Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik
Fachgebiet Telematik
Prof. Dr.-Ing. habil. Dietrich Reschke

Hauptseminar Telematik
im Sommersemester 2002

„Java Performance“

Betreuer: Dipl.-Inf. Thorsten Strufe

vorgelegt von:
Torsten Hildebrandt (THildebrandt@gmx.de)
Matrikel-Nr.: 27150

Ilmenau, den 22. Juli 2002

Inhaltsverzeichnis

1 Einleitung 3

2 Allgemeines 3
2.1 Optimierungen 3
2.2 Das 80/20-Prinzip 4
2.3 Vorgehensweise 4
2.4 Prinzipien 5

3 Java-Spezifika 6
3.1 Speicherverwaltung 6
3.2 Strings & Co 7
3.3 Nebenläufigkeit 9
3.4 Containerklassen 11

4 Fazit 12

Literatur 13

1 Einleitung

Die Programmiersprache Java steht im Ruf, in der Ausführung recht langsame Programme zu erzeugen.[1] Die vorliegende Arbeit versucht einige Aspekte darzustellen, die bei der Erstellung performanter Anwendungen zu beachten sind. Dabei soll der sicherlich auch geschwindigkeitsrelevante Bereich der Java Virtual Machine (VM) und der darin Anwendung findenden Techniken[2] hier ausgeklammert bleiben. Vielmehr soll der Schwerpunkt darauf liegen, was ein Anwendungsentwickler beachten kann/sollte. Die vorgestellten Punkte sollten unabhängig von der verwendeten Java-VM Verbesserungen bewirken, wenn auch in unterschiedlicher Höhe.

Sofern nicht anders angegeben, wurden die in der Arbeit erwähnten Laufzeiten auf einem 1 GHz AMD Duron mit 256 MB RAM unter Windows XP Professional mit Sun Java-SDK 1.4 unter Nutzung der Server-VM ermittelt.

Die Arbeit ist derart gegliedert, dass zunächst im zweiten Kapitel auf einige programmiersprachenunabhängige Aspekte und Prinzipien der Optimierung von Programmen eingegangen wird.

Im dritten Kapitel werden ausgewählte Probleme vorgestellt, die Java-spezifisch sind. Sicherlich liesse sich zu praktisch jeder der mittlerweile mehreren hundert Klassen des Java SDKs Anmerkungen machen, es wurde hier jedoch versucht, einige Aspekte auszuwählen, die in sehr vielen Anwendungen auftreten.

Das vierte Kapitel beschliesst die Arbeit mit einigen abschliessenden Bemerkungen.

2 Allgemeines

2.1 Optimierungen

Die Optimierung von Quellcode lässt sich charakterisieren als eine Transformation bestehenden Codes mit folgenden Eigenschaften[3]:

- Korrektheit der Anwendung muss erhalten bleiben. Um dies sicherzustellen, bieten sich automatisierte Tests an.
- Der neue Code ist effizienter als das Original. Unter Effizienz ist hier im Allgemeinen Laufzeiteffizienz zu verstehen.
- Eingeschränkte Wiederverwendbarkeit des neuen Codes, da dieser meist weniger universell ist.
- Resultierender Code ist meist komplexer und damit schwieriger lesbar sowie schwerer zu warten.

Wie bereits an den beiden letzten Punkten ersichtlich, geht die Laufzeit-Optimierung gewöhnlich zu Lasten anderer Software-Qualitätsmerkmale. Es ist deshalb

[1] vgl. etwa [Verv01]
[2] etwa Algorithmen zur Garbage Collection oder zur Compilierung des Java Byte-Codes
[3] vgl. [Bulka00, S. xxiii]

nötig, die richtige Balance zwischen der Ausführungsgeschwindigkeit und der Erreichung anderer Software-Qualitätsmerkmale zu finden.[4]

2.2 Das 80/20-Prinzip

Um diese Balance zu finden, ist die Kenntnis des 80/20-Prinzips nützlich, das sich als Faustregel auch hier anwenden lässt. Es lassen sich gleich mehrere Ausprägungen identifizieren, die sich allerdings teilweise gegenseitig bedingen.[5]

So sind es in aller Regel nur 20 % des Codes, die aber 80 % der Laufzeit ausmachen, bzw. in 20 % der Funktionen wird 80 % der Zeit benötigt.

80 % der nötigen Anwendungsszenarios werden nur 20 % des Codes berühren. Das heisst, dass oft grosse Mengen an Code zur Behandlung diverser Sonderfälle benötigt werden, die aber letztendlich nur sehr selten zur Ausführung kommen. Dies ist nicht zuletzt der Tatsache geschuldet, dass der Grossteil der Eingabe-Daten aus einem nur kleinen Bereich der möglichen Daten kommen.

Das 80/20-Prinzip findet sich allerdings auch bei der Optimierung direkt. Hier kann man davon ausgehen, dass 20 % der möglichen Optimierungen bereits 80 % der erreichbaren Laufzeitverbesserung bewirken. Anders formuliert, gibt es einige wenige Optimierungen, die eine sehr grosse Verbesserung der Laufzeit bewirken.

Als Konsequenz ergibt sich, dass zufällige Optimierungen zu 80 % sinnlos sind, da sie ausserhalb des kritischen Pfades, das heisst der 20 % des Codes erfolgen, die für 80 % des Laufzeit-Bedarfs verantwortlich sind. Ziel muss es sein, einen möglichst guten „Return“ für die Investitionen zu erhalten. Diese Investitionen bestehen zum einen aus der Verschlechterung der anderen Qualitätsmerkmale und zum anderen aus der Zeit, die zur Durchführung der Optimierungen notwendig ist. Es ist also anzustreben, die 20 % der Optimierungen herauszufinden und durchzuführen, die den Grossteil der möglichen Verbesserungen bringen.

Dazu ist es nötig, den kritischen Pfad der Anwendung zu kennen. Dies ist mittels eines Profilers möglich. Bereits das Java SDK von Sun verfügt über entsprechende (einfache) Funktionen (aktivierbar mittels der Kommandozeilen-Optionen `-Xrunhprof`, `-prof`, `-Xprof`, `-Xaprof`, `-Xhprof`[6]). Des Weiteren sollten für den späteren Einsatz möglichst repräsentative Testdaten zur Verfügung stehen und eine Testumgebung geschaffen werden, mit der gezielt einzelne Aspekte getestet und überprüft werden können.

2.3 Vorgehensweise

Eine mögliche Vorgehensweise ist etwa nach folgendem iterativen Schema:[7]

1. Identifikation des Flaschenhalses

2. Setzen eines Geschwindigkeitszieles

---

[4] DIN 66272 ([DIN94]) unterscheidet etwa die sechs Hauptqualitätsmerkmale Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit.

[5] für die folgenden 3 Absätze vgl. [Bulka00, S. xxiv, 45f]

[6] für eine Beschreibung siehe etwa [Pier01]

[7] in Anlehnung an [Shir02a]

3. Nutzen repräsentativer Daten zur
4. Messung des Ist-Zustands
5. Analyse durch
    - Profiling
    - Analyse des Algorithmus
    - Analyse von Abweichungen in Laufzeiten
6. Durchführen und testen von Änderungen, wenn keine Verbesserung wieder zu 5.
7. Bereits schnell genug? Wenn nein, zurück zu 5. oder 1.

2.4 Prinzipien

Bei der Optimierung von Programmen lassen sich einige grundlegende (programmiersprachenunabhängige) Prinzipien identifizieren, die bei der Erstellung laufzeiteffizienter Programme zu beachten sind.

- *Wahl geeigneter Datenstrukturen.* Bei der Auswahl von Datenstrukturen sind deren Spezifika zu beachten. Wird z. B. eine Liste benötigt, auf deren Elemente mittels eines Indexes zugegriffen werden muss, so ist etwa eine `LinkedList` dafür ungeeignet und es sollte etwa eine `ArrayList` verwendet werden. Bei dieser Klasse ist etwa darauf zu achten, dass Einfügen und Entfernen von Elementen nur am Ende der Liste schnell möglich ist.

  Derartige Besonderheiten sollten möglichst beachtet werden, auch wenn sich die Klassen dank Objektorientierung als Black-Box betrachten lassen und aus Bequemlichkeit für den Nutzer Operationen anbieten, obwohl diese nur ineffizient ausgeführt werden können.

- *Wahl geeigneter Algorithmen.* Bei der Auswahl von Algorithmen sollte möglichst der Algorithmus mit der geringsten Laufzeitkomplexität bevorzugt werden. Doch auch hierbei gibt es Ausnahmen, wie bereits durch „geeignet“ angedeutet. So kann etwa ein BubbleSort durchaus schneller sein als ein Quicksort oder eine sequentielle Suche Hashing überlegen sein, wenn die Zahl der zu sortierenden bzw. zu durchsuchenden Elemente klein ist. Hier bietet es sich einmal mehr an, repräsentative Test-Daten zur Verfügung zu haben.

- *Vermeiden unnötiger und doppelter Berechnungen.*

- *Verschieben von Berechnungen.* Berechnungen auf dem kritischen Pfad, die dort nicht unbedingt nötig sind, weil sie erst später benötigt werden, oder konstant sind, sollten bereits vorher bzw. in einem Nachverarbeitungsschritt errechnet werden. Eine Variante davon ist auch das Herausziehen von schleifeninvarianten Ausdrücken vor die Schleife.

- *Ausnutzen von Anwendungsspezifika.* Wie bereits im Abschnitt „Optimierungen" erwähnt, beruhen Optimierungen oft auf der Ausnutzung von Spezifika des Anwendungsbereichs, die effizienteren Code ermöglichen, da dieser nicht so allgemeingültig seien muss. Ein Beispiel hierzu ist etwa die Verwendung einer einfachen Zuweisung bei der Umwandlung eines Bytes in ein (Unicode-)Zeichen statt der Verwendung eines `CharsetDecoder`, wenn feststeht, dass nur ASCII-Zeichen als Eingabe vorkommen können.

- *Zwischenspeichern von Berechnungen.* Eine effizientere Implementierung lässt sich ebenfalls durch ein Zwischenspeichern von (aufwendig zu ermittelnden) Ergebnissen erreichen, die sich gar nicht oder nur selten ändern. Neben dem eigentlichen Zwischenspeichern (Caching) fällt auch Resource-Pooling in diese Kategorie, d.h. das Zwischenspeichern und Wiederverwenden von Objekten, die nur sehr aufwendig zu erzeugen sind (etwa Threads oder Datenbankverbindungen).

Obwohl einige der Punkte recht trivial klingen mögen, ist es in der Praxis oft nur sehr schwierig möglich, sie zu entdecken und ihnen gerecht zu werden.

3 Java-Spezifika

3.1 Speicherverwaltung

Während Speicherlöcher im herkömmlichen Sinne in Java ausgeschlossen sind, gibt es auch hier eine sehr ähnliche Möglichkeit. Normalerweise spricht man von einem Speicherloch, wenn eine Anwendung vergisst, dynamisch reservierten Speicher wieder freizugeben, obwohl dieser nicht mehr benötigt wird. Dies führt speziell bei lange laufenden Anwendungen (etwa Server) zu einem sehr hohen Speicherbedarf.

In Java ist diese Art von Fehler durch den automatischen Garbage Collector (GC) ausgeschlossen, es kann sich jedoch ein ähnlicher Effekt aus zwei Gründen ergeben. Zum Einen dadurch, dass bisherige GC-Implementierungen nicht völlig genau arbeiteten und es vorkommen kann, dass nicht entscheidbar ist, ob es sich bei einem Wert um eine Zahl oder eine Referenz handelt.[8] Ein anderer Grund sind vom Entwickler vergessene Referenzen, z. B. Referenzen in Container-Objekten, wenn Objekte gecacht werden.

Ersteres Problem wird etwa durch den GC der HotSpot-VM von Sun gelöst, der genau arbeitet.[9]

Zur Lösung des zweiten Problems existieren seit Java 1.2 die Klassen im Paket `java.lang.ref`, die eine begrenzte Interaktion mit dem GC erlauben und dem Entwickler eine feinere Steuerung und Kontrolle des Speicherverbrauchs einer Anwendung ermöglichen.

[8] vgl. [Sun01, S. 8]
[9] vgl. [Sun01, S. 7–10]

Referenz-Objekte[10]

Diese Steuerung wird ermöglicht durch eine abgestufte Erreichbarkeit der Objekten und der Möglichkeit, über Änderungen in der Erreichbarkeit benachrichtigt zu werden.[11]

Es gibt 5 Erreichbarkeitsstufen, die ein Objekt haben kann (aufgeführt in abnehmender Stärke):

stark sind alle Objekte, auf die eine Java-Anwendung ohne Nutzung von Referenzobjekten zugreifen kann. Stark erreichbar sind also alle Objekte, auf die eine Anwendung direkt Referenzen hält, bzw. die indirekt über diese Referenzen erreichbar sind.

soft ein Objekt ist „softly reachable“, wenn es nicht mehr stark , aber noch über mindestens ein `SoftReference`-Objekt von der Anwendung erreichbar ist.

weak schwach erreichbar ist ein Objekt immer dann, wenn es weder stark erreichbar noch „softly reachable“ ist, aber über mindestens ein `WeakReference`-Objekt.

phantom „phantom reachable“ sind alle Objekte, die nicht in die drei vorhergehenden Klassen fallen, für die aber noch mindestens ein `PhantomReference`-Objekt existiert.

unerreichbar sind alle Objekte, die von einer Anwendung weder stark erreichbar, noch über ein Referenz-Objekt zu erreichen ist.

Prinzipiell dürfen alle ausser stark erreichbaren Objekten vom GC freigegeben werden. Phantom references sind hierbei die schwächste Form, die referenzierten Objekte sind nicht mehr in starke Referenzen zu verwandeln, sobald sie einmal „phantom reachable“ wurden. Schwach erreichbare Objekte sind bis zu einem GC-Lauf noch über ihr Referenz-Objekt zu ermitteln, können dabei aber freigegeben werden. „Softly reachable“-Objekte werden vom GC freigegeben, wenn nur noch wenig Speicher verfügbar ist (bevor ein `OutOfMemoryError` erzeugt werden darf, müssen alle „softly reachable“ Objekte freigegeben sein). Des Weiteren sollen zuerst die ältesten, bzw. am längsten nicht mehr benutzten Objekte freigegeben werden.

Zusammen mit den Benachrichtigungen mittels `ReferenceQueue`s hat man damit eine Möglichkeit, Speicherverbrauch und Objektfreigabe in einer Anwendung sehr viel besser zu steuern als dies sonst etwa mittels `finalize()` möglich war.

3.2 `Strings` & Co

Eine Möglichkeit, Laufzeit zu verschenken, besteht in der Besonderheit von Java z. B. gegenüber C++, dass `String`s unveränderlich sind.[12] Das folgende Stück Code etwa benötigt 5,7 s:

---

[10] für den folgenden Abschnitt vgl. [ApiDoc, Paket `java.lang.ref`]

[11] mittels einer `ReferenceQueue`

[12] für den Abschnitt vgl. [ApiDoc, Beschreibung der Klasse `String`] sowie [Bulka00, S. 1–6]

```
1 long start = System.currentTimeMillis();
2 String s = "";
3 for (int i = 0; i < 20000; i++) {
4     s = s + "a";
5 }
6 long stop = System.currentTimeMillis();
```

Diese Design-Entscheidung macht es einerseits möglich, dass sich mehrere `String`-Objekt ein Character-Array teilen können. So ist etwa bei einem `substring()`-Aufruf kein neues Anlegen eines Char-Arrays und Kopieren des Inhaltes nötig. Nachteil dieser Lösung ist andererseits, dass Veränderungen nur nach Anlegen eines `StringBuffer`-Objektes durchgeführt werden können. Es kommt hinzu, dass dieser Schritt bei Verwendung des `+`- oder `+=`-Operators automatisch durch den Compiler erfolgt, so dass Zeile 4 zu

```
s = (new StringBuffer(s)).append("a").toString();
```

wird. Dabei werden bei jedem Schleifendurchlauf mindestens 3 neue Objekte erzeugt, nämlich zum einen offensichtlich das `StringBuffer`- und `String`-Objekt des weiteren aber bei der Erzeugung des `StringBuffer`s auch noch ein neues Character-Array.

Eine bessere Variante ist:

```
1 StringBuffer sb = new StringBuffer();
2 for (int i = 0; i < 20000; i++) {
3     sb.append("a");
4 }
5 String s = sb.toString();
```

Diese benötigt weniger als 50 ms. Weitere Verbesserungen[13] sind möglich:

```
1 StringBuffer sb = new StringBuffer(20000);
2 for (int i = 0; i < 20000; i++) {
3     sb.append('a');
4 }
5 String s = sb.toString();
```

Diese beruhen auf der vorherigen Kenntnis der Zielgrösse des Strings, was die sonst gelegentlich nötigen Expandierungen des internen Character-Arrays verhindert und der Verwendung der `append(char)`-Methode statt `append(String)`[14].

Abschliessend im Kapitel über `String`s soll noch die Frage behandelt werden, ob sich die Auflösung eines Konstrukts wie:

```
String s = "Test " + i + "!";
```

[13]bei 2 Mio Schleifendurchläufen von 330 ms auf 125 ms

[14]vgl. [Wiede01, S. 11–12]

in

```
StringBuffer sb = new StringBuffer();
sb.append("Test ");
sb.append(i);
sb.append("!");
String s = sb.toString();
```

lohnt. Hierzu ist zu sagen, dass sich dies in der dargestellten Form nicht lohnt, da diese Umwandlung ziemlich genau dem entspricht, was der Compiler ohnehin erzeugt, Variante 1 aber deutlich besser lesbar ist. Sinn macht sie nur, wenn abzuschätzen ist, wie lang der `String` letztendlich wird und man diese Länge im `StringBuffer`-Constructor angibt, so dass eventuelle Erweiterungen des Character-Arrays vermieden werden.

3.3 Nebenläufigkeit

Hier soll zunächst der Frage nachgegangen werden, wieviel Laufzeit die Verwendung von synchronisierten Methoden kostet. Dazu wurde der der Zeitbedarf von 20 Mio Aufrufen der folgenden Methoden mit verschiedenen Versionen des Java-SDKs von Sun gemessen.

```
1  public void test1(StringBuffer sb)
2  {
3      sb.append('a');
4  }
5
6  public synchronized void test2(StringBuffer sb)
7  {
8      sb.append('a');
9  }
```

Abbildung 1 zeigt die ermittelten Ergebnisse. Dabei wurden auf der rechten Ordinate die benötigten Zeiten und auf der linken Ordinate die zusätzlich benötigte Laufzeit mit Synchronisierung in Prozent aufgetragen. Für das SDK 1.1 bedeutet dies, dass ohne Synchronisierung eine Zeit von ca. 6 s, mit Synchronisierung aber knapp 12 s benötigt wurden. Dies ist eine um ca. 90 % höhere Laufzeit. Die mit den Zusätzen „s“ bzw. „c“ versehenen Einträge stehen dabei für die mit den Server- bzw. Client-VMs ermittelten Ergebnisse.

Diese Art von „Micro-Benchmarks“ ist allerdings seit Einführung der HotSpot-VM (Standard ab Sun Java-SDK 1.3) nur begrenzt aussagekräftig, da der Compiler (zur Laufzeit) besonders häufig aufgerufene Methoden und oft ausgeführte Schleifen optimiert und unter Anderem möglichst die Methodenaufrufe eleminiert (inlining).[15]

[15] vgl. [Sun02]

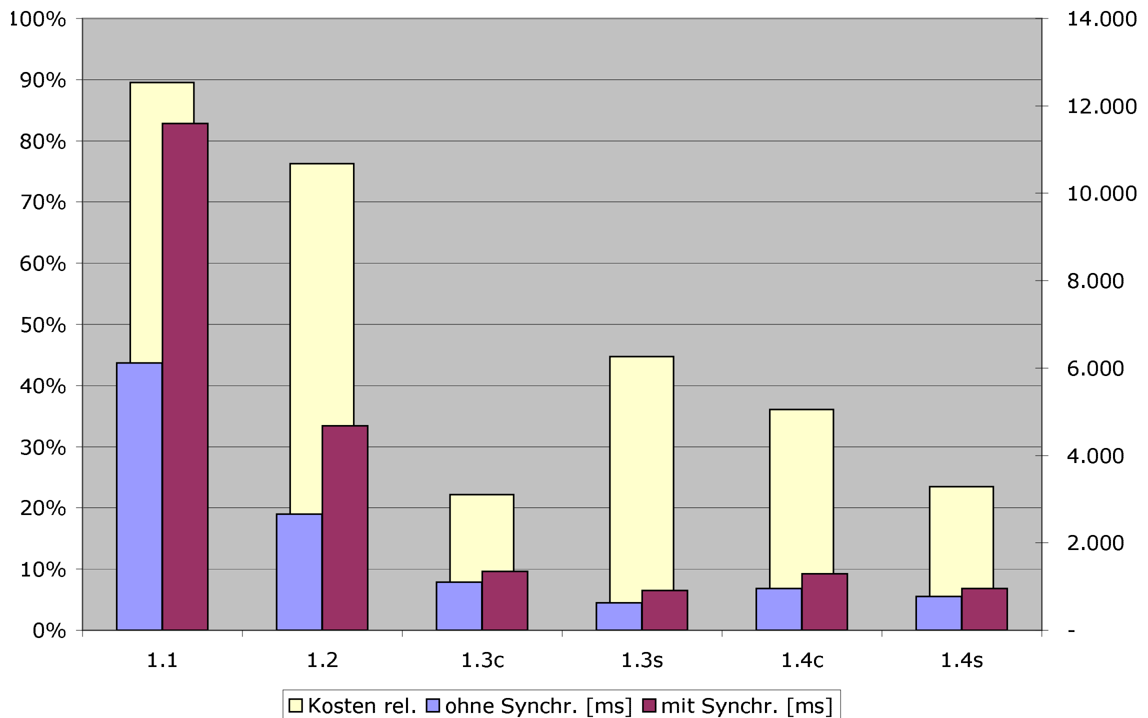

Abbildung 1: Kosten der Synchronisierung verschiedener VMs

Beim Thema Nebenläufigkeit/Multithreading ist generell zu beachten, dass der mögliche Geschwindigkeitszuwachs durch die sequentiell auszuführenden Teile begrenzt wird (Amdahl's Law).[16] Sequentielle Teile sind dabei zum einen alle vor und nach den parallelen Teilen auszuführenden, zum anderen aber auch die Teile, auf die der Zugriff durch mehrere Threads synchronisiert werden muss. Letzteres wirkt sich dabei umso stärker aus, je mehr Threads auf eine gemeinsame Ressource zugreifen müssen. Trotzdem ist eine Synchronisierung meist unvermeidlich. Dabei gibt es ein Granulierungsproblem, d. h. einerseits verursachen grosse, lang dauernde synchronisierte Bereiche ein langes Blockieren anderer Threads. Andererseits bedeuten viele, kleine synchronisierte Blöcke einen höheren Synchronisationsaufwand.[17]

Verbesserungen in nebenläufigen Passagen lassen sich etwa durch eine Vervielfältigung der bisher gemeinsam genutzten Ressourcen erreichen, wenn also z. B. statt eines gemeinsamen Datencaches für jeden Thread ein eigener erzeugt wird. Möglichkeiten mit dem dabei auftretenen höheren Speicherbedarf umzugehen, wurden unter Abschnitt 3.1 behandelt. Eine weitere Möglichkeit der Verbesserung ist der Einsatz weitergehender Synchronisationsmechanismen, etwa Read-/Write-Locks, wenn auf eine Ressource oft lesend, aber nur relativ selten schreibend zugegriffen werden muss.[18]

---

[16] vgl. [Bulka00, S. 132–133]
[17] vgl. [Wiede01, S. 35]
[18] eine Implementierung findet sich etwa in [Lea99]

Des Weiteren ist zum Thema Threads anzumerken, dass ihre Erzeugung teuer ist, so dass hier Pooling[19] und Wiederverwendung bereits erzeugter Objekte eingeführt werden sollte.[20]

3.4 Containerklassen

Exemplarisch für Geschwindigkeitsaspekte bei der Nutzung von Klassen des Java-Container-Frameworks soll hier der Frage nachgegangen werden, welche Möglichkeit des Iterierens über die Einträge in einer `ArrayList` zu bevorzugen ist.[21] Dazu wurde zum einen die Variante mittels Indizes (Zeilen 1 bis 5), zum anderen die Variante mittels `Iterator` (Zeilen 7 bis 11) getestet.

```
1  for (int i=0, n=l.size(); i<n; i++)
2  {
3      Integer o = (Integer) l.get(i);
4      sum += o.intValue();
5  }
6
7  for (Iterator it = l.iterator(); it.hasNext();)
8  {
9      Integer o = (Integer) it.next();
10     sum += o.intValue();
11 }
```

Man beachte in Zeile 1 das Herausziehen eines schleifeninvarianten Ausdrucks. Da feststeht, dass sich die Anzahl der Elemente der Liste nicht ändert, muss der Methodenaufruf `size()` nur einmal vor Ausführung der Schleife durchgeführt werden und nicht jedesmal zur Prüfung der Abbruchbedingung.

Der Unterschied zwischen beiden Varianten war bei 500000 Durchläufen für eine Liste mit 1000 Elementen mit 3,4 s gegenüber 12,7 s sehr gross. Dies liegt auch an der Tatsache, da in der Schleife nur die Summe der Elemente gebildet wird. In den meisten praktischen Anwendungen dürfte der Unterschied nicht derart stark ausfallen, wenn der Zeitbedarf der Anweisungen in der Schleife grösser ist. Dieses Ergebnis legt trotzdem nahe, in Anwendungen (gerade bei Verwendung fremder APIs, die z. B. nur `Collection`-Referenzen übergeben) auf eine Implementierung des Interfaces `RandomAccess` zu prüfen und gegebenenfalls diese Tatsache auszunutzen.[22]

Von der Ausnutzung der Tatsache, dass `next()` eine `NoSuchElementException` erzeugt, ist im Übrigen abzuraten, es ergab sich eine weitere Verschlechterung der

---

[19] Es bietet sich ausserdem eine Verwaltung nach dem LIFO-Prinzip (Last In First Out) an, da auf diese Weise davon profitiert werden kann, dass die zuletzt verwendeten Thread-Objekte und damit zusammenhängende Daten sich noch in Caches des Prozessors oder Betriebssystems befinden.

[20] vgl. [Bulka00, S. 81–85]

[21] generell ist `ArrayList` gegenüber `Vector` zu bevorzugen, es sei denn, die Synchronisierung der Methoden von `Vector` ist nötig.

[22] vgl. [Shir01]

Laufzeit. Dies spart zwar den Aufruf von `hasNext()` in der Abbruchbedingung, die Erzeugung der Exception kostet aber (wahrscheinlich durch das Ausfüllen des Stacktraces) ungleich mehr Zeit. Diese Variante scheint daher höchstens sinnvoll, wenn die Liste sehr viele Einträge enthält.

Zusammenfassend ist zu sagen, dass die Java-Collection-Klassen auf möglichst universelle und einfache Benutzbarkeit hin optimiert scheinen. In geschwindigkeitskritischem Code können sich durch die Erzeugung eigener speziellerer Klassen Optimierungsmöglichkeiten ergeben:

- Elemente sind immer vom Typ `Object`. Dadurch ist es meist nötig, diese Objekte zum gewünschten Typ umzuwandeln. Diese Typumwandlung erfolgt zur Laufzeit immer mit Überprüfung der Zulässigkeit der Umwandlung. Wenn feststeht, dass Elemente nur von einem bestimmten Typ seien könnnen, ist diese Prüfung nicht unbedingt nötig.
  Die Festlegung auf Objekte als Elemente erweist sich auch als hinderlich, wenn native Datentypen gespeichert werden sollten. Dazu muss erst ein entsprechendes Wrapper-Objekt konstruiert werden.

- Die etwa in der Klasse `ArrayList` durchgeführten Bereichsprüfungen[23] scheinen in doppelter Hinsicht redundant. Zum Einen dürfte meist durch den aufrufenden Code bereits sichergestellt sein, dass Indizes innerhalb gültiger Bereiche liegen, zum Anderen erfolgt bei jedem Array-Zugriff eine weitere Bereichsprüfung[24] durch die VM.[25]

4 Fazit

Diese Arbeit stellte einige Aspekte vor, die bei der Erstellung performanter Java-Anwendungen zu berücksichtigen sind.

Dass damit praktisch sehr starke Verbesserungen möglich sind, auch ohne dabei auf das Java Native Interface oder Java-Compiler, die direkt Maschinencode erzeugen, zurückgreifen zu müssen, zeigt etwa [Bulka00, Kapitel 11], wo der unter [Brown97] verfügbare einfache Web-Server auf die etwa 4-fache Anzahl an behandelten Anfragen pro Sekunde optimiert wird.

Dennoch ist eine auch nur annähernd umfassende Behandlung des Themas auf derart wenigen Seiten nicht möglich. Für weitergehende Beschäftigung empfielt sich etwa [Bulka00] sowie als Web-Site die zahlreiche weitere Dokumente zum Thema auflistet und zusammenfasst [Shir02].

[23] etwa bei jedem Aufruf der `get()`-Methode

[24] Die VM sorgt ebenfalls dafür, dass Variablen und insbesondere Arrays mit 0 bzw. `null` initialisiert werden, so dass entsprechde nochmalige Initialisierungen (etwa bei Puffer-Variablen) nicht nötig sind.

[25] vgl. [Gosl00, S. 211], bzw. Kapitel 10.4

Literatur

[ApiDoc] Sun Microsystems: Java™ 2 Platform, Standard Edition, v 1.4.0, API Specification.
http://java.sun.com/j2se/1.4/docs/api/, Abruf am 2002-06-23.

[Bulka00] Dov Bulka: Java Performance and Scalability.
Boston u.a., 2000.

[Brown97] David Brown: A Simple, Multithreaded Web Server.
http://developer.java.sun.com/developer/technicalArticles/Networking/Webserver/index.html, Abruf am 2002-06-23.

[DIN94] DIN (Hrsg.): DIN 66272: Informationstechnik - Bewerten von Softwareprodukten - Qualitätsmerkmale und Leitfaden zu ihrer Verwendung. Identisch mit ISO/IEC 9126: 1991.
Berlin, 1994.

[Gosl00] James Gosling u.a.: The Java Language Specification.
2. Auflage, Boston u.a., 1999.
auch abrufbar unter
http://java.sun.com/docs/books/jls/second_edition/html/,
Abruf am 2002-06-23.

[Lea99] Doug Lea: Concurrent Programming in Java: Design Principles and Patterns.
2. Auflage, Boston u.a., 1999.

[Pier01] Bill Pierce: Diagnose common runtime problems with hprof.
http://www.javaworld.com/javaworld/jw-12-2001/jw-1207-hprof.html, Abruf am 2002-06-22.

[Shir01] Jack Shirazi: Faster List Iteration with RandomAccess Interface.
http://www.onjava.com/pub/a/onjava/2001/10/23/optimization.html,
Abruf am 2002-06-21.

[Shir02] Jack Shirazi: Java Performance Tuning.
http://www.javaperformancetuning.com, Abruf am 2002-06-21.

[Shir02a] Jack Shirazi: Micro-Tuning Step-by-Step.
http://www.onjava.com/pub/a/onjava/2002/03/20/optimization.html,
Abruf am 2002-06-22.

[Sun01] Sun Microsystems: The Java HotSpot™ Virtual Machine - Technical White Paper.
http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.pdf, Abruf am 2002-06-21.

[Sun02] Sun Microsystems: Benchmarking the Java HotSpot VM
http://java.sun.com/docs/hotspot/PerformanceFAQ.html#benchmarking, Abruf am 2002-06-22.

[Verv01] Erwin Vervaet: Java: It's a good thing. Why Java isn't slow, ugly, or irrelevant.
http://www.javaworld.com/javaworld/jw-02-2001/jw-0202-javasalon.html, Abruf am 2002-06-21.

[Wiede01] Michael Wiedekind: Gute JAVA-Programme schneller machen.
http://www.mathema.de/de/archive/download/jax2001/s4.pdf, Abruf am 2002-06-20.