Resource Control in J-SEAL2[*]

Walter Binder
CoCo Software Engineering GmbH, Margaretenstr. 22/9, A-1040 Vienna, Austria
E-mail: w.binder@coco.co.at

Jarle G. Hulaas
University of Geneva, rue Général Dufour 24, CH-1211 Geneva 4, Switzerland
E-mail: Jarle.Hulaas@cui.unige.ch

Alex Villazón
University of Geneva, rue Général Dufour 24, CH-1211 Geneva 4, Switzerland
E-mail: Alex.Villazon@cui.unige.ch

Cahier du CUI No 124
Technical Report, version of 2001/01/03

Abstract

Resource control, i.e., accounting and limiting the consumption of physical resources like CPU and memory, and of logical resources like threads, is necessary for distributed agent systems to execute securely. This is however a missing feature on standard Java platforms. Moreover, prevailing approaches to resource control in Java require substantial support from native code libraries, which is a serious disadvantage with respect to portability, since it prevents the deployment of applications on large-scale heterogeneous networks. This report describes the model and implementation mechanisms underlying the new resource-aware version of the J-SEAL2 mobile agent kernel. The resource control model is based on a set of requirements, where portability is very significant, as well as a natural integration with the existing programming model. The implementation consists of a combination of Java byte-code rewriting with well-chosen enhancements in the J-SEAL2 kernel. Realization of a resource control system may be prompted by motivations such as the need for application service providers to guarantee a certain quality of service, or to create the support for usage-based billing. In this report the design strategy is however focussed on security, and more specifically on preventing denial-of-service attacks originating from mobile agents running on the platform. Initial performance measurements are also presented, which back our approach.

Target Audience

This report is suitable for developers of Java applications who are interested in resource control issues, and it gives some general guidelines on how to achieve a completely portable implementation. Whereas J-SEAL2 is primarily designed for mobile agents, the approach described here is directly applicable to many other distributed programming paradigms practiced in Java.

[*]This work was funded by CoCo Software Engineering GmbH and the Swiss National Science Foundation grants 20-54014.98 and 5003-057753.

1 Introduction

Java [14] was designed as a general-purpose programming language, with special emphasis on portability in order to enhance the support of distributed applications. Therefore, it is natural that access to low-level, highly machine-dependent mechanisms were not incorporated from the beginning. New classes of applications are however being conceived, which rely on the facilities offered by Java, and which at the same time push and uncover the limits of the language. These novel applications, based on the possibilities introduced by code mobility, open up traditional environments, move arbitrarily from machine to machine, execute concurrently, and compete for resources on devices where everything from modest to plentiful configurations can be found. We are therefore witnessing increased requirements regarding fairness and security, and it becomes indispensable to acquire a better understanding and grasp of low-level issues such as resource management.

Operating system kernels provide mechanisms to enforce resource limits for processes. The scheduler assigns processes to CPUs reflecting process priorities. Furthermore, only the kernel has access to all memory resources. Processes have to allocate memory regions from the kernel, which verifies that memory limits for the processes are not exceeded. Likewise, a mobile agent kernel must prevent denial-of-service attacks, such as agents allocating all available memory. For this purpose, accounting of physical resources (i.e., memory, CPU, network bandwidth, etc.) and logical resources (i.e., number of threads, number of protection domains, etc.) is crucial.

Whereas J-SEAL2 [5, 6] is primarily designed for mobile agents, the approach described here is directly applicable to many other distributed programming paradigms practiced in Java, since mobile agent technology is a very general one. J-SEAL2 is thus perfectly fitted to serve as a secure execution platform for Java applets, or traditional distributed applications, where strong protection domains and resource control mechanisms are often needed. Further use cases include technologies such as World-Wide-Web server extensions (Java servlets [20]) and Java application servers (e.g., Enterprise JavaBeans containers [19]).

The great value of resource control is that it is not restricted to serve as a base for implementing security mechanisms. Application service providers may e.g. need to guarantee a certain quality of service, or to create the support for usage-based billing, in order to amortize investments in hardware and software set at customers' disposal. The basic kernel extensions described here will be necessary to schedule the quality of service or to support the higher-level accounting system, which will bill the clients for consumed computing resources. This report is however restricted to the kernel extensions that were necessary to add resource control to J-SEAL2; faithful to the micro-kernel approach, J-SEAL2 relegates to the higher levels the mechanisms which do not absolutely have to be part of the kernel.

This report is organized as follows. The next section presents the design goals and the resulting resource control model, and section 3 the corresponding APIs. Section 4 explains our implementation techniques, for which section 5 presents some initial performance measurements. Section 6 compares our approach with related work, whereas section 7 gives a glimpse on future investigations and concludes the report.

2 Objectives and Resulting Model

The ultimate objective of this work is to enable the creation of execution platforms where anonymous agents, or more general programs, may securely coexist without harming each other, and without harming their environment. Examples of such platforms are user-extensible databases [13] or decentralized e-commerce and trading systems as e.g. in [15]. Java applet execution platforms – World-Wide-Web browsers – as well as embedded Java devices also need such guarantees. The desire to deploy this kind of platforms translates into the following requirements:

- Sufficiently abstract concepts, in order to make mapping of policies into implementations more straightforward, and with a view to making resource control and eventual billing more manageable.

- Accounting of low-level, physical resources as

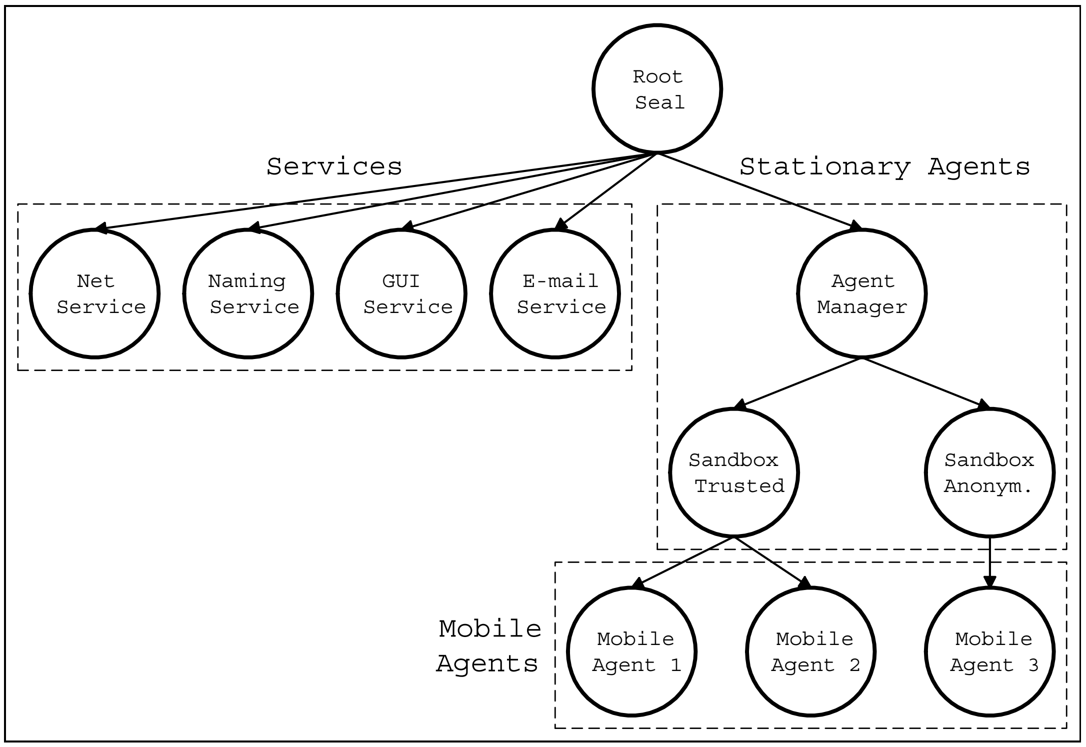

Figure 1: J-SEAL2 nested protection domains.

well as higher-level, logical resources, such as threads.

- Prevention against denial-of-service attacks, be they based on CPU, memory, or communication misuse.
- Fair distribution of resources among concurrent domains, even outside the context of malicious activities.
- Fine-grained load-balancing of mobile agent applications on a cluster of machines.

Since some aspects of resource control are to be manageable by the application developer, it is important that the general model integrate well with the existing J-SEAL2 programming model [5]. The J-SEAL2 kernel manages a tree hierarchy of nested protection domains, which may be either agents or service components. Each agent and service executes in a protection domain of its own, called a sealed object or *seal* for short. Figure 1 shows some frequently used service seals. This model of hierarchically organized protection domains stems from the JavaSeal mobile agent kernel [8].

The resource control facilities shall reflect the hierarchical system structure. Hierarchical process models have been used successfully by operating system kernels, such as the Fluke micro-kernel [11]. The Fluke kernel employs a hierarchical scheduling protocol, CPU Inheritance Scheduling [12], in order to enforce scheduling policies. In this model, a parent domain donates a certain percentage of its own CPU resources to a child process. Initially, the root of the hierarchy possesses all CPU resources.

A generalization of CPU Inheritance Scheduling fits very well to the J-SEAL2 hierarchical domain model. At system startup the root domain, *RootSeal*, owns by default all physical and logical resources, for example 100% CPU, the entire virtual memory, unlimited network usage, the maximum number of threads the underlying Java Virtual Machine (JVM) [16] is able to cope with, an unlimited number of subdomains, etc. Moreover, the root domain, along with the other domains loaded at platform startup, are considered as completely safe, and, consequently, no resource accounting will be enforced on them. This default behavior may however easily be overridden if specific configurations should require accounting even for trusted domains.

When a nested protection domain is created, the creator donates some part of its own resources to the new domain. Figure 2 illustrates the way resources are either shared or distributed inside a seal hierarchy. In the formal model of J-SEAL2, the Seal Calculus [24], the parent seal supervises all its subdomains, and inter-domain communication management was the main concern so far. Likewise, in the resource control model proposed here, the parent seal is responsible for the resource allocation with its subseals. This produces a nested structure, where the parent seal is initially the sole owner of its resources, and it may either share them or dispatch fractions of them to its subseals. However, the sum of all resources within a protection domain, e.g., in the *Untrusted application* of figure 2, remains constant.

Our resource control model stems from further design goals, such as portability and transparency: the next subsections are dedicated to describing these.

2.1 Portability and Transparency

Portability is crucial for the success of any mobile agent platform. There are already some Java-based systems offering resource control facilities, such as Alta [23], GVM [4], KaffeOS [1, 2], etc. However, they rely on modified Java runtime systems, which are not portable. As a result, these systems are not suited for large-scale applications that have to support a wide variety of different hardware platforms and operating systems. Our goal is to provide a general-purpose model which is not dependent on specific implementation techniques, and to explore primarily completely portable solutions. This entails that we have to cope with certain restrictions and with performance levels sometimes inferior to those of existing realizations. Our portable approach will nevertheless show its advantages in the longer term: our solution will always perform somewhat slower than the fastest JVMs without resource control mechanisms, but, on the other hand, we will be able to exploit the latest techniques in Java implementation optimizations, which will often not be possible with non-portable implementations.

A related important requirement of our resource control model is that unmodified off-the-shelf applications must be able to execute on our platform. In other words, resource control must be transparent to applications which do not explicitly manage their pool of resources.

For portability reasons, it should also be stressed that the goal of this work is not to implement any kind of real-time guarantee. The resources that are managed and distributed internally to the JVM are thus entirely dependent upon what the JVM process itself is given by the underlying operating system.

2.2 Minimal Overhead for Trusted Domains

Since J-SEAL2 is designed for large-scale applications, where a large number of services and agents are executing concurrently, design and implementation must minimize the overhead of resource accounting. Some domains, such as core services, are fully trusted. Their resource consumption need not be controlled by the kernel.

2.3 Support for Resource Sharing

In certain situations protection domains that are neighbours in the hierarchy may choose to share some resources. In this case, resource limits are enforced together for a set of protection domains. As a result, resource fragmentation is minimized. For example, consider an agent creating a subagent for a certain task. Frequently, the creating agent does not want to donate some resources to the subagent, but it rather prefers to share its own resources with the subagent. A property of our approach is that if a domain has unlimited access to a resource, this means that it is sharing it with RootSeal.

2.4 Managed Resources

Within each untrusted protection domain, the J-SEAL2 kernel shall account for the following resources:

- CPU_RELATIVE defines the relative share of

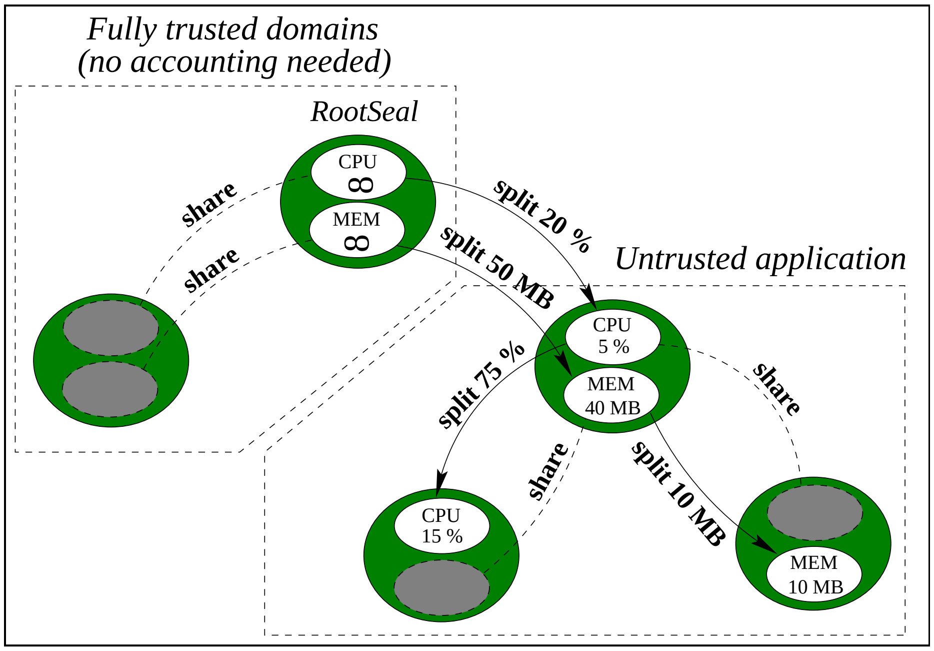

Figure 2: Illustration of the general resource control model.

CPU, and is expressed as a fraction of the parent domain's own relative share[1].

- MEM_ACTIVE is the highest amount of volatile memory that a protection domain is allowed to use at any given moment.
- THREADS_ACTIVE specifies the maximal number of active threads by protection domain at any moment. Uncontrolled creation of threads has to be avoided, as it results in increased load for the scheduler; it may even crash the JVM, as there is currently no standard Java construct allowing one to inquire about a JVM's maximum number of threads.
- THREADS_TOTAL limits the number of threads that may be created throughout the lifetime of a protection domain, as thread creation is an expensive (kernel-level) operation.
- DOMAINS_ACTIVE specifies the maximal number of active subdomains a protection domain is allowed to have at any given moment. This limit is to minimize management overhead inside the kernel by controlling the complexity of the seal hierarchy at any time.
- DOMAINS_TOTAL bounds the number of subdomains that a protection domain may generate throughout its lifetime, as domain creation and termination are expensive kernel operations.

Note that the kernel of J-SEAL2 is not responsible for network control. This is because the microkernel does not provide access to the network. Instead, network access can be provided by multiple services. These network services or some mediation layers in the hierarchy are responsible for network accounting according to application-specific security policies. Let us stress that the network is not a special case, since J-SEAL2, thanks to its homogenous model, may limit communication with any services, like e.g. file IO.

Another resource kind that could be expected in the above list of kernel-managed resources is the total amount of CPU allocated to a given protec-

[1]In our current implementation, this resource is controlled by periodic sampling of the amount of executed bytecode instructions. The precision of the measurement is implementation dependent; there is indeed a bias induced by the fact that the CPU resource is not allocated by absolute values, but by relative shares, while in the implementation, the reference value is the aggregated consumption measured among untrusted domains and is not, as would be expected, the resource taken as a whole.

tion domain throughout its lifetime. It is however not clear what the unit of measurement should be for this resource, while still preserving a completely hardware-independent model. The main objective of this kind of resource accounting would be to prevent applications from indefinitely cluttering up platforms; in a heterogeneous set of servers it gives however more sense to express total life time abstractly as the wall clock time elapsed since the application was started, than as the number of consumed CPU cycles. Using as unit of measurement the amount of executed Java byte-codes, although portable, was also regarded as too low-level. Measuring wall clock time can be achieved at the application level, by establishment of a controller agent with sufficient rights to kill all misbehaving applications; this is a viable approach, since in J-SEAL2, when a parent disposes of a child seal, all resources are guaranteed to be freed properly. Accounting of total CPU time was therefore discarded from the kernel.

Finally, there is also no such resource as MEM_TOTAL, a limit to the accumulated amount of memory used throughout the lifetime of a protection domain. It could be needed to prevent the kind of denial-of-service attacks where a malicious agent creates a lot of dynamic objects in order to keep the CPU busy with garbage collection. Its implementation would however require maintenance of an additional counter, which we preferred to avoid. Instead, J-SEAL2 will take preventive action by charging an abstract amount of CPU as a compensation for the garbage collection induced by each object created.

The six basic resource types retained for management by the J-SEAL2 kernel are discussed in more detail in the API section below.

3 API

Integrating resource control into the J-SEAL2 kernel requires extensions to the kernel API. There are 2 new kernel abstractions: A resource object of type `Res` represents a resource of a certain type available for a protection domain. Resource sets of type `ResSet` ease the management of multiple resources.

A domain may split its resources and donate some parts to subdomains during their creation. The `unwrap` method of the already existing kernel class `Seal` creates a nested protection domain. Thus, the `Seal` API has been extended to allow a parent domain to restrict the resources of its children.

3.1 Definitions

In this section we provide some definitions, which simplify the description of the resource control API. In the following definitions let S denote an arbitrary domain in the hierarchy.

Root `Res` object: A root `Res` object of the domain S is a `Res` object responsible for resource control in S. A root `Res` object is returned by an invocation of the method `getCurrentRes` in class `Res` (details about this method are covered by the next section).

Descendant `Res` object: A descendant `Res` object D of the domain S is the result of splitting a root `Res` object R of S. R is also called the parent `Res` object of D. When a descendant `Res` object is used in a `ResSet` object to create a nested domain, it will be used for resource control in the unwrapped child domain.

Note that these definitions are relative to the domain S. A descendant `Res` object D of the domain S is a root `Res` object in a child C of S, if D was in the `ResSet` object used for unwrapping C. When we use the terms root and descendant `Res` objects in the description of a method, we implicitly assume `Res` objects of the domain invoking the method.

3.2 Class `Res`

For each type of resource, a protection domain has an associated root `Res` object reflecting how much of the resource the domain has been granted. A `Res` object defines a resource limit and provides information on the current resource usage in order to support resource aware computations. It offers an operation allowing a domain to split up some part of the resource. This operation yields a new descendant `Res` object that may be donated to children domains. Table 1 summarizes the interface of a `Res` object.

```
public final class Res {
   public static final int
      CPU_RELATIVE = 0,
      MEM_ACTIVE = 1,
      THREADS_ACTIVE = 2,
      THREADS_TOTAL = 3,
      DOMAINS_ACTIVE = 4,
      DOMAINS_TOTAL = 5;

   public static Res
                 getCurrentRes(int type);
   public int getType();
   public long getLimit();
   public long getUsage();
   public Res split(long limit);
   public void setLimit(long limit);
   public void combine();
   public /*hidden*/ void finalize();
}
```

Table 1: The new `Res` API.

3.2.1 Resources, Limits, and Consumption

The static method `getCurrentRes` returns the root `Res` object for a given type of resource of the invoking domain. The constants `CPU_RELATIVE`, `MEM_ACTIVE`, `THREADS_ACTIVE`, `THREADS_TOTAL`, `DOMAINS_ACTIVE`, and `DOMAINS_TOTAL` (i.e., relative CPU share, active memory in bytes, as well as active and cumulative threads and subdomains) are used to indicate the requested resource type. `IllegalArgumentException` is thrown, if an invalid resource identifier is given as argument.

The information, for which type of resource a `Res` object is responsible, is permanently associated with the `Res` object in order to prevent the programmer from mixing up different types of resources by mistake. The `getType` method returns the type of resource a `Res` object is representing.

`getLimit` returns the resource limit of a `Res` object. A negative value means that there is no resource limit. Note that in general it is not possible for a domain to determine whether it shares resources with other domains, except for the case where the limit is negative, which implies that the domain shares the resource with RootSeal.

Concerning the semantics of the resource limit, the relative CPU share (`CPU_RELATIVE`) is treated differently from all other resource types. A relative CPU share of n means that domains created with the corresponding `Res` object may use at most a fraction of $\frac{n}{\text{sum of all CPU limits in the system}}$ of the CPU time available to domains with a CPU limit ≥ 0.

`getUsage` returns the resource consumption of all domains sharing the same root `Res` object. A negative value means that the J-SEAL2 kernel does not account for the resource. In general, the following implication holds for all `Res` objects: $\texttt{getUsage} < 0 \rightarrow \texttt{getLimit} < 0$. That is, a domain which has a limit for a certain type of resource can always ask about its current resource usage. The reverse implication may hold only for certain types of resources, depending on the implementation (if there is no limit for a certain resource, the J-SEAL2 kernel need not account for that resource). If `getUsage` is invoked on a `Res` object representing a relative CPU share, `UnsupportedOperationException` is thrown.

3.2.2 Splitting Resources

As the `Res` API does not expose any public constructor, the `split` operation has to be used in order to create descendant `Res` objects that may be donated to subdomains. `split` may be invoked only on root `Res` objects. It returns a new descendant `Res` object responsible for the same type of resource as the root `Res` object, which becomes the parent of the descendant. Therefore, all `Res` objects in a J-SEAL2 system form a tree hierarchy. The descendant `Res` object has the resource limit, which was passed to `split` as argument, and an initial resource usage of zero. The resource usage of the parent `Res` object is incremented by the limit given to the descendant. `IllegalArgumentException` is raised, if the limit passed to `split` is negative or smaller than the minimal limit for the resource type the `Res` object is responsible for. For each resource type, the minimal limit is defined by the system administrator in a configuration file. `IllegalStateException` is thrown, if `split` is invoked on a descendant. `ResourceOveruseException`, a subtype of `RuntimeException`, is raised, if the `split` operation would exceed the limit of the root `Res` object.

`split` allows a `Res` object without resource limit to act as a theoretically unlimited generator of descendants. For practical reasons, we require that the sum of all limit assigned to descendants does not exceed the maximum `long` value `Long.MAX_VALUE`. This restriction enables the calculation of relative CPU shares using only integer arithmetic.

3.2.3 Adjusting Resource Limits

The `setLimit` method provides a mechanism to modify the resource limit of a `Res` object. The new resource limit is passed as argument. The resource usage of the parent `Res` object is adjusted accordingly. This operation enforces the following constraints:

- `setLimit` may be invoked only on a `Res` object with a limit ≥ 0. Otherwise, `IllegalStateException` is raised.
- A limit cannot be removed, i.e., the argument must be a non-negative value, which is larger or equal to the minimal limit for the corresponding resource type. Otherwise, `IllegalArgumentException` is thrown.
- The new resource limit must not be smaller than the current resource usage. Otherwise, `ResourceOveruseException` is thrown.
- If invoked on a root `Res` object, the given resource limit must not be larger than the current limit of the object. Otherwise, `IllegalStateException` is raised. This constraint ensures that a domain may only reduce its own limit.
- If invoked on a descendant `Res` object, either the new limit has to be smaller than the current one, or the resource usage of the parent object (which is a root `Res` object in the calling domain) plus the difference between the new limit and the current one must not exceed the limit of the parent `Res` object. Otherwise, `ResourceOveruseException` is thrown. This rule allows a parent domain to increase or reduce the resource limits of its children, but not exceeding its own limit.

A parent domain may use descendant `Res` objects in order to monitor the resource usage of children domains. With the aid of `setLimit`, the parent is able to adjust the resource limits for the children domains.

3.2.4 Combining Resources

The `combine` operation allows to merge `Res` objects that have been split before. If it is invoked on a root `Res` object, `combine` has no effect. If it is called on a descendant `Res` object, the descendant is combined with its parent `Res` object, i.e., the resource usage of the parent object (if it is accounted for) is reduced by the limit of the descendant. The descendant `Res` object is marked as invalid and cannot be used anymore. Combination is only possible, if the descendant `Res` object is not used by any subdomain (i.e., all subdomain created with the descendant `Res` object must be terminated before). Otherwise, `IllegalStateException` is thrown.

The `Res` API also provides a finalizer to be invoked by the garbage collector, when a `Res` object becomes eligible for garbage collection. If the `Res` object to be reclaimed has not been combined with its parent before, the finalizer does the combination in order to avoid resource leakage. Note that the constraint of the `combine` operation is satisfied, because when the finalizer is called, the `Res` object is not in use anymore. The `finalize` method is marked as hidden in order to prevent user code from invoking the finalizer. J-SEAL2 uses extended byte-code verification in order to ensure that hidden class members are not accessed, even though they may be declared as public. For details regarding extended byte-code verification in J-SEAL2, see [6].

3.3 Class `ResSet`

A `ResSet` object offers a convenient way to manage all resources given to a domain. It holds exactly one `Res` object for each type of resource. Table 2 summarizes the public interface of a `ResSet` object:

The static method `getCurrentResSet` returns a `ResSet` object with the root `Res` objects of the domain the calling thread is executing in. This `ResSet` object may be used to access the individual `Res` objects of the domain. The `copy` method creates a shallow copy of a `ResSet` object. The copy contains the same references to `Res` objects as the original `ResSet` object. The `getCurrentResSet` and `copy` methods are the only mechanisms allow-

```
public final class ResSet {
   public static ResSet
                 getCurrentResSet();
   public ResSet copy();
   public Res getRes(int type);
   public void setRes(Res r);
   public void combine();
}
```

Table 2: The new `ResSet` API.

ing to allocate new `ResSet` objects. There is no public constructor, because the API enforces the constraint that a `ResSet` always holds exactly 1 `Res` object for each type of resource.

The `getRes` method return the `Res` object for a given type of resource. The argument is a resource constant defined in the class `Res`. `getRes` throws an `IllegalArgumentException`, if the argument does not represent a valid resource type. The `setRes` method replaces the `Res` object in the set, which has the same resource type as the `Res` object given as argument. `setRes` throws `NullPointerException`, if the argument is `null`.

The `combine` method offers a convenient way to invoke `combine` on all `Res` objects in the set. It raises `IllegalStateException`, if a `combine` operation on a `Res` objects throws an exception. In this case, some `Res` objects in the set may have been combined.

3.4 Class `Seal`

The `Seal` abstraction provides methods for domain creation (unwrapping) and removal (wrapping or disposing). Table 3 summarizes the `wrap` and `unwrap` methods of the `Seal` class. Other methods (e.g., domain disposal) are not shown, because they are not affected by the resource control extension.

The `wrap` method takes the name of the child domain to remove as argument and returns a wrapped representation of the child. If the child is not a leaf node in the hierarchy, all of its subdomains are wrapped recursively as well. However, in this case all information about the resource distribution in the wrapped hierarchy is lost. The `wrap` operation may throw an `InvalidName` exception, if there is no domain with the given name, or a `WrapFailed` exception in case of a serialization failure (the domain is terminated nonetheless).

The `unwrap` method with 3 arguments requires a wrapped representation of the subdomain to resurrect, its name, as well as a `ResSet` object with the resources for the new subdomain. If the wrapped representation is not a leaf node, all domains in the wrapped hierarchy are resurrected and share[2] the resources passed as arguments to `unwrap`. The `unwrap` operation with 2 arguments implicitly shares the resources of the unwrapping domain with the created child domain. `unwrap` throws an `InvalidName` exception, if there is already a subdomain with the given name, an `UnwrapFailed` exception in case of a deserialization failure, or `IllegalStateException`, if there is an invalid `Res` object in the `ResSet` object passed as argument[3].

When a domain is created, the parent's `DOMAINS_ACTIVE` and `DOMAINS_TOTAL` `Res` objects are charged for the created subdomain(s), while the child's resource objects are charged for the CPU time consumed for unwrapping (involving classloading and linking), for memory allocation, as well as for the child's initializer thread.

3.5 Example

The code fragment in table 4 demonstrates the use of the resource control API of the J-SEAL2 kernel. In this example a protection domain (e.g., a sandbox domain controlling the execution of incoming mobile agents) creates 5 subdomain, *childA*, *childB*, *childC*, *childD*, and *childE*. We assume that the creating domain has an assigned CPU share > 0.

While *childA* shares all resources with its parent, the other subdomains receive only limited resources. *ChildB* gets 10% of its parent's CPU share and 1 MB active memory. *ChildB* shares the limits for threads and subdomains with its parent. *ChildC*, *childD*, and *childE* share 20% of the parent's CPU share, 10 MB active memory, a limit of 10 active subdomains, and a total limit of 12 sub-

[2]This limitation does not have much practical significance, because wrapping of deep hierarchies is rarely used. When a domain is wrapped with its children, all affected domains are usually closely related sharing all resources. Furthermore, it is always possible to provide hierarchical wrapping protocols at the user-level, which preserve information about resource distribution.

[3]This can only happen, if an already combined descendant `Res` object is in the `ResSet` object.

```
public class Seal implements Runnable, Serializable {
   public static WrappedSeal wrap(SealName n) throws InvalidName, WrapFailed;
   public static void unwrap(WrappedSeal w, SealName n, ResSet r)
      throws InvalidName, UnwrapFailed;
   public static void unwrap(WrappedSeal w, SealName n)
      throws InvalidName, UnwrapFailed
   {
      unwrap(w, n, ResSet.getCurrentResSet());
   }
   ...
}
```

Table 3: The `wrap` and `unwrap` methods.

domain creations. While *childC* and *childD* share their thread limits with the parent, *childE* (and its subdomains) must not create more than 15 threads. When the subdomains are terminated, the parent combines resources as soon as possible.

4 Implementation

In this section we present the techniques we are using for the implementation of the resource control model discussed in the previous sections. Since accounting for logical resources, such as active and cumulative threads and subdomains, requires only simple modifications to a few J-SEAL2 kernel primitives, we focus on accounting for physical resources, such as memory and CPU consumption.

4.1 No Direct Sharing

Since its initial release the J-SEAL2 kernel is designed to ease the integration of resource control facilities. It guarantees accountability, i.e., user-visible objects belong to exactly one protection domain. References to an object exist only within a single domain[4], i.e., in J-SEAL2 there is no direct sharing of object references between distinct domains. Therefore, it is possible to account each allocated object to exactly one protection domain. This feature not only simplifies resource accounting, but it is also crucial for immediate resource reclamation during domain termination.

[4]The only exception to this rule are `Res` objects (see section 3.2) used for resource sharing.

4.2 Byte-code Rewriting

In our approach we employ byte-code rewriting techniques both for memory and CPU accounting. This is because it is to our understanding the only entirely portable way to implement the needed accounting mechanisms. It is unrealistic to expect the source code of every application to be available for modification. Moreover, if we want guarantees against denial-of-service attacks, we cannot rely on foreign code to perform any voluntary self-limiting operations, whereas if we modify its byte-code before it starts executing, we can 'oblige' it to provide any information needed by the kernel and to obey any restriction imposed on it by the environment. Instead of rewriting byte-code for CPU control, the J-SEAL2 kernel might e.g. ask the underlying operating system for information about the CPU consumption of each thread, but this is possible only when Java threads are directly mapped into operating system threads. Another approach would be to run a modified JVM; the arguments against this are however exposed elsewhere in this paper. A further discussion of existing (and non-portable) approaches is to be found in section 6.1.

In the present report, the byte-code of a Java class is modified before it is loaded by the JVM [16]. Code for memory accounting is inserted before each memory allocation instruction (for details, see section 4.9). CPU accounting uses an abstract measure, the number of executed byte-code instructions. Therefore, code for CPU accounting is inserted in every basic block of code (details are presented in section 4.10).

Rewriting for memory accounting must be done

```
long MB = 1024*1024;

Seal.unwrap(childA, nameOfChildA);

ResSet rP = ResSet.getCurrentResSet();
Res cpu = rP.getRes(Res.CPU_RELATIVE);
long cpuLimit = cpu.getLimit();
Res mem = rP.getRes(Res.MEM_ACTIVE);

ResSet rB = rP.copy();
rB.setRes(cpu.split(cpuLimit/10));
rB.setRes(mem.split(1*MB));
Seal.unwrap(childB, nameOfChildB, rB);

ResSet rCD = rP.copy();
rCD.setRes(cpu.split(cpuLimit/5));
rCD.setRes(mem.split(10*MB));
rCD.setRes(rP.getRes(
  Res.DOMAINS_ACTIVE).split(10));
rCD.setRes(rP.getRes(
  Res.DOMAINS_TOTAL).split(12));
Seal.unwrap(childC, nameOfChildC, rCD);
Seal.unwrap(childD, nameOfChildD, rCD);

ResSet rE = rCD.copy();
Res splitThreads =
  rE.getRes(Res.THREADS_TOTAL).split(15);
rE.setRes(splitThreads);
Seal.unwrap(childE, nameOfChildE, rE);

Seal.dispose(nameOfChildE);
splitThreads.combine();

Seal.dispose(nameOfChildB);
rB.combine();

Seal.dispose(nameOfChildC);
Seal.dispose(nameOfChildD);
rCD.combine();

Seal.dispose(nameOfChildA);
```

Table 4: Resource control example.

before rewriting for CPU accounting, because memory accounting inserts additional byte-code instructions to enforce memory limits, while accounting CPU consumption does not involve any object allocation.

4.3 Domain Types

The resource control model supports trusted domains that have unlimited access to certain types of resources. For performance reasons, the J-SEAL2 kernel does not account for the consumption of these resources. Regarding CPU and memory accounting, we distinguish 4 types of domains:

NO-ACC: Domains without memory limit and without CPU control may execute unmodified Java code, as they do not need to execute any accounting instructions.

CPU-ACC: Domains without memory limit, but with CPU control have to execute CPU accounting instructions. However, code for memory accounting is not required in such domains.

MEM-ACC: Domains with a memory limit, but without CPU control have to execute memory accounting instructions. However, code for CPU accounting is not required in such domains.

CPU-MEM-ACC: Domains with a memory limit and with CPU control have to execute accounting code for memory allocation as well as for CPU consumption.

4.4 Accounting Objects

In MEM-ACC and in CPU-MEM-ACC domains objects of the class `MemAccount` represent memory limit and current usage. In CPU-ACC and in CPU-MEM-ACC domains objects of the class `CPUAccount` maintain CPU consumption. These objects are used only by the J-SEAL2 kernel, they are not accessible by user code. Each thread has associated the `MemAccount` object and a `CPUAccount` object of the domain it is executing in; `null` values indicate that a domain does not need a `MemAccount` or `CPUAccount` object. Java thread-local variables (instances of `java.lang.ThreadLocal`) are used to implement this association. The `MemAccount` and `CPUAccount` classes offer a static method `getCurrentAccount`, which returns the corresponding accounting object of the domain the calling thread is executing in.

Because access to `MemAccount` and above all to `CPUAccount` objects may be extremely frequent, accessing these objects from thread-local variables in every method would cause a significant performance penalty[5]. Therefore, non-native methods are rewritten in order to pass the necessary accounting objects as additional arguments. Native methods are excluded from rewriting, because we cannot account for memory allocated and CPU time consumed by native code. We are relying on modern inter-modular register allocation algorithms implemented by state-of-the-art JVMs to minimize the overhead of passing the accounting objects through the whole method call-graph.

As an example for the rewriting process, consider method `a` given in table 5. The rewritten[6]

```
void a(int x)
{
   b(null, x);
}
```

Table 5: Method `a` before rewriting.

version of method `a` for a CPU-MEM-ACC domain is given in table 6. Here we are only presenting the additional arguments, while the inserted accounting code is discussed in sections 4.9 and 4.10. In

```
void a(int x,
       MemAccount mem, CPUAccount cpu)
{
   b(null, x, mem, cpu);
}
```

Table 6: Method `a` rewritten for a CPU-MEM-ACC domain.

this example, method `a` receives two additional arguments for the `CPUAccount` and `MemAccount` objects[7]. The additional arguments are passed to all invoked methods/constructors (in this example to method `b`).

[5]In Sun's JDK 1.3 implementation thread-local variables are managed as hash-maps, i.e., each access to a thread-local variable requires a hash-map lookup.

[6]For the sake of easy readability, we present rewriting transformations at the Java level, even though the implementation works at the JVM byte-code level.

4.5 Callbacks from Native Code

Native code invoking Java methods complicates the resource control implementation, because the native code is not aware of the accounting objects to be passed to Java methods as extra arguments. The following three scenarios of Java method invocation by native code are particularly important:

- Thread creation: The Java runtime system (native code) invokes the `run` method of a thread object when a thread is started with the aid of the `start` method.
- Static initializers: Static initializers are invoked directly during class-loading, i.e., they are invoked by native code.
- Reflection: The `invoke` method of the class `java.lang.reflect.Method` and the `newInstance` method of the class `java.lang.reflect.Constructor` are native methods.

When the thread invoking a Java method from native code has already set its tread-local accounting objects, it is sufficient to provide for each method an additional one with the same signature, which takes the required accounting objects from thread-local variables and passes them to the rewritten method. In the rewriting example given in tables 5 and 6 we have to supplement the rewritten method with method `a` in table 7. Note that when a constructor is rewritten according to this scheme, the invocation of another constructor of the same class or of a constructor of the superclass has to antecede the lookup of the accounting objects.

However, when a new thread starts executing the `run` method, the thread-local accounting objects have not been initialized yet. As protection domains in J-SEAL2 do not have direct access to the class `java.lang.Thread` but have to employ a safe wrapper class instead [6], the wrapper initializes

[7]Note that in a CPU-ACC or MEM-ACC domain only one additional argument would be necessary to hold the accounting object.

```
void a(int x)
{
   MemAccount mem =
      MemAccount.getCurrentAccount();
   CPUAccount cpu =
      CPUAccount.getCurrentAccount();
   a(x, mem, cpu);
}
```

Table 7: Solving callbacks from native code.

the thread-local accounting variables with the accounting objects of the protection domain the new thread belongs to. These objects are passed to the constructor of the wrapper by the J-SEAL2 kernel.

When a new protection domain is created, the J-SEAL2 kernel allocates a new initializer thread with the accounting objects for the new domain. While starting this thread, the thread wrapper initializes the thread-local accounting variables and starts to load the classes of the new protection domain. The class-loading already happens in the accounting context of the new domain.

4.6 Shared Classes

The J-SEAL2 kernel distinguishes between shared and replicated classes [6]. Shared classes are loaded by the system class-loader (they exist only once in the JVM), while replicated classes, such as the classes of a mobile agent, are loaded by the class-loader of a protection domain (they are reloaded in each domain). All JDK classes as well as most classes from the J-SEAL2 kernel are shared. Certain J-SEAL2 library classes that are frequently used may be shared as well, in order to avoid the overhead of reloading them multiple times.

ln JDK 1.2 it is not possible to load a JDK class with a class-loader different from the system class-loader. Depending on the JVM implementation, certain core JDK classes (e.g., `java.lang.Object`, `java.lang.String`, `java.lang.Throwable`, etc.) are assumed to exist only once in the system. Replicating such classes crashes the JVM. Furthermore, the class-loader API of JDK 1.2 specifies that all classes in the `java` package or in a subpackage thereof can only be defined by the bootstap class-loader. As a consequence, we have the following constraints for accounting for resources used in the JDK:

- All JDK classes are loaded by the system class-loader; there is only a single version of each JDK class.
- Since the same JDK class may be used in different types of domains (NO-ACC, CPU-ACC, MEM-ACC, or CPU-MEM-ACC), JDK classes have to include the accounting code for all domain types.
- The rewriting of JDK classes must be off-line (e.g., during the installation of the J-SEAL2 platform), because JDK classes are always loaded by the system class-loader, which we cannot modify.

The example in table 8 shows how method `a` given in table 5 would be rewritten, if it was defined in a shared class. A method with the same signature as the original method dispatches to the appropriate implementation, when it is invoked from native code. For each type of domain, there is a different method implementation. In this example we distinguished the signature of the NO-ACC implementation from the dispatcher method by adding a dummy argument of type `NoAccount`. The compilers of state-of-the-art JVMs may be able to remove this useless argument.

Alternatively, it is possible to rename the NO-ACC implementation. This approach complicates rewriting, since a table of renamed methods of shared classes has to be maintained, but it has the advantage that replicated classes of trusted domains (e.g., classes of an authenticated, fully trusted agent) can be rewritten very efficiently, because only method signatures in the constant-pool [16] are affected, whereas the method code remains unchanged (in contrast, passing the extra `NoAccount` argument requires additional byte-code instructions).

4.7 Optimizations

Rewriting shared classes as discussed in the previous section increases the code size by more than factor 4. Because the increased code size affects the memory requirements and the startup overhead of the J-SEAL2 kernel (more methods may be compiled), the following optimizations are being implemented:

```
void a(int x)
{
   MemAccount mem =
      MemAccount.getCurrentAccount();
   CPUAccount cpu =
      CPUAccount.getCurrentAccount();
   if (cpu == null)
      if (mem == null) a(x, null);
      else a(x, mem);
   else
      if (mem == null) a(x, cpu);
      else a(x, mem, cpu);
}

void a(int x, NoAccount dummy)
{
   b(null, x, null);
}

void a(int x, CPUAccount cpu)
{
   b(null, x, cpu);
}

void a(int x, MemAccount mem)
{
   b(null, x, mem);
}

void a(int x, MemAccount mem,
              CPUAccount cpu)
{
   b(null, x, mem, cpu);
}
```

Table 8: Rewriting methods in shared classes.

- A leaf method, which does not allocate any objects, requires neither MEM-ACC nor CPU-MEM-ACC implementations (i.e., the NO-ACC implementation can be used in MEM-ACC domains, and CPU-MEM-ACC domains can employ the CPU-ACC implementation).
- As a generalization of this optimization, we do not need to provide MEM-ACC and CPU-MEM-ACC implementations for methods without any memory allocation instructions, if they invoke only methods satisfying the same condition.
- We can optimize the code of shared kernel classes by hand in order to minimize the overhead for resource control (e.g., when allocating a set of objects, we can account for the total size of these objects at once).

4.8 Rewriting Abstract Methods

There are 2 different approaches for dealing with abstract Java methods (including interface methods) in shared types (classes or interfaces):

1. Abstract methods are not rewritten. This approach simplifies the rewriting process, but invoking a method on a variable of a (static) type, in which the called method is declared as abstract (e.g., interface method call), incurs high overhead, because the dispatching method has to access the accounting objects from thread-local variables.
2. Abstract methods are rewritten. For each abstract method in a shared class, the type signatures of the 4 possible implementations (NO-ACC, CPU-ACC, MEM-ACC, CPU-MEM-ACC) are added. As a result, a class implementing the abstract method has to provide all 4 implementations, even if the implementing class is a replicated one (in this case, 3 implementations may be dummies). This approach allows to pass the accounting objects directly to the invoked method, no matter whether it is an interface method or not.

The J-SEAL2 implementation follows the second approach, because method calls on interface types are very frequent in Java programs. Thus, we can avoid the invocation of the dispatcher method.

4.9 Memory Control

Enforcing memory limits requires exact pre-accounting for memory resources, i.e., an overuse exception is raised before a thread can exceed the memory limit of the domain it is executing in. In contrast to JRes [9], which maintains a separate memory limit for each thread, J-SEAL2 enforces a single memory limit for a multithreaded domain or even for a set of domains in the case of resource sharing.

4.9.1 Class `MemAccount`

Because a single `MemAccount` object has to maintain the memory consumption and limit of a set of domains sharing the same memory resources, access to the `MemAccount` must be synchronized. Furthermore, accounting for an object as well as its allocation and initialization has to be an atomic action.

Before the object is allocated, J-SEAL2 ensures that the memory limit is not exceeded and updates the `MemAccount`. If the memory allocation fails (i.e., throwing a `java.lang.OutOfMemoryError`), if the constructor raises an exception, or if the allocating thread is terminated asynchronously (e.g., termination of the protection domain the thread is executing in), we have to ensure that the modification of the `MemAccount` is undone. Otherwise, other threads or even other domains (using the same `MemAccount`) could suffer from memory leakage.

The J-SEAL2 kernel offers a special kernel mode of execution, which prevents a thread from being terminated asynchronously (details about the kernel mode can be found in [6]). However, since entering and leaving kernel mode are rather expensive operations and because memory allocation occurs frequently in object-oriented programs, J-SEAL2 employs a sophisticated rewriting scheme for memory allocation instructions in order to avoid entering kernel mode.

When the garbage collector reclaims some objects, the `MemAccount` that has been charged for these objects must be updated. For this reason, the `MemAccount` maintains a weak reference for each allocated object, which does not prevent the object from being reclaimed. When an object referenced by a weak reference is garbage collected, the weak reference is enqueued in a reference queue, which can be polled by the `MemAccount` implementation. J-SEAL2 uses `AccountRef`, a subclass of `java.lang.ref.WeakReference`, in order to store information about the size of an object. The simple implementation of `AccountRef` is depicted in table 9.

Table 10 shows some parts of the `MemAccount` implementation[8]. The `MemAccount` maintains memory limit and usage, as well as a reference queue and a reference set. The reference set is used in order to maintain strong references to the weak reference objects. It is a highly optimized set implementation, which handles an asynchronous `java.lang.ThreadDeath` exception gracefully (i.e., the set always remains in a consistent state).

The `checkLimit` method ensures that a memory allocation does not exceed the `MemAccount` limit. If there are not enough memory resources available, `checkLimit` polls the reference queue in order to check for reclaimed objects[9]. If the reference queue returns weak references that are also in the reference set, the memory usage of the `MemAccount` object is reduced (discussing the rewriting algorithm for memory allocation sites, we will see that in case of an exception a weak reference may be allocated, but not inserted into the set). If there are still not enough free memory resources after polling the reference queue, `checkLimit` forces the garbage collector to run. In case the garbage collector is not able to reclaim enough objects belonging to the `MemAccount` object, a `ResourceOveruseException` is raised.

Since polling the reference queue and running the garbage collector are kernel-level operations (they affect the internal state of the JVM), they are executed in kernel mode. In line 12 the `Kernel.lockCond` operation enters kernel mode, unless the calling thread is already executing in kernel mode. Details about the conditional kernel lock primitive are discussed in [6]. In line 30 the `Kernel.unlock` operation returns back to user mode (if `checkLimit` has been invoked in user

[8]For instance, we omitted the static `getCurrentAccount` method mentioned in section 4.4.

[9]In order to prevent the reference queue and the reference set from becoming too large, the `checkLimit` method should poll the queue periodically, even though the memory limit is not exceeded. To simplify matters, we omitted this detail in the given code fragment.

```
public final class AccountRef extends WeakReference {
   public final int size;

   AccountRef(Object referent, ReferenceQueue q, int size)
   {
      super(referent, q); this.size = size;
   }
}
```

Table 9: The `AccountRef` implementation.

```
public final class MemAccount {
   public long limit;
   public long usage = 0;
   public final ReferenceQueue refQueue = new ReferenceQueue();
   public final ReferenceSet refSet = new ReferenceSet();

   public MemAccount(long limit) { this.limit = limit; }

   public void checkLimit(int size)
   {
      if (usage + size > limit) { // not enough resources
         boolean locked = Kernel.lockCond();
         try {
            AccountRef ref;
            while ((ref = (AccountRef)refQueue.poll()) != null) {
               if (refSet.remove(ref)) // check whether ref was in the set
                  usage -= ref.size;
            }
            while (usage + size > limit) { // still not enough resources
               System.gc(); // force the gc to run
               long oldUsage = usage;
               while ((ref = (AccountRef)refQueue.poll()) != null) {
                  if (refSet.remove(ref)) // check whether ref was in the set
                     usage -= ref.size;
               }
               if (usage == oldUsage) // gc was not successful
                  throw new ResourceOveruseException();
            }
         } finally {
            if (locked) Kernel.unlock();
         }
      }
   }
   ...
}
```

Table 10: The `MemAccount` implementation.

mode).

4.9.2 Rewriting Algorithm

Table 11 demonstrates how a memory allocation site is rewritten. This rewriting scheme is based on the following assumptions:

- An asynchronous `java.lang.ThreadDeath` exception is thrown at most once in each thread. This property is guaranteed by the J-SEAL2 kernel.
- An operation requiring exclusive kernel mode [6] must not synchronize on an accounting object. Otherwise, a deadlock could arise (before `checkLimit` is called, a lock on the accounting object is obtained; during the execution of `checkLimit`, non-exclusive kernel mode may be entered).
- The state of a `ReferenceSet` object remains consistent, even if an operation is stopped asynchronously.

We are using a boolean flag `undo` in order to indicate whether a modification to the `MemAccount` object must be undone. In lines 7–15, the memory limit is enforced and the `MemAccount` object is updated. Updating the `MemAccount` object and setting the `undo` flag is implemented as an atomic action: both operations are idempotent, thus we simply repeat these operations in case of an asynchronous exception. In line 15 the lock on the `MemAccount` object is released, because in line 16 object initialization is user code and might block arbitrarily. In line 17 a weak reference to the allocated object is created. If an exception occurs in lines 16–17, the weak reference is not inserted in the reference set, and the exception handlers in lines 25–46 are responsible to undo the modification of the `MemAccount` object. In lines 18–24 the weak reference is inserted into the reference set and the `undo` flag is cleared as an atomic action.

Although the transformation of memory allocation sites shown in table 11 increases the code size significantly, most parts of the code are rarely executed. In the common case no exception occurs, thus the exception handlers, representing the bigger part of the code, are not activated. Moreover, in the common case, when the memory limit is not exceeded, the `checkLimit` method returns immediately without obtaining any kernel locks. In order to reduce the code size, exception handlers may be reused, if multiple memory allocation instructions occur in the same method.

4.9.3 Object Size

The size of an object is calculated from the number of fields for each Java basic type, the number of fields holding object references, a constant for the object overhead, as well as a constant for the accounting overhead (i.e., the size of an `AccountRef` object, as well as the space used in the reference queue and in the reference set). For arrays, the actual size must be computed from the array dimensions available on the execution stack. Depending on the Java runtime system, the overhead for array objects may be larger than for non-array objects, because of the size information stored within arrays.

Constants for the object overhead and for the size of Java basic types and object references are managed in a configuration file by the system administrator. Since in general the administrator does not know the object representation of the underlying Java runtime system, a tool helps to approximate these constants (e.g., by avoiding garbage collection and measuring the difference of allocated memory before and after creating certain types of objects). However, object alignment is not taken into account.

4.9.4 Optimizations

While our approach works for objects as well as for arrays, we are also implementing an optimization for non-array objects: Similar to JRes [9], in each allocated object we store a reference to the corresponding `MemAccount` object. Rewritten finalizers are responsible for updating the `MemAccount` when an object is reclaimed by the garbage collector. Thus, we can avoid the significant overhead of maintaining weak references, which is particularly important for small objects.

For arrays, we cannot see any portable alternative. However, in practice the overhead for accounting for allocated arrays is not a serious problem, because arrays frequently are large objects (compared to the accounting overhead they cause).

```
MemAccount mem; // the accounting object passed as a method argument
Object o; // the object to allocate
int size; // the object size including overhead of deallocation information
boolean undo = false; // undo update in case of an exception
try { // only ThreadDeath possible (at most once!)
   try { // any exception possible (object allocation and initialization)
      synchronized (mem) {
         mem.checkLimit(size); // might throw ResourceOveruseException
         long newUsage = mem.usage + size;
         try { // only ThreadDeath possible (at most once!)
            mem.usage = newUsage; undo = true; // idempotent operations
         } catch (Throwable t) { // no exception possible
            mem.usage = newUsage; undo = true; throw t;
         }
      } // unlock the MemAccount, because object initialization is user code
      o = new ...; // allocation and initialization (user code!)
      AccountRef ref = new AccountRef(o, mem.refQueue, size);
      synchronized (mem) {
         try { // only ThreadDeath possible (at most once!)
            mem.refSet.add(ref); undo = false; // idempotent operations
         } catch (Throwable t) { // no exception possible
            mem.refSet.add(ref); undo = false; throw t;
         }
      }
   } catch (Throwable t) { // only ThreadDeath possible
      if (undo) {
         synchronized (mem) {
            long oldUsage = mem.usage - size;
            try { // only ThreadDeath possible (at most once!)
               mem.usage = oldUsage; undo = false; // idempotent operations
            } catch (Throwable t2) { // no exception possible
               mem.usage = oldUsage; undo = false; throw t2;
            }
         }
      }
      throw t;
   }
} catch (Throwable t) { // only ThreadDeath possible, but in this case the
                        // previous handler already undid the update
   if (undo) { // if undo == true, no more exception possible
      synchronized (mem) {
         mem.usage -= size;
      }
   }
   throw t;
}
```

Table 11: Rewriting a memory allocation site.

4.10 CPU Control

For CPU control, we are accounting the number of executed byte-code instructions for each thread running in a CPU-ACC or CPU-MEM-ACC domain. A high-priority scheduler thread, which is part of the J-SEAL2 kernel, executes periodically in order to ensure that assigned CPU limits are respected. The scheduler thread calculates the number of executed byte-code instructions for each set of domains sharing a CPU limit by summing up the CPU consumption of all threads executing in a domain in the set. The scheduler compares the number of executed byte-codes with the desired schedule. If a set of domains has exceeded its CPU limit, the priorities of threads executing in these domains are lowered.

4.10.1 Class `CPUAccount`

In contrast to a `MemAccount` object, which is shared by all threads executing in a domain with memory accounting, each thread running in a domain with CPU accounting has associated its own `CPUAccount` object. Since CPU accounting occurs very frequently, it is important that multiple threads do not have to synchronize on a common accounting object. As only the scheduler thread makes any scheduling decisions, it is sufficient to account for each thread separately. The scheduler is responsible for accumulating the accounting data of all threads executing in a set of domains sharing a CPU limit.

A `CPUAccount` object simply maintains an integer counter, which is updated by the thread owning the object. Table 12 shows some parts of the `CPUAccount` implementation[10]. Because the scheduler thread has to read the counter value, we are using a volatile variable in order to force the JVM to immediately propagate every update from the working memory of a thread to the master copy in the main memory [14, 16].

```
public final class CPUAccount {
   public volatile int usage;
   ...
}
```

Table 12: The `CPUAccount` implementation.

In general, updating the counter requires loading the `usage` field of the `CPUAccount` object from memory (it is volatile), incrementing the loaded value accordingly, and storing the new value in the memory. A counter update requires about 6 byte-code instructions.

4.10.2 Scheduler

In this section we describe how the scheduler thread computes the CPU consumption of a set of domains, and how it employs different JVM priority levels in order to prevent CPU overuse. However, we do not present a particular scheduling algorithm, because we are still experimenting with different policies.

For each `CPUAccount` object, the scheduler thread always stores the value of the counter it has read most recently. The scheduler calculates the difference between the current value and the previously stored value in order to determine the amount of byte-code instructions executed during the last time-slice (because of the lack of synchronization, the scheduler must not reset any `CPUAccount` object). If a thread has not existed before, the scheduler assumes the previously stored value to be zero. When a thread terminates, its `CPUAccount` object is not disposed of immediately, but it is maintained until the scheduler has examined it.

The scheduler has to deal with an overflow in the counter of a `CPUAccount` object. The size of the counter must be large enough so that its full range cannot be used in a single time-slice. For current JVMs and a reasonably small time-slice, a Java `int` is sufficient. However, in future high-performance systems, `CPUAccount` objects may have to maintain `long` values[11].

We are using different JVM priority levels to control the CPU consumption of individual domains. As protection domains in J-SEAL2 do not have direct access to the class `java.lang.Thread` (they have to use a safe wrapper class instead [6], which does not offer any mechanism to change the priority of a thread), an user-level thread cannot raise its own priority.

[10]For instance, we omitted the static `getCurrentAccount` method mentioned in section 4.4.

[11]For a `long` variable, the volatile declaration is crucial, because the JVM does not treat non-volatile `long` values atomically [16].

Even though the Java language specification [14] does not define any scheduling policy, current JVM implementations respect assigned thread priorities. Many JVMs employ fixed priority scheduling, where a low-priority thread cannot execute, if there is a high-priority thread ready to run. The J-SEAL2 kernel uses the distinct JVM thread priority levels as follows:

- MAX_PRIORITY: The maximum priority is reserved to JVM internal tasks, such as handling weak references. J-SEAL2 does not run any threads with the maximum priority.
- MAX_PRIORITY-1: J-SEAL2 uses this priority level for kernel-level operations in order to prevent priority inversion, i.e., when a high-priority thread is waiting for an exclusive kernel lock (see [6]) because of a low-priority thread T executing in kernel mode, the priority of T is temporarily boosted until thread T releases the kernel lock.
- MAX_PRIORITY-2: This priority level is used by the J-SEAL2 scheduler thread.
- NORM_PRIORITY–MIN_PRIORITY[12]: The scheduler assigns these priority levels to threads according to the CPU consumption of the corresponding domain and the assigned CPU share. Threads executing in NO-ACC or in MEM-ACC domains are always assigned NORM_PRIORITY. If a domain exceeds its CPU limit, the priorities of its threads are reduced (or at least the priorities of those threads overusing the CPU). If a domain does not consume its assigned CPU resources, the priorities of its threads may be increased again (but never exceeding NORM_PRIORITY). We are experimenting with different scheduling algorithms regarding the history of CPU consumption.

4.10.3 Rewriting Algorithm

In the description of the rewriting algorithm we use the following definition of an accounting block, which is related to the concept of a basic block of code. In order to minimize the accounting overhead, we are considering blocks of maximal length. An accounting block is a byte-code sequence fulfilling the following constraints:

- If a byte-code instruction, which is neither a method/constructor invocation nor a JVM subroutine invocation, changes the control-flow non-sequentially (e.g., method return, exception raising, branch, JVM subroutine return, etc.), it must be the last instruction in the accounting block. That is, with the exception of method/constructor and JVM subroutine invocations, only the last byte-code instruction in the block may change the control-flow non-sequentially. A method invocation does not terminate an accounting block, because otherwise the average block size would be reduced significantly, as method invocations are very frequent in object-oriented programs.
- Only branches to the begin of the block are allowed. There is no byte-code instruction branching to another instruction in the same method, which is not the first one in its block. Furthermore, the first instruction of an exception handler must be always the first instruction in its block.

The byte-code rewriting algorithm involves the following 4 steps (an efficient implementation may perform multiple steps together):

1. Method/constructor invocations are rewritten in order to pass the `CPUAccount` object as extra argument. Because the `CPUAccount` is always the last argument[13], it can be pushed onto the stack immediately before the method/constructor invocation instruction.
2. An accounting block analysis (similar to a basic block analysis in traditional compilers) partitions the method code into a set of accounting blocks. Each block has an attribute indicating the accounting size of the block. Initially, this attribute holds the number of byte-

[12]In this description we assume that NORM_PRIORITY $<$ MAX_PRIORITY-2.

[13]Since rewriting for memory accounting is done before rewriting for CPU control, the `MemAccount` argument is passed always before the `CPUAccount` object.

code instructions in the block[14]. Furthermore, a control-flow graph with the accounting blocks as nodes has to be constructed, if optimizations are to be performed in order to minimize the accounting overhead. Without any optimizations, accounting instructions have to be inserted into every block.

3. Optimizations, such as those presented in following section, analyze the control-flow graph in order to detect situations where acounting for multiple different blocks may be combined. The optimizations may decrement the accounting size attribute of one block and add it to the accounting size of another block. If the accounting size of a block becomes zero, it does not require any accounting instructions.

4. For every block with a positive accounting size, accounting instructions are inserted at the begin of the block. The only exception to this rule is the first block in a constructor: The invocation of another constructor of the same class or of the superclass has to antecede the accounting code. The included instructions add the accounting size of the block plus the number of inserted accounting instructions to the `CPUAccount` object. For performance reasons, updates of the `CPUAccount` object are not synchronized.

This approach ensures that a thread is charged for at least the number of byte-code instructions it executes. For each accounting block, a thread is charged for the number of instructions in the block, before it executes these instructions (pre-accounting). When an instruction, which is not the last one in its accounting block, raises an exception, the thread has been charged for more instructions than it has consumed. However, since the number of executed byte-code instructions is only an approximation of the exact CPU consumption, and because exception handling is expensive on many JVM implementations, this possible inexactness does not pose any problem.

[14]In order to improve the accuracy of measurement, the J-SEAL2 administrator may configure a weighting of byte-code instructions (integer values) according to their complexity. To simplify matters, we assume that all byte-code instructions have a weighting of 1.

4.10.4 Optimizations

In order to minimize the accounting overhead, the rewriting algorithm may perform certain optimizations. If classes are rewritten off-line, such as shared JDK classes (see section 4.6), the optimization algorithm may perform some complex and time-consuming analysis. However, for replicated classes, only simple optimizations are possible, since these classes are rewritten on-line. In the following paragraphs we present some simple rules that are well suited for on-line optimization.

In the following optimization O1 we assume that the accounting block B has n ($n > 0$) predecessors A_i ($1 \leq i \leq n$) in the control-flow graph. We denote the accounting size attributes of B and A_i as b and a_i.

O1 If all A_i are different from B, and for each A_i the only successor is B, then all a_i are incremented by b and b is set to zero.

For the following optimizations O2 and O3 we assume that the accounting block A has n ($n > 0$) successors B_i ($1 \leq i \leq n$) in the control-flow graph. We denote the accounting size attributes of A and B_i as a and b_i, the minimum accounting size $\min b_i$ as b_{min}, and the maximum accounting size $\max b_i$ as b_{max}.

O2 If all B_i are different from A, and for each B_i the only predecessor is A, then a is incremented by b_{min} and all b_i are decremented by b_{min}. Consequently, the value of at least one b_i becomes zero.

O3 If all B_i are different from A, and for each B_i the only predecessor is A, and the difference $b_{max} - b_{min}$ does not exceed a given threshold T, then a is incremented by b_{max} and all b_i are set to zero. Less formally: If the values of the accounting size attributes of successor blocks are not too much different, the common predecessor block accounts for the longest successor block. This optimization is an aggressive version of rule O2. The threshold controls the aggressiveness of this optimization. A threshold T means that a thread executing a block B_i may be charged for up to T byte-code instructions, which it did not execute. In general, T should not be smaller than the number of byte-code instructions necessary to update the `CPUAccount` object (a thread would be charged for the update instructions, if the optimization was not applied). In order to find effective values for the threshold, we can perform static analysis of typical Java programs (the smallest value T allowing to avoid a significant fraction of the accounting code).

The optimization rules O1, O2, and O3 aim at combining the accounting for a set of blocks that represent conditional statements, but they do not allow to remove the accounting code from loops. For instance, rules O1 and O2 (or alternatively, O1 and O3) may be applied to optimize the accounting for `if-else` statements. However, these rules are not sufficient to reduce the accounting overhead for `if` statements without a matching `else`. Therefore, we are working on further optimizations.

In general, multiple optimization rules can be applied to a given control-flow graph. The order of application is important, since it may affect the quality of the accounting code. Most importantly, the optimization algorithm must ensure termination. In particular, certain loops allow an infinite application of rule O1. The following heuristics help to guide the optimization process:

- An optimization rule may be applied only if the application increases the number of blocks with an accounting size attribute of zero. Since the number of blocks in a method is finite, obeying this rule ensures termination of the optimization algorithm.
- Optimization O1 shall be applied before optimizations O2 and O3.
- Optimization O3 shall be applied before optimization O2. There is no need to apply optimization O2, if optimization O3 (which is more aggressive) succeeds on a certain node in the control-flow graph.
- If there are leaf nodes in the control-flow graph, they should be considered first, afterwards their predecessor nodes, etc.

While optimizations O1, O2, and O3 aim at removing accounting code from certain blocks, the following rule O4 helps to reduce the overhead of accounting by caching the counter maintained by the `CPUAccount` in a local variable. This optimization improves performance only for certain JVM implementations (measurements are given in section 5). Optimization O4 must be considered after application of the rules O1, O2, and O3.

O4 In general, a block with a positive accounting size requires accounting instructions to load, update, and store the `usage` field of the `CPUAccount` object (see section 4.10.1). We introduce a local variable `localUsage` caching the value of the `usage` field in order to avoid reloading this field in every accounting block. The following algorithm marks exactly those accounting block that have to reload the `usage` field of the `CPUAccount` object. All other blocks may directly update the `localUsage` variable and propagate the new value to the `usage` field of the `CPUAccount` object.

- Initially, we mark the first block in the method, in each JVM subroutine, and in each exception handler.
- If a block contains a method/constructor invocation, all of its successors in the control-flow graph are marked.
- If a block with an accounting size attribute of zero is marked, all of its successors have to be marked as well.

The algorithm terminates, if no further blocks can be marked.

4.11 Accounting for Garbage Collection

In order to prevent denial-of-service attacks by causing the garbage collector to consume a considerable amount of CPU time (e.g., an attacker may create a lot of garbage without exceeding its memory limit), the J-SEAL2 kernel has to account for the time spent by the garbage collector. Only CPU-MEM-ACC domains can be charged for the garbage they produce, because accounting for garbage collections requires the information, which domain has allocated a certain object (such information is not available in NO-ACC or CPU-ACC domains), and because the time spent by the garbage collector affects the CPU consumption of a domain (CPU consumption is not measured in NO-ACC or MEM-ACC domains).

Since the exact CPU time spent by the garbage collector is not known, we are using an abstract measure. The J-SEAL2 administrator defines a rough approximation of the number of byte-code instructions required to reclaim an object.

A simple solution is to charge the `CPUAccount` object of a thread, before it allocates an object. That is, a domain has to 'pay' for the garbage it eventually will produce at the time it 'buys' an object. This approach has the advantage that a CPU-MEM-ACC domain is charged for all garbage it produces, even if the domain has already terminated when some objects are reclaimed.

In a more complicated approach the `CPUAccount` object of a thread is charged during execution of the `checkLimit` method of the class `MemAccount`, whenever a weak reference is obtained from the reference queue (table 10, lines 15 and 22). The signature of the `checkLimit` method has to be extended to take an additional `CPUAccount` argument. Furthermore, the `CPUAccount` object may be charged additionally for each time a garbage collection is initiated explicitly (table 10, line 20).

However, the second solution has the drawback that domains are not charged for the garbage they leave in the system after termination (e.g., a malicious mobile agent may allocate as many objects as its memory limit allows and migrate to another site before it is charged for the garbage). In addition to this, the second approach is difficult to implement, when finalizers, which are invoked by unknown garbage collector threads, are rewritten to perform accounting tasks as discussed at the end of section 4.9. For these reasons, the J-SEAL2 kernel implements the first solution.

4.12 Compensating for Native Code

With the aid of byte-code rewriting techniques, it is not possible to account for memory allocation and CPU consumption in native code. Untrusted applications are not allowed to bring native code libraries into the system. Concerning JVM-provided standard operations, the J-SEAL2 kernel tries to compensate for resources used by native code and prevents untrusted domains from using certain functionality leading to a significant resource consumption by native code. In the following we describe some important cases of resource consumption in native code and how J-SEAL2 solves them:

- Class-loading: The Java runtime system manages an internal table of loaded classes. Memory for compiled methods is allocated by the Just-in-Time compiler, which is usually implemented in native code. However, the set of classes untrusted domains (e.g., mobile agents) are allowed to access is limited and known to the J-SEAL2 kernel. Therefore, the kernel accounts for the classes using an approximation, which is proportional to the size of the class files.

- Deserialization: J-SEAL2 uses Java serialization in order to create messages to be transferred across domain boundaries. When the receiving domain opens a message, it is being deserialized using the class-loader of the receiving domain to resolve class names. The class `java.io.ObjectInputStream` employs native methods to allocate objects without invoking their constructors. J-SEAL2 solves this hurdle by storing the amount of objects for each type, which is part of the serialized object graph, in the message. The receiver performs resource checks before deserializing the message. Note that we are not directly storing the size of the message, because the message may be deserialized on a different host using a JVM with a distinct object representation (e.g., consider a mobile agent carrying a message to another location).

- Object cloning: Java supports a way to create a shallow copy of an object of a type implementing the interface `java.lang.Cloneable`. The shallow copy is allocated by a native method. A simple solution is to forbid untrusted domains to use object cloning. Another somewhat more complicated approach is to rewrite invocations of the `clone` method accordingly.

- Reflection: The Java reflection API provides a mechanism to indirectly create a new instance of a class. The `newInstance` method of the class `java.lang.reflect.Constructor` is native. J-SEAL2 prevents untrusted domains from using the reflection API. However, note that object allocated by a constructor invoked with the aid of the `newInstance` method would be accounted for (see section 4.5).

5 Evaluation

Because the implementation of our resource control model in J-SEAL2 is work in progress, we are currently not able to provide performance and scalability evaluations of real applications running in a J-SEAL2 environment with resource control. Nevertheless, in this section we present some performance measurements proofing that the overhead due to accounting is acceptable on modern JVM implementations.

While in J-SEAL2 the overhead for memory control is comparable to the overhead caused by JRes[15] [9], the overhead of CPU control based on byte-code rewriting techniques has to be examined carefully, because such an approach has not been used before.

JRes [9] uses native code for CPU accounting, although the authors mention that CPU accounting could be accomplished with the aid of byte-code rewriting techniques. The authors argued that the resulting execution time would be prohibitive when a reasonable degree of accuracy was to be achieved. However, our initial performance measurements show that the overhead due to our completely portable implementation of CPU accounting is not prohibitive on modern JVM implementations[16]. We measured the following well-known benchmark programs:

Fib: This is the recursive algorithm for the calculation of fibonacci numbers. We used this benchmark to calculate the 35th fibonacci number.

Sort: This is bubble-sort, an iterative sorting algorithm for arrays. It consists of 2 nested loops, where the inner loop exchanges 2 adjacent array elements, if they are not in the desired order. We used this benchmark to sort an array of 10000 `int` values in ascending order. Initially, the input array was sorted in descending order.

Table 13 summarizes our measurements, which were collected on a Windows NT 4.0 workstation (Intel Pentium II, 400MHz clock rate) with 5 different JVM implementations. In order to minimize the impact of compilation and garbage collection, all results represent the median of 5 different measurements. For each measurement, table 13 shows the execution time of the benchmark in milliseconds, as well as the speedup of the original code compared to the rewritten version. We measured code rewritten without any optimizations, as well as the code resulting from the application of the optimizations rules presented in section 4.10.4.

The fibonacci benchmark consists of 5 accounting blocks. Optimization O3 with a minimum threshold $T_{\mathrm{m}in} = 11$ allows to combine accounting for the whole method in the first block, i.e., this optimization avoids 80% of the accounting code. Because the optimized fibonacci method accounts for all instructions in the first block of the method, optimization rule O4 cannot reduce the accounting overhead anymore.

Our measurements for the recursive fibonacci method show that our optimizations allow to reduce the accounting overhead to 12–19% on modern JVM implementations, such as Sun's Hotspot VMs and IBM's Classic VM. These results also indicate that the overhead of passing the additional `CPUAccount` argument is reasonably small.

The bubble-sort method comprises 10 accounting blocks. A combination of the optimization rules O1 and O3 with a minimum threshold $T_{\mathrm{m}in} = 6$ allows to remove the accounting code from 5 blocks, i.e., these optimizations avoid 50% of the accounting code. Furthermore, the optimization O4 can be applied to all accounting blocks but the first one in the method.

The bubble-sort benchmark shows that optimization rule O4 is beneficial only on some JVM implementations, such as Sun's and IBM's Classic VMs, whereas on Sun's Hotspot VMs this rule has a bad impact on the performance. While the performance on IBM's Classic VM suffers significantly from the volatile `CPUAccount` counter (removing the volatile declaration reduces the accounting overhead drastically), the performance impact of the volatile vari-

[15]For an application allocating a new object every 250 byte-code instructions, the overhead for memory control is less than 18%, if no memory limit is exceeded.

[16]We are not measuring the overhead for CPU control incurred by the scheduler, as it can always be kept small by choosing an appropriate time-slice.

		Sun JDK 1.3				Sun JDK 1.2.2				IBM JDK 1.3	
		Classic (Interpreter)		Hotspot Client		Classic (JIT)		Hotspot Server 2.0		Classic (JIT)	
Fib	original	13029	(1,00)	1542	(1,00)	1031	(1,00)	1032	(1,00)	991	(1,00)
	rewritten	22933	(1,76)	2123	(1,38)	1773	(1,72)	1502	(1,46)	1522	(1,54)
	O3	16825	(1,29)	1732	(1,12)	1432	(1,39)	1232	(1,19)	1131	(1,14)
Sort	original	16954	(1,00)	1812	(1,00)	1212	(1,00)	1352	(1,00)	782	(1,00)
	rewritten	44344	(2,62)	2774	(1,53)	2434	(2,01)	2564	(1,90)	2323	(2,97)
	O1+O3	35351	(2,09)	2423	(1,34)	1752	(1,45)	2143	(1,59)	1623	(2,08)
	O1+O3+O4	34650	(2,04)	2633	(1,45)	1513	(1,25)	2294	(1,70)	1543	(1,97)

Table 13: Benchmarks measuring the overhead for CPU accounting.

able is rather small on Sun's Hotspot VMs. Nevertheless, for our benchmark programs, IBM's JVM implementation offers the best overall performance in absolute terms.

6 Related Work

We distinguish two broad categories of related work on adding resource control to Java: those which have security as main objective, and those which follow other motivations.

6.1 Resource Control for Security Purposes

Compared to existing proposals for realizing resource control in Java, we broadly differentiate our approach in two ways: first, whether the model supports a process-based approach, with well-defined domain boundaries and resource allocation for each application, and, second, to which extent the implementation is portable or not.

JRes [9] is a resource control system which takes CPU, memory, and network resource consumption into account. The resource management model of JRes works at the level of individual Java threads; in other words, there is no notion of application as a group of threads, and the implementation of resource control policies is therefore cumbersome. JRes is a pure resource accounting system and does not enforce any separation of domains; covering this other aspect is the goal of J-Kernel [25], a complementary project of the same research team. For its implementation, JRes does not need any modification to the JVM, but relies on a combination of byte-code rewriting and native code libraries. To perform CPU accounting, the approach of JRes is to make calls to the underlying operating system, which requires native code to be accessed[17]. For memory accounting, it essentially uses byte-code rewriting, but still needs the support of a native method to account for memory occupied by array objects. Finally, to achieve accounting of network bandwidth, the authors of JRes also resort to native code, since they swapped the standard `java.net` package with their own version of it.

KaffeOS [1] is a Java runtime system which supports the operating system abstraction of *process* to isolate applications from each other, as if they were run on their own JVM. Thanks to KaffeOS, a modified version of the freely available Kaffe virtual machine [26], it is possible to achieve resource control with a higher precision than what is possible with byte-code rewriting techniques, where e.g. memory accounting is limited to controlling the respective amounts consumed in the common heap, and where CPU control does not account for time spent by the common garbage collector working for the respective applications. The KaffeOS approach should by design result in better performance, but is however inherently non-portable. This means that optimizations found in compilers and standard JVMs are not benefited from: in a recent publication [2] the authors report that, in absence of denial-of-service attack, IBM's compiler and JVM [18] is 2–5 times faster than theirs.

Developed by the same team as KaffeOS, Alta [23] is a prototype based on the Fluke hierarchical process model, and implemented on the Kaffe virtual machine. The main differences with KaffeOS

[17]More precisely, CPU accounting in JRes is based on native threads, a feature not supported by every JVM.

are that a single garbage collector is responsible for all applications, and that Alta entirely respects the hierarchical process model of Fluke by providing resource control APIs, whereas KaffeOS only retains a more implicit nested CPU and memory management scheme.

Many other systems are proposed in the literature, but none of them are as complete as JRes, Alta, and KaffeOS. An excellent recent overview is provided in [3]. To summarize, we might say that J-SEAL2 proposes a protection model inspired both from Alta and J-Kernel, and a memory accounting implementation that is more reminiscent of JRes. For all other aspects, J-SEAL2 however clearly constitutes an independent effort.

6.2 Other Java-centric Approaches to Resource Control

There are several lines of research, where environments and analysis tools have been designed that can be exploited more or less with the same objectives as exposed in this article.

The Real-Time for Java Experts Group [7] has published a proposal to add real-time extensions to Java. One important focus of this work is to ensure predictable garbage collection characteristics in order to meet real-time guarantees. For instance, the specification provides for several memory management schemes, such as areas with limited lifetime or bounded allocation rates, which could be implemented – or at least simulated – with the J-SEAL2 extensions described in the present article. Another real-time system, PERC [17], extends Java to support real-time performance guarantees. To this end, the PERC system analyzes Java byte-codes to determine memory requirements and maximal execution times, and feeds that information to a real-time scheduler. The objective of real-time systems is to provide precise guarantees e.g. for worst-time execution; our focus, on the other hand, is on computing approximated resource consumptions in order to prevent denial-of-service attacks. We are more interested in the relative values of applications, and less in absolute figures. This is confirmed by the fact that we are not trying to estimate their real CPU consumption, but rather to compare the respective number of executed byte-codes.

Profilers constitute another class of tools that have many things in common with resource control: both intend to gather information about resource usage. Profilers however are designed to help developers optimize the efficiency of their applications, and not to externally control their resource consumption. The Java Virtual Machine Profiling Interface (JVMPI) [21] is an API created by Sun; it is a set of hooks to the JVM which signals interesting events like thread start and object allocations. Java Usage Monitor (JUM) [10] is a tool which builds upon JVMPI to help the developer determining how much CPU is consumed by the different threads of an application and how much memory they use. JUM needs native code to obtain information from the underlying operating system about how CPU time is allocated, and is therefore not portable. Interestingly, JUM is able to also account for objects allocated by native code. However, JUM is not able to enforce memory limits. While J-SEAL2 allows for exact pre-accounting of memory resources, where an overuse exception is generated before a thread exceeds its memory limit, a resource control mechanism based on JUM can only react after a memory overuse is detected. In addition to these limitations, JVMPI is an experimental interface, it is not yet a standard profiling interface.

Finally, we mention some approaches that rely on economics-based theories, using virtual currencies to achieve natural load-balancing of concurrent applications, as well as recycling of unused resources in open distributed environments, with the anticipated side-effect of preventing denial-of-service attacks [22]. Our focus is however more on how to implement the basic resource accounting mechanisms on a specific platform, Java, than on the design of high-level – and distributed – resource allocation policies. Nevertheless, whereas the spirit of this report is rather conservative, it does not exclude the application of the presently described techniques to the implementation of open computational markets.

7 Current Status and Conclusion

The techniques described in this report have been tested by rewriting Java sources by hand; the actual byte-code rewriting tool is not finished yet. The

CPU accounting scheme, with its many optimization tricks, could benefit from a formal proof that no denial-of-service will get unnoticed, and that a client will not be charged for much more than it actually consumed. Also on our immediate todo-list is the development of high-level programming tools in order to support a friendlier event notification mechanism than the overuse exceptions generated by the J-SEAL2 kernel. User-specified thresholds should enable applications to receive warnings in a timely manner before the actual overuse happens.

Whereas other approaches focus on high performance, or demonstrate a long-term, deep redesign of the Java runtime system, our proposal might be grossly characterized as a language-based patch. Our resource control system does indeed not provide the same level of accuracy of measurements and execution speed. On the other hand, J-SEAL2 perfectly fulfills its job of isolating applications from each other, and particularly of preventing denial-of-service attacks originating from inside the execution platform. Moreover, the complete compatibility and portability of our approach makes it immediately usable for the benefit of large-scale distributed agent systems, especially when mobile code is involved.

Acknowledgements

The authors would like to thank Julien Francioli, Rudolf Freund, Andreas Krall, Patrik Mihailescu, and Klaus Rapf for their valuable support.

References

[1] G. Back and W. Hsieh. Drawing the red line in Java. In *Seventh IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, Mar. 1999.

[2] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego, CA, USA, Oct. 2000.

[3] G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, USA, June 2000.

[4] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, Aug. 6, 1998.

[5] W. Binder. J-SEAL2 – A secure high-performance mobile agent system. In *IAT'99 Workshop on Agents in Electronic Commerce*, Hong Kong, Dec. 1999.

[6] W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *Sixth ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages*, Cannes, France, June 2000. `http://cui.unige.ch/~ecoopws/ws00/papers/js.ps`.

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

[8] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, Oct. 1999.

[9] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 21–35, New York, Oct. 18–22 1998. ACM Press.

[10] F.-X. Le Louarn. JUM, a Java Usage Monitor. Web pages at `http://www.iro.umontreal.ca/~lelouarn/jum.html`.

[11] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the fluke kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI-99)*, pages 101–116, Berkeley, CA, Feb. 22–25 1999. Usenix Association.

[12] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[13] M. Godfrey, T. Mayr, P. Seshadri, and T. von Eicken. Secure and portable database extensibility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 390–401, New York, June 1–4 1998. ACM Press.

[14] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[15] J. Hulaas, L. Gannoune, J. Francioli, S. Chachkov, F. Schütz, and J. Harms. Electronic commerce of internet domain names using mobile agents. In *Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC'99)*, Nashville, TN, USA, Oct. 1999.

[16] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

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

[18] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[19] Sun Microsystems, Inc. Enterprise JavaBeans Technology. Web pages at `http://java.sun.com/products/ejb/`.

[20] Sun Microsystems, Inc. Java Servlet Technology. Web pages at `http://java.sun.com/products/servlet/`.

[21] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.3/docs/guide/jvmpi/index.html`.

[22] C. F. Tschudin. Open resource allocation for mobile code. In *Proceedings of The First Workshop on Mobile Agents*, Berlin, Apr. 1997.

[23] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[24] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.

[25] T. Von Eicken, C.-C. Chang, G. Czajkowski, and C. Hawblitzel. J-Kernel: A capability-based operating system for Java. *Lecture Notes in Computer Science*, 1603:369–394, 1999.

[26] T. Wilkinson. Kaffe - a Java virtual machine. Web pages at `http://www.transvirtual.com`.