Program Transformations for Portable CPU Accounting and Control in Java

Jarle Hulaas
Software Engineering Laboratory
School of Computer and Comm. Sciences
Swiss Federal Institute of Technology Lausanne (EPFL)
CH–1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Walter Binder
Artificial Intelligence Laboratory
School of Computer and Comm. Sciences
Swiss Federal Institute of Technology Lausanne (EPFL)
CH–1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

ABSTRACT

In this paper we introduce a novel scheme for portable CPU accounting and control in Java, which is based on program transformation techniques at the bytecode level and can be used with every standard Java Virtual Machine. In our approach applications, middleware, and the standard java runtime libraries (i.e., the Java Development Kit, or JDK) are modified in order to expose details regarding the execution of threads. This paper presents the details of how we re-engineer Java bytecode for CPU management, including the strategies developed for transforming the JDK itself in a fully portable way.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures, Frameworks*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Management, Performance, Reliability, Security, Languages

Keywords

Java, Resource Management, Bytecode Engineering, Program Transformations

1. INTRODUCTION

Resource management (i.e., accounting and controlling physical resources like CPU, memory, bandwidth) is a useful, yet rather unexplored aspect of software. Increased security, reliability, performance, and context-awareness are some of the benefits that can be gained from a better understanding of resource management. For instance, accounting and controlling the resource consumption of applications and of individual software components is crucial in server environments that run components on behalf of external clients, in order to protect the host from malicious or badly programmed code. Resource accounting may also provide valuable feedback about actual usage by end-clients and thus enable precise billing and provisioning policies. Java [8] and the Java Virtual Machine (JVM) [10] 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. A more recent research topic addresses the problem of designing and implementing agent-oriented, context-aware software components; awareness of resource availability and usage policies is inherent to this approach, where the realization of self-organizing and self-healing properties are among the long-term objectives. Yet another interesting target domain is resource-constrained embedded systems, because software run on such platforms has to be aware of resource restrictions in order to prevent abnormal termination. For many of the above mentioned systems, Java is the predominant programming language.

However, currently the Java language and standard Java runtime systems lack mechanisms for resource management that could e.g. 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 [11], KaffeOS [1], and the MVM [5] are specialized JVMs supporting resource control. JRes [6] is a resource control library for Java, which uses native code for CPU control and rewrites the bytecode of Java programs for memory control.

On the other hand, resource control with the aid of program transformations offers an important advantage over the other approaches, because it is independent of any 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PEPM'04, August 24–25, 2004, Verona, Italy.
Copyright 2004 ACM 1-58113-835-0/04/0008 ...$5.00.

modified [4]. This is the direction followed in this paper, since it is based completely on program transformations: the bytecode of 'legacy' applications is rewritten in order to make the resource consumption of programs explicit. Thus, rewritten programs will unknowingly 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.

These ideas were first implemented in the Java Resource Accounting Framework J-RAF [3], which has undergone a complete revision in order to provide far better reliability, programmability and performance. This paper presents the details of how we currently re-engineer Java bytecode for CPU management, including the strategies developed for transforming the JDK (Java Development Kit, i.e., the standard java runtime libraries) itself in a fully portable way. These advances are implemented in the second generation of this tool, called J-RAF2[1].

This paper is structured as follows: In the next section we explain the details of our new approach for CPU accounting. Section 3 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. OUR CPU ACCOUNTING SCHEME

In this section we explain the details of our latest CPU accounting scheme. In J-RAF2 each thread permanently accounts for its own CPU consumption, taking the number of executed JVM bytecode instructions as platform-independent measurement unit. Periodically, each thread will aggregate the collected information concerning its own CPU consumption within an account that is shared by all threads of a software component, and execute management code, e.g., scheduling decisions, 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 threads may be delayed in order to meet a restriction placed on execution rates). In this way, the CPU accounting scheme of J-RAF2 does not rely on a dedicated supervisor thread, but the scheduling task is distributed among all threads in the system. We call this approach $self-accounting$.

Hence, and this is for us a guarantee of portability and reliability, we do not rely on the underlying scheduling of the JVM, which is left loosely specified in the Java language, probably to make it easier to implement Java across a wide variety of environments: 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.

2.1 Bytecode Transformation Scheme

Concerning the bytecode rewriting schemes, our two main design goals are to ensure portability (by following a strict adherence to the specification of the Java language and virtual machine) and performance (i.e., minimal overhead due to the additional instructions inserted into the original classes).

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

Each thread has an associated `ThreadCPUAccount` which is shown in figure 5 of subsection 2.5. During normal execution each thread updates the `consumption` counter of its `ThreadCPUAccount`. In order to prevent overflows of the `consumption` counter, which is a simple 32-bit integer, and to schedule regular activation of the shared management tasks, the counter is checked against an adjustable `granularity` limit. More precisely, each time the counter has been incremented by `granularity` (executed bytecodes), its value will be registered and reset to an initial value by the invocation of a `consume()` method. In other words, each thread invokes the `consume()` method of its `ThreadCPUAccount`, when the local `consumption` counter exceeds a certain limit 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. In order to apply this CPU accounting scheme, methods of applications 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 method. 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`). In other words, leaf methods may be omitted (especially accessors of abstract data types), as well as entry points of methods inside which the first method invocation happens after another inserted conditional, typically inside or after a loop.

   (b) In the beginning of each loop.

   (c) In the beginning of each JVM subroutine. This ensures that the conditional is present in the execution of recursive JVM subroutines.

   (d) In the beginning of each exception handler.

   (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 2.7 for an explanation). In practice, hand-written Java methods never exceed a few hundred bytecodes in length, so this is a very weak constraint.

2. The `run()` method of each class that implements the `Runnable` interface is rewritten according to figure 1 in order to invoke `consume()` before the thread terminates. After the thread has terminated, its `ThreadCPUAccount` becomes eligible for garbage collection.

Figure 1: The rewritten `run()` method.

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

3. Finally, the instructions that update the `consumption` counter are inserted at the beginning of each accounting block[2]. 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.

In [7], a problem similar to the one of minimizing the number of inserted conditionals was studied, but in the context of compiling functional languages; this leads to slightly different results than with our scheme, which allows explicit loop constructs, and where modularity and object-orientation precludes certain kinds of optimizations relying on global program analysis.

2.2 Rewriting Example

Figure 2: Exemplary method to be rewritten for CPU accounting.

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

Figure 3 illustrates how the exemplary method of figure 2 is transformed using the proposed CPU accounting scheme.

[2]Here we define the term *accounting block* as the longest possible sequence of bytecode instructions where only the first instruction may be the target of a branch, and where the last instruction is one that changes the control flow (i.e., a branch, return, etc., but not an invocation).

We assume that the code block X includes a method invocation, hence the conditional at the beginning of the method cannot be omitted. Here we do not show the concrete values by which the consumption variable is incremented; these values are calculated statically by the rewriting tool and represent the number of bytecodes that are going to be executed in the next accounting block.

Figure 3: Exemplary method rewritten for CPU accounting.

```
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;
   }
}
```

2.3 Obtaining a Reference to the Associated ThreadCPUAccount

The astute reader may have noticed, by looking at figure 3, that `ThreadCPUAccount` references were passed by argument to all rewritten methods, thus changing their profiles. In reality, this works in the ideal case where we know that absolutely all code has been transformed, i.e., when both callers and callees refer to the same method profiles, and the account reference can be simply and directly transmitted in a chain. But in order to cope with special cases where this is not possible, we have to leave in all classes stubs with the original method profiles, which act as *wrappers* for the methods with the additional `ThreadCPUAccount` argument, as in figure 4.

Figure 4: Exemplary wrapper method for CPU accounting.

```
void f() {
   // Obtain a reference to the account
   // of the current Thread:
   ThreadCPUAccount cpu =
       ThreadCPUAccount.getCurrentAccount();
   // Invoke the real method:
   f(cpu);
}
```

Now, another question arises: how can the `getCurrentAccount()` method know which account is associated to the current thread? Unfortunately, the JVM provides no means for applications to be alerted each

time a thread switch occurs. Therefore, we have to call the standard `Thread.currentThread()` method each time this information is needed. Then, using the reference to the current thread, we can obtain the associated CPU account either through a hash table, or we can patch the real `Thread` class and add the needed reference directly to its set of instance fields. In the measurements presented in this paper, we opted for the second solution, because it is far more efficient, while still respecting the language specifications[3]. The `getCurrentAccount()` method does exactly this.

2.4 Bootstrapping with a Rewritten JDK

The functionality of `getCurrentAccount()` described just above is however not valid during the bootstrapping of the JVM. During this short, but crucial period, there is an initial phase where the `Thread.currentThread()` pretends that no thread is executing and returns the value `null` (this is in fact because the `Thread` class has not yet been asked by the JVM to create its first instance). As a consequence, in all code susceptible of being executed during bootstrapping, i.e., in the JDK, as opposed to application code, we have to make an additional check whether the current thread is undefined; for those cases, we have to provide a dummy, empty `ThreadCPUAccount` instance, the role of which is to prevent all references to the `consumption` variable in the rewritten JDK from generating a `NullPointerException`. This special functionality is provided by the `jdkGetCurrentAccount()` method, which replaces the normal `getCurrentAccount()` whenever we rewrite JDK classes.

A second issue that arises when bootstrapping the JVM with a rewritten JDK, is that we have no means for being informed *when the bootstrapping is terminated.* This is important, since before that moment, we are not allowed to actually use classes of the JDK inside the implementation of our runtime support, like `ThreadCPUAccount`. If we did, it would disturb the normal class loading sequence of the JVM, and most probably make it crash. But unless we want to implement all our runtime support from scratch, and not only ignore the benefits of code reuse for our own base classes, but also impose the same constraints to third-party implementations of `CPUManager` classes (as described in the next section), we have to know from what moment we are allowed to use helper classes like `java.util.Vector`. Our solution goes like this: application classes, i.e., all those which are not part of the JDK, are loaded by the JVM at the end of the bootstrapping. We can thus try to detect when such non-JDK classes are initialized by the JVM, because that will signal the end of the bootstrapping. Concretely, J-RAF2 exploits the *static initializer* method, named `<clinit>`, that may exist inside every class for the sake of initializing class (`static`) variables. Whenever the JVM initializes a class, it will invoke the `<clinit>` method of that class, if it exists. Our rewriting tool can thus insert an invocation to the J-RAF2 runtime system inside every `<clinit>` method (and, if necessary, create it from scratch) of every non-JDK class.

[3]We investigated another approach, which was to make the standard `Thread` class inherit from a special class provided by us, and thus receive the required additional field by inheritance. This elegant alternative would however not conform to the Java language API, which stipulates that `Thread` is a direct subclass of `Object`.

At that call, the more advanced functionalities of CPU management can be loaded and launched. Note that this latter step has to be performed by meta-programming (using the `java.lang.reflect` facility), in order to hide such dependencies away from the JVM; if the dependencies were explicit, the JVM might try to load those classes eagerly, and, of course, break the bootstrapping.

A third issue concerns the wrapper rewriting of the JDK. At some instances, native code in the JDK assumes that it will find a required piece of information at a statically known number of stack frames below itself. This is unfortunately incompatible with the generation of wrapper methods as described previously, because at run-time, it would induce additional stack frames that would break the kind of native methods mentioned above. For this reason, the JDK cannot take advantage of the more efficient wrapper rewriting scheme, and has to invoke `jdkGetCurrentAccount` at the beginning of every method. We can however, at the expense of a comprehensive class hierarchy analysis process, completely duplicate most JDK methods, so that there always is one version with the original profile, and a second with the additional `ThreadCPUAccount` argument. This approach works well, but is quite complex, and results in an appreciable speedup only with older JVMs, therefore we did not include it in the measurements presented in this paper.

2.5 Implementation of ThreadCPUAccount

Normally, each `ThreadCPUAccount` refers to an implementation of `CPUManager` (see figure 6), 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, the `setManager(CPUManager)` method is used to associate `ThreadCPUAccount` objects that had been allocated 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()`.

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

Figure 5: Excerpt of the `ThreadCPUAccount` class.

```
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;
         }
      }
   }

   ...
}
```

Figure 6: The (simplified) `CPUManager` interface.

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

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, and the association of `ThreadCPUAccount` with `CPUManager` 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.

2.6 Exemplary CPUManager Implementations

The following figures 7 and 8 show simplified examples how the accounting information of multiple threads may be aggregated and used. Both `CPUAccounting` and `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.

Figure 7: 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;
   }
}
```

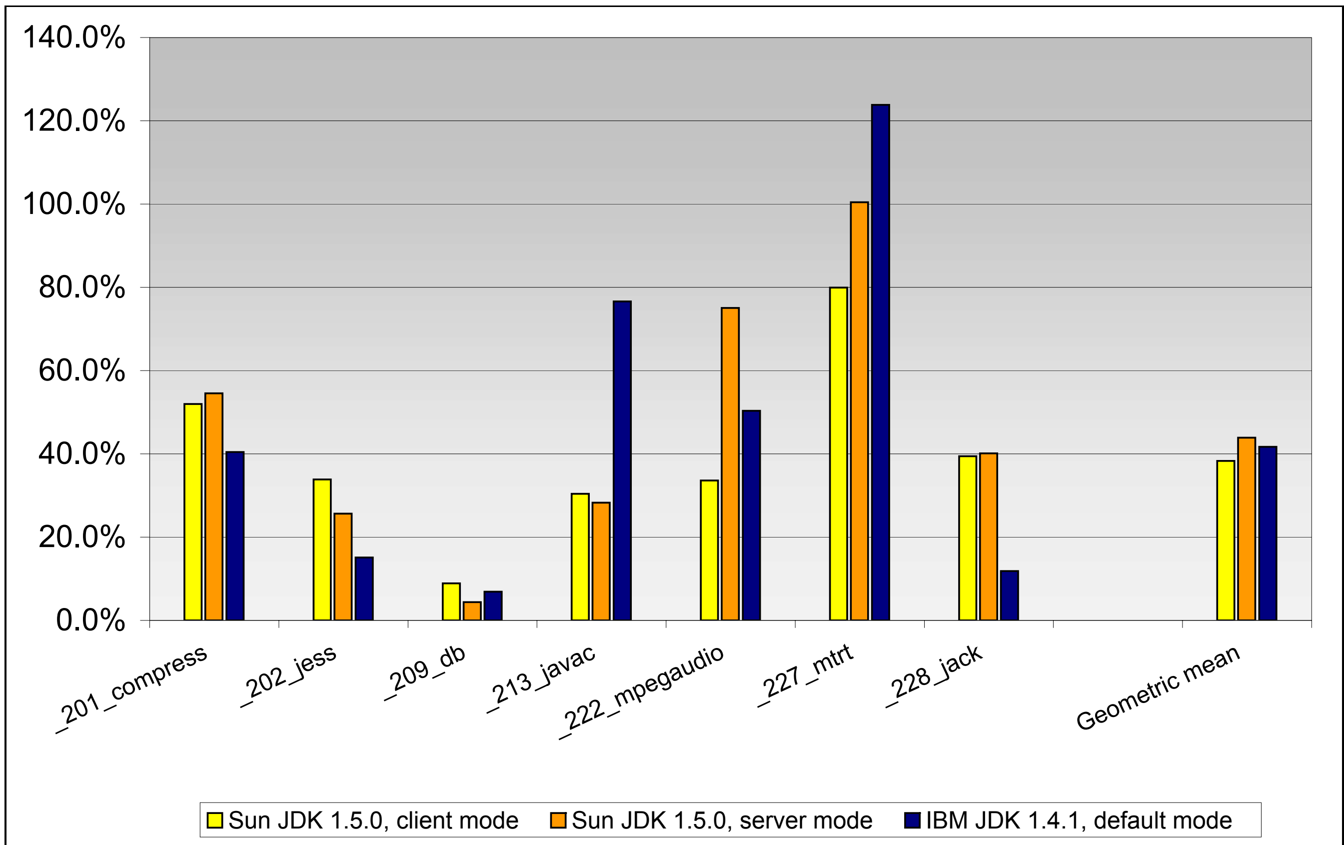

Figure 9: Overheads due to simple CPU accounting in SPEC JVM98.

Figure 8: 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();
   }
}
```

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 [9] 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.

2.7 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

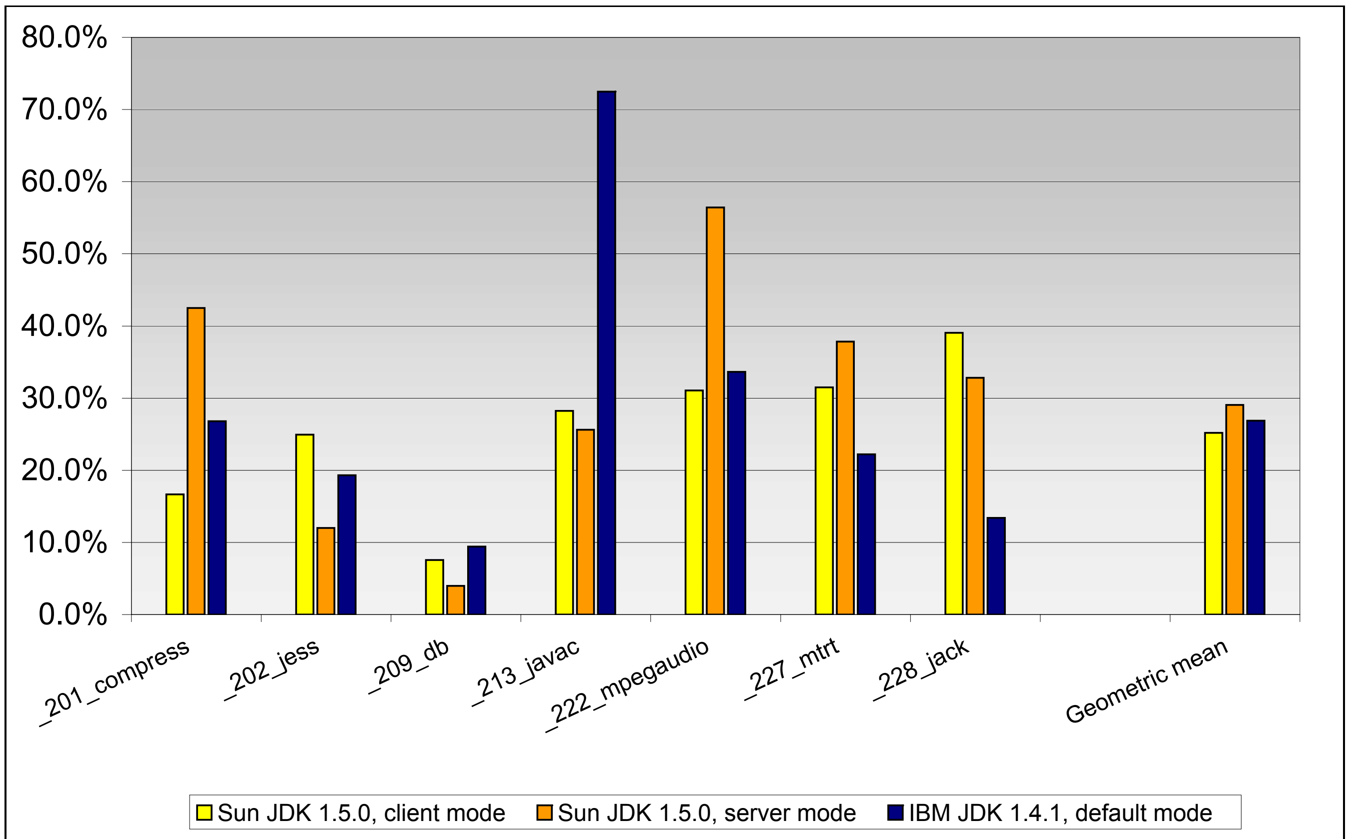

Figure 10: Overheads due to optimized CPU accounting in SPEC JVM98.

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 L has $MAXPATH - 1$ bytecode instructions, no JVM subroutines, 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. 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 [12] 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 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 figure 7, 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 overhead (execution time of modified code vs. execution time of unmodified code) is about 40% (see figure 9). This average is slightly better for the IBM JDK, and much better for the Sun JDK[4] than pre-

[4]The Sun virtual machine has evolved considerably since our former measurements: with the same rewriting scheme, overheads fall from around 300% down to about 40% when going from Sun's 1.3.1 generation to its latest release.

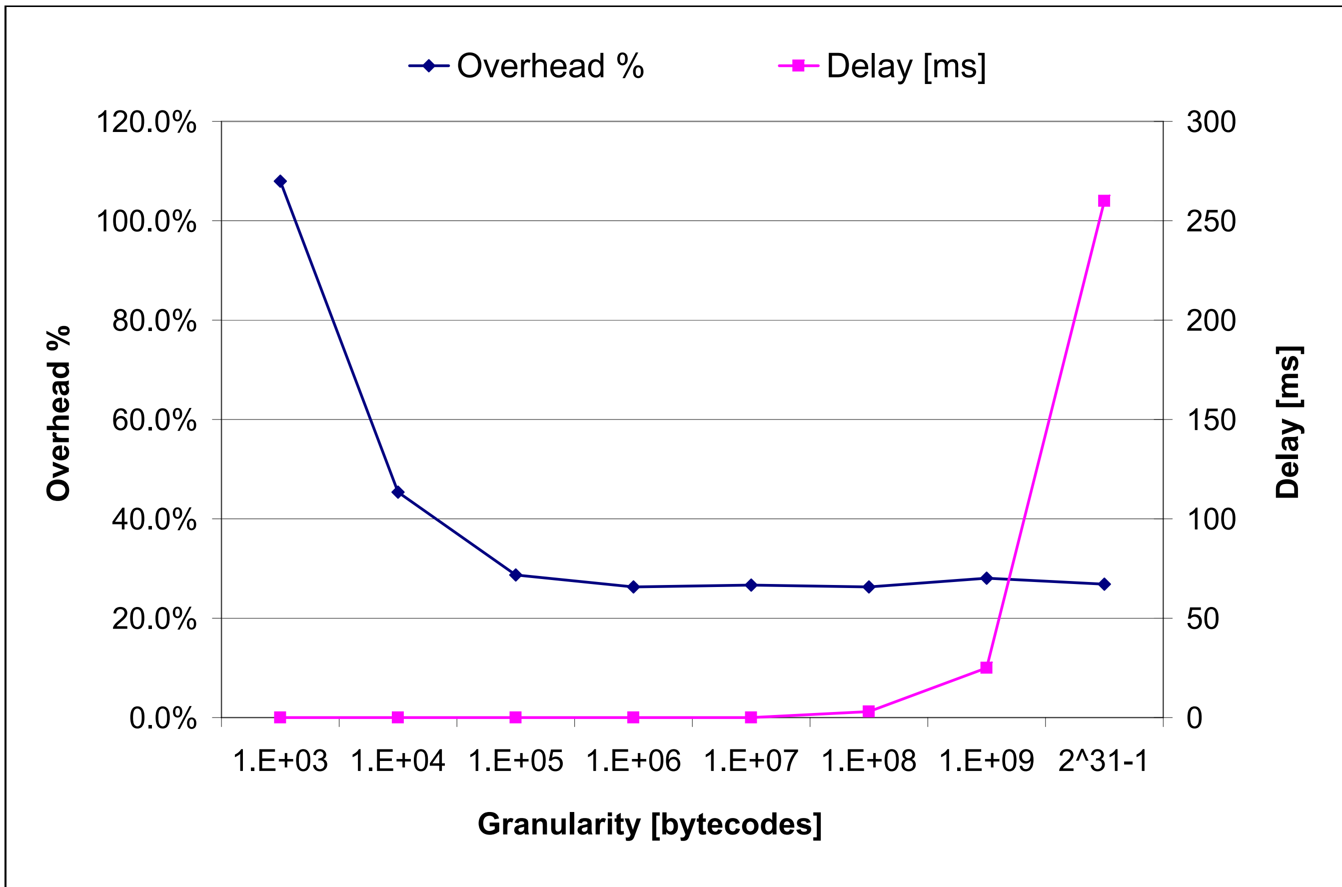

Figure 11: Granularity versus overhead and delay.

viously published results [2], which were based on a former, less portable and less reliable rewriting scheme.

Another interesting measurement we made was to determine the impact of the choice of a granularity (see figure 11). 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[5], hence the asymptotical values slightly different from the above mentioned average overhead.[6]

There are still many ways to reduce these overheads by improving the rewriting schemes. We have for instance experimented with inlining carefully hand-crafted bytecode sequences instead of invoking very frequent methods like `getCurrentAccount()`, or with duplicating all method bodies instead of resorting to short wrappers, or also with detecting leaf calls that do not need to implement the check of consumption vs. granularity (a simple optimization that is applicable to about 19% of the methods of both JDKs considered here); these optimizations decrease overheads drastically, but this is ongoing work, and could not be described further here due to the lack of space. Figure 10 shows that overheads fall down to approximately 27% with what has lately become the default set of optimizations of J-RAF2.

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

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

4. CONCLUSIONS

Resource control with the aid of program transformations offers an important advantage over the other approaches, because it is independent of any particular JVM and underlying operating system. It works with standard Java runtime systems and may be integrated into existing server and mobile object environments, as well as into embedded systems based on Java processors.

To summarize, the CPU accounting scheme described here 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 and all transformations follow a strict adherence to the specification of the Java language and virtual machine. It has been tested with several standard JVMs in different environments, including also the Java 2 Micro Edition [4].
- 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 2.7, 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 present CPU accounting scheme of J-RAF2 does not rely on thread priorities.
- Moderate overhead. We have shown that our CPU accounting scheme does not produce excessive overhead. At the same time, it has 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. It is nevertheless possible to wrap several expensive native operations, like (de-)serialization and class loading, with libraries that deduce the approximate CPU consumption from the size and value of the arguments.

Work is in progress to provide a complete optimization framework, which allows to trade-off between accounting precision and overhead. Security has not been addressed either in this paper; we do nevertheless have load-time bytecode verification algorithms designed to prevent applications from tampering with their own CPU consumption accounts.

Acknowledgements

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

5. 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 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/`.

[3] 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.

[4] 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.

[5] G. Czajkowski and L. Daynès. 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.

[6] 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.

[7] M. Feeley. Polling efficiently on stock hardware. In *the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 179–187, 1993.

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

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

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

[11] 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.

[12] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`.