HPJMeter –

Leistungsanalyse von Java Anwendungen

Verteilte und parallele Systeme

Seminararbeit von

Björn Adam
Michael Adams

Fachhochschule Bonn-Rhein-Sieg
Angewandte Informatik

Betreuender Dozent: Prof. Dr. Rudolf Berrendorf

Stand: 12.01.2004

Inhalt

Einleitung

Diese Seminararbeit ist im Rahmen der Vorlesung „Verteilte und parallele Systeme 2“ der Fachhochschule Bonn-Rhein-Sieg entstanden.
Sie ist eine Vorraussetzung zum bestehen der Fachprüfung.

Das Thema dieser Seminararbeit ist die Leistungsanalyse von Java Anwendungen mit Hilfe des HPJMeter.

Wir haben uns dieses Seminarthema ausgesucht, weil wir denken dass der Aspekt der Performance in Java Anwendungen immer wichtiger wird. Träge Benutzeroberflächen, die auf schlechte Antwortzeiten zurückzuführen sind, sind das Resultat von Flaschenhälsen und schlechter Speicher- und Ressourcenverwaltung. Diese Defizite im Sourcecode werden durch Profiler ermittelt. Die Profiler erstellen sehr große Profile Daten die aus einem sehr unübersichtlichen Wust an Informationen bestehen und somit nicht effektiv analysierbar sind.
Der HPJMeter ermöglicht eine schnelle Analyse und Deutung dieser Informationen.

1 Profiling

Im Folgenden Kapitel werden wir kurz auf das Profiling eingehen. Das Profiling an sich wurde schon in einer anderen Seminararbeit in diesem Semester genau beschrieben. Wir werden uns deshalb auf dieses Wissen beziehen.

1.1 Grundlagen und Definitionen

Unter „Profiling“ versteht man die Analyse der Laufzeit Performance indem während der Ausführung der Anwendung Messdaten ermittelt werden.
An Hand dieser Messdaten können dann Rückschlüsse auf versteckte Optimierungspotentiale des Sourcecode gezogen werden.

Im Folgenden möchten wir ein Beispielprogramm „profilen“, wir werden diese Profile Datei auch später im HPJMeter analysieren.

Als Beispiel-Code verwenden wir ein kleines, selbst erstelltes Java Programm.
Wir haben versucht mit diesem kleinen Programm möglichst viele typische Berechnungen zu simulieren.
Das Programm implementiert einen QuickSort-Algorithmus, Berechnungen mit sin(x), cos(x) und tan(x) Funktionen, eine rekursive Ackermann Funktion sowie eine String Konkatenation.

```
import java.util.Random;


/**
 * @author Björn Adam (mailto: b.adam@gmx.net)
 *
```

```
 * Beispiel Anwendung zur Seminararbeit zu der Vorlesung
 * "Verteilte und parallele Systeme 2"
 */

class QuickSort {

 static void exchange (int a[], int m, int n) {
       int t = a[m];
       a[m] = a[n];
       a[n] = t;
 }

 static int partition (int a[], int m, int n) {
       int x = a[m];
       int j = n + 1;
       int i = m - 1;

       while (true) {
             j--;
             while (a[j] > x) j--;
                   i++;
             while (a[i] < x) i++;
                   if (i < j) exchange (a, i, j);
                   else return j;
             }
 }

 static void qsort (int a[], int l, int r) {
       if (l < r) {
             int r2 = partition (a, l, r);
             qsort (a, l, r2);
             qsort (a, r2 + 1, r);
       }
 }

 static void appendString(){
       String text="";
       for (int i=0; i<10000;i++)
             text = text+"X";
 }
```

```
	static void rechneSinTanCos(int[] a) {
		for (int i = 0;i<a.length; i++) {
			double wert =
Math.log(Math.sin(a[i]))/Math.tan(Math.cos(a[i]));
		}
	}

	public static void main (String args[]) {
		int test[];
		int size=1000000;

		test=new int[size];

		Random r = new Random();
		for (int i=0;i<size;i++){
			test[i] =r.nextInt(size);
		}

		System.out.println("QuickSort...");
		qsort (test, 0, test.length - 1);

		System.out.println("50x Berechnungen Sin,Cos,Tan ...");
		fuenfzigMalrechneSinCosTan(test);

		System.out.println("50x Ack(3,3) ...");
		fuenfzigMalAck();

		System.out.println("50x String append ...");
		appendString();

		System.out.println("Fertig!");
	}

	public static void fuenfzigMalAck() {
		for (int i=0;i<50;i++)
			Ack(3,3);
	}

	public static void fuenfzigMalrechneSinCosTan(int[] a) {
		for (int i=0;i<50;i++)
			rechneSinTanCos(a);
```

```
    }

    public static int Ack(int M, int N) {
        if (M == 0) return( N + 1 );
        if (N == 0) return( Ack(M - 1, 1) );

        return( Ack(M - 1, Ack(M, (N - 1))) );
    }
}
```

Abbildung 1.1-1: Beispiel Code zum Profilen

Mach dem Ausführen der Anwendung unter der Verwendung eines Java Profilers entsteht die folgende Profile Datei:

```
JAVA PROFILE 1.0.1, created Mon Dec 08 12:58:36 2003

Header for -Xhprof ASCII Output

Copyright 1998 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto,
California, 94303, U.S.A.  All Rights Reserved.

WARNING!  This file format is under development, and is subject to
change without notice.

This file contains the following types of records:

THREAD START
THREAD END      mark the lifetime of Java threads

TRACE           represents a Java stack trace.  Each trace consists
                of a series of stack frames.  Other records refer to
                TRACEs to identify (1) where object allocations have
                taken place, (2) the frames in which GC roots were
                found, and (3) frequently executed methods.

HEAP DUMP       is a complete snapshot of all live objects in the Java
                heap.  Following distinctions are made:

                ROOT    root set as determined by GC
                CLS     classes
                OBJ     instances
                ARR     arrays

SITES           is a sorted list of allocation sites.  This identifies
                the most heavily allocated object types, and the TRACE
                at which those allocations occurred.
```

```
CPU SAMPLES     is a statistical profile of program execution.  The VM
                periodically samples all running threads, and assigns
                a quantum to active TRACEs in those threads.  Entries
                in this record are TRACEs ranked by the percentage of
                total quanta they consumed; top-ranked TRACEs are
                typically hot spots in the program.

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
                "count" field indicates the number of times each TRACE
                is invoked.

MONITOR TIME    is a profile of monitor contention obtained by measuring
          the time spent by a thread waiting to enter a monitor.
          Entries in this record are TRACEs ranked by the percentage
          of total monitor contention time and a brief description
          of the monitor.  The "count" field indicates the number of
          times the monitor was contended at that TRACE.

MONITOR DUMP     is a complete snapshot of all the monitors and threads in
          the System.

HEAP DUMP, SITES, CPU SAMPLES|TIME and MONITOR DUMP|TIME records are generated
at program exit.  They can also be obtained during program execution by typing
Ctrl-\ (on Solaris) or by typing Ctrl-Break (on Win32).

--------

THREAD START (obj=2b67268, id = 2, name="Finalizer", group="system")
THREAD START (obj=2b67e20, id = 1, name="Reference Handler", group="system")
THREAD START (obj=2b67ed8, id = 3, name="main", group="main")
THREAD START (obj=2b6afe0, id = 4, name="Signal Dispatcher", group="system")
THREAD END (id = 3)
THREAD START (obj=2b689d0, id = 5, name="DestroyJavaVM", group="main")
THREAD END (id = 5)
TRACE 1: (thread=3)
  <empty>
TRACE 25: (thread=3)
  java.lang.StringBuffer.<init>(<Unknown>:Unknown line)
  QuickSort.appendString(QuickSort.java:Unknown line)
  QuickSort.main(QuickSort.java:Unknown line)
TRACE 4: (thread=3)
  QuickSort.exchange(QuickSort.java:Unknown line)
  QuickSort.partition(QuickSort.java:Unknown line)
  QuickSort.qsort(QuickSort.java:Unknown line)
  QuickSort.qsort(QuickSort.java:Unknown line)
TRACE 14: (thread=3)
  sun.misc.URLClassPath$FileLoader.getResource(<Unknown>:Unknown line)
```

```
    sun.misc.URLClassPath.getResource(<Unknown>:Unknown line)
    java.net.URLClassLoader$1.run(<Unknown>:Unknown line)
    java.security.AccessController.doPrivileged(<Unknown>:Unknown line)
  TRACE 28: (thread=3)
    QuickSort.qsort(QuickSort.java:Unknown line)
    QuickSort.qsort(QuickSort.java:Unknown line)
    QuickSort.qsort(QuickSort.java:Unknown line)
    QuickSort.qsort(QuickSort.java:Unknown line)
  TRACE 3: (thread=3)
    java.lang.StringBuffer.toString(<Unknown>:Unknown line)
    QuickSort.appendString(QuickSort.java:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 5: (thread=3)
    java.util.Random.next(<Unknown>:Unknown line)
    java.util.Random.nextInt(<Unknown>:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 36: (thread=3)
    sun.nio.cs.SingleByteEncoder.encodeArrayLoop(<Unknown>:Unknown line)
    sun.nio.cs.SingleByteEncoder.encodeLoop(<Unknown>:Unknown line)
    java.nio.charset.CharsetEncoder.encode(<Unknown>:Unknown line)
    sun.nio.cs.StreamEncoder$CharsetSE.implWrite(<Unknown>:Unknown line)
  TRACE 11: (thread=3)
    java.security.ProtectionDomain.<init>(<Unknown>:Unknown line)
    java.security.SecureClassLoader.getProtectionDomain(<Unknown>:Unknown line)
    java.security.SecureClassLoader.defineClass(<Unknown>:Unknown line)
    java.net.URLClassLoader.defineClass(<Unknown>:Unknown line)
  TRACE 12: (thread=3)
    java.util.Random.nextInt(<Unknown>:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 15: (thread=3)
    sun.misc.URLClassPath$3.run(<Unknown>:Unknown line)
    java.security.AccessController.doPrivileged(<Unknown>:Unknown line)
    sun.misc.URLClassPath.getLoader(<Unknown>:Unknown line)
    sun.misc.URLClassPath.getLoader(<Unknown>:Unknown line)
  TRACE 35: (thread=3)
    QuickSort.partition(QuickSort.java:Unknown line)
    QuickSort.qsort(QuickSort.java:Unknown line)
    QuickSort.qsort(QuickSort.java:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 21: (thread=3)
    java.lang.StrictMath.log(<Unknown>:Unknown line)
    java.lang.Math.log(<Unknown>:Unknown line)
    QuickSort.rechneSinTanCos(QuickSort.java:Unknown line)
    QuickSort.fuenfzigMalrechneSinCosTan(QuickSort.java:Unknown line)
  TRACE 2: (thread=3)
    sun.misc.AtomicLongCSImpl.attemptUpdate(<Unknown>:Unknown line)
    java.util.Random.next(<Unknown>:Unknown line)
    java.util.Random.nextInt(<Unknown>:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 38: (thread=3)
    QuickSort.rechneSinTanCos(QuickSort.java:Unknown line)
    QuickSort.fuenfzigMalrechneSinCosTan(QuickSort.java:Unknown line)
```

```
  QuickSort.main(QuickSort.java:Unknown line)
TRACE 29: (thread=3)
  java.lang.StringBuffer.<init>(<Unknown>:Unknown line)
  java.net.URLStreamHandler.toExternalForm(<Unknown>:Unknown line)
  java.net.URL.toExternalForm(<Unknown>:Unknown line)
  java.net.URL.toString(<Unknown>:Unknown line)
TRACE 34: (thread=3)
  QuickSort.partition(QuickSort.java:Unknown line)
  QuickSort.qsort(QuickSort.java:Unknown line)
  QuickSort.qsort(QuickSort.java:Unknown line)
  QuickSort.qsort(QuickSort.java:Unknown line)
TRACE 18: (thread=3)
  java.lang.String.getChars(<Unknown>:Unknown line)
  java.lang.StringBuffer.append(<Unknown>:Unknown line)
  QuickSort.appendString(QuickSort.java:Unknown line)
  QuickSort.main(QuickSort.java:Unknown line)
TRACE 24: (thread=3)
  java.lang.Math.tan(<Unknown>:Unknown line)
  QuickSort.rechneSinTanCos(QuickSort.java:Unknown line)
  QuickSort.fuenfzigMalrechneSinCosTan(QuickSort.java:Unknown line)
  QuickSort.main(QuickSort.java:Unknown line)
TRACE 20: (thread=3)
  java.lang.String.getChars(<Unknown>:Unknown line)
  java.lang.StringBuffer.append(<Unknown>:Unknown line)
  java.lang.StringBuffer.<init>(<Unknown>:Unknown line)
  QuickSort.appendString(QuickSort.java:Unknown line)
TRACE 19: (thread=3)
  java.lang.String.charAt(<Unknown>:Unknown line)
  sun.net.www.ParseUtil.decode(<Unknown>:Unknown line)
  sun.net.www.protocol.file.FileURLConnection.getPermission(<Unknown>:Unknown line)
  java.net.URLClassLoader.getPermissions(<Unknown>:Unknown line)
TRACE 17: (thread=3)
  java.lang.StringBuffer.length(<Unknown>:Unknown line)
  java.lang.String.<init>(<Unknown>:Unknown line)
  java.lang.StringBuffer.toString(<Unknown>:Unknown line)
  QuickSort.appendString(QuickSort.java:Unknown line)
TRACE 26: (thread=3)
  java.lang.StrictMath.tan(<Unknown>:Unknown line)
  java.lang.Math.tan(<Unknown>:Unknown line)
  QuickSort.rechneSinTanCos(QuickSort.java:Unknown line)
  QuickSort.fuenfzigMalrechneSinCosTan(QuickSort.java:Unknown line)
TRACE 7: (thread=3)
  sun.misc.JarIndex.getJarIndex(<Unknown>:Unknown line)
  sun.misc.URLClassPath$JarLoader.<init>(<Unknown>:Unknown line)
  sun.misc.URLClassPath$3.run(<Unknown>:Unknown line)
  java.security.AccessController.doPrivileged(<Unknown>:Unknown line)
TRACE 33: (thread=3)
  java.lang.Math.log(<Unknown>:Unknown line)
  QuickSort.rechneSinTanCos(QuickSort.java:Unknown line)
  QuickSort.fuenfzigMalrechneSinCosTan(QuickSort.java:Unknown line)
  QuickSort.main(QuickSort.java:Unknown line)
TRACE 30: (thread=3)
```

```
    java.lang.System.arraycopy(<Unknown>:Unknown line)
    java.lang.String.getChars(<Unknown>:Unknown line)
    java.lang.StringBuffer.append(<Unknown>:Unknown line)
    QuickSort.appendString(QuickSort.java:Unknown line)
  TRACE 32: (thread=3)
    java.lang.String.toString(<Unknown>:Unknown line)
    java.lang.String.valueOf(<Unknown>:Unknown line)
    QuickSort.appendString(QuickSort.java:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 9: (thread=3)
    java.lang.StringBuffer.setShared(<Unknown>:Unknown line)
    java.lang.String.<init>(<Unknown>:Unknown line)
    java.lang.StringBuffer.toString(<Unknown>:Unknown line)
    QuickSort.appendString(QuickSort.java:Unknown line)
  TRACE 37: (thread=3)
    QuickSort.Ack(QuickSort.java:Unknown line)
    QuickSort.Ack(QuickSort.java:Unknown line)
    QuickSort.Ack(QuickSort.java:Unknown line)
    QuickSort.Ack(QuickSort.java:Unknown line)
  TRACE 22: (thread=3)
    java.lang.String.valueOf(<Unknown>:Unknown line)
    QuickSort.appendString(QuickSort.java:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 8: (thread=3)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 23: (thread=3)
    java.lang.String.<init>(<Unknown>:Unknown line)
    java.lang.StringBuffer.toString(<Unknown>:Unknown line)
    QuickSort.appendString(QuickSort.java:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 31: (thread=3)
    sun.misc.AtomicLongCSImpl.get(<Unknown>:Unknown line)
    java.util.Random.next(<Unknown>:Unknown line)
    java.util.Random.nextInt(<Unknown>:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 27: (thread=3)
    QuickSort.appendString(QuickSort.java:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 6: (thread=3)
    QuickSort.exchange(QuickSort.java:Unknown line)
    QuickSort.partition(QuickSort.java:Unknown line)
    QuickSort.qsort(QuickSort.java:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 10: (thread=3)
    QuickSort.partition(QuickSort.java:Unknown line)
    QuickSort.qsort(QuickSort.java:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 16: (thread=3)
    java.lang.StringBuffer.append(<Unknown>:Unknown line)
    QuickSort.appendString(QuickSort.java:Unknown line)
    QuickSort.main(QuickSort.java:Unknown line)
  TRACE 13: (thread=3)
```

```
  java.lang.System.arraycopy(<Unknown>:Unknown line)
  java.lang.String.getChars(<Unknown>:Unknown line)
  java.lang.StringBuffer.append(<Unknown>:Unknown line)
  java.lang.StringBuffer.<init>(<Unknown>:Unknown line)
CPU TIME (ms) BEGIN (total = 673970) Mon Dec 08 13:12:51 2003
rank   self  accum   count trace method
   1 25.31% 25.31%      50    38 QuickSort.rechneSinTanCos
   2 23.89% 49.20% 50000000    33 java.lang.Math.log
   3 18.92% 68.12% 50000000    24 java.lang.Math.tan
   4 15.56% 83.68% 50000000    21 java.lang.StrictMath.log
   5 11.88% 95.56% 50000000    26 java.lang.StrictMath.tan
   6  0.91% 96.47%  999996    34 QuickSort.partition
   7  0.89% 97.36% 1999992    28 QuickSort.qsort
   8  0.88% 98.23% 4557679     4 QuickSort.exchange
   9  0.52% 98.75% 1000207     5 java.util.Random.next
  10  0.44% 99.18% 1000000    12 java.util.Random.nextInt
  11  0.21% 99.40% 1000207     2 sun.misc.AtomicLongCSImpl.attemptUpdate
  12  0.19% 99.58% 1000207    31 sun.misc.AtomicLongCSImpl.get
  13  0.17% 99.75%       1     8 QuickSort.main
  14  0.05% 99.80%       1    10 QuickSort.partition
  15  0.04% 99.84%  121250    37 QuickSort.Ack
  16  0.04% 99.88%  245901     6 QuickSort.exchange
  17  0.02% 99.91%       2    35 QuickSort.partition
  18  0.01% 99.92%   10000    23 java.lang.String.<init>
  19  0.01% 99.93%   10000    13 java.lang.System.arraycopy
  20  0.01% 99.94%   10000    20 java.lang.String.getChars
  21  0.01% 99.95%       1    27 QuickSort.appendString
  22  0.01% 99.96%   10000    25 java.lang.StringBuffer.<init>
  23  0.01% 99.97%   10000    16 java.lang.StringBuffer.append
  24  0.00% 99.97%   10000     3 java.lang.StringBuffer.toString
  25  0.00% 99.98%   10000    22 java.lang.String.valueOf
  26  0.00% 99.98%   10000    32 java.lang.String.toString
  27  0.00% 99.98%   10000     9 java.lang.StringBuffer.setShared
  28  0.00% 99.99%   10000    18 java.lang.String.getChars
  29  0.00% 99.99%   10000    30 java.lang.System.arraycopy
  30  0.00% 99.99%       4    29 java.lang.StringBuffer.<init>
  31  0.00% 99.99%       5    15 sun.misc.URLClassPath$3.run
  32  0.00% 99.99%       1    11 java.security.ProtectionDomain.<init>
  33  0.00% 99.99%      10    36 sun.nio.cs.SingleByteEncoder.encodeArrayLoop
  34  0.00% 100.00%       4     7 sun.misc.JarIndex.getJarIndex
  35  0.00% 100.00%       1    14 sun.misc.URLClassPath$FileLoader.getResource
  36  0.00% 100.00%      33    19 java.lang.String.charAt
  37  0.00% 100.00%   10000    17 java.lang.StringBuffer.length
  38  0.00% 100.00%       0     1 <empty trace>
CPU TIME (ms) END
```

Abbildung 1.1-2: Profile Daten

1.2 Probleme beim Profiling

Bei der Erstellung von Profile Dateien müssen einige Dinge beachtet werden um ein möglichst genaues Mess Ergebnis zu erhalten.
Ein grosses Problem beim Erstellen der Profile Daten entsteht dadurch, dass Profiling an sich auch Speicher und Rechenleistung benötigt.
Das verfälscht das Ergebnis. Ausgehend von diesem Effekt kann man zwei Problemarten definieren:

- Overhead — Die Laufzeit verlängert sich bei dem Profilen dadurch das die Messdaten ermittelt werden müssen.
- Intrusion — Wenn der Profiler dieselben Ressourcen verwendet wie die Anwendung die gemessen werden soll, so misst man auch die Effekte die der Profiler verursacht.

Es existieren zwei Konzepte um diese Effekte möglichst gering zu halten.

- Sampling — Beim Sampling stoppt die Laufzeit Umgebung nach fest eingestellten Intervallen die Ausführung der Anwendung und ermittelt dann in welcher Methode sich das Programm zu diesem Zeitpunkt befindet. Das führt dazu dass die Methode die, die meiste CPU Zeit verwendet auch am häufigsten in der Statistik auftaucht.
  Diese Messmethode verursacht einen geringen Overhead und ermöglicht somit ein relativ genaues Ermitteln der timing data. Abstriche werden hier allerdings im Bereich der call counts und des call graphs gemacht.
- Tracing — Beim Tracing werden immer beim Methodenaufruf Daten ermittelt. Daten sind hier zum Beispiel die aufgerufene Methode sowie die Zeit die in der Methode verbracht wurde.

Diese Methode ist komplementär zu dem Sampling. Hier werden die call counts und der call graph relativ genau ermitteln. Allerdings verursacht der auftretende Overhead eine leichte Verfälschung des timing data.

1.3 Performance Analyse Prozess

Um eine Java Anwendung in der Performance zu steigern, verfolgt man typischerweise immer die folgenden drei Schritte:

1. Ausführen der Anwendung und Generierung der Profile Daten
2. Analyse der Profile Daten
3. Ändern der Anwendung um das Performance Problem zu beseitigen

Um genauerer Ergebnisse zu bekommen, sollte man das erstellen der Profile Dateien mehrmals unter gleichen Bedingungen wiederholen und die Ergebnisse danach vergleichen bzw. die Mittelwerte bestimmen.

In den meisten Fällen werden nach der Erstellung der Profile Daten, Codeteile geändert und danach eine erneute Erstellung der Profile Daten durchgeführt.
Man kann dann an den neu erstellten Profile Daten erkennen ob die letzte Optimierung erfolgreich war und ob noch Optimierungsbedarf besteht.

Bei diesem Optimierungs- Prozess sind jedoch zwei einfache Regeln zu beachten:

- Korrektheit ist wichtiger als Performance

  Wenn man Algorithmen im Sourcecode in der Performance verbessern möchte so muss man immer beachten das die Korrektheit des Algorithmus gewährleistet bleibt.

- Messen des Vorschritts

  Wenn man den Sourcecode optimiert,

sollte man das in kleinen Schritten tun und zwischen diesen Schritten den Vorschritt neu messen.
Dies beugt der Gefahr vor aus einer Optimierung eine Verschlechterung zu machen.

1.4 Erstellen von Profile Dateien

Es gibt verschiedene Java Profiler. Der bekannteste Profiler ist der hprof Profiler.

Um eine Java Anwendung mit dem hprof Profiler zu analysieren muss man folgende Kommandozeile verwenden:

```
java … -Xrunhprof:<options> ApplicationClassName
```

Der Parameter `-Xrunhprof:<options>` gibt an das die Java Anwendung profiled werden soll. Es wird in diesem Fall der hprof-Profiler verwendet.

1.4.1 Nützliche Einstellungen des hprof-Profiler:

Unter `<options>` kann man dem hprof-Profiler einige Parameter übergeben.

Einige nützliche Parameter für den hprof-Profiler:

Profile Art	**Parameter (`<options>`)**
• Performance Analyse	`cpu=samples,thread=y,depth=10,cutoff=0,format=a`
• Performance	`cpu=timer,threads=y,cutoff=0,format=a`

Analyse mit Methoden Aufrufszähler	
• Analyse von Objekt Allokation	`heap=sites,cpu=samples,thread=y,cutoff=0,format=a`
• Komplette Liste der möglichen Optionen anzeigen	`java … -Xrunhprof:help`

2 HPJMeter

Im Folgenden Kapitel möchten wir den HPJMeter etwas genauer vorstellen.

2.1 Einführung

Der HPJMeter ist ein kostenloses, Java-basiertes, grafisches Performance Analyse Tool. Es zeigt, grafisch aufbereitet, Profil Dateien an und ermöglicht so eine einfache und schnelle Analyse von Sourcecode.

Der HPJMeter unterstützt Profile Dateinen von folgenden Profilern:

- Prof (Java 1.1)
- Eprof (HP-UX JDK 1.1.8)
- Xeprof (Java 2, HotSpot VM für HP-UX)
- Xrunprof (Chai VM)
- Xrunhprof (Java 2, Classic oder HotSpot VM)
- Prospect (Java 2, Classic oder HotSpot HP-UX)

Mit der grafischen Darstellung der Messdaten im HPJMeter ist es sehr leicht möglich potentielle Flaschenhälse in dem Sourcecode zu ermitteln.

2.2 Die Oberfläche

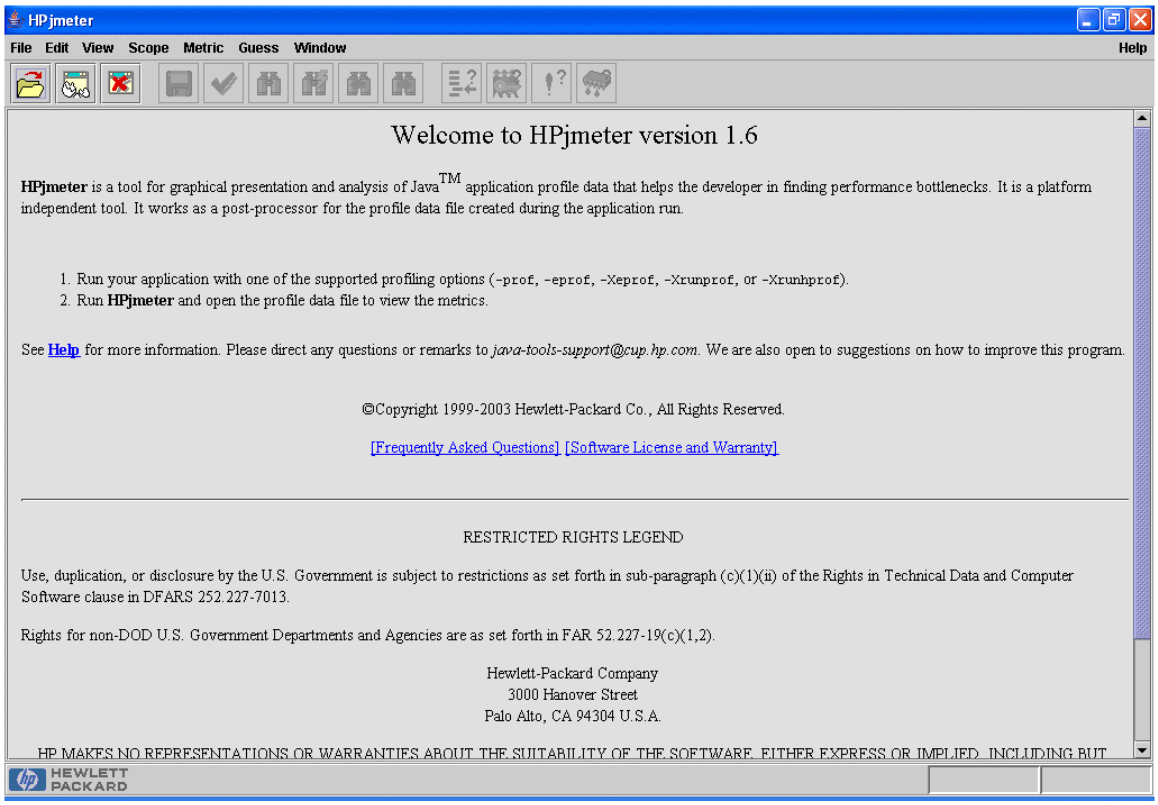

Abbildung 2.2-1: Oberfläche HPJMeter

Die Oberfläche des HPJMeters hat den Aufbau eines typischen Windows Programms. Das Pulldown Menu ermöglicht einen schnellen Zugriff auf die einzelnen Funktionen.

2.2.1 Öffnen von Profile Dateien

Das öffnen einer Profile Datei im HPJMeter ist sehr einfach. Wenn man unter dem Menupunkt „File“ den Punkt „Open“ auswählt, öffnet sich ein Datei Auswahlmenu indem man die gewünschte Profile Datei anwählen kann. Nach dem Bestätigen mit „OK“ wird die Profile Datei geladen und man bekommt eine kurze Zusammenfassung der wichtigsten Daten angezeigt.

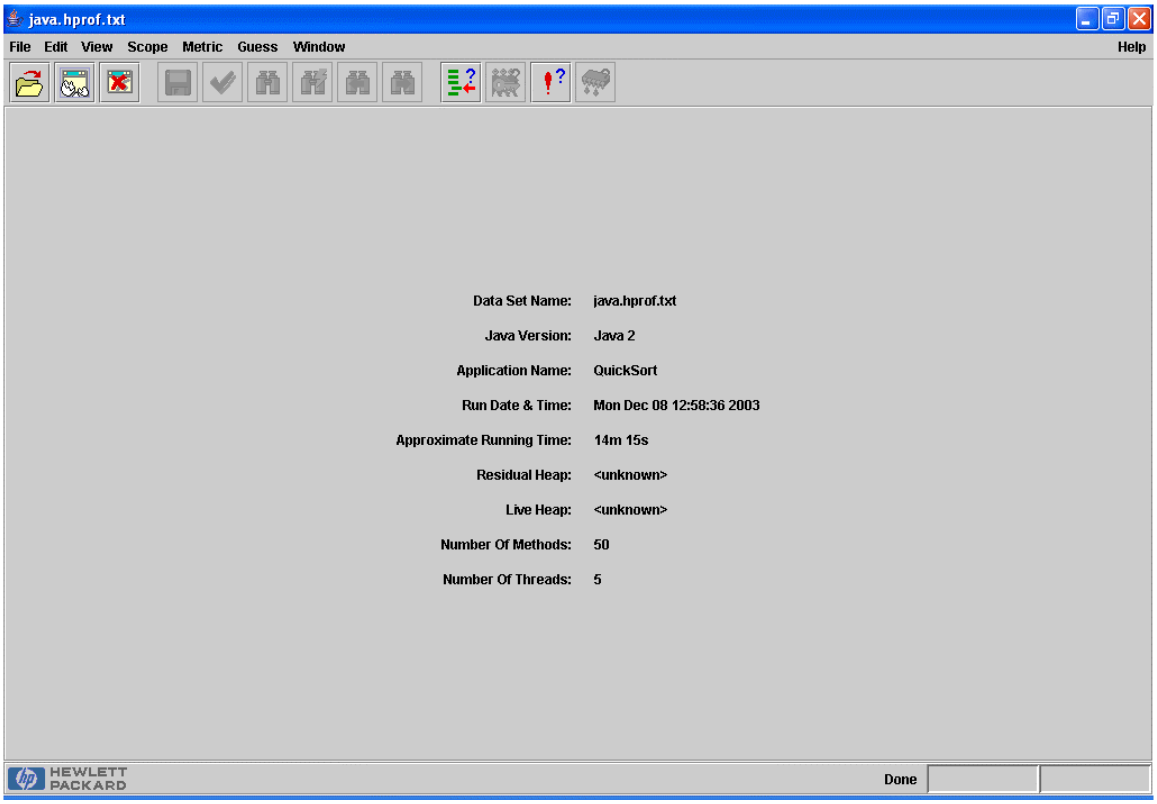

Abbildung 2.2-2: Zusammenfassung der wichtigsten Daten der Profile Datei

Man bekommt sehr schnell und übersichtlich einen Eindruck über die in der Profile Datei gespeicherten Daten. Dazu zählt zum Beispiel die „Running time“ sowie die „Number of Methods“.

2.3 Funktionen

2.3.1 Method/Class Times

Hpjmeter bietet die Möglichkeit, die Methodenaufrufe in CPU Zeit oder in der realen Zeit anzuzeigen (Abbildung 4-4). Es stehen die Funktionen Method Times (CPU) und Method Times (Clock) im Menüeintrag „Metric“ zur Verfügung. Bei der ersten Funktion wird nur die benötigte CPU Zeit angezeigt, im anderen Fall die Gesamtzeit welche die Methode benötigt hat. Hier ist anzumerken, dass der Funktionsumfang von der Profiling-Datei abhängt. In dem Fall der Methodenaufrufen bedeutet dies, dass die reale Zeit nur bei den eprof-Dateien zur Analyse vorhanden sind.

Es wird darüber hinaus zwischen inklusiven und exklusiven Methoden differenziert.

Inclusive: Summe inklusive den aufgerufenen Methoden
Exclusive: Summe der aufgerufenen Methode, d.h. wenn die Methode 5x aufgerufen wurde, wird auch die Summe von 5 Methodenaufrufen angezeigt.

Die Zeiten in der Grafik sind in Millisekunden angegeben und standardmäßig absteigend sortiert. Die Dauer ist zusätzlich mit einem Balken dargestellt so dass die Relationen zu den einzelnen Methoden deutlich ersichtlich ist.

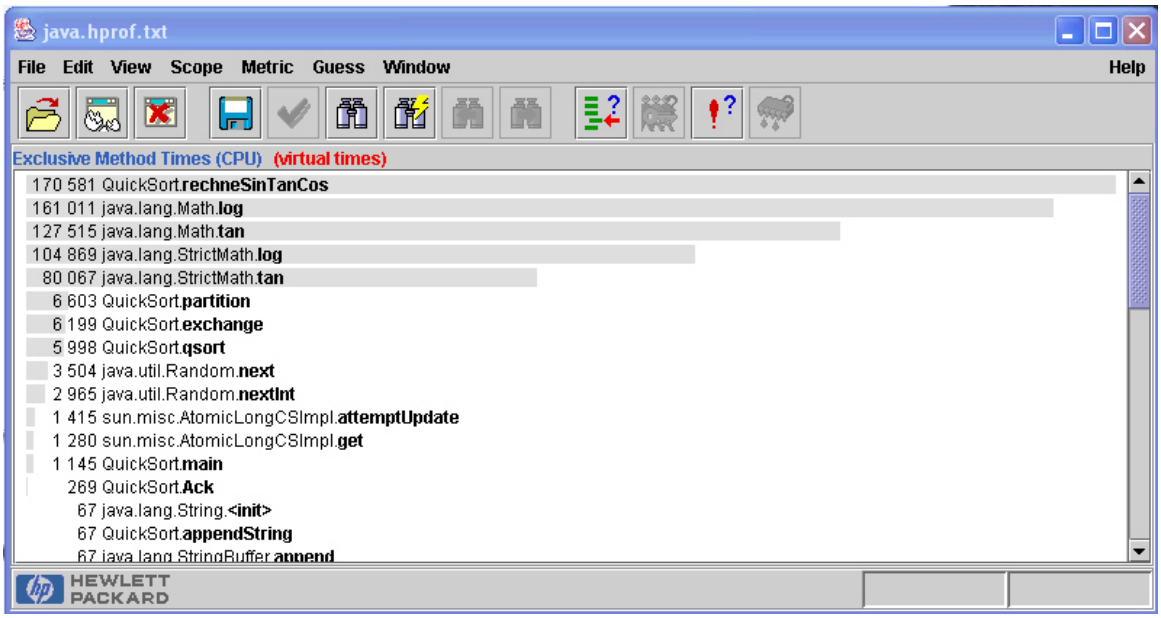

Abbildung 2.3-1: Method Times (CPU)

Eine ähnliche Ansicht kann für die Klassen aufgerufen werden. Unter dem Namen „Class Times“ wird eine Übersicht mit einer Selektion auf die Klassen des Programms vorgenommmen.

2.3.2 Call Graph

Die in einer Profile-Datei enthalten Methoden sind in Hpjmeter auch als hierarchische Struktur darstellbar und nennt sich „Call Graph Tree“ (Abbildung. 4-5)

Durch Mausklicken kann man sich so abwärts durch den Graphen arbeiten und demnach die Aufrufer des Methode in Erfahrung bringen. Die Eigenschaften des Baumes sind farblich dargestellt. Ein Blatt wird schwarz dargestellt, ein besuchter Knoten violett usw. Mit einem Doppelklick auf den gewünschten Knoten, öffnet sich ein Fenster mit allen Aufrufern dieser Methode.

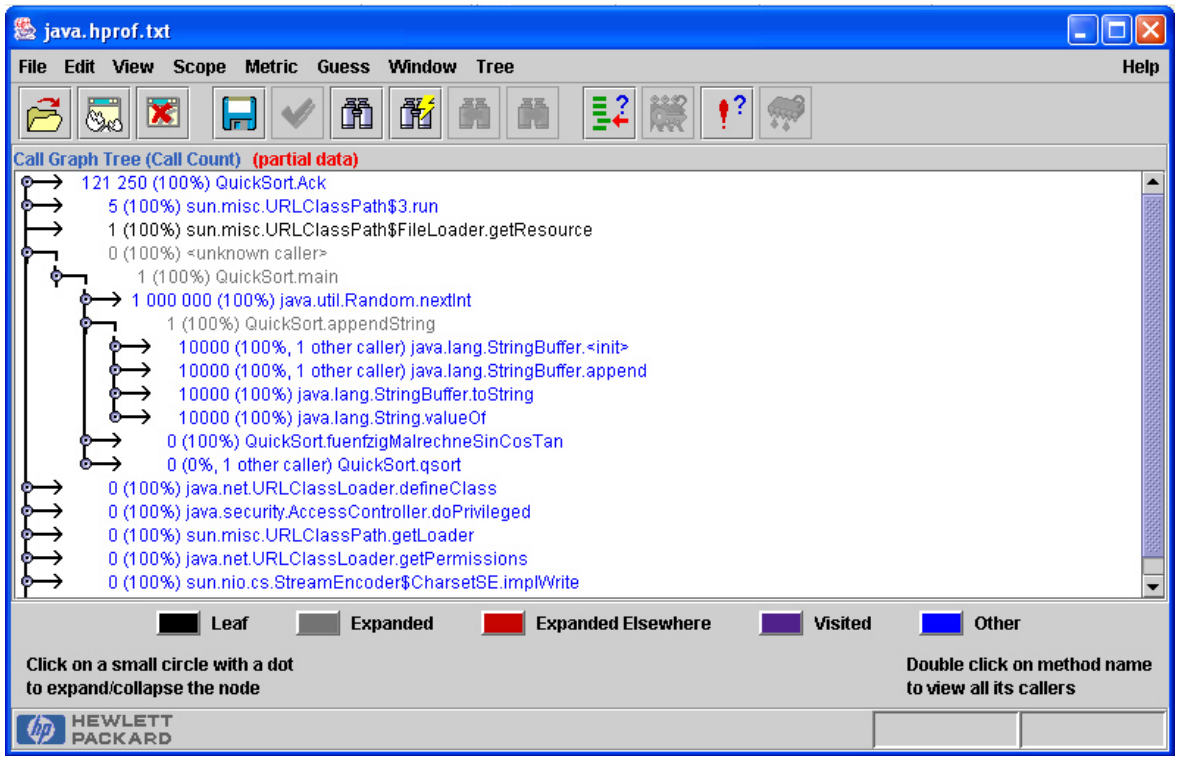

Abbildung 2.3-2: Call Graph Tree (Call Count)

2.3.3 Created Objects

Oft ist es hilfreich zu wissen, welche Objekte wie oft erzeugt werden.

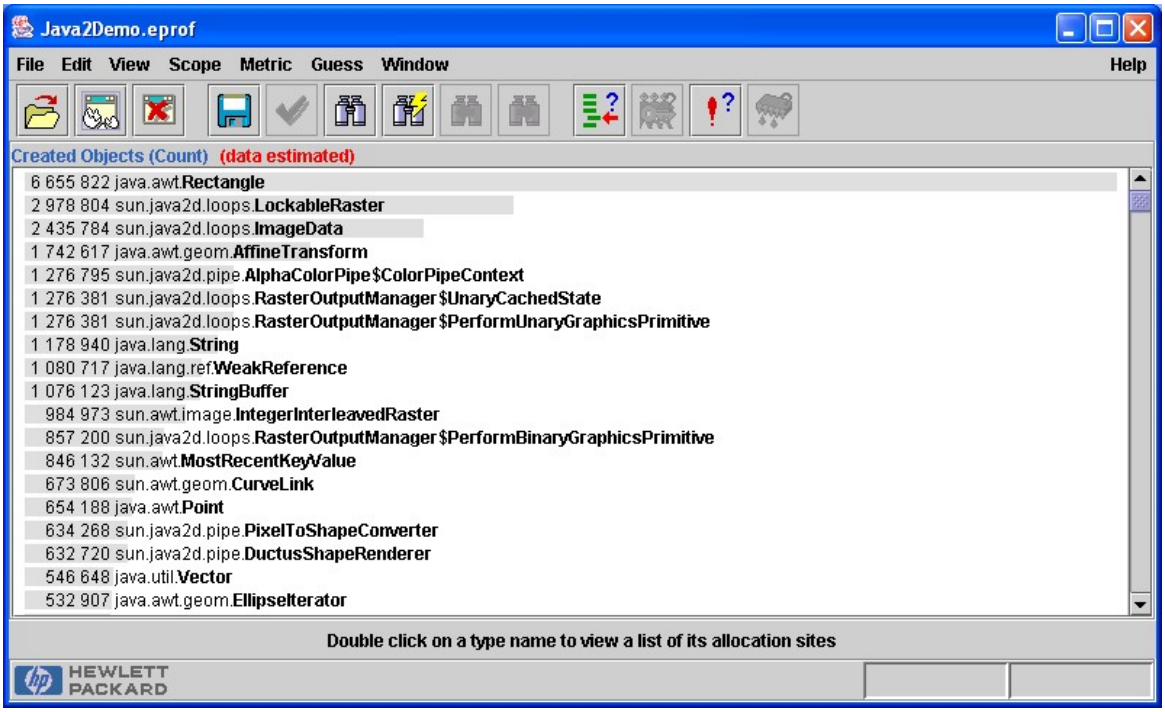

Abbildung 2.3-3: Created Objects (Count)

Um hier eine Hilfe zu schaffen, hat Hpjmeter eine Funktion namens „Created Objects (Count)“ (Abbildung 4-6).

Wie in allen Übersichten in diesem Tool sind auch hier die Anzahl der erzeugten Objekte absteigend sortiert dargestellt. So kann man von Oben nach Unten die Objekte analysieren und evtl. ungewöhnlich häufig erzeugte Objekte bearbeiten und anpassen um eine bessere Performance im Programm zu erreichen. Beispiel hierfür ist das String Objekt, welches häufig neu erzeugt wird und somit viel CPU-Zeit benötigt. Abhilfe schafft hier oft der Stringbuffer, welcher wesentlich schneller ist.

2.3.4 Thread Histogramm

Eine andere Funktion ist die Darstellung von Threads in einem Histogramm. Hier kann man die Aufteilung der Threads auf die verschiedenen Zustände. Gleichzeitig zeigt die Länge des Balkens die Lebensdauer des Threads an.

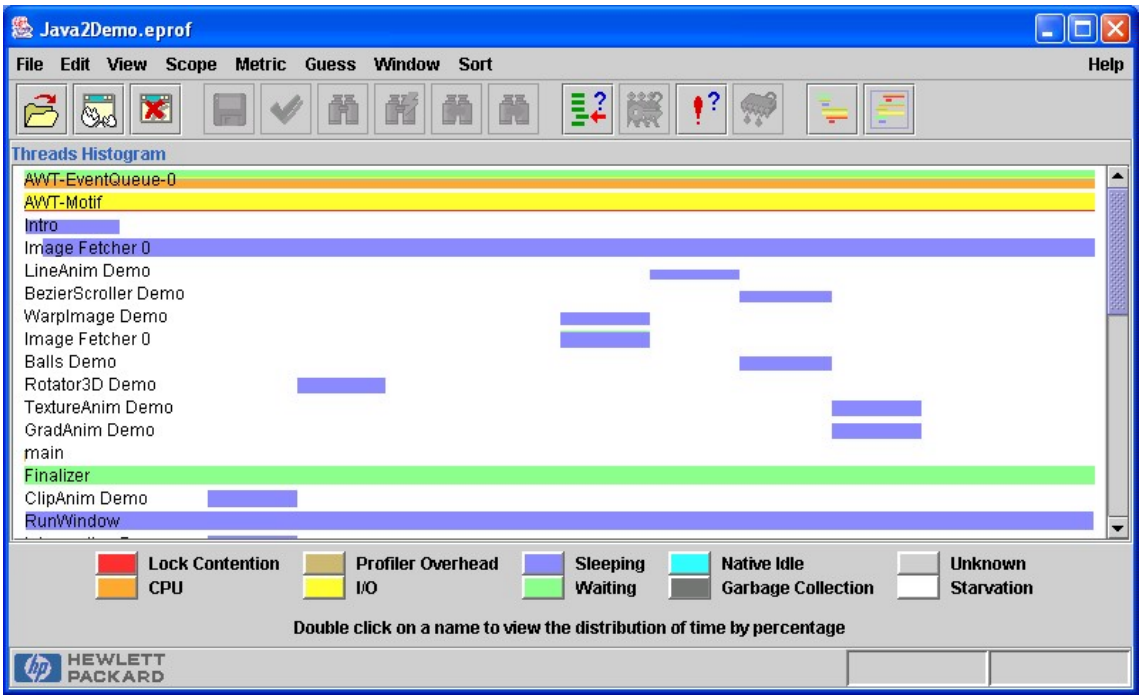

Abbildung 2.3-4: Threads Histogramm

Um eine bessere Übersicht für die einzelnen Threads zu bekommen kann man mit Doppelklick eine Kuchenstatistik erreichen. Hier wird zum Beipspiel nach CPU, I/O oder schlafenden Status unterteilt. Je wärmer die Farben um so größer ist die Wahrscheinlichkeit eine Verbesserung der Performance zu erreichen.

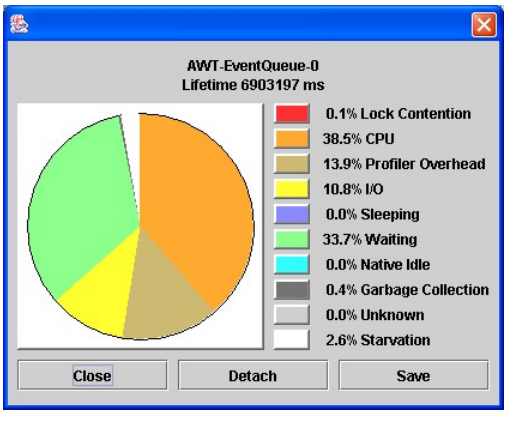

Abbildung 2.3-5: Thread Grafik

2.3.5 Heuristische Funktionen

2.3.5.1 Inline candidates

Performanceverlust kann durch unnötig viele Funktionen entstehen. Hpjmeter bietet hier eine Möglichkeit, diese mit der Funktion „Inline Candidates“ (Abbildung 4-9) zu entdecken um den Fehler zu beheben. Das Programm verwendet eine Heuristik, die Methoden aufspürt, welche wenig und einfachen Code enthalten. Ein anderes Kriterium ist eine geringe Anzahl von verschiedenen Aufrufern. Eine eigene Funktion die im Worst-case nur von einer anderen Methode aufgerufen wird, kann gleichermaßen in der Methode integriert werden und somit ein Performancegewinn bringen.
Die Funktion „Inline Candidates“ kann u.a. sperrenden Methoden aufspüren. Egal ob sie als synchronized deklariert oder synchronized im Code enthalten. Anzumerken ist dass es sich um eine Heuristik handelt, d.h. ein Vorschlag von Hpjmeter der einer weiteren Kontrolle bedarf.

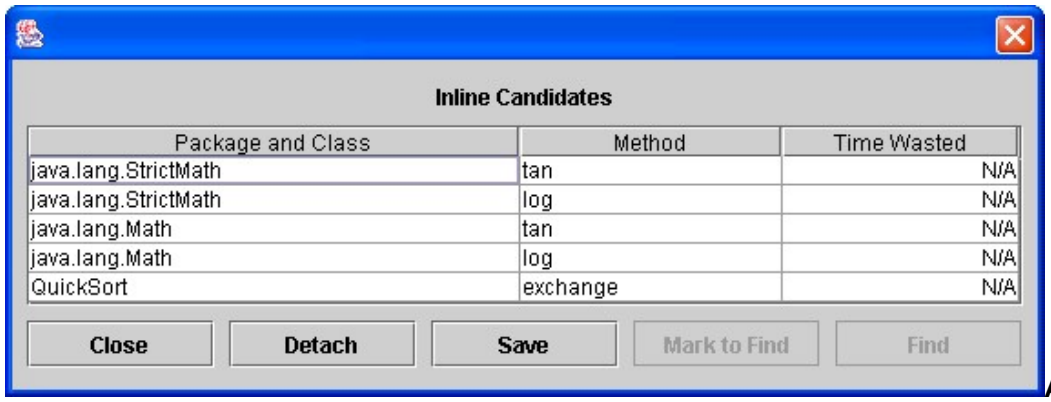

Abbildung 2.3-6: Übersicht Inline Candidates

2.3.5.2 Exceptions

Hpjmeter kann die Anzahl der Exceptions aufgrund einer heuristischen Funktion ermitteln und ausgeben (Abbildung 4-10).

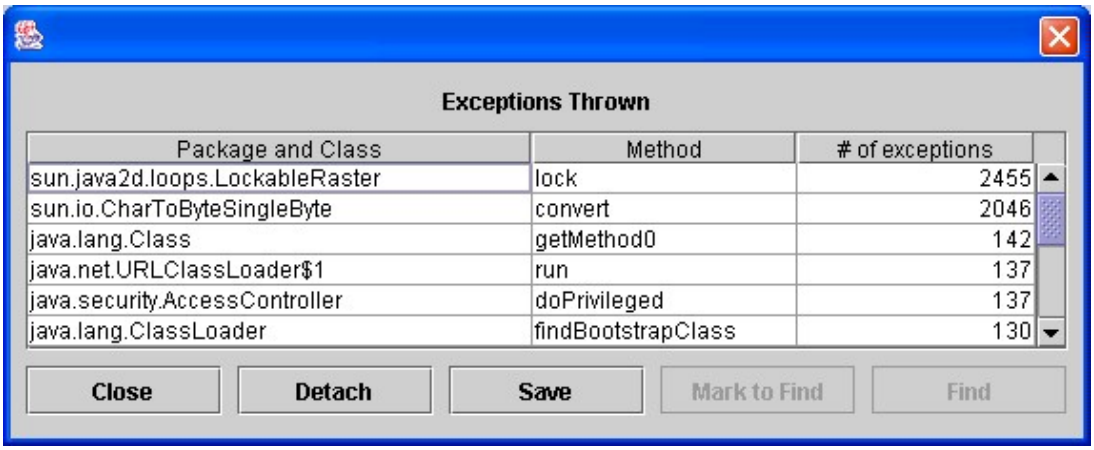

Abbildung 2.3-7: Exceptions

Durch die Heuristik stellt das Ergebnis lediglich Exceptions dar, die auftreten können .

2.3.6 Hilfe & Handling

Hpjmeter ist für ein entgeltfreies Tool sehr komfortable umfangreich. Zum besseren Handling sind einige kleine Funktionen integriert wie das Markieren und Suchen von Methoden oder das Ausklinken von Fenstern zum Vergleichen.
Zusätzlich ist das Programm mit einer gut verständlichen Hilfe ausgestattet.

2.3.7 Mark & Search

Man kann sich in allen Funktionsübersichten mit Doppelklick eine kleine Übersicht über die Methodenaufrufe anzeigen lassen.

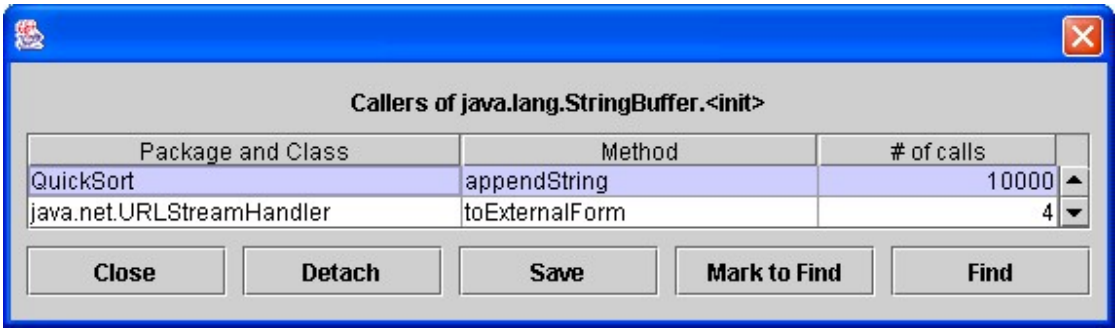

Abbildung 2.3-8: Callers

Um komfortabel die Funktion mit anderen Features zu bearbeiten, können die gewählten Methoden markiert und somit in den Zwischenspeicher geladen werden. In anderen Ansichten kann dann mit den Suchfunktionen die markierte Funktion leicht wiedergefunden werden.

2.3.8 Tree Pruning

Die Graphen eines komplexen Programms können sehr unübersichtlich werden. Aus diesem Grund besteht die Möglichkeit, den Baum unter bestimmten Kriterien abzuschneiden.

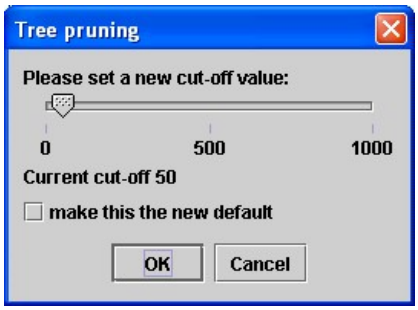

Abbildung 2.3-9: Tree Pruning

Diese Funktion findet man unter dem Namen „Tree Pruning“ (Abbildung 4-11). Parameter ist der sogenannte „cut-off value“. Standardmäßig ist der Wert auf 50 und abhängig von der Baumart kann damit eine Mindestanzahl von Methodenaufrufen oder eine minimale Zeit in Millisekunden festgelegt werden.
Die Funktion steht allerdings nicht für alle Grafiken zur Verfügung.

2.3.9 Selektion nach Process und Thread

Standardmäßig werden die Übersichtsfunktionen auf das ganze Programm angewendet. Es besteht die Möglichkeit, nach Prozess oder Thread zu selektieren. In dieser Ansicht lassen sich dann z.B. die Methodenaufrufe der einzelnen Thread vergleichen (Abbildung 4-13). So können rechts in der

Übersicht der gewählte Thread oder Prozess markiert werden und die entsprechende Übersicht wird mit dem gewünschten Filter aktualisiert.

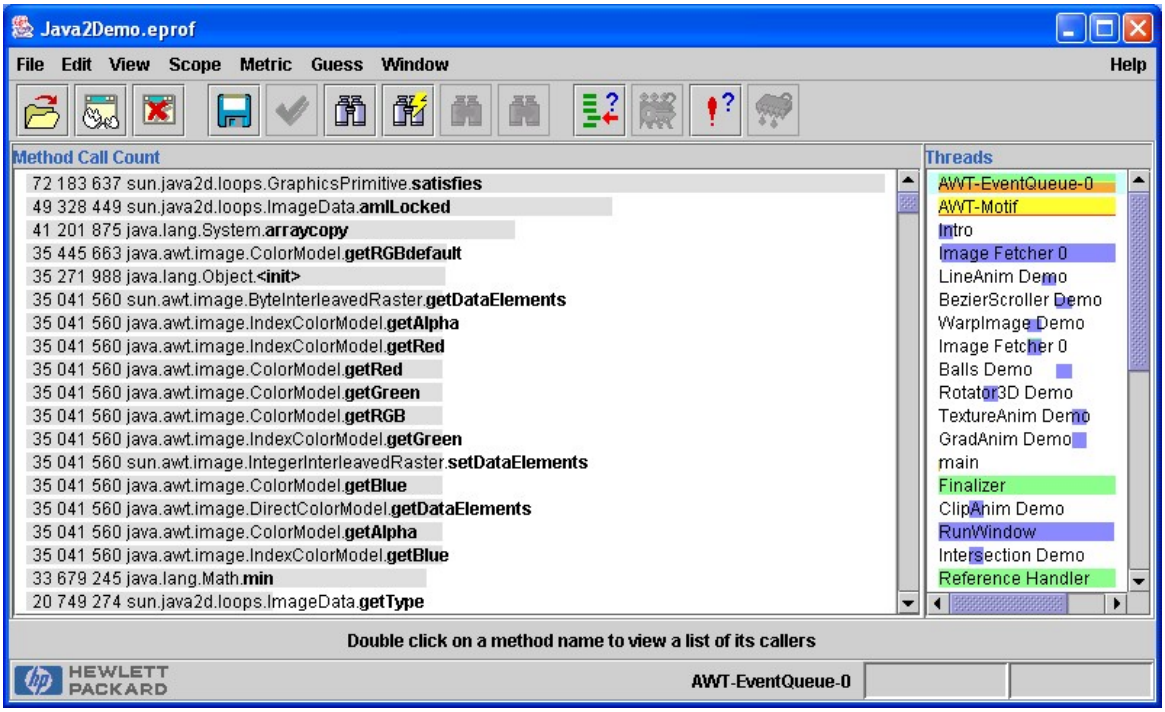

Abbildung 2.3-10: Selektion nach Prozess

3 Abbildungsverzeichnis: