S-RVM: a Secure Design for a High-Performance Java Virtual Machine

Yuval Yarom Katrina Falkner David S. Munro
The University of Adelaide
{yval,katrina,dave}@cs.adelaide.edu.au

Abstract

Reference protection mechanisms, which control the propagation of references, are commonly used to isolate and to provide protection for components that execute within a shared runtime. These mechanisms often incur an overhead for maintaining the isolation or introduce inefficiencies in the communication between the components.

This paper proposes a novel approach for component isolation that avoids runtime overheads by controlling references at compile time. We use the proposed approach to build S-RVM, a Java Virtual Machine based on JikesRVM, which enhances JikesRVM's security by isolating the VM from the application. Our experiments show that on the average S-RVM incurs no performance overhead when executing optimised code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Design, Languages, Measurement, Performance, Security

1. Introduction

Virtual Machines (VMs) frequently need to execute several components of varying levels of trust. Examples includes mobile code e.g. applets downloaded to the browser, third party extensions to software systems and tasks in multi-tasking VMs.

VMs rely on software-based protection mechanisms to isolate trusted components from the untrusted ones. Software protection mechanisms can operate at two levels: reference protection[1] and type protection [21]. Reference protection is the ability to declare objects that are limited to a scope such that only code within that scope can name or hold a reference to these objects. Type protection, most often implemented as protection scopes, refers to the ability to allow access to members of an object based on the scope of the accessing code.

While reference protection is not as ubiquitous as type protection, extensive research into it has been done within the context of component isolation [2, 4, 10, 18] and in alias control [9, 25].

This paper focuses on isolating the application from the VM in metacircular VMs such as Singularity [2], JikesRVM [3] and JNode [1]. Metacircular VMs are implemented in the same language they target and execute within the same runtime environment they provide. As the application and the VM share the runtime environment, isolating the two is paramount for maintaining the VM's integrity. Nevertheless, the existing mechanisms for isolation are considered insufficient for metacircular VMs due to the performance overhead they introduce [8]. We argue that this overhead is avoidable and that with the right design component isolation can be provided at virtually no cost to the steady state performance.

To gain a better understanding of systems providing reference protection we introduce the concept of *zones* which are groupings of objects based on the references allowed to and from these objects. All objects in a zone share the same permissions.

In this paper we present a classification of zones based on the permitted references and show that a handful of zone types are sufficient to describe the reference protection properties of component isolation. This classification captures the salient features of the reference protection in the system and is, therefore, useful for comparing the properties of reference protection schemes.

We also identify the *sealed zone*—a zone type that can prevent external references to the components' internal data while supporting a public interface that can be accessed by other components. While sealed zones promise efficient communication through direct access, existing systems, such as the J-Kernel [18], are implemented as libraries and rely on proxy objects for en-

[1] The original term [21] use for this level is *Memory Protection*. As this term has since been overloaded we elect to use the less confusing term *reference protection*.

forcing the isolation. Consequently the J-Kernel incurs a 10% performance overhead.

We have built S-RVM—a Java VM based on Jikes-RVM—that uses sealed zones to isolate the VM from the application. In S-RVM the VM and the application code execute within separate *tasks*, each with its own type name space. The application can only name its own types and types that the VM exports. As types of internal VM objects are not assignment compatible with any application type and as the application cannot name these types, references from the application to VM internal objects cannot be created. Reference protection in S-RVM is, therefore, enforced at compile time and does not incur any runtime cost.

Restricting the references gives S-RVM its main advantage over JikesRVM. In JikesRVM an application has free access to the VM. A malicious application can use that access to breach the Java language security e.g. by bypassing the type safety. In S-RVM, the application only has access to the interface exposed by the VM, reducing the VM attack surface area. Hence S-RVM provides a more secure environment for running untrusted code than JikesRVM.

We have tested the performance of our system with the DaCapo benchmark suite [7]. Due to it's design, S-RVM takes slightly longer than JikesRVM to achieve optimal performance. When achieving steady-state, the mean performance over all the benchmarks in the suite is the same for both the original JikesRVM and for S-RVM.

2. Related Work

In this section we review the existing work on reference protection. Most of the research into reference protection has been done within the context of Multi-tasking VMs where it is used to provide some degree of task isolation, which can be restrictive, relaxed or anywhere along that spectrum. As most of the multi-tasking VMs are written in Java or languages very similar to it, we focus on Java in our discussion. We begin by reviewing the tools the Java language provides for component isolation and discuss their limitations. We proceed with a description of systems that provide reference protection ordered by decreasing level of isolation.

2.1 The Java Security Architecture

The Java programming language relies on three tools to create protection domains [16]. The first is the Java class loader concept [19]. One of the functionalities of the class loader is to partition the type name space. In addition to resolving the problem of name conflicts, this partitioning prevents code that has no access to a class loader from naming and using type information of types loaded by that class loader.

The second tool is the security manager. The security manager is a runtime authorisation mechanism that gets invoked whenever security sensitive operations are executed. The security manager verifies that the code has the required privilege level to execute the operation.

The third tool is the Java language type safety that ensures that malicious code cannot bypass the other two tools.

While these tools can and have been used to create a multi-tasking VM [6], three aspects of the Java security architecture render it less than ideal for isolating components. First, the isolation properties of the class loader mechanism are weak. Class loaders only protect the type information. They do not control the propagation of references to objects and do not control the use of objects through any of their supertypes (except for those loaded by the class loader). Second, the dynamic nature of the security manager introduces a runtime overhead. The reliance on examining the runtime stack for finding the calling context exacerbates the overhead. Third, the Java security manager is notoriously complex and designing a comprehensive security policy is hard.

In other words, the Java security architecture puts the onus of controlling object reference propagation on the programmer. To control propagation, the programmer needs to have an intimate knowledge of the system libraries and any other potential sharing sources as well as a good understanding of the extent of sharing incurred by disclosing each object reference. At the same time, the tools that Java provides to assist the programmer are both complex and expensive to use.

Many techniques for improving the Java security architecture have been suggested. These are divided into two main approaches: implement reference protection for complete or partial isolation of components; and provide the programmer with better control over either the propagation or the use of references. These techniques are described below.

2.2 Complete Isolation

MVM [10, 11] and JNode [1] isolate applications based on the observation that the only initially shared values in classes are the static fields, the associated `Class` object and `String` literals. By providing each application with its own set of values these systems completely isolate the applications. The exception is the sharing of strings between applications, which is supported by JNode and by early versions of MVM.

Isolation does not come without cost. In complete isolation the only way to share objects is through communication mechanisms, introducing the cost of marshalling and copying of data.

The Microsoft .Net framework [13] also provides complete isolation. Applications in .Net run within application domains, which are completely isolated from each other. .Net supports the concept of remote types which allows applications to communicate across the application domain boundaries. Remote types are implemented as proxies that transparently marshall and send data across the communication channels. As such, they provide simpler communication semantics but do not reduce the communication overhead.

2.3 Object Sharing

Isolation with object sharing relaxes the requirements of complete isolation by allowing references from the component objects to shared objects. Component isolation is maintained by prohibiting external references to component objects.

KaffeOS [4, 5], Singularity [2, 14], XMem [26] and CoLoRS [27] provide object sharing. To enforce task isolation, KaffeOS and XMem use write barriers which introduce runtime checks for reference store operations. Singularity and CoLoRS avoid the runtime costs associated with write barriers by using a separate type hierarchy for shared objects. XMem and CoLoRS mostly rely on the OS process boundaries for protection. Reference protection in these system is used for maintaining referential integrity within shared memory sections.

Although sharing data reduces the overhead associated with marshalling objects across communication channels, it does not allow for remote method invocation. In those systems remote method invocation mechanisms are implemented on top of communication channels between applications.

2.4 Partial Isolation

Partial isolation makes a distinction between private and public objects within the component space. Cross-component references are permitted only to public objects.

Partial isolation is supported by the J-Kernel [18], where it is implemented using automatically generated wrapper objects, called capabilities, that can be shared between tasks. The advantage of this design is that it supports remote method invocation via the capabilities. However the extra processing required for creating a capabilities and converting data when invoking remote methods results in an overhead of about 10%.

2.5 No Isolation

Some multi-tasking VMs do not make guarantees in respect to isolation. Instead they provide the programmer with better tools for controlling propagation of references and for limiting the use of cross-task references.

I-JVM [15] provides initial isolation of components using the techniques described by MVM. It also provides components with access to a shared name service that can be used for publishing and accessing remote objects. While components are initially isolated, cross-component references are passed using the shared name service or other remote objects. The initial isolation provides the programmer with better controls of propagation. I-JVM, however, does not prohibit any cross-component references.

Secure Java [24] and Luna [17] do not control the propagation of references. Instead, they control the use of remote references. Secure Java uses hardware protection mechanism to restrict access to remote objects. Luna adds a remote protection scope to objects types.

2.6 Reference Protection for Alias Control

Confined Types [25] and Ownership Types [9] are both techniques for alias control that use reference protection to protect internal data structures.

Confined types are an extension of package scoped classes in Java. The type information of packaged scope classes is only accessible to code in the package. That is, code outside the package cannot extend these classes and cannot invoke methods or access fields defined in a package scoped class. Confined types extend the restrictions of package scoped classes by ensuring also that references to objects of a confined type cannot escape the package scope.

Ownership types are a method of protecting the internal representation of compound data structures. With ownership types, references can be tagged as “owned” by an object, limiting their propagation to a scope defined by the owner object.

Like the J-Kernel, confined types and ownership types allow indirect access to protected objects. Confined types are accessible through public classes in the package and owned types are accessible via the owner. Unlike the J-Kernel both confined types and ownership types rely on the static nature of the type system to avoid runtime overhead.

While both designs offer reference protection at no performance cost, the inherent limitations of the designs preclude their use for component isolation.

2.7 Summary

The different methods of providing component isolation vary in the level of isolation and the methods of providing it, but they do share one thing in common. They all report a performance overhead.

For some benchmarks in some systems the overhead can be as small as 1% [11]. For other benchmarks it can be over 20% [4]. However, the mean overhead of component isolation is always several percents.

While a few percents overhead may be an acceptable price for the better security of reference protection, we argue that this price is avoidable. We argue that the combination of a static type system and indirect access to protected objects is the key for providing reference protection at no performance cost.

3. Reference Protection

As seen in the previous section, many different approaches for reference protection have been suggested. To better understand the landscape it is useful to introduce the concept of *zones*, where a zone is a group of objects that share the same permissions for referring to and being referred to from objects outside the zone.

As it is much easier to make decisions based on groups of objects than to track permissions for each and every object, the use of zones is implicit in the design of reference protection systems. Hence, the classification we provide in this section adds clarity and allows comparing systems from across the domain. The classification identifies four types of zones that are used as building blocks for constructing the reference protection policy.

A *privileged* zone is a zone that can hold references to any other zone, regardless of the restrictions otherwise imposed on these zones. Privileged zones typically implement system functionality.

Objects in *isolated* zones can only be referenced from within the zone. External references into isolated zones are prohibited.

Shared zones are those that can have incoming references from multiple (non privileged) zones.

A *sealed* zone is a zone that can only have incoming references from a corresponding *interface* zone. References to the interface zone are typically always allowed.

Figure 1 shows the zone types and demonstrates possible allowed cross-zone references. For clarity we do not display privileged zones in this diagram.

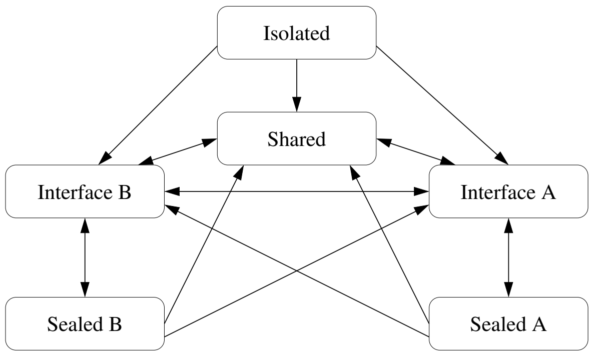

Figure 1. Non-privileged Zone Types

We apply our classification method to the systems described in Section 2 and show the results in Table 1, identifying the system, the system's name for each zone and our classification for that zone. For example, processes in Singularity [14] maintain independent heaps and do not share memory with each other. Hence, these heaps are isolated zones. Singularity processes communicate using channels, which are bi-directional message conduits. A channel can have an associated exchange heap that holds data shared by the processes at the ends of the channel, and is, therefore, a shared zone.

System	Zone	Zone Type
MVM	Isolates	Isolated
JNode	Root Isolate	Isolated (Priv.)
	App Isolates	Isolated
.Net	App. Domain	Isolated
J-Kernel	Domain Capabilities	Interface
	Domain objects	Sealed
KaffeOS	Kernel Heap	Shared (Priv.)
	Shared Heaps	Shared
	Process Heaps	Isolated
Singularity	Process	Isolated
	Exchange heap	Shared

Table 1. Zones in Multi-Tasking VMs

Sealed zones and their corresponding interface zones map naturally into a component system model where each component consists of private data, which is not accessible from other components, and a Remote Procedure Call (RPC) interface, which is used for communication between components. Under this mapping, a sealed zone encloses the private data of each component and the corresponding interface zone forms the RPC interface. The reference protection protects the private data, while RPCs are nothing more than method invocation on objects within the interface.

4. S-RVM

S-RVM is a metacircular Java Virtual Machine based on JikesRVM and designed to use reference protection with the sealed zones model to isolate the VM from the application.

The basic architecture of JikesRVM is depicted in Figure 2. The VM executes at the bottom layer. The VM provides the basic services for the Java libraries but, as most of it is written in Java, it uses some small part of the Java libraries (shown as *Core Libraries* in the diagram). The application itself executes on top of the Java Libraries.

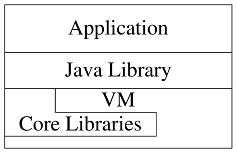

Figure 2. JikesRVM

Using the same environment for executing both the application and itself allows JikesRVM to achieve a high performance. This performance, however, comes at the price that there is no reliable way of separating VM objects from application objects [20].

The lack of clear boundaries between the VM and the application implicitly means that the application code has direct access to VM objects. As the VM maintains the type safety of the language, direct access to the VM can be used to bypass type safety. Java language security depends on type safety, hence direct access to VM objects allows applications to bypass the Java security.

Blurred boundaries between the application and the VM also cause difficulties for debugging and performance measurements. For example, without clear boundaries it is hard to collect performance data for application code only or to perform an object graph analysis for the application.

S-RVM reinstates a clear boundary between the application and the VM by treating the VM and the application as separate tasks. An interface layer is inserted between the VM and the application task. This layer provides the system services required for running a Java library. Reference protection ensures that the only VM objects accessible to the application are those defined to be in the interface layer. Figure 3 shows the main components of S-RVM.

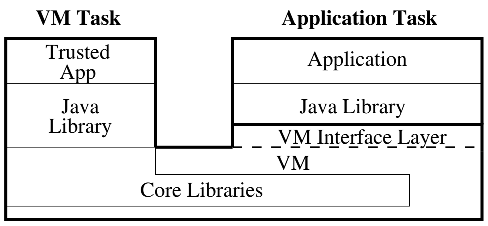

Figure 3. S-RVM

In addition to providing a clear boundary between the VM and the application, the interface layer also forms a trust boundary. The VM does not trust any code running within the application task, including the application task's copy of the Java library. Not trusting the application's library code reduces the VM attack surface area. It also decouples the Java library from the VM allowing the use of different implementations of the Java library for the VM and the application.

The rest of this section describes the implementation of reference protection and the interface layer and discusses some issues specific to securing the VM. These include the asymmetric privileges required, issues related to the trust boundary introduced in S-RVM and exception handling.

4.1 Reference Protection

The crux of separating the application and the VM lies in using a separate base class loader for each task. The base class loader in Java is responsible for loading the Java library classes. Using separate base class loaders effectively creates a separate class name space for each task.

With separate class name spaces, the type hierarchy as viewed by application code is completely distinct from the hierarchy seen by the VM code. Consequently, objects of application types are not assignment compatible with any VM type and vice versa. Thus, the separate name spaces, together with the type safety, isolate the VM from the application and lay the basis for reference protection.

In addition to the tasks name spaces, S-RVM creates a name space for shared types which is used for VM interface types. The `@Export` class annotation marks the classes that belong in the interface layer. When these classes are loaded, their names are added to the shared name space making them available to the application task.

Listing 1 demonstrates a snippet of the class `RVMType` the access to which is required for the implementation of Java reflection. The `@Remote` member annotation introduces a new protection scope by indicating which members of the exported class can be accessed from the application.

```
import org.vmmagic.mu.Export;
import org.vmmagic.mu.Remote;
@Export
public class RVMType {
  @Remote
  public RVMClassLoader getClassLoader() {
    // Implementation of getClassLoader
  }
}
```

Listing 1. An Interface Class

Importing a type is done by loading a stub corresponding to the type. Listing 2 shows the stub code corresponding to the snippet of the `RVMType` class. As can be seen in Listing 2 the stub code includes the class definition with the annotation `@Import`. As the VM only requires the member signatures when importing types, only minimal declarations of remote members are required. Stub classes are automatically generated from the exported class's class file.

```
import org.vmmagic.mu.Import;
@Import
public class RVMType {
  public RVMClassLoader getClassLoader()
          { return NULL; }
}
```

Listing 2. A Stub of an Interface Class

The presented design supports reference protection with the sealed zones model. The application cannot hold references to VM objects unless they are of types in the interface layer. Hence, the interface layer forms the interface zone in the model and other VM objects are in the sealed zone.

As presented, this design is not specific to isolating the VM from the application. The protection it provides is symmetric and the same design could be used for providing reference protection in other environments such as multi-tasking VMs or for isolating third party plugins. One of the challenges of the VM environment is the asymmetric privilege levels. This challenge is discussed below.

4.2 Privileged Access for the VM Task

The VM requires privileged access to application objects for a few special purposes, including garbage collection, passing references between application code and native code, exception throwing and array copying.

To allow passing of references between the VM and the application we add the type `MuObject` which is an unboxed reference to any object in the system. `MuObject` supports a single generic constructor that creates a `MuObject` reference from any object reference and a generic `get` method which returns the referenced object. JikesRVM uses `Object` references to refer to application objects. To avoid rewriting unnecessary sections of JikesRVM we also allow assigning any object to VM `Object` references.

References that can refer to all the objects in the system present the risk of breaching the reference protection. S-RVM does not, currently, verify that VM objects are not passed to the application using `MuObject` references. Faults in the VM may, therefore result in the application getting a reference to internal VM objects. Nevertheless, the generic nature of `MuObject.get()` introduces a dynamic type check before the referenced object can be used for any purpose, providing some level of protection against such VM faults.

Unboxed wrapper types are used to wrap arrays when these are passed to the VM `arraycopy` methods. The use of a different wrapper type for each array type allows us to keep the array element type information across the VM/application boundary and to avoid an otherwise required dynamic type check.

4.3 Creating a Trust Boundary

As discussed above, the interface layer is also a trust boundary. Unlike most Java Virtual Machines, S-RVM does not trust the Java library code used by the application. Instead, the interface layer is designed to provide a security barrier and to protect the VM from malicious applications.

Lack of trust is manifested in extra tests in interface methods. For example, to prevent the application code from using reflection on VM types the VM method `getObjectType()` which returns the type of an object is replaced with a secure version which, when invoked by the application, ensures that the object is an application object.

Lack of trust also implies that arrays cannot be shared between the application and the VM. Instead, S-RVM uses wrapper classes to provide the application with read-only access to VM arrays.

A slightly more involved consequence of the lack of trust is the handling of string objects. JikesRVM uses the `String` implementation from the GNU Classpath library. A `String` object uses an array of characters as a backing store. It also records the offset into the array and the length of the string. The contents of the backing store is considered to be constant and is only shared with code trusted to maintain this property. The backing store can, therefore, be shared between `String` objects.

Strings are frequently transferred across the boundary between the VM and the application. Common operations that manipulate `String` objects are JNI functions and creation of `String` literals during class loading. Copying `String` objects when transferring them between the application and the VM would introduce a significant overhead. On the other hand, sharing `String` objects or their backing stores would reduce isolation and require the VM to trust the application not to modify the contents of `String` objects.

To avoid copying yet maintain the safety of `String` objects, S-RVM introduces two unboxed wrappers for VM character arrays. The first, `MuCharArray`, provides read only access to the character array and is used as the backing store for `String` objects. The second, `MuWriteableCharArray`, provides write access to an underlying character array for the purpose of creating a `String` object. When a `MuWriteableCharArray` is converted to a `MuCharArray`, it is *sealed* and write permission to it is revoked, ensuring that `String` objects remain constant once created.

The S-RVM interface includes some utility methods that allow copying to unsealed `MuWriteableCharArrays` and from `MuCharArrays` using the VM implementation of the array copy functions. It also includes the `MuString` class which, like `String`, contains a `MuCharArray`, an offset and a length. `MuString` is used for packing the information about `String` objects when these are transferred between the application and the VM.

4.4 Exceptions

When Java code encounters an exceptional situation it can abort execution and *throw* an object of type `Throwable` or any of its subtypes. The VM is required to generate and throw exceptions when certain conditions, e.g. when the application references a `null` pointer or when running out of memory, occur.

Due to the separate type hierarchies, exceptions generated in the VM task in S-RVM are not compatible with exceptions in the application task and vice versa. S-RVM combines two methods for ensuring exceptions are signaled and handled as expected.

When the exception is the result of a hardware trap, such as when it is the result of a `null` pointer reference or a division by zero, S-RVM uses an upcall to the task to generate the required exception object. For other exceptions, S-RVM wraps each remote method with an exception handler that converts the exception to the invoking task.

When S-RVM converts an exception it may lose some type information. This loss occurs because the application can declare exception types that are not recognised by the VM. S-RVM, therefore, converts application exceptions to the most specific supertype defined in the VM.

This loss of information cannot affect the VM response to the exception because the VM can only handle the exception types it recognises. If, however, the VM does not catch the exception before returning to the application or if the VM re-throws the exception, the loss of information may affect the application As all the exception conversions that occur in our benchmark do not lose type information, this is a theoretical rather than an actual problem.

To rectify the information loss, the conversion code can keep a reference to the original exception and use it instead of converting the exception back to the application. As the number of exceptions that are actually converted is very low (less than 2,000 conversions in eclipse and less than 1,000 in any of the other DaCapo benchmarks), and as these conversions only occur in the initial loading of classes, we expect changes in the conversion algorithm will only have a negligible effect on performance.

4.5 Summary

S-RVM is a proof-of-implementation of our design of component isolation using sealed zones. S-RVM represents a substantial change in JikesRVM's design and as such requires significant modifications to the software system of which we have described the main architectural changes required to implement sealed zones. However, retrofitting an existing system, enables us to make performance comparison with the underlying single-tasked VM. The results of this comparison are described in the next section.

5. Results

We have tested the performance of S-RVM on an IBM x3500 server with two quad-core Xeon E5345 processors and 24GB of RAM; running Fedora release 16. The VM was compiled using the production configuration, with edge count profiling information collected from running the DaCapo `fop` benchmark. We have retrofitted both S-RVM and JikesRVM with the `Double.toString()` implementation from the OpenJDK [22].

To measure the VM start up time we ran the time-honoured "Hello World!" program on both S-RVM and on JikesRVM 3.1.1. The JikesRVM image contains significant parts of the Java library precompiled at a high optimisation level. By contrast, the application task in S-RVM has no classes preloaded. In addition to loading the code for the application, the application task needs to load and compile those classes of the Java library that the application uses. S-RVM, therefore, takes much longer to start than JikesRVM. Running "Hello World!" in JikesRVM 3.1.1 takes 68ms. Running the same program in the S-RVM application task takes 245ms—almost four times longer.

To measure the performance of optimised code in the VM we use the DaCapo benchmark suite release 2006-10-MR2. The DaCapo suite test harness runs each test multiple times to allow adaptive compilers time to learn and adjust to the program patterns. Table 2 shows the performance of the DaCapo benchmarks in the first, third, tenth and twentieth iteration on both JikesRVM 3.1.1 and on S-RVM. For each benchmark we report the geometric mean of the results from 40 runs of the test, rounded to the nearest millisecond. For a measure of the relative overall performance, we also report the geometric mean of the running times of all benchmarks for each configuration.

Figure 4 shows the performance results for S-RVM relative to JikesRVM. The value of 100 represents the performance of JikesRVM for each benchmark scenario. As can be seen in the diagram, in the first iteration S-RVM consistently underperforms, with a mean performance indicating an overhead of over 10%. (The actual figure is around 12.54%.) However, as the number of repetitions increases, the gap closes until it disappears at 10 iterations.

We have also tested the performance of the S-RVM code without the added security of a separate task. (That is, the application code was executed within the context of the VM, as in JikesRVM.) The results, summarised in Figure 5 demonstrate a much more consistent behaviour, with an overall overhead of around 1.5%.

As S-RVM adds complexity to JikesRVM, a small overhead is expected. The results of running S-RVM without a separate task seem to match this expectation.

Benchmark	1 Iteration		3 Iterations		10 Iterations		20 Iterations	
	JikesRVM	S-RVM	JikesRVM	S-RVM	JikesRVM	S-RVM	JikesRVM	S-RVM
antlr	2,129	2,541	1,629	1,762	1,548	1,584	1,475	1,506
bloat	6,302	7,259	5,301	5,551	5,012	5,323	4,919	5,257
chart	9,373	10,083	6,444	6,353	6,383	6,214	6,403	6,217
eclipse	30,752	32,023	24,787	25,181	23,861	24,044	23,494	23,623
fop	2,202	2,600	1,636	1,694	1,546	1,533	1,501	1,498
hsqldb	2,909	3,273	1,959	2,011	1,881	1,836	1,775	1,677
jython	6,899	7,597	4,488	4,503	4,002	4,017	3,774	3,756
luindex	8,001	8,907	7,054	7,218	6,959	6,985	6,896	6,893
lusearch	2,583	2,980	1,657	1,605	1,459	1,454	1,438	1,443
pmd	5,122	5,793	4,085	4,094	3,907	3,828	3,774	3,711
xalan	2,598	2,910	1,845	1,905	1,623	1,626	1,598	1,595
mean	4,983	5,607	3,694	3,767	3,472	3,478	3,379	3,375

Table 2. Performance of JikesRVM and S-RVM

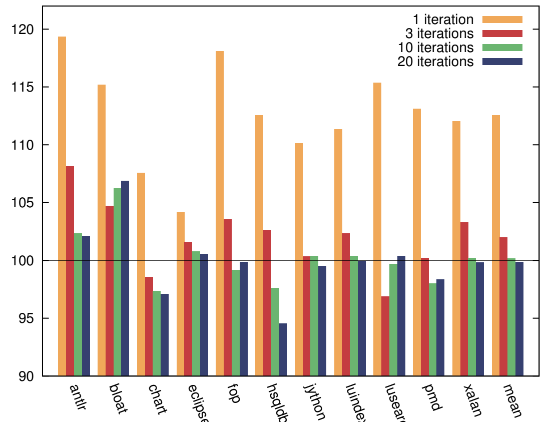

Figure 4. Relative Performance of S-RVM

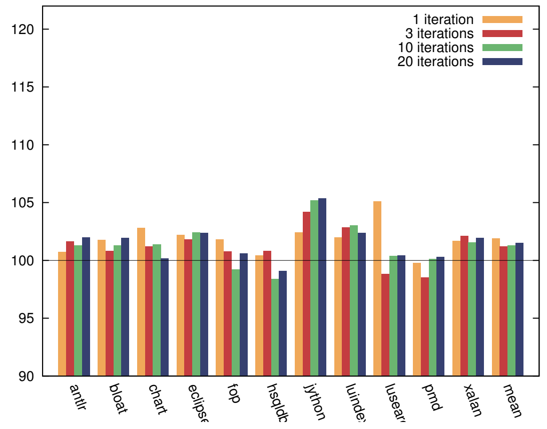

Figure 5. Performance of S-RVM Without Task Separation

The improved performance seen in most of the benchmarks when running the application in a separate task conflicts with our expectations.

We believe this improved performance is the result of having two separate copies of the Java library. With two copies of the Java library, each copy is optimised for a different workload. The library in the VM is optimised for the workload the VM generates while the library in the application is optimised for the workload of the application.

This hypothesis is supported by two observations. The first is that edge count information for some "hot" methods show significant differences between the VM and the application. E.g. DaCapo `fop` tends to invoke `AbstractMap.equals` with identical objects whereas the VM tends to use different objects (68% identical for the application vs. 28% for the VM). Edge count information improves optimisation. Consequently, significant differences in edge count profiling data are likely to translate to noticeable changes in performance.

The second observation is that the optimising compiler eliminates some code when compiling some application task methods. An example is the search of the `Atom` table in `String.intern()`, which is only required when executing within the VM.

In JikesRVM and when S-RVM is run without task separation, both these workloads share a single copy of the library. The optimising compiler cannot optimise the code to either workloads, resulting in a less than optimal performance of the Java library code.

In most benchmarks, the added performance of the specialised Java libraries is enough to more than offset the overhead introduced by the added complexity of S-RVM, resulting in a better performance for these benchmarks. The most notable exception is the `bloat` benchmark which shows a significant slowdown and the gap widens as the number of iterations increases.

Table 3 shows the minimum heap size required for executing the DaCapo benchmarks on JikesRVM and on S-RVM. S-RVM incurs a fairly constant overhead of about 8MB. While the relative overhead is significant, especially for the smaller tests, its absolute value does not seem to change much between the tests. With current memory sizes an overhead of 8MB is not expected to be significant except for the most extreme scenarios.

Benchmark	JikesRVM	S-RVM	Overhead
antlr	22MB	30MB	8MB (36%)
bloat	40MB	45MB	5MB (13%)
chart	36MB	45 MB	9MB (25%)
eclipse	60MB	69MB	9MB (15%)
fop	31MB	40MB	9MB (29%)
hsqldb	102MB	110MB	8MB (8%)
jython	35MB	44MB	9MB (26%)
luindex	24MB	31MB	7MB (29%)
lusearch	45MB	53MB	8MB (18%)
pmd	37MB	45MB	8MB (22%)
xalan	47MB	54MB	7MB (15%)
Average			7.9MB

Table 3. Memory Footprint of S-RVM

6. Conclusion

This paper addresses the issue of the overhead incurred by controlling reference propagation for isolating components. We present a study of zone types used in reference protection and propose the use of sealed zones for achieving isolation. We validate the proposed type system design by using it to implement S-RVM.

S-RVM is a proof-of-implementation Java VM which provides better security properties than JikesRVM, on which it is based. S-RVM segregates application objects and controls references between them and VM objects, thereby restricting application access to internal VM data structures. We measure the performance of S-RVM with the DaCapo benchmark suite and demonstrate that on the average S-RVM incurs no performance overhead over JikesRVM. These performance results confirm that the design works and that it works well.

7. Future Work

Rather than replicating library code used by the VM and the application, it may be possible to share the classes metadata, including both the byte code and the compiled code. Techniques for class sharing have been investigated in the past [11, 12]. Applying these and similar techniques to S-RVM can reduce the memory footprint and can provide pre-compiled library code to the application, reducing the application startup time. To provide the steady-state performance, "hot" methods would still need to be specialised and their code cannot be shared.

S-RVM runs two separate components—the VM and the application. Two natural extensions to this model are supporting multiple components and multi-tasking.

At this stage, a rudimentary support for multi-tasking is available. Initial performance evaluation indicates that While some parallel benchmarks perform reasonably well, in most cases multi-tasking in S-RVM is less efficient than executing multiple VMs. As the scheduling and memory management subsystems in JikesRVM are tuned for executing a single application, this result is to be expected. Past research [23] suggested algorithms for multi-tasking memory management which may alleviate the problem.

Resource accounting and task termination, are not addressed by the current implementation. Accounting for I/O and time resources is orthogonal to memory isolation. As the type information of objects in S-RVM indicate the task the object belongs to, we believe that the work we have done will facilitate the accounting of memory resources. Further research is required to validate this.

Task termination is a more difficult issue in the framework we have chosen. The main limitation is that JikesRVM does not, currently, support class unloading. Each task loads a significant number of classes. Without class unloading a terminating task would leave these classes in the system. Each of these classes consumes resources. Hence the total number of tasks that could, potentially, run on JikesRVM and derivative systems is limited.

Another issue that will need addressing is the conversion of legacy software. The level of effort required will depend to a large extent on the design of the specific legacy software. Monolithic software which lacks clear boundaries will, probably, require a major reengineering effort. On the other side of the spectrum, software designed for a distributed environment and which uses well defined explicit communication mechanisms might only require replacing the implementation of these mechanisms.

It would be interesting to see how much effort is required for converting applications built with the OSGi framework. Some reengineering would, probably, be required, but conversion may be facilitated by the use of automated tools.

8. Acknowledgements

This work was sponsored in part by the Defence Science and Technology Organisation, Australia.

References

[1] Jnode. `http://www.jnode.org/`.

[2] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 1–10, New York, New York, United States, 2006. ACM.

[3] B. Alpern, J. J. Barton, S. Flynn-Hummel, T. Ngo, J. C. Shepherd, C. R. Attanasio, A. Cocchi, D. Lieber, M. Mergen, and S. Smith. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 314–328, Denver, Colorado, United States, Nov. 1999.

[4] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Transactions on Programming Languages and Systems*, 27(4):583–630, July 2005.

[5] G. Back, W. C. 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*, San Diego, California, United States, October 2000.

[6] D. Balfanz and L. Gong. Experience with secure multi-processing in java. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS '98)*, pages 398–405, Amsterdam, Netherlands, May 1998.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 169–190, Portland, Oregon, USA, Oct. 2006. ACM Press. ISBN 1-59593-348-4.

[8] S. M. Blackburn, S. I. Salishev, M. Danilov, O. A. Mokhovikov, A. A. Nashatyrev, P. A. Novodvorsky, V. I. Bogdanov, X. F. Li, and D. Ushakov. The Moxie JVM experience. Technical Report TR-CS-08-01, Australian National University, Department of Computer Science, May 2008.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 33 (10):48–64, 1998. ISSN 0362-1340.

[10] G. Czajkowski. Application isolation in the Java Virtual Machine. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 354–366, Minneapolis, Minnesota, United States, Oct. 2000.

[11] G. Czajkowski and L. Daynés. Multitasking without comprimise: a virtual machine evolution. *SIGPLAN Not.*, 36(11):125–138, 2001. ISSN 0362-1340.

[12] G. Czajkowski, L. Daynés, and N. Nystrom. Code sharing among virtual machines. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP02)*, pages 155–177, Malaga, Spain, June 2002.

[13] *Standard ECMA-335: Common Language Infrastructure (CLI)*. ECMA, fourth edition, June 2006.

[14] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. *SIGOPS Opearing Systems Review*, 40 (4):177–190, 2006. ISSN 0163-5980.

[15] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java virtual machine for component isolation in OSGi. In *the 39th International Conference on Dependable Systems and Networks (DSN 2009)*, pages 544–553, Estoril, Portugal, June 2009. IEEE Computer Society.

[16] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. pages 103–112.

[17] C. Hawblitzel and T. von Eicken. Luna: A flexible Java protection system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 391–401, Boston, Massachusetts, United States, December 2002.

[18] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, June 1998.

[19] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 36–44, Vancouver, British Columbia, Canada, 1998. ACM. ISBN 1-58113-005-8.

[20] Y. Lin, S. M. Blackburn, and D. Frampton. Unpicking the knot: Teasing apart vm/application interdependencies. In *VEE '12: Proceedings of the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 181–190, London, United Kingdom, 2012. ACM. ISBN 978-1-60558-375-4.

[21] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, January 1973.

[22] Oracle Corporation. OpenJDK. `http://openjdk.java.net/`.

[23] S. Soman, C. Krintz, and L. Daynès. Mtm2: Scalable memory management for multi-tasking managed runtime environments. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP08)*, pages 335–361, Paphos, Cypress, July 2008. Springer-Verlag.

[24] L. van Doorn. A secure Java virtual machine. In *Proceedings of the USENIX Security Symposium*, pages 21–35. USENIX Association, 2000.

[25] J. Vitek and B. Bokowski. Confined types in Java. *Software–Practice and Experience*, 31(6):507–532, May 2001.

[26] M. Wegiel and C. Krintz. XMem: Type-safe, transparent, shared memory for cross-runtime communication and coordination. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 327–338, New York, NY, USA, June 2008. ACM Press.

[27] M. Wegiel and C. Krintz. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'10)*, pages 223–240, Reno/Tahoe, Nevada, USA, Oct. 2010.