Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 153 (2006) 57–77
www.elsevier.com/locate/entcs

Using Bytecode Instruction Counting as Portable CPU Consumption Metric

Walter Binder and Jarle Hulaas

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

Abstract

Accounting for the CPU consumption of applications is crucial for software development to detect and remove performance bottlenecks (profiling) and to evaluate the performance of algorithms (benchmarking). Moreover, extensible middleware may exploit resource consumption information in order to detect a resource overuse of client components (detection of denial-of-service attacks) or to charge clients for the resource consumption of their deployed components. The Java Virtual Machine (JVM) is a predominant target platform for application and middleware developers, but it currently lacks standard mechanisms for resource management.
In this paper we present a tool, the Java Resource Accounting Framework, Second Edition (J-RAF2), which enables precise CPU management on standard Java runtime environments. J-RAF2 employs a platform-independent CPU consumption metric, the number of executed JVM bytecode instructions. We explain the advantages of this approach to CPU management and present five case studies that show the benefits in different settings.

Keywords: Java, CPU Consumption Metric, Resource Management, Bytecode Engineering, Program Transformations

1 Introduction

Resource management (i.e., accounting and controlling resources like CPU and memory) is essential for software development and monitoring of deployed software. Profiling allows a detailed analysis of the resource consumption of programs. It helps to detect hot spots and performance bottlenecks, guiding the developer in which parts of a program optimizations may pay off. While profiling provides detailed execution statistics on the basis of individual methods (e.g., calling context, invocation counter, CPU time, etc.), benchmarking

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2005.10.032

evaluates the overall performance (CPU consumption, memory utilization, etc.) of a program. Benchmarking is a common technique to compare the efficiency of different algorithms for a given input.

Monitoring of server systems is important to quickly detect performance problems and to tune the system depending on the workload. Moreover, resource management is a prerequisite to prevent resource overuse in extensible middleware that allows hosting of foreign, untrusted, potentially erroneous or malicious software components (prevention of denial-of-service attacks). In a scenario where a provider hosts client components on his server, billing may be based on actual resource consumption, i.e., the provider may charge the client for the resource consumption of the hosted components.

Java [15] and the Java Virtual Machine (JVM) [19] represent a predominant programming language and deployment platform for application and middleware developers. However, current standard Java runtime systems lack mechanisms for resource management.

Profilers for Java are based on the JVM Profiler Interface (JVMPI) [24], which allows native code profiler agents to intercept various events, such as method invocations. Unfortunately, these profiler agents are written in platform-dependent native code, contradicting the Java motto 'write once and run everywhere'. More problematic, exact profiling based on the JVMPI results in enormous overhead. With exact profiling programs are usually running more than factor 10 slower, in extreme cases we even experienced a slowdown of more than factor 4000 (!) due to profiling. As a result, profiling based on the JVMPI is not suited for complex software systems, such as application servers, and impossible to perform on production systems. Developers spend considerable effort to extract parts of their applications to profile them separately, because profiling the whole system would not be feasible due to the extreme overhead. Furthermore, often the measurements affect the runtime characteristics of the profiled application so that the obtained execution statistics are of limited value.

Most Java developers resort to the wall clock time in order to benchmark their algorithms. However, this approach usually gives imprecise results, due to the measurement granularity (which may be in the order of several milliseconds), the varying workload on the benchmarking machine, or the dynamics of just-in-time compilation and garbage collection. Frequently, results are not reproducible and, thus, strictly speaking not scientific. In order to obtain meaningful results, usually a dedicated machine has to be set up for benchmarking (without any backgound processes), the input has to be large enough to significantly exceed the measurement granularity, and the benchmarks have to be executed multiple times, using statistical methods to consolidate the re-

sults. Therefore, correctly benchmarking Java programs is a complex and time-consuming task.

Dynamic class loading and linking in Java eases the development of extensible middleware. Moreover, language safety (achieved by a combination of type safety, automatic memory management, memory protection, and bytecode verification [29]) and class loader namespaces provide some basic mechanisms to isolate software components. However, due to the lack of resource management mechanisms it is not easily possible to detect denial-of-service attacks. In other words, one of the good reasons for choosing Java is that it helps building extensible systems, but Java fails to bring an essential consequent support, namely that of resource management.

In this paper we present a portable resource management framework for Java, called the Java Resource Accounting Framework, Second Edition (J-RAF2) [1], which solves many of the aforementioned shortcomings of Java. J-RAF2 extends standard Java runtime systems with resource management features. As it is written in pure Java, it can be directly used with arbitrary JVMs. So far, we have successfully tested it in Java 2 Standard Edition (J2SE), Java 2 Enterprise Edition (J2EE), and Java 2 Micro Edition (J2ME) environments.

This paper is structured as follows: In the next section we stress the need for portable resource management, focusing on the CPU resource. We introduce bytecode instruction counting as portable metric for CPU consumption. In Section 3 we give an overview of our techniques for portable CPU accounting and control. In Section 4 we present five case studies of successful applications of J-RAF2. Section 5 summarizes related work, followed by a discussion of the strengths and limitations of our approach in Section 6. Finally, Section 7 concludes this paper.

2 Portable CPU Management

In the following we focus on CPU consumption, because it is the most challenging resource to manage in a portable way: One cannot identify explicit CPU consumption sites in the code and, contrary to other resources, it is rather considered continuous, which is reflected by the fact that quantities of CPU can hardly be manipulated as first-class entities in conventional programming environments.

Usually, CPU consumption is measured in seconds. Prevailing approaches to add CPU management to Java, such as JRes [13], periodically access the

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

operating system to obtain the CPU consumption (CPU time) of individual threads. However, this approach requires native code (either a native code library or a modified JVM) and therefore hampers portability. Moreover, it assumes an operating system supporting threads as well as a JVM that maps Java threads to operating system threads, which may not always be available (e.g., there are Java processors providing a JVM in hardware). More importantly, the overhead may become excessive, if detailed execution statistics are needed (e.g., profiling).

Another problem with measuring CPU consumption in seconds is the varying amount of processing that can be achieved within one CPU second. On a modern server based on a processor with a high clock rate an application may process a large amount of data in one CPU second, while on an embedded device only a small fraction of the same workload could be accomplished. Even on the the same hardware and operating system, the performance of distinct JVM versions may differ significantly. Using the CPU second as metric to specify 'execution contracts' (e.g., CPU limits for mobile code) is problematic, as client and service provider (who will host the client code) may use a different CPU as reference. Charging clients for the CPU consumption of deployed components is also complicated, since the client may not exactly know (and may not be able to verify) how much effective processing power he purchases with one CPU second. For these reasons, the CPU second is not an appropriate metric in a distributed, heterogeneous environment.

In order to avoid these problems, we designed and implemented a fully portable CPU management scheme that can be installed on any existing Java runtime systems, just like a normal Java application. We exploit the number of executed JVM bytecode instructions as metric for CPU consumption. With the aid of bytecode rewriting techniques, Java class files are transformed so that during execution each thread maintains a bytecode instruction counter indicating the number of executed bytecode instructions. For this purpose, we insert accounting instructions at well chosen locations. More details of our accounting scheme are presented in the next section. Our bytecode transformation scheme can be applied to application classes and to libraries, including the classes of the Java Development Kit (JDK).

As the JVM bytecode is a portable code format, the bytecode instruction counters will refer to the same number of executed bytecode instructions no matter on which JVM the transformed code is executed. Using bytecode instruction counting as CPU consumption metric has many advantages:

- 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.

- As the bytecode instruction counting is encoded directly into the program code, the just-in-time compiler of the JVM will optimize the whole transformed program, including the accounting code. Hence, the resulting overhead for CPU management can be reduced.
- Independent of the execution platform, a given program will compute the same CPU consumption value.
- CPU consumption statistics are exactly reproducible and comparable across different platforms.

Consequently, exploiting the number of executed bytecode instructions as metric is key to providing a single CPU management tool across all kinds of Java platforms, to build reliable and sufficiently efficient profiling and benchmarking tools, and to establish 'execution contracts' in a distributed, heterogeneous environment.

3 J-RAF2

In our CPU accounting scheme the bytecode of Java classes is rewritten in order to make its CPU consumption explicit. Each thread computes its own CPU consumption, expressed as the number of executed JVM bytecode instructions. Periodically, each thread aggregates the collected information concerning its own CPU consumption within an account that may be shared with a number of other threads. We call this approach *self-accounting*. During these information update routines, the thread will also execute management code, e.g., to ensure that a given resource quota is not exceeded. In this way, the CPU management scheme of J-RAF2 does not rely on a dedicated supervisor thread, since the management activity is distributed among all threads in the system, thus effectively implementing a form of *self-control*.

Hence, and this is for us a guarantee of portability and reliability, we do not rely on the underlying scheduling provided by the JVM, which is left loosely specified in the Java language [15] and JVM [19] specifications: While some JVMs seem to provide preemptive scheduling ensuring that a thread with high priority will execute whenever it is ready to run, other JVMs do not respect thread priorities at all. This is a major difference to our previous accounting scheme [5], which relied on thread priorities for scheduling. In contrast, J-RAF2 enables the user to write platform-independent CPU management code (such as custom schedulers), which works across all kinds of Java platforms.

In the following subsections we summarize our portable CPU accounting scheme. Low-level implementation details of the J-RAF2 runtime classes are presented elsewhere [16], as are the programming APIs together with program-

ming examples from the viewpoint of a middleware developer [4]. In contrast to these previous publications, this paper focuses on the benefits and general applicability of our CPU management scheme.

3.1 Associating Accounting Information with Threads

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

Each thread has an associated `ThreadCPUAccount`. Fig. 1 summarizes part of the public interface. The semantics of the methods and fields are explained in this and in the following subsections. The association of a thread with its `ThreadCPUAccount` persists for the whole life-time of the thread. When a new thread object is initialized, it automatically receives a fresh `ThreadCPUAccount` object [16]. The `getCurrentAccount()` method returns the `ThreadCPUAccount` object of the calling thread.

```
public final class ThreadCPUAccount {
   public static ThreadCPUAccount getCurrentAccount();

   public int consumption;
   public void consume();

   public void setManager(CPUManager m);
   ...
}
```

Fig. 1. Part of the `ThreadCPUAccount` API.

3.2 Bytecode Transformation Scheme

During normal execution each thread updates the `consumption` counter of its `ThreadCPUAccount`. In order to schedule regular activation of the shared management tasks, the counter is checked against an adjustable limit, the *accounting granularity* [16]. More precisely, each thread invokes the `consume()` method of its `ThreadCPUAccount`, when the local `consumption` counter exceeds a certain limit defined by the accounting granularity. In order to optimize the comparison whether the `consumption` counter exceeds this limit, the counter runs from the granularity value multiplied by -1 to zero, and when it equals or exceeds zero, the `consume()` method is called. In the JVM bytecode there are dedicated instructions for the comparison with zero. In order to apply this CPU accounting scheme, (non-native and non-abstract) methods are rewritten in the following way:

(i) At the beginning of each method the current thread's `ThreadCPUAccount` has to be obtained using the static method `getCurrentAccount()`.

(ii) Conditionals are inserted in order to invoke the `consume()` method periodically. The general idea is to minimize the number of checks whether `consume()` has to be invoked for performance reasons, but still to make sure that malicious code cannot execute an unlimited number of bytecode instructions without invocation of `consume()`. The conditional '`if (cpu.consumption >= 0) cpu.consume();`' is inserted at the beginning of each method and in each loop, ensuring that the conditional will be re-evaluated during iterative computations (recursions or loops). In the conditional the variable `cpu` refers to the `ThreadCPUAccount` of the currently executing thread.

(iii) Finally, the instructions that update the `consumption` counter are inserted at the beginning of each accounting block. An accounting block is related to the concept of basic block of code with the difference that method and constructor invocations may occur at any place within an accounting block. Details concerning the definition of accounting blocks can be found elsewhere [5]. In order to reduce the accounting overhead, the conditionals inserted before are not considered as separate accounting blocks.

3.3 Rewriting Example

Fig. 2 illustrates how a method is transformed using our CPU accounting scheme. In this example 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.[2] Depending on the application, the concrete value for each accounting block can be computed in different ways:

- The number of bytecode instructions in the accounting block before the rewriting takes place. I.e., the resulting CPU consumption reflects the number of bytecode instructions that the original, unmodified program would execute. This setting is particularly useful for profiling and benchmarking.
- 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 con-

[2] For the sake of better readability, we show the transformations on Java code, whereas our implementation works at the JVM bytecode level.

sumption of the deployed client components.

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

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

Fig. 2. Exemplary method before and after rewriting.

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.[3] The `CPUManager` implementation is provided by the middleware developer and implements the actual CPU accounting and control strategies, e.g., custom scheduling schemes. J-RAF2 provides an inheritance mechanism that guarantees that a spawned thread is initially subjected to the same CPU management scheme as its creator thread, i.e., the spawned thread's `ThreadCPUAccount` inherits the `CPUManager` reference from the creator thread's `ThreadCPUAccount`. The `setManager(CPUManager)` method of `ThreadCPUAccount` allows the programmer to explicitly change the `CPUManager` instance.

The `CPUManager` interface includes the method `consume(long)`. Whenever a thread invokes `consume()` on its `ThreadCPUAccount`, this method will in turn report its collected CPU consumption data (stored in the `consumption` field) to the `CPUManager` associated with the `ThreadCPUAccount` by calling `consume(long)`. The `consume(long)` method implements the custom CPU accounting and control policy. It may simply aggregate the

[3] Here the term 'component' takes the meaning of an informal group of threads subjected to the same `CPUManager` object, and hence, logically (but not necessarily), to the same management policy. Depending on the setting, a component may translate e.g. to a pool of reusable threads, or to a concrete protection domain like an *isolate* [18].

reported CPU consumption (and write it to a log file or database), it may enforce absolute limits and terminate components that exceed their CPU limit, or it may limit the execution rate of threads of a component (i.e., putting threads temporarily to sleep if a given execution rate is exceeded). This is possible without breaking security assumptions, since the `consume(long)` invocation is synchronous (i.e., blocking), and executed directly by the thread to which the policy applies.

As an example, a trivial `CPUManager` implementation is depicted in Fig. 3. The `consume(long)` method is `synchronized`, as multiple threads may invoke it concurrently. The `SimpleCPUManager` implementation maintains the sum of all reported consumption information, it does not enforce any CPU limit.

```
public class SimpleCPUManager implements CPUManager {
   protected long consumption = 0;
   public synchronized void consume(long c) {consumption += c;}
   public synchronized long getConsumption() {return consumption;}
}
```

Fig. 3. `CPUManager` implementation: CPU accounting without control.

3.5 Performance

Performance evaluations revealed that CPU management based on bytecode instruction counting causes 17–30% overhead on recent JVMs. We ran the SPEC JVM98 benchmark suite [27] on a Linux RedHat 9 computer (Intel Pentium 4, 2.6 GHz, 512 MB RAM). The JVM98 classes as well as all JDK classes were rewritten for resource management. The entire JVM98 benchmark (which consists of several sub-tests) was run 10 times, and the final results were obtained by calculating the geometric mean of the median of each sub-test.

The most promising results were obtained with IBM's JDK 1.4.2 platform, where the overhead could be kept as low as 17%. With Sun's JDK 1.5.0, the overhead was about 25% for the HotSpot Client VM and 30% for the HotSpot Server VM. Interestingly, in absolute time, on IBM's JVM the benchmarks with accounting executed 20% faster than on Sun's HotSpot Client VM without accounting, and as fast as on Sun's HotSpot Server VM without accounting. This is an indication that a standard JVM (in this case IBM's JVM) enhanced with J-RAF2's portable CPU management mechanisms may offer competitive performance.

4 Case Studies

In this section we discuss five recent case studies where we have applied J-RAF2 in order to evaluate and compare the performance of algorithms, to enhance existing middleware with CPU management features, and to improve security and load-balancing in computational grids, as well as software deployment in a pervasive computing scenario.

4.1 Benchmarking and Profiling

As discussed in Section 1, benchmarking and profiling of Java applications is a complex and time-consuming task. On the one hand, existing Java profilers cause an enormous slowdown of usually factor 10 for exact profiling. Hence, they are not well suited to evaluate complex applications. On the other hand, simple benchmarking of algorithms based on the elapsed wall clock time requires a well prepared and isolated benchmarking machine (without background processes), multiple runs, as well as a statistical processing of the data, since usually measurement results are not exactly reproducible, because of the measurement granularity and the dynamics of just-in-time compilation and garbage collection.

We have made good experience in using J-RAF2 for performance evaluations. Thanks to the abstract measurement unit introduced by J-RAF2, results are exactly reproducible, although the benchmarking may run as a background process on the developer's machine. The accounting is always exact, there is no need to blow up the input data in order to reduce the imprecisions due to the measurement granularity. We are now using J-RAF2 in the evaluation of various service composition algorithms [10]. The use of J-RAF2 for performance evaluation has resulted in a gain of producitvity, because we do not have to maintain a dedicated benchmarking environment.

Fig. 4 illustrates the simplicity of our benchmarking solution. We assume that the main class of the benchmark implements the `Runnable` interface and that its class name is passed as command line argument. The shown program dynamically loads and rewrites the class files of the benchmark. First the `JRAF2ClassLoader` dynamically applies the J-RAF2 bytecode rewriting tool to the loaded classes. Then the program creates a fresh `CPUManager` instance and associates it with the `ThreadCPUAccount` object of the current thread (i.e., the primordial application thread). The invocation of the `consume()` method of the `ThreadCPUAccount` ensures that the value of the `ThreadCPUAccount`'s `consumption` counter is propagated to the `CPUManager` instance. To compute the number of bytecode instructions executed by the benchmark, we simply compare the difference of the CPU

consumption before and after the benchmark's execution. The `CPUManager` inheritance mechanism ensures that threads spawned by the benchmark will report their CPU consumption to the same `CPUManager` object.

```
public static void main(String[] args) throws Exception {
  ClassLoader cl = new JRAF2ClassLoader(); // dynamically rewrites loaded classes
  Class c = cl.loadClass(args[0], true); // load, rewrite, and link desired class
  Runnable benchmark = (Runnable)c.newInstance(); // instantiate rewritten class
  SimpleCPUManager manager = new SimpleCPUManager();
  ThreadCPUAccount cpu = ThreadCPUAccount.getCurrentAccount();
  cpu.setManager(manager);
  cpu.consume();                    // flush ThreadCPUAccount consumption counter
  long consumptionBefore = manager.getConsumption();
  benchmark.run();
  cpu.consume();                    // flush ThreadCPUAccount consumption counter
  long consumptionAfter = manager.getConsumption();
  System.out.println("Bytecode instructions executed by benchmark: " +
                     (consumptionAfter - consumptionBefore));
}
```

Fig. 4. Benchmarking applications with J-RAF2.

If the benchmark classes employ functionality of the JDK (which usually is the case), the JDK has to be rewritten for resource management as well. While in this example the benchmark classes are dynamically rewritten by a special J-RAF2 classloader, the JDK classes always have to be rewritten offline, before the benchmark is run. This happens once during the installation of J-RAF2, which creates a resource-aware version of a previously installed JDK. Many recent JVMs support the '`-Xbootclasspath`' option, which can be used to force the JVM to bootstrap the resource-aware JDK. If a JVM does not support this option, the core classes have to be replaced with the rewritten versions (e.g., this may be the case in a J2ME setting).

By using distinct `CPUManager` instances for different parts of the program execution, it is possible to differentiate the CPU consumption (profiling). However, as this approach is not practical for fine-grained profiling, we are developing another tool, the Java Profiler JP. JP is based on similar bytecode rewriting techniques as J-RAF2, but it maintains separate information for each method: The full call stack (invocation context), a method invocation counter, and the number of executed bytecode instructions by all invocations of the method. We have already obtained detailed execution statistics for the SPEC JVM98 benchmarks [27], significantly faster than with any other exact Java profiler we could find (on average, JP causes an overhead of factor 2–3 for exact profiling).

4.2 Accounting in an Application Server

In addition to the ubiquitous issues of security and reliability, E-commerce infrastructure, such as application servers, also need to be highly available as well as profitable. In order to investigate resource accounting in such an environment in support of billing strategies, we applied our framework to the Apache Tomcat servlet engine[4]. We were able to precisely monitor CPU and network bandwidth consumption on a per-request basis and to report this data in real-time to a database server.

The main challenges were, first, to be able to assign a semantic, real-world meaning to the unstructured, low-level accounting information gathered, and, second, to cope with the fact that Tomcat uses thread pooling for the execution of http requests. This is further complicated by the fact that a servlet may freely spawn worker threads which will not be noticed by the Tomcat engine, resulting in additional resource consumption that has to be correctly reported. The constraint we imposed ourselves was to consider servlet code as legacy (i.e., the source code is not always available for modification). On the other hand, we allowed ourselves to take advantage of Tomcat being an open-source project to extract the semantic association between an http request and the identity of the main thread elaborating the corresponding reply. Exploiting the manager inheritance mechanism mentioned in the previous section, we let each worker thread inherit from the dedicated `CPUManager` object of its respective main request thread. This combination allows integrating the consumption of worker threads, and thus we achieve a fairly complete and straightforward solution to the mentioned challenges.

4.3 Absolute CPU Limits in Extensible Directories

In our previous work [3,8,9] a directory for (semantic) web services has been presented. It offers specific features to enable the efficient composition of web services, taking type constraints of input and output messages into account [10]. Service composition algorithms access the directory retrieving descriptions of web services that can be combined in order to fulfill given requirements, defined as a set of required output messages. As the number of relevant services for a particular service composition problem may be very large, the directory allows for the incremental retrieval of results. The performance of the service composition algorithm depends very much on the order in which (partially) matching results are returned by the directory. Because the research on service composition is still in the beginnings and needs a lot of experimentation to develop industrial-strength algorithms, the directory

[4] http://jakarta.apache.org/tomcat/

shall be flexible to support various ordering heuristics, as for different service composition algorithms distinct heuristics may be more effective.

The directory was later extended to support user-defined pruning and ranking functions, which enable the dynamic installation of application-specific heuristics directly within the directory. That is, the new version of the directory is extensible. The custom pruning and ranking functions are written in Java for the following reasons: Java is well known to many programmers, there are lots of programming tools for Java, and, above all, it integrates very well with the directory, which is completely written in Java.

Integrating user-defined code into the directory leverages state-of-the-art optimizations in recent JVM implementations. For instance, the HotSpot VM [23] first interprets JVM bytecode and gathers execution statistics. If code is executed frequently enough, it is compiled to optimized native code for fast execution. In this way, frequently used pruning and ranking functions are executed as efficiently as algorithms directly built into the directory.

In order to protect the directory against erroneous or malicious client code, it imposes severe restrictions on user-defined pruning and ranking functions. For instance, the client code may use only a very limited API, it is not allowed to allocate memory on the heap, it must not use synchronization primitives, and it cannot define exception handlers. Efficient, extended bytecode verification to enforce restrictions on JVM bytecode for the safe execution of untrusted mobile code has been studied in the JavaSeal [28] and in the J-SEAL2 [2,5] mobile object kernels. The aforementioned restrictions are enforced at load-time and partly at runtime, and ensure that the user-defined code cannot interfere with the internals of the directory causing unwanted side-effects. In order to prevent denial-of-service attacks, an early version of the extensible directory even required client code to be acyclic, i.e., loops were disallowed.

Recently, we have used J-RAF2 to overcome this limitation and to rewrite the custom pruning and ranking functions for CPU control. These functions may now use loop constructs, as their CPU consumption is limited by a CPU control policy defined by the provider of the directory. The execution of a query requires the repeated invocation of the client code. Before calling the user-defined function, the thread attaches to a `CPUManager` that enforces a strict absolute limit on the CPU consumption of each query. The CPU limit is expressed as the number of allowed bytecode instructions to be executed by the user-defined code throughout the processing of the whole query. If the limit is exceeded, the `consume(long)` method throws an exception which will abort the execution of the user code (remember that the custom functions are not allowed to define exception handlers). The directory will then catch the exception and terminate the whole query.

In this setting, the runtime overhead for CPU accounting is negligible, as only the untrusted, user-defined code is rewritten for CPU accounting. The directory itself is not rewritten, and its execution is not accounted for. As service composition clients are likely to use the same set of pruning and ranking functions for multiple queries, the directory keeps a cache of recently used pruning and ranking functions (verified, rewritten, and loaded). Therefore, the overhead of dynamic bytecode rewriting is mitigated.

4.4 Security and Load-balancing in Computational Grids

In other previous work [17] we described a model of computational grid relying on mobile agents running inside a secure Java-based kernel. The objective of this model is to propose a realistic deployment scenario, both from an economic and technical point of view, since we describe a setting where providers of computing resources (individuals or enterprises) may receive rewards in proportion to their service, and where issues like performance and security are addressed extensively, relying on actual tools and environments.

Using the number of executed bytecode instructions as a metric helps solving two important problems that are more specific to the grid setting. The first is that it is hard to distribute the computational load in an extremely heterogenenous environment. To address this issue, all providers of computing resources feed the grid operator with their up-to-date load information expressed as the number of recently executed Java bytecodes. This data is compared to the respective load capacities and declared preferences of the resource providers to adjust the distribution of new computational tasks.

The second issue that has to be solved in this scenario, where resource providers may receive (possibly financial) rewards, is to prevent cheating. Our proposed grid business model relies on the one hand, on clients paying the grid operator for the distributed execution of their application, and on the other hand, on the operator paying the resource providers for offering their idle computing resources. The client buys execution tickets (special tokens) from the operator, which the deployment agent passes to the resource providers for their services. The latter redeem the received execution tickets at the operator. The execution tickets resemble a sort of cryptographically protected currency, valid only within the grid.

In case a resource provider does not offer the desired service against a given execution ticket, the grid operator will quickly notice it. If it turns out that a resource provider collects tickets without delivering the appropriate service, the operator may ultimately decide to remove him from the grid. The detection of such malicious actors is possible by correlating the amount of requested tickets with the work actually performed, which is measured by

CPU monitoring inside the dedicated execution environment.

4.5 Software Deployment in Pervasive Computing

In the EU project PalCom [5] we explore the following scenario: In the domain of pervasive computing, a commonly expected feature is the ability of software to spread from one device to another, resulting in a kind of epidemic dissemination and upgrading of software components. As in the above mentioned grid example, this is a strongly heterogeneous setting. Therefore, it is necessary to express the computing power of each device in a portable way, because of the risk of overloading a scarcely configured device with a heavy component.

The other way round, a target device will want to know in advance whether the proposed software component has static or dynamic requirements that exceed the local possibilities. To solve this issue, we propose to equip each component (e.g., a Java jar file) with a text file describing its static (i.e., load-time and activation) resource requirements. Additionally, the dynamic needs of the component and policies of the environment shall be expressed in terms of absolute and rate-based CPU consumption (using bytecode instruction counting as metric), in order to guarantee portability of resource-aware behaviours.

If stronger, e.g., real-time, guarantees are needed for the correct execution of a software component, it is advisable to profile it off-line, in order to obtain the upper bounds of all required resources in portable units of measurement. The component would declare these values as its load-time requirements, and would then only be loaded by target devices that can comply with them.

While the previous grid computing scenario is rather static and centralized (the operator knows in advance if a component is adequate for a given target host), this pervasive computing example is as decentralized as possible, because each device and each component have to be able to determine whether they are compatible.

5 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 [26]. Our approach also fits this description, since we actually reify the CPU consumption, which is an original idea. Whereas many tools have been developed for engineering bytecode, J-RAF2 relies on an existing one, BCEL [14], for the low-level operations.

[5] http://www.palcom.dk/

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 [25], KaffeOS [1], and the MVM [11] are specialized JVMs supporting resource control. JRes [13] 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 was also applied to enable proper cooperation with the OS via native code libraries. Bytecode-level accounting was not done, 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.

More recently, researchers at Sun have published a report relating their approach to incorporating resource management as an integral part of the Java language [12]. They have embraced a very broad field of investigation, since their ambitions are, e.g., to account for physical as well as logical resources (like ports or file descriptors), and to provide direct support for sophisticated management policies with multi-party decision taking and notification (whereas J-RAF2 focuses on the lower-level facilites, while leaving a lot of flexibility to the application and middleware developer). One notable aspect of their proposal is that it requires the prior deployment of Java isolates [18].

Several other management and runtime monitoring APIs have been offered by Sun along with the successive releases of Java platforms, especially for heap memory, but currently no solution is applicable as widely across environments, nor is there one also usable as a basis for implementing control policies (as opposed to monitoring), or as well integrated with the language as the present framework.

Previous work on J-RAF [5] has shown that other basic resources such as heap memory and network bandwidth may be accounted for within a single homogeneous conceptual and technical framework. More research is needed to advance accounting for these resources to the same level of maturity as the CPU management framework presented here.

A formal model for resource management in a hierarchy of groups (e.g., components, processes) is presented by Moreau and Queinnec [20]. In this model, the parent group provides the resources for its children groups. A notification mechanism informs the parent when the child runs out of resources. The authors also describe a Java prototype, but they do not mention any support for CPU management. A similar hierarchical resource management model was used by the J-SEAL2 kernel [5], which also supported CPU management, but based on a much less reliable accounting scheme than the one

presented in this paper.

The Real-Time for Java Experts Group [7] has published a proposal to add real-time extensions to Java. One important focus of this work is to ensure predictable garbage collection characteristics in order to meet real-time guarantees. Another real-time system, PERC [21], extends Java to support real-time performance guarantees. To this end, the PERC system analyzes Java bytecodes to determine memory requirements and maximal execution times, and feeds that information to a real-time scheduler. The objective of real-time systems is to provide precise guarantees, e.g., for worst-time execution. In contrast, our focus is on computing resource consumption in a hardware-independent measurement unit, which is sufficient for monitoring and to prevent denial-of-service attacks.

6 Discussion

In this section we discuss the strengths and limitations of our CPU management framework.

First and most importantly, our CPU management scheme is fully portable. J-RAF2 is implemented in pure Java and all transformations follow a strict adherence to the specification of the Java language and virtual machine. It has been successfully tested with several standard JVMs in different environments, including also the Java 2 Micro Edition [6].

A novelty of our approach is the use of a portable, hardware-independent unit of measurement for CPU consumption. In our approach, the modified program itself computes its CPU consumption, expressed as the number of executed JVM bytecode instructions. This has significant advantages for the profiling and benchmarking of applications, as results are exactly reproducible. Moreover, on this basis server and client can agree upon CPU quotas without referring to machine characteristics, such as processor model, clock rate, etc. In distributed applications this is a clear advantage, since we may then envision platform-independent 'execution contracts' between heterogeneous hosts, as well as the specification of platform-independent schedulers and scheduling policies.

As business models where service providers host client software components become more widespread, middleware for servers will have to offer resource management functionality to monitor deployed client applications and to charge them for their resource consumption. Whereas standard JVMs will not be equipped with CPU management capabilities in the foreseeable future

(a JSR [6] on resource management has not been initiated yet), J-RAF2 offers a solution today to enhance standard Java runtime systems with CPU management features. An alternative would be to use a modified JVM, but most available JVMs supporting resource management perform significantly worse than standard Java runtime systems. For instance, the Aroma VM [25] does not include any just-in-time compiler, and the authors of KaffeOS [7] [1] reported that in the absence of denial-of-service attacks IBM's compiler and JVM [22] was 2–5 times faster than theirs (standard JVMs have significantly improved performance since this performance comparison was made). The MVM [11], which may offer resource management with less overhead, has not been released to the public. Compared with these approaches, the 17–30% overhead of our portable CPU management scheme seems acceptable.

J-RAF2 offers a small but flexible API. On the one hand, it supports the installation of custom `CPUManager`s for legacy applications without requiring any manual changes to the classes. On the other hand, middleware developers may exploit the `ThreadCPUAccount` API in order to aggregate CPU consumption for individual components in a flexible way. While J-RAF2 provides only a low-level API, advanced features, such as triggers and callbacks [12], may be added in a `CPUManager` implementation by the programmer.

Our proposal for CPU management is built on the idea of self-accounting. Thus, we probably offer the most precise, fine-grained accounting basis available. Moreover, this approach solves one important weakness of all existing solutions based on a polling supervisor thread: The Java specification does not formally guarantee that the supervisor thread will ever be scheduled, whatever its priority is set to. In contrast, in J-RAF2 any resources consumed will be accounted by the consuming thread itself (provided that the consuming code is implemented in Java, and not in some native language), and, if required, the thread will eventually take self-correcting measures.

Concerning limitations, the major hurdle of our approach is that it cannot directly account for the execution of native code. We believe that a range of diverse solutions must be put to work, some of which we have described previously [5], especially concerning memory attacks. In particular, a combination of memory and CPU control is needed in order to prevent denial-of-service attacks through the garbage collector. Allocating a large amount of objects may require only relatively few bytecode instructions, but may cause considerable work for the garbage collector. A simple but effective solution is to charge applications at the moment of object allocation for the estimated future garbage

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

[7] The JanosVM, a successor of KaffeOS conceived as a Java-oriented active network operating system, is available at: http://www.cs.utah.edu/flux/janos/

collection costs [5]. This approach also has the advantage that in the case of termination of a component (or migration of a mobile agent), the component has already been charged for the garbage it leaves in the system.

Security has not been addressed in this paper. Nevertheless, we have load-time and runtime verification algorithms designed to prevent untrusted applications from tampering with their `ThreadCPUAccount` objects, be it directly or indirectly by reflection.

7 Conclusion

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

Exploiting the number of executed bytecode instructions as portable metric for CPU consumption allows us to build reliable benchmarking and profiling tools. Moreover, it is a common base to establish 'execution contracts' in a heterogeneous environment. In this paper we have illustrated the advantages of our portable CPU management scheme with several case studies.

Among our forthcoming investigations, one exciting test will be to explore to which extent our hypotheses remain valid across other virtual machines, the most obvious challenger being the *.Net* platform, for which our initial experiments have confirmed that our approach is also largely applicable to this other environment. Furthermore, we are working on a new profiling system, the Java Profiler JP, which builds on the program transformation techniques we developed for J-RAF2, but offers exact profiling information on a per-method basis. Initial measurements indicate that JP performs significantly faster than traditional Java profilers.

References

[1] Back, G., 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, 2000.

[2] Binder, W., *Design and implementation of the J-SEAL2 mobile agent kernel*, in: *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, 2001, pp. 35–42.

[3] Binder, W., I. Constantinescu and B. Faltings, *A directory for web service integration supporting custom query pruning and ranking*, in: L.-J. Zhang, editor, *European Conference on Web Services (ECOWS 2004)*, Lecture Notes in Computer Science **3250** (2004), pp. 87–101.

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

[5] Binder, W., J. G. Hulaas and A. Villazón, *Portable resource control in Java*, ACM SIGPLAN Notices **36** (2001), pp. 139–155, proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[6] Binder, W. 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).

[7] Bollella, G., B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin and M. Turnbull, "The Real-Time Specification for Java," Addison-Wesley, Reading, MA, USA, 2000.

[8] Constantinescu, I., W. Binder and B. Faltings, *Directory services for incremental service integration*, in: *First European Semantic Web Symposium (ESWS-2004)*, Heraklion, Greece, 2004.

[9] Constantinescu, I. and B. Faltings, *Efficient matchmaking and directory services*, in: *The 2003 IEEE/WIC International Conference on Web Intelligence*, 2003.

[10] Constantinescu, I., B. Faltings and W. Binder, *Large scale, type-compatible service composition*, in: *IEEE International Conference on Web Services (ICWS-2004)*, San Diego, CA, USA, 2004.

[11] Czajkowski, G. 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, 2001.

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

[13] Czajkowski, G. 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)*, ACM SIGPLAN Notices **33, 10**, New York, USA, 1998.

[14] Dahm, M., *Byte code engineering*, in: *Java-Information-Tage 1999 (JIT'99)*, 1999, http://jakarta.apache.org/bcel/.

[15] Gosling, J., B. Joy, G. L. Steele and G. Bracha, "The Java language specification," Java series, Addison-Wesley, Reading, MA, USA, 2000, second edition, xxv + 505 pp.

[16] Hulaas, J. and W. Binder, *Program transformations for portable CPU accounting and control in Java*, in: *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, Verona, Italy, 2004, pp. 169–177.

[17] Hulaas, J., W. Binder and G. D. M. Serugendo, *Enhancing Java grid computing security with resource control*, in: *International Conference on Grid Services Engineering and Management (GSEM 2004)*, Erfurt, Germany, 2004.

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

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

[20] Moreau, L. and C. Queinnec, *Resource aware programming*, ACM Trans. Program. Lang. Syst. **27** (2005), pp. 441–476.

[21] Nilsen, K., *Java for real-time*, Real-Time Systems Journal **11(2)** (1996).

[22] Suganuma, T., T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu and T. Nakatani, *Overview of the IBM Java Just-in-Time compiler*, IBM Systems Journal **39(1)** (2000).

[23] Sun Microsystems, Inc., *Java HotSpot Technology*, Web pages at http://java.sun.com/products/hotspot/.

[24] Sun Microsystems, Inc., *Java Virtual Machine Profiler Interface (JVMPI)*, Web pages at http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/.

[25] Suri, N., 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, 2000.

[26] Tanter, E., 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*, LNCS **2487**, 2002.

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

[28] Vitek, J., C. Bryce and W. Binder, *Designing JavaSeal or how to make Java safe for agents*, Technical report, University of Geneva (1998), http://cui.unige.ch/OSG/pubs/OO-articles/TechnicalReports/98/javaSeal.pdf.

[29] Yellin, F., *Low level security in Java*, in: *Fourth International Conference on the World-Wide Web*, MIT, Boston, USA, 1995.