University of Tartu
Faculty of Mathematics and Computer Science
Institute of Computer Science

Viktor Karabut

Comparison of allocation trackers in JVM

Bachelor's thesis

Supervisor: Vladimir Šor

Author: …… "….." may 2012

Supervisor: …….. "….." may 2012

Approved for defense

Professor: …….. "….." ……. 2012

TARTU 2012

Contents

Introduction

Memory leaks in Java are not the same as memory leaks in, for example, the C programming language. When a C-programmer wants to use memory on the heap, he should manually allocate a memory region. After application finishes using this memory, it should be manually freed. If the pointer to the allocated region is lost, then there is no appropriate way to release this memory. This situation is called a "memory leak". In Java the Java Virtual Machine (JVM) handles all work with memory. When a developer wants to create and use a new object, the JVM allocates a necessary amount of memory. During an application's life the JVM periodically checks for objects in memory that are not used anymore. Objects, which are not referenced, will be discarded and memory reclaimed to be used again. This process is called garbage collection. A memory leak in Java is a situation, where an application is not logically using objects, to which references still exist, meaning the Garbage Collector (GC) can't mark them as unused and free memory. When memory management of JVM cannot allocate any more memory, `java.lang.OutOfMemoryError` exception is thrown.

When a developer is faced with an "OutOfMemoryError" on production server, he can try to reproduce the problem in a test environment. Unfortunately, oftentimes test environments do not allow for reproduction of such errors. It's not always possible to mimic all parameters of a real environment. Developer often doesn't have all required input data or he just does not know how and why the memory leak occurred. This can be also caused by all sorts of bureaucratic obstacles and barriers in large companies with separate operations and development departments and developers just do not have full access to machines in a real environment.

However, even if it is possible to search for memory leaks in a production environment – use of many developer's tools, such as full-featured profilers is not possible, due to the memory and performance overhead not suitable for production environment. It is useful to apply all possible offline methods, such as analysis of heap dumps and collection of allocations' logs. And when a memory leak is localized, only then it would be helpful to turn on an allocation tracker for a specific set of objects or allocation sites to find out what code is responsible for creating objects that are eventually leaked. In order to do it in a production environment, we need effective methods and algorithms for allocation tracking.

Memory allocations tracker is a tool, which works in runtime and logs memory allocation by specific objects or sites. Usually it is a part of a profiler, but standalone solutions also exist. The aim of this work is to review and compare existing open source solutions for allocation tracking in JVM.

The first chapter of this work describes benchmarking techniques, which will be used for comparison of different allocations trackers. In this work will SPECjvm2008 will be used. It is a benchmark suite for measuring the performance of a Java Runtime Environment (JRE). It contains several real life applications and benchmarks focusing on core java functionality. The SPECjvm2008 workload mimics a variety of common general-purpose application computations.

As SPECjvm2008 does not give any information about memory usage a lightweight JVM Tool Interface (JVM TI) agent was created. JVM TI allows a program to inspect the state and

to control the execution of other application running in the JVM [1]. The agent works in a separate thread and writes memory usage statistics to a CSV file every second. The agent creates an insignificant overhead, so it does not distort the SPECjvm2008 results.

In the next chapters of this work open-source allocations trackers are reviewed. Their work principles, algorithms and memory structures are examined. For measuring allocations tracking efficiency the SPECjvm2008 suite will be run with every tested allocations tracking solution. The obtained data allows comparing memory and performance overhead of different approaches in memory allocations tracking. Chapter 2 introduces HPROF – an example profiler, shipped with the Oracle Java Development Kit (JDK) and uses the JVM TI. In chapter 3 we review the NetBeans profiler, previously known as JFluid. This is full-weight Java profiler integrated with the NetBeans IDE. Chapter 4 is about Eclipse Test and Performance platform. This is a collection of open- source frameworks and services that allows software developers to build test and performance tools. In the last chapter results of using Google's Allocation Instrumenter are presented. It uses `java.lang.instrument` package and ASM Java byte code manipulation and analysis framework.

For each examined profiler brief description along with implementation details and benchmarking results are given.

Chapter 1
Benchmarking methodology

1.1 SPECjvm2008

To compare performance of different profilers we need a benchmarking application. Measuring application performance is not as straightforward as it seems. Just-In-Time (JIT) compilation and others JVM optimizations need to be taken into account. So for this purpose one of the most commonly used and a proven solution was chosen – SPECjvm2008 benchmark suite.

SPECjvm2008 benchmark suite was developed by Standard Performance Evaluation Corporation (SPEC) and is freely available from the SPEC website [2]. This suite is designed for measuring the performance of a Java Runtime Environment (JRE) in near-to-real conditions. Before every test SPEC does a short warm-up run to obtain all benefits of JIT compilation. SPECjcm2008 measures performance in operation per second (op/s) and calculates composite result as the geometric mean of benchmarks scores. It contains several real life applications and benchmarks, focusing on core java functionality. The following summary mostly cites the official SPEC documentation:

compiler	This benchmarks uses OpenJDK java compiler to compile a lot of *.java source files. The source files are read from memory rather from disk, using a virtual file system. This benchmark heavily stresses the memory subsystem.
compress	This benchmark compresses data, using a modified LZW algorithm.
crypto	This benchmark encrypts and decrypts sample data using AES, DES and RSA algorithms.
derby	This benchmark uses an open-source database written in pure Java. The purpose of this benchmark is to stress `BigDecimal` library and database logic.
mpegadudio	This benchmark decodes mp3 audio files using JLayer, a LGPL mp3 library. It does a huge amount of floating point computations.
scimark	This benchmark was developed by NIST and is widely used by the industry as a floating point benchmark. There are two version of the test, "small" (dataset 512 Kbytes), which stresses CPU and "big" (dataset 32 Mbytes) which stresses memory subsystem [3]. Included sub-benchmarks: • `scimark.fft` – implementation of Fast Fourier Transform. • `scimark.sor` – Jacobi Successive Over-relaxation – a method to solve linear systems of equations. • `scimark.monte_carlo` – Monte Carlo integration approximates the value of Pi Algorithm stresses random-number generators, synchronized function calls, and function inlining. There is no big version for this sub-benchmark.

	• `scimark.spares` – Sparse matrix multiplication. • `scimark.lu` – dense LU matrix factorization.
serial	Benchmark, which serializes and deserializes objects.
startup	This benchmark starts all other benchmarks for one operation. Needed for enabling some JVM optimizations.
sunflow	This benchmark tests graphics visualization using an open-source, internally multi-threaded global illumination rendering system.
xml	This benchmark tests XML verification and transformation performance. XML validation tests use `javax.xml.validation` API by validating *.xml sample files against XML schema (*.xsd files). XML transformation tests exercises implementation of `javax.xml.transform` and related API by applying style sheets (*.xsl files) to XML documents. This benchmark heavily stresses memory subsystems.

1.2 Memory usage measurement

SPEC doesn't have any memory usage metrics, so for measuring native and heap memory usage a lightweight JVM TI [1] agent was developed. The agent is written in pure C, using JVM TI API. The agent is started in a separate thread on the JVM startup and periodically logs amount of consumed memory to a CSV file. After that, any spreadsheet program, for example by Microsoft Excel, can process the resulting file.

For collecting heap usage statistics the agent calls `java.lang.Runtime.totalMemory()` and `java.lang.Runtime.freeMemory()` methods [4]. Agent also tracks memory consumed by the whole JVM. For this purpose the native WinApi `GetProcessMemoryInfo()`function is used [5]. Overhead created by this agent is constant and extremely small, so it does not affect the SPECjvm2008 benchmarking result.

1.3 System information

On every SPECjvm2008 run the maximum heap size was increased to 1024 Mbytes by setting the JVM startup parameter `-Xmx1024m`. All benchmarks were performed on the following system configuration:

Model:	Notebook MSI GT623x
CPU:	Intel Core 2 Duo P7350 2.00Ghz
RAM:	4 GB DDR2
Operation system:	Window 7 service pack 1 32-bit
JRE version:	1.6.0_26

For further comparisons a clean run without any attached profiler was performed. Benchmark detailed results with overall score 21.07 op/s are shown on Figure 1-1:

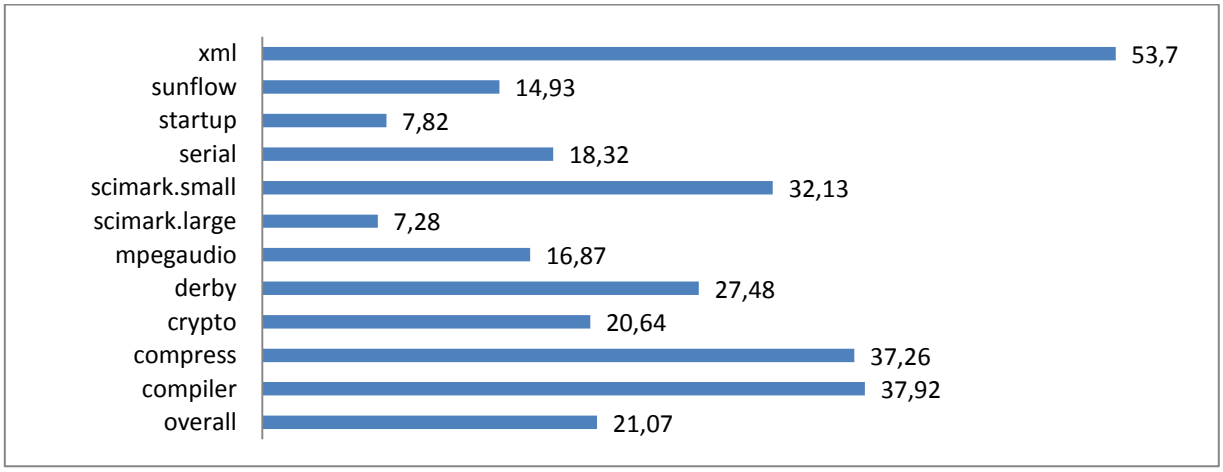


Figure 1-2 SPECjvm2008 results

Memory usage is shown on Figure 1-2, where working set size is the total amount of memory allocated by Windows OS to running JVM instance. Average heap usage was **311 Mbytes**, with peak usage 747 Mbytes. Average working set size was **590 Mbytes**, with peak value of 888 Mbytes. Working set size is the total amount of memory allocated by Windows OS to running JVM instance.

Note that on time intervals with high object allocation density, the heap usage rapidly changing. It is explained by the frequent triggering of the Garbage Collector (GC). In contrast, the time intervals, when the GC was not triggered are the `scimark.sor.small` and `scimark.monte_carlo` benchmarks. It is logical to assume that memory allocation trackers should not affect the performance of these benchmarks.

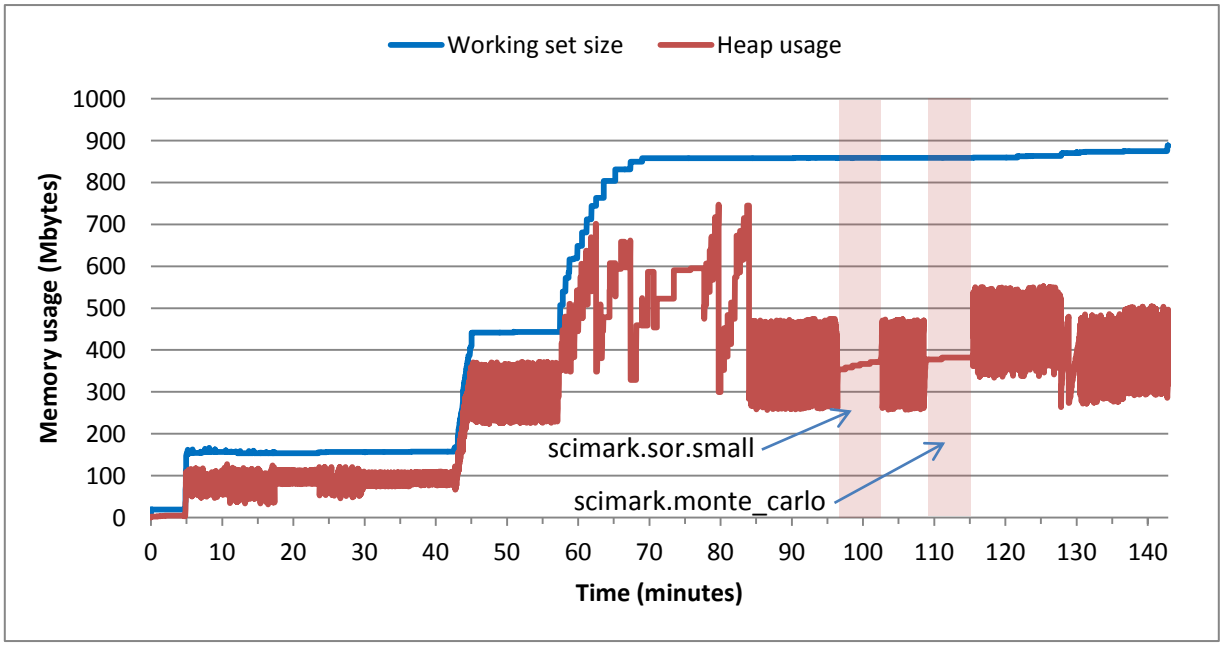


Figure 1-3 SPECjvm2008 memory usage

Chapter 2
HPROF

HPROF is a command line profiler shipped with Java Development Kit (JDK). HPROF source code is provided under a slightly modified BSD license. Oracle has positioned HPROF as a sample JVM TI agent, so all source code is well structured and commented.

HRPOF output format is used by many third-party memory analyzing utilities, such as Eclipse MAT and Heap Walker. For this reason, the majority of commercial and open-source profilers support export to HPROF report format.

For this work we are interested in memory allocations tracking capabilities of HPROF. To start collection of statistics about memory allocations run HPROF with heap=sites parameter:

```
> java -Xrunhprof:heap=sites javaClass
```

In this mode HPROF tracks the amount of memory that was allocated in a specific site. A site, in the terminology of HPROF, is a unique stack trace of a fixed depth. After JVM finishes, HPROF writes the collected data to a file called *java.hprof.txt*. For every allocation site, there are class name of allocated object, amount of used memory and allocations count. A part of the profiler output:

```
rank  self  accum     bytes  objs  bytes  objs trace name
    1  4.16% 29.22%  355072 11096 355072 11096 305920 org.hsqldb.index.NodeAVL
    2  4.15% 37.53%  353472 11046 354944 11092 302346 java.math.BigInteger
```

To relate allocation sites to the source code there are stack traces that lead to the heap allocation. Another part of the *java.hprof.txt* file that contains the stack traces referred to by the top two allocation sites in the output shown above:

```
TRACE 305920:
       org.hsqldb.index.NodeAVL.<init>(NodeAVL.java:313)
       org.hsqldb.RowAVL.setNewNodes (RowAVL.java:345)
       org.hsqldb.RowAVL.<init>(RowAVL.java:156)
       org.hsqldb.persist.RowStoreAVLMemory.getNewCachedObject(RowStoreAVLMemory.ja
va:647)
TRACE 302346:
       java.lang.Number.<init>(Number.java:131)
       java.math.BigInteger.<init>(BigInteger.java:54)
       java.math.BigInteger.valueOf(BigInteger.java:163)
       java.math.BigDecimal.inflate(BigDecimal.java:215)
```

Each frame contains a method name, source file and line number. Maximum stack trace depth can be changed via ‘depth’ run parameter [6].

2.1 Implementation details

HPROF is written in pure C, excluding a small Java class `sun.hprof.Tracker`, which is used to handle calls from Java code. For allocation tracking HPROF does load-time bytecode instrumentation (BCI) by using JVM TI ClassFileLoadHook event and `RedifeneClass`

function. BCI is the ability to alter the Java virtual machine bytecode instructions, which comprise the target program. Inserted code is a standard bytecode, so JVM can run at full speed and get all benefits from Just-In-Time compilation (JIT). For allocations tracking HPROF does following bytecode instrumentations:

1. At the beginning of `java.lang.Object.<init>()` method, an `invokestatic` call to `sun.tool.hprof.Tracker.ObjectInit(object)` is injected.
2. On any newarray type opcode, the array object is duplicated on the stack and an `invokestatic` call to `sun.tools.hprof.Tracker.NewArray(object)` is injected.

Tracker's methods, in turn, return a call to native functions. HPROF gets the stack trace from the current thread and searches for this specific allocation site in a huge hash table. If the allocation site is found, then allocated object count and total memory size will be incremented, otherwise a new site record will be created and added to the hash table. On shutdown the profiler writes collected data from the hash table to *java.hprof.txt* file. An important detail is that the hash table is not designed for concurrent updates. For thread-safety it is locked on every access. Such a solution may lead to significant performance loss in multi-threaded applications.

2.2 Benchmark results

SPEC was run with these JVM parameters:

```
set SPEC_HOME=D:\Documents\diplom\spec
set MEMORY_MONITOR_DLL=D:\Documents\diplom\agent\memoryMonitor.dll
set OUT_FILE=log.txt
java ^
    -Xmx1024m ^
    -Xrunhprof:heap=sites ^
    -agentpath:%MEMORY_MONITOR_DLL% ^
    -Dspecjvm.home.dir=%SPEC_HOME% -Dspecjvm.result.dir=.^
    -jar %SPEC_HOME%\SPECjvm2008.jar -ikv > %OUT_FILE%
```

MemoryMonitor.dll is a JVMTI agent, which is described in section 1.2. Key `-ikv` is used to disable checksum verification. Checksum verification does not affect benchmark results, but adds irrelevant information about memory consumption. SPECjvm2008 run results with composite score 3.15 are shown on Figure 2-1:

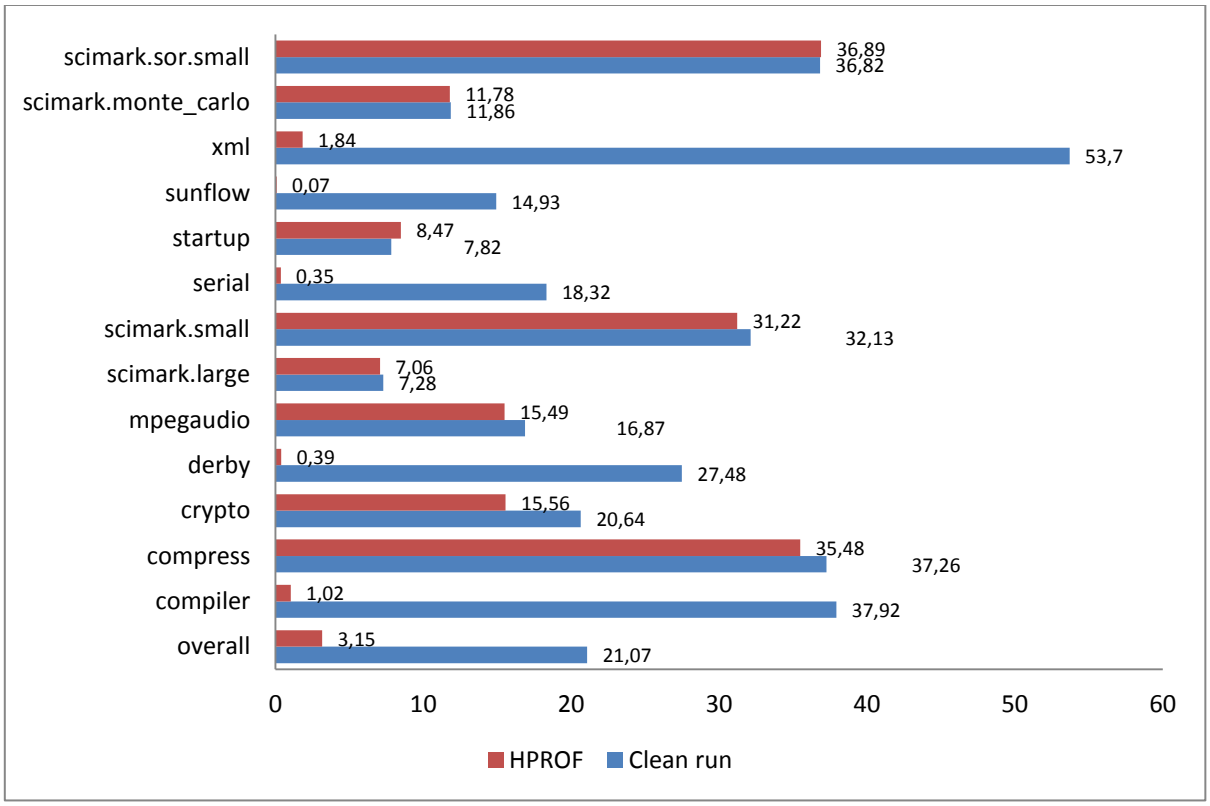
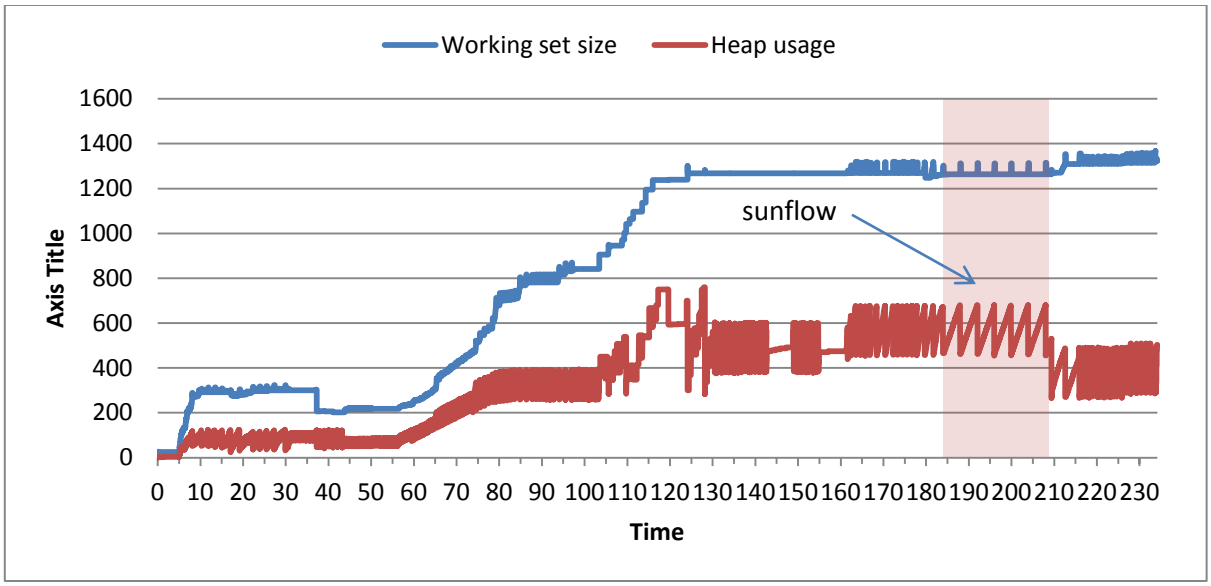


Figure 2-1 SPECjvm2008 benchmark results with and without HPROF

As expected `scimark.sor.small` and `scimark.monte_carlo` sub-benchmarks didn't suffer any performance loss from attached allocations tracker. These tests do not allocate any significant number of objects and thus run at full speed. Scimark, compress and mpegaudio benchmarks also show almost the same results as in the reference run. These benchmarks stress the CPU more than memory.

In other tests, performance overhead has been disastrous. In the test sunflow, which renders 3d scene in multiple threads, suffered a performance drop of more than 200(!) times, from 14.93 op/s to 0.07 op/s. There are a few reasons for such a low result:

- During run Sunflow creates more than 200 000 000 instances of `org.sunflow.core.Ray` class. Which strongly stress the memory subsystem.
- Sunflow uses multiple threads for rendering. HPROF obtains object monitor on every allocation, which lead to performance loss.



2-2Working set size and heap usage during SPEC run with attached HPROF

Memory usage for this benchmark is shown on Figure 2-2. Average heap usage was **349 Mbytes**, which is 12% more than in reference run (311Mbytes). Average working set size was **876 Mbytes**, with peak value of **1367 Mbytes**, which is almost 1.5 times more. Such huge memory consumption is caused by the hash table in native memory, where HRPOF holds collected data.

Chapter 3
NetBeans profiler

This profiler's development was started as a research project by Sun Laboratories in 2003. Profiler uses dynamic byte code instrumentation, which in those days was an experimental technology [7]. Since the regular Java Virtual Machine didn't have the needed bytecode instrumentation support, JFluid used to run on a modified virtual machine, also known as JFluid VM. Later Sun announced what JFluid will be integrated with Sun open source NetBeans IDE. JFluid was renamed to just NetBeans profiler. Starting from NetBeans version 6.0 (December, 2007) profiler's source code is open and available under CDDL and GPLv2 with classpath exception licenses [8].

NetBeans profiler consists of server and client side. They communicate with each other by a socket connection. It allows you to run the profiler on a remote host. Server side is a JVM TI client, which can be attached to an application by "-agentpath" JVM startup parameter.

Client side is a NetBeans module, which collects data from the agent and provides a graphic user interface. For memory profiling there are only few settings:

- **Record garbage collection.** In this mode profiler records the full object lifecycle, from their allocation until garbage collection.
- **Track only every n-th object**. In documentation I found what this option sets the interval for stack sampling. Setting the value to 10, means that for each class only every 10th object allocation will be recorded completely, with stack trace. This option can substantially increase the profiling performance.
- **Record stack trace for allocations.** Same functionality as in HPROF. Allows profiler to record allocation sites. Also, it is possible to set the maximum stack trace depth.

3.1 Implementation details

NetBeans profiler server side is actually a JVM TI agent. The profiler is almost entirely written in Java. Just a small part of the code is written in C.

Just as HPROF, NetBeans uses `ClassFileLoadHook` event and `RedefineClass` function to do byte code instrumentation.

Unlike HPROF, NetBeans profiler doesn't inject any byte code to `java.lang.Object<init>` method. Instead, it just writes a static call to `org.netbeans.lib.profiler.server.ProfilerRuntimeObjAlloc.traceObjAlloc` method after every new object or array creation

HPROF gets stack trace only on every 10-th (can be changed in settings) object allocation of every class. During a single run there are thousands of object allocations, so probability that some allocation site will be skipped is extremely small. On the other hand this optimization leads to a significant performance gain.

3.2 Benchmark results

SPECjvm2008 was run with the NetBeans 7.1.1 profiler. For better comparison, the profiler was run with parameters most similar to the HPROF run configuration:

- Track object allocations only.
- Record stack traces for allocations.
- Limit stack to 3 frames.
- Record only every 10-th object. This option was leaved by default value 10. In the documentation is it stated, what this value is a preferable value and doesn't affect on profiling accuracy.

Target JVM was run with following parameters:

```
set SPEC_HOME=D:\Documents\diplom\spec
set MEMORY_MONITOR_DLL=D:\Documents\diplom\agent\memoryMonitor.dll
set NETBEANS_AGENT="C:\Program Files\NetBeans
7.1.1\profiler\lib\deployed\jdk15\windows\profilerinterface.dll=\"C:\Program
Files\NetBeans 7.1.1\profiler\lib\"",5140
set OUT_FILE=log.txt
java ^
    -Xmx1024m ^
    -agentpath:%MEMORY_MONITOR_DLL% ^
    -agentpath:%NETBEANS_AGENT% ^
    -Dspecjvm.home.dir=%SPEC_HOME% -Dspecjvm.result.dir=.^
    -jar %SPEC_HOME%\SPECjvm2008.jar –ikv > %OUT_FILE%
```

Path to the TPTP agent is taken from NetBeans documentation.

At the first attempt, the profiler has worked for an hour and crashed with `JVMTI error 60: malformed class file`. Error occurs on `mpegaudio` benchmark and was caused by a damaged instrumented class `javazoom.jl.decoder.huffcodetab`. A lot of reports [9], [10], [11] of this issue can be found on the NetBeans bugtracker, but no solution exists as of now. The same error is received while using NetBeans 6.9.1. As a result of this issue, this test was run without the mpegaudio benchmark.

```
set SPEC_HOME=D:\Documents\diplom\spec
set MEMORY_MONITOR_DLL=D:\Documents\diplom\agent\memoryMonitor.dll
set NETBEANS_AGENT="C:\Program Files\NetBeans
7.1.1\profiler\lib\deployed\jdk15\windows\profilerinterface.dll=\"C:\Program
Files\NetBeans 7.1.1\profiler\lib\"",5140
set OUT_FILE=log.txt
java ^
    -Xmx1024m ^
    -agentpath:%MEMORY_MONITOR_DLL% ^
    -agentpath:%NETBEANS_AGENT% ^
    -Dspecjvm.home.dir=%SPEC_HOME% -Dspecjvm.result.dir=.^
    -jar %SPEC_HOME%\SPECjvm2008.jar ^
    startup compiler compress crypto derby scimark serial sunflow xml ^
    –ikv > %OUT_FILE%
```

Figure 3-1 shows benchmark detailed result with composite score 11.09. This is almost 2 times better than of HPROF.

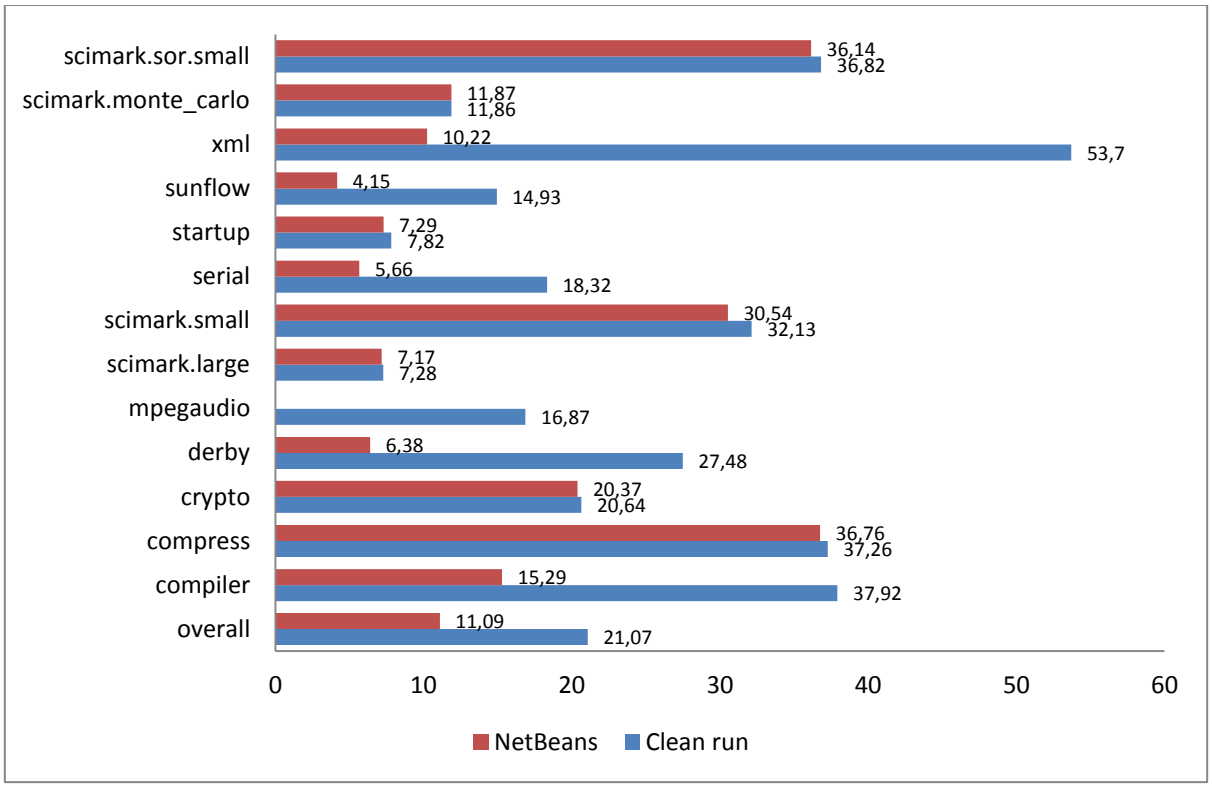


Figure 3-1 SPECjvm2008 results with attached NetBeans profiler

Memory usage for this benchmark is shown on Figure 3-1. The biggest overhead was on tests, which allocate a lot of objects. This is `xml`, `sunflow`, `serial` and `compiler` benchmarks. Tests `scimark.sor.small` and `scimark.monte_carlo` show almost the same results as on reference run.

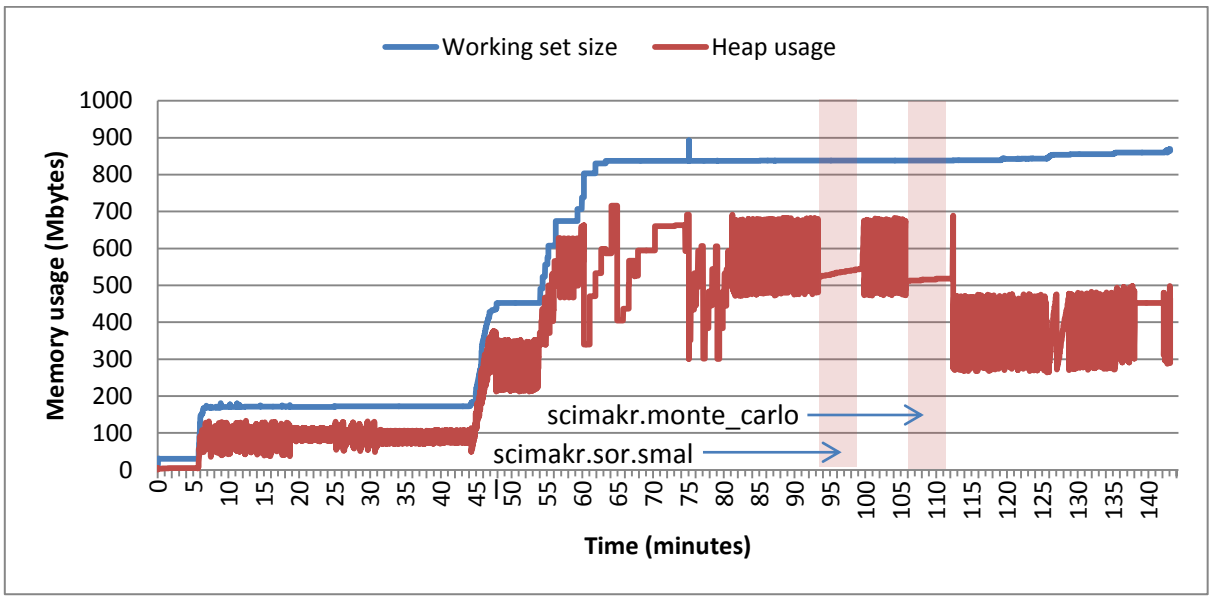


Figure 3-2 Working set size and heap usage during SPECjcm2008 run with attached NetBeans profiler

Chapter 4
Eclipse TPTP

Eclipse Testing and Performance Tools Platform (TPTP) is developed as part of Eclipse open-source project. TPTP is a set of supplying platform frameworks and services that allow software developers to build test and performance tools [9].

TPTP addresses the entire test and performance life cycle, from early testing to production application monitoring, including test editing and execution, monitoring, tracing and profiling, and log analysis capabilities. In this chapter the TPTP profiler subproject is reviewed. Profiler source code is distributed under the Eclipse Public License [10] and is available to download from public Eclipse CVS repository.

Eclipse profiler was designed as part of the Eclipse IDE, but the last supported IDE version is Helios, which was released in June 2012. There were plans to port the project to the latest Eclipse version, but as of now the project is suspended due to a lack of developers.

4.1 Implementation details

Profiler's server runtime is written in the C++ programming language. It attaches to JVM versions 5 or 6 as JVM TI agent. For the older versions of virtual machine the TPTP profiler can use JVM Profiling Interface (JVM PI). This profiler can be run in server or in standalone mode. In server mode it communicates with the IDE via socket connections. In standalone mode it just dumps all collected data to a file.

Unlike all previously reviewed profilers TPTP does not use any byte code instrumentations. Instead, TPTP profiler registers callbacks to these JVM TI events:

```
jvmtiEventObjectFree ObjectFree;
jvmtiEventVMObjectAlloc VMObjectAlloc;
```

Contrary to expectations, the agent profiler does not perform any complex data processing operations. It just logs every event and sends all collected data to the IDE. In standalone mode it writes all logged events to a XML file.

4.2 Running TPTP profiler in server mode

Profiler can be downloaded from the project website with a configured Eclipse Helios IDE. I used the latest TPTP version 4.7.2. The latest version (4.7.2) was used for the tests. The profiler can be run directly from the Eclipse IDE and provides a convenient user interface for profiling third-party jar files. Profile configuration interface can be started via the following menu option:

```
Run -> Profile configurations...
```

An external Java application profile was created, with parameters matching those of HPROF and NetBeans tests as closely as possible. The whole configuration is following:

```
Host:
    localhost[10002]
```

```
Main:
    Main class: spec.harness.Launch
    Class path: D:\Documents\diplom\spec\SPECjvm2008.jar
Arguments:
    Program arguments: -ikv
    JVM arguments:
        -Xmx1024m
        -agentpath:D:\Documents\diplom\agent\memoryProfiler.dll
        -Dspecjvm.home.dir=D:\Documents\diplom\spec
        -Dspecjvm.result.dir=.
    Working directory: D:\Documents\diplom\results\tptp
Profile Settings:
    Filters:
        * * INCLUDE
    Memory Analysis: record stack traces
```

The default port value was used, heap size was increased to 1024 Mbytes, class filters were disabled and profiler records stack traces enabled for every allocation.

After start Eclipse with TPTP profiler has been working for half an hour and has crashed with an `OutOfMemoryError`. Seems what Eclipse IDE did not have enough memory for processing of all collected data. The maximum heap size was increased to 1 Gigabyte by changing `-Xmx384` startup option to `-Xmx1024` in an *eclipse.ini* configuration file. Unfortunately this did not resolve the issue. At the next try Eclipse crashed with same error. Increasing heap size to 1576 Mbytes made the profile work for approximately one hour and then subsequently crash due to an unknown error. Eclipse logs did not help determine the root of the problem.

Further increase of memory size was not applicable due to memory constrains on the test system. Using older versions of both the virtual machine (JVM5) and TPTP (4.7.1) did not help in resolving the issue.

4.3 Running TPTP profiler in standalone mode

Fortunately there exists a standalone mode. In this mode the profiler, attached to a JVM, just writes raw data to a XML file. Then the resulting file can be analyzed by utilities shipped with TPTP. The following run parameters were taken from the official profiler documentation:

```
set JAVA_HOME=C:\Program Files\Java\jdk1.6.0_20
set SPEC_HOME=D:\Documents\diplom\spec
set SPEC_PARAMS=-Dspecjvm.home.dir=%SPEC_HOME% -Dspecjvm.result.dir=.
set MEMORY_MONITOR_DLL=D:\Documents\diplom\agent\memoryMonitor.dll
set TPTP_PATH= D:\TPTP\agent\plugins\org.eclipse.tptp.javaprofiler\
set TPTP_AGENT=JPIBootLoader.dll
set TPTP_PARAMS= JPIAgent:server=standalone,format=xml;HeapProf:allocsites=true
set OUT_FILE=log.txt
java ^
    -Xmx1024m ^
    -agentpath:%MEMORY_PROFILER_DLL% ^
    -agentpath:%TPTP_PATH%%TPTP_AGENT%=%TPTP_PARAMS% ^
    %SPEC_PARAMS% -jar %SPEC_HOME%\SPECjvm2008.jar ^
    -ikv > %OUT_FILE%
```

Again, the profiler crashed on benchmark `startup.compiler.sunflow` with the following error:

```
EXCEPTION_UNCAUGHT_CXX_EXCEPTION (0xe06d7363) at pc=0x756bd36f, pid=4400, tid=7476
```

On the next run it passed all startup tests and crashed on the first iteration of `compiler.compiler` sub-benchmark. Seems, the same error prevents SPECjvm2008 from execution with the Eclipse IDE graphical interface.

Neither inspecting the TPTP project mailing lists and forum for a solution, nor changing run parameters helped to resolve this issue. Errors occurred only on `compiler` and `derby` benchmarks, so it was decided to exclude these tests for the current profiler.

4.4 Benchmark results

The main conclusion of the test - TPTP profiler is unstable and may lead to a crash of the entire application. This problem may be specific to the test system, but results suggest one should be wary of possible issues when using this profiler for actual work.

SPECjvm2008 results with attached TPTP profiler a shown on Figure 4-1. As usual, sub-benchmarks `scimark.monte_carlo` and `scimark.sor.small` don't suffer any performance loss. On other tests TPTP showed the worst results among all previously reviewed allocation trackers. Sunflow benchmark result was about **300 times** less than on reference run. `Xml` benchmark fared just as poorly – 100 times less than on reference run.

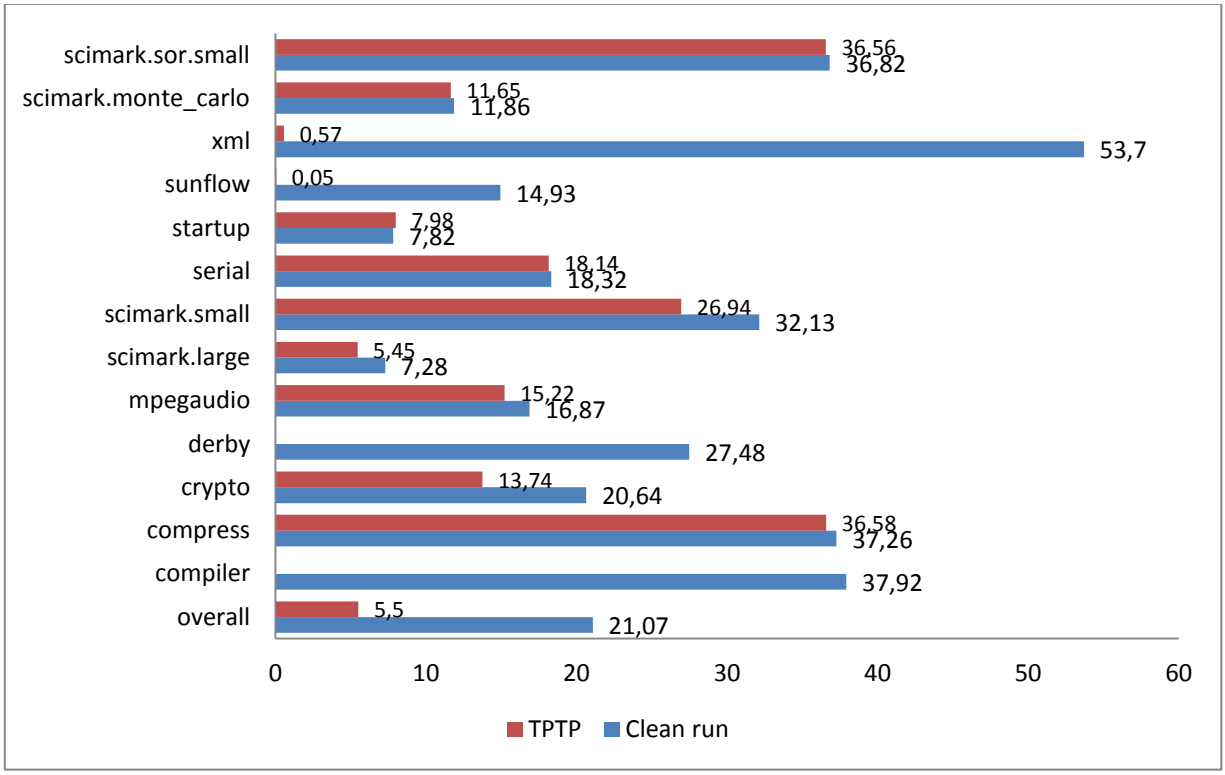


Figure 4-1 SPECjvm2008 benchmarks result with attached TPTP memory profiler

Memory usage for this benchmark is shown on Figure 4-2. Unfortunately, no conclusion regarding TPTP memory overhead can be made, since it was performed on an incomplete set of benchmarks. Note that sub-benchmarks `scimark.sor.small` and `scimark.monte_carlo` a clearly seen on the memory consumption chart.

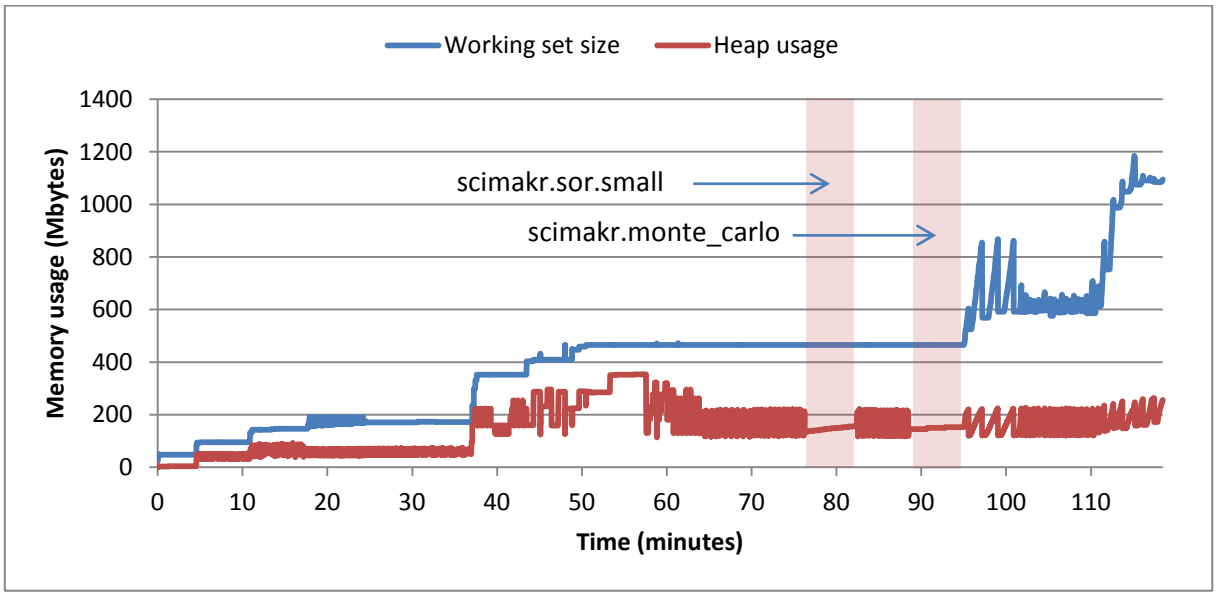


Figure 4-2 SPECjvm2008 memory consumption with attached TPTP profiler

Chapter 5
Allocation Instrumenter project

Unlike the other reviewed solutions, Allocation Instrumenter is not a full functional profiler, but a specialized tool for tracking object allocation events only [11].

The Allocation Instrumenter is a written using the `java.lang.instrument` API [12] and ASM Java byte code manipulation framework [13]. Each allocation in a Java program is instrumented and can be handled by a user-defined callback.

In order to write allocation tracking code, you have to implement the Sampler interface and pass an instance of that to `AllocationRecorder.addSampler()`:

```
AllocationRecorder.addSampler(new Sampler() {
    public void sampleAllocation(int count, String desc, Object newObj, long size)
{
        //your code here
    }
});
```

The source code of Allocation Instrumenter is freely available under Apache License 2.0 [14].

5.1 Implementation details of Allocation Instrumenter

Allocation Instrumenter uses java.lang.instrument package to get the system class loader and redefine the loaded Java classes. After every new object or array creation, it injects a call to the following method:

```
com.google.monitoring.runtime.instrumentation.
    AllocationRecorder.recordAllocation(Class class, Object object)
```

In this method instrumenter gets the class name and calculates the size of the object. After that, it iterates over array of the registered allocation samplers. Then it invokes the method `sampleAllocation` for every registered sampler.

In order to reduce the amount of expensive object size calculations `AllocationRecorder` stores the sizes of 100000 last encountered classes in a concurrent hash map.

In order to prevent recursive calls of `recordAllocation` method there is `ThreadLocal<Boolean>` flag. Thread local variable is unique for every thread, so it is possible to quickly check whether the method has already been invoked in this thread and prevent recursive calls.

5.2 Implementation of allocation tracker

As you can see Allocation Instrumenter offers a simple API for allocation tracking. This section presents a simple allocation tracker, written using the Allocation Instrumenter API, which works on the same principles as HPROF. It tracks every allocation, gets the last 4 frames from the current thread stack and stores it in the hash map. If a specific class with stack trace is already present in hash map, it should increment counter. So after run we get all allocation sites with statistics in one large hash table.

First we define how allocations should be stored:

```
private final class AllocationEntry {
    public String desc; // class name
    public StackTraceElement frame1;
    public StackTraceElement frame2;
    public StackTraceElement frame3;
    public StackTraceElement frame4;
    public long size = 0; // total size of allocated objects
    public int count = 0; // number of allocated object
}
```

String `desc` is a class name of the allocated object. Variables `frame1`, `frame2`, `frame3` and `frame4` hold the last frames of the stack trace. It is better to use arrays for code readability purposes, but in attempt to avoid new array for every `AllocationEntry` instance they are not used. 64-bit integer variable `size` stores the total size of allocated objects. The integer `count` stores the number of allocated objects

To store objects in the hash table a hash function, that depends on class name and stack trace only, is needed.

```
@Override
public int hashCode() {
    return desc.hashCode()^frame1.hashCode()^frame2.hashCode()
                          ^frame3.hashCode()^frame4.hashCode();
}
```

Additional requirement of storing data in a hash table is an equality function, where the general rule is that two equal objects must have the same hash value. Obviously the equality function must be commutative and transitive.

```
@Override
public boolean equals(Object o) {
    if (o == this) return true;
    if (o instanceof AllocationEntry) {
        AllocationEntry e = (AllocationEntry)o;
        return desc.equals(e.desc)
            && frame1.equals(e.frame1)
            && frame2.equals(e.frame2)
            && frame3.equals(e.frame3)
            && frame4.equals(e.frame4);
    } else {
        return false;
    }
}
```

Finally, the following code is used for allocation tracking:

```
private final HashMap<AllocationEntry, AllocationEntry> map =
    new HashMap<AllocationEntry, AllocationEntry>(10000);
private final AllocationEntry DUMMY_ENTRY = new AllocationEntry();
```

```
public void sampleAllocation(int count, String desc, Object newObj, long size) {
    synchronized (lock) {
        StackTraceElement[] trace = Thread.currentThread().getStackTrace();
        DUMMY_ENTRY.desc = desc;
        DUMMY_ENTRY.frame1 = trace[START_FRAME + 0];
        DUMMY_ENTRY.frame2 = trace[START_FRAME + 1];
        DUMMY_ENTRY.frame3 = trace[START_FRAME + 2];
        DUMMY_ENTRY.frame4 = trace[START_FRAME + 3];

        AllocationEntry entry = map.get(DUMMY_ENTRY);
        if (entry == null) {
            entry = DUMMY_ENTRY.clone();
            map.put(entry, entry);
        }

        entry.count++;
        entry.size+=size;
    }
}
```

All allocations are stored in an instance of `HashMap`. This is a commonly used non-synchronized hash table implementation in Java. In order to avoid creating object for every search, there exists a static `DUMMY_ENTRY` object. The last few stack trace elements contain information about the allocation tracker methods and aren't relevant, theses frames are skipped. The method's content is synchronized on a global lock to avoid possible race conditions.

5.3 Benchmarking result

The created profiler was run using the following batch script:

```
set SPEC_HOME=D:\Documents\diplom\spec
set MEMORY_MONITOR_DLL=D:\Documents\diplom\agent\memoryMonitor.dll
set OUT_FILE=log.txt
java ^
    -Xmx1024m ^
    -Xrunhprof:heap=sites ^
    -agentpath:%MEMORY_MONITOR_DLL% ^
    -javaagent:allocation.jar
    -Dspecjvm.home.dir=%SPEC_HOME% -Dspecjvm.result.dir=.^
    -jar %SPEC_HOME%\SPECjvm2008.jar –ikv > %OUT_FILE%
```

Execution of the SPECjvm2008 benchmark suite took much more time than usual – over 13 hours. Benchmarks results with composite score 1.43 is shown on Figure 4-2, which is the worst result among all reviewed allocation trackers.

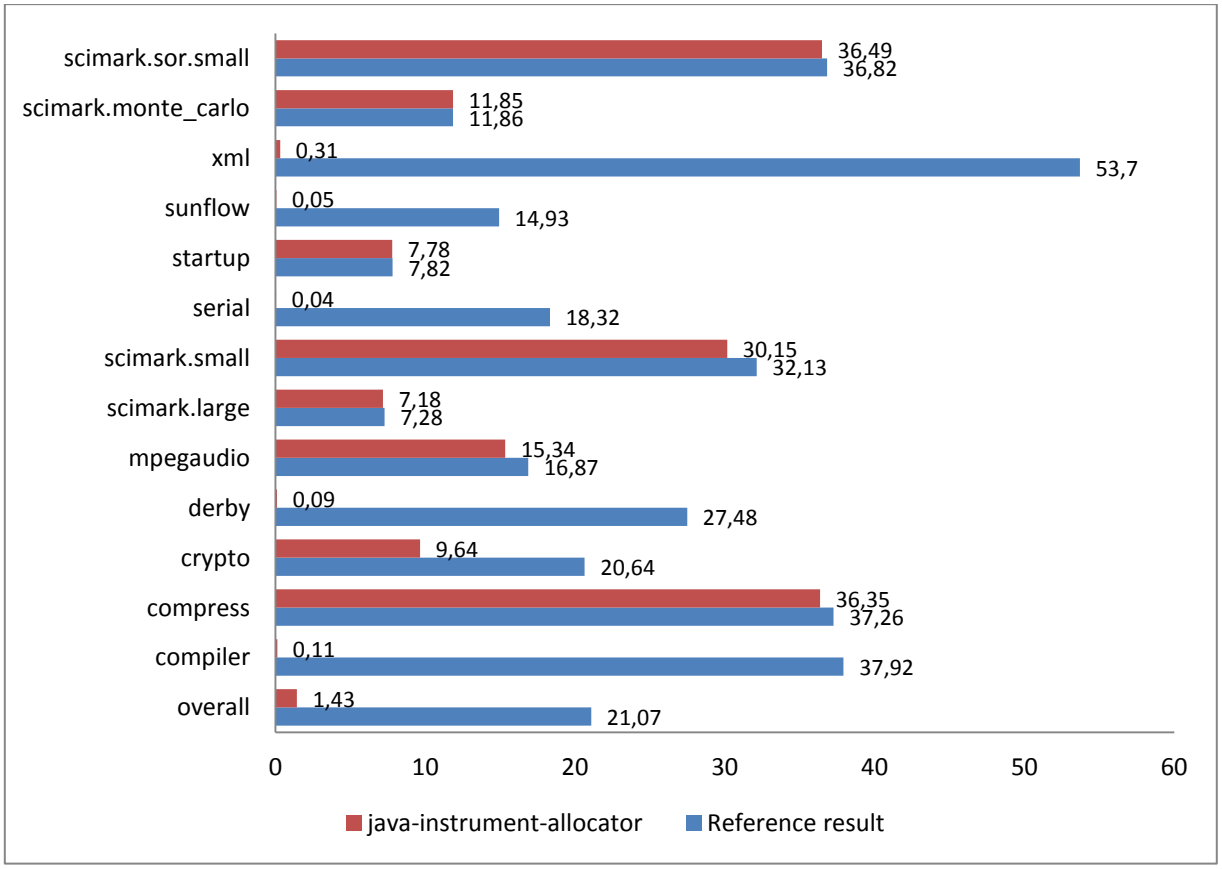


Figure 5-1 SPECjvm2008 results with attached allocation tracker

SPECjvm2008 memory consumption with attached Allocation Instrumenter is shown on Figure 5-1. Despite the fact that all the collected data is stored on the heap, average heap usage of **268 Mbytes** was even less than in the reference run (311 Mbytes). Seem that due to a longer run duration GC managed to collect unused objects more efficiently and keep memory consumption on a low level.

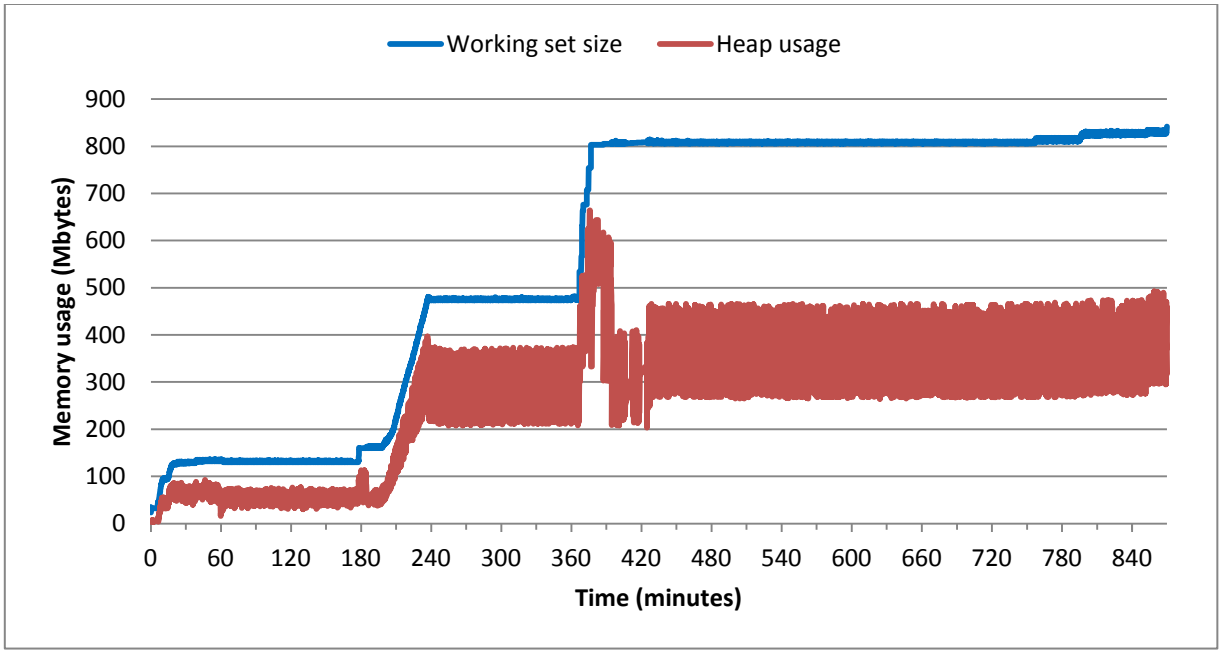


Figure 5-2 SPECjvm2008 memory consumption with attached Allocation Instrumenter

Summary

We looked at several open-source memory allocations trackers. Two of them, HPROF and NetBeans profiler are developed by The Oracle Corporation. The third TPTP profiler was developed for the Eclipse IDE, but now this project is not active due a lack of developers. The last one is Allocation Instrumenter, which is a written in pure java by Google developers.

The SPECjvm2008 benchmark suite was used for measuring performance and a JVMTI agent was developed for monitoring memory consumption overhead.

Two of four allocation trackers had stability problems. NetBeans incorrectly instrumented `javazoom.jl.decoder.huffcodetab` class from mpegaudio benchmark. Eclipse TPTP profiler crashed several times, on different benchmarks, so a full SPECjvm2008 run was not possible with it.

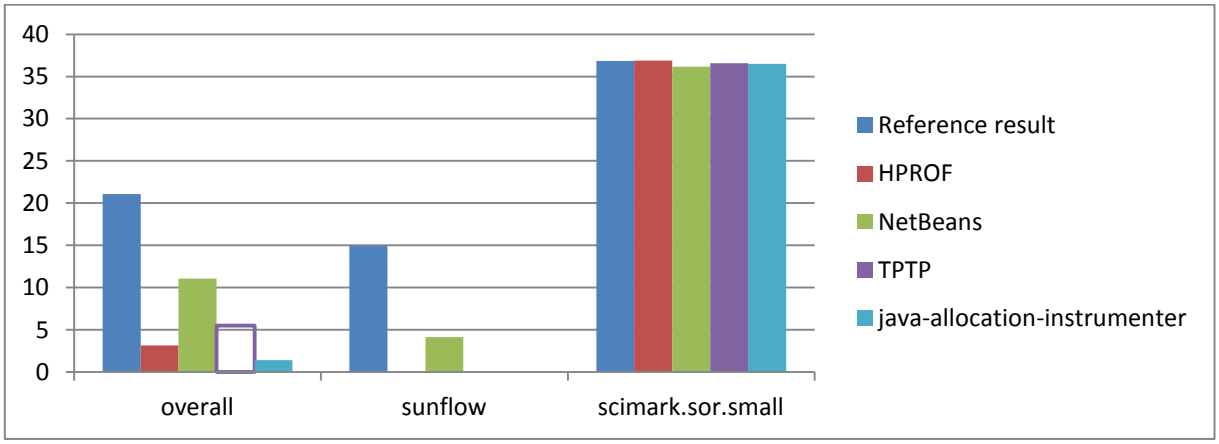


Figure 6-1 SPECjvm2008 results

SPECjvm2008 composite results are shown on Figure 6-1. NetBeans profiler is the most technological advanced and the most efficient of all considered solutions. It's not surprising, as it was written by Sun's engineers in parallel with Java Virtual Machine development and is still in active development.

Most problematic benchmark is `sunflow`, which executes about 300 times slower with an attached TPTP profiler. HPROF's and Allocation Instrumenter results are not much better. Only NetBeans profiler was able to efficiently work with it.

Table 6-1 Memory usage during SPECjvm2008 run in Mbytes

	Heap		Working set	
	Average	Peak	Average	Peak
Reference	311	747	590	888
HPROF	349	760	875	1367
NetBeans	347	715	590	893
TPTP	irrelevant	irrelevant	irrelevant	irrelevant
java-allocation-instrumenter	268	664	580	841

Allocation tracking did not have much of an impact on memory consumption (see Table 6-1). Only in the case of HRPOF there was a significant increase of native memory usage. TPTP profiler crashes at compiler and derby benchmarks, so its memory usage statistics are not relevant.

All solutions, except TPTP, use byte code instrumentation. They insert a call to a tracker method after every object or array creation. Eclipse TPTP use JVMTI event, but it's most likely an outdated legacy solution from the times of JVM 1.4.2 and JVM Profiling Interface (JVMPI).

Google's Allocation Instrumenter was of particular interest, as it allows to track allocations without a single line of native code, but there are still unresolved problems with performance. Future work involves figuring out the bottlenecks involved with the use of this technology and providing a solution.

Mälueraldamiste jälgijate võrdlemine *JVM*'is

Bakalauresetöö 6 EAP
Viktor Karabut
Resümee

Mälueraldamiste jälgija (Memory allocation tracker) on tööriist, mis registreerib objektide loomisi *JVM*'is. Tavaliselt, mälueraldamiste jälgija on profileerija alamosa. Sams eksisteerivad ka eraldiseisvad lahendused. Töö põhieesmärk on läbi vaadata ja võrrelda olemasolevate avatud lähtekoodiga mälueraldamiste jälgijaid.

Selleks, et mõõta mälueraldamiste jälgijate effektivsus kasutasime SPECjvm2008 jõudlustestide komplekt. Mälu kasutamise mõõtmiseks oli kirjatud oma JVM TI agent, mis perioodiliselt kirjutab kasutatud mälu suurus CSV faili.

Töö käigus olid läbi vaadatud ja testitud selliseid mälueraldamiste jälgimise lahendusi:

- HRPOF – lihtne käsurea kasutajaliidesega profileerimise tööriist, mis pakutakse JavaDevelopment Kit (JDK) koosseisus.
- NetBeans profileerija – varem see oli eraldiseisav avatud lähtekoodiga uurimis proekt nimega JFluid. Praegu see on NetBeansi osa.
- TPTP profileerija – profileerija, mis kasutatakse Eclipse IDE's.
- Project Allocation Instrumenter – mälu eraldamiste jälgija Google'st. On kirjutatud puhtas Javas. Kasutab java.lang.instrument API ja ASM raamistikku baitkoodi analüüsimiseks ja manipuleerimiseks.

Neljast kahel mäluereldamiste jälgijatel olid probleemid stabiilsusega. NetBeans valesti muutus javazoom.jl.decoder.huffcodetab klassi baitkoodi, mille pärast ei saanud *mpegaudio* testi käivitada. Eclipse TPTP profileerija ei suutnud edukalt oma tööd lõpetada mitmel korral erinevate jõudlustestide käivitamise ajal.

NetBeans profileerija on tehniliselt kõige arenenum ja kõike tõhusam mäluerdlamiste jälgimise lahendus. See ei ole üllatav, kuna *Sun*'i insenerid kirjutasid seda parrallelselt Java virtualse masina arenguga.

Kõige problemaatilisem mälueraldamiste jälgimise mõttes oli *sunflow* jõudlustest. Koos TPTP profileerijaga ta jooksis umbes 300 korda aeglasemalt. *HPROF*'i ja *Allocation Instrumenter*'i tulemused ei olnud määrkimväärselt paremad. Ainult NetBeans profileerija said enam-vähem efektiivselt töötada sellel testil.

Kõik läbi vaadatud lahendused, välja arvatud TPTP, kasutavad mälueraldamiste jälgmiseks baitkoodi manipuleerimist (ByteCode Instrumentation, BCI). Nad lisavad oma jälgimise meetodi väljakutse pärast iga objekti loomise baitkoodi (opcode `new`). TPTP oma tööks kasutab JVM TI sündmuseid.

Erilist huvi pakkub Allocation Instrumenter Google'st. See raamistik võimaldab kirjutada mälueraldamiste jälgijat puhtas Java keeles. Potensiaalselt see teeb lahendus platvorimst

sõltumatuks. Kuid veel eksisteerivad lahendamata probleeme jõudlusega. Minu tulevikuplaanis on aru saada, kas saab teha mälueraldamiste jälgimist

Bibliography

[1] Oracle Corporation, "JavaTM Virtual Machine Tool Interface (JVM TI)," [Online]. Available: http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/. [Accessed 15 May 2012].

[2] SPEC, "SPECjvm2008 Benchmark," [Online]. Available: http://www.spec.org/jvm2008/. [Accessed 15 May 2012].

[3] R. Pozo and B. Miller, "SciMark v2.0," [Online]. Available: http://math.nist.gov/scimark2/. [Accessed 15 May 2012].

[4] Oracle Corporation, "Java SE6 reference: java.lang.Runtime," [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/lang/Runtime.html. [Accessed 15 May 2012].

[5] Microsoft, "MSDN library: GetProcessMemoryInfo function," [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms683219.aspx. [Accessed 15 May 2012].

[6] Oracle Corporation, "HPROF: A Heap/CPU Profiling Tool in J2SE 5.0," [Online]. Available: http://java.sun.com/developer/technicalArticles/Programming/HPROF.html. [Accessed 15 May 2012].

[7] NetBeans Community, "The Foundation - JFluid Technology," [Online]. Available: http://profiler.netbeans.org/jfluid.html. [Accessed 15 May 2012].

[8] NetBeans Community, "NetBeans license," [Online]. Available: http://netbeans.org/about/legal/license.html. [Accessed 15 May 2012].

[9] NetBeans Community, "Bug 194076 - Profiling fails with 'JVMTI Redefinition failed with error 60' for ArgumentManager," [Online]. Available: http://netbeans.org/bugzilla/show_bug.cgi?id=194076. [Accessed 15 May 2012].

[10] NetBeans Community, "Bug 195477 - Profiler error 60," [Online]. Available: http://netbeans.org/bugzilla/show_bug.cgi?id=195477. [Accessed 15 May 2012].

[11] NetBeans Community, "Bug 69299 - JVMTI Redefinition failed with error 60," [Online]. Available: http://netbeans.org/bugzilla/show_bug.cgi?id=69299. [Accessed 15 May 2012].

[12] Eclipse Foundation, "Eclipse Test and Performance Tools Platform project," [Online]. Available: http://www.eclipse.org/tptp/. [Accessed 15 May 2012].

[13] Eclipse Foundation, "Eclipse Public License v1.0," [Online]. Available: http://www.eclipse.org/legal/epl-v10.html. [Accessed 15 May 2012].

[14] Google, "Project java-allocation-instrumenter," [Online]. Available: http://code.google.com/p/java-allocation-instrumenter. [Accessed 15 May 2012].

[15] Oracle Corporation, "Java SE6 reference: java.lang.instrument package," [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html. [Accessed 15 May 2012].

[16] OW2, "ASM framework," [Online]. Available: http://asm.ow2.org/. [Accessed 15 May 2010].

[17] Apache Foundation, "Apache License, Version 2.0," [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0. [Accessed 15 May 2012].

Appendices

CD content

The accompanied CD-ROM contains the following content:

1. /memory monitor – source code of memory monitoring agent.
2. /java-allocation-instrumenter – source code of Allocation Instrumenter.
3. /results – benchmarks results and raw data.
4. /results/clean – raw data collected during clean run without any attached profiler.
5. /results/hprof – HPROF raw benchmark data.
6. /results/netbeans – NetBeans profiler's benchmark data.
7. /results/tptp – TPTP profiler benchmark data.
8. /results/results.xlsx – composite results.