Programmiersprachen benchmarken -

Programmiersprache ist nicht gleich Programmiersprache

Nick Zbinden und Matthias Gasser

15. März 2011

Berufsbildungszentrum Bau und Gewerbe, Luzern
Klasse: BML08E

Inhaltsverzeichnis

1 Vorwort

Die Informatik hat auf uns schon immer eine gewisse Faszination ausgeübt. Es war daher für uns beide naheliegend eine Lehre als Informatiker zu machen. Während Nick vor allem an der Softwareentwicklung interessiert ist, bevorzuge ich die Systemtechnik, in der die Installation und Wartung von IT-Systemen der Grosse Schwerpunkt ist. Da wir beide unsere Berufe noch immer sehr interessiert ausführen, war es für uns beide klar für die IDPA ein Thema aus dem Bereich Informatik zu wählen. Es galt nun ein Thema zu finden, in dem sowohl die Applikationsentwicklung wie auch die Systemtechnik gleichermassen eine Rolle spielen. Bereits nach kurzer Recherche stiessen wir auf das Thema „Programmiersprachen Benchmarking“. Sofort war unser beider Interesse geweckt und dadurch, dass dieses Thema sowohl ein Testsystem wie auch selbstgeschriebene Programme erforderte, eignete es sich sehr gut für eine IDPA. Wir entschieden uns, unsere Maturaarbeit diesem Thema zu widmen.

Am gewählten Thema interessiert uns beide in erster Linie ob tatsächlich Performanceunterschiede zwischen den untersuchten Sprachen feststellbar sind, und in welchem Ausmass sie sich zeigen. Heutige Computer sind sehr leistungsfähig, dadurch ist es für kleinere Programme wie wir sie testen werden meist irrelevant ob eine Programmiersprache einige Millisekunden schneller ist als die andere. Für grössere Projekte spielt die Geschwindigkeit jedoch auch heute noch eine wichtige Rolle, wir hoffen daher, dass sich unsere Testergebnisse auf grössere Applikationen projizieren lassen.

2 Abstract

Wir haben uns entschieden für unsere Maturaarbeit die Programmiersprachen Java, Clojure und Scala zu vergleichen. Wir haben dazu ein identisches Programm in jeder der genannten Programmiersprachen geschrieben. Durch das Messen von Prozessor- und Speicherdaten sowie der Ausführungszeiten der Programme wollten wir Unterschiede feststellen, um am Ende aufzeigen zu können, welche der getesteten Sprachen vom Testsystem am wenigsten Leistung abverlangt. Für die Messungen haben wir ein Script geschrieben, welches die erforderlichen Daten in angemessenen Zeitabständen ausliest. Nach Auswertung der gemessenen Daten zeigte sich, dass Clojure das System bei weitem am wenigsten belastet. Jedoch war die Ausführungszeit für das Clojure Programm ungefähr drei Mal so lange wie diejenige der anderen Sprachen. Da in der Programmierung heutzutage Geschwindigkeit wichtiger ist als alles andere, raten wir davon ab Clojure für grössere Projekte zu verwenden, da die sehr junge Sprache schlicht noch nicht genügend ausgereift ist. Scala und Java lieferten beide sehr ähnliche Resultate. Bei einer beinahe identischen Ausführungszeit benötigte Scala jedoch immer leicht weniger Systemressourcen als Java. Da Ressourcen bei aktuellen Computern meist jedoch kaum mehr eine Rolle spielen, sollte wenn immer möglich die schnellste Programmiersprache, d.h. Java, verwendet werden.

3 Einleitung

3.1 Untersuchungsgegenstand

Wir haben uns ein sehr komplexes Thema vorgenommen. Programmiersprachen und Compiler sind, auf die Informatik bezogen, 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 herrscht der Konflikt zwischen hoher Abstraktion und Geschwindigkeit. Umso höher die Abstraktion ist, die eine Programmiersprache bietet, umso schwieriger ist die Programmiersprache auf der darunterliegenden Sprache effizient auszuführen.

3.2 Problemstellung

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

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 Referenz-Implementationen die wir nur erklären und messen.

3.4 Erwartungen

Aufgrund der, unter den Sprachen liegenden Infrastruktur, erwarten wir sehr ähnliche Testresultate. Es ist das erklärte Ziel von Clojure und Scala genauso schnell zu sein wie Java, damit der Programmierer niemals aus Geschwindigkeitsgründen auf Java zurückgreifen muss.

Wir haben die Hypothese aufgestellt, dass es für Scala relativ einfach ist gleichschnellen Code wie Java zu produzieren, da beide Sprachen statische Typeninformation haben. Ausserdem ist Scala Java sehr ähnlich und Scala ist eine der ältesten und reifsten JVM-Sprachen.

Die Erwartung an Clojure sind nicht ganz so hoch da das Clojure-Projekt relativ neu ist. Clojure ist ausserdem eine dynamische Programmiersprache, diese sind schwerer zu optimieren.

4 Material und Methoden

4.1 Allgemeines

4.1.1 Sprachauswahl

In den letzten Jahren ist ein neuer Trend entstanden. Programmiersprachen werden oft nicht mehr von Grund auf neu aufgebaut sondern versuchen bestehende Infrastrukturen zu verwenden.
Dies bringt sowohl Vor- als auch Nachteile mit sich. Hauptvorteil ist, dass man sich als Trittbrettfahrer eine VM zunutze machen kann. In moderne VMs, wie der JVM, wurden hunderte von Mannjahren investiert. Sie sind daher sehr stabil und auf vielen Computern bereits installiert.

Der zweite Vorteil ist die Interoperabilität zwischen den verschiedenen Sprachen. Dadurch kann Code wiederverwendet werden ohne dass viel Zeit in die Portierung von Funktionen investiert werden muss welche jede Sprache braucht. Dies können beispielsweise Datenbanken, Protokolle oder XML Parser sein.

In der Java Programmiersprache sind all diese grundlegenden Funktionen bereits implementiert. Die JVM ist daher eine geeignete Plattform um andere Sprachen darauf aufzusetzen.

Die entscheidene Frage ist nun weshalb die Nutzer die neuen Sprachen benützen sollten. Dafür gibt es viele Gründe. Einige neue Sprachen (Scala) bieten bessere Typsysteme und dadurch mehr Compiletime Sicherheiten. Andere Sprachen, wie z.B. Groovy, versuchen Features von sehr dynamischen Sprachen zu unterstützen.

Alle diese neuen und spannenden Features sind fantastisch, gehen aber oft zu Lasten der Performance. Das kommt teilweise davon, dass manche Features (z.B. Bouncechecking) bereits per Definition performancelastig sind, oft stellt jedoch auch der Unterschied in den Semantiken der Sprachen ein Problem dar. D.h. wenn in einer Sprache ein Feature unterstützt wird in der Host-VM jedoch nicht, muss um das Problem herum gearbeitet werden was zusätzlichen Runtime-Overhead verursacht. In dieser Arbeit haben wir uns zwei der meist verwendeten neuen JVM-Sprachen ausgesucht um herauszufinden ob diese es schaffen die Geschwindikeit der Hostsprache (Java) zu erreichen.

4.1.2 Zielanpassung: Sprachanpassung

Gemäss Exposé wollten wir die drei JVM-Sprachen Clojure, Scala und JRuby Benchmarken. Während der Arbeit haben wir festgestellt, dass JRuby andere Ziele vorfolgt

als maximale Performance zu bieten und es 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 weitere neue JVM-Sprache wählen können (es gibt mehr als einhundert), es gibt jedoch kaum andere Sprachen die weit genug entwickelt sind oder die ähnliche Ziele verfolgen. Deshalb haben wir uns Entschieden zu testen 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 die Performance sehr wichtig ist. Viele Sprachen lassen sich extrem "verbiegen"wodurch sich, auf Kosten von Einfachheit und Klarheit, die Geschwindkeit verbessern lässt.

Wir verfügen nicht über ausreichen Fachwissen um solche Programme zu schreiben und sich dieses anzueignen erfordert eine monatelagen Einarbeitungszeit. Deshalb beschränken wir uns, bei der Implementierung, auf eine Sprache. Für die anderen Tests benutzen wir Referenz-Implementierungen.

4.2 Grundlagen

4.2.1 Virtuelle Maschinen

Um es simpel zu halten haben wir uns Entschieden Programmiersprachen zu verwenden, die auf der JVM (oder anderen VMs die Java Byte Code ausführen) laufen. Dies erlaubt uns Aussagen über die Codegenerierung des Source-to-Bytecode-Compilers der jeweiligen Sprachen machen.
Da VMs einen Bytecode als Input erhalten ist es grundsätzlich möglich jede 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 ausführt:

- 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 kompiliert nicht alles auf einmal sondern, nur den Code der auch wirklich gebraucht wird. Das erlaubt dem Compiler Optimierungen 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 Optimierungen vorzunehmen die speziell für diese Hardware den Code anpassen.
Gute JIT-Compiler sind heutzutage geschwindigkeitsmässig vergleichbar den Native Code Compilern. Die Unterschiede sind in vielen Anwendungsfällen nur noch gering.

4.3 Das Testsystem

4.3.1 Leistungsdaten

Beim verwendeten Testsystem handelt es sich um einen HP Compaq 6710b Notebook mit den folgenden Leistungsdaten.

- Prozessor: Intel Core 2 Duo T7700 @ 2.4 GHz
- Arbeitsspeicher: 2 GB

Wir haben uns entschieden als Betriebssystem die aktuellste Version der Linux-Distribution Debian zu verwenden. Debian hat die Vorteile, dass es sehr stabil läuft und einfacher zu bedienen ist, als andere Linux-Distributionen. Wir haben uns aus verschiedenen Gründen dafür entschieden unsere Tests Linux-basierend durchzuführen. So ist unter Linux keine Lizenzierung nötig, da das Betriebssystem und der grösste Teil der Programme Open-Source sind oder wir haben durch vorgängige Recherchen festgestellt, dass die Auswahl an freien Performance-Monitoring-Tools in der Linux-Welt viel grösser ist. Die genannten Leistungsdaten sind für die Nachvollziehbarkeit der Messdaten sehr wichtig. Werden die Tests auf einem anderen System wiederholt, kann nicht sichergestellt werden, dass die Resultate identisch sind. Des Weiteren ist zu erwähnen, dass obwohl der installierte Prozessor über zwei Cores verfügt, die Testprogramme nur darauf ausgelegt sind, einen zu nutzen. Eine Lastverteilung durch das Betriebssystem findet jedoch trotzdem statt. Die Hauptgründe dafür, dass nur ein Kern genutzt wird, sind die folgenden:

- Der Programmieraufwand ist kleiner, wenn nur ein Kern genutzt wird
- Eine Parallelisierung lohnt sich bei zwei Kernen vielfach nicht da der zusätzliche Rechenaufwand grösser ist als der Geschwindigkeitsgewinn.

4.3.2 Swap-Space

Der Swap-Space ist ein virtueller Speicher welcher vom Betriebssystem genutzt wird, wenn der eingebaute Arbeitsspeicher nicht ausreicht. In der MS Windows-Welt ist der

Swap-Space unter der Bezeichnung „Auslagerungsdatei“ bekannt. Der Swap-Space befindet sich auf der Festplatte und ist daher um ca. Faktor 1000 langsamer als der Arbeitsspeicher. Trotzdem ist es wichtig, dass das Betriebssystem notfalls in der Lage ist auf einen Swap zurückgreifen zu können. Ansonsten kann dies zu abstürzen von einzelnen Prozessen oder gar dem ganzen System führen. Da gerade die JVM relativ speicherintensiv ist, wird dies umso wichtiger.

Ein viel diskutiertes Thema wenn es um den Swap-Space geht, ist dessen Grösse. Grundsätzlich wird empfohlen den Swap-Space mindestens gleich gross wie den Arbeitsspeicher, aber höchstens doppelt so gross zu machen. Ich habe mich daher an die Faustregel Swap-Space = 1.5 * Arbeitsspeicher gehalten. Nach oben sind theoretisch übrigens keine Grenzen gesetzt, jedoch ist es fraglich ob ein System jemals einen noch grösseren Swap-Space auslasten wird. Es ist daher reine Speicherverschwendung grössere Bereiche auf der Festplatte für Swap zu reservieren.

4.4 Aufnahme der Daten

Nach gründlichen Recherchen und dem Testen diverser Monitoring-Tools kamen wir leider zu Schluss, dass kein Programm unsere Anforderungen ausreichend erfüllen konnte. Das Hauptproblem war meist, dass die Datenaufnahme nicht automatisch gestartet werden konnte. Die Aufzeichnung der Daten manuell zu starten, kam für uns jedoch nicht in Frage, da bereits eine Sekunde Verzögerung zwischen Programmstart und Start der Datenaufnahme den Verlust einer Grossen Datenmenge zur Folge hätte. Des Weiteren boten einige Programme keine oder nur beschränkte Möglichkeiten um die gesammelten Daten zu speichern.

Als Alternative haben wir uns für ein selbstgeschriebenes Script entschieden. Linux bringt selbst bereits eine Vielzahl an Funktionen mit um Leistungsdaten auszulesen. Bereits nach kurzer Nachforschung zeigte sich, dass eine Automatisierung des Auslesevorgangs mittels Script relativ einfach zu realisieren ist.

4.4.1 Das Script

Das folgende Script liest die benötigten Leistungsdaten aus. Ich erkläre die einzelnen Bereiche des Scripts nacheinander um Verwirrung zu vermeiden.

Die erste Zeile informiert Linux darüber, dass es sich um ein Shell-Script handelt, damit das Betriebssystem den Code richtig interpretieren kann. Bei einem Perl Script müsste hier beispielsweise das ‚sh‘ durch ein ‚perl‘ ersetzt werden.

Listing 1: Shebang

```
#!/bin/sh
```

Die als Parameter mitgegebenen Parameter werden in Variablen gespeichert.

Listing 2: Parameter

```
lang=$1
code=$2
param=$3
```

Nun folgt die Fehlerbehandlung. Die erste Zeile besagt, dass wenn im Script ein Fehler auftritt, welcher nicht zuvor durch eine andere Funktion abgefangen wird, die Funktion „error“ ausgeführt werden soll. Dabei ist es egal wodurch der Fehler verursacht wird oder an welcher Stelle er auftritt.

Die Error-Funktion gibt ihrerseits eine Meldung aus, die den Benutzer über das Auftreten des Fehlers aufklärt und evtl. mögliche Ursachen dafür nennt. Anschliessend wird das Script beendet.

Listing 3: Fehlerbehandlung

```
trap "error" ERR
#
function error() {
  echo '
##########
Ein unbekannter Fehler ist aufgetreten. Prüfen Sie die folgenden mö
   glichen Fehlerquellen:
- ungültige Java Parameter
- fehlerhafte Programmdatei (*.jar)
##########'
  exit 0
}
```

Die folgenden beiden Vergleichsoperationen (IF-Konstrukte) dienen ebenfalls der Fehlerbehandlung. Die erste Funktion prüft ob die angegebene Sprache zulässig ist (wenn die Sprache im Parameter $lang nicht Clojure ist und nicht Java ist und nicht Scala ist, dann führe den folgenden Code aus). Wenn der Vergleich wahr ist, d.h. eine falsche Sprache angegeben wurde, wird eine entsprechende Meldung ausgegeben und das Script beendet.

Listing 4: Sprachparameter prüfen

```
if [ $lang != clojure -a $lang != java -a $lang != scala ] ; then
  echo 'Enter clojure, java or scala for Parameter 1!'
  exit 0
fi
```

Die zweite Vergleichsoperation überprüft, ob die angegebene auszuführende Programmdatei überhaupt auffindbar ist (wenn die Datei im Parameter $code nicht vorhanden ist, dann führe den folgenden Code aus). Sollte das Script keine Datei finden können, wird der Benutzer informiert und das Script beendet.

Listing 5: Jar-Datei prüfen

```
if [ ! -f $code ] ; then
  echo 'File "'$code'" not found!'
  exit 0
fi
```

Der nun folgende Abschnitt überprüft ob bereits Dateien mit Messdaten vorhanden sind. Sollte das der Fall sein fragt das Script beim Benutzer nach, ob die bestehenden Dateien überschrieben werden sollen oder die neuen Messdaten in den bestehenden Dateien, unterhalb der bereits vorhandenen Daten, angehängt werden sollen.

Die for-Schleife wird hier benötigt, weil sich die im Ablageverzeichnis für die Messdaten möglicherweise mehrere Dateien befinden, welche mit dem Parameter $lang, d.h. der verwendeten Programmiersprache beginnen. Da die Vergleichsoperation, welche prüft ob entsprechende Dateien vorhanden sind, nur einen Parameter erlaubt, würde das Script in diesem Fall abstürzen. Mit der Schleife wird dem Vergleicher jeweils nur eine einzelne, zu prüfende Datei übergeben.

Falls bereits Dateien vorhanden sind, wird der Benutzer nach dem weiteren Vorgehen gefragt und die Antwort in eine Variable eingelesen. Eine weitere Vergleichsoperation prüft nun ob der Benutzer ein kleines oder grosses Ypsilon (für Yes) eingegeben hat. Ist das der Fall, werden mit dem rm-Befehl die vorhandenen Dateien gelöscht, der Benutzer wird informiert und die Schleife abgebrochen. Hat der Benutzer jedoch etwas anderes oder gar nichts eingegeben, teilt das Script ihm mit, dass es die bestehenden Dateien erneut verwenden wird und bricht ebenfalls die Schleife ab. Es ist hier notwendig die Schleife abzubrechen, da diese die vorhandenen Dateien hochzählt. Befinden sich also mehrere Dateien im Verzeichnis, würde der Benutzer pro Datei einmal nach dem weiteren Vorgehen abgefragt.

Listing 6: Benutzerabfrage

```
for tfile in /opt/data/$lang.* ] ; do
  if [ -f $tfile ] ; then
    echo '
Bestehende Dateien überschreiben? Geben Sie "Y" ein um die bestehenden
   Datei zu löschen oder drücken Sie Enter um neuen Messdaten in die
   bestehenden Dateien zu schreiben!'
    read answer
    if [ "$answer" = y -o "$answer" = Y ] ; then
      rm /opt/data/$lang.*
      echo 'Dateien wurden gelöscht!'
      break
    else
      echo 'Daten werden in bestehende Dateien geschrieben!'
      break
    fi
  fi
done
```

In die drei Dateien, in welche die gesammelten Daten gespeichert werden, wird

das aktuelle Datum sowie die Uhrzeit geschrieben. Somit ist in den Dateien ersichtlich, von wann genau die Messdaten stammen.

Listing 7: Titel einfügen

```
echo "#
####$(date +%d.%m.%Y" "%T)####
#" | tee -a /opt/data/$lang.memory /opt/data/$lang.cpu /opt/data/$lang.
    time > /dev/null
```

In einem ersten Schritt werden die Daten zum Arbeitsspeicher gesammelt. Nachdem der Benutzer informiert wurde, wird dazu das zu überwachende Programm gestartet und gleichzeitig (wird mit dem &-Zeichen festgelegt) eine Schleife, welche die Daten ausliest. Beim Programmaufruf ist ersichtlich, dass der Parameter $param mitgegeben wird. Dieser enthält evtl. weitere, notwendige Parameter für die JVM um die Programmausführung zu optimieren. Ohne das &-Zeichen nach dem Programmaufruf, würden die beiden Befehle nicht gleichzeitig ausgeführt. Somit würden die Daten erst ausgelesen, wenn das Programm bereits durchgelaufen ist.

Die Schleife prüft als Bedingung ob die Ausgabe von ps $! Grösser als /dev/null ist. $! Ist eine Variable, in der die Prozess-ID (PID) des zuletzt gestarteten Programms abgelegt ist. PS ist ein Linux-Befehl um Prozessinformationen anzuzeigen. Bei dem Verzeichnis /dev/null handelt es sich um das bekannte schwarze Loch in Linux. Es steht für das Nichts, die Leere. So lange also die Ausgabe von Prozessinformationen zum zuletzt gestarteten Prozess grösser als nichts ist (d.h. der Prozess ist noch aktiv), wird die Schleife ausgeführt.

Der sed-Befehl kopiert die Zeilen 11 bis 20 aus der Datei /proc/$!/status in die Datei $lang.memory. Im Proc-Dateisystem werden die aktuellen System- und Prozessinformationen des Linux-System gespeichert. Auch hier steht $! wieder für die PID des zuletzt gestarteten Prozesses. Anschliessend wird mit dem echo-Befehl eine Trennzeile in die gleiche Datei geschrieben und das Script wird für die Dauer von 0.2 Sekunden pausiert. An dieser Stelle kann somit festgelegt werden, in welchem Intervall die Daten ausgelesen werden sollen. Da unsere Programme nur eine sehr kurze Ausführungszeit haben, reicht eine Wartezeit von 0.2 Sekunden aus. Ohne Wartezeit wäre die Datenmenge zudem extrem hoch.

Listing 8: Speicherdaten sammeln

```
echo '
Collecting Memory Data...
'
java -server $param -jar $code &
while ps $! > /dev/null
do
  sed -n '11,20p' /proc/$!/status >> /opt/data/$lang.memory
  echo '-----------------------' >> /opt/data/$lang.memory
  sleep .2
done
echo '
done'
```

Der zweite Schritt ist das Auslesen der Prozessor-Daten. Dazu wird eine temporäre Datei benötigt. Deren Dateiname wird mit der date-Funktion generiert. Diese schreibt das aktuelle Datum sowie die genaue Uhrzeit in eine Variable. Der Dateiname ist somit einzigartig.

Anschliessend erfolgt der gleiche Programmaufruf wie er bereits für das Auslesen der Speicherdaten benutzt wurde. Die CPU-Daten können leider nicht aus dem proc-Dateisystem ausgelesen werden. Sie werden mit dem ps-Befehl gesammelt und in temporäre Datei geschrieben.

Mit dem ps-Befehl werden jedes Mal zu den Prozessordaten auch die Spaltenüberschriften in die Datei geschrieben. Für eine bessere Darstellung werden diese mit dem sed-Befehl entfernt und die neue Ausgabe in die Datei $lang.cpu geschrieben. Der sed-Befehl entfernt ab der dritten Zeile jede zweite Zeile in der temporären Datei. Anschliessend wird die temporäre Datei gelöscht.

Listing 9: CPU-Daten sammeln

```
echo '
Collecting CPU Data...
'
random=tmp$(date +%d%m%Y_%H%M%S)
java -server $param -jar $code &
while ps $! > /dev/null
do
  ps -p $! -o user,pid,%cpu,time >> /opt/data/$random
  sleep .2
done
sed '3~2d' /opt/data/$random >> /opt/data/$lang.cpu
rm /opt/data/\random
echo '
done'
```

Der letzte Schritt ist nun das Auslesen von Zeitdaten (Wie lange dauert die Programmausführung? Wie viel Zeit davon hat die CPU im Kernel-, wie viel im User-Modus verbracht?). Für diese Daten wird keine Schleife benötigt. Die Daten können mit nur einer Befehlszeile ausgelesen werden. Der Parameter –f definiert das Format

der Ausgabe (Was soll ausgegeben und wie sollen die Ausgaben beschriftet werden?), der Parameter –o definiert die Ausgabedatei (Output) und der Parameter –a sorgt dafür, dass die Ausgabedatei nicht überschrieben wird sondern die neuen Daten einfach angehängt (append) werden. Als letzten Parameter erwartet der time-Befehl das Programm welches ausgeführt werden soll.

Listing 10: Zeitdaten sammeln

```
echo '
Collecting Time Data...
'
/usr/bin/time -f "Elapsed real time: %E\nCPU usage: %P\nTotal CPU-
    seconds in user mode: %U\nTotal CPU-seconds in kernel mode: %S\nName
    and arguments of the command: %C" -o /opt/data/$lang.time -a java -
    server $param -jar $code
echo '
done'
```

4.5 Datenauswertungsmethoden

Um die Methoden für die Datenauswertung zu bestimmen ist es vorerst wichtig zu wissen was die gesammelten Daten genau bedeuten um sie später korrekt auswerten zu können. Ich habe in den folgenden drei Unterkapiteln versucht dies möglichst verständlich darzustellen.

4.5.1 Memory

Der Memory Bereich ist der wohl komplexeste von allen. Der Begriff des virtuellen Speichers ist hier immer wieder anzutreffen. Deshalb folgt hier eine kurze, vereinfachte Erklärung.

Virtual Memory: Einem Prozess wird ein Adressbereich simuliert den er alleine für sich benutzen kann. Somit hat der Prozess den Eindruck, über einen zusammenhängenden sTeil des Hauptspeichers zu verfügen. In Wirklichkeit verteilt das Betriebssystem den vom Prozess genutzten Speicher über den ganzen Arbeitsspeicher und teilweise auch den Swap-Space (siehe Bild).
In der Folge eine Beispielsausgabe aus unserem Messskript.

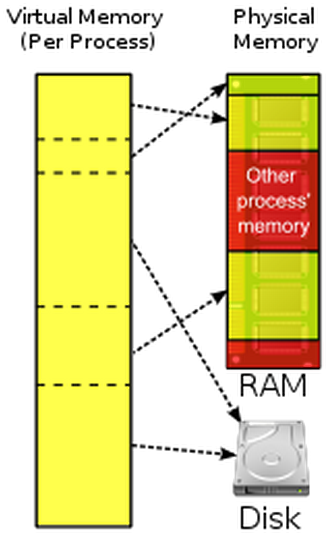

Abbildung 1: Wie virtueller Speicher funktioniert. Quelle: en.wikipedia.org, Virtual memory

Listing 11: Speicherdaten

```
----------------------
VmPeak:  1086096 kB
VmSize:  1036004 kB
VmLck:         0 kB
VmHWM:     15304 kB
VmRSS:     15304 kB
VmData:   988080 kB
VmStk:       220 kB
VmExe:        32 kB
VmLib:     10960 kB
VmPTE:       180 kB
----------------------
```

Wert	**Erklärung**
VmPeak	höchster Verbrauch von virtuellem Speicher den der Prozess seit dem Start jemals hatte
VmSize	aktueller Verbrauch von virtuellem Speicher. Darin ist auch VmData eingerechnet wodurch dieser Wert für die Messungen nicht weiter relevant ist (siehe VmData)
VmLck	Bei der virtuellen Speicherverwaltung wird der virtuelle Adressraum in Pages aufgeteilt. Aufgrund eines Fehlers kann es sein, dass eine solche Page gesperrt (locked) wird. Die Grösse des gesperrten Speichers würde, wenn vorhanden, hier angegeben.
VmHWM	HWM = high water mark, höchster Verbrauch von physikalischem Speicher den der Prozess seit dem Start jemals hatte
VmRSS	RSS = resident set size, aktueller Verbrauch von physikalischem Speicher
VmData	Daten auf denen das Programm ausgeführt wird. Sprich die JVM. Dieser Wert enthält auch Daten welche noch nicht in den Arbeitsspeicher geladen wurden weil sie nicht benötigt werden. Da für unsere Messungen nur die Daten im physikalischen Arbeitsspeicher interessant sind, ist dieser Wert für unsere Messungen daher irrelevant.
VmStk	Benötigte Ressourcen für die Ausführung. Dieser Wert ist eingerechnet in VmRSS.
VmExe	Der ausgeführte Programmcode, bzw. die als ausführbar markierten Pages. Dieser Wert ist eingerechnet in VmRSS.
VmLib	Shared Memory, Speicherbereich auf den auch andere Prozesse zugreifen können
VmPTE	Zwischen dem virtuellen und dem physikalischen Speicher befindet sich eine sog. Page Table welche den virtuellen Speicher dem physikalischen zuordnet (z.B. virt. Adresse 0x01 befindet sich bei 0x0A im Arbeitsspeicher usw.). Die Grösse der Page Table ist in hier angegeben.

Tabelle 1: Erklärung der gemessenen Speicherwerte

Zusammenfassend lässt sich sagen, dass für unsere Messungen lediglich der Wert VmRSS, welcher den tatsächlichen, physikalischen Speicherverbrauch abbildet, relevant ist. Dies weil die anderen Messwerte insofern verfälscht sind, dass sie teilweise gar nie in den Speicher geladen werden (VmPeak, VmSize, VmData) oder weil es für unsere Messungen nicht relevant ist wie gross die einzelnen Datensegmente (VmStk, VmExe) im Speicher sind.

4.5.2 CPU

In der Folge eine Beispielausgabe für die Messung der CPU-Daten mit unserem Script. Neben dem ausführenden Benutzer unter der Prozess-ID sind hier auch die momentane CPU-Auslastung und die vergangene Ausführzeit ersichtlich.

Listing 12: Speicherdaten

```
#
####26.02.2011 13:59:13####
#
USER       PID %CPU     TIME
root      7730  0.0 00:00:00
root      7730  0.0 00:00:00
root      7730 43.0 00:00:00
root      7730 65.0 00:00:00
root      7730 86.0 00:00:00
root      7730  112 00:00:01
root      7730  152 00:00:01
```

Bei Betrachtung dieser Daten fällt auf den ersten Blick auf, dass die CPU-Auslastung 100% teilweise übersteigt. Da es sich beim Testsystem um ein Dual-Core System handelt, kann die CPU-Auslastung bis zu 200% erreichen wenn beide Kerne zu 100% ausgelastet sind. Aufgrund der heute von beinahe allen Betriebssystemen verwendeten SMP-Systemarchitektur, verwenden die Prozessoren unseres Testsystems einen gemeinsamen Adressraum um Aufgaben dynamisch verteilen zu können. Dies führt jedoch auch zu diesen, auf den ersten Blick verwirrenden, Messdaten. Da die gemessenen Daten aufgrund dessen jedoch keineswegs falsch sind, werden wir diese trotzdem für unsere Benchmarkes verwenden.

Die Spalte TIME gibt auf die Sekunde genau an, wie lange der Prozess bereits aktiv ist. Da wir die aktuellen Daten alle 0.2 Sekunden messen, ergeben sich fünf Zeilen pro Sekunde.

4.5.3 Time

Hier eine Beispielausgabe für die Messung der Zeit-Daten mit unserem Script. Erneut tauchen hier evtl. unbekannte Begriffe welche in der Folge kurz erklärt werden.

Listing 13: Speicherdaten

```
#
####26.02.2011 13:59:13####
#
Elapsed real time: 0:02.30
CPU usage: 99%
Total CPU-seconds in user mode: 2.38
Total CPU-seconds in kernel mode: 0.06
Name and arguments of the command: java -server -jar clojure/clojure.jar
```

CPU-Modi: Man unterscheidet zwischen zwei verschiedenen Betriebsmodi in denen

die CPU Prozesse ausführt, es sind dies der Kernel- und der User-Modus.

Kernel Mode: Der ausgeführte Code hat unbeschränkten Zugang zur Hardware, kann jede Anweisung ausführen und jede Speicheradresse adressieren. Der Kernel Mode ist grundsätzlich für vertrauenswürdige Funktionen des Betriebssystems reserviert. Abstürze in diesem Mode können für das System verheerende Folgen haben.

User Mode: Der ausgeführte Code hat keine Möglichkeit direkt auf die Hardware zuzugreifen. Sollten solche Zugriffe nötig sein, müssen sie über Schnittstellen des Betriebssystems erfolgen. Durch die Einschränkungen entstehen bei Abstürzen keine Schäden am System.

CPU-seconds: Zeit in der die CPU tatsächlich mit der Ausführung des Prozesses beschäftigt war. Da durch das Betriebssystem immer auch andere Prozesse aktiv sind, dauert die Programmausführung in der Regel länger als die Anzahl CPU-Seconds die tatsächlich dafür aufgewendet wurden (CPU muss priorisieren). Aufgrund der Multicore Technologie ist es aber durchaus auch möglich, dass für eine Programmausführung mehr CPU-Seconds benötigt werden, als tatsächlich Zeit vergeht da die CPU-Seconds pro Kern gezählt werden.

Wert	**Erklärung**
Elapsed real time	Zeit die für die Programmausführung benötigt wurde
CPU usage	Durchschnittliche CPU-Auslastung während der Programmausführung
Total CPU-seconds in user mode	Im User-Mode verbrachte Zeit
Total CPU-seconds in kernel mode	Im Kernel-Mode verbrachte Zeit
Name and arguments of the command	Ausgeführtes Programm inkl. aller Parameter

Tabelle 2: Erklärung der gemessenen Zeitwerte

4.5.4 Methoden

Für die Auswertung der Daten gehen wir wie folgt vor.

- Memory
  - Vergleich des Verlaufes von VmRSS zwischen allen Programmiersprachen

- CPU
  - Vergleich des Verlaufes der CPU-Auslastung aller Programmiersprachen
  - Vergleich der durchschnittlichen CPU-Auslastung während der Ausführung
- Zeit
  - Vergleich der gesamten Ausführungszeit
  - Vergleich der benötigten CPU-seconds (Kernel- und User-Mode)

Alle Messresultate werden sowohl in tabellarischer wie auch in grafischer Form vorgelegt.

5 Programmiersprachen und Implementierungen

5.1 Java

5.1.1 Beschreibung

Java ist eine Programmiersprache die ab 1992 von Sun Microsystems (oder teilweise im Auftrag von Sun) entwickelt wurde. Java wurde zuerst fr eingebettete Systeme geschrieben. Heutzutage findet man Java Applikation in vielen Anwendungsgebieten. Java ist eine der meist verwendeten Programmiersprachen und wird auch in vielen Universitäten gelehrt.

5.2 Scala

5.2.1 Beschreibung

Die Entwicklung von Scala hat 2001 an der ETH Lausanne unter der Leitung von Martin Odersky begonnen. Die Idee war eine Sprache zu designen, welche die Konzepte von funktionalen und objektorientierten Sprachen in Synthese verwendet. Zwar wurde Scala im akademischen Kontext entwickelt, findet aber auch immer mehr Anwendungen im Business Bereich.

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 eine gute Integration ins Host System (JVM). Diese erlaubt die Wiederverwendung von Java Code, Java Ecosystems (Webservers, Profilers, Debuggers...) und natürlich der VM.

5.3.2 Code

6 Ergebnisse

Im diesem Kapitel werden die Messresultate der durchgeführten Messungen aufgezeigt. Hier ist zu beachten, dass es sich bei den Zeitangaben welche für die Memory- und CPU-Grafiken verwendet wurden, um Schätzungen handelt. Wir waren mit Hilfe unseres Scripts leider nicht in der Lage den genauen Zeitverlauf aufzuzeigen. Eventuelle Unregelmässigkeiten können daher nicht ausgeschlossen werden. Des weiteren beinhalten die in diesem Kapitel aufgezeigten Tabellen, aufgrund der grossen Datenmenge, nicht alle Messresultate. Die genaue Bedeutung der Werte wurde in den vorhergehenden Kapiteln bereits ausführlich behandelt, an dieser Stelle folgen daher keine Erklärungen mehr.

6.1 Java

6.1.1 Allgemeine Daten

Die folgende Tabelle beinhaltet einige allgemeine Messresultate zu Java.

Sprache	Java
Benötigte Zeit	22.28s
Ø Speicherauslastung	308557kB
Ø CPU-Auslastung	109%
Anz. CPU-Sekunden	24.31s
davon im User-Mode	23.74s
davon im Kernel-Mode	0.57s
davon im User-Mode	97.66%
davon im Kernel-Mode	2.34%

Tabelle 3: Erklärung der gemessenen Zeitwerte Java

6.1.2 Memory

Die nachfolgende Tabelle beinhaltet den Verlauf des physikalischen Speicherverbrauchs während der Ausführung des Java-Testprogrammes. Aufgrund der grossen Datenmenge haben wir uns auf einen Wert je zwei Sekunden Programmlaufzeit beschränkt. Die Grafik wurde unter Verwendung aller Messwerte erstellt, allfällige Spitzen sind darauf daher erkenntlich.

Laufzeit [s]	Speicherverbrauch [kB]
0	704
2	64888
4	398824
6	280544
8	287952
10	297992
12	314180
14	342300
16	394120
18	408484
20	419848
22	432032

Tabelle 4: Wertetabelle Speicherverbrauch Java

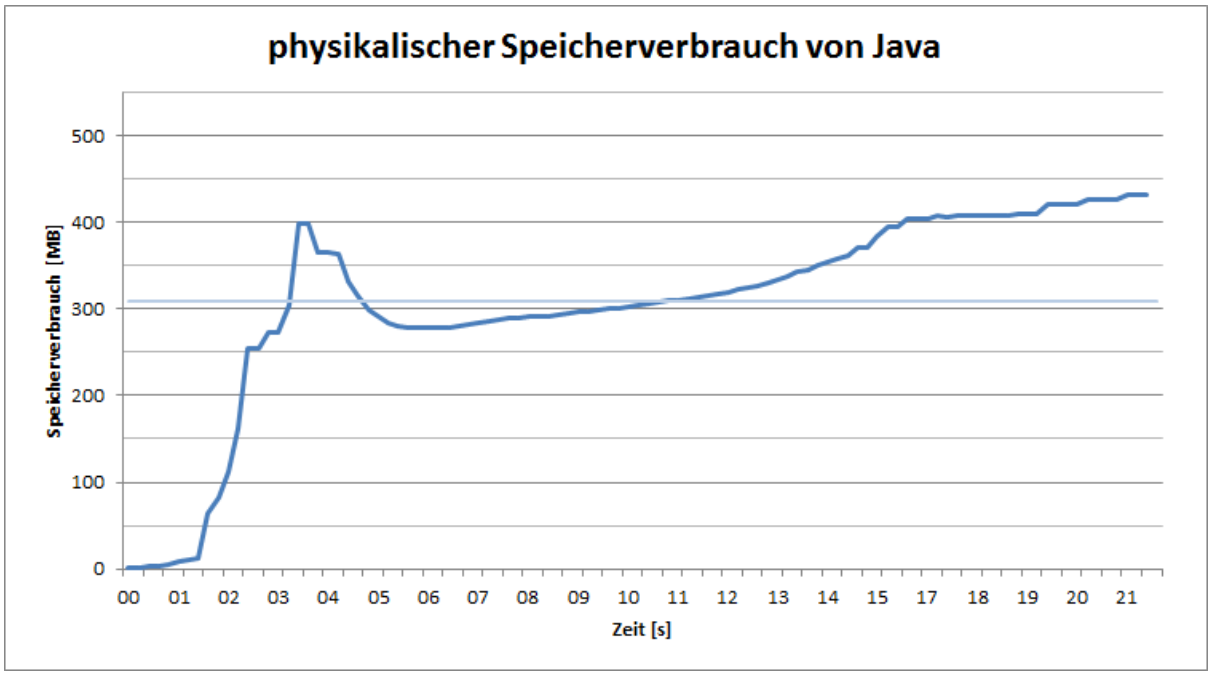

Abbildung 2: Verlauf der Speicherauslastung Java

6.1.3 CPU

Die nachfolgende Tabelle beinhaltet den Verlauf der CPU-Auslastung während der Ausführung des Java-Testprogrammes. Aufgrund der grossen Datenmenge haben wir uns auf einen Wert je zwei Sekunden Programmlaufzeit beschränkt. Die Grafik wurde unter Verwendung aller Messwerte erstellt, allfällige Spitzen sind darauf daher erkenntlich.

Laufzeit [s]	CPU-Auslastung [%]
0	0
2	103
4	131
6	116
8	99.2
10	100
12	99.3
14	100
16	100
18	100
20	106
22	106

Tabelle 5: Wertetabelle CPU-Auslastung Java

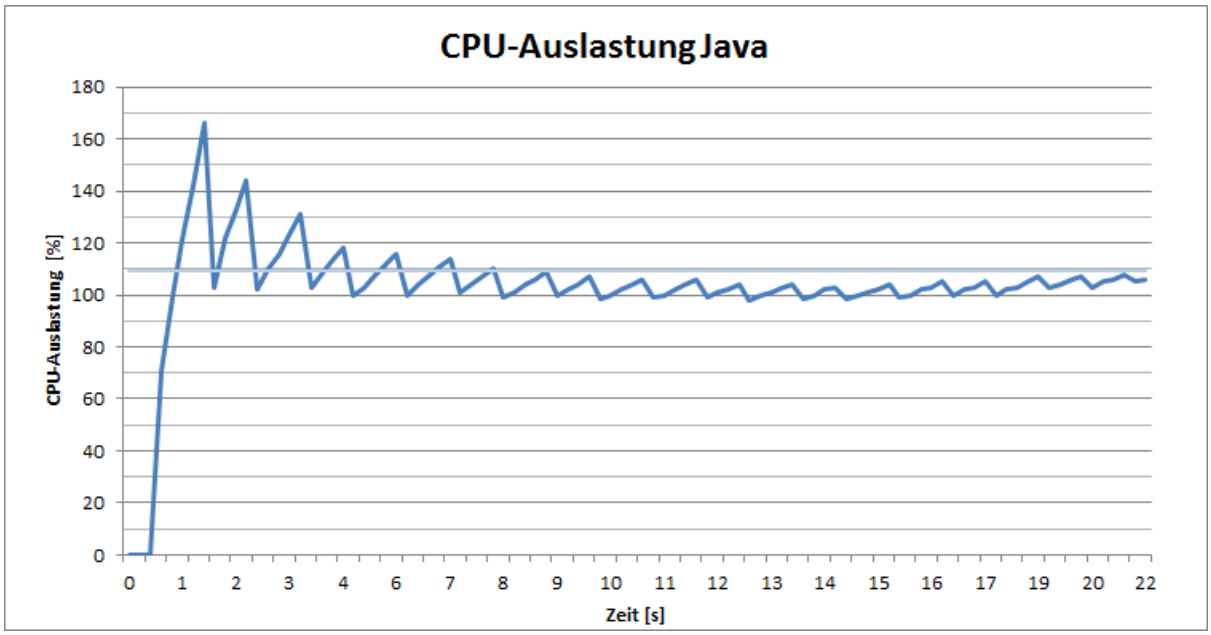

Abbildung 3: Verlauf CPU-Auslastung Java

6.2 Scala

6.2.1 Allgemeine Daten

Die folgende Tabelle beinhaltet einige allgemeine Messresultate zu Scala.

Sprache	Scala
Benötigte Zeit	22.37s
Ø Speicherauslastung	297488kB
Ø CPU-Auslastung	107%
Anz. CPU-Sekunden	24.03s
davon im User-Mode	23.43s
davon im Kernel-Mode	0.6s
davon im User-Mode	97.50%
davon im Kernel-Mode	2.50%

Tabelle 6: Erklärung der gemessenen Zeitwerte Scala

6.2.2 Memory

Die nachfolgende Tabelle beinhaltet den Verlauf des physikalischen Speicherverbrauchs während der Ausführung des Scala-Testprogrammes. Aufgrund der grossen Datenmenge haben wir uns auf einen Wert je zwei Sekunden Programmlaufzeit beschränkt. Die Grafik wurde unter Verwendung aller Messwerte erstellt, allfällige Spitzen sind darauf daher erkenntlich.

Laufzeit [s]	**Speicherverbrauch [kB]**
0	548
2	145256
4	294424
6	228976
8	233204
10	245484
12	262084
14	293188
16	349612
18	443108
20	451776
22	493828

Tabelle 7: Wertetabelle Speicherverbrauch Scala

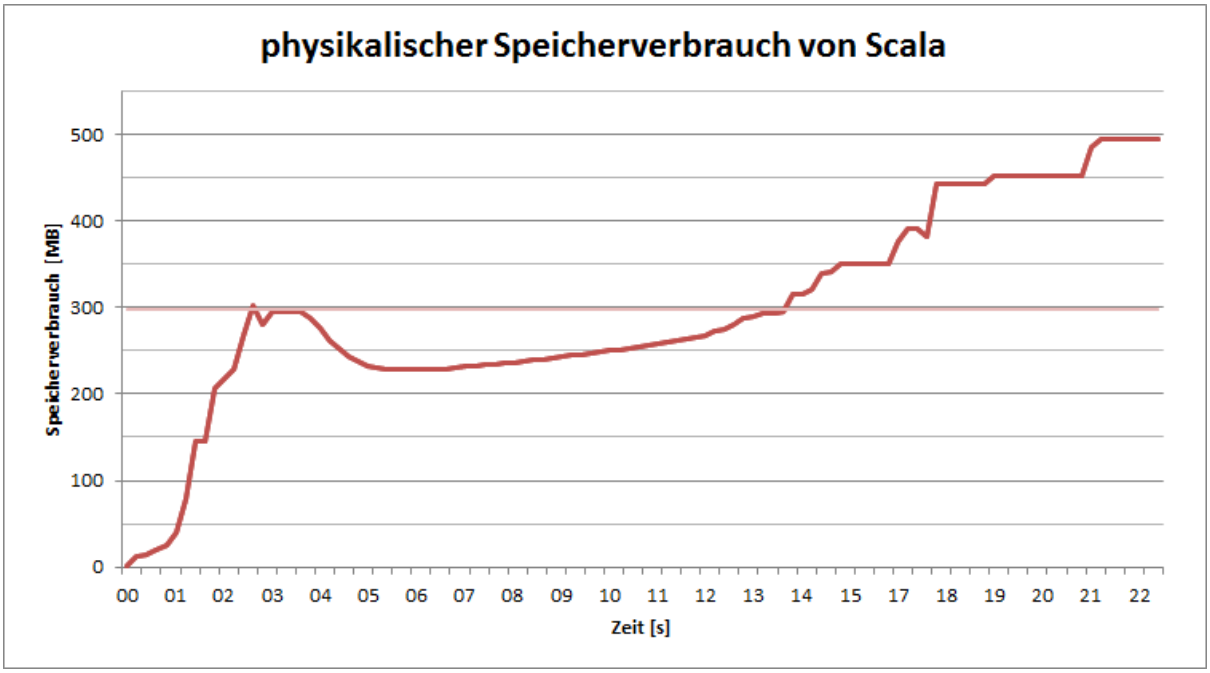

Abbildung 4: Verlauf der Speicherauslastung Scala

6.2.3 CPU

Die nachfolgende Tabelle beinhaltet den Verlauf der CPU-Auslastung während der Ausführung des Scala-Testprogrammes. Aufgrund der grossen Datenmenge haben wir uns auf einen Wert je zwei Sekunden Programmlaufzeit beschränkt. Die Grafik wurde unter Verwendung aller Messwerte erstellt, allfällige Spitzen sind darauf daher erkenntlich.

Laufzeit [s]	CPU-Auslastung [%]
0	0
2	105
4	103
6	100
8	101
10	100
12	100
14	100
16	100
18	99.3
20	100
22	105

Tabelle 8: Wertetabelle CPU-Auslastung Scala

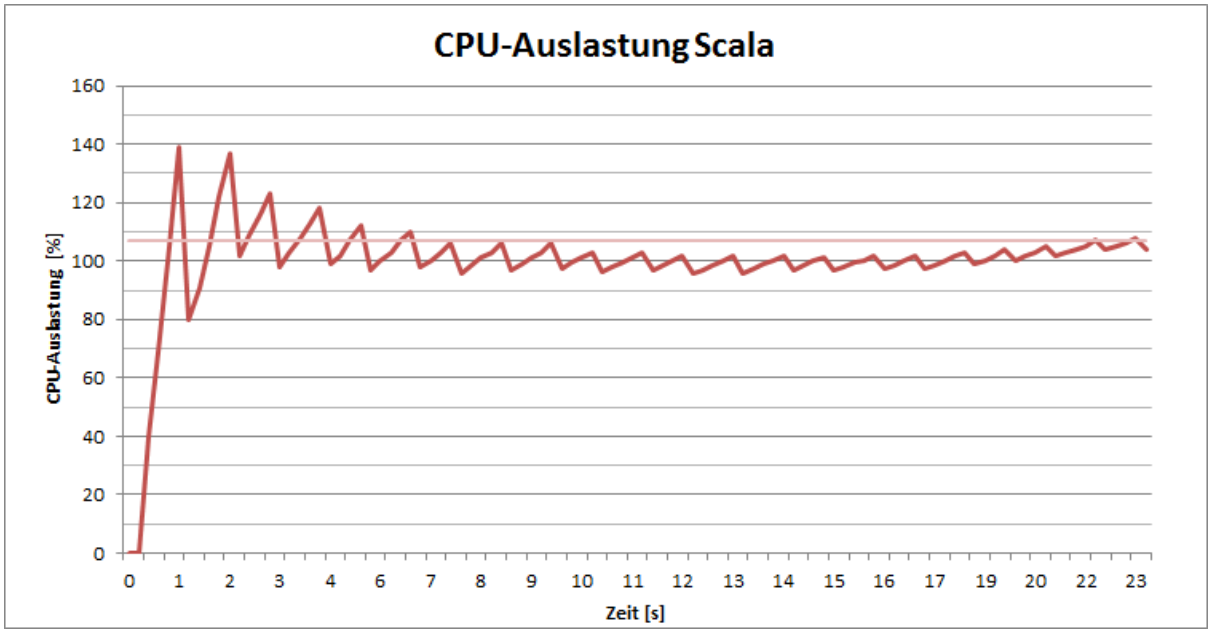

Abbildung 5: Verlauf CPU-Auslastung Scala

6.3 Clojure

6.3.1 Allgemeine Daten

Die folgende Tabelle beinhaltet einige allgemeine Messresultate zu Clojure.

Sprache	Clojure
Benötigte Zeit	62.88s
Ø Speicherauslastung	120264kB
Ø CPU-Auslastung	26%
Anz. CPU-Sekunden	73.32s
davon im User-Mode	72.32s
davon im Kernel-Mode	1.0s
davon im User-Mode	98.64%
davon im Kernel-Mode	1.36%

Tabelle 9: Erklärung der gemessenen Zeitwerte Clojure

6.3.2 Memory

Die nachfolgende Tabelle beinhaltet den Verlauf des physikalischen Speicherverbrauchs während der Ausführung des Clojure-Testprogrammes. Aufgrund der grossen Datenmenge haben wir uns auf einen Wert je fünf Sekunden Programmlaufzeit beschränkt. Die Grafik wurde unter Verwendung aller Messwerte erstellt, allfällige Spitzen sind darauf daher erkenntlich.

Laufzeit [s]	Speicherverbrauch [kB]
0	1300
5	64240
10	120264
15	112896
20	112896
25	112896
30	112896
35	112896
40	112896
45	112896
50	112896
55	112896
60	112896

Tabelle 10: Wertetabelle Speicherverbrauch Clojure

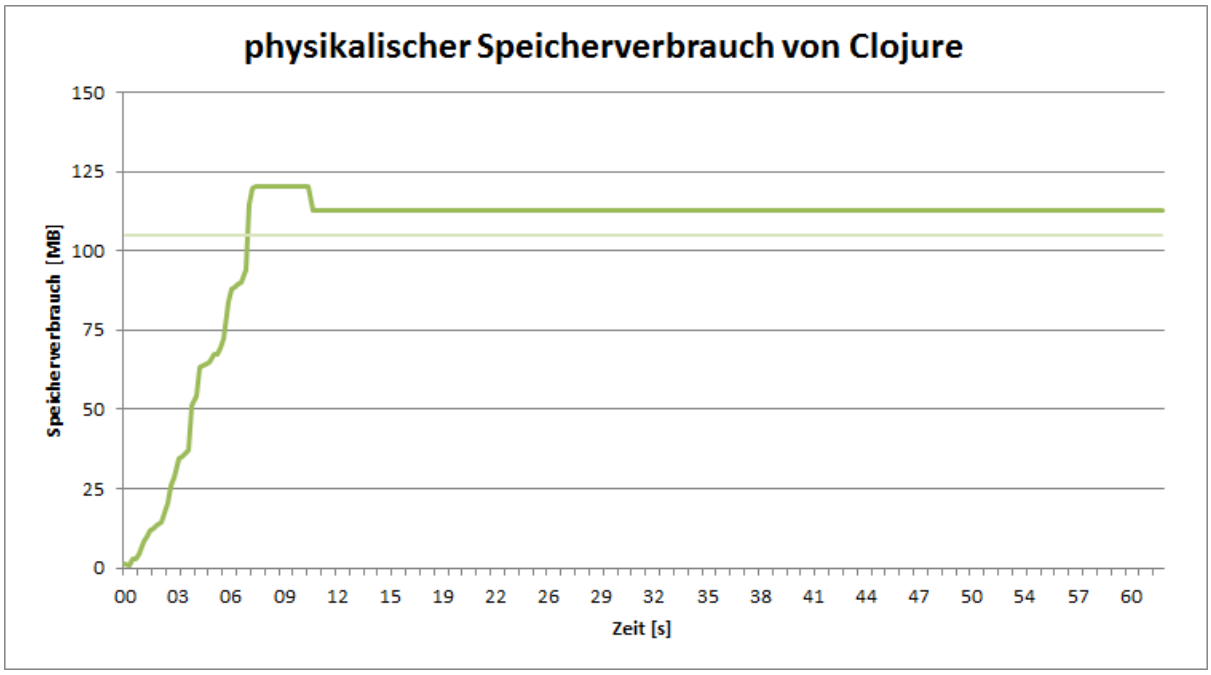

Abbildung 6: Verlauf der Speicherauslastung Clojure

6.3.3 CPU

Die nachfolgende Tabelle beinhaltet den Verlauf der CPU-Auslastung während der Ausführung des Clojure-Testprogrammes. Aufgrund der grossen Datenmenge haben wir uns auf einen Wert je fünf Sekunden Programmlaufzeit beschränkt. Die Grafik wurde unter Verwendung aller Messwerte erstellt, allfällige Spitzen sind darauf daher erkenntlich.

Laufzeit [s]	CPU-Auslastung [%]
0	0
5	92.4
10	42
15	28.8
20	23.2
25	19.3
30	16
35	13.6
40	11.9
45	10.5
50	9.4
55	8.6
60	7.9

Tabelle 11: Wertetabelle CPU-Auslastung Clojure

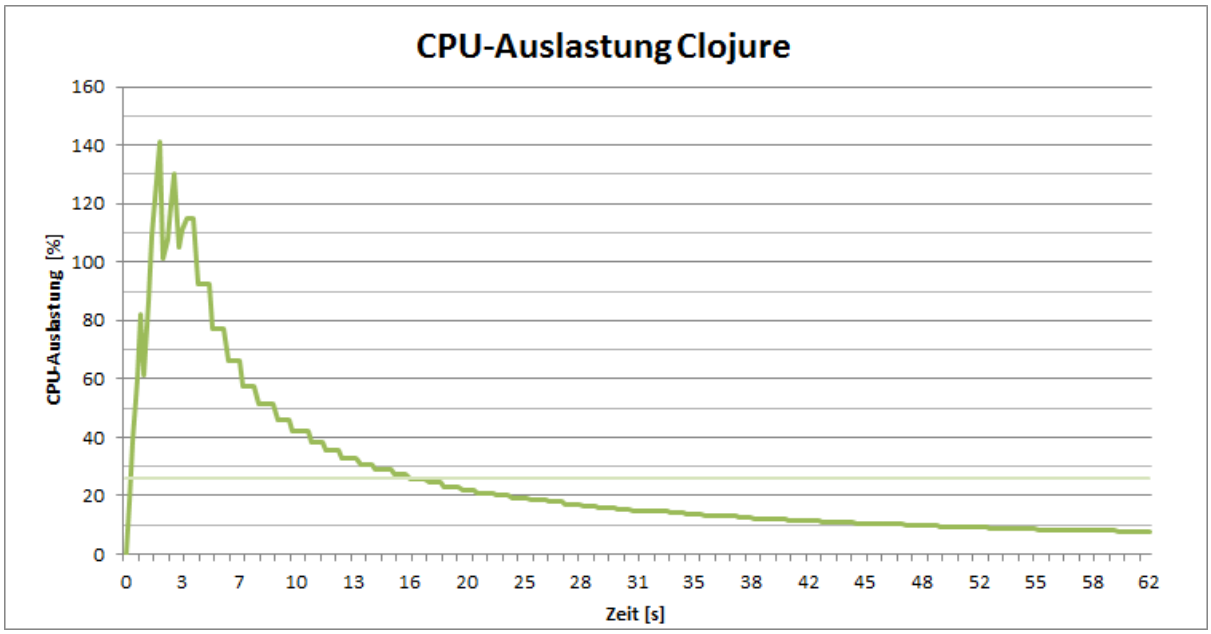

Abbildung 7: Verlauf CPU-Auslastung Clojure

7 Diskussion

7.1 Vergleich der Resultate

7.1.1 Zeitspezifische Daten

In diesem Unterkapitel vergleichen wir die jene Messdaten, welche die Zeit betreffen, d.h. die realen Ausführungszeiten und die benötigten CPU-Sekunden. Genau diese Werte wurden in Abbildung Nr. 8 grafisch dargestellt. Es ist deutlich ersichtlich wie Java und Scala sich ein Kopf-an-Kopf-Rennen liefern während Clojure mit einer ca. dreimal so hohen Ausführungszeit weit abgeschlagen ist. Das gleiche Bild zeigt sich bei den CPU-Sekunden, wo die Differenz noch grösser ist. Vergleicht man die Messungen von Java und Scala etwas genauer, so hat Scala bei beinaher gleicher Ausführungszeit etwas weniger CPU-Sekunden benötigt. Scala könnte daher als knapper 'Sieger' dieser Messung bezeichnet werden.

Sprache	Java	Scala	Clojure
Benötigte Zeit	22.28s	22.37s	62.88s
Anz. CPU-Sekunden	24.31s	24.03s	73.32s
davon im User-Mode	23.74s	23.42s	72.32s
davon im Kernel-Mode	0.57s	0.6s	1.0s
davon im User-Mode	97.66%	97.50%	98.64%
davon im Kernel-Mode	2.34%	2.50%	1.36%

Tabelle 12: Erklärung der gemessenen Zeitwerte Clojure

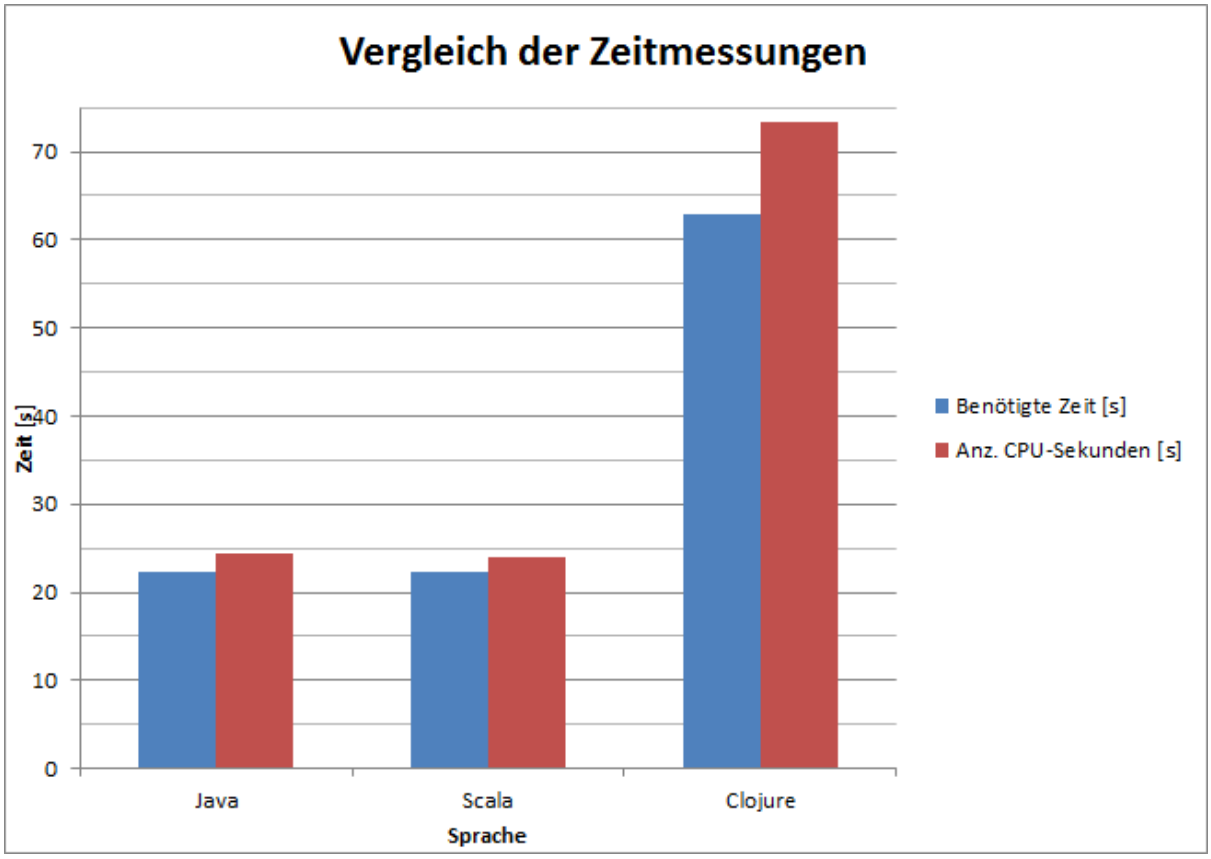

Abbildung 8: Vergleich der Zeitmessungen

Abbildung Nr. 9 zeigt den prozentualen Anteil an CPU-Sekunden auf, welcher im Kernel-Mode verbracht werden musste. Hier sieht das Bild schon wieder ganz anders aus. Clojure verbrachte rund ein Prozent weniger Ausführungszeit im Kernel-Mode als die anderen beiden Sprachen. Scala liegt mit einem Anteil von 2.5% auf dem letzten Platz. Es muss hier jedoch angemerkt werden, dass es unklar ist wann genau die CPU diese Zeit benötigt hat. Sollte sich dies auf den Programmstart beschränken, wird der Prozentsatz der CPU-Sekunden im Kernel-Mode bereits durch die lange Ausführungszeit des Clojure-Programms hinuntergezogen. Leider lässt sich dies nur sehr schlecht prüfen.

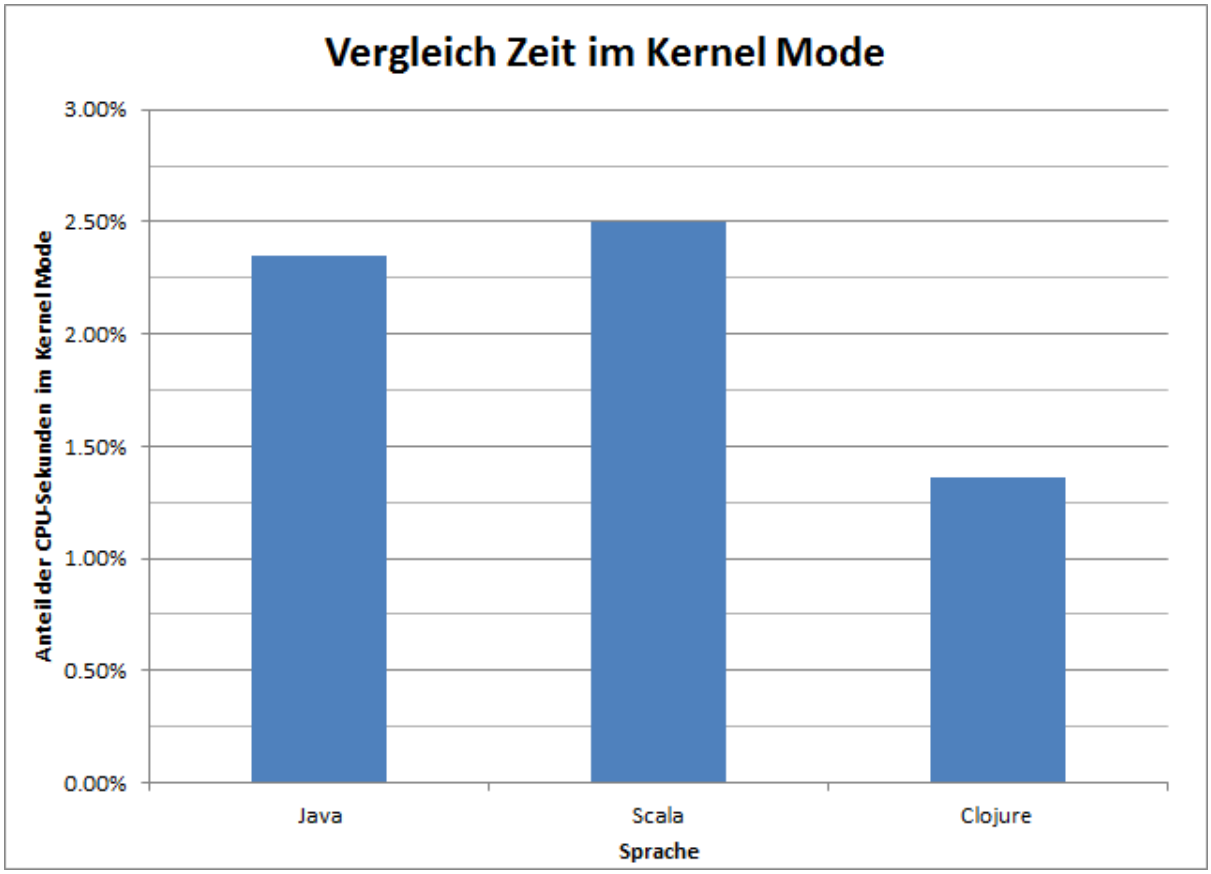

Abbildung 9: Vergleich des Anteils an CPU-Sekunden, welche im Kernel Mode verbracht wurden

7.1.2 Memory

Die folgende Grafik vergleicht die Speicherdaten aller Sprachen. Clojure hat den anderen Sprachen gegenüber einen enorm tiefern Speicherverbrauch. Es muss jedoch bedacht werden, dass das Programm bei einem um 60% tieferen Speicherverbrauch auch eine um ca. 60% längere Ausführungszeit hat. Der Speicherverbrauch von Java und Scala entwickelt sich sehr ähnlich, jedoch ist der Arbeitsspeicherverbrauch von Scala etwas tiefer als jener von Java. Erst kurz vor Programmende steigt der Verbrauch von Scala nochmal stark an und übersteigt jenen von Java. Entsprechend ist auch der durchschnittliche Speicherverbrauch von Java der höchste.

Ein Hinweis zur Tabelle: Aufgrund der grossen Datenmenge enthält die Wertetabelle jeweils nur einen Wert je Sekunde Ausführungszeit. Des weiteren ist die Spalte für Clojure nicht vollständig, da ab Sekunde 23 kein Vergleich mehr mit den anderen Sprachen möglich ist.

Zeit [s]	Java [kB]	Scala [kB]	Clojure [kB]
0	704	548	1300
1	4076	24504	4240
2	64888	145256	12532
3	254820	301740	25872
4	398824	294424	37100
5	315296	243544	64240
6	280544	228976	69396
7	278380	228492	89632
8	287952	233204	120256
9	292080	239428	120256
10	297992	245484	120264
11	305280	252520	120264
12	314180	262084	112896
13	324204	275336	112896
14	342300	293188	112896
15	357680	320268	112896
16	394120	349612	112896
17	404572	376812	112896
18	408484	443108	112896
19	408484	443236	112896
20	419848	451776	112896
21	425884	451780	112896
22	432032	493828	112896
...	-	-	...

Tabelle 13: Vergleich der gemessenen Speicherdaten

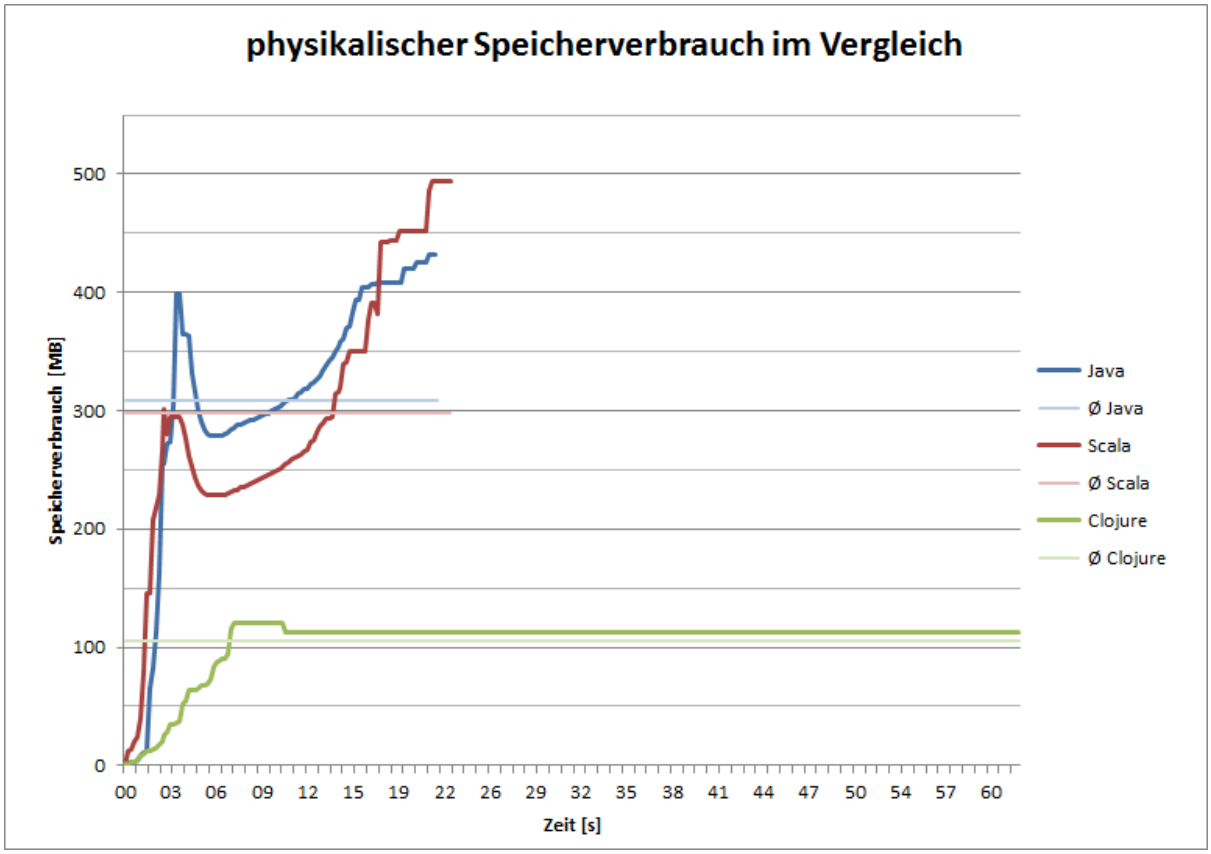

Abbildung 10: Vergleich des Speicherverbrauchs

7.1.3 CPU

Die folgende Grafik zeigt die CPU-Auslastung während der Programmausführung an. Anfangs steigt die CPU-Auslastung bei allen Sprachen ungefäht gleich stark an. Während sich Java und Clojure anschliessend während der ganzen Programmausführung in einem Bereich von ca. 100% bewegen, nimmt die CPU-Auslastung bei Clojure bis zum Programmende stetig ab. Auch hier muss jedoch bedacht werden, dass die Programmausführung bei Clojure weit länger dauert als bei den anderen Sprachen. Beim genaueren Vergleich von Java und Scala fällt auf, dass die CPU-Auslastung bei Java stets leicht höher ist als bei Scala.

Ein Hinweis zur Tabelle: Aufgrund der grossen Datenmenge enthält die Wertetabelle jeweils nur einen Wert je Sekunde Ausführungszeit. Des weiteren ist die Spalte für Clojure nicht vollständig, da ab Sekunde 23 kein Vergleich mehr mit den anderen Sprachen möglich ist.

Zeit [s]	**Java [%]**	**Scala [%]**	**Clojure [%]**
0	0	0	0
1	99	105	82
2	103	105	128
3	144	102	108
4	131	103	115
5	99.8	102	92.4
6	116	100	77
7	114	100	66
8	99.2	101	51.3
9	100	101	46.2
10	100	101	42
11	100	101	38.5
12	99.3	100	35.5
13	99.6	100	33
14	100	100	30.8
15	99.7	100	28.8
16	100	100	27.1
17	100	100	25.6
18	100	100	24.4
19	107	100	23.2
20	103	103	22
21	105	105	22
22	106	105	21
...	-	-	...

Tabelle 14: Vergleich der gemessenen CPU-Auslastungen

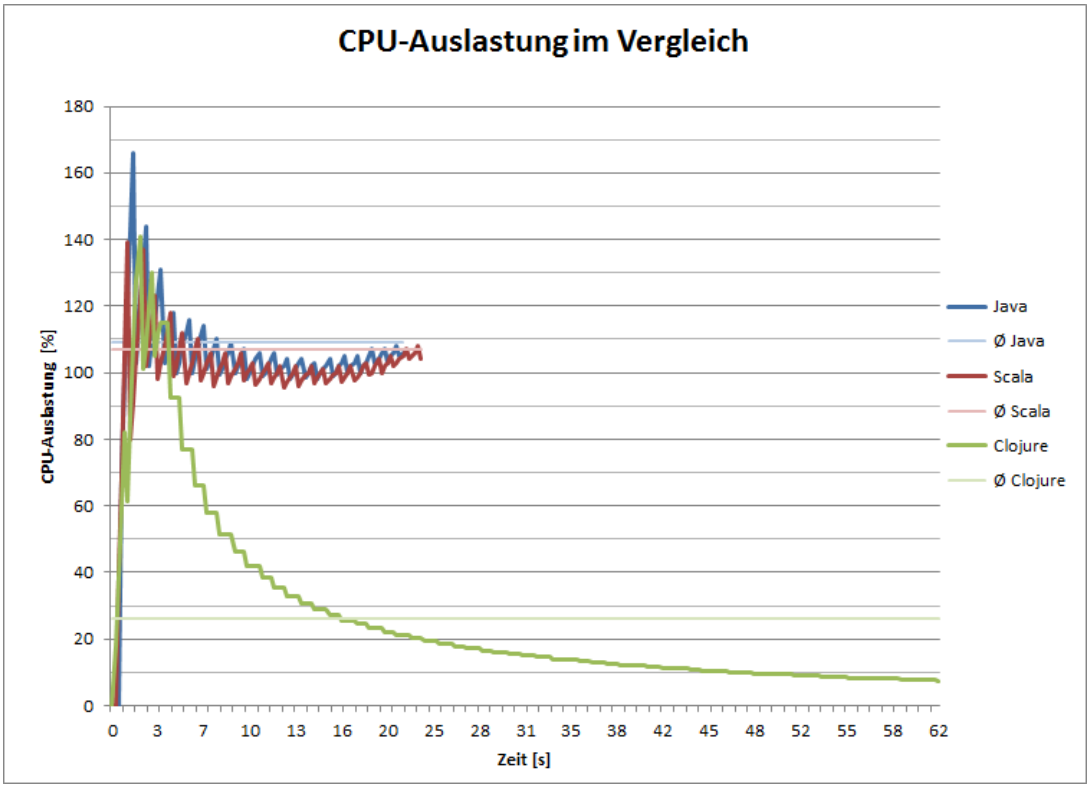

Abbildung 11: Vergleich der CPU-Auslastung

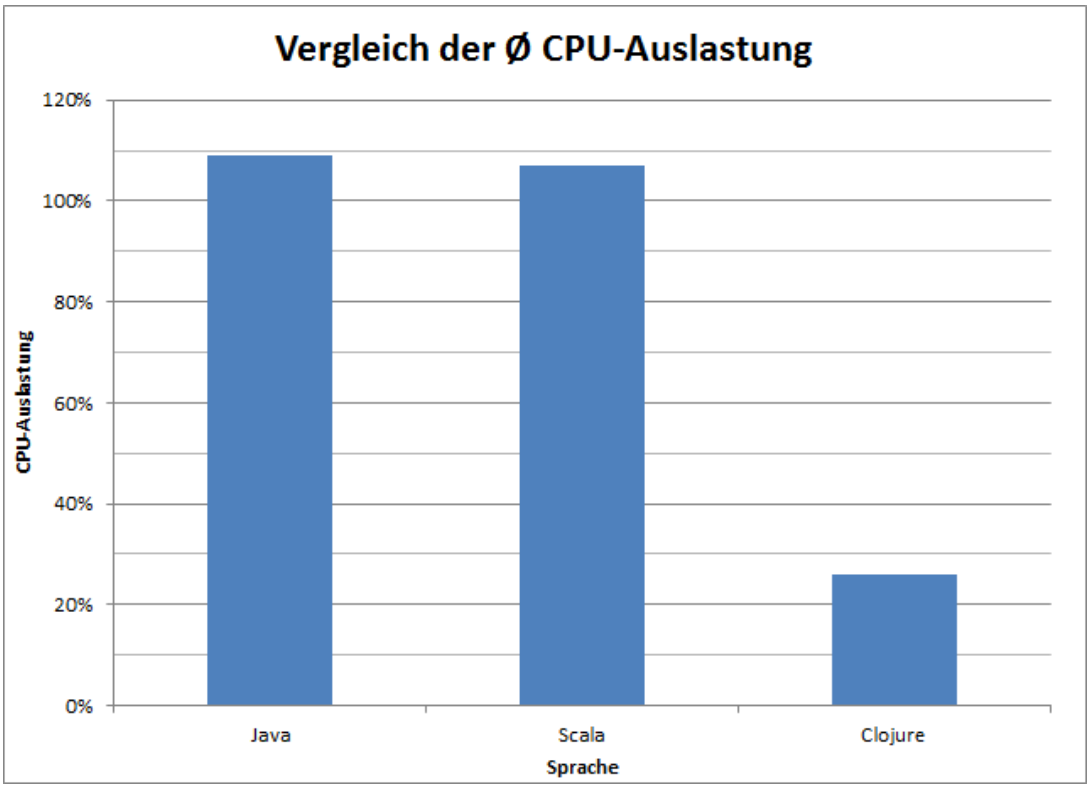

Abbildung 12: Vergleich der durchschnittlichen CPU-Auslastung

7.2 Schlussfolgerung

Die Messresultate entsprechen weitgehend den Erwartungen. Wie es sich auch schon bei anderen Benchmarkes gezeigt hat, sind die Messresultate von Java und Scala sehr ähnlich. Etwas überraschend war lediglich die leicht tiefere Systembelastung von Scala gegenüber Java. Dies bei einer Ausführungszeit welche mit einer Differenz von 0.1 Sekunden beinahe identisch mit derjenigen von Java ist. Clojure ist, wie bei einer so jungen Sprache auch nicht anders zu erwarten, verglichen mit den anderen Sprachen weit abgeschlagen. Zwar ist lag die Belastung von CPU und Memory während der Ausführung stets tiefer als bei den anderen Sprachen, mit einer Zeit von über einer Minute dauerte die Ausführung jedoch ungefähr drei Mal so lange. Da heutige Systeme sehr performant sind und über ausreichend Arbeitsspeicher und Rechenleistung verfügen, wäre eine bessere Nutzung der Ressourcen wünschenswerter gewesen wenn sich dadurch die Ausführzeit verringert hätte. Schliesslich sind die zusätzlichen Ressourcen ja dazu da um genutzt zu werden und damit das System zu verschnellern. Zusammenfassend lässt sich sagen, dass Java und Scala aufgrund der Messresultate auch für Grossprojekte sehr geeignet sind. Diese gehören, wie sich aus externen Tests schliessen lässt, mitunter zu den schnellsten Programmiersprachen überhaupt. Clojure hingegen kann (noch) nicht mit diesen Programmiersprachen mithalten. Es ist daher vor allem bei grösseren Projekte ratsam zu prüfen, ob die von Clojure erbrachte Geschwindigkeit für das Projekt ausreichend ist.

8 Abkürzungsverzeichnis

CPU Central Processing Unit
HWM High Water Mark
IL Intermediate Language
JVM Java Virtual Machine
PID Process Identifier
procfs Process Filesystem
PS Process Status
PTE Page Table Entry
RAM Random Access Memory
RSS Resident Set Size
SED Stream Editor
SMP Symmetric Multiprocessing
VM Virtual Machine

9 Glossar

Arbeitsspeicher	auch Hauptspeicher, dient zum kurzzeitigen Speichern von Daten und ist massiv schneller als Festplatten
Befehlssatz	Eine Menge von Befehlen, welche in Maschinensprache vorliegt
Bytecode	Ein Befehlssatz für eine VM.
CPU-Modi	siehe Kapitel 4.5.3 (Time)
CPU-second	siehe Kapitel 4.5.3 (Time)
Debian	Eine frei erhältiches Client-Betriebssystem welches auf einem Linux-Kernel basiert
Hardware	mechanische und elektronische Teile eines beliebigen Systems, dazu gehören auch Computer
Java Bytecode	Der Bytecode für Java bzw. für die JVM
Java Virtual Machine	Die Implementierung einer VM, welche darauf ausgelegt ist Java Bytecode auszuführen.
Kernel Mode	siehe Kapitel 4.5.3 (Time)
Kompilierung	Übersetzung von Quellcode einer Programmiersprache in Maschinensprache
Maschinensprache	Befehle, die der Prozessor ohne Kompilierung ausführen kann
Open-Source	Software deren Quellcode öffentlich zugänglich ist
Page	eine Einheit in die der Arbeitsspeicher unterteilt wird, es handelt sich um die kleinste Speichereinheit, in der Reservationen durchgeführt werden
Parallelisierung	Aufteilung eines Programms in mehrere Teile, die gleichzeitig ausgeführt werden können. Das Programm wird dadurch multiprozessorfähig
Parameter	ein Wert der einem Programm oder Script mitgegeben wird

Performance-Monitoring-Tool	Programm, welches die Leistungsdaten eines Computers oder Prozesses überwacht
Proc	ein virtuelles Dateisystem welches System- und Prozessinformationen anzeigt
Script	ein in einer Scriptsprache geschriebenes Programm
Scriptsprache	Programmiersprache welche sich speziell für kleine Programme eignet
User Mode	siehe Kapitel 4.5.3 (Time)
Virtual Machine	Eine Programm, welches einen physikalischen Computer simuliert. Sie erhält als Input eine beliebige Sprache und führt diese auf der darunter liegenden Sprache aus.
Virtual Memory	siehe Kapitel 4.5.1 (Memory)

10 Literaturverzeichnis

- Bird, Tim (2009): Runtime Memory Measurement
  http://elinux.org/Runtime_Memory_Measurement (26.02.2011)
- Feiner, Tom (2009): Peak memory usage of a process
  http://serverfault.com/questions/11550/peak-memory-usage-of-a-process
  (26.02.2011)
- Fulgham, Brent (2011): The Computer Language Benchmarks Game.
  http://shootout.alioth.debian.org/ (26.02.2011)
- Gmane.org, Benutzer: shivaligupta (2006): Regarding /proc/<pid>/status
  http://article.gmane.org/gmane.linux.kernel.kernelnewbies/15454/match=
  (26.02.2011)
- Kerrisk, Michael (2010): proc - process information pseudo-file system.
  http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html
  (26.02.2011)
- Mackintosh, David (2010): A definition for a CPU second?
  http://serverfault.com/questions/138703/a-definition-for-a-cpu-second
  (26.02.2011)
- McGrath, Roland (2007): Locking Pages
  http://www.gnu.org/s/libc/manual/html_node/Locking-Pages.htmlLocking-
  Pages
  (26.02.2011)
- Santosa, Mulyadi (2006): When Linux Runs Out of Memory.
  http://linuxdevcenter.com/pub/a/linux/2006/11/30/linux-out-of-
  memory.html
  (26.02.2011)
- Turakhia, Bhavin (2010): Understanding and optimizing Memory utilization.
  http://careers.directi.com/display/tu/Understanding+and+optimizing+Memory+
  utilization (26.02.2011)
- University of Alberta (2010): Understanding Memory
  http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html
  (26.02.2011)
- Unix.com, Benutzer: sysgate (2008): top command + %CPU usage exceeds
  100%? http://www.unix.com/unix-dummies-questions-answers/92541-top-
  command-cpu-usage-exceeds-100-a.html
  (26.02.2011)

- Wikipedia, Benutzer: Guy Harris (2011): Page table
  http://en.wikipedia.org/wiki/Page_table (26.02.2011)
- Wikipedia, Benutzer: MetaEntropy (2010): Code segment.
  http://en.wikipedia.org/wiki/Code_segment (26.02.2011)
- Wikipedia, Benutzer: Mindmatrix (2011): Stack (data structure)
  http://en.wikipedia.org/wiki/Stack_(data_structure) (26.02.2011)
- Wikipedia, Benutzer: Nat682 (2011): Memory segmentation.
  http://en.wikipedia.org/wiki/Segmentation_(memory) (26.02.2011)
- Wikipedia, Benutzer: Rich Farmbrough (2009): Resident set size.
  http://en.wikipedia.org/wiki/Resident_set_size (26.02.2011)
- Wikipedia, unbekannter Autor (2011): Data segment.
  http://en.wikipedia.org/wiki/Data_segment (26.02.2011)
- Wikipedia, unbekannter Autor (2011): Virtual memory.
  http://en.wikipedia.org/wiki/Virtual_memory (26.02.2011)

11 Anhang

Wir erklären hiermit, dass wir die vorliegende interdisziplinäre Projektarbeit eigenständig und ohne unerlaubte fremde Hilfe erstellt haben und dass alle Quellen, Hilfsmittel und Internetseiten wahrheitsgetreu verwendet wurden und belegt sind.

Ort: Datum: Unterschrift: