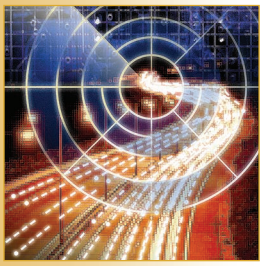

Editors: Doug Lea • *dl@cs.oswego.edu*
Steve Vinoski • *vinoski@ieee.org*

A Portable CPU-Management Framework for Java

The Java Resource Accounting Framework, second edition (J-RAF2), is a portable CPU-management framework for Java environments. It is based on fully automated program-transformation techniques applied at the bytecode level and can be used with every standard Java virtual machine. J-RAF2 modifies applications, libraries, and the Java development kit itself to expose details regarding thread execution. This article focuses on the extensible runtime APIs, which are designed to let developers tailor management policies to their needs.

Walter Binder and Jarle Hulaas
Swiss Federal Institute of Technology, Lausanne

Resource management — accounting and controlling resources such as CPU, memory, and network bandwidth — is a useful, though relatively unexplored, aspect of software. Increased security, reliability, performance, and context-awareness are some potential benefits of a better understanding of resource management. CPU consumption is probably the most challenging resource type to manage. After all, we can't identify explicit CPU-consumption sites in the code because, unlike other resources, it is considered continuous — that is, hardly expressible as discrete amounts (reflected by the fact that we can't manipulate quantities of CPU as first-class entities in conventional programming environments).

Properly managed CPU consumption is extremely valuable, especially with Internet applications. Current standard security mechanisms, such as digital certificates, tend to follow static approaches and focus exclusively on access control. They don't address dynamic aspects such as maximal execution rates or the number of concurrent threads allowed on a given system. These dynamic aspects are nevertheless essential to security and stability. Moreover, they are basic components of tomorrow's context-aware embedded applications.

For families of applications including utility computing, Web services, and grid computing, Java[1] and the Java virtual machine (JVM)[2] represent a dominant programming language and deployment platform. However, the Java language and standard runtime systems lack resource-management mechanisms, which could be used to limit hosted components' resource consumption, for example, or to charge clients for their deployed components' resource consumption.

 Published by the IEEE Computer Society 1089-7801/04/$20.00

Our proposed Java Resource Accounting Framework, second edition (J-RAF2; www.jraf2.org), is a portable CPU-management framework for Java. It introduces an API that middleware developers can use to manage legacy code's CPU consumption, and that application developers can use to implement new forms of resource-aware behaviors. In this article, we describe our approach and illustrate the APIs' use in a series of small implementation examples and two larger case studies.

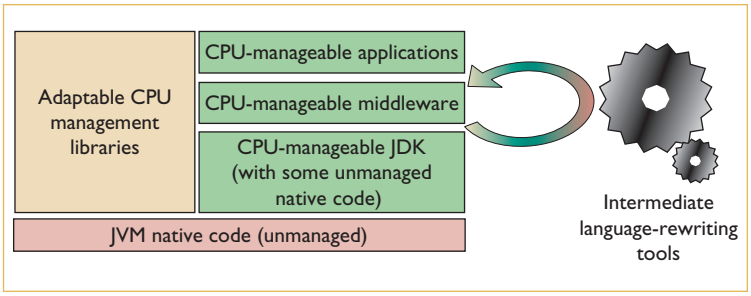

Figure 1. Portable Java CPU-management framework. The J-RAF2 bytecode transformation tool (to the right) makes applications, middleware libraries, and the Java development kit CPU-manageable by making them expose details of their CPU consumption.

Portable CPU Accounting

To develop a resource-management extension that works with arbitrary standard Java runtime systems, our primary design goals have been portability and compatibility. The biggest challenge was to develop a fully portable CPU-management scheme that doesn't cause excessive overhead.

Why Portable?

A resource-management framework should be as easy to deploy as any Java application, especially when the execution overhead is reasonable. Of course, there is a tension between the need for low-level accounting information and control in resource management and ensuring ease of implementation with the JVM and associated runtime libraries. More knowledge is also needed in the field of resource management, which lacks a broadly accepted programming model.

All current approaches to resource management somehow rely on the support of native, nonportable code. In contrast, J-RAF2 achieves portability through a combination of bytecode transformations and runtime libraries implemented in pure Java. Figure 1 illustrates what it would be like if all Java code, including applications, middleware, and the Java development kit (except the subset that is implemented in native code), were CPU-manageable via bytecode transformations (symbolized by the round arrow flowing through the "rewriting machinery") with a small, but adaptable (pure Java) management library, which is required at runtime. Unlike other approaches, ours is independent of any particular JVM or operating system. It works with standard Java runtime systems and can be integrated into existing server and mobile-object environments. Furthermore, this approach enables resource control within embedded systems that use modern Java processors, which provide hard-to-modify JVMs implemented in hardware.

General Approach

J-RAF2 rewrites the bytecode in Java classes to make the programs' resource consumption explicit by tracking the number of executed bytecode instructions. More precisely, each thread permanently accounts for its own CPU consumption, expressed as the number of executed JVM bytecode instructions. Periodically, each thread aggregates the information concerning its CPU consumption within an account shared with several other threads. We call this approach *self-accounting*. During these information-update routines, the thread also executes management code to ensure, for example, that it hasn't exceeded a given resource quota. Because the management activity is distributed among all active threads, J-RAF2's CPU-management scheme implements a form of self-control and avoids reliance on a dedicated supervisor thread.

To guarantee portability and reliability, we avoid relying on the JVM's loosely specified underlying scheduling. While some JVMs seem to provide preemptive scheduling, which ensures that high-priority threads execute whenever they're ready to run, other JVMs don't respect thread priorities at all.

Bytecode-Transformation Scheme

With the bytecode-transformation scheme, our two main design goals are to ensure portability (by strictly adhering to the Java language specification[1] and the JVM specification[2]) and performance (by minimizing the added overhead from inserting instructions into the original classes).

Figure 2 (next page) illustrates part of the public interface for the `ThreadCPUAccount` associated with each thread throughout its lifetime. When a new thread object is initialized, the J-RAF2 runtime system automatically assigns its `ThreadCPUAccount` object.

```
public final class ThreadCPUAccount {
   public static
      ThreadCPUAccount getCurrentAccount();
   public static
      ThreadCPUAccount getAccount(Thread t);
   public Thread getOwningThread();

   public int consumption;
   public void consume();

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

Figure 2. Part of the `ThreadCPUAccount API`. *The* `getCurrentAccount()` *method returns the calling thread's* `ThreadCPUAccount`, *and* `getAccount(Thread)` *returns an arbitrary thread's* `ThreadCPUAccount`. *The* `getOwningThread()` *method returns the thread associated with a* `ThreadCPUAccount`.

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

Figure 3. Part of the `CPUManager` *interface. The* `ThreadCPUAccount` *implementation communicates with the user-defined resource manager through this interface.*

At load time, the J-RAF2 rewriting tool inserts accounting sequences in the beginning of each basic block of code. During normal execution, each rewritten thread increments its `ThreadCPUAccount`'s `consumption` counter with the number of bytecodes it intends to execute in the immediate future. To schedule regular activation of the shared management tasks, each thread periodically checks the consumption counter against an adjustable limit, called the accounting *granularity*. Each time the counter is incremented by a number of bytecodes equal to the granularity, the thread registers the value and resets it to an initial value by invoking the `consume()` method. Conditionals inserted in the beginning of each method and loop check whether `consume()` needs to be invoked. Further details about this accounting scheme appear elsewhere.[3]

Aggregating CPU Consumption

Normally, each `ThreadCPUAccount` refers to an implementation of `CPUManager` that is shared between all threads belonging to a given *component*, which we define as an informal group of threads subjected to the same `CPUManager` object – and hence, logically (but not necessarily) to the same management policy.

Figure 3 shows part of the `CPUManager` interface. The middleware developer provides the `CPUManager` implementation, which implements the actual CPU-accounting and -control strategies. As illustrated in Figure 4, the `ThreadCPUAccount` implementation invokes the `CPUManager` interface's methods. For efficiency, each thread performs the incrementing directly on the `consumption` variable. Because they must perfectly match the characteristics of bytecode rewritten by our tool, the per-thread accounting objects aren't user-extensible.

For resource-aware applications, the `CPUManager` implementation can provide application-specific interfaces for accessing information concerning components' CPU consumption, installing notification callbacks to be triggered when the resource consumption reaches a certain threshold, or when modifying resource-control strategies. In this article, we focus only on the required `CPUManager` interface.

Whenever a thread invokes `consume()` on its `ThreadCPUAccount`, this method reports its collected CPU consumption data (stored in the `consumption` field) to the associated `CPUManager` (if any) by calling the `consume(long)` method, which also resets the `consumption` field. The `consume(long)` method implements the custom CPU-accounting and -control policy. It can simply aggregate the reported CPU consumption (and write it to a log file or database), enforce absolute limits and terminate components that exceed their CPU limits, or limit the execution rate for a component's threads (that is, temporarily put threads to sleep if a given execution rate is exceeded). Given that the `consume(long)` invocation is synchronous (blocking) and executed directly by the thread to which the policy applies, these actions are possible without breaking security assumptions.

The `getGranularity()` method returns the accounting granularity currently defined for a `ThreadCPUAccount` associated with the given `CPUManager`. This adjustable value defines the management activities' frequency (and, indirectly, their

overhead). To prevent excessive delays between invocations to `consume(long)`, the `CPUManager` implementation must adapt the accounting granularity to the number of threads under supervision.

Changing the CPUManager

The `getManager()` method (see Figure 2) returns the current `CPUManager` associated with a `ThreadCPUAccount`, whereas `setManager(CPUManager)` changes it. If a thread invokes `setManager(CPUManager)` on its own `ThreadCPUAccount`, it will report its CPU consumption to the previous `CPUManager` (if any) before attaching to the new one. If a thread calls `setManager(CPUManager)` on a different `ThreadCPUAccount` than its own, the thread owning that `ThreadCPUAccount` will report all consumption to the new `CPUManager` upon the subsequent invocation of `consume()`. This ensures that only the owning thread accesses its `ThreadCPUAccount`'s consumption variable and therefore lets us implement these accesses without synchronization.

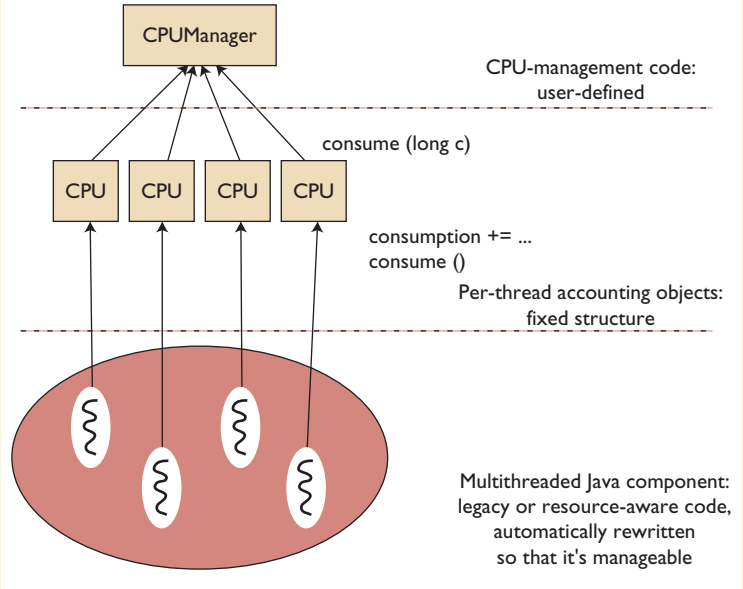

Figure 4. CPU-management hierarchy. Each thread maintains its CPU consumption in the associated per-thread accounting object and periodically reports the accumulated CPU consumption to the shared `CPUManager`.

Providing a CPUManager for Legacy Code

When the JVM is first started, initial threads receive `ThreadCPUAccount` objects without associated `CPUManager` objects. Because user-defined code could depend on arbitrary JDK classes, the J-RAF2 runtime system can load a user-defined `ManagerFactory` (as specified by a system property) only after the startup (or bootstrapping) process's completion. Otherwise, the user-defined code might violate assumptions about the JDK classes' loading sequence and crash the JVM. A user-defined `ManagerFactor` has to implement the following interface:

```
public interface ManagerFactory {
   CPUManager getManager(Thread t);
}
```

If a `ManagerFactory` is specified, all threads collected during the bootstrapping phase are associated with a `CPUManager` provided by the `ManagerFactory`. This approach allows the user to install `CPUManagers` without modifying application classes by hand.

In the simplest case, the user-defined `ManagerFactory` provides a single default `CPUManager`. However, it could also inspect each thread passed as argument and exploit other contextual information available at runtime to help decide which `CPUManager` to assign. This feature can be useful in separating system threads from application threads, for instance.

CPUManager Inheritance

When a new thread *T* is created, it receives its own `ThreadCPUAccount`. If the creating thread has an associated `CPUManager`, *T* will be associated with the same one. Otherwise, the `ManagerFactory` (if installed) will provide an appropriate `CPUManager`.

If a thread is not associated with a `CPUManager`, invocations of `consume()` for its `ThreadCPUAccount` collect the consumption data internally within the `ThreadCPUAccount`. When the thread becomes associated with a `CPUManager`, it reports all the internally collected consumption data to that `CPUManager`. A thread can use `setManager(null)` to disconnect a `ThreadCPUAccount` from any `CPUManager`.

Code Examples

To illustrate the APIs we've described thus far, let's look at some simple `CPUManager` implementations and common use cases. We follow a minimalist approach to design, while ensuring that middleware developers can add new features as needed — possibly in coordination with application developers, as with the more advanced interaction schemes. For instance, several alternative

Related Work in Resource Management

Prevailing approaches to providing resource control in Java-based platforms rely on three main mechanisms:

- modified Java virtual machines,
- native code libraries, and
- program transformations.

For instance, the Aroma VM,[1] KaffeOS,[2] and the Multitasking Virtual Machine (MVM)[3] are specialized JVMs that support resource control.

As an example of the second approach, JRes[4] is a resource-control library for Java. For memory control, it rewrites the bytecode in Java programs; for CPU control, it uses native code with some light bytecode rewriting applied to enable proper cooperation with operating systems. Rather than using bytecode-level accounting, which the authors considered prohibitive in terms of performance, JRes obtains information by polling the OS about threads' CPU consumption. It therefore requires a JVM with OS-level threads.

Researchers at Sun recently published a report on their approach to incorporating resource management as an integral part of the Java language.[5] They've embraced a broad field of investigation, aiming to care for physical as well as logical resources (such as ports and file descriptors) and provide direct support for sophisticated management policies with multiparty decision-making and notification. In contrast, J-RAF2 focuses on lower-level facilities, leaving much flexibility to the application and middleware developer to address higher-level concerns.

One notable aspect of the Sun proposal is that it is based on the Java Isolation API. Isolates provide a very sound base, but the company has yet to release to the general public a JVM that supports them. In turn, this raises questions about the resource-management API's future availability across all environments, including embedded devices, in which programmatic resource-management facilities are already greatly needed.

With successive Java platform releases, Sun has offered several other management and runtime-monitoring APIs, especially for heap memory. Currently, however, no solution is as well integrated with the language, as usable for implementing control policies (as opposed to monitoring), or as applicable across as many environments as J-RAF2.

The J-RAF2 proposal for CPU management is built on the idea of self-accounting. It thus offers what we believe to be the most precise, fine-grained accounting basis available. An important weakness in all existing solutions with polling supervisor threads is that the Java specification doesn't formally guarantee that the supervisor thread will ever be scheduled, regardless of its priority settings. Our approach solves this problem by ensuring that the consuming threads themselves account for any resources they consume (provided that the consuming code is implemented in Java rather than some native language); if required, the threads eventually take self-correcting measures.

References

1. N. Suri et al., "NOMADS: Toward a Strong and Safe Mobile Agent System," *Proc. 4th Int'l Conf. Autonomous Agents* (AGENTS '00), C. Sierra, G. Maria, and J.S. Rosenschein, eds., ACM Press, 2000, pp. 163–164.
2. G. Back, W. Hsieh, and J. Lepreau, "Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java," *Proc. 4th Symp. Operating Systems Design and Implementation* (OSDI '00), Usenix Assoc., 2000.
3. G. Czajkowski and L. Daynès, "Multitasking without Compromise: A Virtual Machine Evolution," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '01), ACM Press, 2001, pp. 125–138.
4. G. Czajkowski and T. von Eicken, "JRes: A Resource Accounting Interface for Java," *Proc. 13th Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '98), ACM Press, 1998, pp. 21–35.
5. G. Czajkowski et al., *A Resource Management Interface for the Java Platform*, tech. report TR-2003-124, Sun Microsystems, May 2003; http://research.sun.com/techrep/2003/abstract-124.html.

measures are possible for handling threads that consume too much CPU power.

Depending on the supervised code's implementation and the seriousness of the misuse, we can choose between slightly delaying the thread or throttling it firmly by repeatedly invoking the Java `sleep()` method, perhaps changing the thread's priority as well. Alternatively, we can terminate the whole isolate by invoking `halt()` or kill the thread by throwing an exception, such as `ThreadDeath`. To prevent the client code from catching `ThreadDeath`, we can rewrite the bytecode for each exception handler that could catch this exception or one of its supertypes, thus ensuring that instances of `ThreadDeath` are immediately rethrown. (This approach was already implemented in the J-SEAL2 mobile object kernel.[4])

If the guilty thread accepts a synchronous callback (which the middleware and application developers must agree on beforehand), finer, less disruptive protocols are possible. For example, we can send a warning, thus allowing the thread to change its behavior by adopting an algorithm that trades off CPU cycles against more memory or network bandwidth (for example, by avoiding CPU-intensive compression), or we can send a notification to guarantee that the supervised thread properly releases all resources (heap memory, locks, worker threads, and so on).

These alternatives are all possible implementations of the `CPUManager` interface. Several inheritance hierarchies are also feasible in which the less

intrusive management schemes are gradually extended with more drastic ones.

CPUManager Implementations

Figures 5 and 6 show simplified examples of how to aggregate and use the accounting information from multiple threads. Both `CPUAccounting` and `CPUControl` implement `CPUManager` and provide specific implementations of the `consume(long)` method. `CPUAccounting` supports dynamic adaptation of accounting granularity. The `granularity` variable is `volatile` in order to ensure that the `consume()` method of the `ThreadCPUAccount` always reads the up-to-date value. (A `volatile` declaration forces the JVM to immediately propagate every update from a thread's working memory to the master copy in the main memory.[1,2])

Note that the `consume(long)` method is synchronized, such that multiple threads can invoke it concurrently. The `CPUAccounting` implementation simply maintains the sum of all reported consumption information, whereas the `CPUControl` implementation enforces a strict limit and terminates a component when its threads exceed that limit. In this example, we assume that the component whose CPU consumption is limited executes within a separate isolate. This is a notional example, however, as the isolation API[5] is missing from current standard JVMs. More sophisticated scheduling strategies could, for instance, delay thread execution when execution rates exceed a given threshold. However, we must take care to prevent deadlocks and priority inversions.

Czajkowski and colleagues' recent proposal for a resource-management API for Java[6] supports user-defined notification methods, which can be invoked upon certain triggering conditions. Notifications are a convenient way to program resource-aware applications. In J-RAF2, it's up to the middleware programmer to provide appropriate notification mechanisms in the `CPUManager` implementation.

Installing CPUManagers

Middleware developers can use the aforementioned APIs in several different ways to attach `CPUManager` implementations to applications.

Gathering execution statistics. To gather an application's CPU consumption (as the number of executed bytecode instructions) without modifying it by hand, the developer can install a custom `ManagerFactory` using a system property. Figure 7 (next page) shows a simple `ManagerFactory` that will associate all threads with a single `CPUManager` instance.

```
public class CPUAccounting implements CPUManager {
   public static final int DEFAULT_GRAN =...;
   protected volatile int granularity;
   protected long consumption = 0;

   public CPUAccounting() {this(DEFAULT_GRAN);}
   public CPUAccounting(int g) {granularity = g;}

   public int getGranularity() {
      return granularity;
   }

   public void setGranularity(int g) {
      granularity = g;
   }

   public synchronized long getConsumption() {
      return consumption;
   }

   public synchronized void consume(long c) {
      consumption += c;
   }
   ...
}
```

Figure 5. Example `CPUManager` implementation: CPU accounting without control. The `consume(long)` method aggregates the reported CPU consumption in a synchronized way.

```
public class CPUControl extends CPUAccounting {
   private Isolate isolate;
   private long limit;

   public CPUControl(int g, Isolate i, long l) {
      super(g);
      isolate = i;
      limit = l;
   }

   public synchronized void consume(long c) {
      super.consume(c);
      if (consumption > limit) isolate.halt();
   }
}
```

Figure 6. Example `CPUManager` implementation: CPU control. The `consume(long)` method terminates an isolate if its threads exceed a given CPU limit.

```
public class DefaultManagerFactory
       implements ManagerFactory {
   private final CPUManager
      defaultManager = new CPUAccounting();

   public CPUManager getManager(Thread t) {
      return defaultManager;
   }
}
```

Figure 7. Example `ManagerFactory` *implementation. All threads are associated with the same* `CPUManager`.

```
public class SystemApplicationManagerFactory
       implements ManagerFactory {
   private final CPUManager
      applicationManager = new CPUAccounting(),
      systemManager = new CPUAccounting();

   public CPUManager getManager(Thread t) {
      ThreadGroup tg = t.getThreadGroup();
      return (tg == null ||
              tg.getParent() == null) ?
             systemManager :
             applicationManager;
   }
}
```

Figure 8. Example `ManagerFactory` *implementation. System threads and application threads use different* `CPUManagers`.

Separating application from system threads. To maintain two different `CPUManagers` — one for system threads and one for application threads — the developer can provide a `ManagerFactory` like the one shown in Figure 8. In this example, we assume that system threads lack parent `ThreadGroups`.[7] While this test might not be appropriate for all Java runtime systems, it works well with many current JVMs.

Alternatively, we can achieve a clean separation of application and system threads with the following approach. Initially, a `ManagerFactory` attaches a system `CPUManager` *S* with each thread after bootstrapping. When the main application thread invokes the `main(String[])` method, this thread creates a second `CPUManager` *A*. `ThreadCPUAccount.getCurrentAccount().setManager(`*A*`)` detaches the main application thread from *S* and attaches it to *A*. Thereafter, all new application threads inherit `CPUManager` *A*, whereas spawned system threads inherit *S*. Thus, *S* will be in charge of all system threads, and *A* will handle all application threads.

Asynchronous execution of a component with its own CPUManager. The example in Figure 9 shows how a new thread executes a component (implementing the `Runnable` interface) asynchronously. Before the thread is started, the middleware sets the `CPUManager` responsible for the component. All subsequent threads created by the component inherit the same `CPUManager`.

Synchronous execution of a component with its own CPUManager. The example in Figure 10 illustrates how a component is executed synchronously. The middleware programmer can use this approach to implement thread pooling. Before the component is executed synchronously (by invoking its `run()` method), the current `CPUManager` is saved and the calling thread switches to the `CPUManager` in charge of the component. After executing the component, the thread restores the previous `CPUManager`. As the calling thread changes the `CPUManager` of its own `ThreadCPUAccount`, all previous CPU consumption is accurately reported to the previous `CPUManager`.

Hierarchical scheduling. For a hierarchy of components, a hierarchical scheduling model inspired by the Fluke kernel's CPU-inheritance scheduling[8] fits very well. In this model, a parent component donates a certain percentage of its own CPU resources to a child. Initially, the root of the hierarchy possesses all CPU resources. The J-SEAL2 kernel employed a similar model for CPU management.[9]

The easiest and most effective way to implement such a scheduling strategy with J-RAF2 is to simulate it with a single, centralized `CPUManager` for all application threads and to represent the hierarchy internally with appropriate data structures. When a thread invokes the `consume(long)` method, the `CPUManager` looks up the component the thread belongs to, registers the reported consumption on behalf of its component, and eventually delays the thread if the corresponding component exceeds its CPU share.

If the components are isolated from each other (for example, encapsulated by different isolates), the `CPUManager` can't be shared among them. In

this case, each component needs its own `CPUManager`. All `CPUManagers` in the system must communicate using a general communication mechanism (such as sockets or remote method invocation) or inter-isolate links, which are part of the Isolation API,[5] to maintain an (approximate) global view of CPU consumption in the system, which is necessary for computing a component's actual CPU share.

Case Studies

To illustrate how J-RAF2 works in practice, we briefly summarize two recent cases in which we successfully applied it to enhance existing middleware with CPU-management features.

Accounting in an Application Server

In addition to the ubiquitous issues of security and reliability, high availability and profitability are vital considerations with any e-commerce infrastructure. To investigate resource accounting in support of billing strategies, we applied our framework to the Apache Tomcat servlet engine (http://jakarta.apache.org/tomcat/) to monitor CPU and network-bandwidth consumption on a per-request basis and report this data in real time to a database server.

In this environment, we faced two main challenges:

- assigning semantic, real-world meaning to the unstructured, low-level accounting information gathered; and
- coping with the fact that Tomcat uses thread pooling to execute HTTP requests.

Further complications arose from the fact that servlets can freely spawn worker threads that the Tomcat engine doesn't notice, thus creating additional resource consumption that we must record. Because the source code is not always available for modification, we chose to consider servlets as legacy code. We took advantage of Tomcat's open-source design to extract semantic associations between HTTP requests and the identities of the main threads elaborating the corresponding replies. Exploiting J-RAF2's manager-inheritance mechanism, we then let each worker thread inherit its main request thread's dedicated `CPUManager` object. This combination lets us integrate worker threads' consumption, and thus achieve a fairly complete and straightforward solution to the mentioned challenges.

```
void executeAsynchronously(Runnable component,
                           CPUManager m) {
   Thread t = new Thread(component);
   ThreadCPUAccount.getAccount(t).setManager(m);
   t.start();
}
```

Figure 9. Asynchronous execution of a component with its own `CPUManager`. *All threads spawned by the component will inherit the same* `CPUManager`.

```
void executeSynchronously(Runnable component,
                          CPUManager m) {
   ThreadCPUAccount cpu =
      ThreadCPUAccount.getCurrentAccount();
   CPUManager middlewareManager =
      cpu.getManager();
   cpu.setManager(m);
   try {
      component.run();
   }
   finally {
      cpu.setManager(middlewareManager);
   }
}
```

Figure 10. Synchronous execution of a component with its own `CPUManager`. *The thread switches between different* `CPUManagers`.

Absolute CPU Limits in Extensible Directories

Constantinescu and colleagues' Hotblu directory offers specific features that facilitate efficient (semantic) Web service composition, taking input and output messages' type constraints into account.[10,11] Service-composition algorithms access the directory to retrieve descriptions of Web services that can be combined to fulfill given requirements. Because the number of relevant services for a particular service-composition problem can be very large, the directory allows for incremental result retrieval. The service-composition algorithm's performance depends very much on the order in which the directory returns (partially) matching results. Because research on service composition is still in its early stages and requires much experimentation to develop industrial-strength algorithms, the directory must be flexible enough to support various ordering heuristics for different service-composition algorithms.

Binder and colleagues later extended the directory to support user-defined pruning and ranking functions, which enable the dynamic installation of application-specific heuristics directly within the directory.[12] Like the directory, the custom functions are written in Java. To protect against erroneous or malicious client code, the directory imposes severe restrictions on user-defined pruning and ranking functions. For instance, the client code

- can use only a very limited API;
- is not allowed to allocate memory on the heap;
- must not use synchronization primitives; and
- can't define exception handlers.

The directory enforces these restrictions at load time and partly at runtime, ensuring that the user-defined code can't interfere with the directory's internals and cause unwanted side effects. To prevent denial-of-service attacks, an early version of the extensible directory even required client code to be acyclic — that is, it disallowed loops.

We've recently used J-RAF2 to overcome this limitation by rewriting the custom pruning and ranking functions: their CPU consumption is now limited by a CPU control policy defined by the directory provider. Query execution 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 each query's CPU consumption. If the limit is reached, the `consume(long)` method throws an exception that aborts execution of the user code (remember that custom functions aren't allowed to define exception handlers). The directory then catches the exception and terminates the query.

In this setting, the runtime overhead for CPU accounting is negligible, as we rewrite only the untrusted, user-defined code for CPU accounting. The directory itself is not rewritten, and J-RAF2 doesn't account for its execution. Because service-composition clients will likely use the same set of functions for multiple queries, the directory keeps a cache of recently used pruning and ranking functions (verified, rewritten, and loaded), thus mitigating the overhead for dynamic bytecode rewriting.

Performance Measurements

To evaluate the runtime overhead our CPU management scheme induces, we ran the Standard Performance Evaluation Corporation's (SPEC) JVM98 benchmark suite (www.spec.org/osg/jvm98/) on top of a rewritten JDK. In our test, we used a single `CPUManager` with the most basic accounting policy (shown in Figure 5) and the highest possible granularity. The resulting average performance slowdown (measured on various Java 2, Standard Edition distributions) amounts to about 20 percent. More detailed information about the performance results appears elsewhere.[3]

We're constantly working to improve our rewriting schemes to reduce overhead. Note, however, that these results correspond to a perfectly accurate accounting of executed bytecode instructions — a level of precision that's not always necessary in practice. We're currently working on a complete optimization framework that will let J-RAF2 users tune accounting precision, thus decreasing the consequent overhead.

Conclusions

Because it is independent of any particular JVM or underlying operating system, program-transformation-based resource control offers important advantages over existing approaches. It works with standard Java runtime systems and can be integrated into many existing Internet applications involving server or mobile-object environments, as well as embedded systems based on Java processors.

Among our forthcoming investigations, one exciting test will be to explore the extent to which our hypotheses remain valid across other virtual machines, the most obvious challenger being the .NET platform.

Our approach's major limitation is that it can't directly account for native code execution. We believe it will take a range of solutions to solve this problem, especially concerning memory attacks.[9] Certain native functions, like (de-)serialization and class loading, can be protected with wrapper libraries, which inspect the arguments. It is also possible to run a calibration process, once per platform, to evaluate the actual consumption of certain categories of native system calls, such as those for which we can safely estimate a constant or linear execution time. As a higher-level measure, we can restrict untrusted applications' access to such native system calls. While these are some answers to the issue of native methods, we must still take into account that different bytecodes have different execution costs. To that end, we also propose a calibration process to collect cost information to feed into the rewriting tool.

Another difficulty is that some JVMs won't let certain core classes be rewritten — the JVM could

crash if such classes are modified. Others simply use internal, hardwired implementations and disregard the corresponding bytecode-level representations. The `Object` class is one such case, and indeed, J-RAF2 doesn't rewrite it. We also give special treatment to the `Thread` class to implement the `CPUManager` inheritance mechanism.

We haven't addressed security in this article, but we do have load-time and runtime verification algorithms that are designed to prevent applications from tampering with their own CPU-consumption accounts, whether directly or indirectly by reflection.

Our previous work on J-RAF2 has shown that we can account for other basic resources, such as heap memory and network bandwidth, within a single homogeneous conceptual and technical framework. More research is needed to advance these resources to the same level of maturity as the CPU-management framework presented here.

Acknowledgments

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

References

1. J. Gosling et al., *The Java Language Specification*, 2nd ed., Addison-Wesley, 2000.
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed., Addison-Wesley, 1999.
3. J. Hulaas and W. Binder, "Program Transformations for Portable CPU Accounting and Control in Java," *Proc. ACM SIGPLAN Symp. Partial Evaluation & Program Manipulation* (PEPM '04), ACM Press, 2004, pp. 169–177.
4. W. Binder, "Design and Implementation of the J-SEAL2 Mobile Agent Kernel," *Proc. Symp. Applications and the Internet* (SAINT '01), IEEE CS Press, 2001, pp. 35–42.
5. *Java Specification Request 121: Application Isolation API Specification*, Java Community Process, work in progress; www.jcp.org/jsr/detail/121.jsp.
6. G. Czajkowski et al., *A Resource Management Interface for the Java Platform*, tech. report TR-2003-124, Sun Microsystems, May 2003; http://research.sun.com/techrep/2003/abstract-124.html.
7. S. Oaks and H. Wong, *Java Threads*, O'Reilly & Assoc., 1997.
8. B. Ford and S. Susarla, "CPU Inheritance Scheduling," *Proc. Usenix Assoc. 2nd Symp. Operating Systems Design and Implementation* (OSDI), Usenix Assoc., 1996, pp. 91–105.
9. W. Binder et al., "Portable Resource Control in Java: The J-SEAL2 Approach," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '01), ACM Press, 2001, pp.139–155.
10. I. Constantinescu, W. Binder, and B. Faltings, "Directory Services for Incremental Service Integration," *Proc. 1st European Semantic Web Symp.* (ESWS '04), LNCS 3053, Springer-Verlag, 2004.
11. I. Constantinescu, B. Faltings, and W. Binder, "Large-Scale, Type-Compatible Service Composition," *Proc. IEEE Int'l Conf. Web Services* (ICWS '04), IEEE CS Press, 2004.
12. W. Binder, I. Constantinescu, and B. Faltings, "A Directory for Web Service Integration Supporting Custom Query Pruning and Ranking," *Proc. European Conf. Web Services* (ECOWS '04), LNCS 3250, Springer-Verlag, 2004.

Walter Binder is a senior researcher at the Artificial Intelligence Laboratory of the Swiss Federal Institute of Technology, Lausanne. His research interests include program transformations, object-oriented systems, mobile code, directory services, and service composition. He received a PhD in computer science from the Vienna University of Technology. He is a member of the ACM and the IEEE. Contact him at walter.binder@epfl.ch.

Jarle Hulaas is a senior researcher at the Software Engineering Laboratory of the Swiss Federal Institute of Technology, Lausanne, where he leads the software engineering group. His research interests include the architectures, implementation paradigms, and security of distributed systems. He received a PhD in computer science from the Swiss Federal Institute of Technology, Lausanne. He is a member of the ACM and the IEEE. Contact him at jarle.hulaas@epfl.ch.