In Proceedings of the 1998 ACM OOPSLA Conference, Vancouver, BC, October 1998.

JRes: A Resource Accounting Interface for Java

Grzegorz Czajkowski and **Thorsten von Eicken**
Department of Computer Science
Cornell University
Ithaca, NY 14853 USA
+1 607 255 5951
{grzes,tve}@cs.cornell.edu

ABSTRACT

With the spread of the Internet the computing model on server systems is undergoing several important changes. Recent research ideas concerning dynamic operating system extensibility are finding their way into the commercial domain, resulting in designs of extensible databases and Web servers. In addition, both ordinary users and service providers must deal with untrusted downloadable executable code of unknown origin and intentions.

Across the board, Java has emerged as the language of choice for Internet-oriented software. We argue that, in order to realize its full potential in applications dealing with untrusted code, Java needs a flexible resource accounting interface. The design and prototype implementation of such an interface — JRes — is presented in this paper. The interface allows to account for heap memory, CPU time, and network resources consumed by individual threads or groups of threads. JRes allows limits to be set on resources available to threads and it can invoke callbacks when these limits are exceeded. The JRes prototype described in this paper is implemented on top of standard Java virtual machines and requires only a small amount of native code.

Keywords

Java, resource management, extensible systems.

INTRODUCTION

The spreading use of active content on the Internet makes the use of untrusted code a fact of life for connected users as well as for providers of Internet-based services. On the client side, downloadable executable content, like Java applets [2] and Active X technology [11], serves to increase the attractiveness of Web pages. On the server side, dynamic server extensibility promises full customizability of various tasks, from information retrieval to accessing proprietary databases to dynamically uploading services. These tasks can be accomplished via a concept of *servlets* [23] — untrusted code executed on the server. The programming language Java is currently a premier tool for implementing such uploadable content.

Despite the attention and publicity given to Java, most notably to its security features, there are still areas where Java either cannot be applied or applying the language is cumbersome and requires non-portable native code support. The language has been extended in various ways and its performance has been continuously improving. However, the current definition of Java (version 1.2) does not have an interface for resource management. The application domains that would benefit from providing such an interface as a standard and integral part of the language specification include both well known paradigms, such as application-controlled load balancing and preventing some denial-of-service attacks, as well as emerging concepts, such as extensible servers.

Not addressing the issue of resource management may in fact be an adequate decision when designing a "plain" programming language. This is not so in the case of Java, which is both a language and a runtime environment and includes many features of an extensible operating system (a Java-enabled browser is in essence an operating system running applets). Server-side environments running untrusted code are being developed and may well become common [20,23]. The fact that Java provides neither a model nor mechanisms for controlling resource consumption of applications (apart from changing thread priorities) is a problem in all these systems for two reasons: malicious code cannot be prevented from using too many resources and code cannot be charged for the resources it is consuming. Both issues represent a significant obstacle to deploying commercial extensible servers.

Some ad-hoc resource controlling solutions are possible in Java. For example, using native code it is possible to monitor CPU time consumed by each thread. Such approaches tie an otherwise portable Java environment to an underlying operating system, which results in loss of portability and maintenance problems. Enforcing appropriate resource consumption patterns is also possible for applications where one has full control over the sources and can "make the code behave". However, in the case of untrusted and unknown code, like applets and servlets, no assumptions can be made about resource consumption.

In this paper we propose JRes — a Java interface which allows per-thread accounting of heap memory, CPU time, and the number of bytes sent and received. In addition to tracking resource consumption, applications can be informed when new threads are created. JRes provides mechanisms for setting limits on the resources available to particular threads and associate *overuse callbacks* to be invoked whenever any such resource limit is exceeded. Although the interface is simple, it is flexible enough to support a variety of resource consumption enforcement policies. For instance, a policy which ensures that no thread gets more than 100 milliseconds of CPU time out of every second is easily expressible, as will be shown later. Similarly, a few lines of code suffice to create a policy in which no thread group can send more than 2MB of data, or a policy in which the combined size of all alive objects allocated by threads belonging to a particular thread group does not exceed 1MB.

Although the design of JRes is general and the system can be used for any Java application, its main intended usage is for extensible environments, like Web browsers or extensible Internet servers. Such environments would use JRes to control resources consumed by their extensions.

The contribution of this paper is twofold. First, it motivates considering incorporating a resource management interface in the language specification of Java and any other language of similar aspirations and scope. By describing several possible applications of JRes we aim at making a convincing argument that future extensions of Java with resource management support should provide at least the functionality included in JRes. The second goal is to demonstrate that although it was possible to implement JRes in its current form without modifying the underlying JVM, it comes at a price of certain functionality limitations and performance overheads. The lessons learned in the course of developing JRes can be important guidelines in future work on resource management infrastructure for Java, possibly through changing the implementation of JVMs. Although the JRes project is carried out in the context of Java, it is applicable to other languages with features similar to Java. To date we are not aware of any other work providing the functionality of JRes in a general purpose programming language.

The rest of the paper is structured as follows. The next section motivates extending Java with resource management capabilities. This is followed by a description of the JRes interface and a presentation of selected implementation and performance details. The JRes limitations are discussed next. The following two sections discuss related work, the current status of the system and planned future directions. The last section summarizes the paper.

MOTIVATING APPLICATIONS

Several motivating examples are described below in order to clarify why it is crucial that Java (and any other language of similar scope) provide a resource management interface. Such an interface should at least allow the resources consumed by applications (be it applets or servlets) to be monitored and limits on the consumption of particular resources to be enforced.

The first group of applications that can readily benefit from JRes are extensible, Java-based Internet servers, which allow service extensions to be uploaded by clients to the server. Consider image processing services created for profit by an independent developer. An architecture of an extensible Web server makes it possible for the developer to upload the code of the service. This makes the service available to a large group of potential users. Clients access the services either via a standard Web browser interface or, in case of more sophisticated or special demands, are able to upload their own code accessing the image-rendering services.

The J-Server, a practical application of the J-Kernel, is an example of a dynamically extensible Web server [20], adequate for implementing such scenarios. Both the system core and the uploaded code execute in a single Java Virtual Machine (JVM). Constructing a server as a single JVM has two considerable advantages. First, the performance of cross-domain calls between the core and extensions is much better than when relying on traditional OS protection mechanisms (e.g. separate processes encapsulating extensions and the core). Second, the security restrictions under which bytecode is run can be finely controlled and Java provides adequate mechanisms for extensibility and implementing protection domains [20,15].

While the issues of dynamic extensibility and protection are well understood in the context of Java, the lack of support for resource accounting has two serious consequences. First, in a single-address space environment described above, it is difficult to identify servlets that consume too many resources. Malicious servlets can easily mount denial-of-service attacks, that is, attacks aimed at preventing other applications from being able to use resources. Second, one of the main incentives behind attractive, high quality, extensible Internet servers is the revenue they may potentially bring. Lack of support for resource accounting leaves only one alternative for billing users: flat, undifferentiated fees. Although in principle flat rates are not bad, typically there needs to be several levels of them and a way to force clients to use up only as much resources as purchased.

The second motivating example is the technology of extensible database servers. The functionality of such systems can be augmented by user-defined functions

(UDFs) [14]. For example, consider a database of stock market data that is accessible through a Web site. A potential user is any investor with a Web browser, a credit card, and an investment formula `InvestVal`. The following SQL query would then find technology stocks of interest to the user:

```
SELECT *
FROM Stocks S
WHERE S.type = “tech” and
      InvestVal(S.history) > 5
```

Here, `InvestVal` is a user-defined function. It should be possible (and relatively straightforward) for a large number of such users in a Web environment to create their own UDFs and use them within queries. The motivation for facilitating such database extensions is the next generation of database systems that will be deployed over the Web [14]. In such applications, a large number of physically distributed end-users working on diverse platforms interact with the database through their Web browsers. Because of the large size and diversity of the user community, the utility of UDFs increases. Java is ideally suited for the implementation of such extensible databases because of its security and portability features as long as the resource management concerns are resolved.

Active Networks [7] is yet another example of an extensible environment for which portability and safety features of Java are desirable. The goal of Active Networks is to associate executable content with network packets and execute such programs on hosts and routers of the Internet. This can result in easier management of networks. However, using standard Java-based environments as an infrastructure for implementing Active Networks is possible only when the programs are trusted because malicious network programs can simply consume enough resources to halt routers, for instance. Extending Java with resource management facilities is very desirable in this context.

The final motivation for JRes is preventing denial-of-service attacks aimed at Web browsers. A typical form of a denial-of-service attack mounted by an applet is when the applet is consuming excessive amounts of resources and thereby prevents other applications on the local host from performing as expected. In severe cases, a hostile applet can completely take over a particular resource and crash other applications as well as prevent new applications from being started. It is important to point out that applets may hog resources not only because of malicious intentions, but also because of programming errors. Although denial-of-service attacks are classified as proper security attacks, to date this issue has not been addressed by Java implementations.

THE PROPOSED INTERFACE

The purpose of JRes is to add resource accounting and limiting facilities to Java. The main motivation is the creation of portable and extensible Web environments. This motivation is reflected in the design of JRes. One part of the interface is accessible to both trusted and untrusted code; this part of JRes allows extensions to learn about resource limits and usage. The purpose of the other part of JRes, requiring authenticating the caller as a trusted system component, is to manage and enforce resource limits. This part is designed to be used only by privileged system modules, e.g. by a resource controlling part of an enhanced Web browser.

Through JRes, the trusted core of Java-based environments can (i) be informed of all new thread creations, (ii) state an upper limit on memory used by all live objects allocated by a particular thread or thread group, (iii) limit how many bytes of data a thread can send and receive, (iv) limit how much CPU time a thread can consume, and (v) register *overuse callbacks*, that is, actions to be executed whenever any of the limits is exceeded.

The JRes interface is presented in Figure 1. It consists of two Java interfaces, one exception and the class `ResourceManager.` Except for `initialize()`, all presented methods of `ResourceManager` deal with threads and each of them has a counterpart method (not shown) managing resources on a per thread group basis. Despite the fact that the interface is simple, it is flexible enough to build rather complex and diverse resource management policies, as will be discussed later.

The `ResourceManager` class defines constants identifying resources and exports six public methods. The first one, `initialize(cookie)`, handles an authenticating object (a *cookie*; it can be any object) to the resource accounting subsystem and initializes it. The purpose of the cookie is to ensure that only one party can have privileged access to the resource management subsystem. This prevents untrusted code from interfering with the resource management policies of a given system. For instance, a browser would call `initialize(cookie)`, before any applets are loaded. Since applets cannot get hold of `cookie`, they cannot call certain method of `ResourceManager`. For instance, applets could not be able to grant themselves more resources than granted by the browser.

The next method presented in Figure 1, `setThreadRegistrationCallback(cookie, tCallback)`, hands an object implementing the `ThreadRegistrationCallback` interface to the resource management subsystem. The effect of this operation is that whenever a new Java thread `t` is created, `tCallback.threadRegistrationNotification(t)` will be invoked. This callback is meant to work in conjunction with the method `setLimit(cookie,`

```
public interface ThreadRegistrationCallback {
  public void threadRegistrationNotification(Thread t);
}

public interface OveruseCallback {
  public void resourceUseExceeded(int resType, Thread t, int resValue);
}

public class ResourceOverusedException extends RuntimeException {
  ResourceOverusedException(int resType, long limitingValue, long usedValue);
}

public final class ResourceManager {

  public static final int RES_CPU;
  public static final int RES_MEM;
  public static final int RES_NET_RECV;
  public static final int RES_NET_SEND;

  public static boolean initialize(Object cookie);

  public static boolean setThreadRegistrationCallback(Object cookie,
                                     ThreadRegistationCallback tCallback);

  public static boolean setLimit(Object cookie, int resType,
                         Thread t, long limit, OveruseCallback oCallback);
  public static boolean clearLimit(Object cookie, int resType, Thread t);

  public static long getResourceUsage(int resType, Thread t);
  public static long getResourceLimit(int resType, Thread t);
}
```

Figure 1. The JRes interface.

`resType, t, limit, oCallback`), which can be invoked each time a new thread creation is detected and has the effect of setting a limit of `limit` units of a resource `resType` for a thread `t`. Whenever this limit is exceeded by the thread `t`, the method `resourceUseExceeded(resType, t, resValue)` will be invoked on the object `oCallback` (note that `oCallback` must implement the `OveruseCallback` interface). The parameter `resValue` passed to the callback provides information about current resource consumption. Resource limits are cleared by invoking `clearLimit(cookie, resType, t)`.

The method `getResourceUsage(resType, t)` queries the resource management subsystem about resource usage of a particular thread `t`; `getResourceLimit(resType, t)` can be used to query resource limits. No caller authentication via a cookie is necessary in order to invoke "get" methods, which makes it possible for untrusted code to monitor its resource consumption and limits.

A particular resource management policy may choose to throw the exception `ResourceOveruseException` as an action taken against applications using too many resources. The arguments to the constructor specify the type of resource, the allowed value and the value actually used. This information can be retrieved from a caught exception.

The values in which resource limits are expressed and usage values are reported are as follows: memory and network are expressed in bytes and CPU time in milliseconds.

JRes allows for exact *pre-accounting* for memory and network resources — that is, an overuse callback is generated before a thread can execute an action leading to exceeding a limit. This is especially desirable when dealing with memory, since once it is allocated it cannot be reclaimed unless the offending thread is killed and there are no other references to the new objects. However, in the current implementation enforcing CPU limits is done by a periodic polling thread, which results in a certain small and configurable delay in "reaction time".

```
public class Example {
  public static void main(String[] args) {
    Object cookie = new Object();
    ResourceManager.initialize(cookie);
    ResPolicy erp = new ResPolicy(cookie);
    ResourceManager.setThreadRegistrationCallback(cookie, erp);
    // Start the computation proper …
  }
}

class ResPolicy
  implements ThreadRegistrationCallback, OveruseCallback {

  private Object ck;
  ResPolicy(Object cookie) { ck = cookie; }

 public void threadRegistrationNotification(Thread t) {

    ResourceManager.setLimits(ck, Resources.RES_MEM, t, 3000000, this);
    ResourceManager.setLimits(ck, Resources.RES_CPU, t, 100, this);

    if (!paidForNetwork(t)) {
      ResourceManager.setLimits(ck, RESOURCE_NET_SEND, t, 0, this);
      ResourceManager.setLimits(ck, RESOURCE_NET_RECV, t, 0, this);
    }
  }

  public void resourceUseExceeded(int resType, Thread t, long value) {
     switch (resType) {
       case RESOURCE_MEM:
         terminateThread(t);
         break;
       case RESOURCE_CPU:
        if (intervalSinceLastCallback(t) < 1000) {
          if (t instanceof EarlyWarning) {
             ((EarlyWarning) t).cpuExceeded(value);
          }
          lowerPriority(t);
         }
         ResourceManager.setLimits(ck, resType, t, value + 100, this);
         break;
       case RESOURCE_NET_SEND:
       case RESOURCE_NET_RECV:
         String msg =
            “This thread has not paid for network resources”;
         throw new ResourceOverusedException(msg);
    }
}
```

Initialization. The cookie will be used later for authentication.

A new instance of an object managing resources.

The object `erp` will be notified whenever a new thread is created.

Invoked by JRes whenever a new thread is created.

Set a limit of almost 3MB of heap memory available to this new thread `t`

Set a limit of 100ms of CPU time.

This method (not defined here) returns true only if using the network by the thread has been paid for.

Prevent the thread from using the network.

Invoked by JRes whenever a resource limit is exceeded.

Terminate threads attempting to use more memory than allowed.

If less than 1000ms have passed since the last CPU overuse callback on `t`, take an appropriate action.

If a certain, specific to the system, interface is defined, warn the thread that is uses too much CPU.

Lower offending thread’s priority.

Add another 100ms to the thread’s CPU limit.

If the offending thread is trying to use the network, an appropriate JRes exception is thrown.

Figure 2. Example resource management policy using the JRes interface.

Example Use of JRes

Figure 2 shows a resource consumption limiting policy taking advantage of the proposed interface. The code can be thought of as a resource management module of an extensible server or a Web browser. After being initialized, the code is informed of every thread creation and sets resource limits for newly created threads. Whenever any limit is exceeded, `resourceUseExceeded()` is called and enforces that a thread cannot have more than 3MB of memory allocated to objects this thread created, at any given point of time; it can use only 100 ms of CPU time out of every wall-clock second; and can use network resources only if thread's owner paid for this. The actual resource policies for real servers will naturally be more complex and will most likely operate on thread groups, but will use the same mechanisms as in this simplified example.

The first four lines of `main()` install a custom resource management policy. Specifically, an object of type `ResPolicy` is registered with the resource management subsystem. This object will be receiving the upcalls generated by new thread creations, handled by the method `threadRegistrationNotification()` and the upcalls caused by exceeding resource limits, handled by the method `resourceUseExceeded()`.

Consider the method `resourceUseExceeded()`. If any thread exceeds heap memory limits (the first `case` statement) a method is invoked to terminate the offending thread in an application-specific way. If any thread attempts to send or receive data without having "paid" for being able to use the network (the last `case` statement) an exception is being generated. The notion of "paying" is not defined here and is very system-specific; in our example, a thread `t` has not "paid" for network resources if a call to a method `paidForNetwork(t)` returns `false`. Regardless of the caller's catching this exception or not, the attempt to use the network is unsuccessful.

JRes can be used to implement periodic limits on resources, as demonstrated in handling exceeding a limit on CPU (the second `case` statement). The code detects threads that consume more than 100 milliseconds of CPU time during one-second periods. Initially, each new thread is granted 100ms of CPU time. Each time a thread exceeds its current CPU time limit, the check is made over what period it happened. If the thread received more than 100 milliseconds of CPU time during a second of a wall-clock, the example policy decides to lower the priority of the thread. Regardless whether the thread exceeds the limit or not, the new limit is set, equal to the current usage plus 100ms. This effectively implements the policy that every thread that receives more that 100 milliseconds of CPU time during a second of a wall-clock time is detected and lowered in priority.

In this particular extensible server, some threads may implement an interface `EarlyWarning`. If the CPU limit has been exceeded by such a thread, the cpu`Exceeded()` method of this interface is invoked and the thread is informed of the overuse. "Smart" threads may choose to respond to such a warning by trying to consume less CPU (if it makes sense) to avoid lowering of their priorities.

JRes and the Execution of Applications

The example from Figure 2 demonstrates how extensible environments may use JRes to control resources available to applications. It is important to stress that applications do not have to be aware of this fact. In other words, any valid Java program is able to run in an environment that uses JRes. However, applications (servlets, applets, database extensions, etc) may be affected if they exceed their resource limits. Therefore, applications can call `getResourceUsage()` or `getResourceLimit()` to learn about their current resource consumption and limits.

As will be described in the next section, the bytecode of applications needs to be rewritten before it can be run with JRes. This is done automatically and, apart from influencing the performance, cannot be detected by rewritten Java programs.

JRes in the Context of the Java Architecture

Figure 3 shows how the notion of resource manager fits into the Java architecture. Providing the JRes functionality is an important enhancement to the language. It can be viewed in the context of two core notions of Java: the class loader and the security manager. The function of the class loader is to load new classes into the system; the security manager controls access to resources such as files and network connections, using class loaders to detect untrusted classes. The notion of resource manager gives Java programs control over computational resources.

IMPLEMENTATION AND PERFORMANCE OF JRES

This section contains a discussion of specific implementation issues and performance implications. This is followed by a presentation of selected performance data. The experimental setup, to which some implementation details and all performance results refer, consists of a 300 MHz Pentium II PC with 128 MB of RAM. Microsoft's Visual J++ version 1.1 was used, with the Java SDK 2.0; this environment uses a just-in-time compiler.

CPU accounting and accounting for memory used by allocated arrays requires native code support, that is, a small set of routines written in C and called from within a Java program as `native` methods. All other features of JRes are accomplished through bytecode rewriting, that is, inserting appropriate bytecode instructions at selected places in original classes in order to maintain information about resource consumption.

Rewriting Java Bytecode for JRes

Java is a good environment for load-time code transformations. Java has a number of properties which

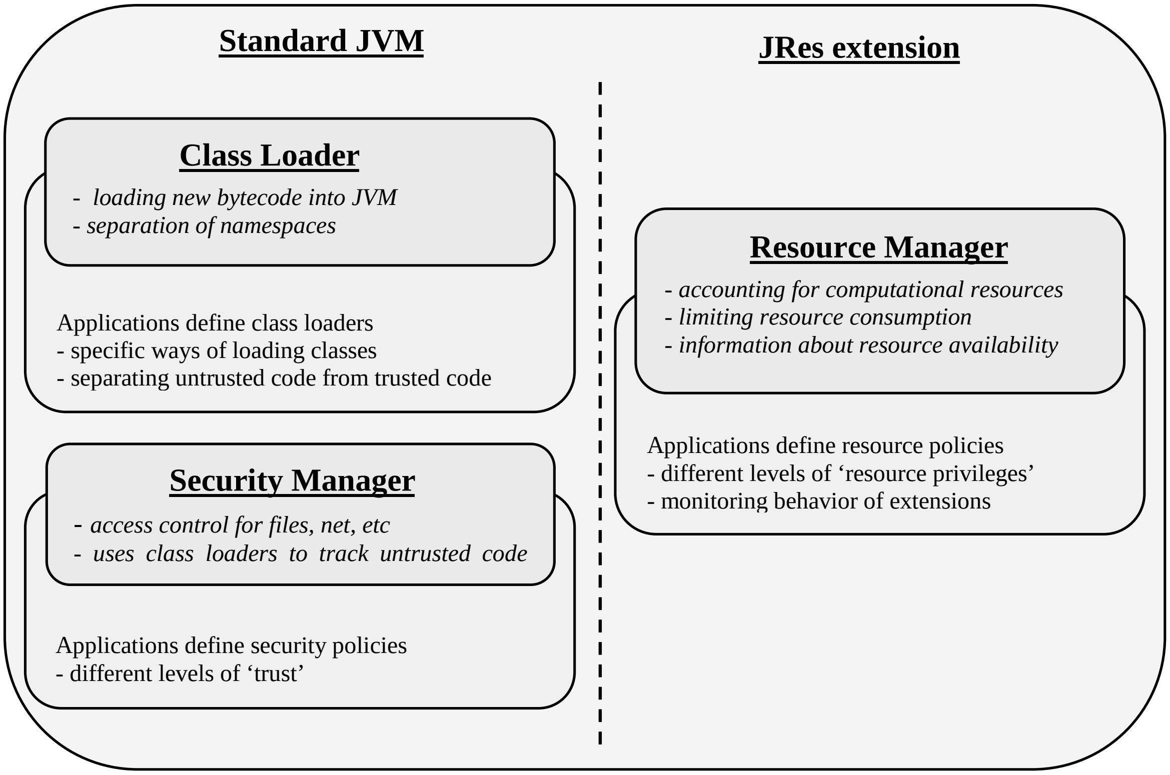

Figure 3. Extending the JVM with resource managing facilities.

assist load-time transformation: (i) classes retain enough symbolic information to enable members to be inserted or renamed, (ii) the JVM is a stack machine, which allows simple code fragments to be inserted into methods, and most importantly, (iii) the JVM uses a user-extensible class loader to locate and add new classes on demand. The class loader's method `loadClass()` is called by the JVM whenever it encounters a reference to a class not yet loaded. Typical class loaders implement `loadClass()` by searching for the class file (possibly accross the network) and then calling `defineClass`(), which converts an array of bytes into an instance of `java.lang.Class`. This instance of `Class` is verified to ensure that the class is semantically valid and that all operations are used with appropriate types before is ready for execution [9].

In JRes, classes can be either rewritten during class loading or off-line, which avoids runtime performance overheads. On-line bytecode rewriting occurs in `loadClass()` before calling `defineClass()`. After the rewriting is complete, the resulting byte array is passed to an invocation of `defineClass()` and the resulting Class object is passed to the JVM, just as if it were the original code. The JVM core does not even know that any changes were made. The current implementation of JRes uses a bytecode rewriting tool developed in the J-Kernel project [20].

Thread Registration

The core component of the current implementation of JRes is thread registration. Its goal is to store information about all threads that are or were active in the system. The module responsible for maintaining this information is contacted whenever a user program sets or clears a resource limit of a particular thread and whenever new resource consumption information becomes available. Another responsibility of the thread registration module is to generate upcalls for newly created threads. This happens if such upcalls have been registered by `ResourceManager.setThreadRegistrationCallback()`.

Thread registration is accomplished via injecting a few bytecode instructions at the beginning of every `run()` method of classes implementing the `Runnable` interface (only objects implementing this interface can be run as threads in Java). When a new thread is started, its `run()` method is invoked. This results in first registering the thread and then executing the original code. Invoking a particular `run()` method more than once or directly (e.g. not a starting method of a thread) is properly handled.

Accounting for Network Resources

JRes tracks the amount of data transferred by a thread. Implementing this was relatively straightforward. There are only a few native methods in classes belonging the `java.net` package which can access the network directly. The methods are either private or protected and are located in non-public classes. This means that they cannot be called by any code outside of `java.net`. Thus, the only action necessary to account for network usage was to rewrite the `java.net` package. The rewriting ensures that whenever one of these native, network-using methods is called, an appropriate information is recorded and a callback informing of exceeding resource limits is generated, if necessary.

A simple ping-pong progam was run between two computers connected by Fast Ethernet in order to estimate the overhead of JRes-instrumented bytecode. The round-trip time increased by about one percent (for messages larger than 50 KB) to 4.5% (for very small message sizes). This result is encouraging and made us resign from further experimenting with an alternative approach to tracking network usage, such as native code on top of sockets layer.

Accounting for CPU Time

In the current implementation of JRes, CPU time is accounted for through a mixture of bytecode rewriting and native code. During thread registration a native code routine is invoked in order to create an operating-system level handle to the new thread. This handle is used later to query the system for information on the CPU time usage by the thread.

As a part of the CPU accounting module startup, a new thread is created. This thread wakes up periodically and, using the stored thread handles, queries the operating system for CPU consumption. When necessary, appropriate overuse callbacks are generated. Except for the overhead of waking up the polling thread and querying the OS, there are no other runtime overheads.

In principle, gauging per-thread consumption of CPU time can be accomplished via bytecode rewriting. Inserting appropriate instructions at well-chosen points of methods can result in statistical information on per-thread CPU usage. We did not experiment with this approach, because the resulting execution time seems to be prohibitive when a reasonable degree of accuracy is to be achieved.

Accounting for Heap Memory

The goal of memory accounting is to know, at any point of program's execution, how much memory is used by objects allocated by each thread. This is accomplished by bytecode rewriting. Constructors of non-array objects are modified to record information about the allocating thread and finalizers are either modified or generated so that JRes can detect when the garbage collector frees an object. The code of every method is changed such that appropriate bytecode is inserted whenever an object allocating instruction occurs in the original method code.

When an object or an array is allocated, a method invocation is inserted just before the allocating instruction. The method will increment a counter by the new object's or array's size and verify that the memory limit has not been exceeded. If the limit is exceeded, the overuse callback is invoked.

When an array allocation instruction is encountered, code computing the number of entries in the array is inserted into the original bytecode. Computing the number of entries takes advantage of the fact that at runtime, just before an array is allocated, the size of every dimension is stored on top of the operand stack. In the case of multidimensional arrays these stack entries must be multiplied out.

An additional JRes method call is inserted after an array allocation bytecode instruction. The reason for doing so is to obtain a *weak pointer* to the newly allocated array; this pointer is used to detect deallocation of the array. Weak pointers are available in Java native code interfaces; their role is to reference objects without preventing their deallocation. If the referenced object is deallocated, every weak pointer referencing this object is changed to null. JRes stores weak pointers to all arrays allocated by a thread in a per-thread vector. Whenever an object allocation causes a thread to exceed its memory limit, the thread's list of weak pointers is scanned and sizes of all recently garbage-collected arrays (i.e. arrays whose corresponding weak pointers have become null) are subtracted from the thread's memory usage counter.

Figures 4 and 5 contain an example of a rewritten class. Figure 4 shows the source of an original class and the effect of transformations shown as source-to-source transformations. Figure 5 shows the actual effect of the rewriting on the bytecode.

It is important to note two things before analyzing the code. First, the numbers 0 and 1, used in calls to internal JRes methods, are identifiers for the classes `Foo` and `java.lang.Object`, respectively. These numbers are generated during bytecode rewriting. Second, the virtual machine of Java is a stack machine; in particular, methods expect their arguments on the stack (not in registers). The signatures of methods are shown in the bytecode in Figure 5. The figure gives a general idea of the increase in code size due to memory accounting transformations. The increase is proportional to the number of object and array allocations in the original code. Methods which do not allocate memory will not change during bytecode transformations.

Performance of Memory Accounting and Usage Limiting

Four diverse Java applications were run in order to evaluate the overheads of JRes memory accounting. The test suite consists of (i) running JavaCC, a parser generator from SunTest [33], on a C++ grammar; (ii)

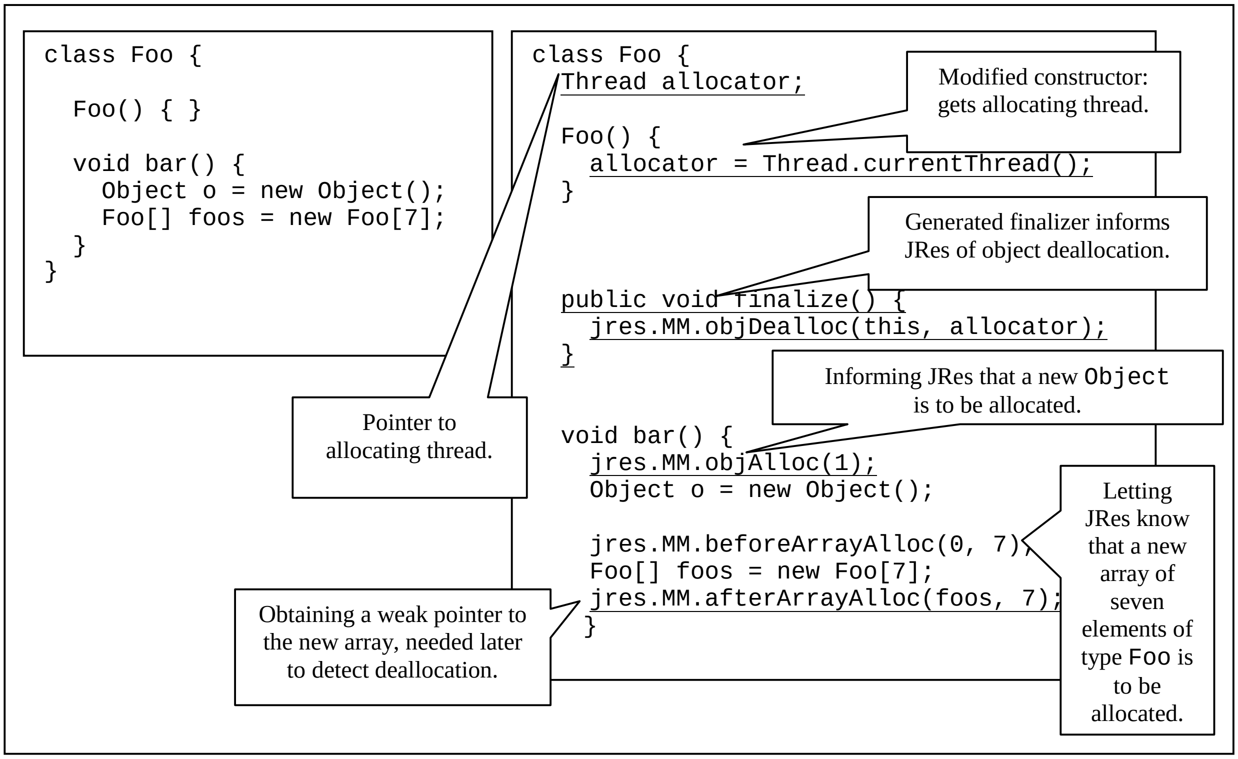

Figure 4. Effects of bytecode rewriting on a class `Foo`. The class has been rewritten in order to account for memory usage. The box on the left shows the original source file `Foo.java`. The box on the right shows the effects of transformations as if they were performed source-to-source.

executing a set of sorting and list manipulating functions in a Lisp interpreter [22]; (iii) running the embedded version of Caffeine Mark suite of tests [31]; and (iv) serving a set of Web pages with the Jigsaw Web server [24]. Classes of all the benchmarks programs were rewritten off-line for memory accounting. Memory limits were set to values that were never exceeded; thus, no overuse callbacks were invoked but after each object and array allocations memory limits have to checked.

Figure 6 shows the overheads caused by JRes as a percentage of the total execution time of rewritten programs. This percentage is split into three components. The first component reports the overhead of invoking empty finalizers on objects of modified classes and the increase in the duration of garbage collection. Since most Java objects do not have finalizers, garbage collectors optimize for that fact; however, in JRes-rewritten bytecode every non-array object has a finalizer. The second component is the overhead of accounting for memory usage, i.e. the cost of executing JRes routines maintaining memory usage statistics. The last component reports the overhead of checking for resource limits. The total overheads caused by the memory accounting and limiting do not exceed 18% on any of the benchmarks. The biggest contributor to the overhead is accounting for resource usage; this involves a method call, getting a reference to the current thread and incrementing an integer stored in a hashtable. Checking for limits involves a method call and looking up an object in a hashtable; this is the second biggest source of overheads. Finally, introducing the finalizers results in a small overhead compared to the cost of accoutning and limiting.

Table 1 contains several run-time and compile-time statistics of the benchmarked programs and their executions. The left side of the table (*"runtime statistics"*) helps to understand the data from Figure 6. As might be expected, the number of created objects has a direct impact on the cost of memory accounting. Lisp and Jigsaw allocate an object roughly every 250 bytecode instructions and they suffer the highest overheads. The Caffeine Marks benchmark allocates an object or array approximately every 14000 instructions, and the overheads of JRes memory management are very small on this program.

Across all four programs arrays are allocated much less frequently than non-array objects, which indicates that future JRes optimizations should be targeted towards improving the handling of the latter.

Compile-time statistics (right-hand side of Table 1) provide information on the cost of performing bytecode rewriting on the classes belonging to benchmark programs. The classes were rewritten off-line, which

```
Foo() {
  aload_0    // push reference from local 0
  invokespecial java/lang/Object.<init>() // invoke superclass constructor

  aload_0    // push reference from local 0
  invokestatic java/lang/Thread.currentThread()
  putfield Foo.allocator   // allocator = Thread.currentThread()
  return
}

void bar() {
  int local1;
  int local2, local3;

  iconst_0   // push 0 onto stack
  istore_2   // pop int from stack into local2
  iconst_1   // push 1 onto stack; 1 is a code of class java.lang.Object
  invokestatic jres/MM.objAlloc(int)

  new java/lang/Object
  invokespecial java/lang/Object.<init>() // constructor

  bipush 7   // push 7 onto stack

  iconst_0   // push 0 onto stack; 0 is a code of class Foo
  istore_3   // pop int from stack into local3
  dup        // duplicate top stack word
  istore_2   // pop int from stack into local2

  dup        // duplicate top stack word
  iload_2    // push int from local2

  invokestatic jres/MM.beforeArrayAlloc(java/lang/Object , int)

  anewarray Foo     // push a reference to a new array of Foo

  dup         // duplicate top stack word
  iload_2     // push int from local2
  invokestatic jres/MM.afterArrayAlloc(java/lang/Object , int )

  astore_1    // pop reference into local1
  aload_1     // push reference from local1
  iconst_0    // push 0 onto stack
  aconst_null // push null object reference
  aastore     // pop reference into array
  return
}

public void finalize() {
  aload_0    //push reference from local 0
  getfield Foo.allocator   // push contents of field Foo.allocator
  iconst_0   // push 0 onto stack; 0 is a code of class Foo
  invokestatic  jres/MM.deleteObject(java/lang/Thread , int )
  return
}
```

Figure 5. Bytecode resulting from transforming class `Foo` from Figure 4 so that it cooperates with JRes memory accounting. The underlined instructions are present both in the original class `Foo` and in the rewritten one; all the other instructions were generated in the process of bytecode rewriting. The original bytecode of Foo.class was generated with `javac` from JDK 1.1.1.

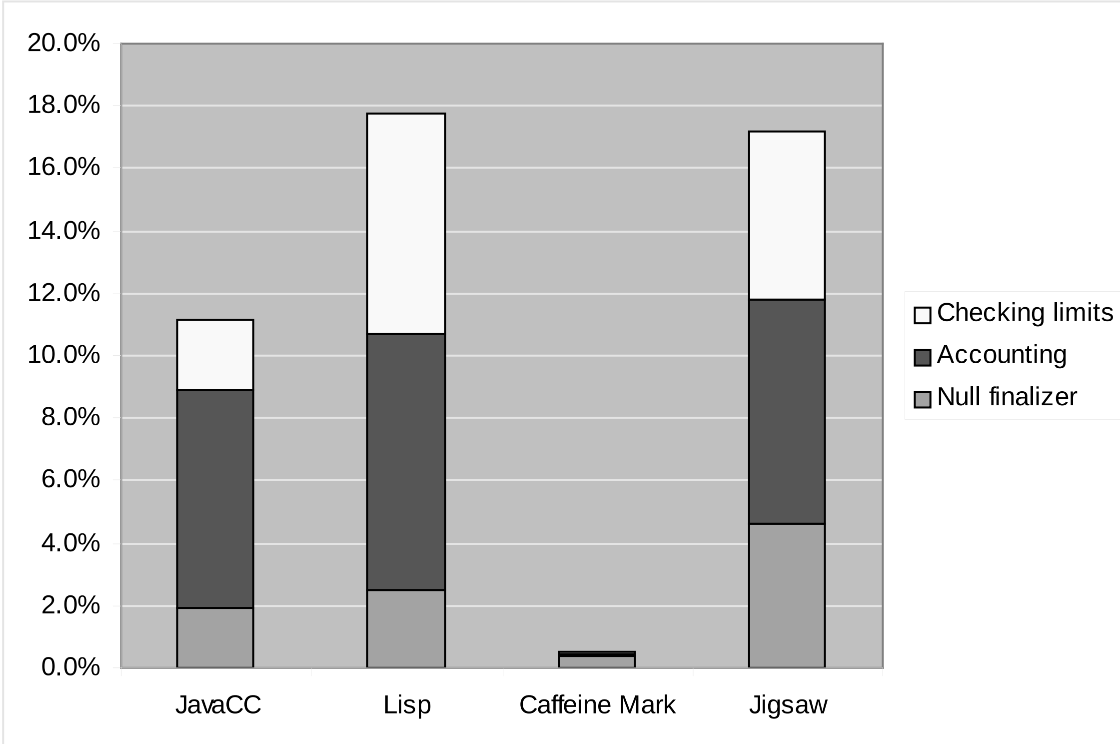

Figure 6. Overheads of accounting for memory and memory limiting.

means that the cost of rewriting is not included in the overheads reported in Figure 6. However, as discussed below, there are important cases in which bytecode rewriting has to happen on-line.

The *Classes* column reports the number of classes rewritten for a particular run of the program (e.g. our test execution of Jigsaw needed 47 classes, but the whole server code has many more classes). The last column in the table shows the time taken to rewrite these classes. With our current rewriting system, the costs of rewriting classes are substantial, even on the programs that contain very few object allocation instructions. The costs grow with the total number of bytecode instructions and the number of object and array creations.

It is not clear what the overheads of other bytecode rewriting systems [7,27] are. In the environments JRes is primarily aimed at, it may be necessary to rewrite classes on-the-fly — for instance, in extensible Web servers. Thus, rewriting will enter the critical path between the remote client issuing the request and obtaining the results. Fortunately extensions are likely to be small. For instance, it took about 2.4 seconds to rewrite *all* 134 example

Program	Runtime statistics			Compile-time statistics				
	Bytecode instructions	Object creations	Array creations	Bytecode instructions	Object creations	Array creations	Classes	Rewrite time (s)
JavaCC	134763243	85750	10749	154113	1885	193	62	15.3
Lisp	73620868	288842	0	5429	155	0	40	2.5
Caffeine	14108912	1011	10	1801	38	5	13	0.6
Jigsaw	1830860	6775	1079	30919	598	98	47	1.9

Table 1. Runtime and compile-time statistics of the benchmark programs.

applet classes from the JDK 1.1.1 distribution. The average size of these classes is 350 bytecode instructions. In systems where an extension can be reused after being loaded the rewriting time will be amortized over all executions of this extension.

DISCUSSION

The resource accounting interface presented in this paper significantly enhances the functionality of dynamically extensible Internet-oriented environments that subsume the functionality of an operating system for uploaded extensions. JRes enables the construction of such environments with the capability of tracking resource usage of extensions. The interface is small, simple, and allows for building complex policies on top of it.

JRes can be used to control access to resources other than CPU, heap memory and the network. For instance, thread creation callbacks can impose limits on the number of threads a particular thread group (and its child thread groups) can create. Although not included in the standard interface of JRes, the system can be extended to limit the number of objects of a certain type alive in the system at any given point of time. For instance, the number of windows opened by applets or the number of JDBC queries created by user defined functions in an extensible database can be controlled in this way.

JRes has several limitations. One of them is that the actions taken because of exceeding a resource limit are restricted by what Java allows. For instance, it is possible to lower a thread's priority but it is impossible to change the thread scheduling algorithm. Having full control over the scheduler would allow us to provide resource guarantees as well, in addition to enforcing resource limits as JRes currently does. Performance overheads of memory accounting are directly linked to the fact that JRes is not part of the JVM. For instance, being able to tap into the memory allocator and garbage collector would make it unnecessary to rewrite bytecode, which would certainly cut the overheads dramatically. The decision not to modify the JVM was dictated by two reasons. First, JRes is now a Java library with a small native component, and as such can be easily ported to most virtual machines. Second, not tying JRes into a JVM implementation allows for fast experimentation with various ideas. Ultimately, however, we would like to incorporate the functionality of JRes into an industry-strength JVM.

Another limitation of JRes is that it does not handle the sharing of objects by threads well. Consider a scenario in which a thread A creates a large object O and hands it off to a thread B. Thread A exits but O cannot be garbage collected and is not counted into B's memory consumption. This problem does not exist in environments where it is possible to identify cooperating threads and ensure that no data is shared between such groups of threads. In fact, this constraint is not as severe as it may appear. For instance, applet security rules ensure that applets cannot access threads in other thread groups. Another example of architecture where it is easy to define and identify collections of cooperating threads is the J-Kernel [20]. An important decision made in the J-Kernel design was to disallow object sharing between protection domains. Similar designs can be expected in future designs of extensible Java-based environments.

RELATED WORK

JRes is related to several research areas: resource accounting and enforcing resource limits in traditional and extensible operating systems, safe language technologies, and binary code transformations. In this section we summarize the most important work from these areas influencing our research.

Operating Systems

Enforcing resource limits has long been a responsibility of operating systems. For instance, many UNIX shells export the `limit` command, which sets resource limitations for the current shell and its child processes. Among others, available CPU time and maximum sizes of data segment, stack segment, and virtual memory can be set. Enforcing resource limits in traditional operating systems is coarse-grained in that the unit of control is an entire process. The enforcement relies on kernel-controlled process scheduling and hardware support for detecting memory overuse.

Single address space operating systems take advantage of the radical address space increase available to operating systems and applications with the appearance of 64-bit architectures. An early example of a 64-bit operating system is Opal [6]. Opal provides coarse-grained allocation and reclamation of resources (a set of memory pages, for instance), similar to that used in conventional operating systems. The basic storage management mechanism is explicit reference counting, which applications and support facilities use to allocate and release untyped storage in coarse units. A mechanism of *resource groups* is provided as the basis for a resource control policy, e.q. quotas or billing, to encourage or require users and their applications to limit resource consumption. Each time an application wants to obtain rights to use a particular resouce, it has to pass its own resource group capability as a hidden argument on system calls.

Another example of a single-address-space operating system is Mungi [21]. An interesting feature of that system is that it uses an economics-based model to manage backing store management. Applications obtain "bank" accounts from which "rent" is collected for the storage occupied by objects. Rent automatically increases as available storage runs low, forcing users to release unneeded storage. Bank accounts receive regular "income". In addition, a "taxation system" is used to prevent the excessive buildup of funds by inactive applications.

The architecture of the SPIN extensible operating system allows applications to safely change the operating system's interface and implementation [4]. SPIN and its extensions are written in Modula-3 and rely on a certifying compiler to guarantee the safety of extensions. The CPU consumption of untrusted extensions can be limited by introducing a time-out. Another example of an extensible operating system concerned with constraining resources consumed by extensions is the VINO kernel [32]. VINO uses software fault isolation as its safety mechanism and a lightweight transaction system to cope with resource hoarding. Timeouts are associated with time-constrained resources. If an extension holds such a resource for too long, it is terminated. The transactional support is used to restore the system to a consistent state after aborting an extension.

The main objective of extensible operating systems is to allow new services to be added to the kernel and for core services to be modified. Their built-in and "hard-coded" support for resource management is adequate for an operating system. In contrast, the main motivation behind JRes is building extensible, safe and efficient Internet environments implemented entirely in a safe language, such as Java. An extension may be an entire application and various billing, accounting and enforcing policies may have to be effective at the same time.

Programming Language Approaches

Except for the ability to manipulate thread priorities and invoke garbage collection, Java programmers are not given any interface to control resource usage of programs. Several extensions to Java attempt to alleviate this problem, but none of them shares the goals of JRes. For instance, the Java Web Server [23] provides an administrator interface which allows to view resource usage in a coarse-grained manner, e.g. the number of running threads can be displayed. PERC (a real-time implementation of Java) [30] provides an API for obtaining guaranteed execution time and assuring resource availability. While the goal of real-time systems is to ensure that applications obtain *at least* as many resources as necessary, the goal of JRes is to ensure that programs do not exceed their resource limits.

A very recent specialized programming language PLAN [16] aims at providing infrastructure for programming Active Networks. PLAN is a strictly functional language based on a dialect of ML. The programs replace packet headers (which are viewed as 'dumb' programs) and are executed on Internet hosts and routers. In order to protect network availability PLAN programs must use a bounded amount of space and time on active routers and bandwidth in the network. This is enforced by the language runtime system. JRes is similar to PLAN in that it limits resources consumed by programs. The main difference is that PLAN pre-computes resources available to programs based on the length of the program. The claim is that resources for an Active Networks program associated with a packet should be bounded by a linear function of the size of the packet's header.

Another approach to constraining resource consumption uses the ideas underlying Proof Carrying Code (PCC) [29]. PCC associates a proof of certain safety properties with a program. The program may contain some runtime checks necessary to allow a proof to be generated (for instance, a proof that no array access exceeds array bounds). After downloading, the proof is verified against the program, which is executed only if the verification is successful. In principle, proofs can be constructed guaranteeing that no more than a certain amount of memory will be allocated or that no more that a specified number of instructions will be executed. However, the feasibility of this approach is still unknown. Currently the size of such resource-constraining proofs exceeds the size of the original code by an order of magnitude. This is a serious problem in extensible Web environments since upload time is on the time-critical path.

Binary Rewriting

Code instrumentation through rewriting of binaries has been used for various purposes: emulation, tracing and debugging, profiling and optimization. Tools and toolkits such as Epoxie [38], Pixie [34] and QPT [25] are designed to rewrite programs so that the modified code generates address traces and instruction counts. Generating address and instruction traces is handled by ATUM [1], which allows for detailed *post-mortem* tracing. Emulation of other architectures is the domain of the PROTEUS [5] and Shade [8] binary rewriting tools. The well-known Purify [17] software testing tool detects memory leaks and access errors using binary rewriting techniques as well.

Another class of tools employing binary rewriting techniques consists of tools editing executable code. The tools offer a library of routines for modifying executable files so that users can design their own binary rewriting strategies. Examples include OM [36], ATOM [35], EEL [26] and recently Etch [13]. A detailed summary of binary code rewriting tools and techniques can be found in [27].

The importance of Java bytecode transformation has been recognized by other research groups recently. For instance, several recent projects rely on bytecode rewriting to provide increased levels of security [20,38]. Publications on BIT [27] and JOIE [7] utilities contain a detailed list of issues that need to be solved when designing such bytecode rewriting tools. Overheads reported in [27] are evidence that Java bytecode rewriting is extremely performance sensitive.

CURRENT STATUS AND FUTURE WORK

The system described in this paper is operational and consists of two packages and one native library. Currently it requires Microsoft's JVM but we plan to port it to Sun's JDK as well.

JRes has been used to alleviate the problem of denial-of-service attacks targeted at browsers. The interface provides enough infrastructure to define a simple policy preventing applets from using more heap memory, CPU time and network resources than a user specifies in a configuration file. As a practical demonstration of usefulness of JRes, the classes implementing JRes and the class implementing the resource consumption policy were added to the core classes of JVM executing inside Microsoft Internet Explorer. The result is an enhanced browser in which applets are prevented from mounting denial-of-service attacks through monopolizing the use of any of the three mentioned resources.

In addition to being an enabling technology for a class of software systems, JRes is being used to understand resource management for an emerging model of Internet computing []. The main characteristics of this model are high code mobility and large numbers of anonymous users, which result in different kind of resource demands made by applications than in traditional operating system environments. Efficient use of distributed computational resources becomes a must if the model is to be useful. JRes provides an experimental infrastructure to research these issues.

The design and implementation of JRes demonstrate that many interesting and useful resource management functions can be added to Java without modifying the JVM, although certainly we would like to incorporate JRes into the implementation of the JVM. Currently we are working on incorporating JRes into the Jaguar extensible database [14] and the J-Kernel [20]. Another topic of current work is the transfer of object ownership.

SUMMARY

This paper presents JRes — an interface extending Java with resource accounting and limiting capabilities. The main motivation behind JRes is facilitating the creation of extensible Internet environments, onto which untrusted code can be uploaded. Currently Java lacks resource management support, which is a severe shortcoming when building such environments. JRes addresses this problem with a small interface that is flexible enough to allow complex resource management policies to be built. The implementation uses a combination of bytecode rewriting and native code to add resource management to off-the-shelf Java virtual machines. The performance overheads are small in the case of accounting for CPU time and network resources. On a set of four well-known Java applications memory accounting resulted in overheads between 0.5-18%.

The current implementation of JRes did not require modifications of the Java Virtual Machine. The discussion of the usefulness of the interface and its limitations and sources of performance overheads presented in this paper can be important when adding a functionality of JRes to an industry-strength implementation of the JVM.

ACKNOWLEDGEMENTS

The authors are grateful to Chi-Chao Chang, Chris Hawblitzel, Li Gong, Greg Morrisett, and Praveen Seshadri for for comments and discussions.

This research is funded by DARPA ITO contract ONR-N00014-92-J-1866, NSF contract CDA-9024600, a Sloan Foundation fellowship, and Intel Corp. hardware donations.

REFERENCES

1. Agarwal, A, Sites, R, and Horowitz, M.. *ATUM: A New Technique for Capturing Address Traces Using Microcode*. Proc. 13th International Symposium on Computer Architecture, June 1986.
2. Arnold, K and Gosling, J. *The Java Programming Language.* Addison-Wesley.
3. Bershad, B, Savage, S and Pardyak P. *Protection is a Software Issue*. Fifth Workshop on Hot Topics in Operating Systems, Orcas Island, WA, May 1995.
4. Bershad, B, Savage, S, Pardyak, P, Sirer, E, Fiuczynski, M, Becker, D, Eggers, S, and Chambers, C. *Extensibility, Safety and Performance in the SPIN Operating System*. 15th ACM Symposium on Operating Systems Principles, Copper Mountain, CO, December 1995.
5. Brewer, E, Dellarocas, C, Colbrook, A and Weihl, W. *PROTEUS: A High-Performance Parallel-Architecture Simulator*. Massachusetts Institute of Technology technical report MIT/LCS/TR-516, 1991.
6. Chase, J, Levy, H, Feeley, M and Lazowska, E. *Sharing and Protection in a Single Address Space Operating System*. In ACM Transactions on Computer Systems, May 1994.
7. Clark, D and Tennenhouse, D. *Architectural Considerations for a New Generation of Protocols* Proc. SIGCOMM Symposium on Communications Architectures and Protocols, Philadelphia, PA, September 1990.
8. Cmelik, R and Keppel, D. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1994.
9. Cohen, G, Chase, J, and Kaminsky, D. *Automatic Program Transformation with JOIE.* Proc. USENIX Annual Conference, New Orleans, LA, June 1998.
10. Czajkowski, G, Chang, C-C, Hawblitze, C, Hu, D and von Eicken, T. *Resource Management for Extensible Internet Servers.* Proc. of
11. Denning, A. *ActiveX Controls Inside Out.* Microsoft Press, 1997.
12. Engler, D, Kaashoek, F, and O'Toole, J. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. Proc. ACM

Symposium on Operating Systems Principles, Copper Mountain, CO, December 1995.

13. Etch.http://memsys.cs.washington.edu/memsys/html/ etch.html.
14. Godfrey, M , Mayr, T, Seshadri P, and von Eicken, T. *Secure and Portable Database Extensibility*. Proc. ACM SIGMOD International Conference on Management of Data, SIGMOD Record, vol 27, Issue 2, June 1998.
15. Gong, L and Schemers, R. *Implementing Protection Domains in the Java Development Kit 1.2*. Internet Society Symposium on Network and Distributed System Security, San Diego, CA, March 1998.
16. Gosling, J, Joy, B, and Steele, G. *The Java language specification.* Addison-Wesley, 1996.
17. Hastings, R and Joyce, B. *Purify: Fast Detection of Memory Leaks and Access Errors.* Proc. Winter USENIX Conference, January 1992.
18. Graham, S, Kessler, P and Marshall K. *An Execution Profiler for Modular Programs*. Software Practice and Experience, pages 671-685, vol 13, 1983.
19. Hicks, M, Kakkar, P, Moore, J, Gunter, C, and Nettles, S. *PLAN: A Programming Language for Active Networks.* Submitted to ACM SIGPLAN Conference on Programming Language Design and Implementation, 1998.
20. Hawblitzel, C, Chang, C-C, Czajkowski, G, Hu, D, and von Eicken, T. *Implementing Multiple Protection Domains in Java.* In Proceedings of USENIX Annual Conference, New Orleans, LA, June 1998.
21. Heiser, G, Lam, F and Russel, S. *Resource Management in the Mungi Single-Address-Space Operating System.* In Proceedings of the 21st Australasian Computer Science Conference, Perth, Australia, February, 1998.
22. Jackson, J and McClelan, A. *Java By Example*. SunSoft Press, April 1996.
23. Java Web Server Home Page. http://jserv.javasoft.com/products/webserver/index.ht ml.
24. Jigsaw Overview. http://www.w3c.org/Jigsaw.
25. Larus, J and Ball, T. *Rewriting Executable Files to Measure Program Behavior*. Software, Practice and Experience, vol 24, no. 2, February 1994
26. Larus, J and Schnarr, E. *EEL: Machine-Indenepdent Executable Editing.* Proc. SIGPLAN Conference on Programming Language Design and Implementation, June 1995.
27. Lee, H and Zorn, B. *BIT: A Tool for Instrumenting Java Bytecodes.* In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, CA, December 1997.
28. Lindholm, T and Yellin, F. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
29. Necula, G. *Proof-Carrying Code.* In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Paris, France, January 1997.
30. Nilsen, K. *Issues in the Design and Implementation of Real-Time Java*. Java Developer's Journal, 1996,
31. Pendragon Software. Caffeine Mark 3.0. http://www.pendragon-software.com/pendragon/cm3/index.html.
32. Seltzer, M, Endo, Y, Small, C, and Smith, K. *Dealing with Disaster: Surviving Misbehaved Kernel Extensions.* Proceedings of the Second Symposium on Operating System Design and Implementation, Seattle, WA, November 1996.
33. Sirer, E, Fiuczynski, M, and Pardyak, P. *Writing an Operating System with Modula-3*. First Workshop on Compiler Support for System Software, Tucson, AZ, February 1996.
34. Smith, M. *Tracing with Pixie*. Memo from Center for Integrated Systems, Stanford University, April 1991.
35. Srivastava, A and Eustace, *A. ATOM A System for Building Customized Program Analysis Tools*. Proc. of the SIGPLAN '94 Conference on Programming Language Design and Implementation, June 1994.
36. Srivastava, A and Wall, D. *A Practical System for Intermodule Code Optimization at Link-Time.* Journal of Programming Languages, vol 1, no 1, March 1993.
37. SunTest. JavaCC - Java Parser Generator. http://www.sun.com/suntest/JavaCC.
38. Wall, D. *Systems for Late Code modification*. In Robert Giegerich and Susan L. Graham, eds., Code Generation – Concepts, Tools, Techniques, Springer-Verlag, 1992.
39. Wallach, D, Balfanz, D, Dean, D, and Felten, E. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, p. 116–128, Saint-Malo, France, October 1997.