Program Transformations for Light-Weight CPU Accounting and Control in the Java Virtual Machine

A Systematic Review

Jarle Hulaas
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH–1015 Lausanne, Switzerland
*(*jarle.hulaas@acm.ch*)*

Walter Binder
Faculty of Informatics
University of Lugano
CH–6904 Lugano, Switzerland
*(*walter.binder@unisi.ch*)*

Abstract.
This article constitutes a thorough presentation of an original 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 even the standard Java runtime libraries (i.e., the Java Development Kit) are modified in a fully portable way, in order to expose details regarding the execution of threads. These transformations however incur a certain overhead at runtime. Further contributions of this article are the systematic review of the origin of such overheads and the description of a new static path prediction scheme targeted at reducing them.

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

1. Introduction

Resource management (i.e., accounting and controlling the consumption of resources, such as CPU and memory) is extremely useful for monitoring deployed software. Run-time monitoring of server systems is important to quickly detect performance problems and to tune the system according to the workload. Resource management also is a prerequisite to prevent malicious or accidental resource overuse, such as denial-of-service attacks, in extensible middleware that allows hosting of foreign, untrusted software components. In commercial application servers, providers may charge their clients for the resources consumed by executed software components; the corresponding contracts should then state the maximal quantities of computing resources that the client is allowed to use, preferably in terms of platform-independent metrics

Published in the Journal of *Higher-Order and Symbolic Computing (HOSC)*, 2008.
Article DOI: 10.1007/s10990-008-9026-4

such as the number of executed bytecodes. In emerging agent-oriented, context-aware software systems, self-tuning abilities are expected; these will in turn require awareness of resource availability and usage policies. Lastly, in resource-constrained embedded systems, software has to be aware of resource restrictions in order to prevent abnormal termination.

Currently, predominant programming languages and environments, such as Java [19] and the Java Virtual Machine (JVM) [26], lack standardized resource management mechanisms. Whereas some prototypes have been proposed to address this lack (see the related work section), they are unfortunately all dependent on substantial amounts of native code, and thus prevent the deployment of resource-managed or resource-aware systems throughout widely heterogeneous networks. Therefore, we propose *portable resource management* with the aid of program transformations. We call our approach J-RAF2 (Java Resource Accounting Framework, 2nd edition) [4, 7, 21], which has been implemented in a tool with the same name.[1] J-RAF2 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 Java processors, which provide a JVM implemented in hardware that cannot be easily modified [8].

This article concentrates on CPU management, since this is a very useful, and at the same time particularly challenging, resource to study. Program transformations have also been applied to the management of other kinds of resources, such as memory management [7, 14], but the low run-time overheads obtained did not require the development of such advanced analysis and transformation techniques as the ones presented here; notably, they did not involve any control flow or data flow analysis.

The program transformations underlying J-RAF2 were first published in reference [7]. As the transformations for CPU management incur a certain overhead at runtime, we started developing several optimizations [21]. Applying these techniques to the JDK itself is a difficult task [3]. Finally, we have also presented and illustrated the actual use of J-RAF2 by an application or middleware developer [4].

The present article is an expanded version of [21]. Its specific, original contributions are:

1. A systematic analysis of the origin of the overheads induced by our CPU accounting scheme.

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

2. A complete description of our optimization schemes targeted at reducing the overheads. One of them, a static path prediction scheme, is novel as compared to our earlier work.

This article is structured as follows: In Section 2 we present the design goals pursued. Section 3 introduces the basic ideas and principles of CPU accounting through bytecode instrumentation. Section 4 analyses the origin of various overheads that are due to the extra bytecode instructions introduced by the instrumentation, and the following two sections propose and evaluate several optimization schemes for reducing these overheads. Section 7 evaluates the overall performance of applications using our new CPU accounting scheme and discusses the benefits and limitations of our approach. This article ends with related work and a conclusion.

2. Design Goals

Traditionally, the CPU consumption of a program is measured in seconds. This approach, however, has several drawbacks: It is platform-dependent (for the same program and input, the CPU time differs depending on hardware, operating system, and virtual machine), measuring it accurately may require platform-specific features (such as special operating system functions) limiting the portability of the CPU management services, and the resulting CPU consumption may not be easily reproducible, as it may depend on external factors such as the system load.

For these reasons, we use the *number of executed JVM bytecode instructions* as our CPU consumption metric. While this metric is not directly translatable into real CPU time, it has many advantages:

- **Platform-independence**: The number of executed bytecode instructions is a platform-independent, dynamic metric [17]. It is independent of the hardware and virtual machine implementation (e.g., interpretation versus just-in-time compilation). However, the availability of different versions and implementations of the Java class library (the classes of the Java development kit) may limit the platform-independence of this metric.
- **Reproducibility**: For deterministic programs, the CPU consumption measured in terms of executed bytecode instructions is exactly reproducible, if the same Java class library is used. However, reproducibility cannot be guaranteed for programs with non-deterministic thread scheduling.

- **Comparability**: CPU consumption statistics collected in different environments are directly comparable, since they are based on the same platform-independent metric.

- **Portability and compatibility**: Because counting the number of executed bytecode instructions does not require any hardware- or operating system-specific support, it can be implemented in a fully portable way. Our CPU management scheme is implemented in pure Java and it is compatible with any standard JVM.

Our CPU management scheme, which is presented in detail in Section 3, supports the installation of *CPU managers* that are periodically activated at run-time in order to collect information regarding the number of executed bytecode instructions and to enforce CPU consumption policies. The following design goals underlie our CPU management approach:

- **Extensibility**: Concrete CPU management policies are not hard-coded in our scheme. User-defined CPU managers may implement custom CPU consumption policies and schedulers in pure Java.

- **Fine-grained activation control**: Our CPU management scheme offers a fine-grained, dynamically adjustable activation control of CPU managers. CPU managers themselves specify the interval between subsequent activations. This interval is expressed in terms of the number of bytecode instructions to be executed until the CPU manager will be re-activated.

- **Deterministic activation**: CPU managers are activated in a deterministic way by each thread. For each thread, the activation of a CPU manager depends neither on a timer, nor on the priority of the thread. Independent of the JVM, a thread activates a CPU manager after the execution of a given number of bytecode instructions. Hence, the activation of CPU managers does not rely on the underlying scheduling of the JVM, thereby preserving the portability of our resource management scheme.[2] Note, however, that although a CPU manager is activated by each thread in a deterministic manner, the bytecode instrumentation and the execution of management code may affect the scheduling

[2] Thread scheduling is left loosely specified in the Java language [19] and JVM [26], in order to facilitate the implementation of 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.

of non-deterministic programs. I.e., such programs may exhibit a different thread scheduling when they are executed with or without instrumentation.

Because of the fine-grained, deterministic activation control of management code, a CPU manager can precisely restrict the further execution of a thread. A CPU manager allows each thread to execute only a limited number of its own bytecode instructions before re-executing management code. This is essential to thwart denial-of-service attacks, which is one of our goals. Our CPU manager activation scheme allows for *pre-accounting*, since a CPU manager may permit a thread to execute a certain number of bytecode instructions only if this execution is guaranteed not to exceed a given quota.

Our primary design goal is full portability of the CPU management scheme, which shall be compatible with any JVM. As we rely on bytecode instrumentation, one important issue is to keep the overhead caused by the inserted bytecode low. While a certain overhead inevitably is the price one has to pay for platform-independent, fully portable CPU management, we have devised several optimization schemes to reduce the overhead, as discussed in Sections 4, 5, and 6.

Another design goal is to support the coexistence of code that has been transformed for CPU management with unmodified code. For instance, a middleware system may want to account only for the execution of deployed components (e.g., Servlets, Enterprise Java Beans, etc.), but not for the execution of certain management tasks within the middleware platform. Accounting only for the execution of those parts of the code where the accounting information is actually needed helps to reduce the overall accounting overhead. For this reason, and because of the possibility of dynamic class loading in Java, we currently abstain from global (interprocedural or intermodular) program analysis and transformation, to enable the user to decide for each method whether it shall be rewritten for accounting or not. In other words, we require that all transformations maintain the compatibility of rewritten code with unmodified code.

3. Principles of the CPU Accounting Scheme

The general idea of our approach is that the bytecode of software components is rewritten in order to make their resource consumption explicit. We informally define a component as a group of threads running under the supervision of the same CPU Manager. As shown in Figure 1, threads maintain the count of executed bytecodes inside

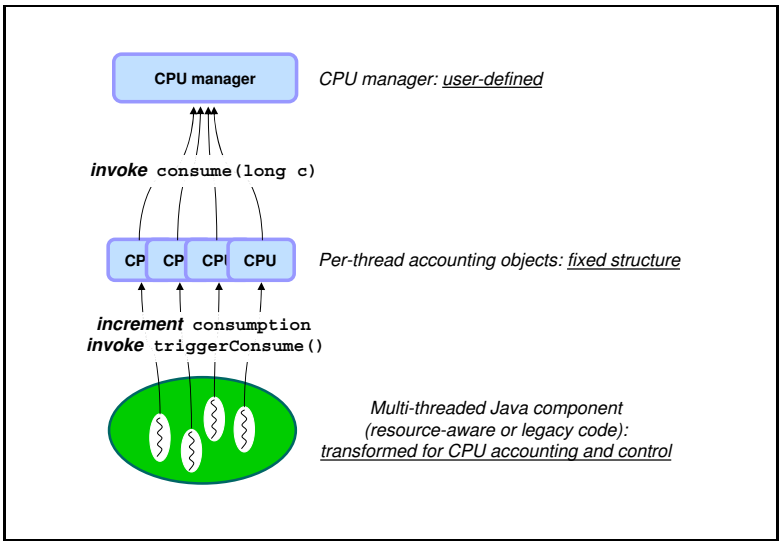


Figure 1. Runtime Organization of a CPU-Managed Component.

individual CPU accounting objects by incrementing an integer field called `consumption`.

The CPU accounting scheme of J-RAF2 does not rely on a dedicated supervisor thread. Instead, the execution of management tasks is distributed among all threads in the system. Periodically, each thread invokes `triggerConsume()`, which itself calls `consume(long c)` on the CPU Manager (if any), in order to aggregate the collected information concerning its own CPU consumption within an account that is shared by all threads of the same software component. While inside `consume(long c)`, each thread has the opportunity to execute management code, such as scheduling decisions, to ensure that a given resource quota is not exceeded. E.g., the thread may terminate the component if there is a hard limit on the total number of bytecode instructions it may execute, or it may delay itself (i.e., put itself to sleep) in order to meet a restriction placed on the execution rate.

We call our approach *self-accounting*, because each thread accounts for its own CPU consumption. Self-accounting is essential for the portability of management code, since its activation does not rely on the under-specified scheduling of the JVM. Moreover, when a thread manages itself, we avoid many deadlocking and access rights issues that arise with a dedicated supervisor thread, since the `consume(long c)` invocation is synchronous (i.e., blocking), and executed directly by the thread to which the policy applies.

3.1. Bytecode Transformation Scheme

Our two main design goals for the bytecode rewriting schemes 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). In this section we present a simple, unoptimized transformation scheme.

Each thread is permanently associated with a `ThreadCPUAccount` accounting object, of which the public interface is listed in Figure 2. First, upon entering a method of a transformed component, it is necessary to determine the `ThreadCPUAccount` belonging to the currently executing thread: this is achieved through a call to `getCurrentAccount()`. Then, as execution proceeds through the body of the method, the thread updates the `consumption` counter of its `ThreadCPUAccount`: this is the actual CPU accounting.

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

Figure 2. Part of the ThreadCPUAccount API.

To prevent overflows of the `consumption` counter, which is a simple 32-bit integer,[3] and, more fundamentally, to ensure the regular execution of the shared management tasks, the counter has to be steadily checked against an adjustable `granularity` limit. More precisely, each thread invokes the `triggerConsume()` method of its `ThreadCPUAccount`, when the local `consumption` counter exceeds the limit defined by the `granularity` variable. In the following, we refer to this periodic check as *polling*.

There are dedicated JVM bytecode instructions for the comparison with zero. Hence, in order to optimize the comparison of the `consumption` counter to the `granularity`, the counter runs from `-granularity` to zero, and when it equals or exceeds zero, the `triggerConsume()` method is called. We use the `iflt` instruction, which branches if the value on top of the operand stack is smaller than zero, in order to skip the invocation of `triggerConsume()` in the preponderant case where `consumption` is below zero.

[3] 64-bit `long` integers unfortunately still impose a prohibitive runtime overhead in Java.

Concretely, to apply this CPU accounting scheme, each non-native and non-abstract Java method (respectively constructor) is rewritten in the following way:

1. Insert a call to `getCurrentAccount()` at the beginning of the method and save the result in a local variable (let us call it `cpu`).
2. Insert as few conditionals as possible in order to implement the polling efficiently. The conditional "`if (cpu.consumption >= 0) cpu.triggerConsume();`" is inserted in the following locations:
   a) At the beginning of the method and before method termination (before a `return` or `throw` statement). This is to ensure that the conditional is regularly evaluated in presence of recursive methods and more generally of deeply nested call stacks. It is a form of call/return polling as described by Feeley [18].
   b) At the beginning of each JVM subroutine and before return from the JVM subroutine. This ensures that the conditional is regularly evaluated in the (possibly nested) execution of JVM subroutines. Again, this is a form of call/return polling.
   c) At the beginning of each exception handler. This is important for complete call/return polling, since a method may be terminated by an exception which was not thrown explicitly. E.g., a JVM `invoke` instruction may throw an exception, if the callee method throws an exception. In this case, the exception handler that catches the exception will perform the return polling.
   d) At the beginning of each loop.
   e) In each possible execution path (excluding backward jumps, since they are already taken care of as loops) 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 to *MAXPATH*. In order to avoid an overflow of the `consumption` counter, *MAXPATH* should not exceed 2^{15} (see Section 3.5 for an explanation).

   We omit superfluous call/return polling in leaf methods, i.e., in methods that do not invoke any other method (unless the invocation is inside a structure that always requires polling, such as a loop, a subroutine or an exception handler). Furthermore, within the constraints cited above, we try to limit polling to the paths

that actually require it, so that execution paths that do not contain method invocations may appear as being leaf methods on their own. Other optimizations, such as Feeley's balanced polling [18], could be applied as well.

3. In order to ensure that `triggerConsume()` is also invoked just before the thread terminates, the `run()` method of each class that implements the `Runnable` interface is rewritten according to Figure 3. Hence, it is certain that the very last amount of CPU consumed is reported correctly, after which the thread will terminate.

```
public void run(ThreadCPUAccount cpu) {
   ThreadCPUAccount cpu = ThreadCPUAccount.getCurrentAccount();
   try {
       ... // Here is the original code.
   } finally {cpu.triggerConsume();}
}
```

Figure 3. The rewritten `run()` method.

4. Finally, the instructions that update the `consumption` counter are inserted in the beginning of each *accounting block*, which we define as the longest possible sequence of bytecode instructions where only the first instruction may be the target of a branch and only the last instruction may change the control flow (e.g., a branch or jump). Thus, in contrast to the classical notion of *basic block*, method invocations do not end a block, as this yields larger accounting blocks without harming correctness.[4] In order to reduce the accounting overhead, the conditionals inserted for polling are not considered as distinct accounting blocks.

3.2. Rewriting Example

Figure 4 illustrates how a method is transformed according to the proposed, unoptimized accounting scheme. For the sake of readability, in this article we show all transformations on Java code, whereas in reality they take place at the JVM bytecode level.

In this example, a polling conditional might seem necessary as a final instruction at the end of method `f(int)`, in order to protect against the risk of executing more than *MAXPATH* instructions since the last invoked method. However, this is not required, because all execution

[4] The accuracy of this solution is actually slightly degraded when exceptions are thrown at run-time, as explained in Section 7.2.

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

Figure 4. Unoptimized rewriting of a method for CPU accounting.

paths leading through an invocation to the end of this method will necessarily pass through the beginning of the `while` loop, which itself already contains a compulsory polling conditional.

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.

Depending on the application, the concrete value for each accounting block can be computed in different ways:

1. The number of bytecode instructions in the accounting block before the rewriting takes place. In this strategy, the resulting CPU consumption reflects the number of bytecode instructions that the original, unmodified program would execute. This approach is particularly useful for benchmarking.

2. The number of bytecode instructions in the accounting block after the rewriting, including the inserted accounting instructions. I.e., the resulting CPU consumption includes the accounting overhead. In particular, this setting allows a service provider to charge a client for the overall CPU consumption of the deployed client components.

3. For each of the previous two settings, each JVM bytecode instruction may receive a different weight, as the complexity of the various

classes of JVM bytecode instructions varies significantly. This allows to calibrate the accounting for a particular JVM, which enables a better modeling of the effective CPU load on the given JVM.

3.3. Refining `getCurrentAccount()`

The functionality of the `getCurrentAccount()` method is to return the `ThreadCPUAccount` associated with the currently executing thread. To this end, `getCurrentAccount()` first obtains a reference to the thread via the standard `Thread.currentThread()` method. Then, using this reference, it could obtain the associated CPU account through a hash table.[5] This is however a costly solution, since it has to occur at each method entry of the rewritten application. A far more efficient approach taken here is to patch the real `Thread` class and add the needed reference directly to its set of instance fields.[6]

The required functionality of `getCurrentAccount()` is unfortunately hard to implement during the bootstrapping of the JVM. During this short, but crucial period, there is an initial phase where `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 to execution 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.

As this article focuses on the optimization of bytecode transformation schemes, we will not discuss further the issues related to rewriting the JDK. Interested readers can find more information on this subject in [3].

[5] Using the predefined `ThreadLocal` class of Java also amounts to searching in hash tables.

[6] We investigated also 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, however, does not conform to the Java API, which stipulates that `Thread` is a direct subclass of `Object`.

3.4. Aggregating CPU Consumption

Normally, each `ThreadCPUAccount` object refers to an implementation of `CPUManager`, which is shared between all threads belonging to a component. 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 `triggerConsume()` method of `ThreadCPUAccount`. Figure 5 shows part of the `CPUManager` interface.

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

Figure 5. 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. Here we focus only on the required `CPUManager` interface.

The intended semantics of `consume(long c)` has already been presented. The `getGranularity()` method has to return the accounting granularity currently defined by the given `CPUManager` for the calling thread. The accounting granularity defines the frequency of the management activities, and may be adapted at run-time, e.g. to make low-priority threads check often enough if they should yield the CPU in favour of threads with imminent deadlines. The range of legal values the granularity may take is however also bounded by choices made during the code transformation, as explained in the next section.

Further details and examples concerning the management of `CPUManager` objects, and the association of `ThreadCPUAccount` objects with `CPUManager` objects can be found in our previous work [4].

3.5. Triggering Delay

The delay until a thread triggers the `consume(long c)` method of its `CPUManager` is affected by the following factors:

1. The current accounting granularity for the thread. It is possible to allow the granularity to range all the way up to `Integer.MAX_VALUE`, i.e., $2^{31} - 1$, but it may be further restrained

by the nature of the application to be supervised, as shown in the following.

2. The number of bytecode instructions until the next conditional C is executed that checks whether the `consumption` variable has reached or exceeded zero. This value is bounded by the number of bytecode instructions on the longest execution path between two such conditionals C. The worst case is a method M that has a series of $MAXPATH$ 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 `triggerConsume()` method of `ThreadCPUAccount` (which in turn invokes 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` (a negative value) and it will not exceed 2^{30} (i.e. $MAXPATH^2$), well below `Integer.MAX_VALUE`. Using current hardware and a state-of-the-art JVM, the execution of 2^{32} bytecode instructions may take only a fraction of a second,[7] of course depending on the complexity of the executed instructions.

For a component with n concurrent threads, in total at most $n * MAXDELAY$ bytecode instructions are executed before a thread triggers the `consume(long c)` function. If the number of threads in a component can be high, the accounting granularity may be reduced in order to achieve a finer-grained management. However, as this delay is not only influenced by the accounting granularity, it may be necessary to use a smaller value for $MAXPATH$ during the rewriting.

An interesting measurement we made was to determine the impact of the choice of a granularity. We used the 'compress' program of the SPEC JVM98 benchmark suite to this end. As shown in Figure 6, the lower the granularity, the higher the overhead will be, and the more frequently the management actions will take place. In our current implementation, and on the given computer, this interval is not

[7] On our test machine (see Section 4.1) this is made possible on some favorable code segments by the combination of an efficient just-in-time Java compiler and a CPU architecture taking advantage of implicit instruction-level parallelism.

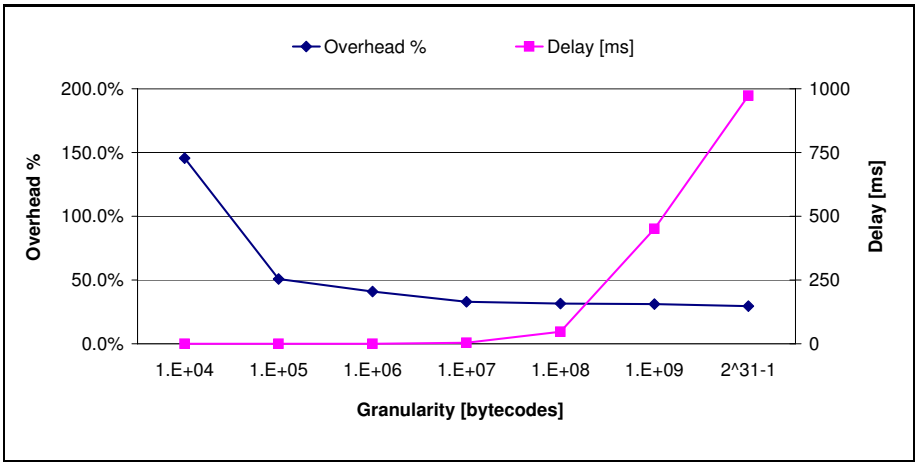


Figure 6. Granularity versus overhead and delay.

measurable[8] with granularities below 10 000 000 bytecode instructions. Another lesson learned is that granularities of 100 000 and more exhibit rather similar levels of overhead, ranging approximately from 30% to 50%, whereas a granularity of only 10 000 results in an overhead as high as 150%.

4. Introduction to Rewriting Optimizations

In this section we present an analysis of the origin of the overheads that our rewriting scheme entails. The next two sections each present a set of targeted optimizations designed to systematically reduce these overheads.

4.1. Evaluation Methodology

The benchmarking was performed with SPEC JVM98 [31], a well-known general-purpose benchmark suite, which consists of the following Java programs:

- *compress*: a popular utility used to compress/uncompress files;
- *jess*: a Java expert system shell;
- *db*: a small data management program;
- *javac*: an old Java compiler, compiling 225,000 lines of code;

[8] Intervals below the resolution of the `System.currentTimeMillis()` function, i.e., one millisecond, are not measurable.

- *mpegaudio*: an MP3 audio stream decoder;
- *mtrt*: a dual-threaded program that ray traces an image file;
- *jack*: a parser generator with lexical analysis;

We ran SPEC JVM98 on a Linux Fedora Core 2 computer (Intel Pentium 4, 2.6 GHz, 512 MB RAM). For all experiments, the entire JVM98 benchmark suite was run 40 times. Then, for each program, the median execution time was used to compute the slowdown ratio (i.e., the overhead) compared to the original, unmodified version of the program. Finally, a global overhead indicator was obtained by calculating the geometric mean of these ratios. All executions are automatically verified for semantic correctness by the SPEC JVM98 benchmark suite itself, by comparing actual output with expected data. It should be noted that with this setup our execution time measurements have a remaining imprecision of 1%,[9] although the test machine is each time restarted and booted into a mode with only the strictly necessary system services active.

Measurements were made on the IBM JDK 1.4.2 platform in its default execution mode only, as well as the Sun JDK 1.5.0 platform in its following three modes:

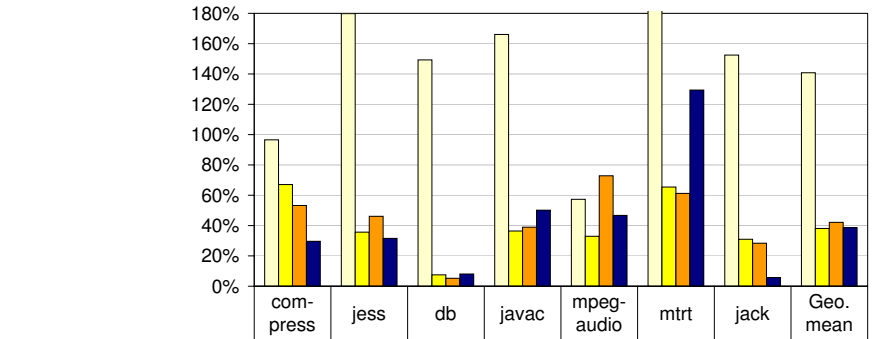
1. Purely interpreted mode (`-Xint` command-line option): this completely deactivates the just-in-time (JIT) compiler, and gives us an idea of how the relative overheads and consecutive optimizations might manifest themselves in a primitive Java environment, such as on certain embedded systems with limited resources.

2. Client mode (`-client` command-line option): this starts the Sun Java HotSpot [28] Client VM, which is tuned for reducing start-up time and memory footprint, and has a JIT that incrementally compiles bytecodes into native instructions.

3. Server mode (`-server` command-line option): this launches the Sun Java HotSpot Server VM, which is designed for maximum program execution speed and therefore contains a JIT that optimizes code more systematically at load-time.

All other JVM-specific options take their default values. In our test we used a single `CPUManager` with the most basic accounting policy, i.e., one which simply aggregates announced consumptions of application threads, and with the highest possible granularity. This means that the `triggerConsume()` method is invoked only once in about 2^{31} bytecode

[9] This was experimentally determined as the maximal difference between the final results of any two complete benchmark suite executions.



	com-press	jess	db	javac	mpeg-audio	mtrt	jack	Geo. mean
Sun JDK 1.5.0, interpreted	96.6%	179.8%	149.3%	166.1%	57.3%	224.0%	152.5%	140.8%
Sun JDK 1.5.0, client	67.0%	35.7%	7.5%	36.4%	33.0%	65.4%	31.0%	38.1%
Sun JDK 1.5.0, server	53.3%	46.1%	5.3%	38.9%	72.8%	61.2%	28.4%	42.1%
IBM 1.4.2, default	29.6%	31.6%	8.1%	50.1%	46.7%	129.4%	5.7%	38.6%

Figure 7. Overhead of CPU accounting without optimizations.

instructions, which makes its influence on the overheads negligible in practice.

For all tests, unless noted otherwise, both the JDK and the JVM98 benchmarks were rewritten according to the tested strategy. One difficulty is that many JVMs do not allow certain core JDK classes to be transformed: they will either crash, or simply use internal, hardwired implementations and disregard the corresponding bytecode-level representations. With the `java.lang.Object` class, we were therefore able to rewrite only a subset of the methods. The `java.lang.Thread` class, on the other hand, was patched, as explained in Section 3.3. On the virtual machines considered here, no other classes required any special treatment.

The measurement to which all optimizations will be compared is the result of the unoptimized transformation scheme introduced in Section 3, which we call the *Simple* rewriting scheme. The corresponding execution time overhead introduced by this scheme as compared to the original, non-rewritten benchmark, is shown in Figure 7. The rewriting increased the size of the corresponding class files by 17% (we refer to Section 7.3 for a discussion on code size).

4.2. Origin of Overhead

We next analyze the origin of the overhead of our bytecode transformations, in order to be able to devise targeted optimization schemes.

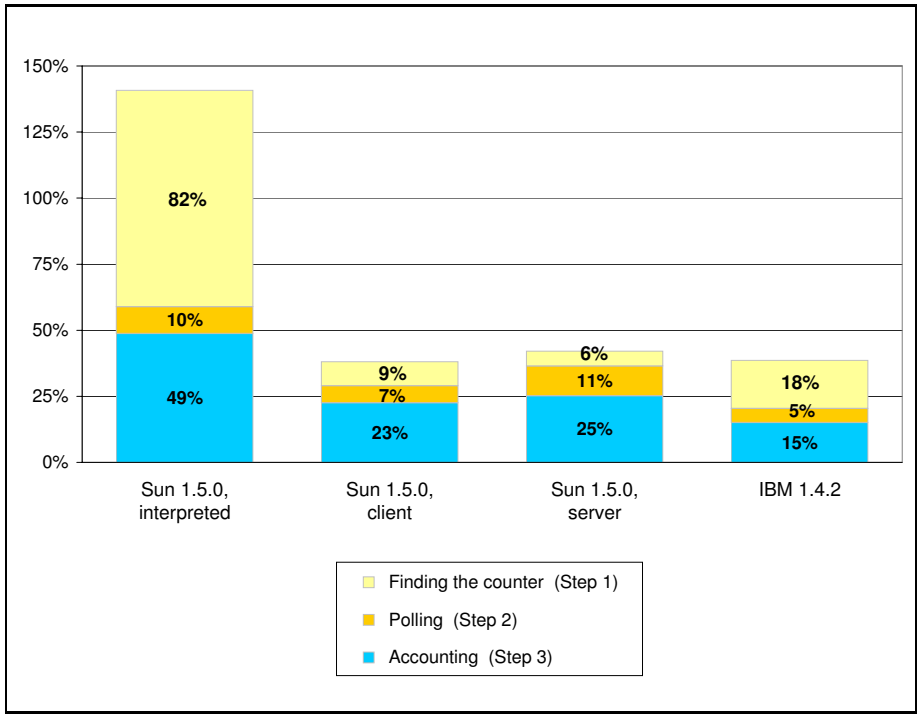


Figure 8. Evaluation of the distribution of overheads (with *Simple* rewriting, no optimizations).

Although our accounting scheme only works when implemented in its entirety, let us decompose it in order to find out which parts of the code cause execution time overheads. To this end, we measured three different versions of the *Simple* rewriting scheme presented in Section 3, which each represents an incremental step in instrumenting the bytecode. The three steps are:

1. Finding the proper instruction counter (with an invocation to `getCurrentAccount()`)
2. Polling the value of the instruction counter (with the conditional "`if (cpu.consumption >= 0) cpu.triggerConsume();`")
3. Accounting, i.e. updating the instruction counter (with "`cpu.consumption += ...;`")

Figure 8 shows the measured average distribution of overheads among the different steps in a rewritten method. The stacked columns are made of the contributed overheads, summing up to the respective

per-JVM totals of 140.8%, 38.1%, 42.1% and 38.6% (the geometric means shown in Figure 7). We can see from this figure that on average the most expensive part is the accounting itself (*Step 3*), followed by the obtaining of the `consumption` counter (*Step 1*), and finally the polling (*Step 2*). For the purely interpreted platform, *Step 1* is clearly the most expensive. This comes from the fact that here we insert an unconditional method invocation (for obtaining the reference to the right `ThreadCPUAccount`), and invocations are relatively costly on that platform. The second most expensive step is the accounting (*Step 3*), while polling (*Step 2*) is a relatively light operation.

It must be noted that these numbers can only be considered as an approximation. Indeed, the storing of the `ThreadCPUAccount` reference into a local variable at the beginning of each method may be discarded by the JIT, and never executed (hence never measured), unless this store is followed by code for reading the variable, e.g., for polling and accounting. This means that the overhead calculated for *Step 1* may be slightly underestimated, while the overhead for *Step 2* would be slightly overestimated, except on the JVM without a JIT (the Sun JVM in purely interpreted mode).

Based on the numbers of Figure 8, we introduce differentiated optimizations in the next two sections.

5. Reducing the Overhead of Finding the Proper Instruction Counter

The per-thread instruction counter is encapsulated inside a `ThreadCPU-Account` object, itself to be found via a reference to the current Thread. We have explained in Section 3.3 how the `getCurrentAccount()` method is already optimized for obtaining this information by patching the `Thread` class. As part of our standard optimization settings, we also decided to directly inline the contents of `getCurrentAccount()` instead of generating an invocation to it.

In order to avoid these repetitive executions of `getCurrent-Account()`, we instead pass the `ThreadCPUAccount` as additional argument from method to method, by changing all method signatures during the transformation process. We describe this approach in the following section, and thereafter we present an enhanced version of it.

5.1. Wrapper Rewriting

Figure 9 illustrates how the method shown in Figure 4 is transformed using a CPU accounting scheme that passes the `ThreadCPUAccount` as extra argument.

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

Figure 9. Method rewritten for CPU accounting. The `ThreadCPUAccount` is passed as extra argument.

This approach works when all client classes of a transformed class can also be transformed, such that all method invocation sites are updated to use the corresponding extended method signatures. Sometimes, however, this is not possible, such as when a Java method is invoked by native code or via the reflection API. In order to cope with such cases, we have to leave in all classes stubs with the original method signatures, which act as *wrappers* for the methods with the additional `ThreadCPUAccount` argument, as shown in Figure 10. Therefore we call this general argument passing scheme the *Wrapper* rewriting scheme.

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

Figure 10. Example wrapper method with unmodified signature.

Another issue concerns the *Wrapper* rewriting of the JDK. In some cases, 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, which introduces extra stack frames into the stack at runtime. For this reason, the JDK cannot take advantage of the more efficient wrapper rewriting scheme, and has to invoke `jdkGetCurrent-`

Table I. Overheads and optimizations for *Step 1*.

Rewriting mode: JVM98-JDK	Sun 1.5.0 interpreted	Sun 1.5.0 client	Sun 1.5.0 server	IBM 1.4.2
Simple–Simple	81.9%	9.1%	5.9%	18.2%
Wrapper–Simple	35.9%	4.2%	5.7%	5.3%
Wrapper–Wrapper	8%	4.3%	7.0%	2.5%
+Fixed arg passing	8.8%	4.6%	7.2%	2.4%

`Account()` in 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 signature, and a second one with the additional `ThreadCPUAccount` argument. This approach works well, but is quite complex, and unfortunately results in an appreciable speedup only with certain JVMs. Further details on this are to be found in [3].

Looking at Table I, we can see that considerable progress has been made, starting from our reference, at the first line, i.e., the overhead when rewriting both SPEC JVM98 and JDK in *Simple* mode. The optimizations designed for *Step 1* are particularly beneficial on the Sun JVM in interpreted mode, which does not have the ability to dynamically inline the ubiquitous invocation to `jdkGetCurrentAccount()`, nor the invocations that the inlined body of `getCurrentAccount()` itself contains. As expected, *Wrapper* mode rewriting brings the best speedup to applications that have the highest ratios of method invocations, like 'mtrt' and 'compress' [17]. In fact, rewriting the application layer (here: SPEC JVM98) in *Wrapper* mode is always a benefit, but rewriting also the JDK in *Wrapper* mode is only interesting on half of the platforms (Sun JVM in interpreted mode and IBM). The last line of the table, entitled *+Fixed arg passing*, will be described in the next section.

5.2. Fixed Argument Passing

In this section we further improve the way the `ThreadCPUAccount` reference is passed as an extra argument upon method invocation. Our hypothesis is that if this extra argument is passed always at the same, fixed position in the argument list, the JIT (if any) may skip some machine instructions that would be required to shift this argument from one position to another between two invocations.

To achieve this effect, the bytecode transformation tool will sometimes have to insert dummy arguments at the lower positions, when

the original method has a shorter argument list. We found by systematic testing that, for the IBM Java platform, the best position for the `ThreadCPUAccount` is the 2^{nd}, both for virtual methods (the `this` object reference being number zero), and for static methods. Figure 11 illustrates the insertion of dummy arguments for this configuration, with methods 1–3 being virtual, and methods 5–7 static. Note that for the first two methods in Figure 11, `ThreadCPUAccount` is successfully set as second argument. In contrast, the third example method already takes three arguments (including the implicit `this` reference), therefore `ThreadCPUAccount` has to come at the third position.[10]

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

Figure 11. Examples of the insertion of 'dummy' arguments.

According to Table I, this scheme only seems to be profitable to the IBM platform, and then only marginally so. However, with all accounting code incorporated, this transformation reduces the total overhead by 2–3% on that JVM.

6. Reducing the Overhead of Accounting

As we have seen in Figure 7, the accounting itself (i.e. keeping the value of the CPU account up-to-date), is a fairly expensive task, responsible for about 50% of the overhead. This is due to the frequent accesses to the CPU account, which take place at each conditional branch in code transformed with the *Simple* scheme. Therefore, we introduce a new path prediction scheme, the role of which is to reduce the number of updates made to the CPU account, by trying to predict *statically* – i.e. during the bytecode transformation – the outcome of the conditional branches that will be taken at runtime. The resulting effect is that chains of accounting blocks that are predicted to be executed in sequence will be merged into a single block requiring only one update of the CPU account. As a consequence, overheads will be reduced if

[10] Whereas it is relatively trivial to append new items to the end of an argument list, shifting the position of existing arguments is more difficult to implement efficiently, since it requires the transformation tool to analyze the operand stack upstream of each invocation site.

Table II. Overheads of unoptimized vs. aggressive approximation (incl. unoptimized *Steps 1–2*).

Accounting optimization mode	Sun 1.5.0 interpreted	Sun 1.5.0 client	Sun 1.5.0 server	IBM 1.4.2
No optimization	140.8%	38.1%	42.1%	38.6%
With aggressive approx.	110.6%	30.4%	31.9%	34.3%

the predictions succeed, and the size of transformed class files will be substantially lower (see Section 7.3).

Since the static prediction of execution paths (*SPP*) is not an easy task, the probability of missing a prediction will remain non-negligible. Our goal is therefore to achieve a solution where even mispredicted branches will not, at least in theory, incur a higher overhead than with our initial, *Simple* rewriting scheme.[11] This solution will be described in Section 6.2.

But first, we want to present another approach, called *Aggressive approximation*, to reducing the overhead of *Step 3*, which will experimentally give us a very low overhead for the accounting task, and hence provide valuable information on what maximal performance can be expected from *SPP* schemes.

6.1. Aggressive Approximation

Depending on the application, it may not always be necessary to do a very precise accounting of CPU consumption. There are e.g. many occurrences in Java code where the number of executed bytecodes only differs by one or two units between the two outcomes of a given conditional. Therefore, it might be acceptable to lose some precision in exchange for a lower run-time overhead. An extreme case is one where we only update the consumption once per method, or per loop body, i.e. essentially only in conjunction with polling sites in the code: this is what we call the *aggressive approximation*. The actual value to account may be chosen statically as the average length of all possible executions, or as the length of the longest, or most probable path.

Having implemented the latter solution, the resulting overhead is as indicated by Table II, compared to the completely unoptimized solution.

[11] As shown later, there is some additional jumping required to implement the scheme, thus in practice the benefit may not be complete.

The *SPP* scheme should then have an overhead between those two settings, preferably closer to the lowest values.

6.2. Static Path Prediction

The goal of the scheme we call *Static Path Prediction* (*SPP*) is to identify sequences of accounting blocks to be conceptually merged into execution paths that can be accounted just once. Recall that paths diverge at each accounting block that ends with a conditional branch (i.e. essentially `if` instructions at the Java source code level). The concrete idea of *SPP* is to initially update the consumption with the entire cost of the predicted path. Hence, whenever a branch does not take the direction that was predicted during the rewriting (thus we have a *miss* of the prediction), the predicted path is left, and a corresponding correction must be made to the CPU account, subtracting the unused part of the exited path, and adding the cost of the new path that is then started.

6.2.1. *SPP Rewriting Example*

```
void f(int x) {            void f(int x) {
                              ThreadCPUAccount cpu;
                              cpu = ThreadCPUAccount.getCurrentAccount();
                              cpu.consumption += ...;
                              if (cpu.consumption >= 0) cpu.triggerConsume();
   g();                       g();
   while (x > 0) {           Start:
                              cpu.consumption += ...;
                              if (cpu.consumption >= 0) cpu.triggerConsume();
                              if (x > 0) {
      if (h(x)) {     -->        if (h(x)) {
         i(x);                      i(x);
      }                          } else {
                                    cpu.consumption -= ...; // misprediction
                                 }
      --x;                       --x;
   }                             goto Start;
                              }
                              cpu.consumption -= ...;  // misprediction
}                          }
```

Figure 12. Example of *Simple*, *SPP*-optimized transformation scheme.

Figure 12 shows an example of the proposed transformation scheme, as an improvement on top of the *Simple* rewriting. Compared to the transformation of Figure 4, which illustrates the unoptimized *Simple* scheme, we can see that there are two fewer modifications of the `consumption` variable inside the loop, and that at some places, the

variable is decremented, reflecting the fact that a mispredicted branch was taken. The presence of such possibly negative corrections imply that the *MAXPATH* value now also has to act as a safety barrier with respect to subtractions occurring right after a reset of the `consumption` variable by the `triggerConsume()` method. If *MAXPATH* is too small, the `consumption` variable might then overflow on the negative side, to become strongly positive, which is of course wrong.

This example also shows that entirely new accounting blocks (implemented by the additional `else` block) sometimes have to be added to the control-flow graph in order to implement corrections due to mispredictions. Another case that requires the insertion of dedicated accounting blocks is the following: if a block constitutes the join point of two execution paths that both are considered as mispredictions, then it will not be able to do the accounting correction for the two of them. To solve this, we create a second accounting block so that each path may implement its own correction.

6.2.2. *Chosen Heuristics*

It is a challenge to define adequate heuristics for identifying successful paths, only on the basis of the bytecode that is being processed. Such heuristics should be relatively inexpensive to evaluate, and, for simplicity, use only information available from the control flow graph. This is, first, because we know from the *aggressive approximation* experiment that the margin for improvement is relatively narrow (see Table II). Another reason is that there is a trend towards load-time, or even runtime instrumentation practices, which in turn urge for the development of computationally efficient transformation tools.

The heuristics are expressed relative to the control flow graph, where each node is a primitive accounting block. The heuristics we explored try to identify the following kinds of branches (loosely following the classification of [2]):

1. Loop entry branches are branches that decide whether execution will continue into a subsequent loop or avoid it. These are expected to lead into the loop.

2. Loop branches are branches inside a loop that determine whether execution will continue inside or exit the loop.

   a) exit branches are predicted to fail;

   b) backward branches are predicted to succeed;

3. Non-loop branches, i.e. branches that are not inside loops, or that are inside a loop, but do not exit or branch backwards:

a) Opcode-based equality heuristics: a number of simple tests are easily recognizable at the bytecode level (e.g. `ifnull`/`ifnonnull` and `ifeq`/`ifne`), and are expected to allow predicting the outcome:
   i) Null test for reference types: testing if a reference is equal to `null` is deemed to usually fail;
   ii) Zero test and equality test: testing if a value of a primitive type (`boolean`, `integer`, `double`, etc) is equal to zero is deemed to usually fail, as well as the equality test between two primitive types or two references.[12]

b) Successors-based heuristics: depending on properties of the closest successor nodes, we want to determine if the branch will be taken or not.
   i) `if-then` without an `else` structures: the `then` part is assumed to perform some special-case processing, and is therefore predicted not to be taken.
   ii) direct successors containing a `throw` statement: they are of course expected not to be taken, since they raise an exception.
   iii) direct successors containing a `return` statement: they are also intuitively deemed to be treating special cases, such as to terminate a recursive iteration.

We implemented these heuristics and inserted additional probes into the transformed classes, in order to collect statistics on the quality of the predictions throughout the whole SPEC JVM98 benchmark suite.

We decided that the heuristics are to be evaluated in turn, starting with those for which our statistics show the highest probability of success (or *hit*), respectively the lowest *miss* rate, as detailed in the following. We chose to consider them as exclusive, i.e. as soon as a heuristic decides that a given successor node of the current branching node shall be taken, the algorithm stops there.

We found that the best sequence of heuristics, which yielded an overall miss rate of 28%, was the following:

1. Loop branches and loop entry branches are well-known to provide a hit with good confidence [2]. In our setting, they had a total miss rate of 23%, which is not quite as good as expected.

[12] This aggregation of different cases inside the same heuristic is due to the nature of the JVM instruction set, which has several uses for the `ifeq`/`ifne` opcodes. Therefore, a finer analysis would be needed to distinguish between the different cases.

2. Successors-based heuristics, by order of efficiency:
   a) Avoid direct successors that raise an exception (3% miss rate).
   b) Avoid direct successors that end with a `return` statement (27% miss rate).
   c) In the presence of an `if-then` without an `else` structure, take the *then* branch (31% miss rate); this contradicts the hypothesis we made at point 3.b.i, but conforms to the findings of Ball and Larus [2] in the case of the C language. This heuristic nevertheless suffers from a high standard deviation, with e.g. a miss rate as high as 96% with the 'db' program of JVM98, and as low as 9% with 'jess'.
3. Our default strategy is to choose the successor that follows the current branching node sequentially instead of the one targeted by the branch (41% miss rate). Part of the explanation for this may stem from the fact that each Java `if` condition is usually compiled into a bytecode that tests for the negation of the condition (in order to branch to the `else` part, if any, otherwise to the `endif`); therefore it could mean that the Java programmer tends to write his `if` tests in such a way that the most common treatment is located inside the `then` block, instead of the `else` block, if any.

We found that all other heuristics were unreliable (i.e. close to 50% of miss rate and/or a high standard deviation), such as the opcode-based tests for equality or nullness of operands. It is likely that such heuristics must be enhanced with data-flow analysis to become effective. One could imagine further heuristics to experiment with, but at the same time, it is good to have only a few that do not require too much processing on each branching node.

As a caution, one should generalize from these results only if the SPEC JVM98 suite actually represents "standard Java coding practices".

6.2.3. *Benchmarking the* SPP *Scheme*

In addition to measuring the overhead due to our *SPP* scheme with the given heuristics, we also benchmarked a version of *SPP* where all heuristics were replaced by a completely random choice, using the random number generation of the Java class library. We can see that the random version performs almost as well as *SPP with heuristics*, and that the *aggressive approximation* performs clearly better, which is somewhat disappointing. This is because the random version may take good paths by chance, short-circuiting accounting operations that would otherwise exist with the *Simple* scheme. At places the gain of

Table III. Overheads of various *SPP* settings (incl. unoptimized *Steps 1–2*).

Accounting optimization mode	Sun 1.5.0 interpreted	Sun 1.5.0 client	Sun 1.5.0 server	IBM 1.4.2
No optimization	140.8%	38.1%	42.1%	38.6%
Random SPP	124.7%	36.4%	40.5%	37.5%
SPP with heuristics	119.8%	35.8%	40.5%	37.2%
Aggressive approx.	110.6%	30.4%	31.9%	34.3%

SPP (both random and with heuristics) is minimal, and the main explanation is probably that the resulting frequency of accesses to the `ThreadCPUAccount` remains fairly high, due to the overall misprediction rate of 28%. Comparatively, the *aggressive approximation* modifies the `ThreadCPUAccount` far less frequently, but at the cost of a significant loss of precision. Table III summarizes these results, along with the numbers that were already given in Table II. Section 7.4 outlines some opportunities for further improvements of our optimizations, including the *SPP* scheme.

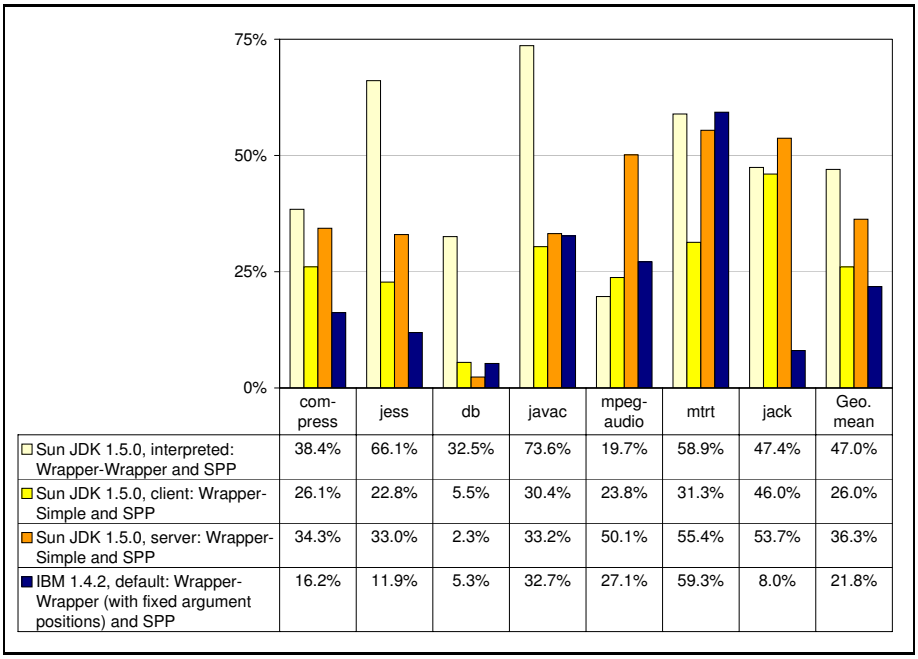
7. Evaluation and Discussion

We present in this section the benefits, limitations, and possible extensions of this work, starting with an overall quantitative assessment of the proposed optimizations.

7.1. Optimal Optimization Combinations

At this point, we know from all preceding experiments which partial optimizations are best, and can aggregate them in order to present the per-platform optimal settings:

- For Sun JDK 1.5.0, interpreted mode: *SPP* on top of *Wrapper* rewriting for JVM98 and *Wrapper* rewriting (with code duplication) for JDK.
- For Sun JDK 1.5.0, client mode: *SPP* on top of *Wrapper* rewriting for JVM 98, *Simple* rewriting for JDK.
- For Sun JDK 1.5.0, server mode: *SPP* on top of *Wrapper* rewriting for JVM 98, *Simple* rewriting for JDK.



	com-press	jess	db	javac	mpeg-audio	mtrt	jack	Geo. mean
Sun JDK 1.5.0, interpreted: Wrapper-Wrapper and SPP	38.4%	66.1%	32.5%	73.6%	19.7%	58.9%	47.4%	47.0%
Sun JDK 1.5.0, client: Wrapper-Simple and SPP	26.1%	22.8%	5.5%	30.4%	23.8%	31.3%	46.0%	26.0%
Sun JDK 1.5.0, server: Wrapper-Simple and SPP	34.3%	33.0%	2.3%	33.2%	50.1%	55.4%	53.7%	36.3%
IBM 1.4.2, default: Wrapper-Wrapper (with fixed argument positions) and SPP	16.2%	11.9%	5.3%	32.7%	27.1%	59.3%	8.0%	21.8%

Figure 13. Per-platform lowest overheads with CPU accounting.

– For IBM JDK 1.4.2, default mode: *SPP* on top of *Wrapper* rewriting for JVM 98, *Wrapper* rewriting (with code duplication) for JDK, and fixed `ThreadCPUAccount` parameter position 2 both for virtual and static methods.

The overheads corresponding to these optimal settings are detailed in Figure 13.

7.2. Accuracy of Accounting

The aggressive approximation was designed primarily for determining a lower bound on the accounting overhead, i.e. a limit we should try to reach in our search for new optimization schemes targeted at *Step 3*. The aggressive approximation incurs an obvious loss of accounting precision, that remains to be evaluated. On the other hand, all the other optimizations presented here are designed to be as accurate as possible, with the compromise that exceptions thrown at runtime may distort the result, as described in the following.

In our approach, we always update the consumption counter with the statically computed number of bytecodes at the beginning of each accounting block. Exceptions at runtime may however occur at arbitrary instructions within the accounting block, causing the remaining

Table IV. Relative inaccuracy of the computed number of executed bytecodes.

compress	**jess**	**db**	**javac**	**mpegaudio**	**mtrt**	**jack**
$4{,}9 \cdot 10^{-8}$	$7{,}6 \cdot 10^{-8}$	$1{,}5 \cdot 10^{-7}$	$1{,}0 \cdot 10^{-4}$	$2{,}7 \cdot 10^{-8}$	$6{,}2 \cdot 10^{-6}$	$6{,}4 \cdot 10^{-3}$

instructions in the accounting block not to be executed, even though they have been included in the consumption counter. Consequently, the computed consumption may indicate more bytecodes than really executed. The degree of this inaccuracy was determined by comparison with the exact number of executed bytecodes, which was obtained by modifying our accounting block analysis so that any bytecode that may throw an exception ends a block. While this allows fully precise bytecode counting, the overhead is high (code size and execution time), because many more locations are instrumented.

As illustrated in Table IV, the relative inaccuracy is below 0,1% for all benchmarks except 'jack', which is known to be a particularly exception-intensive program [10, 27]. We conclude that our chosen accounting block analysis reconciles accurate bytecode counting with reduced instrumentation overhead.

7.3. Impact on Code Size

One of our goals in designing the optimizations is to limit the number of bytecode instructions added during the transformations. This is to avoid negative effects on locality, which may reduce overall performance, and also to help making J-RAF2 a reasonable option for embedded devices, with less memory than desktop and server systems. The *Simple* rewriting scheme expands the code by 17%, whereas the *SPP* transformation increases the size by 14% (averages calculated on the JDK class files of both Sun 1.5.0 and IBM 1.4.2).

The increased size may result in code validity problems if, before transformation, a method already is close to the limits defined by the JVM specification [26] (the maximum number of bytecodes in a method is 64K) and its instruction set (standard unconditional jumps and conditional branches may only express positive and negative offsets of 32K bytecodes). It is the task of the lower layer of our tool (currently provided by BCEL [15]) to ensure the transparent conversion of jumps and branches into their *wide*, 64K bytecodes-enabled counterparts when

the required offsets are too large.[13] Whereas the *Simple* transformation scheme only slightly expands existing offsets, the more advanced schemes, like the *SPP* transformation, may from time to time generate new jumps over longer distances inside the method body, in order to execute accounting correction code. An enhanced version of *SPP* should try harder than now to minimize jumping.

Other limits that might - in extreme situations - be reached after the transformation are e.g. the number of local variables in a method, or the number of methods in a class (in the case of *Wrapper* rewriting with code duplication). If any of these limits is exceeded, BCEL will throw an exception and our tool will in turn reject the guilty method.

7.4. Opportunities for Further Optimizations

First, it should be noted that in our performance measurements the number of iterations of SPEC JVM98 was changed from the default setting of 2–4 up to 40. This has the effect of enabling each JVM to fully utilize its internal optimization capabilities, and hence it increases the reproducibility of the experiments. As a negative consequence, some JVMs now exhibit perceptibly higher overheads than were found in our previous experiments [21] (this concerns essentially the IBM JVM 1.4.2). We may therefore conclude that the overheads presented here are especially relevant to server environments, whereas short-running applications will often show a better relative performance.

The optimizations that we have presented here reveal the limits of our design goal of resorting only to intraprocedural optimizations. We see several research directions for further reducing the overhead:

- Improving the *SPP* prediction capability on a per-application basis with feedback from off-line profiling, as described in [25]: this would make the *SPP* scheme semi-static instead of purely static, but the predictions and resulting transformations for each given application would still remain portable. The interest of this technique is supported by the measurements we made, which show that the current *SPP* implementation strongly benefits 'mpegaudio', which seems to have fairly predictable branches, whereas it actually increases the overhead of 'jack', which has a rather unusual control flow, notably a frequent use of exceptions [10, 27].

[13] Unfortunately, the current version of BCEL (v 5.1) does not transparently transform conditional branches since they do not have *wide* equivalents, and would require instead a transformation into a combination of instructions with one short and one long offset.

- Whereas polling was a relatively minor source of overhead with our unoptimized transformations, it becomes the second most important source of overhead after application of all optimizations. Loops remain a source of overhead because they increase the frequency of polling. We are currently working on a loop unrolling technique where a dedicated, local counter is used to create another level of (less frequent) polling and accounting inside loops; this idea is somewhat related to the "virtual" check mentioned by Feeley [18]. Whereas this optimization seems promising for loop-intensive, scientific applications, it is not yet certain that there will be a real benefit for more varied code, like SPEC JVM98.

- Method inlining would contribute significantly to the reduction of all overheads: by eliminating invocations, many occurrences of *Step 1* and *Step 2* would disappear, and by merging method bodies, it would create further opportunities for *SPP* to identify long, likely execution paths. We have made a preliminary test with the Soot Framework[14] [32], to apply method inlining on SPEC JVM98 (but excluding the JDK) before rewriting with J-RAF2 (using the per-platform best settings, as described above): compared to the geometric means in Figure 13, the resulting overhead was unchanged for the IBM JVM, whereas we obtained a drastic 10–15% decrease in overhead for the Sun JVM in all execution modes. Since the effect of method inlining with Soot on JVM98 (*before* J-RAF2 rewriting) is noticeably lower, it confirms that inlining has a very welcome lever effect on our own optimizations. As a disadvantage, however, whole-program optimizations are not necessarily compatible with dynamic class-loading; dynamic instrumentation capabilities would be required to address this issue.

- If the aggregation of all other optimizations does not yield sufficiently low overheads, it may be advisable to relax the accounting accuracy. Beyond the extreme example of *aggressive approximation*, the exact relationship between the level of approximation and the resulting overhead remains to be studied.

7.5. Benefits

The CPU accounting scheme based on our bytecode transformation techniques offers the following benefits, which make it an ideal candidate for enhancing Java server environments and mobile object systems with resource management features:

[14] http://www.sable.mcgill.ca/soot/

- 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 [8].

- 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, the number of executed bytecode instructions is independent of system properties of the server environment.

- Flexible accounting/controlling strategies. J-RAF2 allows custom implementations of the `CPUManager` interface.

- Fine-grained control of scheduling granularity. The accounting delay can be adjusted within the constraints specified in Section 3.5: the `CPUManager` is allowed to dynamically adapt the `granularity` of each thread between the bounds defined by `Integer.MAX_VALUE` and the $MAXPATH$ value chosen at rewrite time.

- Independence of JVM thread scheduling. The present CPU accounting scheme of J-RAF2 does not make any assumptions concerning the scheduling of threads. E.g., in contrast to other approaches to CPU management [7, 14], J-RAF2 does not rely on a dedicated supervisor thread running at the highest priority level.

- Moderate overhead. We have shown that our CPU accounting approach results in moderate overhead. Whereas we have focussed on designing optimization algorithms that are specific to this bytecode rewriting scheme, it is also possible to apply prior general-purpose transformations to further increase the performance, such as loop unrolling, inlining or parallelization techniques, using a bytecode–to–bytecode optimization framework like Soot. We have also tried to target optimizations that promise a broad usability, i.e. that bring benefits on a large number of JVMs, and that do not result in excessive increase of code size.

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 expensive native operations, such as (de-)serialization and class loading, with libraries that deduce the approximate CPU consumption from the size and value of the arguments.

It should be noted that bytecode instruction counting and CPU time are distinct metrics for different purposes. While CPU accounting based on bytecode instruction counting has many advantages as discussed in Section 2, more research is needed in order to assess to which extent and under what conditions it can be used as an accurate prediction of real CPU time for a concrete system. For this purpose, individual (sequences of) bytecode instructions may receive different weights according to their complexity. This weighting would be specific to a particular execution environment and may be generated by a calibration mechanism. Therefore, such an approach would sacrifice the platform-independence of the accounting, but would still be applicable with on-the-fly rewriting, as promoted starting with JDK 1.5.0 (see discussion in next section on further deployment possibilities).

Security of our transformations (e.g. with respect to combination with other transformations or protection from tampering with CPU-consumption accounts, whether directly or indirectly by reflection) has not been addressed in this article. Nevertheless, it is fairly straightforward to implement load-time bytecode verification algorithms to prevent applications from tampering with their own CPU consumption accounts. Moreover, the reflection methods of `java.lang.Class` (e.g., `getFields()`, `getMethods()`, etc.), can be patched in order to prevent access to the internals of our CPU accounting mechanism by untrusted code.

Another issue related to reflection is that our optimized program transformation scheme, which passes a `ThreadCPUAccount` instance as extra argument, may break existing code that relies on the reflection API. After introduction of wrapper methods, the arrays of reflection objects returned by `getConstructors()`, `getDeclaredConstructors()`, `getMethods()`, and `getDeclaredMethods()` of `java.lang.Class` (i.e., instances of `java.lang.reflect.Constructor` respectively `java.lang.reflect.Method`) will contain both the wrapper methods (with the unmodified signatures) as well as methods with extended signatures. If an application selects a method from this array considering only the method name (but not the signature), it may try to invoke a method with extended signature, but fail to provide the extra argument, resulting in an `IllegalArgumentException`. We solve this

issue by patching the aforementioned methods of `java.lang.Class` to filter out the reflection objects that represent methods with extended signatures. This modification is straightforward, because in standard JDKs these methods are implemented in Java (and not in native code). Note that this issue is only relevant if an optimized program transformation scheme is used; for the *Simple* rewriting scheme, such problems with reflection cannot happen.

7.7. Further Extensions

Whereas the primary motivation for this work was to monitor and control the CPU consumption of deployed software, we are exploring several other applications and extensions of our bytecode rewriting techniques:

1. *Profiling* at the bytecode level may employ transformation techniques closely related to the ones described here. Profiling allows a detailed analysis of the resource consumption of programs under development. It helps to detect hot spots and performance bottlenecks, guiding the developer to the parts of a program that should be improved. We have shown in [6] that this approach allows exact profiling with overheads that are about two orders of magnitude lower[15] than standard tools for Java. Thus, the developer no longer has to carefully select the sub-parts of his application that he wants to profile.

2. While profiling provides detailed execution statistics about individual methods (e.g., calling context, invocation counter, CPU time, etc.), *benchmarking* evaluates the overall performance (CPU consumption, memory utilization, etc.) of a program. Benchmarking at the bytecode level is a tool which helps compare the cost of different algorithm implementations (for a given input), at a higher level than what platform-dependent, plain execution times offer [5].

3. Most tools for *aspect-oriented programming*, such as AspectJ [24], allow to define pointcuts, such as method invocations, the beginning of exception handlers, etc. However, they do not give access to control flow entities like loops and basic blocks, and are therefore not appropriate tools for the insertion of accounting code. Moreover, it would be difficult to specify our particular scheme of passing accounting objects as extra arguments (which involves the automated creation of wrapper methods or the duplication of code)

[15] This overhead is however higher than with J-RAF2 because there is much more information to collect and manage.

with current aspect languages. Therefore, from a technical point of view, a tool with the same bytecode manipulation capabilities as ours is required. We are considering defining an aspect-oriented language in order to factor out the parameters of the rewriting process, and make our current tool more user-friendly. This language would enable the developer to more conveniently define the sites where accounting is needed and what actions should be taken there. We are aware of no prior work on aspect languages for CPU management, and resource management in general.

Whereas our bytecode transformations are currently implemented as a distinct tool, there are also other deployment opportunities, depending on the needs of the envisaged application. The JDK 1.5.0 platform introduced services that allow Java agents to instrument programs running on the JVM. Such Java agents exploit the instrumentation API (package `java.lang.instrument`) to let users install bytecode transformations that are integrated with the JVM instead of being provided by a separate tool. Java agents are invoked after the JVM has been initialized, but before the real application. They are even allowed to redefine already loaded system classes.

However, JDK 1.5.0 imposes several restrictions on the transformation of previously loaded classes. For instance, new fields or methods cannot be added, and method signatures cannot be changed. Therefore it is not compatible with our most advanced transformations. Nonetheless, if all JDK classes are rewritten according to the *Simple* scheme, where each method obtains the `ThreadCPUAccount` instance of the calling thread upon method entry, our bytecode transformations can be packaged as a standard Java agent for JDK 1.5.0. This approach also requires that the transformations themselves do not take too long, as the rewriting would be done on the fly. As an indication, a rewriting of the 2443 classes of IBM JDK 1.4.2 with the *Simple* scheme takes 40 seconds on our benchmarking machine.

Our previous work on J-RAF2 has shown that we can account for other basic resources, such as heap memory [7] and network bandwidth [22], within a single homogeneous conceptual and technical framework. More research is needed to advance the management of these resources to the same level of maturity as the CPU. However, from the perspective of studying program transformations, they are not quite as challenging, since the corresponding consumption sites are easy to locate in the bytecode and can therefore be essentially handled with wrapper techniques, and they are less critical – again purely from a program transformation point of view – because they are consumed at a slower pace than the CPU.

8. Related Work

Altering Java semantics via bytecode transformations has been used for many purposes that can be generally characterized as adding reflection or aspect-orientedness to off-the-shelf software components [9, 23, 30]. Our approach also fits this description, since we reify the CPU consumption, which is an original idea. J-RAF2 relies on the Byte Code Engineering Library BCEL [15] for the low-level operations.

In the following we present related work on the two main subjects of this article, namely resource management in Java, and static path prediction schemes.

8.1. Resource Management in Java

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 [29], KaffeOS [1], and the MVM [12] are specialized JVMs supporting resource control. In contrast, our approach does not require any JVM modifications and is compatible with any standard JVM.

JRes [14] is a resource control library for Java, which uses native code for CPU control and rewrites the bytecode of Java programs for memory control. For CPU control, some light bytecode rewriting is also applied to enable proper cooperation with the OS via native code libraries. JRes does not support bytecode-level accounting, as this seemed prohibitive in terms of performance. Another difference is that in JRes information is obtained by polling the OS about the CPU consumption of threads, and therefore requires a JVM with OS-level threads, which is not always available. Researchers at Sun are also working on incorporating resource management as an integral part of the Java language [13]; this proposal requires substantial additional support that has to be first introduced into the platform.

8.2. Static Path Prediction

Our path prediction scheme may be seen as just another name for branch prediction [11]. However our purpose is slightly different from existing approaches, which focus on lower-level objectives such as designing compilers that generate optimal code for a given family of processors [16], or predicting the worst-case execution time in view of implementing a hard real-time system [20]. Some approaches use per-application tailored feed-back received from off-line simulations: that kind of prediction is in fact *semi-static*, as opposed to purely static approaches like ours, which rely only on the information that can be

extracted through static analysis of the program [25]. Purely static branch prediction performs of course worse than the semi-static and dynamic approaches.

Our work on static path prediction bears many similarities to that of Ball and Larus [2], since they employ largely the same set of techniques for statically analyzing programs. Whereas they studied static prediction with programs in the C language, we are working with Java and could therefore introduce the (obvious) heuristic related to throwing exceptions.

To our knowledge this is the first static prediction scheme that works on the bytecode-to-bytecode level, with purely platform-independent strategies and impact assessment.

9. 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.

This article has made a thorough review of the bytecode transformations that are put to work in our J-RAF2 framework, including a systematic review of the origin of the associated overheads, and the optimizations designed to mitigate them. It has also presented and assessed an entirely new static path prediction scheme which contributes to the reduction of accounting overheads.

We have thus shown that runtime overheads can be reduced to fairly moderate levels, although at first the bytecode rewriting approach might seem overly expensive. Part of the remaining overhead may still be reduced by enhancing our intraprocedural analysis and transformation schemes. But, in the light of our experiments, it seems that the possibility of making further significant progress strongly depends on the adoption of complementary interprocedural analysis tools.

Acknowledgements

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

The authors would like to thank Andrea Camesi, Francesco Devittori and Alex Villazón for their support, as well as the anonymous reviewers and the editors for many valuable comments and suggestions.

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. T. Ball and J. R. Larus. Branch prediction for free. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, 1993.
3. W. Binder and J. Hulaas. Extending standard Java runtime systems for resource management. In *Software Engineering and Middleware (SEM 2004)*, volume 3437 of *LNCS (Lecture Notes in Computer Science)*, pages 154–169, Linz, Austria, Sept. 2004.
4. W. Binder and J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, 8(5):74–83, Sep./Oct. 2004.
5. W. Binder and J. Hulaas. Using bytecode instruction counting as portable CPU consumption metric. In *QAPL'05 (3rd Workshop on Quantitative Aspects of Programming Languages)*, volume 153(2) of *ENTCS (Electronic Notes in Theoretical Computer Science)*, Edinburgh, Scotland, Apr. 2005.
6. W. Binder and J. Hulaas. Exact and portable profiling for Java using bytecode instruction counting. In *QAPL'06 (4th Workshop on Quantitative Aspects of Programming Languages)*, to appear in ENTCS (Electronic Notes in Theoretical Computer Science), Vienna, Austria, Apr. 2006.
7. W. Binder, J. G. Hulaas, and A. Villazón. Portable resource control in Java. *ACM SIGPLAN Notices*, 36(11):139–155, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).
8. 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*, 2535:154–170, 2002.
9. S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.
10. M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. *ACM SIGPLAN Notices*, 35(5):13–26, May 2000.
11. H. G. Cragon. *Branch Strategy Taxonomy and Performance Models*. IEEE Computer Society Press, 1992.
12. 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)*, pages 125–138, Tampa Bay, Florida, Oct. 2001.
13. G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce. A resource management interface for the Java platform. *Software Practice and Experience*, 35(2):123–157, Nov. 2004.
14. 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.
15. M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. `http://jakarta.apache.org/bcel/`.

16. B. L. Deitrich, B.-C. Cheng, and W. W. Hwu. Improving static branch prediction in a compiler. In *IEEE PACT*, pages 214–221, 1998.
17. B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.
18. M. Feeley. Polling efficiently on stock hardware. In *the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 179–187, June 1993.
19. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, MA, USA, second edition, 2000.
20. J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2), June 1998.
21. 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.
22. J. Hulaas and D. Kalas. Monitoring of resource consumption in Java-based application servers. In *Proceedings of the 10th HP Openview University Association Plenary Workshop (HP-OVUA'2003)*, Geneva, Switzerland, July 2003.
23. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.
24. I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, 2003.
25. A. Krall. Improving semi-static branch prediction by code replication. In *Conference on Programming Language Design and Implementation*, volume 29(7), pages 97–106, Orlando, 1994.
26. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
27. T. Ogasawara, H. Komatsu, and T. Nakatani. A study of exception handling and its dynamic optimization in Java. *ACM SIGPLAN Notices*, 36(11):83–95, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).
28. Sun Microsystems, Inc. Java HotSpot Technology. Web pages at `http://java.sun.com/products/hotspot/`.
29. 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 *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00)*, NY, June 2000.
30. E. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002), USA*, volume 2487 of *LNCS*, Oct. 2002.
31. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`.
32. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.