Extending Standard Java Runtime Systems for Resource Management

Walter Binder[1] and Jarle Hulaas[2]

[1] Artificial Intelligence Laboratory, EPFL, CH–1015 Lausanne, Switzerland
[2] Software Engineering Laboratory, EPFL, CH–1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract. Resource management is a precondition to build reliable, extensible middleware and to host potentially untrusted user components. Resource accounting allows 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. In order to accurately monitor the resource utilization of Java applications, the application code as well as the libraries used by the application – in particular, the classes of the Java Development Kit (JDK) – have to be transformed for resource accounting. However, the JDK classes are tightly interwoven with the native code of the Java runtime system. These dependencies, which are not well documented, have to be respected in order to preserve the integrity of the Java platform. We discuss several hurdles we have encountered when rewriting the JDK classes for resource management, and we present our solutions to these problems. Performance evaluations complete this paper.

Keywords: Bytecode rewriting, Java, JDK, program transformations, resource management.

1 Introduction

Resource management (i.e., accounting and controlling physical resources like CPU and memory) 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 host components on behalf of various 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. Such information will currently be furnished in an ad-hoc way by the underlying operating system, but higher software layers would definitely benefit from receiving it through standardized APIs in order to

T. Gschwind and C. Mascolo (Eds.): SEM 2004, LNCS 3437, pp. 154–169, 2005.

enable portable and tightly integrated implementations of policies at the middleware level.

Java [9] and the Java Virtual Machine (JVM) [12] 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 agent systems and middleware that can be extended and customized by mobile code. 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.

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 management in Java-based platforms rely on a modified JVM, on native code libraries, or on program transformations. For instance, KaffeOS [1] and the MVM [6] are specialized JVMs supporting resource control. JRes [7] is a resource control library for Java, which uses native code for CPU control and rewrites the bytecode of Java programs for memory control.

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 middleware. 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]. In this approach the bytecode of 'legacy' applications is rewritten in order to make its resource consumption 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) [4], which has undergone a complete revision in order to provide far better reliability, programmability, and performance. Details concerning the new bytecode rewriting scheme of J-RAF2[1] can be found in [10]. The drawback of this approach is that we cannot account for the resource consumption of native code.

In this paper we focus on the solutions developed specifically for correctly and efficiently rewriting the Java runtime support, called the Java Development Kit (JDK). Typically, rewriting the bytecode of an application is not sufficient to account and control its resource consumption, because Java applications use the comprehensive APIs of the JDK. Therefore, resource-aware versions of the JDK classes are needed in order to monitor the total resource consumption of an application. Ideally, the same bytecode rewriting algorithm should be used to rewrite application classes as well as JDK classes. However, the JDK classes are tightly interwoven with native code of the Java runtime system, which causes subtle complications for the rewriting of JDK classes. In this paper we report on the difficulties we encountered with JDK rewriting and on our solutions

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

to these problems. While we describe the problems and solutions in the context of CPU management, they apply for memory management in a similar way. Finally, we present benchmark results of different strategies for JDK rewriting on various Java runtime systems.

This paper is structured as follows: In the next section we explain the basic idea of our program transformations for CPU management. In section 3 we discuss the problems in applying these transformations to the JDK classes, and in section 4 we show the necessary refinements for JDK rewriting. Section 5 presents a tool that helped us extend certain JDK classes. In section 6 we discuss performance measurements of different program transformation strategies. Finally, section 7 concludes this paper.

2 Transformation for Resource Control

In our approach the bytecode of each Java method is rewritten to expose its CPU consumption.[2] Every thread has an associated `ThreadCPUAccount` that is updated while the thread is executing the rewritten code. In each basic block of code the number of executed bytecode instructions is added to a counter within the `ThreadCPUAccount`. Periodically, the counter is checked and if it exceeds a dynamically adjustable threshold, a method is invoked on the `ThreadCPUAccount` which reports the thread's CPU consumption to a user-defined CPU manager. As multiple `ThreadCPUAccounts` may be associated with the same CPU manager, the manager is able to aggregate the CPU consumption of a set of threads. For instance, a CPU manager may be responsible for a set of threads executing within a component (such as a Servlet or a mobile agent). The CPU manager may implement an application-specific accounting, scheduling, and controlling policy. For example, the CPU manager may log the reported CPU consumption, it may try to terminate threads that exceed their allowed CPU quota, or it may delay threads if their execution rate is too high. Because the task of executing scheduling policies is distributed among all threads in the system, we call this approach *self-accounting*. While the details and APIs of our self-accounting scheme are presented in [10, 2], in this paper we exclusively focus on the particularities of transforming the classes of standard Java runtime systems.

The code in table 1 illustrates how the `ThreadCPUAccount` could be bound to a thread using a thread-local variable (thread-local variables are bound to `Thread` instances, i.e., each thread has its own copy). The method `ThreadCPUAccount.getCurrentAccount()` returns the `ThreadCPUAccount` associated with the calling thread. The thread-local variable has to be set whenever a thread is created.

As each Java method has to access the `ThreadCPUAccount`, a simple transformation scheme may load the `ThreadCPUAccount` on method entry as shown in table 2.[3]

[2] The rewriting operation is normally performed statically, and this can happen either once for all (as would be expected in the case of standard classes like the JDK), or at load-time (for application-level classes).

[3] For the sake of easy readability, we show the transformation at the level of the Java language, whereas our implementation operates at the JVM bytecode level.

Table 1. Binding `ThreadCPUAccounts` to threads using thread-local variables

```
public class ThreadCPUAccount {
  private static final ThreadLocal currentAccount = new ThreadLocal();

  public static ThreadCPUAccount getCurrentAccount() {
    return (ThreadCPUAccount)currentAccount.get();
  }
  ...
}
```

Table 2. Simple rewriting scheme: The `ThreadCPUAccount` is loaded on method entry

```
void f() {
  ThreadCPUAccount cpu = ThreadCPUAccount.getCurrentAccount();
  ... // method code with accounting
  g();
  ...
}
```

Table 3. Optimized rewriting scheme: The `ThreadCPUAccount` is passed as extra argument

```
void f(ThreadCPUAccount cpu) {
  ... // method code with accounting
  g(cpu);
  ...
}
```

Here we only show the method entry as well as the exemplary invocation of a method `g()`, whereas the actual accounting code is not presented in this paper (see [10] for details).

Unfortunately, it turns out that an entry sequence as depicted in table 2 causes high overhead. Access to thread-local variables requires loading of the `Thread` object representing the currently executing thread. Thus, we opted for a different transformation scheme as illustrated in table 3. In this approach the `ThreadCPUAccount` is passed as additional argument to methods/constructors. This scheme works insofar as all invocation sites are updated to provide the additional actual argument. 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.

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.[4] For this reason, we add wrapper methods as shown in table 4, which load the `ThreadCPUAccount`

[4] One example is the `main()` method, which by convention has to have exactly one argument, an array of strings containing the command-line arguments; the `main()` method will very likely be invoked at startup by native code, or by Java code making use of the reflection API, hence the invocation sites cannot be updated by our systematic scheme.

Table 4. Wrapper method with unmodified signature

```
void f() {
  ThreadCPUAccount cpu = ThreadCPUAccount.getCurrentAccount();
  ... // account for execution of wrapper
  f(cpu);
}
```

Table 5. Reverse wrapper for native method

```
void n(ThreadCPUAccount cpu) {
  ... // account for execution of reverse wrapper
  n();
}

native void n();
```

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.

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 table 5.

3 Applying the Transformation to JDK Classes

While the transformations presented in the previous section are conceptually simple and work well with application classes, they cannot be directly applied to JDK classes. In this section we summarize the difficulties we encountered when rewriting the JDK. In the following section we will elaborate solutions to these problems.

3.1 Thread Without `Thread` Object

The implementation of `ThreadCPUAccount.getCurrentAccount()` invokes `ThreadLocal.get()`, which calls `Thread.currentThread()`. During the bootstrapping of the JVM there is no `Thread` object associated with the thread that loads and links the initial JDK classes. If `Thread.currentThread()` is executed during the bootstrapping process, it will return `null`. Hence, at this initial stage thread-local variables must not be used. Consequently, if we use the `ThreadCPUAccount` implementation shown in the previous section and rewrite all JDK classes (including, for instance, the static initializers of `Object` and `Thread`) according to the scheme presented before, the bootstrapping of the JVM will fail.

3.2 Endless Recursion When Accessing Thread-Local Variables

Another problem is related to the implementation of the class `ThreadLocal`. If all JDK classes – including `ThreadLocal` – are rewritten according to the transformations given in the previous section, the execution of

`ThreadCPUAccount.getCurrentAccount()` will result in an endless recursion, since the wrappers of the `ThreadLocal` methods will invoke `ThreadCPUAccount.getCurrentAccount()` again.

3.3 Native Code Depending on a Fixed Call Sequence

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 shall be 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.

While in [13] the authors claim to have added wrapper methods to all JDK methods without any problems, we discovered the problem mentioned before during the execution of the SPEC JVM98 benchmarks [14]. The problem was not easy to detect, simple applications may execute successfully in a JDK with wrapper methods.

4 Solving the Difficulties of JDK Rewriting

In this section we refine the implementation of `ThreadCPUAccount.getCurrentAccount()` and the transformation rules for the rewriting of JDK classes.

4.1 Refined Implementation of `ThreadCPUAccount`

To solve the problems discussed in sections 3.1 and 3.2, we decided to avoid the JDK implementation of thread-local variables and to attach the `ThreadCPUAccount` directly to the `Thread` object. For this purpose, we add the field '`public ThreadCPUAccount org_jraf2_cpuAccount;`' to the `Thread` class.[5] Moreover, we modify the `Thread` constructors in order to allocate a new instance of `ThreadCPUAccount` and to store it in the field `org_jraf2_cpuAccount`. Consequently, whenever a `Thread` is allocated, it will receive its own `ThreadCPUAccount` instance. For these modifications, we have developed a simple but convenient tool to patch and extend legacy code, which we present in section 5. Another advantage of this approach is that the access to the `org_jraf2_cpuAccount` variable is faster than using the JDK implementation of thread-local variables, because we avoid the lookup in a hashtable.

In table 6 we show some part of the code of the refined implementation of `ThreadCPUAccount`. If `Thread.currentThread()` returns `null` during the bootstrapping of the JVM, a default `ThreadCPUAccount` is returned. This simple check solves the problem outlined in section 3.1. Moreover, during bootstrapping `Thread.currentThread()` may return a `Thread`

[5] To ensure that malicious applications do not directly access the added public variable, we verify each class before rewriting to ensure that it does not refer to that variable. The methods of the reflection API can be modified as well in order to prevent access to the variable.

Table 6. Implementation of `ThreadCPUAccount` based on a modified `Thread` class

```
public class ThreadCPUAccount {
  private static final ThreadCPUAccount defaultCPU = new ThreadCPUAccount();

  public static ThreadCPUAccount getCurrentAccount() {
    Thread t = Thread.currentThread();
    if (t == null) return defaultCPU;
    ThreadCPUAccount cpu = t.org_jraf2_cpuAccount;
    return cpu == null ? defaultCPU : cpu;
  }
  ...
}
```

object which has not yet been completely initialized. In this case, a default `ThreadCPUAccount` is returned, too. To avoid an endless recursion when calling `ThreadCPUAccount.getCurrentAccount()` (see section 3.2 for details), we have to ensure that `Thread.currentThread()` does not receive a wrapper calling `ThreadCPUAccount.getCurrentAccount()`. Usually, this is not an issue, if `Thread.currentThread()` is implemented as a native method. For the same reason, the method `ThreadCPUAccount.getCurrentAccount()` itself is excluded from rewriting.

4.2 Analysis and Refined Rewriting of 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 table 2. 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 table 5 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*.

1. 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.
2. Mark all native methods.
3. 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

Table 7. Rewriting scheme for JDK methods: The code is duplicated

```
void f() {
  ThreadCPUAccount cpu = ThreadCPUAccount.getCurrentAccount();
  ... // method code with accounting
  g(cpu);
  ...
}

void f(ThreadCPUAccount cpu) {
  ... // method code with accounting
  g(cpu);
  ...
}
```

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.

4. 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 table 5.
- 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 table 2.

So far, we have ensured that native JDK methods are always invoked directly. However, as we have mentioned in section 3.3, 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 table 7.[6] 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 does not suffer significantly, because the `ThreadCPUAccount` is passed as argument whenever possible.

5 Extending Java Legacy Code

We have developed a simple tool called MergeClass, which allows to insert new functionality into compiled Java classfiles. With the aid of MergeClass it is possible to plant new features into standard JVMs and to experiment with the modified JVM. After the desired extension has been written in Java and has been compiled, MergeClass directly merges it into given classfiles. This approach is more favorable than resorting to low-level tools, such as disassemblers, assemblers, or decompilers, which would require to patch each class manually and separately. Moreover, many Java decompilers have problems to correctly decompile certain Java classfiles, e.g., obfuscated classes. Compared

[6] In this sample we assume that method `g()` is not in the list *L*. Otherwise, the extra argument must not be passed to `g()`.

with tools for aspect-oriented programming, such as AspectJ [11], our MergeClass tool is simple and limited, but it is easy to use, it does not require to learn new language features, and it enables a very fast development and experimentation cycle.

5.1 The MergeClass Tool

The MergeClass tool takes 3 or 4 arguments. The first two arguments refer to existing Java classfiles. The first one is the class to be extended (usually legacy code), the second one is the extension to be merged into the first one (usually developed in Java and compiled with a standard Java compiler). The third argument specifies the output file to hold the resulting Java classfile. The fourth argument is optional, it defines a configuration file to parametrize the merging process.

MergeClass reads the original input classfile I_O and the extension input classfile I_E. In order to merge I_E into I_O, I_E has to fulfill several restrictions:

- I_E must extend one of the following 3 classes:
  1. `java.lang.Object`, allowing to merge simple extensions that are independent of I_O.
  2. The superclass of I_O, enabling the merging of classes with the same superclass.
  3. I_O, allowing to merge the most specific subclass into its superclass (if all constraints are met, this process may be iterated).
- I_E must not define more than 1 constructor. The constructor must not take any arguments. (Its signature has to be '`()V`'.)
- I_E must not have inner classes.

The resulting output class O has the same name, accessibility, and superclass as I_O. It implements the union of the interfaces implemented by I_O and I_E. O is final if I_E is final, i.e., the extensibility of I_E overrules the extensibility of I_O. (This feature may be used to 'open' a final legacy class by adding interfaces and making it extensible.)

If no special configuration file is defined, MergeClass first copies all members (fields, methods, and constructors) of I_O into an in-memory representation of O. Then it copies or merges all members of I_E into O with the following transformations or checks:

T1: All references to the name of I_E have to be replaced with a reference to the name of I_O.

T2: If I_E extends I_O and code in I_E invokes a private method in I_O (how this is possible will be explained at the end of this section), the `invokevirtual` bytecode instruction is replaced with an `invokespecial` bytecode instruction, as for private methods (in the resulting class) `invokespecial` has to be used.

C1: If there is a name clash (i.e., I_E and I_O define a field with the same name or a method with the same name and signature), an exception is thrown and no result classfile is created (as will be explained later, the optional configuration file can be used to resolve such name conflicts).

The static initializer and the constructor of I_E cannot be simply copied into O, but they have to be integrated with the code copied from I_O. More precisely, if I_O and I_E both define a static initializer, the code in the static initializer of I_E has to be appended

to the code of the static initializer taken from I_O. In a similar way, if I_E defines a non-trivial constructor (i.e., a constructor that does more than just invoking the superclass constructor), the code of the constructor of I_E has to be appended to the code of each constructor taken from I_O. The following transformations are necessary to append code from I_E to code from I_O:

T3: `return` instructions in the code taken from I_O are replaced with `goto` instructions that jump to the begin of the code to be appended from I_E. Redundant `goto` instructions are removed.
T4: As the structure of the appended code remains unchanged, exception handlers in I_E are to be preserved.
T5: In the constructor of I_E the initial invocation of the superclass constructor is stripped off.

With the aid of a special configuration file, the merging process can be customized. The configuration file allows to deal with name clashes and to mark special methods in I_E whose code shall be merged into certain methods of I_O.

For each member in I_E (except for the constructor and static initializer), the configuration file may define one of the following properties:

- `DiscardInConflict`: If there is a name clash, the member in I_O will be preserved.
- `TakeInConflict`: If there is a name clash, the member in I_E will replace the member in I_O.

Moreover, for a void method M in I_E that takes no arguments, the property `InsertAtBegin(regular expression)` or `AppendAtEnd(regular expression)` may be defined. As a consequence, M will not be copied directly into O, but its code will be merged into all methods of I_O whose name match the given regular expression. If M is a static method, only static methods in I_O are considered for a match, otherwise only instance methods may match.

The code of M may be inserted in the beginning or at the end. If it is appended at the end, the transformations T1–T4 are applied. In addition, the following transformation is needed:

T6: If M is appended to a non-void method in I_O, the method result is stored in an otherwise unused local variable and each `return` instruction in the appended code is replaced with a code sequence to load the result onto the stack and to return it.

If the code of M is to be inserted at the beginning, the transformations T1, T2, and T4 have to be complemented with the following transformations (note that T3 is replaced with T7):

T7: `return` instructions in the code to be inserted from I_E are replaced with `goto` instructions that jump to the begin of the code taken from I_O. Redundant `goto` instructions are removed.
T8: All local variable indices in the code inserted from I_E are incremented accordingly in order to avoid clashes with local variables used in the code of I_O. For an instance

method, the local variable 0 remains unchanged, since by default it holds the reference to `this`. This transformation ensures that the inserted code cannot mess up with the arguments passed to the method taken from I_O.

If multiple `InsertAtBegin` or `AppendAtEnd` expressions apply to a given method in I_O, the code merging happens in the order of the definition of the properties in the configuration file.

As mentioned before, it may be necessary that I_E references private or package-visible members in I_O. For this purpose, we offer a complementary tool, MakePublicExtensible, which takes as arguments the names of two Java classfiles, an input file and an output file. The output file is created by making the input class public and non-final, as well as making all its members public. Hence, the Java source of I_E may extend the class that results of applying MakePublicExtensible and access all of its members. The compilation of the Java source of I_E will succeed, because the accessibility and extensibility constraints have been removed. Afterwards, I_E is merged with the original I_O. Of course, in the merged class the code taken from I_E may access all members. The class resulting from applying MakePublicExtensible is discarded. It is only needed temporarily in order to be able to compile the sources of I_E.

5.2 Extending `Thread` Using MergeClass

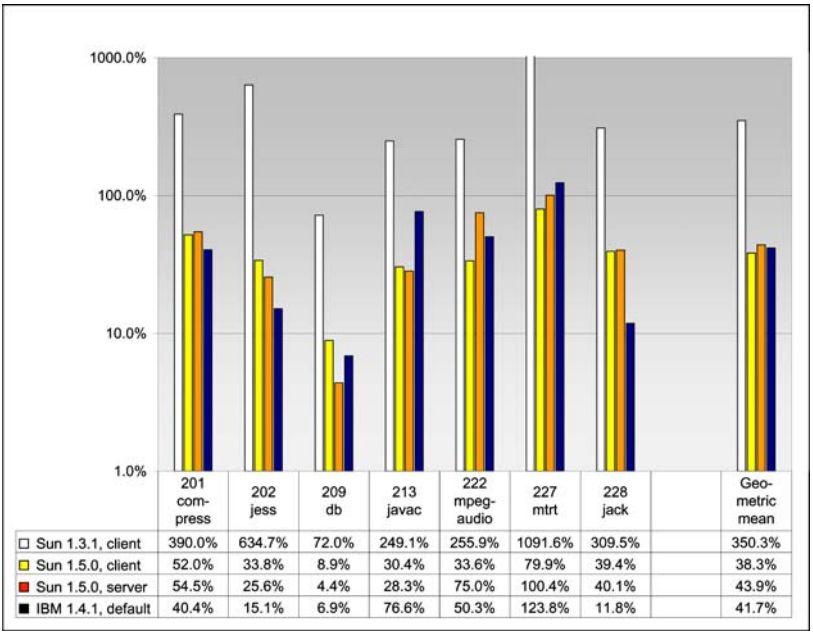
Table 8 illustrates how the `Thread` extensions described in section 4 can be separately implemented, compiled, and injected into the `Thread` class using our MergeClass tool. MergeClass adds the field `org_jraf2_cpuAccount` to `java.lang.Thread`. Moreover, it appends the allocation of a `ThreadCPUAccount` object to each constructor in Thread.

Table 8. `Thread` extension

```
public class ThreadExtension extends java.lang.Thread {
  public ThreadCPUAccount org_jraf2_cpuAccount;

  // to be appended to each constructor:
  public ThreadExtension() {org_jraf2_cpuAccount = new ThreadCPUAccount();}
}
```

Using MergeClass we have been able to experiment with different strategies of maintaining accounting objects. We have implemented the thread extensions in pure Java and compiled them with a standard Java compiler. Furthermore, we were able to test the thread extensions with various versions of the JDK without any extra effort (apart from applying MergeClass). We also integrated some more elaborate features into the `Thread` class, for instance a mechanism that initializes the new `ThreadCPUAccount` with the CPU manager of the calling thread's `ThreadCPUAccount`. This ensures that a spawned thread will execute under the same CPU accounting policy as its creator thread. For details concerning CPU managers, see [10]. Moreover, the thread extensions can be easily adapted for other accounting objects, such as memory accounts (for details concerning memory accounting in Java, see [3]).



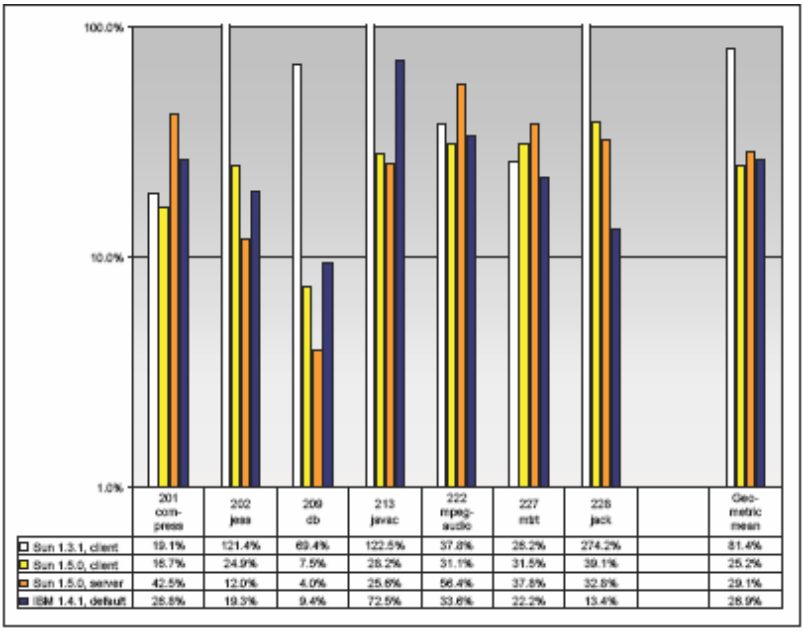
	201 compress	202 jess	209 db	213 javac	222 mpeg-audio	227 mtrt	228 jack		Geometric mean
Sun 1.3.1, client	390.0%	634.7%	72.0%	249.1%	255.9%	1091.6%	309.5%		350.3%
Sun 1.5.0, client	52.0%	33.8%	8.9%	30.4%	33.6%	79.9%	39.4%		38.3%
Sun 1.5.0, server	54.5%	25.6%	4.4%	28.3%	75.0%	100.4%	40.1%		43.9%
IBM 1.4.1, default	40.4%	15.1%	6.9%	76.6%	50.3%	123.8%	11.8%		41.7%

Fig. 1. Overhead of CPU accounting: JDK and benchmarks transformed with the simple scheme

6 Evaluation

In this section we present some benchmark results comparing the accounting overhead of different rewriting strategies on various JVMs. We ran the SPEC JVM98 benchmark suite [14] 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 IBM's JDK 1.4.1 platform in its default execution mode, with Sun's JDK 1.3.1 in its 'client' mode, as well as with Sun's JDK 1.5.0 beta platform in its 'client' and 'server' modes. While IBM JDK 1.4.1 and Sun JDK 1.5.0 represent recent JVMs with state-of-the-art just-in-time compilers, we intentionally added an older JVM for comparison.

The most significant setting we measured was the performance of a rewritten JVM98 application on top of a rewritten JDK. Figure 1 shows the relative overhead of the *simple* transformation scheme of table 2 applied to the JDK as well as to the benchmark classes. In the beginning of each method `Thread.currentThread()` is invoked. We expected this rewriting scheme to result in high overhead on certain JVMs (worst case). In particular, the older Sun JDK 1.3.1 performs badly in this setting, the overhead is up to 1090% for the mtrt benchmark, the geometric mean is about 350% overhead. Apparently, `Thread.currentThread()` is not implemented efficiently on this JVM. For the other JVMs, the average overhead is about 40% in this setting. Because of this big difference, we used a logarithmic scale in figure 1.



	201 compress	202 jess	209 db	213 javac	222 mpeg-audio	227 mtrt	228 jack		Geometric mean
Sun 1.3.1, client	19.1%	121.4%	69.4%	122.5%	37.8%	28.2%	274.2%		81.4%
Sun 1.5.0, client	16.7%	24.9%	7.5%	28.2%	31.1%	31.5%	39.1%		25.2%
Sun 1.5.0, server	42.5%	12.0%	4.0%	25.6%	56.4%	37.8%	32.8%		29.1%
IBM 1.4.1, default	25.8%	19.3%	9.4%	72.5%	33.6%	22.2%	13.4%		28.9%

Fig. 2. Overhead of CPU accounting: JDK transformed with the simple scheme, benchmarks with the wrapper scheme

For the measurements in figure 2, the JDK classes were rewritten according to the *simple* transformation scheme (table 2), whereas the benchmark classes were transformed using the *wrapper* strategy of table 3. In this setting, Sun JDK 1.3.1 incurred an overhead of about 80%, whereas on the other JVMs we measured only 25–30% overhead.

For figure 3, the JDK classes were transformed based on the scheme of table 7 (*no wrappers*, code duplication), while the benchmark classes were rewritten according to the *wrapper* scheme of table 3. In this setting, all JVMs incur an overhead of 30–40%. For Sun JDK 1.3.1, this scheme gives the best results. Interestingly, for the other more recent JVMs, this scheme does not improve the performance, 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 `Thread.currentThread()` is reduced. For recent JVMs with a rather fast implementation of `Thread.currentThread()` this rewriting scheme does not pay off, but for the older Sun JDK 1.3.1 the benefits of reducing the number of `Thread.currentThread()` invocations outweigh the overheads due to the increased code size.

We can conclude that current JDKs shall be transformed using the simple scheme, whereas older releases will perform better when rewritten with code duplication. For libraries other than the JDK and for application classes, the wrapper scheme performs best.

Note that we did not apply any optimization apart from passing `ThreadCPUAccount` objects as extra arguments. Currently, we are evaluating optimizations that reduce the number of accounting sites in the rewritten code. For

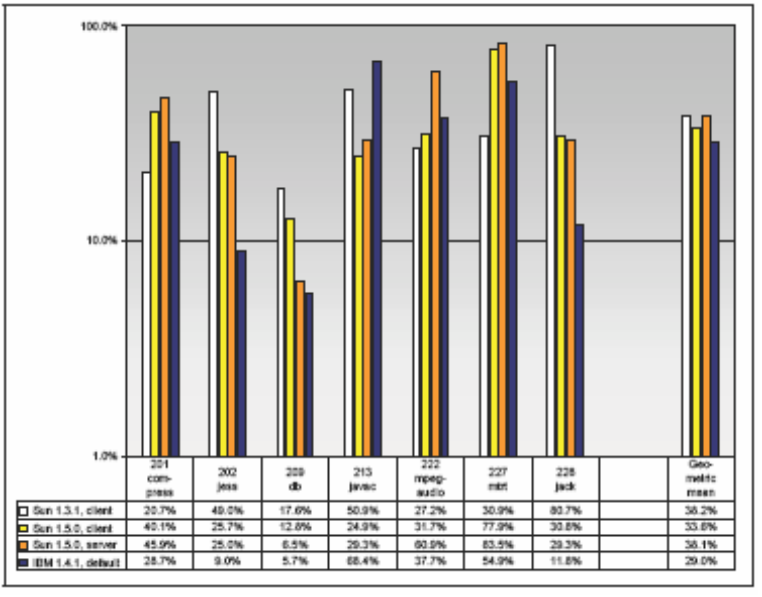


Fig. 3. Overhead of CPU accounting: JDK transformed with code duplication, benchmarks with the wrapper scheme

instance, we are working on control flow optimizations that increase the average size of basic blocks of code and therefore reduce the proportion of accounting instructions during the execution of rewritten programs. In particular, we are considering special cases of loop unrolling which do not result in a significant increase of the code size, but allow to remove accounting instructions from loops. Initial measurements of optimized CPU accounting, which are not presented in this paper due to space limitations, indicate that such optimizations allow to further reduce the overhead to about 20% and below on current standard JVMs.

The rewriting process itself takes only a very short time, although our tools have not yet been optimized to minimize the rewriting time. For example, rewriting the 20660 methods of the 2790 core classes of IBM's JDK 1.4.1 takes less than one minute on our test machine. The implementations of our tools are based on the bytecode engineering library BCEL [8], which provides an object representation for the individual bytecode instructions. Optimizing our tools for dynamic rewriting of classes during class loading may require resorting to a lower-level bytecode representation.

We have also considered using tools for aspect-oriented programming in order to define the sites where accounting is needed and to insert the accounting code there. However, in our approach accounting sites are ubiquitous, as they are based on the low-level concept of basic blocks of code. Most tools for aspect-oriented programming, such as AspectJ [11], allow to define only higher-level pointcuts, such as method invocations, the beginning of exception handlers, etc. With such tools it is not easily possible to express that accounting code is to be inserted into each basic block of code. Moreover, it would be difficult to specify our particular scheme of passing acocunting objects as

extra arguments (which involves the creation of wrapper methods or the duplication of code) with current aspect languages. For our purpose, the direct manipulation of the JVM bytecode is the best suited approach, which allows us to implement low-level transformations and optimizations.

7 Conclusion

Program transformation techniques allow to transparently integrate resource management into Java-based systems, although current Java runtime systems do not support this feature. To account for the total resource consumption of an application component, it is not sufficient to rewrite its classes, but all used libraries, including middleware and JDK classes, have to be transformed, too.

In this paper we have outlined the difficulties of modifying the classes of standard JDKs. The native code of the Java runtime system relies on several low-level assumptions regarding the dependencies of Java methods in certain JDK classes. Thus, program transformations that are correct for pure Java may break native code in the runtime system. Unfortunately, these dependencies are not well documented, which complicates the task of defining transformation rules that work well with the Java class library.

Moreover, the transformed JDK classes may seem to work as desired even with large-scale benchmarks, while the transformation may have compromised the security model of Java. Such security malfunctions are hard to detect, as they cannot be perceived when running well behaving applications. We have experienced that a minor restructuring of the method call sequence completely breaks several security checks, which are based on stack introspection and assume a fixed call sequence. Consequently, modifications and updates of the JDK are highly error-prone.

In this paper we have presented program transformations for resource management, in particular focusing on CPU accounting, which are applicable to application classes as well as to the Java class library. We have developed different transformation schemes and evaluated their respective performance. The most elaborate scheme results in an overhead for CPU accounting of about 25–30% (without optimizations to reduce the number of accounting sites).

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 J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, 8(5):74–83, Sep./Oct. 2004.
3. W. Binder, J. Hulaas, and A. Villazón. Resource control in J-SEAL2. Technical Report Cahier du CUI No. 124, University of Geneva, Oct. 2000. `ftp://cui.unige.ch/pub/tios/papers/TR-124-2000.pdf`.
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)*, 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. 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.
7. 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*, New York, USA, Oct. 1998.
8. M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. `http://jakarta.apache.org/bcel/`.
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. J. Hulaas and W. Binder. Program transformations for portable CPU accounting and control in Java. In *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, pages 169–177, Verona, Italy, August 24–25 2004.
11. I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, 2003.
12. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
13. A. Rudys and D. S. Wallach. Enforcing Java run-time properties using bytecode rewriting. *Lecture Notes in Computer Science*, 2609:185–200, 2003.
14. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`.