HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Performance-Scoring-Framework

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2013

Autor(en):	Daniel Marty, Peter Camastral
Betreuer:	Thomas Letsch
Projektpartner:	-
Experte:	-
Gegenleser:	-

Performance-Scoring-Framework

Studenten

- Peter Camastral
- Daniel Marty

Einführung

Mit der Studienarbeit 'Exercise-Repository' können im Praktikumsbetrieb (z.B. Modul Prog2:Algorithmen und Datenstrukturen) einzelne Übungs-Aufgaben mit einem Client-Programm auf einen Server hochgeladen werden wo sie dann unmittelbar einem automatischen Test unterzogen werden. Dem Client werden daraufhin die Resultate zurück übermittelt.
'Exercise-Repository' unterstützt u.a. auch das Führen von Ranglisten aufgrund dieser Resultate.

Um die Performance eines Programms resp. einer Java-Methode solch einer Übungs-Aufgabe zu bestimmen werden diese mit Testdaten zur Ausführung gebracht und entsprechend die Laufzeit gemessen.

Das Problem ist nun, dass 'einfache' Laufzeitmessungen einer grossen Streuung unterliegen was dazu führt, dass beinahe bei jedem Hochladen eine andere Laufzeit als Resultat gemessen wird und somit auch die 'Rangierung' permanent ändert, obwohl keine Änderung am Programm resp. Algorithmus vorgenommen wurde.

Aufgabenstellung

Mit dieser Studienarbeit soll für Java ein genereller Ansatz gesucht und realisiert werden bei welchem solch resultierende Zeiten aus der Laufzeitmessung keiner (oder nur einer minimalsten) Streuung unterliegen.

Mögliche Ansätze könnten u.a. sein: JVM Tool Interface (JVM TI) Agent, Realtime-Extension, Java Native Interface (JNI), Multicore-Allocation, dezidierte JVM mit spezieller Performance-Measurement-Library, etc.

Technologien

- Java
- Eclipse
- Enterprise Architect

Generelles

- Die Vorgaben der Abteilung Informatik [1] sind einzuhalten.
- Die "Generelle Richtlinien für Studien- und Bachelorarbeiten" [2] sind einzuhalten.
- Mit dem CASE-Tool Enterprise Architect ist ein UML-Modell zu führen, welches synchron mit den Programm-Sourcen und der Projekt-Dokumentation ist.
- Ein Java-Entwickler muss mit der Projekt-Dokumentation in die Lage versetzt werden, die Applikation in Betrieb zu nehmen und weiter entwickeln zu können.

Termine

- Montag, 16.09.13 Beginn der Studienarbeit
- Freitag, 20.12.13 17:00 Uhr Abgabe der Studienarbeit

Betreuung

- Betreuer
  Thomas Letsch
  tletsch@hsr.ch
  055 - 22 24 567 (HSR Büro 5.204); 055 - 214 43 50 (Geschäft)
- Besprechungen
  Wöchentliche Besprechung jeweils Donnerstag 17:15 Uhr

Referenzen

[1] www.hsr.ch>HSR-intern>Bachelor-Studiengänge>Informatik>Allgemeine Infos Diplom-, Bachelor- und Studienarbeiten
https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html
[2] "Generelle Richtlinien für Studien- und Bachelorarbeiten"
(v1.5 / 18.09.2011, Thomas Letsch)

Rapperswil, 16.September 2013

Thomas Letsch

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Erklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.
- das ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Ort, Datum:
Rapperswil, 19.12.2013

Name, Unterschrift:

Daniel Marty
D. Marty

Peter Camastral
P. Camastral

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Kurzfassung der Studienarbeit

Abteilung	**Informatik**
Name[n] der Studierenden	**Daniel Marty** **Peter Camastral**
Studienjahr	**HS 2013**
Titel der Studienarbeit	**Performance-Scoring-Framework**
Examinatorin / Examinator	**Thomas Letsch**
Themengebiet	**Software**
Projektpartner	**-**
Institut	**Diverses**

Das Performance-Scoring-Framework bietet eine Möglichkeit, Java Programme oder Methoden in ihrer Ausführungsgeschwindigkeit zu bewerten. Mit Zeitmessungen ist das nicht möglich, da andere Programme und Threadwechsel das Resultat extrem verzerren und somit bei mehreren Durchläufen nie ein konstantes Resultat entstehen kann. Die Lösung wurde mit dem Zählen der von der Virtuellen Maschine ausgeführten Operationen erarbeitet. Dies wird mit einem JVMTI Agent erreicht, der beim Start eines Java Programmes an die Virtuelle Maschine angehängt wird und dann mittels Callbacks die verschiedenen Operationen zählt. Da diese jedoch unterschiedliche Ausführungszeiten haben, musste zusätzlich noch ein Kalibrierprogramm entwickelt werden. Dieses ermittelt die relativen Ausführungszeiten von einzelnen Operationen und generiert eine Datei aus dem das Framework dann Scores für die Operationen bezieht.
Da die Operationen in wenigen Nanosekunden abgearbeitet werden, ist es nur in Linux möglich genaue Messungen zu erstellen, denn unter Windows sind die präzisesten Timer (QueryPerformanceCounter) lediglich auf 300 ns genau. Damit keine anderen Programme die Kalibration beeinflussen, wird mit Shell Skripten der Scheduler so eingerichtet, dass ein Core komplett frei ist und vom System nicht mehr gebraucht werden kann.

Studienarbeit
Performance-Scoring-Framework
Management Summary

Peter Camastral Daniel Marty

19. Dezember 2013

Änderungstabelle

Datum	Version	Änderung	Person
01.10.2013	1.0rc01	Ausgangslage	d1marty
09.10.2013	1.0	Überarbeitung des Textes	d1marty
27.11.2013	1.1rc01	Vorgehen, Technologien	d1marty
04.12.2013	1.1rc02	Ergebnisse, Ausblick	pcamastr
19.12.2013	1.1	Ergebnisse, Ausblick	pcamastr

Inhaltsverzeichnis

1 Ausgangslage

Die Studienarbeit Performance-Scoring-Framework erweitert eine frühere Studienarbeit Exercise-Repository, welche eine Client-Server Applikation realisiert die folgende Dienste erfüllt:

- Übertragung von Binär- und Textdateien vom Client zum Server.
- Kompilierung des eingecheckten Codes.
- Erstellung von Ranglisten.
- Darstellung der Resultate.

Um die Rangliste erstellen zu können braucht die Server Applikation ein Programm, das ihm aus dem übergebenen Code einen eindeutigen Score zurückliefert. Dieses wird in dieser Studienarbeit erarbeitet.

2 Vorgehen, Technologien

Für die Realisierung wurden verschiedene Ansätze geprüft und mit Messungen validiert. Folgende Technologien wurden eingesetzt:

- Agent Libraries mit dem Java Virtual Machine Tool Interface (in C++)
- Skripte für Einrichtung der Umgebung bzw. für den Start der Messungen (in bash)
- Betriebssystemunabhängige Zeitmessungsmethoden (in Assembler)
- Modifizierung der Java Virtual Machine (in C)

3 Ergebnisse

Das Framework zur Auswertung von Javaprogrammen wurde erstellt und liefert Scores zu den Bytecodes aus einer vorkalibrierten Konfigurationsdatei. Die Kalibration dieser Konfigurationsdatei konnte jedoch nicht umgesetzt werden, da die Zeitmessung in der JVMTI zu ungenau ist um exakte Resultate zu garantieren. Die Zeitmessung direkt in der JVM konnte aus zeitlichen Gründen nicht abgeschlossen werden, da der Code zu umfangreich ist.

4 Ausblick

Um die Zeitmessung genügend genau zu machen muss entweder die JVM selber erweitert oder ein eigener Bytecode Interpreter erstellt werden. Für eine Lösung ohne Zeitmessungen könnte vielleicht über ein Simulator wie 'bochs' die Ausführung genau in CPU Cycles gemessen werden.

Studienarbeit
Performance-Scoring-Framework
Technischer Bericht

Peter Camastral Daniel Marty

20. Dezember 2013

Änderungstabelle

Datum	Version	Änderung	Person
27.11.2013	1.0rc01	Einleitung und Übersicht	d1marty
05.12.2013	1.0rc02	Perspektive, Funktion & Abhängigkeiten	pcamastr
18.12.2013	1.1	Erreichte, nicht erreichte Ziele, Schlussfolgerung	d1marty

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabe

Das Performance-Scoring-Framework misst den Aufwand einer Java-Klasse/ Methode und liefert das Ergebnis dann zurück. Das Ergebnis sollte bei gleichbleibendem Code stets dasselbe sein.

1.2 Abhängigkeiten

Unsere Applikation wird von dem Excercise Repository (eine frühere Studienarbeit) aufgerufen und hat keine Abhängigkeiten zu diesem.

1.3 Übersicht über die Arbeit

Der grösste Teil der Arbeit ist der Evaluationsbericht/das Evaluationsdokument, in welchem verschiedene Ansätze angeschaut und beurteilt werden. Thematisiert werden die Debug-Option der JVM, das Java Virtual Machine Tool Interface, die openJDK und verschiedene Arten von Zeitmessungen. In der Abgabe enthalten sind alle geforderten Dokumente sowie der Evaluationsbericht.

- Aufgabenstellung
- Kurzfassung
- Management Summary
- Technischer Bericht
- Evaluationsdokument
- Glossar
- Literaturverzeichnis

2 Resulate

2.1 Erreichte Ziele

Es ist gelungen eine Umgebung für das Performance-Scoring-Framework einzurichten, indem alle Prozesse auf eine andere CPU verschoben werden, damit der zu messende Prozess auf einer isolierten CPU laufen kann. Das Zählen und Unterscheiden der Bytecodes mit einem Agent, ist ebenfalls gelungen, sowie das Bewerten der Resultate mit Werten aus einem Configfile. Es ist auch möglich, nur eine gewisse Methode zu messen. Werte welche noch nicht im Configfile enthalten sind/Bytecodes die noch nie vorgekommen sind (z.B. Initialisierung eines unbekannten Objektes) werden gekennzeichnet und könnten somit kalibriert werden. Der ganze Prozess wurde so gestaltet, dass nur ein Skript gestartet werden muss, welches den Klassen- und Methodennamen verlangt und danach den Score in eine Datei schreibt. Wir konnten ebenfalls zeigen, dass die Messmethoden unter Windows zu ungenau sind um aussagekräftige Messungen zu machen.

2.2 Nicht erreichte Ziele

Es ist nicht gelungen eine brauchbare Kalibrierung durchzuführen. Die Zeit, die einige Bytecodes für ihre Ausführung benötigen, ist abhängig von verschiedenen Faktoren. Diese Faktoren sind zum Beispiel das Objekt beziehungsweise die Anzahl und Art der zu initialisierenden Datenfelder, die Grösse eines Arrays oder die Methode, die mittels invokestatic aufgerufen wird. Diese sind schwer oder garnicht zu ermitteln, da die JVMTI keine Möglichkeit bietet den Operand Stack der Virtuellen Maschine auszulesen und weil das Auslesen des Constant Pools nicht korrekt funktioniert (wird möglicherweise in zukünftigen Java Versionen behoben). Zusätzlich ist der Timer unter Windows zu ungenau um solche Messungen erfolgreich zu machen, denn die Bytecodes haben Ausführungszeiten im Nanosekundenbereich. Aus diesem Grund wurde die Lösung auf das Linux-Betriebsystem eingeschränkt.

3 Schlussfolgerungen

Eine Bewertung der Performance einer Java-Methode ist schwer, da der gleiche Bytecode verschieden lang dauern kann. Die Timer müssen dafür sehr genau sein und einen Unterschied von nur einigen Nanosekunden messen können. Alle getesteten Ansätze hatten zu viel Overhead. Ein möglicher Ansatz wäre die JVM direkt zu modifizieren, was aber nicht sehr portabel sein würde, da man dann nur immer diese modifizierte JVM benutzen könnte, um die Messungen zu machen. Es wäre auch möglich einen eigenen Bytecode Interpreter mit eingebauten Messungen zu erstellen, jedoch könnte dies sehr zeit- und wartungsaufwendig (neue Java Versionen) sein. Ein anderer möglicher Ansatz wäre es einen Emulator wie zum Beispiel bochs zu benutzen und da die Cycles zu zählen, so könnte man die Zeit durch die Anzahl Cycles ersetzten.

Studienarbeit
Performance-Scoring-Framework
Evaluationsdokument

Peter Camastral Daniel Marty

19. Dezember 2013

Änderungstabelle

Datum	Version	Änderung	Person
24.09.2013	1.0rc01	Bytecodeinterpreter (Debug JVM)	pcamastr
30.09.2013	1.1rc01	Profiling Agent mit JVMTI	pcamastr
08.10.2013	1.2rc01	Code eingefügt, JVMTI unter Linux, Ausführungzeiten des Bytecodes	d1marty
15.10.2013	1.2rc02	Analyse der Zeitmessungen	d1marty
16.10.2013		JVMTI Events	pcamastr
23.10.2013	1.2rc03	Analyse im Main	d1marty
30.10.2013	1.2rc04	Overhead, RDTSC	d1marty
30.10.2013		new Opcode Objekt Abhängigkeiten	pcamastr
05.11.2013	1.2rc05	Instanzierung eines Arrays	pcamastr
06.11.2013		Manuelle Einstellung der Ausführung unter Linux	d1marty
12.11.2013	1.2rc06	Isolierung einer CPU unter Linux	d1marty
21.11.2013	1.2rc07	Zweck des Dokumentes, Use Cases	d1marty
26.11.2013		Zeitmessung mit JVMTI	d1marty
26.11.2013		Modifizierung der JVM	d1marty
05.12.2013	1.2rc08	Auslesen des Constant Pools	pcamastr
12.12.2013	1.2rc09	Lösung	d1marty
18.12.2013	1.2	Korrektur	d1marty

Inhaltsverzeichnis

1 Übersicht

1.1 Zweck des Dokumentes

Dieses Dokument zeigt verschiedene Lösungsansätze, um die Performance einer Java-Applikation zu messen. Es werden verschiedene Messungen gemacht, um die Resultate validieren zu können.

1.2 Testsystem

Alle Messungen werden auf einem 64bit Linux Mint 14 Nadia System mit einem 3.5.0-17-generic Kernel und einem Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz Prozessor gemacht.

1.3 Use Cases

Mit den verschiedenen Technologien müssen folgende zwei Use Cases erfüllt werden können:

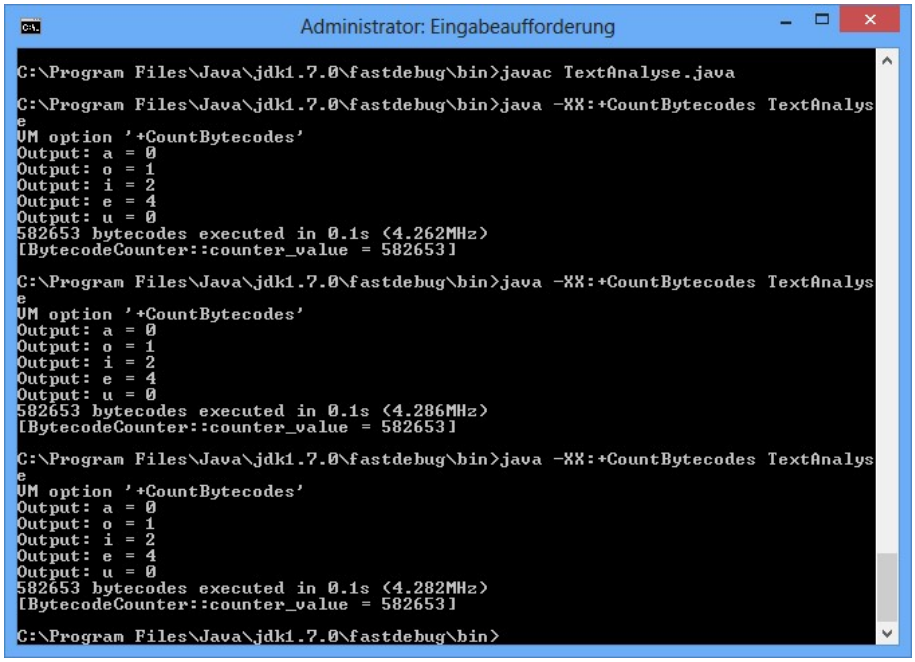
- Kalibrierung: Für jeden Computer der die Systemvoraussetzungen erfüllt, muss eine Kalibrierung erstellt werden können. Während der Kalibrierung werden die Bytecodeausführungszeiten sowie Overheads ermittelt und abgespeichert. Die Kalibrierung kann eine Ausführungszeit von bis zu 2h beanspruchen.
- Ermittlung des Scores: Für eine Applikation wird die Anzahl und die Art der benötigten Bytecodes ermittelt und mit Hilfe der in der Kalibrierung gemessenen Ausführungszeiten beurteilt. Dies sollte im Bereich von Minuten geschehen (vorausgesetzt die Methode kann in diesem Zeitraum ohne das Performance-Scoring-Framework durchlaufen).

2 Bytecode Interpreter Cycle Count mit Debug JVM

Die Debug-Version der HotSpot JVM von Oracle unterstützt spezielle Optionen, die beim Start der JVM mit -XX:+option aktiviert werden können. Die aktuelle Version kann für alle Betriebssysteme auf http://download.java.net/jdk7/archive/b142/binaries/ unter der DEBUG Spalte bezogen werden.

Für die Messung muss das Programm mit dem mitgelieferten Compiler übersetzt und dann mit der Option -XX:+CountBytecodes gestartet werden. Diese Option ist nirgends von Oracle selber dokumentiert http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html! Gefunden wurde sie unter http://www.progdoc.de/papers/Jax2012/jax2012.html#(4).

2.1 Test von Programmieren 2 Übung 1 Aufgabe 2



Die Anzahl ausgeführter Bytecodes bleibt konstant und wird auf der Konsole sowie in dem nach jedem JVM Start erzeugten hotspot.log ausgegeben

3 Profiling Agent mit JVMTI

Mit dem JVM Tool Interface kann in C/C++ eine Agent Library erstellt werden, die beim Programmstart an eine beliebige JVM angehängt wird und vollständige Kontrolle über diese erhält.
Die Standard Profiling Library hprof wird folgendermassen an ein Javaprogramm angehängt:

java –agentlib:hprof=cpu=times TextAnalyse

Dieser Agent produziert eine Datei java.hprof.txt mit den gemessenen Zeitinformationen:
CPU TIME is a profile of program execution obtained by measuring the time spent in individual methods (excluding the time spent in callees), as well as by counting the number of times each method is called. Entries in this record are TRACEs ranked by the percentage of total CPU time. The countfield indicates the number of times each TRACE is invoked.

Da diese Zeitinformation zu ungenau ist, muss ein eigener Agent entwickelt werden, der wie ein Debugger nach jeder Instruktion der JVM etwas Code ausführt. Mit Hilfe von http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#EventSection wurde dies umgesetzt.

Listing 1: instructionagent.cpp

```
1  #include <jvmti.h>
2  #include <algorithm>
3  #include <iostream>
4  #include <array>
5  #include "string.h"
6  #include "BytecodeInfo.h"
7
8  static jvmtiEnv *jvmti;
9  static jint stepcounter = 0;
10 const int BYTECODE_COUNT = 256;
11 std::array<BytecodeInfo, BYTECODE_COUNT> bArray{};
12
13 void JNICALL callbackSingleStep(jvmtiEnv *jvmti_env, ↩
       JNIEnv* jni_env,
14     jthread thread, jmethodID method, jlocation location) ↩
         {
15   // stop timer
16   jvmtiError error;
17   jint bytecodeCountPtr;
18   unsigned char *bytecodesPtr;
19   error = jvmti->GetBytecodes(method, &bytecodeCountPtr, &↩
       bytecodesPtr);
```

```
20    if (error != JVMTI_ERROR_NONE) {
21      printf("error getting bytecode");
22      return;
23    }
24
25    // bArray[bytecodesPtr[location]].addMeasurement(↩
          elapsedTime);
26
27    error = jvmti->Deallocate(bytecodesPtr);
28
29    if (error != JVMTI_ERROR_NONE) {
30      printf("error im deallocate");
31      return;
32    }
33
34    stepcounter++;
35    // start timer
36  }
37
38  JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *↩
        options,
39      void *reserved) {
40
41    jint result;
42    jvmtiError error;
43    jvmtiCapabilities capa;
44    jvmtiEventCallbacks callbacks;
45
46    for (int i = 0; i < BYTECODE_COUNT; i++) {
47      bArray.at(i) = BytecodeInfo(i);
48    }
49    result = jvm->GetEnv((void **) &jvmti, JVMTI_VERSION);
50    if (result != JNI_OK || jvmti == NULL) {
51      printf("unable to access JVM");
52      return JNI_ERR;
53    }
54
55    memset(&capa, 0, sizeof(jvmtiCapabilities));
56    capa.can_generate_single_step_events = 1;
57    capa.can_get_bytecodes = 1;
58    error = jvmti->AddCapabilities(&capa);
59    if (error == JVMTI_ERROR_NOT_AVAILABLE) {
60      printf("error registering capabilities!");
61      return JNI_ERR;
62    }
63
64    // If thread is NULL, the event is enabled or disabled ↩
          globally
65    error = jvmti->SetEventNotificationMode(JVMTI_ENABLE,
```

```
66        JVMTI_EVENT_SINGLE_STEP, NULL);
67    if (error != JVMTI_ERROR_NONE) {
68      printf("step event registration failed");
69      return JNI_ERR;
70    }
71
72    memset(&callbacks, 0, sizeof(callbacks));
73    callbacks.SingleStep = &callbackSingleStep;
74    error = jvmti->SetEventCallbacks(&callbacks, (jint) ↩
          sizeof(callbacks));
75    if (error != JVMTI_ERROR_NONE) {
76      printf("error adding callback");
77      return JNI_ERR;
78    }
79
80    printf("instructionagent loaded successfully \n");
81
82    // start timer
83    return JNI_OK;
84  }
85
86  JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm) {
87    printf("%i steps\n", stepcounter);
88    std::stable_sort(bArray.begin(), bArray.end());
89    std::reverse(bArray.begin(), bArray.end());
90    for (auto  b : bArray) {
91      std::cout << "-------------------------------------\n";
92      std::cout << "Bytecode = " << b.getBytecodeNumber() <<↩
            "\n";
93      std::cout << "-------------------------------------\n";
94      std::cout << " Anzahl = " << b.getCount() << "\n";
95      std::cout << " Zeit = " << b.getElapsedTime() << "\n";
96      std::cout << " Zeit pro Ausfuehrung = " << b.↩
          getAverageTime()
97          << "\n";
98    }
99  }
```

3.1 JVMTI Events

Das SingleStep Event wird abhängig von der Implementation einer JVM nach dem Ausführen einer Instruktion ausgeführt. In der am weitest verbreiteten JVM Hotspot gilt dies für jedes erhöhen des Program Counters. http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/0ed9a90f45e1/src/share/vm/interpreter/bytecodeInterpreter.cpp beschrieben auf Zeilen 174-184.

Listing 2: Auszug aus bytecodeInterpreter.cpp

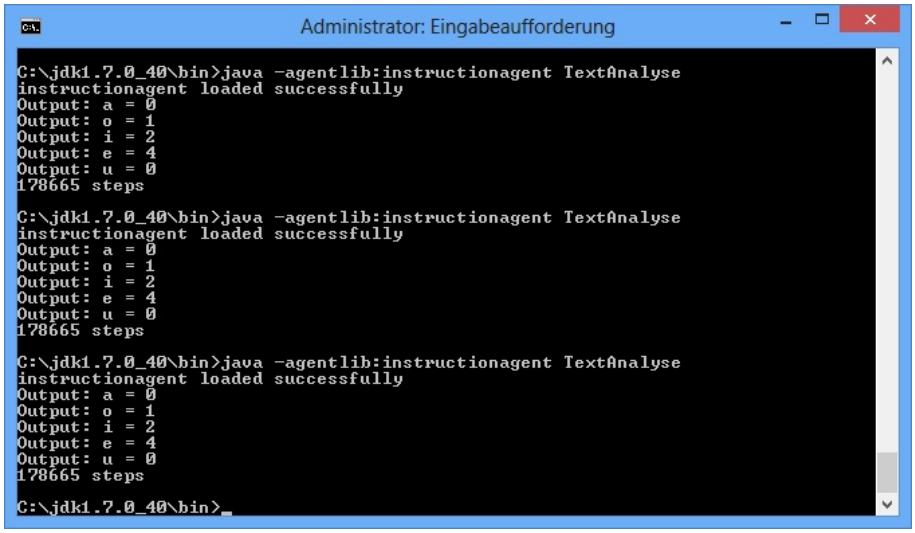
```
1  ...
2  #ifdef VM_JVMTI
3  /* NOTE: (kbr) This macro must be called AFTER the PC has ↩
       been
4     incremented. JvmtiExport::at_single_stepping_point() ↩
          may cause a
5     breakpoint opcode to get inserted at the current PC to ↩
          allow the
6     debugger to coalesce single-step events.
7
8     As a result if we call at_single_stepping_point() we ↩
          refetch opcode
9     to get the current opcode. This will override any other↩
           prefetching
10    that might have occurred.
11 */
12 #define DEBUGGER_SINGLE_STEP_NOTIFY()
13 ...
```

3.2 Übersetzen des Agents unter Windows

Der Agent muss im gleichen Modus wie die ausführende JVM übersetzt werden (64/32 Bit). Fürs Übersetzen der Library ist wichtig, dass der Header jvmti.h von einer JDK im Verzeichnis include zur Verfügung steht. Zusätzlich muss auch das include/win32 Verzeichnis noch angegeben werden für den Header jni_md.h.
Die fertige Library als .dll muss danach in ein Verzeichnis von der Umgebungsvariable PATH oder ins bin Verzeichnis der JDK kopiert werden und kann mit
java –agentlib:instructionagent TextAnalyse
aufgerufen werden.

3.2.1 Übersetzen des Agents unter Linux

Der Agent kann unter Linux[1] mit Eclipse CDT kompiliert werden. Dazu muss ein Shared Library Projekt erstellt werden und die Toolchain Linux GCC ausgewählt werden. Die Quellcodedatei des Agents muss in das Projekt kopiert werden. Über Properties - C / C++ General - Path and Symbols muss für die Sprache GNU C++ noch /usr/lib/jvm/default-java/include[2] inkludiert werden und das Symbol __GXX_EXPERIMENTAL_CXX0X__ hinzugefügt werden für C++ 11. Um die Library nun kompilieren zu können, muss unter Properties - C/C++ Build - Settings bei Miscellaneous das Flag -fPIC gesetzt werden. Um die C++ 11 Funktionalität zu nutzen muss zusätzlich noch bei other flags –std=c++0x angehängt werden. Die Shared Library kann nachher entweder in /usr , /usr/lib kopiert werden oder der Debug Ordner kann in /etc/ld.so.conf[3] hinzugefügt werden. Beim Aufruf von **java** wird dann nur der Name der Library angegeben, das heisst ohne die Endung .so oder den Zusatz lib.



3.3 Bytecodeausführungszeit messen unter Linux

Für die Zeitmessung wird eine Hilfsklasse verwendet, in der die Anzahl der Aufrufe und die Ausführungszeit jedes Bytecodes gespeichert wird. Danach wird die durchschnittliche Zeit der Aufrufe ausgerechnet. Für die nachfolgenen Ausführungen wird die Prog2-Klasse Text-Analyse verwendet.

[1] Getestet unter Linux Mint 14 mit g++ 4.7.2 und CDT 8.2.1 for Eclipse Kepler
[2] Pfade können bei anderen Distributionen abweichen
[3] Um den Library Cache zu aktualisieren muss noch **ldconfig** ausgeführt werden.

Listing 3: BytecodeInfo.h

```
#ifndef BYTECODEINFO_H_
#define BYTECODEINFO_H_

class BytecodeInfo {
public:
  BytecodeInfo();
  BytecodeInfo(int bytecodeNumber);

  int getBytecodeNumber();
  int getCount();
  int getElapsedTime();
  int getAverageTime();
  void addMeasurement(int time);

  inline bool operator <(const BytecodeInfo &b) const {
    return (averageTime < b.averageTime);
  }

private:
  int bytecodeNumber, count, elapsedTime, averageTime;
  void setCount(int count);
  void incrementCount();
  void setTime(int time);
  void addTime(int time);
};

#endif /* BYTECODEINFO_H_ */
```

Listing 4: BytecodeInfo.cpp

```
#include "BytecodeInfo.h"

BytecodeInfo::BytecodeInfo() {
}

BytecodeInfo::BytecodeInfo(int bytecodeNumber) :
    bytecodeNumber(bytecodeNumber), count(0), elapsedTime(0), averageTime(0) {
}

int BytecodeInfo::getBytecodeNumber() {
  return bytecodeNumber;
}

int BytecodeInfo::getCount() {
  return count;
}
```

```
17
18 void BytecodeInfo::setCount(int count) {
19   this->count = count;
20 }
21
22 void BytecodeInfo::incrementCount() {
23   this->count++;
24 }
25
26 int BytecodeInfo::getElapsedTime() {
27   return elapsedTime;
28 }
29
30 int BytecodeInfo::getAverageTime() {
31   if (count) {
32     return averageTime;
33   }
34   return 0;
35 }
36
37 void BytecodeInfo::setTime(int time) {
38   this->elapsedTime = elapsedTime;
39 }
40
41 void BytecodeInfo::addTime(int time) {
42   this->elapsedTime += time;
43 }
44
45 void BytecodeInfo::addMeasurement(int time){
46   incrementCount();
47   addTime(time);
48   averageTime = elapsedTime / count;
49 }
```

3.3.1 Zeitmessung mit der Funktion clock()

Um die Zeit mit der Bibliotheksfunktion clock() zu messen muss folgender Code im Agent hinzugefügt werden.

Listing 5: clock() im instructionagent.cpp

```
1  ...
2  #include <time.h>
3  clock_t stepbegin, stepend;
4  ...
5  void JNICALL callbackSingleStep(jvmtiEnv *jvmti_env, ↩
       JNIEnv* jni_env, jthread thread, jmethodID method, ↩
       jlocation location) {
6  stepend = clock();
7  ...
8  stepbegin = clock();
9  }
10 ...
```

Das Ergebnis dieser Messung (outputClock.txt) zeigt, dass die Messung mittels clock() keine genügend hohe Auflösung bietet und uns somit keinen Aufschluss auf die Ausführungslängen der Bytecodes gibt.

3.3.2 Zeitmessung mit JVMTI

Das JVMTI beinhaltet eine Funktion um die Zeit zu messen. Folgender Code ermöglicht es die verstrichene Zeit zu messen.

Listing 6: JVMTI Zeitmessung im instructionagent.cpp

```
1  ...
2  static jlong stepbegin, stepend;
3  ...
4  void JNICALL callbackSingleStep(jvmtiEnv *jvmti_env, ↩
       JNIEnv* jni_env, jthread thread, jmethodID method, ↩
       jlocation location) {
5  jvmti->GetTime(&stepend);
6  ...
7  jvmti->GetTime(&stebegin);
8  }
9  ...
```

Diese Funktion bietet eine bessere Auflösung und es sind Unterschiedliche Ausführungszeiten sichtbar (outputJVMTI.txt). Nachfolgend wird gezeigt wie die GetTime Funktion in der JVM implementiert worden ist.

Listing 7: .../openjdk/hotspot/src/share/vm/prims/jvmtiEnv.cpp

```
1  jvmtiError
2  JvmtiEnv::GetTime(jlong* nanos_ptr) {
3    *nanos_ptr = os::javaTimeNanos();
4    return JVMTI_ERROR_NONE;
```

Listing 8: .../openjdk/hotspot/src/os/linux/vm/os_linux.cpp

```
1  jlong os::javaTimeNanos() {
2    if (Linux::supports_monotonic_clock()) {
3      struct timespec tp;
4      int status = Linux::clock_gettime(CLOCK_MONOTONIC, &tp↩
           );
5      assert(status == 0, "gettime error");
6      jlong result = jlong(tp.tv_sec) * (1000 * 1000 * 1000)↩
           + jlong(tp.tv_nsec);
7      return result;
8    } else {
9      timeval time;
10     int status = gettimeofday(&time, NULL);
11     assert(status != -1, "linux error");
12     jlong usecs = jlong(time.tv_sec) * (1000 * 1000) + ↩
           jlong(time.tv_usec);
13     return 1000 * usecs;
14   }
15 }
```

In der Implementation für Linux sieht man, dass zuerst geprüft wird ob die monotonic_clock unterstützt wird. Ist dies nicht der Fall, wird mit µs Auflösung gemessen und dann mit 1000 multipliziert (der Datentyp von **gettimeofday()** hat nur ein usec Datenfeld). Ob das System monotonic_clock unterstützt, kann durch das Auslesen des Symbols POSIX_TIMERS überprüft werden.

Listing 9: Ausgabe von **man clock_gettime**

```
1  AVAILABILITY
2  On POSIX systems on which these functions  are  available,
3  the  symbol _POSIX_TIMERS  is defined in <unistd.h> to
4  a value greater than 0.
5  The symbols _POSIX_MONOTONIC_CLOCK, _POSIX_CPUTIME,
6  _POSIX_THREAD_CPUTIME indicate that
7  CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID,
8  CLOCK_THREAD_CPUTIME_ID are available.
9  (See also sysconf(3).)
```

Ist die Ausgabe von
getconf -a | grep POSIX_TIMERS | '{awk print $2}' und
getconf -a | grep POSIX_MONOTONIC | '{awk print $2}' grösser als 0, sind die Funktionen verfügbar. Die Genauigkeit von **clock_gettime()** beträgt etwa 1/Hz (1/Taktfrequenz) und kann mit der Funktion **clock_getres()** überprüft werden. In unserem Fall beträgt sie 1ns.

3.3.3 Analyse der Zeitmessungen

Da die Mehrfachausführung der Zeitmessung zu sehr unterschiedlichen Resultaten führt, muss die Streuung untersucht werden. Zusätzlich könnte es auch von Nutzen sein, den Minimal- und Maximalwert zu speichern. Eventuell könnte die minimale Ausführzeit eines Bytecodes verwendet werden. Um mehrere Outputs auswerten zu können, wird die BytecodeInfo Klasse serialisierbar gemacht, deshalb werden die Operatoren >> und << wie folgt definiert.

Listing 10: Erweiterung im BytecodeInfo.h

```
1  ...
2  public:
3    int getMinTime();
4    int getMaxTime();
5  ...
6    friend std::ostream& operator<<(std::ostream& os, const ↩
       BytecodeInfo& b)
7    {
8      os << b.bytecodeNumber << '\n';
9      os << b.count << '\n';
10     os << b.averageTime << '\n';
11     os << b.minTime << '\n';
12     os << b.maxTime << '\n';
13     return os;
14   }
15
16   friend std::istream& operator>>(std::istream& is, ↩
       BytecodeInfo& b)
17   {
18     is >> b.bytecodeNumber >> b.count >> b.averageTime >> ↩
         b.minTime >> b.maxTime;
19     return is;
20   }
21
22 private:
23 ...
24 int minTime, maxTime;
25 ...
```

Listing 11: Anpassungen im BytecodeInfo.cpp

```
...
BytecodeInfo::BytecodeInfo(int bytecodeNumber) :
    bytecodeNumber(bytecodeNumber), count(0), elapsedTime(0), averageTime(0), minTime(0), maxTime(0){
}
...
int BytecodeInfo::getMinTime()
{
  return minTime;
}

int BytecodeInfo::getMaxTime()
{
  return maxTime;
}

void BytecodeInfo::addTime(int time) {
  if (time < minTime || minTime == 0)
    minTime = time;
  if (time > maxTime)
    maxTime = time;
  this->elapsedTime += time;
}
...
```

Die gemessenen Resultate der einzelnen Bytecodes können mit folgendem Code in ein File geschrieben werden. Der Modus ios::app bewirkt, dass der Inhalt angehängt wird, somit können die Resultate mehrerer Durchläufe in einer Datei festgehalten werden.

Listing 12: Anpassungen im instructionagent.cpp

```
...
#include <fstream>
...
JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm) {
...
  std::ofstream ofs{};
  ofs.open("PSFResults.txt", std::ios::app);
  for (auto b : bArray)
    ofs << b;
  ofs.close();
...
}
```

Um für eine Testklasse mehrere Durchläufe zu machen, kann die Klasse unter Linux mit dem Bash-Skript startMeasurement.sh gestartet werden. Die Klasse TextAnalyse.class kann so im Terminal 1000mal ausgeführt werden.

Listing 13: Commands im Terminal

```
$ ./startMeasurement.sh
zu testende Klasse
/home/<user>/TextAnalyse
Anzahl Durchläufe
1000
```

Listing 14: startMeasurement.sh

```
1  #!/bin/bash
2  echo "zu testende Klasse"
3  read CLASSPATH
4  echo "Anzahl Durchläufe"
5  read ROUNDS
6  PATH=$(dirname $CLASSPATH)
7  CLASSNAME=$(/usr/bin/basename $CLASSPATH)
8
9  if [ -f $PATH/PSFResults.txt ]
10 then
11     /bin/rm -f $PATH/PSFResults.txt
12   echo "alte Messungen wurden gelöscht"
13 fi
14
15 for (( c=1; c<=$ROUNDS; c++ ))
16 do
17   echo "Messung $c von $ROUNDS gestartet"
18   /usr/bin/java -agentlib:agentLinux -classpath $PATH ↩
         $CLASSNAME
19   echo "Messung $c von $ROUNDS abgeschlossen"
20 done
```

Die BytecodeInfo-Objekte der 1000 Ausführungen sind nun alle in der Datei PSFResults.txt gespeichert, was ziemlich unübersichtlich und gross ist. Das Programm MeasurementAnalyzer wird verwendet um den Output zu analysieren. Dabei werden jeweils die Minimal- und die Maximalwerte für den Minimal-, den Average- und den Maximalwert jedes Bytecodes von allen Durchläufen extrahiert. Der Wertebereich für die obengenannten Werte wird dann ausgegeben.

Listing 15: MeasurementAnalyzer.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <fstream>
4  #include <vector>
5  #include "BytecodeInfo.h"
6
7  int main(int argc, char * argv[]) {
8
9    if (argc != 2) {
10     std::cout
11         << "Ungueltige Angabe der Parameter -> ↩
               MeasurementAnalyzer <file> \n";
12     return 1;
13   }
14
15   std::ifstream ifs { };
16   ifs.open(argv[1]);
17
18   BytecodeInfo b { };
19
20   int counter { 0 };
21   std::vector<std::vector<BytecodeInfo>> bVVector { };
22   std::vector<BytecodeInfo> bVector { };
23   while (ifs >> b) {
24     bVector.push_back(b);
25     counter = (counter+1) % 256;
26     if (!counter) {
27       bVVector.push_back(bVector);
28       bVector.clear();
29     }
30   }
31
32   std::cout << "Analyse von " << bVVector.size() << " ↩
        Messungen \n";
33
34   for (int i = 0; i < 255; i++){
35     int minMin { 0 }, maxMin { 0 }, minMax { 0 }, maxMax {↩
           0 }, minAvg { 0 }, maxAvg { 0 }, count { 0 };
36     for (std::vector<BytecodeInfo> v : bVVector){
37       BytecodeInfo b = v.at(i);
38       if (!count)
39         count = b.getCount();
40       if (b.getMinTime() < minMin || minMin == 0)
41         minMin = b.getMinTime();
42       if (b.getMinTime() > maxMin)
43         maxMin = b.getMinTime();
44
45       if (b.getMaxTime() < minMax || minMax == 0)
```

```
46          minMax = b.getMaxTime();
47        if (b.getMaxTime() > maxMax)
48          maxMax = b.getMaxTime();
49
50        if (b.getAverageTime() < minAvg || minAvg == 0)
51          minAvg = b.getAverageTime();
52        if (b.getAverageTime() > maxAvg)
53          maxAvg = b.getAverageTime();
54      }
55      std::cout << "-----------------------------------\n";
56      std::cout << "Bytecode = " << i << "\n";
57      std::cout << "-----------------------------------\n";
58      std::cout << " Anzahl  = " << count << '\n';
59      std::cout << " MinTime = " << minMin << '-' << maxMin ↩
            << '\n';
60      std::cout << " MaxTime = " << minMax << '-' << maxMax ↩
            << '\n';
61      std::cout << " AvgTime = " << minAvg << '-' << maxAvg ↩
            << '\n';
62    }
63  }
```

Der Output des Programmes mit einem nach Listing 13 erstellten PSFResults.txt ist in der Datei 1000Measurements.txt sichtbar. Eine Liste aller Bytecodes ist in Bytecodes.pdf zu finden. Die Fixierung auf den Minimalwert alleine bringt nichts, denn es gibt Bytecodes bei denen sowohl der niedrigste wie auch der höchste Maximalwert erheblich höher als der Minimalwert ist. Die Bytecodes 1 [aconst_null], 3 [iconst_0], 5 [iconst_2], 6 [iconst_3], 19 [ldc_w], 89 [dup], 153 [ifeq], 178 [getstatic], 184 [invokestatic], 187 [new] haben immer mindestens einmal eine Ausführzeit von über 100µs, die Bytecodes 42 [aload_0] und 183 [invokespecial] haben sogar mindestens einmal eine Ausführungszeit von 40ms pro Durchlauf, unabhängig von der geladenen Klasse[4].

3.3.4 Analyse im Main

Die oben erwähnten Werte könnten aber auch nur bei der Initialisierung der Virtual Machine auftauchen. Um dies herauszufinden, muss erkannt werden, ob man in der Mainmethode ist, was mit folgender Codeerweiterung erreicht werden kann.

[4] Auswertung von TextAnalyse.class und Sequence.class gespeichert in TextAnalyse10000.txt bzw. Sequence100.txt

Listing 16: Anpassungen im instructionagent.cpp

```
1  ...
2  static jint realstepcounter = 0;
3  static bool inMain = false;
4  ...
5
6  void JNICALL callbackSingleStep(jvmtiEnv *jvmti_env, ↩
       JNIEnv* jni_env, jthread thread, jmethodID method, ↩
       jlocation location) {
7    ...
8    if (in_main) {
9      realstepcounter++;
10     bArray[bytecodesPtr[location]].addMeasurement(stepend ↩
         - stepbegin); // nur noch Bytecodes im Main -> log
11   }
12   ...
13 }
14
15 void JNICALL callbackMethodEntry(jvmtiEnv *jvmti_env, ↩
       JNIEnv* jni_env, jthread thread, jmethodID method) {
16   char* name_ptr;
17   jvmtiError error = jvmti->GetMethodName(method, &↩
       name_ptr, NULL, NULL);
18   if (error != JVMTI_ERROR_NONE) {
19     printf("method name failed");
20     return;
21   }
22   std::string name(name_ptr);
23   if (!name.compare(std::string("main"))) {
24     std::cout << "entering: " << name.c_str() << std::endl↩
         ;
25     in_main = true;
26   }
27   jvmti->Deallocate((unsigned char*) name_ptr);
28 }
29
30 void JNICALL callbackMethodExit(jvmtiEnv *jvmti_env, ↩
       JNIEnv* jni_env, jthread thread, jmethodID method, ↩
       jboolean was_popped_by_exception, jvalue return_value)↩
        {
31   char* name_ptr;
32   jvmtiError error = jvmti->GetMethodName(method, &↩
       name_ptr, NULL, NULL);
33   if (error != JVMTI_ERROR_NONE) {
34     printf("method name failed");
35     return;
36   }
37   std::string name(name_ptr);
38   if (!name.compare(std::string("main"))) {
```

```
39      std::cout << "leaving: " << name.c_str() << std::endl;
40      in_main = false;
41    }
42    jvmti->Deallocate((unsigned char*) name_ptr);
43  }
44
45  JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *↩
      options,
46      void *reserved) {
47    ...
48    capa.can_generate_method_entry_events = 1;
49    capa.can_generate_method_exit_events = 1;
50    ...
51    error = jvmti->SetEventNotificationMode(JVMTI_ENABLE,
52        JVMTI_EVENT_METHOD_ENTRY, NULL);
53    if (error != JVMTI_ERROR_NONE) {
54      printf("ENTRY event registration failed");
55      return JNI_ERR;
56    }
57    error = jvmti->SetEventNotificationMode(JVMTI_ENABLE,
58        JVMTI_EVENT_METHOD_EXIT, NULL);
59    if (error != JVMTI_ERROR_NONE) {
60      printf("EXIT event registration failed");
61      return JNI_ERR;
62    }
63    ...
64    callbacks.MethodEntry = &callbackMethodEntry;
65    callbacks.MethodExit = &callbackMethodExit;
66    ...
67  }183 to hex
68
69  JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm) {
70    ...
71    printf("%i steps in main\n", realstepcounter);
72    ...
73  }
```

Misst man nur die Ausführungszeiten der Bytecodes innerhalb der Mainmethode sind die maximalen Zahlen kleiner. Es hat aber immer noch viele Bytecodes die mindestens einmal eine Ausführungszeit von etwa 10-50µs haben und die zwei Bytecodes 42 [aload_0] und 187 [new], bei denen es 300µs sind[5]. Folgende Optionen des Java Application Launchers bewirken keine Veränderung:
-Xint -> interpreter-mode only.
-Xnoclassgc -> disable class garbage collection.
-Xmsn -> specify the initial size, in bytes, of the memory allocation pool.
-XmXn -> specify the max size, in bytes, of the memory allocation pool.

Auch eine Prozessprioritäterhöhung mittels nice bleibt ohne Effekt.[6]

```
sudo nice -n -20 sudo -u $USER Befehl
```

3.3.5 Overhead Messung

Der Overhead von den Zeitmessungen sowie den verschiedenen Events muss ermittelt werden, damit man weiss, welche Verhältnisse die einzelnen Messungen zueinander haben. Um den Overhead von der Zeitmessungsmethode zu bestimmen, wird diese sehr oft gestartet und gleich wieder gestoppt. Der minimale Wert wird dann in eine Datei Config.txt gespeichert, wo sie beim nächsten Mal, anstatt neu zu berrechnen, was ziemlich lange dauert, geholt werden kann. Der gemessene Wert wird dann bei der Zeitmessung wieder abgezogen. Wird ein Event gehookt, so wird am Schluss der Methode der "beginstep-Zähler" zurückgesetzt[7].

Listing 17: Anpassungen im instructionagent.cpp

```
1 #include <boost/filesystem.hpp>
2 ...
3 static jlong minNanoSec{};
4 ...
5 void JNICALL callbackSingleStep(jvmtiEnv *jvmti_env, ↩
    JNIEnv* jni_env, jthread thread, jmethodID method, ↩
    jlocation location) {
6   ...
7   bArray[bytecodesPtr[location]].addMeasurement(stepend - ↩
      stepbegin - minNanoSec);
8   ...
```

[5] aus SequenceMain100.txt bzw. TextAnalyseMain10000.txt
[6] Auswertung mit allen Optionen und Priorität in TextAnalyseMainOptimized1000.txt
[7] Um den Code unter Linux kompilieren zu können muss das Paket libboost-filesystem-dev installiert sein und im Eclipse unter Properties - C/C++ General - Paths and Symbols - Libraries noch der String boost_filesystem hinzugefügt werden

```
9  }
10 ...
11 JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *↩
       options, void *reserved) {
12   ...
13   printf("instructionagent loaded successfully \n");
14   jlong nanoSec { };
15   if (!boost::filesystem::exists("Config.txt")) {
16     std::cout << "Overhead der Zeitmessung wird gemessen \↩
         n";
17     const unsigned long LOOPS = 1000000000;
18     for (volatile unsigned long i = 0; i < LOOPS; i++) {
19       jvmti->GetTime(&stepbegin);
20       jvmti->GetTime(&stepend);
21       nanoSec = stepend - stepbegin;
22       if (nanoSec < minNanoSec || minNanoSec == 0)
23         minNanoSec = nanoSec;
24     }
25     std::ofstream ofs { };
26     ofs.open("Config.txt");
27     ofs << minNanoSec << '\n';
28     ofs.close();
29   } else {
30     std::ifstream ifs { };
31     ifs.open("Config.txt");
32     ifs >> minNanoSec;
33     ifs.close();
34   }
35   ...
36   jvmti->GetTime(&stepbegin);
37   return JNI_OK;
38 }
39 ...
```

3.3.6 Zeitmessung mittels RDTSC

Da die Zeitmessung mittels jvmti->GetTime(jlong* ptr) unter Windows eine zu geringe Auflösung besitzt muss eine Alternative dazu entwickelt werden. Der TSC (Time Stamp Counter) könnte sich dafür eignen. Der TSC ist ein 64bit Register, das auf allen x86 Prozessoren vorhanden ist. Zusätzlich zu den unten aufgeführten Ergänzungen, müssen auch die Zeilen jvmti->GetTime(stepbegin) und jvmti->GetTime(stepend) zu start = rdtsc() bzw. stop = rdtsc() geändert werden.

Listing 18: Ergänzungen im instructionagent.cpp

```
...
//  Windows
#ifdef _WIN32

#include <intrin.h>
unsigned long long rdtsc() {
  return __rdtsc();
}
unsigned long long start, stop;

//  Linux/GCC
#else

__inline__ uint64_t rdtsc() {
  uint32_t lo, hi;
  __asm__ __volatile__ (
      "xorl %%eax,%%eax \n        cpuid"
      ::: "%rax", "%rbx", "%rcx", "%rdx");
  __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
  return (uint64_t) hi << 32 | lo;
}
uint64_t start, stop;
#endif
...
```

Die Messwerte[8] sind im Schnitt etwa doppelt so hoch wie mit GetTime. Unter Windows sind die Werte aber deutlich genauer.

3.3.7 Vergleich der Bytecodemessung mit einer Zeitmessung

Um die zwei Varianten vergleichen zu können braucht es einen zweiten Agent, welcher die Zeit vom Eintreten einer Methode (in diesem Fall Main) bis zum Verlassen misst. Der Code ist ähnlich dem instructionagent.cpp einfach ohne Bytecodeauswertungen und Singlestepevents.

[8] TextAnalyseMain1000RDTSC.txt gemessen unter Linux

3.3.8 Manuelle Einstellung der Ausführung unter Linux

Wird ein Prozess gestartet so entscheidet das Betriebsystem automatisch auf welchen CPUs und mit welcher Taktfrequenz die CPUs laufen. Da wir die Zeit messen und bei einem geringeren Takt mehr Zeit vergeht bis die gleiche Arbeit erledigt ist, wäre es von Vorteil die Kontrolle über den CPU-Takt zu haben. Mit **cat /proc/cpuinfo | grep cpu MHz** kann man die aktuelle Taktfrequenz der CPUs auslesen und mit **taskset -p <pid>** sieht man die Maske welcher dem Prozess mitgegeben wurde. 1 bedeutet zum Beispiel, dass der Prozess nur auf dem CPU 0 laufen darf. 3 hingegen, dass der Prozess sowohl auf CPU 0 sowie auch auf CPU 1 laufen darf. Standardmässig besteht die Maske nur aus Einsen, was bedeutet, dass jeder CPU benutzt wird. Will man dies nun ändern startet man einen Prozess mit **taskset -c 0 <prozess>**. Die Taktfrequenzstufen und Grenzen einer CPU sind durch die Hardware vorgegeben. Unter Linux entscheidet ein sogenannter Scalinggovernor wie die Taktfrequenz geregelt wird. Standardmässig ist der Governor **ondemand** eingestellt. Das bedeutet, dass die Frequenz den aktuellen Bedürfnissen angepasst wird und sprunghaft angehoben oder gesenkt werden kann. Der Governor **performance** sorgt dafür, dass die CPU immer mit der im Governor definierten maximalen Frequenz läuft. Um dies zu erreichen muss "performance" in den scaling_governor geschrieben werden. Dies wird erreicht mit **echo performance | sudo tee /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor**[9]. Wird **taskset** und das Modifizieren des Governors kombiniert, kann man einen Prozess auf nur einem CPU, der jederzeit auf der maximalen Taktfrequenz läuft, starten.

[9] Die Änderung ist flüchtig und wird nach einem Neustart wieder auf **ondemand** gesetzt

3.3.9 Isolierung einer CPU unter Linux

Das Problem an **taskset** ist, dass nicht sichergestellt werden kann dass andere Prozesse ebenfalls auf dieser CPU ausgeführt werden. Um dieses Problem zu beheben müssen cpusets erstellt werden. Cpuset ist ein Kernelobjekt, welchem ein Set von CPUs zugeordnet werden kann. Wenn ein Prozess einem Cpuset zugeteilt wird, kann er nur auf den im Cpuset enthaltenen CPUs ausgeführt werden. Um ein Cpuset zu erstellen muss man das Cpuset- Filesystem zuerst mounten. Das Hauptverzeichnis repräsentiert das ganze System mit allen CPUs und Memory Nodes. Ein Prozess gehört genau zu einem Cpuset. Child-Prozesse werden dem gleichen Cpuset zugeordnet wie der Parent-Prozess. Um eine CPU zu isolieren werden auf einem System mit n CPUs ein Cpuset mit (n-1) CPUs für das System und ein Cpuset mit einer CPU für das Performance Scoring Framework erstellt. Die Erstellung erfolgt mit **mkdir**. Alle benötigten Dateien werden vom System automatisch erstellt. Durch das Eintragen der gewünschten Werte in die Dateien, können die Cpusets konfiguriert werden. Bei beiden Sets wird das cpu_exclusive-Flag gesetzt, damit andere Cpusets nicht überlappen dürfen. Welche CPUs den Prozessen zugeteilt sind, kann mit folgendem Skript ermittelt werden.

Listing 19: show_cpu_assignment.sh

```
1  #!/bin/bash
2
3  for stat_file in `ls -1 /proc/*/task/*/status | sort -u`
4  do
5      content=`cat ${stat_file} 2>/dev/null`
6      name=`printf "${content}" | grep "Name:" | awk '{ ↩
           print $2 }'`
7      pid=`printf "${content}" | grep "Pid:" | head -1 | awk↩
           '{ print $2 }'`
8      cpus_allowed=`printf "${content}" | grep "Cpus_allowed↩
           :" | awk '{ print $2 }'`
9      cpus_allowed_list=`printf "${content}" | grep "↩
           Cpus_allowed_list:" | awk '{ print $2 }'`
10
11     if [ "${name}" != "" ]
12     then
13         echo "Name: $name"
14         echo "Pid: $pid"
15         echo "CPUs allowed: $cpus_allowed"
16         echo "CPUs allowed list: $cpus_allowed_list"
17     fi
18 done
```

Die Umbegung kann mit folgendem Skript eingerichtet werden. Es werden zwei Cpusets erstellt und alle laufenden Prozesse auf das Cpuset “sys“ verschoben indem die PID’s aller Prozesse in die tasks Datei des gewünschten Sets geschrieben wird. Das Skript benötigt Rootrechte und die Änderung wird bei einem Neustart rückgängig gemacht.

Listing 20: init_environment.sh

```
1  #!/bin/bash
2
3  if [ "`whoami`" != "root" ]
4  then
5    echo "ERROR: Skript muss mit Rootrechten ausgeführt ↩
       werden"
6    exit 1
7  fi
8
9  posix_timers=`getconf -a | grep POSIX_TIMERS | awk '{ ↩
     print $2 }'`
10 if [ $posix_timers -lt 1 ]
11 then
12   echo "ERROR: Timer auf diesem System ist zu ungenau"
13   exit 2
14 fi
15
16 cpu_count=$(grep family /proc/cpuinfo | wc -l)
17 echo "Anzahl CPUs: "$cpu_count
18 if [ $cpu_count == 1 ]
19 then
20   echo "ERROR: nicht genügend CPUs vorhanden"
21   exit 3
22 elif [ $cpu_count == 2 ]
23 then
24   cpu_sys=0
25   echo "CPU $cpu_sys wird für das System reserviert"
26 else
27   cpu_sys=0-$((cpu_count - 2))
28   echo "CPUs $cpu_sys werden für das System reserviert"
29 fi
30 cpu_psf=$(($cpu_count - 1))
31 echo "CPU $cpu_psf wird für das Performance Scoring ↩
     Framework reserviert"
32
33 if [ ! -d /dev/cpuset ]
34 then
35   mkdir /dev/cpuset
36   mount -t cpuset cpuset /dev/cpuset/
37 fi
38
```

```
39 cd /dev/cpuset
40 file=$(find -maxdepth 1 -iname '*cpus')
41 prefix=${file%cpus}
42 prefix=${prefix#./}
43 mkdir sys
44 cd sys
45 echo $cpu_sys > ${prefix}cpus
46 echo 1 > ${prefix}cpu_exclusive
47 echo 0 > ${prefix}mems
48 cd ..
49 mkdir psf
50 cd psf
51 echo $cpu_psf > ${prefix}cpus
52 echo 1 > ${prefix}cpu_exclusive
53 echo 0 > ${prefix}mems
54 echo 0 > ${prefix}sched_load_balance
55 echo 1 > ${prefix}mem_hardwall
56 cd ..
57 #move all tasks to the sys cpu set
58 for task in `cat tasks`
59 do
60   echo $task > sys/tasks 2> /dev/null
61 done
```

Vor dem Ausführen des Skriptes dürfen alle Prozesse auf allen CPUs laufen[10]. Sind die Cpusets dann erstellt sind fast alle Prozesse auf die CPUs 0 - (n-1) beschränkt. Es gibt aber einige Ausnahmen, welchen nicht verschoben werden können zum Beispiel SoftIRQs oder Watchdogs[11]. Neu gestartete Applikationen werden ebenfalls dem Cpuset "sys" zugeteilt[12].

[10] assignment_before_init.txt
[11] assignment_after_init.txt
[12] assignment_after_init_app.txt (Steam, Openoffice, Firefox, pluma)

3.3.10 Messung von Bytecodes mit Abhängigkeiten

Da einige der Bytecodes in ihrer Laufzeit von Objekten(new, newarray usw.) oder verschiedener Betriebssystemfunktionen(invokestatic usw.) abhängen, muss dies bei der Messung berücksichtigt werden, damit diese voneinander getrennt ausgewertet werden können.

Listing 21: Ergänzungen im instructionagent.cpp

```
1 ...
2 if(bytecodesPtr[location] == 187){ //new: next 2 bytes in ↩
    bytecode are a 16 bit big endian reference to ↩
    classname in constant pool
3 int objectid = (bytecodesPtr[location+1]<<8)+bytecodesPtr[↩
    location+2];
4 ...
```

3.3.11 Auslesen vom Constant Pool

Um die Referenzen im Bytecode auflösen zu können, muss in der entsprechenden Klasse der Constant Pool ausgelesen werden. Dieser beinhaltet alle festen Werte die im Programm vorkommen, vorallem Methodennamen. Die JVMTI liefert den ganzen Constant Pool als rohe Bytefolge, mit einem zusätzlichen Pointer auf dessen Länge. Mit Hilfe von http://en.wikipedia.org/wiki/Java_class_file#The_constant_pool wird diese Folge dekodiert und als XML zwischengespeichert.

Listing 22: Ergänzungen im instructionagent.cpp

```
1  ...
2  jvmti->GetConstantPool(mainclass, &poolcount, &↩
     poolbytecount, &poolptr);
3  while (i < poolbytecount) {
4    std::stringstream val;
5    switch (poolptr[i]) {
6    case 1:
7      stringsize = (poolptr[i + 1] << 8) + poolptr[i + 2];
8      for (int j = 0; j < stringsize; j++) {
9        val << poolptr[i + 3 + j];
10     }
11         stringMap.insert(std::pair<int,std::string>(↩
             poolentrycounter, val.str()));
12     i += 2 + stringsize + 1;
13     break;
14 ...
```

Methoden werden im Bytecode zuerst mit einer 2 Byte langen ID referenziert, die im Constant Pool ein Eintrag mit 2 IDs ist. Die erste zeigt auf den Klassennamen und die zweite auf den Methodennamen.

Da der Constant Pool aus den Klassen kompiliert wird, sind die vorhandenen IDs immer unterschiedlich. Als Konsequenz müssen für den vergleich zweier Klassen alle Referenzen bis auf die Stringnamen aufgelöst werden.

Die Funktion wurde im Agent deaktiviert, da in der aktuellen JVMTI Version ein Bug existiert, der es nicht erlaubt den Speicher, der durch GetConstantPool reserviert wird, mit Deallocate wieder freizugeben. Daraus resultiert auch bei kleinen Programmen eine extreme Belastung vom RAM.

3.3.12 Messung der Instanzierung eines Arrays

Die Instanzierung eines neuen Arrays ist abhängig vom primitiven Typ und der Grösse, was beides im Bytecode eine Stelle vor bzw. nach dem newarray Opcode zu finden ist.

Listing 23: Ergänzungen im instructionagent.cpp

```
1  ...
2  if(bytecodesPtr[location] == 188){ //newarray
3              jint size;
4              if(previousstep < 9)//iconst
5                  size = previousstep - 3;
6              else if(previousstep < 18)//bipush
7                  size = (int)bytecodesPtr[location-1];
8              else if(previousstep < 27){//iload
9                  jint slot = previousstep - 26;
10                 jint stacksize;
11                 jvmti->GetFrameCount(NULL, &stacksize);
12                 jvmti->GetLocalInt(NULL, stacksize, slot, ↩
                       &size);
13             }
14 ...
```

Die Grösse von einem neuen Array, der mit newarray oder multianewarray erstellt wird, kann nicht ausgelesen werden, wenn sie über arraylength von einem anderen Array übernommen wird. Dies ist nicht möglich weil JVMTI keine Möglichkeit anbietet auf den Operand Stack der JVM zuzugreiffen, auf dem diese Grösse abgelegt ist.

4 Modifizierung der JVM

4.1 Einleitung

Da der Profiling Agent mit JVMTI einige Ungenauigkeiten verursachte, wird nun untersucht, was für Resultate erzielt werden, wenn die Modifikationen direkt im Sourcecode der JVM vorgenommen werden.

4.2 Kompilieren der JDK

Zuerst muss der Sourcecode der JDK heruntergeladen werden. Der Sourcecode der OpenJDK kann unter http://openjdk.java.net/ heruntergeladen werden. In unserem Fall wird das Bundle 7u benutzt, das direkt von https://jdk7.java.net/source.html bezogen werden kann. Nach dem Entpacken muss in der Konsole ins openjdk Verzeichnis gewechselt werden. Mit **make sanity** kann geprüft werden, ob alle Abhängigkeiten erfüllt sind. Wahrscheinlich wird die Ausgabe etwa so aussehen:

Listing 24: Ausgabe von **make sanity**

```
1  WARNING: LANG has been set to en_US.UTF-8, this can cause
2  build failures. Try setting LANG to 'C'.
3
4  ERROR: Your LD_LIBRARY_PATH environment variable is set.
5  This may produce binaries binaries incorrectly.
6  Please unset it and start your build again.
7
8  ERROR: FreeType version  2.3.0  or higher is required.
9  make[2]: Entering directory
10 `/home/dani/openjdk/jdk/make/tools/freetypecheck'
11 /bin/mkdir -p /home/dani/openjdk/build/linux-amd64/btbins
12 rm -f /home/dani/openjdk/build/linux-amd64/btbins/
13 freetype_versioncheck
14 make[2]: Leaving directory `/home/dani/openjdk/jdk/make/
15 tools/freetypecheck'
16 Failed to build freetypecheck.
17
18 ERROR: You do not have access to valid Cups header files.
19 Please check your access to
20 /usr/include/cups/cups.h
21 and/or check your value of ALT_CUPS_HEADERS_PATH,
22 CUPS is frequently pre-installed on many systems,
23 or may be downloaded from http://www.cups.org
```

Um diese Probleme zu beseitigen, muss zuerst die Umgebungsvariable LANG verändert werden. Dies kann mit **export LANG=C** erreicht werden. Die Umgebungsvariable LD_LIBRARY_PATH kann mit
unset LD_LIBRARY_PATH gelöscht werden. Für FreeType und CUPS müssen zusätzliche Pakete installiert werden. Die Installation erfolgt mittels **sudo apt-get install libcups2-dev libfreetype6-dev**. Nun sollte **make sanity** "Sanity check passed" augeben. Sollten immer noch Fehler auftreten, müssen mit oben gezeigten Methoden entweder die entsprechenden Umgebungsvariablen gesetzt werden oder die benötigten Pakete nachinstalliert werden. Zusätzlich müssen noch die Umgebungsvariablen ALT_BOOTDIR und ALLOW_DOWNLOADS mittels
export ALT_BOOTDIR=/usr/lib/jvm/default-java
export ALLOW_DOWNLOADS=true
gesetzt werden und **sudo apt-get install libxrender-dev** ausgeführt werden. Anschliessend kann **make all** ausgeführt werden.
Unter .../openjdk/build/linux-amd64/bin kann nun die gerade kompilierte JDK ausgeführt werden.

5 Lösung

Da es mit dem Agent nicht möglich ist genaue Zeitmessungen zu machen und sich die Modifkation als schwierig herausstellt, ist nur der Usecase “Ermittlung des Scores“ abgedeckt.

5.1 Umfang

- Skript um Umgebung einzurichten (Listing 20)
- Start-Skript für die Berechnung des Scores (get_score.sh)
- JVMTI-Agent für die Berechnung des Scores (agentScore.cpp)
- Skript um Umgebung aufzulösen (terminate_environement.sh)

Listing 25: get_score.sh

```
1  #!/bin/bash
2
3  echo "zu testende Klasse"
4  read class_path
5  echo "zu testende Methode (ENTER = main)"
6  read method_name
7  echo $method_name > method.txt
8  path=$(dirname $class_path)
9  class_name=$(/usr/bin/basename $class_path)
10 if [ -f score.txt ]
11 then
12     /bin/rm -f score.txt
13   echo "alter Score gelöscht"
14 fi
15 /usr/bin/java -Xint -agentlib:agentScore -classpath $path ↩
       $class_name
16 /bin/rm -f method.txt
```

Listing 26: agentScore.cpp

```
1  #include <jvmti.h>
2  #include <iostream>
3  #include <sstream>
4  #include <fstream>
5  #include <stdint.h>
6  #include <map>
7  #include "string.h"
8  #include <boost/property_tree/ptree_fwd.hpp>
9  #include <boost/property_tree/xml_parser.hpp>
10
11 std::string getMethodNameFromFile();
12 void loadConfigFile();
13 void writeScoreToFile(std::string finalScore);
14 std::string convertScoreToString(unsigned long long score)↩
       ;
15 void reloadConstantPool(jmethodID method);
16
17 static const std::string CONFIG_FILE = "Config.xml";
18 static const int NO_SCORE = -1;
19 static const int NOT_CALIBRATED_SCORE = 0;
20 static const int DEFAULT_SCORE = 1;
21
22 static jvmtiEnv *jvmti;
23 static jint previousstep = 0;
24 static unsigned long long int stepcounter = 0;
25 static unsigned long long int score = 0;
26 static bool inMethod = false;
27 static bool methodAvailable = false;
28 static bool calibrated = true;
29 static std::string methodName;
30 boost::property_tree::ptree bTree; //bytecodes
31
32 std::map<int, std::string> stringMap;//constantpool
33 std::map<int, int> classReferenceMap;
34 std::map<int, int> typeReferenceMap;
35
36 int getScoreFromConfig(const std::stringstream& ss);
37
38 void JNICALL callbackSingleStep(jvmtiEnv *jvmti_env, ↩
       JNIEnv* jni_env,
39     jthread thread, jmethodID method, jlocation location) ↩
           {
40   jvmtiError error;
41
42   if (inMethod) {
43     stepcounter++;
44     jint bytecodeCountPtr;
45     unsigned char *bytecodesPtr;
```

```
46      error = jvmti->GetBytecodes(method, &bytecodeCountPtr,&bytecodesPtr);
47      if (error != JVMTI_ERROR_NONE) {
48        std::cout << "ERROR: Fehler beim Holen des Bytecodes\n";
49        return;
50      }
51
52      //some bytecodes have additional dependencies
53      std::stringstream ss;
54      int dependencyid, classref, typeref;
55      std::string classname, typeName;
56      switch (bytecodesPtr[location]) {
57
58      case 182: // invokevirtual dependency METHODREF
59      case 183: // invokespecial dependency METHODREF
60      case 184: // invokestatic dependency METHODREF
61      case 185: // invokeinterface dependency INTERFACEREFREF
62      case 186: // invokedynamic dependency METHODREF
63        dependencyid = (bytecodesPtr[location + 1] << 8)
64            + bytecodesPtr[location + 2]; // ref to constant pool entry with 2 values
65              try{
66              classref = classReferenceMap.at(dependencyid);
67              classname = stringMap.at(classref);
68              typeref = typeReferenceMap.at(dependencyid);
69              typeName = stringMap.at(typeref);
70              }
71              catch(std::out_of_range e){ //constant pools of some core java libraries are not accessible(obfuscated)
72                  score += DEFAULT_SCORE;
73                  break;
74              }
75        ss << "bytecodeid." << (int) bytecodesPtr[location] << ".methodname."
76            << typeName << "/" << classname << ".min";
77          score += getScoreFromConfig(ss);
78        break;
79
80      case 187:
81        //new: next 2 bytes in bytecode are a 16 bit big endian reference to classname in constant pool
82        dependencyid = (bytecodesPtr[location + 1] << 8)
83            + bytecodesPtr[location + 2];
84              try{
85              classref = classReferenceMap.at(dependencyid);
86              classname = stringMap.at(classref);
```

```
87              }
88              catch(std::out_of_range e){ //constant pools ↩
                    of some core java libraries are not ↩
                    accessible(obfuscated)
89                  score += DEFAULT_SCORE;
90                  break;
91              }
92          ss << "bytecodeid." << (int) bytecodesPtr[location] ↩
                << ".objectid."
93              << classname << ".min";
94          score += getScoreFromConfig(ss);
95          break;
96
97      case 188:
98          //newarray: value on top of stack is size of array, ↩
                next byte of bytecode is its type
99              //for some preceding bytecodes we can get what↩
                    was pushed on stack
100         jint size;
101         if (previousstep > 2 && previousstep < 9)     //↩
                iconst
102           size = previousstep - 3;
103         else if (previousstep == 16)      //bipush
104           size = (int) bytecodesPtr[location - 1];
105         else if (previousstep == 17)      //sipush
106           size = (bytecodesPtr[location - 2] << 8)
107             + bytecodesPtr[location - 1];
108         else {
109           //arraylength not measurable, but same sized array↩
                    was already measured
110           score+=DEFAULT_SCORE;
111           break;
112         }
113         ss << "bytecodeid." << (int) bytecodesPtr[location] ↩
                << ".type."
114             << (int) bytecodesPtr[location + 1] << ".size." ↩
                    << size
115             << ".min";
116         score += getScoreFromConfig(ss);
117         break;
118
119     default:
120         ss << "bytecodeid." << (int) bytecodesPtr[location] ↩
                << ".min";
121         score += getScoreFromConfig(ss);
122         break;
123     }
124     previousstep = bytecodesPtr[location];
125     error = jvmti->Deallocate(bytecodesPtr);
```

```
126     if (error != JVMTI_ERROR_NONE) {
127       std::cout << "ERROR: Fehler beim Freigeben \n";
128       return;
129     }
130   }
131 }
132 
133 void JNICALL callbackMethodEntry(jvmtiEnv *jvmti_env, ↩
      JNIEnv* jni_env,
134     jthread thread, jmethodID method) {
135     jvmtiError error;
136 
137     if(inMethod){
138         //reloadConstantPool(method); // disabled because ↩
              bytes of constant pool cannot be Deallocated ↩
              in current JVMTI
139         return;
140     }
141 
142   char* name_ptr;
143   error = jvmti->GetMethodName(method, &name_ptr, NULL, ↩
        NULL);
144   if (error != JVMTI_ERROR_NONE) {
145     std::cout << "ERROR: Holen des Methodennamen ↩
          fehlerhaft \n";
146     return;
147   }
148   std::string name(name_ptr);
149   if (!name.compare(methodName)) {
150     std::cout << "entering: " << name.c_str() << std::endl↩
          ;
151     inMethod = true;
152         //reloadConstantPool(method); // disabled because ↩
              bytes of constant pool cannot be Deallocated ↩
              in current JVMTI
153 
154       error = jvmti->SetEventNotificationMode(JVMTI_ENABLE↩
            ,
155           JVMTI_EVENT_SINGLE_STEP, NULL);
156       if (error != JVMTI_ERROR_NONE) {
157         std::cout << "ERROR: step event registration ↩
              fehlerhaft \n";
158         return;
159       }
160 
161     methodAvailable = true;
162 
163     error = jvmti->SetEventNotificationMode(JVMTI_ENABLE,
164         JVMTI_EVENT_METHOD_EXIT, NULL);
```

```
165     if (error != JVMTI_ERROR_NONE) {
166       std::cout << "ERROR: exit event registration ↩
              fehlgeschlagen \n";
167     }
168   }
169   jvmti->Deallocate((unsigned char*) name_ptr);
170 }
171
172 void JNICALL callbackMethodExit(jvmtiEnv *jvmti_env, ↩
      JNIEnv* jni_env,
173     jthread thread, jmethodID method, jboolean ↩
          was_popped_by_exception,
174     jvalue return_value) {
175   char* name_ptr;
176   jvmtiError error = jvmti->GetMethodName(method, &↩
        name_ptr, NULL, NULL);
177   if (error != JVMTI_ERROR_NONE) {
178     std::cout << "ERROR: Holen des Methodennamen ↩
          fehlerhaft \n";
179     return;
180   }
181   std::string name(name_ptr);
182   if (!name.compare(methodName)) {
183     std::cout << "leaving: " << name.c_str() << std::endl;
184     inMethod = false;
185
186         error = jvmti->SetEventNotificationMode(↩
              JVMTI_ENABLE,
187           JVMTI_EVENT_SINGLE_STEP, NULL);
188       if (error != JVMTI_ERROR_NONE) {
189       std::cout << "ERROR: singlestep event disable ↩
              fehlgeschlagen \n";
190       }
191
192         error = jvmti->SetEventNotificationMode(↩
              JVMTI_DISABLE,
193         JVMTI_EVENT_METHOD_ENTRY, NULL);
194     if (error != JVMTI_ERROR_NONE) {
195       std::cout << "ERROR: entry event disable ↩
              fehlgeschlagen \n";
196     }
197
198     error = jvmti->SetEventNotificationMode(JVMTI_DISABLE,
199         JVMTI_EVENT_METHOD_EXIT, NULL);
200     if (error != JVMTI_ERROR_NONE) {
201       std::cout << "ERROR: exit event registration ↩
              fehlgeschlagen \n";
202     }
203   }
```

```
204   jvmti->Deallocate((unsigned char*) name_ptr);
205 }
206
207 JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *↩
      options,
208     void *reserved) {
209   jint result;
210   jvmtiError error;
211   jvmtiCapabilities capa;
212   jvmtiEventCallbacks callbacks;
213   methodName = getMethodNameFromFile();
214
215   result = jvm->GetEnv((void **) &jvmti, JVMTI_VERSION);
216   if (result != JNI_OK || jvmti == NULL) {
217     std::cout << "ERROR: Zugang zur JVM nicht möglich \n";
218     return JNI_ERR;
219   }
220
221   memset(&capa, 0, sizeof(jvmtiCapabilities));
222   capa.can_generate_single_step_events = 1;
223   capa.can_get_bytecodes = 1;
224   capa.can_generate_method_entry_events = 1;
225   capa.can_generate_method_exit_events = 1;
226   capa.can_get_constant_pool = 1;
227   error = jvmti->AddCapabilities(&capa);
228   if (error == JVMTI_ERROR_NOT_AVAILABLE) {
229     std::cout << "ERROR: Fehler beim Registrieren der ↩
        Capabilities \n";
230     return JNI_ERR;
231   }
232
233   // If thread is NULL, the event is enabled or disabled ↩
      globally
234   error = jvmti->SetEventNotificationMode(JVMTI_ENABLE,
235       JVMTI_EVENT_METHOD_ENTRY, NULL);
236   if (error != JVMTI_ERROR_NONE) {
237     std::cout << "ERROR: ENTRY event registration ↩
        fehlerhaft \n";
238     return JNI_ERR;
239   }
240
241   memset(&callbacks, 0, sizeof(callbacks));
242   callbacks.SingleStep = &callbackSingleStep;
243   callbacks.MethodEntry = &callbackMethodEntry;
244   callbacks.MethodExit = &callbackMethodExit;
245   error = jvmti->SetEventCallbacks(&callbacks, (jint) ↩
      sizeof(callbacks));
246   if (error != JVMTI_ERROR_NONE) {
```

```
247     std::cout << "ERROR: Fehler beim Hinzufügen der ↩
            Callbacks \n";
248     return JNI_ERR;
249   }
250
251   loadConfigFile();
252
253   std::cout << "agentScore erfolgreich geladen \n";
254   return JNI_OK;
255 }
256
257 JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm) {
258   std::cout << "Auswertung: \n";
259   std::cout << "Steps: " << stepcounter << '\n';
260   std::cout << "Score: " << score << '\n';
261   std::string finalScore;
262   if (!methodAvailable) {
263     std::cout << "ERROR: Die angegebene Methode wurde ↩
            nicht gefunden \n";
264     finalScore = "0";
265   } else if (!calibrated) {
266     std::cout << "WARNING: Wert ungenau, Kalibrierung ↩
            notwendig \n";
267     finalScore = "~" + convertScoreToString(score);
268   } else {
269     finalScore = convertScoreToString(score)+ "";
270   }
271   boost::property_tree::xml_writer_settings<char> settings↩
        ('\t', 1);
272   write_xml(CONFIG_FILE, bTree, std::locale(), settings); ↩
        //order?
273   writeScoreToFile(finalScore);
274 }
275
276 void loadConfigFile() {
277   try {
278     read_xml(CONFIG_FILE, bTree,
279         boost::property_tree::xml_parser::trim_whitespace)↩
            ;
280   } catch (boost::property_tree::xml_parser::↩
        xml_parser_error e) {
281     //probably filenotfound
282   }
283 }
284
285 int getScoreFromConfig(const std::stringstream& ss) {
286   int byteCodeScore = bTree.get<int>(ss.str(), NO_SCORE);
287   if (byteCodeScore == NOT_CALIBRATED_SCORE) {
288     calibrated = false;
```

```
289     return DEFAULT_SCORE;
290   }
291   if (byteCodeScore == NO_SCORE) {
292     bTree.put(ss.str(), NOT_CALIBRATED_SCORE);
293     calibrated = false;
294     return DEFAULT_SCORE;
295   }
296   return byteCodeScore;
297     return 1;
298 }
299
300 std::string getMethodNameFromFile() {
301   std::ifstream ifst;
302   ifst.open("method.txt");
303   std::string methodName;
304   if (ifst.is_open())
305     ifst >> methodName;
306   if (methodName.size() == 0)
307     methodName = "main";
308   ifst.close();
309   return methodName;
310 }
311
312 void writeScoreToFile(std::string finalScore) {
313   std::cout << finalScore << '\n';
314   std::ofstream ofst;
315   ofst.open("score.txt");
316   ofst << finalScore.c_str();
317   ofst.close();
318 }
319
320 std::string convertScoreToString(unsigned long long score)↩
      {
321   char string[20];
322   sprintf(string, "%llu", score);
323   std::string finalScore(string);
324   return finalScore;
325 }
326
327 void reloadConstantPool(jmethodID method){
328     stringMap.erase(stringMap.begin(), stringMap.end());
329     classReferenceMap.erase(classReferenceMap.begin(), ↩
          classReferenceMap.end());
330     typeReferenceMap.erase(typeReferenceMap.begin(), ↩
          typeReferenceMap.end());
331
332   jvmtiError error;
333     jclass thisClass;
334
```

```
335   error = jvmti->GetMethodDeclaringClass(method, &↩
          thisClass);
336   if (error != JVMTI_ERROR_NONE) {
337     std::cout << "ERROR: Fehler beim Holen der Klasse \n";
338     return;
339   }
340
341   jint poolcount;
342   jint poolbytecount;
343   unsigned char *poolptr;
344
345   error = jvmti->GetConstantPool(thisClass, &poolcount, &↩
          poolbytecount,
346       &poolptr);
347   if (error != JVMTI_ERROR_NONE) {
348     std::cout << "ERROR: Fehler beim Holen des ↩
            Constantpools \n";
349     return;
350   }
351
352   int stringsize;
353   int i = 0;
354   int poolentrycounter = 0;
355
356   while (i < poolbytecount) {
357
358     poolentrycounter++;
359
360     std::stringstream val;
361
362     switch (poolptr[i]) {
363     case 1: // modified UTF8 string
364       stringsize = (poolptr[i + 1] << 8) + poolptr[i + 2];
365       for (int j = 0; j < stringsize; j++) {
366         val << poolptr[i + 3 + j];
367       }
368             stringMap.insert(std::pair<int,std::string>(↩
                  poolentrycounter, val.str()));
369       i += 2 + stringsize + 1;
370       break;
371
372         // tag 2 does not exist!
373
374     case 3: // 32 bit int
375     case 4: // 32 bit float
376       i += 5;
377       break;
378
```

```
379     case 9: // field refernence : 16 bit class 16 bit typename
380     case 10: // method refernence : 16 bit class 16 bit typename
381     case 11: // interface refernence : 16 bit class 16 bit typename
382     case 12: // name and type descriptor reference : 16 bit name 16 bit type
383             classReferenceMap.insert(std::pair<int, int>(poolentrycounter,(poolptr[i + 1] << 8) + (poolptr[i + 2])));
384             typeReferenceMap.insert(std::pair<int, int>(poolentrycounter,(poolptr[i + 3] << 8) + (poolptr[i + 4])));
385       i += 5;
386       break;
387
388     case 5: // 64 bit long
389     case 6: // 64 bit double
390       i += 9;
391       break;
392
393     case 7: // 16 bit class reference
394     case 8: // 16 bit UTF8 reference
395             classReferenceMap.insert(std::pair<int, int>(poolentrycounter,(poolptr[i + 1] << 8) + (poolptr[i + 2])));
396       i += 3;
397       break;
398
399     default:
400       std::cout << "ERROR: Unbekannter Constant Pool Tag " << poolptr[i] << "\n";
401       return;
402     }
403   }
404
405   error = jvmti->Deallocate(poolptr); // does not work -> huge memory leak
406   if (error != JVMTI_ERROR_NONE) {
407     std::cout << "ERROR: Fehler beim Freigeben \n";
408     return;
409   }
410 }
```

Listing 27: terminate_environment.sh

```
1  #!/bin/bash
2
3  if [ "`whoami`" != "root" ]
4  then
5    echo "ERROR: Skript muss mit Rootrechten ausgeführt ↩
       werden"
6    exit 1
7  fi
8
9  if [ ! -d /dev/cpuset/psf ]
10 then
11   echo "ERROR: Performance Scoring Framework Umgebung ↩
       nicht eingerichtet"
12   exit 2
13 fi
14
15 cpu_count=$(grep family /proc/cpuinfo | wc -l)
16 cpu_psf=$(($cpu_count - 1))
17
18 #move tasks to default cpuset
19 cd /dev/cpuset
20 for task in `cat sys/tasks`
21 do
22   echo $task > tasks 2> /dev/null
23 done
24 for task in `cat psf/tasks`
25 do
26   echo $task > tasks 2> /dev/null
27 done
28 #delete cpusets
29 rmdir /dev/cpuset/sys
30 rmdir /dev/cpuset/psf
31
32 echo ondemand | sudo tee /sys/devices/system/cpu/↩
     cpu$cpu_psf/cpufreq/scaling_governor
```

5.2 Systemvoraussetzungen

Für das Performance-Scoring-Framework wird ein Rechner mit mindestens 2 CPUs und einem Linux Betriebsystem vorausgesetzt sowie einer JRE oder JDK.

5.3 Anleitung

Zuerst müssen 2 exklusive Cpusets erstellt werden und alle momentan laufenden Tasks auf das eine Cpuset verschoben werden, damit ein CPU frei wird. Dies wird automatisch mit dem Skript **init_environment.sh** eingerichtet[13]. Im Terminal kann man den Prozess mit **echo $$ >/dev/cpuset/psf/tasks** auf für das Performance-Scoring-Framework reservierte Cpuset verschieben. Hier würde man die Kalibrierung aus dem Terminal starten, was nun ein Child-Prozess des Terminal-Prozesses ist und somit auch auf dem reservierten Cpuset läuft. Nach der Kalibrierung könnte man mit dem Skript **terminate_environement.sh** die Umgebung wieder beenden. Anschliessend muss die Library **libagentScore.so** in ein Verzeichnis kopiert werden, das dem Linker bekannt ist (Kapitel 3.2.1). Um den Score einer Methode nun zu ermitteln, muss das Skript **get_score.sh** gestartet werden. Das Skript fragt nach einer Klasse und nach einer Methode. Die angegebene Klasse muss eine Mainmethode besitzen, da die JVM eine benötigt [14]. Welche Bytecodes in der angegebenen Methode vorkommen, wird dann beim ersten Durchlaufen eben dieser Methode ermittelt. Für jeden vorkommenden Bytecode wird aus der Config.xml Datei der kalibrierte Wert geladen. Sollte der Wert nicht vorkommen, wird ein neuer Eintrag mit dem Wert 0 erstellt, der dem Programm sagt, dass der Wert nicht kalibriert ist und der Defaultwert (1) genommen werden sollte.

Listing 28: Beispiel einer fiktiven Config.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <bytecodeid>
3   <8>
4     <min>14</min>
5   </8>
6   <89>
7     <min>0</min>
8   </89>
9 </bytecodeid>
```

[13] Die Skripte sollten im Terminal gestartet werden, da man sonst mögliche Fehlermeldungen nicht sieht

[14] The Java virtual machine starts up by creating an initial class, which is specified in an implementation-dependent manner, using the bootstrap class loader (§5.3.1). The Java virtual machine then links the initial class, initializes it, and invokes its public class method void main(String[]). The invocation of this method drives all further execution. Java Virtual Machine Specification: http://docs.oracle.com/javase/specs/

In diesem Beispiel wurde der Bytecode mit der Id 8 vorher schon irgendwo definiert. Der Bytecode mit der Id 89 ist noch nie vorgekommen und wurde deshalb von dem Agent als unkalibriert markiert. Da man mit dem Agent keinen Rückwert geben kann, wird der Score in die Datei Score.txt gespeichert. Ist der Score eine Zahl (z.B. 785097), so ist der Wert nur durch schon kalibrierte Werte entstanden. Ein Score mit einem vorangestellten ˜(z.B. ˜764678) zeigt, dass es einige Bytecodes (unbekannte Objekte, sehr grosses Array ...)gab, die noch nie vorgekommen sind. Bei einem Score von 0 wurde die angegebene Methode nicht gefunden.

5.4 Ausblick

Man könnte versuchen eine Kalibrierung direkt in der JVM zu machen. Es müsste dann eine grössere Klasse mehrmals getestet werden. Der minimale Wert der Bytecodes müsste dann in die Config.xml gespeichert werden. Sollte der **agentScore** dann angeben, dass einige Werte nicht kalibriert sind, müsste mit dieser Klasse/Methode nochmals eine Kalibrierung gemacht werden, um die hinzugefügten unkalibrierten Einträge zu ersetzen.

Studienarbeit
Performance-Scoring-Framework
Glossar

Peter Camastral Daniel Marty

19. Dezember 2013

Änderungstabelle

Datum	Version	Änderung	Person
18.12.2013	1.0	Erstellung	d1marty

A

Agent: Ein Agent ist ein autonomes Gebilde, das eine gewisse Umgebung (in unserem Fall die JVM) beobachtet und bei gewissen Aktionen reagiert.

Array: Ein Array ist eine Liste aus gleichartigen Elementen.

B

Bash: Die "Bourne again Shell" ermöglicht eine Benutzung des Computersystems mittels Kommandos.

Bibliotheksfunktion: Eine Methode/Funktion aus einer Sammlung von Unterprogrammen.

Bytecode: Ein Befehl, der von der JVM abgearbeitet werden kann.

C

C++: Genormte, objektorientierte Programmiersprache. In unserem Fall wird C++11 eingesetzt.

Child-Prozess: Ein Prozess der von einem Parent-Prozess gestartet wurde.

Compiler: Der Compiler übersetzt Sourcecode in ein ausführbares Programm.

Constant Pool: Speicherort für Konstanten.

CPU: Die "Central Processing Unit" ist die Hardware, die Instruktionen ausführt.

Cycle: Maschinenzyklus der normalerweise mit dem CPU-Takt gleichläuft.

D

Debug: Werkzeug für die Diagnose von Programmen.

DRY: "Dont repeat yourself", keine Wiederholungen.

E

Emulator: Ein Programm das einen Computer nachbildet.

Event: Ein Ereignis, das bestimmte Behandlungsroutinen erfordern kann.

F

G

Gantt: Ein Balkenplan zur Planung von Aktivitäten.

Garbage Collection: Automatische Freigebung des Speichers.

H

Header: Schnittstellendeklaration für ein C++ Programm.

hooken: Fremden Programmcode in eine bestehende Anwendung integrieren.

Hotspot: Weitverbreitetste JVM.

I

Instanzierung: Das Erzeugen eines Objektes einer bestimmten Klasse.

Interpreter: Liest Quellcode ein und führt ihn direkt aus. Es wird keine ausführbare Datei erstellt im Gegensatz zum Compiler.

J

Java: Objektorientierte, plattformunabhängige Programmiersprache die auf der JVM ausgeführt wird.
JDK: Das "Java Development Kit" enthält die Java Laufzeitumgebung und Werkzeuge für die Entwicklung von Java.
JRE: Die "Java Runtime Environment" enthält nur die Java Laufzeitumgebung ohne Werkzeuge für die Entwicklung von Java.
JVM: "Die Java Virtual Machine" ist eine virtuelle Maschine, die den Bytecode ausführt.
JVMTI: Das "Java Virtual Machine Tool Interface" ist eine Schnittstelle um die JVM zu steuern.

K

Kernel: Der Kern eines Betriebsystems, der die unterste Softwareschicht bildet.
Kernelobjekt: Ein Speicherblock der bestimmte Datenstrukturen beinhaltet, der nur vom Kernel abgerufen werden kann.
Klasse: Bauplan für Attribute und Methoden eines Objektes.
kompilieren: Übersetzten des Programmcodes in eine ausführbare Datei.

L

Library: Eine Programmbibliothek die eine Sammlung von Unterprogrammen beinhaltet.
Linker: Verbindet einzelne Module zu einem ausführbaren Programm.
Linux: Ein freies, kostenloses Betriebsystem.
Linux Mint: Ein Linuxderivat aufbauend auf Ubuntu.

M

Mainmethode: Einstiegspunkt in eine Klasse.
Memory Allocation Pool: Dynamisch zugeteilter Speicher mit fester Blockgrösse.
Memory Node: Speicherknoten, entweder Arbeitsspeicher oder Grafikspeicher.
Monotonic Clock: Eine Uhr die stets aufwärts zählt.
mounten: Das Einbinden eines Dateisystems in das System.

N

O

Objekte: Instanzen einer Klasse
Opcode: ->Bytecode
OpenJDK: Die OpenSource implementation des Java Development Kit von Oracle
Operand Stack: Der interne Stack einer Virtuellen Maschine, auf dem Argumente für Operationen bereitgestellt werden.
Overhead: Mehraufwand der durch "unnötigen" Programmcode entsteht

P
Parent-Prozess: Prozess der einen oder mehrere Prozesse gestartet hat.
Pointer: Ein Zeiger, der auf eine Speicheradresse zeigt.
Polymorphie: Funktionen die abhängig vom übergebenen Objekttypen verschiedenen Code ausführen
primitiver Typ: Typen die direkt im Speicher abgebildet sind.
Program Counter: Ein inkrementierender Zähler der auch die Speicheradresse des aktuellen oder des nächsten Befehls enthält.
Prozess: Ein Vorgang der auf einem CPU abgearbeitet wird.
Prozesspriorität: Indikator für das System, welche Prozesse mehr Ausführungszeit erhalten.
Q
R
RDTSC: TimeStamp Counter der CPU, heute ungenau wegen automatischem anpassen der Taktfrequenz um Strom zu sparen.
Referenz: Ein Verweis auf ein Objekt im Speicher.
Rootrechte: Die weitreichendsten Zugriffsrechte in einem Betriebssystem.
S
Scalinggovernor: Regelt die Taktfrequenzstufen eines Prozessors.
Score: Resultat das anhand der Performance einer Methode ermittelt wird.
Skript: Programm das von einem anderen direkt interpretiert wird ohne auf Maschinencode übersetzt zu werden.
SoftIRQ: Ein Interrupt/eine Unterbrechung, die durch einen Software-Befehl ausgelöst wird.
String: Eine Kette aus mehreren Zeichen. Ein Text.
Symbol: Zusatzinformation die beim Kompilieren generiert wird um die Umkehrung zu erleichtern.
T
Task: ->Prozess
Terminal: Systemkonsole: Bedienung des Systems ohne grafische Oberfläche.
U
Ubuntu: Ein Linuxderivat der Firma Canonical.
Umgebungsvariable: Eine konfigurierbare Variable des Betriebsystems, die von mehreren Programmen benutzt werden kann und meistens Pfade darstellt.
V
W
Watchdog: Überwacht die Funktion anderer Komponenten.
X
XML: Die “Extensible Markup Language“ ist eine Sprache zur Darstellung hierarchischer Daten.
Y
Z

Studienarbeit
Performance-Scoring-Framework
Literaturverzeichnis

Peter Camastral Daniel Marty

18. Dezember 2013

Änderungstabelle

Datum	Version	Änderung	Person
18.12.2013	1.0	Erstellung	d1marty

Quellenangabe

[1] Debug-Version der Hotspot JVM http://download.java.net/jdk7/archive/b142/binaries/
[2] Bytcode Interpreter Cycle Count Debug Option http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html http://www.progdoc.de/papers/Jax2012/jax2012.html#(4)
[3] Infos zur Standard Profiling Library http://www.cs.cmu.edu/~pattis/15-1XX/15-200/handouts/profilerlab/
[4] Dokumentation zum JVMTI Agent http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#EventSection
[5] Singlestep Definition http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/0ed9a90f45e1/src/share/vm/interpreter/bytecodeInterpreter.cpp Zeilen 174-184.
[6] Manpages zu clock_gettime http://linux.die.net/man/3/clock_gettime
[7] Manpages zu java http://www.unix.com/man-page/all/1/java/
[8] Informationen zu nice http://wiki.ubuntuusers.de/nice
[9] Implementation von JVM Funktionen http://hg.openjdk.java.net/jdk7u/jdk7u/archive/tip.zip in /openjdk/hotspot/src/share/vm/prims/jvmtiEnv.cpp sowie /openjdk/hotspot/src/os/linux/vm/os_linux.cpp
[10] Auflistung der verschiedenen Bytecodes http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
[11] Implementation RDTSC http://stackoverflow.com/questions/11609026/algorithm-speed-tester-in-c-c
[12] Manpages zu taskset http://linuxcommand.org/man_pages/taskset1.html
[13] Informationen über Scalinggovernors http://wiki.ubuntuusers.de/Prozessortaktung
[14] Manpages zu cpuset http://man7.org/linux/man-pages/man7/cpuset.7.html
[15] Informationen zum Constant Pool http://en.wikipedia.org/wiki/Java_class_file#The_constant_pool
[16] Spezifikation der JVM http://docs.oracle.com/javase/specs/