Performance von Java-Anwendungen

Dipl.-Inform. Thilo Hermann
thilo.hermann@web.de
Dipl.-Inform. Dieter Ebhart
dieter_ebhart@freenet.de

Abstract
Die Performance einer Anwendung ist ein kritischer Erfolgsfaktor für die Akzeptanz beim Anwender. Insbesondere bei objektorientierten Hochsprachen wie Java spielen Performanceüberlegungen bei der Konstruktion großer individueller Softwaresysteme eine Rolle. Hier versprechen die Hersteller von Java Virtual Machines (VM) einen erheblichen Performancegewinn durch den Wechsel auf Ihre jeweils neueste Version. Welchen Anteil an der Performance einer Java-Anwendung hat eigentlich die Auswahl der VM und wie stark lässt sich die Performance durch optimierte Algorithmen beeinflussen? Dieser Artikel beleuchtet beide Aspekte anhand einfacher Codebeispiele.

Die Codebeispiele
Oft muss in Java-Anwendungen neben Objekten auch mit elementaren Datentypen gearbeitet werden. Die Verwendung dieser Datentypen hat unserer Erfahrung nach großen Einfluss auf die Performance einer typischen Java-Anwendung. Deshalb haben wir beispielhaft dieses Themenfeld ausgewählt und das Laufzeitverhalten verschiedener Beispielprogramme auf den VMs 1.3.1 und 1.4.2 von SUN, sowie der VM 1.3 von IBM gemessen. Als Betriebssystem diente Windows XP. Die gemessenen Laufzeiten sind mit 100% für den Durchlauf der einfachsten Variante auf der SUN VM 1.4.2 normiert, so dass die Graphiken Aufschluss über die prozentuale Veränderung der Laufzeit geben.

Doch nun zu den Beispielprogrammen. Um einfache Datentypen auch an Stellen verwenden zu können, an denen nur Objekte zulässig sind existieren sogenannte Wrapper-Klassen. Ein Standardbeispiel hierfür sind die Collections-Klassen von Java. Diese Container-Klassen ermöglichen die generische Verwaltung von Daten und bieten zusätzlich Standardalgorithmen auf den Daten an. Intern verwalten die Container nur Objekte. Um also elementare Datentypen darin abzulegen, muss man diese in die Wrapper-Klassen verpacken.

Viele Lehrbücher beinhalten eine Auflistung der Wrapper-Klassen mit den zugehörigen Klassenbeschreibungen. Zur Verwendung der Wrapper schlagen viele Lehrbücher folgendes Vorgehen vor (vgl. [Flan1999] S. 146):

```
Map m = new HashMap();
m.put("key",new Integer(42));
```

Der Datentyp boolean
Um den int-Wert in der assoziativen Liste (Map) zu speichern, wird ein neues Wrapper-Objekt erzeugt, das den Wert enthält. Für int, float und double ist dies die "richtige" Vorgehensweise. Bei booleschen Werten ist dieses Vorgehen sehr kritisch zu beurteilen. Das Beispiel zum Füllen einer Liste mit booleschen Werten soll dies verdeutlichen.

```
private static final int ITERS = 100000;

public Object performTest(){
  List result = new ArrayList(ITERS);
  Random r = new Random();
  for(int i=0;i<ITERS;i++){
    result.add(new Boolean(r.nextBoolean()));
  }
  return result;
}
```

Codefragment 1: Boolean mit Konstruktor

Das Codefragment 1 erzeugt bei jedem Durchlauf ein neues Boolesches-Wrapper-Objekt und legt dieses in der Liste ab.

```
private static final int ITERS = 300000;

public Object performTest(){
  List result = new ArrayList(ITERS);
  Random r = new Random();
  for(int i=0;i<ITERS;i++){
    if(r.nextBoolean()){
      result.add(Boolean.TRUE);
    }else{
      result.add(Boolean.FALSE);
    }
  }
  return result;
}
```

Codefragment 2: Boolean mit vordefinierten Instanzen

Das Codefragment 2 trägt eine vorhandene Referenz auf ein Boolesches-Wrapper-Objekt in die Liste ein. Man mag nun einwerfen, dass die Verwendung eines Objektes an mehreren Stellen zu Seiteneffekten führen kann und deshalb die zweite Variante nicht korrekt funktionieren kann. Dies ist aber bei den Wrapper-Klassen nicht der Fall, da jede Instanz unveränderbar (immutable) ist. Unveränderbare Objekte zeichnen sich dadurch aus, dass ihr interner Zustand nach der Erzeugung nicht mehr veränderbar ist. Dadurch sind Seiteneffekte und Probleme bei nebenläufiger Ausführung (Multithreading) auszuschließen (vgl. [Bloch2001]).

Das Diagramm unten zeigt, dass das Laufzeitverhalten der beiden Varianten stark unterschiedlich ist. Die Cache-Variante ist bei allen VMs schneller. Der Faktor zwischen den beiden Varianten schwankt dabei zwischen 1,4 und 2,5. Demgegenüber schwankt

der Faktor zwischen der langsamsten und der schnellsten VM auf Serverseite zwischen 1,85 bei BooleanCache und 2,78 bei BooleanSimple. Insgesamt ist eine maximale Performancesteigerung auf Serverseite durch Auswahl der SUN VM 1.4.2, sowie durch Verwendung von BooleanCache gegenüber SUN 1.3.1 und BooleanSimple um den Faktor 4,5 möglich. Dabei macht die Auswahl der geeigneten VM über die Hälfte des Performancegewinns aus.

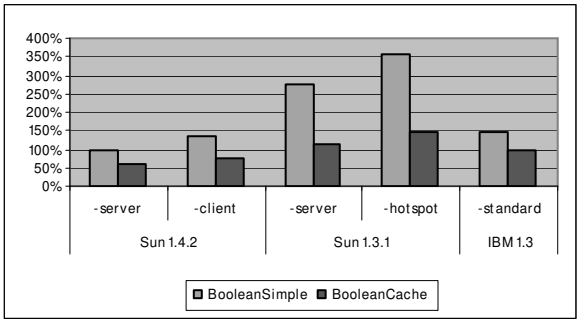

Diagramm 1: Laufzeit der Boolean-Programme

Der Unterschied zwischen den beiden Algorithmen BooleanCache und BooleanSimple lässt sich dadurch erklären, dass die erste Variante im Gegensatz zur zweiten Variante nicht bei jedem Durchlauf ein neues Objekt erzeugt.

Name	Boolean	ArrayList	Random
BooleanSimple	300000	1	1
BooleanCache	0	1	1

Tabelle 1: Erzeugte Objekte der Boolean-Programme

Die Tabelle 1 enthält die Anzahl der erzeugten Objekte der beiden Beispielprogramme. Anhand dieser Daten wird offensichtlich warum BooleanCache schneller ist.

Die Unterschiede zwischen der 1.4.2 und 1.3.1 von SUN sind erstaunlich (vgl. Diagramm 2). Offensichtlich sind Optimierungen bei der Objekterzeugung und bei Methodenaufrufen erfolgt. Diese Ergebnisse sprechen für den Einsatz der VM 1.4.2 von SUN. Generell sind die Server-Varianten bei den Testprogrammen etwas performanter. Die Server-Varianten versuchen die Laufzeit von häufig verwendeten Codeabschnitten maximal zu optimieren. Die Client-Variante bzw. Hotspot-Variante optimiert vor allem die Startzeit und den Speicherverbrauch (vgl. [SUN2003]).

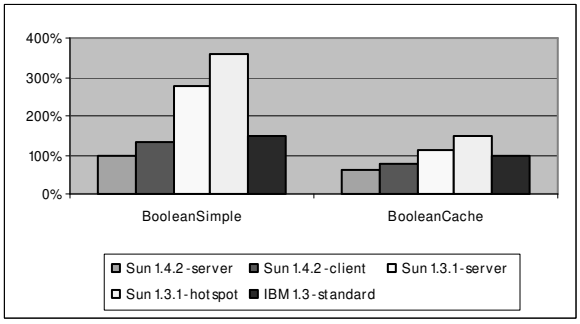

Diagramm 2: Laufzeit der Boolean –Programme (VM)

Die IBM VM 1.3 liegt im Vergleich zur SUN 1.3.1 sehr gut. Der SUN 1.4.2 kann sie aber nicht ganz das Wasser reichen. Vergleichsmessungen mit einer 1.4 VM von IBM wären sehr interessant. Zurzeit ist aber eine solche VM für Windows nicht verfügbar.

Der Vollständigkeit halber zeigt Diagramm 3 die Laufzeiten für die interpretierte Ausführung (Xint und classic), die jedoch aus Performancegesichtspunkten nicht zu empfehlen ist. Im Folgenden verzichten wir daher auf diese Messergebnisse.

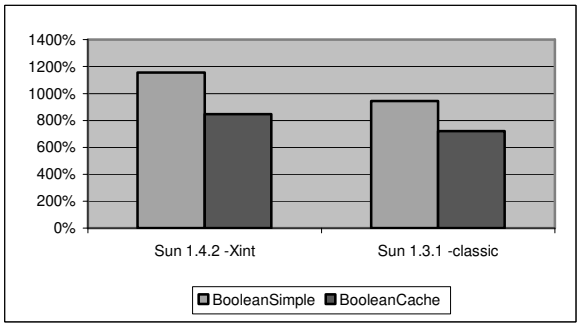

Diagramm 3: Laufzeit Boolean-Programme (Interpreter)

Der Zeitanteil für die Garbagecollection ist in den gemessenen Werten nicht enthalten (siehe Umgebung). Die Faustformel „Je mehr Objekte, umso länger dauert die Garbagecollection“ gilt aber auch hier. Das führt zu der Vermutung, dass die gemessenen Faktoren in realen Programmen noch größer sind. Auf jeden Fall ist das Laufzeitverhalten der Variante 2 besser vorhersagbar. Um diese Aussage zu untermauern wurde der vorhandene Speicher künstlich begrenzt. Dies ist mittels des VM-Parameters Xmx möglich. Das Diagramm 4 zeigt anschaulich, dass der zu erwartende Performancegewinn bei knappem Speicher sehr viel höher liegt. Der Faktor bewegt sich zwischen 2,1 bei der IBM VM und 7,3 bei der SUN 1.3.1. Die Cache Varianten sind im Gegensatz dazu kaum langsamer. Bei der SUN VM 1.3.1 sogar schneller. Die IBM VM scheint bei der Garbagecollection effizienter zu sein.

Der Datentyp byte

Bei der Verwendung des elementaren Datentyps „byte“ kann durch einen entsprechenden Caching-Mechanismus dasselbe Verhalten erreicht werden. Der

Wertebereich eines byte in Java ist -128 <=x<128. In der Standard-API sind aber keine vordefinierten Wrapper-Objekte vorhanden. Das „Codefragment 3: ByteCache“ enthält eine Implementierung für einen

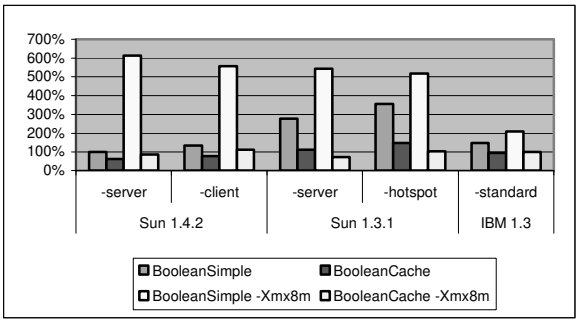

Diagramm 4: Laufzeit der Boolean-Programme mit begrenztem Speicher

Byte-Wrapper-Cache der für die möglichen Ausprägungen (maximal 256) ein Objekt vorhält. Der korrekte Einsatz dieses Caches kann die erzeugte Objektmenge erheblich reduzieren. Die Erzeugung der Instanzen darf dabei nicht über einen der vorhandenen Byte-Konstruktoren erfolgen, sondern über die Factory-Methode (vgl. [Gamm1996]) des Caches. Die Factory-Methode liefert zu dem gegebenen byte-Wert eine Referenz auf die entsprechende Instanz im internen Cache. Dieser Cache ist als Array realisiert und der Zugriff auf die Elemente erfolgt über den gegebenen byte-Wert mit einem Offset zur Normalisierung.

```
public class ByteCache {
  private static final Byte[] BYTE_CACHE;
  private static final int BC_OFFSET =
     Byte.MIN_VALUE * -1;
  static {
   BYTE_CACHE = new Byte[256];
   for(int i=Byte.MIN_VALUE;i<=Byte.MAX_VALUE;i++)
     BYTE_CACHE[i+BC_OFFSET] = new Byte((byte) i);
  }
  private ByteCache(){}
  public static Byte getByte(byte val){
    return BYTE_CACHE[val+BC_OFFSET];
  }
  public static Byte valueOf(String val){
    return BYTE_CACHE[Byte.parseByte(val,10) +
                      BC_OFFSET];
  }
  public static Byte valueOf(String val, int rad){
    return BYTE_CACHE[Byte.parseByte(val,rad) +
                      BC_OFFSET];
  }
}
```

Codefragment 3: ByteCache

Der Cache hat den großen Vorteil, dass er genau 256 Byte-Wrapper-Objekte und ein zusätzliches Array mit 256 Einträgen erzeugt. Dies ist unabhängig von der Anzahl der verwendeten Byte-Wrapper-Objekte im Programm. Der Speicherbedarf ist damit einfach berechenbar (vgl. Tabelle 2). Die Testprogramme sind äquivalent zu den Boolean-Testprogrammen.

Name	Byte	Byte[]	ByteArrayList
ByteSimple	300000	0	0
ByteCache	256	1	0
BytePCJ	0	0	1

Tabelle 2: Erzeugte Objekte der Byte-Testprogramme

Das Laufzeitverhalten ist dem bei der Verwendung von Booleans sehr ähnlich. Nur die Faktoren sind nicht ganz so hoch. Dies liegt sehr wahrscheinlich an der zusätzlichen Indirektion beim Zugriff auf das Array. Auffallend ist der geringe Vorteil des Caches beim Einsatz der IBM VM. Dies lässt die Vermutung zu, dass die IBM VM effizient bei der Erzeugung von Objekten ist.

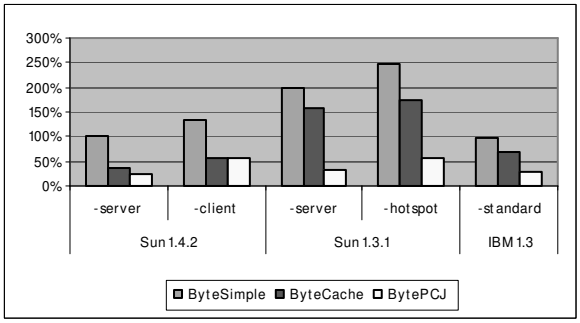

Diagramm 5: Laufzeit der Byte-Programme

Eine weitere Performancesteigerung ist durch den Einsatz von spezialisierten Collections-Klassen erreichbar. Im OpenSource-Bereich gibt es die „Primitve Collections for Java“-API (vgl. [Pcj2003]). Sie beinhaltet spezielle Collections für die primitiven Datentypen. Diese Collection-Klassen arbeiten direkt auf den primitiven Datentypen und verwenden somit keine Wrapper-Klassen zur Ablage. Da die VMs sehr effizient mit elementaren Datentypen umgehen, sollte sich die Performance dadurch nochmals steigern lassen. Das Codefragment 4 zeigt die Benutzung der PCJ-Klassen anstatt der Standard-Collection-Klassen.

```
private static final int ITERS = 300000;

public Object performTest(){
  ByteArrayList l = new ByteArrayList(ITERS);
  Random r = new Random();
  for(int i=0;i<ITERS;i++){
    l.add((byte) (r.nextInt(256)-128));
  }
  return l;
}
```

Codefragment 4: Byte mit PCJ-Container

Die gemessenen Werte für die Klasse BytePCJ im Diagramm 5 bestätigen den erwarteten Performancegewinn. Die Tabelle 2 zeigt, dass nur ein Objekt vom Typ „ByteArrayList“ erzeugt wird. Die Ablage der bytes erfolgt also nicht über die Wrapper-Klassen.
Der Einfluss der verwendeten VM auf die Laufzeit entspricht dem bei den Boolean-Programmen. Erstaunlich ist, dass der Cache bei der SUN 1.3.1 nur

einen kleineren Vorteil gegenüber der Standard-Implementierung bietet. Die IBM VM liegt hier fast gleich auf mit der SUN 1.4.2 (vgl. Diagramm 6).

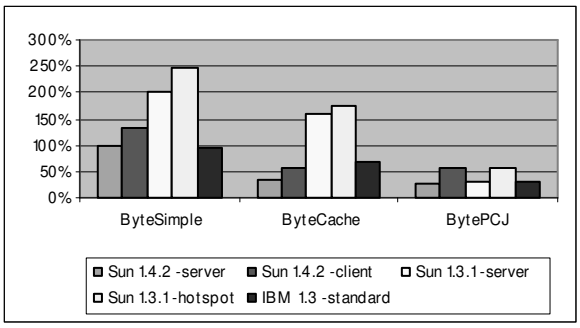

Diagramm 6: Laufzeit der Byte-Programme (VM)

Der Einsatz des Caches bietet sich auch bei der Umwandlung von Strings in die Wrapper-Klassen an. Die valueOf()-Methode der booleschen Wrapper-Klasse verwendet intern die beiden vordefinierten Instanzen Boolean.TRUE und Boolean.FALSE. Eine Optimierung ist in diesem Fall nicht möglich. Bei bytes erfolgt die Erzeugung intern mittels des Byte-Konstruktors. Es bietet sich an für diese Aufgabe den Cache einzusetzen. Deshalb beinhaltet der ByteCache eine Methode valueOf(), die die gecachten Instanzen verwendet. Bei der Verwendung dieser Methode sollte demnach wieder eine Performanceverbesserung messbar sein.

Das Diagramm 7 beinhaltet das gemessene Laufzeitverhalten. Die ermittelten Werte bestätigen die Ergebnisse der vorherigen Messungen. Der Vorteil ist jedoch durchwegs geringer.

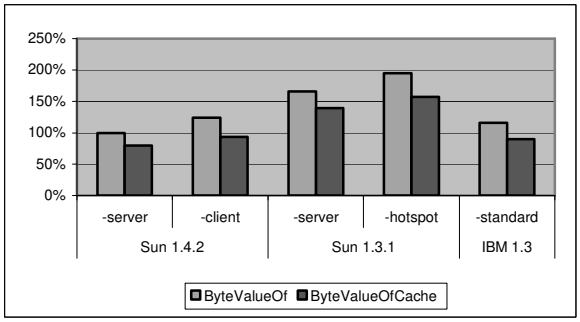

Diagramm 7: Laufzeit ByteValueOf-Programme

Einfache Zahlentypen

Der vorgestellte Cache-Mechanismus ist auch für Shorts verwendbar. Man muss sich dabei aber im klaren sein, dass der Cache 2^16 Objekte enthält. Diese Optimierung lohnt sich also nur bei der Verarbeitung von großen Datenmengen. Die ShortCache Implementierung ist äquivalent zu der des ByteCache.

Bei Integern sind für den Cache 2^32 Einträge notwendig. Von der Verwendung eines kompletten Caches ist bei der zurzeit handelsüblichen Speicherausstattung daher abzuraten. Dennoch sind Varianten denkbar, bei denen der Cache nach einer Heuristik nur häufig verwendete Teile enthält. Bei diesen Varianten bleibt festzuhalten, dass sie nur einen Performancevorteil bringen, falls die zu bearbeitenden Werte nicht normalverteilt sind. Dies wird durch die Werte im Diagramm 8 belegt. Das Verhalten lässt sich dadurch erklären, dass der Verwaltungsaufwand für den Cache im Verhältnis zu den Cachetreffern zu groß ist.

Als Beispiel für eine Performancesteigerung dient folgendes Szenario: 90% der zu verarbeitenden Werte liegen innerhalb eines definierten Intervalls der Rest außerhalb.

```
public static class IntegerIntervalCache {
  private static Integer[] INT_CACHE;
  private static int INT_START;
  private static int INT_END;
  private static int OFF;
  private IntegerIntervalCache(){}
  public static void init(int s, int l){
    INT_CACHE = new Integer[l+1];
    INT_START = s;
    INT_END = s+l;
    OFF = - s;
    for(int i=OFF+INT_START;i<OFF+INT_END+1;i++)
      INT_CACHE[i] = new Integer(i);
  }
  public static Integer getInteger(int i){
    if(i<INT_START || i>INT_END)
      return new Integer(i);
    return INT_CACHE[OFF + i];
  }
}
```

Codefragment 5: IntegerIntervalCache

Für diesen Fall kann man einen Intervall-Cache verwenden, der alle Werte innerhalb des Intervalls enthält und für die Werte außerhalb jeweils ein neues Objekt erzeugt. Eine Beispielimplementierung ist unter Codefragment 5: IntegerIntervalCache zu finden.

Eine weitere denkbare Alternative ist der Einsatz einer assoziativen Liste (HashMap) als Cache. Der Cache überprüft dabei zuerst ob der gewünschte int-Wert schon gespeichert ist. Falls ja wird eine Referenz auf den vorhandenen Wert zurückgeliefert. Andernfalls wird ein neues Wrapper-Objekt erzeugt, im Cache abgelegt und die Referenz zurückgegeben (lazy initialization). Der Einsatz einer Standard-HashMap ist nicht sinnvoll möglich, da der Schlüssel innerhalb der Map ein Objekt sein muss. Dies führt in unserem Beispiel dazu, dass auf jeden Fall ein Objekt für den Zugriff auf die Map erzeugt werden muss. Genau dies sollte der Cache verhindern. Die implementierte Variante basiert daher auf der IntHashMap aus der PCJ. Bei dieser Variante ist aber Vorsicht angesagt, da der Cache wächst, bis alle möglichen Integer-Wrapper-Objekte abgelegt sind oder der Speicher voll ist. Im Gegensatz zum IntervalCache muss bei dieser Variante das Intervall nicht im Voraus bekannt sein. Die gemessenen Werte (vgl. Diagramm 8) zeigen, dass bei diesem Szenario nur der vorinitialisierte IntervalCache einen Vorteil bringt.

Der Verwaltungsaufwand der Map ist im Verhältnis zur Erzeugung des Wrapper-Objekts zu hoch. Die Verwendung einer spezialisierten Collection ist erneut die schnellste Variante.

```
public class IntegerCache {
  private static IntKeyOpenHashMap INT_CACHE =
                          new IntKeyOpenHashMap();
  private IntegerCache(){}
  public static Integer getInteger(int i){
    Integer result = (Integer) INT_CACHE.get(i);
    if(result == null) {
      result = new Integer(i);
      INT_CACHE.put(i, result);
    }
    return result;
  }
}
```

Codefragment 6: IntegerCache

Fazit

Der Einsatz einer „neuen“ VM kann enorme Performancevorteile bringen. Zu beachten ist dabei, dass bei realen Softwaresystemen die Beschleunigung aufgrund ihrer spezifischen Rahmenbedingungen, wie z.B. der Datenhaltung nicht so groß sein wird, wie bei unseren kleinen Beispielprogrammen. Der besondere Charme am Einsatz einer performanceoptimalen VM liegt darin, dass keinerlei Eingriffe in den Code notwendig sind. Gespannt kann man auf die IBM VM 1.4 für Windows sein.

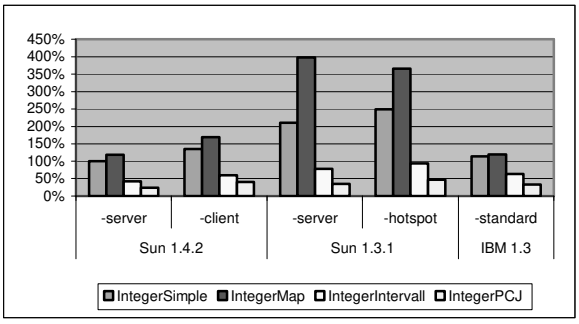

Diagramm 8: Laufzeit der Integer-Programme

Ein Allheilmittel ist der Einsatz einer schnelleren VM aber nicht. Effiziente Algorithmen tragen noch immer einen großen Teil zur Performance eines Systems bei. Nach unseren Messungen sollte man beim Einsatz der Wrapper-Klassen für boolean, byte, short und int immer einen entsprechenden Cache verwenden. Dies reduziert die Anzahl der erzeugten Objekte erheblich. Das Laufzeitverhalten, insbesondere bei Rechnern mit geringem Speicher, wird bei allen VMs positiv beeinflusst. Der Einsatz der Caches wirkt sich dabei nicht negativ auf die Wartbarkeit und die Lesbarkeit der Programme aus. Falls die Performance der Cache-Varianten nicht ausreicht kann auf die spezialisierten Collections zurückgegriffen werden.

Bei individuellen Softwaresystemen werden häufig „eigene“ Datentypen erzeugt. Diese sollte man nach Möglichkeit als unveränderbare Objekte designen. Bei der Verwendung dieser nicht veränderbaren Objekte (immutables) bietet es sich an schon beim Design an Caching-Mechanismen zu denken. Müssen bei der Konstruktion der Datentypen aufwändige Operationen durchgeführt werden, kann ein Caching mit lazy initialization durchaus einen signifikanten Performancevorteil bringen. Das Factory-Pattern mit privaten Konstruktoren hilft dabei, das Caching durchgängig zu verwenden.

Erläuterungen zur Messumgebung

Alle Messungen erfolgten mit einem eigenen Framework auf dem Betriebssystem Windows XP. Das Framework versucht die Durchführung einer Garbagecollection (GC) zwischen den Aufrufen der einzelnen Testprogramme anzustoßen. Dabei ist zu beachten, dass der Aufruf der Methode System.gc() nicht direkt zu einer Durchführung der GC führt. Vielmehr signalisiert er der VM, dass jetzt ein guter Zeitpunkt dafür wäre.

Die benötigte Zeit der GC ist nur schwer zu messen. Dies liegt an der asynchronen Ausführung. Es gibt zwar die Möglichkeit mit dem VM-Schalter „verbosegc“ einen entsprechenden Report zu erzeugen, die Zahlen sind aber nur schwer einzelnen Testläufen zuzuordnen. Deshalb ist bei allen Messungen dieser Faktor nicht enthalten. Die VM erhält bei den Testläufen mit den Parametern Xmx und Xms 256MB Hauptspeicher und somit ist eine asynchronce GC i.d.R. nicht notwendig.

Die in den Tabellen angegeben Objektmengen ermittelt das Framework durch eine modifizierte Object-Klasse (vgl. [SHIR2000]). Die modifizierte Klasse speichert für jeden Objekttyp einen Zähler in einer assoziativen Liste (HashMap). Dies geschieht innerhalb des Object-Konstruktors. Die veränderte Klasse muss als Bootstrap-Klasse noch vor der Runtime-Umgebung der VM bekannt gemacht werden. Mit diesem Verfahren ist es leider nicht möglich die Erzeugung von Arrays zu erfassen. Die vorhandenen Zahlen wurden deshalb anhand des Sourcecodes berechnet und nicht gemessen. Bei den Laufzeitmessungen erfolgt eine Durchschnittsbildung über 100 Testläufe. Der Algorithmus entfernt dabei den minimalen und maximalen Wert. Die Zeitmessungen erfolgen mittels der System.currentTimeMillis()-Methode.

Literatur:

[Bloch2001] Effective Java Programming Language Guide; Joshua Bloch; 2001; AW

[Flan1999] Java in a Nutshell; 3rd edition; David Flanagan, 1999, O’Reilly

[Gamm1996] Entwurfsmuster: Elemente wieder verwendbarer objektorientierter Software; Gamma et al.; 1996; AW

[Pcj2003] http://pcj.sourceforge.net/

[Shir2000] Java Performance Tuning; Jack Shirazi; 2000; O’Reilly

[Sun2003] http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf