Self-accounting as Principle for Portable CPU Control in Java

Walter Binder[1] and Jarle Hulaas[2]

[1] Artificial Intelligence Laboratory, EPFL, CH–1015 Lausanne, Switzerland
Walter.Binder@epfl.ch
[2] Software Engineering Laboratory, EPFL, CH–1015 Lausanne, Switzerland
Jarle.Hulaas@epfl.ch

Abstract. In this paper we present a novel scheme for portable CPU accounting and control in Java, which is based on program transformation techniques and can be used with every standard Java Virtual Machine. In our approach applications, libraries, and the Java Development Kit are modified in order to expose details regarding the execution of threads. Each thread accounts for the number of executed bytecode instructions and periodically the threads of an application component aggregate the information of their respective CPU consumption within a shared account and invoke scheduling functions that are able to prevent applications from exceeding their allowed CPU quota.

Keywords: Bytecode rewriting, CPU accounting and control, Java, program transformations

1 Introduction

Accounting and controlling the resource consumption of applications and of individual software components is crucial in server environments that host components on behalf of various clients, in order to protect the host from malicious or badly programmed code. Java [9] and the Java Virtual Machine (JVM) [11] are being increasingly used as the programming language and deployment platform for such servers (Java 2 Enterprise Edition, Servlets, Java Server Pages, Enterprise Java Beans). Moreover, accounting and limiting the resource consumption of applications is a prerequisite to prevent denial-of-service (DoS) attacks in mobile object (mobile agent) systems and middleware that can be extended and customized by mobile code. For such systems, Java is the predominant programming language.

However, currently the Java language and standard Java runtime systems lack mechanisms for resource management that could be used to limit the resource consumption of hosted components or to charge the clients for the resource consumption of their deployed components.

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 [12], KaffeOS [1], and the MVM [7] are specialized JVMs supporting resource control. JRes [8] is a resource control library for Java, which uses native code for CPU control and rewrites the bytecode of Java programs for memory control.

M. Weske and P. Liggesmeyer (Eds.): NODe 2004, LNCS 3263, pp. 24–38, 2004.

The Java Resource Accounting Framework J-RAF [4] is based completely on program transformations. In this approach the bytecode of applications is rewritten in order to make the CPU consumption of programs explicit. Programs rewritten by J-RAF keep track of the number of executed bytecode instructions (CPU accounting) and update a memory account when objects are allocated or reclaimed by the garbage collector.

Resource control with the aid of program transformations offers an important advantage over the other approaches, because it is independent of a particular JVM and underlying operating system. It works with standard Java runtime systems and may be integrated into existing server and mobile object environments. Furthermore, this approach enables resource control within embedded systems based on modern Java processors, which provide a JVM implemented in hardware that cannot be easily modified [5].

CPU accounting in the initial version of J-RAF [4] relied on a high-priority scheduling thread that executed periodically in order to aggregate the CPU consumption of individual threads and to adjust the running threads' priorities according to given scheduling policies. This approach hampered the claim that J-RAF enabled fully portable resource management in Java, because the scheduling of threads within the JVM is not well specified and the semantics of thread priorities in Java is not precisely defined. Hence, while some JVMs seem to provide preemptive scheduling ensuring that a thread with high priority will execute whenever it is ready to run, other JVMs do not respect thread priorities at all. Therefore, scheduling code written for environments using J-RAF may not exhibit the same behaviour when executed on different JVM implementations.

To overcome this limitation, the new version J-RAF2[1], the Java Resource Accounting Framework, Second Edition, comes with a new scheme for CPU accounting. In J-RAF2 each thread accounts for its own CPU consumption, taking the number of executed JVM bytecode instructions as platform-independent measurement unit. Periodically, each thread aggregates the collected information concerning its CPU consumption within an account that is shared by all threads of a software component and executes scheduling code that may take actions in order to prevent the threads of a component from exceeding their granted CPU quota. In this way, the CPU accounting scheme of J-RAF2 does not rely on a dedicated scheduling thread, but the scheduling task is distributed among all threads in the system. Hence, the new approach does not rely on the underlying scheduling of the JVM.

This paper is structured as follows: In the next section we review the old CPU accounting scheme of J-RAF and show its limitations. Section 3 explains the details of our new approach for CPU accounting. Section 4 evaluates the performance of applications using our new CPU accounting scheme. Finally, the last section summarizes the benefits of J-RAF2 and concludes this paper.

2 The Old CPU Accounting Scheme and Its Limitations

In the old version of J-RAF [4] each thread accounts locally for the number of JVM bytecode instructions it has executed. A periodically executing high-priority scheduling thread accumulates that accounting information from all threads in the system, ag-

[1] http://www.jraf2.org/

gregates it for individual application components, and ensures that CPU quota are respected, e.g., by terminating components that exceed their allowed CPU shares[2] or by lowering the priorities of threads that are excessively consuming processing power.

Each thread has an associated `OldCPUAccount` object maintaining a `long` counter, which is updated by the owning thread[3]. Table 1 shows some parts of the `OldCPUAccount` implementation. Because the scheduler thread has to read the value of the `consumption` variable, it has to be declared as `volatile` in order to force the JVM to immediately propagate every update from the working memory of a thread to the master copy in the main memory [9, 11].

Table 1. The `OldCPUAccount` implementation in the old accounting scheme.

```
public final class OldCPUAccount {
   public volatile long consumption;

   ...
}
```

In general, updating the counter requires loading the `consumption` field of the `OldCPUAccount` object from memory (it is `volatile`), incrementing the loaded value accordingly, and storing the new value in the memory. The accounting instructions are inserted in the beginning of each accounting block. An accounting block is related to the concept of basic block of code with the difference that method and constructor invocations may occur at any place within an accounting block. Details concerning the definition of accounting blocks can be found in [4]. Table 3 shows how the examplary code fragment given in table 2 is transformed according to this rewriting scheme. In this rewriting example we assume that the accounting object is passed as an extra argument to the method. For the sake of easy readability, we show all transformations at the Java level, even though the implementation works at the JVM bytecode level.

The main problem with this accounting scheme is that it relies on the scheduler of the JVM to ensure that a high-priority scheduler thread executes periodically in order to make use of the local accounting information of the individual threads. Unfortunately, the semantics of thread priorities are not well defined in the Java specification [9] and in the JVM specification [11]. For instance, the Java specification states: “Every thread has

[2] Current standard Java runtime systems do not offer a clean way to force the termination of components [6]. In order to enable the safe termination of a Java component (reclaiming all its allocated resources and ensuring the consistency of the system), the runtime system has to provide strong isolation properties. For instance, J-SEAL2 [2] is a Java library which provides strong isolation properties by preventing components from sharing object references. More recently, JSR 121 [10] specifies a Java extension to isolate components.

[3] This association may be implemented by a thread-local variable, by extending the `Thread` class of the Java Development Kit (JDK) with a special field to store the thread’s accounting object, or by extending the method signatures to pass the accounting object as an extra argument. However, these details are outside the scope of this paper, they can be found in [4] and more recently in [3].

Table 2. Exemplary method to be rewritten for CPU accounting.

```
void f() {
   X;
   while (true) {
      if (C) {
         Y;
      }
      Z;
   }
}
```

Table 3. Exemplary method rewritten according to the old CPU accounting scheme.

```
void f(OldCPUAccount cpu) {
   cpu.consumption += ...;
   X;
   while (true) {
      cpu.consumption += ...;
      if (C) {
         cpu.consumption += ...;
         Y;
      }
      cpu.consumption += ...;
      Z;
   }
}
```

a priority. When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion." When testing certain scheduling strategies (to be periodically executed by a scheduling thread) we encountered very different behaviour on different JVMs. On some JVMs our scheduler worked as intended, whereas other JVMs seemed not to respect thread priorities at all. Hence, due to the vague specification of the semantics of thread priorities, it is not possible to write platform-independent scheduling code based on priority assignment.

Another problem with this approach is its inferior performance. Updating a `volatile long` variable in each accounting block causes excessive overhead on several recent state-of-the-art JVMs. Past performance evaluations, such as in [4, 3], did not show this problem, since in the settings of these measurements the `consumption` variable was a `volatile int`. However, on fast processors the whole range of `int` may be exhausted within a millisecond of accounted execution or less. As there are no guarantees that the scheduler would run frequently enough to detect all overflows, we had to revert to a `volatile long` counter in order not to lose accounting information.

3 The New CPU Accounting Scheme and Its Benefits

Our new scheme for CPU accounting solves all the problems mentioned in section 2. It does without a dedicated scheduler thread by distributing the scheduling task to all threads in the system. Periodically, each thread reports its locally collected accounting information to an account that is shared between all threads of a component and executes scheduling functionality in order to ensure that a given resource quota is not exceeded (e.g., the component may be terminated if there is a hard limit on the total amount of bytecode instructions it may execute, or the thread may be delayed in order to meet a restriction on its execution rate). We call this approach *self-accounting*.

3.1 Implementation of `ThreadCPUAccount`

Like in the old approach, each thread has its associated `ThreadCPUAccount` which is shown in table 4. During execution each thread updates the `consumption` counter of its `ThreadCPUAccount` as in the old accounting scheme. Now the `consumption` variable is an `int` and it is not `volatile`, because only a single thread accesses it and ensures that there is no overflow.

Each threads invokes the `consume()` method of its `ThreadCPUAccount`, when the local `consumption` counter exceeds a certain threshold defined by the `granularity` variable. In order to optimize the comparison whether the `consumption` counter exceeds the `granularity`, the counter runs from `-granularity` to zero, and when it equals or exceeds zero, the `consume()` method is called. In the JVM bytecode there are dedicated instructions for the comparison with zero. We use the `iflt` instruction in order to skip the invocation of `consume()` if `consumption` is below zero.

Normally, each `ThreadCPUAccount` refers to an implementation of `CPUManager` (see table 5), which is shared between all threads belonging to a component. The first constructor of `ThreadCPUAccount` requires a reference to a `CPUManager`. The second constructor, which takes a value for the accounting granularity, is used only during bootstrapping of the JVM (`manager == null`). If the JDK has been rewritten for CPU accounting, the initial bootstrapping thread requires an associated `ThreadCPUAccount` object for its proper execution. However, loading complex user-defined classes during the bootstrapping of the JVM is dangerous, as it may violate certain dependencies in the classloading sequence. For this reason, a `ThreadCPUAccount` object can be created without previous allocation of a `CPUManager` implementation so that only two classes are inserted into the initial classloading sequence of the JVM: `ThreadCPUAccount` and `CPUManager`. Both of them only depend on `java.lang.Object`. After the bootstrapping (e.g., when the `main(String[])` method of an application is invoked), the `setManager(CPUManager)` method is used to associate `ThreadCPUAccount` objects that have been allocated and collected (e.g., in an array) during the bootstrapping with a `CPUManager`. As the variable `manager` is `volatile`, the thread associated with the `ThreadCPUAccount` object will notice the presence of the `CPUManager` upon the following invocation of `consume()`.

Table 4. The `ThreadCPUAccount` implementation in the new accounting scheme.

```
public final class ThreadCPUAccount {
   public int consumption;
   private long aggregatedConsumption = 0;
   private int granularity;
   private boolean consumeInvoked = false;
   private volatile CPUManager manager;

   public ThreadCPUAccount(CPUManager m) {
      manager = m;
      granularity = manager.getGranularity();
      consumption = -granularity;
   }

   public ThreadCPUAccount(int g) {
      manager = null;
      granularity = g;
      consumption = -granularity;
   }

   public void setManager(CPUManager m) {
      manager = m;
   }

   public void consume() {
      long amountCons =
         (long)consumption + granularity;
      if (manager == null) {
         aggregatedConsumption += amountCons;
         consumption = -granularity;
      }
      else {
         granularity = manager.getGranularity();
         consumption = -granularity;
         if (consumeInvoked) {
            aggregatedConsumption += amountCons;
         }
         else {
            amountCons += aggregatedConsumption;
            aggregatedConsumption = 0;
            consumeInvoked = true;
            manager.consume(amountCons);
            consumeInvoked = false;
         }
      }
   }
   ...
}
```

Table 5. The (simplified) `CPUManager` interface.

```
public interface CPUManager {
   public int getGranularity();
   public void consume(long c);
}
```

After the bootstrapping the `granularity` variable in `ThreadCPUAccount` is updated during each invocation of the `consume()` method. Thus, the `CPUManager` implementation may allow to change the accounting granularity dynamically. However, the new granularity does not become active for a certain thread immediately, but only after this thread has called `consume()`.

The `consume()` method of `ThreadCPUAccount` passes the locally collected information concerning the number of executed bytecode instructions to the `consume(long)` method of the `CPUManager` which implements custom scheduling policies. As sometimes `consume(long)` may execute a large number of instructions and the code implementing this method may have been rewritten for CPU accounting as well, it is important to prevent a recursive invocation of `consume(long)`. We use the flag `consumeInvoked` for this purpose. If a thread invokes the `consume()` method of its associated `ThreadCPUAccount` while it is executing the `consume(long)` method of its `CPUManager`, it simply accumulates the information on CPU consumption within the `aggregatedConsumption` variable of its `ThreadCPUAccount`. After the `consume(long)` method has returned, the thread will continue normal execution and upon the subsequent invocation of `consume()` the `aggregatedConsumption` will be taken into account. During bootstrapping a similar mechanism ensures that information concerning the CPU consumption is aggregated internally within the `aggregatedConsumption` field, as a `CPUManager` may not yet be available.

Details concerning the management of `CPUManager` objects, the association of `ThreadCPUAccount` with `CPUManager` objects, and of threads with `ThreadCPUAccount` objects are not in the scope of this paper. If J-RAF2 is used to integrate CPU management features into a Servlet or EJB container, the management of `CPUManager` objects is under the control of the container.

3.2 Bytecode Transformation Scheme

In order to make use of the new CPU accounting scheme, methods are rewritten in the following way:

1. Insertion of conditionals in order to invoke the `consume()` method periodically. The rationale behind these rules is to minimize the number of checks whether `consume()` has to be invoked for performance reasons, but to make sure that malicious code cannot execute an unlimited number of bytecode instructions without invocation of `consume()`. The conditional "`if (cpu.consumption >= 0) cpu.consume();`" is inserted in the following locations (the variable `cpu` refers to the `ThreadCPUAccount` of the currently executing thread):

(a) In the beginning of each loop.
(b) In the beginning of each JVM subroutine. This ensures that the conditional is present in the execution of recursive JVM subroutines.
(c) In the beginning of each exception handler.
(d) In the beginning of each method/constructor. This ensures that the conditional is present in the execution of recursive methods. For performance reasons, the insertion in the beginning of methods may be omitted if each possible execution path terminates or passes by an already inserted conditional before any method/constructor invocation (`invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`). For instance, this optimization applies to leaf methods.
(e) In each possible execution path after $MAXPATH$ bytecode instructions, where $MAXPATH$ is a global parameter passed to the bytecode rewriting tool. This means that the maximum number of instructions executed within one method before the conditional is being evaluated is limited to $MAXPATH$. In order to avoid an overflow of the `consumption` counter, $MAXPATH$ should not exceed 2^{15} (see section 3.5 for an explanation).

2. The `run()` method of each class that implements the `Runnable` interface is rewritten according to table 6 in order to invoke `consume()` before the thread terminates. After the thread has terminated, its `ThreadCPUAccount` becomes eligible for garbage collection.
3. Finally, the instructions that update the `consumption` counter are inserted at the beginning of each accounting block, like in the old CPU accounting scheme. In order to reduce the accounting overhead, the conditionals inserted before are not considered as separate accounting blocks. The number of bytecode instructions required for the evaluation of the conditional is added to the size of the accounting block they precede.

Table 6. The rewritten `run()` method.

```
public void run(ThreadCPUAccount cpu) {
   try {...}
   finally {cpu.consume();}
}
```

3.3 Rewriting Example

Table 7 illustrates how the exemplary method of table 2 is transformed using the new CPU accounting scheme. We assume that the code block X includes a method invocation, hence the conditional at the beginning of the method cannot be omitted.

3.4 Exemplary `CPUManager` Implementations

The following tables 8 and 9 show simplified examples how the accounting information of multiple threads may be aggregated and used. Both `CPUAccounting` and

Table 7. Exemplary method rewritten according to the new CPU accounting scheme.

```
void f(ThreadCPUAccount cpu) {
   cpu.consumption += ...;
   if (cpu.consumption >= 0) cpu.consume();
   X;
   while (true) {
      cpu.consumption += ...;
      if (cpu.consumption >= 0) cpu.consume();
      if (C) {
         cpu.consumption += ...;
         Y;
      }
      cpu.consumption += ...;
      Z;
   }
}
```

`CPUControl` implement `CPUManager` and provide specific implementations of the `consume(long)` method. `CPUAccounting` supports the dynamic adaptation of the accounting granularity. The variable `granularity` is `volatile` in order to ensure that the `consume()` method of `ThreadCPUAccount` alyways reads the up-to-date value.

Note that the `consume(long)` method is `synchronized`, as multiple threads may invoke it concurrently. The `CPUAccounting` implementation simply maintains the sum of all reported consumption information, whereas the `CPUControl` implementation enforces a strict limit and terminates a component when its threads exceed that limit. In this example we assume that the component whose CPU consumption shall be limited executes within a separate isolate. This is a notional example, as the isolation API [10] is missing in current standard JVMs. More sophisticated scheduling strategies could, for instance, delay the execution of threads when their execution rate exceeds a given threshold. However, attention has to be paid in order to prevent deadlocks and priority inversions.

3.5 Scheduling Delay

The delay until a thread invokes the scheduling code (as a custom implementation of the `consume(long)` method of `CPUManager`) is affected by the following factors:

1. The current accounting granularity for the thread. This value is bounded by `Integer.MAX_VALUE`, i.e., $2^{31} - 1$.
2. The number of bytecode instructions until the next conditional C is executed that checks whether the `consumption` variable has reached or exceeded zero. This value is bounded by the number of bytecode instructions on the longest execution path between two conditionals C. The worst case is a method M of maximum length that consists of a series of invocations of a leaf method L. We assume that

Table 8. Exemplary `CPUManager` implementation: CPU accounting without control.

```
public class CPUAccounting implements CPUManager {
   protected long consumption = 0;
   protected volatile int granularity;

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

   public int getGranularity() {
      return granularity;
   }

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

   public synchronized long getConsumption() {
      return consumption;
   }

   public synchronized void consume(long c) {
      consumption += c;
   }
}
```

Table 9. Exemplary `CPUManager` implementation: CPU control.

```
public class CPUControl extends CPUAccounting {
   private Isolate isolate;
   private long limit;

   public CPUControl(int g, Isolate i, long l) {
      super(g);
      isolate = i;
      limit = l;
   }

   public synchronized void consume(long c) {
      super.consume(c);
      if (consumption > limit) isolate.halt();
   }
}
```

L has $MAXPATH - 1$ bytecode instructions, no JVM subroutines, no exception handlers, and no loops. M will have the conditional C in the beginning and after each segment of $MAXPATH$ instructions, whereas C does not occur in L. During the execution of M, C is reached every $MAXPATH * (MAXPATH - 1)$ instructions, i.e., before $MAXPATH^2$ instructions.

Considering these two factors, in the worst case the `consume()` method of `ThreadCPUAccount` (which in turn will invoke the `consume(long)` method of `CPUManager`) will be invoked after each $MAXDELAY = (2^{31} - 1) + MAXPATH^2$ executed bytecode instructions. If $MAXPATH = 2^{15}$, the `int` counter `consumption` in `ThreadCPUAccount` will not overflow, because the initial counter value is `-granularity` and it will not exceed 2^{30}, well below `Integer.MAX_VALUE`). Using recent hardware and a state-of-the-art JVM, the execution of 2^{32} bytecode instructions may take only a fraction of a millisecond, of course depending on the complexity of the executed instructions.

For a component with n concurrent threads, in total less than $n * MAXDELAY$ bytecode instructions are executed before all its threads invoke the scheduling function. If the number of threads in a component can be high, the accounting granularity may be reduced in order to achieve a fine-grained scheduling. However, as the delay until an individual thread invokes the scheduling code is not only influenced by the accounting granularity, it may be necessary to use a smaller value for $MAXPATH$ during the rewriting.

3.6 Preventing Malicious Manipulations of `ThreadCPUAccount` Objects

As the accounting code is spread throughout the application classes and libraries, the `consumption` counter of `ThreadCPUAccount` has to be `public`. Therefore, malicious code could try to explicitly reset this counter in order to hide its CPU consumption.

To prevent this kind of attack, our bytecode rewriting tool includes a special verifier that is able to ensure that a class does not use features of `ThreadCPUAccount` before it is being transformed. The verifier is executed before rewriting untrusted code. If untrusted code tries to access `ThreadCPUAccount`, the verifier rejects it and the code shall not be loaded into the JVM. For trusted code, the verifier is not executed, as trusted code may need to access functions of `ThreadCPUAccount` for management purpose.

The verifier checks the constant pool [11] of untrusted classes in order to ensure that the `ThreadCPUAccount` class is not referenced. This technique of extended bytecode verification, which can be implemented very efficiently, has been successfully used in the JavaSeal [14] and J-SEAL2 [2] mobile object kernels. Moreover, if malicious code is allowed to use reflection, a simple check can be inserted in `java.lang.Class` during the rewriting of the JDK in order to prevent malicious code from accessing the accounting internals by reflection.

4 Evaluation

In this section we present a brief overview of the benchmarks we have executed to validate our new accounting scheme. We ran the SPEC JVM98 benchmark suite [13] on a Linux RedHat 9 computer (Intel Pentium 4, 2.6 GHz, 512 MB RAM). For all settings, the entire JVM98 benchmark was run 10 times, and the final results were obtained by calculating the geometric means of the median of each sub-test. Here we present the

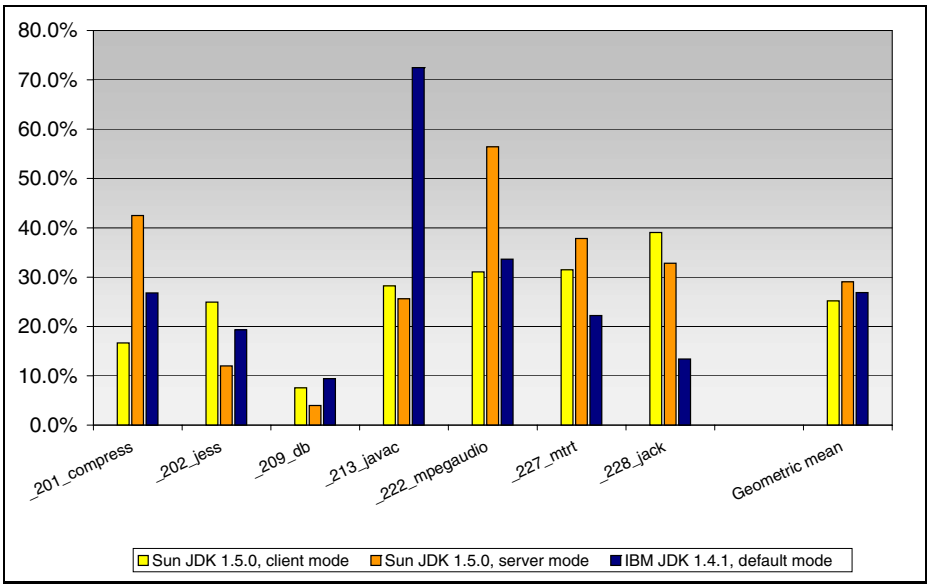

Fig. 1. Overheads due to CPU accounting in SPEC JVM98.

measurements made with the IBM JDK 1.4.1 platform in its default execution mode, as well as the Sun JDK 1.5.0 beta 1 platform in its 'client' and 'server' modes. In our test we used a single `CPUManager` with the most basic accounting policy, i.e., the one described in table 8, and with the highest possible granularity.

The most significant setting we measured was the performance of the rewritten JVM98 application on top of rewritten JDKs, and we found that the average overhead (execution time of modified code vs. execution time of unmodified code) was below 30% (see figure 1).

Another interesting measurement we made was to determine the impact of the choice of a granularity (see figure 2). The granularity has a direct influence on the responsiveness of the implementation w.r.t. the chosen management policy: the lower the granularity, the more frequently the management actions will take place. In our current implementation, and on the given computer, this interval is not measurable with granularities below 10,000,000 bytecode instructions. Another valuable lesson learned is that granularities of 100,000 and more exhibit practically the same level of overhead. The latter measurements were made exclusively on the 'compress' sub-benchmark of SPEC JVM98[4], hence the asymptotical values are slightly different from the above mentioned average overhead[5].

As a final remark, it should be emphasized that these results all correspond to a perfectly accurate accounting of executed bytecode instructions, which is a level of

[4] This was for simplicity, and 'compress' was chosen because it exhibits a performance which is usually closest to the overall average value.

[5] All performance measurements have an intrinsic imprecision of 2–3% depending on complex factors such as the load history of the test machine.

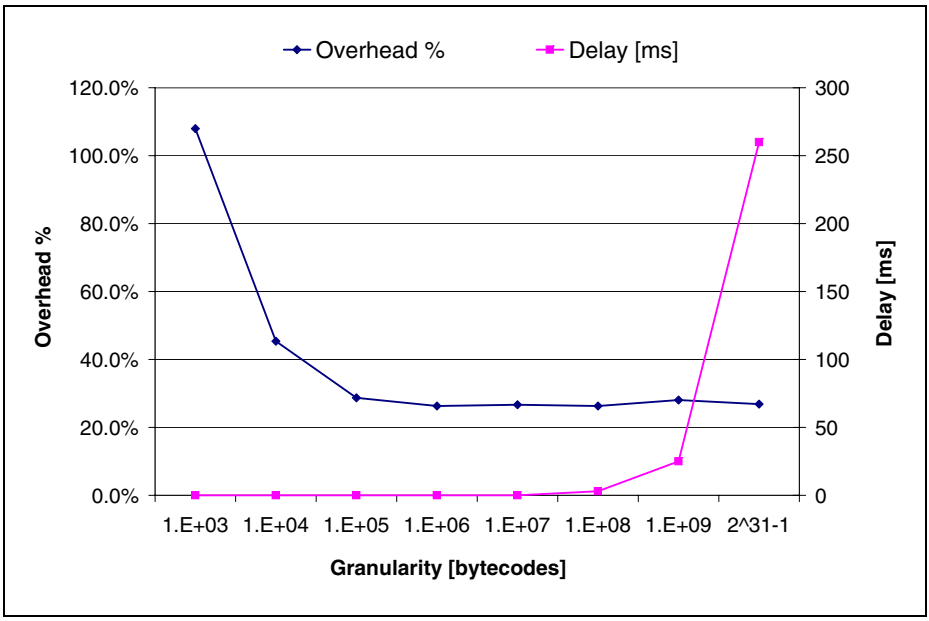

Fig. 2. Granularity versus overhead and delay.

precision not always necessary in practice. Currently, we are working on approximation schemes, which already enable us to reduce the overheads down to below 20%.

With the algorithms described here, the rewriting process takes only a very short time. For instance, rewriting the 20660 methods of the 2790 core classes of IBM JDK 1.4.1 takes less than one minute on our test machine. Each method is treated separately, but some algorithms could be enhanced with limited forms of interprocedural analysis. We do however not allow ourselves to do global analysis, as this might restrict the developer's freedom to extend classes gradually, and to load new sub-classes dynamically.

5 Conclusions

To summarize, the new CPU accounting scheme of J-RAF2 offers the following benefits, which make it an ideal candidate for enhancing Java server environments and mobile object systems with resource management features:

- Full portability. J-RAF2 is implemented in pure Java. In has been tested with several standard JVMs in different environments, including also the Java 2 Micro Edition [5].
- Platform-independent unit of accounting. A time-based measurement unit makes it hard to establish a contract concerning the resource usage between a client and a server, as the client does not exactly know how much workload can be completed within a given resource limit (since this depends on the hardware characteristics of the server). In contrast, using the number of executed bytecode instructions is

independent of system properties of the server environment. To improve accounting precision, various JVM bytecode instructions could be associated with different weights. At the moment, our implementation counts all instructions equally.
- Flexible accounting/controlling strategies. J-RAF2 allows custom implementations of the `CPUManager` interface.
- Fine-grained control of scheduling granularity. As described in section 3.5, the accounting delay can be adjusted; to some extend dynamically at runtime, to some extend during the rewriting process.
- Independence of JVM thread scheduling. The new CPU accounting scheme of J-RAF2 does not rely on thread priorities anymore.
- Moderate overhead. We have shown that the new CPU accounting scheme does not increase the overhead with respect to the old scheme. However, the new scheme brings many benefits, such as the independence of the JVM scheduling, or the prevention of overflows.

Concerning limitations, the major hurdle of our approach is that it cannot account for the execution of native code. We should also note that our evaluation was done without any optimizations to reduce the number of accounting sites. Work is in progress to provide a complete optimization framework, which allows to trade-off between accounting precision and overhead.

Acknowledgements

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

References

1. 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.
2. W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, Jan. 2001.
3. W. Binder and V. Calderon. Creating a resource-aware JDK. In *ECOOP 2002 Workshop on Resource Management for Safe Languages*, Malaga, Spain, June 2002. http://www.ovmj.org/workshops/resman/.
4. 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-2001)*, Tampa Bay, Florida, USA, Oct. 2001.
5. W. Binder and B. Lichtl. Using a secure mobile object kernel as operating system on embedded devices to support the dynamic upload of applications. *Lecture Notes in Computer Science*, 2535, 2002.
6. W. Binder and V. Roth. Secure mobile agent systems using Java: Where are we heading? In *Seventeenth ACM Symposium on Applied Computing (SAC-2002)*, Madrid, Spain, Mar. 2002.
7. 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.

8. 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*, pages 21–35, New York, USA, Oct. 18–22 1998. ACM Press.
9. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, MA, USA, second edition, 2000.
10. Java Community Process. JSR 121 – Application Isolation API Specification. Web pages at http://jcp.org/jsr/detail/121.jsp.
11. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
12. 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, editors, *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00)*, pages 163–164, NY, June 3–7 2000. ACM Press.
13. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at http://www.spec.org/osg/jvm98/.
14. J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal or how to make Java safe for agents. Technical report, University of Geneva, July 1998. http://cui.unige.ch/OSG/publications/OO-articles/ TechnicalReports/98/javaSeal.pdf.