Graphical Monitoring of CPU Resource Consumption in a Java-based Framework

Andrea Camesi and Jarle Hulaas
Software Engineering Laboratory

Walter Binder
Artificial Intelligence Laboratory

Swiss Federal Institute of Technology Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

ABSTRACT. Monitoring of CPU consumption is a very basic requirement in many areas of software. It is especially valuable in the frame of Internet applications, in support of specific aspects such as security, reliability, and adaptability. This paper is set in the context of J-RAF2, a Java-based framework exploiting bytecode rewriting techniques in order to guarantee the portability of CPU-managed applications. We briefly recall this implementation technique and present an example of CPU monitoring module to validate this new approach. Depending on the application, resource management may involve simple low-level accounting of CPU usage, or higher-level tools that collect accounting information from several sources in order to enforce sophisticated strategies. This paper shows that CPU monitoring modules may be easily built to work on top of classes rewritten with J-RAF2, and that they will be entirely portable across many kinds of Java deployment technologies (standalone applications, applets, servlets).

Keywords: CPU monitoring, CPU control, Bytecode Rewriting, Java, Management Framework, Security, Software architecture.

1. INTRODUCTION

Resource monitoring i.e., accounting and controlling physical resources like CPU, memory, bandwidth is a prerequisite in many platforms to increase security and reliability, e.g., to avoid denial-of-service attacks [1]. Context-awareness is another benefit that can be gained from a better understanding of resource control. Run time profiling of applications is crucial in server environments in order to protect the host from malicious or badly programmed code. Java [2] and the Java Virtual Machine (JVM) [3] are being increasingly used as the programming language and deployment platform for such server systems as Java 2 Enterprise Edition, Servlets, Java Server Pages and Enterprise Java Beans. However, the Java language and standard Java runtime systems currently lack mechanisms for resource management that could be used to limit the resource consumption of hosted applications or to charge the clients for their resource consumption. Thus, the use of Java helps building extensible systems, but conversely Java lags the necessary support of resource control.

In this paper we concentrate on the CPU resource [6]. Its monitoring is probably the most challenging of all resource kinds, due to the specificity that we cannot identify explicit CPU consumption places in the code. Indeed, to the opposite of other resources, it is rather considered continuous, i.e., to the opposite of memory that is explicitly always allocated. For that reason, it would be extremely valuable to be able to properly manage CPU consumption, especially if we consider the broad families of Internet applications that exist such as utility computing, web services, grid computing, embedded systems, and mobile object (mobile agent) systems [5] that can be extended and customized by mobile code. Moreover, implementing agent-oriented, context-aware software entities needs the realization of notions of self-organization and self-healing.

This article is set in the context of a Java-based portable CPU management framework called J-RAF2 (Java Resource Accounting Framework, 2nd edition, http://www.jraf2.org). It proposes a CPU monitoring architecture that can be used by engineers who need to manage resources to monitor and control CPU resources and by developers to manage the CPU consumption. The framework allows to implement new resource monitors using the provided adaptable and extensible mechanism.

This article is structured as follows: In the next section we explain the design goals of our approach for rewriting bytecode in support of CPU accounting. In Section 3 we illustrate some monitoring techniques that allow to manage CPU resource consumption. In Section 4 we continue exploring the management capabilities through graphical monitoring of applications. Then in Section 5, we discuss our solutions from the monitoring and management point of view. Finally, in Section 6 we present related work, and we conclude in Section 7.

2. CPU ACCOUNTING

In order to explain our CPU-management structure, we first briefly sketch our approach of bytecode rewriting. In J-RAF2 each thread, along its life time, accounts for its own CPU consumption into an associated account. The accounting is done by keeping track of the number of executed JVM bytecode instructions. Periodically, each thread aggregates its own consumption within an account that is shared with a number of other threads. This approach is called *self-accounting*. Interleaved with this accounting, the thread also executes management and monitoring code, e.g., to ensure that a given resource quota is not exceeded. In this way, the CPU management of J-RAF2 does not rely on a dedicated supervisor thread, but on a distribution among all threads in the system, thus effectively implementing a form of *self-control*.

2.1. Design goal to ensure portability

All current approaches somehow rely on the support of native, i.e., non-portable, code for their own implementation. Our framework, in contrast, draws its portability from the combination of bytecode transformations and runtime libraries implemented in pure Java (see Figure 1).

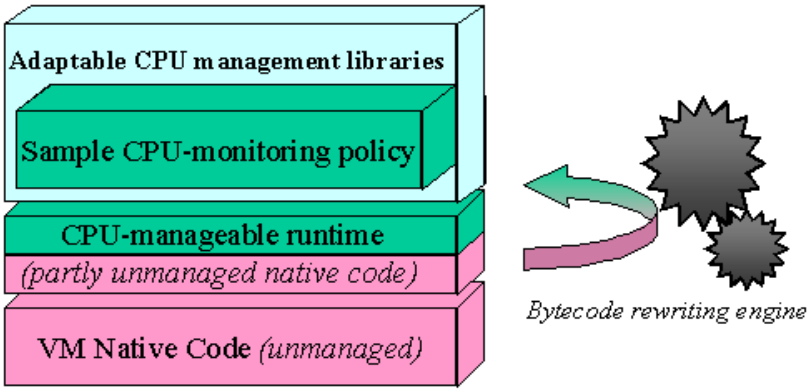

Figure 1 depicts the layered architecture of J-RAF2 offering a sample CPU-monitoring policy.
All Java code, including the JDK (except the subset that is implemented in native code) is made CPU manageable by bytecode transformations (a process symbolized by the round arrow flowing through the rewriting engine).

Figure 1. Monitoring using the portable Java CPU management framework.

This offers an important advantage over the other approaches, because it is independent of any particular JVM and underlying operating system. It works with every standard Java runtime systems and may be integrated into existing server and mobile object environments.

2.2. Bytecode Transformation Scheme

The two main goals of our bytecode transformation schemes are to ensure portability and performance. Portability is ensured by following a strict adherence to the specification of the Java language and virtual machine, and performance by minimizing the overhead due to the additional instructions inserted into the original classes.

Table 1: Part of the ThreadCPUAccount API.

```
public final class ThreadCPUAccount{
   public static
      ThreadCPUAccount getCurrentAccount();

   public int consumption;
   private int granularity;
   public void consume();

   private CPUManager manager;
   public CPUManager getManager();
   public void setManager(CPUManager m);
   ...
}
```

Table 2: The CPUManager interface.

```
public interface CPUManager {
   public void consume(long c);
   public int getGranularity();
   public void attach(Thread t);
   public void detach(Thread t);
}
```

Each thread has its own `ThreadCPUAccount`, of which Table 1 summarizes a part of the API interface. When a new thread object is initialized, the method `getCurrentAccount()` returns the associated `ThreadCPUAccount`. Then, each rewritten thread increments the *consumption* counter of its account with the number of bytecodes it intends to execute in the immediate future (the corresponding accounting sequences having been inserted at load-time by the J-RAF2 rewriting tool). When the counter has been incremented by a number of bytecodes equal to or greater than an adjustable limit, the accounting *granularity*, the shared management operation takes place by invocation of the `consume()` method. Conditionals that check whether `consume()` has to be invoked are inserted in the beginning of each method as well as in each loop [6]. Each `ThreadCPUAccount` refers to an implementation of `CPUManager`, the *manager* object, which is shared between all threads. The method `getManager()` returns the current `CPUManager`, whereas `setManager(CPUManager)` changes it. If a thread invokes `setManager(CPUManager)` on its own `ThreadCPUAccount`, the CPU consumption will be reported to its `CPUManager`, then the thread detaches from its `CPUManager` and attaches to the new `CPUManager`.

2.3. Aggregating CPU Consumption

The `CPUManager` is to implements the actual CPU accounting and control strategies, e.g., custom scheduling schemes [7]. Table 2 shows the `CPUManager` interface. Its methods are invoked by the `ThreadCPUAccount` implementation, as depicted in Figure 2. For performance reasons, the per-thread accounting objects have a fixed structure, since they have to perfectly match the characteristics of bytecode rewritten by our tool. When a thread invokes `consume()` on its `ThreadCPUAccount`, this method in turn reports its collected CPU consumption to the `CPUManager` by calling `consume(long)`. Inside the `CPUManager`, it may simply aggregate the reported CPU consumption and write it to a monitor, or it may enforce absolute limits and terminate threads or groups of threads that exceed their CPU limit, or it may limit the execution rate of threads, i.e., putting threads temporarily to sleep if they exceed a given execution rate. This is possible without breaking security assumptions, since the `consume(long)` invocation is synchronous, i.e., blocking, and executed by the thread to which the policy applies.

The `getGranularity()` method returns the accounting granularity currently defined for a `ThreadCPUAccount` associated with the given `CPUManager`. It is an adjustable value, which defines the frequency, and thus indirectly the overhead of the management activities: it has to be adapted to the number of threads under supervision in order to prevent excessive delays between invocations to `consume(long)`. The methods `attach(Thread t)` and `detach(Thread t)` allow the manager to assign and remove the associated `ThreadCPUAccount`.

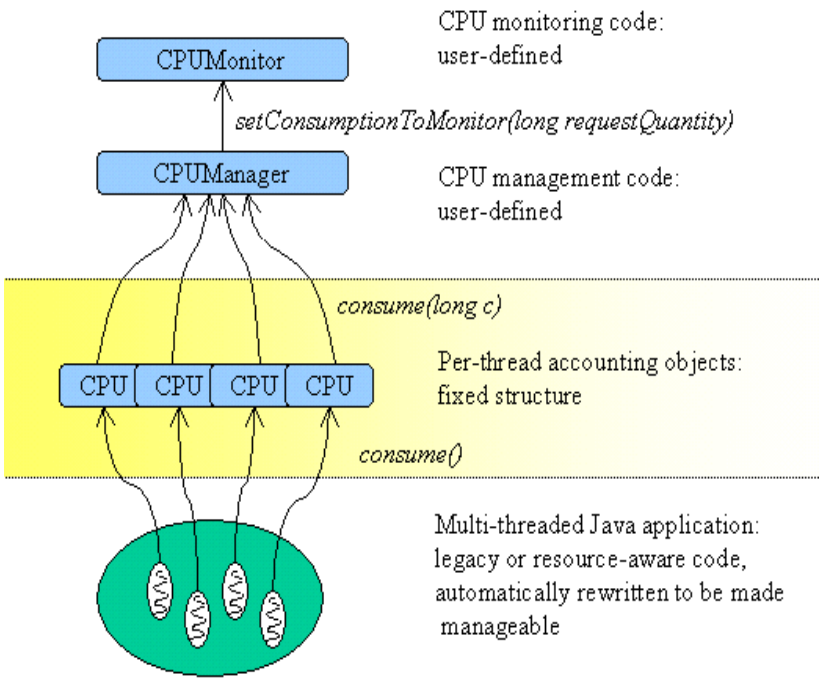

Figure 2. Overview of the CPU monitoring hierarchy.

3. MONITORING CPU RESOURCES

3.1. Managing Granularity

To manage the granularity, developers could write a class `GranularityManager` as shown in Table 3. The variable `granularity` is volatile in order to ensure that the `consume()` method of `ThreadCPUAccount` always reads the up-to-date value. The value is read by the threads that perform the `consume()` call (see Figure 2). [1] The method `setGranularity(int g)` allows the manager to adjust the granularity in order to adapt it to the number of monitored threads. And `getGranularity()` simply returns the actual value defined by the manager.

3.2. Managing Threads

Table 4 shows a simplified example of how the accounting information of multiple threads may be managed. The `ThreadCPUManager` class extends `GranularityManager` and implements `CPUManager`. The manager maintains a map of active threads in order to maintain the sum of all reported consumption information. It provides a specific implementation of the `consume(long)` method. Note that the `consume(long)` simply calls the `consume(Thread,long)` method that is synchronized, as multiple threads may invoke it concurrently. Same structure for the `getConsumption()` method that calls the `getConsumption(Thread)` method. It also provides methods to attach and detach threads to/from the hashmap (`attach(Thread)` and `detach(Thread)`). The use of *weak references* has the advantage that keys will automatically be removed by the garbage collector when appropriate.

1. The `CPUMonitor` also has a thread, but the CPU monitoring code is not rewritten, so this thread does not make any call to the `consume()` method.

3.3. Hierarchical structure of Managers

Like with `ThreadCPUManager`, the developer could define many different managers in a hierarchical structure. This allows the developer to assign appropriate CPUManagers depending on the chosen management policy. The next sub-section shows one example of the use of this inheritance structure to provide a `CPUManager` for a specific use.

Table 3: The GranularityManager implementation

```
public class GranularityManager {
   protected static final int
      MAX_GRANULARITY = ...;
   protected volatile int granularity;

   public GranularityManager() {
      granularity = MAX_GRANULARITY;
   }

   public GranularityManager(int g) {
      granularity = g;
   }

   public void setGranularity(int g) {
      granularity = g;
   }

   public int getGranularity() {
      return granularity;
   }
}
```

Table 4: Part of the ThreadCPUManager implementation

```
public class ThreadCPUManager
         extends GranularityManager
         implements CPUManager {

   protected Map activeThreads = new WeakHashMap();
   protected long totalConsumption = 0;

   public void consume(long c) {
      consume(Thread.currentThread(),c);
   }

   public synchronized void
         consume(Thread t, long c) {
      Long C = (Long) activeThreads.get(t);
      c += C.longValue();
      activeThreads.put(t,new Long(c));
   }

   public long getConsumption() {
      return getConsumption(Thread.currentThread());
   }

   public synchronized long getConsumption(Thread t) {
      Long C = (Long)activeThreads.get(t);
      return C.longValue();
   }

   public synchronized void attach(Thread t) {
      activeThreads.put(t, new Long(0));
   }

   public synchronized void detach(Thread t) {
      activeThreads.remove(t);
   }
   ...
}
```

3.4. Providing a CPUManager

When the JVM is first started, initial threads receive their dedicated `ThreadCPUAccount` objects without an associated `CPUManager`. Only once the start-up (or bootstrapping) process is completed, a user-defined `ManagerFactory` may be loaded (see Table 5), as specified by a system property. If a `ManagerFactory` is specified, such as e.g. the `MonitorCPUAccounting` (see Table 6), all threads collected during the bootstrapping phase become associated with one or several `CPUManagers` designated by the `ManagerFactory`[1]. This approach allows the user to install CPUManagers without modifying application classes by hand.

1. Identification and management of JVM internal daemon threads is made possible by the fact that, at some point of their execution, they execute rewritten user or JDK bytecode. In our approach, JVM internal daemon threads are identified as not having a parent ThreadGroup and so ignored.

In the simplest case, the user-defined `ManagerFactory` may provide a single default `CPUManager` like the `MonitorManagerFactory` implementation in Table 6. However, it may also inspect each thread passed as argument, as well as exploit other contextual information available at runtime, to help deciding which `CPUManager` shall be assigned. The `getManager(Thread)` method return the `CPUManager` that is assigned to the `ThreadCPUAccount`. When a new thread is created, it receives its own `ThreadCPUAccount`. If the creating thread has an associated `CPUManager`, the new thread will *inherit* the same `CPUManager`. Otherwise, the `ManagerFactory` will be asked to provide an appropriate `CPUManager`.

If a thread is not associated with a `CPUManager`, invocations of `consume()` on its `ThreadCPUAccount` will collect the consumption internally within the `ThreadCPUAccount`. When the thread becomes associated with a `CPUManager`, all the internally collected consumption will be reported to that `CPUManager`. The call `setManager(null)` may be used to disconnect a `ThreadCPUAccount` from any `CPUManager`.

Table 5: The ManagerFactory interface.

```
public interface ManagerFactory {
   CPUManager getManager(Thread t);
}
```

Table 6: The MonitorManagerFactory implementation.

```
public final class MonitorManagerFactory
         implements ManagerFactory {
   private final CPUManager appManager =
         new MonitorCPUAccounting();

   public Object getManager(Thread t) {
      ThreadGroup tg = t.getThreadGroup();
      return (tg == null || tg.getParent() == null) ?
         null :
         appManager;
}
```

The `MonitorCPUAccounting` class is an example that shows how to monitor the accounting information of multiple threads (see Table 7). It extends `ThreadCPUManager` and implements `CPUManager`. More custom control policies could be implemented in such a way, like the Isolate approach of Sun [8]. It provides, like `ThreadCPUManager`, a specific implementation of the `consume(long)` method. In this class, the monitor provides the communication with the graphical monitoring interface that has a user-defined layout, and allows to show some or all active threads. The graphical monitoring is explained in the next section.

Table 7: The MonitorCPUAccounting

```
public class MonitorCPUAccounting extends ThreadCPUManager {
//graphical monitor instance
private CPUMonitor monitor;

   public MonitorCPUAccounting(CPUMonitor monitor) {
      ...
   }

   public setMonitor(CPUMonitor monitor) {
      this.monitor = monitor;
   }

   public synchronized void consume(long requestQuantity) {
      super.consume(requestQuantity);
      setConsumptionToMonitor(requestQuantity);
   }

   public void setConsumptionToMonitor(long requestQuantity) {
      monitor.setConsumption(requestQuantity);
   }
   ...
}
```

In this simple case, `consume()` calls the `consume()` method of the `ThreadCPUManager` class and provides a new sample value to the graphical monitor through the `setConsumptionToMonitor(long)` method.

3.5. A model for CPU Monitoring

A model for the user-defined part of the CPU Monitoring is shown in the Figure 3. In this picture we distinguish the management part in yellow (with the `ThreadCPUAccount` class and the hierarchical structure of the `ManagerFactory`), the monitoring part in green (with the `GranularityManager` class and the hierarchical structure of the `CPUManager`), and finally the manageable runtime part in pink (with the classes that are used by the graphical monitor `CPUMonitor`).

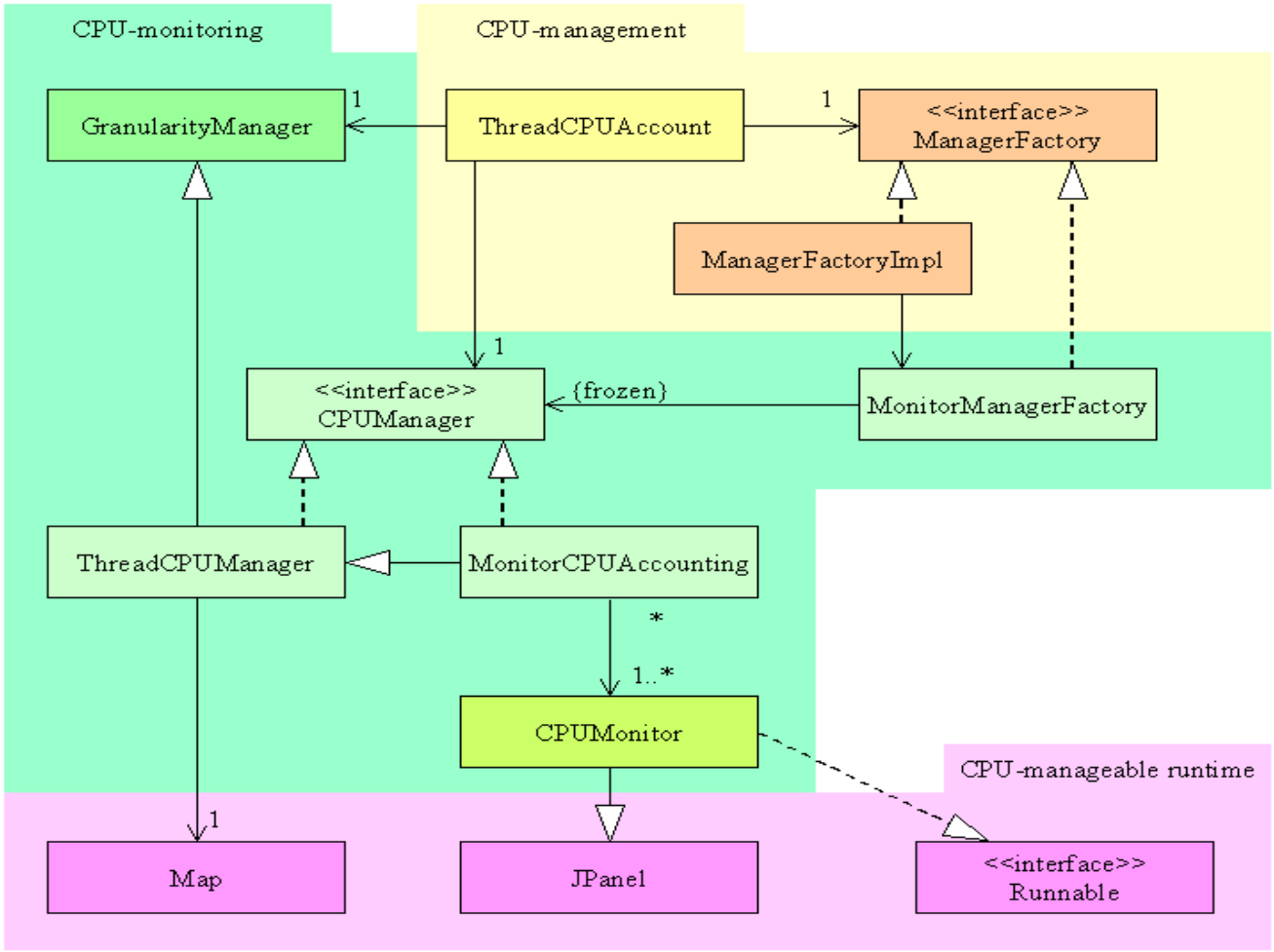

Figure 3. The CPUMonitoring model

4. GRAPHICAL MONITORING

The `CPUManager` allows the implementation of user-defined control strategies, e.g., the `MonitorCPUAccounting` class using the inheritance structure of `CPUManagers`. Moreover, programmers could plug customized graphical monitors showing all or some threads consumption with different layouts. The layout of the graphical interface is independent of the `MonitorCPUAccounting` class that implements the management policies; it could however communicate with this class (e.g., based on input received through the GUI) to enforce active measures against a thread that consumes too much CPU power. Because both classes are user-defined, developers who want to extend this structure have a very flexible architecture that allows more advanced interactions between these two classes.

4.1. JRAF2 in action

In this section we show a simple graphical monitor interface that uses the APIs described in the previous section. An important design guideline for us was to follow a minimalistic approach, while allowing programmers to add new features as needed. Figure 4 shows a screen shot of a graphical application that has been beforehand rewritten by the J-RAF2 tool. We chose the Java 2D Demo (http://java.sun.com/products/java-media/2D/samples/index.html) to demonstrate our approach on a general multi-threaded application. In order to accurately monitor the CPU consumption, applications, libraries, and the Java Development Kit (JDK) itself are transformed to expose details regarding the execution of threads. In this example, we aggregate the CPU consumption of individual threads and display these values in the graphical monitor.

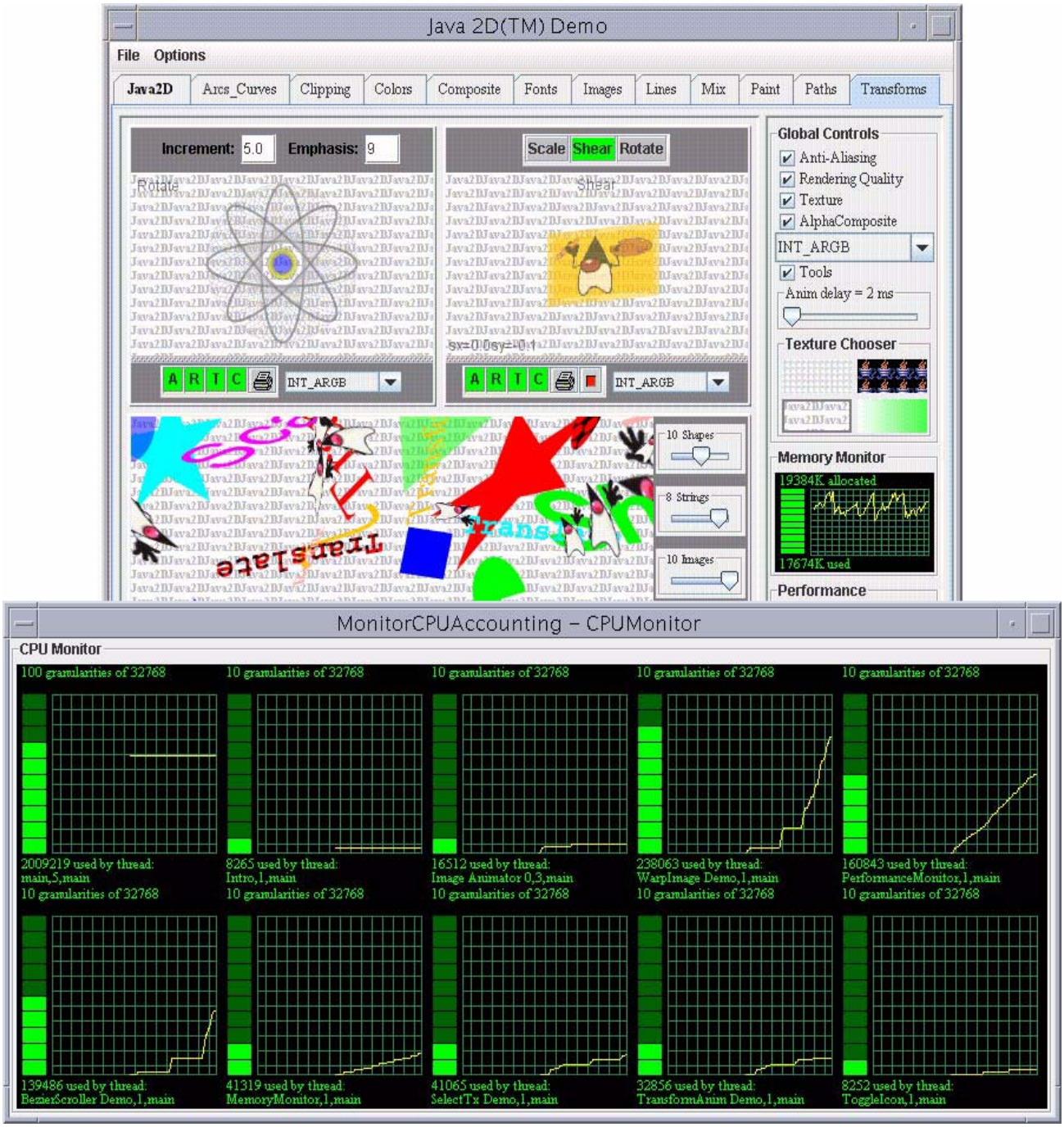

Figure 4. The Java 2D Demo processed by J-RAF2.

4.2. Application with Appletviewer

To run the Java 2D Demo application inside Appletviewer, the user needs to specify two options to give enough permissions to execute the accounting. These permissions are inserted into the file `<jdk_home>\jre\lib\security\java.policy` by adding the lines of Table 8.

Table 8: System properties needed with the Appletviewer

```
permission java.util.PropertyPermission "org.jraf2.cpu.ManagerFactory", "read";
permission java.lang.RuntimePermission "modifyThreadGroup";
```

The first line is required to be able to identify and load a `ManagerFactory`, e.g., the `MonitorManagerFactory` class, using the system property `-Dorg.jraf2.cpu.ManagerFactory=org.jraf2.runtime.cpu.managers.MonitorManagerFactory` on the command line. The second line is necessary for the `ManagerFactory` to be allowed to test if a given thread is a system thread or not, as required by the example of Table 6. Finally the user launches the "Java 2D Demo" application, prepending the -J option before the system property definition mentioned above (see Figure 5).

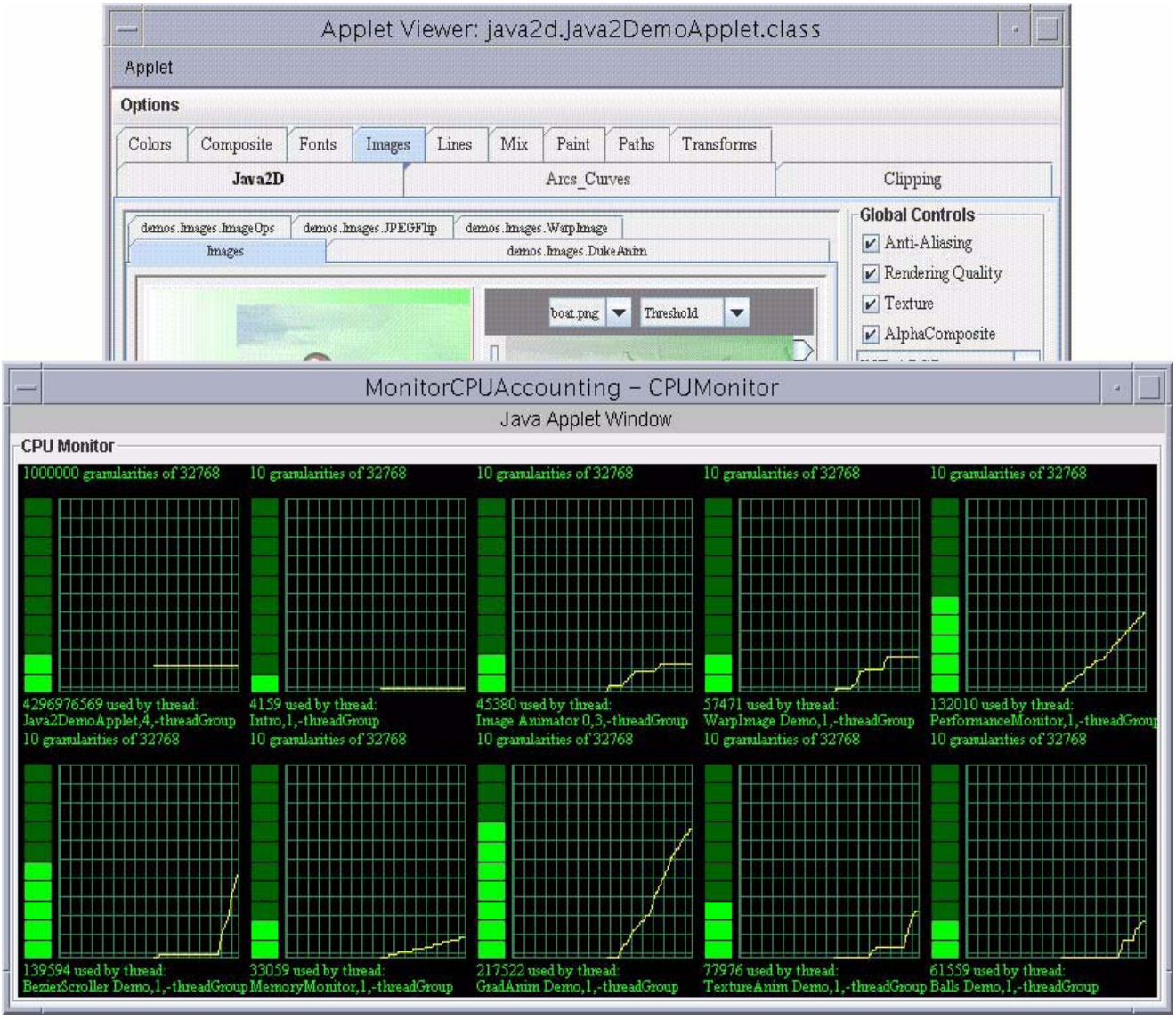

Figure 5. The Java 2D Demo inside the Appletviewer

4.3. Application with the Netscape browser

To run the Java 2D Demo application inside the Netscape browser, the user needs to specify an environment variable to specify to the browser where to find a JVM with a JDK rewritten [4] with J-RAF2 (see Table 9). Then he needs to write a simple html

Table 9: Environment variable needed for the Browser

```
setenv NPX_PLUGIN_PATH
   /users/camesi/jdk1.5.0/jre/plugin/sparc/ns4
```

Table 10. Simple html document with an applet

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <title>Java 2D(TM) Demo</title>
  </head>

  <body>
   <appletcode=java2d.Java2DemoApplet.class
archive=rw_Java2Demo.jar width=710 height=540>
   </applet>
  </body>
</html>
```

document with an applet tag calling our rewritten Java2Demo.jar application, here renamed rw_Java2Demo.jar (see Table 10). Finally, in order to enable our accounting features, the user needs to specify in the Java Control Panel, under the Java tab, choosing the option 'Java Applet Runtime Settings', the runtime parameter `-Dorg.jraf2.cpu.ManagerFactory=org.jraf2.runtime.cpu.managers.MonitorManagerFactory` (see Figure 6).

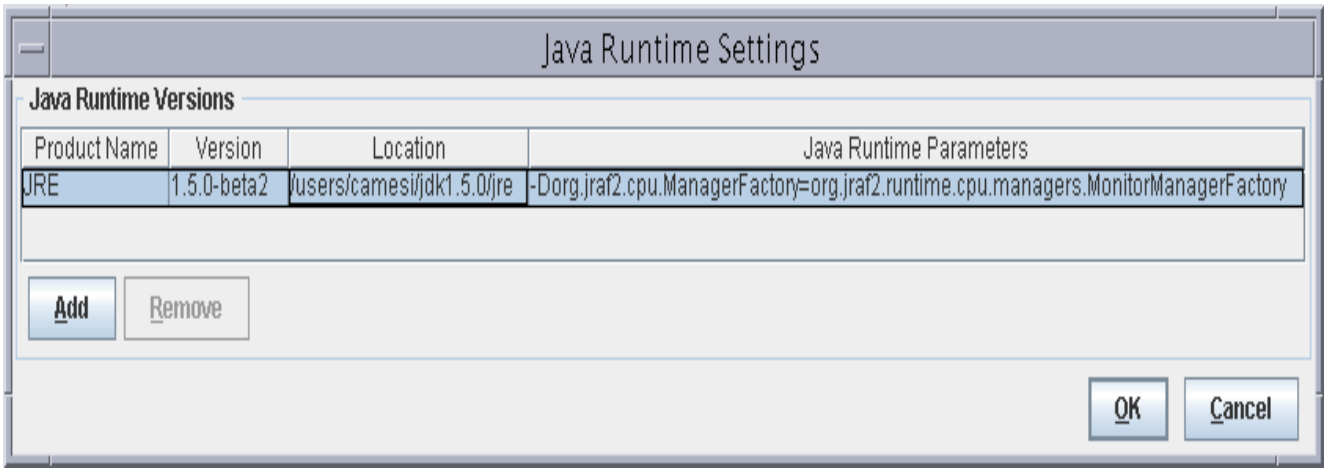

Figure 6. The Java Control Panel window

To view the application running inside the browser, launch Netscape on the simple html page defined above, like this:

`netscape `pwd`/Java2Demo.html` (see Figure 7)

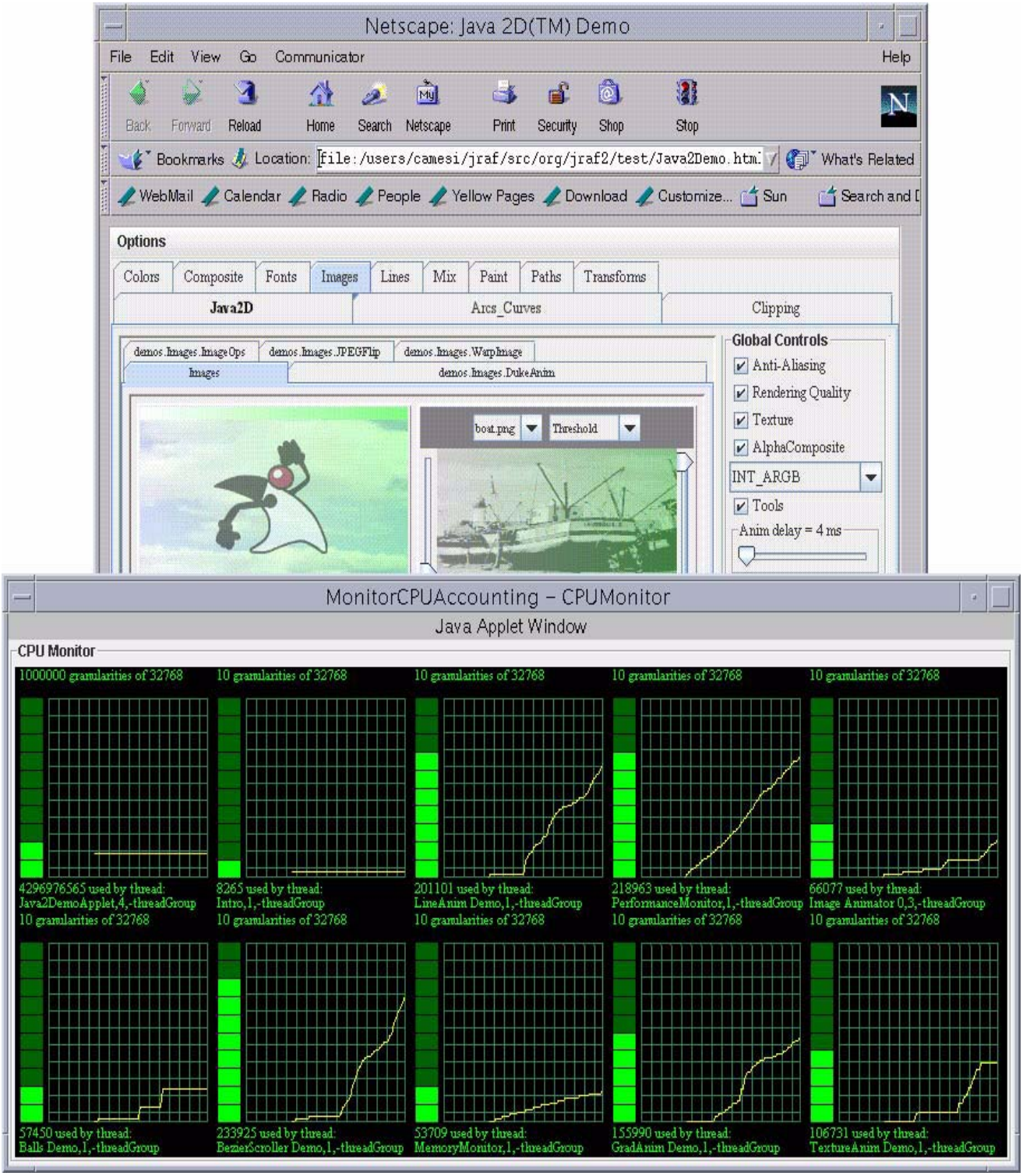

Figure 7. The Java 2D Demo inside the Netscape browser

4.4. Application as a Servlet

The last deployment scenario demonstrated in this paper is a simple ‘helloworld’ executing as a monitored servlet inside Tomcat version 5.5.0. The Servlet Engine needs some environment variables to start accounting the consumption (see Table 11). Note

Table 9: Environment variable needed for Servlet

```
setenv CATALINA_HOME
   /users/camesi/jakarta-tomcat-5.5.0
setenv JAVA_HOME
   /users/camesi/jdk1.5.0
setenv CATALINA_OPTS
   -Dorg.jraf2.cpu.ManagerFactory=org.jraf2.runtime.cpu.managers.MonitorManagerFactory
```

that the option specifying the monitor is set in the CATALINA_OPTS environment variable. Then we may start a web browser and call our servlet at a URL like ‘http://localhost:8080/servlets-examples/servlet/HelloWorldExample’, see Figure 8.

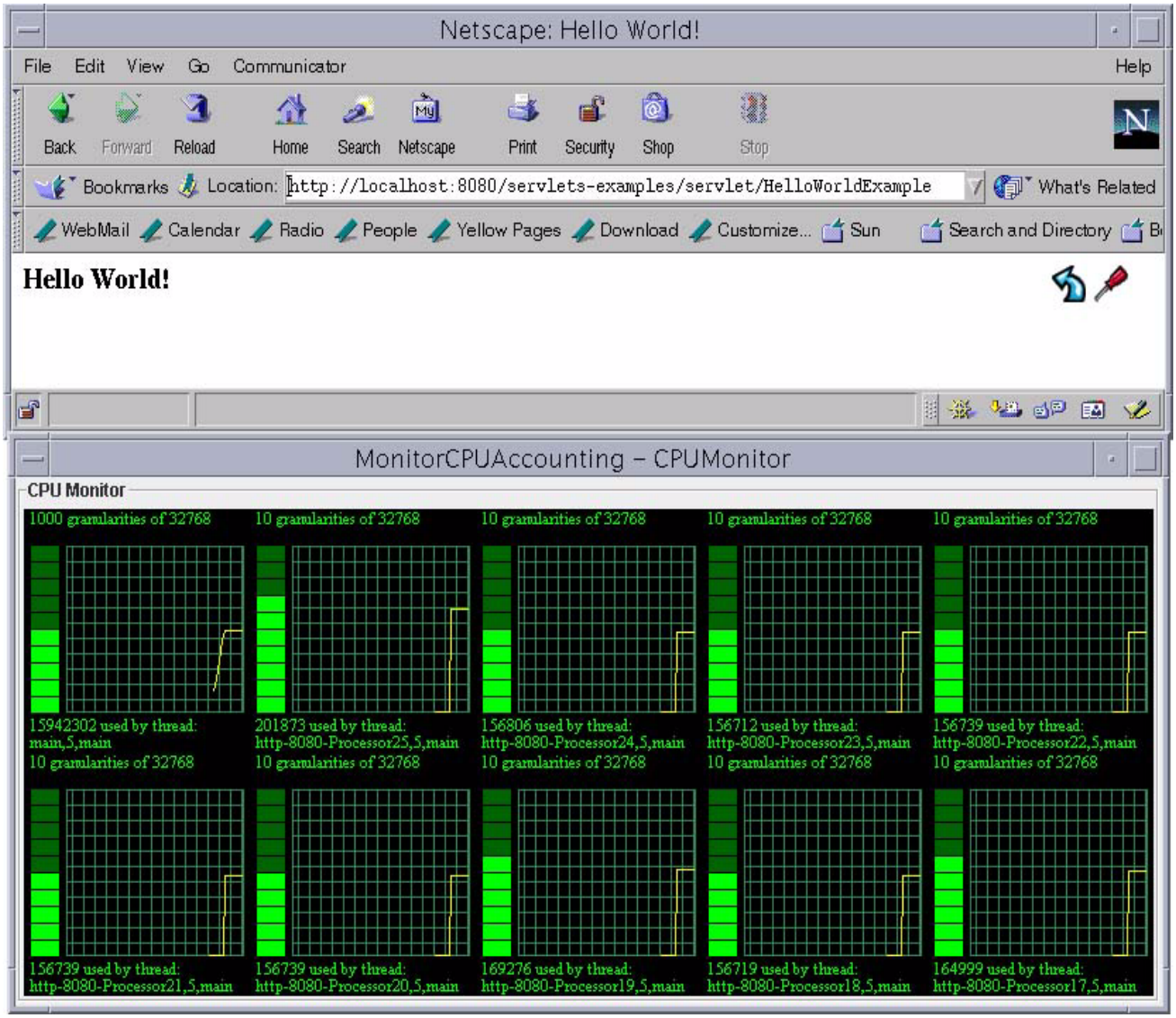

Figure 8. Accounting under Servlet

5. EVALUATION

Our proposal for CPU management and monitoring is built on the idea of self-accounting. We thus probably offer the most precise fine-grained accounting basis available with an extensible and adaptable layer architecture. Nevertheless, this approach is not only intellectually attractive, but it also solves one important weakness of all existing solutions (see related work section) based on a polling supervisor thread: The fact that the Java specification does not formally guarantee that the supervisor thread will ever be scheduled, whatever its priority is set to. On the other hand, in J-RAF2, any resources consumed will be accounted by the consuming thread itself (provided that the consuming code is implemented in Java, and not in some native language), and, if required, the thread will eventually take self-correcting measures. Another interesting contribution of our approach is the use of a portable, hardware-independent unit of measurement for CPU consumption: the number of executed bytecode instructions. Testing our architecture for accounting, monitoring and controlling the CPU resource consumption, we have explored the extension capabilities of our architecture and showed that the programmer is able to extend the hierarchical structure to his needs. Due to the nature of our approach, native code execution can of course not be directly accounted. Another small limitation of our J-RAF2 framework is that it needs to be used with Java version 2.0 or higher [1].

More sophisticated control strategies can be envisaged in different Internet applications. We could imagine a distributed graphical monitoring system e.g. in grid computing, with the needs of accessing real-time CPU usage data and the ability to host foreign, installable applications. In this configuration, new code installation happens almost at each execution. Here, resource management constitutes a building block for efficient load-balancing and therefore we expect graphical monitoring to be profitable.

6. RELATED WORK

Prevailing approaches to provide resource control in Java-based platforms rely on a modified JVM, on native code libraries, or on program transformations. For instance, the Aroma VM [9], KaffeOS [10], and the MVM [11] are specialized JVMs supporting resource control. JRes [12] is a resource control library for Java, which uses native code for CPU control and rewrites the bytecode of Java programs for memory control.

More recently, researchers at Sun have published a report relating their approach to incorporating resource management as an integral part of the Java language [7]. They have embraced a very broad field of investigation, since their ambitions are e.g. to care for physical as well as logical resources, and to provide direct support for sophisticated management policies with multi-party decision taking and notification. On the other hand, J-RAF2 focuses on the lower-level facilities, while leaving a lot of flexibility to developers. Several other management and runtime monitoring APIs have been offered by Sun along with the successive releases of Java platforms, especially for heap memory, but currently no solution is applicable as widely across environments, nor is also usable as basis for implementing control policies (as opposed to monitoring), nor is as well integrated with the language as the present framework.

7. CONCLUSION

In this paper we have given an overview of J-RAF2, an extensible and adaptable resource management framework. Initially, it was conceived to prevent denial-of-service attacks in mobile code environments, but is has become a general tool with different management strategies on top of arbitrary rewritten Java applications. Completely built on automated program transformation techniques applied at the bytecode level, it can be used with every standard JVM. This paper shows that developers are able to extend our layered structure to manage the CPU consumption. By installing hierarchical structures of CPUManagers, they are able to implement arbitrary strategies to supervise thread execution.

1. The principal reason of this choice is the use of the weak references in our approach. This is necessary to allow the GC (Garbage Collector) to clean the thread references to threads that terminated their execution to ensure no intrusive behaviour to the original application.

Acknowledgements

This work was partly financed by the Swiss National Science Foundation.

8. REFERENCES

[1] W. Binder, J. Hulaas, A. Villazón, and R. Vidal. Portable resource control in Java: The J-SEAL2 approach. *In ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-01)*, Tampa Bay, Florida, USA, Oct. 2001.

[2] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification.* Java series. Addison-Wesley, Reading, MA, USA, second edition, 2000.

[3] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, USA, second edition, 1999.

[4] W. Binder and J. Hulaas, Extending Standard Java Runtime Systems for Resource Management, in proceedings of SEM 2004 (the fourth international workshop on Software Engineering and Middleware), Linz, Austria, September 20-21, 2004, post-proceedings in LNCS vol. 3437, Springer Verlag, 2005.

[5] W. Binder and J. Hulaas, *A Portable CPU-Management Framework for Java* in IEEE Internet Computing, Vol. 8, No. 5, September/October 2004, pp. 74-83.

[6] J. Hulaas and W. Binder, *Program Transformations for Portable CPU Accounting and Control in Java*, in proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation), Verona, Italy, August 24-25, 2004, ACM Press, 2004, pp. 169-177, `http://www.jraf2.org/publications/PEPM04.pdf`.

[7] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce. *A resource management interface for the Java Platform*. In SP&E journal (Software Practice and Experience), vol. 35, pp. 123-157, February 2005.

[8] Java Community Process. JSR 121 - Application Isolation API Specification. Web pages at `http://jcp.org/jsr/detail/121.jsp`.

[9] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith, "NOMADS: toward a strong and safe mobile agent system", In C. Sierra, G. Maria, and J. S. Rosenschein (Eds.), *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00)*, pp. 163-164, NY, June 3-7 2000. ACM Press.

[10] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. *In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego, CA, USA, Oct. 2000.

[11] G. Czajkowski and L. Daynes. Multitasking without compromise: A virtual machine evolution. *In ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-01)*, Tampa Bay, Florida, Oct. 2001.

[12] G. Czajkowski and T. von Eicken, JRes: A resource accounting interface for Java. *In Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pp. 21-35, New York, USA, Oct. 18-22 1998. ACM Press.