Stefan Rode

Einfluss von Java Code Optimierungsverfahren auf die Energieeffizienz

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

Betreuender Prüfer : Prof. Dr. Bernd Kahlbrandt
Zweitgutachter : Prof. Dr. -Ing. Martin Hübner

Abgegeben am 18. September 2009

Stefan Rode

Thema der Bachelorarbeit

Einfluss von Java Code Optimierungsverfahren auf die Energieeffizienz

Stichworte

Java Optimierung Profiling Energieeffizienz

Kurzzusammenfassung

In dieser Arbeit werden Java Code Optimierungsverfahren hinsichtlich ihres Einfluss auf die Energieeffizienz untersucht. Dazu werden beispielhafte Programme entwickelt, die in Szenarien mit Java Profiling-Techniken zur Laufzeit analysiert werden und deren Stromverbrauch gemessen wird. Aus den Ergebnissen der Analyse und des Stromverbrauchs wird ein Wert für Energieeffizienz der Programme ermittelt und ein Vergleich zwischen verschiedenen Implementationsvarianten durchgeführt. Abschließend findet eine Bewertung der Optimierungsverfahren statt, ob und unter welchen Umständen sie sich eignen und wie sie sich auf die Qualitätskriterien des Softwareengineerings auswirken.

Stefan Rode

Title of the paper

Influence of Java code optimization techniques on energy efficiency

Keywords

Java Optimization Profiling Energy Efficiency

Abstract

In this report, Java code optimization methods are examined in terms of their influence on energy efficiency. For that purpose exemplary programs must be developed, which are analyzed on runtime by Java profiling techniques according to a scenario and which power consumption is measured. From the results of the analysis and power consumption, a value for energy efficiency is identified and a comparison between different implementation variants is carried out. Finally, an evaluation of the optimization procedure takes place, whether and under what circumstances they are qualified and how they affect the criteria of quality of software engineering.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagungen

Zuallererst möchte ich mich bei meinen Eltern bedanken, die mir das Studium ermöglicht haben. Insbesondere meinem Vater sei gedankt, der mich in allen Belangen unterstützt hat.
Ein weiterer Dank geht an Herrn Prof. Dr. Bernd Kahlbrandt, meinem Betreuer und Prüfer, für die Unterstützung bei dieser Arbeit, vor allem für seine Anstrengungen bei der Beschaffung eines Messgeräts.
Benjamin Vetter sei hier auch der Dank ausgesprochen, für das schnelle Korrekturlesen meiner Arbeit.

1. Einleitung

In den letzten Jahren gewinnen die Themen Klimawandel und Klimaschutz mehr und mehr an Bedeutung. Während historisch gesehen das Klima der Erde ständigen natürlichen Schwankungen unterworfen ist, die sich über Jahrtausende vollziehen, ist seit Mitte des 18. Jahrhunderts – dem Beginn der Industrialisierung – eine deutliche Veränderung des Weltklimas zu beobachten. Laut aktuellen Berichten des IPCC [1] ist die momentane Klimaerwärmung nicht auf natürliche Ursachen zurückzuführen. Verantwortlich dafür ist vielmehr der Mensch, der durch seinen ständig steigenden Verbrauch von Energie und den daraus resultierenden Folgen das Klima belastet. Die Energie wird vor allem aus fossilen Brennstoffen gewonnen, deren Gewinnung alleine schon einen erheblichen Eingriff in die Natur darstellt. Bei der Erzeugung der Energie werden unter Anderem grosse Mengen an Kohlenstoffdioxid freigesetzt, das in der Erdatmosphäre den natürlichen Treibhauseffekt verstärkt und somit zur Erwärmung des Klimas führt. [2] Des Weiteren gilt zu berücksichtigen, dass die natürlichen Ressourcen, wie Kohle, Erdöl und Erdgas nur begrenzt auf der Erde vorhanden und zugänglich sind, was bedeutet, dass in nicht allzu ferner Zukunft der Energiebedarf nicht mehr allein durch natürliche Ressourcen gedeckt werden kann. Aus diesem Grund gibt es vielfältige Maßnahmen, die auf der Welt ergriffen werden, um sowohl knappe Ressourcen zu schonen, als auch die vom Menschen verursachten Klimaveränderungen aufzuhalten.

In Bereich der Informations- und Datenverarbeitung (IT) hat sich ein Begriff durchgesetzt, der für eine andere Denkweise im Bezug auf die Umwelt steht – Green IT. Green IT befasst sich mit der Frage, wie man die Informations- und Datenverarbeitung energieeffizient, umweltverträglich und nachhaltig gestalten bzw. verändern kann. Zweifelsfrei – das zeigen aktuelle Studien – spielt die IT-Branche eine grosse Rolle im Energieverbrauch in Deutschland. Demnach beträgt der von der Informations- und Kommunikationstechnologie bedingte Stromverbrauch 55,4 TWh (Terawattstunden), was einem Anteil von 10,5 % am gesamtdeutschen Stromverbrauch entspricht. Bis 2020 wird ein Anstieg des Stromverbrauchs um 20 % auf 66,7 TWh prognostiziert, sollten keine Maßnahmen ergriffen werden. [3] Obwohl auf der einen Seite der Energieverbrauch und der damit verbundene Aufwand diese zu beschaffen ständig steigt, ist ein Mangel an Umweltbewusstsein und engagiertem Energiemanagement zu beobachten, da die Server in einem Rechenzentrum durchschnittlich nur zwischen 10 und 50 % ausgelastet sind. Dies wurde in einer Studie mit mehr als 5000 Servern über 6 Monate ermittelt. [4] Dabei müssen natürlich tageszeitabhängige und saisonale Schwankungen im Zugriff auf diese Server berücksichtigt werden.
Das bedeutet also, dass ein enormes Einsparpotential für Energie vorhanden ist. Im Bereich der Softwareentwicklung und Programmierung können unter Anderem effizienter gestaltete Programme dazu beitragen weniger Energie zu verbrauchen.
Wir sollten uns darüber bewusst werden, dass wir nicht nur gegenüber unserer Generation, sondern auch gegenüber nachfolgenden Generationen eine besondere Verantwortung zu tragen haben, denn die Knappheit der Ressourcen und die Veränderung des Klimas wird sich erst in den kommenden Jahren und Jahrzehnten richtig auf unseren Alltag auswirken, wenn jetzt nichts unternommen wird.

Ein weiterer Aspekt, der im Bezug auf Green IT betrachtet werden sollte ist, dass die Energiesparmaßnahmen dazu beitragen die Kosten für die Anschaffung, den Betrieb und die Entsorgung von IT-Systemen zu senken, was vor allem in Hinblick auf die steigenden Rohstoffkosten immer mehr betriebswirtschaftlich und privatwirtschaftlich an Relevanz gewinnt.

1.1. Problemstellung

Im Gegensatz zu Programmiersprachen, wie C oder C++, gehört Java zu einer Familie von Sprachen, die statt direkten Maschinencode einen Zwischencode erzeugt – den Java Bytecode. Damit ein Java-Programm ausgeführt werden kann, muss der Bytecode von einem Interpreter zur Laufzeit interpretiert werden. Das übernimmt eine virtuelle Maschine. Was auf der einen Seite ein großer Vorteil ist, da somit Java-Programme sehr leicht auf andere Plattformen portiert werden können, ohne das das Programm an die Plattform angepasst werden muss, ist auf der anderen Seite auch ein Nachteil, denn es entsteht ein gewisser Overhead, der durch den Vorgang des Interpretierens in der virtuellen Maschine entsteht. Aus diesem Grund sind Java-Programme im direkten Vergleich mit z.B. C-Programmen deutlich langsamer.

Die Hypothese lautet, dass ein Programm, das in einer kürzeren Ausführungszeit zu einem äquivalenten Ergebnis kommt, als ein anderes Programm, weniger Energie verbraucht, da es weniger Zeit der CPU benutzt. Diese Hypothese basiert darauf, dass der Energiebedarf der CPU abhängig davon ist, wieviele Operationen die CPU in einer bestimmten Zeit durchführen muss. Bei geringer Last, wenn kein Programm ausgeführt wird, verbraucht die CPU weniger Energie als bei hoher Last, wenn mehrere Programme ausgeführt werden. Das bedeutet, dass bei einem Programm, dass schneller abläuft, die CPU länger in einem Zustand niedrigerer Last laufen kann oder ein anderes Programm dadurch mehr CPU-Zeit bekommen kann. Die Energieeffizienz ist demnach bei einem Programm höher, dessen Ausführungszeit bei gleichem Stromverbrauch kürzer ist, als bei einem äquivalenten Programm oder bei dem der Stromverbrauch bei gleicher Ausführungszeit niedriger ist.

Für die Zunkunftsfähigkeit von Java Anwendungen in Bezug auf die o.g. Hypothese ist es daher notwendig Quellcode zu schreiben, der die Ausführungszeit verringert und damit die Energieeffizienz steigert.

1.2. Zielsetzung

Im Rahmen dieser Arbeit soll untersucht werden, wie sich Optimierungen des Java Codes auf die Energieeffizienz der Anwendung auswirken und die oben genannte Hypothese soll dadurch belegt werden. Dazu müssen Optimierungsverfahren gefunden und geeignete Testbeispiele und Messszenarien erarbeitet werden. Dann muss eine Methode entwickelt werden, mit der man die Energieeffizenz einer Java Anwendung quantifizieren kann. Dazu ist es notwendig Stromverbrauchsmessungen durchzuführen und Java Profiling einzusetzen. Die Ergebnisse der Messungen sollen dahingehend beurteilt werden, inwiefern sich das jeweilige

Optimierungsverfahren auf die Energieeffizienz der Anwendung auswirkt und wie sinnvoll es ist dieses Verfahren einzusetzen. Abschliessend soll eine Bewertung der Optimierungsverfahren hinsichtlich der Qualitätskriterien des Softwareengineerings erfolgen.

Die durch diese Arbeit gewonnenen Erkenntnisse sollen Einsparpotentiale für den Stromverbrauch von Rechnern aufzeigen und damit langfristig einen Beitrag dazu leisten den Stromverbrauch zu senken. Grundsätzlich sind mit Rechnern nicht nur standortgebundene Geräte gemeint, die über eine Stromleitung mit Strom versorgt werden und damit eine unbegrenzte Laufzeit haben, sondern auch die mobilen Geräte, die grösstenteils von Akkus mit Strom versorgt werden und deren Laufzeit aus diesem Grund begrenzt ist. Mobile Geräte finden zunehmend Einzug in unseren Alltag und die Zahl der Anwendungen für diese Geräte wächst ständig. Auch auf diesen Geräten werden Java-Anwendungen eingesetzt, da sie u.a. vergleichsweise einfach auf andere Plattformen zu portieren sind. In diesem Sinne können die gewonnenen Erkenntnisse auch dazu beitragen die Laufzeit von mobilen Geräten zu verlängern. [5]

1.3. Aufbau der Arbeit

Im Kapitel 2 werden Grundlagen geklärt, die zum Verständnis der nachfolgenden Kapitel dienen. Das Kapitel 3 behandelt eine Auswahl von Optimierungsverfahren anhand von Beispielen, stellt Messszenarien auf, stellt die Messergebnisse dar und wertet sie aus. Weiterhin findet in der Auswertung der Messergebnisse und eine Beurteilung der Anwendbarkeit der Optimierungsverfahren statt, sowohl hinsichtlich der Energieeffizienz, als auch hinsichtlich der Qualitätskriterien des Softwareengineerings. Das Kapitel 3 ist damit der Hauptteil der Arbeit. Die Infrastruktur, die für die Messungen nötig ist, wird in Kapitel 4 erklärt. In Kapitel 5 wird ein Resümee gezogen, ob die Zielsetzungen der Arbeit erreicht wurden und welche Erkenntnisse daraus gewonnen werden können. Das Kapitel 6 gibt abschließend einen kurzen Ausblick auf weitere Möglichkeiten, Java Anwendungen energieeffizienter zu gestalten.

1.4. Einordnung

Bisherige Arbeiten haben sich vor Allem mit dem Stromverbrauch und der Energieeffizienz der Java VM beschäftigt. [6,7]
Untersuchungen auf die Energieeffizienz von Java Anwendungen und konkrete Messungen hat es laut den Recherchen im Rahmen dieser Arbeit bisher nicht gegeben.

2. Grundlagen

In diesem Abschnitt werden u.a. Begriffe und Definitionen, die in der Arbeit verwendet werden und zum Allgemeinwissen zählen, geklärt.

2.1. Definitionen / verwendete Begriffe

2.1.1. Energieeffizienz

Im Allgemeinen versteht man unter dem Begriff Energieeffizienz, dass ein bestimmter Nutzen unter Einsatz von möglichst wenig Energie erreicht wird oder dass maximaler Nutzen aus einer bestimmten Menge von Energie gezogen wird (ökonomisches Prinzip). [8]
In dieser Arbeit soll die Energieeffizienz von Java Methoden betrachtet werden. Eine Recherche nach einer Möglichkeit, die Energieeffizienz von Methoden zu quantifizieren, führte zu keinem verwendbaren Ergebnis.
Aus diesem Grund wird im Weiteren die folgende vereinfachte Formel für den durchschnittlichen Stromverbrauch (s. elektrische Energie [9]) eines Methodenaufrufs zur Bewertung der Energieeffizienz herangezogen, die darauf basiert, dass zur Quantifizierung der Energieeffizienz die Laufzeit einer Methode und der Stromverbrauch für einen bestimmten Zeitraum in Relation zueinander gesetzt werden müssen. Der durchschnittliche Stromverbrauch E_D in Wh berechnet sich aus:

$$E_D = \frac{E_S * n}{T_S * 60 * 1000000}$$

wobei E_S die elektrische Energie in Wh bezeichnet, die während des Szenarios verbraucht wird.T_S ist die Gesamtlaufzeit in µs und n die Anzahl der Methodenaufrufe in dem Szenario.

2.1.2. Durchschnittliche Laufzeit einer Methode

Die durchschnittliche Laufzeit T_D einer Methode ist das arithmetische Mittel aus der Gesamtlaufzeit T_G der Methodenaufrufe und der Anzahl der Methodenaufrufe n. Die Formel lautet:

$$T_D = \frac{T_G}{n}$$

2.2. Java VM

Die Grundlage für die Ausführbarkeit einer jeden Java Anwendung ist eine Java VM. Viele Optimierungen auf Quellcodeebene beschäftigen sich damit, wie man sich Eigenheiten in der Spezifikation der Java VM für die Verbesserung der Laufzeit der Anwendungn zu Nutze machen kann.

2.2.1. Methodenaufrufe

Es gibt grundsätzlich 2 Arten von Methoden, die Klassenmethoden und die Instanzmethoden. Der Unterschied zwischen den beiden Arten ist, dass Instanzmethoden auf einem Objekt aufgerufen werden, die Klassenmethoden nicht. Wird eine Methode aufgerufen, wird ersteinmal ein sog. Stack-Frame erzeugt, der Speicherplatz für ein Array von lokalen Variablen der Methode, einen Operanden Stack und eine Referenz zum Laufzeit Konstanten Pool bietet und auf den VM Stack gelegt. Im Falle einer Instanzmethode wird eine Referenz auf das Objekt (this), auf dem die Methode ausgeführt wird und die Parameter vom Operanden Stack der aufrufenden Methode als lokale Variablen auf den Operanden Stack abgelegt. Bei einer Klassenmethode werden nur die Parameter vom Operanden Stack der aufrufenden Methode auf den Operanden Stack abgelegt. Als nächstes werden die Objektreferenz und die Parameter im Array der lokalen Variablen gespeichert und der Program Counter (PC) wird auf die erste Anweisung in der Methode gesetzt. Nach dem Ausführen der Anweisungen innerhalb der Methode, wird das Resultat, wenn es eins gibt, vom Operanden Stack der Methode genommen und auf den Operanden Stack der aufrufenden Methode gelegt. Dann wird der Stack-Frame der Methode vom VM Stack genommen und der PC wird auf die Anweisung, die dem Methodenaufruf folgt gesetzt.

Aufgrund dieses umfangreichen Ablaufs sind Methodenaufrufe gegenüber anderen Anweisungen sehr zeitaufwändig. [10,11]

2.3. Profiling von Java Anwendungen

Unter Profiling versteht man das Analysieren des Laufzeitverhaltens einer Anwendung. Dabei können verschiedene Kenngrößen des Programms, wie z.B. die Ausführungszeit, die Anzahl der Methodenaufrufe, die Auslastung des Heap-Speichers, Informationen über Threads und ihre Zustände, die Objekte im System, uvm., während der Laufzeit gemessen und in anschaulicher Form dargestellt werden. Die Ergebnisse der Analyse können helfen, Stellen im Quellcode zu finden, die die Anforderungen an die Performanz der Anwendung gefährden oder beeinträchtigen.

Die Laufzeit einer Methode kann man einfach ermitteln, indem man vor und nach dem Methodenaufruf z.B. die aktuelle Systemzeit mit der System.currentTimeMillis()-Methode abruft und die Differenz aus beiden Werten berechnet. Die Anzahl der Aufrufe einer Methode könnte man mit einer int-Variable zählen, die bei jedem Methodenaufruf inkrementiert wird. Will man eine komplette Anwendung analysieren, würde der Vorgang alle Methoden auf diese Weise zu untersuchen wahrscheinlich die Entwicklungszeit der Anwendung bei Weitem übertreffen.
Die Firma Sun Microsystems Inc. [12] hat deshalb mit dem Java Virtual Machine Profiler Interface (JVMPI) eine Schnittstelle entwickelt, die sich zwar seit der Java Version 2 im experimentellen Zustand befindet, mit der es aber ermöglicht wird, Laufzeitinformationen einer Anwendungen direkt von der VM bereitgestellt zu bekommen.

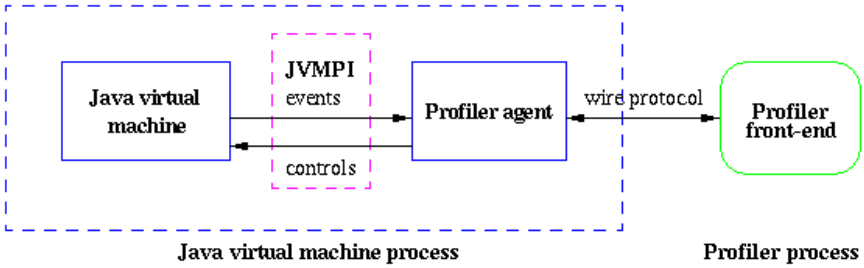

Abb. 2.1: Aufbau und Funktionsweise der JVMPI-Schnittstelle [13]

In der Abb. 2.1 wird der Aufbau und die Funtionsweise der JVMP-Schnittstelle dargestellt. Ein sog. Profiler Agent, der sich innerhalb des VM-Prozess befindet, stellt Anfragen an die VM und wird beim Eintreten von vorher festgelegten Events, bspw. dass eine Methode aufgerufen, ein Objekt erzeugt oder der Garbage Collector aufgerufen wurde, von der VM benachrichtigt. Über ein bisher noch nicht standartisiertes Protokoll, kann eine Profiling Anwendung die Informationen, die der Profiler Agent gesammelt hat, abrufen. [13]

3. Optimierungsverfahren

Optimierungsverfahren kann man je nach der erwarteten Wirkung in unterschiedliche Kategorien einteilen: Laufzeit, Codegröße, Speicherplatzbedarf, usw. Dabei kann man ein Optimierungsverfahren nicht pauschal zu einer Kategorie gehörend abgrenzen, denn die Optimierung kann auch kategorieübergreifend wirken. Zum Beispiel kann durch das verzögerte Erzeugen eines Objekts sowohl die Zeit bis zum Erzeugen des Objektes verschoben und evtl. eingespart werden, andererseits wirkt es sich auch positiv auf den Speicherplatzbedarf der Anwendung aus, wenn das Objekt erst dann erzeugt wird, wenn es tatsächlich benötigt wird.
Diese Arbeit konzentriert sich allein auf die Auswirkungen eines Optimierungsverfahrens auf die Laufzeit. Daher werden bspw. Auswirkungen auf den Speicherplatzbedarf nicht näher betrachtet.

Die im Folgenden genannten und beispielhaft erläuterten Optimierungsverfahren stellen jedoch nur eine Teilmenge der existierenden Optimierungsverfahren für Java Anwendungen dar und dienen dazu die aufgestellte Hypothese zu belegen.
Zu berücksichtigen gilt dabei, dass die Auswirkungen der Optimierungen u.a. abhängig von der verwendeten Hardware, dem Betriebssystem und der eingesetzten Java VM sind und aus diesem Grund je nach Konfiguration unterschiedlich starke Ausprägungen haben können.

Um zu entscheiden, ob ein Optimierungsverfahren in jedem Fall sinnvoll ist, muss man verschiedene Aspekte berücksichtigen. Dazu gehört natürlich in erster Linie, ob das Optimierungsverfahren ein besseres Ergebnis liefert, als die Ausgangsbasis. Auch kleine Unterschiede in der Energieeffizienz zweier Methoden können sich für die Energieeffizienz der Anwendung lohnen, wenn die Häufigkeit, mit der die Methode in der Anwendung aufgerufen werden sehr hoch ist. Auf der anderen Seite lohnt sich eine Optimierung einer

Methode nicht, wenn sie möglicherweise um ein Vielfaches schneller und energieeffizienter als eine andere Methode ist, aber sie nur sehr selten aufgerufen wird und sich die Energieeinsparungen für die gesamte Anwendung kaum bemerkbar machen. Weiterhin muss man bedenken, das das Entwickeln von optimalem Quellcode Zeit und daraus resultierend Geld kostet. Wenn der Entwicklungsaufwand, also der Aufwand, den ein Softwareentwickler oder Programmierer dafür benötigt eine Optimierung für ein Programm zu planen, zu programmieren und zu testen größer ist, als die erwarteten Energieeinsparungen, ist es zumindest betriebswirtschaftlich nicht sinnvoll die Optimierung anzuwenden. Hinzu kommt, dass durch das Optimieren des Codes, dieser möglicherweise schlechter lesbar und nicht mehr so einfach zu verstehen ist, was dazu führt, dass der Wartungsaufwand für dieses Programm steigt.

Auch wenn ein Optimierungsverfahren bewirkt, dass ein Programm schneller ausgeführt wird und es damit hinsichtlich des Qualitätskriteriums Performanz besser ist, kann es doch dazu führen das das Programm sich hinsichtlich anderer Qualitätskriterien verschlechtert. Angenommen, beim Optimieren des Programms werden 2 Methoden, die unabhängige Aufgaben erfüllen, zu einer speziellen, umfangreicheren Methode zusammengefasst, um den Overhead der Methodenaufrufe von 2 Methoden zu senken. Ein Aufruf der neuen Methode ist um einen Faktor n schneller, als der vorherige Aufruf der 2 Methoden.
Das Optimieren hat dann folgendes bewirkt:

- Das Programm ist schneller als vorher und hat damit das Qualitätskriterium Performanz verbessert.
- Durch das Zusammenführen der 2 Methoden zu einer speziellen Methode hat sich die Wiederverwendbarkeit verschlechtert, da die neue Methode nur noch in einem bestimmten Kontext verwendet werden kann. Vorher konnten die beiden Methoden unabhängig voneinander verwendet werden.
- Die Wartbarkeit des Programms hat sich ebenfalls verschlechtert, da eine umfangreichere Methode schwerer zu lesen und zu verstehen ist, als zwei kürzere Methoden.

Es muss also für jede Anwendung und jedes Optimierungsverfahren individuell abgewägt werden, ob die Vorteile, die sich aus dem Geschwindigkeitsgewinn ergeben, die Nachteile überwiegen.

3.1. Logische Operatoren

Die bedingten Operatoren && und || sollten in logischen Ausdrücken anstatt der Operatoren & und | verwendet werden. In logischen Ausdrücken werden bei Verwendung der Operatoren & und | immer beide Operanden ausgewertet. Das ist jedoch nicht immer notwendig. Beim Operator && wird der zweite Operand nur ausgewertet, wenn der erste Operand *true* ist. Bei der Verwendung von || wird der zweite Operand nur ausgewertet, wenn der erste Operand *false* ist. Sinnvoll ist diese Regel vor allem dann, wenn der zweite Operand einen Methodenaufruf mit einer aufwändigen Berechnung erfordert. Weiterhin sollte man die Operanden nach der Eintrittswahrscheinlichkeit anordnen. Bei einer ODER-Verknüpfung, ist

es am effektivsten, wenn der Operand mit der höchsten Eintrittswahrscheinlichkeit als erster ausgewertet wird, da hierdurch in den meisten Fällen der zweite Operand nicht ausgewertet werden muss. Im Gegensatz dazu sollte bei einer UND-Verknüpfung der erste Operand die niedrigere Eintrittswahrscheinlichkeit besitzen, damit der zweite Operand in den meisten Fällen nicht ausgewertet werden muss. [14,15]

Code-Beispiel:

```
public class NameService {

	private Map<String,String> names = new HashMap<String,String>();

	public NameService(Map<String,String> map) {
		names.putAll(map);
	}

	public String lookup1(String name) {
		if(name != null & !"".equals(name) & names.containsKey(name))
			return names.get(name);
		return "";
	}

	public String lookup2(String name) {
		if(name != null && !"".equals(name) && names.containsKey(name))
			return names.get(name);
		return "";
	}
}
```

Es gibt eine Klasse NameService, die eine Map *names* mit Namen und IP-Adressen jeweils als Strings speichert. Die IP-Adresse zu einem bestimmten Namen erhält man, wenn man eine *lookup*-Methode aufruft und den Namen als String *name* übergibt. Ist der übergebene String *null*, ein leerer String oder ist der Name nicht in der Map *names* vorhanden, wird ein leerer String zurückgegeben und andernfalls die zum Namen zugehörige IP-Adresse. Die Methode *lookup1()* verwendet für das Prüfen der Bedingungen den einfachen &-Operator, während die Methode *lookup2()* den &&-Operator verwendet.
Sollte der übergebene String *null* sein, müsste in der *lookup1()*-Methode, obwohl die Bedingung für die if-Anweisung in diesem Fall gar nicht mehr erfüllt werden kann, unnötigerweise geprüft werden, ob der String *name* ein leerer String ist und ob *name* in der Map *names* enthalten ist. Auch wenn *name* ein leerer String ist, wird in dieser Methode das Vorhandensein in der Map *names* geprüft, was an dieser Stelle keinen Sinn macht.

Szenario:

```
public static void szenario_logOps() {
	Map<String,String> names = new HashMap<String,String>();

	for(int i = 0; i < 256; i++) {
		names.put("haw" + i, "141.22.0." + i);
	}

	NameService ns = new NameService(names);
```

```
                List<String> list = new LinkedList<String>(names.keySet()),
                             list2 = new LinkedList<String>();

                for(int i = 0; i < 1000; i++) {
                        list2.add(list.get(i%256));
                        list2.add(null);
                        list2.add(list.get((i+5)%256));
                        list2.add("");
                        list2.add(list.get((i+10)%256));
                }

                while(true) {
                        for(String key : list2) {
                                ns.lookup1(key);

                                // separate Messung
                                // ns.lookup2(key);
                        }
                }
        }
```

Das Szenario sieht vor, dass die Map eines NameService-Objekts mit 256 Name-IP-Adress-Paaren gefüllt werden soll. Dazu wird eine Map *names* erzeugt, die in einer for-Schleife mit der Methode *put()* gefüllt wird. Dann wird ein NameService-Objekt erzeugt, dem die Map *names* übergeben wird. Weiterhin sollen in dem Szenario 5000 Anfragen mit einer *lookup*-Methode erfolgen. Dazu wird eine List *list*, mit allen Namen, die in der Map *names* enthalten sind, erzeugt. Eine weitere List *list2* wird erzeugt, die 5000 Strings für die 5000 Anfragen speichern soll. In einer for-Schleife werden die 5000 Strings in die List *list2* mit der Methode *add()* hinzugefügt. Davon sind 60 % vorhande Namen aus der List *list*, 20 % sind leere Strings und 20 % sind *null*. In einer Endlosschleife soll dann mit Hilfe einer foreach-Schleife über die List *list2* iteriert werden und die 5000 Anfragen mit der lookup-Methode gemacht werden. Die Methoden *lookup1()* und *lookup2()* werden dann separat in unterschiedlichen Messdurchgängen eingesetzt. Die Messdurchgänge dauern jeweils 10 Minuten.

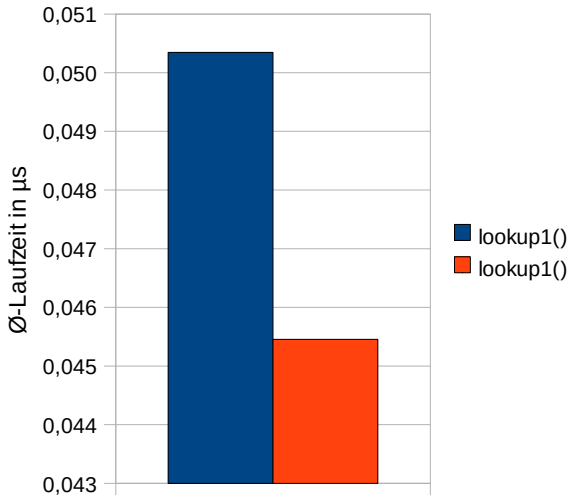

Abb. 3.1: Ø-Laufzeit der lookup-Methoden

Die Messung zeigt, wie in Abb. 3.2 und Abb. 3.1 zu sehen ist, dass die *lookup2()*-Methode 9,7 % schneller ist, als die *lookup1()*-Methode und die Energieeffizienz durch den geringeren durchschnittlichen Stromverbrauch um 9,4 % höher ist.

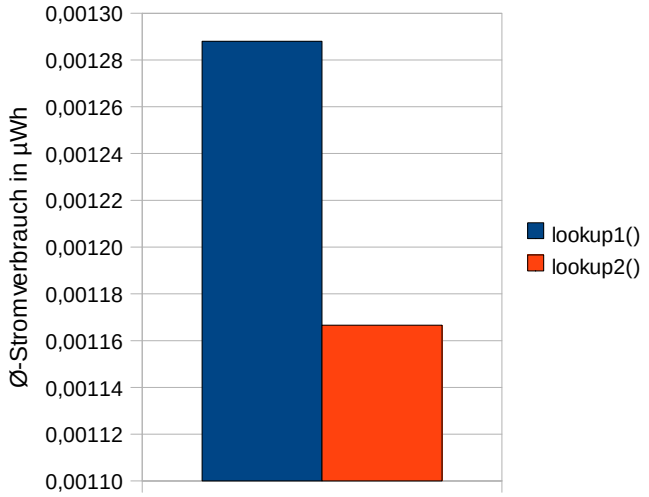

Abb. 3.2: Ø-Stromverbrauch der lookup-Methoden

Da selbst in dem Fall, dass alle Operanden ausgewertet werden müssen, kein Nachteil gegenüber den einfachen logischen Operatoren & und | entsteht, ist es sinnvoll die beiden bedingten Operatoren && und || einzusetzen. Man muss allerdings beachten, dass Zuweisungen und Methodenaufrufe als Operand eines logischen Ausdrucks nicht ausgewertet werden, wenn die Bedingung des logischen Ausdrucks durch die in der Auswertungsreihenfolge voranstehenden Operanden nicht erfüllt ist.
Bis auf das Kriterium Performanz wird die Qualität durch diese Optimierung nicht beeinflusst.

3.2. Objekterzeugung

Die Erzeugung und Initialisierung von Objekten kann unter Umständen viel Zeit kosten, z.B. dann, wenn das Objekt mehrere aggregierte oder komponierte Objekte in seinen Instanzvariablen speichert, die erst erzeugt werden, wenn das Objekt selbst initialisiert wird. Die aggregierten und komponierten Objekte können dann wiederum Aggregations- oder Kompositionsbeziehungen zu anderen Objekten haben. Für die Objekterzeugung muss Speicher auf dem Heap allokiert werden und das erzeugte Objekt muss initialisiert werden. Sind sehr viele Objekte im System und übersteigt der belegte Speicher des Heap regelmäßig die festgelegte Kapazität des Heap, muss der Garbage Collector in kurzen Zeitabständen ausgeführt werden, um wieder freien Speicher auf dem Heap für neue Objekte zu erhalten. Abhängig vom Thread-Scheduling des Betriebssystems und des verwendeten Prozessors, kann der Garbage-Collector die Laufzeit des Programms negativ beeinträchtigen. Gerade bei

Objekten, die eine kurze Lebenszeit haben, d.h. relativ kurz nach der Erzeugung nicht mehr benötigt werden und keine Objektreferenzen mehr auf diese Objekte existieren, sollte geprüft werden, ob es überhaupt notwendig ist diese Objekte zu erzeugen. [16,10]

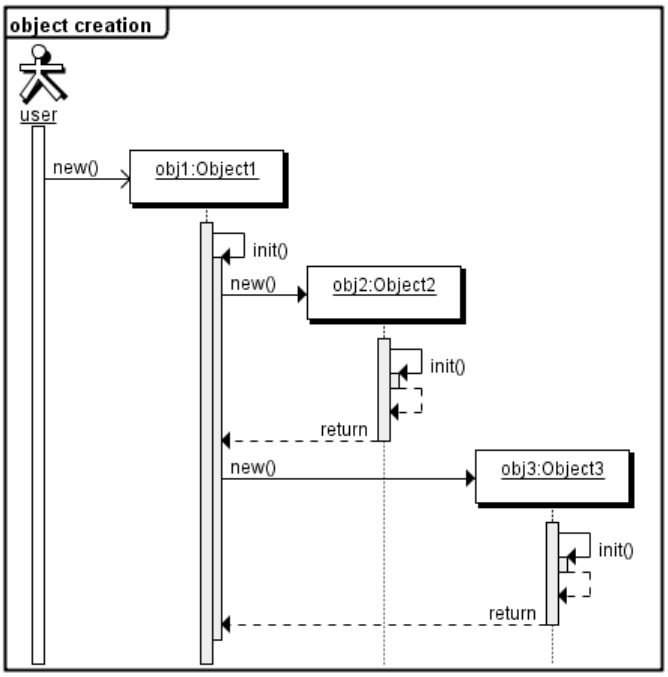

Abb. 3.3: Objekterzeugung und Initialisierung

Die Abb. 3.3 zeigt beispielhaft anhand eines Sequenzdiagramms, wie die Erzeugung und Initialisierung eines Objekts ablaufen könnte. Angenommen es soll ein Objekt *obj1* vom Typ Object1 erzeugt werden. Dieses Objekt hat zwei Instanzvariablen *obj2* und *obj3* vom Typ Object2 und Object3. Nach der Erzeugung des Objekts *obj1* wird dessen *init()*-Methode aufgerufen, in der *obj2* und *obj3* erzeugt werden. Diese beiden Objekte werden ebenfalls initialisiert. Je nachdem in welchen weiteren Aggregations- oder Kompositionsbeziehungen Object2 und Object3 stehen, kann deren Instanziierung weitere Objekterzeugungen nach sich ziehen.

3.2.1. Wiederverwendung von Exception-Objekten

Exception-Objekte zu erzeugen ist vergleichsweise zeitaufwändig, da beim Initialisieren des Objekts der individuelle Stacktrace ermittelt werden muss. Sollte man sich dazu entscheiden in einer Klasse eine oder mehrere Methoden anzubieten, die explizit Exceptions werfen, kann es sinnvoll sein ein Exception-Objekt mehrfach zu verwenden. Hierdurch können die Kosten für die Erzeugung ab der zweiten geworfenen Exception gespart werden, weil nicht jedes Mal ein neues Exception-Objekt erzeugt werden muss. Klar sollte man sich darüber sein, dass dadurch der individuelle Stacktrace, der zum Auslösen der Exception führte, verloren geht.

Denkbar ist diese Optimierung zum Beispiel dann, wenn bereits der Typ der Exception verrät, durch welchen Umstand es zum Abbruch der aufgerufenen Methode gekommen ist und auf einer höheren Abstraktionsebene mit dieser Ausnahme umgegangen werden kann. [16]

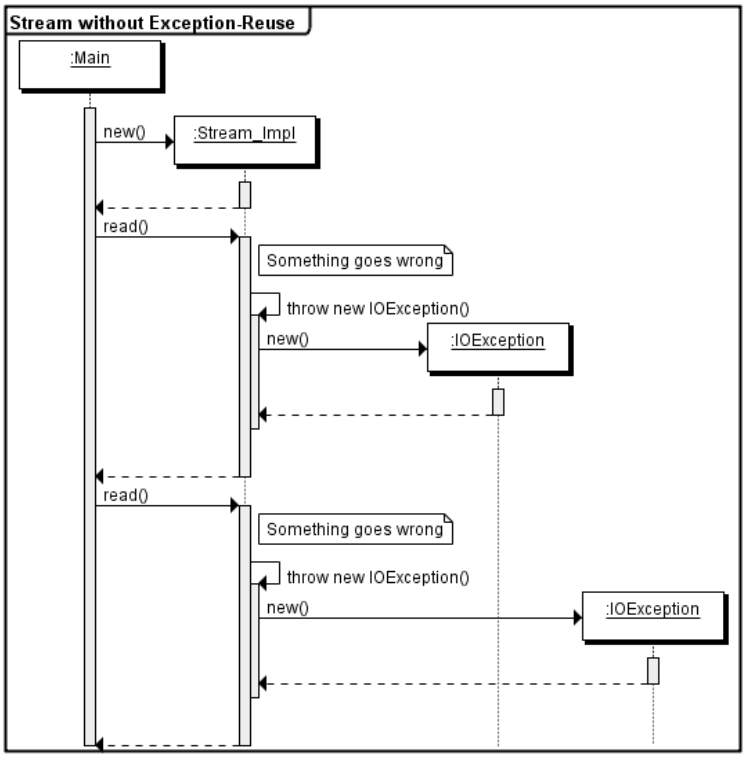

Abb. 3.4: Stream der eine Exception wirft

In der Abb. 3.4 ist ein Sequenzdiagramm zu sehen, das die Verwendung eines Streams zeigt und ohne die Wiederverwendung des Exception-Objekts arbeitet. Tritt beim Aufruf der *read()*-Methode eine Ausnahmesituation ein, wird ein neues IOException-Objekt erzeugt und mit *throw* geworfen.
Abb. 3.5 zeigt hingegen ein Sequenzdiagramm, in dem ein Stream der beim mehrfachem Eintreten einer Ausnahmesituation immer dasselbe Exception-Objekt verwendet. Das Exception-Objekt *e* wird nur einmal bei der Initialisierung des Streams erzeugt.

Code-Beispiel:

```
import java.io.IOException;

public class Stream_Impl {

	private static final IOException ioException = new IOException(
			"hier läuft was schief");

	public String read1() throws IOException {
		//something goes wrong
		throw new IOException("hier läuft was schief");
	}
```

```
        public String read2() throws IOException {
                //something goes wrong
                throw ioException;
        }
}
```

Die Methode *read1()* erzeugt bei jedem Aufruf eine neue IOException, während in der Methode *read2()* bei jedem Aufruf das Exception-Objekt *ioException* der Klasse Stream_Impl geworfen wird.

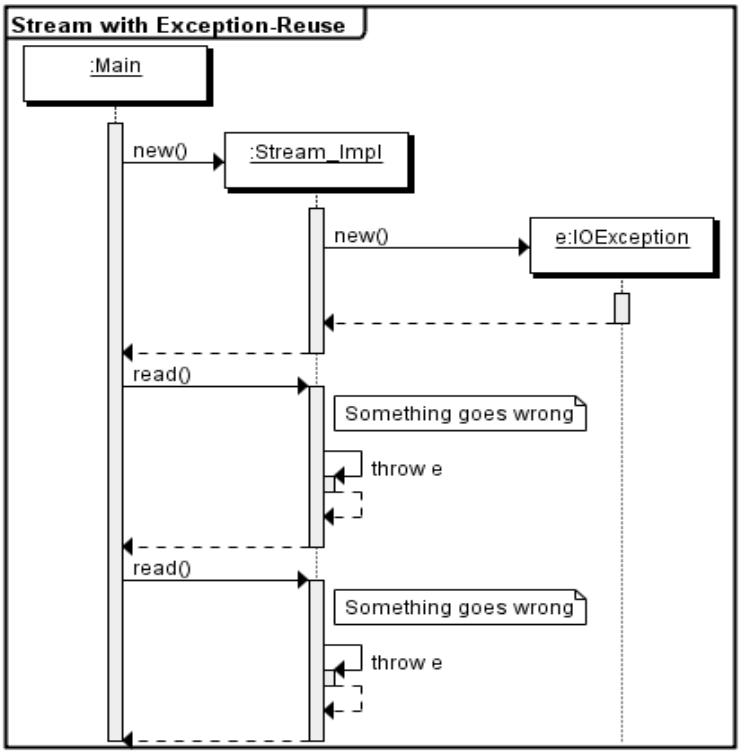

Abb. 3.5: Stream mit Wiederverwendung des Exception-Objekts

Szenario:

In dem Szenario soll zuerst ein Stream_Impl-Objekt erzeugt werden, wie es z.B. beim Lesen oder Schreiben einer Datei geschieht. Danach wird in einer Endlosschleife eine *read*-Methode aufgerufen, zum Lesen der Datei. Wann eine Exception in einer *read*-Methode geworfen wird, soll durch eine Zählervariable *counter* vom Typ int festgelegt werden, die beim Erzeugen des Stream_Impl-Objekts mit dem im Konstruktor übergebenen Wert *c* initialisiert und bei jedem Aufruf der Methode *read* runtergezählt wird. Ist der Zähler gleich 0, wird die Exception geworfen. Als Werte für *counter* sollen 50, 100, 500 und 1000 genommen werden. Dadurch lässt sich vergleichen, wieviele Exceptions geworfen werden müssen, damit sich diese Optimierung lohnt. Für das Szenario muss das o.g. Code-Beispiel etwas angepasst werden:

```
public class Stream_Impl {

        private static final IOException ioException = new IOException(
```

```
                    "hier läuft was schief");

        private int counter = 0, index = 0;

        public Stream_Impl(int c) {
            counter = c;
            index = c;
        }

        public String read1() throws IOException {
            if (index == 0){
                index = counter;
                throw new IOException("hier läuft was schief");
            }
            index--;
            return "";
        }

        public String read2() throws IOException {
            if (index == 0){
                index = counter;
                throw ioException;
            }
            index--;
            return "";
        }
}
```

Die Klasse wird durch zwei Instanzvariablen *counter* und *index* ergänzt, die für das Zählen benutzt werden. In die *read*-Methoden wird das Prüfen, das Dekrementieren und das Zurücksetzen des Zählers *index* hinzugefügt.
Der Code für das Szenario sieht dann folgendermaßen aus:

```
public static void szenario_exception() {
        int counter = 50;

        Stream_Impl stream = new Stream_Impl(counter);
        while(true) {
            try {
                stream.read1();
            } catch (IOException e) {}

            // separate Messung
            // try {
            //     stream.read2();
            // } catch (IOException e) {}
        }
}
```

Die Methoden *read1()* und *read2()* werden separat in verschiedenen Messdurchgängen gemessen. Ein Messdurchgang des Szenarios soll 10 Minuten dauern.

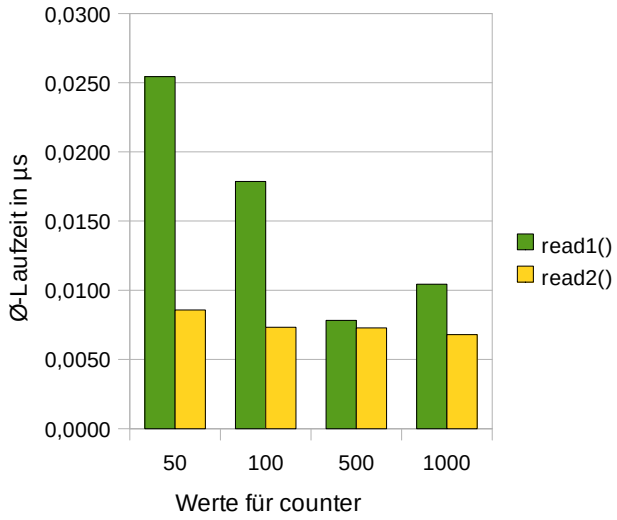

Abb. 3.6: Ø-Laufzeit der read-Methoden

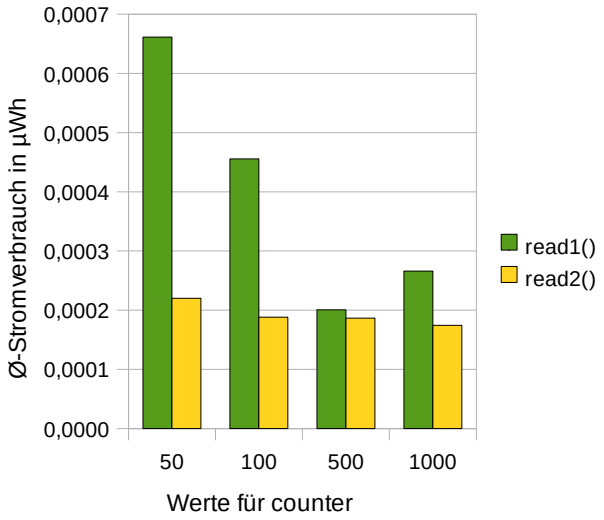

Abb. 3.7: Ø-Stromverbrauch der read-Methoden

Festgestellt werden kann aufgrund der Messungen, dass die durchschnittliche Laufzeit und der Stromverbrauch durch die Häufigkeit, mit der eine Exception in einer der *read*-Methoden geworfen wird, beeinflusst wird. Je seltener es zum Auslösen einer Exception kommt, desto schneller ist die Methode und desto weniger Strom wird durchschnittlich verbraucht. Ein deutlicher Unterschied zwischen den *read*-Methoden ist für Werte von *counter* bis ca. 100 zu erkennen. Die *read1()*-Methode ist bei einem *counter*-Wert von 50 um den Faktor 3 langsamer und energieineffizienter, als die *read2()*-Methode. Bei einem Wert von *counter* gleich 100 ist immerhin noch ein Unterschied von einem Faktor 2,4 zu erkennen. Je seltener eine Exception geworfen wird, desto mehr nähern sich die durchschnittliche Laufzeit und der Stromverbrauch der beiden *read*-Methoden einander an.

In diesem Beispiel lohnt sich also die Wiederverwendung der Exception-Objekte, wenn relativ häufig Exceptions geworfen werden.

Gegen diese Optimierung ist mit Blick auf die Qualitätskriterien nichts einzuwenden, solange beim Abfangen der Exception nicht ausgewertet werden soll, wie es zu der Ausnahmesituation gekommen ist, sondern nur festgestellt werden soll, dass es zu einer bestimmten Ausnahmesituation gekommen ist. Das Weiterleiten einer Exception durch einen sogenannten rethrow ist weiterhin möglich. Seiteneffekte könnten jedoch auftreten, wenn mehrere Objekte das Exception-Objekt verwenden und jeweils nachträglich der aktuelle Stacktrace durch ein *fillInStacktrace()* erzeugt werden soll. Dadurch wird die Wartbarkeit der Anwendung verschlechtert, da es eine potentielle Quelle für Fehler ist, die für einen Anwendungsprogrammierer nicht unbedingt offensichtlich ist und Wissen über die Methode, die die Exception wirft voraussetzt. Geht man von gut dokumentiertem Quellcode aus, ist das Risiko für einen Programmfehler an dieser Stelle relativ gering.

3.2.2. Collection-Objekte und ihre initiale Kapazität

Man sollte soweit die zu verwendende Collection-Klasse es anbietet, dem zu erzeugenden Collection-Objekt im Konstruktor seine initiale Kapazität übergeben, die so groß sein soll, wie es im Programm nötig ist. Dabei ist es besser die initiale Kapazität größer als nötig zu wählen, statt zu klein. Grund dafür ist, dass einige Collection-Implementationen aus dem java.util-Package, wie ArrayList und Vector intern eine Datenstruktur mit einer festen Größe, z.B. ein Array, zur Speicherung der Elemente benutzen. Wenn das interne Array seine Kapazitätsgrenze erreicht und ein weiteres Element hinzugefügt werden soll, muss ein größeres Array erzeugt werden und eine aufwändige Umkopieroperation ist notwendig. Das interne Array einer ArrayList hat standardmäßig eine Kapazität von 10 Elementen. Beim Erreichen der Kapazitätsgrenze k, wächst die Kapazität des internen Arrays mit folgender Formel:

$$k_i = \left\lfloor \frac{k_{i-1} * 3}{2} + 1 \right\rfloor$$

Ein Einblick in den Quellcode der *add()*- und *ensureCapacity()*-Methode der Klasse ArrayList beweist dies:

```
public class ArrayList<E> extends AbstractList<E>
  implements List<E>, RandomAccess, Cloneable, Serializable
{
  private static final long serialVersionUID = 8683452581122892189L;
  private transient Object[] elementData;
  private int size;

public boolean add(E paramE)
  {
    ensureCapacity(this.size + 1);
    this.elementData[(this.size++)] = paramE;
    return true;
  }

public void ensureCapacity(int paramInt)
  {
    this.modCount += 1;
    int i = this.elementData.length;
    if (paramInt > i)
    {
      Object[] arrayOfObject = this.elementData;
      int j = i * 3 / 2 + 1;
      if (j < paramInt)
        j = paramInt;
      this.elementData = Arrays.copyOf(this.elementData, j);
    }
  }
...
}
```

Die Methode *ensureCapacity()* wird hier in der *add()*-Methode mit *this.size + 1* aufgerufen, was zur Folge hat, dass die Kapazität des internen Arrays durch die o.g. Formel erhöht wird, wenn das Hinzufügen eines weiteren Elements die Kapazität des internen Arrays überschreiten würde. Das bedeutet, dass die Kapazität des internen Arrays beim ständigen Hinzufügen von Elementen nicht um eine konstante Anzahl von Elementen steigt, sondern die Anzahl der zusätzlich möglichen Elemente linear steigt. Weiterhin bedeutet dies, dass Umkopieroperationen bei einer steigenden Anzahl von Elementen und der nötigwerdenden Vergrößerung des internen Arrays immer seltener werden. [16,17]
Solange also die Anzahl der hinzuzufügenden Elemente n kleiner gleich der initialen Kapazität ist, sind die Laufzeitkosten für eine Hinzufügeoperation konstant. Danach erfolgt das Hinzufügen eines weiteren Elements bei linear ansteigenden Laufzeitkosten.

Code-Beispiel:

```
public static void initialCapacity1(int n) {
	List<String> list = new ArrayList<String>();
	for(int i = 0; i < n; i++) {
		list.add("abc");
	}
}

public static void initialCapacity2(int n) {
	List<String> list = new ArrayList<String>(n);
	for(int i = 0; i < n; i++) {
		list.add("abc");
	}
}
```

Beide Methoden haben einen Übergabeparameter *n* vom Typ int, der bestimmt, wieviele Elemente in die Liste eingefügt werden sollen. Während in der Methode *initialCapacity1()* eine ArrayList mit der standardmäßigen initialen Kapazität von 10 erzeugt wird, wird in der Methode *initialCapacity2()* eine ArrayList mit der initialen Kapazität von *n* Elementen erzeugt. Danach wird jeweils *n*-mal ein String hinzugefügt. Der Einfachheit halber ist der String in diesem Beispiel "*abc*".

Szenario:

```
public static void szenario_ArrayListCreation() {
	int n = 50;

	while(true) {
		CodeOptimizations.initialCapacity1(n);

		// separate Messung
		// CodeOptimizations.initialCapacity2(n);
	}
}
```

Das Szenario sieht vor, dass die o.g. Methoden in einer Endlosschleife mit einer variablen Anzahl von *n* Elementen getestet werden sollen. Die Anzahl *n* soll in verschiedenen Messdurchläufen jeweils 20, 50, 100, 1000 und 10000 Elemente betragen. Dadurch soll eine

ungefähre Einschätzung möglich sein, ab wievielen Elementen es sich lohnt die initiale Kapazität im Voraus festzulegen, wenn die Liste bis zu ihrer Kapazitätsgrenze gefüllt wird. In weiteren Messungen soll ermittelt werden, wie die Laufzeit und die Energieeffizienz beeinflusst wird, wenn die Liste nur bis zu einem bestimmten Teil der Kapazitätsgrenze gefüllt wird. Ausserdem soll gemessen werden, wieviel die Objekterzeugung in Abhängigkeit von der Anzahl der Elemente n kostet. Ein Messdurchgang wird auf 20 Minuten festgelegt.

Elemente n	**20**	**50**	**100**	**1.000**	**10.000**
initialCapacity1() Ø-Laufzeit in µs	0,43440	1,04049	2,11238	23,17610	431,94734
initialCapacity2() Ø-Laufzeit in µs	0,3648	0,8543	1,6834	16,7602	235,7588
initialCapacity1() Ø-Stromverbrauch in µWh	0,01115	0,02671	0,05404	0,59485	11,05065
initialCapacity2() Ø-Stromverbrauch in µWh	0,00936	0,02186	0,04307	0,42878	6,03150

Tabelle 3.1: Messergebnisse des Szenarios Collections und ihre initiale Kapazität Teil 1

Die Messungen (siehe Tabelle 3.1) beim Füllen der Liste bis zur vorgesehenen Kapazitätsgrenze *n* zeigen, dass selbst bei kleiner Anzahl *n* von Elementen ein deutlicher Unterschied zwischen den beiden Methoden *initialCapacity1()* und *initialCapacity2()* besteht. Bei 20 Elementen sind die durchschnittliche Laufzeit und der Stromverbrauch der Methode *initialCapacity2()* um 16 % kleiner. Bei einer größeren Anzahl von Elementen wird der Unterschied noch deutlicher. Bei 100 Elementen beträgt der Unterschied in der Laufzeit und beim Stromverbrauch 20,3 %. Bei 10.000 Elementen ist der Unterschied schon auf 45,4 % zugunsten der *initialCapacity2()*-Methode gewachsen.

Füllgrad	**40,0%**	**50,0%**	**66,6%**	**80,0%**
initialCapacity1() Ø-Laufzeit in µs	21,9059	18,8478	35,0355	34,9765
initialCapacity2() Ø-Laufzeit in µs	28,4275	27,6666	23,5471	25,8153
initialCapacity1() Ø-Stromverbrauch in µWh	0,5623	0,4869	0,8992	0,8919
initialCapacity2() Ø-Stromverbrauch in µWh	0,7296	0,7147	0,6044	0,6583

Tabelle 3.2 Messergebnisse des Szenarios Collections und ihre initiale Kapazität Teil 2

Der Tabelle 3.2 kann man entnehmen, dass die Methode *initialCapacity1()* bei einem Füllgrad der Liste von 40 % um ca. einen Faktor 1,2 schneller und energieeffizienter ist, als die *initialCapacity2()*-Methode. Bei einem Füllgrad der Liste von 50 % ist der Faktor gegenüber der *initialCapacity2()*-Methode bei ca. 1,4. Ab einem Füllgrad von 66,6 % erweist sich aber die *initialCapacity1()*-Methode als langsamer und energieineffizienter, als die *initialCapacity2()*-Methode. Der Unterschied liegt hier bei einem Faktor 1,4 und bei einem Füllgrad von 80 % auch bei einem Faktor von 1,4, zugunsten der *initialCapacity2()*-Methode.

Kapazität n	20	50	100	1.000	10.000
Objekterzeugung initialCapacity1() Ø-Laufzeit in µs	0,0176	0,0237	0,0132	0,0340	0,2455
Objekterzeugung initialCapacity2() Ø-Laufzeit in µs	0,0592	0,0905	0,2009	11,9985	144,7797

Tabelle 3.3 Messergebnisse der Kosten für die Erzeugung von ArrayList-Objekten

Die Objekterzeugungskosten, die in der Tabelle 3.3 dargestellt sind, zeigen dass die mit einer internen Kapazität *n* initialisierten ArrayList-Objekte in der Methode *initialCapacity2()* deutlich mehr Zeit für die Erzeugung benötigen, als die ArrayList-Objekte in der Methode *initialCapacity1()*. Bei steigender Kapazität *n* wird der Unterschied immer größer. Während bei einer Kapazität *n* von 50 Elementen das ArrayList-Objekt in der Methode *initialCapacity1()* noch mit einem Faktor 3,8 schneller erzeugt werden kann, als in der *initialCapacity2()*-Methode, ist der Faktor bei einer Kapazität *n* von 10.000 Elementen schon auf 589,7 gewachsen.

Bei der Entscheidung, ob die hier genannte Optimierung geeignet ist, muss man abwägen, ob es wichtiger ist, dass das Collection-Objekt schnell erzeugt werden kann oder ob die Hinzufügeoperationen schnell erfolgen sollen. Die Messergebnisse haben klar gezeigt, dass sich das Vorinitialisieren einer ArrayList mit eine Kapazität n lohnt, wenn nach der Erzeugung des ArrayList-Objekts eine Anzahl von Elementen von mindestens 66 % der internen Kapazität des ArrayList-Objekts hinzugefügt wird. Bei einer Echtzeit-Anwendungen ist es möglicherweise wichtig, dass die Hinzufügeoperationen in konstanter Zeit erfolgen sollen. In diesem Fall ist das Initialisieren des Collection-Objekts mit der zu erwartenden Anzahl von Elementen gut geeignet. Wird das Collection-Objekt dagegen nur für wenige Elemente benötigt, ist es nicht sinnvoll es mit einer hohen Kapazität zu initialisieren, da dann die Objekterzeugung zu einem Overhead führt.

Die Qualität der Anwendung wird – bis auf die Performanz – nicht positiv oder negativ verändert.

3.2.3. Enumerating Constants

Konstante Objekte sollten, soweit dies möglich ist, intern durch eine eindeutige Abbildung auf einen Wert vom Typ int ersetzt werden. Wenn diese Abbildung konsistent angewendet wird, können dadurch Geschwindigkeits- und Speicherplatzvorteile entstehen. Der Grund dafür ist, dass Vergleiche von int-Werten schneller erfolgen können, als Vergleiche auf die Wertgleichheit von Objekten. Bei Vergleichen von Objekten sind Aufrufe der Methode equals() notwendig und diese kosten mehr Zeit, als ein Identitätsvergleich. [16]

Code-Beispiel:

```
public interface Direction {
```

```
        public static final Direction NORTH = null;
        public static final Direction EAST = null;
        public static final Direction SOUTH = null;
        public static final Direction WEST = null;
}


public class Direction_const implements Direction {
        public static final Direction NORTH = new Direction_const("North");
        public static final Direction EAST = new Direction_const("East");
        public static final Direction SOUTH = new Direction_const("South");
        public static final Direction WEST = new Direction_const("West");

        private String name;

        private Direction_const(String name) {
                this.name = name;
        }

        @Override
        public boolean equals(Object obj) {
                if (this == obj) return true;
                if (! (obj instanceof Direction_const))
                        return false;
                Direction_const other = (Direction_const) obj;
                if (name == null) {
                        if (other.name != null)
                                return false;
                } else if (!name.equals(other.name))
                        return false;
                return true;
        }
}


public class Direction_enum implements Direction{
        public static final Direction NORTH = new Direction_enum(0);
        public static final Direction EAST = new Direction_enum(90);
        public static final Direction SOUTH = new Direction_enum(180);
        public static final Direction WEST = new Direction_enum(270);

        private int degree;

        private Direction_enum(int degree) {
                this.degree = degree;
        }

        @Override
        public boolean equals(Object obj) {
                if (this == obj) return true;
                if (! (obj instanceof Direction_enum))
                        return false;
                Direction_enum other = (Direction_enum) obj;
                if (degree != other.degree) return false;
                return true;
        }
}
```

In diesem Code-Beispiel gibt ein Interface Direction, dass die vier Himmelsrichtungen als Konstanten für alle implementierenden Klassen vorgibt. In der Klasse Direction_const ist die Repräsentation einer Himmelsrichtung als String in der Instanzvariable *name* gespeichert. Beim Prüfen auf Wertgleichheit von zwei Direction_const-Objekten müssen – sofern sie nicht

identisch sind – u.a. die in den beiden Instanzvariablen *name* gespeicherten Strings mit der von String angebotenen *equals()*-Methode verglichen werden. Im Gegensatz dazu verwendet die Klasse Direction_enum für die Himmelsrichtungen int als Datentyp. Der Vergleich auf Wertgleichheit von zwei Direction_enum-Objekten kann durch Vergleich der Identität der jeweiligen Instanzvariable *degree* erfolgen.

Seit der Java Version 5 gibt es sogenannte enum-Klassen. Die konstanten Instanzen einer enum-Klasse haben eine natürliche Ordnung, da jede Instanz einem eindeutigem int-Wert zugeordnet ist. Somit kann hier auch bei einem Vergleich auf Wertgleichheit ein Identitätsvergleich durchgeführt werden. Ein weiterer Vorteil ist, dass viel weniger Code geschrieben werden muss, was man am nächsten Code-Beispiel sehen kann. [18]

Code-Beispiel:

```
public enum EDirection implements Direction{
	NORTH("North"), EAST("East"), SOUTH("South"), WEST("West");

	private String name;

	EDirection(String s) {
		name = s;
	}
}
```

Hier ist das Beispiel von oben mit einer enum-Klasse umgesetzt. Eine *equals()*-Methode muss in diesem Fall nicht extra geschrieben werden, da diese schon von der Oberklasse java.lang.Enum implementiert wird. Auch die von dem Interface geerbten Konstanten müssen nicht explizit initialisiert werden, wie es bei den Klassen Direction_const und Direction_enum notwendig ist, da für die Instanzen einer enum-Klasse automatisch Konstanten angelegt werden.

Szenario:

Für das Szenario sollen zwei Listen *list1* und *list2* mit jeweils 3000 Direction-Objekten gefüllt werden. Anschließend soll über die beiden Listen iteriert werden und die Elemente mit gleichem Index verglichen werden. Da der Java Profiler die Methodenaufrufe der *equals()*-Methode in der enum-Klasse nicht explizit analysiert bzw. die Analyseergebnisse nicht darstellt, muss der Aufruf der jeweiligen *equals()*-Methode von allen drei Klassen in eine Hilfsmethode namens *CodeOptimizations.obj_enum_const1()* (siehe Anhang) gelegt werden. Das ganze Szenario wird wieder in einer Endlosschleife ausgeführt. Ein Messdurchgang dauert 20 Minuten.

```
public static void szenario_enumConstants() {
	List<Direction> list1 = new ArrayList<Direction>(),
	                list2 = new ArrayList<Direction>();

	for(int i = 0; i < 1000; i++) {
		list1.add(Direction_const.EAST);
```

```
                list1.add(Direction_const.NORTH);
                list1.add(Direction_const.WEST);

                list2.add(Direction_const.NORTH);
                list2.add(Direction_const.SOUTH);
                list2.add(Direction_const.WEST);

                // separate Messung für Direction_enum
                // list1.add(Direction_enum.EAST);
                // list1.add(Direction_enum.NORTH);
                // list1.add(Direction_enum.WEST);

                // list2.add(Direction_enum.NORTH);
                // list2.add(Direction_enum.SOUTH);
                // list2.add(Direction_enum.WEST);

                // separate Messung für EDirection
                // list1.add(EDirection.EAST);
                // list1.add(EDirection.NORTH);
                // list1.add(EDirection.WEST);

                // list2.add(EDirection.NORTH);
                // list2.add(EDirection.SOUTH);
                // list2.add(EDirection.WEST);
        }

        while(true) {
                for(int i = 0, s = list1.size(); i < s; i++) {
                        CodeOptimizations.obj_enum_const1(list1.get(i),
                                list2.get(i));

                        // separate Messung für Direction_enum
                        // CodeOptimizations.obj_enum_const1(list1.get(i),
                                        list2.get(i));

                        // separate Messung für EDirection
                        // CodeOptimizations.obj_enum_const1(list1.get(i),
                                        list2.get(i));
                }
        }
}
```

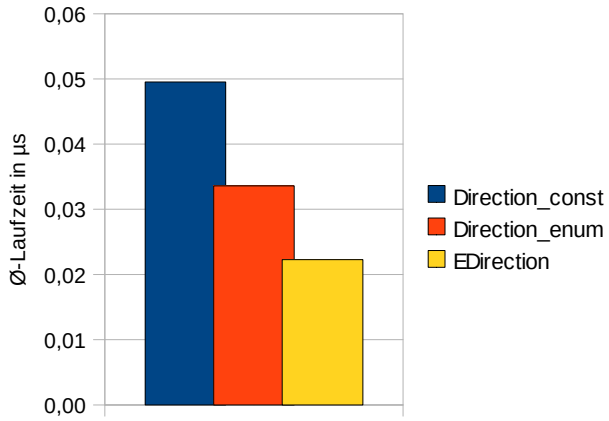

Abb. 3.8: Ø-Laufzeit des enum-Szenarios

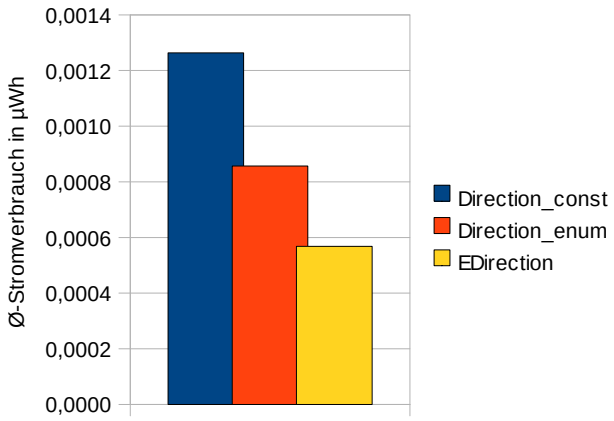

Abb. 3.9: Ø-Stromverbrauch des enum-Szenarios

Die gemessenen Werte – dargestellt in der Abb. 3.8 - für die durchschnittliche Laufzeit der *obj_enum_const1()*-Methode zeigen, dass die *equals()*-Methode der Direction_enum-Klasse um 32,2 % schneller ist als die der Direction_const-Klasse. Im Gegensatz zur EDirection-Klasse ist sogar ein Unterschied von 55 % zu erkennen. Abb. 3.9 zeigt, dass die Energieeffizienz durch den geringeren durchschnittlichen Stromverbrauch der *obj_enum_const1()*-Methode der Direction_enum-Klasse um 32 % gesteigert wird, im Gegensatz zur Direction_const-Klasse. Im Vergleich mit der EDirection-Klasse wird die Energieeffizienz sogar um 55 % gesteigert.

Ein Vorteil von enum-Klassen ist, dass man für die gleiche Funktionalität, im Vergleich zu einer normalen Klasse, weniger und somit viel übersichtlicheren Code schreiben kann, was die Wartbarkeit der Anwendung erhöht, weil der Quellcode besser zu lesen und zu verstehen ist. Weiterhin können die enum-Konstanten aufgrund der internen Abbildung auf einen int-Wert in einem switch-Konstrukt verwendet werden, was bei anderen Objekten nicht möglich ist. Nachteile von enum-Klassen sind bspw., dass man keine Objekte von Ihnen zur Laufzeit erzeugen kann und es nicht möglich ist von enum-Klassen zu erben.
Ist es allerdings notwendig zur Laufzeit zusätzliche Objekte zu Erzeugen oder soll die Klasse in eine Vererbungshierarchie eingegliedert werden, ist es sinnvoll, ähnlich wie es in der Direction_enum-Klasse gezeigt wurde, vorzugehen.

3.2.4. Lazy Initialization

Die sogenannte Lazy Initialization kann den Vorgang der Objekterzeugung beschleunigen, indem Initialisierungen verzögert werden. Dabei wird eine Instanzvariable nicht sofort im Konstruktor initialisiert, sondern erst dann, wenn im späteren Verlauf des Programms auf diese Variable zugegriffen werden soll und eine Initialisierung wirklich von Nöten ist. Gerade wenn die Initialisierung die Erzeugung eines großen Objekts erfordert und auf das Objekt bzw. die Variable im Programmverlauf möglicherweise gar nicht zugegriffen wird, spart man die Zeit für das Erzeugen des komponierten oder aggregierten Objekts ein. Im Falle, dass das Objekt tatsächlich gebraucht wird, wird es direkt vor dem Zugriff erzeugt. Durch das direkte

Erzeugen des Objekts vor dem Zugriff, wird zumindest der erste Zugriff verlängert, da das komponierte oder aggregierte Objekt erzeugt werden muss. Im schlechtesten Fall, dass bei allen Objekten eine Initialisierung der Instanzvariablen zu einem verzögerten Zeitpunkt erfolgt, fällt ein Overhead für das Prüfen, ob die Instanzvariable schon initialisiert wurde, an. Beim Einsatz von mehreren Threads sollte die Umsetzung der verzögerten Initialisierung synchronisiert werden, da sonst nicht ausgeschlossen werden kann, dass mehrere Threads gleichzeitig die Initialisierung einer Instanzvariable auslösen und damit eine Wettlaufsituation darum entsteht, welches neu erzeugte Objekt die Instanzvariable referenziert. [16,19]
Gibt es eigentlich einen optimalen Zeitpunkt für das Initialisieren einer Instanzvariablen ?
Ziel einer effizienten Implementation einer Klasse ist es doch, dass ein Objekt schnell erzeugt werden kann und das auf die Instanzvariable zugegriffen werden kann, ohne das dies zu einer spürbaren Verzögerung durch die Initialisierung führt. Dazu muss die Instanzvariable soweit vor dem Zugriff auf sie initialisiert werden, wie es für das Erzeugen des komponierten oder aggregierten Objekts und der Zuweisung zur Variable nötig ist. Diesen Zeitpunkt jedoch genau zu bestimmen ist sehr abhängig davon, ob Wissen über den tatsächlichen Zeitpunkt des Zugriffs auf die Instanzvariable vorliegt. Dieses Wissen kann jedoch nur vorliegen, wenn der tatsächliche Programmverlauf bekannt ist, was zwar theoretisch möglich ist, praktisch aber nur unter großem Aufwand zu bewerkstelligen ist. In den meisten Fällen wird es deshalb nicht möglich sein den optimalen Zeitpunkt zu finden.

Code-Beispiel:

```
public class File_Impl {

	private String file;
	private OutputStream out;

	public File_Impl(String f) throws IOException {
		file = f;
		out = new FileOutputStream(f);
	}

	public OutputStream getOutputStream() {
		return out;
	}
}

public class LazyFile_Impl {

	private String file;
	private OutputStream out;

	public LazyFile_Impl(String f) {
		this.file = f;
	}

	public OutputStream getOutputStream() throws
		FileNotFoundException {
		return (out == null) ? out = new FileOutputStream(file) :
			out;
	}
}
```

In der Klasse File_Impl wird die Instanzvariable *out* im Konstruktor initialisiert, indem ein FileOutputStream mit dem übergebenen String *f* erzeugt wird. Im Gegensatz dazu wird in der Klasse LazyFile_Impl erst ein FileOutputStream erzeugt und die Variable *out* initialisiert, wenn mit *getOutputStream()* tatsächlich ein FileOutputStream angefordert wird.

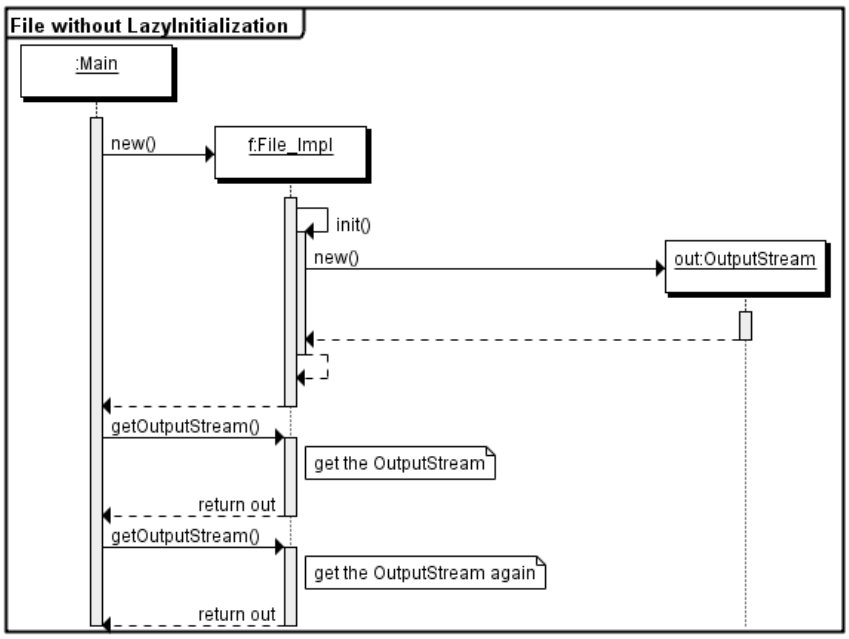

Abb. 3.10: File mit gewöhnlicher Initialisierung

Die Abbildung Abb. 3.10 zeigt ein Sequenzdiagramm einer Erzeugung eines File_Impl-Objekts aus dem Code-Beispiel. Bei mehrfachem Aufruf der Methode getOutputStream() wird stets das OutputStream-Objekt out zurückgegeben.
In der folgenden Abbildung Abb. 3.11 ist ein Sequenzdiagramm zu sehen, in dem eine Instanz der Klasse LazyFile_Impl erzeugt wird. Beim ersten Aufruf der Methode getOutputStream() muss der OutputStream out erzeugt werden. Jeder weitere Aufruf gibt dann ähnlich wie in der Abbildung Abb. 3.10 das Objekt, das in der Variable out referenziert wird zurück.

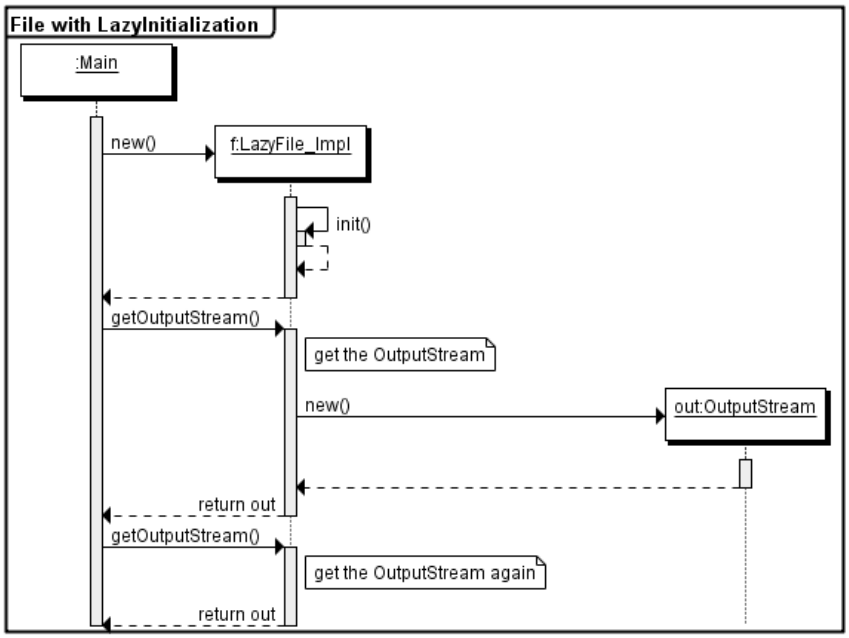

Abb. 3.11: File mit verzögerter Initialisierung

Szenario:

Es werden alle Dateien eines Verzeichnisses *dir* mit 1000 unterschiedlich großen Dateien erfasst und die Pfade der Dateien als Strings in einer Liste *fileNames* gespeichert. Für eine Anzahl von *n* Dateien soll dann jeweils ein OutputStream angefordert, wobei *n* die Werte 10, 100, 500, 700 und 1000 annehmen soll. Damit soll ein Vergleich möglich werden, ab wievielen angeforderten OutputStreams von den 1000 Dateien sich die verzögerte Initialisierung lohnt.

```
public static void szenario_LazyInitialization() {
	String dir = "D:/BA_Tests/files";
	int n = 100;

	List<String> fileNames = new ArrayList<String>();

	for(File f: (new File(dir).listFiles())) {
			fileNames.add(f.getName());
	}

	while(true) {
		CodeOptimizations.obj_file(fileNames, n);

		// separate Messung
		// CodeOptimizations.obj_lazyFile(fileNames, n);
	}
}
```

Damit man die Laufzeit für die Objekterzeugung und die Anforderung der OutputStreams besser messen kann, werden zwei weitere Methoden *CodeOptimizations.obj_file()* und *CodeOptimizations.obj_lazyFile()* benötigt, denen wie im Folgenden zu sehen, die Liste der Dateipfade *fileNames* und die Anzahl der Dateien *n*, zu denen ein OutputStream angefordert werden soll, übergeben wird.

```
public static void obj_file(List<String> fileNames, int n) {
	List<File_Impl> files = new ArrayList<File_Impl>();

	try {
		for (String s : fileNames)
			files.add(new File_Impl(s));
	} catch (IOException e) {}

	for (int i = 0; i < n; i++)
		files.get(i).getOutputStream();
}

public static void obj_lazyFile(List<String> fileNames, int n) {
	List<LazyFile_Impl> files = new ArrayList<LazyFile_Impl>();

	for(String s: fileNames)
		files.add(new LazyFile_Impl(s));

	try {
		for (int i = 0; i < n; i++)
			files.get(i).getOutputStream();
	} catch (IOException e) {}
}
```

In den beiden Methoden wird zu jedem Dateipfad ein File_Impl- bzw. ein LazyFile_Impl-Objekt erzeugt und in einer List *files* gespeichert. Anschließend wird von *n* Dateien ein OutputStream angefordert. Die Methoden *obj_file()* und *obj_lazyFile()* werden separat voneinander gemessen. Ein Messdurchgang soll 20 Minuten dauern.

Die Messergebnisse, die in Abb. 3.12 und Abb. 3.13 dargestellt sind, zeigen dass das Szenario mit der *obj_lazyFile()*-Methode in den meisten Fällen um ein Vielfaches schneller und energieeffizienter ist. Während die durchschnittliche Laufzeit und der Stromverbrauch der *obj_file()*-Methode relativ konstant ist, steigen beide Werte bei der *obj_lazyFile()*-Methode fast proportional bei steigendem Wert für *n* und nähern sich bei einem Wert für *n* zwischen 700 und 1000 der Laufzeit und dem Stromverbrauch der *obj_file()*-Methode an. Dass sich die Messergebnisse beider Methoden zwischen 700 und 1000 einander annähern, ist damit zu erklären, dass bei *n* gleich 1000 für jedes File_Impl- bzw. LazyFile_Impl-Objekt ein OutputStream erzeugt wird, während bei kleineren Werten für *n*, nicht für jedes LazyFile_Impl-Objekt ein OutputStream erzeugt wird.

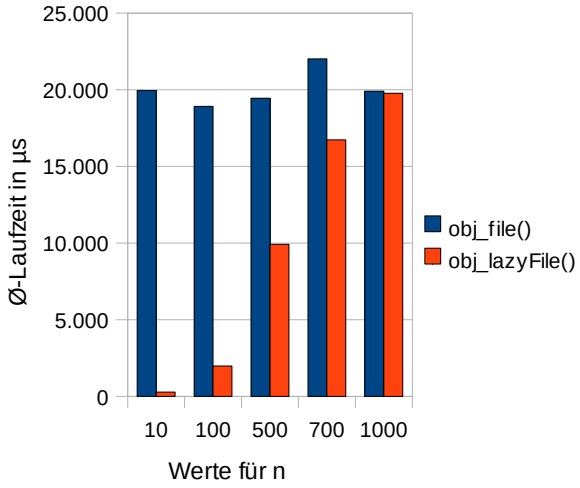

Abb. 3.12: Ø-Laufzeit der obj_file-Methoden

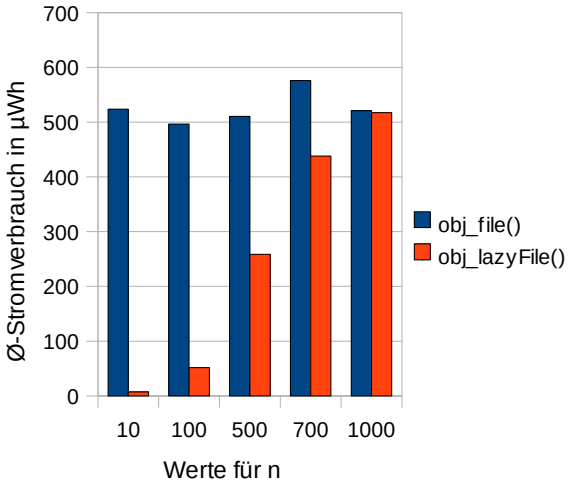

Abb. 3.13: Ø-Stromverbrauch der obj_file-Methoden

Weitere Messungen (siehe Tabelle 3.4) zeigen, dass die Erzeugung eines File_Impl-Objekts im Mittel um den Faktor 200 langsamer ist, als die Erzeugung eines LazyFile_Impl-Objekts. Dagegen ist es bei der Laufzeit der *getOutputStream()*-Methode genau anders herum, denn im Mittel ist sie in der Klasse File_Impl um den Faktor 242 schneller, als in der LazyFile_Impl-Klasse.

n	**10**	**100**	**500**	**700**	**1000**
File_Impl Objekterzeugung Ø-Laufzeit in µs	19,06443	17,93813	18,32725	20,91774	18,92391
LazyFile_Impl Objekterzeugung Ø-Laufzeit in µs	0,02849	0,05139	0,10979	0,14822	0,15409
File_Impl.getOutputStream() Ø-Laufzeit in µs	0,14103	0,06956	0,06089	0,05725	0,05178
LazyFile_Impl.getOutputStream() Ø-Laufzeit in µs	14,32643	15,08060	17,96873	21,87801	18,30865

Tabelle 3.4: Messergebnisse des Szenarios Lazy Initialization

Es lohnt sich also die verzögerte Initialisierung in diesem Beispiel einzusetzen, wenn man vorher weiß, dass es eine grosse Anzahl von Dateien gibt und nicht für alle Dateien ein OutputStream angefordert wird. In diesem Fall machen sich die geringen Kosten für die Objekterzeugung bezahlt. Wenn dagegen von fast allen Dateien ein OutputStream angefordert wird und das vielleicht sogar mehrfach pro Datei erfolgt, lohnt sich die verzögerte Initialisierung nicht, denn dann werden die Einsparungen durch die geringen Kosten der Objekterzeugung von den hohen Kosten für die Anforderung eines OutputStreams übertroffen.

Die Qualitätskriterien – bis auf die Performanz – der Anwendung wird durch das Einsetzen der verzögerten Initialisierung nicht negativ oder positiv berührt. Es ist zwar ein wenig mehr Quellcode zu schreiben, dieser wird dadurch aber nicht schwerer zu lesen und zu verstehen. Der Anwendungsprogrammierer muss für die Benutzung der LazyFile_Impl-Klasse nicht wissen, dass ein OutputStream erst erzeugt wird, wenn er durch die *getOutputStream()*-Methode angefordert wird und hat dadurch keine Nachteile bei der Verwendung dieser Klasse.

3.3. Schleifen

Da Programme die meiste Zeit mit dem Ausführen von Schleifen beschäftigt sind, sind diese ein wichtiger Ansatzpunkt, die Laufzeit der Programme zu verbessern. Gerade in Schleifen machen sich unperformante Operationen bemerkbar, da sie mehrfach ausgeführt werden und sich der Overhead bei jedem Schleifendurchlauf summiert. Abhängig sind die Auswirkungen auf die Laufzeit und die Energieeffizienz selbstverständlich auch davon, wie oft eine Schleife durchlaufen wird.

3.3.1. Invariante Operationen

Invariante Operationen, die in Schleifen ausgeführt werden, d.h. Operationen, die bei mehrfacher Ausführung immer dasselbe Ergebnis liefern, sollten außerhalb der Schleife durchgeführt werden, da die mehrfache Ausführung unnötig ist und einen Overhead erzeugt. Sinnvoll ist diese Regel vor allem dann, wenn die invariante Operation aufwändig ist. [14]

Laufzeit-Analyse:

Angenommen eine Schleife wird n mal durchlaufen, die Operationen in der Schleife haben – zur Vereinfachung – eine konstante Laufzeit von 1 und es gibt k invariante Operationen, die ebenfalls eine konstante Laufzeit von 1 haben. Dann gilt für die Laufzeit der Schleife mit invarianten Operationen:

$$T_1(n,k) = n + kn = n(k+1)$$

Für die Laufzeit der Schleife ohne invariante Operationen gilt:

$$T_2(n,k) = n + k$$

Die k invarianten Operationen werden in der Formel T_2 dazu addiert, da sie im Vorfeld der Schleife berechnet werden.

Code-Beispiel:

```
public static int loop_invariant(int x, int y, int n) {
	int result = 0;

	for(int i=0; i <= n; i++) {
		result += x / y;
	}

	return result;
}

public static int loop_invariant_motion(int x, int y, int n) {
	int result = 0, temp = x / y;

	for(int i=0; i <= n; i++) {
		result += temp;
	}

	return result;
}
```

In diesem Beispiel, wird in der Methode *loop_invariant()* eine invariante Operation *x / y* in der for-Schleife bei jedem Schleifendurchlauf ausgeführt. Die Methode *loop_invariant_motion()* zeigt, wie die invariante Operation vor dem Eintritt in die Schleife ausgeführt wird, indem das Ergebnis der Operation einer lokalen Variable zugewiesen wird, die dann in der Schleife verwendet wird. In beiden Methoden wird die for-Schleife *n*-Mal durchlaufen.

Szenario:

```
public static void szenario_loop_motion() {
	int n = 10;
	int x = 250, y = 5;

	while(true) {
		CodeOptimizations.loop_invariant(x, y, n);
		CodeOptimizations.loop_invariant_motion(x, y, n);
	}
}
```

In einer Endlosschleife sollen die Methoden *loop_invariant()* und *loop_invariant_motion()* mit Werten für *n* von 10, 500 und 1000 aufgerufen werden. Ein Messdurchgang soll 10 Minuten dauern.

n	**10**	**500**	**1000**
loop_invariant() Ø-Laufzeit in µs	0,01052	9,37765	23,64843
loop_invariant_motion() Ø-Laufzeit in µs	0,00768	0,01298	0,01929
loop_invariant() Ø-Stromverbrauch in µWh	0,00027	0,23913	0,59909
loop_invariant_motion() Ø-Stromverbrauch in µWh	0,00020	0,00033	0,00049

Tabelle 3.5 Messergebnisse des Szenarios invariante Operationen

Die in der Tabelle 3.5 dargestellten Messergebnisse zeigen, dass sich schon bei relativ wenigen Iterationsschritten einer Schleife eine invariante Operation negativ auf die Laufzeit und die Energieeffizienz auswirkt. Bei *n* gleich 10 ist die durchschnittliche Laufzeit und der Stromverbrauch bei der *loop_invariant()*-Methode um einen Faktor 1,3 höher, als bei der *loop_invariant_motion()*-Methode. Bei größeren Werten für *n* steigt die durchschnittliche Laufzeit und der Stromverbrauch der *loop_invariant()*-Methode mehr als linear an, da für jeden Iterationsschritt die invariante Operation berechnet werden muss. Im Gegensatz dazu steigt die durchschnittliche Laufzeit und der Stromverbrauch bei der *loop_invariant_motion()*-Methode nur linear bzw. weniger als linear, weil sich der Aufwand für die eine invariante Operation, die im Vorfeld der Schleife berechnet wird, auf die Anzahl aller Iterationsschritte aufteilt. Bei 1000 Iterationsschritten ist die durchschnittliche Laufzeit der *loop_invariant()*-Methode um den Faktor 1226 langsamer und der Stromverbrauch um den Faktor 1223 höher, als bei der *loop_invariant_motion()*-Methode.

Auch invariante Operationen in der Abbruchbedingung der Schleife sollten vermieden werden, da die Abbruchbedingung bei jedem Schleifendurchlauf geprüft wird.

Code-Beispiel:

```
public static void loop_invariant_breakCond(List<Integer> l) {
	for(int i=0; i < l.size(); i++) {
		; // do something
	}
}

public static void loop_invariant_motion_breakCond(List<Integer> l) {

	for(int i=0, size = l.size(); i < size; i++) {
		; // do something
	}
}
```

In beiden Methoden soll über die übergebene Liste *l* in einer for-Schleife iteriert werden. Zum Prüfen der Abbruchbedingung zieht die Methode *loop_invariant_breakCond()* die Länge der Liste mit deren *size()*-Methode heran. Im Gegensatz dazu wird in der Methode *loop_invariant_motion_breakCond()* die Abbruchbedingung gegen eine lokale Variable *size* geprüft, die ebenfalls mit der Länge der Liste *l* initialisiert wird.

Szenario:

```
public static void szenario_loop_motion_breakCond() {
	int n = 10;

	List<Integer> liste = new ArrayList<Integer>(n);
	for(int i = 0; i < n; i++) liste.add(i);

	while(true) {
		CodeOptimizations.loop_invariant_breakCond(liste);
		CodeOptimizations.loop_invariant_motion_breakCond(liste);
	}
}
```

In dem Szenario soll eine Liste liste mit Integer-Objekten gefüllt werden. Die Länge der Liste soll durch die Variable *n* vom Typ int festgelegt werden und in verschiedenen Messungen 10 und 1000 betragen. Ein Messdurchgang dauert 10 Minuten.

n	**10**	**1000**
loop_invariant_breakCond() Ø-Laufzeit in µs	0,15456	14,31433
loop_invariant_motion_breakCond() Ø-Laufzeit in µs	0,02046	0,02096
loop_invariant_breakCond() Ø-Stromverbrauch in µWh	0,00397	0,36740
loop_invariant_motion_breakCond() Ø-Stromverbrauch in µWh	0,00053	0,00054

Tabelle 3.6 Messergebnisse des Szenarios invariante Operationen in der Abbruchbedingung

Die Messergebnisse in der Tabelle 3.6 zeigen ähnliche Werte, wie in der Tabelle 3.5. Bei einem Wert von *n* gleich 10 ist der Unterschied zwischen der *loop_invariant_breakCond()*-Methode und der *loop_invariant_motion_breakCond()*-Methode mit einem Faktor von 7,5 für die durchschnittliche Laufzeit und den Stromverbrauch zugunsten der *loop_invariant_motion_breakCond()*-Methode zu beziffern. Für einen Wert von *n* gleich 1000 ist der Faktor ca. 680.

Die Messungen haben also gezeigt, dass sich diese Optimierung in jedem Fall lohnt, da bei jedem zusätzlichen Schleifendurchlauf mit invarianten Operationen der Overhead immer weiter linear wächst.
Die Qualität der Anwendung wird beim Ersetzen einer einzigen invarianten Operation in einer Schleife nicht beeinträchtigt. Sind jedoch eine größere Menge von invarianten Operationen vorhanden, wird der Quellcode durch das Anlegen weiterer Variablen leicht unleserlich und schwerer zu verstehen, was die Wartbarkeit negativ beeinträchtigt. Dadurch entstehen auch wieder zusätzliche Fehlerquellen.

3.3.2. Loop Interchange

Beim Loop Interchange soll bei mehreren ineinander verschachtelten Schleifen die Ordnung abhängig von der Anzahl der Iterationen geändert werden. Die Schleife mit der größeren Anzahl von Iterationen soll nach Innen gelegt werden. Je größer die Anzahlen der Iterationen pro Schleife voneinander abweichen, desto größer sollte auch die zeitliche Ersparnis durch das Vertauschen sein.

Laufzeit-Analyse:

Angenommen, es gibt eine äußere Schleife, die n mal durchlaufen wird und eine innere Schleife, die k mal durchlaufen wird. Die Operationen in der inneren Schleife, einschließlich der Prüfung der Abbruchbedingung und dem Inkrementieren/Dekrementieren des Schleifenindex, sollen – zur Vereinfachung – in konstanter Zeit von 1 erfolgen. Das Anlegen der lokalen Indexvariable erfolgt auch in konstanter Zeit von 1.
Für die Laufzeit gilt:

$$T(n,k) = n(k+1)+1$$

Die Laufzeit für eine verschachtelte Schleife mit z.B. n = 2000 und k = 5 ist also T(2000, 5) = 12001. Vertauscht man hingegen n und k, ist die Laufzeit T(5, 2000) = 10006. Der Unterschied ist ca. ein Faktor von 1,2.

Code-Beispiel:

```
public static long loop_nested(int x, int y) {
	long result = 0;
	for (int k=0; k <= x; k++) {
		for (int i=0; i <= y; i++) {

			result += i + k;
		}
	}
	return result;
}

public static long loop_nested_inter(int x, int y) {
	long result = 0;
	for (int i=0; i <= y; i++) {
		for (int k=0; k <= x; k++) {
			result += i + k;
		}
	}
	return result;
}
```

Beiden Methoden wird die Anzahl der vorgesehenen Iterationen als Variablen *x* und *y* übergeben. In der Methode *loop_nested()* wird eine verschachtelte Schleife verwendet, die äußere Schleife zählt dabei *x* Iterationen und die innere Schleife *y* Iterationen. Bei der Methode *loop_nested_inter()* ist es genau umgekehrt, die äußere Schleife zählt *y* und die innere Schleife *x* Iterationen. Der Einfachheit halber, wird für *x* der größere Wert erwartet. Eine Prüfung, welcher Wert kleiner ist, wird in diesem Beispiel nicht durchgeführt.

Szenario:

Das Szenario sieht vor, dass die Methoden *loop_nested()* und *loop_nested_inter()* in einer Endlosschleife mit verschiedenen Werten für *x* und *y* aufgerufen werden. Die Kombinationen für (x,y) sollen in separaten Messdurchgängen (2000,5), (2000, 50), (2000,200), (2000,500) und (2000,1000) sein. In einer Endlosschleife werden dann die o.g. Methoden aufgerufen. Ein Messdurchgang dauert 10 Minuten.

```
public static void szenario_loop_interchange() {
	int x = 2000, y = 5;

	while(true) {
		CodeOptimizations.loop_nested(x, y);
		CodeOptimizations.loop_nested_interchanged(x, y);
	}
}
```

(x,y)	(2.000, 5)	(2.000, 50)	(2.000, 200)	(2.000, 500)	(2.000, 1.000)
loop_nested() Ø-Laufzeit in µs	47,3174	389,6926	1.482,8886	3.667,0498	7.310,1790
loop_nested_inter() Ø-Laufzeit in µs	38,8816	366,9720	1.456,5623	3.618,5921	7.167,6239
loop_nested() Ø-Stromverbrauch in µWh	1,3012	10,7815	41,0266	101,4550	202,2483
loop_nested _inter() Ø-Stromverbrauch in µWh	1,0692	10,1529	40,2982	100,1144	198,3043

Tabelle 3.7 Messergebnisse des Szenarios Loop Interchange

Aus den Messergebnissen in der Tabelle 3.7 wird ersichtlich, dass der größte Unterschied in der durchschnittlichen Laufzeit und des Stromverbrauchs zwischen den beiden Methoden *loop_nested()* und *loop_nested_inter()* besteht, wenn die Abweichung zwischen den Werten für *x* und *y* groß ist. Bei Werten für (x,y) von (2000,1000), ist die *loop_nested()*-Methode ca. 2 % langsamer und energieineffizienter, als die *loop_nested_inter()*-Methode. Der Unterschied in der durchschnittlichen Laufzeit und dem Stromverbrauch wächst bei Werten von (2000,50) auf ca. 6 % und liegt bei Werten von (2000,5) sogar bei ca. 21 % zugunsten der *loop_nested_inter()*-Methode.

Aufgrund dieser Ergebnisse ist diese Optimierung hinsichtlich der durchschnittlichen Laufzeit und der Energieeffizienz also in jedem Fall sinnvoll. Ob es allerdings tatsächlich möglich ist, in der Anwendung diese Optimierung einzusetzen, hängt z.B. auch davon ab, ob die Reihenfolge der Verschachtelung der Schleifen semantisch eine Rolle spielt, welche Variablen in der Abbruchbedingung der Schleife verwendet werden und wie die Indexvariablen innerhalb der Schleifen verwendet werden. Bspw. ist es nicht möglich von zwei ineinander verschachtelten Schleifen die Ordnung zu ändern, wenn die Anzahl der Iterationen für die innere Schleife dynamisch mit jedem Iterationsschritt der äußeren Schleife festgelegt wird.

Die Qualität der Anwendung wird – bis auf das Kriterium der Performanz – durch diese Optimierung nicht negativ oder positiv beeinflusst.

3.3.3. Indexvariablen

Als Indexvariablen für Schleifen sollten – soweit möglich – nur Variablen vom Typ int verwendet werden, da die Java VM nicht für alle Typen den gleichen Umfang an Bytecode-Instruktionen anbietet. Die beste Unterstützung in Form von Bytecode-Instruktionen hat der Typ int, da dies die Voraussetzung für eine effiziente Implementation der Operanden-Stacks und der Arrays für die lokalen Variablen der Java VM ist. Die Verwendung von byte, short und char haben implizite Typecasts zu oder von int zur Folge, da es keine Bytecode-Instruktionen, wie store, load und add für diese Typen gibt. [16]
Die folgende Methode *iloop()* zeigt eine typische for-Schleife mit einer Indexvariable vom Typ int:

```
public void iloop() {
	for(int i = 0; i < 100; i++) {
		;  // do something
	}
}
```

Der kompilierte Bytecode ist dann folgender:

```
 0 iconst_0
 1 istore_0
 2 goto 8 (+6)
 5 iinc 0 by 1
 8 iload_0
 9 bipush 100
11 if_icmplt 5 (-6)
14 return
```

Anders sieht es aus, wenn man statt einer int- eine short-Variable verwendet. Der Java Code unterscheidet sich kaum in der Methode *sloop()*:

```
public static void sloop() {
	for(short i = 0; i < 100; i++) {
		; // do something
	}
}
```

Der kompilierte Bytecode zeigt jedoch, dass ein etwas größerer Aufwand notwendig ist:

```
 0 iconst_0
```

```
 1 istore_0
 2 goto 10 (+8)
 5 iload_0
 6 iconst_1
 7 iadd
 8 i2s
 9 istore_0
10 iload_0
11 bipush 100
13 if_icmplt 5 (-8)
16 return
```

Dass die Instruktionen mit dem Buchstaben i beginnen, wie iload, iconst, iadd verdeutlicht, dass hier mit einer Variable vom Typ int gearbeitet wird. Anhand der Instruktion i2s kann man sehen, dass das Ergebnis der Inkrementierung der Indexvariable nach der iadd-Instruktion zu einer short-Variable gecastet wird.
Noch aufwändiger wird es, wenn man eine Indexvariable vom Typ double, float oder long verwendet. Die Methode *dloop()* verwendet eine double-Variable:

```
public static void dloop() {
	for(double i = 0.0; i < 100.0; i++) {
		; // do something
	}
}
```

Der kompilierte Bytecode sieht so aus:

```
 0 dconst_0
 1 dstore_0
 2 goto 9 (+7)
 5 dload_0
 6 dconst_1
 7 dadd
 8 dstore_0
 9 dload_0
10 ldc2_w #167 <100.0>
13 dcmpg
14 iflt 5 (-9)
17 return
```

Bei Variablen vom Typ double, float und long müssen bei den Additionen und Vergleichen jeweils 2 Variablen auf den Operanden-Stack gelegt werden, im Gegensatz zu int-Variablen, bei denen nur eine lokale Variable notwendig ist. Weiterhin gibt es keine Instruktion zum Inkrementieren, wie beim Typ int die iinc-Instruktion. Es sind deshalb insgesamt 4 Operationen zum Inkrementieren dieser Variablen notwendig, also 3 Operationen mehr als beim Typ int. Auch der Vergleich ist bei einer int-Variable mit einer einzigen Instruktion

namens if_cmplt möglich, während bei den drei anderen Typen 2 Operationen durchgeführt werden müssen. Bei einer double-Variable muss z.B. erst eine dcmpg- und dann eine iflt-Instruktion ausgeführt werden. [16,10]

Code-Beispiel:

```
public static void loop_indexVar1(int n) {
	for(int i = 0; i < n; i++) {
		;
	}
}

public static void loop_indexVar2(short n) {
	for(short i = 0; i < n; i++) {
		;
	}
}

public static void loop_indexVar3(double n) {
	for(double i = 0.0; i < n; i++) {
		;
	}
}
```

Analog zu den Methoden *iloop()*, *sloop()* und *dloop()*, haben diese drei Beispielmethoden *loop_indexVar1()*, *loop_indexVar2()* und *loop_indexVar3()* auch keine Operationen in der Schleife, um besser miteinander verglichen zu werden. Die Anzahl der Iterationen pro Schleife wird hier als Parameter *n* übergeben.

Szenario:

In diesem Szenario sollen die drei o.g. Methoden *loop_indexVar1()*, *loop_indexVar2()* und *loop_indexVar3()* mit Werten für *n* von 100, 1000, 2000, 5000 und 10000 in einer Endlosschleife aufgerufen werden. Ein Messdurchgang dauert 10 Minuten.

```
public static void szenario_loop_indexVar() {
	int in = 100;
	short sn = 100;
	double dn = 100;

	while(true) {
		CodeOptimizations.loop_indexVar1(in);
		CodeOptimizations.loop_indexVar2(sn);
		CodeOptimizations.loop_indexVar3(dn);
	}
}
```

Anzahl Iterationen	100	1.000	2.000	5.000	10.000
loop_indexVar1() Ø-Laufzeit in µs	0,00719	0,01511	0,02153	1,14952	7,22778
loop_indexVar2() Ø-Laufzeit in µs	0,00787	0,02238	0,03266	4,18048	13,26799
loop_indexVar3() Ø-Laufzeit in µs	0,01372	4,20996	13,37038	40,83202	86,55332
loop_indexVar1() Ø-Stromverbrauch in µWh	0,00018	0,00040	0,00057	0,03027	0,19033
loop_indexVar2() Ø-Stromverbrauch in µWh	0,00020	0,00059	0,00086	0,11009	0,34939
loop_indexVar3() Ø-Stromverbrauch in µWh	0,00035	0,11016	0,35209	1,07524	2,27924

Tabelle 3.8 Messergebnisse des Szenarios Indexvariablen

Anhand der Messergebnisse, die in der Tabelle 3.8 dargestellt sind, kann man erkennen, dass tatsächlich bei allen Werten für *n* die Methode *loop_indexVar1()* schneller und energieeffizienter als die anderen beiden Methoden ist, gefolgt von der *loop_indexVar2()*-Methode. Die durchschnittliche Laufzeit und der Stromverbrauch der *loop_indexVar1()*-Methode ist z.B. bei *n* gleich 100 um den Faktor 1,1 kleiner gegenüber der *loop_indexVar2()*-Methode und um den Faktor 1,9 kleiner gegenüber der *loop_indexVar3()*-Methode. Bei größeren Werten für *n* wird der Unterschied zwischen den Methoden noch deutlicher. Z.B. bei *n* gleich 10.000 unterscheidet sich die durchschnittliche Laufzeit und der Stromverbrauch der *loop_indexVar1()*-Methode um einen Faktor 1,8 gegenüber der *loop_indexVar2()*-Methode und sogar um einen Faktor 11,9 gegenüber der *loop_indexVar3()*-Methode.

Solange also der Speicherplatz, den die Indexvariable belegt, nicht relevant ist und der niedrigste bzw. höchste Wert der Indexvariable der Schleife innerhalb des Wertebereichs einer int-Variable von -2^{31} bis $2^{31}-1$ bleibt, sollte in jedem Fall eine int-Variable verwendet werden. Die Verwendung einer double-Variable als Schleifenindex ist aufgrund der Messergebnisse nicht zu empfehlen.

Die Qualität der Anwendung wird – bis auf die Performanz – in keinster Weise durch das Verwenden von int-Variablen als Schleifenindex negativ oder positiv beeinflusst.

3.3.4. Integer Vergleiche

Ein Vergleich in der Abbruchbedingung einer Schleife ist schneller, wenn der Vergleich mit einer der int-Konstanten -1, 0, 1, 2, 3, 4 oder 5 erfolgt. Der Grund dafür ist nicht etwa, dass die Vergleichsinstruktion im Bytecode schneller ist, sondern dass es eine besondere Instruktion iconst gibt, die speziell diese Konstanten auf den Operanden-Stack ablegt, weil die Verwendung dieser Konstanten besonders häufig ist. Andere Konstanten müssen z.B. mit der bipush-Instruktion auf den Operanden-Stack abgelegt werden. Während für die iconst-Instruktion kein weiterer Aufwand nötig ist, um einen der o.g. Konstanten auf den Operanden-Stack abzulegen, da die Information, welche Konstante abzulegen ist, bereits implizit in der Instruktion enthalten ist, muss innerhalb der bipush-Instruktion erst der explizite Operand angefordert und decodiert werden, um ihn dann auf den Operanden-Stack abzulegen.

Eine for-Schleife, die den Schleifenindex rückwärts zählt und in der Abbruchbedingung einen Vergleich wie z.B. gegen 0 durchführt, sollte demnach schneller sein, als eine for-Schleife, deren Schleifenindex hochgezählt wird und bei der in der Abbruchbedingung ein Vergleich mit einer anderen int-Konstante erfolgt. [16,10]

Code-Beispiel:

```
public static void loop_comparison1(int n) {
	for(int i = 0; i < n; i++) {
		;
	}
}

public static void loop_comparison2(int n) {
	for(int i = n - 1; i >= 0; i--) {
		;
	}
}
```

Den beiden Methoden *loop_comparison1()* und *loop_comparison2()* wird die Anzahl der Iterationen für die for-Schleife als int-Variable *n* übergeben. In der Methode *loop_comparison1()* wird der Schleifenindex *i* von 0 hochgezählt, solange *i* kleiner *n* ist. Dagegen wird in der Methode *loop_comparison2()* der Index *i* von *n* runtergezählt, solange *i* kleiner oder gleich 0 ist.

Szenario:

```
public static void szenario_loop_comparison() {
	int n = 10;
	while(true) {
		CodeOptimizations.loop_comparison1(n);

		// separate Messung
		// CodeOptimizations.loop_comparison2(n);
	}
}
```

In einer Endlosschleife sollen die beiden Methoden *loop_comparison1()* und *loop_comparison2()* mit Werten für *n* von 10, 100, 1000 und 10000 aufgerufen werden. Ein Messdurchgang dauert 10 Minuten.

n	**10**	**100**	**1000**	**10000**
loop_comparison1() Ø-Laufzeit in µs	0,006901	0,007101	0,015737	10,200459
loop_comparison2() Ø-Laufzeit in µs	0,006546	0,007068	0,015178	10,185152
loop_comparison1() Ø-Stromverbrauch in µWh	0,000176	0,000182	0,000412	0,277112
loop_comparison2() Ø-Stromverbrauch in µWh	0,000167	0,000181	0,000397	0,276697

Tabelle 3.9 Messergebnisse des Szenarios Vergleiche in Schleifen

Die Messergebnisse in der Tabelle 3.9 zeigen, dass die durchschnittliche Laufzeit und der Stromverbrauch für alle Werte von *n* bei der Methode *loop_comparison2()* kleiner ist und sie damit energieeffizienter ist, als die *loop_comparison1()*-Methode. Bei einem Wert für *n* von 10, ist ein Unterschied in der durchschnittlichen Laufzeit und des Stromverbrauchs von 5,4 % zu erkennen, während bei einem Wert für *n* von 10.000 der Unterschied gerade einmal 0,15 % beträgt. Die durchschnittliche Laufzeit und der Stromverbrauch nähern sich bei größer werdenden Werten für n einander an.

Aufgrund der Erkenntnisse aus der Messung, ist der Einsatz dieser Optimierung zu empfehlen. Die tatsächliche Verwendbarkeit hängt davon ab, ob der Schleifenindex innerhalb der Schleife verwendet wird und ob die Reihenfolge der Werte des Schleifenindex eine Rolle spielt. Soll bspw. über ein Array iteriert werden, wobei bei einem Array-Index von 0 gestartet werden und auf die Elemente zugegriffen werden soll, ist die Umsetzung mit einer rückwärts laufenden for-Schleife nur umständlich möglich und wird deshalb nicht angeraten. Wird der Schleifenindex innerhalb der Schleife nicht verwendet, spricht nichts gegen eine rückwärtslaufende Schleife.

Die Qualität der Anwendung wird – bis auf die Performanz – durch die Umsetzung von for-Schleifen mit rückwärts laufendem Index nicht beeinträchtigt.

3.3.5. Arrayzugriffe

Soweit möglich sollten Arrayzugriffe innerhalb von Schleifen vermieden werden und stattdessen sollte der Arrayzugriff außerhalb der Schleife erfolgen und das Ergebnis in einer temporären Variable gespeichert werden, die dann in der Schleife verwendet wird. Hintergrund dafür ist, dass die Java VM bei jedem Arrayzugriff prüfen muss, dass die Array-Referenz nicht auf null zeigt und das der angegebene Index innerhalb der Array-Grenzen liegt. Das bedeutet, dass durch das wiederholte Prüfen ein Overhead entsteht. [14,16]

Als Beispiel eignet sich hier der Sortieralgorithmus Counting Sort, mit dem man ein Eingangs-Array *a* von positiven int-Werten sortieren kann. [20] Das Ergebnis ist ein sortiertes Array *b*. Dazu wird ein Hilfsarray *c* benötigt. Die Sortierung erfolgt in diesem Fall out-of-place.

Code-Beispiel:

```
public static int[] countingSort1(int[] a) {

	int max = a[0], b[] = new int[a.length];

	for (int i = 0; i < a.length; i++) {
		if (a[i] > max) max = a[i];
	}

	int[] c = new int[max+1];

	for (int i = 0; i < a.length; i++) {
		c[a[i]] += 1;
```

```
        }

        for(int i = 1; i <= max; i++) {
            c[i] += c[i-1];
        }

        for(int j = a.length -1; j >= 0; j--) {
            b[c[a[j]] -1] = a[j];
            c[a[j]] -= 1;
        }
        return b;
}
```

Als Ausgangspunkt ist hier eine naive Implementation des Algorithmus in der Methode *countingSort1()* gegeben. In der ersten for-Schleife wird der größte int-Wert in dem Eingabe-Array *a* ermittelt und in der Variable *max* gespeichert. Dann wird ein Hilfsarray *c* mit der Länge *max+1* erzeugt. Die zweite for-Schleife zählt die Vorkommen der int-Werte und speichert sie mit dem durch sich selbst repräsentierten Index in *c* ab. In der dritten for-Schleife werden die Werte im Hilfsarray *c* aufsummiert, sodass der Inhalt angibt, bis zu welcher Stelle der Wert des Index im Ausgabe-Array *b* steht. Die letzte for-Schleife überträgt die Werte aus *a* nach *b*. Dazu wird das Hilfsarray *c* genutzt.

Der nächste Schritt ist, die bekannten Optimierungen einzusetzen:

```
public static int[] countingSort1_opt(int[] a) {

        int max = a[a.length -1], b[] = new int[a.length];

        for (int i = a.length -1; i >= 0; i--) {
            if (a[i] > max) max = a[i];
        }

        int[] c = new int[max+1];

        for (int i = a.length -1; i >= 0; i--) {
            c[a[i]] += 1;
        }

        for(int i = 1; i <= max; i++) {
            c[i] += c[i-1];
        }

        for(int j = a.length -1, numj; j >= 0; j--) {
            numj = a[j];
            b[c[numj] -1] = numj;
            c[numj] -= 1;
        }
        return b;
}
```

In der Methode *countingSort1_opt()* wurden die ersten beiden for-Schleifen durch rückwärtslaufende for-Schleifen ausgetauscht (siehe Abschnitt 3.3.4.). Außerdem wird in der letzten for-Schleife eine temporäre Variable *numj* angelegt, die bei jedem Schleifendurchlauf den Wert *a[j]* zugewiesen bekommt. Im Folgenden wird dann auf *numj* zugegriffen, wenn auf *a[j]* zugegriffen werden soll. Es wird also in dieser Schleife nur noch ein Arrayzugriff auf *a*

durchgeführt, statt drei Zugriffen in der Methode *countingSort1()*. Insgesamt werden in der letzten for-Schleife der *countingSort1()*-Methode bei einer Größe des Arrays *a* von n: 6 * n Arrayzugriffe durchgeführt. Dagegen werden in der letzten for-Schleife der *countingSort1_opt()*-Methode nur 4 * n Arrayzugriffe durchgeführt.

Als nächstes kann man sich das Wissen über die Bedeutung des Inhalts des Hilfsarrays *c* zu Nutze machen. Für jeden Index des Arrays ist nämlich die Anzahl der Elemente mit dem Wert des Index darin gespeichert. Damit kann man eine weitere, schnellere Implementation des Algorithmus schreiben:

```
public static int[] countingSort2_opt(int[] a) {

	int max = a[0], b[] = new int[a.length];

	for (int i = a.length -1; i >= 0; i--) {
		if (a[i] > max) max = a[i];
	}

	int[] c = new int[max+1];

	for (int i = a.length -1, numi; i >= 0; i--) {
		numi = a[i];
		c[numi] += 1;
	}

	int fillheight = a.length -1;

	for (int i = max; i >= 0; i--) {
		for (int j = c[i]; j > 0; j--) {
			b[fillheight--] = i;
		}
	}
	return b;
}
```

Die Methode *countingSort2_opt()* spart sich das Aufsummieren des Hilfsarrays *c*, iteriert direkt über *c* und schreibt den Wert *i* mit der Anzahl *c[i]* in das Ausgabe-Array *b*. Die Anzahl der Arrayzugriffe in der letzten for-Schleife wird somit auf n weiter reduziert.

Szenario:

Ein Array *artikel* mit Artikelnummern von Produkten soll sortiert werden. Die Größe *n* des Arrays *artikel* soll in verschiedenen Messungen 10, 100, 1000 und 10000 Elemente betragen. Ein Messdurchgang dauert 10 Minuten.

```
public static void szenario_counting_sort() {
	int n = 10000;
	int artikel[] = new int[n];

	for(int i = 0; i < n; i++) {
		artikel[i] = 123 + i % n;
	}
```

```
            while(true) {
                  CodeOptimizations.countingSort1(artikel);

                  // separate Messung
                  // CodeOptimizations.countingSort1_opt(artikel);

                  // separate Messung
                  // CodeOptimizations.countingSort2_opt(artikel);
            }
      }
```

n	10	100	1000	10000
countingSort1() Ø-Laufzeit in µs	0,37367	3,97776	38,10428	411,30603
countingSort1_opt() Ø-Laufzeit in µs	0,18216	3,63852	38,20459	375,08063
countingSort2_opt() Ø-Laufzeit in µs	0,19306	3,92295	32,79096	316,90300
countingSort1() Ø-Stromverbrauch µWh	0,00984	0,10541	1,04152	11,03671
countingSort1_opt() Ø-Stromverbrauch in µWh	0,00477	0,09642	1,02516	10,12718
countingSort2_opt() Ø-Stromverbrauch in µWh	0,00512	0,10396	0,87443	8,50356

Tabelle 3.10: Messergebnisse des Szenarios Counting Sort

Anhand der in der Tabelle 3.10 dargestellten Messergebnisse kann man erkennen, dass bei einem Wert für *n* von 10 die Methode *countingSort_opt1()* gegenüber der *countingSort1()*-Methode um ca. 51,2 % schneller und energieeffizienter ist und gegenüber der *countingSort_opt2()*-Methode um ca. 5,6 % schneller und energieeffizienter ist. Bei *n* gleich 1000 zeigt die Messung, dass die *countingSort_opt2()*-Methode um den Faktor 1,16 schneller und energieeffizienter als die beiden Methoden *countingSort1()* und *countingSort_opt1()*. Bei größeren Werten für *n*, z.B. bei 10.000 wird besonders ein Unterschied in der durchschnittlichen Laufzeit und dem Stromverbrauch deutlich. Die *countingSort_opt()*-Methode ist 22,9 % schneller und energieeffizienter, als die *countingSort1()*-Methode und 15,5 % schneller und energieeffizienter als die Methode *countingSort_opt1()*.

Das Reduzieren der Arrayzugriffe hat also einen positiven Einfluss auf die Energieeffizienz und ist aus diesem Grund auch zu empfehlen. Anwendbar ist diese Optimierung in der Praxis dann, wenn sich der Wert des Index bei einem Arrayzugriff in einer Schleife nicht bei jedem Iterationsschritt ändert oder wenn mehrmals in einem Iterationsschritt auf dasselbe Element des Arrays zugegriffen wird. In diesem Fall verringert sich die Anzahl der Arrayzugriffe durch das Verwenden einer temporären Variable. Ist der Wert des Index bei den Arrayzugriffen in der Schleife in jedem Iterationsschritt ein anderer, lohnt sich das Verwenden einer temporären Variable nicht.

Verwendet man zum Reduzieren von Arrayzugriffen mehrere temporäre Variablen, kann der Quellcode schnell schwerer lesbar und zu verstehen werden. Damit wird die Wartbarkeit der Anwendung negativ beeinflusst. Es entstehen durch die zusätzlichen temporären Variablen auch neue Fehlerquellen.

3.3.6. Array kopieren

Zum Kopieren eines Arrays sollte die native Methode *System.arraycopy()* statt eines Schleifenkonstrukts verwendet werden, da diese – in C bzw. Assembler implementiert – performanter ist. Welche Auswirkungen das Verwenden dieser nativen Methode tatsächlich hat, hängt davon ab, wie die native Methode implementiert ist und wie sie die Ressourcen der Plattform nutzt, also z.B. ob effiziente Programmbibliotheken existieren und verwendet werden. [16,21]

Code-Beispiel:

```
public static int[] loop_arraycopy(int[] array, int[] copy, int length) {
	for (int i = 0; i < length; i++) {
		copy[i] = array[i];
	}
	return copy;
}
```

Die Methode *loop_arraycopy()* verwendet eine for-Schleife für das Kopieren des Arrays *array*.

Szenario:

```
public static void szenario_arrayCopy() {
	int n = 10;
	int array[] = new int[n], arrayLength = n;

	for (int i=0; i < n; i++) {
		array[i] = i;
	}

	while (true) {
		int copy1[] = new int[arrayLength];

		CodeOptimizations.loop_arraycopy(array, copy1,
			arrayLength);

		// separate Messung
		// System.arraycopy(array, 0, copy1, 0, arrayLength);
	}
}
```

In dem Szenario sollen Arrays verschiedener Größe *n* kopiert werden. Die Größe *n* soll in verschiedenen Messungen 10, 100, 1000 und 10000 Elemente betragen. Ein Messdurchgang soll 10 Minuten dauern.

n	10	100	1000	10000
loop_arraycopy() Ø-Laufzeit in µs	0,01104	0,03173	0,09068	28,50573
System.arraycopy() Ø-Laufzeit in µs	0,01396	0,03461	0,07208	12,36988
loop_arraycopy() Ø-Stromverbrauch in µWh	0,00029	0,00084	0,00242	0,76965
System.arraycopy() Ø-Stromverbrauch in µWh	0,00037	0,00091	0,00189	0,32368

Tabelle 3.11: Messergebnisse des Szenarios Array kopieren

Bei kleinen Werten für *n* bis 100 zeigen die Messergebnisse in der Tabelle 3.11, dass die Methode *loop_arraycopy()* mit dem Schleifenkonstrukt schneller und damit auch energieeffizienter ist, als die *System.arraycopy()*-Methode. Bei einem Wert für *n* von 10 ist ein Unterschied von einem Faktor 1,26 und bei *n* gleich 100 ein Faktor von 1,09 den Messergebnissen zu entnehmen. Je größer allerdings die Anzahl der Elemente *n* ist, desto besser in der Laufzeit und des Stromverbrauchs wird die *System.arraycopy()*-Methode im Vergleich zur *loop_arraycopy()*-Methode. Bei *n* gleich 1000 ist der Unterschied mit einem Faktor 1,25 und bei *n* gleich 10000 mit einem Faktor 2,3 zugunsten der Methode *System.arraycopy()* zu beziffern.

Es ist also für Arrays bis zu einer Größe von ca. 100 Elementen weiterhin sinnvoll ein Schleifenkonstrukt zum Kopieren zu verwenden. Bei größeren Arrays ist auf jeden Fall zu empfehlen die *System.arraycopy()*-Methode zum Kopieren einzusetzen.

Da die Verwendung der *System.arraycopy()*-Methode weniger Quellcode erfordert, wird die Lesbarkeit und damit die Wartbarkeit der Anwendung verbessert. Eine native Methode zu verwenden, würde die Portierbarkeit einer Anwendung auf andere Plattformen verschlechtern. Die System-Klasse mit der *arraycopy()*-Methode gehört jedoch zur Java Klassenbibliothek und ist für alle Plattformen, für die es eine VM gibt, verfügbar.

3.3.7. Iteratoren

Zum Iterieren über eine Collection oder Ähnliches kann man in Java ein indexbasiertes Schleifenkonstrukt oder einen Iterator verwenden. Es gibt keine pauschale Regel, welche der beiden Methoden sich performanter verhält. Am Beispiel der ArrayList und der LinkedList soll gezeigt werden, wo sich ein Iterator und wo sich ein Schleifenkonstrukt besser eignet. Eine ArrayList benutzt intern ein Array, um die Elemente abzuspeichern, während eine LinkedList eine doppelt verkettete Liste ist. Will man indiziert mit *get(i)* auf ein Element der Liste zugreifen, muss bei der ArrayList nur das Element aus dem internen Array mit dem Index *i* geholt werden, während bei der LinkedList die Elemente bis zum gegebenen Index *i* traversiert werden müssen, um das Element zu finden. Das bedeutet also, dass der indizierte Zugriff auf ein Element in der ArrayList bei konstanter Zeit erfolgt und bei der LinkedList die Zugriffszeit linear steigt, je nachdem wieviele Elemente bis zu dem Index *i* traversiert werden müssen. Bei der Verwendung eines Schleifenkonstrukts zum Iterieren über eine LinkedList, das einen indizierten Zugriff benutzt, entsteht also beim Anfordern des Elements mit dem Index *i* ein Overhead für das Traversieren der Elemente von 0 bis *i - 1*, da im vorherigen Iterationsschritt der gleiche Aufwand für das Element *i - 1* entstanden ist. Ein Iterator hingegen kann bei einer LinkedList in konstanter Zeit das nächste Elemente lesen, da er immer eine Referenz auf das nächste Element in der verketteten Liste speichert und somit der Overhead für das Traversieren entfällt. Ein Ausschnitt aus dem Quellcode der inneren Klasse ListItr von LinkedList zeigt dies:

```
private class ListItr
   implements ListIterator<E>
  {
   private LinkedList.Entry<E> lastReturned = LinkedList.access$000(this.this$0);
```

```
   private LinkedList.Entry<E> next;
   private int nextIndex;
…
   public E next()
   {
    checkForComodification();
    if (this.nextIndex == LinkedList.access$100(this.this$0))
      throw new NoSuchElementException();
    this.lastReturned = this.next;
    this.next = this.next.next;
    this.nextIndex += 1;
    return this.lastReturned.element;
   }
…
```

In dem Feld *next* ist das nächste Element der verketteten Liste gespeichert. Beim Aufruf der *next()*-Methode wird dann u.a. der Nachfolger des *next*-Elements als *next*-Element gesetzt.

Die Anwendung eines Iterators bei einer ArrayList erzeugt aber einen Overhead, der durch zusätzliche Methodenaufrufe von *hasNext()* und *next()* entsteht, die dazu dienen, zu prüfen ob noch ein Element in der Liste ist und um das nächste Element zu lesen. [14,22]

Code-Beispiel:

```
public static void loop_index(List<String> list) {
	String temp = null;
	for(int i = 0, n = list.size(); i < n; i++) {
		temp = list.get(i);
	}
}

public static void loop_iter(List<String> list) {
	String temp = null;
	for(String elem: list) {
		temp = elem;
	}
}
```

Die Methode *loop_index()* iteriert mit einer gewöhnlichen for-Schleife indiziert über die als Parameter *list* übergebene Liste. In der Schleife wird das Element an *i*-ter Position mit der Methode *get(i)* angefordert. Im Gegensatz dazu wird in der Methode *loop_iter()* eine foreach-Schleife benutzt, die intern die Methoden des Iterator-Interface *hasNext()* und *next()* verwendet, was anhand der invokeinterface-Instruktionen 13 und 22 im Bytecode ersichtlich ist:

```
 0 aconst_null
 1 astore_1
 2 aload_0
 3 invokeinterface #149 <java/util/List.iterator> count 1
 8 astore_3
 9 goto 21 (+12)
12 aload_3
13 invokeinterface #153 <java/util/Iterator.next> count 1
```

```
18 astore_2
19 aload_2
20 astore_1
21 aload_3
22 invokeinterface #159 <java/util/Iterator.hasNext> count 1
27 ifne 12 (-15)
30 return
```

Szenario:

Über ein Liste *liste* von Artikelbeschreibungen soll iteriert werden. Die Größe der Liste *liste* soll in verschiedenen Messungen 100, 500 und 1000 betragen. Die Artikelbeschreibungen werden mit einem BufferedReader *in* Zeile für Zeile aus einer Datei *file* eingelesen und dann der Liste *liste* hinzugefügt. Danach werden die beiden Methoden *loop_index()* und *loop_iter()* mit der Liste *liste* als Argument aufgerufen. Ein Messdurchlauf soll 10 Minuten dauern.

```
public static void szenario_loop_iterator() {
	String file = "/home/dradoslav/BA_Tests/Artikelbeschreibungen.txt";

	// separate Messung
	// String file = "/home/dradoslav/
	//     BA_Tests/Artikelbeschreibungen100.txt";

	// separate Messung
	// String file = "/home/dradoslav/
	//     BA_Tests/Artikelbeschreibungen500.txt";

	String line;

	List<String> liste = new ArrayList<String>();

	// separate Messung
	// List<String> liste = new LinkedList<String>();

	try {
		BufferedReader in = new BufferedReader(
			new FileReader(new File(file)));
		while((line = in.readLine()) != null) {
			liste.add(line);
		}
	} catch (IOException e) {}

	while(true) {
		CodeOptimizations.loop_index(liste);
		CodeOptimizations.loop_iter(liste);
	}
}
```

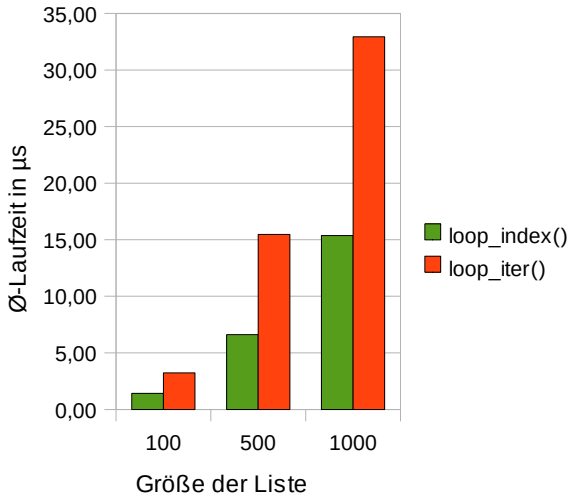

Abb. 3.14: Ø-Laufzeit der ArrayList Iteration

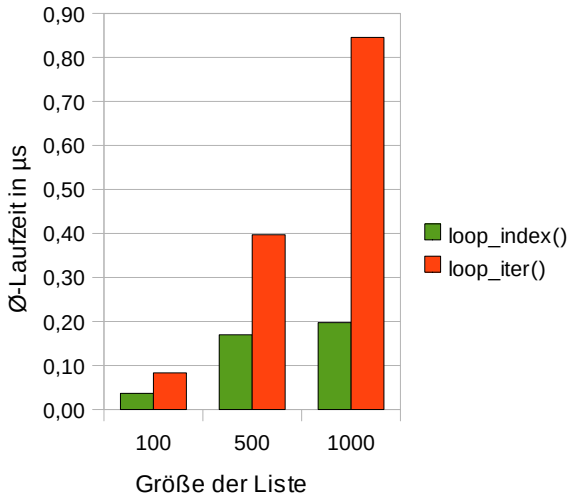

Abb. 3.15: Ø-Stromverbrauch mit ArrayList

Den in den Abb. 3.14 und Abb. 3.15 dargestellten Messergebnissen ist zu entnehmen, dass die Iteration über eine ArrayList mit einer foreach-Schleife, die in der *loop_iter()*-Methode verwendet wird, für alle Listengrößen weniger durchschnittliche Laufzeit und Stromverbrauch verursacht.

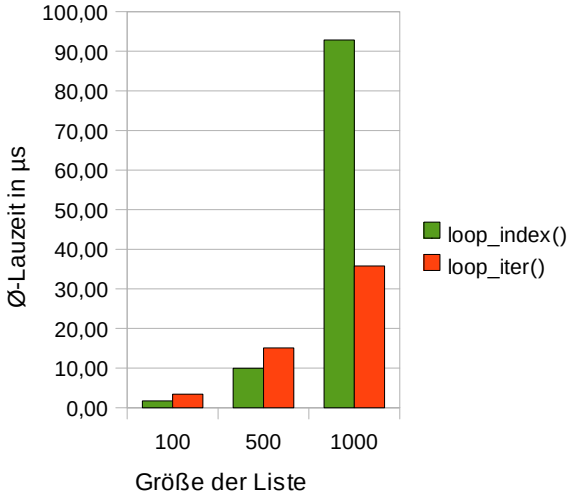

Abb. 3.16: Ø-Laufzeit der LinkedList Iteration

Abb. 3.17: Ø-Stromverbrauch mit LinkedList

Die Abb. 3.16 und Abb. 3.17 zeigen, dass bei der LinkedList bis zu einer Listengröße von 500 die Methode *loop_index()* schneller und energieeffizienter ist, als die *loop_iter()*-Methode. Bei 500 Elementen in der Liste ist der Unterschied mit einem Faktor 1,5 zugunsten der *loop_index()*-Methode zu beziffern, während bei 1000 Elementen die Methode *loop_iter()* um einen Faktor 2,6 deutlich schneller und energieeffizienter ist, als die *loop_index()*-Methode.

Es ist also aufgrund der Messergebnisse der Energieeffizienz sinnvoll für das Iterieren über eine ArrayList keinen Iterator zu verwenden. Bei einer LinkedList hingegen ist das Verwenden eines Iterators laut der Messergebnisse ab einer Anzahl von 1000 Elementen lohnenswert. Wenn man in der Praxis mit Variablen vom Typ des List-Interface arbeitet, könnte es aufwändig werden immer die beste Art der Iteration für die jeweiligen List-Objekte auszuwählen, da erst zur Laufzeit wirklich feststeht, von welcher Klasse das List-Objekt ist. Man müsste dann vor jeder Iteration über eine Liste eine Typüberprüfung durchführen, was zwar nicht wesentlich mehr Zeit kostet, aber was den Quellcode ziemlich unleserlich und schwieriger zu verstehen macht. Damit würde die Wartbarkeit der Anwendung verschlechtert werden. Ebenfalls würde man damit Abhängigkeiten von konkreten Klassen produzieren, die man mit durch die Verwendung des List-Interface eigentlich verhindern will. Das würde bspw. die Wiederverwendbarkeit und die Übertragbarkeit der Anwendung verschlechtern.

3.4. Strings

Da Strings in Java *immutable Objects* sind, d.h. Objekte, die nach ihrer Erzeugung nicht mehr verändert werden können, bewirken alle Methoden, die einen String verändern sollen, wie z.B. substring() oder concat(), dass ein neuer String mit den veränderten Eigenschaften erzeugt und returniert werden muss. Dieses wiederum erzeugt einen gewissen Overhead, da nicht einfach nur das interne char-Array geändert wird, sondern eine Objekterzeugung erfolgt. Im Abschnitt 3.2. wurde bereits geklärt, warum die Erzeugung von Objekten viel Zeit kosten kann. Den Sinn von *immutable Objects* zu diskutieren ist jedoch nicht Gegenstand dieser Arbeit.

3.4.1. Strings konkatenieren

Typischerweise wird zum Konkatenieren eines Strings an einen anderen der +-Operator verwendet. Der Compiler wandelt einen Ausdruck der Form String1 + String2 um, indem er ein StringBuffer-Objekt oder ein StringBuilder-Objekt, das mit dem ersten String initialisiert ist erzeugt, mit der Methode *append()* den zweiten String anhängt und dann mit der Methode *toString()* wieder in einen String umwandelt. Der Unterschied zwischen StringBuffer und StringBuilder liegt im Großen und Ganzen darin, dass der StringBuffer im Gegensatz zum StringBuilder synchronisiert ist. Solange String1 und String2 keine Literale sind – die bereits vom Compiler zu einem einzigen String-Literal verkettet werden können – und mehrfach weitere Strings angehängt werden, sollte statt dem +-Operator direkt ein StringBuffer oder StringBuilder verwendet werden, da ansonsten ein Overhead entsteht, weil für n Konkatenationen 2 * (n-1) Objekterzeugungen stattfinden, während bei der Verwendung von StringBuffern bzw. StringBuilder bei n Konkatenationen nur 2 Objekterzeugungen notwendig sind. Das StringBuffer- bzw. StringBuilder-Objekt sollte mit einer angemessenen Kapazität initialisiert werden, ähnlich wie im Absatz 3.2.2. mit Collection-Objekten verfahren wurde. Die Kapazität sollte besser zu groß, als zu klein gewählt werden, da man sich dadurch

unnötige Umkopieroperationen des im StringBuffer bzw. StringBuilder intern verwendeten char-Arrays spart. Standardmäßig hat das interne char-Array eine Kapazität von 16 Elementen.
Auch das Konkatenieren von Strings mit der *concat()*-Methode aus der String-Klasse sollte durch eine äquivalente Operation mit einem StringBuffer bzw. StringBuilder ersetzt werden. Die *concat()*-Methode ist zwar effizienter implementiert, als der +-Operator, denn es werden direkt die beiden internen char-Arrays zu einem neuen char-Array aneinander gehängt, trotzdem wird bei jedem Aufruf ein neuer String erzeugt, dem das neue char-Array übergeben wird. Bei mehrfach aufeinander folgenden Aufrufen von *concat()* kann man sich durch Verwendung eines StringBuffers oder StringBuilders den Overhead für die Objekterzeugung bei jedem Aufruf sparen. [14]

Code-Beispiel:

```
public static void stringAppend_concat(String x, int n) {
	String s = "";
	for(int i = 0; i < n; i++) {
		s.concat(x);
	}
}

public static void stringAppend_plus(String x, int n) {
	String s = "";
	for(int i = 0; i < n; i++) {
		s += x;
	}
}

public static void stringAppend_StringBuilder(String x, int n) {
	StringBuilder s = new StringBuilder(x.length() * n);
	for(int i = 0; i < n; i++) {
		s.append(x);
	}
	s.toString();
}
```

In den drei o.g. Methoden wird jeweils *n* mal der übergebene String *x* konkateniert. Die Methode *stringAppend_concat()* benutzt dafür die von String angebotene Methode *concat()*. In der Methode *stringAppend_plus()* wird der +-Operator zum Konkatenieren verwendet. Die Methode *stringAppend_StringBuilder()* verwendet einen mit einer Kapazität von *c* initialisierten StringBuilder und dessen *append()*-Methode.

Szenario:

Das Szenario besteht darin, in einer Endlosschleife n-mal einen String *s* mit sich selbst zu konkatenieren. Ein Messdurchgang soll 10 Minuten dauern.

```
public static void test_StringAppend() {
	int n = 200;

	String s = "Das ist ein viel längerer String";
```

```
            while(true) {
                  CodeOptimizations.stringAppend_concat(s, n);
                  CodeOptimizations.stringAppend_plus(s, n);
                  CodeOptimizations.stringAppend_StringBuilder(s, n);
            }
      }
```

n	5	10	200	500	1000
stringAppend_plus() Ø-Laufzeit in µs	106,35	227,99	46.068,96	265.219,07	1.055.688,26
stringAppend_concat() Ø-Laufzeit in µs	26,88	50,91	890,68	2.511,72	5.441,39
stringAppend_StringBuilder() Ø-Laufzeit in µs	39,11	62,38	778,20	2.772,74	5.369,21
stringAppend_plus() Ø-Stromverbrauch in µWh	2,75	5,89	1.197,79	6.895,70	27.447,89
stringAppend_concat() Ø-Stromverbrauch in µWh	0,69	1,32	23,16	65,30	141,48
stringAppend_StringBuilder() Ø-Stromverbrauch in µWh	1,01	1,61	20,23	72,09	139,60

Tabelle 3.12 Messergebnisse des Szenarios Strings konkatenieren

Die Tabelle 3.12 zeigt die Messergebnisse des Szenarios, aus denen ersichtlich wird, dass die Konkatenation mit dem +-Operator schon bei *n* gleich 5 eine annähernd 3-fach höhere durchschnittliche Laufzeit und Stromverbrauch hat, als mit dem StringBuilder. Der Unterschied der Verwendung des +-Operators zur Konkatenation mit der *concat()*-Methode beträgt sogar annähernd den Faktor 4 in der Laufzeit und im Stromverbrauch. Bei einem *n* gleich 1000 ist das Konkatenieren mit dem +-Operator im Gegensatz zur *concat()*-Methode und dem StringBuilder fast 200-mal langsamer und energieineffizienter.
Bei kleineren Werten für *n* ist zu beobachten, dass die *concat()*-Methode schneller und energieeffizienter als der StringBuilder ist, was sich jedoch bei steigenden Werten für *n* umkehrt. Bei 200 Konkatenationen ist die durchschnittliche Laufzeit und der Stromverbrauch des StringBuilders um 14,5 % geringer, als bei der *concat()*-Methode.

Ob diese Optimierung einen positiven Unterschied in der Energieeffizienz und der Laufzeit des Programms bewirkt, hängt also unter Anderem davon ab, wie lang und wieviele zu verkettende Strings vorhanden sind. Des Weiteren ist die Kapazität, mit welcher ein StringBuffer- bzw. StringBuilder-Objekt initialisiert wird, entscheidend. Dauert nämlich dessen Erzeugung länger, als die Erzeugung der neuen Strings in der Konkatenationskette, lohnt sich der ganze Aufwand nicht. Gemessen wurden die Kosten für die Erzeugung eines StringBuffer- bzw. StringBuilder-Objekts mit der initialen Kapazität nicht, da hier Querverbindungen zum Abschnitt 3.2.2. gezogen werden können.

Die Qualität der Anwendung wird verschlechtert, weil sich die Lesbarkeit des Quellcodes und daraus resultierend die Wartbarkeit der Anwendung dadurch verschlechtert, dass statt des einfachen +-Operator zum Konkatenieren zweier Strings, plötzlich mehrere Zeilen

zusätzlichen Quellcodes notwendig werden. Sollen also mehrere Strings verkettet werden, wird der Quellcode schnell ziemlich unübersichtlich und es entstehen zusätzliche Fehlerquellen. Weiterhin muss immer daran gedacht werden den StringBuffer oder StringBuilder mit einer ausreichenden Kapazität zu initialisieren, um unnötige Umkopieroperationen des internen char-Arrays zu vermeiden.

3.4.2. Strings einfügen

Einen String in einen anderen String mit den von der Klasse String angebotenen Methoden *substring()* und *concat()* einzufügen, erzeugt durch das nötigwerdende Kopieren der Strings und der damit notwendigen Objekterzeugung einen Overhead. Mit einem StringBuffer bzw. StringBuilder kann man sich zumindest einen Teil der Kosten für die Objekterzeugung sparen, da nur 2 Objekte erzeugt werden müssen, während bei der Verwendung der o.g. String-Methoden 4 Objekte erzeugt werden. [14]

Code-Beispiel:

```
	public static void stringInsert_String(String s, String i, int position) {
		String sub1 = s.substring(0, position),
		sub2 = s.substring(position),
		result = sub1.concat(i).concat(sub2);
	}

	public static void stringInsert_StringBuffer(String s, String i, int
position) {
		StringBuffer sb = new StringBuffer(s);
		String result = sb.insert(position, i).toString();
	}
```

Die Methode *stringInsert_String()* teilt den übergebenen String *s* mit *substring()* in zwei Strings *sub1* und *sub2* auf. Danach wird der übergebene String *i* und *sub2* mit *concat()* an *sub1* angehängt. In diesem Fall wird durch die Methode *substring()* jeweils ein neues String-Objekt erzeugt. Durch das zweimalige Konkatenieren mit *concat()* werden noch zwei String-Objekte erzeugt. Im Gegensatz dazu wird in der Methode *stringInsert_StringBuffer()* ein StringBuffer – mit dem String *s* initialisiert – erzeugt, der übergebene String *i* mit *insert()* an der vorgesehenen Stelle eingefügt und mit *toString()* wieder zu einem String umgewandelt. Es wird hier ein StringBuffer-Objekt und ein String-Objekt durch einen Aufruf von *toString()* erzeugt.

Szenario

```
		public static void test_StringInsert() {
			String s = "Hier wird eingefügt",
				i = "ein String ";
			int n = 10;

			// separate Messung
			// String s = "ac",
```

```
//         i = "b";
// int n = 1;

while(true) {
	CodeOptimizations.stringInsert_String(s,i,n);
	CodeOptimizations.stringInsert_StringBuffer(s,i,n);
}
}
```

Wieder soll in einer Endlosschleife ein String *i* in einen String *s* an der Position *n* eingefügt werden. Dazu wird in der ersten Messung der String *“einString “* in den String *“Hier wird eingefügt“* an der Position 10 eingefügt. Das Ergebnis ist der String *“Hier wird ein String eingefügt“*. In der zweiten Messung soll der String *“b“* in den String *“ac“* an der Position 1 eingefügt werden. Hier ist das Ergebnis dann der String *“abc“*. Ein Messdurchgang dauert 10 Minuten.

Resultat-String	**“Hier wird ein String eingefügt“**	**“abc“**
stringInsert_String() Ø-Laufzeit in µs	0,14619	20,88681
stringInsert_StringBuffer() Ø-Laufzeit in µs	0,11472	16,48286
stringInsert_String() Ø-Stromverbrauch in µWh	0,00380	0,53958
stringInsert_StringBuffer() Ø-Stromverbrauch in µWh	0,00298	0,42581

Tabelle 3.13 Messergebnisse des Szenarios String einfügen

Laut den Messergebnissen in der Tabelle 3.13, ist das erste Beispiel mit dem String *“Hier wird ein String eingefügt“* unter Verwendung eines StringBuffers um 21,5 % schneller und energieeffizienter, als bei der Verwendung der String-Methoden. Das Beispiel mit dem String *“abc“* zeigt ebenfalls einen Unterschied in der durchschnittlichen Laufzeit und des Stromverbrauchs von 21 % zugunsten der Umsetzung mit dem StringBuffer.
Aufgrund der Ergebnisse wird offensichtlich, dass selbst bei kleinen Strings das Einfügen mit dem StringBuffer schneller und energieeffizienter erfolgen kann. Da der StringBuffer im Gegensatz zum StringBuilder synchronisiert ist, kann aufgrund der Ergebnisse davon ausgegangen werden, dass die Umsetzung mit einem StringBuilder mindestens genau so schnell und energieeffizient ist, wie mit dem StringBuffer. Die Verwendung von StringBuffer bzw. StringBuilder zum Einfügen von Strings ist also zu empfehlen.
Auch die Qualität der Anwendung kann durch die Verwendung StringBuffer bzw. StringBuilder verbessert werden, da weniger Quellcode im Gegensatz zur Umsetzung mit den String-Methoden geschrieben werden muss. Das Beispiel zeigt, mit dem String-Methoden sind 4 Methodenaufrufe notwendig, während mit dem StringBuffer nur 2 Methodenaufrufe notwendig sind. Dadurch verbessert sich die Wartbarkeit der Anwendung, weil der Quellcode besser zu lesen und zu verstehen ist. Ebenfalls reduzieren sich die potentiellen Fehlerquellen durch die geringere Anzahl von Methodenaufrufen.

3.4.3. Strings vergleichen

Will man zwei Strings auf Wertgleichheit vergleichen, verwendet man gewöhnlich die *equals()*-Methode aus der Klasse String. Die *equals()*-Methode ist so implementiert, dass sie zuerst den einfachsten Fall prüft, ob das übergebene Objekt identisch mit sich selbst ist. Ist dies nicht der Fall, wird geprüft, ob das Objekt überhaupt eine String-Instanz ist und sollte das nicht zutreffen, werden die internen char-Arrays verglichen. Ein Blick auf den Quellcode der *equals()*-Methode der Klasse String beweist das:

```
public boolean equals(Object paramObject)
{
 if (this == paramObject)
  return true;
 if (paramObject instanceof String)
 {
  String str = (String)paramObject;
  int i = this.count;
  if (i == str.count)
  {
   char[] arrayOfChar1 = this.value;
   char[] arrayOfChar2 = str.value;
   int j = this.offset;
   int k = str.offset;
   do
    if (i-- == 0)
     break label84;
   while (arrayOfChar1[(j++)] == arrayOfChar2[(k++)]);
   return false;
   label84: return true;
  }
 }
 return false;
}
```

Das bedeutet, dass es am günstigsten ist, wenn die beiden zu vergleichenden Strings identisch sind und am schlechtesten, wenn beide Strings zwar inhaltlich gleich, aber nicht identisch sind. Eine Besonderheit in Java ist, dass die VM einen Konstantenpool unterhält, in dem alle Stringliterale gespeichert sind. Zwei wertgleiche aber nicht identische String-Objekte haben intern eine Referenz auf dasselbe Stringliteral. Arbeitet man nicht nur mit Stringliteralen, sondern mit zur Laufzeit erzeugten Strings, kann man sich beim Vergleich auf Wertgleichheit, das Wissen über die *equals()*-Methode zu Nutze machen und sich zu all diesen Strings, die Referenz des Stringliterals im Konstantenpool mit der Methode *intern()* holen und danach beide Strings vergleichen. [14,16]

Code-Beispiel:

```
public static void stringComp(String s1, String s2) {
	s1.equals(s2);
}

public static void stringComp_intern(String s1, String s2) {

	s1.intern().equals(s2.intern());
}
```

Die Methoden *stringComp()* und *stringComp_intern()* führen jeweils einen Vergleich der beiden übergebenen Strings *s1* und *s2* mit der *equals()*-Methode durch. In der Methode *stringComp_intern()* wird jedoch von beiden Strings die *intern()*-Methode aufgerufen, um zu erreichen, dass in der *equals()*-Methode bei wertgleichen aber nicht identischen String-Objekten ein Identitätsvergleich stattfindet.

Szenario:

Das Szenario sieht vor, dass in einer Endlosschleife jeweils 2 Strings *s1* und *s2* miteinander verglichen werden. Die Strings sollen in verschiedenen Messungen unterschiedliche Stringlängen von 18, 43, 116 und 1000 haben. Ein Messdurchgang soll 10 Minuten dauern.

```
public static void test_StringCompare() {
	String s1 = "Das ist ein String",
	s2 = new String("Das ist ein String");

	// separate Messung
	// String s1 = "Das ist ein String, etwas länger als vorher",
	// s2 = new String("Das ist ein String, etwas länger als
		vorher");

	// separate Messung
	// String s1 = "Und noch eiiiiiiiiiiiiin
	//     gaaaaaaaaaaaaaaaaaaaaaaaaaanz
	//     laaaaaaaaaaaaaaaaaaaaaaaaanger
	//     Striiiiiiiiiiiiiiiiiiiiiiiing",
	// s2 = new String("Und noch eiiiiiiiiiiiiin
	//     gaaaaaaaaaaaaaaaaaaaaaaaaaanz
	//     laaaaaaaaaaaaaaaaaaaaaaaaanger
	//     Striiiiiiiiiiiiiiiiiiiiiiiing");

	// separate Messung
	// for(int i = 0; i < 1000; i++) {
	//           s1 += "a";
	//           s2 += new String("a");
	// }

	while(true) {
		CodeOptimizations.stringComp(s1,s2);
		CodeOptimizations.stringComp_intern(s1,s2);
	}
}
```

Stringlänge	18	43	116	1000
stringComp() Ø-Laufzeit in µs	0,02127	0,02163	0,02609	2,42225
stringComp_intern() Ø-Laufzeit in µs	0,06029	0,05821	0,07156	4,91039
stringComp() Ø-Stromverbrauch in µWh	0,00055	0,00056	0,00068	0,06500
stringComp_intern() Ø-Stromverbrauch in µWh	0,00155	0,00149	0,00186	0,13176

Tabelle 3.14 Messergebnisse des Szenarios String Vergleiche

Entgegen den Erwartungen, zeigen die Messergebnisse in der Tabelle 3.14, dass die Methode *stringComp_intern()* bei allen Stringlängen langsamer und energieineffizienter als die *stringComp()*-Methode ist. Das ist vermutlich auf den zweifachen Aufruf der *intern()*-Methode zurückzuführen. Die Einsparungen der Laufzeit durch den Identitätsvergleich sind offensichtlich geringer, als die Kosten für den Aufruf der *intern()*-Methode. Aus diesem Grund ist von der Anwendung dieser vermeintlichen Optimierung abzuraten.

3.5. Synchronized

Die Verwendung von synchronized Methoden oder Blöcken sollte an Stellen, an denen Synchronisation nicht notwendig ist, vermieden werden, da dadurch ein Overhead entsteht. Das ist z.B. der Fall, wenn Operationen auf einem immutable Object ausgeführt werden, da dieses nach der Erzeugung nicht mehr geändert werden kann. Zwei Threads, die parallel auf das immutable Object zugreifen, können somit keine kritischen Veränderungen am Object durchführen, die dazu führen, dass der jeweils andere Thread mit falschen, inkonsistenten Werten arbeitet. Der Grund für den Overhead ist, dass synchronized Methoden und Blöcke in der VM dazu führen, dass der aktuelle Thread einen sogenannten Monitor anfordern muss, in den nur er alleine eintreten und den Code dort ausführen kann. Nach der Ausführung muss der Thread den Monitor wieder verlassen. [10,14]

Code-Beispiel:

```
public class SynchronizedStringWrapper {
	private final String internal_String;

	public SynchronizedStringWrapper(String s) {
		this.internal_String = s;
	}

	public synchronized SynchronizedStringWrapper toUppercase() {
		return new SynchronizedStringWrapper(
			internal_String.toUpperCase());
	}
}

public class StringWrapper {
	private final String internal_String;

	public StringWrapper(String s) {
		this.internal_String = s;
	}

	public StringWrapper toUppercase() {
		return new StringWrapper(internal_String.toUpperCase());
	}
}
```

In der Klasse SynchronizedStringWrapper wird die Methode *toUppercase()* der Klasse String gewrappt und synchronisiert, indem der Aufruf dieser Methode an String delegiert wird und mit dem resultierenden String wieder ein neues SynchronizedStringWrapper-Objekt erzeugt

wird. Im Gegensatz dazu verwendet StringWrapper keine Synchronisation in der *toUppercase()*-Methode, da die Instanzvariable *internal_String* jeweils als Konstante deklariert ist und ein String-Objekt referenziert, das *immutable* ist.

Szenario:

In dem Szenario soll von jeweils einem StringWrapper- und einem SynchronizedStringWrapper-Objekt die *toUppercase()*-Methode in einer Endlosschleife aufgerufen werden. Dabei sollen separate Messungen mit einem und mit zwei Threads gemacht werden. Ein Messdurchgang soll 10 Minuten dauern.

```
public static void szenario_SynchronizedMethod() {
	String x = "d";
	final StringWrapper a = new StringWrapper(x);
	final SynchronizedStringWrapper sa = new
				SynchronizedStringWrapper(x);

	Thread t1 = new Thread("t1") {
		@Override
		public void run() {
			while(true) {
				sa.toUppercase();
				a.toUppercase();
			}
		}
	};

	// separate Messung
	// Thread t2 = new Thread("t2") {
	//	@Override
	//	public void run() {
	//		while(true) {
	//			sa.toUppercase();
	//			a.toUppercase();
	//		}
	//	}
	//};

	t1.start();

	// separate Messung
	//t2.start();
}
```

Thread Anzahl	1	2
StringWrapper.toUppercase() Ø-Laufzeit in µs	0,07411	8,06006
SynchronizedStringWrapper.toUppercase() Ø-Laufzeit in µs	0,07856	0,96238
StringWrapper.toUppercase() Ø-Stromverbrauch in µWh	0,00190	0,20822
SynchronizedStringWrapper.toUppercase() Ø-Stromverbrauch in µWh	0,00202	0,02486

Tabelle 3.15 Messergebnisse des Szenarios synchronisierte Methoden

Während die Methode *toUppercase()* der StringWrapper-Klasse laut den Messergebnissen in der Tabelle 3.15 bei einem Thread um einen Faktor 1,06 schneller und energieeffizienter im Gegensatz zur SynchronizedStringWrapper-Klasse ist, kann man bei zwei Thread einen Unterschied von einem Faktor 8,3 zugunsten der *toUppercase()*-Methode der SynchronizedStringWrapper-Klasse feststellen.

Warum an dieser Stelle die synchronisierte Version der Methode *toUppercase()* bei zwei Threads so deutlich schneller ist, könnte durch Compiler- oder VM interne Optimierungen verursacht worden sein.

Eine weitere Empfehlung aus der Fachliteratur im Umgang mit Synchronisation ist z.B., dass nicht ganze Methoden als synchronized deklariert werden, sondern nur die Codeabschnitte, die tatsächlich synchronisiert werden müssen. Dadurch erreicht man, dass wenn zwei Threads gleichzeitig auf eine Methode zugreifen wollen, der eine Thread nicht solange warten muss, bis der andere Thread die komplette Methode abgearbeitet hat, sondern jeweils nur bis die kritischen Codeabschnitte verlassen worden sind. Das lohnt sich vor allem dann, wenn viele oder vergleichsweise aufwändige unkritische Operationen in der Methode ausgeführt werden müssen. [10,14]

Code-Beispiel:

```
public class SynchronizedAccount {
	private int balance = 0;
	private Logger logger;

	public SynchronizedAccount(Logger log) {
		this.logger = log;
	}

	public synchronized void deposit(int d) throws Exception {

		if (d < 0) throw new Exception("Ungültiger Betrag");

		logger.log("Start Transaktion: " + d + " " +
			System.currentTimeMillis());
		balance += d;
		logger.log("Einzahlung: " + d + " " +
			System.currentTimeMillis());
		logger.log("Ende Transaktion: " + d + " " +
			System.currentTimeMillis());
	}
}

public class BlockSynchronizedAccount {
	private int balance = 0;
	private Logger logger;

	public BlockSynchronizedAccount(Logger log) {
		this.logger = log;
	}

	public void deposit(int d) throws Exception {
		if (d < 0) throw new Exception("Ungültiger Betrag");
```

```
                        logger.log("Start Transaktion: " + d + " " +
                                System.currentTimeMillis());
                        synchronized (this) {
                                balance += d;
                        }
                        logger.log("Einzahlung: " + d + " " +
                                System.currentTimeMillis());
                        logger.log("Ende Transaktion: " + d + " " +
                                System.currentTimeMillis());
                }
        }
```

In der Klasse SynchronizedAccount wird in der als synchronized deklarierten Methode *deposit()* die Instanzvariable *balance* um den Übergabeparameter *d* erhöht, was eine kritische Operation darstellt. Jeweils davor und danach finden nicht kritische Operationen statt, die in diesem Fall das Senden einer Nachricht an ein Logger-Objekt mit *log()*-Methode. Der Logger-Objekt speichert die Nachricht dann in einer *log*-Datei. Die Logger-Klasse ist für das Verständnis des Beispiels nicht von Bedeutung und ist deshalb nur im Anhang abgebildet. Anders in der Klasse BlockSynchronizedAccount, in der in der Methode *deposit()* nur die kritische Operation – die Zuweisung der Variable *balance* – mit einem Block synchronisiert wird.

Szenario:

Das Szenario sieht vor, dass es eine Instanz der Klasse SynchronizedAccount *sa* und eine Instanz der Klasse BlockSynchronizedAccount *ba* gibt. Den beiden Objekte wird bei der Erzeugung eine Referenz auf ein vorher erzeugtes Logger-Objekt *logger* übergeben. Zwei Threads sollen dann in ihrer *run()*-Methode in einer Endlosschleife abwechselnd die *deposit()*-Methoden von *sa* und *ba* aufrufen. Ein Messdurchgang dauert 10 Minuten.

```
public static void szenario_SynchronizedBlock() {
        Logger logger = null;
        try {
                logger = new Logger("/dev/null");
        } catch (FileNotFoundException e1) {}

        final SynchronizedAccount sa = new SynchronizedAccount(logger);
        final BlockSynchronizedAccount ba = new
                BlockSynchronizedAccount(logger);

        Thread t1 = new Thread("t1") {
                @Override
                public void run() {
                        while(true) {
                                try {
                                        ba.deposit(5);
                                        sa.deposit(5);
                                } catch (Exception e) {}
                        }
                }
        };

        Thread t2 = new Thread("t2") {
                @Override
                public void run() {
```

```
                        while(true) {
                              try {
                                    ba.deposit(5);
                                    sa.deposit(5);
                              } catch (Exception e) {}
                        }
                  }
            };

            t1.start();
            t2.start();
      }
```

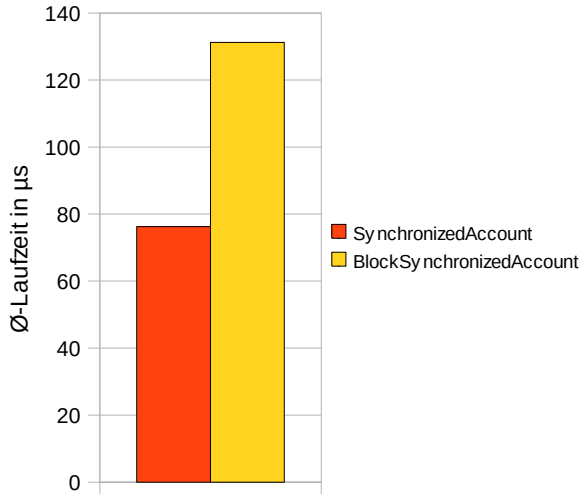

Abb. 3.18: Ø-Laufzeit der deposit()-Methoden

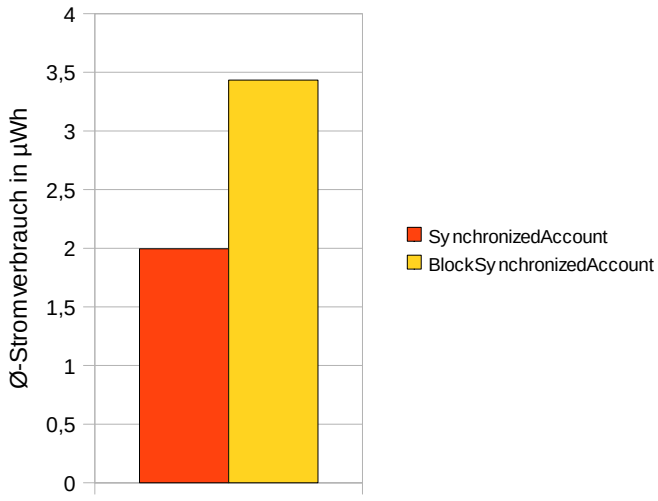

Abb. 3.19: Ø-Stromverbrauch der deposit()-Methoden

Die Messergebnisse in den Abb. 3.18 und Abb. 3.19 zeigen, ähnlich wie die Messergebnisse in der Tabelle 3.15, dass die mit einem Synchronisationsblock versehene Methode *deposit()* in der BlockSynchronizedAccount-Klasse um einen Faktor 1,7 langsamer und energieineffizienter ist als die *deposit()*-Methode in der SynchronizedAccount-Klasse und damit die Erwartungen, durch einen Synchronisationsblock eine kürzere Laufzeit und höhere Energieeffizienz zu erzielen, nicht erfüllt wurden.

Auch an dieser Stelle könnten die unerwarteten Auswirkungen eines Synchronisationsblocks von Compiler- oder VM internen Optimierungen hervorgerufen werden.
Abschließend ist festzustellen, dass sich die in diesem Abschnitt genannten Optimierungen in dieser Konfiguration nicht eignen die Energieeffizienz zu steigern und diese sogar negativ beeinflussen.

3.6. Eingabe und Ausgabe

E/A-Geräte stellen traditionell einen Flaschenhals für den Datenfluss dar, da sie in den meisten Fällen eine mechanische Verbindung über einen E/A-Controller und eine langsamere Übertragungsgeschwindigkeit und Reaktionszeit zu dem Rechner haben, als fest verdrahtete und mit einem schnelleren Bus verbundene Komponenten, wie z.B. der Hauptspeicher. Dieses

ist natürlich nicht nur abhängig von der Geschwindigkeit des Busses mit dem das E/A-Gerät mit dem Rechner verbunden ist, sondern auch davon, wie hoch die Lese- und Schreibgeschwindigkeit des Geräts selber ist. Externe Speicherzugriffe z.B. auf eine Festplatte können einige 1000 mal länger dauern, als Zugriffe auf den RAM-Speicher. Deshalb sollte man versuchen Operationen mit E/A-Geräten mit so wenig Zugriffen, wie möglich zu realisieren. Zwei Strategien, die sich unter Anderem damit beschäftigen, sind das Puffern und Cachen von Daten.

3.6.1. Puffer

Um die Anzahl der Zugriffe auf ein E/A-Gerät zu reduzieren, kann man anstatt vieler Zugriffe, die jeweils ein Byte anfordern, einen Zugriff bevorzugen, der dann eine größere Anzahl an Bytes anfordert und diese Bytes dann in einem Puffer im Hauptspeicher zwischenspeichert. Weiterhin kann man je nachdem, wie groß der Hauptspeicher ist und wieviele Bytes insgesamt gelesen oder geschrieben werden sollen, die Größe des Puffers variieren. Für den Puffer muss Speicher allokiert werden, was Zeit beim Erzeugen der Datenstruktur des Puffers und damit des Objekts, das den Puffer beinhaltet, kostet. Deshalb sollte so vorgegangen werden, dass der Puffer vergleichsweise größer gewählt werden sollte, wenn viele Bytes transferiert werden, weil man dadurch die Anzahl der Zugriffe reduziert. Wenn eine vergleichsweise kleinere Menge an Bytes transferiert wird, sollte der Puffer auch kleiner gewählt werden, da sonst ein Overhead für das Anlegen des Puffers entsteht. Natürlich kann man die Puffergröße nicht willkürlich nach oben hin steigern, sondern muss auch die in der Java VM konfigurierte Größe des Heap-Speichers und auf physikalischer Ebene die Größe des zur Verfügung stehenden Hauptspeichers bei der Festlegung berücksichtigen. Weiterhin kommt hinzu, dass die Garbage Collection bei einem großen Puffer, der viel Speicher im Heap belegt, öfter ausgeführt werden muss, damit freier Speicher für neue Objekte vorhanden ist. Hinsichtlich dieser Einschränkungen ist es sicherlich sinnvoll die Puffergröße dynamisch, anhand der Größe des freien Heap-Speichers und der zu übertragenden Menge an Bytes, festzulegen, sofern diese dynamische Berechnung nicht mehr Zeit in Anspruch nimmt, als die zusätzlichen Zugriffe auf das E/A-Gerät. Damit wird auch das Problem eines statischen Puffers gelöst, wenn dieser größer ist als die Anzahl der zu übertragenden Bytes und die Erzeugung der Datenstruktur des Puffers mehr Zeit, als eigentlich nötig ist, brauchen würde. Der Einfachheit des folgenden Beispiels halber, wird aber nur der statische Puffer untersucht. [14,16]

Code-Beispiel:

```
public static void copy_unbuffered(File source, File target) throws
IOException {
	InputStream in = new FileInputStream(source);
	OutputStream out = new FileOutputStream(target);
	int b;
	while((b = in.read()) != -1) {
		out.write(b);
	}
	in.close();
	out.close();
```

```
}

public static void copy_buffered(File source, File target, int bufferSize)
throws       IOException {
       InputStream in = new FileInputStream(source);
       OutputStream out = new FileOutputStream(target);
       int length;
       byte buffer[] = new byte[bufferSize];
       while((length = in.read(buffer)) != -1) {
              out.write(buffer, 0, length);
       }
       in.close();
       out.close();
}
```

Die Methode *copy_unbuffered()* kopiert eine Quelldatei *source*, indem immer ein Byte von einem FileInputStream *in* gelesen wird und dann mit einem FileOutputStream *out* in eine Zieldatei *target* geschrieben wird. Anders in der Methode *copy_buffered()* – dort wird ein Puffer *buffer* vom Typ byte angelegt. Zuerst wird immer der Puffer durch das Lesen des FileInputStreams *in* gefüllt, dann werden die gepufferten Bytes in den FileOutputStream *out* geschrieben.

Szenario:

Alle Dateien eines Quellverzeichnis *dirSource* sollen in ein Zielverzeichnis *dirTarget* kopiert werden. Der Inhalt des Verzeichnis *dirSource* besteht aus Dateien unterschiedlicher Größe, von 1 KB bis zu 50 MB. Insgesamt sind in dem Verzeichnis *dirSource* 113 Dateien mit einer Gesamtgröße von 100 MB. Die Größe des Puffers wird auf 512 KB festgelegt. Ein Messdurchgang soll 60 Minuten dauern.

```
public static void szenario_FileCopy() {
       String dirSource = "/home/dradoslav/BA_Tests/files",
              dirTarget = "/home/dradoslav/BA_Tests/files_target";

       List<File> sourceList = new ArrayList<File>();

       for(File f: (new File(dirSource).listFiles())) {
              sourceList.add(f);
       }

       while(true) {
              CodeOptimizations.copyFiles(sourceList, dirTarget);
       }
}
```

Für das Szenario wird noch die folgende Methode *copyFiles()* benötigt, in der über die übergebene Liste von Dateien *files* iteriert wird und die o.g. Methoden zum Kopieren angewendet werden.

```
public static void copyFiles(List<File> files, String targetDir) {
       try {
              for (File f : files) {
                     FileCopier.copy_unbuffered(f,
                            new File(targetDir + f.getName()));
```

```
                        // separate Messung
                        // FileCopier.copy_buffered1(f,
                              new File(targetDir + f.getName()), 512);
                    }
                } catch (IOException e) {}
            }
```

Weiterhin sollen noch Messungen mit unterschiedlichen Puffergrößen stattfinden. Als weitere Puffergrößen werden 64 KB, 128 KB und 1024 KB verwendet.

Puffergröße	kein Puffer	64 KB	128 KB	512 KB	1024 KB
copyFiles() Ø-Laufzeit in s	371,5343	2,3903	2,3601	2,3401	2,7388
copyFiles() Ø-Stromverbrauch in Wh	9,6702	0,0612	0,0607	0,0615	0,0723
copy_unbuffered() Ø-Laufzeit in s	3,4190	-	-	-	-
copy_buffered() Ø-Laufzeit in s	-	0,0212	0,0209	0,0207	0,0242
copy_unbuffered() Ø-Stromverbrauch in mWh	88,9879	-	-	-	-
copy_buffered() Ø-Stromverbrauch in mWh	-	0,5417	0,5372	0,5440	0,6394

Tabelle 3.16: Messergebnisse des Szenarios Puffer

Aus den in der Tabelle 3.16 dargestellten Ergebnissen, ist eindeutig zu entnehmen, dass das Puffern im Mittel eine Einsparung von 99 % der Laufzeit und des Stromverbrauchs der *copyFiles()*-Methode bewirkt. Auch zwischen den verschiedenen Puffergrößen sind Unterschiede zu erkennen. Die *copyFiles()*-Methode ist mit einem Puffer von 512 KB am schnellsten, jedoch ist die Energieeffizienz bei einer Puffergröße von 128 KB am größten. Diese Erkenntnis ist auch anhand der Messwerte für die *copy_buffered()*-Methode zu sehen.

Puffer bringen erst einen spürbaren Zeitvorteil, wenn die Menge der zu übertragenden Bytes ausreichend groß ist. Das richtet sich vor allem danach, wieviel Zeit für das Allokieren des Speichers nötig ist, wie schnell das E/A-Gerät die Daten zur Verfügung stellt und wie schnell der Bus zwischen dem E/A-Gerät und der CPU ist. Bei einem vergleichsweise langsamen Bus und einer langsamen Reaktionszeit des E/A-Geräts wird sich das Puffern in jedem Fall lohnen. Weiterhin ist die Größe des Puffers entscheidend dafür, ob Zeiteinsparungen möglich sind. Der Puffer muss mindestens doppelt so groß sein, wie die Anzahl der Bytes, die bei einem Verarbeitungsvorgang gleichzeitig verarbeitet werden können. Ansonsten reduziert sich die Anzahl der Zugriffe auf das Ein- oder Ausgabegerät nicht, was ja Ziel des Pufferns ist.
In der in diesem Szenario und der Messung verwendeten Konfiguration hat das Puffern zu einer Erhöhung der Energieeffizienz der Anwendung beigetragen und ist daher zu empfehlen.

Die Qualität der Anwendung wird insofern negativ beeinträchtigt, dass eine Datenstruktur für das Zwischenspeichern der gelesenen Daten angelegt werden muss. Dadurch wird der Quellcode etwas umfangreicher und damit schwerer zu lesen und zu verstehen, was die Wartbarkeit der Anwendung negativ beeinflusst. Weiterhin können zusätzliche Fehlerquellen entstehen.

3.6.2. Caches

Reduzierung von Zugriffen auf E/A-Geräte kann auch dadurch erreicht werden, dass man häufig zu verwendende Daten, die gelesen oder geschrieben werden sollen, im Hauptspeicher hält, statt bei jedem Zugriff auf diese Daten auch auf das E/A-Gerät zuzugreifen. Dadurch spart man ab dem zweiten Zugriff auf die Daten eine Menge Zeit, da auf den vergleichsweise schnellen Hauptspeicher zugegriffen werden kann. Beim ersten Zugriff werden die Daten in den Hauptspeicher geladen. [16]

Code-Beispiel:

```
public class SimpleFileCache {
	public static interface CachedFile {
		public File getFile();
		public InputStream getInputStream() throws IOException;
		public long length();
	}

	private Map<File,CachedFile> fileMap = new HashMap<File,CachedFile>();

	public CachedFile getCachedFile(File file) throws IOException {

		CachedFile result = null;
		if(fileMap.containsKey(file)) result = fileMap.get(file);
		else {
			result = new ByteArrayFile(file);
			fileMap.put(file, result);
		}
		return result;
	}

	public boolean registerCachedFile(CachedFile f) {
		if (! fileMap.containsKey(f)) {
			fileMap.put(f.getFile(), f);
			return true;
		}
		return false;
	}

	public void reset() {
		fileMap.clear();
	}
}
```

Am Beispiel eines einfachen Datei-Caches soll die Wirkung des Zwischenspeicherns von Dateien im RAM-Speicher gezeigt werden. Das Beispiel hat keinen Anspruch in jeder Hinsicht vollständig zu sein und dient allein dazu, schnell verstanden zu werden. Aus diesem Grund kann hier auch nur lesend auf die Dateien zugegriffen werden.

Die Klasse SimpleFileCache dient dazu ein Verzeichnis von Dateien, die gecacht wurden, zu verwalten. Soll auf eine Datei zugegriffen werden, kann mit der Methode *getCachedFile()* eine gecachte Version der Datei angefordert werden. Sofern die gecachte Datei nicht schon in der internen Map *fileMap* vorliegt, wird ein gecachtes Dateiobjekt erzeugt, das das CachedFile-Interface implementiert, in diesem Fall ein ByteArrayFile und dieses ByteArrayFile wird dann in die interne Map eingetragen. Ein CachedFile-Objekt ist

sozusagen ein Proxy-Objekt zu einem File-Objekt. Auf jeden Fall entfällt ab dem zweiten Zugriff auf eine gecachte Datei, dass ein CachedFile-Objekt erzeugt werden muss, was bedeutet, dass die Datei in den Hauptspeicher geladen werden muss. Ebenfalls kann ein CachedFile mit der Methode *registerCachedFile()* in das Verzeichnis eingetragen werden. Das CachedFile-Interface verlangt, dass die implementierenden Klassen drei Methoden *getFile()*, *length()* und *getInputStream()* konkret implementieren müssen. Mit der Methode *getInputStream()* kann ein InputStream erzeugt werden, mit dem dann die Datei gelesen werden kann. Die Methode *reset()* setzt die interne Map *fileMap* zurück. [14]

Es gibt beispielsweise zwei CachedFile-Implementationen:

```
public class ByteArrayFile implements SimpleFileCache.CachedFile {

	private File file;
	private byte[] content;
	private long length;

	public ByteArrayFile(File file) throws IOException {
		this.file = file;
		this.length = file.length();
		load(file);
	}

	@Override
	public File getFile() {
		return file;
	}

	public long length() {
		return length;
	}

	private void load(File f) throws IOException {
		content = new byte[(int) length];
		InputStream in = new FileInputStream(f);
		try {
			in.read(content);
		} finally {
			if(in != null) try {in.close();} catch (IOException e) {}
		}
	}

	@Override
	public InputStream getInputStream() throws IOException {
		return new ByteArrayInputStream(content);
	}
}
```

Die Klasse ByteArrayFile cacht eine Datei, indem beim Erzeugen einer Instanz dieser Klasse im Konstruktor eine Methode *load()* aufgerufen wird, in der ein byte-Array *content* von der Größe der Datei erzeugt wird und mit den Bytes der Datei gefüllt wird. Der InputStream, der in der Methode *getInputStream()* erzeugt wird, ist ein ByteArrayInputStream, dem das byte-Array *content* als Argument übergeben wird.

```
public class MemoryMappedFile implements SimpleFileCache.CachedFile {

	private File file;
	private MappedByteBuffer mBuf;
	private long length;

	public MemoryMappedFile(File file) throws IOException {
		this.file = file;
		this.length = file.length();
		map(file);
	}

	@Override
	public File getFile() {
		return file;
	}

	public long length() {
		return length;
	}

	private void map(File f) throws IOException {
		FileInputStream in = new FileInputStream(f);
		try {
			mBuf = in.getChannel().map(FileChannel.MapMode.READ_ONLY,
				0, length);
		} finally {
			if(in != null) try {in.close();} catch (IOException e) {}
		}
	}

	@Override
	public InputStream getInputStream() throws IOException {
		return new InputStream() {
			private int pos;

			@Override
			public int read() throws IOException {
				if(pos == mBuf.limit()) return -1;
				return mBuf.get(pos++);
			}

			@Override
			public int read(byte[] b, int off, int len) throws
				IOException {
				if(pos == mBuf.limit()) return -1;
				synchronized (mBuf) {
					mBuf.position(pos);
					int readBytes = Math.min(len,
						mBuf.remaining());
					mBuf.get(b, off, readBytes);
					pos += readBytes;
					return readBytes;
				}
			}
		};
	}
}
```

In der Klasse MemoryMappedFile wird beim Erzeugen einer Instanz im Konstruktor die Methode *map()* aufgerufen. In dieser Methode wird ein FileInputStream *in* erzeugt, von dem ein FileChannel erzeugt wird und mit dessen Methode *map()* wird die Datei direkt in den

Hauptspeicher gemappt und ein MappedByteBuffer *mBuf* für den Zugriff darauf erzeugt. Der MappedByteBuffer wird dann in der Methode *getInputStream()* verwendet, um in einer anonymen Klasse einen InputStream zu erzeugen, mit dem die gemappte Datei gelesen werden kann.

Szenario:

Alle Dateien des Quellverzeichnis aus dem Szenario des Abschnitt 3.6.1. sollen als Schlüssel in einer Map *files* erfasst werden und bis zu 20 Mal gelesen werden. Wie oft die jeweilige Datei gelesen werden soll, wird als Wert des Schlüssels in der Map *files* gespeichert. Nach jedem Ablauf des Szenarios in der Endlosschleife muss das FileCache-Objekt mit der Methode *reset()* wieder auf den Anfangszustand gesetzt werden und der Garbage Collector muss ausgeführt werden, da sonst beim nächsten Schleifendurchlauf die Dateien noch im Heap und damit im Hauptspeicher sind. Auswirkungen hat die Aufräumaktion nur bei den gecachten Dateien. Ein Messdurchgang soll 40 Minuten dauern.

```
public static void szenario_FileCache() {
	String dir = "/home/dradoslav/BA_Tests/files";
	Map<File, Integer> files = new HashMap<File,Integer>();
	SimpleFileCache cache = new SimpleFileCache();

	int counter = 0;

	for(File f : (new File(dir)).listFiles()) {
		files.put(f, counter % 20);
		counter++;
	}

	while(true) {
		try {
			CodeOptimizations.io_readFile(files);

			// separate Messung
			// CodeOptimizations.io_readFile_cached(files, cache);

			// aufräumen
			cache.reset();
			System.gc();
		} catch (IOException e) {}
	}
}
```

Die beiden in der Endlosschleife verwendeten Methoden *CodeOptimizations.io_readFile()* und *CodeOptimizations.io_readFile_cached()* sind wie folgt implementiert:

```
public static void io_readFile(Map<File, Integer> files) throws IOException {
	for(File f : files.keySet()) {
		for(int i = 0, count = files.get(f); i < count; i++) {
			InputStream in = new FileInputStream(f);
			byte content[] = new byte[(int) f.length()];
			in.read(content);
			in.close();
		}
```

```
        }
    }

    public static void io_readFile_cached(Map<File, Integer> files,
            SimpleFileCache s) throws IOException {
        for(File f : files.keySet()) {
            for(int i = 0, count = files.get(f); i < count; i++) {
                SimpleFileCache.CachedFile cf = s.getCachedFile(f);
                InputStream in = cf.getInputStream();
                byte content[] = new byte[(int) cf.length()];
                in.read(content);
                in.close();
            }
        }
    }
```

Es wird über die Schlüssel der in der als Parameter *files* übergebenen Map in einer for-Schleife iteriert und innerhalb dieser Schleife wird mit einer weiteren for-Schleife mit einer bestimmten Häufigkeit, deren Wert als Wert des jeweiligen Schlüssels in der Map *files* gespeichert ist, das Lesen einer Datei mit einem FileInputStream vollzogen. In der Methode *io_readFile_cached()* geschieht das Lesen der Datei *f*, indem von dem als Parameter übergebenen SimpleFileCache-Objekt *s* die Datei *f* als CachedFile-Objekt *cf* und anschließend ein InputStream für *cf* angefordert wird.

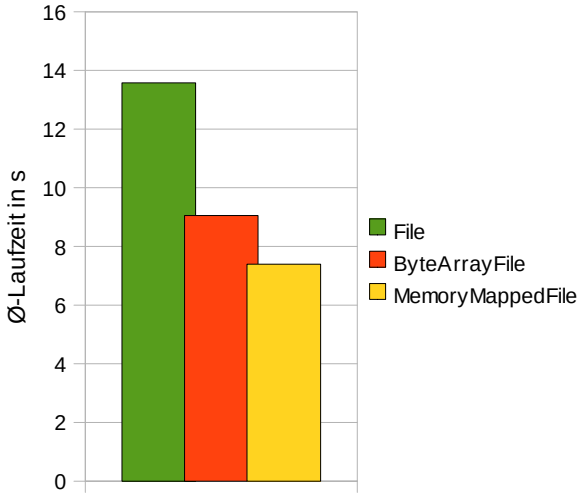

Abb. 3.20: Ø-Laufzeit der readFile-Methoden

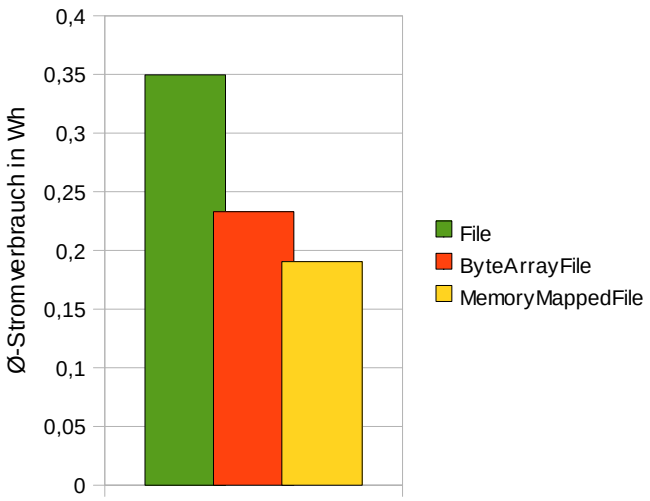

Abb. 3.21: Ø-Stromverbrauch der readFile-Methoden

Die Messergebnisse in den Abb. 3.20 und Abb. 3.21 zeigen, dass das Cachen positive Auswirkung auf die durchschnittliche Laufzeit und den Stromverbrauch hat. Die *io_readFile()*-Methode ist um einen Faktor 1,5 langsamer und energieineffizienter als die *io_readFile_cached()*-Methode mit einem ByteArrayFile und sogar um einen Faktor 1,8 langsamer und energieineffizienter als die *io_readFile_cached()*-Methode mit einem MemoryMappedFile.

Ob signifikante Unterschiede in der Laufzeit und der Energieeffizienz durch das Cachen von Daten entstehen, ist nicht zuletzt abhängig davon, wie oft diese Daten angefordert werden. In der Messung wurden die Dateien ein bis 20 Mal nacheinander gelesen. Die durchschnittliche Laufzeit und der Stromverbrauch sind in diesem Fall durch das Cachen positiv beeinflusst worden und die Anwendung eines Caches ist zu empfehlen. Werden aber insgesamt nur sehr wenig Zugriffe auf die gecachten Dateien gemacht, würden die Abweichungen zwischen den gecachten und ungecachten Dateien geringer werden, da die Daten im Hauptspeicher gehalten werden müssen, was mutmaßlich mehr Energieverbrauch für den Hauptspeicher bedeutet. Weiterhin ist zu bedenken, dass der Hauptspeicher begrenzt ist und größere, mehrere Gigabyte große Dateien möglicherweise nicht komplett in den Hauptspeicher passen, was das Cachen nur für einen Teil der Daten möglich macht. Das Ausführen des Garbage Collectors wird unter Umständen öfter nötig, wenn der Heap-Speicher der VM an seine Kapazitätsgrenzen gelangt, um Speicher von nicht mehr referenzierten Objekten freizugeben, was wiederum die Laufzeit und den Stromverbrauch des Programms negativ beeinflussen kann, je nachdem ob man einen Einzelkern- oder Mehrkernprozessor verwendet und wieviele weitere Prozesse ausgeführt werden. Ein weiterer Punkt, der in diesem Zusammenhang zu berücksichtigen ist, ist das Auslagern von Seiten aus dem Hauptspeicher. Wenn z.B. eine Datei im Hauptspeicher gecacht wurde und die Hauptspeicherverwaltung des Betriebssystems die Datei komplett oder teilweise in eine Auslagerungsdatei auf einer Festplatte auslagert, weil sie eine bestimmte Zeit nicht mehr angefordert wurde und andere Prozesse bspw. mehr Speicher benötigen, hat das Cachen nur eine begrenzt bessere Wirkung bzgl. des Zeitbedarfs, da beim Anfordern der gecachten Datei aus dem Hauptspeicher Seitenfehler auftreten würden, was dazu führt, das die Hauptspeicherverwaltung die fehlenden Seiten aus der Auslagerungsdatei von der Festplatte anfordern muss und gerade den zusätzlichen Zugriff auf die Festplatte will man durch das Cachen verhindern.

Die Qualität der Anwendung wird insofern negativ beeinflusst, dass zusätzliche Klassen notwendig sind, um das Cachen umzusetzen. Dadurch entstehen auf der einen Seite zusätzliche Fehlerquellen, auf der anderen Seite wird die Verständlichkeit des Quellcodes und damit die Wartbarkeit verschlechtert. Für den Benutzer der Klassen, die Caches einsetzen, ändert sich an der Benutzbarkeit nichts.

3.7. Anwendungsbeispiel

Das im folgenden behandelte Anwendungsbeispiel dient u.a. noch einmal dazu, einen Teil der kennengelernten Optimierungen an einem konkreten praktischen Beispiel anzuwenden.

3.7.1. Boyer-Moore Algorithmus

Der Boyer-Moore Algorithmus dient dazu, in einem String *S* ein Muster-String *T* zu suchen und im Falle, dass *T* in *S* vorkommt, die erste Stelle von *T* in *S* anzugeben. Die Idee des Algorithmus besteht darin, den Muster-String *T* von rechts nach links mit dem String *S* zu vergleichen. Bei Nichtübereinstimmung des letzten Zeichens von *T* mit dem Zeichen, dass in S an derselben Stelle ist, wird das Muster um die Länge des Musters weiter hinter das Zeichen verschoben. Bei Übereinstimmung der beiden Zeichen von *T* und *S*, wird von *T* das vorletzte

Zeichen mit dem Vorgänger des in S verglichenen Zeichens durchgeführt. Dies geschieht solange, bis der String komplett verglichen wurde und alle Zeichen gleich sind und damit das Muster erfolgreich gefunden wurde oder bis zwei Zeichen nicht übereinstimmen. Im Falle der Nichtübereinstimmung wird T weiter verschoben. [23]

Eine naive Implementation des Algorithmus [24] sieht folgendermaßen aus:

```
public class BoyerMoore {

	public static final int ALPHABET_SIZE = Character.MAX_VALUE + 1;

	private String text;
	private String pattern;

	private int[] last;
	private int[] match;
	private int[] suffix;

	public BoyerMoore(String text, String pattern) {
		this.text = text;
		this.pattern = pattern;
		last = new int[ALPHABET_SIZE];
		match = new int[pattern.length()];
		suffix = new int[pattern.length()];
	}

	/**
	 * Searches the pattern in the text. Returns the position of the first
	 * occurrence, if found and -1 otherwise.
	 */
	public int match() {
		// Preprocessing
		computeLast();
		computeMatch();

		// Searching
		int i = pattern.length() - 1;
		int j = pattern.length() - 1;
		while (i < text.length()) {
			if (pattern.charAt(j) == text.charAt(i)) {
				if (j == 0) {
					// the left-most match is found
					return i;
				}
				j--;
				i--;
			} else { // a difference
				i += pattern.length() - j - 1
					+ Math.max(j - last[text.charAt(i)],
					match[j]);
				j = pattern.length() - 1;
			}
		}
		return -1;
	}

	/**
	 * Computes the function last and stores its values in the array
	 * last.
	 */
	private void computeLast() {
```

```
        for (int k = 0; k < last.length; k++) {
            last[k] = -1;
        }
        for (int j = pattern.length() - 1; j >= 0; j--) {
            if (last[pattern.charAt(j)] < 0) {
                last[pattern.charAt(j)] = j;
            }
        }
}

/**
 * Computes the function match and stores its values in the array
 * match. The function is defined as follows:
 */
private void computeMatch() {
        /* Phase 1 */
        for (int j = 0; j < match.length; j++) {
            match[j] = match.length;
        }

        computeSuffix();

        /* Phase 2 */
        // Uses an auxiliary array, backwards version of the KMP failure
        // function.
        // if such s exists,
        for (int i = 0; i < match.length - 1; i++) {
            int j = suffix[i + 1] - 1;
            if (suffix[i] > j) {
                match[j] = j - i;
            } else {
                match[j] = Math.min(j - i + match[i], match[j]);
            }
        } // End of Phase 2

        /* Phase 3 */
        // Uses the suffix array to compute each shift s such that and
        // stores the minimum of this shift and the previously computed
        // one.
        if (suffix[0] < pattern.length()) {
            for (int j = suffix[0] - 1; j >= 0; j--) {
                if (suffix[0] < match[j]) {
                    match[j] = suffix[0];
                }
            }
            int j = suffix[0];
            for (int k = suffix[j]; k < pattern.length(); k =
                suffix[k]) {
                while (j < k) {
                    if (match[j] > k)
                        match[j] = k;
                    j++;
                }
            }
        }// endif
}

/**
 * Computes the values of suffix, which is an auxiliary array,
 * backwards version of the KMP failure function.
 */
private void computeSuffix() {
        suffix[suffix.length - 1] = suffix.length;
        int j = suffix.length - 1;
```

```
                for (int i = suffix.length - 2; i >= 0; i--) {
                    while (j < suffix.length - 1
                            & pattern.charAt(j) != pattern.charAt(i)) {
                        j = suffix[j + 1] - 1;
                    }
                    if (pattern.charAt(j) == pattern.charAt(i)) {
                        j--;
                    }
                    suffix[i] = j + 1;
                }
            }
        }
```

Im Weiteren wird eine optimierte Klasse BoyerMoore_opt entwickelt. Zur Optimierung wird jede Methode aus der BoyerMoore Klasse separat betrachtet.
Zuerst einmal werden – zusätzlich zu den schon vorhandenen Instanzvariablen – zwei neue Konstanten *text_length* und *pattern_length* hinzugefügt.

```
    private final int text_length, pattern_length;
```

Die Konstanten werden benötigt, um invariante Operationen, die sich durch die ganze Klasse ziehen zu ersetzen. Im Konstruktor werden die neuen Konstanten initialisiert. Dadurch ist es jetzt möglich bei der Erzeugung der Arrays *last* und *match* auf den doppelten Aufruf der Methode *pattern.length()* zu verzichten.

```
    public BoyerMoore_opt(String text, String pattern) {
        this.text = text;
        this.text_length = text.length();
        this.pattern = pattern;
        this.pattern_length = pattern.length();
        last = new int[ALPHABET_SIZE];
        match = new int[pattern_length];
        suffix = new int[pattern_length];
    }
```

In der Methode match() findet die eigentliche Arbeit des Algorithmus statt. Es wird eine lokale Variable pLength_minus1 vom Typ int angelegt, die den Wert von pattern_length - 1 speichert, da diese Berechnung invariant ist. Es werden alle diese Berechnungen durch pLength_minus1 ersetzt. Des Weiteren wird eine lokale Variable textAtI vom Typ char angelegt, der in while-Schleife immer den aktuellen Wert von text.charAt(i) zugewiesen wird und somit eine mehrfache Berechnung in dem if-else-Ausdruck vermeidet. In der Abbruchbedingung der while-Schleife wird der Aufruf der Methode text.length() durch die Konstante text_length ersetzt.

```
    public int match() {
        computeLast();
        computeMatch();

        final int pLength_minus1 = pattern_length - 1;

        int i = pLength_minus1;
```

```
        int j = pLength_minus1;
        char textAtI;
        while (i < text_length) {
            textAtI = text.charAt(i);
            if (pattern.charAt(j) == textAtI) {
                if (j == 0) return i;
                j--;
                i--;
            } else {
                i += pLength_minus1 - j
                        + Math.max(j - last[textAtI], match[j]);
                j = pLength_minus1;
            }
        }
        return -1;
    }
```

In der Methode *computeLast()* wird zuerst das Array *last* mit dem Wert -1 gefüllt. Dazu wurde in der Klasse BoyerMoore eine for-Schleife verwendet. Hier allerdings kommt eine Utility-Methode der Klasse Arrays namens *fill()* zum Einsatz, da diese Methode nach eigenen Messungen ab einer Array-Größe von ca. 100 Elementen schneller ist, als ein Schleifenkonstrukt. Dann wird auch in der for-Schleife die Konstante *pattern_length*, anstatt des Methodenaufrufs *pattern.length()* verwendet und es wird eine neue lokale Variable *patternAtJ* vom Typ char angelegt, die in der for-Schleife immer den aktuellen Wert von *pattern.charAt(j)* speichert. Damit wird pro Schleifendurchlauf ein Methodenaufruf eingespart.

```
    private void computeLast() {
        Arrays.fill(last, -1);

        char patternAtJ;

        for (int j = pattern_length - 1; j >= 0; j--) {
            patternAtJ = pattern.charAt(j);
            if (last[patternAtJ] < 0) {
                last[patternAtJ] = j;
            }
        }
    }
```

In der nächsten Methode *computeMatch()* wird ähnlich, wie in der Methode *match()* eine lokale Variable *mLength_minus1* vom Typ int angelegt, die das Ergebnis der invarianten Operation *match.length - 1* speichert. Anstatt der invarianten Operation wird der Wert den die Variable *mLength_minus1* beinhaltet verwendet. Besonders in der zweiten for-Schleife ist dieses Vorgehen sinnvoll, da dort die invariante Operation jedes mal beim Prüfen der Abbruchbedingung ausgewertet werden muss. Außerdem wird der Aufruf der Methode *pattern.length()* durch die Konstante *pattern_length* ersetzt.

```
    private void computeMatch() {

        int mLength_minus1 = match.length - 1;

        for (int j = mLength_minus1; j >= 0; j--) {
```

```
                match[j] = match.length;
            }

            computeSuffix();

            for (int i = 0; i < mLength_minus1; i++) {
                int j = suffix[i + 1] - 1;
                if (suffix[i] > j) {
                    match[j] = j - i;
                } else {
                    match[j] = Math.min(j - i + match[i], match[j]);
                }
            }

            if (suffix[0] < pattern_length) {
                for (int j = suffix[0] - 1; j >= 0; j--) {
                    if (suffix[0] < match[j]) {
                        match[j] = suffix[0];
                    }
                }
                int j = suffix[0];
                for (int k = suffix[j]; k < pattern_length; k =
                    suffix[k]) {
                    while (j < k) {
                        if (match[j] > k)
                            match[j] = k;
                        j++;
                    }
                }
            }
        }
```

Abschließend wird in der Methode *computeSuffix()* eine lokale Variable *suffLength_minus1* vom Typ int angelegt, die das Ergebnis von der invarianten Operation *suffix.length - 1* speichert. Die Aufrufe dieser invarianten Operation werden durch *suffLength_minus1* ersetzt. Des Weiteren wird eine weitere lokale Variable *patternAtI* vom Typ char angelegt, die auch wie in den vorherigen Methoden gezeigt, den aktuellen Wert des Methodenaufrufs von *pattern.charAt(i)* speichert. Somit muss in der for-Schleife nur 1 mal der Methodenaufruf von *pattern.charAt(i)* erfolgen. Das Speichern von *pattern.charAt(i)* in einer lokalen Variablen kann in diesem Fall nicht erfolgen, da *j* sowohl in der while-Schleife, als auch in der if-Anweisung neu berechnet wird. In der while-Schleife wird ausserdem in dem logischen Ausdruck der Operator & durch den &&-Operator ausgetauscht. Damit wird im Fall, dass der erste Operand mit *false* ausgewertet wird, der zweite Operand nicht mehr ausgewertet, da der gesamte logische Ausdruck nicht mehr nach *true* ausgewertet werden kann und die Bedingung der while-Schleife damit nicht erfüllt ist.

```
    private void computeSuffix() {
        int suffLength_minus1 = suffix.length - 1;
        suffix[suffLength_minus1] = suffix.length;
        int j = suffLength_minus1;
        char patternAtI;

        for (int i = suffix.length - 2; i >= 0; i--) {
            patternAtI = pattern.charAt(i);
            while (j < suffLength_minus1 && pattern.charAt(j) !=
                                patternAtI) {
                j = suffix[j + 1] - 1;
```

```
                }
                if (pattern.charAt(j) == patternAtI) {
                        j--;
                }
                suffix[i] = j + 1;
        }
    }
```

Szenario:

In einer Liste *productList* von 1000 Artikelbeschreibungen wird nach einem Stichwort *pattern* gesucht und gezählt, wie oft dieses Stichwort unter den Artikelbeschreibungen vorkommt. Zuerst müssen die 1000 Artikelbeschreibungen aus einer Datei *f* mit Hilfe eines BufferedReaders *in* eingelesen und in der Liste *productList* gespeichert werden. Es gibt drei verschiedene Dateien für *f*, in denen Strings einer Länge von 93, 129 und 178 Zeichen gespeichert sind. Damit soll ein Vergleich möglich sein, ob die Stringlänge einen Einfluss auf die Laufzeit und die Energieeffizienz hat. Dann wird in einer Endlosschleife die Methode *CodeOptimizations.search()* aufgerufen. Ein Messdurchgang soll 40 Minuten dauern.

```
public static void szenario_BoyerMoore() {
        File f = new File("D:/BA_Tests/Artikelbeschreibungen.txt");

        //separate Messung
        // File f = new File("/home/dradoslav/
                BA_Tests/Artikelbeschreibungen2.txt");

        //separate Messung
        // File f = new File("/home/dradoslav/
                BA_Tests/Artikelbeschreibungen3.txt");

        String pattern = "16,4 Zoll";

        List<String> productList = new ArrayList<String>();
        String product = "";

        try {
                BufferedReader in = new BufferedReader(new FileReader(f));

                while((product = in.readLine()) != null) {
                        productList.add(product);
                }
        } catch (IOException e) {}

        while(true) {
                CodeOptimizations.search(productList, pattern);

                // separate Messung
                // CodeOptimizations.search2(productList, pattern);
        }
}
```

Die Methode *CodeOptimizations.search()* ist eine Hilfsmethode, um das Szenario besser analysieren und messen zu können. Sie ist wie folgt implementiert:

```
public static int search(List<String> productList, String pattern) {
        int count = 0;
```

```
            for(String p : productList) {
                  BoyerMoore bm = new BoyerMoore(p, pattern);

                  // separate Messung
                  // BoyerMoore_opt bm = new BoyerMoore_opt(p, pattern);

                  if(bm.match() != -1) count++;
            }

            return count;
      }
```

Über die übergebene Liste *productList* wird iteriert. Für jede Artikelbeschreibung *p* wird ein neues BoyerMoore- bzw. BoyerMoore_opt-Objekt erzeugt und anschließend der Zähler *count* inkrementiert, wenn die aufgerufene Methode *match()* eine erfolgreiche Suche mit einem Wert ungleich -1 signalisiert.

Stringlängen	**93**		**129**		**178**	
	BM	**BM_opt**	**BM**	**BM_opt**	**BM**	**BM_opt**
search() Ø-Laufzeit in ms	789,0462	891,3031	801,1417	769,7746	792,0543	765,8099
match() Ø-Laufzeit in µs	224,8449	327,4589	233,0954	201,3509	249,5450	219,4783
computeLast() Ø-Laufzeit in µs	221,1735	325,2432	223,2299	196,5618	236,8757	214,2891
computeMatch() Ø-Laufzeit in µs	0,8323	0,5133	2,2101	1,1598	1,7408	1,1375
computeSuffix() Ø-Laufzeit in µs	0,7351	0,4422	2,0240	1,0261	1,5886	1,0016
search() Ø-Stromverbrauch in Wh	0,0207	0,0230	0,0208	0,0198	0,0207	0,0199
match() Ø-Stromverbrauch in µWh	5,8928	8,4594	6,0508	5,1848	6,5194	5,7064
computeLast() Ø-Stromverbrauch in µWh	5,7966	8,4021	5,7947	5,0615	6,1884	5,5715
computeMatch() Ø-Stromverbrauch in µWh	0,0218	0,0133	0,0574	0,0299	0,0455	0,0296
computeSuffix() Ø-Stromverbrauch in µWh	0,0193	0,0114	0,0525	0,0264	0,0415	0,0260

Tabelle 3.17: Messergebnisse des Szenarios Boyer Moore

Die Messergebnisse in der Tabelle 3.17 zeigen, dass die *search()*-Methode mit einem BoyerMoore-Objekt für eine Stringlänge von 93 um den Faktor 1,1 schneller und energieeffizienter ist, als bei der Verwendung eines BoyerMoore_opt-Objekts. Bei größeren Stringlängen ist dann, entsprechend den Erwartungen, die *search()*-Methode mit einem BoyerMoore_opt-Objekt schneller und energieeffizienter. Bei einer Stringlänge von 129 ist

der Unterschied in der durchschnittlichen Laufzeit und dem Stromverbrauch ca. ein Faktor 1,04 und bei einer Stringlänge von 178 ist ein Faktor von 1,03 zugunsten der BoyerMoore_opt-Klasse zu beziffern. Die *match()*-Methode zeigt ähnliche Ergebnisse, wie die *search()*-Methode. Bei einer Stringlänge von 93 ist sie mit einem Objekt der BoyerMoore-Klasse um einen Faktor 1,4 schneller und energieeffizienter, als bei der Verwendung eines BoyerMoore_opt-Objekts. Bei einer Stringlänge von 129 ist der Unterschied zugunsten der BoyerMoore-Klasse bei einem Faktor 1,15 und bei einer Stringlänge von 178 ist der Faktor 1,13. Die Methode *computeLast()* ist ebenfalls bei einer Stringlänge von 93 mit einem BoyerMoore_opt-Objekt langsamer und energieineffizienter und bei Stringlängen von 129 und 178 kehrt sich dieser Unterschied im Gegensatz zu einem Objekt der BoyerMoore-Klasse um. Bei den beiden Methoden *computeMatch()* und *computeSuffix()* der BoyerMoore_opt-Klasse ist erstmals zu beobachten, dass die durchschnittliche Laufzeit und der Stromverbrauch für alle Stringlängen geringer ist, als bei der BoyerMoore-Klasse. Der Unterschied für die Methoden *computeMatch()* und *computeSuffix()*, ist bei einer Stringlänge von 93 mit jeweils einem Faktor von 1,6, bei einer Stringlänge von 129 mit jeweils einem Faktor von 1,9 und bei einer Stringlänge von 178 mit jeweils einem Faktor 1,5, zugunsten der BoyerMoore_opt-Klasse zu beziffern.

Die Wirkung der Optimierungen in Abhängigkeit von der Stringlänge konnte an diesem Beispiel gut gezeigt werden. Es hat auch gezeigt, dass Erfolge in der Optimierung von Methoden, wie z.B. bei den beiden Methoden *computeMatch()* und *computeSuffix()* durch Misserfolge bei der Optimierung, wie z.B. bei der *computeLast()*-Methode neutralisiert und sogar übertroffen werden können. Die Verwendung der BoyerMoore_opt-Klasse kann also erst für Strings mit mindestens einer Länge von 129 Zeichen empfohlen werden.

Die Qualität der BoyerMoore-Anwendung hat sich mit der Entwicklung der BoyerMoore_opt-Klasse nicht negativ verändert.

4. Messinfrastruktur

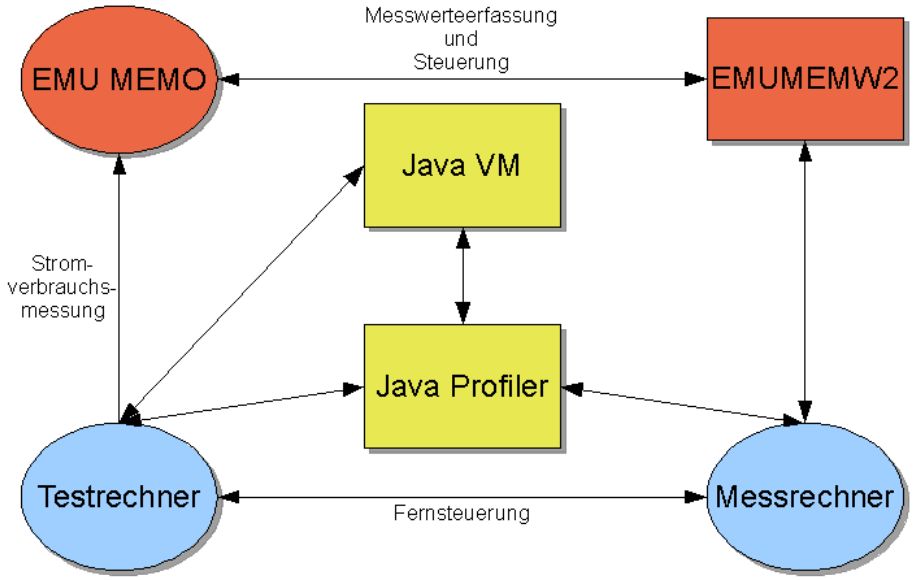

Abb. 4.1: Aufbau der Messinfrastruktur

In der Abb. 4.1 ist der technische Aufbau der Messungen für den Stromverbrauch und der Analyse der Beispielanwendungen dargestellt. Die Komponenten in den ovalen Flächen sind dabei die Hardware- und die Komponenten in den rechteckigen Flächen sind die Softwarekomponenten, die für die Messungen verwendet werden. Zwischen den Komponenten sind die Kommunikationsverbindungen durch Pfeile dargestellt.
Auf dem Testrechner werden die Beispielanwendungen ausgeführt und der Stromverbrauch dieses Rechners wird gemessen.
Der sogenannte Messrechner dient dazu Prozesse, wie z.B. die Java VM auf dem Testrechner zu starten und die Messergebnisse, die von der EMUMEMW2-Software erfasst werden auszuwerten. Da der Messrechner keinen Einfluss auf die Ergebnisse der Messungen hat, wird er in den folgenden Abschnitten nicht näher betrachtet.
Die Verbindung zwischen dem Messrechner und dem Testrechner wird durch ein lokales Netzwerk realisiert. Das EMU MEMO Messgerät ist mit dem Schuko-Stecker des Testrechners und über ein USB-Kabel mit dem Messrechner verbunden.

4.1. Testrechner Hardware

Mainboard: tyan kt400-8235

Prozessor: AMD Athlon XP 2000+

Hauptspeicher: 512 MB

Festplatte: Seagate Barracuda 40GB

Netzteil: Fortron FSP250-61GT, 250W

4.2. Konfiguration des Testrechners

Als Betriebssystem wird eine Linux Distribution namens Ubuntu in der Server-Version 9.04 eingesetzt. Die Entscheidung fiel zugunsten eines Linux-Betriebssystems aus, da es kostenlos verfügbar und für die Zwecke dieser Arbeit vergleichsweise einfach zu konfigurieren ist. Für die Stromverbrauchsmessungen wird keine grafische Benutzeroberfläche benötigt, daher wird hier die Server-Version von Ubuntu eingesetzt, in der standardmäßig keine grafische Benutzeroberfläche installiert ist. Eine grafische Benutzeroberfläche, wie bspw. das X-Window-System in Linux würde zusätzlich Rechnerressourcen, wie CPU-Zeit in Anspruch nehmen und damit die Ergebnisse der Messungen verfälschen. Gesteuert und administriert werden kann der Testrechner über SSH. Dazu muss ein SSH-Dienst installiert werden. Aus diesem Grund kann auf eine Grafikkarte verzichtet werden. Alle Laufwerke, die nicht für die Messungen benötigt werden, wie z.B. DVD-Laufwerk, Disketten-Laufwerk, usw., werden von der internen Stromversorgung getrennt.
Das Auslagern von Seiten aus dem Hauptspeicher in eine swap-Datei wird deaktiviert, da der Hauptspeicher für die durchzuführenden Messungen eine ausreichende Größe hat. Wenn die Auslagerung aktiv wäre, könnte nicht beeinflusst werden, ob und wann das Betriebssystem Seiten auslagert. Das bedeutet, dass zusätzliche Betriebssystemaktivitäten, sowie Hauptspeicher- und Festplattenzugriffe während den Messungen stattfinden und diese die Ergebnisse der Messungen verfälschen würden.

4.3. Java VM

Das Java Runtime Environment (JRE) der Firma Sun Microsystems, Inc. [12] in der Version 6 wird eingesetzt. Im JRE ist sowohl eine Client-, als auch eine Server-Version der Java VM enthalten. Für die Messungen wird nur die Server-Version verwendet, da sie für eine schnelle Ausführbarkeit der Programme entwickelt wurde, während die Client-Version für eine geringe Startzeit und Speicherauslastung entwickelt wurde. [25]

4.4. Java Profiler

Die Anforderungen an das Profiling-Programm sind, dass eine Java-Anwendung zur Laufzeit über die JVMPI-Schnittstelle der Java VM zu analysieren und die Ergebnisse auf einem entfernten Rechner auszuwerten und ggf. grafische und textuelle Repräsentationen der Ergebnisse zu erzeugen ist. Zur Analyse gehören vorrangig die aufgerufenen Methoden und deren Ausführungszeit.

Entsprechend den Anforderungen wird der jprofiler5 [26] zum Analysieren der Java-Programme eingesetzt. Mit diesem Programm ist es nicht nur möglich die Methodenaufrufe und deren Ausführungszeit zu messen, sondern darüber hinaus ist es möglich Informationen über die laufenden Threads, den Heap-Speicher, die lebenden Objekte, den Garbage-Collector, uvm. ausgeben zu lassen.
Mann kann sich den jprofiler als verteilte Anwendung vorstellen, wie in Abb. 4.1 zu sehen, bei der die Kommunikation nach dem Client/Server-Modell [27] über ein lokales Netzwerk erfolgt. Der Testrechner nimmt dabei die Rolle des Servers ein und der Messrechner die des Clients.

Der typische Ablauf einer Laufzeitanalyse ist folgendermaßen: Auf dem Testrechner wird eine Java VM gestartet, die einen jprofiler-Dienst startet, der an einem vorher festgelegten Netzwerk-Port auf eine Verbindung von einem Client wartet. Sobald man auf dem Messrechner eine Messung beginnt, wird eine Verbindung zu dem Server aufgebaut und der jprofiler-Dienst startet die zu analysierende Java-Anwendung. In regelmäßigen konfigurierbaren Abständen fordert der Client vom Server die gesammelten Analysedaten ab, dazu muss der jprofiler-Dienst über die JVMPI-Schnittstelle mit der Java VM kommunizieren, in der die Analysedaten erhoben werden. Auf dem Messrechner kann man dann die Analyseergebnisse darstellen lassen und auswerten.

Das Analysieren einer Anwendung erfordert von der VM einen zusätzlichen Aufwand, als bei der gewöhnlichen Ausführung der Anwendung. Um den Aufwand und die dadurch möglichen Auswirkungen auf die späteren Messergebnisse einschätzen zu können, muss ermittelt werden, wie hoch der zusätzliche Aufwand ist.
Mit dem unter der GNU GPL stehendem Programm OProfile [28] lassen sich unter Linux Programme auf niedriger Systemebene analysieren, bspw. danach wieviel CPU-Zyklen sie in einem bestimmten Zeitraum in Anspruch genommen haben.
Eine Analyse einer Java Anwendung (siehe Anhang) über einen Zeitraum von 22 Minuten hat folgendes Ergebnis:

```
CPU_CLK_UNHALT...|
  samples|      %|
------------------
 20493888 59.2365 java
        CPU_CLK_UNHALT...|
          samples|      %|
        ------------------
         16755872 81.7603 anon (tgid:3100 range:0xb5a42000-0xb5ac2000)
          3398507 16.5830 anon (tgid:11645 range:0xb59ce000-0xb5a5e000)
           208774  1.0187 [vdso] (tgid:11645 range:0xb7f12000-0xb7f13000)
           124905  0.6095 anon (tgid:11645 range:0xb5a5e000-0xb79ce000)
             5598  0.0273 [vdso] (tgid:3100 range:0xb7f86000-0xb7f87000)
              176 8.6e-04 anon (tgid:3100 range:0xb5ac2000-0xb7a42000)
               56 2.7e-04 java
  9832571 28.4205 no-vmlinux
  3052110  8.8220 libjprofilerti.so
   854461  2.4698 libjvm.so
…
```

Die Java Anwendung selbst hat 20.493.888 CPU-Zyklen benötigt. Der Kernel des Betriebssystems (no-vmlinux) benötigte 9.832.571 CPU-Zyklen, der jrofiler (libjprofilerti.so) hat 3.052.110 CPU-Zyklen benötigt und die Java VM (libjvm.so) benötigte 854.461 CPU-Zyklen. Der jprofiler Prozess hat also ca. 8,8 % der gesamten CPU-Zyklen in dem Messzeitraum und ca. 14,9 % der CPU-Zyklen der Java Anwendung in Anspruch genommen.
An der Repräsentativität der Messergebnisse des Stromverbrauchs ändert dies nichts, denn in der Berechnung des durchschnittlichen Stromverbrauchs für einen Methodenaufruf wird die tatsächliche Ausführungszeit der Methode miteinbezogen.

4.5. EMU MEMO 10

Zum Messen des Stromverbrauchs des Testrechners wird in dieser Arbeit das EMU MEMO 10 der EMU Elektronik AG [29] eingesetzt. Die Entscheidung für dieses Gerät wurde aufgrund der Genauigkeit der Messwerte für die Wirkenergie von 0,1 Wh und der Möglichkeit, die Messergebnisse über eine USB-Schnittstelle auf den Messrechner zu übertragen und auszuwerten, getroffen.

Das Gerät wird für eine Messung zwischen der Steckdose und dem Schuko-Stecker gesteckt. Über einen optischen Auslesekopf, der über ein USB-Kabel mit dem Messrechner verbunden ist, kann das Gerät für die Messungen parametriert und die Messergebnisse übertragen werden.

5. Fazit und Rückblick

Das Ziel war es, die beispielhaft erklärten Optimierungsverfahren hinsichtlich ihres Einflusses auf die Laufzeit und damit auf die Energieeffizienz zu untersuchen. Mit Hilfe der Messungen ist es gelungen bei der Mehrheit der Optimierungsverfahren einen positiven Einfluss auf die Energieeffizienz nachzuweisen. Die Ausprägung der Effizienzsteigerung bei den verschiedenen Optimierungsverfahren hängt, wie schon einmal erwähnt, von der Konfiguration und dem Kontext der Anwendung ab.
Im Fall der Synchronisation haben sich die Annahmen über die Auswirkungen auf die Energieeffizienz nicht bestätigt.
Wenn die in dieser Arbeit genannten Java Code Optimierungsverfahren konsequent angewendet werden, ist auf Basis der Messungen davon auszugehen, dass sich der Energieverbrauch von Rechnern – auf denen optimierte Java Programme im Gegensatz zu naiv implementierten Java Programmen ausgeführt werden – mittel- bis langfristig senken lässt. Bei einem privaten Haushalt, in dem ein Rechner steht und auf dem gelegentlich mal eine Java Anwendung ausgeführt wird, werden die Energieeinsparungen allerdings geringer ausfallen, als in einem großen Unternehmen, in dem auf vielen Rechnern und im großen Umfang Java Anwendungen ausgeführt werden.
Die Qualität der Anwendungen hat sich bei den meisten Optimierungen bis auf die schlechtere Lesbarkeit und Verständlichkeit nicht negativ verändert. Der Zugewinn durch die Verbesserung der Performanz gleicht jedoch die weniger gute Lesbarkeit und Verständlichkeit aus.

Abschließend ist festzustellen, dass diese Arbeit längst nicht alle Möglichkeiten, Java Programme auf Quellcode-Ebene hinsichtlich der Energieeffizienz zu optimieren, dargestellt hat. Eine Untersuchung des Stromverbrauchs von einzelnen Komponenten, wie bspw. CPU, Speicher und Festplatte, könnte genauer zeigen, wo und wodurch die Energieeffizienz gesteigert wird. Es sind also noch weitere Potentiale vorhanden, Java Programme zu optimieren, für die weitere Untersuchungen notwendig sind.

6. Ausblick

Das Optimieren von Java Anwendungen auf Quellcode-Ebene ist nur ein erster Schritt, um Java Programme energieeffizienter zu machen. Anknüpfend an dieses Thema gibt es eine Reihe weiterer Ansätze. Intelligenter werdende Compiler erkennen bspw. Ausdrücke im Quellcode, die durch effizientere Ausdrücke ersetzt werden können und sorgen somit dafür, dass ein Anwendungsprogrammierer weniger Zeit für die Optimierung des Quellcodes einsetzen muss. Weiterhin gibt es von der VM eingesetzte JIT-Compiler, die zur Laufzeit bestimmte Codeabschnitte in nativen Code übersetzen, der dann schneller ausgeführt werden kann, weil er auf einer niedrigeren Abstraktionsebene läuft.
Bei Rechnern, die durch die Optimierungen weniger ausgelastet sind, kann fortführend das Energiemanagement des Betriebssystems Energiesparmaßnahmen ergreifen, indem es z.B. die CPU-Taktfrequenz drosselt oder es können mehr Anwendungen auf dem Rechner ausgeführt werden, was unter günstigen Bedingungen dazu führen kann, dass in einem Rechenzentrum insgesamt weniger Rechner benötigt werden.

7. Abkürzungsverzeichnis

CPU	-	Central Processing Unit
E/A	-	Eingabe- und Ausgabe
GB	-	Gigabyte
GNU GPL	-	GNU General Public License
JIT	-	Just-In-Time
JVMPI	-	Java Virtual Machine Profiler Interface
KB	-	Kilobyte
MB	-	Megabyte
PC	-	Program Counter
Schuko	-	Schutzkontakt
SE	-	Softwareengineering
SSH	-	Secure Shell
VM	-	Virtuelle Maschine

8. Glossar

Bytecode: Sammlung von Instruktionen, die durch das Kompilieren des Java Codes erzeugt und von der VM interpretiert wird.

Cache: Zwischenspeicher von Daten, um bei mehrfacher Anforderung der Daten, schneller auf diese zugreifen zu können.

Controller: Wird hier verwendet, um eine Einheit zu bezeichnen, die ein E/A-Gerät steuert und den Datentransfer zwischen dem Rechner und dem E/A-Gerät regelt.

Event: Bezeichnet ein Ereignis, dass zur Steuerung des Programmflusses dient.

Garbage Collection: Bezeichnet ein Verfahren, dass automatisch und in regelmäßigen Abständen den Heap-Speicher von nicht mehr benötigten Objekten und Arrays bereinigt und damit Speicher frei gibt.

Heap: Datenstruktur, die in der Java VM genutzt wird, um Instanzen von Klassen und Arrays zu speichern. Der Heap wird beim Start der VM angelegt und wird während der Laufzeit von einem Speicherüberwachungssystem überwacht (siehe Garbage Collection). [10]

Immutable Object: Bezeichnet ein Objekt, dessen Eigenschaften nach der Erzeugung nicht mehr verändert werden können.

Monitor: Programmiersprachliches Konzept zur Abstraktion der expliziten Synchronisation des Zugriffs von mehreren Threads auf gemeinsame Datenstrukturen und Ressourcen

Out-of-place: Ein Algorithmus überschreibt die Eingangsdaten nicht mit den Ausgangsdaten. Bei einer Methode, die eine Operation auf einem Array durchführt, ist das Eingangs-Array nicht identisch mit dem Ausgangs-Array.

Overhead: Bezeichnet einen Überschuss an Laufzeit, die theoretisch notwendig für die Ausführung einer Operation ist.

Plattform: Bezeichnet eine Kombination aus Hardware und Betriebssystem.

Profiling: Bezeichnet die Analyse des Laufzeitverhaltens eines Programms.

Secure Shell: Bezeichnet ein Netzwerkprotokoll und ein Programm, mit dem man eine verschlüsselte Verbindung zu einem entfernten Rechner herstellen und lokal eine Kommandozeile des entfernten Rechners betreiben kann.

Seiteneffekt: Der Begriff wird hier als unbeabsichtigte Nebenwirkung einer Operation verwendet, die zu inkonsistenten Werten führen kann.

Stacktrace: Zu Diagnosezwecken wird eine Liste der Methodenaufrufe, die zu einem bestimmten Ausnahmezustand führten, aus dem Inhalt des Stacks aufbereitet.

Virtuelle Maschine: Laufzeitumgebung, die die plattformunabhängige Ausführung von Java-Bytecode ermöglicht. In der virtuellen Maschine werden die Bytecode-Instruktionen interpretiert und auf der physischen Maschine – dem eigentlichen Rechner – zur Ausführung gebracht.

Wettlaufsituation: Wenn das Ergebnis einer Operation abhängig von der zeitlichen Konstellation bestimmter Einzeloperationen ist, handelt es sich um eine Wettlaufsituation.

9. Quellen

[1] Solomon, S., D. Qin, M. Manning, Z. Chen, M. Marquis, K.B. Averyt, M.Tignor und H.L. Miller, Eds., *IPCC 2007: Zusammenfassung für politische Entscheidungsträger. In: Klimaänderung 2007: Wissenschaftliche Grundlagen.*
Beitrag der Arbeitsgruppe I zum Vierten Sachstandsbericht des Zwischenstaatlichen Ausschusses für Klimaänderung (IPCC), Bern/Wien/Berlin: Cambridge University Press, Cambridge, United Kingdom und New York, NY, USA; Deutsche Übersetzung durch ProClim-, österreichisches Umweltbundesamt, deutsche IPCC-Koordinationsstelle, .

[2] Greenpeace e.V., “Verursacht der Mensch die Erderwärmung ?,” *http://www.greenpeace.de/themen/klima/klimawandel/artikel/verursacht_der_mensch_die_erderwaermung/*, Apr. 2007.

[3] *IKT-Stromverbrauch in Deutschland: Status und Prognose bis 2020*, Berlin: Fraunhofer-Institut für Zuverlässigkeit und Mikrointegration, IZM Berlin, 2009.

[4] Luiz André Barroso und Urs Hölzle, *The Case for Energy-Proportional Computing*, Google, 2007.

[5] Heiko Mosemann, *Java me: Anwendungsentwicklung für Handys, PDA und Co*, Hanser Verlag, 2008.

[6] Chiyoung Seo, Sam Malek, und Nenad Medvidovic, “An Energy Consumption Framework for Distributed Java-Based Systems.”

[7] Sébastien Lafond und Johan Lilius, “An Opcode Level Energy Consumption Model for a Java Virtual Machine.”

[8] “Ökonomisches Prinzip – Wikipedia,” *http://de.wikipedia.org/wiki/%C3%96konomisches_Prinzip*.

[9] “Elektrische Energie – Wikipedia,” *http://de.wikipedia.org/wiki/Elektrische_Energie*.

[10] Tim Lindholm und Frank Yellin, “The JavaTM Virtual Machine Specification - Second Edition,” 1999.

[11] Bill Venners, “How the Java virtual machine handles method invocation and return,” *http://www.javaworld.com/javaworld/jw-06-1997/jw-06-hood.html*, 1997.

[12] “Sun Microsystems Inc.,” *http://www.sun.com/*.

[13] Sun Microsystems Inc., “The Java Virtual Machine Profiler Interface (JVMPI),” *http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html*.

[14] Hendrik Schreiber, *Performant Java programmieren*, ADDISON-WESLEY, 2002.

[15] James Gosling, Bill Joy, Guy Steele, und Gilad Bracha, “The Java Language Specification, Third Edition.”

[16] Jack Shirazi, *Java Performance Tuning*, O'Reilly, 2003.

[17] Sun Microsystems Inc., “JavaTM Platform Standard Edition 6 API Specification.”

[18] Nell B. Dale und Chip Weems, *Programming and problem solving with Java*, Jones & Bartlett Publishers, 2007.

[19] Joshua Bloch, *Effective Java*, ADDISON-WESLEY, 2008.

[20] Gary Pollice, George Heineman, und Stanley Selkow, *Algorithms in a Nutshell*, O'Reilly, 2008.

[21] Guido Krüger, *Go To Java 2, Handbuch der Java-Programmierung*, ADDISON-WESLEY, 2000.

[22] Jack Shirazi, “The Performance of Java's Lists | O'Reilly Media,” 2001.

[23] Hans-Werner Lang, *Algorithmen: In Java*, Oldenbourg Wissenschaftsverlag, 2006.

[24] V.Boutchkova, *Boyer Moore Pattern Matching Algorithm - Java Implementation*, http://www.fmi.uni-sofia.bg/fmi/logic/vboutchkova/sources/BoyerMoore.java, .
[25] Sun Microsystems Inc., “JavaTM Virtual Machine Technology,” *http://java.sun.com/javase/6/docs/technotes/guides/vm/index.html*.
[26] ej-technologies GmbH, “Java Profiler - JProfiler,” *http://www.ej-technologies.com/products/jprofiler/overview.html*, 2009.
[27] Andrew S. Tanenbaum und Maarten van Steen, *Verteilte Systeme: Prinzipien und Paradigmen*, Pearson Studium, 2008.
[28] John Levon und Philippe Elie, “OProfile - Linux Profiler,” *http://oprofile.sourceforge.net/about/*.
[29] “EMU Elektronik AG,” *http://www.emuag.ch/index_d.htm*.

Anhang

Messergebnisse

Alle Messergebnisse sind auf der beiliegenden CD unter dem Namen Messungen.ods.

Quellcodes

Methode obj_enum_const1():

```
public static void obj_enum_const1(Direction d1, Direction d2) {
	d1.equals(d2);
}
```

Klasse Logger:

```
public class Logger {

	String logFile;
	FileOutputStream out;

	public Logger(String logFile) throws FileNotFoundException {
		this.logFile = logFile;
		out = new FileOutputStream(new File(logFile));
	}

	public void log(String s) throws IOException {
		out.write(s.getBytes());
	}
}
```

Anwendung, mit der der Overhead des jprofilers gemessen wurde:

```
public static void test() {
	while(true) {
		CodeOptimizations.fibonacci_iterativ(2000);
	}
}

public static long fibonacci_iterativ(int n) {
	if (n <= 1) return 1;
	else {
		long i0 = 1, i1 = 1;
		long temp;

		for (int i = 2; i <= n; i++) {
			temp = i1;
			i1 = i0+i1;
			i0 = temp;
		}
		return i1;
	}
}
```

Alle Quellcode-Dateien sind auf der beiliegenden CD unter dem Namen Sourcen.zip.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach § 24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 18. September 2009 _____

Ort, Datum Unterschrift