Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Projektarbeit Informatik

Best-Practices für performante Java-Programmierung (Software Engineering)

Autoren	Rémi Georgiou André Stocker
Hauptbetreuung	Dr. Mark Cieliebak Marek Arnold
Datum	20.12.2016

Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

.....

Unterschriften:

.....

.....

.....

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Projektarbeiten zu Beginn der Dokumentation nach dem Titelblatt mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Zusammenfassung

In dieser Arbeit wird die Effizienz von Daten- und Codestrukturen der Programmiersprache Java untersucht. Es wird auf Wissen und Erfahrungen, welche in zwei vorangegangenen Bachelorarbeiten gesammelt wurden, zurückgegriffen.

Eine qualitativ hochwertige Software soll korrekt, effizient und wartungsfreundlich sein. Unter dem Aspekt der Effizienz stellt man sich die Frage, welche Datenstrukturen und Programmierparadigmen die beste Wahl zur Problemlösung sind. Aufgrund des nicht deterministischen Verhaltens der Java Virtual Machine, ist das Testen von Benchmarks nicht trivial [1]. Die Performance von Java-Code hängt stark von den Optimierungen des Just-in-time-Compilers (JIT-Compiler) und des Garbage Collectors ab [2].

Für die Performancemessungen wird eine Testumgebung, welche in zwei früheren Bachelorarbeiten aufgebaut wurde, verwendet. Laufzeitmessungen zu einigen interessanten Fragestellungen werden mit Java Microbenchmark Harness (JMH) von OpenJDK durchgeführt. Die detaillierten, textbasierten Messergebnisse von JMH werden geparst und grafisch dargestellt. Es wird an frühere Experimente angeknüpft und deren Resultate kritisch hinterfragt. Dank neuem Wissen über die Java Virtual Machine und JMH werden neue Experimente getestet, mit dem Ziel eine Sammlung von Best-Practices für performante Java-Programmierung zu erstellen.

Verglichen mit der ArrayList war der Lesezugriff auf einzelne Elemente eines Arrays ein wenig schneller. Die funktionale for-each-Schleife war die optimale Wahl für Iterationen. Eine lokale Schleifenindexvariable ist immer noch die beste Lösung. Es konnte kein zusätzlicher Aufwand beim Aufruf von Zugriffsmethoden (Getter/Setter) festgestellt werden. Dieselbe Feststellung traf für das Experiment mit der Vererbungshierarchie zu.

Abstract

This paper investigates the efficiency of data and code structures of the Java programming language. It is based on the knowledge and experiences gained in two previous Bachelor's theses.

High-quality software should be bug-free, efficient and maintenance-friendly. Questions arise as to which data structures and programming paradigms are the best choice for solving a given problem. Due to the non-deterministic behavior of the Java Virtual Machine, evaluating the performance of java code is not straightforward [1]. The JVM makes a number of optimizations when it compiles the code that greatly affect the performance [2].

A test system which was developed during two previous Bachelor's theses is used for performance measurements. Runtime measurements are performed with Java Microbenchmark Harness (JMH) from OpenJDK. The detailed, text-based measurement results of JMH are parsed and converted to graphs. Individual existing experiments from a previous Bachelor's thesis are analyzed. Thanks to new knowledge about the Java Virtual Machine and JMH, new experiments are tested with the aim of creating a collection of best practices for efficient Java programming.

Compared to the ArrayList, the read access of elements in an array was a bit faster. The functional for-each loop was the optimal choice for iterations. A local loop index variable is still the best solution. No additional run time was measured when calling accessors (getter/setter). The same observation was true for the experiment with the class hierarchy.

Danksagung

Danken möchten wir in erster Linie Dr. Mark Cieliebak, welcher uns die Bearbeitung dieses interessanten Themas ermöglicht hat. Ebenfalls möchten wir unseren Dank an Marek Arnold aussprechen. Wir konnten von ihrer Expertise auf dem Gebiet der Java-Programmierung profitieren. Ihre konstruktive Kritik und wertvollen Hinweise während der Bearbeitungszeit dieser Arbeit schätzten wir sehr.

Vielen Dank!

Rémi Georgiou, André Stocker

Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage

In den vergangen zwei Jahren haben sich Studierende der Zürcher Hochschule für Angewandte Wissenschaften mit dem Thema Java-Effizienz befasst. Diese Arbeit knüpft an die Erkenntnisse und Resultate an, welche in den Bachelorarbeiten von Simbürger/Thöni [3] und De Tomasi/Rutz [4] gewonnen wurden. Einige frühere Experimente werden aufgegriffen und in etwas abgewandelter Form mit veränderten Parametern getestet.

Die Performance von Java-Programmen zu testen ist nicht trivial [1]. Java-Programme unterliegen zur Laufzeit verschiedenen nichtdeterministischen Einflüssen, wie z. B. dem Just-in-time-Compiler (JIT-Compiler), die Heap-Speichergrösse und der Garbage Collector (GC) [1]. Ausser diesen mit der Java Virtual Machine (JVM) im Zusammenhang stehenden Effekten, kommen noch Thread-Scheduling und Interrupts auf der Betriebssystemebene hinzu.

Für Laufzeitmessungen von Java-Code existieren spezialisierte Frameworks wie Caliper [5] oder das Java Microbenchmark Harness (JMH) [6] von OpenJDK. JMH wurde in den oben zitierten Bachelorarbeiten eingesetzt. Aufgrund des nichtdeterministischen Verhaltens der JIT-Kompilierung stellt das Messen von Java-Benchmarks eine Herausforderung dar. Aleksey Shipilëv [7], ein langjähriger Mitarbeiter von Sun/Oracle, Entwickler von JMH und Experte auf dem Gebiet der Java-Performancemessungen, findet die folgenden Worte dafür: „Benchmarking is the (endless) fight against the optimizations“ [8].

Die Verwendung des Benchmark-Harness JMH, welches auf Oracles Java Virtual Machines (HotSpot und OpenJDK) abgestimmt ist, hilft diese Schwierigkeiten zu umgehen. [8]. Ein entscheidender Vorteil ist, dass bei richtiger Konfiguration verlässliche und reproduzierbare Resultate erzeugt werden. Die offizielle Samplecode-Website des JMH-Projekts stellt eine Sammlung von Benchmark-Best-Practices und Dokumentation bereit. [9].

In der Literatur zum Thema Java-Performance findet man verschiedene Strategien zur Bewertung von Benchmarks. Es werden Mittelwert, Bestwert, Median oder der schlechteste Messwert für Performance-Analysen herangezogen [1].

1.2 Zielsetzung und Aufgabenstellung

Im Fokus dieser Projektarbeit liegt das Thema Effizienz von Java-Code. Wie mit unserem Betreuer Dr. Mark Cieliebak vereinbart, sollen objektive und nachvollziehbare Daten und Messungen aus strukturierten Laufzeit-Experimenten erhoben werden (siehe offizielle Aufgabenstellung in der Vereinbarung im Anhang).

Das oben erwähnte Performance-Messsystem, nachfolgend Testsystem genannt, wird eingesetzt und erweitert. JMH wird weiterhin für Laufzeitmessungen eingesetzt.

Um Missverständnisse und Verwechslungen zu vermeiden, werden einleitend die folgenden Begriffe definiert.

Begriff	**Bedeutung**
Experiment	Eine Sammlung von Benchmarks zu einem Thema
Benchmark	Testmethode, deren Laufzeit gemessen wird
Iteration	Eine Menge von Aufrufen (Invocations) desselben Benchmarks JMH berechnet daraus die (Durchschnitts-)Laufzeit einer Iteration
Invocation	Ein Aufruf eines Benchmarks

Tabelle 1: Wichtige Begriffe und deren Bedeutung in dieser Arbeit

Diese Arbeit befasst sich mit den folgenden sechs Themen:

Themen / Experimente
Zufällige Zugriffe auf den Datenstrukturen Array und ArrayList
Iterationsvarianten
Schleifenkopf mit lokaler Zählvariable oder Instanzvariable
Direktzugriff auf Instanzvariablen und Zugriff über Getter- und Setter-Methoden
Lese- und Schreibzugriff auf Instanzvariablen, deklariert als static/final/volatile
Overhead von Abstraktion und Vererbung

Tabelle 2: Themen / Experimente

Für jedes Experiment werden mehrere Benchmarks geschrieben, ausgeführt und untersucht. In JMH werden Benchmark-Methoden mit der Annotation `@Benchmark` versehen.

```
@Benchmark
public String helloWorldTest() {
      return "Hello, world!";
}
```

Listing 1: Beispiel eines JMH-Benchmarks

Im Kapitel 2 werden die Möglichkeiten zum Tuning der JVM anhand von einigen Beispielen für den JIT-Compiler und den Garbage Collector besprochen.

Das Kapitel 3 befasst sich mit dem Aufbau der Testumgebung und der Problematik der Laufzeitmessung von Java-Code. Darüber hinaus werden die verwendete JMH-Konfiguration des Testsystems und die Messverfahren beschrieben.

Die eigentlichen Messresultate der Benchmarks sind im Kapitel 4 tabellarisch und grafisch dargestellt.

Zu guter Letzt stellt das Kapitel 5 eine Diskussion und einen Ausblick bereit.

Für das Verständnis der hier behandelten Thematik wird vorausgesetzt, dass die Leserin oder der Leser über Grundkenntnisse der objektorientierten Programmierung und der Programmiersprache Java verfügt.

2 JVM-Einstellungen

In diesem Kapitel werden die Möglichkeiten, wie man die Performance der Java Virtual Machine beeinflussen kann, betrachtet. Die Ausführungszeit eines Java-Programmes hängt massgeblich von der JIT-Kompilierung ab [2]. Die Aktivität des Garbage Collectors hat ebenfalls einen gewissen Einfluss.

Das Ziel ist, bestmögliche Voraussetzungen für die Laufzeitmessungen zu schaffen.

2.1 JIT-Compiler

Das Testsystem setzt Oracles HotSpot Java Virtual Machine ein. Die Performance eines Java-Programms hängt davon ab, wie effizient der von der JIT-Kompilierung generierte architekturspezifische Assemblycode ist. Der Name HotSpot rührt daher, dass Quellcode einer Methode umso „heisser“ ist, je öfter er ausgeführt wird [2].

Der Just-in-time-Compiler lässt sich beim Starten der JVM mit dem Flag `–Xint` deaktivieren. Wenn diese Kompilieroption gesetzt ist, arbeitet die JVM im Interpreter-Only-Modus [10].

Die JVM unterscheidet zwischen zwei JIT-Compilern mit den Namen `server` und `client`. Der Client-Compiler beginnt früher mit der Bytecode-Kompilierung. Der Server-Compiler spielt seine Stärke bei längerer Programmausführung aus. Bevor mit der Kompilierung begonnen wird, analysiert der Server-Compiler den Bytecode und entscheidet über die beste Optimierung. Dies führt schlussendlich dazu, dass der Server-Compiler schnelleren Code produziert als der Client-Compiler [11].

Ein guter Kompromiss zwischen Client- und Server-Compiler lässt sich mit **Tiered Compilation** erreichen (`-XX:+TieredCompilation`). Dabei wird der Java-Bytecode zuerst vom Client-Compiler (C1) kompiliert und anschliessend vom Server-Compiler (C2) optimiert. In Java SE 8 ist Tiered Compilation standardmässig aktiviert [12].

Der JIT-Compiler übersetzt Java-Bytecode in eine Assemblersprache für die darunterliegende Maschinenarchitektur. Der Assemblycode wird im sogenannten Code-Cache gehalten. Dieser Code-Cache hat eine feste Grösse, und wenn er seine Kapazität erreicht, kann die JVM keinen neuen Assemblycode produzieren [13].

Der Code-Cache wird über zwei Flags gesteuert (Grössenangaben in Bytes):
`-XX:ReservedCodeCacheSize=`*`N`*
`-XX:InitialCodeCacheSize=`*`N`*

Um zu verhindern, dass die Code-Cache-Grösse während den Laufzeitmessungen vergrössert werden muss, setzt man beide Flags auf denselben Wert. Auf dem Computersystem steht ausreichend Arbeitsspeicher (16 Gigabytes) zur Verfügung, deshalb wird die Code-Cache-Grösse auf ein Gigabyte festgelegt.

2.2 Garbage Collection

Java fällt in die Kategorie der sogenannten „Managed Programming Languages“, was bedeutet, dass sich ein Softwareentwickler nicht um das Anfordern und Freigeben von Speicher kümmern muss, wie in anderen Hochsprachen (C, C++). Die JVM übernimmt diese komplexe Aufgabe automatisch bei Programmausführung [14]. Das Speichermanagementtool Garbage Collector (GC) ist zuständig für das Anfordern von Speicher für neue Objekte und die Speicherfreigabe von nicht mehr referenzierten Objekten [15].

Grundsätzlich entscheidet die JVM darüber, wann der GC aktiv wird. Dennoch gibt es einige interessante Konfigurationsmöglichkeiten. Oracles HotSpot Java Virtual Machine stellt vier Garbage-Collection-Algorithmen zur Verfügung:

- Serial Garbage Collector
  (standardmässig aktiviert für `client`-Compiler, siehe Absatz 2.1 JIT-Compiler)
- Throughput Collector
  (standardmässig aktiviert für `server`-Compiler, siehe ebenfalls Absatz 2.1)
- Concurrent Mark Sweep (CMS) Collector
- G1 Collector (garbage first)

Die Details der einzelnen GC-Algorithmen werden hier nicht behandelt. Es würde den Rahmen dieser Projektarbeit sprengen. Vielmehr wird auf die offizielle Dokumentation [15] und Fachliteratur [16] zum Thema hingewiesen. Einfach ausgedrückt teilt ein Garbage Collector den Heapspeicher in sogenannte Generationen (young und old) ein. Diese Unterteilung macht Sinn, weil in einem typischen Java-Programm viele Objekte eine sehr kurze Lebensdauer haben und nur ein Teil des Heapspeichers der JVM nach toten Objekten durchsucht werden muss. Wenn der GC die Young-Generation nach toten Objekten durchsucht, werden die überlebenden Objekte in andere Bereiche des Heapspeichers verschoben [17].

Eine Möglichkeit den GC zu beeinflussen, liegt im Festlegen der Heapspeichergrösse. Dies erreicht man mittels zwei Tuningflags.

Flag	**Beschreibung**
-Xms*N*	Legt die Startgrösse des Heapspeichers in Bytes fest
-Xmx*N*	Legt die maximale Grösse des Heapspeichers in Bytes fest

Tabelle 3: Tuningflags für die Heapspeichergrösse

In einem Experiment werden drei grosse Arrays mit je 500‘000 Integer-Objekten vor jeder Benchmarkiteration angelegt. Es besteht das Risiko, dass bei zu kleinem Heapspeicher die JVM den Garbage Collector oft anspringen lässt. Dies könnte sich negativ auf die Performance der Benchmarks auswirken. Um dies zu vermeiden, wird die Start- und Maximalgrösse des Heapspeichers der JVM auf acht Gigabytes festgesetzt.

Es kann keine eindeutige Aussage darüber getroffen werden, welcher Garbage Collector am geeignetsten ist oder welchen Einfluss die vier Garbage-Collection-Algorithmen auf die Experimente haben.

Scott Oaks ausgezeichnetes Buch mit dem Titel „Java Performance The Definitive Guide“ [16] enthält eine leicht verständliche Einführung zum Thema Garbage Collection. Nach eingehender Lektüre dieses Kapitels wurde entschieden, den Garbage First (G1) Collector zu verwenden. Folgende Gründe sprechen dafür:

- Der G1 Collector hat gegenüber dem Serial Garbage Collector den entscheidenden Vorteil, dass er die Old-Generation nach referenzlosen Objekten ohne Programmunterbrechung durchsucht.
- Der GC-Prozess wird in separate Threads ausgelagert und läuft somit parallel zu den Benchmark-Threads.
- Ausserdem skaliert der G1 Collector gegenüber dem CMS Collector besser bei grossem Heapspeicher [17].

Der G1 Collector wird beim Starten einer JVM-Instanz mit dem Flag `-XX:+UseG1GC` aktiviert.

3 Vorgehen

3.1 Grundsätzliches

In diesem Absatz wird erläutert, weshalb die Performance von Java-Code mit einer spezialisierten Lösung gemessen werden sollte. In naiver Weise könnte man Java-Quellcode zwischen zwei Zeitmarken einbetten. Javas `System`-Klasse bietet für diese Fälle eine Methode an: `System.nanoTime()` [18]. Dann bildet man die Differenz der beiden Zeitpunkte und erhält die verstrichene Zeit. Aber welche Aussagekraft hat dieser Messwert? Der folgende Test stellt die Problematik dar.

```
private void test() {
    long start = System.nanoTime();

    for (int i=0; i < BIG_NUMBER; i++)
        doHeavyWork();

    System.out.println(System.nanoTime() - start);
}
```

Listing 2: Beispiel eines Benchmarks zur Laufzeitmessung

Einer der Fallstricke liegt in der Verwendung der Methode `System.nanoTime()`. Ein Blick in die Java Standard Edition 8 API [18] gibt Aufschluss. Diese Methode garantiert zwar Nanosekundenpräzision, aber keine Nanosekundenauflösung.

Auf einem modernen Betriebssystem ist jeder Aufruf einer Betriebssystemfunktion (Systemcall) mit einer gewissen Latenz verbunden. Der Grund ist der folgende: GNU/Linux und Windows sind Multiprogramming-Betriebssysteme und halten zu jeder Zeit mehrere Programme (Prozesse) im Speicher. Prozesse konkurrieren untereinander um CPU-Zeit. Eine spezielle Betriebssystemkomponente, der Scheduler, entnimmt der Runqueue einen Prozess und teilt ihn der CPU zu. Der Scheduler entzieht dem Prozess die CPU, z. B. beim Eintreten eines Interrupts oder wenn sein Zeitquantum[1] aufgebraucht ist. Eine laufende Instruktion wird aber stets abgeschlossen. Dies bezeichnet man als Preemptive Scheduling [19].

Aleksey Shipilëv [7] trägt in seinem Vortrag über Java-Benchmarking [20] unter anderem die Problematik beim Messen mit `System.nanoTime()` vor. Er beschreibt darin ein Experiment, bei dem die **Latenz eines Aufrufs** von `System.nanoTime()` unter Linux, Solaris und Windows gemessen wird. Aufbauend auf diesen Informationen wird ein Messversuch gestartet.

```
@Benchmark
public long getNanoTime() {
    return System.nanoTime();
}
```

Listing 3: Benchmark von System.nanoTime()

[1] Das Quantum ist eine Zeitscheibe (engl. „time slice“) für einen Thread [31].

Dieser simple Benchmark wurde unter Java Version 1.8.0_111 auf der Java HotSpot 64 Bit Server Virtual Machine ausgeführt. Die JVM lief auf einem Computersystem mit Intel Core-i7 3770K Prozessor und Windows 7 Pro 64 Bit. Es stellt sich heraus, dass ein Aufruf von `System.nanoTime()` unter Windows 7 Pro 64 Bit im Mittel eine Latenz von 11.626 Nanosekunden hat.

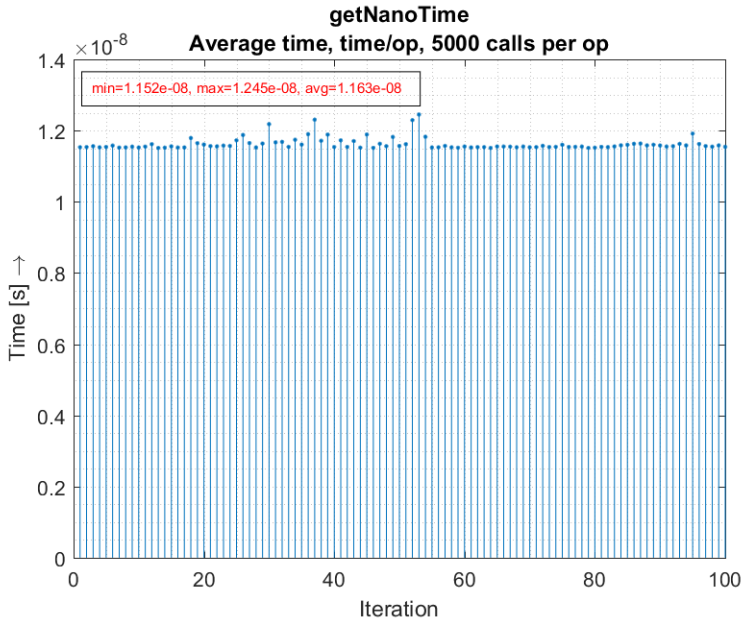


Abbildung 1: Mittlere Laufzeit von System.nanoTime()

Diese Erkenntnis soll als Input für einen weiteren Versuch herangezogen werden.

```
private static void testNanoTimeResolution() {
    long[] nanos = new long[ONE_SECOND];

    for (int i = 0; i < ONE_SECOND; i++)
        nanos[i] = System.nanoTime();

    int index = -1;
    while (nanos[++index] == nanos[0]) {}

    System.out.println("First change of nanoTime() value at index " + index);
}
```

Listing 4: System.nanoTime() Benchmark

Der oben aufgeführte Test soll beweisen, weshalb die **Nanosekundenauflösung** nicht garantiert ist. Die Testmethode speichert die Rückgabewerte von `System.nanoTime()` in einem Array ab und gibt den Index der ersten neuen Nanozeit aus.

Das folgende Listing zeigt die ersten 25 Werte des Arrays `nanos` aus oben erwähnter Testmethode `testNanoTimeResolution()`.

```
[ 0]  long  2633990407092
[ 1]  long  2633990407092
[ 2]  long  2633990407384   ┐
[ 3]  long  2633990407384   │
[ 4]  long  2633990407384   ├  Verletzt die Nanosekundenauflösung!
[ 5]  long  2633990407384   │
[ 6]  long  2633990407384   ┘
[ 7]  long  2633990407677
[ 8]  long  2633990407677
[ 9]  long  2633990407677
[10]  long  2633990407677
[11]  long  2633990407677
[12]  long  2633990407677
[13]  long  2633990407969
[14]  long  2633990407969
[15]  long  2633990407969
[16]  long  2633990407969
[17]  long  2633990407969
[18]  long  2633990407969
[19]  long  2633990408262
[20]  long  2633990408262
[21]  long  2633990408262
[22]  long  2633990408262
[23]  long  2633990408262
[24]  long  2633990408262
```

Listing 5: Gespeicherte Rückgabewerte von System.nanoTime() im Array nanos

Angenommen das Testsystem hätte eine ideale Latenz (< 1 Nanosekunde) und eine perfekte Nanosekundenauflösung, dann dürften zwei konsekutive Aufrufe niemals denselben Wert liefern.

Wie bereits weiter oben erwähnt, garantiert die Implementierung von `System.nanoTime()` zwar Nanosekundenpräzision, aber keine Nanosekundenauflösung. Das bedeutet im Zusammenhang mit der Java Virtual Machine und Multiprogramming-Betriebssystemen, dass **Laufzeiten unter zwölf Nanosekunden nicht gemessen** werden können. Die JVM würde das Betriebssystem lediglich mit `System.nanoTime()`-Aufrufen überfluten.

Diese Erkenntnis erweckt den Anschein, dass Laufzeitmessungen von Java-Benchmarks keine leichte Sache sind. An dieser Stelle wird deutlich darauf hinweisen, dass ein professionelles Benchmarktool wie JMH einer selbst entwickelten Lösung für Milli/Mikro/Nano-Benchmarks vorzuziehen ist.

3.2 Testumgebung

Hardwareplattform

Alle Benchmarks werden auf einem Computersystem mit den folgenden Spezifikationen ausgeführt:

Intel Xeon E3-1230 @ 3.3 GHz, 16 GB RAM, S-ATA III Harddisk

Diese CPU [21] verfügt über vier physikalische Cores und unterstützt Hyperthreading. Intel SpeedStep wurde deaktiviert, damit alle Cores mit der maximalen Schaltfrequenz betrieben werden.

Die Überlegung dabei ist die folgende: Die thermische Verlustleistung (Englisch: Thermal Design Power, TDP) einer CPU legt die Anforderungen an das Prozessorkühlsystem fest. Moderne Prozessoren fahren die Schaltfrequenz herunter, wenn sich ihre Temperatur der kritischen Zone nähert [22]. Solange die CPU während der Ausführung der Benchmarks ihren TDP-Wert nicht erreicht, soll das gesamte Leistungsbudget des Prozessors zur Verfügung stehen. Die CPU des Computersystems hat einen TDP-Wert von 80 Watt [21].

Betriebssystem und Java Virtual Machine (JVM)

Auf dem Computersystem ist die 64-Bit-Version von Debian 8.6 und Oracles HotSpot JVM installiert:

```
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)
```

Testsystem

Das Testsystem besteht aus drei Java-Maven-Projekten, welche als Teil der Bachelorarbeit von Marco De Tomasi und Markus Rutz [3] entstanden sind. Es wurde erweitert, um neue Benchmarks zu verwalten und spezifische Parser-Funktionalität bereitzustellen. JMH-Benchmarks produzieren viel Output, welcher standardmässig in der Konsole der integrierten Entwicklungsumgebung ausgegeben wird. Das Testsystem leitet den Outputstream in Textdateien um. Diese Dateien enthalten alle Messresultate und werden von einem neuen Parser eingelesen und aufs Wesentliche reduziert. Die kondensierte Information wird in Textdateien gespeichert. Ein MATLAB-Script liest diese Textdateien ein und erstellt daraus Stem-Plots und Histogramme im PNG-Format.

Das Testsystem verwendet die Version 1.12 des Java Microbenchmark Harness.

3.3 Messverfahren

Da ein bestehendes Testsystem verwendet wird, hat es den Vorteil, dass auf einer gewissen Expertise aufgebaut werden kann. Das führt aber dazu, dass man Systemkonfigurationen und Laufzeitmessungen mit einem frischen Auge betrachten und die erhaltenen Resultate kritisch hinterfragen kann.

Im Unterschied zu den vorangegangenen Experimenten aus früheren Bachelorarbeiten [3] [4], sind in dieser Arbeit nicht nur die durchschnittlichen Laufzeiten der Benchmarks von Interesse. Vielmehr soll versucht werden, mehr Information aus den einzelnen Messiterationen herauszuholen. Für Best-Practices-Empfehlungen werden Minimal- und Durchschnittslaufzeiten in Betracht gezogen.

Alle Benchmarks werden am Stück in derselben JVM-Instanz ausgeführt. Die Startup-Phase der JVM spielt für diese Experimente keine Rolle, weil die Tests während mehreren Stunden laufen. Sie wird deshalb vernachlässigt.

Beim Versuch die Zugriffszeit auf Elemente eines Arrays und einer ArrayList zu bestimmen, ist man auf Datenstrukturen mit zufälligen Daten angewiesen. Vor jeder Iteration werden neue Datenstrukturen instanziiert und mit pseudozufälligen `int`-Werten bestückt. Die Ausführungszeit dieser Setup-Phase ist für die Tests nicht von Interesse und würde die Messwerte verfälschen. Darum wird Setup-Code in separate Methoden ausgelagert, deren Laufzeiten nicht in die Messungen einfliessen. JMH stellt dafür die Annotation `@Setup` bereit.

```
@Setup(Level.Iteration)
public void setup() {
    arrayWithObjects = new Integer[SIZE];
    arrayListWithObjects = new ArrayList<>(SIZE);
    for (int i = 0; i < SIZE; i++) {
        int randomInt = RANDOM.nextInt(RANGE);
        arrayWithObjects[i] = randomInt;
        arrayListWithObjects.add(randomInt);
    }
    setupRandomIndexes();
}

private void setupRandomIndexes() {
    randomIndexes = new int[RANGE];
    for (int i = 0; i < RANGE; i++)
        randomIndexes[i] = RANDOM.nextInt(SIZE);
}
```

Listing 6: Setupmethode

Wenn man die Laufzeit eines Benchmarks mit JMH messen möchte, kann man zwischen zwei Betriebsmodi [23] wählen:

AverageTime	Bildet den Mittelwert von N Aufrufen innerhalb einer Iteration.
SingleShotTime	Angenommen die Arbeitslast (workload) eines Benchmarks hat keinen sogenannten festen Zustand (steady state), sondern sei variabel. Möchte man diesen Benchmark N-mal innerhalb einer Iteration ausführen und davon den Mittelwert bilden, so gibt das ein völlig falsches Bild wider, denn das Resultat der Laufzeitmessung hängt von verschiedenen Arbeitslasten ab. Stattdessen sollte man N Aufrufe des Benchmarks zu einem Batch zusammenfassen und die Gesamtlaufzeit des Batches messen [23].

Tabelle 4: Gegenüberstellung der Benchmark-Modi AverageTime und SingleShotTime

Wie aus der offiziellen JMH-Dokumentation [23] ersichtlich wird, macht es nur Sinn den SingleShotTime-Benchmarkmodus in Benchmarks zu verwenden, die keinen festen Zustand haben. Diese Projektarbeit untersucht Benchmarks, welche Datenstrukturen mit fester Länge verwenden, aber mit pseudozufälligen Werten bestückt werden. Es soll herausgefunden werden, ob es einen Unterschied macht, welcher Benchmark-Modus verwendet wird.

Es besteht die Möglichkeit, die Aktivität des Just-In-Time-Compilers zu inspizieren. Das Setzen des Compiler-Flags `-XX:+PrintCompilation` veranlasst die JVM zur Ausgabe des Kompilierprozesses.

```
    927  249       3       java.lang.String::getChars (16 bytes)
    932  225 %     3       sun.nio.cs.UTF_8$Decoder::decodeArrayLoop @ -2 (691 bytes)   made not entrant
    953  250       4       java.lang.String::startsWith (72 bytes)
    954   22       3       java.lang.String::startsWith (72 bytes)   made not entrant
    954  251       3       java.lang.Class::searchFields (41 bytes)
    954  252     n 0       java.lang.String::intern (native)
    955  253     n 0       sun.reflect.Reflection::getCallerClass (native)   (static)
    955  255     n 0       java.lang.Throwable::fillInStackTrace (native)
    955  256       3       java.lang.Class::checkInitted (19 bytes)
    955  254  s    3       java.lang.Throwable::fillInStackTrace (29 bytes)
    956  257       3       org.openjdk.jmh.util.Optional::orElse (16 bytes)
    956  258  s    3       sun.misc.PerfCounter::add (18 bytes)
    957  259  s    3       sun.misc.PerfCounter::get (9 bytes)
    961  260       3       java.lang.Class::checkMemberAccess (49 bytes)
    962  261       3       java.lang.Character::toUpperCase (6 bytes)
    962  262       3       java.lang.Character::toUpperCase (9 bytes)
    962  263       3       java.lang.CharacterDataLatin1::toUpperCase (53 bytes)
    962  264       3       java.nio.DirectLongBufferU::put (18 bytes)
    963  265       3       sun.net.www.ParseUtil::encodePath (336 bytes)
    964  266     n 0       java.lang.ClassLoader::findLoadedClass0 (native)
    964  267       3       sun.misc.URLClassPath::getResource (83 bytes)
    965  268   !   3       java.util.zip.InflaterInputStream::read (138 bytes)
    965  270       4       java.util.concurrent.ConcurrentHashMap::setTabAt (19 bytes)
    965  105       3       java.util.concurrent.ConcurrentHashMap::setTabAt (19 bytes)   made not entrant
    966  271   !   3       java.lang.ref.ReferenceQueue::poll (28 bytes)
    966  269       3       java.util.zip.InflaterInputStream::ensureOpen (18 bytes)
    966  273     n 0       java.util.zip.Inflater::inflateBytes (native)
    966  272       3       java.util.TreeMap::parentOf (13 bytes)
    967  274  s    3       java.lang.StringBuffer::append (13 bytes)
    967  275       3       java.lang.ref.Finalizer::register (10 bytes)
    967  276       3       java.lang.ref.Finalizer::<init> (23 bytes)
    975  277       3       java.io.InputStream::<init> (5 bytes)
    975  278       4       sun.nio.cs.UTF_8$Decoder::decodeArrayLoop (691 bytes)
    975  279       3       java.lang.String::compareTo (78 bytes)
```

Listing 7: Auszug aus dem JIT-Kompilierprozess

Von Interesse sind die Zahlen in der fünften Kolonne. Sie geben den **Level der Tiered Compilation** an und sind ein wichtiger Hinweis auf die Laufzeit-Performance einer Bytecode-Instruktion.

Tier Level	Erklärung
0	Interpretierter Code
1	Simple C1 compiled code
2	Limited C1 compiled code
3	Full C1 compiled code
4	C2 compiled code

Tabelle 5: Tiered Compilation Levels

Wie bereits im Abschnitt 2.1 erklärt, setzt Tiered Compilation zwei Kompiliermodi ein. Jede Bytecode-Instruktion startet zunächst bei Level 0. Der Client-Compiler übersetzt den Java-Bytecode, und wenn dieser oft ausgeführt wird, erreicht er schlussendlich den Tier Level 4 und wird vom Server-Compiler (C2) optimiert [24]. Im Absatz 3.4 wird beschrieben, was das für Konsequenzen hat für die Laufzeitmessungen.

Um verlässliche Aussagen über Effizienz und Best-Practices zu treffen, werden **Minimum- und Durchschnittslaufzeiten** von Benchmarks analysiert.

3.4 JMH-Konfiguration

Bevor man Laufzeitmessungen für Benchmarks durchführen kann, müssen einige Einstellungen des Java Microbenchmark Harness im Testsystem vorgenommen werden. JMH wird über Optionen gesteuert.

Wie bereits erwähnt, benötigt die JIT-Kompilierung etwas Zeit, um optimierten Assemblycode zu generieren. Es muss dafür gesorgt werden, dass Benchmarks vor dem Messvorgang einige Runden (Warmup-Iterationen) warmlaufen. Damit wird der JVM die Chance gegeben, den Bytecode auf Stufe Tiered Compilation Level 4 zu kompilieren.

Mit konkreten Messversuchen soll die erforderliche Anzahl Warmup-Iterationen bestimmt werden. Zu diesem Zweck werden zwei Benchmarks getestet, welche jeweils 5000-mal pro Iteration aufgerufen werden. Die Anzahl durchzuführender Iterationen sind 20, 50, 100, 150 und 200 pro Benchmark. Die Anzahl **Warmup- und Messiterationen** sind pro Test **identisch**.

Die Resultate der Iterationsanalyse sind auf den nachfolgenden Seiten dargestellt.

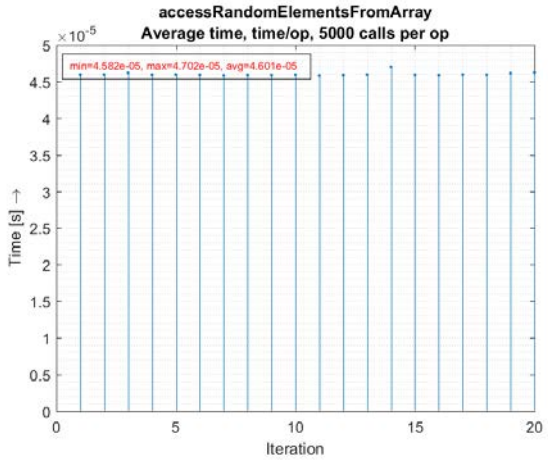


Abbildung 2: 20_AnalyseIteratioaccessRandomElementsFromArray.png

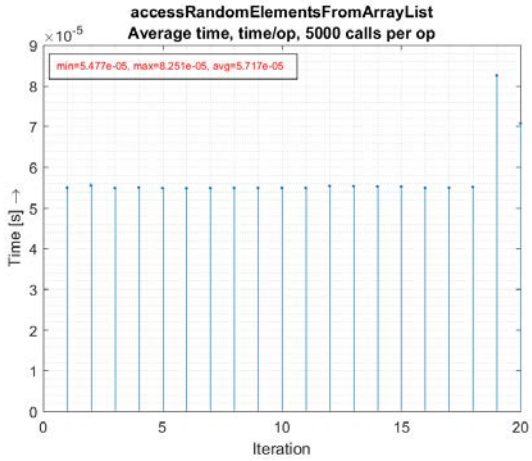


Abbildung 3: 20_AnalyseIteratioaccessRandomElementsFromArrayList.png

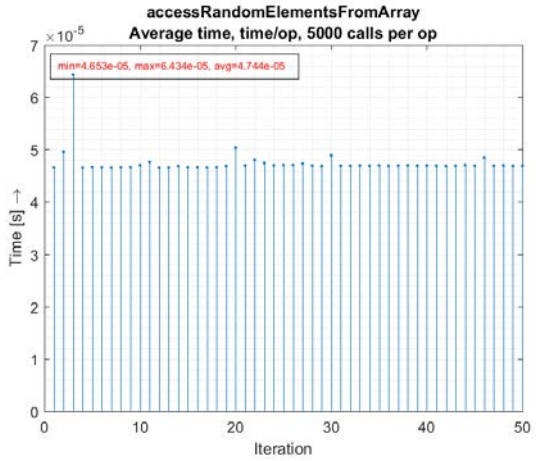


Abbildung 4: 50_AnalyseIteratioaccessRandomElementsFromArray.png

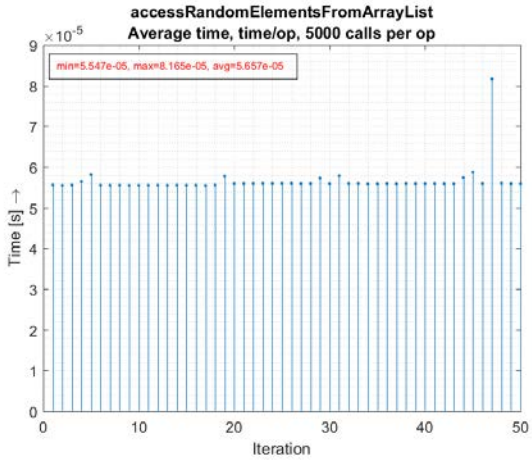


Abbildung 5: 50_AnalyseIteratioaccessRandomElementsFromArrayList.png

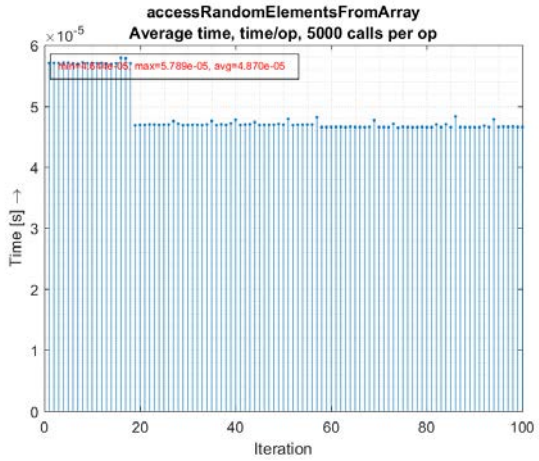


Abbildung 6: 100_AnalyseIteratiaccessRandomElementsFromArra.png

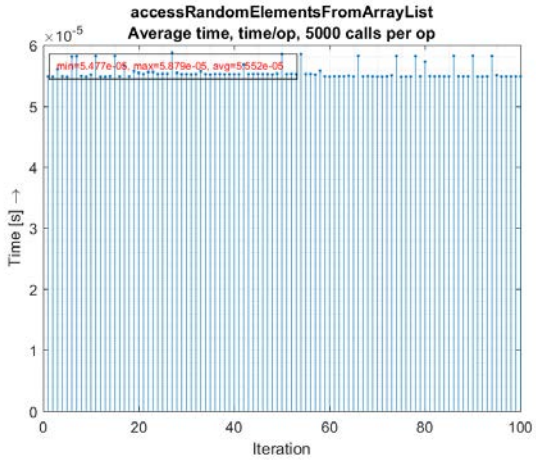


Abbildung 7: 100_AnalyseIteratiaccessRandomElementsFromArrayList.png

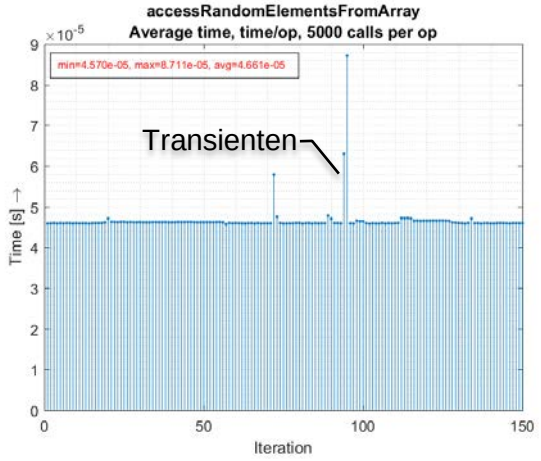


Abbildung 8: 150_AnalyseIteratiaccessRandomElementsFromArray.png

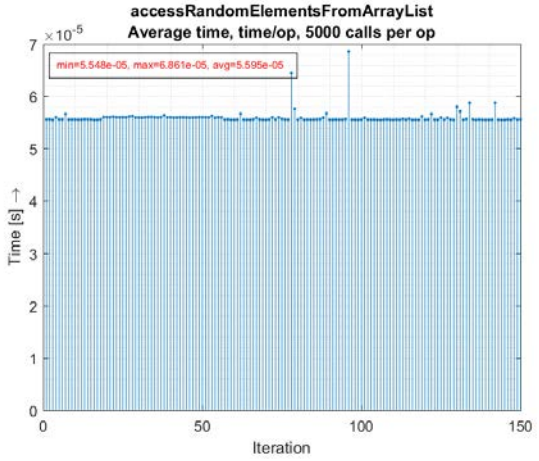


Abbildung 9: 150_AnalyseIteratiaccessRandomElementsFromArrayList.png

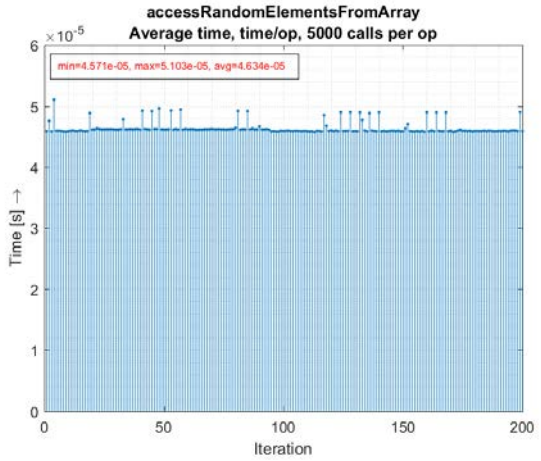


Abbildung 10: 200_AnalyseIteratiaccessRandomElementsFromArray.png

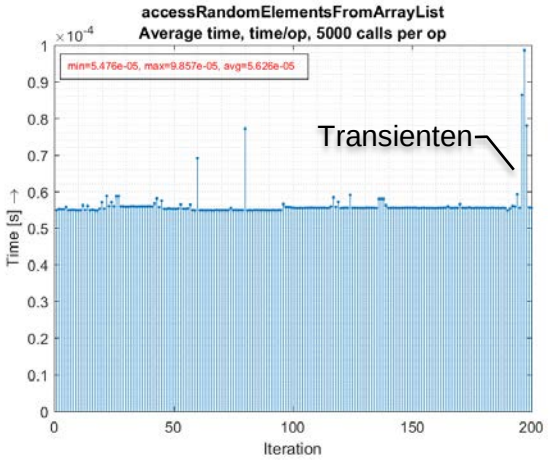


Abbildung 11: 200_AnalyseIteratiaccessRandomElementsFromArrayList.png

Benchmark	Iterationen	Durchschnittslaufzeit in Nanosekunden
accessRandomElementsFromArray	20	46006
accessRandomElementsFromArrayList	20	57170
accessRandomElementsFromArray	50	47439
accessRandomElementsFromArrayList	50	56570
accessRandomElementsFromArray	100	48703
accessRandomElementsFromArrayList	100	55524
accessRandomElementsFromArray	150	46605
accessRandomElementsFromArrayList	150	55945
accessRandomElementsFromArray	200	46342
accessRandomElementsFromArrayList	200	56262

Tabelle 6: Iterationen und ihre durchschnittlichen Laufzeiten des Experiments ArrayVsArrayListAccessTest

Die durchschnittlichen Laufzeiten eines Tests zeigen nur geringfügige Schwankungen im Bereich ± 2 ms. Eine grössere Anzahl Warmup-Iterationen resultiert nicht in einer viel kürzeren Benchmarklaufzeit.

Unabhängig von der Anzahl Iterationen (zur Erinnerung: die Anzahl Warmup- und Messiterationen sind gleich) kommt es immer wieder vor, dass die Ausführungszeit einzelner Iterationen (Ausreisser, Transienten) auffallend über dem Durchschnittswert liegen. Das überrascht nicht, weil die Prozesse der JVM nicht alleine auf der CPU rechnen. Der Linux-Scheduler teilt die begrenzten Rechnerressourcen (CPU) allen Prozessen (oder Threads) fair zu [19]. Die Garbage-Collection der Young-Generation führt ebenfalls zu kurzen Unterbrechungen [17] der JVM-Threads. Diese Unterbrechungen können prinzipiell zu jedem Zeitpunkt stattfinden (auch inmitten einer Messiteration) und lassen sich nicht vermeiden. Die Ausführungszeit solcher Transienten wird also in allen Benchmarks mitgemessen.

Man kann somit zwei Punkte aus dieser Erfahrung mitnehmen:

1) Nach bereits 20 Warmup-Iterationen hat der JIT-Compiler den Code optimiert (Tiered Compilation Level 4)

2) Um Transienten (verfälschte Messiterationen) besser glätten zu können, macht es Sinn 200 Messiterationen für jeden Benchmark durchzuführen. Mit zunehmender Anzahl Messiterationen sollte man kürzere Durchschnittslaufzeiten erreichen. (Ähnlich dem Erwartungswert einer Zufallsvariablen in einem Stichprobenexperiment)

Entsprechend werden die Parameter (im JMH-Jargon Optionen genannt) für JMH festgelegt:

JMH-Option	**Wert**
warmupIterations	20
warmupBatchSize	5000
measurementIterations	200
measurementBatchSize	5000
operationsPerInvocation	5000

Tabelle 7: JMH-Optionen

Diese Einstellungen gelten global für alle Benchmarks.

3.5 Schleifen in Benchmarks

Es kommt vor, dass Schleifen in Benchmarks benötigt werden, um eine Operation auf allen Elementen einer Datenstruktur anzuwenden. Die folgende zwei Benchmarks sollen als Beispiel dienen.

```
@Benchmark
public Integer[] accessRandomElementsFromArray_measureWrong() {
    Integer[] result = new Integer[10000];
    for (int i = 0; i < 10000; i++) {
        result[i] = arrayWithObjects[randomIndexes[i]];
    }
    return result;
}

@Benchmark
public Integer[] accessRandomElementsFromArrayList_measureWrong() {
    Integer[] result = new Integer[10000];
    for (int i = 0; i < 10000; i++) {
        result[i] = arrayListWithObjects.get(randomIndexes[i]);
    }
    return result;
}
```

Listing 8: Falsch messen bei Operationen innerhalb der Schleife

Beim Verwenden von Schleifen in Java-Benchmarks ist äusserste Vorsicht geboten. Das obige Listing zeigt, dass falsche Annahmen über den Zugriff der Elemente aus den Datenstrukturen gemacht wurden. In diesem konkreten Fall liegt das Problem in der Speicherung von Elementen in einem lokalen Objektarray (`Integer[] result`). Die Werte in `randomIndexes` ändern sich nicht innerhalb einer Iteration und pro Iteration wird der Benchmark 5000-mal aufgerufen (siehe Absatz 3.4). Der JIT-Compiler ist sehr clever und wird Optimierungen vornehmen. Die Konsequenz daraus sind verfälschte Laufzeitmessungen [25].

Das folgende Listing zeigt die korrekte Handhabung von Schleifen in Benchmarks.

```
@Benchmark
public void accessRandomElementsFromArray(Blackhole b) {
    for (int i = 0; i < RANGE; i++)
        b.consume(arrayWithObjects[randomIndexes[i]]);
}

@Benchmark
public void accessRandomElementsFromArrayList(Blackhole b) {
    for (int i = 0; i < RANGE; i++)
        b.consume(arrayListWithObjects.get(randomIndexes[i]));
}
```

Listing 9: Richtig messen beim sicheren Durchlaufen von Schleifen

Jedes Array-Element wird vom `Blackhole`-Objekt konsumiert. Man kann der Methode `consume()` des `Blackhole`-Objekts primitive Datentypen und Objekte übergeben. Dadurch wird garantiert, dass jeder Schleifendurchgang vollständig gerechnet wird [25].

Die ersten Messresultate der Benchmarks des Experiments „Zugriffe auf Array und ArrayList“ waren nicht überzeugend. Sie deuteten auf eine erheblich bessere Performance des Arrays gegenüber der ArrayList hin. Der Benchmark-Code wurde umgeschrieben (vgl. Listing 9) und erneut getestet. Die nachfolgenden Grafiken wurden aus den erhaltenen Messresultaten generiert und illustrieren den Sachverhalt.

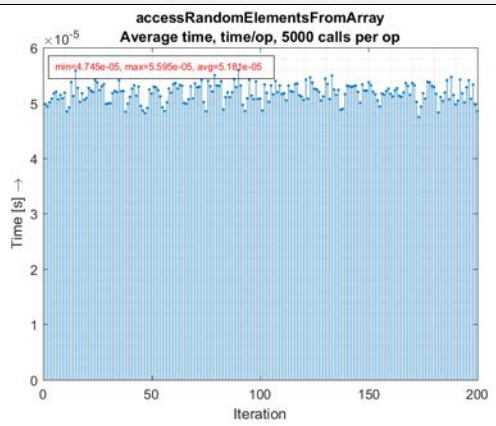
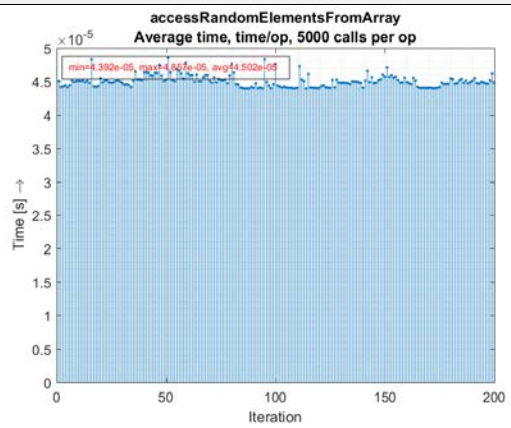
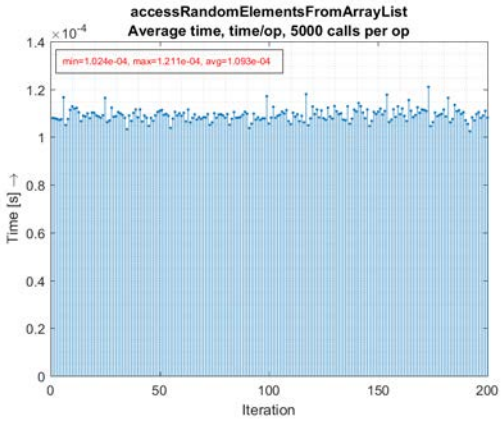
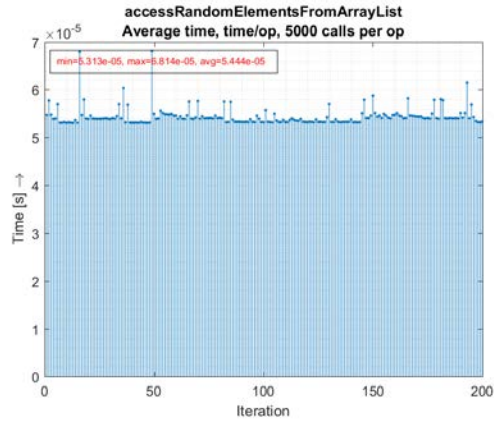
	Falsche Messungen	Richtige Messungen
Array	 *Abbildung 12: 2016_11_12-203453_accessRandomElementsFromArray_iterations.png*	 *Abbildung 13: 2016_11_22-212504_accessRandomElementsFromArray_iterations.png*
ArrayList	 *Abbildung 14: 2016_11_12-203453_accessRandomElementsFromArrayList_iterations.png*	 *Abbildung 15: 2016_11_22-212504_accessRandomElementsFromArrayList_iterations.png*

Tabelle 8: Falsches und richtiges Messen des Experiments „Zugriffe auf Array und ArrayList“

Bei den Messresultaten in der linken Spalte scheint es so, als ob die Zugriffe auf pseudozufällige Elemente der ArrayList im Durchschnitt 2.1-mal schneller sind als beim Array.
$\left(\frac{\mathbf{0.0001093\ s}}{\mathbf{0.00005181\ s}} = \mathbf{2.10963}\right)$

Dieses Ergebnis verblüfft und deckt sich nicht mit dem Eindruck, den man beim Einsatz von ArrayList hat. Gerade deshalb nicht, weil die OpenJDK-Implementation der ArrayList ein Array als interne Datenstruktur verwendet [26]. Die richtigen Messungen in der rechten Spalte zeigen, dass die Zugriffe auf das Array im Mittel etwa 17 % schneller sind als bei der ArrayList.
$\left(\frac{\mathbf{0.00004502\ s}}{\mathbf{0.00005444\ s}} = \mathbf{0.826965}\right)$

Das `Blackhole`-Objekt ist die Lösung!

Wie bereits bei der Methode `System.nanoTime()` kann man sich fragen, welche Latenz ein Aufruf von `Blackhole.consume()` verursacht. Es wird mit 20 Warmup- und 200 Messiterationen die Latenz von einem Aufruf, drei und zehn Aufrufen bestimmt.

AverageTime					
	min	**delta**	**avg**	**delta**	**stdev**
2016_11_25-122849_IteratorForEachLoopTest					
consumeOneStudentFromListTest	3.71E-09	**0.00%**	3.88E-09	**0.00%**	5.17E-10
consume3StudentsFromListTest	1.01E-08	**63.21%**	1.01E-08	**61.62%**	1.96E-10
consume10StudentsFromListTest	2.96E-08	**87.45%**	2.99E-08	**87.03%**	6.81E-10
2016_11_25-220816_IteratorForEachLoopTest					
consumeOneStudentFromListTest	3.84E-09	**0.00%**	3.85E-09	**0.00%**	9.39E-11
consume3StudentsFromListTest	1.01E-08	**62.18%**	1.04E-08	**62.80%**	5.20E-10
consume10StudentsFromListTest	3.02E-08	**87.28%**	3.08E-08	**87.48%**	2.37E-09
2016_11_29-183505_IteratorForEachLoopTest					
consumeOneStudentFromListTest	3.83E-09	**0.00%**	3.84E-09	**0.00%**	6.03E-11
consume3StudentsFromListTest	1.01E-08	**62.19%**	1.02E-08	**62.21%**	1.57E-10
consume10StudentsFromListTest	3.02E-08	**87.28%**	3.02E-08	**87.29%**	5.59E-10
2016_11_30-092512_IteratorForEachLoopTest					
consumeOneStudentFromListTest	3.84E-09	**0.00%**	3.84E-09	**0.00%**	3.60E-11
consume3StudentsFromListTest	1.01E-08	**62.18%**	1.02E-08	**62.21%**	2.11E-10
consume10StudentsFromListTest	3.02E-08	**87.28%**	3.02E-08	**87.26%**	1.11E-10
2016_12_03-112710_IteratorForEachLoopTest					
consumeOneStudentFromListTest	3.64E-09	**0.00%**	3.64E-09	**0.00%**	8.03E-11
consume3StudentsFromListTest	9.70E-09	**62.50%**	9.76E-09	**62.66%**	4.88E-10
consume10StudentsFromListTest	3.11E-08	**88.30%**	3.11E-08	**88.29%**	2.84E-10
2016_12_04-114839_IteratorForEachLoopTest					
consumeOneStudentFromListTest	3.50E-07	**0.00%**	3.60E-07	**0.00%**	2.49E-09
consume3StudentsFromListTest	9.50E-07	**63.14%**	9.57E-07	**62.41%**	2.76E-09
consume10StudentsFromListTest	3.01E-06	**88.37%**	3.04E-06	**88.18%**	1.09E-08

Listing 10: Latenz von Blackhole.consume()

Wie unschwer zu erkennen ist, entwickeln sich die Laufzeiten nicht perfekt linear zueinander.

Aus der Analyse von Performance-Modellen von Aleksey Shipilëv [20] geht hervor, dass die Laufzeit eines Gesamtsystems im Allgemeinen **keine lineare Kombination** aus den Laufzeitmessungen der enthaltenen Untersysteme ist („*performance is not usually composable*“ [20]).

3.6 Benchmarks

Die nachfolgende Tabelle stellt eine Übersicht aller durchgeführten Benchmarks dar.

Absatz	Experiment	Benchmarks
4.1	Zugriffe auf Array und ArrayList	accessRandomElementsFromArray accessRandomElementsFromArrayList
4.2	Iterationsvarianten	iteratorStudentListTest forEachStudentListTest streamStudentListTest iteratorStudentSetTest forEachStudentSetTest streamStudentSetTest
4.3	Schleifenkopf mit lokaler Zählvariable oder Instanzvariable	instanceVariableLoopCounter localVariableLoopCounter staticInstanceVariableLoopCounter
4.4	Direktzugriff auf Instanzvariablen vs. Getter- und Setter-Methoden	getValueViaGetterMethodeTest getValueViaInstanceVariableTest setValueViaSetterMethodeTest setValueViaInstanceVariableTest
4.5	Direktzugriff auf Instanzvariablen vs. Getter- und Setter-Methoden	getValue getStaticValue getFinalValue getStaticFinalValue getVolatileValue writeValue writeValueStatic writeValueVolatile
4.6	Overhead von Abstraktion und Vererbung	doInLineTestAbstraction doInLineTestNoAbstraction doNotInLineTestAbstraction doNotInLineTestNoAbstraction testAbstraction testNoAbstraction
4.7	JIT-Performance	

Tabelle 9: Übersicht aller Experimente und deren Benchmarks

4 Resultate

4.1 Zugriffe auf Array und ArrayList

4.1.1 Fragestellung

Ein Array und eine ArrayList werden vor jeder Iteration instanziiert und mit zufälligen Integer-Werten bestückt. Ist das Array beim Zugriff auf einzelne Elemente schneller als die ArrayList?

```
@Setup(Level.Iteration)
public void setup() {
    arrayWithObjects = new Integer[SIZE];
    arrayListWithObjects = new ArrayList<>(SIZE);
    for (int i = 0; i < SIZE; i++) {
        int randomInt = RANDOM.nextInt(RANGE);
        arrayWithObjects[i] = randomInt;
        arrayListWithObjects.add(randomInt);
    }
    setupRandomIndexes();
}

private void setupRandomIndexes() {
    randomIndexes = new int[RANGE];
    for (int i = 0; i < RANGE; i++)
        randomIndexes[i] = RANDOM.nextInt(SIZE);
}

@Benchmark
public void accessRandomElementsFromArray(Blackhole b) {
    for (int i = 0; i < RANGE; i++)
        b.consume(arrayWithObjects[randomIndexes[i]]);
}

@Benchmark
public void accessRandomElementsFromArrayList(Blackhole b) {
    for (int i = 0; i < RANGE; i++)
        b.consume(arrayListWithObjects.get(randomIndexes[i]));
}
```

Listing 11: Benchmarks Array und ArrayList

4.1.2 Vermutung (Hypothese)

Eine ArrayList speichert die Elemente intern in einem Array ab [26]. Beim Zugriff auf ein Element aus der ArrayList wird ein zusätzlicher Aufruf der Methode `get()` ausgeführt. Der Bytecodes der beiden Benchmarkmethoden bestätigt dies. Der nachfolgende Bytecode wurde mit dem Java Class File Disassembler (`javap`) aus dem Class-File `ArrayVsArrayListAccessTest.class` extrahiert [27].

```
public void accessRandomElementsFromArray(org.openjdk.jmh.infra.Blackhole);
  Code:
     0: iconst_0
     1: istore_2
     2: iload_2
     3: sipush        10000
     6: if_icmpge     30
     9: aload_1
    10: aload_0
    11: getfield      #5          // Field arrayWithObjects
    14: aload_0
    15: getfield      #14         // Field randomIndexes
    18: iload_2
    19: iaload
    20: aaload
    21: invokevirtual #15         // Method org/openjdk/jmh/infra/Blackhole.consume
    24: iinc          2, 1
    27: goto          2
    30: return

public void accessRandomElementsFromArrayList(org.openjdk.jmh.infra.Blackhole);
  Code:
     0: iconst_0
     1: istore_2
     2: iload_2
     3: sipush        10000
     6: if_icmpge     34
     9: aload_1
    10: aload_0
    11: getfield      #8          // Field arrayListWithObjects
    14: aload_0
    15: getfield      #14         // Field randomIndexes
    18: iload_2
    19: iaload
    20: invokeinterface #16,  2   // InterfaceMethod java/util/List.get
    25: invokevirtual #15         // Method org/openjdk/jmh/infra/Blackhole.consume
    28: iinc          2, 1
    31: goto          2
    34: return
```

Listing 12: Bytecode-Auszug aus der kompilierten Klasse ArrayVsArrayListAccess.class

Die Bytecode-Instruktionen aus den zwei Testmethoden sind identisch, bis auf die zusätzliche Instruktion `invokeinterface` im Benchmark `accessRandomElementsFromArrayList`.

Es wird vermutet, dass die Lesezugriffe auf das Array schneller sind als bei der ArrayList.

4.1.3 Beobachtung der Messwerte

AverageTime					
	min	**delta**	**avg**	**delta**	**stdev**
2016_11_22-212504_ArrayVsArrayListAccessTest					
accessRandomElementsFromArray	4.39E-05	**0.00%**	4.50E-05	**0.00%**	9.17E-07
accessRandomElementsFromArrayList	5.31E-05	**17.33%**	5.44E-05	**17.30%**	1.92E-06
2016_11_25-122849_ArrayVsArrayListAccessTest					
accessRandomElementsFromArray	4.41E-05	**0.00%**	4.51E-05	**0.00%**	6.19E-07
accessRandomElementsFromArrayList	5.31E-05	**16.98%**	5.50E-05	**18.04%**	4.31E-06
2016_11_25-220816_ArrayVsArrayListAccessTest					
accessRandomElementsFromArray	4.58E-05	**0.00%**	4.65E-05	**0.00%**	1.33E-06
accessRandomElementsFromArrayList	5.48E-05	**16.45%**	5.58E-05	**16.54%**	2.72E-06
2016_11_29-183505_ArrayVsArrayListAccessTest					
accessRandomElementsFromArray	4.76E-05	**0.00%**	4.78E-05	**0.00%**	1.96E-07
accessRandomElementsFromArrayList	5.55E-05	**14.24%**	5.60E-05	**14.59%**	9.65E-07
2016_11_30-092512_ArrayVsArrayListAccessTest					
accessRandomElementsFromArray	4.58E-05	**0.00%**	4.72E-05	**0.00%**	1.42E-06
accessRandomElementsFromArrayList	5.55E-05	**17.49%**	5.62E-05	**15.88%**	1.15E-06
2016_12_03-112710_ArrayVsArrayListAccessTest					
accessRandomElementsFromArray	4.57E-05	**0.00%**	4.61E-05	**0.00%**	4.46E-07
accessRandomElementsFromArrayList	5.54E-05	**17.41%**	6.76E-05	**31.80%**	1.24E-05

Listing 13: Auswertung der Testläufe von ArrayVsArrayListAccessTest im AverageTime-Modus

SingleShotTime					
	min	**delta**	**avg**	**delta**	**stdev**
2016_11_23-101221_ArrayVsArrayListAccessTest					
accessRandomElementsFromArray	2.39E-01	**0.00%**	2.45E-01	**0.00%**	1.47E-02
accessRandomElementsFromArrayList	2.70E-01	**11.73%**	2.83E-01	**13.40%**	1.70E-02
2016_11_25-184721_ArrayVsArrayListAccessTest					
accessRandomElementsFromArray	2.02E-01	**0.00%**	2.45E-01	**0.00%**	1.39E-02
accessRandomElementsFromArrayList	2.39E-01	**15.21%**	2.82E-01	**13.07%**	2.69E-02
2016_11_25-210309_ArrayVsArrayListAccessTest					
accessRandomElementsFromArray	2.13E-01	**0.00%**	2.41E-01	**0.00%**	3.88E-03
accessRandomElementsFromArrayList	2.59E-01	**17.71%**	2.80E-01	**13.87%**	6.24E-03

Listing 14: Auswertung der Testläufe von ArrayVsArrayListAccessTest im SingleShotTime-Modus

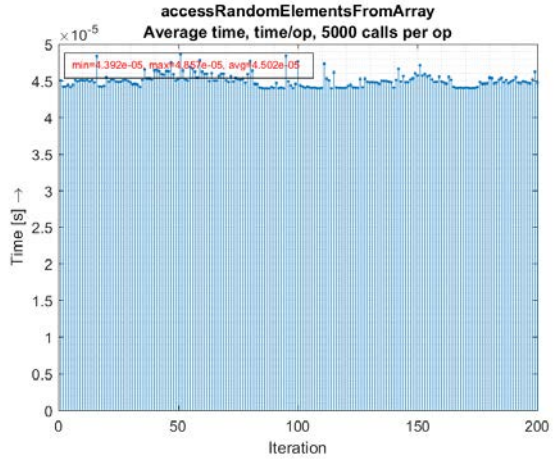


Abbildung 16: 2016_11_22-212504_accessRandomElementsFromArray_iterations.png

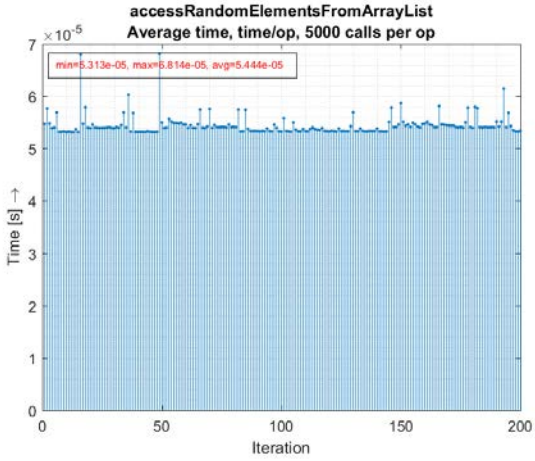


Abbildung 17: 2016_11_22-212504_accessRandomElementsFromArrayList_iterations.png

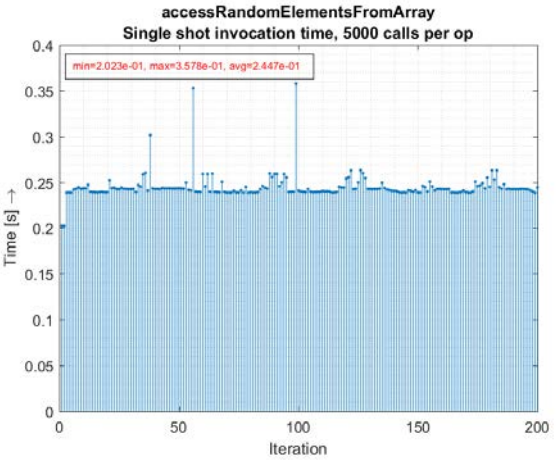


Abbildung 18: 2016_11_25-184721_accessRandomElementsFromArray_iterations.png

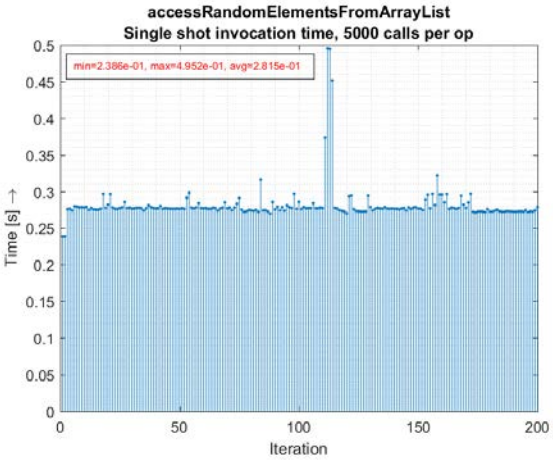


Abbildung 19: 2016_11_25-184721_accessRandomElementsFromArrayList_iterations.png

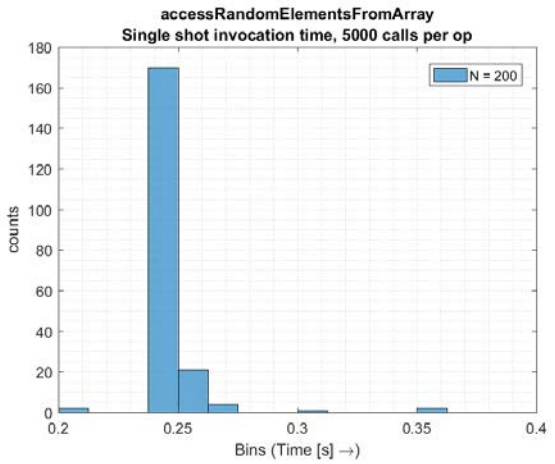


Abbildung 20: 2016_11_25-184721_accessRandomElementsFromArray_histogram.png

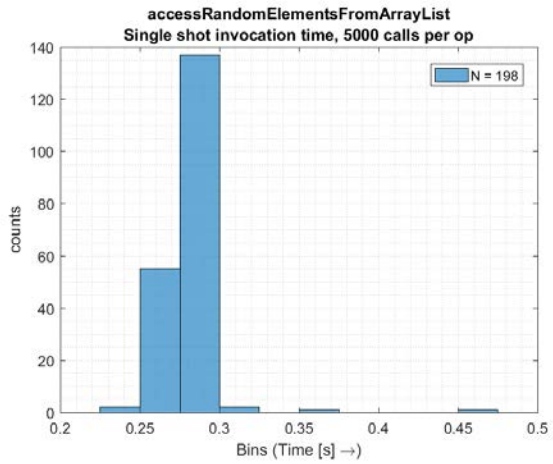


Abbildung 21: 2016_11_25-184721_accessRandomElementsFromArrayList_histogram.png

4.1.4 Auswertung und Empfehlung

Der Lesezugriff auf zufällige Elemente eines Arrays ist ungefähr 12-18 % schneller als bei der ArrayList, unabhängig vom Benchmark-Modus. Der Performancegewinn des Arrays gegenüber der ArrayList ist gering. Somit behalten beide Datenstrukturen je nach Situation ihre Daseinsberechtigung. Es wird empfohlen, für Collections mit unbekannter oder flexibler Länge eine ArrayList und für Datensätze fester Länge ein Array zu verwenden.

4.2 Iterationsvarianten

4.2.1 Fragestellung

Eine ArrayList und ein HashSet werden mit der gleichen Anzahl Objekten des Typs `Student` bestückt. Die Datenstrukturen werden mit einer klassischen for-each-Schleife, einem Iterator-Objekt und einer funktionalen for-each-Schleife durchlaufen. In jedem Schleifendurchgang wird ein Objekt aus der ArrayList oder dem HashSet vom `Blackhole`-Objekt konsumiert. Damit wird sichergestellt, dass die JIT-Kompilierung keine Optimierungen der Schleifendurchgänge vornimmt [28].

Bei Programmen, welche viel Zeit in Schleifen verbringen, stellt sich die Frage, ob die Wahl der Iterationsvariante einen entscheidenden Unterschied ausmacht.

```
@Setup(Level.Iteration)
public void setup() {
    Random random = new Random();
    studentList = new ArrayList<Student>(INITIAL_CAPACITY);
    studentSet = new HashSet<Student>(INITIAL_CAPACITY);
    for (int i = 0; i < INITIAL_CAPACITY; i++) {
        studentList.add(new Student(random.nextInt()));
        studentSet.add(new Student(random.nextInt()));
    }
}

@Benchmark
public void iteratorStudentListTest(Blackhole b) {
    Iterator<Student> iter = studentList.iterator();
    while (iter.hasNext())
        b.consume(iter.next());
}

@Benchmark
public void forEachStudentListTest(Blackhole b) {
    for (Student student : studentList)
        b.consume(student);
}

@Benchmark
public void streamStudentListTest(Blackhole b) {
    studentList.stream().forEach((student) -> b.consume(student));
}
```

```
@Benchmark
public void iteratorStudentSetTest(Blackhole b) {
    Iterator<Student> iter = studentSet.iterator();
    while (iter.hasNext())
        b.consume(iter.next());
}

@Benchmark
public void forEachStudentSetTest(Blackhole b) {
    for (Student student : studentSet)
        b.consume(student);
}

@Benchmark
public void streamStudentSetTest(Blackhole b) {
    studentList.stream().forEach((student) -> b.consume(student));
}
```

Listing 15: Benchmarks Iterationsvarianten

4.2.2 Vermutung (Hypothese)

Der Iterator-Benchmark enthält eine zusätzliche Objektinstanziierung und in jedem Schleifendurchgang erfolgen zwei Methodenaufrufe. Bei der klassischen for-each-Schleife wird pro Schleifendurchgang ein Objekt erzeugt. Die funktionale for-each-Schleife basiert auf einem Stream, welcher erzeugt werden muss.

Die klassische for-each-Schleife wird voraussichtlich die kürzesten Messwerte liefern.

4.2.3 Beobachtung der Messwerte

AverageTime					
	min	**delta**	**avg**	**delta**	**stdev**
2016_11_25-122849_IteratorForEachLoopTest					
iteratorStudentListTest	4.42E-06	**8.25%**	4.58E-06	**4.46%**	1.84E-07
forEachStudentListTest	4.83E-06	**15.98%**	5.52E-06	**20.70%**	9.69E-07
streamStudentListTest	4.06E-06	**0.00%**	4.37E-06	**0.00%**	7.32E-08
iteratorStudentSetTest	1.65E-05	**75.26%**	1.82E-05	**75.14%**	6.51E-07
forEachStudentSetTest	1.37E-05	**70.25%**	1.52E-05	**70.23%**	3.53E-07
streamStudentSetTest	4.08E-06	**0.00%**	4.52E-06	**0.00%**	3.08E-07
2016_11_25-220816_IteratorForEachLoopTest					
iteratorStudentListTest	4.60E-06	**9.63%**	4.76E-06	**5.17%**	5.47E-07
forEachStudentListTest	5.25E-06	**20.80%**	5.29E-06	**14.75%**	1.82E-07
streamStudentListTest	4.16E-06	**0.00%**	4.51E-06	**0.00%**	4.04E-07
iteratorStudentSetTest	1.51E-05	**72.52%**	2.02E-05	**77.98%**	2.75E-06
forEachStudentSetTest	1.55E-05	**73.32%**	1.86E-05	**76.00%**	5.32E-07
streamStudentSetTest	4.14E-06	**0.00%**	4.46E-06	**0.00%**	2.25E-07
2016_11_29-183505_IteratorForEachLoopTest					
iteratorStudentListTest	4.60E-06	**11.89%**	4.64E-06	**4.83%**	2.71E-07
forEachStudentListTest	5.25E-06	**22.81%**	5.27E-06	**16.19%**	9.21E-08
streamStudentListTest	4.05E-06	**0.00%**	4.42E-06	**0.00%**	3.78E-08
iteratorStudentSetTest	1.66E-05	**75.62%**	2.23E-05	**80.14%**	8.47E-07
forEachStudentSetTest	1.43E-05	**71.68%**	1.86E-05	**76.25%**	7.88E-07
streamStudentSetTest	4.05E-06	**0.00%**	4.42E-06	**0.00%**	3.91E-08
2016_11_30-092512_IteratorForEachLoopTest					
iteratorStudentListTest	4.60E-06	**10.78%**	4.62E-06	**4.39%**	8.96E-08
forEachStudentListTest	5.25E-06	**21.89%**	5.27E-06	**16.14%**	7.66E-08
streamStudentListTest	4.10E-06	**0.00%**	4.42E-06	**0.00%**	3.36E-08
iteratorStudentSetTest	1.62E-05	**74.97%**	2.25E-05	**80.36%**	9.51E-07
forEachStudentSetTest	1.40E-05	**70.96%**	1.88E-05	**76.55%**	6.92E-07
streamStudentSetTest	4.06E-06	**0.00%**	4.42E-06	**0.00%**	3.77E-08
2016_12_03-112710_IteratorForEachLoopTest					
iteratorStudentListTest	4.60E-06	**11.13%**	4.62E-06	**4.24%**	8.84E-08
forEachStudentListTest	5.25E-06	**22.18%**	6.07E-06	**27.10%**	8.75E-07
streamStudentListTest	4.09E-06	**0.00%**	4.42E-06	**0.00%**	4.60E-08
iteratorStudentSetTest	1.64E-05	**75.05%**	2.24E-05	**80.12%**	8.03E-07
forEachStudentSetTest	1.64E-05	**75.03%**	1.89E-05	**76.37%**	6.21E-07
streamStudentSetTest	4.10E-06	**0.00%**	4.46E-06	**0.00%**	1.43E-07

Listing 16: Auswertung der Testläufe von IteratorForEachLoopTest im AverageTime-Modus

SingleShotTime					
	min	**delta**	**avg**	**delta**	**stdev**
2016_11_25-210309_IteratorForEachLoopTest					
iteratorStudentListTest	1.99E-02	**24.60%**	2.13E-02	**27.22%**	3.55E-04
forEachStudentListTest	2.19E-02	**31.35%**	2.33E-02	**33.55%**	2.89E-03
streamStudentListTest	1.50E-02	**0.00%**	1.55E-02	**0.00%**	1.98E-03
iteratorStudentSetTest	2.98E-02	**49.60%**	3.08E-02	**50.80%**	1.20E-03
forEachStudentSetTest	2.99E-02	**49.77%**	3.11E-02	**51.27%**	8.45E-04
streamStudentSetTest	1.50E-02	**0.00%**	1.52E-02	**0.00%**	7.35E-04

Listing 17: Auswertung der Testläufe von IteratorForEachLoopTest im SingleShotTime-Modus

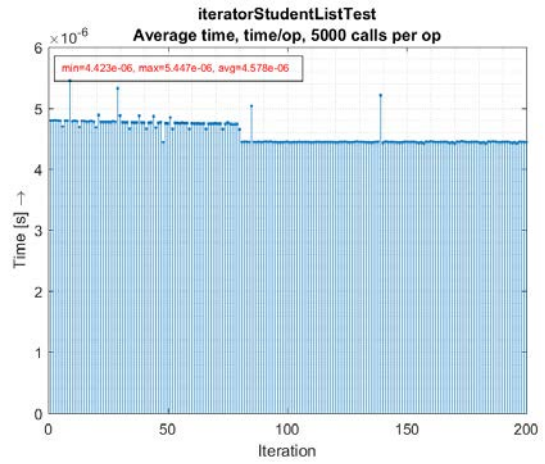


Abbildung 22: 2016_11_25-122849_iteratorStudentListTest_iterations.png

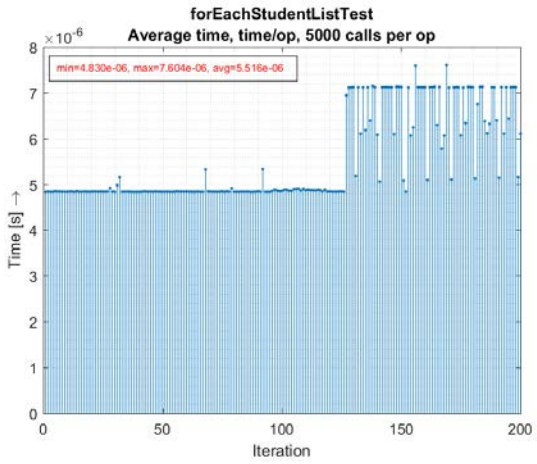


Abbildung 23: 2016_11_25-122849_forEachStudentListTest_iterations.png

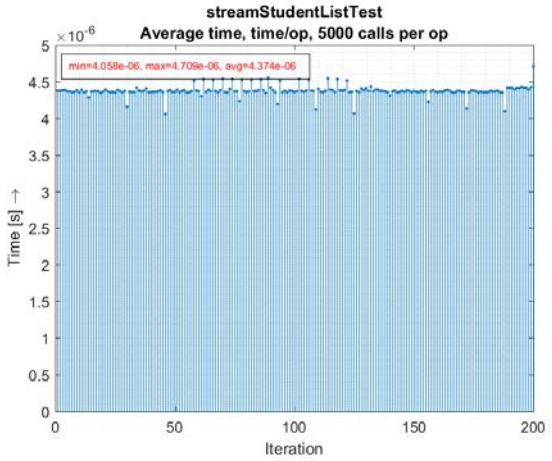


Abbildung 24: 2016_11_25-122849_streamStudentListTest_iterations.png

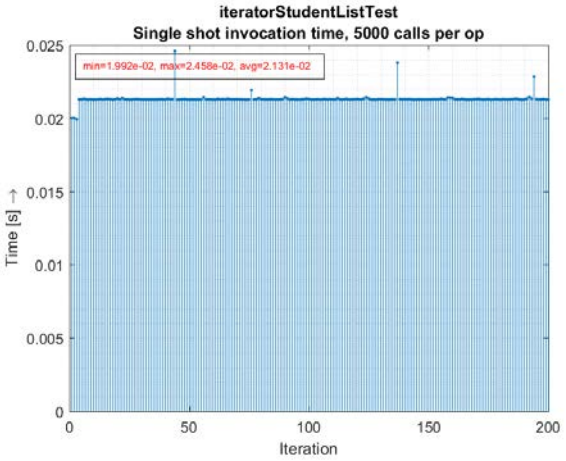


Abbildung 25: 2016_11_25-210309_iteratorStudentListTest_iterations.png

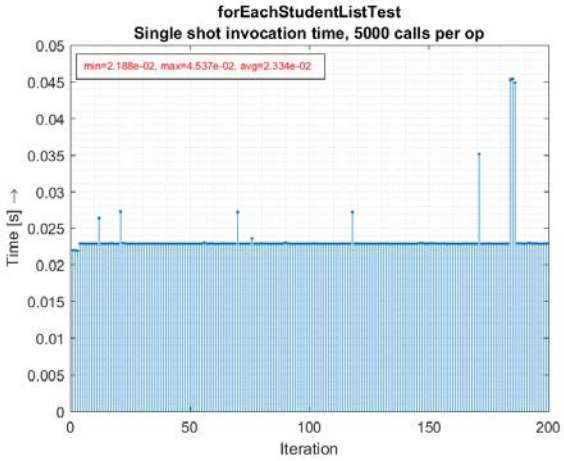


Abbildung 26: 2016_11_25-210309_forEachStudentListTest_iterations.png

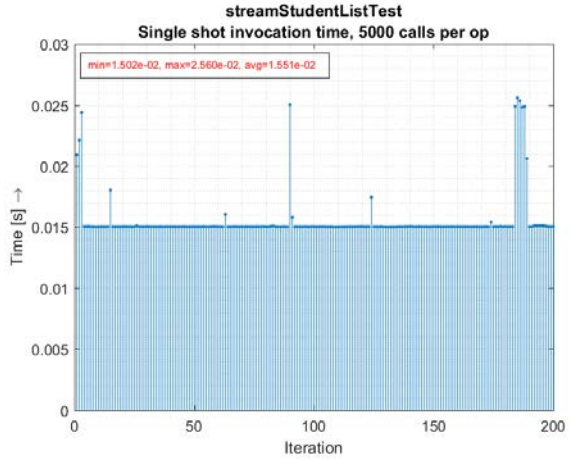


Abbildung 27: 2016_11_25-210309_streamStudentListTest_iterations.png

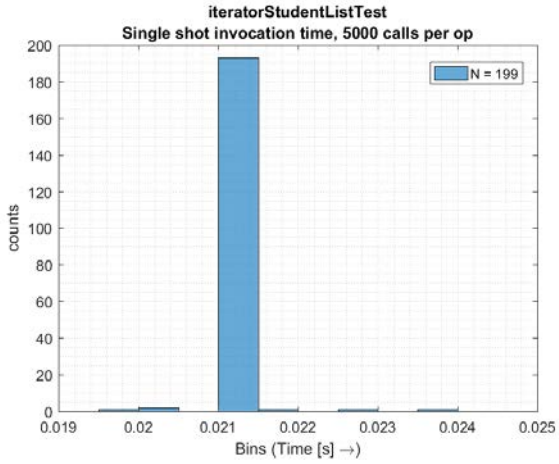


Abbildung 28: 2016_11_25-210309_iteratorStudentListTest_histogram.png

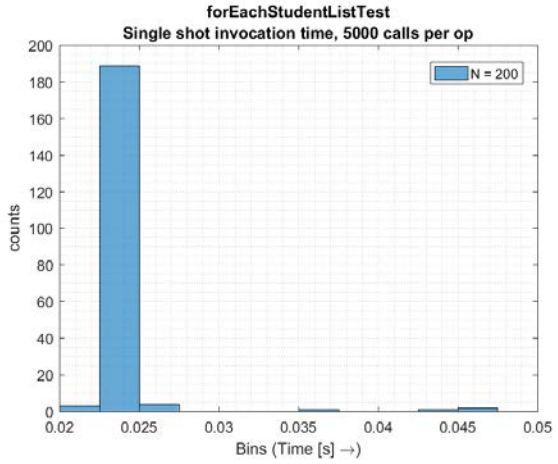


Abbildung 29: 2016_11_25-210309_forEachStudentListTest_histogram.png

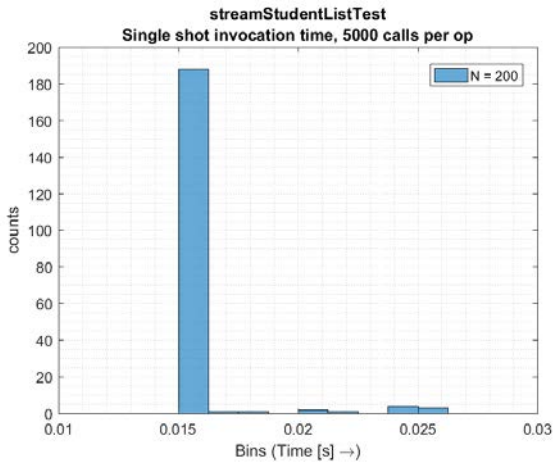


Abbildung 30: 2016_11_25-210309_streamStudentListTest_histogram.png

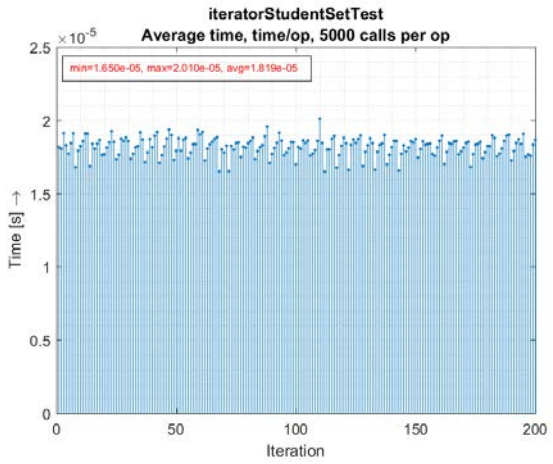


Abbildung 31: 2016_11_25-122849_iteratorStudentSetTest_iterations.png

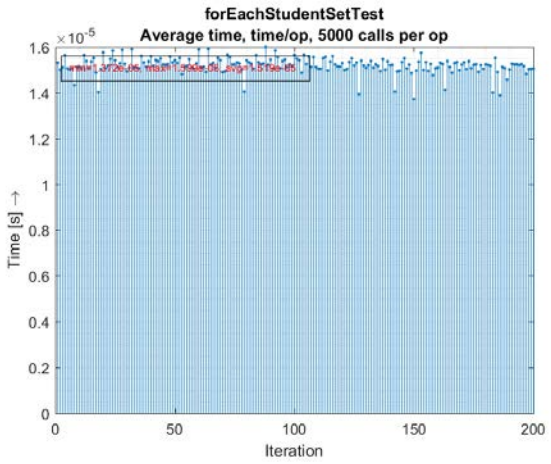


Abbildung 32: 2016_11_25-122849_forEachStudentSetTest_iterations.png

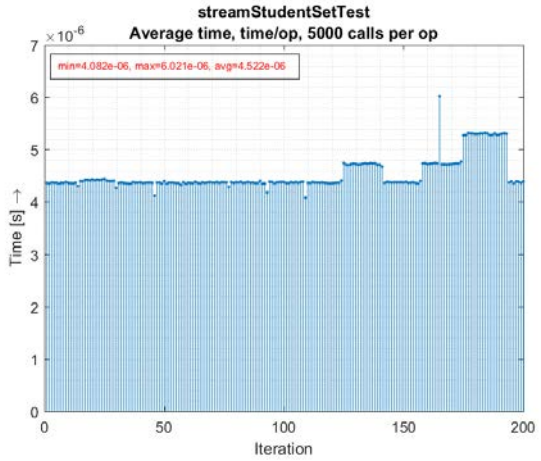


Abbildung 33: 2016_11_25-122849_streamStudentSetTest_iterations.png

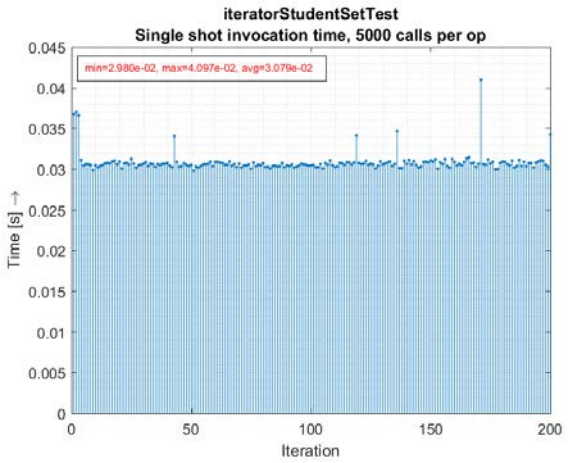


Abbildung 34: 2016_11_25-122849_iteratorStudentSetTest_iterations.png

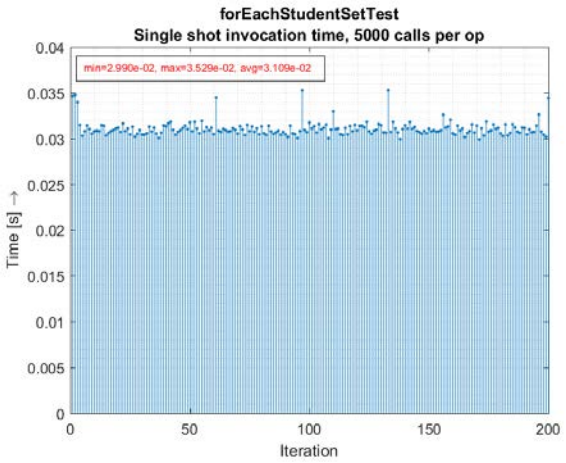


Abbildung 35: 2016_11_25-210309_forEachStudentSetTest_iterations.png

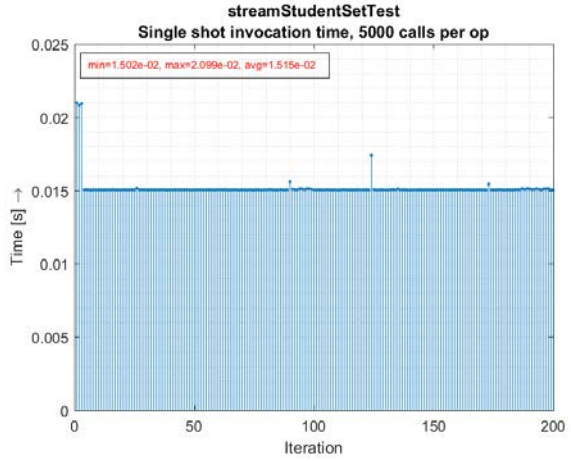


Abbildung 36: 2016_11_25-210309_streamStudentSetTest_iterations.png

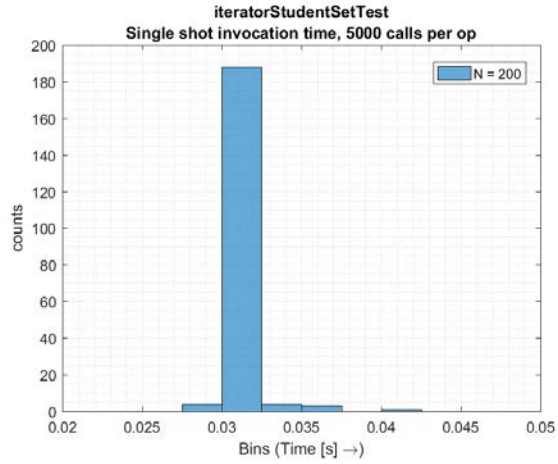


Abbildung 37: 2016_11_25-210309_iteratorStudentSetTest_histogram.png

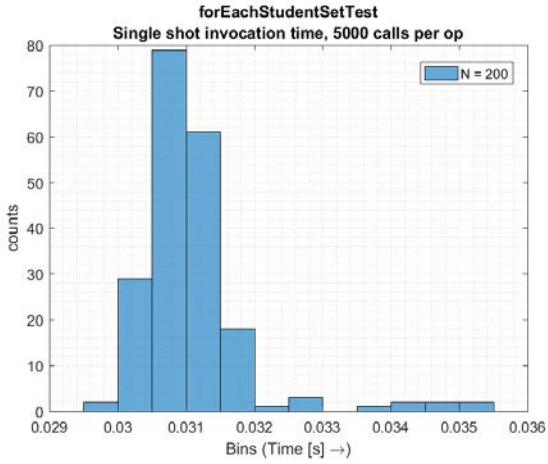


Abbildung 38: 2016_11_25-210309_forEachStudentSetTest_histogram.png

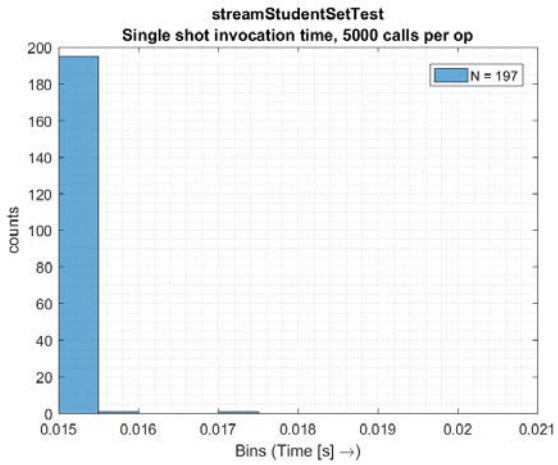


Abbildung 39: 2016_11_25-210309_streamStudentSetTest_histogram.png

4.2.4 Auswertung und Empfehlung

In allen Testszenarien schneidet die funktionale for-each-Schleife am besten ab, gefolgt vom Iterator und der etwas langsameren klassischen for-each-Schleife. Bei der HashSet-Iteration bringt die funktionale for-each-Schleife wesentliche Performance-Vorteile gegenüber den beiden anderen Varianten.

Es wird empfohlen, wann immer möglich eine funktionale for-each-Schleife zu verwenden.

4.3 Schleifenkopf mit lokaler Zählvariable oder Instanzvariable

4.3.1 Fragestellung

Ein `int`-Array wird mit zufälligen Werten bestückt. Anschliessend wird durch das Array iteriert und das aktuelle Arrayelement einem `Blackhole`-Objekt übergeben. Mit dem Experiment soll der optimalste Zählvariablentyp eruiert werden. Getestet werden lokale Zählvariablen, Instanzvariablen und statische Instanzvariablen.

```
@Setup(Level.Iteration)
public void setup() {
    randomValues = new int[RANGE];
    for (int i = 0; i < randomValues.length; i++) {
        randomValues[i] = RANDOM.nextInt();
    }
}

@Benchmark
public void instanceVariableLoopCounter(Blackhole b) {
    for (index = 0; index < RANGE; index++)
        b.consume(randomValues[index]);
}

@Benchmark
public void localVariableLoopCounter(Blackhole b) {
    for (int i = 0; i < RANGE; i++)
        b.consume(randomValues[i]);
}

@Benchmark
public void staticInstanceVariableLoopCounter(Blackhole b) {
    for (staticIndex = 0; staticIndex < RANGE; staticIndex++)
        b.consume(randomValues[staticIndex]);
}
```

Listing 18: Benchmarks lokale Variable vs. Instanzvariable in Schleifen

4.3.2 Vermutung (Hypothese)

Es wird angenommen, dass die Laufzeiten der drei Varianten gleich sind.

4.3.3 Beobachtung der Messwerte

AverageTime					
	min	**delta**	**avg**	**delta**	**stdev**
2016_11_22-212504_LocalVsInstanceVariableLoopCounterTest					
staticInstanceVariableLoopCounter	2.65E-05	**1.09%**	2.65E-05	**1.06%**	9.88E-08
instanceVariableLoopCounter	2.73E-05	**4.10%**	2.74E-05	**4.37%**	8.20E-07
localVariableLoopCounter	2.62E-05	**0.00%**	2.62E-05	**0.00%**	1.79E-07
2016_11_25-122849_LocalVsInstanceVariableLoopCounterTest					
staticInstanceVariableLoopCounter	2.65E-05	**1.06%**	2.68E-05	**0.00%**	4.33E-07
instanceVariableLoopCounter	2.73E-05	**4.10%**	2.74E-05	**2.12%**	4.40E-07
localVariableLoopCounter	2.62E-05	**0.00%**	2.70E-05	**1.00%**	3.33E-06
2016_11_25-220816_LocalVsInstanceVariableLoopCounterTest					
staticInstanceVariableLoopCounter	2.74E-05	**0.40%**	2.75E-05	**0.44%**	6.14E-07
instanceVariableLoopCounter	2.84E-05	**3.81%**	2.87E-05	**4.57%**	9.95E-07
localVariableLoopCounter	2.73E-05	**0.00%**	2.74E-05	**0.00%**	2.21E-07
2016_11_29-183505_LocalVsInstanceVariableLoopCounterTest					
staticInstanceVariableLoopCounter	2.74E-05	**0.00%**	2.74E-05	**0.00%**	1.33E-07
instanceVariableLoopCounter	2.87E-05	**4.36%**	2.88E-05	**4.69%**	1.39E-07
localVariableLoopCounter	2.75E-05	**0.29%**	2.75E-05	**0.29%**	1.52E-07
2016_11_30-092512_LocalVsInstanceVariableLoopCounterTest					
staticInstanceVariableLoopCounter	2.74E-05	**0.00%**	2.74E-05	**0.00%**	8.08E-09
instanceVariableLoopCounter	2.88E-05	**4.73%**	2.88E-05	**4.79%**	1.71E-07
localVariableLoopCounter	2.75E-05	**0.29%**	2.75E-05	**0.33%**	8.75E-08
2016_12_03-112710_LocalVsInstanceVariableLoopCounterTest					
staticInstanceVariableLoopCounter	2.74E-05	**0.00%**	2.74E-05	**0.00%**	1.06E-08
instanceVariableLoopCounter	2.87E-05	**4.33%**	2.88E-05	**4.79%**	1.43E-07
localVariableLoopCounter	2.75E-05	**0.25%**	2.75E-05	**0.36%**	1.45E-07

Listing 19: Auswertung der Testläufe von LocalVsInstanceVariableLoopCounterTest im AverageTime-Modus

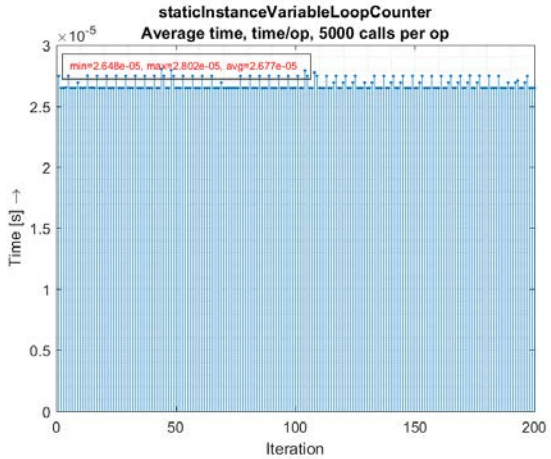


Abbildung 40: 2016_11_25-122849_staticInstanceVariableLoopCounter_iterations.png

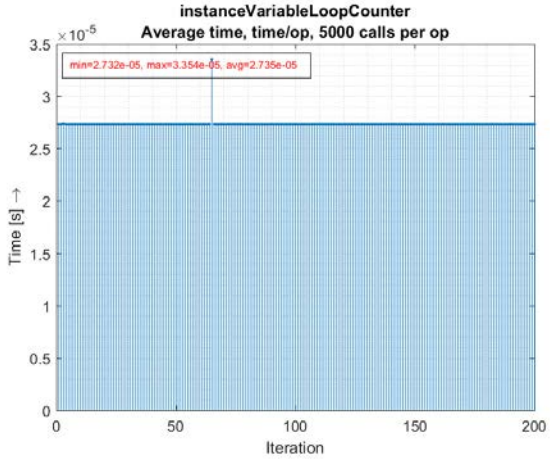


Abbildung 41: 2016_11_25-122849_instanceVariableLoopCounter_iterations.png

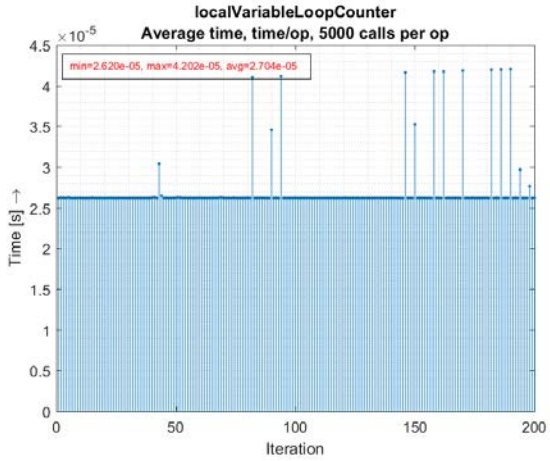


Abbildung 42: 2016_11_25-122849_localVariableLoopCounter_iterations.png

4.3.4 Auswertung und Empfehlung

Instanzvariablen sind nur geringfügig langsamer als eine lokale Variable. Für Schleifen werden lokale Indexvariablen empfohlen.

4.4 Direktzugriff auf Instanzvariablen vs. Getter- und Setter-Methoden

4.4.1 Fragestellung

Zwei Instanzen der Klasse `Vector` werden mit zufälligen x- und y-Koordinaten initialisiert. Es soll via Zugriffsmethoden oder mittels Direktzugriff eine Public-Instanzvariable gelesen und gesetzt werden. Welche Variante ist zu bevorzugen? Soll man Datenkapselung als Best-Practice aus der objektorientierten Programmierung anwenden oder Clean-Code-Regeln missachten?

```
@Setup(Level.Iteration)
public void setup() {
    vector1 = new Vector(RANDOM.nextDouble(), RANDOM.nextDouble());
    vector2 = new Vector(RANDOM.nextDouble(), RANDOM.nextDouble());
}

@Benchmark
public double getValueViaGetterMethodeTest() {
    return vector1.getX();
}

@Benchmark
public double getValueViaInstanceVariableTest() {
    return vector2.x;
}

@Benchmark
public Vector setValueViaSetterMethodeTest() {
    vector1.setX(42);
    return vector1;
}

@Benchmark
public Vector setValueViaInstanceVariableTest() {
    vector2.x = 42;
    return vector2;
}
```

Listing 20: Benchmarks Get/Set und Direktzugriff

4.4.2 Vermutung (Hypothese)

Es wird vermutet, dass kein zusätzlicher Aufwand beim Aufruf von Zugriffsmethoden (Getter/Setter) entsteht.

4.4.3 Beobachtung der Messwerte

AverageTime					
	min	**delta**	**avg**	**delta**	**stdev**
2016_11_22-212504_GetterSetterVsDirectAccessTest					
getValueViaGetterMethodeTest	2.61E-09	**0.00%**	2.62E-09	**0.08%**	7.23E-11
getValueViaInstanceVariableTest	2.61E-09	**0.00%**	2.61E-09	**0.00%**	6.56E-11
setValueViaSetterMethodeTest	2.77E-09	**0.04%**	2.79E-09	**0.00%**	4.05E-11
setValueViaInstanceVariableTest	2.77E-09	**0.00%**	2.84E-09	**1.80%**	2.87E-10
2016_11_25-122849_GetterSetterVsDirectAccessTest					
getValueViaGetterMethodeTest	2.61E-09	**0.04%**	2.61E-09	**0.00%**	2.67E-11
getValueViaInstanceVariableTest	2.61E-09	**0.00%**	2.86E-09	**8.60%**	4.26E-10
setValueViaSetterMethodeTest	2.77E-09	**0.00%**	2.80E-09	**0.00%**	1.32E-10
setValueViaInstanceVariableTest	2.78E-09	**0.18%**	2.85E-09	**1.79%**	2.90E-10
2016_11_25-220816_GetterSetterVsDirectAccessTest					
getValueViaGetterMethodeTest	2.62E-09	**0.04%**	2.63E-09	**0.00%**	5.25E-11
getValueViaInstanceVariableTest	2.62E-09	**0.00%**	2.64E-09	**0.23%**	6.20E-11
setValueViaSetterMethodeTest	2.83E-09	**0.00%**	3.00E-09	**3.54%**	4.05E-10
setValueViaInstanceVariableTest	2.83E-09	**0.04%**	2.89E-09	**0.00%**	1.72E-10
2016_11_29-183505_GetterSetterVsDirectAccessTest					
getValueViaGetterMethodeTest	2.62E-09	**0.08%**	2.62E-09	**0.00%**	8.77E-12
getValueViaInstanceVariableTest	2.62E-09	**0.00%**	2.63E-09	**0.08%**	1.81E-11
setValueViaSetterMethodeTest	2.82E-09	**0.00%**	2.85E-09	**0.00%**	4.14E-11
setValueViaInstanceVariableTest	2.82E-09	**0.07%**	2.85E-09	**0.21%**	6.85E-11
2016_11_30-092512_GetterSetterVsDirectAccessTest					
getValueViaGetterMethodeTest	2.62E-09	**0.00%**	2.63E-09	**0.00%**	1.17E-11
getValueViaInstanceVariableTest	2.62E-09	**0.00%**	2.63E-09	**0.00%**	1.50E-11
setValueViaSetterMethodeTest	2.81E-09	**0.00%**	2.85E-09	**0.00%**	3.79E-11
setValueViaInstanceVariableTest	2.83E-09	**0.81%**	2.87E-09	**0.73%**	2.39E-10
2016_12_03-112710_GetterSetterVsDirectAccessTest					
getValueViaGetterMethodeTest	2.62E-09	**0.00%**	2.65E-09	**0.60%**	1.44E-10
getValueViaInstanceVariableTest	2.62E-09	**0.00%**	2.63E-09	**0.00%**	2.72E-11
setValueViaSetterMethodeTest	2.83E-09	**0.21%**	2.87E-09	**0.24%**	3.19E-10
setValueViaInstanceVariableTest	2.82E-09	**0.00%**	2.86E-09	**0.00%**	1.28E-10

Listing 21: Auswertung der Testläufe von GetterSetterVsDirectAccessTest im AverageTime-Modus

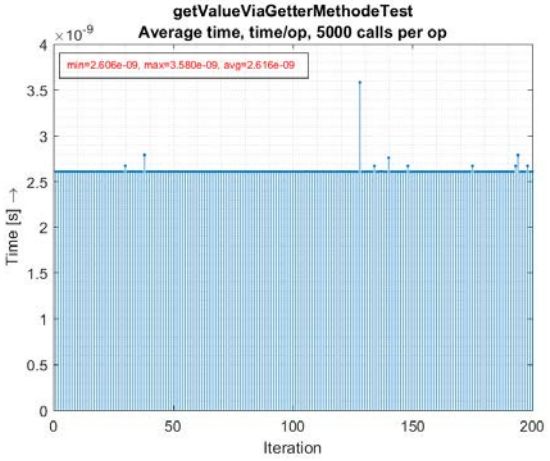


Abbildung 43: 2016_11_22-212504_getValueViaGetterMethodeTest_iterations.png

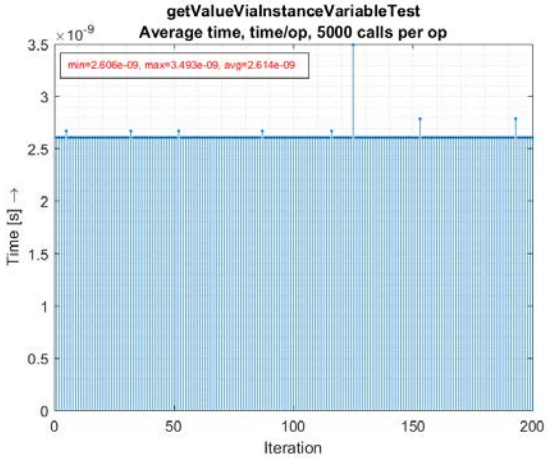


Abbildung 44: 2016_11_22-212504_getValueViaInstanceVariableTest_iterations.png

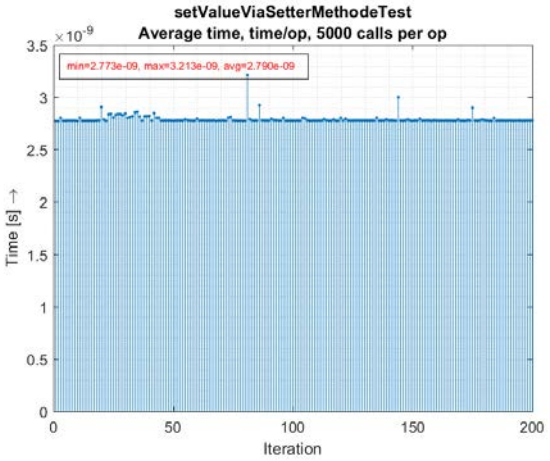


Abbildung 45: 2016_11_22-212504_setValueViaSetterMethodeTest_iterations.png

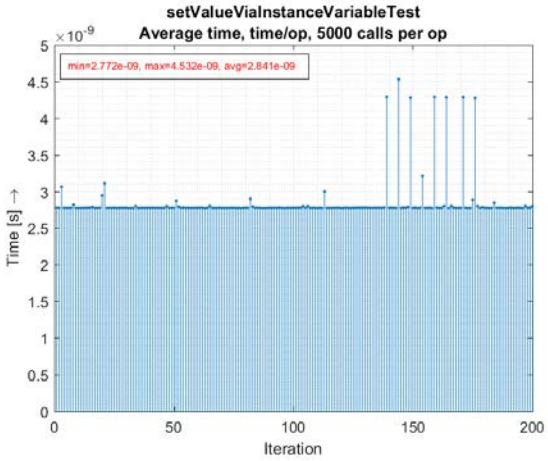


Abbildung 46: 2016_11_22-212504_setValueViaInstanceVariableTest_iterations.png

4.4.4 Auswertung und Empfehlung

Es lassen sich keine Unterschiede feststellen. Aus diesem Grund sollte man weiterhin Objektattribute `private` deklarieren und bei Bedarf Zugriffsmethoden bereitstellen. Dadurch wird eine Schnittstelle nach aussen definiert.

4.5 Lese- und Schreibzugriff auf Instanzvariablen (static, final, volatile)

4.5.1 Fragestellung

Verschiedene Modifikatoren, wie static, final, static final und volatile werden auf Instanzvariablen angewendet. Es werden Lese- und Schreibzugriffszeiten auf den Variablen verglichen.

```
@Param("2.71828")
public double arg;

@Benchmark
public double getValue() {
    return value;
}

@Benchmark
public double getStatic() {
    return staticValue;
}

@Benchmark
public double getfinal() {
    return FINAL_VALUE;
}

@Benchmark
public double getStaticFinal() {
    return STATIC_FINAL_VALUE;
}

@Benchmark
public double getVolatile() {
    return volatileValue;
}

@Benchmark
public boolean writeValue() {
    value = arg;
    return true;
}

@Benchmark
public boolean writeStatic() {
    staticValue = arg;
    return true;
}

@Benchmark
public boolean writeVolatile() {
    volatileValue = arg;
    return true;
}
```

Listing 22: Benchmarks static, volatile und final Instanzvariablen

4.5.2 Vermutung (Hypothese)

Der Zugriff auf die verschiedenen Variablen wird ungefähr gleich schnell sein. Das Schreiben auf eine volatile Variable könnte etwas länger brauchen, als auf die anderen beiden Typen.

4.5.3 Beobachtung der Messwerte

AverageTime					
	min	**delta**	**avg**	**delta**	**stdev**
2016_11_25-122849_StaticVolatileFinalTest					
getValue	2.60E-09	**2.00%**	2.62E-09	**2.29%**	3.95E-11
getFinalValue	2.57E-09	**0.90%**	2.59E-09	**1.16%**	5.75E-11
getStaticValue	2.73E-09	**6.64%**	2.74E-09	**6.71%**	2.98E-11
getStaticFinalValue	2.54E-09	**0.00%**	2.56E-09	**0.00%**	9.07E-12
getVolatileValue	2.60E-09	**2.00%**	2.60E-09	**1.43%**	6.70E-13
writeValue	2.43E-09	**0.00%**	2.44E-09	**0.00%**	3.87E-11
writeValueStatic	2.53E-09	**4.07%**	2.57E-09	**5.36%**	3.50E-11
writeValueVolatile	7.88E-09	**69.17%**	7.88E-09	**69.10%**	1.01E-12
2016_11_25-220816_StaticVolatileFinalTest					
getValue	2.60E-09	**2.12%**	2.60E-09	**1.84%**	3.30E-11
getFinalValue	2.54E-09	**0.00%**	2.56E-09	**0.08%**	3.91E-11
getStaticValue	2.73E-09	**6.82%**	2.73E-09	**6.44%**	3.70E-11
getStaticFinalValue	2.54E-09	**0.04%**	2.56E-09	**0.00%**	2.53E-11
getVolatileValue	2.60E-09	**2.12%**	2.60E-09	**1.66%**	1.27E-11
writeValue	2.43E-09	**0.00%**	2.46E-09	**0.00%**	1.14E-10
writeValueStatic	2.53E-09	**4.07%**	2.58E-09	**4.62%**	1.11E-10
writeValueVolatile	7.88E-09	**69.17%**	7.89E-09	**68.88%**	6.41E-11
2016_11_26-184833_StaticVolatileFinalTest					
getValue	2.60E-09	**2.35%**	2.62E-09	**2.94%**	6.11E-11
getFinalValue	2.56E-09	**0.82%**	2.57E-09	**1.13%**	8.14E-12
getStaticValue	2.73E-09	**7.07%**	2.77E-09	**8.31%**	2.12E-10
getStaticFinalValue	2.54E-09	**0.00%**	2.54E-09	**0.00%**	1.21E-11
getVolatileValue	2.60E-09	**2.35%**	2.61E-09	**2.53%**	8.63E-11
writeValue	2.43E-09	**0.00%**	2.44E-09	**0.00%**	3.75E-11
writeValueStatic	2.53E-09	**4.07%**	2.57E-09	**5.03%**	9.56E-11
writeValueVolatile	7.88E-09	**69.17%**	7.95E-09	**69.36%**	1.83E-10
2016_11_29-183505_StaticVolatileFinalTest					
getValue	2.60E-09	**2.12%**	2.60E-09	**1.69%**	1.25E-11
getFinalValue	2.54E-09	**0.00%**	2.55E-09	**0.00%**	9.66E-12
getStaticValue	2.73E-09	**6.75%**	2.73E-09	**6.38%**	3.62E-13
getStaticFinalValue	2.57E-09	**0.97%**	2.58E-09	**1.01%**	7.01E-12
getVolatileValue	2.60E-09	**2.12%**	2.60E-09	**1.62%**	7.19E-13
writeValue	2.43E-09	**0.00%**	2.43E-09	**0.00%**	1.04E-12
writeValueStatic	2.53E-09	**4.07%**	2.55E-09	**4.86%**	8.48E-12
writeValueVolatile	7.88E-09	**69.17%**	7.88E-09	**69.17%**	5.97E-13
2016_11_30-092512_StaticVolatileFinalTest					
getValue	2.60E-09	**1.81%**	2.60E-09	**1.35%**	7.42E-13
getFinalValue	2.55E-09	**0.00%**	2.56E-09	**0.00%**	7.15E-12
getStaticValue	2.73E-09	**6.53%**	2.73E-09	**6.12%**	3.82E-13
getStaticFinalValue	2.56E-09	**0.23%**	2.57E-09	**0.23%**	6.04E-12
getVolatileValue	2.60E-09	**1.81%**	2.60E-09	**1.39%**	7.37E-12
writeValue	2.43E-09	**0.00%**	2.43E-09	**0.00%**	7.82E-13
writeValueStatic	2.53E-09	**4.07%**	2.55E-09	**4.86%**	1.00E-11
writeValueVolatile	7.88E-09	**69.17%**	7.88E-09	**69.17%**	1.97E-13

Listing 23: Auswertung der Testläufe von StaticVolatileFinalTest im AverageTime-Modus

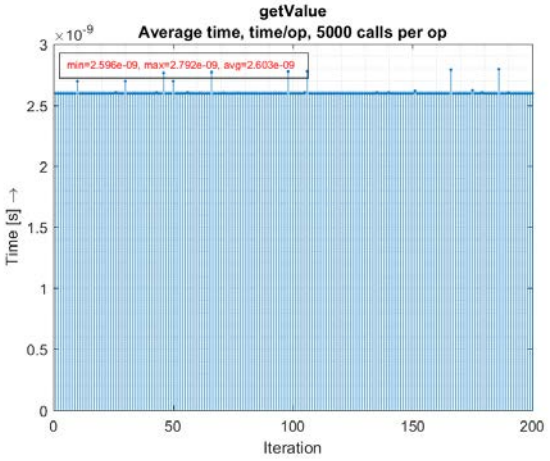


Abbildung 47: 2016_11_25-220816_getValue_iterations.png

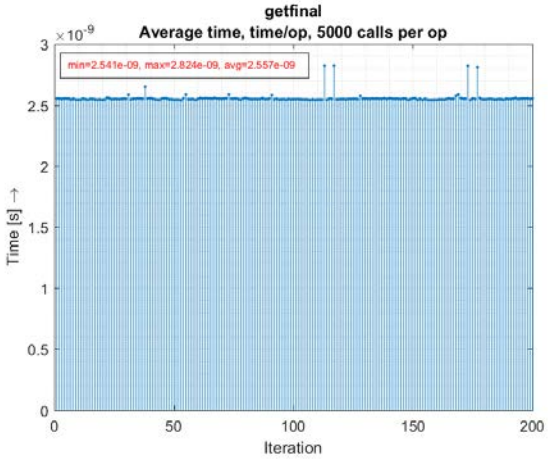


Abbildung 48: 2016_11_25-220816_getfinal_iterations.png

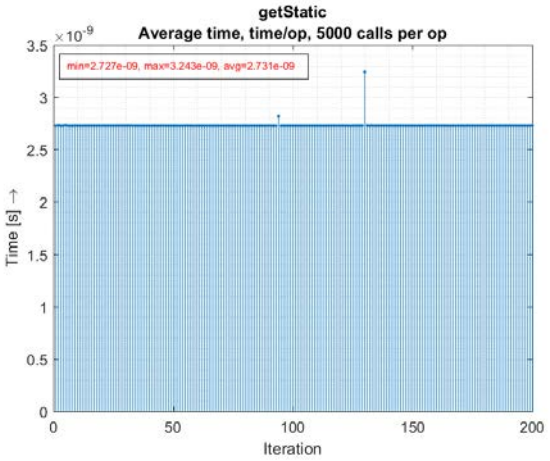


Abbildung 49: 2016_11_25-220816_getStatic_iterations.png

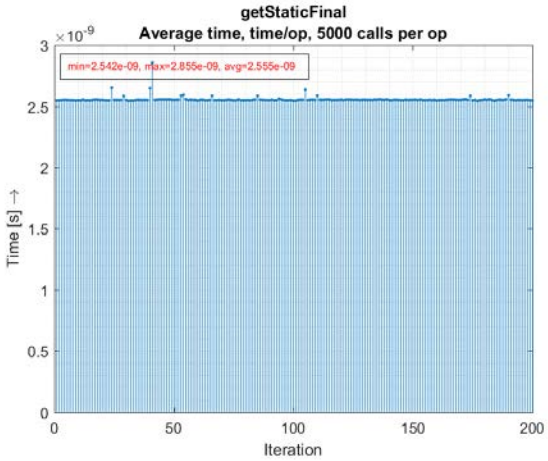


Abbildung 50: 2016_11_25-220816_getStaticFinal_iterations.png

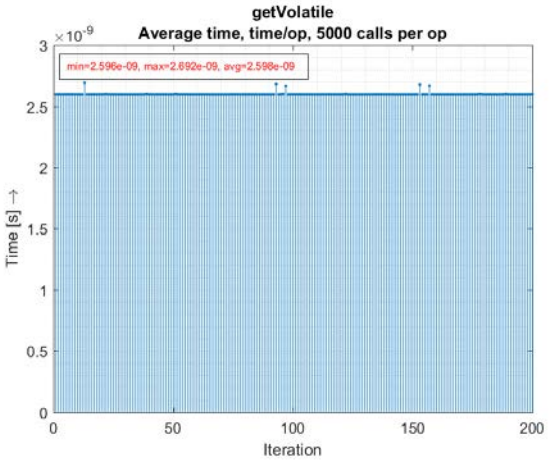


Abbildung 51: 2016_11_25-220816_getVolatile_iterations.png

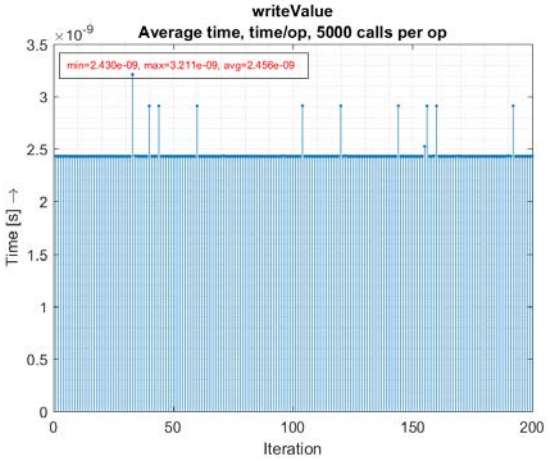


Abbildung 52: 2016_11_25-220816_writeValue_iterations.png

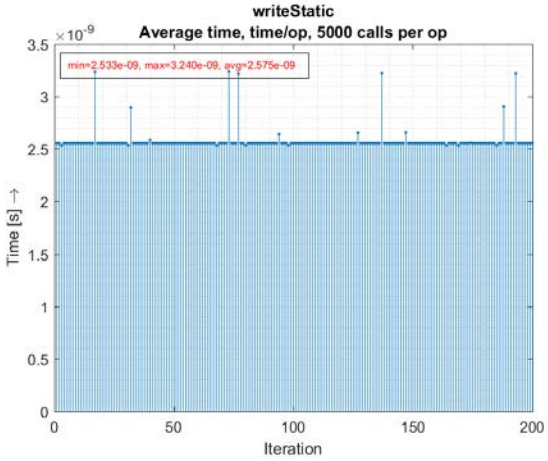


Abbildung 53: 2016_11_25-220816_writeStatic_iterations.png

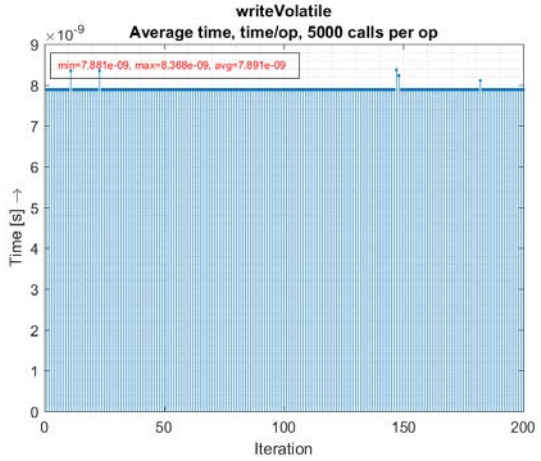


Abbildung 54: 2016_11_25-220816_writeVolatile_iterations.png

4.5.4 Auswertung und Empfehlung

Die Lesezugriffszeiten auf eine final und static final Instanzvariable sind etwa gleich schnell. Der Zugriff auf eine normale und volatile Instanzvariable braucht nur gering mehr. Am langsamsten ist eine static Instanzvariable. Die Unterschiede sind über alle Varianten sehr gering.

Die Schreibzugriffszeiten auf eine normale Instanzvariable sind schneller als auf eine static Instanzvariable. Das Schreiben auf eine volatile Instanzvariable braucht ungefähr dreimal länger.

Es wird empfohlen, die Modifikatoren je nach Verwendungszweck einzusetzen. Schreibzugriffe auf volatile Variablen sollten wann immer möglich vermieden werden.

4.6 Overhead von Abstraktion und Vererbung

4.6.1 Fragestellung

In der objektorientierten Programmierung gibt es den Grundsatz des Single Responsibility Principle [29]. Ein Objekt hat eine einzige Verantwortlichkeit und eine klare Schnittstelle, über die von aussen mit dem Objekt interagiert werden kann. Das führt zu übersichtlichem und wiederverwendbarem Code. In diesem Experiment wird untersucht, ob der Zugriff auf Attribute eines Objekts, welches nach Clean-Code-Best-Practice programmiert wurde, mit zusätzlichem Aufwand verbunden ist.

Die Benchmarks verwenden zwei Testklassen `Turbolaser` und `TurbolaserNoCleanCode`. `Turbolaser` wird nach objektorientiertem Design entworfen und erweitert zwei abstrakte Basisklassen `Projectile` und `GameObject`. `TurbolaserNoCleanCode` erbt von keiner Superklasse, enthält jedoch dieselben Methoden und Attribute wie `Turbolaser`.

4.6.2 Vermutung (Hypothese)

Der Overhead der Vererbung sollte gering sein, weil eine Optimierung (Code-Inlining) durchgeführt wird. JMH erlaubt die Steuerung des JIT-Compilers mittels Annotationen:

```
@CompilerControl(CompilerControl.Mode.INLINE)
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
```

Es soll herausgefunden werden, ob diese Annotationen die Messwerte beeinflussen.

```
private Turbolaser tl1;
private TurbolaserNoCleanCode tl2;
long then = System.nanoTime();

@Benchmark
public Point testAbstraction() {
    long now = System.nanoTime();
    tl1.update(now - (double) then);
      return tl1.getPosition();
}

@Benchmark
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
public Point doNotInLineTestAbstraction() {
    long now = System.nanoTime();
    tl1.update(now - (double) then);
      return tl1.getPosition();
}

@Benchmark
@CompilerControl(CompilerControl.Mode.INLINE)
public Point doInLineTestAbstraction() {
    long now = System.nanoTime();
    tl1.update(now - (double) then);
      return tl1.getPosition();
}
```

```
@Benchmark
public Point testNoAbstraction() {
    long now = System.nanoTime();
    tl2.update(now - (double) then);
      return tl2.getPosition();
}

@Benchmark
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
public Point doNotInLineTestNoAbstraction() {
    long now = System.nanoTime();
    tl2.update(now - (double) then);
      return tl2.getPosition();
}

@Benchmark
@CompilerControl(CompilerControl.Mode.INLINE)
public Point doInLineTestNoAbstraction() {
    long now = System.nanoTime();
    tl2.update(now - (double) then);
    return tl2.getPosition();
}
```

Listing 24: Benchmarks Vererbungshierarchie

4.6.3 Beobachtung der Messwerte

AverageTime					
	min	**delta**	**avg**	**delta**	**stdev**
2016_11_22-212504_AbstractionOverheadTest					
testAbstraction	3.38E-08	**0.65%**	3.46E-08	**1.62%**	8.85E-10
testNoAbstraction	3.36E-08	**0.00%**	3.40E-08	**0.00%**	7.75E-10
doInLineTestAbstraction	3.36E-08	**0.00%**	3.40E-08	**0.15%**	8.51E-10
doInLineTestNoAbstraction	3.36E-08	**0.00%**	3.40E-08	**0.00%**	6.69E-10
doNotInLineTestAbstraction	3.44E-08	**0.00%**	3.48E-08	**0.40%**	8.24E-10
doNotInLineTestNoAbstraction	3.44E-08	**0.03%**	3.46E-08	**0.00%**	3.50E-10
2016_11_25-122849_AbstractionOverheadTest					
testAbstraction	3.36E-08	**0.03%**	3.39E-08	**0.09%**	5.45E-10
testNoAbstraction	3.36E-08	**0.00%**	3.39E-08	**0.00%**	3.76E-10
doInLineTestAbstraction	3.36E-08	**0.03%**	3.39E-08	**0.00%**	3.07E-10
doInLineTestNoAbstraction	3.36E-08	**0.00%**	3.39E-08	**0.15%**	6.91E-10
doNotInLineTestAbstraction	3.44E-08	**0.00%**	3.46E-08	**0.00%**	2.06E-10
doNotInLineTestNoAbstraction	3.44E-08	**0.00%**	3.48E-08	**0.37%**	3.42E-10
2016_11_25-220816_AbstractionOverheadTest					
testAbstraction	3.37E-08	**0.33%**	3.44E-08	**1.57%**	2.83E-10
testNoAbstraction	3.36E-08	**0.00%**	3.38E-08	**0.00%**	2.00E-10
doInLineTestAbstraction	3.37E-08	**0.00%**	3.39E-08	**0.06%**	3.98E-10
doInLineTestNoAbstraction	3.37E-08	**0.00%**	3.39E-08	**0.00%**	1.80E-10
doNotInLineTestAbstraction	3.41E-08	**0.00%**	3.43E-08	**0.20%**	3.76E-10
doNotInLineTestNoAbstraction	3.41E-08	**0.00%**	3.42E-08	**0.00%**	1.13E-10
2016_11_29-183505_AbstractionOverheadTest					
testAbstraction	3.36E-08	**0.00%**	3.38E-08	**0.00%**	1.40E-10
testNoAbstraction	3.37E-08	**0.39%**	3.39E-08	**0.06%**	1.46E-10
doInLineTestAbstraction	3.36E-08	**0.59%**	3.38E-08	**0.00%**	1.39E-10
doInLineTestNoAbstraction	3.34E-08	**0.00%**	3.38E-08	**0.03%**	1.21E-10
doNotInLineTestAbstraction	3.41E-08	**0.00%**	3.42E-08	**0.00%**	1.15E-10
doNotInLineTestNoAbstraction	3.41E-08	**0.00%**	3.42E-08	**0.06%**	1.42E-10
2016_11_30-092512_AbstractionOverheadTest					
testAbstraction	3.37E-08	**0.39%**	3.38E-08	**0.00%**	1.15E-10
testNoAbstraction	3.35E-08	**0.00%**	3.38E-08	**0.03%**	1.28E-10
doInLineTestAbstraction	3.36E-08	**0.00%**	3.38E-08	**0.06%**	1.08E-10
doInLineTestNoAbstraction	3.37E-08	**0.30%**	3.38E-08	**0.00%**	1.03E-10
doNotInLineTestAbstraction	3.41E-08	**0.00%**	3.42E-08	**0.03%**	1.92E-10
doNotInLineTestNoAbstraction	3.41E-08	**0.00%**	3.42E-08	**0.00%**	1.39E-10
2016_12_03-112710_AbstractionOverheadTest					
testAbstraction	3.36E-08	**0.00%**	3.42E-08	**0.67%**	1.49E-09
testNoAbstraction	3.36E-08	**0.12%**	3.40E-08	**0.00%**	7.80E-10
doInLineTestAbstraction	3.36E-08	**0.00%**	3.38E-08	**0.00%**	2.35E-10
doInLineTestNoAbstraction	3.37E-08	**0.15%**	3.39E-08	**0.27%**	5.77E-10
doNotInLineTestAbstraction	3.41E-08	**0.00%**	3.43E-08	**0.26%**	9.01E-10
doNotInLineTestNoAbstraction	3.41E-08	**0.00%**	3.43E-08	**0.00%**	4.65E-10

Listing 25: Auswertung der Testläufe von AbstractionOverheadTest im AverageTime-Modus

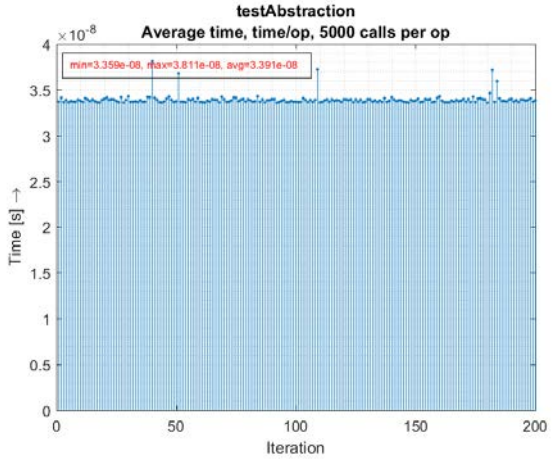


Abbildung 55: 2016_11_25-122849_testAbstraction_iterations.png

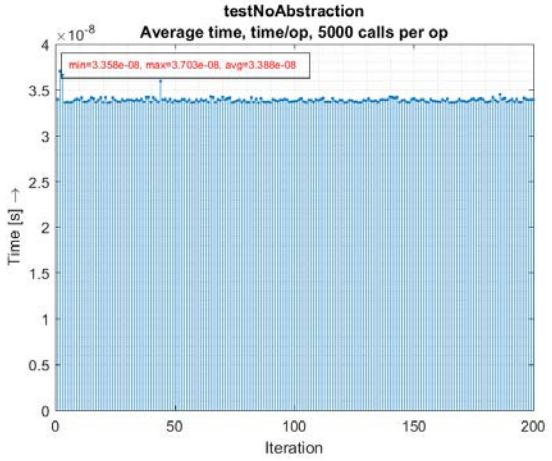


Abbildung 56: 2016_11_25-122849_testNoAbstraction_iterations.png

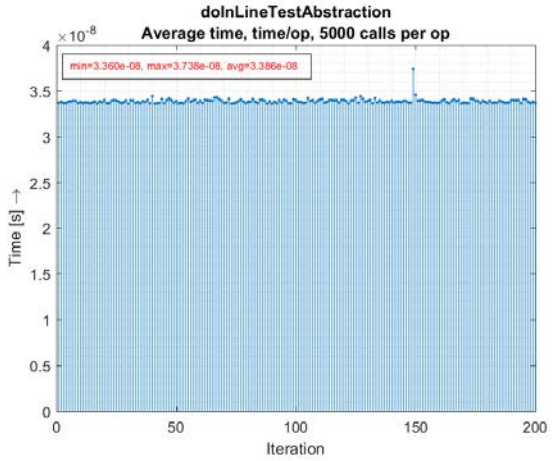


Abbildung 57: 2016_11_25-122849_doInLineTestAbstraction_iterations.png

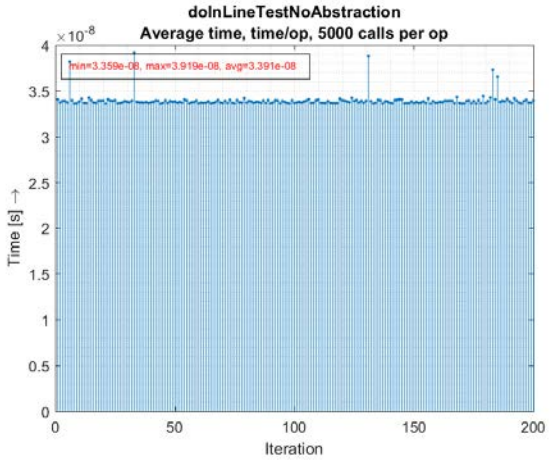


Abbildung 58: 2016_11_25-122849_doInLineTestNoAbstraction_iterations.png

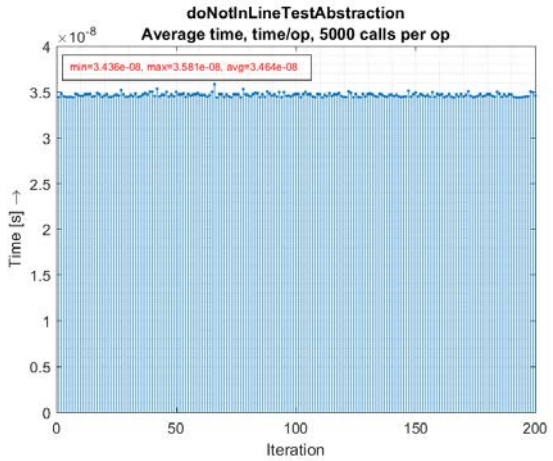


Abbildung 59: 2016_11_25-122849_doNotInLineTestAbstraction_iterations.png

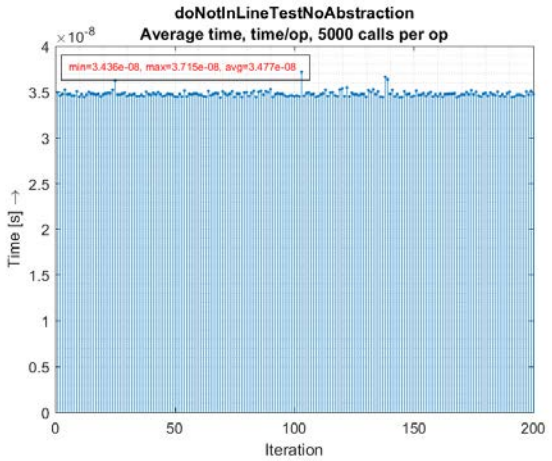


Abbildung 60: 2016_11_25-122849_doNotInLineTestNoAbstraction_iterations.png

4.6.4 Auswertung und Empfehlung

Das durchgeführte Experiment ist ergebnislos. Die erhaltenen Messwerte sind wenig überzeugend.

Die Messresultate deuten darauf hin, dass der Abstraktions-Overhead vom JIT-Compiler wegoptimiert wurde. Die Laufzeiten der Benchmarks sind zu kurz um eine Aussage zu treffen. Man könnte ein aufwendigeres Szenario aufbauen, bei dem beispielsweise künstlich CPU-Zeit verbraucht wird. JMH stellt dazu die Methode `Blackhole.consumeCPU()` bereit.

4.7 JIT-Performance

Zum Abschluss soll getestet werden, wie viel Performancegewinn die Just-in-time-Kompilierung gegenüber rein interpretiertem Bytecode auf einer HotSpot Java Virtual Machine liefern kann. Dazu werden zwei komplette Durchläufe aller Experimente verglichen, einmal mit aktivierten Compiler-Optimierungen (siehe Kapitel 2.1) und einmal mit deaktivierter JIT-Kompilierung.

Bleibt der JIT-Compiler ausgeschaltet, so arbeitet die JVM im Interpreter-Only-Mode und kompiliert keinen Assemblycode. Wie bereits erwähnt, lässt sich die JIT-Kompilierung mittels `–Xint` beim Starten der VM deaktivieren [10].

Benchmark	**Laufzeit [ns]**	
	JIT	**Interpretiert**
doInLineTestAbstraction	33.822	1405.824
doInLineTestNoAbstraction	33.804	1397.348
doNotInLineTestAbstraction	34.24	1395.578
doNotInLineTestNoAbstraction	34.226	1403.844
testAbstraction	33.796	1416.48
testNoAbstraction	33.807	1414.379
accessRandomElementsFromArray	47242.192	831228.116
accessRandomElementsFromArrayList	56158.053	2942086.726
getValueViaGetterMethodeTest	2.626	217.996
getValueViaInstanceVariableTest	2.626	155.393
setValueViaSetterMethodeTest	2.847	224.687
setValueViaInstanceVariableTest	2.868	156.205

iteratorStudentListTest	4621.463	302241.572
forEachStudentListTest	5267.759	309901.2
streamStudentListTest	4418.09	225237.559
iteratorStudentSetTest	22492.81	419452.791
forEachStudentSetTest	18826.973	427572.694
streamStudentSetTest	4416.224	228919.575
instanceVariableLoopCounter	28796.968	882350.029
localVariableLoopCounter	27507.372	792863.645
staticInstanceVariableLoopCounter	27416.86	813779.389
getValue	2.596	156.748
getStaticValue	2.728	156.01
getFinalValue	2.561	154.114
getStaticFinalValue	2.567	155.93
getVolatileValue	2.596	154.835
writeValue	2.43	158.307
writeValueStatic	2.554	155.996
writeValueVolatile	7.881	178.619

Tabelle 10: Laufzeitvergleich zwischen nativ kompiliertem und interpretiertem Code

Die Ergebnisse sprechen eine deutliche Sprache. Die JIT-Kompilierung liefert eine 17 bis 83-fache Effizienzsteigerung.

5 Diskussion und Ausblick

5.1 Rückblick auf die Resultate

Wie man gesehen hat, ist es beim Testen von Java-Benchmarks von entscheidender Bedeutung, dass man Fallstricke umgeht. Setzt man JMH korrekt ein, dann lassen sich gezielte Fragen zur Performance und Effizienz von Java-Code beantworten.

Die Ergebnisse aus allen Experimenten mit Ausnahme von „Overhead von Abstraktion und Vererbung“ haben ihren Erwartungen erfühlt. Mehrere Testläufe derselben Benchmarks haben im AverageTime-Benchmarkmodus wiederholt zuverlässige und aussagekräftige Messwerte geliefert. Vergleiche zwischen Laufzeitmessungen ergaben eine geringe Streuung der Messwerte. Der SingleShotTime-Benchmarkmodus erwies sich nur für die Experimente „Zugriffe auf Array und ArrayList“ und „Iterationsvarianten“ als brauchbar.

5.2 Interpretation und Validierung der Resultate

5.2.1 Zugriffe auf Array und ArrayList

Falls eine ArrayList verwendet werden kann, sollte sie nicht durch ein Array ersetzt werden. Da der Vorteil der Flexibilität der ArrayList gegenüber dem Array den Nachteil der marginalen Performanceeinbusse von etwa 12-18 % überwiegt.

5.2.2 Iterationsvarianten

Die funktionale for-each-Schleife erzielte in den untersuchten Benchmarks die kürzesten Laufzeiten. Sie sollte einem Iterator oder einer klassischen for-each-Schleife vorgezogen werden.

5.2.3 Schleifenkopf mit lokaler Zählvariable oder Instanzvariable

Die beobachteten Messwerte lassen darauf schliessen, dass die Wahl zwischen einer lokalen und einer Instanzzählvariablen ohne Bedeutung ist. Die drei Benchmarks lieferten fast identische Messresultate.

5.2.4 Direktzugriff auf Instanzvariablen vs. Getter- und Setter-Methoden

Das Konzept der Datenkapselung in der objektorientierten Programmierung sollte weiterhin angewendet werden. Lese- und Schreibzugriffe auf Objektattribute über Getter- und Setter-Methoden sind nicht langsamer als Direktzugriffe.

5.2.5 Lese- und Schreibzugriff auf Instanzvariablen

Schreibzugriffe auf volatile Variablen sollten aufgrund der erhöhten Latenz vermieden werden. Aus der Sicht der Performance gibt es bei den anderen Modifikatoren keine Favoriten.

5.2.6 Overhead von Abstraktion und Vererbung

Die Resultate aus diesem Experiment können nicht validiert werden. Es wurde angenommen, dass Methodenaufrufe mit einem kleinen aber messbaren zusätzlichen Aufwand verbunden sind. Die Messwerte offenbaren ein anderes Bild. Es wäre denkbar, dass der JIT-Compiler den Code mittels Inlining so weit optimiert hat, dass keine messbaren Unterschiede feststellbar sind. Weitere Tests mit komplexeren und rechenintensiveren Benchmark-Methoden sind notwendig.

5.3 Rückblick auf Aufgabenstellung

Obwohl nicht alle Fragen restlos geklärt werden konnten, wurden dennoch einige interessante Best-Practices zusammengetragen. Zudem konnte neues Wissen über das Java Microbenchmark Harness aufgebaut werden. Die Anforderungen an die Aufgabenstellung konnten somit abgedeckt werden.

5.4 Erweiterungsmöglichkeiten

Das Thema Vererbung, insbesondere die Auswirkung des Methoden-Inlining, bietet sich für zukünftige Forschungsarbeiten an. Auch Programmierparadigmen aus der funktionalen Programmierung bieten ausreichend Material für weitere Benchmarks. Mit dem Release-Datum von Java SE 9 [30] in nicht allzu weiter Ferne kommen wieder einige neue Features und Optimierungen hinzu, welche untersucht werden können.

6 Verzeichnisse

6.1 Quellenverzeichnis

[1] A. Georges, D. Buytaert und L. Eeckhout, «Statistically rigorous java performance evaluation,» in *ACM SIGPLAN Notices - Proceedings of the 2007 OOPSLA*, Montreal, Quebec, Canada, 2007.

[2] S. Oaks, Java Performance: The Definitive Guide, Sebastopol, California: O'Reilly Media, 2014, pp. 73-76.

[3] M. De Tomasi und M. Rutz, Best-Practices für performante Java-Programmierung [Software Engineering], Winterthur: Zürcher Hochschule für Angewandte Wissenschaften, 2016.

[4] M. Simbürger und R. Thöni, Best-Practices zur Performance-Optimierung bei der Software-Entwicklung in Java, Winterthur: Zürcher Hochschule für angewandte Wissenschaften, 2015.

[5] «Github: google/caliper wiki,» [Online]. Available: https://github.com/google/caliper/wiki. [Zugriff am 19.11.2016].

[6] OpenJDK, «Java Microbenchmark Harness (JMH),» [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh. [Zugriff am 14.11.2016].

[7] «Aleksey Shipilëv: One Stop Page,» [Online]. Available: https://shipilev.net/. [Zugriff am 24.11.2016].

[8] A. Shipilëv, «Java Microbenchmark Harness (the lesser of two evils),» 2013. [Online]. Available: https://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf. [Zugriff am 19.11.2016].

[9] OpenJDK, «JMH Samples,» [Online]. Available: http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/. [Zugriff am 19.11.2016].

[10] Oracle, «Oracle Java™ Documentation: Java Platform, Standard Edition Tools Reference,» [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html. [Zugriff am 03.12.2016].

[11] S. Oaks, Java Performance: The Definitive Guide, Sebastopol, California: O'Reilly Media, 2014, pp. 77-81.

[12] B. Evans, «Oracle Technology Network: Understanding Java JIT Compilation with JITWatch, Part 1,» [Online]. Available: http://www.oracle.com/technetwork/articles/java/architect-evans-pt1-2266278.html. [Zugriff am 03.12.2016].

[13] S. Oaks, Java Performance: The Definitive Guide, Sebastopol, California: O'Reilly Media, 2014, pp. 85-87.

[14] J. Gosling, B. Joy, G. Steele und G. Bracha, The Java Language Specification, Second Edition Hrsg., Addison-Wesley, 2000, p. 1.

[15] Oracle, «Oracle Java™ Documentation: Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide,» [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/. [Zugriff am 08.12.2016].

[16] S. Oaks, Java Performance: The Definitive Guide, Sebastopol, California: O'Reilly Media, 2014.

[17] S. Oaks, Java Performance: The Definitive Guide, Sebastopol, California: O'Reilly Media, 2014, pp. 105-132.

[18] Oracle, «Java™ Platform Standard Edition 8 API Specification: Class System,» [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime. [Zugriff am 11.14.2016].

[19] P. Mandl, Grundkurs Betriebssysteme: Architekturen, Betriebsverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung, 4. Auflage Hrsg., Wiesbaden: Springer Fachmedien, 2014, pp. 111-145.

[20] A. Shipilëv, «Java Benchmarking: as easy as two timestamps,» [Online]. Available: https://shipilev.net/talks/jvmls-July2014-benchmarking.pdf. [Zugriff am 14.11.2016].

[21] Intel, «Intel ARK,» [Online]. Available: http://ark.intel.com/products/52271/Intel-Xeon-Processor-E3-1230-8M-Cache-3_20-GHz. [Zugriff am 19.11.2016].

[22] J. L. Hennessy und D. A. Patterson, Computer Architecture: A Quantitative Approach, 5. Ausgabe Hrsg., Boston, MA: Elsevier/Morgan Kaufmann, 2012, p. 22.

[23] OpenJDK, «JMH Samples: Batch Size,» [Online]. Available: http://hg.openjdk.java.net/code-tools/jmh/file/e0764df2e4ee/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_26_BatchSize.java. [Zugriff am 15.11.2016].

[24] S. Oaks, Java Performance: The Definitive Guide, Sebastopol, California: O'Reilly Media, 2014, pp. 101-104.

[25] OpenJDK, «JMH Samples: Safe Looping,» [Online]. Available: http://hg.openjdk.java.net/code-tools/jmh/file/bb860b8d1eb6/jmh-

samples/src/main/java/org/openjdk/jmh/samples/JMHSample_34_SafeLooping.java. [Zugriff am 23.11.2016].

[26] «GrepCode Class Source: ArrayList,» [Online]. Available: http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/ArrayList.java/. [Zugriff am 23.11.2016].

[27] Oracle, «javap - The Java Class File Disassembler,» [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html. [Zugriff am 25.11.2016].

[28] J. Ponge, «Avoiding Benchmarking Pitfalls on the JVM,» Oracle, [Online]. Available: http://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html. [Zugriff am 17.12.2016].

[29] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Upper Saddle River, N.J.: Prentice Hall, 2009, pp. 135-150.

[30] «OpenJDK: JDK 9,» [Online]. Available: http://openjdk.java.net/projects/jdk9/. [Zugriff am 17.12.2016].

[31] P. Mandl, Grundkurs Betriebssysteme: Architekturen, Betriebsverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung, 4. Auflage Hrsg., Wiesbaden: Springer Fachmedien, 2014, p. 117.

6.2 Abbildungsverzeichnis

6.3 Listings

6.4 Tabellenverzeichnis

7 Anhang

- Offizielle Aufgabenstellung
- Projektplan
- Datenträger

Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

Best-Practices für performante Java-Programmierung (Software Engineering) PA16_tham_1

BetreuerInnen:	Markus Thaler, tham Mark Cieliebak, ciel
Fachgebiete:	Software (SOW)
Studiengang:	IT
Zuordnung:	Institut für angewandte Informationstechnologie (InIT)
Gruppengrösse:	2

Kurzbeschreibung:
Qualitativ hochwertige Software ist ein entscheidender Faktor für den nachhaltigen Erfolg von ICT-Produkten. Vereinfacht ausgedrückt gilt: Gute SW = korrekt + effizient + wartungsfreundlich.

In dieser Projektarbeit liegt der Focus auf dem Thema Effizient: Es soll eine Sammlung von Best-Practices entwickelt werden, wie man die Effizient von Java-basierten Software-Systemen optimieren kann. Einige exemplarische Fragestellungen sind: Ist ein Array schneller als eine ArrayList? Welche JVM-Einstellungen wirken sich wie auf die Performance aus? Welchen Einfluss hat Autoboxing auf die Laufzeit? Wie schnell sind die neuen Konstrukte in Java 8?

Die Best-Practices sollen auf objektiven und nachvollziehbaren Daten und Messungen beruhen, die in strukturierten Laufzeit-Experimenten erhoben wurden.

Grundlage: In den letzten zwei Jahren wurde in mehreren Projekt/Bachelorarbeiten ein System (Hardware und Software) aufgebaut, mit dem man auf Knopfdruck vergleichende Effizienz-Messungen machen kann: man programmiert z.B. mehrere Methoden, die jeweils ein Array komplett durchlaufen; das System führt diese Methoden automatisch aus, ermittelt die Laufzeit der einzelnen Methoden und erzeugt eine entsprechende Grafik. Dieses System können Sie für Ihre Experimente verwenden.

Im Rahmen der Arbeit werden Sie u.a. folgende Teilaufgaben bearbeiten:

- Einarbeitung ins Thema Effizienz-Messung
- das Framework in Betrieb nehmen und überprüfen, dass es funktioniert (keine Messfehler)
- Konkrete Software-Fragmente implementieren und vergleichen
- Übersichtliche Darstellung der Ergebnisse und Best-Practices
- Optional: Ausbau des Frameworks, sodass neben der Laufzeit weitere Faktoren wie Speicherverbrauch oder Anzahl erzeugte Objekte gemessen werden

Literatur

- Joshua Bloch: Effective Java
- Carl Stephen Lebsack: Performance Analysis and Optimization of the Java Memory System

Voraussetzungen:
- Gute Java-Kenntnisse
- Struturiertes experimentelles Arbeiten

Die Arbeit ist vereinbart mit:
Rémi Georgiou (georgrem)
André Stocker (stockan1)

Weiterführende Informationen:
http://buytaert.net/files/oopsla07-georges.pdf

Samstag 26. November 2016 17:20

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

Projektplan für Projektarbeit IT - HS 2016

Best-Practices für performante Java-Programmierung

Projektstart **19.09.2016** Version: **1.2**
Projektende **23.12.2016** Status:
Projektleiter Rémi Georgiou, André Stocker

Aufgabe	Start		Ende	Status	Kalenderwoche 38	39	40	41	42	43	44	45	46	47	48	49	50	51
Bericht schreiben Dokumentation der Experimente, Literaturübersicht	14.10.2016		11.12.2016	Abgeschlossen														
Experimente (5-10 Wow-Effekte) untersuchen	15.10.2016		30.11.2016	Abgeschlossen														
Projektarbeit fertigstellen Arbeit drucken, Ringbuch erstellen			18.12.2016	Abgeschlossen														
Einarbeitung in JMH Benchmarking-Framework	19.09.2016		09.10.2016	Abgeschlossen														
Testsystem von BA 2016 in Betrieb nehmen (MySQL DB + parser + jfreechart graphs)	19.09.2016		09.10.2016	Abgeschlossen														
Refactoring von bestehenden Microbenchmarks	10.10.2016		31.10.2016	Abgeschlossen														
Literatur studieren	19.09.2016		15.12.2016	Abgeschlossen														
Analyse und Dokumentation der Problematiken mit Java Performance-Messungen	17.10.2016		23.11.2016	Abgeschlossen														
Benchmarks auf Testsystem ausführen und Charts generieren	01.10.2016		04.12.2016	Abgeschlossen														
Projektarbeit abgeben Deadline: 23.12.2016																		

Der beigefügte Datenträger beinhaltet folgende Ordner- und Dateistruktur:

Dateityp	Name	Beschreibung
Ordner	java-create-charts	Mit diesem Java-Maven-Projekt können Diagramme erstellt werden. Dieses Projekt wurde in der Bachelorarbeit von Marco De Tomasi und Markus Rutz entwickelt. In dieser Projektarbeit wurde es nicht verwendet. Es wird nur der Vollständigkeit halber mitgeliefert.
Ordner	java-performance-tests	Dieses Java-Maven-Projekt beinhaltet die JMH-Testumgebung mit allen Experimenten und Benchmarks aus dieser Projektarbeit und der Bachelorarbeit von Marco De Tomasi und Markus Rutz.
Ordner	java-result-parser	Mit diesem Java-Maven-Projekt können JMH-Resultate geparst und in die Datenbank geschrieben werden. Dieses Projekt wurde in der Bachelorarbeit von Marco De Tomasi und Markus Rutz entwickelt. In dieser Projektarbeit wurde es nicht verwendet. Es wird nur der Vollständigkeit halber mitgeliefert.
Ordner	java-test-parameter-parser	Dieses Java-Projekt enthält zwei Parser, mit welchen die JMH-Messresultate für die weitere Verarbeitung in MATLAB vorbereitet werden können.
Ordner	libs	Beinhaltet die ausführbaren jar-Files der Java-Maven-Projekte.
Ordner	sql	Beinhaltet die SQL-Skripts für das Erstellen oder Wiederherstellen der Datenbank.
Ordner	tmp	Hier befinden sich die Ergebnisse aus den Testläufen aus der Bachelorarbeit von Marco De Tomasi und Markus Rutz. Sie sind nur der Vollständigkeit halber mitgeliefert.
Ordner	AnalyseIterationen	Beinhaltet alle Messresultate aus den Analyseiterationen, welche zur Bestimmung der JMH-Optionen verwendet wurden.
Ordner	Bytecode	Enthält den extrahierten Bytecode aus dem Class-File `ArrayVsArrayListAccessTest.class`
Ordner	Experimente	Beinhaltet alle Messresultate und Diagramme, welche in dieser Arbeit verwendet wurden.

Ordner	MATLAB_createGraphs	Mit diesem MATLAB-Skript wurden aus den Messresultaten Stem-Plots und Histogramme erstellt.
Ordner	NanoTime	Beinhaltet alle Messresultate und Diagramme zum NanoTime-Experiment.
Ordner	SafeLooping	Beinhaltet alle Messresultate zum Schleifen-Experiment (Absatz 3.5)
Excel-Datei	Auswertung der Resultate.xlsx	Diese Datei enthält Auswertungen aller Experimente.
PDF-Datei	PAIT16_Best-Practices_fuer_performante_Java-Programmierung_Georgiou_Stocker.pdf	Projektarbeit im PDF-Format