JVMPI -
Das Java Virtual Machine Profiling Interface
Verteilte und Parallele Systeme

Seminararbeit von
Oliver Zilken und Philipp Wever

Fachhochschule Bonn-Rhein-Sieg
Studiengang Angewandte Informatik

Betreuender Dozent: Prof. Dr. Rudolf Berrendorf

Stand: 28.11.2003

Inhaltsverzeichnis

1 Einleitung und Motivation

Die Ausführungsgeschwindigkeit von Programmen ist in heutigen Tagen von großer Wichtigkeit. Gerade bei der Betrachtung von Java bemängeln Kritiker häufig Performance- und Geschwindigkeitsdefizite. Schlechte Antwortzeiten und träge grafische Benutzeroberflächen sind häufig das Resultat von ineffizientem Source-Code.

Zur Behebung bzw. Lokalisierung von Flaschenhälsen, schlechter Speicherausnutzung oder Ressourcenbelegung werden so genannte Profiler verwendet. Sie bieten eine komfortable Möglichkeit oben genannte Problemstellungen zu analysieren.

Diese Arbeit befasst sich mit der Anwendung des Profiling und stellt am Beispiel des Java Virtual Machine Profiler Interface (JVMPI) Framework die Funktionsweise und den Aufbau von Profiling-Tools dar. Des Weiteren werden verschiedene Beispiel-Profiler betrachtet.

2 Das JVMPI Framework

Seit der JDK Version 1.2 beinhaltet Java das Java Virtual Machine Profiler Interface (JVMPI). Das JVMPI bildet eine Schnittstelle innerhalb eines Prozessablaufs der Java Virtual Machine (JVM). Durch diese kann die JVM mit einem Profiler Agenten, auf den später noch näher eingegangen wird, kommunizieren. Das Framework bietet umfangreiche Möglichkeiten an, ein Profiling durchzuführen.

2.1 Aufbau

Die Implementierung der JVMPI-Schnittstelle basiert auf der Programmiersprache C. Bestandteil der Kommunikation zwischen JVM und Agent sind relevante Ereignisse, die während der Programmausführung auftreten. Es besteht die Möglichkeit, der JVM vorzugeben, welche Ereignisse an den Agenten weitergegeben werden sollen. Die folgende Abbildung stellt die Architektur der JVMPI Implementierung schematisch dar.

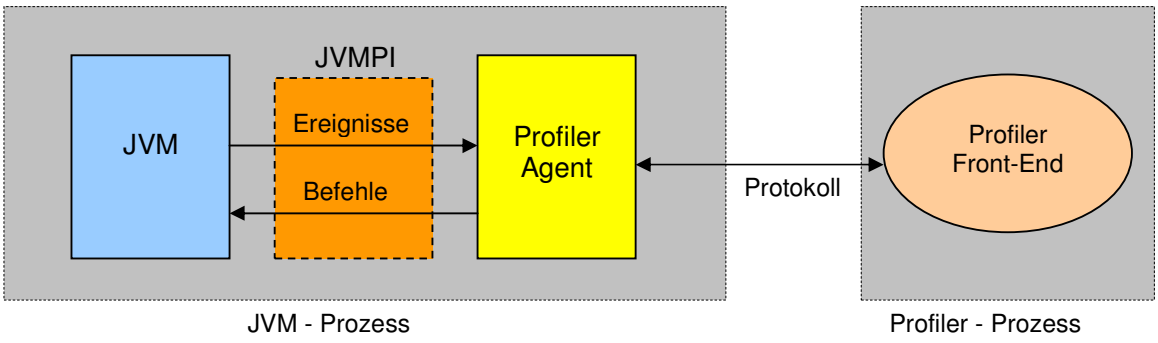

Abbildung 1: Architektur von JVMPI

Ein Profiler Front-End wird als Zusatzpaket von verschiedenen Herstellern angeboten. Es dient zur Visualisierung und komfortableren Bedienung der JVMPI-Schnittstelle. Die Ausführung des Front-End kann auf verschiedene Weise erfolgen. Zum einen kann es direkt im Prozess der JVM ausgeführt werden. Andererseits kann die Ausführung auch in einem eigenen, separat erzeugten Prozess stattfinden. Dies bietet wiederum die Möglichkeit das Front-End auf einem entfernten Rechner als verteile Anwendung zu betreiben.

Das Protokoll, welches zur Übertragung von Daten zwischen Profiler Agent und Front-End benötigt wird, ist nicht Bestandteil des JVMPI, sondern wird von den jeweiligen Front-End Paketen zur Verfügung gestellt.

2.2 Der Agent

Der Profiler Agent ist eine Bibliothek, die je nach Bedarf, beim starten der JVM zusätzlich aufgerufen und aktiviert werden kann. Die aktuelle Version des JVMPI unterstützt lediglich einen Agenten pro JVM. Da sowohl Agent als auch die JVM eine Reihe von Funktionen unterstützen, wird beim Start eine so genannte Function-Pointer Tabelle generiert. Diese besteht aus zwei Mengen von Funktionen. Die eine Menge beinhaltet Funktionen, die von der JVM unterstützt werden und die andere Menge beinhaltet die Funktionen, die vom Agenten bereitgestellt werden. In der Tabelle werden sämtliche Funktionen beider Seiten spezifiziert. Nun können Agent und JVM sich, über Funktionsaufrufe gegenseitig Anweisungen erteilen.

Der Aufruf des Profiler Agenten kann in zwei Wegen erfolgen. In beiden Fällen muss vorher ein Name für den Agenten festgelegt werden. Im Folgenden werden die beiden Methoden vorgestellt, wobei der Agent den Namen „myprofiler“ erhalten hat:

- Der Benutzer ruft den Agenten Mittels Kommandozeile auf. Hierbei wird er direkt beim Starten der JVM aktiviert. Es können so auch noch zusätzliche Optionen angegeben werden. Beispiel:

  ```
  java -Xrun myprofiler:heapdump=on,file=log.txt ToBeProfiledClass
  ```

  Die JVM durchsucht nun das Java-Standardverzeichnis für Bibliotheken nach dem angegebenen Agenten (für SPARC/Solaris z.B.: /lib/sparc/libmyprofiler.so). Sollte die Suche fehlschlagen, werden noch einige weitere, wichtige Systemverzeichnisse durchsucht.

  Bei erfolgreicher Suche wird der Profiler Agent geladen und muss dabei folgende Methode enthalten:

  ```
  JVMPI_Interface *jvmpi_interface;

  JNIEXPORT jint JNICALL JVM_OnLoad(JavaVM *jvm, char *options, void *reserved) {
      int res = (*jvm)->GetEnv(jvm, (void **)&jvmpi_interface, JVMPI_VERSION_1);
      if (res < 0) {
          return JNI_ERR;
      }
      ... /* Einträge im jvmpi_interface */
  }
  ```

  Bei dieser Funktion definiert das erste Argument einen Zeiger auf die JVM-Instanz, und das zweite Argument einen Zeiger auf Optionen für die Datenaufzeichnung. Das letzte Argument ist reserviert und wird auf NULL gesetzt. Durch den GetEnv – Aufruf wird der Agent bei der JVM als Profiler registriert.

  Bei erfolgreichem Methodenaufruf muss die Funktion JNI_OK zurückgeben. Bei fehlgeschlagener Initialisierung wird JNI_ERR zurückgegeben.

- Der Agent kann auch innerhalb des Java-Programms selbst aufgerufen werden. Bei dieser Methode wird er erst gestartet wenn folgender Aufruf getätigt wird:

  ```
  System.loadLibrary("myprofiler");
  ```

  Dabei muss in der Bibliothek des Agenten die Funktion

  ```
  JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *vm, void *reserved);
  ```

  zur Verfügung stehen.

2.3 Funktionen und Ereignisse

Um Funktionen aufzurufen und Ereignisse zu überwachen ist im Profiler Agent die Methode „NotifyEvent“ implementiert. Mit Hilfe dieser Funktion hat die JVM die Möglichkeit, Nachrichten über Ereignisse an den Agenten zu übergeben. Die Methode ist die einzige innerhalb des JVMPI-Interfaces, die im Agenten und nicht in der JVM implementiert ist und hat folgende Struktur:

```
void (*NotifyEvent)(JVMPI_Event *event);
```

Der Agent meldet die Ereignisse, die er mitgeteilt bekommen möchte, mit Hilfe der Funktionen „EnableEvent“ bei der JVM an. Über die Funktion „DisableEvent“ können die Ereignisse wieder abgemeldet werden. Standardmäßig sind alle Mitteilungen deaktiviert.

Jedes Java-Element (Threads, Klassen, Methoden, Objekte, heap arenas und JNI globale Referenzen) hat eine Identifikationskennung (ID), die systemweit einzigartig ist. Die folgende Tabelle zeigt die ID-Form der einzelnen Elemente, deren Datentyp und Gültigkeitsbereiche:

ID	Datentyp	Anfang der Gültigkeit	Ende der Gültigkeit
thread ID	JNIEnv *	thread_start	thread_end
object ID	jobjectID	object_alloc	object_free, object_move, arena_delete
class ID	jobjectID	class_load	class_unload, object_move
method ID	jmethodID	defining_class_load	defining_class_unload
arena ID	jint	arena_new	arena_delete
JNI global ref ID	jobject	global_ref_alloc	global_ref_free

Tabelle 1: Identifikationskennungen von Java-Elementen

Es folgt eine Tabelle mit den wichtigsten Ereignissen, die von JVMPI unterstützt werden. Es werden hier hauptsächlich die Ereignisse aufgeführt, die zum Verständnis der Gültigkeitsbereiche beitragen. Die vollständige Liste kann man in der JVMPI – Beschreibung von Sun einsehen.

Ereignis	Beschreibung
THREAD_START, THREAD_END	Werden generiert, wenn ein Thread startet oder stoppt.
OBJECT_ALLOC, OBJECT_FREE	Werden generiert, wenn Objekte erzeugt oder gelöscht werden.
CLASS_LOAD, CLASS_UNLOAD	Werden generiert, wenn Klassen geladen oder entladen werden.
METHOD_ENTRY, METHOD_EXIT	Werden generiert, wenn die JVM einen Methodenaufruf startet, bzw. wenn die Methode verlassen wird.
ARENA_NEW, ARENA_DELETE	Werden generiert, wenn eine Heap Arena erstellt oder gelöscht wird.
JVM_INIT, JVM_SHUTDOWN	Werden generiert, um mitzuteilen, ob die JVM vollständig gestartet bzw. beendet wurde.

Tabelle 2: JVMPI Ereignisse

3 Beispielimplementation

Um die grundlegende Funktionsweise, den Ablauf und Aufbau eines Profiling zu veranschaulichen, wird im Folgenden ein kurzes Testprogramm analysiert. Die Untersuchung erfolgte auf dem Betriebssystem Windows XP.

3.1 Der Source-Code

Das Beispielprogramm, das es zu analysieren gilt, besteht aus zwei Java-Klassen. Die Klasse JCalc.java beinhaltet eine einzige, aber rechenintensive Funktion. Diese Funktion befasst sich mit der Berechnung von Fließkommazahlen in der komplexen Ebene (Mandelbrotmenge). Die Klasse JTest.java dient als Testprogramm, welches die Kalkulationsfunktion aufruft und ihr zur Berechnung relevante Parameter übergibt. Außerdem wird durch diese Klasse die Zeit gemessen, die zur Programmausführung benötigt wurde.

Klasse JCalc.java

```
/* ================================================================================
Beschreibung: Klasse zur Berechnung von Punkten in der komplexen Ebene.
@author: Philipp Wever
@version: 27.11.2003
================================================================================ */

public class JCalc {
      // Klassenvariablen deklarieren
      float xmin, ymin, xmax, ymax;
      int maxiter;

      // Konstruktor der Klasse JTest
      public JCalc(float xmin, float ymin, float xmax, float ymax, int
                                                                        maxiter) {
            this.xmin = xmin;
            this.ymin = ymin;
            this.xmax = xmax;
            this.ymax = ymax;
            this.maxiter = maxiter;
      }

/* ================================================================================ */
/* Methode zur Berechnung der Punkte */

      public void calc() {
            int i, j, k;
            float absvalue, temp;
            float c_real, c_imag, z_real, z_imag, dx, dy;

            // Auflösung vorgeben
            int x_res = 10;
            int y_res = 10;

            dx = (xmax - xmin) / x_res;
            dy = (ymax - ymin) / y_res;

            for(i = 0; i < x_res; i++) {
                  for(j = 0; j < y_res; j++) {
                        /* map point to window */
                        c_real = xmin + i * dx;
                        c_imag = ymin + j * dy;
                        z_real = z_imag = 0;
                        k = 0;
```

```
                        do  {
                          temp = z_real*z_real - z_imag*z_imag + c_real;
                          z_imag = 2*z_real*z_imag + c_imag;
                          z_real = temp;
                          absvalue = z_real*z_real+z_imag*z_imag;
                          k++;
                        } while (absvalue < 4.0 && k < maxiter);

                        System.out.print(absvalue+" ");
                  }
            }
      }
}
/* ============================  EOF  ================================== */
```

Klasse JTest.java

```
/* ============================================================================
Beschreibung: Testprogramm zum Aufruf der Funktion calc().
@author: Philipp Wever
@version: 27.11.2003
============================================================================ */

public class JTest {
      public static void main(String[] args) {
            // Variablen deklarieren
            float a, b, c, d;
            double t1, t2, t;
            a = b = -2;
            c = d = 2;

            // Objekt der Klasse JTest erzeugen
            JCalc j = new JCalc(a, b, c, d, 1000);

            /****************** Analyse *****************/
            //Systemzeit vorher
            t1=System.currentTimeMillis();

            // Methode zur Berechnung aufrufen
            System.out.println("\n\nBerechnug gestartet...");
            j.calc();
            System.out.println();

            //Systemzeit nachher
            t2=System.currentTimeMillis();

            // Benötigte Zeit augeben
            System.out.println("---------------------------------------");
            System.out.println("Calculation took "+(t2-t1)+" ms.");
      }
}
/* ============================  EOF  ================================== */
```

3.2 Der Profiler

Wie schon erwähnt, ist der Profiler-Agent in C programmiert. Im folgenden Code sind die grundlegenden Eigenschaften implementiert, die jeder Profiler erfüllen muss.

C-Code myprofiler.cc

```
/* ========================================================================
Beschreibung: Programm zur Erstellung des Profiler-Agenten.
@author: Philipp Wever
@version: 27.11.2003
======================================================================== */

#include <jvmpi.h>

// Globaler Zeiger auf JVMPI-Interface
static JVMPI_Interface *jvmpi_interface;

// Funktion zur Behandlung von Ereignisnachrichten
void notifyEvent(JVMPI_Event *event) {
  switch(event->event_type) {
  case JVMPI_EVENT_CLASS_LOAD:
    fprintf(stderr, "myprofiler> Class Load : %s\n", event-
                                              >u.class_load.class_name);
    break;
  }
}

// Eintrittspunkt des Profiler-Agenten
extern "C" {
  JNIEXPORT jint JNICALL JVM_OnLoad(JavaVM *jvm, char *options, void
                                                        *reserved) {
    fprintf(stderr, "myprofiler> initializing ..... \n");

    // JVMPI-Interface Zeiger holen
    if ((jvm->GetEnv((void **)&jvmpi_interface, JVMPI_VERSION_1)) < 0) {
      fprintf(stderr, "myprofiler> error in obtaining jvmpi interface
                                                        pointer\n");
      return JNI_ERR;
    }

    // JVMPI-Interface initialisieren
    jvmpi_interface->NotifyEvent = notifyEvent;

    // Aktivieren der Ereignisse (Hier: CLASS_LOAD)
    jvmpi_interface->EnableEvent(JVMPI_EVENT_CLASS_LOAD, NULL);

    // Nachricht bei erfolgreicher Initialisierung
    fprintf(stderr, "myprofiler> .... ok \n\n");
    return JNI_OK;
  }
}
/* ============================  EOF  ================================== */
```

Als erstes wird ein Zeiger, der auf das JVMPI-Interface verweist, erzeugt. Direkt danach wird die schon erwähnte Funktion notifyEvent() definiert, die zum Laden von Funktionen und Überwachen von Ereignissen verantwortlich ist.

Danach erfolgt die Sektion, die nach dem Aufruf des Profiler-Agenten relevant wird. Beim Laden des Profilers verlangt die JVM die Methode JVM_OnLoad, die die JVM-Instanz

referenziert. Wichtig zu erwähnen wäre noch die Funktion EnableEvent(), die in diesem Beispiel lediglich das Ereignis des Ladens einer Klasse aktiviert.

3.3 Das Ergebnis

Nachdem durch die Kompilierung des Profiler-Codes eine Systembibliothek namens „myprofiler.dll“ erzeugt wurde, konnte sie mit der Java-Option „-Xrun“ aufgerufen werden. Der Aufruf java –Xrunmyprofiler JTest ergab folgende Ausgabe (Da beim Start eines Java-Programms sehr viele Standardklassen aufgerufen werden, wurde die Ausgabe hier gekürzt und nur relevante Teile werden gezeigt):

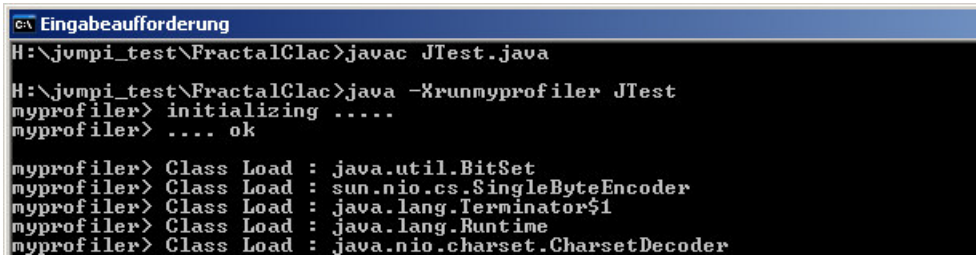

Abbildung 2: Start des Profiling mit Initialisierung und Profiler-Überprüfung

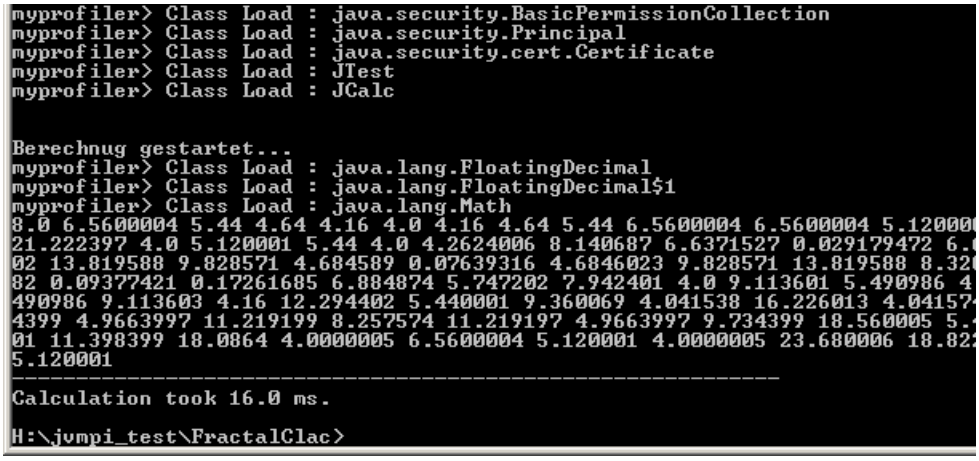

Abbildung 3: Ausgabe am Ende des Profiling

Aus Abbildung 2 geht hervor, dass die Initialisierung des Profilers erfolgreich war. Abbildung 3 zeigt den Berechnungsvorgang und es wird ersichtlich, welche zusätzlichen Klassen beim Aufruf von JCalc geladen wurden.

Bei der Ausführung des Programms ohne Aktivierung des Profilers ist auffällig, dass die Ausführungszeit erheblich geringer ist. Das ist das Resultat aus der Tatsache, dass das Programm durch den Vorgang der Protokollierung der Ereignisse, die der Profiler durchführt, sehr oft unterbrochen wird. Dieses Phänomen stellt gerade bei der Analyse umfangreicherer Softwareimplementierungen ein Problem dar.

4 Der einfache JVMPI-Profiler: HPROF

4.1 Allgemeines

Seit der Version 1.3 des JDK gibt es schon einen eingebauten Profiler, der auf der JVMPI-API basiert: Der Profiler hprof.

In ihm sind zwar nur grundlegende Funktionen des Profiling implementiert, er reicht aber durchaus aus, um ein erstes Profiling durchführen zu können. Er besitzt keine grafische Oberfläche; die Ergebnisse des Profiling mit hprof werden in eine Datei ausgegeben. Da in dieser Datei alle Statistiken und Traces enthalten sind, benötigt es jedoch immer einen gewissen Aufwand, die Text-Datei so nachzubearbeiten, dass man eine übersichtliche Auflistung der wichtigsten Parameter bekommt.

Trotz allem ist hprof eine hilfreiche Möglichkeit, um mit relativ geringem Aufwand Engpässe der Applikation im CPU- und Speichermanagement zu erkennen.

4.2 Syntax

Der Profiler hprof ist über die Kommandozeilen beim Ausführen der Java-Applikation zu steuern. Die Syntax lautet:

```
java –Xrunhprof:[<option>=<wert>,]*  MeineKlasse
```

Beispiel: `java –Xrunhprof:cpu=samples,heap=sites,depth=6 MyClass`

Als Optionen können folgende Möglichkeiten gewählt werden:

Option	**mögliche Werte**	**Default-Wert**
help	-	-
file	<Dateiname der Ausgabedatei>	java.hprof.txt
cpu	samples | times | old | off	off
heap	dump | sites | all	all
depth	<Tiefe des Stack trace>	4
monitor	y | n	n
thread	y | n	n
doe	y | n	y
format	a | b	a
net	<host>:<port>	<in lokale Datei schreiben>
cutoff	<Wert>	0.0001
lineno	y | n	y
gc_okay	y | n	y

Tabelle 3: Die Syntax von hprof

Die einzelnen Optionen haben hierbei folgende Bedeutung:

help

Diese Option wird ohne Wert aufgerufen (:help). Bei ihrer Benutzung läuft die Java-Applikation nicht ab; der Profiler tritt nicht in Kraft. Stattdessen wird ein kurzer Hilfetext auf die Standartausgabe gegeben, in der die Benutzung von hprof und die wichtigsten Optionen beschrieben sind.

file

Mit dieser Option kann man den Dateinamen der auszugebenden Text-Datei bestimmen. Hierbei sind sämtlich Namen im lokalen Dateisystem möglich.

Der Standardwert dieser Option ist den Dateinamen "java.hprof.txt" gesetzt. Die Datei wird dann im gleichen Verzeichnis angelegt, in dem sich die ausgeführte .class Datei befindet. In dieser Ausgabdatei befinden sich immer folgende Informationen:

- Ein kurzes Glossar der möglichen Optionen
- Die Auflistung alles Thread mit der Auflistung ihres Starts und ihrer Beendigung
- Die TRACEs. Hierbei wird zu jeder in der Applikation aufgerufenen Methode ihr StackTrace angegeben.

cpu

cpu=samples

Mit dieser Option kann die Auslastung der CPU festgestellt werden. Hierbei wird in sehr kurzen Zeitabständen die momentane Beschäftigung des Prozessors für jeden Thread abgefragt und protokolliert. In der Ausgabedatei wird dann später eine Tabelle erzeugt. In dieser wird jede während dem Ablauf der Applikation benutzte Methode mit der relativen Prozessor - Benutzungshäufigkeit aufgeführt.

Ebenso wird angegeben, wie oft jede Methode im Prozessor bearbeitet wurde und in welchem TRACE der Ausgabe-Datei die Methode zu finden ist.

Der Übersichtlichkeit halber wird die Tabelle nach Aufrufhäufigkeit geordnet.

cpu=times

Diese Option funktioniert ähnlich der cpu=samples Option. Der Unterschied besteht darin, dass hier nicht die Anzahl der Methodenaufrufe protokolliert werden, sondern nur die tatsächliche Zeit, die jede Methode im Prozessor berechnet wurde.

Das Benutzen der Option cpu kann zu einer signifikanten Verlangsamung der Applikation führen. Dies ist jedoch notwendig, um die exakte Prozessorauslastung bestimmen zu können.

heap

heap=sites

Mit dieser Option kann man die Benutzung des Java - heaps festgestellt werden. Hierbei wird in sehr kurzen Abständen das momentan erste Objekt des heaps abgefragt und protokolliert.

In der Ausgabedatei wird dann später eine Tabelle erzeugt. In dieser werden die verschiedenen benutzten Objekte, nach Häufigkeit geordnet, aufgelistet. In der Tabelle stehen zu jedem Objekt neben der relativen Häufigkeit noch einige andere Informationen:

Der für dieses Objekt allozierte Speicherplatz mit der Anzahl der darin enthaltenen Objekte

Der für dieses Objekt tatsächlich benutzte Speicherplatz mit der Anzahl der darin enthaltenen, tatsächlich benutzten Objekte

Eine Klassizifierung der Objekte. “[I“ bedeutet hierbei den Datentyp Integer, “[S“ den Datentyp String usw.

Heap=dump

Mit dieser Option wird ein momentaner Speicherauszug des Java-Heaps abgefragt und in der Ausgabe-Datei protokolliert. In diesem wird jedes einzelne, im Heap befindliche Objekt aufgeführt. Hierbei gibt es folgende Klassizifierungen:

ROOT	vom root des Garbage Collectors festgelegt.
CLS	Java – Klasse
OBJ	Instanz einer Java – Klasse
ARR	Array

depth

Mit dieser Option wird die maximale Tiefe des StackTrace bestimmt, der bei jeden TRACE angegeben ist. Dies ist nützlich, um die tatsächliche Herkunft einer ausgeführten Methode heraus zu finden.

monitor

Wenn diese Option aktiviert ist, wird zusätzlich protokolliert, welchem Thread vom Monitor Prozessorzeit gewährt wurde.

thread

Wenn diese Option aktiviert ist, dann werden die TRACEs nach den abgelaufenen Threads differenziert angezeigt.

doe

Diese Option bedeutet “Dump on Exit“. Wenn sie deaktiviert ist, dann wird nach Beendigung des Programmablaufs keine Information mehr in der Ausgabe – Datei abgelegt.

format

Mit dieser Option kann das Dateiformat der Ausgabedatei festgelegt werden. Es werden folgende Möglichkeiten angeboten:

a : Ausgabe im ASCII - Format

b : Ausgabe im Binär – Format. Dieses Format kann nützlich sein, wenn man die Profiling – Informationen an ein anderes Programm weiterleitet, z.B. HAT (http://java.sun.com/people/billf/heap).

net

Mit dieser Option ist es möglich, die Ausgabedatei über eine Socket - Verbindung an einen beliebigen anderen Rechner zu senden.

cutoff

Mit dieser Option, die nur eine untergeordnete Rolle spielt, kann der Anhalte-Wert des Profilings geändert werden.

lineno

Wenn diese Option deaktiviert ist, werden in den TRACEs keine Zeilennummern mehr angezeigt.

gc_okay

Wenn diese Option deaktiviert ist, dann wird während des Profilings der Garbage Collector nicht aufgerufen.

4.3 Beispiel-Ausgabe

Zur Erzeugung der Beispiel Ausgabe wurde folgendes Programmbeispiel verwendet:

Klasse SimpleProfilingTest.java

```
/* ================================================================================
 * Class SimpleProfilingTest
 * Used for profiling.
 * @Author: Oliver Zilken
 * @date: Oct 24, 2003
 * Created in project: VUPSII_JVMPI
================================================================================ */

public class SimpleProfilingTest {

      private static final int iterations = 10000;
      private static final int doubleIterations=iterations*1000;

      public static String generateString() {
            String testString = "";
            for(int i = 0; i < iterations; ++i) {
                  testString += "X";
            }
            return testString;
      }

      public static double generateDouble() {
            double testDouble = 1e25D;
            for (int i = 0; i <doubleIterations; i++) {
                  testDouble /= (1.27435634e136D / 1.48973456342786e135D);
            }
            return testDouble;
      }

      public static void main(String[] args) {
            double time1, time2,time3;
            time1 = System.currentTimeMillis();
            String string = generateString();
            time2 = System.currentTimeMillis();
            double d = generateDouble();
            time3 = System.currentTimeMillis();
            System.out.println(iterations+" strings were computed in
                                                "+(time2-time1)+" ms.");
            System.out.println(doubleIterations+" doubles were computed in
                                                "+(time3-time2)+" ms.");
            System.out.println("Total = "+(time3-time1)+"ms.");
      }
}
/* ============================  EOF  ================================== */
```

Die o.g. Beispiel – Klasse wird mit hprof mit folgenden Optionen ausgeführt:

java -Xint -Xrunhprof:cpu=samples,heap=sites,depth=5 SimpleProfilingTest

Die Option -Xint ist notwendig, um das Programm im Interpreter-Modus laufen zu lassen, um mögliche Verfälschungen bei der Arbeit der Profilers durch den Hotspot – Compiler vermeiden. Das Programm erzeugt dann folgende Ausgabe auf stdout:

```
10000 strings were computed in 2291.0 ms.
  10000000 doubles were computed in 197.0 ms.
  Total = 2488.0ms.
  Dumping allocation sites ... CPU usage by sampling running
       threads ... done.
```

In der Ausgabedatei java.hprof.txt kann man nun folgende Einzelheiten erkennen (Auszüge):

Thread – informationen

```
THREAD START (obj=809e4e8, id = 3, name="main", group="main")
[...]
THREAD END (id = 3)
THREAD START (obj=81a7618, id = 6, name="DestroyJavaVM", group="main")
THREAD END (id = 6)
```

Es werden 2 Threads benutzt:

- Der erste Thread hat die id=3. In ihm fndet der gesamte Programmablauf statt.
- Der zweite Thread wird zur Beendigung der JVM benötigt.

Heap - Informationen

```
SITES BEGIN (ordered by live bytes) Fri Oct 24 12:44:22 2003
          percent          live       alloc'ed  stack class
 rank self  accum    bytes  objs  bytes   objs  trace  name
1 79.65% 79.65%  840168  42  100460000 10000    229   [C
2  5.26% 84.91%  55488  371  56600        394        1   [C
3  4.35% 89.26%  45896  218  45912     219        1   [B
4  1.64% 90.89%  17256  319  17256     319        1   java.lang.Object
5  1.41% 92.30%  14856  291  14856     291        1   [S
6  1.30% 93.60%  13664  244  13664     244        1   java.lang.Class
7  1.06% 94.66%  11232  213  601440    225        1   [I
8  0.84% 95.50%   8832  368  9384      391        1   java.lang.String
[...]
106  0.01% 98.74% 112     1  112        1       244   [C
SITES END
```

Anhand dieser heap-Tabellen kann man erkennen, dass ca. 80 % des gesamten Java – Heaps von einer Methode belegt wird, die ein Char-Array erzeugt. Im dazugehörigen TRACE 229 steht:

```
TRACE 229:
      java.lang.StringBuffer.<init>(StringBuffer.java:115)
      java.lang.StringBuffer.<init>(StringBuffer.java:130)
      SimpleProfilingTest.generateString(SimpleProfilingTest.java:18)
      SimpleProfilingTest.main(SimpleProfilingTest.java:36)
```

Hieran sieht man, dass diese 80% allein durch die Zeile " testString += "X"; " in Zeile 36 der Datei "SimpleProfilingTest.java" belegt wurden.

Es wurden 10000 Objekte alloziert, die insgesamt ca.100 MB Speicher beanspruchen.

Tatsächlich am Leben sind aber nur 42 Objekt mit einem Speicherbedarf von 800 kb.

Dies zeigt, das bei der Benutzung des "+=" - Operators viele unnötige Objekte erzeugt wurden und hier ein Ansatzpunkt für eine mögliche Optimierung ist (z.B. Durch Verwendung eines StringBuffers).

CPU - Informationen

```
CPU SAMPLES BEGIN (total = 95) Sat Oct 25 14:35:53 2003
rank   self  accum   count trace method
   1 47.37% 47.37%      45   226 java.lang.StringBuffer.<init>
   2 26.32% 73.68%      25   231 SimpleProfilingTest.generateDouble
   3 12.63% 86.32%      12   225 SimpleProfilingTest.generateString
   4  6.32% 92.63%       6   227 java.lang.StringBuffer.toString
   5  3.16% 95.79%       3   229 java.lang.System.arraycopy
   6  2.11% 97.89%       2   232 SimpleProfilingTest.generateDouble
   7  1.05% 98.95%       1   228 java.lang.StringBuffer.<init>
   8  1.05% 100.00%      1   230 java.lang.StringBuffer.append
CPU SAMPLES END
```

Bei der Betrachtung der CPU- Zuteilungszeiten sieht man, daß ca. 75% aller Samples in String- verarbeitende Methoden fielen. Die restlichen 25% fielen in die Verarbeitung von double – Zahlen.

Auch diese Fakten geben dem Optimierer einen Hinweis, dass bei der Verarbeitung von Strings im Programm ein großes Optimierungspotential bestehen kann.

5 Einfacher Profiler mit Front-End: PerfAnal

5.1 Allgemeines

Um die unübersichtliche Ausgabe von hprof strukturiert und einfach analysieren zu können, wurde PerfAnal entwickelt. Es besteht aus einer GUI, welche die Ausgabedatei von hprof einliest und die darin enthaltenen Informationen übersichtlich darstellt.

PerfAnal ist ein sinnvolles Werkzeug, um sich die Ergebnisse des Profilings mit hprof übersichtlich anzeigen zu lassen.

Die GUI bietet jedoch keine eigenen Profiling – Funktionalitäten und ist deshalb, genau wie hprof, nur bei einfachen Profiling – Aufgaben wirklich hilfreich.

5.2 Beispiel-Ausgabe

Wenn man die Ausgabedatei des oben durchgeführten hprof - Profilings als Argument an PerfAnal weiterleitet, so sieht die GUI von PerfAnal folgendermaßen aus:

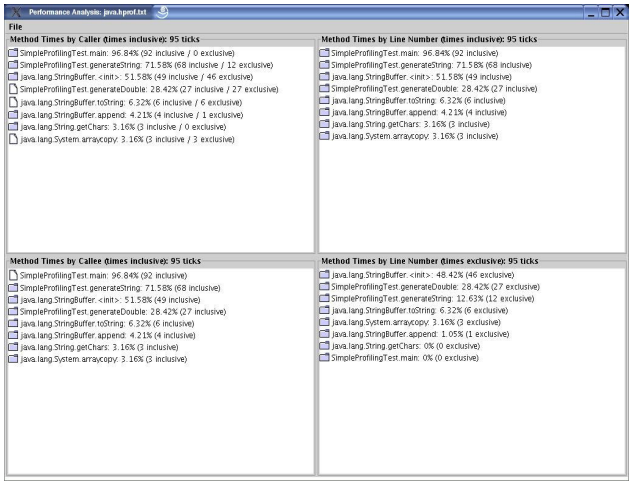

Abbildung 4: Die Oberfläche von PerfAnal

Die GUI ist in vier Bereiche gegliedert, die jede einen anderen Perspektive auf die Ergebnisse des Profiling liefern und jeweils mit dem Jtree – Interface präsentiert sind.

1. Quadrant: Oben Links – Method times by Caller

In diesem Bereich sind die Anteile der Prozessorauslastung durch die aufrufenden Methoden dargestellt.

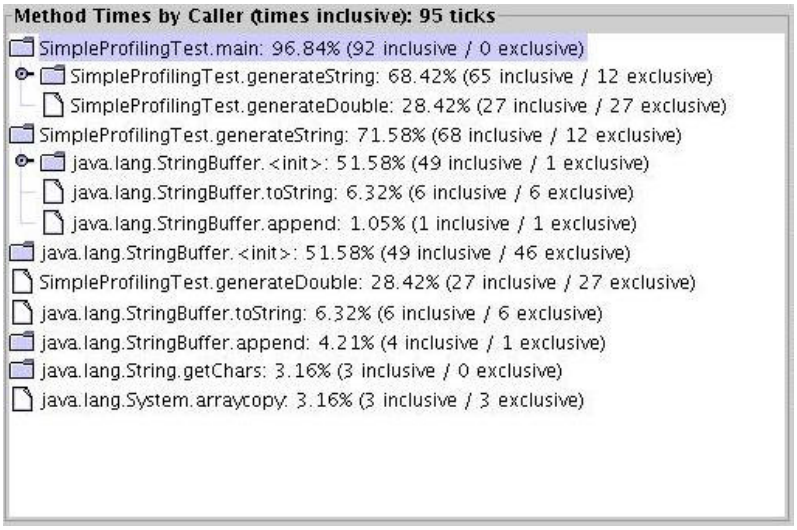

Abbildung 5: Das obere linke Fenster der Oberfläche von PerfAnal

Wie erwartet verbraucht die main – Methode die meiste Rechenzeit (ca.97%). Davon verbraucht die Subroutine generateString 68%, die Subroutine generateDouble 28%. Es wird auffällig, das innerhalb der generateString Subroutine nur 1% der Rechenzeit zum eigentlichen Ausführen des Programmablaufes, StringBuffer.append, verwendet wird, während für das wiederholte Initialisieren des Stringbuffers, Stringbuffer.<init>, 50 mal so viel Rechenzeit verbraucht wird.

2. Quadrant: Unten Links – Method times by Callee

In diesem Bereich sind die Zeiten der Prozessorauslastung durch die aufgerufenen Methoden dargestellt.

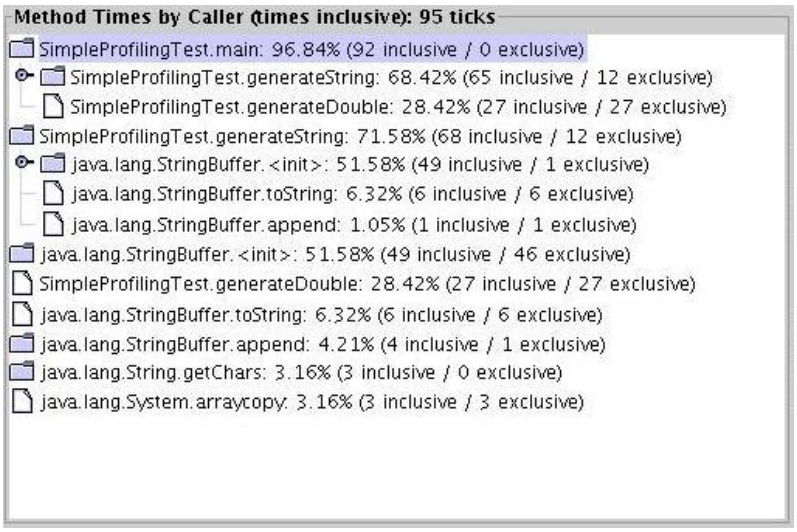

Abbildung 6: Das untere linke Fenster der Oberfläche von PerfAnal

Dieser Bereich bietet in diesem Beispiel keine neuen Erkenntnisse über den Programmverlauf.

3. Quadrant: Oben Rechts – Method times by Line Number (times inclusive)

In diesem Bereich werden erneut die Anteile der Prozessorauslastung durch die aufgerufenen Methoden dargestellt, nur diesmal nach Zeilennummern geordnet.

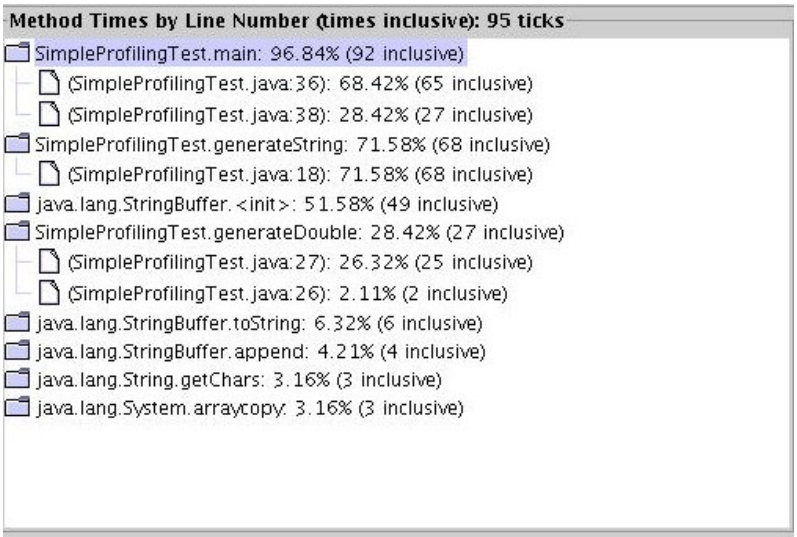

Abbildung 7: Das obere rechte Fenster der Oberfläche von PerfAnal

Hieran sieht man, dass ca. 97% der gesamten Prozessorauslastung allein durch zwei Zeilen ausgelöst wurde:

- 68% durch die Zeile 36: "`String string = generateString();`"
- 28% durch die Zeile 38: "`double d = generateDouble();`"

4. Quadrant: Unten Rechts – Method times by Line Number (times exclusive)

In diesem Bereich wird angezeigt, wo genau der Prozessor wirklich ausgelastet wird. Diese Ansicht zeigt nur die tatsächlich benutzten Programmzeilen, keine Subroutinen.

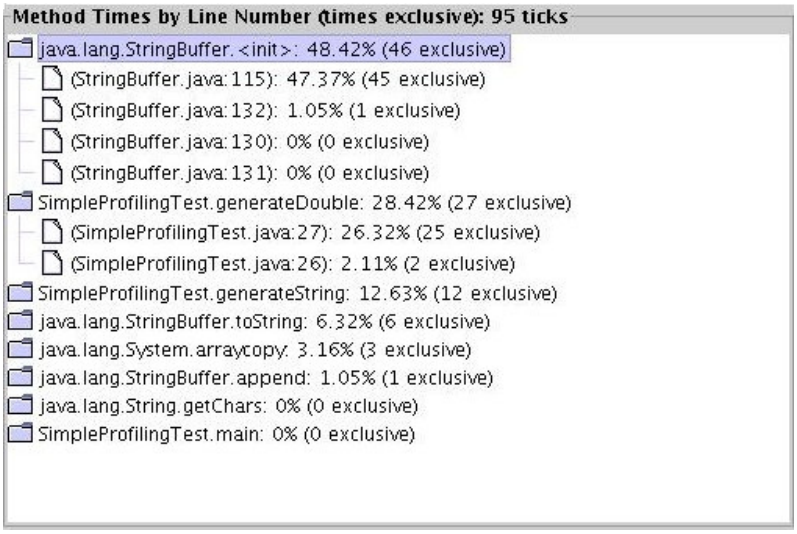

Abbildung 8: Das untere rechte Fenster der Oberfläche von PerfAnal

Hier sieht man, dass ca. 48% aller Berechnungen nicht im Beispielprogramm stattfinden, sondern in der JDK-Klasse StringBuffer.

5.3 Weitere Möglichkeiten

Mit einem Rechtklick kann auf der GUI ein Popup-Menu aufgerufen werden, in dem 3 Möglichkeiten zur Wahl stehen:

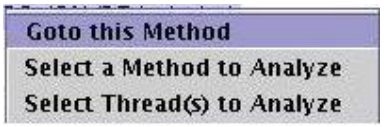

Abbildung 9: Das Methoden-Popup-Menü von PerfAnal

Goto this method

Mit dieser Option kann man sich die Position einer bestimmen Methode in allen 4 Quadranten anzeigen lassen.

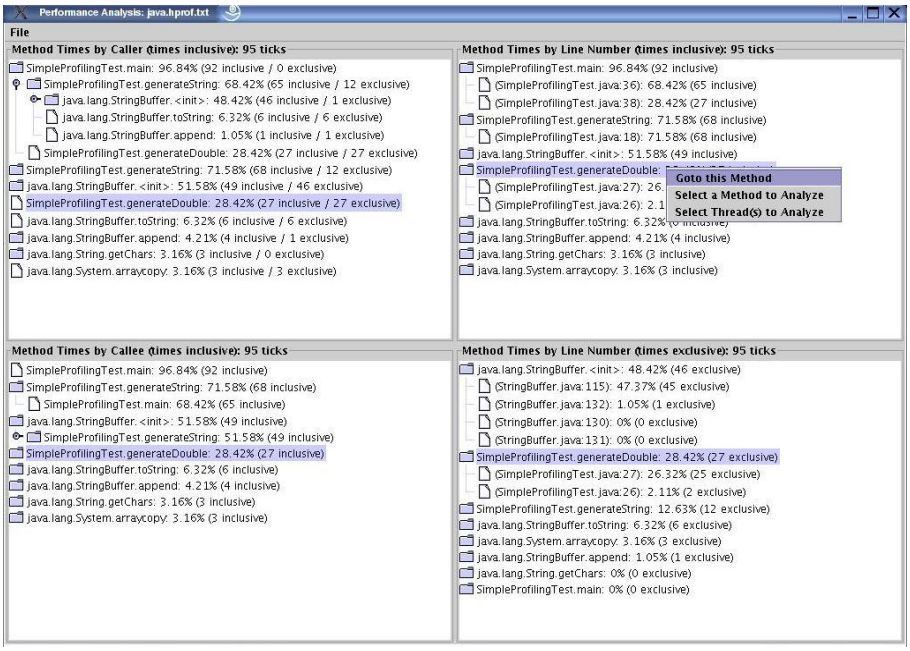

Abbildung 10: Die Option “Goto this Method“ in PerfAnal

Select a method to analyze

Mit dieser Option kann man die Jtrees der Quadranten aus der Sicht der ausgewählten Methode anordnen lassen.

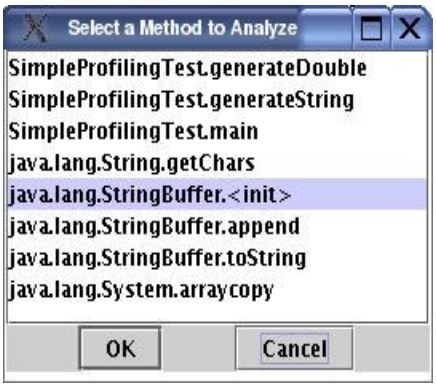

Abbildung 11: Die Option “Select a Method…“ in PerfAnal

Select Thread(s) to analyze

Mit dieser Option kann man die GUI sich neu aufbauen lassen, wobei nur die angewählten Threads, in der die betreffende Method abgelaufen ist, in die GUI einbezogen werden.

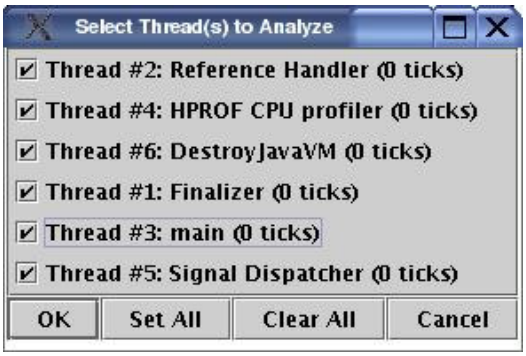

Abbildung 12: Die Option “Select Thread(s)...“ in PerfAnal

6 Komplexer Profiler: Borland®OptimizeIt™

OptimizeIt™ ist eine kommerzielle Software der Firma Borland®, die auf dem JVMPI basiert. Es soll hier der Vollständigkeit halber kurz vorgestellt werden.

Da man eine kostenpflichtige Lizenz benötigt, um alle Funktionalitäten des Programms nutzen zu können, kann das Programm hier nur anhand einer einfachen Heap - Statistik demonstriert werden.

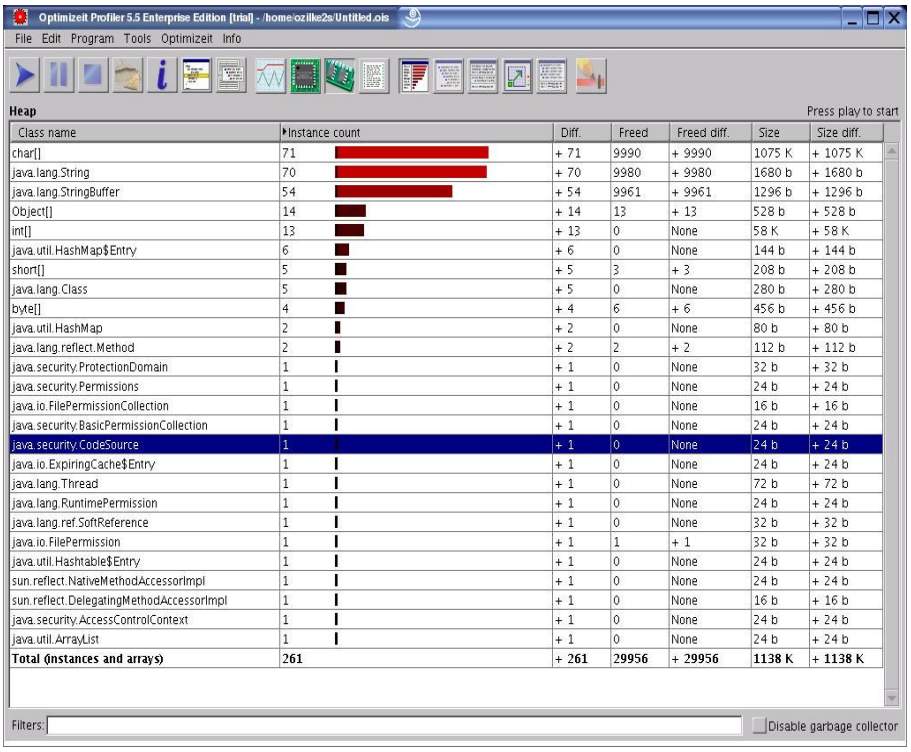

Abbildung 13: Die Oberfläche von Borland® OptimizeIt™

Borland® OptimizeIt™ bietet viele zusätzliche, nützliche Funktionen zum Profiling von Java – Applikationen. Außerdem enthält es einen Code - Coverage Tester, einen Thread - Debugger und einen Progress Tracker, mit dem man verschiedene Profiling – Ergebnisse miteinander vergleichen kann.

7 Literaturverzeichnis

(1) Shirazi, Jack „ Java Performance Tuning “ , 2000, O’Reilly

(2) Krüger, Guido „ GoTo Java 2 – Handbuch der Java-Programmierung “ , 2001, Addison Wesley

(3) „JSR-000163 Java™ Platform Profiling Architecture “, http://www.jcp.org/aboutJava/communityprocess/review/jsr163/

(4) „ JSR-163: Java™ Platform Profiling Architecture Public Review ”, API-Entwurf von

(5) The Java Virtual Machine Profiler Interface (JVMPI) http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html

(5) hprof Tech Tips, http://developer.java.sun.com/developer/TechTips/2000/tt0124.html

(6) PerfAnal: A java Performance Profiling Tool, http://developer.java.sun.com/developer/technicalArticles/GUI/perfanal/

(7) Borland® OptimizeIt™, kommerzielle Profiling Software, http://www.borland.com/optimizeit/

(8) Jprof Profiler Homepage http://starship.python.net/crew/garyp/jProf.html