Java Performance

HS Telematik
Torsten Hildebrandt
2002-07-05

Gliederung

1. Allgemeines
2. Strings
3. Nebenläufigkeit
4. Speicherverwaltung
5. Container
6. I/O
7. Sonstiges

Allgemeines

Optimierung ist Transformation bestehenden Codes:

- Korrektheit muss erhalten bleiben
- Neuer Code effizienter als Original
- Neuer Code weniger universell → schränkt Wiederverwendbarkeit ein
- Resultierender Code meist komplexer, schwieriger lesbar, schwerer zu warten

→Balance zw. Ausführungsgeschwindigkeit und anderen SW-Qualitätsmassen

80/20-Prinzip

Ausprägungen:

- 20% des Codes nehmen 80% Laufzeit in Anspruch
- 80% der benötigten Zeit in nur 20% der Funktionen
- 80% der mgl. Anwendungsszenarios werden nur 20% des Codes berühren
- 20% der Optimierungen bringen 80% des Geschwindigkeitszuwachses
- Grossteil der Eingabe-Daten wird nur aus einem kleinen Bereich der mgl. Daten kommen

Schlüsse aus 80/20

- Zufällige Optimierungen zu 80% sinnlos, Optimierungen ausserhalb des kritischen Pfades bewirken keine signifikanten Geschwindigkeits-verbesserungen
- Investitionen in Optimierung sollen signifikante Verbesserungen bringen

- Ermittlung des kritischen Pfads mittels Profiler, im SDK 1.4 mittels `-prof`, `-Xprof` , `-Xrunhprof`, `-Xaprof` , `-Xhprof`
- Schaffung einer Testumgebung

Vorgehensweise

1. Identifikation des Flaschenhalses
2. Setzen eines Zieles für Geschwindigkeit
3. Nutzen repräsentativer Daten
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.

Prinzipien

- Wahl geeigneter Datenstrukturen
- Wahl eines geeigneten Algorithmus
- Vermeiden unnötiger und doppelter Berechnungen
- Verschieben von Berechnungen, z.B. Herausziehen von Schleifen-invarianten Ausdrücken
- Ausnutzen von Spezifika des Anwendungsbereichs
- Zwischenspeichern von Berechnungen

Strings (1/3)

- schlecht:

```
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();
```

- Warum so langsam?

  Zeile 4 wird zu:

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

Strings (2/3)

- besser:

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

- noch besser:

```
6 StringBuffer sb = new StringBuffer(20000);
7 for (int i = 0; i < 20000; i++) {
8     sb.append('a');
9 }
10 String s = sb.toString();
```

Strings (3/3)

- Sinnvolle Umwandlung?

```
 1 String s = "Test " + i + "!";
→
 2 StringBuffer sb = new StringBuffer();
 3 sb.append("Test ");
 4 sb.append(i);
 5 sb.append("!");
 6 String s = sb.toString();
```

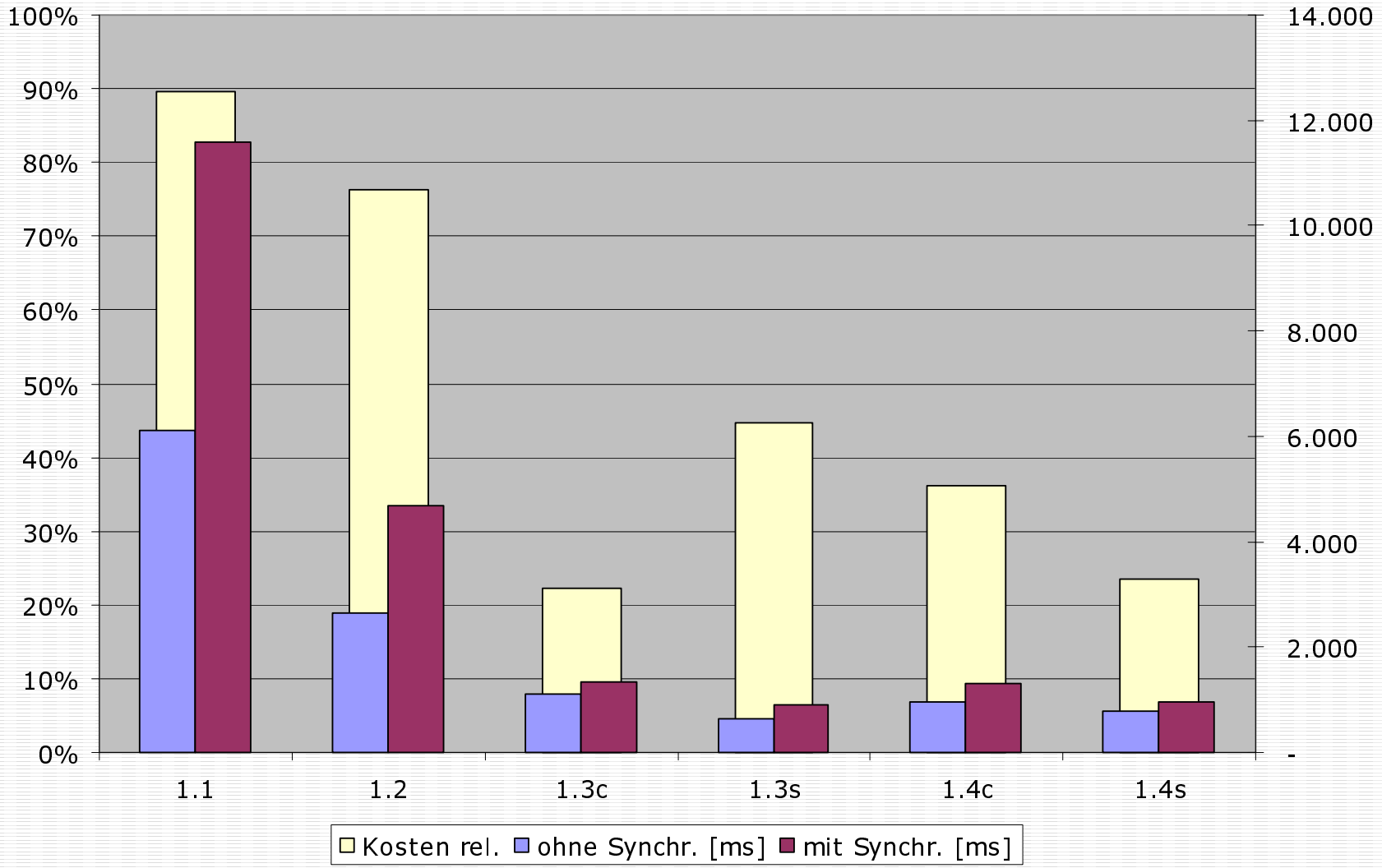
Kosten der Synchronisierung (1/2)

- Testcode (Aufruf 20 Mio. mal):

```
1 public void test1(StringBuffer sb)
2 {
3         sb.append('a');
4 }

5 public synchronized void test2(StringBuffer sb)
6 {
7         sb.append('a');
8 }
```

Kosten der Synchronisierung (2/2)



Nebenläufigkeit

- Möglicher Geschwindigkeitszuwachs begrenzt durch sequentiell auszuführenden Teile
- Granulierungsproblem:
  - durch grosse synchronisierte Bereiche langes Blockieren anderer Threads
  - kleine synchronisierte Blöcke bedeuten höheren Synchronisationsaufwand
- Keine Verzögerungsschleifen (stattdessen `sleep()`)
- Thread-Pooling
- Evtl. weitergehende Synchronisationsmethoden impl. (etwa Read-/Write-Locks)
- Vermeidung durch Duplizierung von Ressourcen

Speicherverwaltung (1/2)

- Begrenzte Interaktion mit GC durch Reference-Objekte
- Arten der Erreichbarkeit von Objekten:
  1. Strongly reachable – über normale Refernz erreichbar
  2. Softly reachable – nur noch über SoftReference erreichbar
  3. Weakly reachable – weder 1. noch 2., aber über mind. 1 WeakReference-Objekt zu erreichen
  4. Phantom reachable – wenn weder 1.-3., aber noch über PhantomReference erreichbar
  5. Unreachable – weder Referenz noch Referenzobjekt verweisen auf dieses Objekt
- Nutzbar z.B. für intelligente Caches

Speicherverwaltung (2/2)

- `WeakHashMap`
- Objekterzeugung auf Minimum beschränken
- Objekte recyclen, evtl. Pooling einführen

Container

- ArrayList statt Vector
- Bei Hashtable:
  - `equals()` und `hashcode()` wichtig für Performance, evtl. auch für Strings und Container neue Methoden implementieren
- Wie effizient iterieren?

```
 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 for (Iterator it = l.iterator(); it.hasNext();)
 7 {
 8   Integer o = (Integer) it.next();
 9   sum += o.intValue();
10 }
```

IO

- Byte-IO vs. Character-IO
- BufferedReader/-Writer/-... verwenden

```
    1 printstream.write("Test "+i+"!");

→   2 printstream.write("Test ");
    3 printstream.write(i);
    4 printstream.write("!");
```

NIO (1/3)

- Seit SDK 1.4
- unterstützt Memory-mapped files
- Direkte Puffer
- Non-blocking I/O

- Server bisher:

```
1 ServerSocket s = new ServerSocket();
2 Socket conn = s.accept( );
3 InputStream in = conn.getInputStream();
4 ... // aus 'in' lesen
5 OutputStream out = conn.getOutputStream();
6 ... // Response schreiben
7 conn.close();
```

NIO (2/3)

Vorarbeiten:

```
1 InetSocketAddress isa =
      new InetSocketAddress(port);
2 Selector selector = Selector.open();
3 ServerSocketChannel channel =
      ServerSocketChannel.open();
4 channel.configureBlocking(false);
5 channel.socket().bind(isa);
6 channel.register(selector,SelectionKey.OP_ACCEPT);
```

NIO (3/3)

```
1 while (selector.select() > 0) {
2   Set readyKeys = selector.selectedKeys();
3   Iterator readyItor = readyKeys.iterator();
4   while (readyItor.hasNext()) {
5     SelectionKey key = (SelectionKey)readyItor.next();
6     readyItor.remove();
7     if (key.isAcceptable()) {
8       ServerSocketChannel ssc = (ServerSocketChannel)
        key.channel();
9       SocketChannel socket = (SocketChannel) ssc.accept();
10      socket.configureBlocking(false);
11      socket.register(selector,
   SelectionKey.OP_READ|SelectionKey.OP_WRITE);
12    } // key.isAcceptable()
13    if (key.isReadable()) {
        ... // aus Channel lesen
14    }
    ...
```

Sonstiges

- autom. Initialisierung aller Variablen mit 0 bzw. `null`
- Date-Objekte erzeugen teuer
- `Exceptions` sind teuer
- Konstanten möglichst `static final` deklarieren
- `instanceof` vor Cast wirklich nötig?
- XML-Parsing: SAX statt DOM

Wenn alles nichts hilft...

- Java Native Interface (JNI)?
- Java To Native Compiler?
  (z.B. GNU Compiler for Java)

Ende

Vielen Dank für Ihre Aufmerksamkeit!

http://www.javaperformancetuning.com/