# IDPA: Programmiersprachen benchmarken

Nick Zbinden und Matthias Gasser

24. Februar 2011

# Inhaltsverzeichnis

# 1 Vorwort

Dieses Projrkt ist aus dem Willen heraus entstanden eine Arbeit zu machen die Applikations. und Systemtechnikenkenntniss nuetzt und verbindet. Nach verschiedenen Ideen einigten wir uns ...

# 2 Abstract

# 3 Einleitung

## 3.1 Untersuchungsgegenstand

Wir haben uns ein sehr kompliertes Thema vorgenommen. Programmiersprachen und Compiler sind, zumindes für die Informatik, einen sehr altes Thema.

Hochsprachen wie wir Sie in dieser Arbeit verwenden werden seit mehr als fünfzig Jahren immer wieder weiter entwickelt und verbessert. Seit der Anfangszeit herscht der Konflikt zwischen hoher Abstraktion und Geschwindikeit. Umso höher die Abstraktion ist die eine Programmiersprache bietet umso schwieriger ist die Programmiersprache auf der darunterliegenden effizient auszuführen.

## 3.2 Problemstellung

Wir haben uns die Aufgabe gestellt drei Programmiersprachen unter dem asbekt der Geschwindigkeit anzuschauen und zu vergleichen. Eine definitives Ergebniss ist in diesem Bereich fast unmöglich da die Anwendungsgebiete einer Programmiersprache zu verschieden sind und jeder Anwendungsfall andere Requirements hat. Auch die unter der Programmiersprache liegenden Layers (Betriebssysteme und Hardware) sind von entscheidender Bedeutung. Um korrekte Aussagen zu machen müssen diese Layers entweder herausgerechnet werden oder identisch sein. Das Ziel ist es für den von uns ausgewählten Anwendungsfall Messungen zu machen, Aussagen über die Geschwindikeiten zu treffen und wenn möglich klähren warum die Sprachen so verhält.

## 3.3 Wissenslücken

Um komplizierte Algorithmen in drei verschiedenen Sprachen zu programmieren braucht man viel Erfahrung in diesen Sprachen. Um den Aufwand nicht zu hoch werden zu

lassen haben wir uns auf die Implementierung in einer Programmiersprache spezialisiert und vergleichen diese mit reference Implementationen die wir nur Erklähren und messen.

## 3.4 Erwartungen

Wegen er unter der Sprachen liegenden Infrastruktur erwarten wir sehr ähnliche Testresultate. Es ist das erklährte Ziel von Clojure und Scala genau so schnell zu sein wie Java um nie auf Java zurück fallen zu müssen.

Was wir allso testen ist die effizienz der Sourcecode zu Bytecode Compiler. Diese Technischen Details werden später in dieser Arbeit verwendet.

# 4 Material und Methoden

## 4.1 Allgemeines

### 4.1.1 Sprachauswahl

In den letzten Jahren ist ein neuer Trend erwachsen. Programmiersprachen werden oft nicht mehr von Grund auf neu aufgebaut sondern versuchen bestehende Infrastrukturen zu verwenden.
Diese bringt Vorteile und Nachteile mit sich. Der Vorteil ist, dass man sich als Trittbrettfahrer an einer VM anhängen kann. In moderne VM wie der JVM wurden hunderte von Mannjahren investiert und sind daher sehr stabiel und schon fast überall installiert.

Der zweite Vorteil ist das Interoptabilität zwischen den verschiedenen Sprachen erreicht werden kann. Somit kann Code wiederverwendet werden ohne, dass man Zeit in portierung von Programmen stecken muss die jede Sprach braucht um im echten Leben verwendung zu finden beispielsweise Datenbank Protokolle, XML Parsers usw.

In der Java Programmiersprach sind all diese Grundlegenden Programme schon implementiert und die JVM ist eine geeignet Plattform um ander Sprachen darauf aufzusetzten.

Die entscheidene Frage ist nun warum sollten die Nutzer die neuen Sprachen benützen. Die Antwort ist sehr vielfälltig. Einige neue Sprachen (Scala) bieten bessere Typsysteme und dadurch besser Compiletime Sicherheiten andere Sprachen wie z.B. Groovy versuchen Features von sehr dynamischen Sprachen zu unterstützen.

Alle diese neuen und spannenden Features sind fantastisch kommen aber oft auf Kosten der Performance. Das kommt davon das manche Features (Bouncechecking) einfach per definition Laufzeit brauchen aber oft ist das Problem auch der Unterschied in den Sprache Semantiken d.h. wenn in einer Sprache ein Feature unterstützt ist aber in der Host-VM nicht muss um das Problem herum gearbeitet was zusätzlichen Runtime-Overhead verursacht. In dieser Arbeit haben wir uns zwei der meiste

verwendeten neuen JVM-Sprachen ausgesucht um herauszufinden ob diese es schaffen die Geschwindikeit der Hostsprache (Java) zu erreichen.

### 4.1.2 Zielanpassung: Sprachanpassung

Im Expose haben wir die drei JVM-Sprachen Clojure, Scala und JRuby gennant die wir Benchmarken wollten. Während der Arbeit haben wir festgestellt, dass JRuby andere Ziele vorfolgt als maximale Performance zu bieten und deshalb weder produktiv noch sinnvoll ist JRuby mit Clojure und Scala, die beide diesen Anspruch haben, zu vergleichen.

Um JRuby zu ersetzen hätten wir eine weiter neue JVM-Sprache hernehmen können (es gibt mehr als einhundert) aber es gibt kaum Sprachen die weit genug entwickelt sind oder die ähnliche Ziele verfolgen. Deshalb haben wir uns Entschieden den test zu machen ob Clojure und Scala wirklich die Geschwindigkeit von Java erreichen

### 4.1.3 Zielanpassung: Implementierung

Zuerst war die Idee ein Algorithmus in drei verschiedenen Sprachen zu implementieren und dann die Performence zu messen. Um einen mehr oder weniger komplexen Algprithmus in drei Sprachen zu implementieren muss man diese Sprachen sehr gut kennen und verstehen. Noch schwieriger wird es wenn Performance noch von grosser wichtigkeit ist. Viele Sprachen kann man extrem verbiegen was es, auf Kosten von Einfachheit und Klarheit, erlaubt die Geschwindkeit zu verbessern.

Um solche Progamme zu schreiben sind wir fähig und sich sowas anzueignen ist of mit Monate langer einarbeitung Verbunden. Deshalb beschränken wir uns, bei der implementierung, auf eine Sprache. In den anderen nehmen Reference implementierung und werden diese nur testen und erklären.

## 4.2 Grundlagen

### 4.2.1 Virtuelle Maschienen

Um es simpel zu halten haben wir uns Entschieden Programmiersprachen zu nehmen die auf der JVM (oder anderen VMs die Java Byte Code ausführen) laufen. Dies erlaubt uns Aussagen über die Codegenerierung des Source-to-Bytecoden compilers der Sprache zu machen.
Da VM einen Bytecode als Input erhalten ist es grundsätzlich möglich jeder Programmiersprache auf einer VM laufen zu lassen. Wie dieser Bytecode in der VM ausgeführt wird ist den darüber liegenden Sprachen egal.
Einige Möglichkeiten wie eine VM den Bytecode ausfühen:

- Interpreter
- JIT-Compiler

- Native Code Compiler
- Direkt auf Hardware

Ich möchte nur auf den JIT-Compiler etwas näher eingehen da die JVM die wir benützen einen solchen verwendet (Hotspot). Um zu verstehen warum ein Programm langsam oder schnell ist muss man bis zu einem gewissen grad den Compiler verstehen.

#### 4.2.2 JIT Compiler

Ein JIT-Compiler compiliert nicht alles auf einmal sonder, nur den Code den auch wirklich braucht. Das erlaubt es dem Compiler obtimierungen an den wichtigen Stellen anzubringen. Ein weiterer grosser Vorteil ist es, dass dem JIT-Compiler die Umgebung auf der er sich befindet bekannt ist. Das erlaubt es obtimierungen anzubringen die speziell für diese Hardware den Code anpassen.
Gute JIT-Compiler sind heutzutage vergleichbar mit der Geschwindikeiten von Native Code Compilern. Die Unterschiede sind in vielen Anwendungsfällen nur noch gering.

# 5 Programmiersprachen und Implementierungen

## 5.1 Java

### 5.1.1 Beschreibung

Java ist eine Programmiersprache die von 1992 an von Sun Microsystems (oder teilweise im Auftrag von Sun) entwickelt wurde. Java wurde zuerst für Embedet Systems geschrieben. Heutzutage findet man Java Applikation in vielen Anwendungsgebieten, insbesondere Enterprice Applikationen.

## 5.2 Scala

### 5.2.1 Beschreibung

Die Entwicklung von Scala hat 2001 an der ETH Lausanne unter Martin Odersky begonnen. Die Idee ist es eine Sprache zu designen die Konzepte eine Funktionalen und Objektorientiertersprachen in syntese verwendet. Zwar wurde Scala im Akademischen Kontext entwickelt findet aber immer mehr Anwendungen im Bussness Bereichen

## 5.3 Clojure

### 5.3.1 Beschreibung

Clojure ist eine dynamische Programmiersprache die 2007 von Rich Hicky für die JVM geschrieben wurde. Im Vordergrund der Entwicklung standen hohe Abstraktion, vor allem bei Concurency Programming, und gut integration ins Host System (JVM). Diese erlaubt Wiederverwednung von Java Code, Java Ecosystems (Webservers, Profilers, Debuggers ...) und natürlich der VM.

### 5.3.2 Code

```
(ns binarytrees
  (:gen-class))

(defn TreeNode [left right item]
  {:left left
   :right right
   :item item})

(defn bottom-up-tree [item depth]
  (if (zero? depth)
      (TreeNode nil nil item)
      (TreeNode
       (bottom-up-tree (dec (* 2 item))
                       (dec depth))
       (bottom-up-tree (* 2 item)
                       (dec depth))
       item)))

(defn item-check [node]
  (if (nil? (:left node))
    (:item node)
    (+ (:item node)
       (item-check (:left node))
       (- (item-check (:right node))))))

(defn iterate-trees [mx mn d]
  (let [iterations (bit-shift-left 1 (int (+ mx mn (- d))))]
    (format                                          (* 2 iterations) d
            (reduce + (map (fn [i]
                             (+ (item-check (bottom-up-tree i d))
                                (item-check (bottom-up-tree (- i) d))))
                           (range 1 (inc iterations)))))))

(def min-depth 4)

(defn main [max-depth]
```

```
    (let [stretch-depth (inc max-depth)]
      (let [tree (bottom-up-tree 0 stretch-depth)]
        (println (format
           stretch-depth (item-check tree))))
      (let [long-lived-tree (bottom-up-tree 0 max-depth)]
        (doseq [trees-nfo (map (fn [d]
                                 (iterate-trees max-depth min-depth d))
             (range min-depth stretch-depth 2)) ]
          (println trees-nfo))
        (println (format
           max-depth (item-check long-lived-tree)))
        (shutdown-agents))))

(defn -main [& args]
  (let [n (if (first args) (Integer/parseInt (first args)) 0)
        max-depth (if (> (+ min-depth 2) n) (+ min-depth 2) n)]
    (main max-depth)))
```



VM - Virtuall Maschine
JVM - Java Virtuell Maschine
IL - Interniediet Language



Virtuall Maschine:
Eine Virtuall Maschine ist eine Programm welches simuliert eine echte Maschine zu

sein. Sie bekommt als input irgend eine Sprache und fuert diese auf der darunter liegenden Sprache aus.
Java Virtuall Maschiene:
Eine implementation einer VM die dafuer ausgelegt ist Java Bytecode auszufueren.

Bytecode:
Ein Instruktionsset für eine VM.
Java Bytecode:
Java Bytecode ist der für Java bzw. für die JVM entwickelte Bytecode.

# F Anhang