Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 53–73
www.elsevier.com/locate/entcs

Java Bytecode Transformations for Efficient, Portable CPU Accounting

Walter Binder and Jarle Hulaas

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract

Resource management is essential to build reliable middleware and to host potentially untrusted software components. Resource accounting allows to study and optimize program performance and to charge users for the resource consumption of their deployed components, while resource control can limit the resource consumption of components in order to prevent denial-of-service attacks. In the approach presented here, program transformations enable resource management in Java-based environments, even though the underlying runtime system may not expose information concerning the resource consumption of applications. We present a fully portable program transformation scheme to enhance standard Java runtime systems with mechanisms for CPU management. We implemented several optimizations in order to reduce the overhead of our CPU accounting scheme. Detailed performance measurements quantify this overhead and show the impact of various optimizations.

Keywords: Java, Resource Management, Bytecode Engineering, Program Transformations

1 Introduction

Management of physical resources (i.e., accounting and controlling resources like CPU and memory) is an interesting 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.

The case of CPU consumption is very challenging because one cannot identify explicit consumption sites in the code and, contrary to other resources, it is rather considered continuous (which is reflected by the fact that quantities

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2005.02.037

of CPU can hardly be manipulated as first-class entities in conventional programming environments). This paper presents an approach which addresses these two issues.

Accounting and controlling CPU consumption has many valuable applications. But currently, this capability is furnished in an ad-hoc way by the underlying operating system. However, higher software layers would definitely benefit from standardized APIs and tools in order to enable portable and tightly integrated implementations of resource management code at the middleware or application level.

CPU accounting is a fundamental element of monitoring and profiling. Many advanced tools rely on the continuous collection of consumption information in order to automatically adapt and dynamically optimize the execution of applications or to guarantee the best possible utilization of a pool of computing resources. Therefore, monitoring and profiling are not only performed with dedicated development-time tools, but become an integral part of the runtime environment.

Software run on resource-constrained embedded systems has to be aware of resource restrictions in order to avoid situations of sudden and abnormal termination. This capability subsumes a programming model including proactive resource management facilities. Similarly, emerging agent-oriented, context-aware applications will need such facilities for realizing their self-organizing and self-healing objectives.

In applications that allow the installation of foreign software components or need the flexibility of mobile code, CPU management is a required building block. This is because current standard security mechanisms, like digital certificates, tend to be too coarse-grained, to follow a very static approach, and to focus exclusively on access control. They do not cover purely dynamic aspects such as maximal execution rates or the number of concurrent threads allowed on a given system. These dynamic aspects are nevertheless essential to security and stability.

For all of the above mentioned families of applications, Java and the Java Virtual Machine (JVM) [5] represent a predominant programming language and deployment platform. 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 components or to charge the clients for the resource consumption of their deployed components.

This paper presents a new, portable CPU management framework for Java called J-RAF2 (Java Resource Accounting Framework, 2nd edition) [1]. With

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

the aid of program transformations at the JVM bytecode level, applications and libraries, including the Java Development Kit (JDK), are rewritten in order to expose information concerning their CPU consumption. Each thread in the system records the number of bytecode instructions it has executed and periodically invokes a management and scheduling mechanism that can be customized by the middleware programmer. I.e., we exploit bytecode instruction counting as platform-independent metrics for CPU consumption.

The main, original contributions of this paper are several optimizations to our basic resource accounting scheme in order to reduce the overhead of CPU accounting. Detailed performance measurements illustrate the impact of these optimizations. J-RAF2 is a continuation of our previous work [2], compared to which we have now achieved much stronger reliability, portability, and programmability.

This paper is structured as follows: The next section gives an overview of our portable CPU accounting scheme. Section 3 presents a series of optimizations that help to reduce the CPU accounting overhead, while Section 4 provides a detailed performance evaluation. Section 5 discusses the strengths and limitations of our approach. Finally, Section 6 concludes this paper.

2 Portable CPU Accounting

In this section we explain our CPU accounting scheme. To start with, 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 for CPU accounting. More precisely, each thread permanently accounts for its own CPU consumption, expressing it as the number of executed JVM bytecode instructions. This constitutes a platform-independent measurement unit, which has some practical advantages, as explained in Section 5. Periodically, each thread aggregates the collected information concerning its own CPU consumption within an account that is shared with a number of other threads. We call this approach *self-accounting*. During these information update routines, the thread will also execute management code, e.g., to ensure that a given resource quota is not exceeded. In this way, the CPU management scheme of J-RAF2 does not rely on a dedicated supervisor thread, since the management activity is distributed among all threads in the system, thus effectively implementing a form of *self-control*.

Hence, and this is for us a guarantee of portability and reliability, we do not rely on the underlying scheduling provided by 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. This is a major difference to our previous accounting scheme [2], which relied on thread priorities for scheduling.

In the following subsections we summarize our portable CPU accounting scheme. Low-level implementation details are covered in [4]. In [1] the programming APIs are presented, together with programming examples from the viewpoint of a middleware developer and two case studies of successful applications of J-RAF2. In contrast to these previous publications, this paper focuses on optimizations and on a detailed evaluation.

2.1 Associating Accounting Information with Threads

Concerning the bytecode transformation (or rewriting) scheme, 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 (by minimizing the overhead due to the additional instructions inserted into the original classes).

Each thread has an associated `ThreadCPUAccount`. Fig. 1 summarizes part of the public interface. The semantics of the methods and fields are explained in this and in the following subsections. The association of a thread with its `ThreadCPUAccount` persists for the whole life-time of the thread. When a new thread object is initialized, it automatically receives a fresh `ThreadCPUAccount` object. This is achieved by statically modifying the bytecode of the `Thread` class, adding a field to hold a reference to the thread's `ThreadCPUAccount` object. Moreover, the thread constructors are patched in order to initialize that field with a new instance of `ThreadCPUAccount` whenever a thread is created.The `getCurrentAccount()` method in Fig. 1 returns the `ThreadCPUAccount` of the calling thread.

```
public final class ThreadCPUAccount {
   public static ThreadCPUAccount getCurrentAccount();
   public int consumption;
   public void consume();
   ...
}
```

Fig. 1. Part of the ThreadCPUAccount API.

2.2 Bytecode Transformation Scheme

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 limit, the *accounting granularity* (in Section 2.4 we will explain how this value is provided). More precisely, each time the `consumption` counter is incremented by a number of bytecodes greater than or equal to the granularity, its value is registered and reset to an initial value by the invocation of the `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 accounting granularity. In order to optimize the comparison whether the `consumption` counter exceeds this limit, the counter runs from the granularity value multiplied by -1 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 thus 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, (non-native and non-abstract) methods are rewritten in the following way:

(i) At the beginning of each method the current thread's `ThreadCPUAccount` has to be obtained using the static method `getCurrentAccount()`. In Section 3.2 we will present a more efficient alternative.

(ii) Conditionals are inserted 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) At the beginning of each method. This ensures that the conditional is present in the execution of recursive methods. In Section 3.1 we will show how this rule can be relaxed in order to reduce the CPU accounting overhead.
   (b) At the beginning of each JVM subroutine. This ensures that the conditional is present in the execution of recursive JVM subroutines.
   (c) At the beginning of each exception handler.
   (d) At the beginning of each loop.

(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 evaluated is limited by $MaxPath$. Consequently, a larger value of $MaxPath$ may increase the effective accounting granularity.

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

(iv) Finally, the instructions that update the `consumption` counter are inserted at 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 [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.

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

Fig. 2. The rewritten `run()` method.

2.3 Rewriting Example

The example in Fig. 3 illustrates how a method is transformed using the proposed CPU accounting scheme. The conditional that checks whether the `consumption` counter has reached zero is inserted at the beginning of the method and in the loop, whereas the `consumption` variable is updated in each accounting block. Here we do not show the concrete values by which the variable `consumption` 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. [2]

[2] For the sake of better readability, in this paper we show all transformations on Java code or pseudo-code, whereas our implementation works at the JVM bytecode level.

```
void f(int x) {                    void f(int x) {
                                      ThreadCPUAccount cpu;
                                      cpu = ThreadCPUAccount.getCurrentAccount();
                                      cpu.consumption += ...;
                                      if (cpu.consumption >= 0) cpu.consume();
   g();                               g();
   while (x > 0) {                    while (x > 0) {
                                         cpu.consumption += ...;
                           -->           if (cpu.consumption >= 0) cpu.consume();
      if (h(x)) {                        if (h(x)) {
                                            cpu.consumption += ...;
         i(x);                              i(x);
      }                                  }
                                         cpu.consumption += ...;
      --x;                               --x;
   }                                  }
}                                  }
```

Fig. 3. Rewriting of a method for CPU accounting.

2.4 Aggregating CPU Consumption

Normally, each `ThreadCPUAccount` object refers to an implementation of `CPUManager`, which is shared between all threads belonging to a component. [3] Fig. 4 shows part of the `CPUManager` interface. The `CPUManager` implementation is provided by the middleware developer and implements the actual CPU accounting and control strategies, e.g., custom scheduling schemes. The methods of the `CPUManager` interface are invoked by the `consume()` method of `ThreadCPUAccount`.

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

Fig. 4. Part of the `CPUManager` interface.

For resource-aware applications, the `CPUManager` implementation may provide application-specific interfaces to access information concerning the CPU consumption of components, to install notification callbacks to be triggered when the resource consumption reaches a certain threshold, or to modify resource control strategies. In this paper we focus only on the required `CPUManager` interface.

Whenever a thread invokes `consume()` on its `ThreadCPUAccount`, this method will in turn report its collected CPU consumption data (stored in the

[3] In this paper the term 'component' takes the meaning of an informal group of threads subjected to the same `CPUManager` object, and hence, logically (but not necessarily), to the same management policy.

`consumption` field) to the `CPUManager` associated with the `ThreadCPUAccount` (if any) by calling `consume(long)`. `consume()` also resets the `consumption` field. The `consume(long)` method implements the custom CPU accounting and control policy. It may simply aggregate the reported CPU consumption (and write it to a log file or database), it may enforce absolute limits and terminate components that exceed their CPU limit, or it may limit the execution rate of threads of a component (i.e., putting threads temporarily to sleep if a given execution rate is exceeded). This is possible without breaking security assumptions, since the `consume(long)` invocation is synchronous (i.e., blocking), and executed directly by the thread to which the policy applies.

`getGranularity()` 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 of `consume(long)`. The constant $MaxPath$ introduced by the rewriting rule (ii)(e) in Section 2.2 affects the perceived accounting granularity, too. At most $MaxPath$ bytecode instructions may be executed without performing the granularity check. Hence, the effective accounting granularity ranges between the value returned by `getGranularity()` and the sum `getGranularity`() + $MaxPath$.

3 Optimizations

In this section we give an overview of some optimization techniques that reduce the CPU accounting overhead.

3.1 Optimizing Leaf Methods

According to the bytecode transformation rule (ii)(a) in Section 2.2, a granularity check is inserted at the beginning of each method. In general, this is necessary to ensure that the granularity check will be performed in recursive methods.

To improve performance, the check at the beginning of methods can be omitted if each possible execution path terminates (i.e., returns or throws an exception) or passes by an otherwise inserted conditional (e.g., at the beginning of a loop) before any method/constructor invocation. In particular, this optimization pays off for leaf methods.

Nevertheless, this optimization may increase the perceived accounting granularity. A leaf method L with fewer than $MaxPath$ bytecode instructions will

not perform the granularity check. Another method may invoke L in series up to $MaxPath$ times before the granularity check will be enforced. In total, the granularity check may be delayed by up to $MaxPath^2$ bytecode instructions (worst case).

3.2 Passing `ThreadCPUAccount` *as Argument*

As we will show in Section 4, the invocation of `ThreadCPUAccount.getCurrentAccount()` at the beginning of each method causes high overhead, because it requires loading of the `Thread` object representing the current thread. Here we present an optimization that allows to significantly reduce the number of invocations of this method.

3.2.1 Extending Method Signatures

We extend the signatures of methods to take an extra `ThreadCPUAccount` argument and pass the `ThreadCPUAccount` object of the current thread to method and constructor invocations. Fig. 5 illustrates this rewriting scheme for the example given in Fig. 3.

```
void f(int x, ThreadCPUAccount cpu) {
   cpu.consumption += ...;
   if (cpu.consumption >= 0) cpu.consume();
   g(cpu);
   while (x > 0) {
      cpu.consumption += ...;
      if (cpu.consumption >= 0) cpu.consume();
      if (h(x, cpu)) {
         cpu.consumption += ...;
         i(x, cpu);
      }
      cpu.consumption += ...;
      --x;
   }
}
```

Fig. 5. Method rewritten for CPU accounting. The `ThreadCPUAccount` is passed as extra argument.

Because native code may invoke Java methods and we do not modify native code, we have to preserve a method with the same signature as before rewriting. For this reason, we add wrapper methods as shown in Fig. 6, which load the `ThreadCPUAccount` and pass it to the resource-aware methods that take the `ThreadCPUAccount` as extra argument. Compatibility with non-rewritten and non-rewritable code is thus ensured. In the best case, the `ThreadCPUAccount.getCurrentAccount()` method will be invoked only once at program startup, and then the resulting account will flow through the extra arguments during the rest of the execution.

```
void f(int x) {
   ThreadCPUAccount cpu = ThreadCPUAccount.getCurrentAccount();
   cpu.consumption += ...; // account for execution of wrapper
   f(x, cpu);
}
```

Fig. 6. Wrapper method with unmodified signature.

As we do not change native methods, they do not receive the additional `ThreadCPUAccount` argument. Because rewritten Java methods will invoke methods with the extra argument, we provide reverse wrappers for native methods, as depicted in Fig. 7.

```
native void n();

void n(ThreadCPUAccount cpu) {
   cpu.consumption += ...; // account for reverse wrapper
   n();
}
```

Fig. 7. Reverse wrapper for native method.

As passing the `ThreadCPUAccount` as extra argument significantly reduces the CPU accounting overhead, we would like to apply this technique to all classes, including the JDK. However, adding wrapper methods to JDK classes causes some subtle problems. In the JDK certain methods rely on a fixed invocation sequence. Examples include methods in `Class`, `ClassLoader`, `DriverManager`, `Runtime`, and `System`. These methods inspect the stack frame of the caller to determine whether an operation is permitted. If wrapper methods (or reverse wrappers for native methods) are added to the JDK, the additional stack frames due to the invocation of wrapper methods will violate the assumptions of the JDK programmer concerning the execution stack.

3.2.2 Extending Method Signatures in JDK Classes

In order not to violate assumptions regarding the structure of the call stack when a JDK method is invoked, we have to make sure that there are no extra stack frames of wrappers of JDK methods on the stack. A trivial solution is to rewrite the JDK classes according to the transformation shown in Fig. 3. However, as we have mentioned before, such a rewriting scheme may cause high overhead on certain JVMs.

A first step towards a more efficient solution is to ensure that native JDK methods are always invoked directly. That is, reverse wrappers as depicted in Fig. 7 are to be avoided for native JDK methods. For this purpose, we have developed a simple tool to analyze the JDK, which gives out a list of methods

L that must not receive wrappers. This list is needed for the subsequent rewriting of JDK and of application classes, since invocations of methods in L must not pass the extra `ThreadCPUAccount` argument.

Obviously, L includes all native JDK methods. Additionally, we have to consider polymorphic call sites that may invoke native JDK methods. In this case, the extra `ThreadCPUAccount` argument must not be passed, since the target method may be native and lack a reverse wrapper. Hence, if a native method overwrites/implements a method m in a superclass/interface, m has to be included in L. We use the following simple marking algorithm to compute L.

(i) Compute the class hierarchy of the JDK. For each class, store the class name, a reference to the superclass, references to implemented interfaces, and a list of the signatures and modifiers of all methods in the class.

(ii) Mark all native methods.

(iii) Propagate the marks upwards in the class hierarchy. Let m_c be a marked method, which is neither static nor private. Furthermore, let C be the class defining m_c, and A the set of ancestors of C, including direct and indirect superclasses as well as all implemented interfaces. For each class or interface X in A, if X defines a method m_x with the same signature as m_c, which is neither static nor private, mark m_x.

(iv) All marked methods are collected in the list L.

The JDK methods in the list L are rewritten as follows:

- Native methods do not receive the reverse wrapper shown in Fig. 7.
- Abstract methods are not modified; the signature extended with the extra argument is not added.
- The signature of Java methods is not touched either; they are transformed according to the simple rewriting scheme given in Fig. 3.

So far, we have ensured that native JDK methods are always invoked directly. However, there are JDK methods which require that their callers are not invoked through wrappers either. To respect this restriction, the code of each JDK method not included in L is duplicated, as presented in Fig. 8.[4] As there are no wrappers for JDK methods, the call sequence within the JDK remains unchanged. While the code is approximately duplicated (with respect to the rewriting scheme for application classes), the execution performance may be improved, because the `ThreadCPUAccount` is passed as argument

[4] In this sample we assume that methods `g()`, `h(int)`, and `i(int)` are not in the list L. Otherwise, the extra argument must not be passed to them.

whenever possible.

```
void f(int x) {
   ThreadCPUAccount cpu =
      ThreadCPUAccount.getCurrentAccount();
   cpu.consumption += ...;
   if (cpu.consumption >= 0)
      cpu.consume();
   g(cpu);
   while (x > 0) {
      cpu.consumption += ...;
      if (cpu.consumption >= 0)
         cpu.consume();
      if (h(x, cpu)) {
         cpu.consumption += ...;
         i(x, cpu);
      }
      cpu.consumption += ...;
      --x;
   }
}
```

```
void f(int x, ThreadCPUAccount cpu) {

   cpu.consumption += ...;
   if (cpu.consumption >= 0)
      cpu.consume();
   g(cpu);
   while (x > 0) {
      cpu.consumption += ...;
      if (cpu.consumption >= 0)
         cpu.consume();
      if (h(x, cpu)) {
         cpu.consumption += ...;
         i(x, cpu);
      }
      cpu.consumption += ...;
      --x;
   }
}
```

Fig. 8. Rewriting scheme for JDK methods: The code is duplicated.

3.2.3 Improving Register Allocation

Passing the `ThreadCPUAccount` reference as extra argument enables the just-in-time compiler of the JVM to keep it in a processor register (interprocedural register allocation). However, depending on the arity of the methods, the `ThreadCPUAccount` reference appears at different argument positions, which may result in a suboptimal register allocation on certain JVMs.

In order to help register allocation, the `ThreadCPUAccount` reference should be passed in a fixed argument position. Initially, we tried to pass it as the first argument, which however did not improve performance and considerably complicated the rewriting algorithm (detecting the right location to insert instructions to load the `ThreadCPUAccount` reference requires control-flow and stack analysis).

Therefore, we followed a different approach, inserting 'dummy' arguments resulting in methods/constructors of the same arity. For virtual (resp. static) methods, let $N_{virtual} \geq 0$ (resp. $N_{static} \geq 0$) denote the JVM local variable [5] with the smallest index allowed to receive the inserted `ThreadCPUAccount` reference. For a virtual method/constructor (resp. static method) that needs $A_{virtual}$ (resp. A_{static}) JVM local variables to receive its arguments, we append $max(0, N_{virtual} - A_{virtual})$ (resp. $max(0, N_{static} - A_{static})$ 'dummy' arguments of a type that does not occur in the original program to the signature. Upon invocation of the rewritten method, `null` references are passed for the 'dummy' arguments. According to the JVM specification [5], long and double argu-

ments require 2 JVM local variables, while all other types require a single one. Virtual methods/constructors receive the `this` object reference in the JVM local variable 0.

Fig. 9 illustrates the insertion of dummy arguments for $N_{virtual} = N_{static} = 2$. After the transformation, methods 1–2 and 5–7 pass the `ThreadCPUAccount` reference in the same JVM local variable 2.

```
1:        v()             -->          v(Dummy,ThreadCPUAccount)
2:        v(int)          -->          v(int,  ThreadCPUAccount)
3:        v(long)         -->          v(long, ThreadCPUAccount)
4:        v(int,long)     -->          v(int,  long, ThreadCPUAccount)

5: static s()             -->   static s(Dummy,Dummy,ThreadCPUAccount)
6: static s(int)          -->   static s(int,  Dummy,ThreadCPUAccount)
7: static s(long)         -->   static s(long,       ThreadCPUAccount)
8: static s(int,long)     -->   static s(int,  long, ThreadCPUAccount)
```

Fig. 9. Examples of the insertion of 'dummy' arguments.

3.3 Control Flow Optimizations

In order to reduce the accounting overhead, it is necessary to reduce the number of accounting sites in the code. One way to achieve this is to increase the average accounting block size using bytecode transformations that enlarge accounting blocks or reduce the number of them. Currently, J-RAF2 supports two optimizations of this kind. The first one optimizes simple jumps, the second one transforms loops. Both optimizations are applied before the accounting code is inserted.

3.3.1 Jump Optimization

The first optimization aims at avoiding accounting blocks that consist only of a single jump instruction. If accounting code was added to such a block J, the overhead of accounting for the jump instruction would by far exceed the execution cost of the jump itself. Our first optimization tries to shortcut such jumps: First, all branches to block J are redirected to the target of J.

Then, if J is neither the first block in the method, in an exception handler, nor in a JVM subroutine, we try to completely remove J. If the control cannot flow from the block preceding J in the bytecode into the block J, the previous redirection of branches has made J become dead code that can be removed. If the block preceding J does not branch (i.e., control flows only sequentially into J), the block J is joined with its predecessor in the control flow graph, which will result in a combined accounting for both J and its predecessor.

Otherwise, if the predecessor of J branches and control may flow into J, we check whether the code sequence matches the scheme presented in Fig. 10 on the left, where the block preceding J branches to the block following J in the bytecode. This structure occurs frequently in the bytecode generated by standard Java compilers. In this case, we invert the condition of the branch, replace the branch target with the target of the jump, and remove the jump (see Fig. 10 on the right). Whenever we remove a jump, we take care to update all affected exception handlers accordingly.

```
     if (C) goto Then;                    if (!C) goto Else;
J:   goto Else;              -->
Then:                                Then:
```

Fig. 10. Removing a jump.

3.3.2 Loop Optimization

Many Java compilers transform a `while()` loop according to the scheme in Fig. 11. To the left is the original loop, while in the middle is the pseudo code of the compiled loop. X and Y are accounting blocks. The result of inserting accounting instructions is shown in Fig. 11 to the right.[5] That is, for each iteration of the loop, accounting code is executed twice, because the branch occurs in between the execution of C and Y, which are separate accounting blocks.

```
                                          consumption += ...;
X;                X;                      X;
                Start:                  Start:
                                          consumption += ...;
while (C)  -->    if (!C) goto End;  -->  if (!C) goto End;
{                                         consumption += ...;
   Y;             Y;                      Y;
}                 goto Start;             goto Start;
                End:                    End:
```

Fig. 11. Compilation of a `while()` loop and insertion of accounting instructions.

In order to reduce the accounting overhead, the branch can be moved to the end of the loop, which corresponds to transforming a `while()` loop into a `do-while()` loop. Fig. 12 shows the result of this transformation, which allows moving one accounting site outside of the loop. Moreover, if X is an accounting block that does not change the control flow at the end (no branch, no jump, etc.) and if the `while()` loop can only be reached through X, we can combine the accounting for the initial evaluation of C with the accounting

[5] For the sake of easy readability, we omit the granularity checks in this example.

for X, as shown in Fig. 12 to the right. Nevertheless, while this transformation enables a more efficient accounting, it may increase the code size, because the code of the conditional C is duplicated.

```
                                                consumption += ...;
X;                    X;                        X;
if (C) {              if (!C) goto End;         if (!C) goto End;
  do {              Start:                    Start:
            -->                       -->       consumption += ...;
    Y;                Y;                        Y;
  } while (C);        if (C) goto Start;        if (C) goto Start;
}                   End:                      End:
```

Fig. 12. Compilation of transformed loop and insertion of accounting instructions.

In J-RAF2 we have implemented a loop detection and transformation algorithm that works on the control flow graph of a method. The execution time of the algorithm is linear in the number of nodes in the control flow graph. Our loop optimization algorithm detects the loop pattern shown in Fig. 11 in the middle and transforms it into the structure illustrated in Fig. 12 in the middle. The loop transformation is applied only if the code increase is small (e.g., max. 20 JVM bytecode instructions) and if the block containing the branch is covered by exactly the same set of exception handlers as the jump instruction. This is important, as the jump will be replaced with a copy of the block including the branch. Therefore, this condition ensures that both instances of the branch block will be managed by the same set of exception handlers.

4 Evaluation

In this section we present an overview of the benchmarks we have executed to validate our CPU accounting scheme. We ran the SPEC JVM98 benchmark suite [6] on a Linux RedHat 9 computer (Intel Pentium 4, 2.6 GHz, 512 MB RAM). All benchmarks were run in single-user mode (no networking) and we removed background processes as much as possible in order to obtain reproducible results. For all settings, the entire JVM98 benchmark (consisting of several sub-tests) was run 10 times, and the final results were obtained by calculating the geometric mean of the median of each sub-test. Here we present the measurements made with the IBM JDK 1.4.2 platform in its default execution mode, as well as the Sun JDK 1.5.0 (final) platform in its 'client' and 'server' modes. In our test we used a single `CPUManager` with a simple accounting policy (just aggregating the CPU consumption of all threads

[6] http://www.spec.org/osg/jvm98/

in the system) and with the highest possible granularity. An analysis of the performance impact of the granularity can be found in [4].

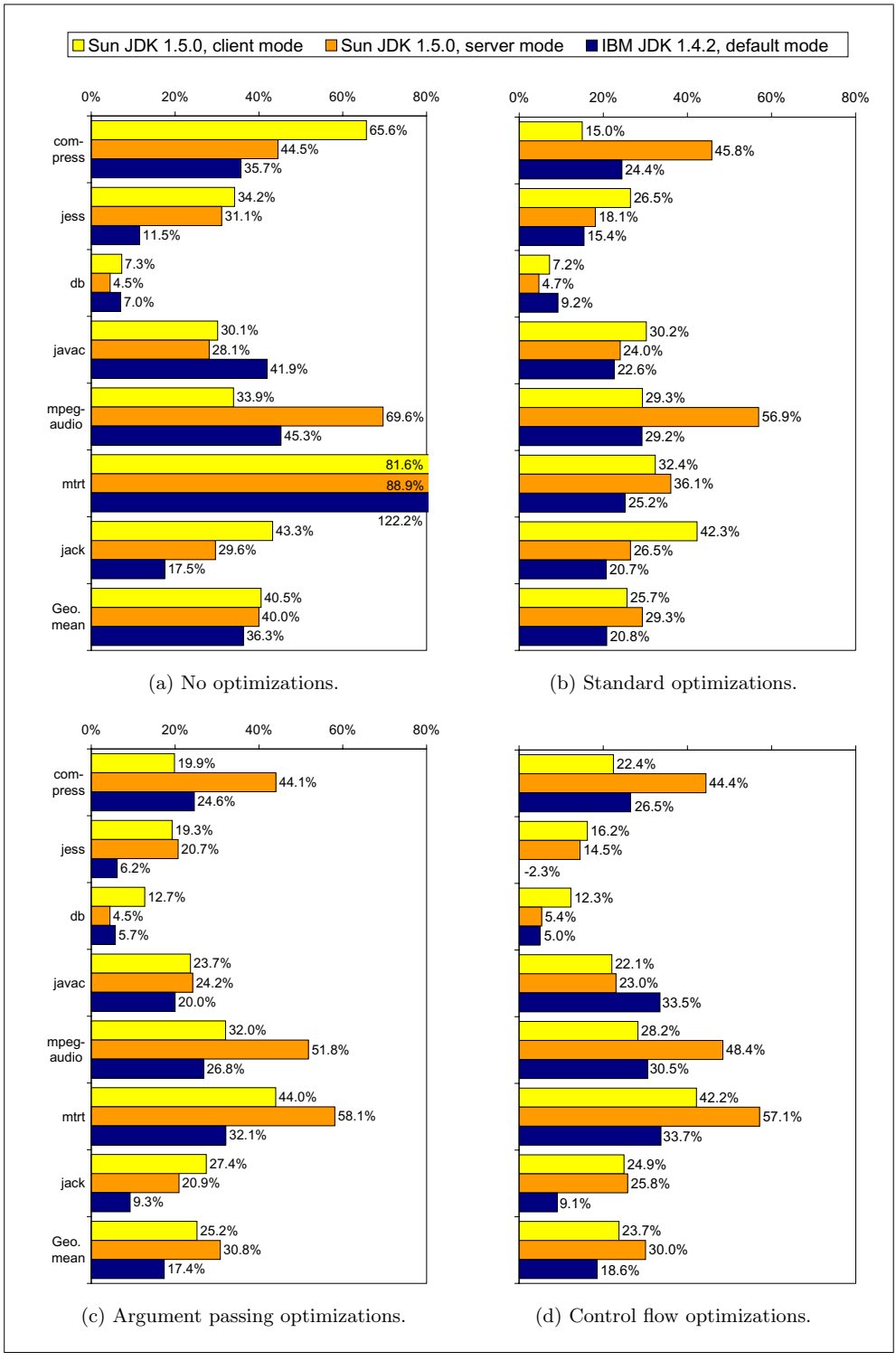
We measured the performance of a rewritten SPEC JVM98 application on top of a rewritten JDK with different optimizations. Fig. 13 (a) shows the overhead when no optimization was applied. On average the overhead is 36–41%.

Fig. 13 (b) illustrates the impact of our optimizations. We applied the optimization for leaf methods explained in Section 3.1 and extended the signature of all methods in the SPEC JVM98 classes in order to pass the `ThreadCPUAccount` as extra argument (see Section 3.2), while the whole JDK was rewritten according to the scheme presented in Fig. 3. We call this set of optimizations our standard optimizations. In this setting the average overhead is reduced to 21–29%.

Fig. 13 (c) shows the impacts of passing the `ThreadCPUAccount` reference also to JDK methods whenever possible (see Fig. 8). Moreover, we examined the impact of passing the `ThreadCPUAccount` in a well-defined argument position. We evaluated a large number of combinations of possible values for $N_{virtual}$ and N_{static}. For IBM's JVM, a setting of $N_{virtual} = N_{static} = 2$ resulted in the lowest average overhead of 17%.

Interestingly, for Sun's JVMs, this scheme did not improve the performance (average overhead 25–31%), which can be explained as follows: On the one hand, the rewriting of the JDK classes significantly increases the code size and hence causes overheads during class loading and just-in-time compilation. On the other hand, the number of invocations of `getCurrentAccount()` (involving a call to `Thread.currentThread()`) is reduced. For JVMs with a rather fast implementation of `Thread.currentThread()`, the overheads due to the increased code size may outweigh the benefits of less invocations to `Thread.currentThread()`.

Finally, we evaluated the impact of the simple control flow optimizations presented in Section 3.3 (see Fig. 13 (d)). We applied first the jump optimization (Section 3.3.1) and second the loop optimization (Section 3.3.2). Afterwards, the classes were transformed with the previous optimizations. While the geometric means are not much different from the previous ones, the results for some benchmarks, in particular 'jess', are interesting. On IBM's JVM there is even a slight speedup despite of CPU accounting (all results are fully reproducible!). This is because the SPEC JVM98 classes have not been compiled with IBM's Java compiler and therefore some code patterns can be improved for IBM's JVM. Further investigations are needed to be able to characterize the code patterns where speedups and where slowdowns are observed. Whereas the practical value of these transformations is not fully



(a) No optimizations.

(b) Standard optimizations.

(c) Argument passing optimizations.

(d) Control flow optimizations.

Fig. 13. Overhead of CPU accounting with different optimizations.

assessed, the observed speedup constitutes a rather unusual result.

In absolute time, on IBM's JVM the benchmarks with accounting executed 20% faster than on Sun's HotSpot Client VM without accounting, and as fast as on Sun's HotSpot Server VM without accounting.

Further reduction of the runtime overhead is still possible, e.g., by extending and generalizing the notion of leaf method with class-level interprocedural analysis, and having callers account for the CPU consumption of callees when possible. Moreover, some speed can be gained in well-chosen code segments, especially inside loops, by temporarily storing consumption values in local variables, which will normally be allocated to registers by the just-in-time compiler. We are also currently refining a heuristics-based optimization which consists of identifying and accounting whole execution paths, instead of working at the level of isolated accounting blocks; this is not so simple, since, for preventing denial-of-service attacks, we have to account for a code sequence before it is actually executed, and since we do not want a wrong guess (of the path that will be taken) to lead to a higher overhead than our current scheme. In addition to this, we are exploring approximation schemes that significantly reduce the number of accounting sites, but result in an imprecise accounting.

5 Discussion

In this section we review the strengths and limitations of our CPU management framework. A discussion of related work on resource management in virtual execution environments can be found in [1].

First and most importantly, our CPU management scheme is fully portable. 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 successfully tested with several standard JVMs in different environments.

As business models where service providers host client software components become more widespread, middleware for servers will have to offer resource management functionality to monitor deployed client applications and to charge them for their resource consumption. Whereas standard JVMs will not be equipped with CPU management capabilities in the foreseeable future (a JSR [7] on resource management has not been initiated yet), J-RAF2 offers a solution today to enhance standard Java runtime systems with CPU management features. An alternative would be to use a modified JVM, but most available JVMs supporting resource management perform significantly worse than standard Java runtime systems.

[7] Java Specification Request: http://www.jcp.org/

J-RAF2 offers a small but flexible API. On the one hand, it supports the installation of custom `CPUManager`s for legacy applications without requiring any manual changes to the classes. On the other hand, middleware developers may exploit the `ThreadCPUAccount` API in order to aggregate CPU consumption for individual components in a flexible way. While J-RAF2 provides only a low-level API, advanced features, such as triggers and callbacks [3], may be added in a `CPUManager` implementation by the programmer.

Our proposal for CPU management is built on the idea of self-accounting. Thus, we probably offer the most precise, fine-grained accounting basis available. Moreover, this approach solves one important weakness of all existing solutions based on a polling supervisor thread: The Java specification does not formally guarantee that the supervisor thread will ever be scheduled, whatever its priority is set to. In contrast, 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. On this basis server and client can agree upon CPU quotas without referring to machine characteristics, such as processor model, clock rate, etc. In distributed applications this is a clear advantage, since we may then envision platform-independent 'execution contracts' between heterogeneous hosts, as well as the specification of platform-independent schedulers and scheduling policies.

Concerning limitations, the major hurdle of our approach is that it cannot directly account for the execution of native code. We believe that a range of diverse solutions must be put to work, some of which we have described previously [2], especially concerning memory attacks. In particular, a combination of memory and CPU control is needed in order to prevent denial-of-service attacks through the garbage collector. Allocating a large amount of objects may require only relatively few bytecode instructions, but may cause considerable work for the garbage collector. A simple but effective solution is to charge applications at the moment of object allocation for the estimated future garbage collection costs [2]. This approach also has the advantage that in the case of termination of a component (or migration of a mobile agent), the component has already been charged for the garbage it leaves in the system.

Certain native functions, such as (de-)serialization and class loading, can be protected with wrapper libraries which inspect the arguments. It is also possible to run a calibration process, once per platform, to evaluate once for all the actual consumption of certain categories of native system calls, e.g., those for which we can safely estimate that they will have a constant

or linear execution time. As a higher-level measure, it can be decided that untrusted applications shall only have restricted access to such native system calls. Whereas these are some answers to the issue of native methods, it remains that different bytecodes have different execution costs (e.g., a simple incrementation of a variable may take several bytecodes). To take this into account, we also propose a calibration process to collect cost information that will be fed into the rewriting tool.

A minor issue concerns exceptions which may cause some imprecision in the accounting, as we always account for all instructions in an accounting block, even though some instructions may not be executed in case of an exception. I.e., we may account for more instructions than have been executed. Normally, this is not a big problem, as the average size of accounting blocks is not very big. Moreover, as exception handling is computationally more expensive, an accounting policy could declare that exception handling will cause extra charges. If it is necessary to avoid this potential imprecision, each JVM instruction that may throw an exception should end an accounting block (in particular, method invocations would end accounting blocks). However, such a scheme would significantly decrease the average size of accounting blocks and therefore cause considerably more accounting overhead. For this reason, we favored a larger average block size at the expense of a possible loss in accounting precision.

6 Conclusion

CPU management based on program transformations offers an important advantage over existing 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 and enhance many existing Internet applications, including server environments, as well as embedded systems.

This paper gives an overview of the new CPU management scheme of J-RAF2, including the self-accounting and control mechanism, the bytecode transformation algorithms, as well as several optimizations. We have shown that thanks to these optimizations, the overhead can be quite reasonable, whereas on the face of it this approach might seem not viable because of performance issues. J-RAF2 does not define a high-level programming model for resource aware middleware and applications, but it is a flexible, low-level system researchers and developers can experiment with and extend.

References

[1] Binder, W. and J. Hulaas, *A portable CPU-management framework for Java*, IEEE Internet Computing **8** (2004), pp. 74–83.

[2] Binder, W., 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)*, USA, 2001.

[3] Czajkowski, G., S. Hahn, G. Skinner, P. Soper and C. Bryce, *A resource management interface for the Java platform*, Technical Report TR-2003-124, Sun Microsystems (2003).

[4] Hulaas, J. 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, 2004, pp. 169–177.

[5] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, Reading, MA, USA, 1999, second edition, xv + 473 pp.