelle 6.2 werden die relativen Deltas der Mittelwerte zwischen Java und Kotlin bei dem Variationstest aufgelistet. Dabei werden die statistisch signifikanten Deltas hervorgehoben dargestellt.
Die besprochenen Verteilungen sind im Anhang als Abbildungen A.4 bis A.12 zu finden.
Für die Kernelzeit bei dem **A*-Algorithmus**, der iterativen **binären Suche**, der **Matrix-Ketten-Multiplikation** und dem **Quicksort** gibt es eine Differenz zwischen Mittelwert und Median, bei dem der Mittelwert eine Signifikanz zeigt, die der Median nicht widerspiegelt. Aufgrund der Verteilung wird sich hier dennoch für den Mittelwert entschieden.
Die rekursive **binäre Suche** hat auch diese Differenz für die Kernelzeit. Hier hat die Verteilung bei Kotlin aber einen Ausläufer nach rechts, weshalb der Median verwendet wird. Im weiteren Verlauf dieses Kapitels wird der Median verwendet, wenn es entsprechende Ausläufer gibt. Sollten weitere Gründe eine Rolle bei der Entscheidung spielen, so werden diese explizit erwähnt.
Die **Matrix-Ketten-Multiplikation** hat bei der RAM-Watt-Messung neben einem Ausläufer auch eine Clusterbildung. Beides wirkt sich auf die Mittelwerte aus, weshalb auf den Median zurückgegriffen wird.
Die letzte Differenz befindet sich bei der **Tokenization**. Hier hat die Echtzeitmessung ein signifikantes Delta beim Mittelwert. Es wird jedoch der Median verwendet, da es kleine Ausläufer nach rechts gibt.

In einigen Fällen hat sich die Bedeutung der relativen Deltas in Tabelle 6.2 im Vergleich zur Tabelle 6.1 geändert. Diese Werte werden in der Tabelle mit einem + markiert. Dabei haben einige Werte das Vorzeichen gewechselt, während andere zwischen signifikant und insignifikant gewechselt sind. Der Großteil der Tabelle hat aber immer noch dieselbe Bedeutung.

Für die statistischen Tests gilt eine Ablehnung der Nullhypothese, wenn der berechnete p-Wert kleiner als 0,05 ist. Eine Ablehnung der Nullhypothese bedeutet beim Shapiro-Wilk-Test, dass die Messwerte nicht normalverteilt sind. Bei dem Kolmogorow-Smirnow-Test bedeutet eine Ablehnung, dass keine identischen Verteilungen vorliegen. Der Welch's t-Test gibt bei einer Ablehnung an, dass die Mittelwerte unterschiedlich sind.

Tabelle 6.2: Relatives Delta (%) der Mittelwerte aller Algorithmen im Variationstest

Softwaremetrik	**A-Stern**	**Iter. Binär**	**Rek. Binär**	**Heapsort**
Echtzeit	**11,11**	**-39,29**	**-23,26**	**-36,99**
Kernelzeit	**15.82**	**-1,25**	0,00*+	**-22,67**
Userzeit	**6,72**	**-98,01**	**-47,83**	**-91,08**
CPU-Auslastung	**-3,20+**	**-25,88**	**-10,58**	**-19,77+**
Max. RAM	**4,59**	**-7,27**	**-4,21**	**-7,97**
Package-Watt	**-14,30**	**-19,05**	**-7,97**	**-8,93+**
CPU-Watt	**-16,46**	**-22,61**	**-9,07**	**-11,92+**
RAM-Watt	**-9,52**	**-7,00**	**-2,26**	**-1,75**
	Huffman	**Matrix**	**Quicksort**	**Token**
Echtzeit	**-60,84**	**-75,43**	**-34,58**	0,00*
Kernelzeit	**-24,18**	**-31,82**	**-5,68**	**-12,08**
Userzeit	**-179,70**	**-166,41**	**-74,76**	**-61,76**
CPU-Auslastung	**-29,28**	**-25,04**	**-17,58**	**-21,60**
Max. RAM	**-5,50**	**-11,01**	**-4,62**	**-4,86**
Package-Watt	**3,23+**	**-9,53+**	**-12,29+**	**-9,20**
CPU-Watt	**3,51+**	**-10,75+**	**-14,93+**	**-10,3**
RAM-Watt	**-2,37**	**2.86**	**-2,92+**	**-3,11**

Hervorgehobene Werte markieren ein signifikantes Delta. Das * markiert Werte, wo der Median verwendet wird, während das + eine signifikante Änderung zum Einzeltest markiert. Ein negativer Wert entspricht einem größeren Messwert bei Kotlin.

Der Shapiro-Wilk-Test ergab, dass die meisten Messwerte nicht normalverteilt sind. Außer der Kernelzeit, wo nur die **Matrix-Ketten-Multiplikation** nicht die gleiche Verteilung hat, sind bei den anderen Softwaremetriken laut dem Kolmogorow-Smirnow-Test kaum gleiche Verteilungen. Durch die fehlenden identischen Verteilungen kann der Welch's t-Test nur auf wenigen Messwerten angewendet werden.

Für die Kernelzeit bei dem **A*-Algorithmus**, dem **Heapsort** und dem **Huffman Encoding** wird die Nullhypothese abgelehnt. Das bedeutet, dass es mit einer hohen Wahrscheinlichkeit unterschiedliche Mittelwerte gibt. Für die maximale RAM-Messung bei beiden **binären Suchen** und dem **Quicksort** sowie der Package-Watt und der CPU-Watt-Messung bei der **Matrix-Ketten-Multiplikation** wird auch die Nullhypothese abgelehnt. Diese Ablehnungen bestätigen die Signifikanz der entsprechenden Werte in Tabelle 6.2.
Für die Kernelzeit bei beiden **binären Suchen**, dem **Quicksort** und der **Tokenization** sowie der RAM-Watt-Messung bei der **Matrix-Ketten-Multiplikation** wird die Nullhypothese nicht abgelehnt. Bei der Kernelzeit der rekursiven **binären Suche** wird die Nichtsignifikanz, die durch den Median gegeben ist, bestätigt. Bei den anderen Messwerten widerspricht es den Werten in Tabelle 6.2. Dies kann bedeuten, dass das Signifikanzniveau von 1 % zu niedrig ist. Eine Nichtablehnung ist jedoch kein definitiver Beweis für denselben Mittelwert. Die p-Werte der statistischen Tests sind im Anhang zu finden.

Die Direktkosten der Ressourcen zeigen für die meisten Algorithmen eine höhere Last, während die Last beim **A*-Algorithmus** zurückgegangen ist. Der Overhead macht jedoch immer noch den Großteil der Messung aus, wobei die Echt- und Userzeitmessung bei der Java-Implementierung des **A*-Algorithmus** die Ausnahme bilden. Hier ist die Differenz zwischen Java und Kotlin größer als die Messunterschiede. Die Direktkosten der Leistungsmessungen sind zum Großteil negativ. Wie schon bei dem Einzeltest sind die Unterschiede der Algorithmen nicht durch die Direktkosten erklärbar.

Die Untersuchung mithilfe des Spearman'schen Rangkorrelationskoeffizienten ergibt eine Ablehnung der Nullhypothese nur bei dem maximalen RAM des **A*-Algorithmus**. Das bedeutet, nur bei diesem einen Messwert kann eine Unabhängigkeit beider Datensätze wahrscheinlich ausgeschlossen werden. Dabei tritt eine negative ordinale Korrelation mit einem t-Wert von -0,200 auf, was bedeutet, dass mit größeren Java-Messwerten kleinere Kotlin-Messwerte einhergehen. Alle anderen Messdaten sind nach dieser Untersuchung entweder unabhängig oder haben eine nicht monotone Korrelation.

Zu der Operationsdomäne **Algorithmen** lässt sich abschließend sagen, dass die Last der Algorithmen wahrscheinlich zu gering ist, demnach wären die Unterschiede allein durch Overhead der JVM und der Implementierung erklärbar. Gerade die Leistungsmessungen haben vor allem negative Direktkosten oder einen sehr geringen Anteil. Da einige Direktkosten im Einzeltest positiv, im Variationstest jedoch negativ sind und umgekehrt, kann hier kein Einfluss der Algorithmen mit Sicherheit festgestellt werden. Die Unterschiede zwischen dem Einzel- und dem Variationstest könnten an den verschiedenen Inputs liegen. Es ist jedoch wahrscheinlicher, dass Variationen durch in- oder externe Faktoren eine größere Rolle spielen. Hinzu kommt, dass die Signifikanz der Unterschiede schwer einzuschätzen ist. Das gewählte Signifikanzniveau von 1 % ist eventuell zu gering und die gewählten statistischen Tests sind wenig aussagekräftig und der Welch's t-Test selten anwendbar, da die dafür notwendigen Kriterien nicht erfüllt sind.

6.2 Einzeltest

In diesem Unterkapitel wird sich mit den Einzeltests der Operationsdomänen *Dateien*, *Listen* und *(De-)Serialisierung* auseinandergesetzt. Die Operationsdomänen werden nacheinander getrennt betrachtet. Zuerst wird eine Tabelle aus den relativen Deltas der Mittelwerte erstellt. Danach wird diese Tabelle betrachtet und mit den Direktkosten in Verbindung gebracht.

Die Verteilungen des Einzeltests sind im Anhang als Abbildungen A.13 bis A.28 zu finden.

6.2.1 Dateien

Wie bei der Operationsdomäne *Algorithmen* gibt es auch bei der Operationsdomäne *Dateien* einige Fälle, in denen der Median und der Mittelwert unterschiedliche Aussagen über die statistische Signifikanz zulassen. In einigen Fällen wird, wie in Unterkapitel 6.1.2 erwähnt, aufgrund von Ausläufern der Median verwendet.

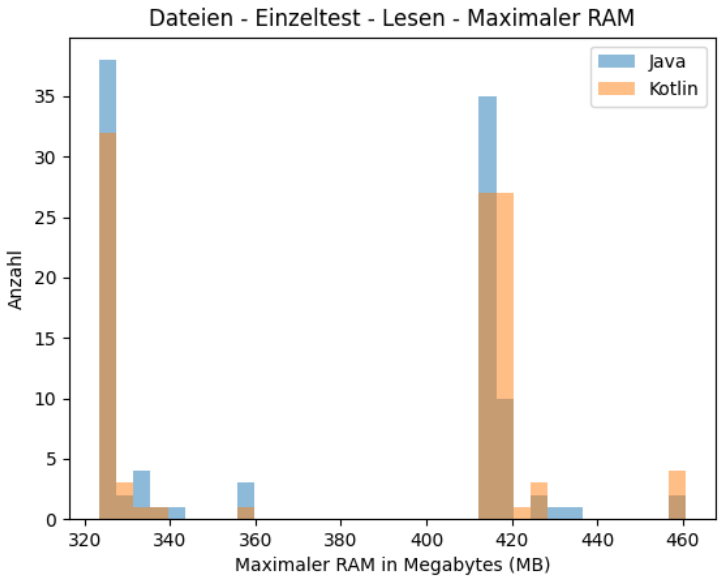


Abbildung 6.1: Verteilung der maximalen RAM-Messung für Dateien Lesen

Beim Lesen der Dateien kommt es bei der Messung des maximalen RAMs zu einer schwierig zu interpretierenden Ansammlung von Messwerten. Wie in Abbildung 6.1 zu sehen ist, existieren zwei Cluster, wodurch es zu einer größeren Differenz zwischen Mittelwert und Median kommt. Der Mittelwert beträgt 374,20 MB bei Java und 384,50 MB bei Kotlin, was einem relativen Delta von -2,75 % entspricht. Die Mediane liegen hingegen bei 414,48 MB beziehungsweise 415,62 MB, was einem relativen Delta von -0,28 % entspricht. Aufgrund der Clusterbildung werden beide Cluster noch einmal getrennt betrachtet. Dabei kann weder bei den Mittelwerten noch bei den Medianen ein signifikanter Unterschied festgestellt werden. Deshalb wird für die Tabelle der Median verwendet.

Die Clusterbildung kann mehrere Ursachen haben. Die erste Ursache könnte die nicht immer vollständig kontrollierbare Memory Allocation sein, sodass bei einigen Durchläufen mehr RAM als benötigt reserviert wird. Dies könnte durch die Implementierung oder durch die JVM geschehen. Die zweite Ursache könnte der Garbage Collector sein. In den verwendeten Methoden werden Variablen erstellt und verworfen. Dabei könnte in einigen Prozessen der Garbage Collector schneller alte Variablen löschen als in anderen, wodurch weniger RAM maximal belegt ist. Die dritte Ursache wäre das *time*-Werkzeug. Eventuell hat es eine zu geringe Abtastrate, wodurch der Messwert nicht den wahren maximalen RAM widerspiegelt. Es ist unbekannt, in welchen Abständen *time* den RAM misst.

Wird nun neuer RAM reserviert, aber die Messung beendet, bevor der neue RAM ausgelesen wurde, könnte ein zu kleiner Wert angegeben werden. Für diese Ursache spricht auch, dass die Kotlin-Messwerte zwischen 1,02 und 1,03 Sekunden Echtzeit das Cluster wechseln. Es gibt jedoch einen Messwert mit 1,07 Sekunden, der im ersten Cluster liegt. Bei Java kommt es auch zu einem Wechsel, wobei die Messwerte zwischen 1,02 und 1,04 Sekunden zwischen den Clustern schwanken. Ein kausaler Zusammenhang ist jedoch nicht ersichtlich, da die Prozesse, die mehr RAM benötigen, auch einfach eine höhere Laufzeit benötigen können. Eine genaue Ursachenforschung liegt außerhalb des zeitlichen Rahmens dieser Untersuchung.

Die Userzeit weist beim Erstellen und Löschen eine Anomalie auf. Bei den Mittelwerten kommt es zu einem relativen Delta von 4,02 % beziehungsweise 3,91 %. Das relative Delta der Mediane liegt bei 0,00 %. Keine der Verteilungen hat einen großen Ausläufer, der dies erklären kann. Aus diesem Grund wird der Mittelwert verwendet. Bei der Kernelzeitmessung des Löschens und Schreibens zeigen die Mediane ein relatives Delta von 0,00 %, da keine entsprechenden Ausläufer und Cluster erkennbar sind, wird wie bei dem **A*-Algorithmus** und der **Tokenization** der Mittelwert verwendet.

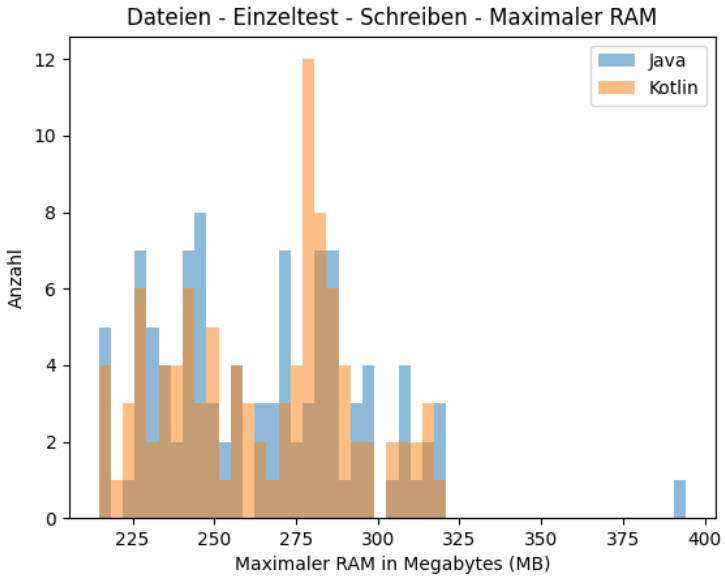


Abbildung 6.2: Verteilung der maximalen RAM-Messung für Dateien Schreiben

Zum Schluss gibt es auch bei der Grundoperation Schreiben eine Auffälligkeit bei den Messwerten des maximalen RAMs. Die Mittelwerte zeigen mit 264,73 MB bei Java und 264,77 MB bei Kotlin ein relatives Delta von -0,01 %. Der Median von Java ist mit einem Wert von 264,82 nur 0,11 MB größer als der Mittelwert. Bei Kotlin gibt es jedoch eine Differenz von 6,74 MB, was dazu führt, dass der Median einen signifikanten Unterschied mit einem relativen Delta von -2,52 % feststellt. Bei näherer Betrachtung der Verteilung, die in Abbildung 6.2 dargestellt wird, stellt man fest, dass sich bei Kotlin drei Cluster ausgebildet haben. Der erste Cluster enthält 49 der 100 Messwerte, während der zweite Cluster 41 und der dritte Cluster 10 Messwerte beinhaltet. Dies legt die Vermutung nahe, dass bei einer höheren Wiederholungsanzahl mehr als 50 % der Messwerte in das erste Cluster fallen oder der Übergang ausgeglichener wird. Deshalb wird sich in diesem Fall für den Mittelwert entschieden, obwohl der Median normalerweise gewählt werden würde.

Tabelle 6.3: Relatives Delta (%) der Mittelwerte aller Dateigrundoperationen

Softwaremetrik	**Erstellen**	**Löschen**	**Lesen**	**Schreiben**
Echtzeit	**9,15**	**7,74**	-0,5	-0,74
Kernelzeit	**-13,33**	**-3,65**	**-4,23**	**2,11**
Userzeit	**4,02**	**3,91**	-0,09	-0,98
CPU-Auslastung	**-10,06**	**-5,95**	-0,28	-0,01
Max. RAM	**-3,14**	**-3,22**	-0,28*	-0,01
Package-Watt	-0,38*	-0,09	0,31	-0,05
CPU-Watt	**-2,19***	**-1,04***	0,16	-0,07
RAM-Watt	**8,34**	**-3,07**	0,7	-0,20

Hervorgehobene Werte markieren ein signifikantes Delta. Das * markiert Werte, wo der Median verwendet wird. Ein negativer Wert entspricht einem größeren Messwert bei Kotlin.

Im Gegensatz zu den Algorithmen zeigt Tabelle 6.3 bei den Grundoperationen der Operationsdomäne *Dateien* viele nicht statistisch signifikante Unterschiede. Vor allem beim Lesen oder Schreiben gibt es nur bei der Kernelzeit signifikante Unterschiede. Dabei verbringen die Kotlin-Prozesse beim Lesen mehr Zeit im Kernelmode und weniger Zeit beim Schreiben. Bis auf die Package-Watt-Messung haben alle anderen Messungen beim Erstellen und Löschen einen signifikanten Unterschied. Dabei benötigt Java bei beiden Grundoperationen mehr Echt- und Userzeit sowie mehr RAM-Watt beim Erstellen.

Kotlin benötigt bei beiden Grundoperationen mehr Kernelzeit, CPU-Auslastung und maximalen RAM sowie mehr RAM-Watt beim Löschen.
Die Leistungswerte für alle Grundoperationen sind auf einem ähnlichen Niveau, obwohl das Erstellen und Löschen von Dateien eine sehr geringe Laufzeit im Vergleich zum Schreiben hat.
In Anbetracht der in Unterkapitel 5.2 vorgestellten Mittelwerte stellt man fest, dass mit Ausnahme der CPU-Auslastung-Messung die absoluten Differenzen beim Erstellen und Löschen äußerst gering sind.

Die Direktkosten zeigen, dass im Gegensatz zu der Operationsdomäne *Algorithmen* große Teile der Ausführungskosten bei den Ressourcen durch die Dateioperationen kommen. Vor allem beim Lesen fällt der Overhead fast überhaupt nicht ins Gewicht. Bei der Leistung hingegen scheinen die Grundoperationen nicht so stark ins Gewicht zu fallen, denn hier sind viele Direktkosten negativ. Eine wirklich große Rolle spielt nur die RAM-Watt beim Lesen und Schreiben. Beim Erstellen und Löschen sind die Variationen wieder größer als die Grundoperationskosten.

Es gibt zwar einige statistisch signifikante Unterschiede zwischen Java und Kotlin, jedoch sind diese vor allem beim Erstellen und Löschen zu finden. Hier wurde die Leistungsmessung durch Variationen stark beeinflusst. Der Unterschied ist also entweder durch diese in- oder externen Variationen oder wieder durch den Overhead erklärbar. Bei der Ressourcenmessung sind diese Unterschiede zwar eher messbar, jedoch ist anzumerken, dass diese beiden Grundoperationen die Operationen mit einer geringen Last sind, wo Variationen verstärkt eine Rolle spielen. Die Unterschiede sind auch dadurch erklärbar, dass die ihnen zugrundeliegenden Verteilungen unterschiedlich sind. Kotlins Messwerte haben beispielsweise in Abbildung A.13 breitere Verteilung von als Javas Messwerte. Dadurch wird auch die Standardabweichung einer Sprache beeinflusst, dies ist bei der CPU-Watt-Messung bei dem Erstellen von Dateien in Abbildung 5.6 sichtbar. Die Verteilung der CPU-Watt-Messung ist im Anhang in Abbildung A.14 dargestellt. Für das Erstellen und Löschen wäre ebenfalls eine höhere Last innerhalb einer Messung empfehlenswert, damit die Annahmen entsprechend untersucht werden können.

6.2.2 Listen

Bei der Operationsdomäne *Listen* gibt es nur bei wenigen Grundoperationen Besonderheiten.

In Tabelle 6.4 sieht man, dass die CPU-Auslastung nur beim zufälligen Hinzufügen und beim „Löschen vom Anfang der Liste“ statistisch signifikant ist. In Abbildung 5.11 findet sich jedoch derselbe Trend bei allen Grundoperationen. Daraus lässt sich schließen, dass der Unterschied eventuell bei allen Grundoperationen dieselbe Signifikanz hat. Dies könnte ein Indikator für ein nicht optimal gewähltes Signifikanzniveau sein. Kotlin hat bei der CPU-Auslastung eine sehr breite Verteilung im Gegensatz zu Java, was zeigt, dass Kotlin hier nicht stabil läuft, aber das Zentrum der Verteilung nahe an Java liegt. Ansonsten sind alle Unterschiede zwischen Java und Kotlin bei den verschiedenen Grundoperationen sehr ähnlich. Java scheint also ressourcensparender und schneller zu sein, dafür aber mehr Leistung zu benötigen.

Die negativen relativen Deltas fallen auch hier bei den Direktkosten auf. Nur die Kernelzeit bei Java scheint größere Differenzen zu haben. Bei den geringen Mittelwerten, welche kleiner als 0,02 Sekunden sind, ist es aber fragwürdig, wie stark diese ins Gewicht fallen.

Für die Operationsdomäne *Listen* scheint Java performanter zu sein, während Kotlin energieeffizienter ist. Da die Direktkosten-Messungen aber gezeigt haben, dass der Overhead fast 100 % der Last ausgemacht hat, wäre es auch hier sinnvoll, innerhalb einer Wiederholung die Last zu erhöhen. Die CPU-Auslastung zeigt auf, dass das Signifikanzniveau eventuell falsch gewählt wurde.

Tabelle 6.4: Relatives Delta (%) der Mittelwerte aller Listengrundoperationen

Softwaremetrik	**Erstellen**	**Add Anf.**	**Add Ende**	**Add Rand.**
Echtzeit	**-83,67**	**-83,97**	**-84,65**	**-82,56**
Kernelzeit	**-48,39**	**-43,27**	**-54,06**	**-46,23**
Userzeit	**-120,67**	**-124,54**	**-119,50**	**-121,56**
CPU-Auslastung	-0.86*	-0.86*	-0.86*	**-2,50**
Max. RAM	**-13,40**	**-13,39**	**-13,38**	**-13,41**
Package-Watt	**14,55**	**14,25**	**14,52**	**14,64**
CPU-Watt	**15,98**	**15,58**	**15,93**	**15,40**
RAM-Watt	**6,32**	**6,51**	**6,43**	**6,93**
	Read Seq.	**Read Rand.**	**Update**	
Echtzeit	**-80,78**	**-86,09**	**-84,80**	
Kernelzeit	**-45,95**	**-55,40**	**-52,30**	
Userzeit	**-121,13**	**-119,44**	**-119,36**	
CPU-Auslastung	0.00*	0,86*	-0,86*	
Max. RAM	**-13,39**	**-13,38**	**-13,38**	
Package-Watt	**15,44**	**16,84**	**14,99**	
CPU-Watt	**16,01**	**17,95**	**16,42**	
RAM-Watt	**7,37**	**8,60**	**6,33**	
	Del. Anf.	**Del. Ende**	**Del. Rand.**	
Echtzeit	**-84,34**	**-85,25**	**-85,26**	
Kernelzeit	**-54,57**	**-45,66**	**-56,39**	
Userzeit	**-119,87**	**-123,63**	**-119,96**	
CPU-Auslastung	**-2,15**	-0,86*	-0,86*	
Max. RAM	**-13,40**	**-13,39**	**-13,40**	
Package-Watt	**15,23**	**15,69**	**16,14**	
CPU-Watt	**16,74**	**16,79**	**16,86**	
RAM-Watt	**6,46**	**7,22**	**7,76**	

Hervorgehobene Werte markieren ein signifikantes Delta. Das * markiert Werte, wo der Median verwendet wird. Ein negativer Wert entspricht einem größeren Messwert bei Kotlin.

6.2.3 (De-)Serialisierung

Bei der Operationsdomäne *(De-)Serialisierung* gibt es nur bei der RAM-Watt-Messung der Deserialisierung eine Auffälligkeit.

Tabelle 6.5: Relatives Delta (%) der Mittelwerte aller (De-)Serialisierung-Grundoperationen

Softwaremetrik	Deserialisierung	Serialisierung
Echtzeit	**-49,16**	**45,84**
Kernelzeit	**-25,29**	**16,75**
Userzeit	**-107,34**	**58,98**
CPU-Auslastung	**-16,89**	**6,15**
Max. RAM	**-8,33**	**5,23**
Package-Watt	0,06	**-7,00**
CPU-Watt	-0,05	**-6,88**
RAM-Watt	0,00*	**-1,47**

Hervorgehobene Werte markieren ein signifikantes Delta. Das * markiert Werte, wo der Median verwendet wird. Ein negativer Wert entspricht einem größeren Messwert bei Kotlin.

Die Tabelle 6.5 zeigt, dass es bei der Deserialisierung keinen signifikanten Unterschied zwischen Java und Kotlin gibt. Ansonsten benötigt Kotlin mehr Ressourcen bei der Deserialisierung. Bei der Serialisierung benötigt Java hingegen mehr Ressourcen und Kotlin mehr Leistung.

Die Direktkosten zeigen größere relative Deltas bei der Laufzeit von Kotlin in der Deserialisierung und den Ressourcen von Java in der Serialisierung. Bis auf die Echt- und Userzeit von Java bei der Serialisierung macht der Overhead 75 % bis 100 % aus. Auch hier wäre also eine höhere Last innerhalb einer Wiederholung sinnvoll.

6.3 Kombinationstest

Zuerst wird eine gemeinsame Tabelle für alle Operationsdomänen erstellt. Dann wird für jede Operationsdomäne die Tabelle betrachtet und mit dem Einzeltest in Verbindung gebracht. Danach werden statistische Tests ausgeführt, um das Ergebnis mit der Tabelle zu vergleichen. Dabei werden die Nullhypothesen abgelehnt, wenn ein p-Wert kleiner als 0,05 ist. Danach werden die Direktkosten mit dem Einzeltest verglichen.

Die hier besprochenen Verteilungen sind im Anhang als Abbildungen A.29 bis A.31 zu finden.
Bei der Echtzeitmessung der Dateien hat Java einen Ausreißer mit hoher Distanz nach rechts, der den Mittelwert beeinflussen würde, weswegen der Median stattdessen verwendet wird. Die Kernelzeitmessung der Listen hat keine Ausreißer oder Ausläufer, entsprechend wird hier der Mittelwert verwendet. Bei der Kernelzeitmessung der (De-)Serialisierung wird aufgrund der Ausläufer, wie in Unterkapitel 6.1.2 beschrieben, der Median verwendet.

Tabelle 6.6: Relatives Delta (%) der Mittelwerte der Kombinationstests

Softwaremetrik	**Dateien**	**Listen**	**(De-)Serialisierung**
Echtzeit	0,80*	**5,92**	**17,36**
Kernelzeit	**1,96**	**16,1**	0,00*
Userzeit	0,83	**13,27**	**22,66**
CPU-Auslastung	-0,26	**6,63**	**4,07**
Max. RAM	**1,23**	**5,25**	**-2,26**
Package-Watt	-0,45	**-9,25**	**2,96**
CPU-Watt	-0,23	**-10,58**	**3,75**
RAM-Watt	0,00	**-3,25**	**2,46**

Hervorgehobene Werte markieren ein signifikantes Delta. Das * markiert Werte, wo der Median verwendet wird. Ein negativer Wert entspricht einem größeren Messwert bei Kotlin.

6.3.1 Dateien

Die Tabelle 6.6 zeigt nur für die Kernelzeit und den maximalen RAM einen signifikanten Unterschied. Bei der Kernelzeit benötigt Java mehr Zeit, was ähnlich zum Einzeltest für die Grundoperation Schreiben ist. Der Unterschied beim maximalen RAM ist beim Einzeltest nicht signifikant oder zugunsten Javas gewesen, während hier Java mehr RAM benötigt.
Die geringen Standardabweichungen zeigen, dass die unterschiedliche Anzahl von Zeilen kaum Einfluss auf den Energie- und Ressourcenverbrauch hat.

Aufgrund des Shapiro-Wilk-Tests wird für jede Softwaremetrik die Nullhypothese einer Normalverteilung abgelehnt. Dafür kann für keine Softwaremetrik außer der Kernelzeit aufgrund des Kolmogorow-Smirnow-Tests die Nullhypothese einer gleichen Verteilung abgelehnt werden. Anhand des Welch's t-Tests kann nur für die Laufzeiten die Nullhypothese eines gleichen Mittelwerts abgelehnt werden. Dies widerspricht nur bei der Echt- und Userzeit sowie bei dem maximalen RAM der Tabelle 6.6. Für die Echtzeit sind die Mittelwerte aber unterschiedlich, weshalb für die Tabelle der Median genommen wurde. Für die Userzeit wird mit einem p-Wert von 0,049 die Nullhypothese nur knapp abgelehnt und für den maximalen RAM kann wie bei den Algorithmen das Signifikanzniveau hinterfragt werden.

Bei der Kombinationsmessung der Dateien kann mithilfe des Spearman'schen Rangkorrelationskoeffizienten die Nullhypothese bei den Laufzeiten abgelehnt werden. Für die Laufzeiten ist eine Unabhängigkeit damit unwahrscheinlich. Dabei haben alle Laufzeiten eine positive, ordinale Korrelation. Für die Echt- und Userzeit ist diese Korrelation stärker, mit einem t-Wert von 0,574 beziehungsweise 0,688. Die Korrelation der Kernelzeit fällt mit einem t-Wert von 0,232 schwächer aus, ist dennoch immer noch positiv. Alle übrigen Messdaten sind entweder unabhängig oder besitzen eine nicht monotone Korrelation.

Bei den Direktkosten ist die CPU-Auslastung im Vergleich zum Einzeltest gesunken, was zeigt, dass die Dateigrundoperationen keine hohe CPU-Last haben. Für die Laufzeit- und maximalen RAM-Messungen ist die Last im Vergleich zu den meisten Grundoperationen gestiegen. Nur die Lese-Grundoperation hatte eine höhere Auslastung.
Bei den Leistungsmessungen hat nur die RAM-Watt-Messung positive Direktkosten, wenn jedoch die Mittelwerte der Messungen betrachten werden, so sind die gemessenen Watt geringer als alle Grundoperationen bis auf Schreiben.

Da hier die gleichen internen Java-Klassen zur Umsetzung benutzt und auf der JVM ausgeführt wurden, ist es nicht überraschend, dass in den meisten Softwaremetriken keine signifikanten Unterschiede gefunden wurden. Für die CPU-Auslastung und die Leistung kann für die Operationsdomäne *Dateien* keine Aussage getroffen werden, da scheinbar erneut die Last zu gering ist.

6.3.2 Listen

Die Tabelle 6.6 zeigt für jede Softwaremetrik einen signifikanten Unterschied. Dabei verbraucht Java mehr Ressourcen und Kotlin benötigt mehr Leistung. Dies ist ein kompletter Wechsel zum Einzeltest, wo bei jeder Grundoperation mehr Ressourcen von Kotlin und mehr Leistung von Java benötigt wird.

Bis auf den maximalen RAM bei Kotlin wird bei jeder Softwaremetrik die Nullhypothese einer Normalverteilung abgelehnt. Eine gleiche Verteilung ist nur bei der Kernelzeit wahrscheinlich, wo durch den Welch's t-Test die Nullhypothese eines gemeinsamen Mittelwerts abgelehnt wird. Damit widersprechen die statistischen Tests nicht der Tabelle 6.6.

Für alle Messdaten kann die Nullhypothese von unabhängigen Datensätzen zwischen Java und Kotlin mithilfe des Spearman'schen Rangkorrelationskoeffizienten nicht abgelehnt werden.

Bei den Direktkosten macht der Overhead in allen Fällen den Großteil der Last aus. Die Direktkosten von Java und Kotlin haben bei der CPU-Auslastung eine Differenz von 2 %, was in etwa der Unterschied zwischen den Mittelwerten von Java und Kotlin ist. Der Unterschied könnte also durch die Direktkosten erklärbar sein.

Aufgrund der fehlenden Last kann zu der Operationsdomäne *Listen* keine Aussage getroffen werden. Die gemessenen Unterschiede können der Overhead oder eine ungünstige Variation sein.

6.3.3 (De-)Serialisierung

Die Tabelle 6.6 zeigt einen höheren Ressourcen- und Leistungsverbrauch bei Java, wobei die Kernelzeit keinen Unterschied aufweist und Kotlin mehr maximalen RAM verbraucht. Dieses Muster ist weder bei der Deserialisierung noch bei der Serialisierung im Einzeltest zu finden. Entsprechend scheint keine Grundoperation den Kombinationstest zu dominieren.

Wie bei der Operationsdomäne *Listen* wird auch hier bei jeder Softwaremetrik bis auf den maximalen RAM bei Kotlin die Nullhypothese einer Normalverteilung abgelehnt. Bei der Kernelzeit, der Package-Watt und der CPU-Watt-Messung kann eine gleiche Verteilung nicht abgelehnt werden. Nur bei der Kernelzeit kann es einen gemeinsamen Mittelwert geben, was die Wahl des Median bestätigt.

Die Nullhypothese von unabhängigen Datensätzen zwischen Java und Kotlin kann, wie bei der Operationsdomäne *Listen*, mithilfe des Spearman'schen Rangkorrelationskoeffizienten für keine Messdaten abgelehnt werden.

Wie bei der Serialisierung im Einzeltest ist die höchste Last bei den Direktkosten auf der Laufzeit, wobei auch die CPU-Auslastung einen Anstieg verzeichnet. Es gibt bei der CPU-Auslastung eine Differenz von 2,6 % zwischen Java und Kotlin, was größer als der gemessene Unterschied ist. Bei den Laufzeiten sind Kotlins Direktkosten geringer als Javas, was auch in etwa dem gemessenen Unterschied entspricht.

Es konnten leichte Unterschiede bei den Laufzeiten und der CPU-Auslastung in den Direktkosten festgestellt werden, aber eine höhere Last während der Messung könnte die Unterschiede stärker ausprägen. Für die Leistung und den maximalen RAM kann keine Aussage getroffen werden.

7 Fazit

Die Wahl von Rosetta Code als Datenquelle wurde anfangs gewählt, um Zeit zu sparen. Dabei hat sich gezeigt, dass dies für den Großteil der Algorithmen funktioniert hat. Es waren zwar Anpassungen nötig, diese waren aber nicht zeitintensiv. Die Verwendung der **A*-Algorithmus**-Implementierung bildet dabei eine Ausnahme, da es hier so viele Probleme gab, dass eine Eigenimplementierung eventuell weniger Zeit gekostet hätte.

Aus den nun gewonnenen Erkenntnissen lassen sich einige Beobachtungen ableiten. Die Messung des **A*-Algorithmus** hat gezeigt, dass der Algorithmus in Java eine höhere Laufzeit hat, die wahrscheinlich nicht der Overhead ist.
Die Operationsdomäne *Dateien* hat für die Lesen-Grundoperation und den Kombinationstest eine ausreichende Last. Eine Verringerung des Overheads würde die Messwerte genauer machen, aber wahrscheinlich keine weiteren Aussagen zur CPU-Auslastung oder der Leistung ermöglichen. Die Laufzeit und der benötigte maximale RAM scheinen für beide Programmiersprachen in der Operationsdomäne *Dateien* vergleichbar zu sein.
Bei der Operationsdomäne *(De-)Serialisierung* benötigt die Kotlin-Implementierung wahrscheinlich eine geringere Laufzeit und CPU-Auslastung.
Zu den restlichen Algorithmen und der Operationsdomäne *Listen* sind Aussagen wegen der Dominanz des Overheads nicht möglich.

Ein möglicher Schwachpunkt der Arbeit stellt das gewählte Signifikanzniveau von 1 % dar. Es gab einige Fälle, wo es laut dem Welch's t-Test eine hohe Wahrscheinlichkeit für denselben Mittelwert gab, der Unterschied der Mittelwerte aber als signifikant eingestuft worden. Dies war beispielsweise beim maximalen RAM bei der Kombinationsmessung der Operationsdomäne *Dateien* der Fall. Dies zeigt, dass die Höhe des Signifikanzniveaus eventuell zu niedrig gewählt ist.

Aufgrund der Verteilung der Messwerte, die meistens weder normalverteilt sind noch die gleiche Verteilung haben, ist die Anwendung der hier verwendeten statistischen Tests sehr gering. Der Shapiro-Wilk-Test und der Kolmogorow-Smirnow-Test haben alleine keine große Aussagekraft über die Unterschiede der Sprachen. Der Welch's t-Test konnte in den meisten Fällen nicht verwendet werden. Es sollten andere Methodiken und Tests für weitere Untersuchungen in Betracht gezogen werden. Insbesondere die Tests, die keine Annahme über die darunter liegende Verteilung machen, damit ein Großteil der Messungen abgedeckt werden kann. Es könnte aber auch an der Last der Experimente liegen. Bei der Operationsdomäne *Dateien* wurden häufiger gleiche Verteilungen festgestellt. Dort war die Last am höchsten. Ein anderer Faktor könnte die Größe der Variation und die Anzahl der Wiederholungen sein, da gerade im Kombinations- und Variationstest nur 100 Wiederholungen ausgeführt werden. Ist die Variation der Messwerte bei mindestens einer Programmiersprache entsprechend hoch, können der Shapiro-Wilk-Test und der Kolmogorow-Smirnow-Test eine Normalverteilung beziehungsweise eine gleiche Verteilung fälschlicherweise ausschließen. Durch die hohe Last bei der Operationsdomäne *Dateien* könnte die Variation weniger ins Gewicht fallen als bei den anderen Operationsdomänen. Eine erhöhte Wiederholungsanzahl oder eine Reduktion der Variation könnte bei den statistischen Tests zu einer erhöhten Aussagekraft führen.

Eine Korrelation zwischen den Programmiersprachen konnte in den wenigsten Fällen festgestellt werden. Dies war nur für den maximalen RAM des *A*-Algorithmus* und die Laufzeiten der Operationsdomäne *Dateien* möglich. Der p-Wert beim *A*-Algorithmus* liegt mit 0,046 jedoch sehr nahe am festgelegten α von 0,05 und auch die Korrelation ist mit einem Wert von -0,2 schwach. Für die Operationsdomäne *Dateien* sieht es anders aus, da hat die Kernelzeit den höchsten p-Wert mit einem Wert von 0,02. Die Kernelzeit hat auch eine schwache Korrelation. Die Echt- und Userzeit haben eine hohe monotone Korrelation. Bei den Messdaten, wo die Nullhypothese nicht abgelehnt wurde, könnten dennoch korrelieren, jedoch nicht monoton. Hier wäre eine nähere Betrachtung sinnvoll. Auch hier könnten mehr Wiederholungen oder eine generelle Reduktion der Variation, durch zum Beispiel eine Verringerung von externen Einflüssen auf das Systemn hilfreich sein.

Die Ergebnisse von [13, 55], die in Unterkapitel 2.5 beschrieben werden, konnten zwar in einigen Messungen beobachtet werden, die Beobachtungen sind aber aufgrund der Direktkosten nicht aussagekräftig. Das Ergebnis, dass die Unterschiede nicht signifikant sind, welches in [36] auftritt, kann von dieser Ausarbeitung zum Teil bestätigt werden.

Eine Beobachtung, die sich durch alle Experimente zieht, ist eine zu geringe Last bei den Grundoperationen. Es gibt zwar einige Messungen, in denen für bestimmte Softwaremetriken Aussagen getroffen werden können, aber für viele Softwaremetriken war die Last zu gering. Gerade die Leistungsmessungen haben sehr geringe Direktkosten. Eventuell haben die Grundoperationen sogar keine Auswirkung auf die Leistungsmessung, da zum Beispiel die JVM den Großteil der Leistung benötigt. Die einzige Ausnahme bildet hier die RAM-Watt-Messung bei der Operationsdomäne *Dateien*, wo höhere Direktkosten mehrmals auftreten. Dies ist ein möglicher Hinweis darauf, dass gerade die Leistungsmessungen, aber auch die Ressourcenmessungen nicht unbedingt auf einzelne Grundoperationen heruntergebrochen werden sollten, sondern im größeren Umfang getestet werden müssen.

Einzig bei der Operationsdomäne *(De-)Serialisierung* kann eine eindeutige Sprachempfehlung getroffen werden. Kotlin sollte bevorzugt benutzt werden, sofern die Laufzeit ein wichtiges Entscheidungskriterium ist.

7.1 Ausblick

Bei der Untersuchung der Forschungsfrage sind einige weitere Fragen aufgekommen. Da die Forschungsfrage nur einen sehr kleinen Bereich des Forschungsgebiets zum Energie- und Ressourcenverbrauch von Programmiersprachen abbildet, gibt dies Hinweise auf den wahren Umfang der möglichen Forschung. Die Untersuchung in dieser Arbeit hat zusätzlich gezeigt, dass die Forschung mit vielen trickreichen Eigenheiten verbunden ist. Diese Ausarbeitung kann als Ausgangspunkt für weitere Forschungen in dem Bereich genutzt werden.

Eine Möglichkeit wäre es, die statistische Signifikanz weiter zu untersuchen. Gerade die gewählte Höhe des Signifikanzniveaus, das unterschiedliche Delta bei der Testmessung oder eine zu geringe Last haben in dieser Arbeit Fragen aufgeworfen. Dabei kann die Leitfrage der zu wählenden Höhe des Signifikanzniveaus bei dem Vergleich von Programmiersprachen gewählt werden. Zusätzlich kann näher untersucht werden, welche Maßnahmen getroffen werden können, um externe Faktoren und Einflüsse zu reduzieren.

Eine Möglichkeit, um Fluktuationen reduzieren zu können, könnte das Durchführen der Messungen mit erhöhter Prozesspriorität sein. Es wäre ebenfalls hilfreich, wenn die Ausführungsplattform von allen nicht benötigten Programmen und Prozessen befreit werden. Um die Eindeutigkeit bei Verteilungen zu verbessern, sollte eine deutlich erhöhte Auslastung von den Messungen erzeugt werden, da sonst der Overhead die Messung dominiert.

Die beobachtete Clusterbildung beim Lesen der Dateien ist ungeklärt geblieben. Hier sollte die Ursache näher untersucht werden.

Da beide Programmiersprachen in dieser Arbeit für die JVM kompiliert wurden, stellen sich die Fragen, welchen Einfluss Compiler Flags auf den Ressourcen- und Leistungsverbrauch haben und wie vergleichbar der Kotlin/Native-Compiler für Kotlin und GraalVM für Java ist.

Das im Rahmen dieser Ausarbeitung eingeführte Testframework ist darüber hinaus vertikal sowie horizontal skalierbar. Eine horizontale Skalierung bedeutet in diesem Zusammenhang, dass dieselbe Betrachtungsebene beibehalten wird, während bei der vertikalen Skalierung die Betrachtungsebene verändert wird.

Bei einer horizontalen Skalierung könnten mehr Operationsdomänen und Algorithmen entweder mit möglichst gleichen Implementierungen, wie in dieser Ausarbeitung, oder mit spracheigenen Features umgesetzt werden, um einen sehr breiten Programmiersprachvergleich zu ermöglichen. Es können auch mehrere Operationsdomänen in einem Experiment verbunden werden, um näher an eine reelle Anwendung zu kommen.

Für die vertikale Skalierung gibt es verschiedene Betrachtungsebenen.
Es könnte ein Projekt in beiden Programmiersprachen umgesetzt werden, bei dem sprachspezifische Features zum Einsatz kommen, um eine reelle Anwendung zu simulieren.
In einer anderen Betrachtungsebene wird auch ein Projekt umgesetzt, allerdings können hier Bibliotheken und Frameworks verwendet werden, wodurch nicht mehr die reine Programmiersprache, sondern das ganze Ökosystem untersucht wird.
Die letzte hier vorgestellte Möglichkeit wäre es, ein Projekt umzusetzen, in dem Java und Kotlin kombiniert wird. Dieses Projekt könnte mit jeweils einem reinen Java-Projekt verglichen werden, um abzuschätzen, was ein Wechsel zu Kotlin in einem laufenden Projekt für Auswirkungen hat.

Literaturverzeichnis

[1] Amazon.com Inc.: *Amazon Homepage*. – URL https://www.amazon.de/. – Zugriff am: 28.01.2024

[2] AndroidTimeMachine Team: *AndroidTimeMachine*. 2020. – URL https://androidtimemachine.github.io/. – Zugriff am: 18.12.2023

[3] Ardito, Luca ; Coppola, Riccardo ; Malnati, Giovanni ; Torchiano, Marco: Effectiveness of Kotlin vs. Java in android app development tasks. In: *Information and Software Technology* 127 (2020), S. 106374. – URL https://doi.org/10.1016/j.infsof.2020.106374. – ISSN 0950-5849

[4] Benchmarksgame Team: *Computer Language Benchmarks Game*. – URL https://benchmarksgame-team.pages.debian.net/benchmarksgame/. – Zugriff am: 18.12.2023

[5] Benčević, Marin: *What is Lazy Evaluation? — Programming Word of the Day*. 2018. – URL https://medium.com/background-thread/what-is-lazy-evaluation-programming-word-of-the-day-8a6f4410053f. – Zugriff am: 11.12.2023

[6] Blobel, Volker ; Lohrmann, Erich: *Statistische und numerische Methoden der Datenanalyse*. 2. URL http://www-library.desy.de/preparch/books/BloLoBuch.pdf, 2012

[7] Brown, Len: *Turbostat Manual*. – URL https://manpages.debian.org/testing/linux-cpupower/turbostat.8.en.html. – Zugriff am: 21.01.2024

[8] Daigle, Kyle ; Github Staff: *Octoverse: The state of open source and rise of AI in 2023*. 2023. – URL https://github.blog/2023-11-08-the-state-of-open-source-and-ai/#the-most-popular-programming-languages. – Zugriff am: 28.01.2024

[9] eBay Inc.: *eBay Homepage*. – URL https://www.ebay.com/. – Zugriff am: 28.01.2024

[10] Georgiou, Stefanos ; Kechagia, Maria ; Spinellis, Diomidis: Analyzing Programming Languages' Energy Consumption: An Empirical Study. (2017), Nr. 42. – URL https://doi.org/10.1145/3139367.3139418. ISBN 9781450353557

[11] Georgiou, Stefanos ; Rizou, Stamatia ; Spinellis, Diomidis: Software Development Lifecycle for Energy Efficiency: Techniques and Tools. In: *ACM Comput. Surv.* 52 (2019), Nr. 4. – URL https://doi.org/10.1145/3337773. – ISSN 0360-0300

[12] Github Inc.: *GitHub Homepage*. – URL https://github.com/. – Zugriff am: 28.01.2024

[13] Glukhov, D.V. ; Mullayanov, B.I.: The Performance Evaluating of Kotlin and Java Implementations. In: *2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)* (2020), S. 1–7. – URL https://doi.org/10.1109/FarEastCon50210.2020.9271621

[14] GNU: *time(1) - Linux manual page*. – URL https://man7.org/linux/man-pages/man1/time.1.html. – Zugriff am: 21.01.2024

[15] ISO/IEC JTC1/SC22/WG14: *ISO/IEC 9899:2018*. – URL https://www.iso.org/standard/74528.html. – Zugriff am: 21.05.2024

[16] JetBrains s.r.o.: *The State of Developer Ecosystem 2023*. 2023. – URL https://www.jetbrains.com/lp/devecosystem-2023/. – Zugriff am: 28.01.2024

[17] JetBrains s.r.o.: *The State of Developer Ecosystem 2023*. 2023. – URL https://www.jetbrains.com/lp/devecosystem-2023/languages/. – Zugriff am: 28.01.2024

[18] Khoirom, Selina ; Sonia, Moirangthem ; Laikhuram, Borishphia ; Laishram, Jaeson ; Singh, Tekcham D.: Comparative Analysis of Python and Java for Beginners. In: *International Research Journal of Engineering and Technology* 7 (2020), Nr. 8, S. 4384 – 4407. – URL https://www.irjet.net/volume7-issue08. – ISSN 2395-0056

[19] Kotlin Foundation: *Kotlin docs*. – URL https://kotlinlang.org/docs/home.html. – Zugriff am: 19.12.2023

[20] Kotlin Foundation: *Comparison to Java*. 2022. – URL https://kotlinlang.org/docs/comparison-to-java.html. – Zugriff am: 19.12.2023

[21] Kotlin Foundation: *Kotlin compiler options*. 2023. – URL https://kotlinlang.org/docs/compiler-reference.html. – Zugriff am: 13.12.2023

[22] Kotlin Foundation: *Kotlin Native*. 2023. – URL https://kotlinlang.org/docs/native-overview.html. – Zugriff am: 13.12.2023

[23] Kruglov, Artem ; Succi, Giancarlo: *Developing Sustainable and Energy-Efficient Software Systems*. 1. Springer Nature Switzerland AG, 2023. – URL https://doi.org/10.1007/978-3-031-11658-2

[24] Martinez, Matias ; Gois Mateus, Bruno: Why Did Developers Migrate Android Applications From Java to Kotlin? In: *IEEE Transactions on Software Engineering* 48 (2022), Nr. 11, S. 4521–4534. – URL https://doi.org/10.1109/TSE.2021.3120367

[25] McDonald, J. H.: *Handbook of Biological Statistics*. 3. Sparky House Publishing, 2014. – URL http://www.biostathandbook.com/index.html

[26] McKenna, Hazel J. ; Chang, Leo ; Brinkerhoff, M. R.: *Numeracy*. Utah Valley University, 2023. – URL https://uen.pressbooks.pub/uvumqr/front-matter/cover/. – ISBN 9798989680207

[27] O'Grady, Stephen: *The RedMonk Programming Language Rankings: January 2023*. 2023. – URL https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/. – Zugriff am: 28.01.2024

[28] Oracle: *GraalVM Documentation*. – URL https://www.graalvm.org/latest/docs/. – Zugriff am: 19.12.2023

[29] Oracle: *JAR File Specification*. – URL https://docs.oracle.com/en/java/javase/22/docs/specs/jar/jar.html. – Zugriff am: 03.05.2024

[30] Oracle: *Learn Java*. – URL https://docs.oracle.com/en/java/. – Zugriff am: 03.05.2024

[31] Oracle: *OpenJDK*. – URL https://openjdk.org/. – Zugriff am: 12.04

[32] Oracle: *Oracle*. – URL https://www.oracle.com/. – Zugriff am: 19.12.2023

[33] Oracle: *Native Image*. 2023. – URL https://www.graalvm.org/latest/reference-manual/native-image/. – Zugriff am: 13.12.2023

[34] Pereira, Rui ; Couto, Marco ; Ribeiro, Francisco ; Rua, Rui ; Cunha, Jácome ; Paulo Fernandes, Jo ao ; Saraiva, Jo ao: Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate? (2017), S. 256–267. – URL https://doi.org/10.1145/3136014.3136031. ISBN 9781450355254

[35] Pereira, Rui ; Couto, Marco ; Ribeiro, Francisco ; Rua, Rui ; Cunha, Jácome ; Paulo Fernandes, Jo ao ; Saraiva, Jo ao: Ranking programming languages by energy efficiency. In: *Science of Computer Programming* 205 (2021), S. 102609. – URL https://doi.org/10.1016/j.scico.2021.102609. – ISSN 0167-6423

[36] Peters, Michael ; Scoccia, Gian L. ; Malavolta, Ivano: How does Migrating to Kotlin Impact the Run-time Efficiency of Android Apps? In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2021), S. 6–46. – URL https://doi.org/10.1109/SCAM52516.2021.00014

[37] Python Software Foundation: *Python Documentation*. – URL https://docs.python.org/3.10/library/subprocess.html. – Zugriff am: 01.05.2024

[38] Python Software Foundation: *Python Documentation*. – URL https://docs.python.org/3. – Zugriff am: 01.05.2024

[39] Red Hat: *What is a REST API?* 2020. – URL https://www.redhat.com/en/topics/api/what-is-a-rest-api. – Zugriff am: 21.01.2024

[40] Rosetta Code: *A* search algorithm*. – URL https://rosettacode.org/wiki/A*_search_algorithm. – Zugriff am: 11.03.2024

[41] Rosetta Code: *Binary search*. – URL https://rosettacode.org/wiki/Binary_search. – Zugriff am: 11.03.2024

[42] Rosetta Code: *Huffman coding*. – URL https://rosettacode.org/wiki/Huffman_coding. – Zugriff am: 11.03.2024

[43] Rosetta Code: *Matrix chain multiplication*. – URL https://rosettacode.org/wiki/Matrix_chain_multiplication. – Zugriff am: 11.03.2024

[44] Rosetta Code: *Rosetta Code*. – URL https://rosettacode.org/wiki/Rosetta_Code. – Zugriff am: 05.12.2023

[45] Rosetta Code: *Sorting algorithms/Heapsort*. – URL https://rosettacode.org/wiki/Sorting_algorithms/Heapsort. – Zugriff am: 11.03.2024

[46] Rosetta Code: *Sorting algorithms/Quicksort*. – URL https://rosettacode.org/wiki/Sorting_algorithms/Quicksort. – Zugriff am: 11.03.2024

[47] Rosetta Code: *Tokenize a string with escaping*. – URL https://rosettacode.org/wiki/Tokenize_a_string_with_escaping. – Zugriff am: 11.03.2024

[48] SciPy community: *SciPy documentation*. – URL https://docs.scipy.org/doc/scipy/index.html. – Zugriff am: 21.01.2024

[49] SciPy community: *scipy.stats.shapiro*. – URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html. – Zugriff am: 21.01.2024

[50] Shapiro, S. S. ; Wilk, M. B.: An analysis of variance test for normality (complete samples)†. In: *Biometrika* 52 (1965), 12, Nr. 3-4, S. 591–611. – URL https://doi.org/10.1093/biomet/52.3-4.591. – ISSN 0006-3444

[51] Stackoverflow: *2023 Developer Survey*. 2023. – URL https://survey.stackoverflow.co/2023/#technology. – Zugriff am: 28.01.2024

[52] TIOBE Software BV: *TIOBE Programming Community Index Definition*. – URL https://www.tiobe.com/tiobe-index/programminglanguages_definition/. – Zugriff am: 25.01.2024

[53] TIOBE Software BV: *TIOBE Index for December 2023*. 2023. – URL http://www.tiobe.com/tiobe-index/. – Zugriff am: 28.01.2024

[54] Walmart Inc.: *Walmart Homepage*. – URL https://www.walmart.com/. – Zugriff am: 28.01.2024

[55] Wasilewski, Kamil ; Zabierowski, Wojciech: A Comparison of Java, Flutter and Kotlin/Native Technologies for Sensor Data-Driven Applications. In: *Sensors* 21 (2021), Nr. 10, S. 6–46. – URL https://doi.org/10.3390/s21103324. – ISSN 1424-8220

[56] Wilkens, Andreas: *Computerarchitektur und Betriebssysteme*. – URL https://vfhcab.eduloop.de/loop/Kernel-Mode,_User-Mode_und_Systemaufrufe. – Zugriff am: 20.05.2024

[57] İmamoğlu, Meltem Y. ; Çetinkaya, Deniz: A rule based decision support system for programming language selection. (2017), S. 71–75. – URL https://doi.org/10.1109/ICKEA.2017.8169904

A Anhang

Abbildungen

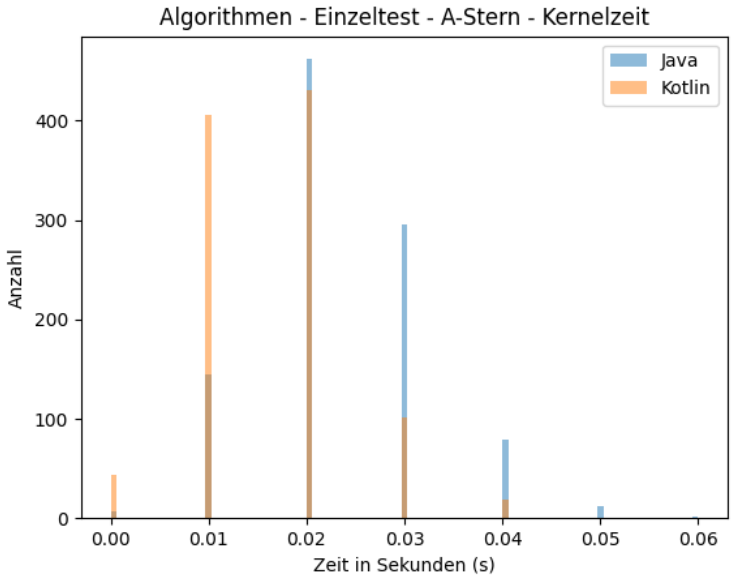


Abbildung A.1: Verteilung der Kernelzeit-Messung für den A*-Algorithmus

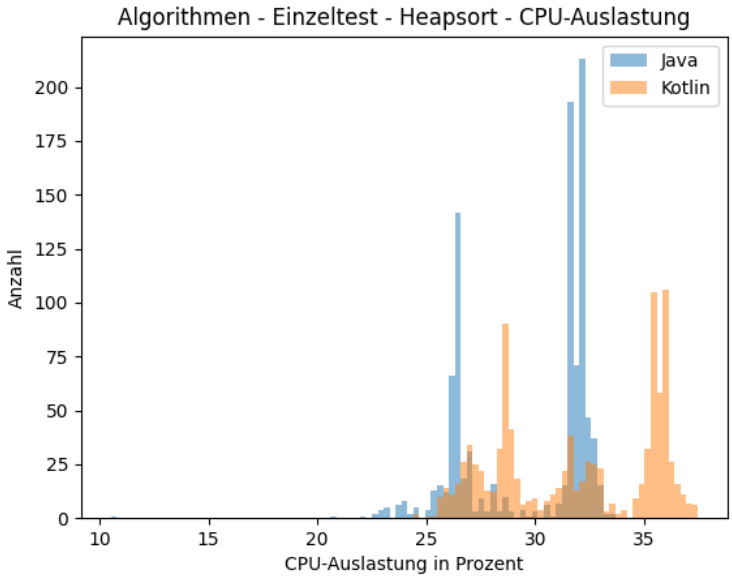


Abbildung A.2: Verteilung der CPU-Auslastung-Messung für den Heapsort

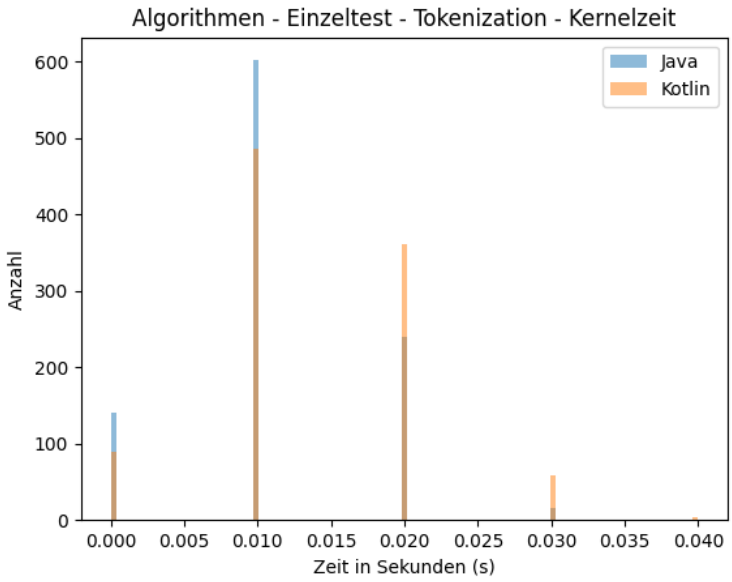


Abbildung A.3: Verteilung der Kernelzeit-Messung für die Tokenization

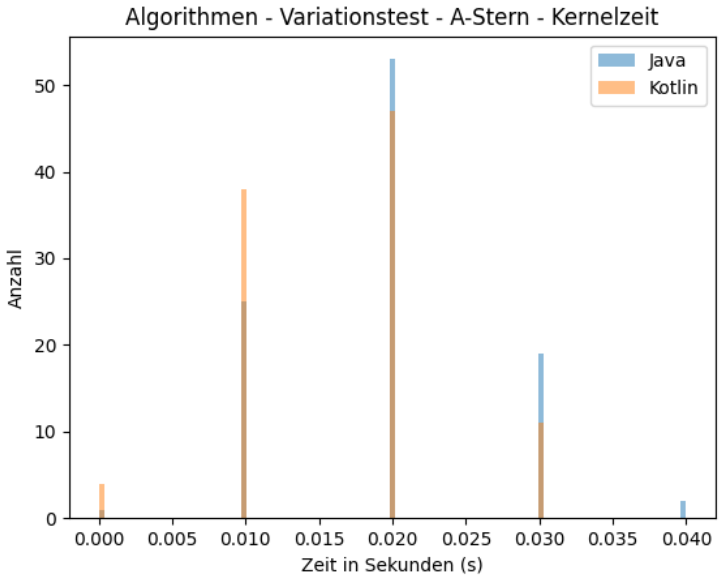


Abbildung A.4: Verteilung der Kernelzeit-Messung für den A*-Algorithmus

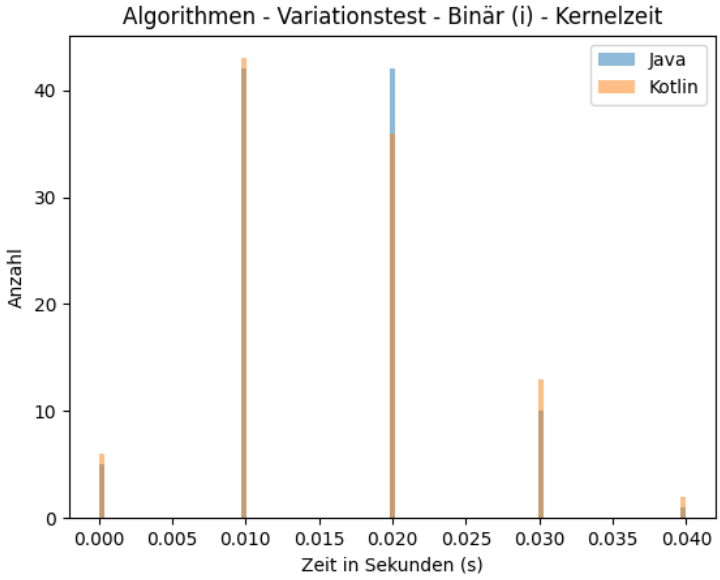


Abbildung A.5: Verteilung der Kernelzeit-Messung für die iterative binäre Suche

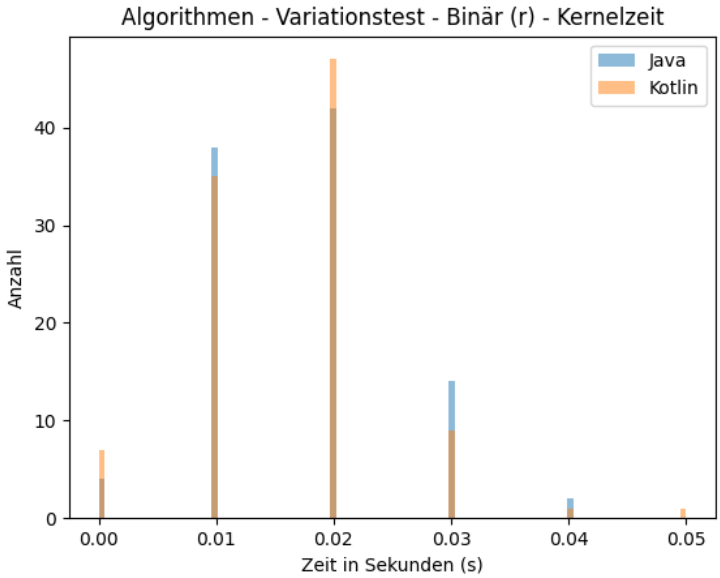


Abbildung A.6: Verteilung der Kernelzeit-Messung für den rekursiven binäre Suche

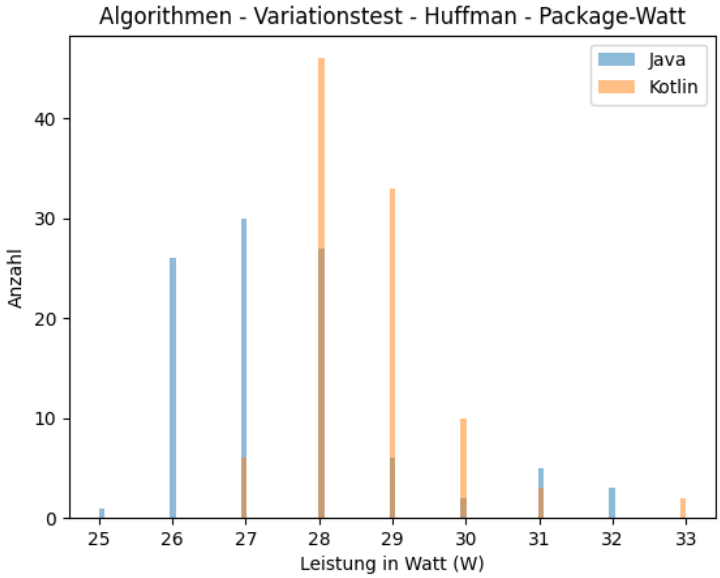


Abbildung A.7: Verteilung der Package-Watt-Messung für das Huffman Encoding

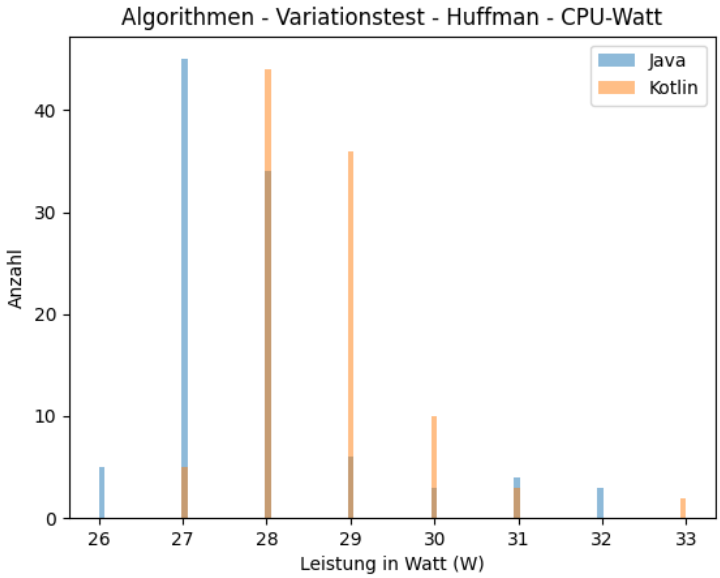


Abbildung A.8: Verteilung der CPU-Watt-Messung für das Huffman Encoding

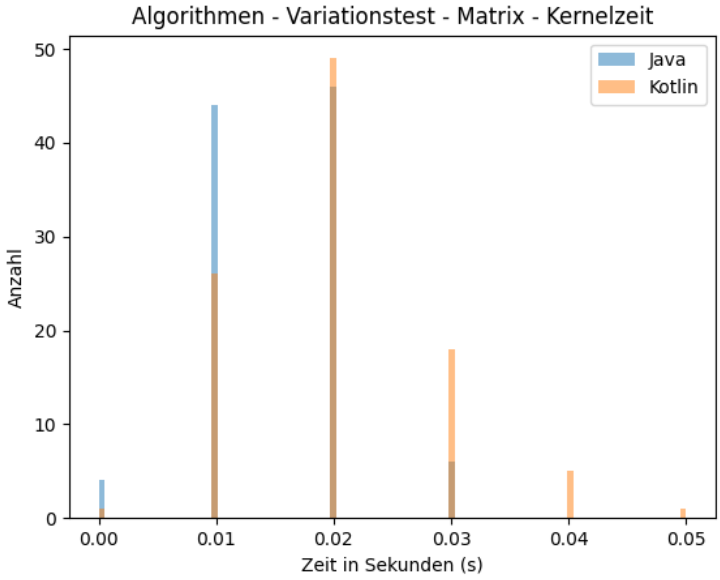


Abbildung A.9: Verteilung der Kernelzeit-Messung für die Matrix-Ketten-Manipulation

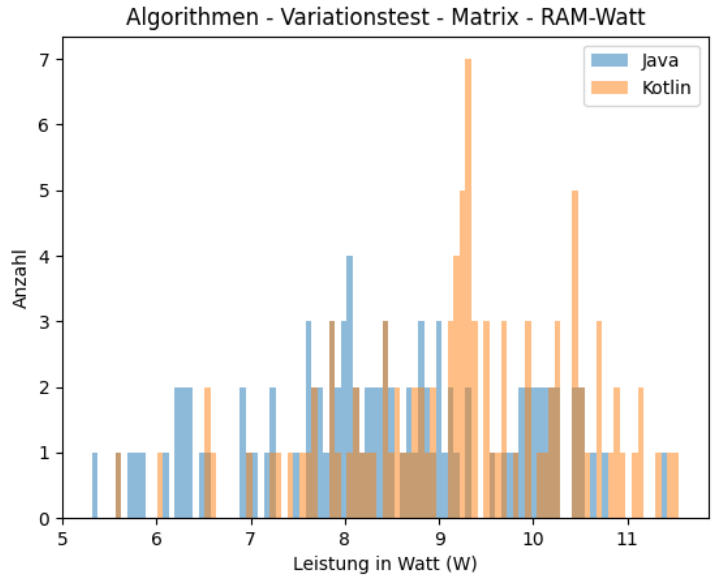


Abbildung A.10: Verteilung der RAM-Watt-Messung für die Matrix-Ketten-Manipulation

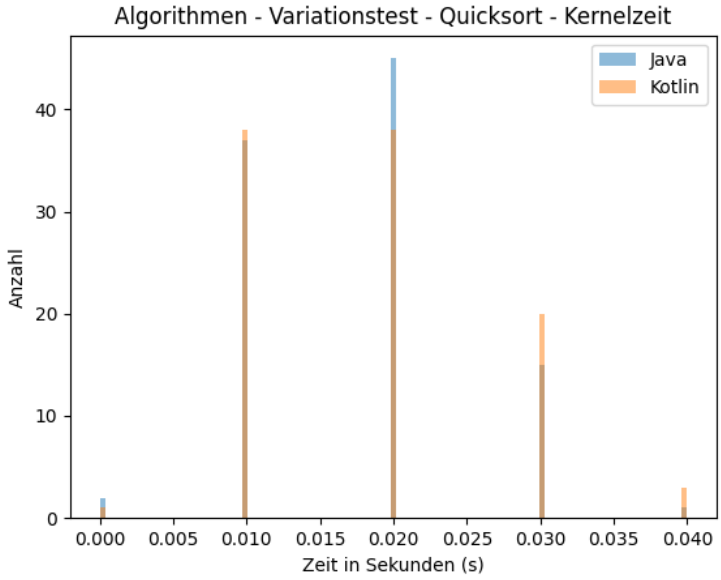


Abbildung A.11: Verteilung der Kernelzeit-Messung für den Quicksort

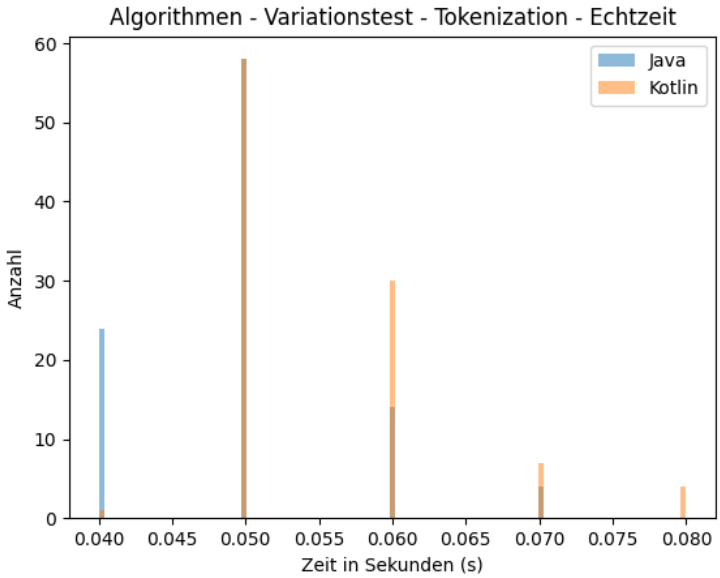


Abbildung A.12: Verteilung der Echtzeit-Messung für die Tokenization

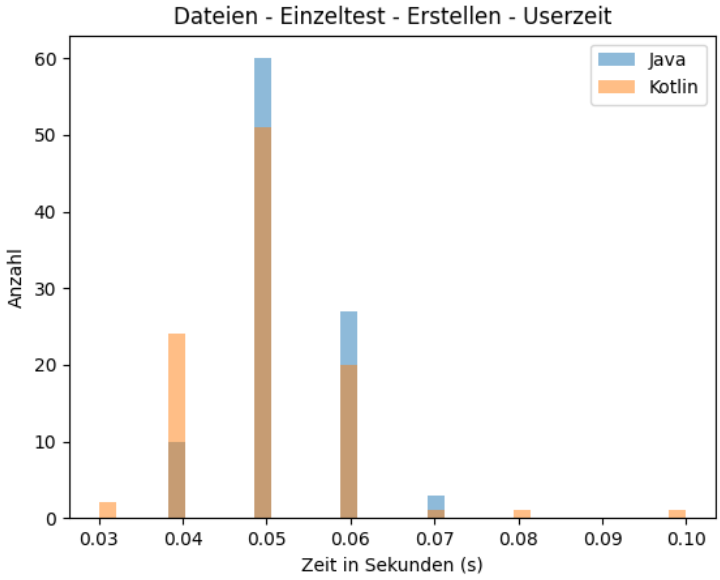


Abbildung A.13: Verteilung der Userzeit-Messung für Dateien Erstellen

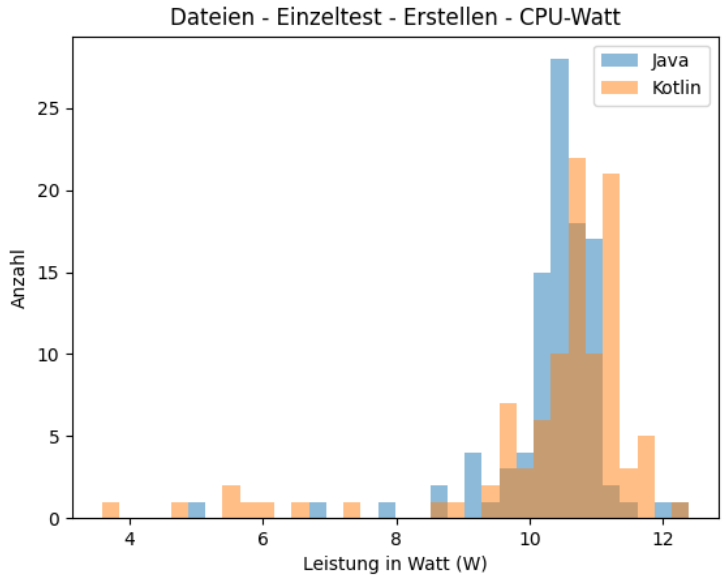


Abbildung A.14: Verteilung der CPU-Watt-Messung für Dateien Erstellen

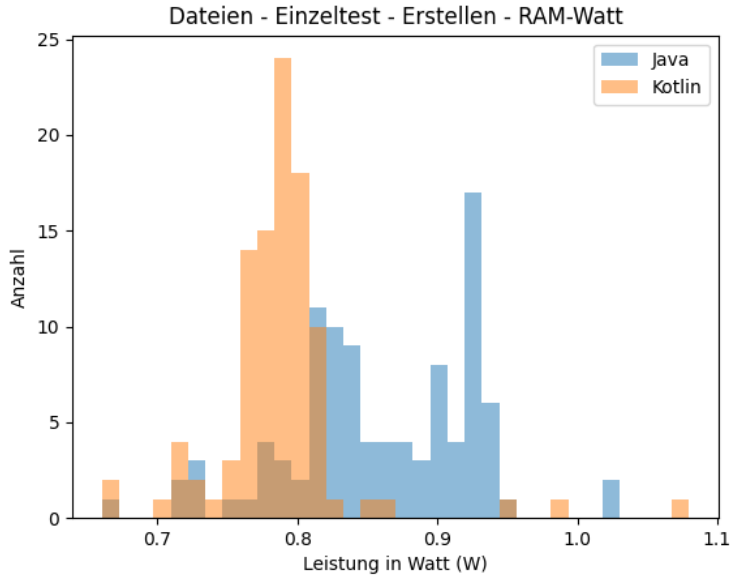


Abbildung A.15: Verteilung der RAM-Watt-Messung für Dateien Erstellen

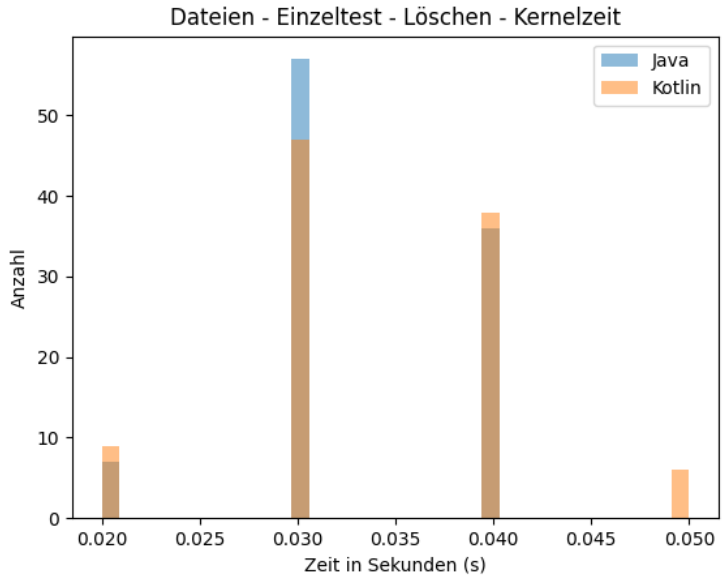


Abbildung A.16: Verteilung der Kernelzeit-Messung für Dateien Löschen

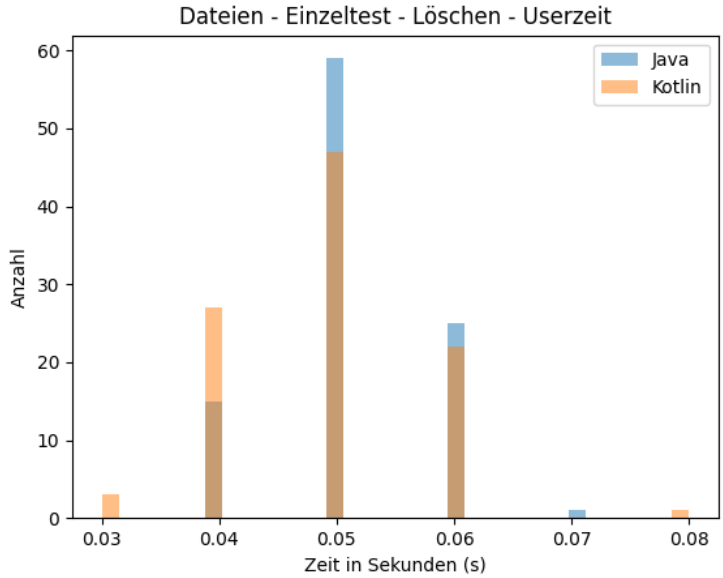


Abbildung A.17: Verteilung der Userzeit-Messung für Dateien Löschen

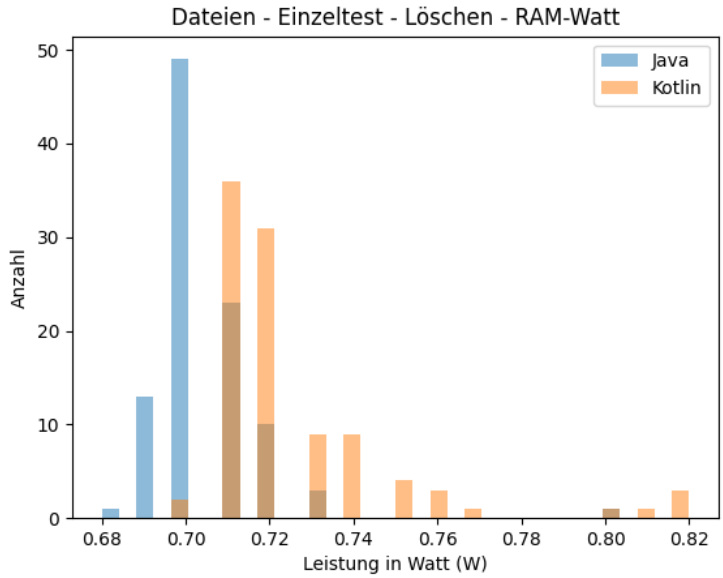


Abbildung A.18: Verteilung der RAM-Watt-Messung für Dateien Löschen

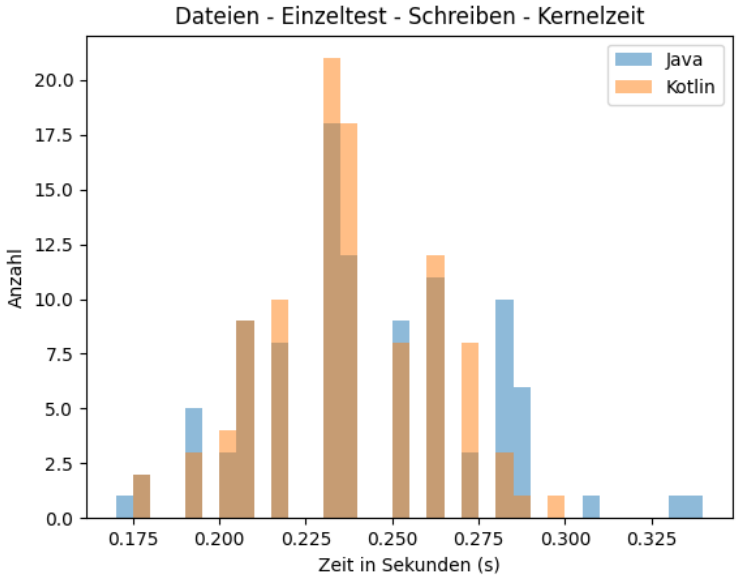


Abbildung A.19: Verteilung der Kernelzeit-Messung für Dateien Schreiben

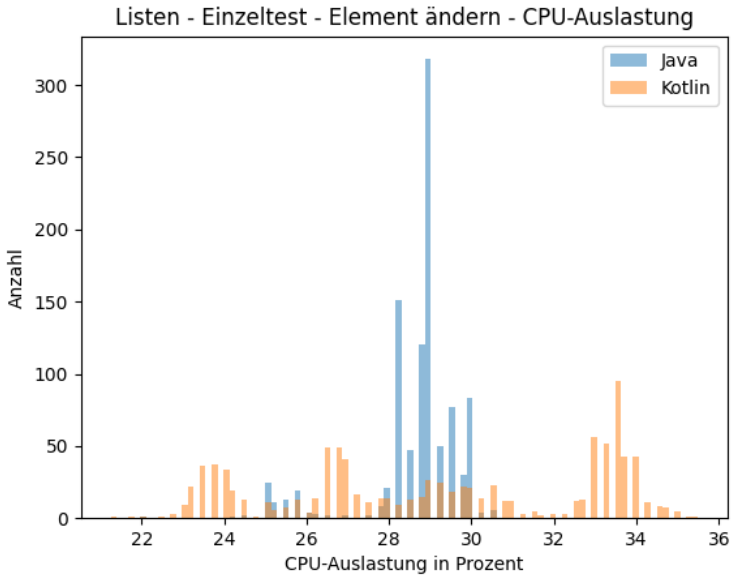


Abbildung A.20: Verteilung der CPU-Auslastung-Messung für Listen - Element ändern

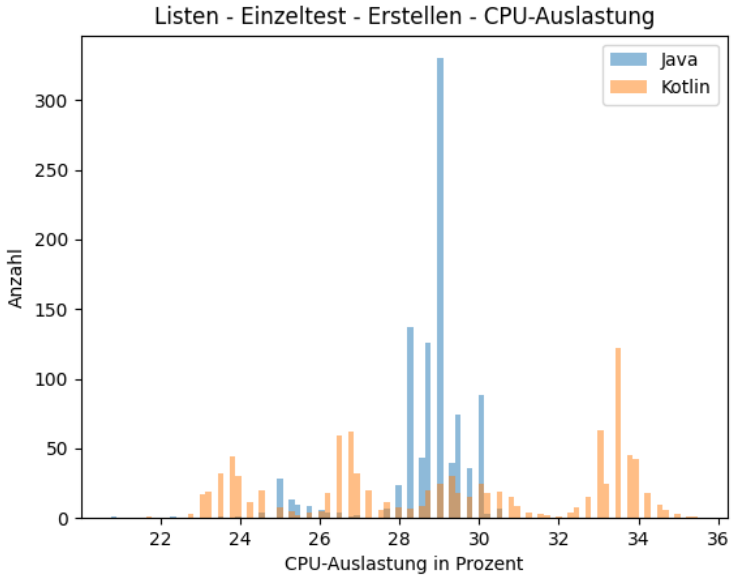


Abbildung A.21: Verteilung der CPU-Auslastung-Messung für Listen - Erstellen

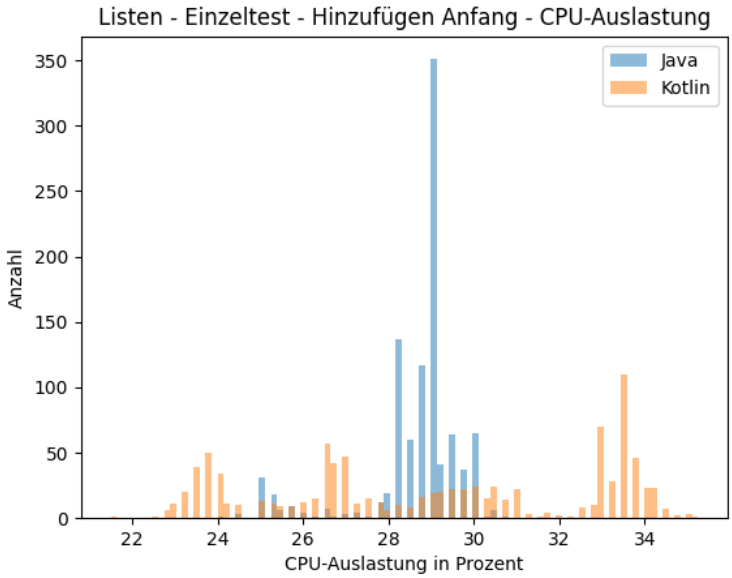


Abbildung A.22: Verteilung der CPU-Auslastung-Messung für Listen - Hinzufügen Anfang

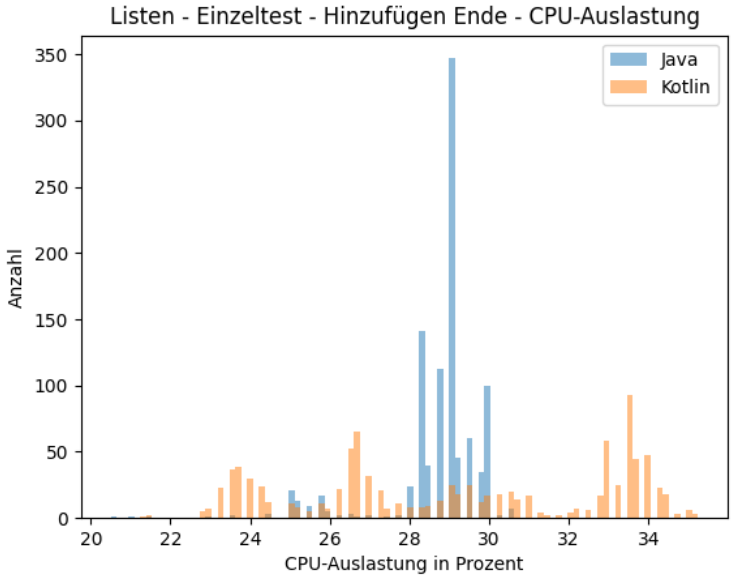


Abbildung A.23: Verteilung der CPU-Auslastung-Messung für Listen - Hinzufügen Ende

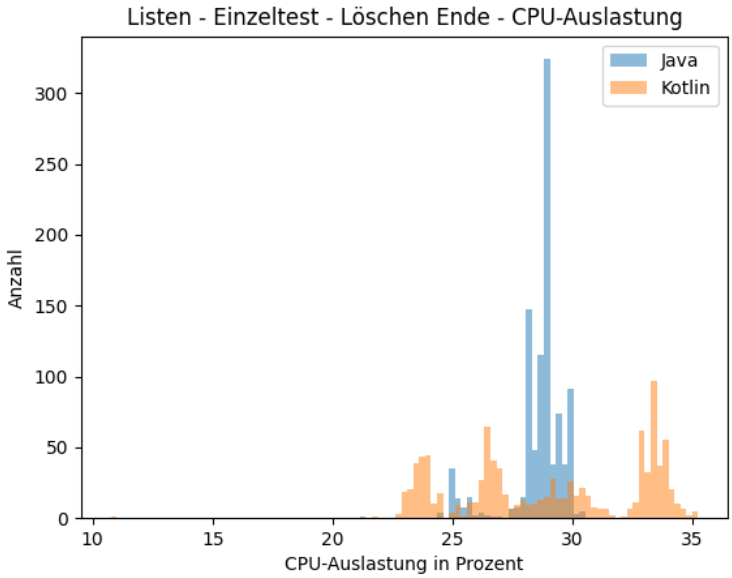


Abbildung A.24: Verteilung der CPU-Auslastung-Messung für Listen - Löschen Ende

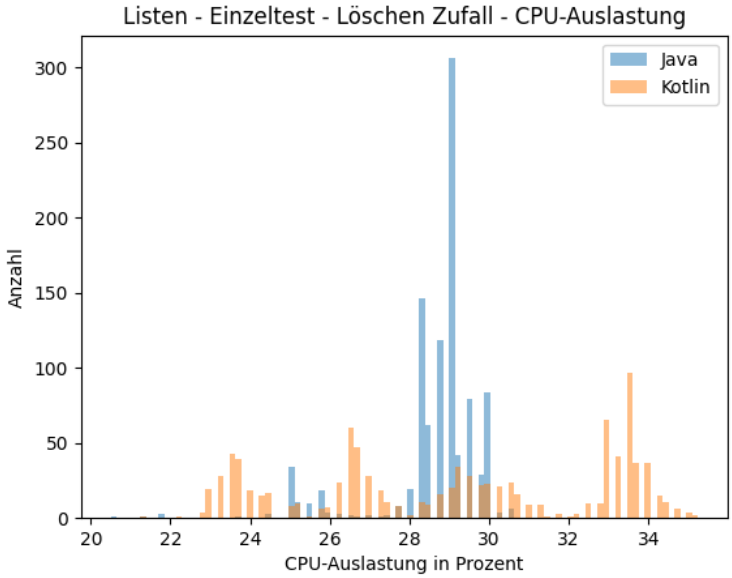


Abbildung A.25: Verteilung der CPU-Auslastung-Messung für Listen - Löschen Zufall

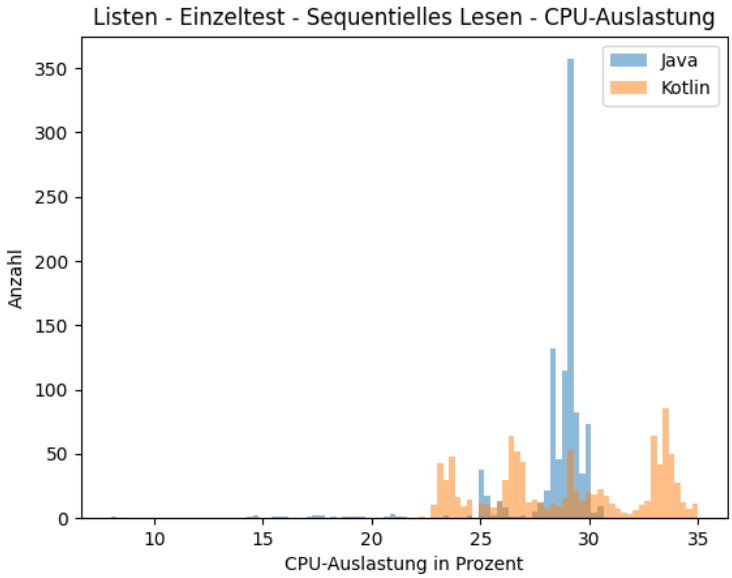


Abbildung A.26: Verteilung der CPU-Auslastung-Messung für Listen - Sequentielles Lesen

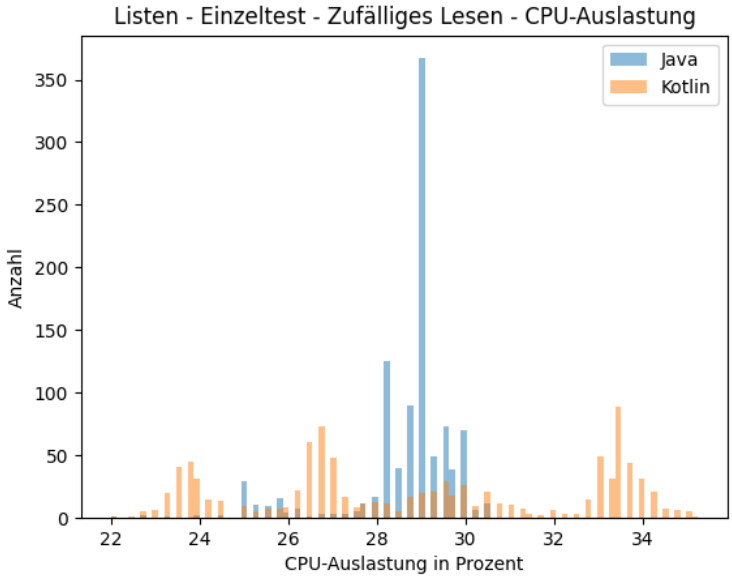


Abbildung A.27: Verteilung der CPU-Auslastung-Messung für Listen - Zufälliges Lesen

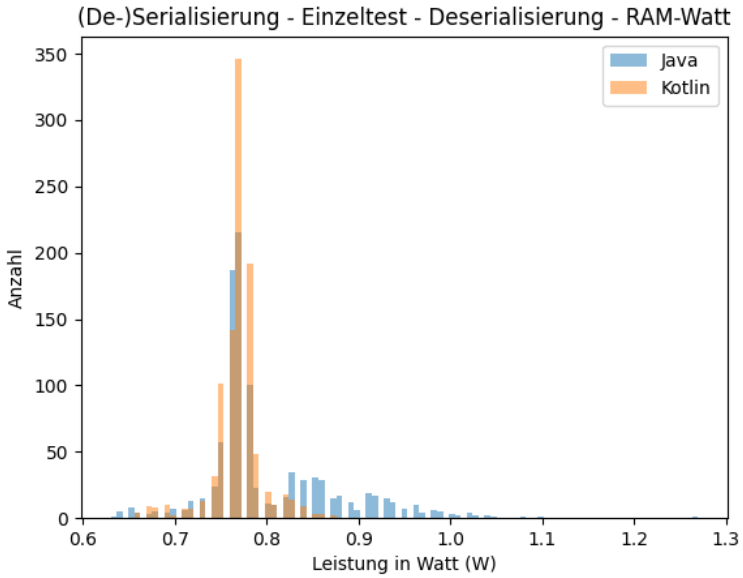


Abbildung A.28: Verteilung der RAM-Watt-Messung für die Deserialisierung

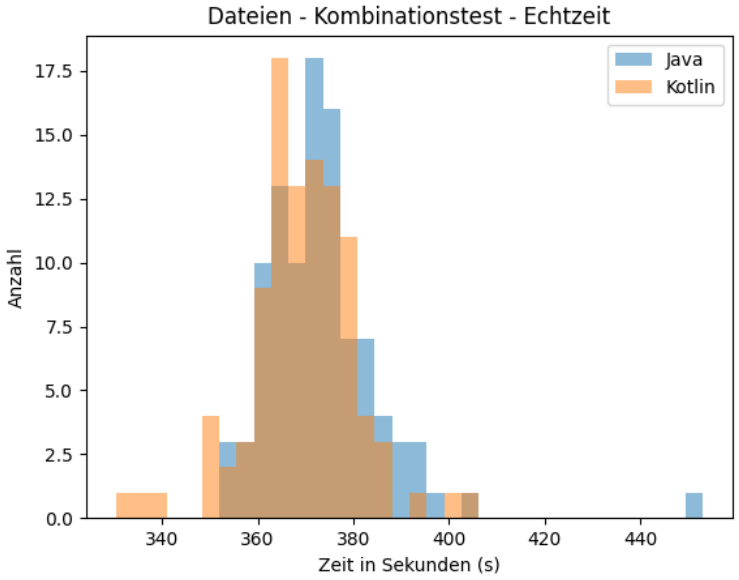


Abbildung A.29: Verteilung der Echtzeit-Messung für den Dateien-Kombinationstest

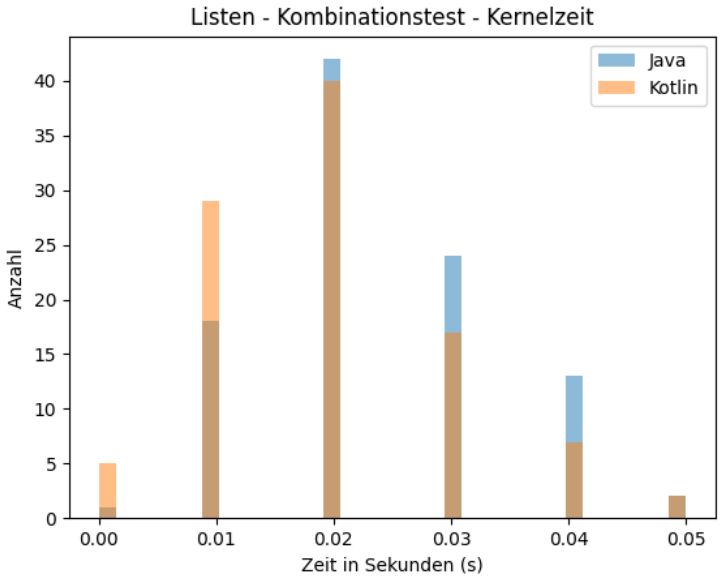


Abbildung A.30: Verteilung der Kernelzeit-Messung für den Listen-Kombinationstest

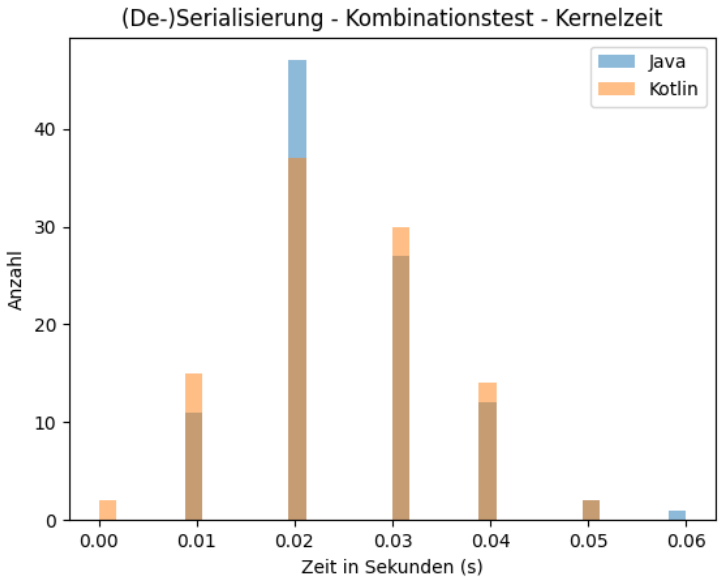


Abbildung A.31: Verteilung der Kernelzeit-Messung für den (De-)Serialisierung-Kombinationstest

Abläufe

Ablauf A.1 Shapiro-Wilk-Test nach [50]

1. Sortiere die Messergebnisse so, dass es gilt: $x_1 \leq x_2 \leq \cdots \leq x_n$
2. Berechne:
$$S^2 = (n-1) * \sigma^2$$
3. Mit a_{n-k+1} aus Tabelle 5 in [50]:
$$n = \begin{cases} 2m, & n \text{ ist gerade} \\ 2m+1, & n \text{ ist ungerade} \end{cases}$$
$$b = \sum_{k=1}^{m} a_{n-k+1} * x_k$$
4. Berechne: $W = \frac{b^2}{S^2}$
5. W mit Tabelle 6 in [50] vergleichen
   - Liegt W unter dem Wert in der Tabelle ist das Ergebnis signifikant und die Messergebnisse nicht normalverteilt

Ablauf A.2 Kolmogorov-Smirnov-Test nach [6]

1. Verteilungsfunktion
$$F(x) = \int_{-\infty}^{x} f(x')dx'$$
2. Sortiere die Messergebnisse nach ihrer Größe
3. Kumulative Größe konstruieren:
$$F_n(x) = \frac{\text{Anzahl der } x_i\text{-Werte} \leq x}{n}$$
4. Testgröße berechnen:
$$t = \sqrt{n} * max|F_n(x) - F(x)|$$

Ablauf A.3 Welch's t-Test nach [25]

1. Berechne die Anzahl der Freiheitsgrade:
$$df = \frac{(\frac{\sigma_x^2}{n_x} + \frac{\sigma_y^2}{n_y})^2}{\frac{\sigma_x^2}{(n_x - 1) * n_x^2} + \frac{\sigma_y^2}{(n_y - 1) * n_y^2}}$$
2. Berechne:
$$t = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{\sigma_x^2}{n_x} + \sigma_y^2 n_y}}$$
3. Mit der Anzahl der Freiheitsgrade und dem t-Wert den p-Wert aus einer Tabelle auslesen.

Ablauf A.4 Spearman'scher Rangkorrelationskoeffizienten nach [25]

Für mehr als zehn Messwerte gilt:

1. Ordne jedem Messwert einen Rang zu.
   - Der größte Wert bekommt Rang 1, der kleinste Rang N.
   - Bei einem Unentschieden wird der Mittelwert berechnet und als Rang gegeben.
2. Berechne mit R = Rang und $\bar{R}$ = Durchschnittlicher Rang:

$$\rho = \frac{\sum_{k=1}^{n}(R(x_k) - \bar{R}(x))(R(y_k) - \bar{R}(y))}{\sqrt{\sum_{k=1}^{n}(R(x_k) - \bar{R}(x))^2 * \sum_{k=1}^{n}(R(y_k) - \bar{R}(y))^2}}$$

3. Berechne:

$$t = \frac{\sqrt{(n-2) * \rho^2}}{\sqrt{(1-\rho^2)}}$$

Tabellen

Tabelle A.1: CLBG-Programmiersprachen nach [4]

Programmiersprache
Ada
C, Chapel, C++, C#
Dart
Erlang
Fortran, F#
Go
Haskell
Java, JavaScript, Julia
Lisp, Lua
OCaml
Pascal, Perl, PHP, Python
Racket, Ruby, Rust
Smalltalk, Swift

Tabelle A.2: CLBG-Benchmarks nach [35]

Benchmark	Description
n-body	Double precision N-body simulation
fannkuch-redux	Indexed access to tiny integer sequence
spectral-norm	Eigenvalue using the power method
mandelbrot	Generate Mandelbrot set portable bitmap file
pidigits	Streaming arbitrary precision arithmetic
regex-redux	Match DNA 8mers and substitute magic patterns
fasta	Generate and write random DNA sequences
k-nucleotide	Hashtable update and k-nucleotide strings
reverse-complement	Read DNA sequences, write their reverse-complement
binary-trees	Allocate, traverse and deallocate many binary trees
chameneos-redux	Symmetrical thread rendezvous requests
meteor-contest	Search for solutions to shape packing puzzle
thread-ring	Switch from thread to thread passing one token

Tabelle A.3: p-Werte des Shapiro-Wilk-Tests für alle Java-Algorithmen im Variationstest

Softwaremetrik	**A-Stern**	**Iter. Binär**	**Rek. Binär**	**Heapsort**
Echtzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Kernelzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Userzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Auslastung	**0,057**	$< 0,001$	$< 0,001$	$< 0,001$
Max. RAM	$< 0,001$	**0,482**	**0,142**	0,04
Package-Watt	$< 0,001$	**0,147**	$< 0,001$	**0,135**
CPU-Watt	$< 0,001$	**0,170**	$< 0,001$	**0,143**
RAM-Watt	$< 0,001$	$< 0,001$	$< 0,001$	**0,263**
	Huffman	**Matrix**	**Quicksort**	**Token**
Echtzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Kernelzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Userzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Auslastung	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Max. RAM	$< 0,001$	$< 0,001$	**0,064**	0,002
Package-Watt	$< 0,001$	**0,1**	$< 0,001$	$< 0,001$
CPU-Watt	$< 0,001$	**0,121**	$< 0,001$	$< 0,001$
RAM-Watt	$< 0,001$	0,004	$< 0,001$	$< 0,001$

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05.

Tabelle A.4: p-Werte des Shapiro-Wilk-Tests für alle Kotlin-Algorithmen im Variationstest

Softwaremetrik	**A-Stern**	**Iter. Binär**	**Rek. Binär**	**Heapsort**
Echtzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Kernelzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Userzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Auslastung	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Max. RAM	**0,052**	**0,068**	**0,070**	**0,311**
Package-Watt	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Watt	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
RAM-Watt	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
	Huffman	**Matrix**	**Quicksort**	**Token**
Echtzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Kernelzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Userzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Auslastung	$< 0,001$	$< 0,001$	0,001	$< 0,001$
Max. RAM	**0,678**	0,001	**0,057**	0,019
Package-Watt	$< 0,001$	**0,11**	$< 0,001$	$< 0,001$
CPU-Watt	$< 0,001$	**0,104**	$< 0,001$	$< 0,001$
RAM-Watt	$< 0,001$	0,012	$< 0,001$	$< 0,001$

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05.

Tabelle A.5: p-Werte des Kolmogorow-Smirnow-Tests für alle Algorithmen im Variationstest

Softwaremetrik	**A-Stern**	**Iter. Binär**	**Rek. Binär**	**Heapsort**
Echtzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Kernelzeit	**0,155**	**1,000**	**1,000**	**0,155**
Userzeit	0,01	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Auslastung	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Max. RAM	$< 0,001$	-	-	$< 0,001$
Package-Watt	$< 0,001$	$< 0,001$	$< 0,001$	0,004
CPU-Watt	$< 0,001$	$< 0,001$	$< 0,001$	0,001
RAM-Watt	$< 0,001$	$< 0,001$	$< 0,001$	0,024
	Huffman	**Matrix**	**Quicksort**	**Token**
Echtzeit	$< 0,001$	$< 0,001$	$< 0,001$	0,01
Kernelzeit	**0,078**	0,024	**0,968**	**0,583**
Userzeit	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Auslastung	$< 0,001$	$< 0,001$	$< 0,001$	$< 0,001$
Max. RAM	$< 0,001$	$< 0,001$	-	$< 0,001$
Package-Watt	$< 0,001$	-	$< 0,001$	$< 0,001$
CPU-Watt	$< 0,001$	-	$< 0,001$	$< 0,001$
RAM-Watt	$< 0,001$	**0,47**	$< 0,001$	$< 0,001$

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05. Ein - markiert einen nicht ausgeführten Test.

Tabelle A.6: p-Werte des Welch's t-Tests für alle Algorithmen im Variationstest

Softwaremetrik	**A-Stern**	**Iter. Binär**	**Rek. Binär**	**Heapsort**
Echtzeit	-	-	-	-
Kernelzeit	0,003	**0,863**	**0,558**	0,007
Userzeit	-	-	-	-
CPU-Auslastung	-	-	-	-
Max. RAM	-	$< 0,001$	$< 0,001$	-
Package-Watt	-	-	-	-
CPU-Watt	-	-	-	-
RAM-Watt	-	-	-	-
	Huffman	**Matrix**	**Quicksort**	**Token**
Echtzeit	-	-	-	-
Kernelzeit	0,002	-	**0,384**	**0,105**
Userzeit	-	-	-	-
CPU-Auslastung	-	-	-	-
Max. RAM	-	-	$< 0,001$	-
Package-Watt	-	$< 0,001$	-	-
CPU-Watt	-	$< 0,001$	-	-
RAM-Watt	-	**0,871**	-	-

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05. Ein - markiert einen nicht ausgeführten Test.

Tabelle A.7: p-Werte des Spearman'schen Rangkorrelationskoeffizients für alle Algorithmen im Variationstest

Softwaremetrik	**A-Stern**	**Iter. Binär**	**Rek. Binär**	**Heapsort**
Echtzeit	**0,994**	**0,619**	**0,615**	**0,983**
Kernelzeit	**0,700**	**0,104**	**0,213**	**0,217**
Userzeit	**0,379**	**0,988**	**0,483**	**0,092**
CPU-Auslastung	**0,656**	**0,193**	**0,263**	**0,239**
Max. RAM	0,046	**0,283**	**0,116**	**0,563**
Package-Watt	**0,691**	**0,741**	**0,851**	**0,887**
CPU-Watt	**0,629**	**0,954**	**0,921**	**0,928**
RAM-Watt	**0,590**	**0,071**	**0,861**	**0,580**
	Huffman	**Matrix**	**Quicksort**	**Token**
Echtzeit	**0,377**	**0,865**	**0,243**	**0,713**
Kernelzeit	**0,236**	**0,386**	**0,114**	**0,225**
Userzeit	**0,520**	**0,618**	**0,391**	**0,126**
CPU-Auslastung	**0,939**	**0,472**	**0,472**	**0,260**
Max. RAM	**0,257**	**0,594**	**0,302**	**0,832**
Package-Watt	**0,355**	**0,227**	**0,206**	**0,482**
CPU-Watt	**0,396**	**0,217**	**0,147**	**0,278**
RAM-Watt	**0,170**	**0,566**	**0,652**	**0,498**

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05.

Tabelle A.8: p-Werte des Shapiro-Wilk-Tests für alle Java-Implementierungen im Kombinationstest

Softwaremetrik	**Dateien**	**Listen**	**(De-)Serialisierung**
Echtzeit	$< 0,001$	$< 0,001$	$< 0,001$
Kernelzeit	$< 0,001$	$< 0,001$	$< 0,001$
Userzeit	$< 0,001$	$< 0,001$	0,001
CPU-Auslastung	$< 0,001$	$< 0,001$	$< 0,001$
Max. RAM	0,008	0,004	0,009
Package-Watt	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Watt	$< 0,001$	$< 0,001$	$< 0,001$
RAM-Watt	$< 0,001$	$< 0,001$	0,016

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05.

Tabelle A.9: p-Werte des Shapiro-Wilk-Tests für alle Kotlin-Implementierungen im Kombinationstest

Softwaremetrik	**Dateien**	**Listen**	**(De-)Serialisierung**
Echtzeit	0,004	$< 0,001$	$< 0,001$
Kernelzeit	0,004	$< 0,001$	$< 0,001$
Userzeit	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Auslastung	$< 0,001$	$< 0,001$	0,049
Max. RAM	$< 0,001$	**0,222**	**0,209**
Package-Watt	$< 0,001$	$< 0,001$	$< 0,001$
CPU-Watt	$< 0,001$	$< 0,001$	$< 0,001$
RAM-Watt	$< 0,001$	$< 0,001$	0,028

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05.

Tabelle A.10: p-Werte des Kolmogorow-Smirnow-Tests für die Kombinationstests

Softwaremetrik	Dateien	Listen	(De-)Serialisierung
Echtzeit	**0,282**	$< 0,001$	$< 0,001$
Kernelzeit	0,036	**0,211**	**0,994**
Userzeit	**0,470**	$< 0,001$	$< 0,001$
CPU-Auslastung	**0,368**	$< 0,001$	$< 0,001$
Max. RAM	**0,368**	$< 0,001$	$< 0,001$
Package-Watt	**0,282**	$< 0,001$	**0,155**
CPU-Watt	**0,470**	$< 0,001$	**0,078**
RAM-Watt	**0,282**	$< 0,001$	$< 0,001$

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05.

Tabelle A.11: p-Werte des Welch's t-Tests für die Kombinationstests

Softwaremetrik	Dateien	Listen	(De-)Serialisierung
Echtzeit	0,012	-	-
Kernelzeit	-	0,011	**0,727**
Userzeit	0,049	-	-
CPU-Auslastung	**0,178**	-	-
Max. RAM	**0,576**	-	-
Package-Watt	**0,059**	-	0,027
CPU-Watt	**0,311**	-	0,012
RAM-Watt	**1,000**	-	-

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05. Ein - markiert einen nicht ausgeführten Test.

Tabelle A.12: p-Werte des Spearman'schen Rangkorrelationskoeffizients für die Kombinationstests

Softwaremetrik	Dateien	Listen	(De-)Serialisierung
Echtzeit	$< 0,001$	**0,690**	**0,605**
Kernelzeit	0,02	**0,061**	**0,396**
Userzeit	$< 0,001$	**0,378**	**0,170**
CPU-Auslastung	**0,133**	**0,427**	**0,266**
Max. RAM	**0,075**	**0,240**	**0,454**
Package-Watt	**0,995**	**0,414**	**0,172**
CPU-Watt	**0,826**	**0,432**	**0,242**
RAM-Watt	**0,742**	**0,884**	**0,335**

Hervorgehobene Werte entsprechen einer nicht abgelehnten Nullhypothese mit einem α von 0,05.

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit mit dem Thema:

Java vs. Kotlin: Fallstudie zur Energieeffizienz und Performance

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original