Software-based Reference Protection for Component Isolation

By

Yuval Yarom

September 12, 2014

A Thesis Submitted for the Degree of
Doctor of Philosophy
In the School of Computer Science
University of Adelaide

Contents

Abstract **viii**

Declaration **ix**

Acknowledgements **x**

1 Introduction **1**

1.1 Classification of Reference Protection Systems 5
1.2 Exported Types 7
1.3 Implementation of Exported Types 8
1.4 Summary of Contributions 8
1.5 Thesis Structure 9

2 A Model and Classification of Reference Protection **10**

2.1 Zones 10
2.2 Complete Isolation 15
2.2.1 MVM 16
2.2.2 JX 16
2.2.3 The .Net Framework 17
2.2.4 JNode 18
2.2.5 OVM 18
2.2.6 Summary 19
2.3 Object Sharing 20
2.3.1 KaffeOS 20
2.3.2 Singularity 21
2.3.3 Rust 22
2.3.4 XMem 23
2.3.5 CoLoRS 24
2.3.6 Real-Time Java 24
2.3.7 Summary 25
2.4 Partial Isolation 25
2.4.1 J-Kernel 26

2.4.2 Confined Types . . . 27
2.4.3 Ownership Types . . . 27
2.4.4 Summary . . . 28
2.5 Initial Isolation . . . 28
2.5.1 I-JVM . . . 29
2.5.2 Alta . . . 29
2.5.3 Summary . . . 29
2.6 Isolation Techniques . . . 30
2.6.1 Reachability Roots . . . 30
2.6.2 Controlling Reference Propagation . . . 33
2.7 Evaluation of the Classification . . . 34

3 Exported Types **38**
3.1 Type Systems . . . 39
3.2 Partial Isolation and Types . . . 41
3.3 Component Name Spaces . . . 44
3.4 Creating the Interface Zones . . . 45
3.5 The Remote Interface . . . 47
3.6 Compatibility with Language Features . . . 49
3.6.1 Structure-based Type Equivalence . . . 49
3.6.2 Implicit Operations on Types . . . 50
3.6.3 Universal Methods . . . 51
3.6.4 Special Semantics of Types . . . 52
3.6.5 Reflection . . . 53
3.7 Summary . . . 53

4 An Implementation of Exported Types **55**
4.1 The Java Virtual Machine . . . 58
4.1.1 Class Loaders . . . 59
4.1.2 Security Manager . . . 60
4.1.3 Byte Code Verifier . . . 61
4.1.4 Summary . . . 62
4.2 JikesRVM . . . 62
4.2.1 Compilation Framework . . . 63
4.2.2 Memory Management . . . 64
4.2.3 Compiler Magic . . . 65
4.2.4 Library Interface . . . 66
4.2.5 Virtual Machine Build . . . 66
4.2.6 Security . . . 67
4.3 An Overview of S-RVM . . . 69

4.4 Import and Export . . . 72
4.5 The VM Interface Layer . . . 73
4.6 Creating a Trust Boundary . . . 76
4.7 Privileged Access for the virtual machine Task . . . 77
4.8 Exceptions . . . 79
4.9 Implementation Verification . . . 80
4.10 Summary . . . 81

5 Performance Evaluation **82**
5.1 Memory Usage . . . 83
5.2 Application Task Startup . . . 85
5.3 Steady-State Execution Speed . . . 86
5.4 Summary . . . 92

6 Summary and Conclusions **93**
6.1 Classification Framework . . . 93
6.2 The Exported Types Design . . . 94
6.3 S-RVM . . . 95
6.4 Revisiting the Classification Framework . . . 96
6.5 A Multi-tasking Virtual Machine . . . 99
6.6 Structure-based Type Equivalence . . . 100
6.7 Summary . . . 101

A Exported Types Specifications **102**
A.1 Type Systems . . . 102
A.2 Exported Types . . . 103
A.3 Mapping to zones . . . 105

B S-RVM Implementation Details **107**
B.1 The Upcall Interface . . . 107
B.2 Initialising `RVMTask` . . . 110
B.3 `String` Backing Store . . . 111
B.4 Exception Conversion . . . 114

C Performance Data **117**

Bibliography **138**

List of Tables

2.1 Methods for controlling reference propagation 34
2.2 Zones in systems providing reference protection 35
2.3 Classification of reference protection 36
2.4 Classification of reference protection 37

5.1 Minimum heap size 84
5.2 Normalised execution times of the DaCapo benchmarks on S-RVM relative to JikesRVM with 90% confidence intervals (Same Heap Size scenario) 88
5.3 Normalised execution times of the DaCapo benchmarks on S-RVM relative to JikesRVM with 90% confidence intervals (Same Heap Pressure scenario) 89

6.1 Zones in systems providing reference protection 97
6.2 Classification of reference protection 98
6.3 Classification of reference protection 99

C.1 Mean execution times of the DaCapo benchmarks at the 1st iteration (ms) 118
C.2 Mean execution times of the DaCapo benchmarks at the 2nd iteration (ms) 119
C.3 Mean execution times of the DaCapo benchmarks at the 3rd iteration (ms) 120
C.4 Mean execution times of the DaCapo benchmarks at the 4th iteration (ms) 121
C.5 Mean execution times of the DaCapo benchmarks at the 5th iteration (ms) 122
C.6 Mean execution times of the DaCapo benchmarks at the 6th iteration (ms) 123
C.7 Mean execution times of the DaCapo benchmarks at the 7th iteration (ms) 124

C.8 Mean execution times of the DaCapo benchmarks at the 8th iteration (ms) . . . 125
C.9 Mean execution times of the DaCapo benchmarks at the 9th iteration (ms) . . . 126
C.10 Mean execution times of the DaCapo benchmarks at the 10th iteration (ms) . . . 127
C.11 Mean execution times of the DaCapo benchmarks at the 11th iteration (ms) . . . 128
C.12 Mean execution times of the DaCapo benchmarks at the 12th iteration (ms) . . . 129
C.13 Mean execution times of the DaCapo benchmarks at the 13th iteration (ms) . . . 130
C.14 Mean execution times of the DaCapo benchmarks at the 14th iteration (ms) . . . 131
C.15 Mean execution times of the DaCapo benchmarks at the 15th iteration (ms) . . . 132
C.16 Mean execution times of the DaCapo benchmarks at the 16th iteration (ms) . . . 133
C.17 Mean execution times of the DaCapo benchmarks at the 17th iteration (ms) . . . 134
C.18 Mean execution times of the DaCapo benchmarks at the 18th iteration (ms) . . . 135
C.19 Mean execution times of the DaCapo benchmarks at the 19th iteration (ms) . . . 136
C.20 Mean execution times of the DaCapo benchmarks at the 20th iteration (ms) . . . 137

List of Figures

1.1 The spectrum of isolation using reference protection 6

2.1 A component system with isolated zones 12
2.2 A component system with isolated zones and a shared zone 13
2.3 A component system using sealed zones 13
2.4 Structure of the JX system 17
2.5 Heap structure in KaffeOS 21
2.6 The exchange heap in singularity 22
2.7 Taxonomy of methods for handling reachability roots 31

3.1 A component system with partial isolation 42
3.2 Overlaying the sealed zones model over the type hierarchy 43
3.3 Sealed zones 44
3.4 Type hierarchies with exported types 45
3.5 Type hierarchies after import 46
3.6 Extending an imported type 47

4.1 The JikesRVM runtime environment 63
4.2 Type hierarchy in JikesRVM 68
4.3 The S-RVM runtime environment 69
4.4 RVMTask class diagram 71
4.5 Bi-directional communication between tasks 74
4.6 Classloader class hierarchy 74

5.1 S-RVM Boot image overhead 84
5.2 Class loading during task startup 85
5.3 Mean execution time on S-RVM relative to JikesRVM. 90
5.4 Steady state normalised execution times of the DaCapo benchmarks on S-RVM at several heap sizes 91
5.5 Separate vs. combined profiling 92

6.1 A multi-tasking virtual machine 99

Abstract

Reference protection mechanisms are commonly used to isolate and to provide protection for components that execute within a shared run-time environment. These mechanisms often incur an overhead due to maintaining the isolation or introduce inefficiencies in the communication between the components. Past research operated under the assumption that some performance loss is an acceptable price for the added security that comes with better isolation. This thesis sets out to demonstrate that good isolation does not imply performance loss.

While numerous models for implementing reference protection have been suggested, there is a lack of a unified terminology that allows the comparison of systems from across the domain. This thesis presents a classification framework that captures the trade-offs present in the design of reference protection. It identifies four main models of reference protection: *complete isolation*, where components do not share references to objects; *object sharing*, where components can share data while still maintaining private, unshared data; *partial isolation*, where components have private, unshared data and an exposed interface that allows other component's indirect access to the private data; and *initial isolation*, where components are isolated when created, but the model allows the programmer to share references without restriction.

Applying the classification to systems providing reference protection identifies a gap in the prior research. Partial isolation promises the level of security expected from component isolation combined with efficient communication. Yet, the only implementation of partial isolation of components uses expensive run-time checks to enforce the protection.

To bridge this gap, this thesis presents the Exported Types design. Exported Types is a type system design that enforces partial isolation at compile time. Using compile-time checks removes the run-time overhead of enforcing the protection model. The design is applied to a meta-circular Java virtual machine to isolate the virtual machine code from the application. Applying reference protection in this scenario reduces the number of classes the virtual machine exposes to the application by two orders of magnitude. Performance tests demonstrate that reference protection, and the higher security it provides, are achieved at no performance cost.

Declaration

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

I give consent to this copy of my thesis, when deposited in the University Library, being made available for loan and photocopying, subject to the provisions of the Copyright Act 1968.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library catalogue and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

Yuval Yarom
September 12, 2014

Acknowledgements

Ithaka gave you the marvelous journey.
Without her you would not have set out.
She has nothing left to give you now.
—*Constantine Cavafy, "Ithaka"*

The journey described in the following pages would not have been possible without the dedicated guidance of my supervisors Dave Munro and Katrina Falkner. It is through the wisdom they shared, the encouragement they provided and the patience they exercised that this work has come to fruition. It was an honour working with them and the lessons they have taught will stay with me for the rest of my life.

I would also like to thank Henry Detmold who for a long time acted as an unofficial supervisor. Many thanks also to Steve Blackburn, Ron Morrison and John Zigman for the help they provided and their useful suggestions in various stages of this work. Thanks are also due to Richard Jones and to the anonymous examiner for the careful reading of this thesis and for the wise and helpful comments provided.

I am indebted to Kathy Cooper, Cathie Liersch, Sharyn Liersch, Julie Mayo, Jo Rogers and Tracey Young who helped navigating the treacherous waters of the university administration and for providing some geek-free refuge. Thanks also to William Brodie-Tyrrell, Diana Hill, Peter Kelly, Ekim Kocadag, Joseph Kuehn, Matt Lowry, Peter Nguyen, Travis Olds and Stani Ovtcharova who, over the years, shared the working space with me and helped make the time spent on this PhD an enjoyable experience.

The sages teach that without food there is no learning.[1] It would be futile to try listing all fine establishments providing sustenance around the university. However, special mention must be made of the spicy chicken at Raah Cafe, the mayonnaise chicken at Zen Garden and everything at Vego and Love'n It.

Most of all, I would like to thank my wife Karen Gekker and my children Yarden and Itay for their help, support and encouragement throughout the years. I acknowledge and deeply appreciate the sacrifices they made to allow me to continue my studies.

[1] Ethics of the Fathers 3:17 אם אין קמח, אין תורה. Lit.:*If there is no flour, there is no (study of the) Torah.*

Chapter 1

Introduction

In recent years virtual machines[1] that provide the run-time environment for executing programs have proliferated all aspects of life. They run inside Web browsers and inside Web servers. They are an integral part of desktop environments and of embedded electronic devices.

Software running in these virtual machines often comprises multiple software entities which provide abstractions of parts of the functionality of the system. These entities are developed independently of each other, but may cooperate through common interfaces. Examples of such entities include code libraries, packages of mobile code and third-party extensions, to name a few. This thesis refers to these entities as *components*.

Components running in the same system are often developed by multiple, potentially mistrusting parties. In such cases, components cannot trust the code of other components. It is, therefore, desirable to limit the access an untrusted component is given to the data of other components. Furthermore, even when there is mutual trust, components may fail due to software errors. Containing component failures and limiting their effect on the operation of the system is, therefore, desired.

To contain component failures and to protect the system from hostile components, the virtual machine needs to isolate these components from one another. Isolating components, however, reduces the efficiency and the ease of communication between them, hampering cooperation. Thus, the level of component isolation a virtual machine provides presents a trade-off between the level of security and the efficiency of inter-component communication. This trade-off is the domain of this thesis.

This thesis explores the isolation provided by *reference protection*, or the use of

[1] The term "virtual machine" may refer to two distinct types of entities. The first is a software entity that provides a run-time environment for a high-level language, such as Java or C# and the second is a software entity that provides a virtualised system environment, e.g. VMware. "Virtual machine" in this thesis always refers to the former.

software mechanisms to control the propagation of references between components. Software-based protection has a finer granularity than hardware-based protection and, therefore, allows greater flexibility. However, reference protection often incurs a performance overhead due to the need to trace references to control their propagation. Prior research into the use reference protection for component isolation regards this overhead, as well as the costs of inefficient inter-component communication, as an acceptable price for the added security of component isolation. This thesis shows that both performance and security can be delivered and there is no need to sacrifice one to achieve the other.

The thesis presents a new classification framework for reference-protection mechanisms and describes a type system designed to enforce component isolation at compile time while maintaining efficient inter-component communication. The type system design is accompanied by a proof-of-implementation demonstrating that isolation is provided at no performance cost.

The current popularity of virtual machines can be traced back to the introduction of the Java programming language [Gosling and McGilton, 1996, Lindholm and Yellin, 1999]. Virtual machines have existed earlier [Shillington and Ackland, 1979] and the concept that programming languages are implemented through a combination of hardware and software machines is even older [Neuhold and Lawson, 1971]. However, Java was one of the first languages to gain widespread acceptance by users, programmers and researchers. The success of implementing Java using a virtual machine prompted many high level languages to use the approach [Bolz et al., 2009, ECMA, 2006, Evans, 2011, Fagerholm, 2005, Ierusalimschy et al., 2005].

With the numerous languages that are implemented using virtual machines and with the wide spectrum of environments in which virtual machines are employed, it is no surprise that the nature of components is also varied. Examples of components include applets downloaded to the browser [Dean et al., 1996], third-party extensions to software systems, OSGi [OSG, 2011] bundles and tasks in multi-tasking virtual machines [Aiken et al., 2006, Back et al., 2000a, Czajkowski, 2000]. While these components are very different from each other, they do have a few common properties.

In all of these systems, components are run-time software entities. As software entities, they have code and data associated with them and the system keeps track of when a particular component is executing. Furthermore, each component has an identity which can be used for referring and managing it. For example, OSGi bundles have an associated `Bundle` object that manages their life cycle and is used for referring to them.

Components do not execute in a vacuum. For their operation they need to

communicate with the run-time environment and with other components. To that aim, components support some sort of an interface and a communication protocol that specifies how this interface is used. Yet, components also maintain some private state, which is not accessed directly by other components.

Components, like all software, may fail. Furthermore, as components are often downloaded from the Internet, they may include malicious code. Virtual machines implement protection mechanisms whose aim is to isolate components, so that code in one component can only affect the behaviour of other components via the use of the communication protocol between them.

Software-based isolation mechanisms take one of two forms: reference protection[2] and type protection [Morris, 1973]. 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, e.g. `public` and `private` scopes, refers to the ability to allow access to members of an object based on the scope of the accessing code. These forms correspond to implicit and explicit protection, identified by Bershad et al. [1995a].

While reference protection is not as ubiquitous as type protection, extensive research into it has been done within the context of component isolation [Aiken et al., 2006, Back and Hsieh, 2005, Back et al., 2000a, Binder et al., 2001, Czajkowski, 2000, Czajkowski and Daynès, 2001, ECMA, 2006, Golm et al., 2002, Hawblitzel et al., 1998, JNode, Rust Reference Manual, Spring et al., 2007, Tullmann et al., 2001] in alias control [Clarke et al., 1998, Vitek and Bokowski, 2001] and in memory management [Armbruster et al., 2007, Bollella et al., 2000, Grossman et al., 2002].

The *principle of least privilege* [Saltzer and Schroeder, 1975] is a central idea in computer security. As formulated by Saltzer and Schroeder [1975], it states:

> Every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Components communicate through the interfaces they provide. They do not need to have access to the private state of other components. Hence, components do not need references to the private states of each other. Consequently, the principle of least privilege implies that references to the private state of a component should not be accessible from outside that component.

Reference protection can isolate components and maintain the protection required for private data, preventing the propagation of references to the data. Type protection can also be used for controlling reference propagation. However, without

[2]The original term Morris [1973] uses for this form is *Memory Protection*. As this term has since been overloaded this thesis uses the less confusing term *reference protection*.

reference protection, the onus of maintaining the safety of the private data remains with the programmer, and programmers are known to be fallible.

Implementations of reference protection vary with respect to the level of isolation they provide. The choice of the isolation level presents a trade-off between the level of security guarantees provided and the efficiency of the inter-component communication. When isolation is strict, i.e. when components do not share references to memory objects, communication between components must use indirect mechanisms that require data copying and, often, marshalling. On the other hand, strict isolation provides the benefit of a simple operation model which delivers higher security and is, usually, easier to implement. Less strict isolation permits some object sharing. Sharing avoids the need for data copying and marshalling and, therefore, allows for a more efficient communication. However, sharing is less secure than strict isolation because it opens more ways of interaction between components. Furthermore, the need to distinguish between sharable and non-sharable data complicates the implementation of reference protection models that support sharing.

Previous works on reference protection implicitly assume that the added security provided by component isolation justifies the overhead which results from data copying or from tracing references. This thesis challenges this compromise and claims that isolation can be provided at no run-time performance overhead. More specifically, it demonstrates that the performance overheads of isolation can be eliminated by enforcing reference protection at compile time and by allowing direct references from components to the interface of other components.

This thesis provides a new classification of reference-protection systems which captures the trade-offs and creates a unified model for describing systems providing reference protection. As part of the classification, it introduces *partial isolation*—a reference-protection model that permits inter-component references to those objects that provide a communication interface. Although the partial isolation model promises a combination of efficient communication and good isolation, past implementations of the model relied on expensive run-time enforcing of reference protection, which degrades the system performance.

To fill this gap, this thesis presents a design of a type system that provides partial isolation of components. The design allows inter-component references, but includes limitations on types in the system that prevent references from components to the private data of other components. Access to public data promises an efficient communication channel between components, whereas preventing access to private data maintains the required level of protection.

Lastly, the thesis presents S-RVM [Yarom et al., 2012], a proof-of-implementation of the type system design. This implementation is carried out within the context

of a meta-circular Java virtual machine. Meta-circular virtual machines execute the virtual machine in the same run-time environment they provide for the application. Thus, the application and the virtual machine are two separate components executing within that run-time environment. S-RVM uses the type system design to separate these two components and to protect the internal state of the virtual machine from the application.

The rest of this chapter introduces the work presented in this thesis. It presents the classification, the type system and the implementation. The chapter concludes with a summary of the contributions of this thesis and a road map to the rest of the thesis.

1.1 Classification of Reference Protection Systems

As mentioned above, an extensive body of research into reference protection exists. Comparing research in this area is complex due to the absence of a consistent description that applies to all systems. The purpose of the classification framework described in this thesis is to provide such a description.

The classification is based on the concept of a *zone*, which is a group of objects to which protection is applied uniformly. The thesis defines zone types that are sufficient to describes the protection policies used in existing systems.

- An *isolated zone* consists of objects that can only be referenced from within the zone.
- A *shared zone* consists of objects that can be referenced from multiple zones.
- A *sealed zone* contains objects that can be referenced from within the zone as well as from a single *interface* zone. The interface zone itself is a shared zone.

Privileged zones are special zones that are not affected by the restrictions of other zones. That is, references from objects in a privileged zone are permitted to any zone, overriding the restriction of the target zone. Privileged zones appear in components that implement system functionality and may, therefore, require access to any object in the system. The privileged zone itself can be of any of the three zone types.

The combination of zones in a system and the way these zones are associated with the components defines the *protection model* of the system. Existing research on reference protection offer four protection models which represent different trade-offs between the level of security and the efficiency of communication mechanisms.

Complete isolation is achieved by using isolated zones exclusively, where the data of different components resides in different zones. As references between components are not permitted, components cannot *directly* affect the computation of one another. Hence, the components are isolated from each other. With the absence of inter-

component references, communication between components can only be achieved through system-provided communication channels.

Isolation with object sharing extends complete isolation by adding shared zones. Objects in the shared zones can be used for sharing data between components. Sharing facilitates transferring information between domains. Sharing, however, reduces the level of isolation because components can directly affect the computation of other components by modifying shared values.

Partial isolation is achieved by the use of sealed zones. A sealed zone encompasses the private data of each component. Objects in the corresponding interface zone are permitted to hold references into the sealed zone. These objects are, themselves, shared, hence other components may hold references to the objects in the interface zone. Consequently, by invoking methods of objects in the interface zone of a component, other components can get indirect access to the private data of that component. By providing indirect access to private data, partial isolation offers a lower level of security than object sharing. At the same time, the ability to invoke code in another component provides communication options that are not supported by the more strict levels of isolation.

When components are created in systems providing *initial isolation*, the data of a component is initially, i.e. when the component is initialised, isolated. The component may, however, share references to its internal data and the system does not enforce any restrictions on this sharing. With this model the decision on the trade-off between communication and security is left to the developers.

Figure 1.1 shows where the described protection models fit within the spectrum of isolation. Systems to the left of the chart offer fewer restrictions on communication and, consequently, less security. Systems to the right of the chart are more secure but only offer less efficient channels of communication.

Figure 1.1: The spectrum of isolation using reference protection

The spectrum can be extended both to the right, through the use of hardware protection mechanisms or separate hardware, and to the left, for example by not using reference protection or by using languages with no type safety. As the topic of this thesis is reference protection, these extensions are outside the scope of this thesis.

Because the partial isolation model does not allow references to objects in the private data of components, the level of isolation it provides is sufficient for component isolation. As such, of all the models described, partial isolation provides the

most flexible communication options, without breaching the isolation requirements. Yet, past implementations of the model for component isolation enforce the model using run-time tests, which introduce a performance overhead. The next section describes a design of a type system that uses static type rules to enforce partial isolation using sealed zones.

1.2 Exported Types

Exported Types is a design of the essential characteristics of a type system that provide component isolation with sealed zones. Exported Types can be applied to the type system of existing languages as well as be a part of the design of a new type system.

The main benefit of Exported Types is that it enforces partial isolation at compile time. That is, the type information associated with references is sufficient for the compiler to prevent assignments that would breach the reference-protection model. Thus, Exported Types avoids the overhead of making reference protection related decisions at run time.

To separate components, Exported Types partitions the types in the system into multiple name spaces and associates each component with a name space. Textual type names used by a component are resolved within the component's name space. Unless otherwise configured, name spaces are disjoint. The types in different components are not shared and the type safety of the language prevents inter-component references.

Two operations are used for creating the interface zones. Components *export* types to make these types available to other components. Components *import* previously exported types, binding the imported type names to the previously exported types. Thus, objects of exported types are accessible from components importing the types, allowing efficient inter-component communication.

Exported types may include members that should not be accessible from outside the exporting component. This could be the result of the exported type inheriting these members from its superclasses or could be a software engineering decision of providing elevated privileges to the code of the exporting component.

Protecting members from access from outside the component is achieved by adding a *remote* scope for exported types. Members of exported types that can be accessed remotely are annotated to be included in the remote scope. Members not annotated are only accessible from the exporting component. The use of an explicit scope ensures that only methods intended to be used remotely are used from outside the component.

1.3 Implementation of Exported Types

S-RVM is a proof of implementation of providing reference protection using the Exported Types design. It is a Java virtual machine based on JikesRVM [Alpern et al., 1999, 2000]. Like JikesRVM, S-RVM is a meta-circular virtual machine. That is, it is written in the same language and executes in the same run-time environment that it provides. Unlike JikesRVM, S-RVM maintains a strict separation between the virtual machine and the application. Separating the virtual machine from the application protects the internal data and interfaces of the virtual machine and ensures that the application does not access them. Hence, the design of S-RVM is inherently more secure than that of JikesRVM.

S-RVM uses the Exported Types design to separate the application from the virtual machine. The virtual machine and the application each execute as separate components. Reference protection prevents references from the application to private virtual-machine objects. Yet, the use of the Exported Types design, allows the virtual machine to export types of objects that provide virtual-machine services to the application. Application objects may hold references to objects of these exported types.

One consequence of the design of S-RVM is that it maintains separate copies of library types for the virtual machine and for the application. Each of the copies of the library is optimised for its corresponding workload. Consequently, the performance of optimised code in S-RVM is marginally better than in JikesRVM.

Basing S-RVM on an existing virtual machine provides a real, non-artificial baseline to compare the performance of the design. The interface between the application and the virtual machine is frequently exercised during the lifetime of the application. Hence, using Exported Types in this specific settings heavily uses the interface between components, and demonstrates that the Exported Types design can provide the added security with no performance cost.

1.4 Summary of Contributions

This thesis challenges the common point of view that isolation costs and demonstrates that there is no need to sacrifice performance for the sake of security.

The thesis presents a new classification framework for systems providing reference protection. The classification captures the salient features of reference protection with respect to both the level of security and the efficiency of inter-component communication. It creates a unified terminology for comparing systems from across the domain and is useful for specifying isolation properties when designing reference-protection mechanisms.

The thesis also provides the first design of type system that enforces the partial isolation protection model. The design can be used as a basis for a type system for a new language as well as to extend an existing type system.

S-RVM, a proof of implementation of the type system design is described. The implementation enhances the security of the JikesRVM Java virtual machine by isolating the implementation of the run-time environment from the application executing within it. S-RVM slightly outperforms JikesRVM in the steady state. While this improvement is not statically significant, S-RVM does achieve the goal of providing increased security at no performance cost.

1.5 Thesis Structure

The next chapter develops the classification of reference-protection systems and presents a survey of existing systems. It identifies the absence of a solution that enforces partial isolation at compile time. This gap motivates the development of the Exported Types design, described in Chapter 3.

S-RVM, described in Chapter 4, is an application of the Exported Types design to separate the virtual machine from the application in a meta-circular virtual machine. The performance of S-RVM is analysed in Chapter 5, which demonstrates that the design provides the security of isolation with no performance degradation.

Chapter 6 presents the conclusions of this work and suggests directions for future research.

The appendices provide a semi-formal description of the Exported Types design (Appendix A), a more detailed information on the implementation (Appendix B) and detailed performance data (Appendix C).

Chapter 2

A Model and Classification of Reference Protection

This thesis promotes the use of the partial isolation model for component isolation using reference protection. This chapter introduces the vocabulary required for discussing reference-protection models, including the description of the partial isolation model. The chapter presents a classification of reference-protection models that succinctly captures the trade-offs the models make between the level of isolation and efficiency of inter-component communication.

The next section presents the novel concept of *zone*s, which is central to the classification of reference-protection models. The following four sections present the protection models identified in this thesis and surveys existing systems that support each of the models. Section 2.6 discusses the techniques used for providing reference protection. Section 2.7 concludes the chapter with a discussion of the effectiveness of the classification.

2.1 Zones

Memory protection, or the ability to keep memory references local to a scope, is not a new concept [Lampson and Redell, 1980, Morris, 1973]. For many years, however, research into memory protection was focused on hardware mechanisms and how to use them to protect memory [Bershad et al., 1995a]. Early languages and software-based operating systems such as Mesa [Lampson and Redell, 1980, Mitchell et al., 1979], Pilot [Redell et al., 1980], Cedar [Swinehart et al., 1986] and SPIN [Bershad et al., 1994, 1995b, Sirer et al., 1996] do not support software-based memory protection.

Research into software-based control of the propagation of references, or *reference protection*, started shortly after the introduction of the Java programming language.

From its inception, Java was designed to handle mobile code and the security issues related to it [Gosling and McGilton, 1996]. Language mechanisms to control the behaviour of mobile code were included in early versions of Java and were later developed, becoming the security architecture of modern Java implementations. Java, however, does not support reference protection. Consequently, while it may be possible to create a Java-based operating system [Sun Microsystems, Inc., 1996], the language's lack of reference protection limits its use for component isolation and the protection that the operating system can provide for components [Back and Hsieh, 1999, Czajkowski, 2000].

Many approaches for adding component isolation through reference protection to Java and other languages have been suggested [Aiken et al., 2006, Back and Hsieh, 2005, Back et al., 2000a, Binder et al., 2001, Czajkowski, 2000, Czajkowski and Daynès, 2001, ECMA, 2006, Golm et al., 2002, Hawblitzel et al., 1998, JNode, Rust Reference Manual, Spring et al., 2007, Tullmann et al., 2001]. Naturally, reference protection restricts sharing and, therefore, limits communication. In systems that enforce strict isolation between components no sharing of data is supported and all data communication requires copying. In other systems, sharing of data and of computation is permitted at the cost of a less rigid isolation.

While many approaches for component isolation have been suggested there is a lack of a unifying terminology that allows comparing the different approaches to reference protection. To better understand the landscape this thesis introduces the concept of *zone*s, where a zone is a group of objects to which a protection rule is applied uniformly.

The role of zones in a reference-protection system is similar to the role of memory segments in hardware-based memory protection [Bensoussan et al., 1969, Organick, 1972]. To avoid the need to track the permissions for each and every memory address, hardware protection mechanisms combine ranges of addresses into segments and apply protection rules uniformly to all the addresses within a segment. In a similar fashion, because it is easier for the run-time environment to make decisions based on groups of objects than to track permissions for each and every object, zones are implicit in the design of reference-protection systems.

Zones can be classified according to how inter-zone references are permitted and managed. The classification provided here identifies types of zones that represent a trade-off between the level of isolation and the ease of communication that reference-protection systems provide. These zone types form the basic building blocks from which any reference protection system constructs its reference protection policy.

Isolated Zones

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

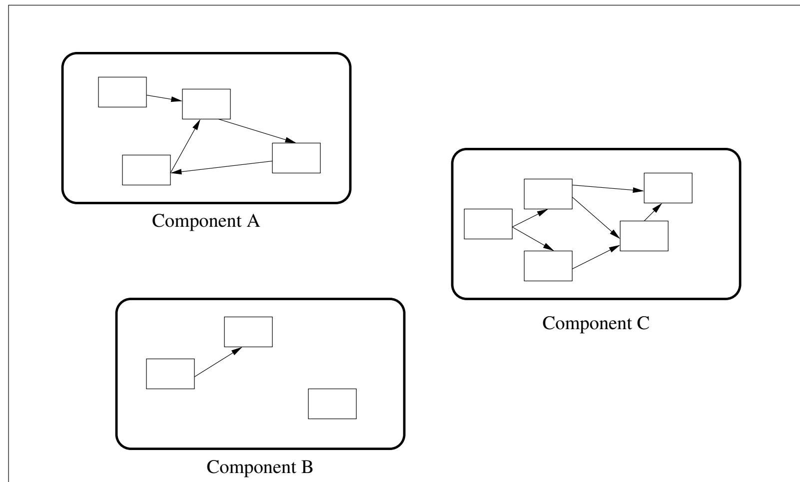

Figure 2.1: A component system with isolated zones

Figure 2.1 shows a system with multiple isolated zones. Objects can hold references to other objects within the zone. However, references across zone boundaries are not permitted. As external references into the isolated zones are prohibited, these zones are the reference-protection equivalent of task-private memory segments in traditional operating systems [Organick, 1972].

The main advantage that isolated zones offer is the guarantee that, due to the lack of external references, the values in the zone can only be accessed by the component associated with it. Thus, isolated zones offer security and simplify programming. This isolation, nevertheless, has a downside. Because external references to the zone are not allowed, components cannot share the data in the zone. The only way to communicate data is using message passing, i.e. by copying it out of the zone.

Shared Zones

Shared zones are zones that can have incoming references from multiple other zones. They are the reference-protection equivalent of shared segments in traditional operating systems and are used in a similar manner [Organick, 1972]. Systems that support shared zones use them in combination with isolated zones. In such systems isolated zones are used for components' private data and shared zones are used for efficient communication between components.

A system supporting multiple components that communicate using shared zones is presented in Figure 2.2. As can be seen in the diagram, each of the two components has its own isolated zone. References from these zones to the shared zone are

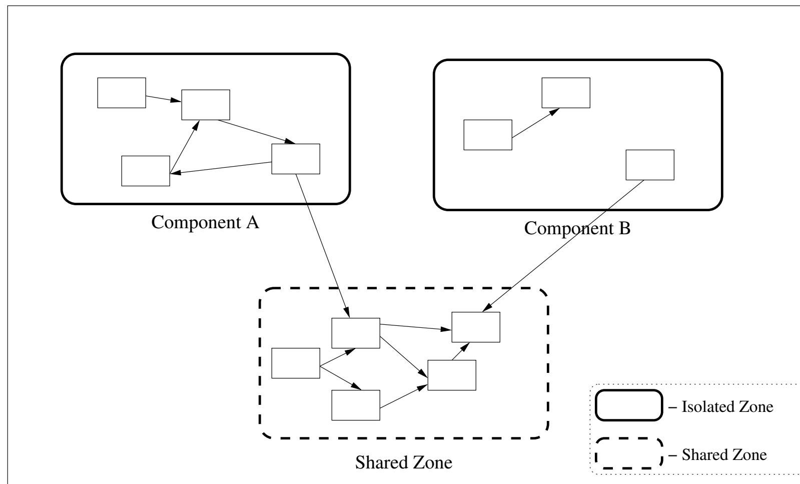

Figure 2.2: A component system with isolated zones and a shared zone

permitted. However references to the isolated zones, either from the shared zone or from other isolated zones, are prohibited.

Sealed Zones

A *sealed* zone is a zone that can only have incoming references from a corresponding *interface* zone. The interface zone itself is shared and external objects can hold references to it. Sealed zones and their corresponding interface zones are used to model the private data and the communication interface of components. The sealed zone contains the private data of the component and the interface zone contains the public data. The interface zone "wraps" the sealed zone controlling all external access to the private data.

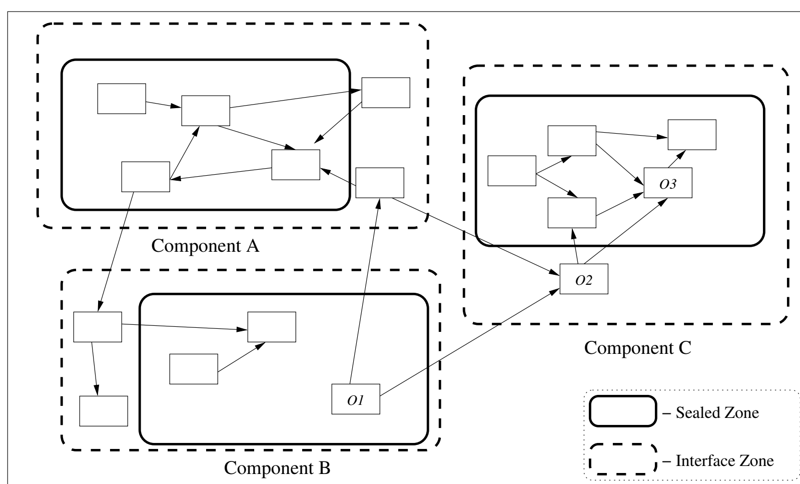

Figure 2.3: A component system using sealed zones

Sealed zones are demonstrated in Figure 2.3. As can be seen in the diagram, external references to objects in the interface zones are permitted. However references

to objects in a sealed zone are only allowed from objects within that zone or from objects in the corresponding interface zone.

Sealed zones provide a more relaxed isolation than shared and isolated zones provide. In sealed zones systems, a component can refer indirectly to another component's private data. Figure 2.3 shows an example of this, where object $O1$ in Component B refers, via object $O2$, to object $O3$—a private object of Component C.

The main advantage of sealed zones is that they support method invocation across component boundaries. Systems using isolated zones only cannot share data. Instead they must use sequential communication mechanisms to copy data.

Systems supporting shared zones reduce the overhead of communication by sharing data, but they do not include a mechanism for transferring control. To share computation between components, the components must agree on a Remote Procedure Call (RPC) protocol.

With sealed zones, a component can hold references to objects in the interface zone of another component. It can use this reference to invoke methods of these objects, effectively gaining access to the other component. This access is not, however, unrestricted. Type protection can be used to limit the access from external components to those members of interface objects that are safe for external access.

Systems using sealed zones can support nested zones. Such systems allow more flexibility in the access control than is possible with a flat system. Ownership types, described in Section 2.4.3 is an example of a system supporting nested sealed zones.

Sealed zones have no immediate counterpart in memory protection. Some hardware-based mechanisms that achieve similar effects have been suggested. System calls in traditional operating systems are the most common example of transferring control across component boundaries. Another example is lightweight remote procedure calls [Bershad et al., 1990], which use a combination of system traps and system code to allow transfer of control between tasks.

Privileged Zones

Each of the zone types described above can be designated as being *privileged*. Outgoing references from privileged zones are allowed to any object, overriding the restrictions of the target zone. Privileged zones are used for system code that may require access to any object in the system. They are, therefore, the reference protection counterpart of the traditional operating-system kernel.

Systems combine the zone types described above to construct their *reference-protection model*, which is a broad classification of the permissions and restrictions the system imposes on references. This thesis identifies four reference-protection models. Systems that provide the *complete isolation* model use isolated zones exclu-

sively. With no references crossing zones boundaries, this model achieves the highest level of isolation. However, inter-component communication in these systems cannot use data sharing, introducing an overhead through the use of data copying and marshalling.

The *object sharing* model uses both isolated and shared zones. Isolated zones are used for the components, whereas shared zones are used for shared data. As the model supports data sharing, inter-component communication in systems using this model is more efficient than when the complete isolation model is used. However, sharing reduces the level of isolation provided by the system.

Partial isolation is a reference-protection model that uses sealed zones and their corresponding interface zones to model components. The model allows indirect references to the private data of a component. Hence, it further relaxes the isolation provided by the object sharing model. At the same time, indirect references allow components to invoke methods on objects within other components, enhancing the communication options provided by the system.

The fourth model is *initial isolation*. In this model, when components are instantiated they are isolated from other components. However, the system provides primitives for sharing object references between components without imposing restrictions on what can be shared. Consequently, in the initial isolation model, the onus of ensuring that references to private data do not escape from the component lies with the programmer.

The next four sections provide a survey of systems supporting reference protection, grouped by the reference-protection model provided by the systems.

2.2 Complete Isolation

Systems providing complete isolation aim to achieve a level of component isolation similar to that provided by a traditional operating-system process abstraction. By preventing any sharing of data, these systems ensure that components can only communicate through system-defined communication channels.

Systems using this approach partition the object space to isolated zones such that each component has its own zone. Using only isolated zones implies that the system prohibits both incoming and outgoing references from the components' data.

Some of the systems in this section use a shared zone for objects that are guaranteed to be immutable. Sharing immutable objects relaxes the isolation slightly, but still prevents a component from changing data visible to other components.

2.2.1 MVM

The Multitasking Virtual Machine (MVM) [Czajkowski, 2000, Czajkowski and Daynès, 2001, Czajkowski et al., 2003] is a Java virtual machine that executes multiple independent tasks, or *isolates* [Java Community Process, 2006] in the same virtual machine. The system aims to isolate the tasks from one another, while sharing as much code and metadata as possible.

MVM isolation is based on the observation that the only initially shared values in Java classes are the static fields, the associated `Class` objects and `String` literals. By providing each application with its own set of values MVM completely isolates the applications.

Reports on MVM describe two versions. The earlier [Czajkowski, 2000] is implemented using bytecode manipulations that replace static fields with arrays and the static field access operations with array access operations. The later version [Czajkowski and Daynès, 2001] modifies the virtual machine itself to achieve a similar effect, however class representation, including both bytecode and compiled code, remains shared between the applications.

Sharing the code amortises the memory footprint of the virtual machine over the running applications, reducing the total memory footprint required for running multiple instances of the same application. It also reduces the application start up time for repeated executions of the same application. Shared compiled code may, however, result in a loss of optimisation opportunities. For example, constant propagation of static fields cannot be used if the field's value is not shared [Czajkowski, 2000].

The overhead of replacing all static fields ranges from almost nil for benchmarks that hardly use static fields to about 70% when static fields are heavily used. When sharing static final primitive fields, `String` literals and constant arrays, the maximum overhead is about 5% for the first version [Czajkowski, 2000] or 7% for the second [Czajkowski and Daynès, 2001].

2.2.2 JX

The JX Operating System [Golm et al., 2001, 2002, Wawersich et al., 2002] is an operating system that employs software protection for component isolation. Its main aim is to demonstrate that a Java-based operating system can be built without a large performance degradation.

JX has a small microkernel written in C and assembly. The rest of the system is composed of *components* loaded into *domains*. JX Components are collections of code and do not have any specific role in protection. Thus, JX components are not components in the sense used for the rest of this thesis. In JX, the unit of isolation,

or what the rest of this thesis refers to as "component," is the JX domain.

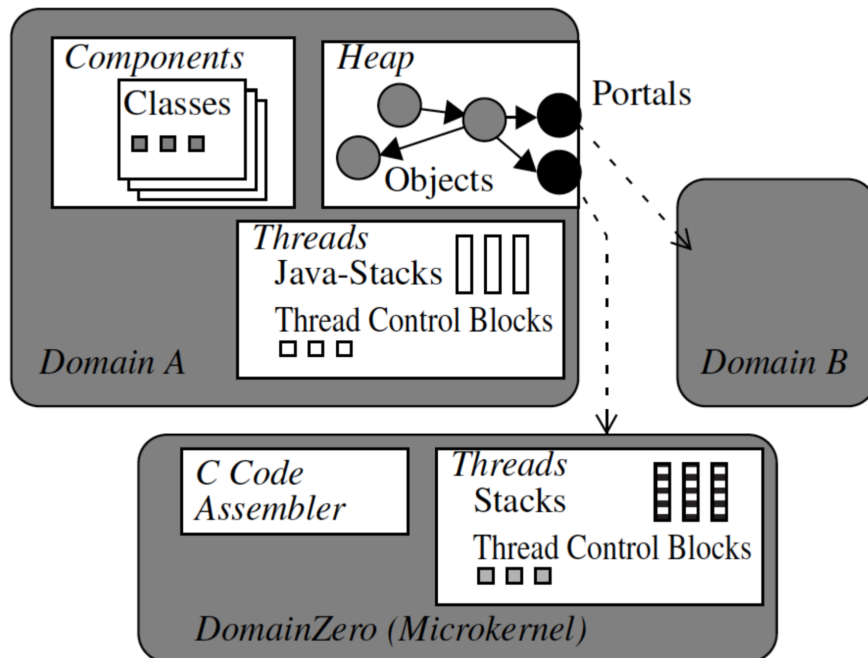

Figure 2.4: Structure of the JX system (from Golm et al. [2002])

Figure 2.4 demonstrates the structure of the JX System. Each domain maintains its own heap, garbage collection and threads and has its own type hierarchy. Objects of one domain are not assignment compatible with types of other domains. Consequently, cross-domain references violate the type safety and cannot be created.

Domains communicate via *portals*, which are communication endpoints attached to *services* in other domains. Portals are, technically, Java interfaces that extend the `Portal` interface. To maintain method invocation semantics, the calling thread blocks on portal calls while a service thread at the target domain handles the request. Arguments of portal calls are deep-copied to avoid cross-domain references.

Deep copying implies that, unlike Java semantics, objects are passed by value. A consequence of passing by value is that objects lose their identity when transferred across portals. This may result, for example, in two copies of the same object being created in the server.

A JX component can be loaded into multiple domains. To reduce the memory footprint the code of the classes in the JX component is shared between all the domains loading it.

2.2.3 The .Net Framework

The Microsoft .Net framework [ECMA, 2006] is a software framework used primarily on Microsoft Windows, with the aim of providing a common environment for running applications written in multiple languages and supporting language interoperability.

Applications in .Net run within *application domains*, which are completely isolated from each other.

Software components (*assemblies*) are loaded into application domains. Assemblies loaded as *domain neutral* share the code across all domains they are loaded into. Assemblies that are loaded as *domain specific* have their own copy of the code. Sharing code allows the system to reduce the memory footprint at a slight performance cost.

.Net supports the concept of remote types which allow applications to communicate across the application-domain boundaries. Remote types are implemented as proxies that transparently marshall and send data across the communication channels. The marshalling process creates proxies for objects of remote types. Deep-copying is used for other objects.

2.2.4 JNode

The Jnode Operating System [JNode] is an attempt to build a Java-based operating system. Jnode implements the Application Isolation API [Java Community Process, 2006], where programs execute within separate entities called *Isolates* and communicate using communication channels called *Links*. Isolates are isolated by providing each with its own set of values for static fields.

The JNode operating system itself executes within the *root isolate*. As the root isolate manages the system, it needs to be able to access all the objects in the system, regardless of the isolate the objects belong to. Consequently, references from the root isolate to other isolates are not restricted, and the root isolate is privileged.

Furthermore, to avoid the communication overhead through Links, isolates hold references to root isolate objects. JNode isolation, therefore, only applies between non-root isolates. As all isolates can hold references to root isolate objects, the root isolate is a shared zone.

2.2.5 OVM

OVM is a real-time meta-circular Java virtual machine. The system architecture consists of one *executive domain* which runs the virtual machine and multiple *user domain*s that execute user programs [Armbruster et al., 2007, Spring et al., 2007].

Each domain in OVM has its own name space which is completely segregated from other domains. Hence, types in one domain are not assignment compatible with types in other domains and cross-domain references are prohibited. Consequently, the objects in each domain form an isolated zone.

The zone of the executive domain is privileged. The executive domain includes a special type `Oop`, which can refer to any object in the system. `Oop` references are

used by the executive domain for managing the user domains.

Cross-domain calls are supported between the executive domain and the user domains. The executive domain uses reflection on `Oop` objects to invoke methods in the user domains. The user domains use methods of the type `LibraryImports` for invoking executive domain methods. The compiler intercepts invocation of `LibraryImports` methods and translates those to calls to methods of a `RuntimeExports` object in the executive domain.

Communication between the user domains and the executive domain in OVM does not, therefore, require data marshalling or copying. However, this communication mechanism relies on the privileged nature of the executive domain and on the special treatment of the `LibraryImports` class. Hence, this communication mechanism cannot be extended to general cross-domain communication without breaking the isolation properties of the system.

2.2.6 Summary

Three observations are made on the systems described above. The first is that static fields and boxed literals are implicitly shared by all code running in a virtual machine. Having a separate copy of these values for each component is, therefore, a necessary condition for complete isolation.

The second observation is that in a type-safe language there is no way to forge or create a reference to an object. The only way code can get a reference to an object is by following the reference chains from objects it already has references to. Combining these observations implies that in the absence of an explicit way of transferring references between components, separating the static values and boxed literals is sufficient for complete isolation.

The third observation is that where performance information is available, sharing code between components presents a trade-off between memory footprint and performance. Sharing reduces the amount of memory required for storing multiple copies of the code, but there are limitations on optimisations that can be applied to shared code, reducing its overall performance.

Components in systems using complete isolation cannot use the memory for communication and must rely on communication channels. Communication channels introduce an overhead for marshalling and unmarshalling data. They also introduce changes in language semantics due to the pass-by-value nature of communication.

Sharing objects between components can reduce the communication overhead.

2.3 Object Sharing

Isolation with object sharing is the reference-protection equivalent of using shared memory. In systems that provide isolation with sharing, components are still associated with isolated zones. In addition, the system provides shared zones which are used for inter-component communication. These systems restrict references coming into the components' private data but allow outgoing references.

Object sharing reduces the overhead required for marshalling and unmarshalling objects across communication channels.

2.3.1 KaffeOS

KaffeOS [Back, 2002, Back and Hsieh, 2005, Back et al., 2000a,b] is a Java operating system that supports the abstraction of a process. It is based on the Kaffe virtual machine [Kaffe] which is a free implementation of a Java virtual machine. Each process in KaffeOS has its own heap, where a process's heap is garbage collected independently of other processes' heaps. References between process heaps are prohibited.

To facilitate efficient inter-process communication, KaffeOS supports sharing objects between multiple processes. Shared objects are located within special shared heaps. Types of objects in shared heaps come from a central name space, ensuring that all processes sharing the heap have the type information for the objects in the heap.

KaffeOS also includes a kernel heap which is used for the implementation of KaffeOS itself. All heaps can hold references to kernel objects and kernel object can hold references to any object in the system. The kernel heap is, therefore, a privileged shared zone. Figure 2.5 demonstrates the heaps structure of KaffeOS.

Once created, a shared heap is frozen. New objects cannot be created and objects are not reclaimed from the heap. Heap objects are reclaimed *en masse* when none of the heap's objects is externally accessible. Freezing a shared heap is required for meeting KaffeOS's goal to account precisely for the memory used by processes [Back and Hsieh, 2005].

While the composition of a shared heap cannot change, objects in the heap are not constant and their fields are mutable. Write barriers [Wilson, 1992] are used to ensure that references from shared heaps do not point to objects in the processes heaps. The write barriers, which are invoked on every reference store, cause a performance overhead of up to 7% [Back et al., 2000a]. The total overhead of KaffeOS over Kaffe is up to 25% [Back and Hsieh, 2005].

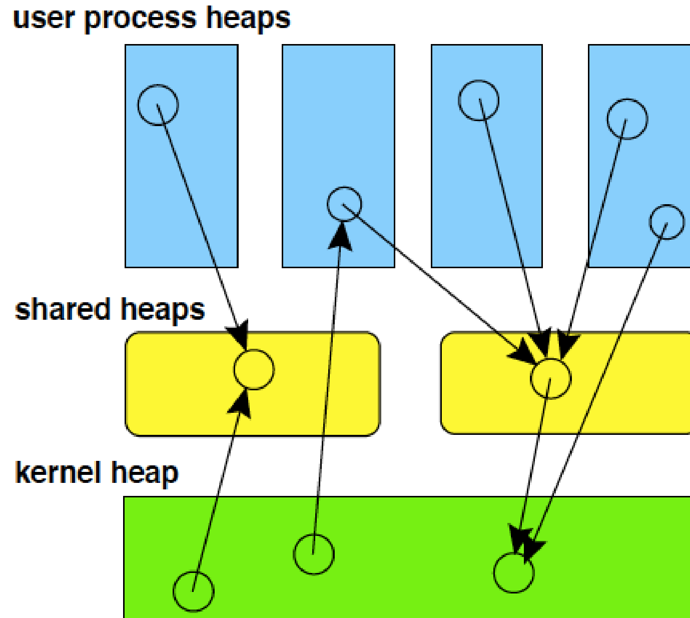

Figure 2.5: Heap structure in KaffeOS (from Back et al. [2000a])

2.3.2 Singularity

Singularity [Aiken et al., 2006, Fähndrich et al., 2006, Hunt et al., 2005] is an operating systems research project in Microsoft. The approach it takes is: *building on advances in programming languages and programming tools to develop and build a new system architecture and operating system (named Singularity), with the aim of producing a more robust and dependable software platform* [Hunt et al., 2005].

The Singularity operating system supports two levels of isolation. Hardware Isolated Processes (*HIP*s) are similar to processes in traditional operating systems. Each HIP has its own virtual address space and protection is maintained using the hardware protection mechanisms of the processor.

Software Isolated Processes (*SIP*s) are, as their name implies, processes that rely on software protection for isolation. Several SIPs can reside in a single HIP, each one of them being a closed object space, with no references crossing between SIPs.

SIPs communicate using *channels*, which are strongly typed connections between processes. The Sing# language used in Singularity is a variant of C# that provides constructs for specifying program behaviour. One of the features Sing# provides is *channel contracts* which specify the legal sequences of messages on channels.

Each HIP also features an *exchange heap* (Figure 2.6) that can be used for sharing data between SIPs. The syntax of Sing# makes an explicit distinction between references to values in the process's heap and references to values in the exchange heap. Singularity restricts the types of values in the exchange heaps to primitive types,

structures and vectors of exchange heap types. Structures are further restricted to not include references to the process's heap.

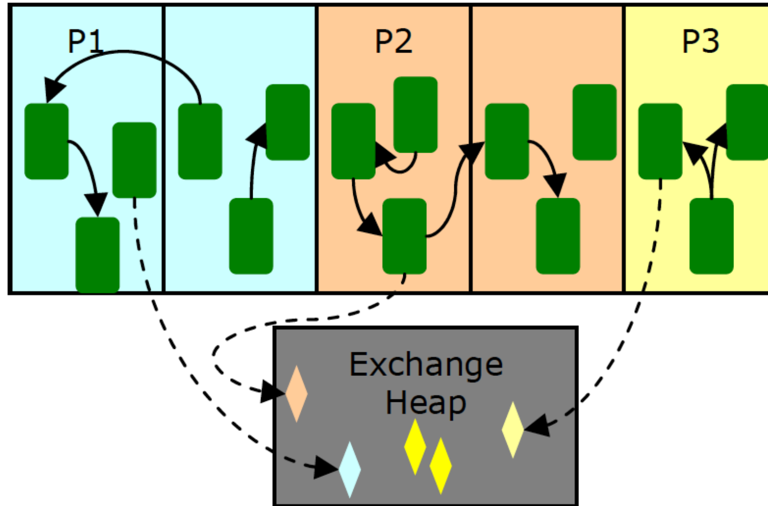

Figure 2.6: The exchange heap in singularity (from Hunt et al. [2005])

The restrictions on values in the exchange heap imply that all references in the exchange heap must point to values in the exchange heap. Hence, the exchange heap cannot propagate references to processes' heaps, ensuring that SIPs are isolated. Also, since the exchange heap cannot contain objects, Singularity does not allow sharing behaviour. Only data can be shared between processes.

The semantics of Singularity prevent concurrent access to shared values. A value in the exchange heap has exactly one owner that can access it. References to shared values can be transferred in messages. Transferring a reference also transfers the ownership of the referenced value preventing any further access by the sender.

2.3.3 Rust

The Rust programming language [Rust Reference Manual] aims to provide a type-safe, memory-safe and concurrent programming platform. Its design is oriented towards *creating and maintaining boundaries—both abstract and operational—that preserve large-system integrity, availability and concurrency* [Rust Reference Manual].

Rust supports lightweight tasks and the primitives required for task communication. Each Rust task has its own garbage-collected heap. These heaps are completely isolated, with no cross-task references allowed.

In addition, Rust supports a shared heap that is used for values that are shared between tasks. As in Singularity, no concurrent access to shared values is supported, yet references to shared values can be transferred between tasks without the need to copy these values.

Tasks communicate using communication channels that allow sending and receiving values. Only scalar values and pointers to shared-heap values can be sent in messages.

The shared heap itself is created implicitly and consists of all the *owned values* of *owned types*. Owned values in Rust are values pointed to by *owning pointers*, which is Rust term for unique pointers [Aldrich et al., 2002, Minsky, 1996]. Each owned value has exactly one owning pointer referring to it. Owning pointers are moved rather than copied, ensuring that the one-to-one relationship between owning pointers and owned values is maintained.

Owned types in Rust are all the scalar types, the types of owning pointers to owned types and structural types whose members are owned types.

The structure of the shared heap prevents data races by not allowing concurrent access to values. By only allowing transfer of owning pointers of owned types, Rust guarantees that the receiving task is the only task that has access to the transferred value or any of the values reachable from it.

One consequence of the design of owned types is that pointers in owned types always refer to values in the shared heap, ensuring that no references from the shared heap to the tasks heaps exist. Another consequence is that it enforces a tree structure on values in the shared heaps. Cyclic structures in the shared heap are not reachable and cannot be used.

Rust initialises new tasks with a closure [Landin, 1964] that specifies both the function the task executes and the values in the environment that the function needs. Only owned closures (closures with data in the shared heap) can be used for task initialisation. The ownership of the closure is transferred to the new task, ensuring that no part of the initial environment of the task is shared with any other task.

2.3.4 XMem

The aim of XMem [Wegiel and Krintz, 2008] is to provide a type-safe shared memory between isolated programs, each executing in a separate operating system process. Shared objects are allocated in a shared memory region which is mapped in the same virtual address of all participating processes.

XMem only shares object representations. Class metadata and static fields are stored in the private data of each process. XMem ensures classes of shared objects are loaded on all participating processes. It also verifies that all participating processes use the same implementation of globally-shared classes. Each program also executes an XMem management thread. These threads cooperate to perform global operations, such as shared class loading, which require synchronisation across the system.

To maintain referential integrity within the shared region, XMem disallows references from it to private objects. XMem employs a write barrier to check each reference-store operation and verify that it does not breach the restriction. As with KaffeOS, the write barriers impose an overhead, which in XMem's case ranges from 0.3% to 3.5% [Wegiel and Krintz, 2008].

XMem does not provide reference protection and relies, instead, on hardware-based mechanisms for protection. However, having a type-safe shared memory region and the need to maintain referential integrity raise issues similar to those found in systems that offer reference protection with object sharing. XMem's approach to these issues is, therefore, relevant to the current discussion.

2.3.5 CoLoRS

Co-Located Runtime Sharing (CoLoRS) [Wegiel and Krintz, 2010], like XMem, is a system that provides type-safe shared memory between isolated programs. Like XMem, CoLoRS creates a shared memory region and maps it in the same address in all participating processes. Unlike XMem, CoLoRS targets heterogenous environments, allowing sharing e.g. between Java and Python programs.

The shared memory region in CoLoRS is managed by a dedicated server process. The use of a separate process obviates the need for global synchronisation and the per-program threads found in XMem.

Referential integrity of the shared region in CoLoRS is maintained by write barriers. To reduce the overhead of these barriers, CoLoRS optimistically compiles methods with the assumption that they only access private objects. CoLoRS guards these methods, and reverts to the interpreter and to recompilation if the assumption does not hold. Even with the speculative optimisation, CoLoRS incurs a performance overhead of 5%.

2.3.6 Real-Time Java

Real-time Java is a specifications of an extension of the Java language to support real-time programming [Bollella et al., 2000]. As part of its memory management, real-time Java supports *scoped regions* [Tofte and Talpin, 1994]. Scoped regions create a hierarchy of regions, such that objects in higher levels of the hierarchy can hold references to objects in lower levels, but not vice versa. This restriction allows collecting the top level regions *en masse* when they become unreachable, thereby reducing the pressure on the garbage collector. By not requiring the use of a garbage collector for collecting scoped regions, the system can meet the real-time guarantees.

The specifications do not specify how to prevent references from lower-level regions to higher-level regions. It does, however, require that assignment are checked

before being executed, thereby hinting at the use of write barriers. It also suggest that optimisations through the use of static analysis are acceptable. OVM implements the real-time Java specifications within its user domains [Armbruster et al., 2007]. The OVM implementation uses wrte-barriers.

Objects in scoped regions can be referenced by objects in multiple other regions. Hence, scoped regions are shared zones. Scoped regions, however, are not designed for component isolation, consequently, the design does not specify any isolated zones. Furthermore, while there are obvious restrictions on reference propagation in the system, the classification presented here is not expressive enough to capture these.

2.3.7 Summary

The main issue that systems providing isolation with object sharing need to address is how to prevent references from the shared zone to isolated zones. Unless restricted, code that has access to both the shared and the isolated zones could break reference protection.

Some systems use write barriers that check reference-store operations at run time to ensure that the operations do not breach reference protection. Validating each and every store operation can incur a significant performance overhead. However, compiler techniques, such as the optimistic optimisation in CoLoRS, can reduce this overhead.

An alternative approach for preventing breaches to reference protection is to limit the types of objects in the shared zones so that references from them to objects in the isolated zones are not type safe. When type safety is enforced at compile time, this approach does not incur a run-time performance overhead.

Shared zones facilitate sharing of large quantities of data between components. However they rely on the use of additional mechanisms to transfer control, e.g. for invoking code in another component. Partial isolation, discussed in the next section, provides direct method invocation across component boundaries.

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.

Systems that support partial isolation use sealed zones and their corresponding interface zones to model components. Objects private to the component are within the component's sealed zone whereas public objects are in the interface zone. Because the interface zone is shared, components can access objects in the interface zones and by invoking methods of object in the interface zones components can get

an indirect access to other components' private data. The design of the system ensures that methods of the objects in the interface zones do not disclose references of objects in the sealed zones.

2.4.1 J-Kernel

The J-Kernel [Hawblitzel, 2000, Hawblitzel et al., 1998] aims to explore developing an operating system in Java and to enhance the protection provided by the language.

Protection domains in the J-Kernel are separated through the use of a special class loader which modifies the bytecode of loaded classes to prevent sharing through the system classes.

The J-Kernel is a capability-based system [Dennis and Van Horn, 1966, Needham and Walker, 1977], which uses capabilities to provide cross-domain access. Capabilities in the J-Kernel are implemented as wrapper objects which wrap objects that implement the `Remote` interface. Capability classes are automatically generated by the class loader when classes that implement the `Remote` interface are loaded.

The J-Kernel imposes a special calling convention on cross-domain method calls. In these calls, arguments and return values are passed by reference only if they are capabilities. The capability wrappers perform deep copy of non-capability objects to avoid sharing of these objects. An immediate consequence of these calling conventions is that cross-domain references are only allowed to capability objects. Thus, domain objects form a sealed zone, with the capabilities that wrap objects in the domain being the corresponding interface zone.

For a domain to get capabilities of other domains, there must be some mode of communication between domains. One mechanism suggested is a system-wide repository, which is basically a name service that allows domains to publish capabilities [Hawblitzel et al., 1998]. A second mechanism suggested relies on the hierarchical relationship between domains. When a parent domain creates a child domain, the parent domain gets a capability to the child which allows it to send capabilities to the child [Hawblitzel, 2000].

The main advantage of the J-Kernel capabilities design is that it provides remote method invocation. Furthermore, in addition to implementing the cross-domain calling conventions, the code in the capabilities supports capability revocation. By revoking a capability when the domain that created it terminates, capability revocation helps achieving clean domain-termination semantics.

The extra processing required for creating capabilities and converting data when invoking remote methods results in an overhead of about 10% [Hawblitzel et al., 1998].

2.4.2 Confined Types

The aim of Confined Types [Vitek and Bokowski, 2001] is to isolate objects internal to a Java package to prevent references from outside the package to these objects. The design imposes statically enforceable restrictions on types identified as *confined* to ensure that references to confined objects do not escape the package scope. These restrictions ensure, for example, that objects of confined types are not returned by public or protected methods of unconfined types. They also prevent inadvertent escape by disallowing widening of references of confined types to an unconfined supertype.

The combined effect of the restrictions is that only objects whose types are in the package can hold references to objects of confined types. As such, objects of the confined types of a package form a sealed zone, with objects of unconfined types in the package being the corresponding interface zone.

A major limitation of the confined types design is its inability to protect the components of compound data structures, i.e. data structures that are created from multiple objects. If these components are not defined in the package, they cannot be confined to the package and might, therefore, escape the scope. External access to these components may compromise the protection of the compound data structure. Consequently, while confined types may be a useful tool for protection, they are less than ideal for the purpose of general component isolation.

2.4.3 Ownership Types

Ownership Types [Aldrich and Chambers, 2004, Aldrich et al., 2002, Clarke, 2001, Clarke et al., 1998, Dietl and Müller, 2005] is a technique for alias control that uses reference protection to protect the internals of compound data structures.

Ownership types impose a hierarchical structure on the heap by assigning each object a single "owner". The ownership relation is specified by tagging references within structures. The syntactic rules of the language ensure that tagged references do not escape the scope they are restricted to.

Several models of ownership types have been suggested. Under the *owner-as-dominator* model [Clarke, 2001, Clarke et al., 1998], all reference paths from a public root to an owned object must go through the owner. With this model, the owned objects are private data which can only be accessed through a single public object—the owner.

The *ownership domains* model [Aldrich and Chambers, 2004] relaxes the strict requirements of the owner-as-dominator model by allowing owners to specify domains of owned objects with different access rules for each domain. This model supports multiple public objects owned by the same owner, all of them can access

the private objects.

The *owner-as-modifier* model [Dietl and Müller, 2005] allows external read-only references to owned objects. n this model, all owned objects are potentially accessible from outside the owner's scope. Hence, this model does not provide reference protection.

In ownership types, reference tagging only applies to instance fields. Consequently, ownership types cannot protect static members of types, limiting their use for component isolation.

2.4.4 Summary

Partial isolation allows direct communication between components. It also maps naturally to a component model where the component has a public interface and private data.

Both confined types and ownership types use separate types for the sealed zones and for the interface zones.[1] As the sealed types can be identified in compile time, reference protection in these system is statically enforced. However, these systems focus on alias control and are too limited for use in component isolation.

The J-Kernel does provide partial isolation of components, but it implements the model using run-time tests which incur a significant performance overhead. Enforcing partial component isolation at compile time is not supported in any system.

All the systems described so far offer reference protection, i.e. guaranteed restrictions on reference propagation. Some systems offer a more relaxed level of isolation, where components are isolated when instantiated, but there is no ongoing system guarantee that references do not propagate. This model is the topic of the next section.

2.5 Initial Isolation

To provide reference protection, the system needs to have two properties. It needs to instantiate components so that they are isolated and it must maintain this state throughout the execution of the code. This section discusses systems that offer the former without the latter.

Strictly speaking, these designs do not offer reference protection. That is, they do not impose any restriction on the propagation of references. Instead, they leave the onus of controlling what is shared with the programmer.

[1] Note that for ownership types, tagged and untagged versions of the same type name are not assignment compatible, hence they are not the same type.

Nevertheless, as these systems do offer some level of isolation and as they use techniques similar to those used by systems described earlier to achieve the initial isolation, a description of these system is included here.

2.5.1 I-JVM

I-JVM [Geoffray et al., 2009] aims to improve the security of the OSGi framework [OSG, 2011]. OSGi supports extensibility through the addition of *bundles*. To increase the security of OSGi, I-JVM employs the same isolation technique used by MVM: each bundle has its own set of static fields, `String` literals and `Class` objects.

When I-JVM initialises a bundle, it also provides the bundle with a reference to the shared OSGi name service. The name service can be used to register object references and to find foreign references. Once an object reference is shared, further references can be accessed through the interface that the object provides. I-JVM does not control this sharing of references or any subsequent reference propagation.

2.5.2 Alta

Alta [Tullmann, 1999, Tullmann and Lepreau, 1998] is a Java-based operating system that implements the nested process model [Ford et al., 1996] using software protection. Processes in Alta can spawn sub-processes which remain under the control of the spawning process.

Each Alta process has its own type space. This type space is controlled by the spawning process, which can substitute the class implementation for the spawned subprocess, if required.

While the type spaces of processes are disjoint, the virtual machine has sufficient information to allow sharing of object references while maintaining type safety. Alta uses interprocess communication channels to send these object references between processes. To restrict sharing, these communication channels can be monitored by the spawning process.

Accesses to shared objects are not considered part of the interprocess communication channels. These accesses are not restricted and may, potentially, allow unrestricted sharing of object references.

2.5.3 Summary

Both Alta and I-JVM improve the isolation that the Java language provides by ensuring that no external references point to the component's data upon initialisation. The components can, theoretically, maintain that state by not disclosing references to their data through the communication mechanisms that the systems provide.

Both systems, however, do not enforce this isolation. Components can share reference to their internal data and once shared, the systems impose no restrictions on the use of these data. Preventing further sharing is left to the programmer.

This concludes the exploration of the classification framework of reference-protection systems. The next section describes the techniques used for providing isolation and how they relate to the classification.

2.6 Isolation Techniques

Preceding sections demonstrate that the suggested zones model is sufficient to classify the existing systems. Examining the techniques used for component isolation further demonstrates the strength of the model. It turns out that there is strong relationship between the techniques a system employs for component isolation and the level of isolation it offers.

Systems that provide reference protection need to make two guarantees. They need to ensure that the reachability roots do not breach the reference-protection model. Reachability roots are those objects that are *a priori* accessible to a components. These include static values in classes, as well as other values that can be directly referenced, e.g. `Class` and `String` literals.

The second guarantee is that the execution of the program does not breach the model. That is, that there are restrictions on reference assignments that ensure that protection is maintained.

Various techniques have been suggested for maintaining both these guarantees. These are discussed below.

2.6.1 Reachability Roots

The *reachability roots* of a program are the set of values that can be accessed by the code directly, without the need to follow other references. In Java, for example, the reachability roots include static fields and the `Class` and `String` literals.

Different systems take different approaches for handling reachability roots. Figure 2.7 presents a taxonomy of the different approaches systems take.

Sharing Reachability Roots

A straightforward approach is to share the reachability roots. Sharing the reachability roots implies that the values referred by these roots are shared between components. This sharing must be consistent with the reference-protection model and is, therefore, not applicable to systems that only support isolated zones.

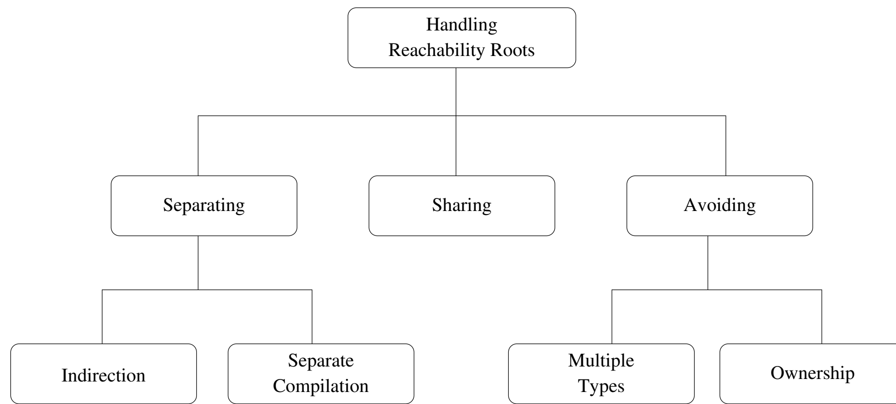

Figure 2.7: Taxonomy of methods for handling reachability roots

KaffeOS is an example of a system that shares reachability roots. In KaffeOS values of static members of library types are considered to be part of the kernel heap which is shared by all user processes. Sharing the reachability roots in KaffeOS is only applied to library classes. Application classes in KaffeOS are not shared between processes and therefore their static members are not shared.

Separating the Reachability Roots

Separating the reachability roots is a technique that provides each component with its own copy of the reachability roots. Systems use one of two implementation methods to achieve separation. The simpler method is to replace each reachability root with an array of values and modify the program code to reference the correct entry in the array based on the accessing component. The main advantage of this method is that it can be implemented without changing the internals of the virtual machine. Systems like MVM implement this method by modifying the Java bytecode as it is loaded into the virtual machine.

It should be noted, however, that this method is not fully transparent. Reflection and handcrafted bytecode can be used to access the array, which would breach the protection. Czajkowski [2000] argues that the reflection system can be modified to behave as it did with unmodified classes and that further checks can prevent loading code that directly accesses the arrays. However, these are not demonstrated and the question of whether it would be possible to implement these modifications with no changes to the virtual machine remains to be settled.

Replacing the reachability roots with arrays introduces an extra level of indirection which affects the performance of the code. It also results in loss of optimisation opportunities. Having an array of possible values for reachability root means that the compiler cannot assume that there is a single possible value for the reachability root and cannot use a constant value in the compilation. Czajkowski [2000] demonstrates that sharing of immutable reachability roots can reduce the overhead

associated with both issues but does not completely eliminate it.

The overheads of indirection can be completely eliminated by specialising the code for each component. With this approach, the code that accesses the reachability roots is compiled separately for each component the code is used in. As the identity of the component is known at compile time, the compiled code can bypass the indirection required for accessing the correct reachability root, thereby avoiding both the overhead of extra indirection and the loss of optimisation opportunities. While this approach generally provides a better performance, it does have some down sides. Multiple copies of compiled code have a larger memory footprint than a single copy would require. Furthermore, recompiling the code for each component uses CPU resources and results in a longer component set-up time. Finally, this approach cannot be implemented without modifying the virtual machine itself and is, therefore, less portable than using indirection.

Avoiding Shared Reachability Roots

Rather than trying to work around the problem of shared reachability roots, the language can be designed to avoid the issue. Two approaches for avoiding shared reachability roots have been suggested. One approach is to use a separate type hierarchy for each component. For example, Singularity avoids the shared roots by using a separate type hierarchy for each Software Isolated Process (SIP). Non shared types used in a Singularity SIP are completely isolated from the types of other SIPs. Even when the same code is loaded into multiple SIPs, the code is separately compiled, reachability roots are separate and even the data layout is not guaranteed to be the same.

Shared types in Singularity are shared between SIPs. These types, however, only specify instance data layout. As discussed, they do not specify behaviour and do not have static members. Thus, shared types do not include any reachability roots.

Rust takes a different approach for avoiding shared reachability roots. The Rust language does not have any static or class members. The reachability roots of a task are the values included in the closure used for initialising the task. Because only owned closures can be used for task initialisation and as the ownership of these closures is transferred to the new task, these values are guaranteed not to be shared between tasks.

Controlling the reachability roots is required for ensuring the reference-protection model. However, as evident from systems supporting initial isolation, it is not always sufficient. Further strategies for controlling the propagation of references are needed to ensure that the reference protection model is maintained throughout the lifetime of the system.

2.6.2 Controlling Reference Propagation

The various strategies for controlling the reachability roots ensure that components do not breach the reference-protection model when initialised. Control of reference propagation ensures that the running code maintains the protection model. Several methods for controlling reference propagation have been suggested. The specific method used is closely tied to the reference-protection model. These methods are further discussed below.

Complete Isolation

Control of propagation is implicit in systems providing complete isolation. In these systems, the reachability sets of the components are disjoint and the system does not provide mechanisms for transferring references between components.

Without mechanisms that transfer references, components can only access references that are reachable from the reachability roots. With disjoint sets of reachability roots, no cross-component references can be reached and isolation is maintained.

When the protection model supports sharing, implicit control of propagation no longer works. The system needs to guarantee that the code running in the system is safe and does not store references in contravention of the reference-protection model.

Object Sharing

In systems supporting shared zones, a running application code may have access to values in its isolated zone and to values in a shared zone. The code, in this case, may create a reference from the shared zone to the isolated zone. The model prohibits external references to the isolated zone. The system, therefore, must ensure that the code does not create them.

Preventing such references at run time is achieved by adding a write barrier for each reference-store operation. The write barrier verifies that the reference-store operation complies with the reference-protection model. Complying stores are allowed to proceed whereas stores that conflict with the protection model are not executed.

The performance overhead of adding a write barrier to each and every reference-store operation may be significant. Compile-time prevention of non-complying reference-store operations avoids this overhead. This method relies on only allowing certain object types in the shared zone. These shared types are restricted to only hold references to shared object types. The type safety of the language, then, guarantees that shared objects do not hold references to isolated objects.

Partial Isolation

Limiting reference stores is not sufficient for ensuring safety in systems supporting sealed zones. In these systems, components may have access to objects in the interface zone of other components. Consequently, the system needs to prevent components from following the references from interface objects to sealed objects.

Transfer of references to sealed objects can be prevented by interposing *transfer barriers* between components. The transfer barriers check each reference as it is moved across the boundary between components. The J-Kernel capability objects are an example of transfer barriers.

Tracking the reference in compile time is used in both ownership types and confined types to prevent external references to sealed zones. These systems distinguish between types of objects in the sealed zones, or *sealed types*, and types of objects that can be shared. References of sealed types are only allowed in code that has access to the sealed zone, preventing the transfer of these references outside of the corresponding interface zone.

Reference Tracking Time

As the discussion above demonstrates, the techniques for controlling reference protection can also be classified based on whether it enforces the protection model in run time or at compile time. Table 2.1 summarises the methods used for propagation control.

Model	**Run Time**	**Compile Time**
Complete Isolation	Implicit	
Object Sharing	Write Barriers	Shared Types
Partial Isolation	Transfer Barriers	Sealed Types

Table 2.1: Methods for controlling reference propagation

2.7 Evaluation of the Classification

Preceding sections describe the classification of systems providing reference protection introduced by this thesis; survey prior research and demonstrate how it fits within the classification; and demonstrate that the techniques a system uses to achieve reference protection are closely tied to the classification of the system. This section evaluates the effectiveness of the classification.

The first measure of the evaluation is whether it captures all of the zone types used in systems supporting reference protection. Table 2.2 presents the classification of zones that are used by the systems reviewed in this chapter. Systems that provide

initial isolation do not control the propagation of references and do not manage zones. As such, these systems are not included in the table.

System	**Zone**	**Zone Type**
CoLoRs	Process heaps Shared region	Isolated Shared
Confined Types	Objects of confined types Objects in the package	Sealed Interface
J-Kernel	Domain Capabilities Domain objects	Interface Sealed
JNode	Root Isolate Application Isolates	Privileged Shared Isolated
JX	Domains	Isolated
KaffeOS	Kernel Heap Shared Heaps Process Heaps	Privileged Shared Shared Isolated
MVM	Isolates	Isolated
.Net	Application Domain	Isolated
OVM	User Domains Executive Domain	Isolated Privileged Isolated
Ownership Types	Owning objects Owned objects	Interface Sealed
Real-time Java	Regions	Shared
Rust	Processes Shared Values	Isolated Shared
Singularity	Process Exchange heap	Isolated Shared
XMem	Process heaps Shared region	Isolated Shared

Table 2.2: Zones in systems providing reference protection

Table 2.2 identifies the system, the system's name for each zone and the type of the zone. For example, each domain in JX has its own heap and its own type hierarchy. Types in one domain are not assignment compatible with types in another. Consequently there are no references between domain heaps and the domains are isolated.

This table demonstrates the effectiveness of the classification for outlining isolation boundaries. Identifying the zone types in a system concisely identifies the allowed references between these zones and, with them, the level of isolation between the components built from these zones. This concise description allows comparing the level of isolation provided by different systems. For example, both JNode and OVM provide a similar level of isolation, where user tasks (called application isolates in JNode and user domains in OVM) cannot access the objects in each other, whereas the system component can access objects in every task.

As can be expected, the conciseness of the classification comes at a cost of losing some details. For example, the classification fails to capture the hierarchical structure of ownership types. It also ignores the temporal nature of ownership transfer in Rust and Singularity.

Table 2.2 also reveals that there are only a few combination of zone types used in the same system. Ignoring privileged zones, which are used for system components, there are only three such combinations in use. These combinations define the reference-protection model provided by the system. Systems that only use isolated zones provide complete isolation. Systems that use isolated and shared zones provide the object sharing model. Systems that use sealed and interface zones provide the partial isolation model.

Table 2.3 summarises the reference-protection models and implementation methods used by each of the systems. As before, systems that use initial isolation are not included in the table. The table illustrates that the classification applies to every system surveyed. It also demonstrates the relationship between the techniques used for reference-propagation control and the reference-protection model provided by a system. That is, implicit propagation control is used with complete isolation, shared types and write barriers with object sharing and sealed types and transfer barriers with partial isolation.

System	**Run Time Model**	**Reachability Control**	**Propagation Control**
CoLoRs	Object Sharing	Multiple Types	Write Barriers
Confined Types	Partial Isolation	Shared	Sealed Types
J-Kernel	Partial Isolation	Multiple Types	Transfer Barriers
JNode	Complete Isolation	Indirection	Implicit
JX	Complete Isolation	Multiple Types	Implicit
KaffeOS	Object Sharing	Shared Multiple Types	Write Barriers
MVM	Complete Isolation	Indirection	Implicit
.Net	Complete Isolation	Indirection Separate Compilation	Implicit
OVM	Complete Isolation	Multiple Types	Implicit
Ownership Types	Partial Isolation	Shared	Sealed Types
Real-time Java	Object Sharing	Shared	Write Barriers
Rust	Object Sharing	Ownership	Shared Types
Singularity	Object Sharing	Multiple Types	Shared Types
XMem	Object Sharing	Separate Compilation	Write Barriers

Table 2.3: Classification of reference protection

Switching the focus from the systems to the reference-protection model and its implementation results in Table 2.4. The table divides the systems based on the technique used for controlling propagation. As demonstrated in Table 2.1, the technique

depends on the reference-protection model and the time of enforcing the control of reference propagation. Hence, Table 2.4 replicates the structure of Table 2.1, but replaces the techniques used with the systems that employ them.

<table>
<tr><th></th><th>Run Time</th><th>Compile Time</th></tr>
<tr><td>Complete Isolation</td><td colspan="2">JNode
JX
MVM
.Net
OVM</td></tr>
<tr><td>Object Sharing</td><td>CoLoRs
KaffeOS
XMem
Real-time Java</td><td>Rust
Singularity</td></tr>
<tr><td>Partial Isolation</td><td>J-Kernel</td><td>Confined Types
Ownership Types</td></tr>
</table>

Table 2.4: Classification of reference protection

It is interesting to note that the only systems that provide the partial isolation model and enforce it in compile time use shared reachability roots, severely limiting their use for component isolation. This is a surprising result because, of all the reference-protection models described, the partial isolation model promises the most flexible communication options and its enforcement at compile time promises minimal implementation overhead.

The next chapter continues the investigation of the partial isolation model. It presents Exported Types—a design of a type system that provides the partial isolation protection model. As the model is supported in the type system, it avoids the performance overhead of run-time checks. The design uses the multiple types approach to avoid sharing the reachability roots and provides a compile-time control of reference propagation, avoiding the costs of transfer barriers.

Chapter 3

Exported Types

The previous chapter identifies a gap in the prior work on reference protection. The classification of systems providing reference protection that Chapter 2 presents demonstrates the trade-offs the systems make between the level of isolation and the flexibility of the inter-component communication options they provide. As noted, the partial isolation reference-protection model provides the most flexible communication options of all the models that achieve the level of protection expected for components. Nevertheless, the only existing implementation of the partial isolation model for component isolation does not achieve efficient inter-component communication because it uses costly transfer barriers that add a significant run-time overhead.

This chapter bridges the gap in the prior work. It presents *Exported Types*, a type system design that merges the protection model with the type system, using the types of references to identify the zone the referenced objects are in. Using the reference type information available to the compiler, the compiler can enforce the protection model during compile time. Enforcing the model at compile time avoids expensive run-time tests and supports an efficient implementation.

The Exported Types design relies on name-based type equivalence [Albano et al., 1989, Connor et al., 1990]. It separates components by associating each component with a name space and resolving type names for the component within that name space. These name spaces are the main mechanism behind the implementation of the sealed zones. Type names of one component do not resolve to types in another component by default and types in different components are not assignment compatible.

In addition to the component name spaces, Exported Types supports a shared name space and two operations which are used to create the interface zones. The *export* operation registers a type name in the shared name space and associates it with a type. The *import* operation binds a type name in the importing component's name space with the type associated with that name in the shared name space.

Importing an exported type provides access to objects of the type. Thus, exporting a type places it and all it subtypes in the interface zone of the exporting component. The programmer may wish to give the exporting component privileged access to objects of the exported type. To provide the programmer with some control over access to objects of exported types, the Exported Types design adds an explicit remote protection scope. Inter-component access to members of an exported type is only allowed if those members are explicitly marked as being in the remote scope.

Exported Types specifies those features that are required for implementing partial isolation. Other features, such as the nature of subtyping or type-protection mechanisms, are to be completed by the language designer. Thus, Exported Types can be used as a basis for a type system for a new language. Alternatively, it can be applied as an extension to the type system of an existing language, assuming the type system of the language supports the features required for implementing Exported Types. It is backwards compatible in the sense that code that does not use the export or import operations executes within its own component as if it was executing in an unmodified language runtime.

The rest of this chapter describes the Exported Types design. The description provided is, mostly, informal, describing the type system design by way of examples. For a more rigorous description, see Appendix A.

This chapter consists of three main parts. The first two sections provide preliminary background on types and type systems and describe the relationship between type systems and partial isolation and how it affects the design of Exported Types. The following three sections describe the design in detail, including the components name spaces, the import and export operations and the remote interface. The last part of this chapter describes language specific issues with examples from the Java language.

3.1 Type Systems

Data values in computer systems are represented as sequences of bits. The type of the value specifies how the value is represented as well as restricts the operations on the sequence of bits to only those operations that conform with the type. For example, a type `int` in Java specifies that the value is a numeric integer, represented as a 32-bits sequence using the two's complement representation and that floating point operations are not to be applied to the value.

Languages provide a set of *primitive types*, which are types that exist *a priori*. These are used as building blocks to compose the complete set of types in the language. The *type algebra* is the subset of the language that allows the programmer

to create *composite types* by combining primitive and other composite types.

Composite types raise the question of type equivalence—how the equivalence of two type declarations is determined [Albano et al., 1989, Connor et al., 1990]. In *structural* type systems, types are considered equivalent if their definitions are considered equivalent. In *nominative* or *name-based* type systems, the equivalence of types depends on other characteristics of the types, such as the type name or the code location the type is defined in.

Languages sometime mix nominative and structural features in their type systems. For example, classes in Java use name-based equivalence. Different class declarations declare different classes even if the declarations specify the same structure. Java arrays, on the other hand, use structural equivalence. Array types are considered equivalent if their element types are equivalent, irrespective of the code location the array types are declared in.

Primitive types are, usually, *unboxed*. That is, the identity of the values of the type is intrinsic to the bit sequence. Values of unboxed types are considered identical if the bits in the values are the same, irrespective of where the values are stored.

Bit sequences representing *boxed types* are stored in "boxes" of memory. The identity of values of these types is the box in which they are stored and is unrelated to the bit sequence that represents the value. Values of boxed types are, therefore, a combination of a memory location and the bit sequence stored in it. This thesis uses the term *object* for values of boxed types.

Having a reference to a value implicitly identifies it by its memory location. Hence, references can only be created to boxed values.

During compilation, semantics and access decisions do not depend on the values. They only depend on the types of the values. Thus, the compiler uses types to represent the potential set of values that can be used in their place. Types, therefore, can be considered to be sets of values, where the members of a type are all the values that have that type.

A *type hierarchy* is a partial order of types using the *subtype* relation [Cardelli and Wegner, 1985, Liskov, 1987]. The subtype relation has the *substitution property* which, essentially, means that values of a subtype can always be used where the program expects values of the supertype [Liskov, 1987]. Consequently, the set of values of a subtype is a subset of the set of values of a supertype. Similarly, a value of a subtype is also a value of all of the subtype's supertypes.

When the program expects a value of a supertype, the program uses the representation of the supertype for the value. For the program to correctly use values of subtypes when it expects values of a supertype, the representation of the supertype must be applicable to bit sequences that represent values of the subtypes. Hence, the representations of subtypes must be extensions of the representations of their

supertypes.

The substitution property also limits the restrictions that the compiler can impose on access to values of subtypes. As values of the subtype can be used wherever values of the supertype are expected, any operation allowed on values of the supertype is also implicitly allowed on values of the subtype. The compiler, therefore, cannot impose restrictions on access to values of the subtype unless the same or stronger restrictions are imposed on access to values of the supertype. The Java language provides an example of this limitation. When a Java class declares a `public` method, subclasses of the class cannot narrow the scope of the method, e.g. by overriding it with a `private` method.

Types and zones are both collections of objects. Using types to encode the zone that objects are in allows the compiler to apply the access rules of reference protection.

3.2 Partial Isolation and Types

This section explores the relationship between types and zones and highlights the underlying properties of type systems that can support partial isolation.

Partial isolation is a reference-protection model that protects the private data of components yet supports inter-component access to interface objects. Under the model a component consists of two zones. The sealed zone of a component encompasses the private data whereas the interface zone contains the component's interface. Figure 3.1 (reproduced from Figure 2.3) embodies the main concepts of the partial isolation model. It demonstrates three components and shows values or objects within the zones and possible references between these objects. The important property of partial isolation is that direct references from one component to the sealed zone of another are prohibited. For example, Object $O1$ cannot hold a direct reference to Object $O3$.

To enforce the model at compile time, the compiler needs to make access decisions regarding permitted references. Compilers make decisions on access to objects based on the objects' types. Hence, to enforce reference protection at compile time, the zone structure should be reflected in the type system. That is, knowing the types of values should be sufficient to determine whether references between them are allowed.

As discussed above, the compiler uses types to represent sets of values. The compiler does not know which specific values of the set will be used at run time and must, therefore, be conservative when permitting access. If a certain access is not permitted to even one of the values of the type, the compiler should either disallow that access to any value of the type or defer the decision to run time. The

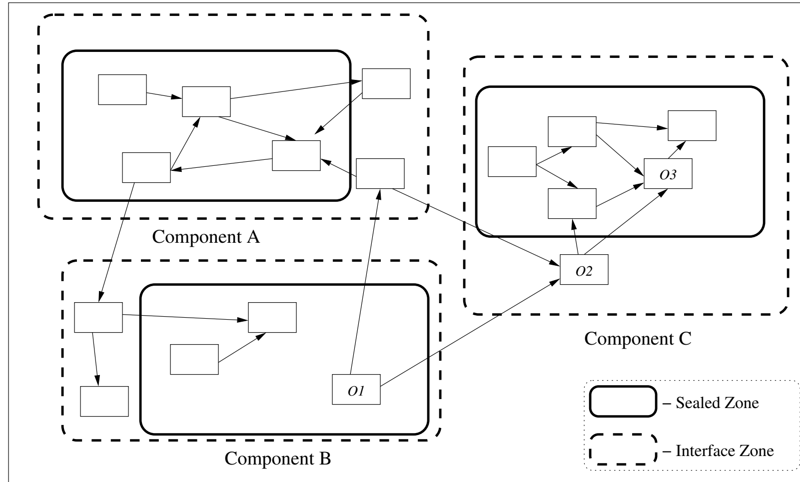

Figure 3.1: A component system with partial isolation

compiler can only permit an access without run-time tests if the access is allowed to all potential values of the type.

In the context of compiler-enforced partial isolation, the compiler makes the decisions about which references are allowed to objects. If a type can include an object in the sealed zone of a component, the compiler must treat objects of the type as potentially being in that sealed zone and prevent any references from outside the component to objects of the type. Also, due to the substitution property, all supertypes of the type are considered to be in the sealed zone.

The association of types that contain objects in the sealed zones with the sealed zone means that types cannot span the sealed zones of multiple components. That is, if an object of a type is in the sealed zone of a component, the type is associated with that sealed zone and no object of the type can be in the sealed zone of another component.

Exported Types builds upon this association of types to zones to overlay the type hierarchy with zone information. In Exported Types, each type is associated with a zone in the component that declares the type. By default, types are associated with the sealed zone of the component and are considered private to the component. Types declared as exported are associated with the interface zone of the component. Diagram 3.2 shows the relationship between types and zones in Exported Types.

It is worth noting that all the supertypes of types in a sealed zone are in the same zone. For example, T_{B1} and T_{B2}, the supertypes of type T_{B3}, are both in the sealed zone of component B. This is a consequence of the substitution property. The set of values represented by a subtype is a subset of the set of values represented by a supertype. If the subtype may include values in the sealed zone, the supertype may include them as well. Hence, the supertype must be in the sealed zone.

The converse does not hold. Subtypes of sealed types may be in the interface

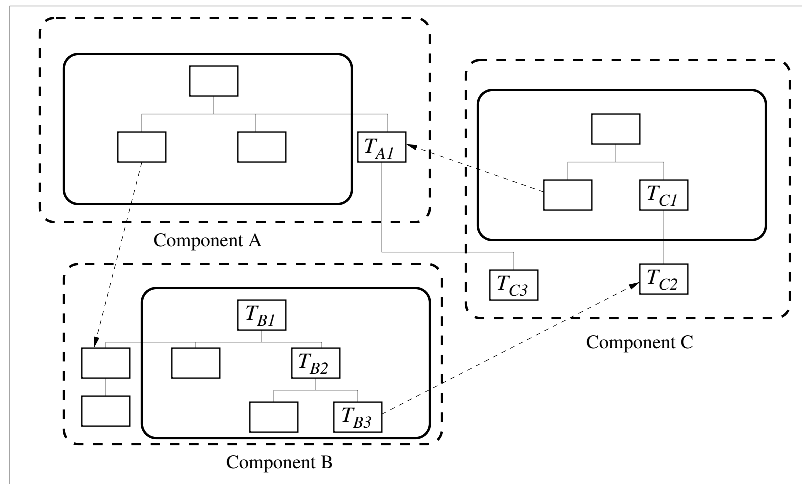

Figure 3.2: Overlaying the sealed zones model over the type hierarchy

zone. For example, type T_{C2} in the interface zone of Component C is a subtype of T_{C1} which is in the sealed zone of the same component. The implication is that type T_{C1}, while being associated with the sealed zone of Component C, may include objects in the interface zone of the component.

Diagram 3.2 also displays some of the inter-component references permitted between object types. All objects of types associated with the interface zone of a component are in the interface zone. Hence, inter-component references are permitted when the referent type is associated with an interface zone of a component. To prevent inter-component references to objects in sealed zones, Exported Types prevent component code from being able to refer to types associated with the sealed zones of other components.

Exporting a type allows access not only to the objects, but also to the type definition itself. Components may extend types exported by other components, creating subtypes of these exported types. For example, type T_{C3} in Component C is a subtype of type T_{A1} of Component A. As a result, objects of type T_{C3} are in the interface zones of both components A and C. Objects in the interface zones of two components can hold references to objects in the sealed zones of both components. Code of such objects is associated with only one of the components and cannot, therefore, access objects in the sealed zone of the other.

Associating types with the component that declares them implies that the type system is not purely structural. Not being purely structural is an inherent property of type systems that support partial isolation. This is because different components often have objects that have identical structures in their private data. With structural type systems these objects would be considered of the same type. As discussed above, objects in the sealed zones of different components, i.e. the private data of the components cannot have the same type. Hence, even though the objects have

the same structure, they are of different types. Thus, type equivalence is not defined solely by the structure of the type.

A nominative type system is required for the Exported Types design. The design uses the concept of type name spaces to associate types with components. The use of name spaces in Exported Types is discussed in the next section.

3.3 Component Name Spaces

The key to component isolation with the Exported Types design is creating a separate name space for each component. Type names for the component are resolved within the name space, creating a separate type hierarchy for each component. Figure 3.3 shows the type hierarchies of two components.

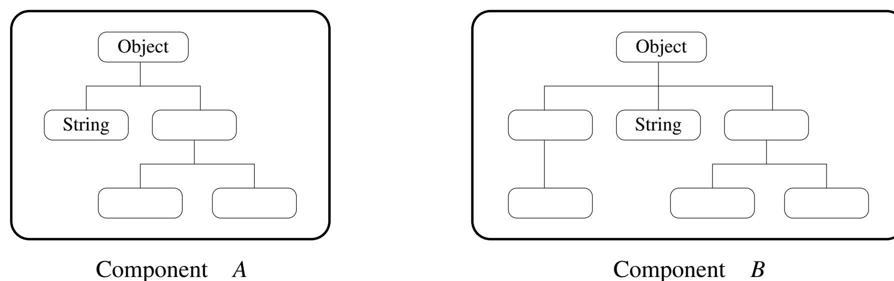

Figure 3.3: Sealed zones

The hierarchies of the components are completely separated. Types in one hierarchy are not assignment compatible with types in another. For example, in Figure 3.3, each name space includes the `Object` and `String` types. A `String` object from Component A cannot be assigned to a `String` reference in component B and vice versa. Likewise, no object from Component A can be assigned to an `Object` reference in component B. As inter-component references are prohibited, separating the name spaces achieves complete isolation.

It should be noted that the name space separation only applies to boxed types. Values of unboxed types are not shared, hence these types can be shared without risk of breaching the reference-protection model. Unboxed types provide a basic set of types shared between components, allowing communication between the components.

When applying the Exported Types design to extend the type system of an existing language, the use of a separate name space for each component results in each component having a type hierarchy that mirrors the original type hierarchy of the language. Consequently, code that does not use the export and import operations of the Exported Types design is enclosed within a component with no incoming or outgoing references. The absence of inter-component references means that the

code cannot directly access the memory space of other components, nor can other components access the memory space of the code.

The concept of separate name spaces is not new. OVM [Armbruster et al., 2007, Spring et al., 2007], .Net [ECMA, 2006] and JX [Golm et al., 2001, 2002, Wawersich et al., 2002] use separate name spaces to achieve complete isolation between components.

The Exported Types design uses the name spaces to create components. The next section describes the mechanisms used for splitting components into sealed and interface zones.

3.4 Creating the Interface Zones

The main contribution of the Exported Types design is the ability to selectively add types to the interface zone of a component while preserving the substitution property. Type export and import are realised through the use of a shared name space. When a type is exported, the type name is registered in this name space. Components wishing to use the type can, then, locate it in the shared name space and gain access to the type.

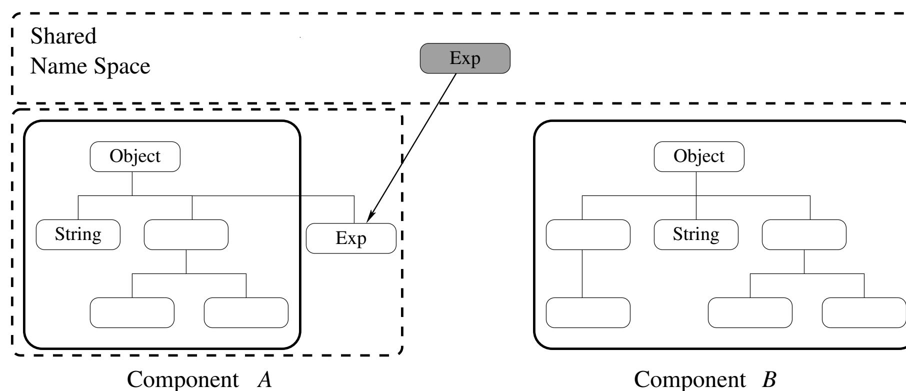

Figure 3.4: Type hierarchies with exported types

Figure 3.4 shows the class hierarchies and the name spaces after component A exports the class `Exp`. As the diagram demonstrates, the `Exp` type is in the interface zone of component A. The name of the type is registered in the shared name space, where it is associated with the type. (Shaded boxes in the diagrams in this chapter indicate names that are bound to types declared in other name spaces.)

Using a shared name space raises the issue of conflicts when multiple components export types with the same name. The Exported Types design leaves the decision on how to address these to the language or system designer. For systems where the likelihood of conflicts is low, a simple first-comes-first-served approach may be suitable. An example of such a scenario is an application, such as a Web browser,

that publishes an interface for third-party extensions or plug-ins. As the application is the only component to publish an interface there is no risk of conflicts.

Other scenarios may present a greater likelihood of conflicts. An example of such a scenario is a fully-fledged multi-tasking virtual machine where tasks publish interfaces for services they provide. Such systems may require a more elaborate approach, such as hierarchical naming schemes or a naming service.

To use an exported type, components must explicitly import it. Explicit imports are required to ensure that the programmer intends to import types. Forcing explicit imports ensures that only code that needs to access remote components has that access. This is particularly important when Exported Types is used to extend an existing language, as it guarantees that the semantics of legacy code do not change by inadvertently importing types.

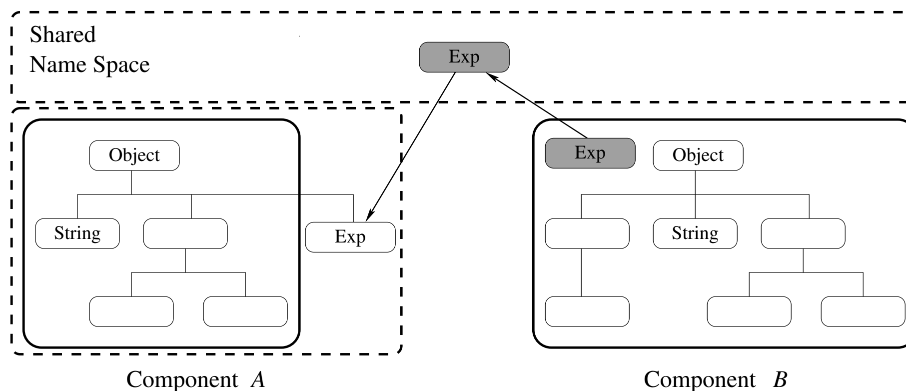

Figure 3.5: Type hierarchies after import

Figure 3.5 shows the type hierarchies described above after component B imports the type `Exp`. As the diagram demonstrates, the type name `Exp` in component B is bound, through the shared name space to the type `Exp` in component A. References of type `Exp` in component B can, therefore, point to objects of the type, which are part of the interface zone of component A.

Importing types gives the importing component access to the type declaration. The importing component can, then, subtype and extend the imported type. One use of extending an imported type is for implementing callbacks, e.g. for use in mutual recursion. Figure 3.6 demonstrates a possible way of implementing callbacks. In this scenario, Component A provides a service and publishes an interface for this service that Component B uses. As part of that interface, Component A occasionally needs to invoke callbacks in Component B.

The approach for implementing the callbacks is for the service provider (Component A) to declare and export an abstract interface type (`AbstractCallback`). The abstract interface type specifies the signature of the possible callback methods that the service provider may invoke. The client (Component B) imports and ex-

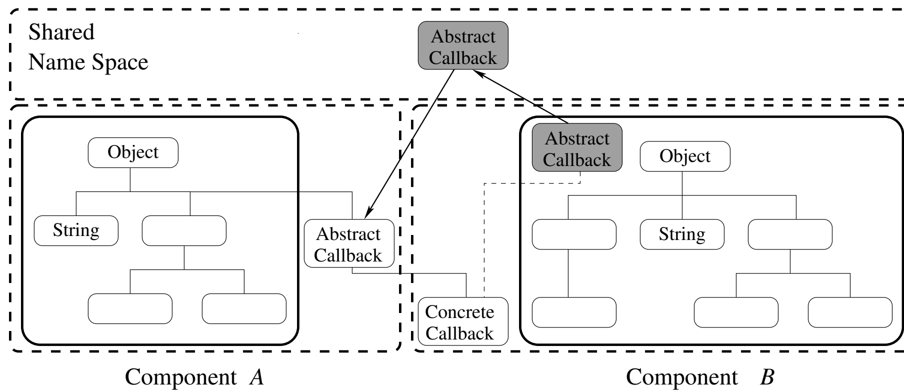

Figure 3.6: Extending an imported type

tends the abstract interface by providing a concrete implementation of the callback methods it responds to (`ConcreteCallback`). Objects of `ConcreteCallback` are in the interface zone of Component B and can, therefore, access its private data. By the substitution principle, these objects are also of type `AbstractCallback`, which is the same type in both Component A and Component B. Thus, Component B can pass an object of `ConcreteCallback` as a method argument to Component A, which can, then, use it for invoking callbacks to Component B.

Objects of `ConcreteCallback` are in the interface zones of the two components. However, the code of these objects is not shared. Each method of the object is associated with the component that provides its implementation and cannot access references to the sealed zone of another component. For example, methods implemented as part of the declaration of the `AbstractCallback` can access objects in the sealed zone of Component A but not those of Component B.

Explicitly marking the members of exported types that can be accessed by code associated with other components provides the mechanism that prevents the propagation of references to objects in the sealed zone between components.

3.5 The Remote Interface

The *remote interface* of an exported type is the set of members of the type that can be used by code external to the component. As the remote interface allows transfer of references between components, it plays a crucial role in maintaining reference protection. References can cross component boundaries by assigning them to or reading them from fields of exported types. They can also cross component boundaries when used as arguments, return values or exceptions of methods of exported types.

The concept of a remote interface is not new. Two approaches have been used to declare a remote interface. The first, which is used in the Luna operating sys-

tem [Hawblitzel, 2000, Hawblitzel and von Eicken, 2002], is to add a `remote` protection scope wider than the `public` scope. Type members annotated with `remote` can be accessed across component boundaries. Other type members are only accessible from within the component.

Another approach for declaring the remote interface is used by the J-Kernel [Hawblitzel et al., 1998] and the Java RMI [Sun Microsystems, Inc., Waldo, 1998]. In this approach, interface types (i.e. Java `interfaces`) are used for the remote interface. These systems use a markup `Remote` interface. The `Remote` interface does not have any members, however, all interfaces that extend it are remotely accessible. Consequently, classes that implement any such interface are remotely accessible and are in the interface zone of the implementing component. The remote interface of these classes is the methods of the implemented interface.

Communication between components uses the remote interface. Hence, by ensuring that references to objects in sealed zones cannot be used on the remote interface, Exported Types can guarantee that references to these objects do not leave the scope of the component they belong to. To maintain this property, Exported Types requires that only unboxed, exported or imported types are used in the signatures of members that are part of the remote interface.

Unboxed types are safe to use in the remote interface because their values are not objects and do not have references pointing to them. Exported and imported types are, by definition, associated with the interface zone of the component that exports them. As discussed in Section 3.2, all the objects in these types are in the interface zone of the component. Consequently, passing references to these objects between components cannot breach the reference protection.

It should be noted that the above restrictions on signatures are sufficient to implicitly define the remote interface. That is, the remote interface could be defined as the collection of all members of types associated with interface zones that only include unboxed, imported and exported types in their signatures. Exported Types, however, requires that the remote interface is explicitly declared. The rationale for this requirement is that implicit declaration of the remote interface does not, necessarily, agree with the programmer's intentions.

When the remote interface is implicitly declared, the programmer does not have a direct control over which members are included in the interface. For example, the programmer cannot prevent external components from accessing methods whose arguments and return values are of unboxed types. When components do not trust one another, such methods need to be secured against potentially malicious access. Explicit declaration of the remote interface ensures that only those members that the programmer intends to expose are indeed exposed and protects against inadvertent exposure of members e.g. through exporting a type that was hitherto sealed.

Explicit declaration of the remote interface also enables early verification that types used on the remote interface are safe for that use. The explicit declaration is a positive indication of the programmer's intent to allow inter-component use of the member. This indication allows the compiler to verify that the member will, indeed, be accessible at run time. If, on the other hand, the system relies on type information to decide which members are in the remote interface, the fact that a member is not in the interface will only be realised when the member is used, rather than declared.

The discussion so far presents the features and properties of Exported Types design. Some language features, however, are not compatible with the design. These are the topic of the next section.

3.6 Compatibility with Language Features

Ideally, it would be possible to decide for each type whether it is exported or not and, consequently, whether to allow its use on the remote interface. However, particularly when extending an existing language, language features may be incompatible with the design and may result in the semantics of exported types differing from those of other types.

This section identifies five language features that cause difficulties with the Exported Types design: structure-based type equivalence, implicit operations on types, special semantics of types, universal methods and reflection. It describes the difficulties and suggests some approaches for handling them. The examples of these issues are taken from features of the Java language. Similar problems are found in other programming languages, such as Simula, C# or Scala.

3.6.1 Structure-based Type Equivalence

To avoid the need to explicitly declare every type, languages do not always use pure name-based type equivalence. They often include some features of structure-based equivalence. Arrays in Java, Simula and C# are examples of types that use structure-based equivalence. Array types in these languages are not explicitly declared and are not named. They are considered equivalent if their element types are equivalent. As exporting relies on type names identifying types, structure-based types cannot be exported.

A possible solution is to use named array types, similar to array types in ADA [American National Standards Institute, Inc., 1983]. ADA uses name-based equivalence for all types, including array types. When declaring a named array type, the programmer associates a name with the type. The name of the array type is used for

variable declarations and arrays are considered of the same type only if they are declared using the same array type name.

Such a solution may be useful when designing a new language. Adding named array types to an existing language may result in significant changes to the language semantics and to its implementation. A possible workaround to structure-based equivalence is to use a wrapper class that hides the structure-based type. The structure-based type is never exported. Hence objects of this type are always in the sealed zone of a component. The wrapper class type is exported to provide remote components with indirect access to objects of the structure-based type it wraps. An example of a wrapper class is provided in the next section.

3.6.2 Implicit Operations on Types

Arrays in Java also highlight the issue of implicit operations. The operations of getting and setting elements of the array are implicitly defined. Consequently, operations on arrays are supported in all contexts in Java and there is no way to limit their use, e.g. to create immutable arrays.

Another example of implicitly defined operation in Java is the use of objects for synchronisation. Every object in Java has an intrinsic lock that can be used for synchronisation. The Java bytecode supports two operations on the objects locks: `monitorenter`, which locks the lock, and `monitorexit`, which releases it. In the Java language, both these operations are implemented using the `synchronized` keyword, which is used to tag critical sections.

Because these operations are implicit, the language does not include the syntax to declare them. Consequently, when exporting a type, there is no easy method of deciding whether an implicit operation is included in the remote interface of the type.

One possible solution to this problem is to use a special syntax for these operations. For example, implementations of Exported Types for Java can use Java annotations to decide whether objects of an exported type can be locked by a remote component.

An alternative solution which is applicable to Java arrays, is to provide indirect access to the array via an exported wrapper class [Gamma et al., 1993]. This wrapper class can, then, filter undesired access to the wrapped array. Listing 3.1 shows the code for a wrapper class that wraps an `int` array, providing external components with read-only access to the array. The class `IntArrayWrapper` in the example has two remote methods: `getLength` and `get`. These methods allow external components to get the length of the array and to get the value of an array entry. As no other methods or members are in the remote interface, no other access to the array

is provided.

```
@Export
public class IntArrayWrapper {
  final int[] intArray;

  public IntArrayWrapper(int[] intArray) {
    this.intArray = intArray;
  }

  @Remote
  public int getLength() { return intArray.length; }

  @Remote
  public int get(int i) { return intArray[i]; }
}
```

Listing 3.1: Int-array wrapper

3.6.3 Universal Methods

Closely related to implicit operations are universal methods. These are methods that can be applied to any object. In the Java language these are the methods of the `Object` class. E.g. `hashCode`, which computes the hash code of the object, or `equals`, which compares two objects for equality.

The signature of many universal methods includes arguments or return values of types that cannot be exported. For example, the `equals` method accepts an `Object` argument. `Object` is the supertype of all other reference types, hence exporting `Object` would implicitly export every class.

These Universal methods cannot be in the remote interface of exported types. Consequently, imported types cannot be subtypes of the importing component's `Object` type.

Language features, such as Java generics [Bracha, 2004], and library code, such as the Java Collection framework, rely on either universal methods or on upcasting to `Object`, and often on both. These features cannot be used with imported types.

As with implicit operations, wrapper classes can be used to overcome the lack of universal methods. Listing 3.2 demonstrates a wrapper that can be used to wrap objects of an imported type `Exp`. As the wrapper class is a subtype of the importing component's `Object` type, these wrappers can be used with the Java Collection framework. It is assumed that `Exp` declares the methods `hashCode()` and `equalsTo(Exp other)` and places them in the remote interface.

```
public class ExpWrapper {
  final Exp theExp;

  public ExpWrapper(Exp exp) {
    this.theExp = exp;
  }

  public int hashCode() { return theExp.hashCode; }

  public boolean equals(Object other) {
    if (!other instanceof ExpWrapper)
      return false;
    return theExp.equalsTo(((ExpWrapper)other).theExp);
  }
}
```

Listing 3.2: A wrapper for an imported type

3.6.4 Special Semantics of Types

Some types have special meaning in the language. An example from Java is exception handling where objects of type `Throwable` are used to signal exceptional termination of program code.

As a `Throwable` object from one component is not of type `Throwable` in any other component, exceptions created in one component are not considered exceptions outside that component, hence exceptions thrown in one component cannot be delivered to code in another component. This incompatibility does not have a straightforward solution.

One possible approach is to share the exception types defined by the system between the components. This approach would allow catching an exception thrown by another component. However, methods of the class `Throwable` include arguments and return values that are not exported. Thus, any implementation that would allow one component to catch exceptions thrown by another would have to take care of these methods to ensure they do not disclose sealed values.

Another approach for handling the cross-component exceptions issue is to convert exceptions that are thrown across component boundaries to the corresponding exception type in the catching component. Exceptions can be converted by serialising them at the throwing component and deserialising it at the catching component. As cross-component exceptions are not common, the overhead of the serialisation process is not expected to have a significant impact on the performance of the system. Furthermore, the overhead can be reduced using the *fast copy* technique suggested by Hawblitzel et al. [1998], that makes direct copies of the data in objects without first serialising them.

3.6.5 Reflection

Reflection is a language feature that allows programs to inspect and affect their own calculation [Smith, 1982]. More specifically, reflection allows programs to query the type information of values at run time and to operate on values based on the queried type information rather than based on statistically assigned type tags. For example, a program can use reflection to test whether an object supports a certain method and to invoke the method if supported.

Values exported from a component can hold references to values private to the exporting component. Unchecked use of reflection on values of exported types can, therefore, leak these references to the importing component, resulting in a breach of the reference protection.

To protect against this breach of reference protection, reflection can be limited to only operate on values that are guaranteed not to hold references to private values of other components. One way of achieving this is to allow reflection on values if all the types the values have are declared by the accessing component. This would allow reflection to operate on all values in the sealed zone of the component. It would also allow reflection to operate on values in the interface zone of the component if these values are not also in the interface zone of other components.

3.7 Summary

The Exported Types design enforces reference protection with partial isolation. It is useful both as a basis for a new type system design and as an extension to an existing language.

The design overlays the partial isolation model over the type hierarchy to create compile-time-enforceable reference protection. One of the consequences of this overlay is that all the supertypes of types in the sealed zone of a component are also in the same sealed zone.

The use of explicit operations in the Exported Types design ensures backwards compatibility when the design is used to extend an existing language. Legacy code does not use any of the Exported Types operations and can continue executing within a component without any change to its semantics.

Exported Types separates components by associating each component with a type name space. Name-based type equivalence is, therefore, inherent in the design. The main novelty of the Exported Types design is its support for exporting and importing types, which lets components access types declared in other components.

The design places significant importance on explicit operations. Types are explicitly imported and exported, ensuring that external access is not inadvertently

granted. Also, to avoid inadvertent exposure of members, the remote interface of types is explicitly declared.

Some language features conflict with the Exported Types design. Many of these can be addressed by using wrapper objects, however in many other cases there is no generic solution to these issues.

The next chapter presents a concrete implementation of the Exported Types design.

Chapter 4

An Implementation of Exported Types

This chapter continues the exploration of the partial isolation reference-protection model. The previous chapter presents Exported Types, a type system design that enforces partial isolation in compile time. This chapter describes a proof-of-implementation of the design, demonstrating that it can be implemented and that it can be applied to an existing language. The next chapter analyses the performance of the implementation, demonstrating that the increased security the isolation provides does not incur a run-time performance cost.

This chapter describes the implementation of S-RVM [Yarom et al., 2012], a Java virtual machine based on JikesRVM [Alpern et al., 2000] that implements Exported Types. JikesRVM is a research-oriented Java virtual machine notable for being meta-circular. Meta-circular virtual machines are virtual machines that execute within the same environment they provide. In the case of JikesRVM it means that it is written in Java and that it runs itself. S-RVM increases the security of JikesRVM by providing a clear boundary between the application code and the virtual machine code, using reference protection to protect the virtual machine code from the application.

The virtual machine code is responsible for enforcing the type safety of the language. To maintain protection for virtual machine code, there is a need for a clear boundary between the trusted virtual machine and the untrusted application [Back and Hsieh, 1999]. That is, the virtual machine needs to be able to easily determine the context within which code executes.

In JikesRVM, the application and the virtual machine share the type hierarchy, including the Java library classes. Hence, the same library code can be executed by both the virtual machine and the application. Sharing the library code, therefore, blurs the boundaries between the virtual machine and the application. As a result, there is no one consistent solution for deciding whether code executes within the

virtual machine or the application context [Jones and Ryder, 2008, Lin et al., 2012].

Sharing the type hierarchy means that application code has access to virtual machine types. Access to virtual machine types allows application code to bypass the type safety of the language and the blurred boundaries in JikesRVM hinder protecting the virtual machine from application access. As the Java language security depends on type safety, direct access to virtual machine objects allows applications to bypass the Java security.

S-RVM increases the security of JikesRVM by using the Exported Types design to reinstate a clear boundary between the virtual machine and the application. It segregates the virtual machine and the application into separate components, called *tasks*, where each object and the code associated with it is unambiguously associated with one task. The inherent protection of the Exported Types design ensures that virtual machine types are not accessible to the application except where explicitly declared to be in the interface zone of the virtual machine task.

The virtual machine executes within the *root task*. The interface zone of the root task provides the application in the *application task* with the virtual machine services it requires for its execution. As the application has direct access to this zone, the services are provided at no performance overhead.

The interface between the virtual machine and the application also forms a trust boundary. That is, the virtual machine does not trust any code that runs within the application task, including the Java library code. Not trusting the application's library code reduces the virtual machine attack-surface area [Manadhata and Wing, 2011]. It also decouples the Java library from the virtual machine allowing the use of different implementations of the Java library for the virtual machine and the application. This facilitates porting versions of the Java library for application use, obviating the need for handling the circular dependencies between the virtual machine and the Java library it uses.

S-RVM uses Java annotations to implement the export and import operations of the Exported Types design. When the virtual machine loads a class annotated with the `@Export` annotation, it adds the name of the class to a shared name space, allowing the application task to import the class. When it loads a class annotated with the `@Import` annotation the virtual machine searches for the class in the shared name space, replacing the previously exported class for the loaded class.

S-RVM also uses annotations for specifying the remote interface of exported classes. Only methods of exported classes annotated with `@Remote` annotation are accessible from the application task.

Implementing the Exported Types design to separate the application from the virtual machine presents several challenges that are specific to this context. Principal among these is the asymmetric nature of access required. Some of the functions

of the virtual machine, e.g. garbage collection, require unrestricted access to application objects. Consequently, virtual machine code needs to be able to bypass the protection provided by the Exported Types design. Hence, the implementation needs to support privileged zones.

Another challenge is the need to maintain the Java library interface. Some library classes, e.g. `Thread` or `ClassLoader`, represent virtual machine entities. The straightforward implementation of these classes is as virtual machine types that are exported to the application. Such implementation, however, is not compatible with the Java language specifications both because these classes are not subclasses of the importing task `Object` class and because the specifications of these classes include types in the sealed zone of the application.

A third challenge stems from enforcing a trust boundary between the virtual machine and the application. In a typical Java virtual machine implementation the Java library is part of the trusted code base. Trusting the library means that it can perform operations that are not allowed for application code. For example, Java library code can modify the backing store of `String` objects and is trusted not to do that in a way that would break the Java language security. To reduce the amount of trusted code and to reduce the attack-surface area of the trusted code, S-RVM does not trust the Java library of the application task. Consequently some parts of the Java library need to be redesigned.

Basing S-RVM on an existing system, rather than building it from scratch, is a double-edged sword. The main advantage of this approach, and the motivator for choosing it, is that JikesRVM is a mature, well-established, high-performance, Java virtual machine. Consequently, it provides a solid basis to compare the performance effects of adding Exported Types. Furthermore, the use of a mainstream language allows using standard performance benchmark suites which include "real" workloads. These would not have been available for a new language designed from the ground up with support for Exported Types.

The downside of using JikesRVM is that neither the Java language nor JikesRVM itself were designed to support Exported Types. As discussed in Chapter 3, some language features are not compatible with the Exported Types design. Furthermore, JikesRVM was designed to support the Java type system. Exported Types changes the semantics of the type system, and this change affect diverse areas of the code of JikesRVM, including the type system, the optimising compiler, the interface to the Java libraries and the Java Native Interface.

With several areas of JikesRVM being affected, it is little surprise that the changes for supporting Exported Types are quite involved. The implementation of S-RVM includes modifications to over 70 JikesRVM classes and almost 60 new class files. Overall more than 25,000 source lines were added or modified, covering

about 3% of the JikesRVM code base.

It is worthwhile noting that most of the changes required in JikesRVM are, basically, the result of a single design change. JikesRVM is designed to support the Java type system where each type in the core library is uniquely identified. That is, there is a single `java.lang.Object` type, a single `java.lang.String` type, etc.

S-RVM adds to JikesRVM the support for multiple, disjoint, type name spaces. Consequently, there are multiple copies of the Java core libraries and the code can no longer assume that unique semantics of a Java core library type apply to only one type. For example string literals in a class file need to be created with the `String` type of the name space the class is loaded into and references can be assigned to variables of type `Object` only if the references types are in the same name space as the `Object` type.

Thus, the single change to the design of the type system percolates through the various parts of JikesRVM code and affects multiple components of JikesRVM. This chapter describes this change and the modifications it induces on the code of JikesRVM.

The next two sections provide a brief introduction to features of the Java language and of JikesRVM that are required for understanding the implementation of S-RVM. This introduction is followed by an overview of S-RVM and its main data structures. The implementation of the export and import operations is discussed next. The discussion, then, proceeds to the design of the interface layer between the application and the virtual machine, followed by the issue of asymmetric access required for the virtual machine and the implementation of cross-task exceptions. This chapter concludes with a discussion on S-RVM implementation verification.

4.1 The Java Virtual Machine

The introduction of the Java language in 1995 reignited the interest in high-level-language virtual machines. Java is an object-oriented, type-safe and secure language. From its inception, the language was designed to support mobile code and the security challenges related to it [Gosling and McGilton, 1996]. This section provides an introduction to relevant aspects of the Java virtual machine, with focus on the language's security features.

The Java front-end compiler `javac` compiles Java code into Java bytecode [Gosling, 1995]. The Java bytecode is an intermediate instruction format for an abstract stack-based machine. Java bytecode is packaged in Java class files which the Java virtual machine executes. Hence, strictly speaking, the Java virtual machine executes Java bytecode [Lindholm and Yellin, 1999].

The Java virtual machine emulates the abstract machine to execute bytecode.

For this emulation it either interprets bytecode instructions one at a time or, for improved performance, compiles the bytecode into machine code and lets the CPU execute the compiled code.

The Java virtual machine is designed to execute mobile code, that is, code that is delivered to the machine over the network. In many cases the origin of this mobile code is unknown and the code cannot be trusted. One of the main functionalities of the Java security architecture is to limit such code to a restricted environment called the *sandbox*, that prevents hostile code from harming the computer it executes on.

The Java security architecture has evolved over the years to an intricate framework that supports encryption, authentication and access control [Gong, 1997, Oaks, 2001]. As reference protection falls under the umbrella of access control, this section focuses on this area.

Access control in Java consists of three main components [Gong, 1997, Gong et al., 1997, Yellin, 1995]. The first is the *class loader* which is both a mechanism for introducing (loading) code into the virtual machine and a naming and a protection scope for the code it loads [Liang and Bracha, 1998]. The second component of the security architecture is the *security manager* which verifies that code has authorisation to execute sensitive operations [Gong, 1997]. The third component is the *bytecode verifier*, which ensures that bytecode is type-safe and that it only manipulates objects using operations sanctioned by their type [Gosling, 1995, Leroy, 2003]. These three components are described in more details below.

4.1.1 Class Loaders

A Java class loader is an object used to locate Java class files and to load them to the virtual machine. A Java virtual machine starts execution with at least two class loaders: the *base class loader* (a.k.a. the *bootstrap class loader*) and the *system class loader*. The base class loader is used for loading the classes of the Java libraries, whereas the system class loader is used for loading the main program code.

In addition to the system-provided class loaders, the program can instantiate user-defined class loaders [Liang and Bracha, 1998]. These class loaders are used, for example, to load classes from remote locations or from different storage back ends. User-defined class loaders can also manipulate the bytecode before it is loaded and even dynamically generate the bytecode for the classes.

Class loaders act both as name spaces and as naming scopes for Java. Each class loader is associated with a name space for the classes it loads. Thus, run-time types are identified by their name and the class loader which loaded them. This scheme prevents conflicts between components that load classes with the same name. When the components are loaded by different class loaders, the types they load are different

even if they have the same name.

As a naming scope, the class loader is responsible for resolving type names appearing in code it loads. The class loader may locate the class and load it. Alternatively, the class loader may delegate the resolution process to another class loader.

To prevent class loaders from overriding critical system classes, the virtual machine ensures that core library classes (classes with names starting with `java.*`) are always loaded by the base class loader. These classes include the root of the Java type hierarchy—`java.lang.Object`.

Having a single root type hampers reference protection. In a static hierarchical type system, supertypes cannot support operations that are prohibited by subtypes. Otherwise, casting to the supertype would allows bypassing the restrictions on the subtype. As there are no restrictions on assigning to `Object` references in Java, the type system in Java cannot control the propagation of references in compile time. All extensions of Java that enforce reference protection at compile time create multiple `Object` types, either explicitly or implicitly.

When the class loader loads a class, it also records the location the code was loaded from, information on digital signatures associated with the code and any default permissions for the code. This information is used by the security manager, described in the next section, for authorising access to resources.

4.1.2 Security Manager

The security manager is the main authorisation module in Java. It manages the security policy and authorises access to resources, including files, URLs, class loaders and others.

Code that requires access control must explicitly invoke the security manager to authorise access. When invoked, the default security manager traverses the run-time stack to establish a security context, which is used to authorise the access. To establish the security context, the security manager relies on the identity of the class loader that loaded the active code and any information that the class loader recorded for that code.

The run-time call to the security manager and the cost of the stack traversal result in an overhead of 5%-100% to the cost of operations that require protection [Herzog and Shahmehri, 2005]. Consequently, the security manager is not used for protecting fine-grained frequent operations. Furthermore, due to the costs involved, the security manager is often disabled.

Requiring an explicit call to the security manager implies that the security it provides is discretionary. Code that fails to invoke the security manager and code

that allows bypassing the call to the security manager is not secure. To prevent bypassing the call to the security manager, programmers rely on the type safety of the language. The bytecode verifier, discussed in the next section, is the principal tool for enforcing type safety.

4.1.3 Byte Code Verifier

Java is a type-safe language. It ensures that access to objects is restricted to only those operations sanctioned by the types of the objects. When Java code is compiled to Java bytecode, `javac` ensures that the code does not breach type safety. For example, the compiler ensures that methods the code invokes actually exist, that their arguments and return values are compatible with those used in the call site and that code within the lexical scope of the call site is allowed to invoke the method.

Two separate issues can cause difficulties with verifying type safety by the Java compiler. The first is that the compiler can only rely on static, compile-time, information. Yet, the correctness of some operations cannot be verified at compile time. Two examples are type cast to a subtype and array bound checking. In both examples, the type safety of the operation depends on the actual values of the operands. Java resolves this issue by introducing run-time checks that verify type safety against the actual operands.

The second issue is that the types used to check type safety of the code are not necessarily the same types that will be used at run time. The Java compiler has no control over which Java bytecode files are shipped to the virtual machine for execution. It also has no control over which class loader is used to load these types and, consequently, over the run-time types used. Using different types at run time may void any type checks done at compile time. To overcome this issue, the Java virtual machine must verify that the bytecode it executes is type safe.

The discrepancy between compile-time and run-time types is not the only motivation for verifying bytecode in the Java virtual machine. Bytecode can be generated by tools other than the Java compiler [Bruneton et al., 2002, Chiba, 2000, Dahm, 1999] and there is no guarantee that code generated by such tools is type safe.

The bytecode verifier is a component of the Java virtual machine that verifies the Java bytecode loaded at a machine. It checks, *inter alia*, that the types of operands match the operators or the methods invoked, that methods do not cause stack overflow or underflow, that all registers and objects are initialised before use, and that scoping rules are not bypassed. Describing the implementation of a bytecode verifier is beyond the scope of this thesis. Leroy [2003] provides an excellent survey of algorithms and strategies for implementing a bytecode verifier.

4.1.4 Summary

This section presents some areas of the Java virtual machine that have relevance to the implementation of S-RVM. First amongst these is the distinction between the Java language and the Java bytecode. The Java virtual machine does not execute Java code. It executes Java bytecode. Consequently, "compile time" as used in this thesis is the time that the virtual machine compiles bytecode into machine code, rather than the time that `javac` compiles Java code into Java bytecode.

The observation that the Java virtual machine executes Java bytecode also implies that, for type safety, the Java bytecode needs be verified. The bytecode verifier, therefore, provides the protection which underlies any security mechanism in the language.

The primary role of class loaders in the Java virtual machine is to find and load class files. They are, however, also used to create type name spaces. These name spaces are not isolated, allowing sharing of types between class loaders. The forced use of the base class loader for Java core types is useful for creating a set of types common to all code. However, sharing `Object`, the top of the type hierarchy, hampers reference protection.

Thus, Java type system does not provide any control of reference propagation. Programmers that need to control the propagation of references in Java have to rely on other mechanisms. The security manager, which is the mechanism Java uses for access control, is both discretionary and adversely impacts performance.

The next section describes JikesRVM, the Java virtual machine which provides the basis for the implementation of S-RVM.

4.2 JikesRVM

JikesRVM is a Java virtual machine oriented toward providing a flexible test-bed for experimentation with virtual machine design. Having been used for many research projects, JikesRVM is a widely accepted experimental platform. Hence, its use as a basis for S-RVM provides a baseline to measure and compare the performance against S-RVM.

JikesRVM is a meta-circular virtual machine. That is, it executes in the same run-time environment it provides. Figure 4.1 shows the high-level structure of the JikesRVM run-time environment. Similar to other Java virtual machines, the run-time environment is layered, with the virtual machine executing at the bottom layer, the Java libraries in the middle layer and the application at the top. But, as most of the code of the virtual machine is written in Java, it uses some core services of the Java library, e.g. class loading and string objects.

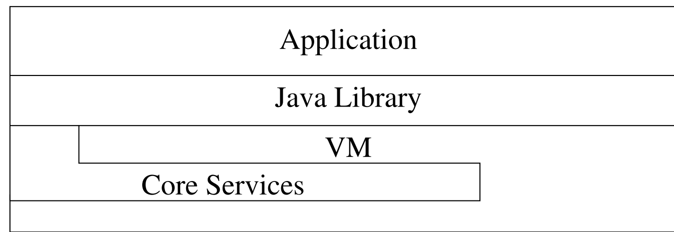

Figure 4.1: The JikesRVM runtime environment

Executing the virtual machine and the application within the same run-time environment offers performance benefits by removing the language barriers between the virtual machine and the application. When the virtual machine and the application share the run-time environment, they use the same object format, the same calling conventions, the same data representation and the same semantics. Thus, there is no need to convert data when it is transferred between the application and the virtual machine.

Furthermore, the use of the same language allows the compiler to inline virtual machine code in the application. When the application code invokes a virtual machine method, either explicitly by calling the method or implicitly when the virtual machine method implements a specific bytecode operation, the code of the virtual machine method can be subsumed into the application code and compiled together with it. The compiler can then use information it has for the specific call site when compiling the virtual machine code and optimise the virtual machine code for the specific call site.

This section presents the structure of JikesRVM with focus on areas required for understanding the implementation of S-RVM. It describes the compilation and memory management frameworks in JikesRVM, the use of *magic*, the interface between the Java library and JikesRVM, virtual machine initialisation and the implication of the shared run-time environment on the security of the virtual machine.

4.2.1 Compilation Framework

Unlike most Java virtual machines, JikesRVM never interprets bytecode. In JikesRVM bytecode is always compiled into native machine code which is executed directly by the processor. JikesRVM supports two compilers which represent a trade-off between compiler efficiency and code performance. The *baseline* compiler consists of a single pass that directly translates bytecode into native code without any intermediate representation. Instead of performing register allocation, the baseline compiler emulates the Java operand stack in memory. This simplistic approach to compilation results in very inefficient code. The compilation process, however, is very quick, reducing the overhead of compilation.

The *optimising* compiler translates bytecode to an intermediate representation on which it performs optimisations. It supports several optimisation levels that attain different trade-offs of compilation overhead vs. native code performance.

JikesRVM also supports an adaptive optimisation framework [Arnold et al., 2004] that uses information about compilation speed and native code quality to choose the appropriate compiler and optimisation level. Adaptive optimisation allows JikesRVM to focus the compilation effort on frequently used code and to adapt the optimisations to characteristics of the code.

In a typical scenario, method code is compiled using the baseline compiler when the method is first invoked. The compiled code collects profiling information on the frequency of execution of the method code. The profiling information is used to identify "hot" methods, which are recompiled using the optimising compiler. The optimising compiler also relies on some profiling information for decisions such as method inlining, which reduces the overhead of popular method calls, and branch prediction, which improves performance by reducing the need to flush the processor pipeline.

4.2.2 Memory Management

For memory management JikesRVM uses the MMTk toolkit [Blackburn et al., 2004, 2006a]. MMTk is an extensible framework for building high-performance garbage collectors.

To achieve high performance, MMTk focuses on the *fast path*—the code execution path taken in the more common or frequent cases. As part of that focus, MMTk uses thread-local data structures to handle the common case, avoiding the overhead of synchronisation on the fast path. To avoid the overhead of method invocation, methods on the fast path are inlined into the calling context. Inlining also allows the called methods to be optimised with the calling code, which allows the inlined code to be specialised to the context.

As Java uses automatic memory management, the application code needs to interface with the memory management in two ways. Memory is allocated through bytecode operations. The JikesRVM compiler implements these operations as calls to MMTk methods.

Memory is reclaimed by an offline garbage collection process. All the collectors in MMTk are stop-the-world collectors [JikesRVM User Guide]—the application code stops while garbage collection executes. To stop the application threads for garbage collection, the JikesRVM compiler inserts *yield points* at the entry code of every method and inside each loop in the code. The yield point checks whether garbage collection is required and synchronises the thread with the garbage collector.

The memory management needs low-level primitives, such as direct access to the memory and to the run-time stack. These primitives are not supported by the Java language because they can be used to bypass the type safety of the language. The next section describes the mechanisms that implement low-level primitives, including those required for memory management.

4.2.3 Compiler Magic

As a meta-circular virtual machine, the code of JikesRVM is written in Java. However, some of the low-level operations that the virtual machine needs to execute cannot be implemented in Java. Examples of such operations include invoking native code generated by the compiler, scanning and rewinding the execution stack when an exception is thrown and the allocation and reclamation of memory.

JikesRVM uses *compiler magic* [Frampton et al., 2009] to implement these operations. Compiler magic is code for which the compiler applies different semantics than specified by the language.

Three main forms of magic are implemented in the JikesRVM compilers. Magic methods are methods for which the compiler implements special semantics. When the compiled code includes a call to a magic method, the compiler intercepts the call, inserting code for the special semantics instead of the code for a method invocation. Examples of such operations include primitives for atomic access to memory, which are required for implementing low-level synchronisation, and operations that access the stack of the current thread.

Type extensions are another form of magic that extends the language type system. These include types like `Address`, which is an unboxed reference to a memory address. The `Address` type defines some operations such as reading and writing data from the memory or converting between `Object` references and the memory address the referenced object is stored in.

As `Address` is an unboxed type, values of the type are not objects and have no method dispatch table. Instead, all methods of `Address` are magic, and the compiler replaces calls to them with the native code that implements them. For example, calls to `Address.loadByte()` are replaced with the machine code to load a byte from the memory and calls to type conversion methods are simply eliminated from the resulting native code.

The third form of magic is using compiler pragmas to change the semantics of code within a scope. Examples of such pragmas are the `@Inline` and `@NoInline` annotations that override the compiler's inlining algorithm by forcing or preventing the inlining of the methods they annotate.

Another example is the `@Uninterruptible` annotation that is used for methods

whose execution should never be interrupted. Interruptions caused by the code are prevented by forbidding the use of bytecode operations, such as synchronisation or memory allocation, that may result in interrupts and by only allowing the code to invoke other uninterruptible methods. Furthermore, the compiler does not generate yield points in uninterruptible methods, avoiding unintended pre-emption during the execution of the method.

4.2.4 Library Interface

The execution environment of a Java program includes both the virtual machine and the Java library. Some of the Java library classes abstract virtual machine entities. Hence, the implementation of the Java library needs to interface with the virtual machine. At the same time, JikesRVM itself is written in Java. As such, it depends on some Java classes and needs an interface to the library.

JikesRVM can be built with either the GNU Classpath [GNU Classpath] implementation of the Java library or with the Apache Harmony [Apache Harmony] implementation. As currently S-RVM only supports GNU Classpath, only the interface with this implementation is described here.

The GNU Classpath implementation is mostly written in Java. A few classes, including the floating point numbers, basic I/O and graphics packages, are implemented in C. To interface with the virtual machine, the GNU Classpath uses a set of *hooks*, which are classes that are expected to be implemented by the virtual machine provider.

Hooks are defined as package-scoped classes. The names of hook classes are the name of the class they provide the implementation to with the prefix `VM`. For example, the class `VMClass` provides the hooks required for the implementation of the `Class` class, and the class `VMString` provides those required for `String`.

For some of the library classes, GNU Classpath also includes a protected field for virtual machine specific data. When the field is provided, GNU Classpath also provides the hooks required for its initialisation.

JikesRVM occasionally needs access to internal members of Java library classes. Instead of modifying the public interface of these classes, JikesRVM adds the class `JikesRVMSupport` to the Java library package. The methods of `JikesRVMSupport` provide access to package-scoped members of library classes.

4.2.5 Virtual Machine Build

A substantial set of services is required for the virtual machine to start operating. The virtual machine needs to be able to load classes, compile code, allocate memory and support other miscellaneous virtual machine services before it can load other

services and execute a program. To provide the initial set of services JikesRVM builds a *boot image* that contains a memory image of the virtual machine.

To create the boot image JikesRVM uses the *boot image writer* program. The boot image writer is a Java program that runs on any existing Java virtual machine. It creates a mock-up of a running JikesRVM, including loaded classes and compiled methods. It then converts objects from the mock-up it created to the JikesRVM object model, creating an image of a running JikesRVM virtual machine. This image is written to the boot image file.

A minimal boot image need only use the baseline compiler and does not need to include the optimising compiler in the image. However, to reduce startup time and improve performance, the boot image can include additional classes and the code in it can be compiled with the optimising compiler.

To run the virtual machine, JikesRVM uses a small C program that maps the boot image into the memory and jumps to the virtual machine initialisation code.

4.2.6 Security

Language-based security is one of the major aims of the Java programming language [Gosling and McGilton, 1996]. JikesRVM, however, significantly lacks in that area. The most obvious security issue in JikesRVM is the absence of a bytecode verifier [Yellin, 1995]. The bytecode verifier is one of the cornerstones of the Java security and without it JikesRVM cannot ensure that code does not breach the Java safety.

Adding a bytecode verifier to JikesRVM does not seem to be a difficult problem. Algorithms for bytecode verification are available [Leroy, 2003] and the bytecode verifier is mostly independent of the rest of the virtual machine. A more subtle problem, yet much more difficult to fix, is the sharing of the run-time environment between the virtual machine and the application.

Sharing the run-time environment blurs the boundaries between the application and the virtual machine. The application and the virtual machine share the same type hierarchy. As Figure 4.2 demonstrates, this hierarchy includes some classes that are categorically application classes and others that are unambiguously within the virtual machine. However, the same Java library classes are used by both the virtual machine and the application.

The direct consequence of the shared type hierarchy is that it provides application code with access to internal virtual machine types. This access can be exploited by application programs to bypass the type safety of language. For example, JikesRVM uses the `Statics` class to manage values of static fields in the run-time. Application code has access to the `Statics` class methods and can use this access to retrieve or

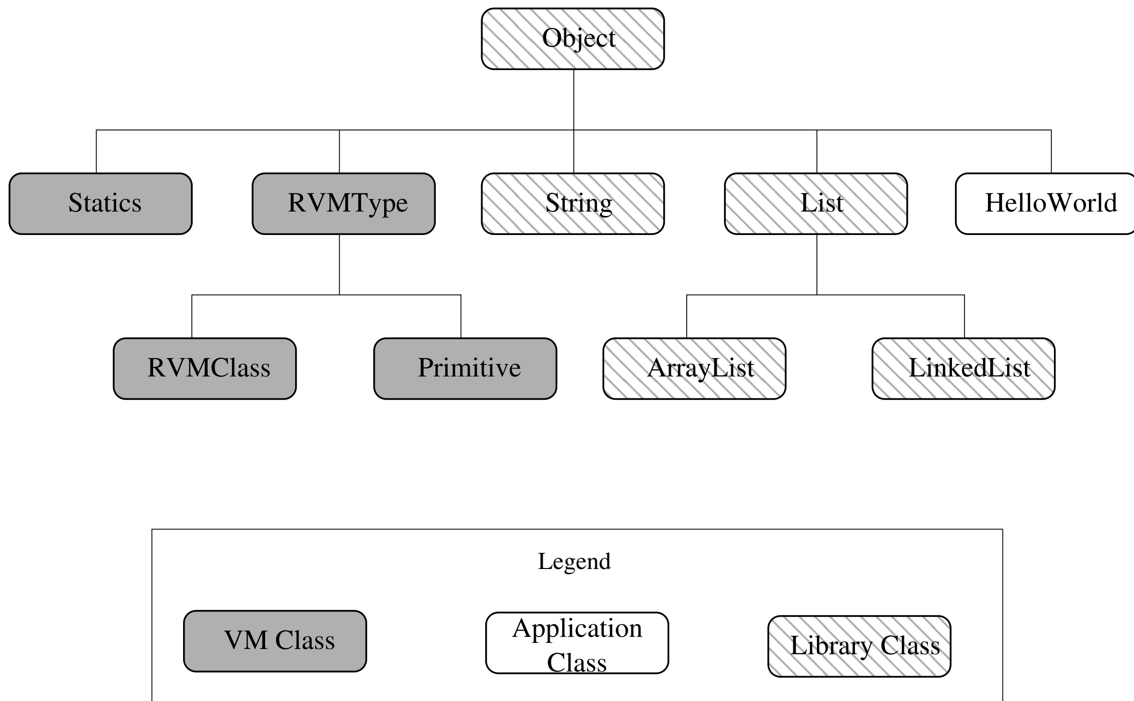

Figure 4.2: Type hierarchy in JikesRVM

update the value of any static field in the run-time, irrespective of the access scope of the fields.

The problem of the `Statics` class can be solved using a simple mechanism. Instead of using public static methods for access, the `Statics` class can be implemented as a Singleton [Gamma et al., 1995], using the instance methods for access. Releasing the reference to the singleton instance only to trusted callers ensures that untrusted application code is unable to use the `Statics` methods.

Three issues with the use of the Singleton pattern make this solution less than ideal. First, JikesRVM consists of over 1,300 classes. Each of these is accessible to application code and is, therefore, in the attack surface [Manadhata and Wing, 2011] of the virtual machine.

The second issue is that as the Java library is shared with the application, identifying trusted code is not easy. Code of Java library classes can execute both in the application and in the virtual-machine context. Identifying the context in which library code executes requires expensive dynamic checks [Jones and Ryder, 2008, Lin et al., 2012].

Thirdly, as Java does not provide any form of reference protection, trusted code should be trusted not to further disclose the reference to the sensitive objects, such as the `Statics` singleton.

S-RVM uses the Exported Types design to separate the type hierarchies of the virtual machine and the application. Consequently, the attack surface of the virtual machine is significantly reduced, identification of the context code and objects

belong to is simple and reference protection ensures that sensitive internal objects are not disclosed to the application. The next section presents an overview of the implementation of S-RVM.

4.3 An Overview of S-RVM

The high-level structure of the S-RVM run-time environment is depicted in Figure 4.3. The run-time environment consists of a *root task*, which executes the virtual machine and an *application task*, which executes an untrusted application. The virtual machine provides services to the application task using a virtual machine interface layer. The root task also executes a trusted application, which is responsible for launching the application task.

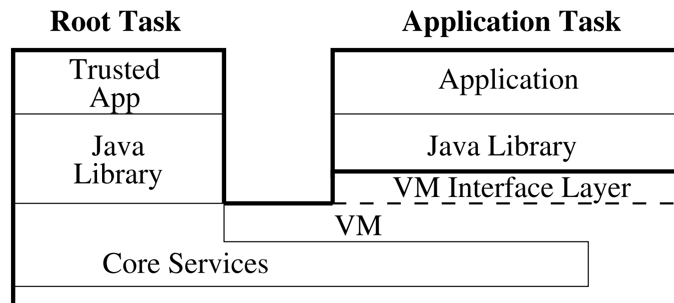

Figure 4.3: The S-RVM runtime environment

To provide reference protection, S-RVM implements the Exported Type design. In a nutshell, the Exported Types design creates a separate name space for each component and supports two type operations. The *export* operation makes a type of the exporting component available for use by other components and the *import* operation binds a type name in a component to a type exported by another component. The Exported Types design also creates a *remote* protection scope to restrict access to members of exported types.

Each S-RVM task consists of a sealed zone that contains private data and an interface zone that is accessible by the other task. The interface zone of the root task consists of types it exports. The interface zone of the application task consists of subtypes of types it imports from the root task.

S-RVM uses Java annotations [Java Community Process, 2004] for the syntax of the Exported Types operations. Types to be exported are annotated with the `@Export` annotation. The `@Remote` annotation identifies methods of an exported type that are in the remote interface of the type. Listing 4.1 shows part of the code of the type `RVMClass`, which is JikesRVM's internal representation of a Java class.

The `@Export` annotation in line 1 indicates that the class is exported. That is, code running in the application task can import this class and use it. The method

```
1  @Export
2  public class RVMClass extends RVMType {
3
4    private final RVMClass superClass;
5    private FieldLayoutContext fieldLayoutContext
6
7    @Remote
8    public RVMClass getSuperClass() {
9      return superClass;
10   }
11
12   public FieldLayoutContext getFieldLayoutContext() {
13     return fieldLayoutContext;
14   }
15 }
```

Listing 4.1: An exported class

`getSuperClass` in line 8 has the `@Remote` annotation and is, therefore, accessible from the application task. The method `getFieldLayoutContext` in line 12, while being `public`, does not have the `@Remote` annotation. Hence it is not accessible from code running in the application task.

To import types, the application task loads a *stub* of the type. Stubs are Java types which specify the remote interface of an exported type and that have the `@Import` annotation. S-RVM provides a tool that parses exported types and automatically generate the stubs. Listing 4.2 shows the stub file generated from the code in Listing 4.1.

```
1  @Import
2  public class RVMClass extends RVMType {
3    public RVMClass getSuperClass() { return null; }
4  }
```

Listing 4.2: An import stub

The `@Import` annotation in line 1 directs the virtual machine to bind the type name (`RVMClass` in this example) to a previously exported type with the same name. The body of the stub type is not used when importing the type.

The body of stub types includes the remote method declarations with a minimal method implementation. The minimal implementation is required to allow compilation of the stub into bytecode. The implementation of `void` methods is empty. For methods returning `boolean`, a number or a reference, the minimal implementation is to return `false`, 0 or `null`, respectively.

The body of the stub type helps in early detection of errors in the declaration

of, and in the use of, the remote interface. When generating the code for the stub type, the signatures of remote methods are checked to ensure that they consist of types that are shared between tasks, i.e. primitive and exported types. Furthermore, compiling client code against the stub class files instead of against the exported type ensures that the client code only uses the remote interface of the exported type.

The Exported Types design is based on creating a separate name space for each task. The crux of separating the application and the virtual machine lies in creating the separate name spaces using a separate base class loader for each task. The base class loader in Java is responsible for loading the Java library classes. Using a separate base class loader for each task implies that the class `ClassLoader` in each task is private to the task. Hence, class loaders cannot be shared between the tasks and class loading cannot be delegated across task boundaries. Using separate base class loaders, therefore, effectively creates a separate class name space for each task. The Exported Types design is a partial design of a type system, which can be used either as a basis for a complete type system or as an extension to an existing one. S-RVM uses the Exported Types design to extend the Java type system with support for reference protection.

Tasks in S-RVM are represented by `RVMTask` objects. The `RVMTask` object representing the root task is created during the S-RVM build time. The trusted application within the root task creates the `RVMTask` object representing the application task. The ability to dynamically create multiple tasks provides an initial support for extending S-RVM to a multi-tasking Java virtual machine.

The key members of the `RVMTask` object are the base class loader of the task and the upcall interface into the task. A partial class diagram is displayed in figure 4.4. The base class loader object, whose type is `BootstrapClassLoader`, provides the functionality required to load the implementation of the classes in the core Java library. The class is a subclass of `RVMClassLoader` which is used for storing the virtual machine's representation of a class loader.

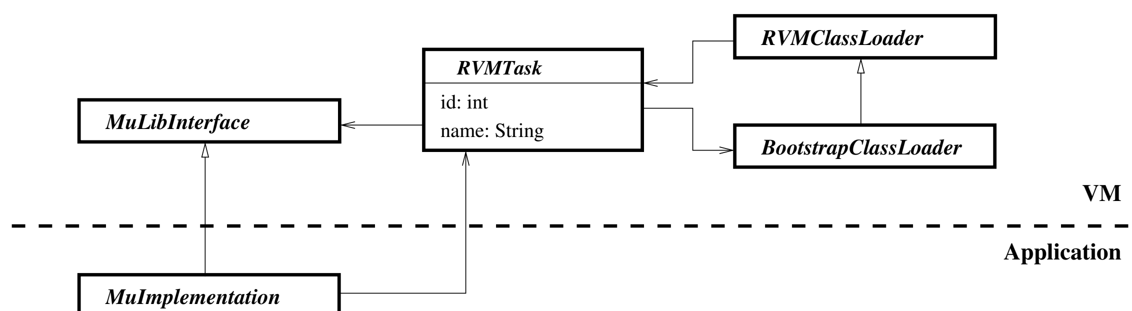

Figure 4.4: RVMTask class diagram

`RVMClassLoader` as well as the upcall interface form part of the *VM Interface Layer*, which provides the mean for communication between the application task

and the root task. The layer consists of types exported by the root task, which the application task can use for accessing the root task, as well as of exported types that the application task extends to allow the root task access to the application task. These are described in further detail in Section 4.5, following the description of the implementation of the import and export mechanisms.

4.4 Import and Export

Having a base class loader for each task creates the tasks name spaces required for the Exported Types design. Type export and import require a shared name space. S-RVM implements this shared name space as a simple hash table that translates from a name to a type.

When S-RVM parses a class file it checks for the existence of the `@Export` and `@Import` annotations. Exporting a type takes two steps. First, when finding an `@Export` annotation while parsing a class file, the name of the class is reserved, ensuring that no concurrent attempts to export a type of the same name succeeds. After parsing completes, the type is created and is associated with the reserved name.

The purpose of the two phase process is to avoid a potential race condition when two classes that share the same name are exported. Reserving the name when parsing the class file ensures that only the first one of these will complete parsing and will be created.

The handling of `@Import` annotations is straightforward. When the class file parser finds the annotation, it searches the shared name space for the type name, returning the exported type if found and throwing an exception otherwise. The only complication is that this approach does not handle imported magic types.

In JikesRVM there are a fixed number of magic types. The JikesRVM compiler identifies type references as magic by comparing them to each of the known type references of magic types. In S-RVM, import occurs when the type is loaded. Hence, when a type is first referenced, S-RVM cannot know whether the type will be imported, let alone whether it is an imported magic type. This results in two problems. First, the number of type references that can refer to magic types is no longer fixed and comparing against a fixed list no longer works. Second, the compiler cannot know if a type reference is for a magic type without loading the type.

S-RVM solves these two problems by forcing the load of imported magic types during task initialisation and by marking type references as magic. (See Listing B.5 in Appendix B.2.) The main downside of this scheme is that the identity of the imported magic types must be known at virtual machine build time. However, this is not a major limitation because *a priori* knowledge of magic types is required for

handling them even when they are not exported.

The virtual machine interface layer consists of 49 exported types. Compared with 1,300 classes in the JikesRVM virtual machine and over 5,000 Java library classes that JikesRVM trusts, this represent a reduction of over two orders of magnitude in the number of classes exposed to the application.

Exporting a class puts it in the VM interface layer. Design patterns used for this layer are described in the next section.

4.5 The VM Interface Layer

The root task provides the run-time environment for the application task. For the application to interact with its environment, it needs to communicate with the root task. This communication includes transferring both data and control between the application and the root task.

Due to the separate base class loaders, types in the root task are incompatible with types in the application task. That is, references to objects in the root task are not type compatible with and cannot be assigned to references in the application task and vice versa.

To support communication between the tasks, S-RVM implements the Exported Types Design. Types that the root task exports are imported by the application task, allowing the application task to hold references to object of these types. These objects form the *VM Interface Layer* (Figure 4.3), which provides the means for communication between the tasks.

Importing types creates a dependency on the task that exports the types. To avoid mutual dependencies between the root and the application tasks, only the root task exports types. A type exported by the root task can be used for downcalls, i.e. calls from the application task to the root task. To support upcalls, or calls from the root task to the application task, S-RVM uses the callback mechanism described in Section 3.4. That is, the application task extends abstract types exported by the root task. As the application task provides the implementation of these extended types, objects of the extended types have access to objects of the application task. References to these objects can be used by the root task, by virtue of these extended types being subtypes of types declared and exported by the root task.

Figure 4.5 presents the class diagram of a typical pattern used for bi-directional communication between the root and the application tasks. The root task exports two types: `DowncallInterface` and `AbstractUpcall`. The application uses downcalls by invoking methods of the class `DowncallInterface`. The class `AbstractUpcall` declares the upcall methods that the root task may invoke.

The pattern consists of two objects, one in each task. The `DowncallInterface`

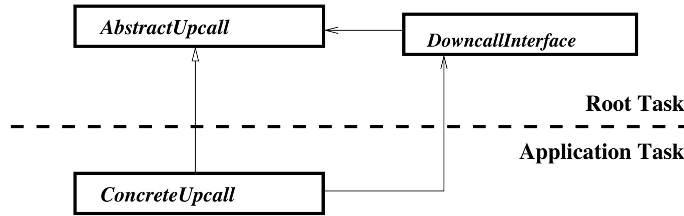

Figure 4.5: Bi-directional communication between tasks

object provides the interface for the application task to invoke downcalls. The `ConcreteUpcalls`, whose type extends the exported type `AbstractUpcalls`, implements the upcalls.

An example of using this pattern is shown in Figure 4.4 where an object of the class `RVMTask` provides a downcall interface for task-related requests from the application task. Examples of services include loading native libraries, access to the base class loader and querying the type of an object.

The upcall interface is declared in the exported abstract class `MuLibInterface` and is implemented in the application task class `MuImplementation`. Upcalls are required when the root task needs to create or access objects in the application task. For example, when the root task loads an application class, it needs to create the `Class` object representing the loaded class. The `Class` class of the created object is within the application task. To create this object, the root task invokes the method `MuLibInterface.createClassForType()`, which calls a protected constructor of the `Class` class. (See Listing B.2 in Appendix B.1.) Other methods of `MuLibInterface` are used for application task initialisation and for implementing Java Native Interface (JNI) calls [Liang, 1999].

Another example of this use of the pattern for bi-directional communication is the implementation of the interface for managing Java threads. A partial class diagram for the classes involved with Java threads is shown in Figure 4.6. The class `Thread` is the Java library implementation of a thread abstraction. `RVMThread` is the class that represents a thread in the root task. `VMThread` provides both the Classpath hook (Section 4.2.4) and the implementation of the upcall interface for the thread. The upcall interface itself is defined in the abstract class `MuThread`.

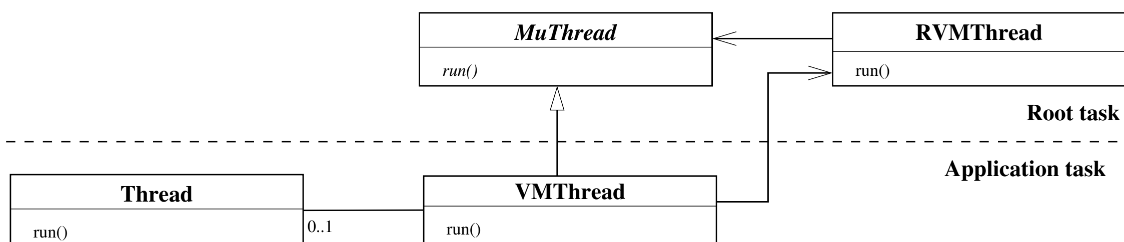

Figure 4.6: Classloader class hierarchy

Hence, for example, when a thread is started, the root task needs to invoke its `run()` method. To achieve this it invokes the `run()` method of `RVMThread` which makes an upcall to the `VMThread` hook object, which, in turn, invokes `Thread.run()`.

The use of this approach introduces an extra object (of class `VMThread` in the example above) that acts as intermediary between the Java library object (`Thread`) and the root task representation of a thread (`RVMThread`). As additional objects incur overheads, it is worth examining why this intermediary object is required.

There are three possible ways to avoid the intermediary object. The upcall functionality can be merged into the root task object, it can be merged into the Java library object and the root task can use reflection for implementing the upcall functionality.

Merging the upcall functionality into the root task object can be achieved by the application task extending the root task object type. There are, however, two problems with this approach. First, it makes the root task object dependent on types defined in the application task. This creates a circular dependency for objects that need be created before the task can be active. For example, when an application class is loaded, the root task needs to create the objects representing methods of the class. If these objects need to be of a type that the application task defines, these objects cannot be created until that type is loaded, preventing the loading of this type until after it is loaded.

The second problem is that the application task needs to extend the root task object, limiting the type hierarchy within the root task. It would mean that the types exported by the root task cannot be `final` and cannot be extended at the root task without affecting the application task.

The second approach for avoiding the intermediary object is to merge it into the Java library class. The problem with this approach is that classes that implement upcall methods need to be subtypes of root task types. The specification of the Java library define the type hierarchy of the Java library classes, preventing this subtyping. Furthermore, even if the specifications are relaxed, Java library classes in the application task are expected to be subtypes of the application task `Object` type. They cannot, therefore, extend root task types.

The third approach, using reflection, is feasible. However, the reflection interface is less intuitive than the method invocation interface and reflection misses optimisation opportunities, such as inlining the invoked method.

Due to the limitations of the alternative approaches, S-RVM uses the pattern described above, with the intermediate object.

To ensure the protection of the root task, code in the root task does not trust the code in the application task. The next section discusses implementation issues that relate to this lack of trust.

4.6 Creating a Trust Boundary

As discussed above, the interface layer is 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 virtual machine 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 virtual machine types the virtual machine 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. Listing B.2 in Appendix B.1 demonstrates another case of extra tests due to lack of trust.

Lack of trust also implies that arrays cannot be shared between the application and the virtual machine. Instead, S-RVM uses wrapper classes to provide the application with read-only access to virtual machine 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, in which 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 virtual machine 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 virtual machine would introduce a significant overhead. On the other hand, sharing `String` objects or their backing stores would reduce isolation and require the virtual machine to trust the application not to modify the contents of `String` objects.

A simple approach for safely sharing the backing store is to only provide the application read access to it. The backing store is an array of characters and, as mentioned in Chapter 3, Java arrays cannot be exported and must be wrapped. Wrapping, however, introduces an overhead due to the extra reference required for accessing the array through the wrapper. As string operations are fairly common, this overhead is not acceptable.

To provide read-only access without the overhead of the wrapper, S-RVM introduces the unboxed wrapper type `MuCharArray`. `MuCharArray` supports two exported magic operations `get()` and `length()` which translate to the corresponding operations on the array. The virtual machine can also unwrap `MuCharArray` to get access

to the backing store. However, the unwrap operation is not exported, ensuring that the untrusted application code cannot modify the backing store.

Read only access to the backing store is insufficient. The Java library needs a way of creating strings, which require modify access to the backing store. The backing store of strings, then, goes through two phases: initialisation and use. During the backing store initialisation phase, the application needs modify access while sharing is prohibited. During the use phase, the application only needs read access to the backing store, while sharing is desired.

S-RVM implements the initialisation phase using the unboxed wrapper `MuWriteableCharArray`. This wrapper is used for read/write access to an underlying virtual machine character array. It also provides a `seal` operation, which both prevents further modifications to the underlying array and returns the `MuCharArray` corresponding to it.

Implementing the seal operation requires keeping the phase information in the backing store. Unfortunately, Java does not support any feature similar to C's flexible array members [International Standardization Organization], and keeping a flag with an array requires an extra level of indirection. To avoid that extra level of indirection, S-RVM uses the first element in the array (at index 0) as a flag. When the value of the first element is 0, the backing store is modifiable. The `seal` operation sets the value to 1, preventing further modifications to the backing store. As the `seal` operation is the only way for the application to get the `MuCharArray` wrapper for the backing store, this scheme guarantees that sharing only occurs after the backing store is sealed.

For efficiency, the S-RVM interface also includes some utility methods that allow copying to unsealed `MuWriteableCharArray`s and from `MuCharArray`s using the virtual machine 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 virtual machine. Appendix B.3 shows details on creating and using the `String` backing store.

The root task, unlike the application task, is privileged and needs unfettered access to all the objects in the system. The next section presents the mechanisms for providing privileged access from the root task to the application task.

4.7 Privileged Access for the virtual machine Task

The virtual machine 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 virtual machine and the application S-RVM adds 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.

References that can refer to all the objects in the system present the risk of breaching the reference protection. S-RVM relies on the virtual machine not transferring `MuObject` references that refer to root task objects into the application task. However, it does not introduce checks to ensure that references passed as `MuObject` do not breach the protection. Instead, for the case of a badly behaved virtual machine, S-RVM relies on the generic nature of the `MuObject.get()` method. Generic methods in Java introduce a dynamic type check. If the virtual machine does transfer a reference which breaches the reference protection, the dynamic type check will fail and the reference will not be used.

In JikesRVM the type hierarchy is shared between the virtual machine and the application. Hence, in JikesRVM an `Object` reference can point to any application object. Some of the JikesRVM code relies on that property and uses `Object` references to point to application objects. Replacing these references with `MuObject` references would require rewriting significant parts of the JikesRVM code. To avoid this rewrite, S-RVM allows references of the virtual machine `Object` type to point to any application object.

Unboxed wrapper types are used for passing arrays from the application task to the root task. These are required for array copying and for transferring the bytecode representation of classes that are loaded to the virtual machine. For efficiency, S-RVM provides a specialised array copy method for each primitive array type. S-RVM cannot, however, trust the application task to use the correct array type for each method. To avoid using dynamic type checks, S-RVM uses multiple unboxed wrapper types—one type for each primitive array type and an additional type for all reference arrays. It relies on the type safety of the language to ensure that the wrapped array type matches the wrapper and, consequently, that the wrapped array types match the specialised array copy methods.

Like the `MuObject` type, the unboxed array types present a risk of breaching the reference protection. This risk is, however, much smaller. S-RVM does not provide any interface that passes wrapped arrays from the virtual machine to the application, nor does it provide any interface that allows the application to unwrap these wrappers.

To use the array wrappers, the virtual machine unwraps them and casts the references to the corresponding virtual machine array types. This implies that virtual machine array type references can refer to application array types.

S-RVM's implementation of the Exported Types design allows sharing the types of method arguments and return values. It does not, however, handles exceptions. S-RVM approach for handling cross-task exceptions is the topic of the next section.

4.8 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 virtual machine is required to generate and throw exceptions when certain conditions occur, e.g. when the application references a `null` pointer or when running out of memory.

Due to the separate type hierarchies, exceptions generated in the root 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 exceptions thrown across the VM interface layer to an equivalent exception type in the receiving task. The code for wrapping remote methods and for converting exceptions is further discussed in Appendix B.4.

When S-RVM converts an exception it may lose some type information. This loss occurs because the application can declare exception types that have no equivalent in the root task. In these cases, S-RVM converts application exceptions to the most specific supertype defined in the virtual machine.

This loss of information cannot affect the virtual machine's response to the exception because the virtual machine can only handle the exception types it recognises. If, however, the virtual machine does not catch the exception before returning to the application or if the virtual machine re-throws the exception, the loss of information may affect the application.

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. However, as relying on the information loss is very rare, this solution has not yet been implemented. The information loss does not affect any of the test cases tried.

4.9 Implementation Verification

The implementation of Exported Types in S-RVM is not complete. The lack of a bytecode verifier means that malicious code can bypass the protection offered by the remote interface and can, consequently, breach the protection model. Further opportunity for breaching the model exists through the trust placed on the virtual machine code to not abuse the `MuObject` unboxed wrapper and the assignment compatibility of every type with the virtual machine's `Object` type. Additionally, there is no guarantee that programming errors in the virtual machine code do not inadvertently expose more risks to the protection model. The implementation of S-RVM itself does not include mechanisms to verify its correctness. This section illustrates some potential approaches for achieving verification.

Techniques for implementing bytecode verifiers have been surveyed in Leroy [2003]. Adapting a bytecode verifier to support Exported Types, as implemented in S-RVM, may require two modifications. Some type information, e.g. the assumption that every object can be assigned to the `Object` type, may be hard-coded within the implementation of the verifier. Such assumptions may no longer hold for S-RVM and the implementation of a bytecode verifier will have to be changed to avoid these assumptions. The second modification is the implementation of the remote interface which is not a part of the Java language. In the absence of a concrete implementation of a bytecode verifier, it is hard to estimate the amount of work required for these modifications.

As discussed above, the `MuObject` type and similar unboxed wrappers do not pose a significant security risk because a type check is often introduced before the use of such values. Nevertheless, a type check is omitted when converting `MuObject` to the task's `Object` type or when the optimising compiler optimises the type check out. Furthermore, even though the reference cannot be used, transferring a reference to a private value constitutes a breach of the protection model. Instrumenting remote methods of exported classes to check the types of values passed across the interface is an effective approach for protecting against such inadvertent breaches.

Another possible breach is unexpected transitions between the application and the virtual machine contexts. The design of S-RVM aims to prevent such transitions, however, verification can increase the confidence in the implementation. A possible way to verify this is to instrument the remote interface to keep track of the current context of each thread. When combined with assertions at key points in the code, such tracking can provide a validation of the correctness of the implementation.

This approach for tracking the current context is based on the work of Lin et al. [2012]. That work identifies transition points between the application and the virtual machine and instruments them to trace the execution context. Lin et al.

[2012] reports identifying 41 classes as transition points. This number is similar to the size of the interface layer in S-RVM—49 types, including 13 unboxed wrappers.

Tracing the execution context, as done by Lin et al. [2012], can be used for dynamically enforcing reference protection. The reported overhead of tracking the context is 0.6%. However, this overhead is only for tracking the context. Tracking the allocation context of objects incurs an overhead of 6% [Lin, 2012] and verification of reference assignments is likely to incur a prohibitive overhead. Hence, while the approach is possible, it is unlikely to be used for reference protection.

4.10 Summary

The shared run-time environment in JikesRVM offers performance benefits by removing obstacles to communication between the virtual machine and the application. At the same time, it blurs the boundaries between the application and the virtual machine, exposes the virtual machine to potential attacks and presents a large attack surface.

S-RVM uses the Exported Types design to separate the virtual machine and the application. It creates a well-defined boundary between the application and the virtual machine. Separating the application from the virtual machine reduces the number of classes exposed to the application by over two orders of magnitude while at the same time maintaining the benefits of the shared run-time environment.

S-RVM is not a panacea to all the security problems in JikesRVM. It does not implement a bytecode verifier and it makes only feeble attempts to harden the virtual machine interface layer. Nevertheless, a smaller attack-surface area implies better security [Manadhata and Wing, 2011].

S-RVM is a proof-of-implementation of the Exported Types design. It represents a substantial change in the design of JikesRVM and as such requires significant modifications to the software system. Many of these changes are generic in nature. In particular, the use of annotations for the syntax of the Exported Types design, the stub file generation tool and the creation of name spaces by using multiple base class loaders can be used for other implementations of Exported Types in Java. Other modifications are specific to the scenario of separating the virtual machine from the application.

The lack of attention to security in the design of JikesRVM is a complete antithesis to the focus put on performance. Dozens of researches and developers have poured countless hours into algorithms and techniques for improving the performance of JikesRVM. Consequently, JikesRVM provides an excellent baseline to measure the performance impact of the Exported Types design. This impact is discussed in the next chapter.

Chapter 5

Performance Evaluation

Preceding chapters have described the Exported Types design and its implementation in S-RVM. S-RVM employs the Exported Types design to separate the virtual machine from the application. Its main benefit is the enhanced security it provides. It does not aim to improve the performance of the virtual machine. While security often comes with a performance cost, the Exported types design is based on the promise of efficient inter-component communication. The purpose of this chapter is to measure the overhead that S-RVM, as an implementation of the Exported Types design, introduces and demonstrate that this overhead is not prohibitive.

The test platform is an IBM x3500 server with two quad-core Xeon E5345 processors and 24GB of RAM; running Fedora release 16. The virtual machine was compiled using the *production* configuration, which uses edge-count profiling information collected from running the DaCapo `fop` benchmark. Both S-RVM and JikesRVM are retrofitted with the `Double.toString()` implementation from the OpenJDK [Oracle Corporation]. The OpenJDK implementation is written in Java, unlike the Classpath implementation which uses native C code, and provides a significantly better performance for both JikesRVM and S-RVM.

The DaCapo benchmark suite version 2006-10-MR2 [Blackburn et al., 2006b, 2008] is used for most of the benchmarks. The suite includes 11 benchmarks of varying characteristics and is a de-facto standard for measuring the performance of a Java virtual machine. The newer version of the benchmark suite (version 9.12) was not used because JikesRVM is not currently compatible with some of its benchmarks.

When an application executes within an S-RVM task, it requires its own copy of the Java library, separate from the copy of the library the virtual machine uses. This extra copy requires more memory, both for the static values that are replicated and for the data used for representing the classes. Consequently, S-RVM has a larger memory footprint than JikesRVM.

The JikesRVM boot image contains all of the virtual-machine classes as well as significant parts of the Java library preloaded. 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 longer to start than JikesRVM.

Furthermore, S-RVM, like JikesRVM, initially uses the baseline compiler to compile methods. Hence, when S-RVM loads library code to the task, the compiled code is slower than library code pre-compiled in the virtual machine. Consequently, until the optimising compiler has time to recompile the library code, the application is slower in S-RVM than in JikesRVM.

On the other hand, the use of a separate library for the application allows profiling data to be collected independently for the virtual machine and for the application. This profiling data better represents the different workloads that the application and the virtual machine generate and allows better optimisation of the library code in both the virtual machine and the application. The better optimisation offsets the increased code complexity in S-RVM. The result of this better optimisation is that at the steady state S-RVM does not underperforms JikesRVM.

This chapter presents the costs associated with S-RVM. It discusses the memory footprint, analyses the application start up costs and presents the steady-state performance of S-RVM.

5.1 Memory Usage

S-RVM uses more memory than JikesRVM, both for the additional copy of the Java library classes and for the automatic exception catch block used for exception conversion. Both the heap usage and the size of the boot image are larger with S-RVM.

The S-RVM boot-image size is 50.6MB or 2.25MB more than JikesRVM. Of these, 16.24MB are used for code and 10.7MB are code maps which hold information required for translating machine code addresses to bytecode addresses, finding the stack layout at machine code addresses and identifying exceptions that are to be captured at a machine code address. Diagram 5.1 shows a breakdown of the boot-image overhead.

1.39MB, or over half of the memory overhead in the boot image is the result of the exception-conversion code in S-RVM. The exception-conversion code automatically wraps methods with an exception handler that converts virtual machine exceptions to application exceptions and vice versa. Wrapped methods include methods on the virtual machine interface layer as well as virtual machine methods that implement Java bytecode operations. These wrapped methods are often inlined into code, resulting in a significant increase in both the code size and the size of code maps used, for example, by the garbage collector and exception handling code.

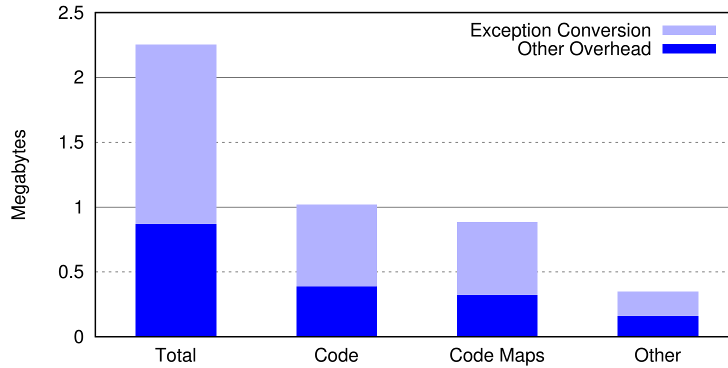

Figure 5.1: S-RVM Boot image overhead

Heap usage is compared by measuring the minimum heap size required for the benchmarks in the DaCapo suite. The minimum heap size is determined experimentally and is the smallest size of a heap that allows 3 iterations of the benchmark to execute to completion, using the JikesRVM *production* configuration. Table 5.1 summarises the results.

	Without Compiled Code			**With Compiled Code**		
Benchmark	**JikesRVM**	**S-RVM**	**Overhead**	**JikesRVM**	**S-RVM**	**Overhead**
antlr	22MB	30MB	8MB (36%)	23MB	32MB	9MB (39%)
bloat	40MB	45MB	5MB (13%)	41MB	48MB	7MB (17%)
chart	36MB	45 MB	9MB (25%)	37MB	47MB	10MB (27%)
eclipse	60MB	69MB	9MB (15%)	66MB	76MB	10MB (15%)
fop	31MB	40MB	9MB (29%)	34MB	41MB	7MB (21%)
hsqldb	102MB	110MB	8MB (8%)	102MB	111MB	9MB (9%)
jython	35MB	44MB	9MB (26%)	39MB	48MB	9MB (23%)
luindex	24MB	31MB	7MB (29%)	26MB	32MB	6MB (23%)
lusearch	45MB	53MB	8MB (18%)	45MB	54MB	9MB (20%)
pmd	37MB	45MB	8MB (22%)	40MB	47MB	7MB (18%)
xalan	47MB	54MB	7MB (15%)	51MB	59MB	8MB (16%)
Average			7.9MB			8.3MB

Table 5.1: Minimum heap size

By default, JikesRVM does not account for compiled code in the heap size. The table reports the minimum heap size both with this default behaviour, i.e. without accounting for compiled code, and when accounting for compiled code.

As demonstrated, S-RVM requires about 8MB more than JikesRVM. The number is fairly consistent across all benchmarks and is the result of having a separate copy of the Java library. The overhead for compiled code is less than 1MB. It is hard to experimentally measure the exact overhead because JikesRVM uses a 1MB granularity on heap size.

The added copy of the library also slows down the application start up. This is discussed in the next section.

5.2 Application Task Startup

The JikesRVM image contains significant parts of the Java library pre-compiled 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. When starting to execute, S-RVM is, therefore, much slower than JikesRVM.

Running the time-honoured "Hello World!" application on JikesRVM and on S-RVM shows that for small applications S-RVM is over four times slower than JikesRVM. (223ms vs 53ms.) Most of the overhead consists of loading and initialising classes. Diagram 5.2 shows the main phases in executing an application task in S-RVM and the number of classes loaded during the execution.

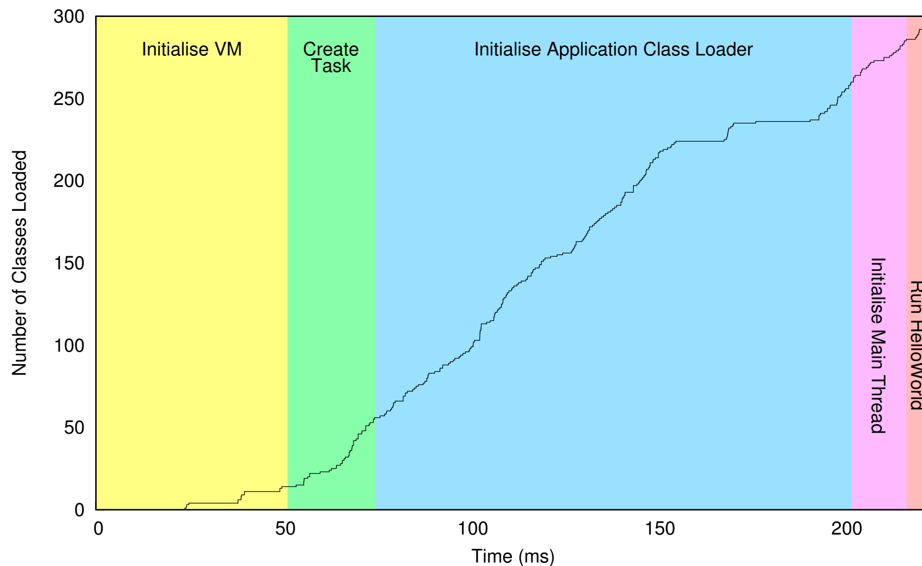

Figure 5.2: Class loading during task startup

The first step is to initialise various components of the virtual machine and to parse the command line arguments. This phase takes about 51ms and is required in both JikesRVM and S-RVM, however, whereas JikesRVM is ready to execute the application at the end of this phase, S-RVM now needs to create the application task.

Creating a task takes about 23ms, and consists of creating an `RVMTask` object, a base class loader and the library interface object and of loading enough classes to be able to create `String` and `Class` literals. At the end of this phase the task is able to load further classes.

The third phase is to create an application class loader for the application task. The application class loader in Java is responsible for loading the code of the main application executed on a virtual machine. During this phase over 200 different classes are loaded. Loading and initialising these classes takes 136ms.

This phase demonstrates the cost of loading classes. In JikesRVM, creating the application class loader is part of the virtual-machine initialisation phase. As all classes required for the application class loader in JikesRVM are already preloaded in the boot image, the virtual machine only needs to create the application class loader object and to initialise it, requiring only 61μs.

The last two phases are completing the initialisation of the main thread (15ms) and executing the main program (6ms). Both these steps require loading further library classes, which are included in the JikesRVM boot image, and, therefore, take considerably more than the equivalent steps in JikesRVM (2ms and 230μs).

Preloading and pre-compiling the libraries for the application task can reduce the start-up overhead of S-RVM. However, this approach is unsuitable for the more general case of a multi-tasking virtual machine. A possible approach for this more general case is to share pre-compiled library code between tasks [Czajkowski et al., 2002]. Further research is required to check the applicability of this approach to S-RVM.

Library classes are only loaded once during a program's execution. The overhead of loading these classes does not, therefore, affect the steady-state performance of applications. The next section analyses the steady-state performance of S-RVM compared with JikesRVM.

5.3 Steady-State Execution Speed

Measuring the steady-state performance of a virtual machine is not trivial. Experimental methodologies vary between authors. See Georges et al. [2007] for a concise survey of available methodologies.

In this thesis, the performance of optimised code is measured using the DaCapo benchmark suite 2006-10-MR2. The DaCapo benchmark evaluation methodology [Blackburn et al., 2008] recommends testing performance under varying heap pressures. This thesis measures the execution times of the benchmarks over heap sizes ranging from twice the minimum heap size, as presented in Table 5.1, to ten times the minimum heap size. For example, the execution times of the **chart** benchmark are measured with heap sizes ranging from 72MB to 360MB.

The execution time of JikesRVM in each of these heap sizes is compared with that of S-RVM with the same heap size. However, because S-RVM requires more memory than JikesRVM, using the same heap size with S-RVM creates a higher heap pressure. To compensate for the overhead in S-RVM, the execution speed of JikesRVM is also compared with that of S-RVM with the heap increased by the S-RVM overhead. Thus, the execution time of the **chart** benchmark on JikesRVM with a heap size of 216MB is compared to S-RVM with heap sizes of 216MB and

225MB. The latter being the heap size used in JikesRVM with additional 9MB to compensate for the added memory requirements.

The DaCapo harness executes multiple iterations of the same benchmark in the same execution environment to allow the virtual machine to adapt to the benchmark. The default harness is modified to garbage-collect the heap before each and every iteration. This is done in order to avoid stale data of one iteration introducing garbage collection in following iterations. This seems to remove most, if not all, of the state dependency between consecutive iterations which is otherwise evident for some benchmarks [Kalibera and Jones, 2013].

To track the progress towards the steady state, the time it takes to complete each of the first 20 iterations is measured. Extending the tests to more than 20 iterations does not seem to change the mean execution time by more than 2%.

To control non-determinism, each measurement is repeated 12 times and the geometric mean of all these measurements is reported. An alternative approach for controlling non-determinism is to use replay compilation [Georges et al., 2008]. However, experimenting with compiler replay, as implemented in JikesRVM and as amended for S-RVM, demonstrates that compiler replay underperforms the adaptive compiler by 3–5%. For the purposes of this thesis, the better performing option of using adaptive compilation was chosen. The complete measurement data can be found in Appendix C.

Tables 5.2 presents the normalised execution times of S-RVM in the first, third, tenth and 20th iterations with heap sizes of twice, three times, five times and ten times the minimum heap sizes under the same heap size scenario. Table 5.3 shows the same results for the same heap pressure scenario. In each result the value of 100 represents the mean time to execute the corresponding test configuration on JikesRVM. The table also reports the geometric mean of the results for each benchmark configuration and the 90% confidence intervals.

Figure 5.3 shows the mean execution time of all the DaCapo benchmarks on S-RVM. The results are normalised to the execution time on JikesRVM, with larger numbers indicating longer execution times. Several observations can be made on this diagram. First, the relative execution time on S-RVM improves as the number of iterations increases. The normalised mean execution time at the first iteration is very low (11–14% slower than JikesRVM). This is partly due to the need to load the library classes used by the benchmark and partly because these library classes are initially compiled using the baseline compiler.

Compiling the library classes with the baseline compiler means that the compiled code is slower than the version of the library loaded in the boot image. It also mean that the optimising compiler needs to optimise more methods before S-RVM reaches its steady state.

Iteration	**Heap Size**	**antlr**	**bloat**	**chart**	**eclipse**	**fop**	**hsqldb**
1st	2X	123.6(122.4–124.7)	115.9(114.8–117.1)	108.1(107.2–109.0)	105.9(105.3–106.5)	115.6(115.1–116.1)	112.2(111.0–113.4)
	3X	121.5(120.3–122.7)	114.9(114.1–115.7)	106.7(106.0–107.4)	102.8(102.3–103.2)	117.4(116.7–118.1)	111.3(110.0–112.6)
	5X	119.2(118.2–120.2)	113.8(112.5–115.0)	104.8(104.0–105.6)	103.3(102.9–103.6)	117.7(117.0–118.5)	110.2(108.3–112.1)
	10X	107.9(107.0–108.8)	104.9(103.9–105.8)	98.6(97.8–99.4)	101.7(101.3–102.1)	105.0(104.2–105.8)	99.7(98.0–101.4)
3rd	2X	115.6(114.3–117.0)	108.8(107.4–110.1)	100.2(99.3–101.1)	102.0(101.7–102.4)	105.3(104.4–106.1)	102.9(100.5–105.4)
	3X	108.9(107.7–110.2)	104.3(102.7–105.8)	99.0(98.2–99.8)	101.7(101.4–102.0)	104.1(103.2–105.1)	103.3(101.5–105.0)
	5X	107.4(106.1–108.6)	104.8(103.5–106.1)	98.2(97.5–98.8)	101.5(101.1–101.9)	104.0(102.7–105.2)	102.0(100.1–104.0)
	10X	100.8(99.3–102.3)	105.4(104.6–106.3)	96.6(95.8–97.4)	100.6(100.5–100.8)	101.4(100.9–101.9)	99.4(97.4–101.4)
10th	2X	107.7(104.4–111.2)	108.0(105.9–110.1)	99.1(98.4–99.8)	101.4(101.2–101.7)	99.6(98.8–100.4)	99.1(97.8–100.5)
	3X	105.7(103.2–108.2)	105.7(104.4–107.1)	97.0(96.5–97.6)	101.3(101.0–101.6)	100.7(99.7–101.7)	97.8(96.0–99.7)
	5X	100.2(99.3–101.1)	106.5(105.4–107.6)	96.6(96.0–97.1)	100.8(100.3–101.2)	100.6(99.8–101.4)	101.1(99.0–103.2)
	10X	99.6(98.2–101.0)	106.6(106.0–107.3)	96.4(96.1–96.8)	99.6(99.3–99.9)	99.1(98.6–99.7)	96.2(95.1–97.4)
20th	2X	110.3(107.2–113.4)	107.3(106.0–108.5)	99.2(98.6–99.8)	101.3(100.8–101.8)	100.5(100.0–101.0)	96.3(95.2–97.4)
	3X	101.2(99.0–103.6)	106.1(104.9–107.3)	96.8(96.2–97.4)	101.0(100.7–101.2)	98.7(97.4–100.0)	96.4(94.7–98.2)
	5X	101.0(100.1–102.0)	106.1(105.4–106.8)	96.5(96.0–97.0)	100.8(100.5–101.1)	99.0(98.1–99.9)	94.4(93.5–95.4)
	10X	99.6(98.2–101.0)	106.6(106.0–107.3)	96.4(96.1–96.8)	99.6(99.3–99.9)	99.1(98.6–99.7)	96.2(95.1–97.4)

Iteration	**Heap Size**	**jython**	**luindex**	**lusearch**	**pmd**	**xalan**	**mean**
1st	2X	113.8(112.5–115.1)	114.4(113.8–115.1)	116.0(107.7–125.0)	118.2(116.4–120.1)	117.8(117.3–118.4)	**114.6(112.1–117.2)**
	3X	111.2(109.8–112.7)	109.6(108.8–110.3)	110.4(105.4–115.6)	116.5(115.3–117.7)	115.1(114.1–116.0)	**112.4(110.6–114.1)**
	5X	109.9(109.1–110.7)	109.9(109.0–110.8)	112.9(105.7–120.5)	113.0(112.1–114.0)	112.6(111.6–113.6)	**111.5(109.2–113.8)**
	10X	100.1(98.9–101.3)	101.1(100.4–101.9)	119.5(108.6–131.5)	101.5(100.7–102.2)	101.9(101.1–102.8)	**103.7(100.7–106.7)**
3rd	2X	104.9(103.9–105.9)	105.6(104.8–106.4)	130.8(125.3–136.6)	107.3(105.9–108.6)	112.5(111.5–113.5)	**108.4(106.7–110.2)**
	3X	100.2(99.6–100.9)	102.8(102.2–103.3)	109.4(108.2–110.6)	106.7(104.5–109.1)	106.4(104.8–108.0)	**104.2(103.0–105.4)**
	5X	100.8(99.8–101.9)	101.5(101.1–101.9)	111.5(109.2–113.8)	100.6(99.6–101.5)	102.1(101.5–102.7)	**103.1(101.9–104.2)**
	10X	99.9(99.1–100.8)	99.8(99.2–100.3)	101.8(100.4–103.2)	97.9(97.2–98.6)	98.9(98.1–99.7)	**100.2(99.2–101.2)**
10th	2X	103.0(101.9–104.1)	102.7(101.9–103.6)	108.9(107.3–110.5)	108.5(107.1–110.0)	116.1(114.2–118.0)	**104.8(103.3–106.3)**
	3X	100.6(99.8–101.4)	100.1(99.4–100.8)	100.6(99.7–101.5)	99.2(97.2–101.2)	104.0(102.6–105.5)	**101.1(99.8–102.4)**
	5X	100.4(99.1–101.6)	98.7(98.2–99.3)	101.7(100.7–102.8)	100.0(99.2–100.8)	100.7(99.7–101.7)	**100.6(99.6–101.6)**
	10X	99.3(98.7–100.0)	99.5(98.9–100.2)	102.5(101.5–103.5)	98.7(98.1–99.4)	99.5(98.4–100.6)	**99.7(98.9–100.5)**
20th	2X	101.2(99.8–102.6)	101.8(101.1–102.4)	108.3(107.2–109.3)	106.1(104.8–107.5)	112.3(110.3–114.4)	**103.9(102.6–105.2)**
	3X	99.9(99.0–100.8)	100.2(99.6–100.8)	100.2(99.6–100.7)	100.3(98.4–102.1)	101.8(99.9–103.7)	**100.2(98.9–101.5)**
	5X	99.1(98.0–100.2)	98.6(97.9–99.2)	101.5(100.3–102.7)	99.3(98.3–100.4)	98.4(97.2–99.7)	**99.5(98.6–100.4)**
	10X	99.3(98.7–100.0)	99.5(98.9–100.2)	102.5(101.5–103.5)	98.7(98.1–99.4)	99.5(98.4–100.6)	**99.7(98.9–100.5)**

Table 5.2: Normalised execution times of the DaCapo benchmarks on S-RVM relative to JikesRVM with 90% confidence intervals (Same Heap Size scenario)

Iteration	Heap Size	antlr	bloat	chart	eclipse	fop	hsqldb
1st	2X	119.4(118.2–120.5)	115.8(114.5–117.2)	105.7(104.5–106.9)	104.9(104.4–105.5)	117.0(116.3–117.7)	112.3(111.1–113.4)
	3X	120.1(119.1–121.1)	115.1(113.9–116.3)	105.4(104.7–106.1)	103.6(103.1–104.0)	117.7(116.9–118.6)	111.5(110.4–112.7)
	5X	118.8(117.6–120.0)	114.1(113.2–115.0)	105.2(104.3–106.1)	103.2(102.9–103.5)	117.8(117.5–118.2)	111.2(110.0–112.3)
	10X	106.1(104.6–107.7)	105.1(104.2–106.1)	98.3(97.5–99.1)	101.5(101.1–101.8)	104.4(103.5–105.3)	100.3(98.3–102.3)
3rd	2X	109.5(107.5–111.6)	107.3(105.3–109.3)	98.4(97.7–99.2)	101.8(101.4–102.2)	102.7(101.9–103.4)	104.4(102.6–106.2)
	3X	107.6(106.5–108.8)	104.4(103.5–105.2)	98.6(97.9–99.3)	101.2(100.7–101.6)	104.1(103.2–105.0)	99.5(97.5–101.6)
	5X	108.8(107.7–110.0)	105.4(104.3–106.6)	98.1(97.4–98.9)	101.4(101.1–101.6)	103.5(102.4–104.7)	99.8(98.0–101.7)
	10X	100.1(98.7–101.5)	106.0(104.9–107.0)	96.6(95.9–97.3)	100.6(100.3–100.8)	101.9(101.3–102.5)	98.1(96.8–99.5)
10th	2X	98.1(94.8–101.5)	107.0(104.9–109.0)	98.0(97.2–98.7)	100.8(100.5–101.1)	97.8(96.6–99.1)	97.6(95.7–99.6)
	3X	101.5(99.6–103.5)	106.6(105.3–107.9)	97.3(96.4–98.2)	100.9(100.6–101.2)	100.3(99.7–100.9)	96.4(94.5–98.4)
	5X	101.0(99.6–102.5)	107.2(106.2–108.3)	96.6(95.8–97.4)	100.7(100.3–101.0)	100.6(99.8–101.4)	99.4(96.4–102.6)
	10X	99.0(98.2–99.9)	105.3(104.1–106.5)	96.0(95.5–96.5)	99.7(99.5–99.9)	99.6(98.6–100.6)	95.5(93.7–97.3)
20th	2X	100.6(98.1–103.2)	107.0(105.0–109.1)	97.0(96.6–97.4)	100.6(100.4–100.8)	100.0(98.4–101.6)	95.8(94.2–97.5)
	3X	98.6(97.2–100.1)	107.1(105.9–108.3)	96.3(96.1–96.6)	100.7(100.4–100.9)	98.8(98.2–99.5)	96.7(95.5–98.0)
	5X	100.3(99.0–101.6)	107.2(105.9–108.4)	96.7(96.3–97.1)	100.3(100.1–100.5)	99.5(98.7–100.3)	94.6(92.9–96.2)
	10X	99.0(98.2–99.9)	105.3(104.1–106.5)	96.0(95.5–96.5)	99.7(99.5–99.9)	99.6(98.6–100.6)	95.5(93.7–97.3)

Iteration	Heap Size	jython	luindex	lusearch	pmd	xalan	mean
1st	2X	112.0(111.1–112.9)	111.9(111.1–112.6)	109.3(107.4–111.3)	113.1(111.4–114.7)	112.9(112.2–113.6)	**112.1(111.0–113.3)**
	3X	110.0(108.8–111.3)	109.8(109.0–110.6)	106.9(104.7–109.2)	114.0(112.4–115.6)	113.2(112.3–114.1)	**111.5(110.3–112.6)**
	5X	110.6(109.2–111.9)	109.7(108.9–110.6)	110.7(105.2–116.5)	112.4(111.7–113.1)	112.0(111.0–113.0)	**111.3(109.5–113.2)**
	10X	99.9(99.2–100.7)	101.1(100.6–101.7)	110.5(108.4–112.6)	101.0(99.7–102.4)	103.3(102.6–104.1)	**102.8(101.7–104.0)**
3rd	2X	101.8(101.2–102.3)	102.5(102.0–103.1)	121.1(117.0–125.2)	102.8(100.3–105.3)	106.3(105.3–107.3)	**105.2(103.5–106.8)**
	3X	100.4(99.4–101.4)	101.6(101.0–102.3)	111.2(109.5–113.0)	103.4(101.5–105.4)	105.2(104.3–106.1)	**103.3(102.1–104.5)**
	5X	100.4(99.8–101.1)	101.8(101.2–102.4)	109.0(107.5–110.6)	99.4(98.2–100.7)	102.5(101.2–103.8)	**102.7(101.6–103.8)**
	10X	99.8(99.3–100.3)	99.5(99.0–100.0)	100.4(99.7–101.1)	99.0(98.2–99.7)	98.8(98.0–99.7)	**100.0(99.2–100.9)**
10th	2X	100.1(99.2–100.9)	99.6(99.1–100.1)	102.4(100.7–104.1)	101.7(99.8–103.7)	107.5(105.6–109.4)	**100.9(99.3–102.6)**
	3X	100.4(99.4–101.4)	98.7(98.1–99.3)	100.5(99.4–101.5)	100.5(98.2–102.8)	103.6(102.6–104.6)	**100.6(99.3–101.8)**
	5X	100.4(99.5–101.2)	98.7(98.3–99.1)	100.6(100.0–101.2)	100.9(100.1–101.7)	101.5(100.0–102.9)	**100.7(99.4–101.9)**
	10X	98.6(97.9–99.4)	98.9(98.3–99.6)	101.9(101.1–102.6)	98.8(98.1–99.6)	98.6(97.9–99.3)	**99.2(98.4–100.1)**
20th	2X	98.6(97.5–99.7)	99.5(98.8–100.2)	101.9(101.1–102.7)	101.4(99.7–103.1)	103.0(101.8–104.3)	**100.4(99.1–101.8)**
	3X	98.9(98.0–99.8)	98.6(97.9–99.4)	100.6(100.0–101.3)	99.9(97.5–102.4)	100.6(99.7–101.4)	**99.7(98.6–100.8)**
	5X	99.1(98.6–99.7)	98.1(97.7–98.5)	101.1(100.5–101.7)	99.4(98.4–100.5)	98.9(97.9–99.8)	**99.5(98.6–100.4)**
	10X	98.6(97.9–99.4)	98.9(98.3–99.6)	101.9(101.1–102.6)	98.8(98.1–99.6)	98.6(97.9–99.3)	**99.2(98.4–100.1)**

Table 5.3: Normalised execution times of the DaCapo benchmarks on S-RVM relative to JikesRVM with 90% confidence intervals (Same Heap Pressure scenario)

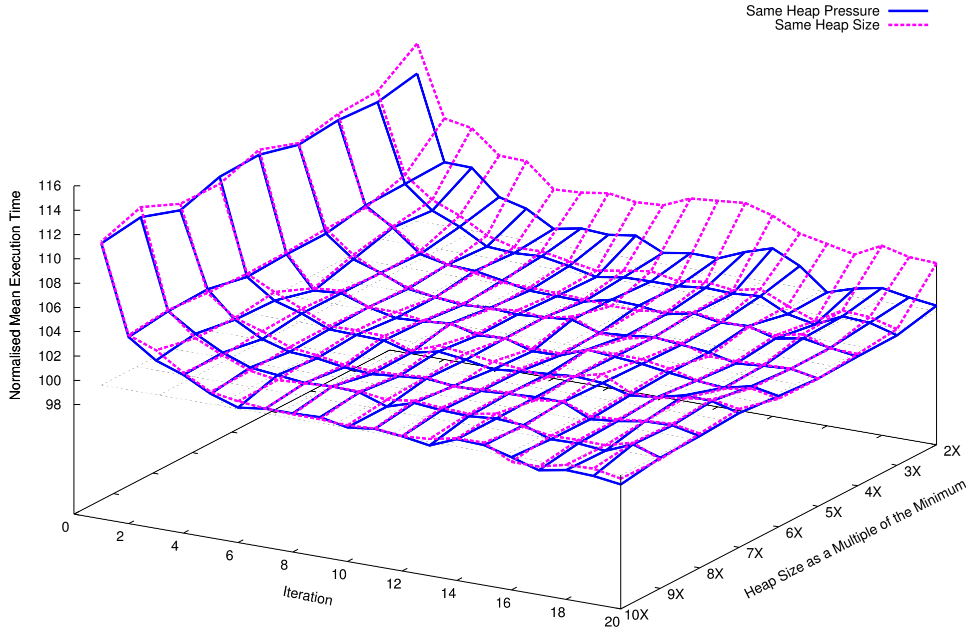

Figure 5.3: Mean execution time on S-RVM relative to JikesRVM.

A second observation is that for the same heap size scenario, the performance of S-RVM improves as the size of the heap increases. The explanation is straightforward. The memory overhead of S-RVM is the same for small and large heaps. For small heaps, the same overhead leaves relatively less heap space for application objects. For larger heap sizes, the overhead is minor relative to the total heap size, resulting in execution times similar to those of the same heap pressure scenario.

The third observation is that at steady state S-RVM marginally outperforms JikesRVM. Figure 5.4 further highlights this observation by showing the relative execution time of each of the benchmarks at steady state, including the 90% confidence interval. As the graph demonstrates, when the heap is large enough, S-RVM outperforms JikesRVM on most of the DaCapo benchmarks. However, while the mean execution time on S-RVM is slightly better than on JikesRVM, the difference is not statistically significant.

Due to the need to support multiple tasks, the implementation of S-RVM is less efficient than that of JikesRVM. In particular, optimisations that assume that every object is assignment compatible with the Java `Object` type are disabled and additional indirections are introduced in the data structures that represent types. Based on this information, one would expect S-RVM to underperform JikesRVM. Consequently, the slight performance improvement, albeit not being significant, is surprising.

This better performance is a consequence of S-RVM use of separate copies of

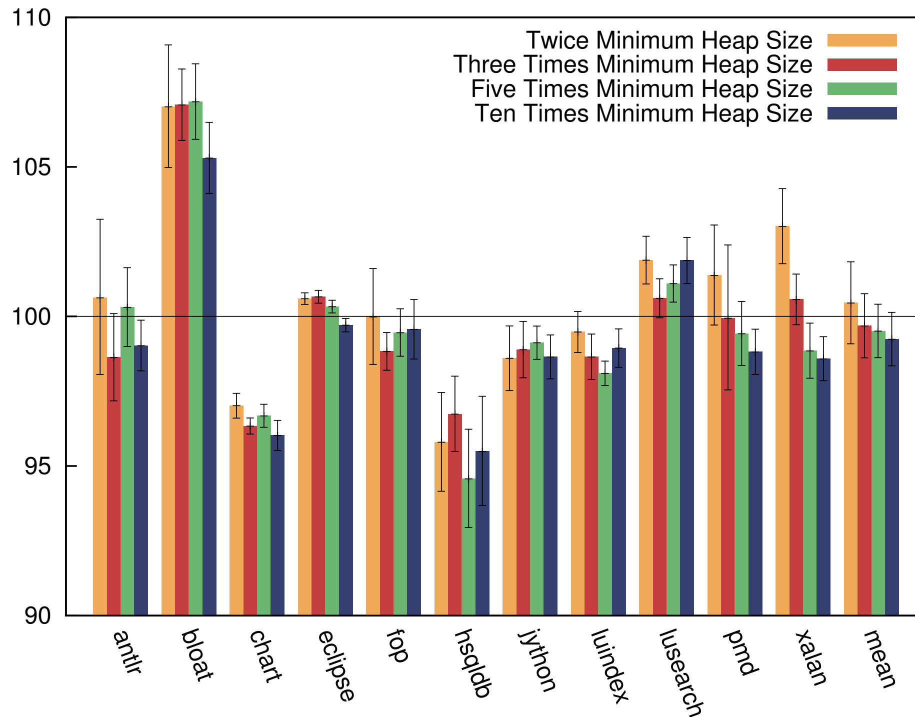

Figure 5.4: Steady state normalised execution times of the DaCapo benchmarks on S-RVM at several heap sizes

the Java libraries for the virtual machine and for the application. These copies are optimised for different workloads, with the virtual machine copy being optimised for the virtual machine workload and the application copy being optimised for the application workload.

The virtual machine and the application generate different workload on library classes. S-RVM, like JikesRVM, collects profiling information on a per-method basis. This profiling information is used for directing the optimising compiler. S-RVM uses a separate copy of library methods in each task. Profiling information for each of these copies matches the workload generated by the task.

In JikesRVM, on the other hand, Java library methods are shared between the virtual machine and the application. Consequently, the profiling information collected for these methods is combined and is less representative of the use by either the virtual machine or the application. Optimising code based on the combined profiling information produces a compromise between workloads resulting in lower than optimal performance.

The effect of the combined profiling information can be demonstrated by forcing S-RVM to combine the profiling information of library methods in the virtual machine and the application. Figure 5.5 shows the results of executing the DaCapo benchmarks on a version of S-RVM that combines edge-count profiling information. (Edge-count information measures the frequencies of taking conditional branches.

It is used *inter alia* in decisions on code reordering to avoid branches in the common case and in inlining decisions.)

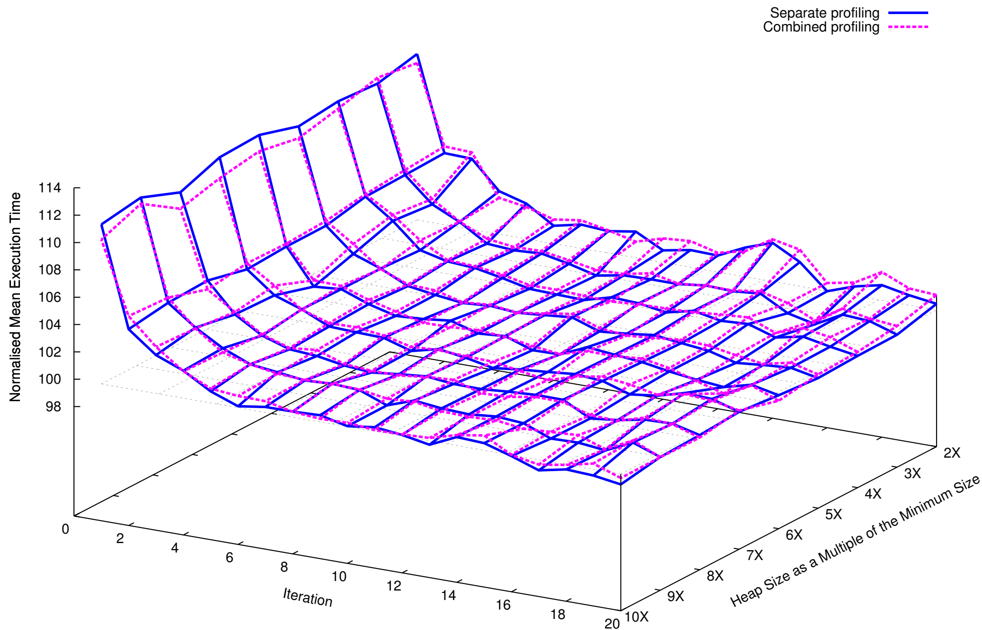

Figure 5.5: Separate vs. combined profiling

While only part of the profiling information is combined, the diagram demonstrates a slight slowdown of about 0.2%. This slowdown confirms the hypothesis that the speedup in S-RVM is the result of the profile information better representing the different workloads. It should be noted, however, that this slowdown, like the performance improvement of S-RVM, is not statistically significant.

5.4 Summary

S-RVM increases the security of JikesRVM by separating the virtual machine from the application. This chapter discusses the costs of the better security provided by S-RVM.

As demonstrated, S-RVM does not incur any cost in the steady state. Due to the need to load the Java library, program start up in S-RVM is slower than in JikesRVM. The two copies of the library also result in a larger memory footprint.

S-RVM is the first virtual machine to provide reference protection at no performance loss. This achievement is the result of its use of the Exported Types design for providing reference protection. As such, S-RVM demonstrates the effectiveness of the Exported Types design.

Chapter 6

Summary and Conclusions

Isolating components increases the safety and security of the system by restricting the way misbehaving components can affect other components. This thesis explores the area of software-based component isolation. It focuses on finding the right balance between effective protection and efficient communication between isolated components.

Work included in this thesis encompasses three main areas: the classification of of systems providing software-based component isolation, a design of a type system that supports software-based component isolation and the implementation of the type system.

This chapter describes the main findings of this work in these areas.

6.1 Classification Framework

The first area of work in this research is the classification of software-based component isolation. The classification is centred around the concept of zones, which are groups of objects to which protection is applied uniformly. It identifies a small number of zone types based on the permitted inter-zone references and classifies systems according to the zone types they use.

The benefit of the classification is that it defines the problem space and provides a context to building reference-protection systems. The main strength of the classification is that it captures the level of isolation between components. Consequently, it identifies possible sharing of data and permitted inter-component method invocation. Thus, the classification identifies the potential use of memory access for direct communication between otherwise isolated components.

Four levels of isolation are identified. Complete isolation provides the highest level of isolation where the system guarantees that memory accessible by one component is not accessible by others. Hence, in systems that provide complete isolation, components cannot use the main memory for communication.

The second level of isolation, the object sharing protection model, allows multiple components to access the same data. It does not, however, support direct transfer of control between components. Additional protocols are required for implementing cross-component procedure calls and other mechanisms for transferring control.

Cross-component procedure calls are supported by the partial isolation protection model. In systems supporting partial isolation each component consists of a sealed zone which is private to the component and an interface zone which is accessible by other components. Code in the interface zone is granted access to the sealed zone and can be used to provide other components with indirect access to the data in the sealed zone.

Initial isolation is a protection model that supports the least level of isolation. Under this model, components are isolated when created, but they can share references with other components. Once references are shared, the system does not provide any guarantees regarding the extent of sharing.

The classification has proven its usefulness by identifying that partial isolation has not been explored in conjunction with compile-time enforcement of reference protection. The partial isolation model maps well to a typical component model in which each component consists of private data and an RPC interface. The interface is used for communication between components whereas the private data is not accessible outside the component. Under the mapping, each component consists of a sealed zone and an interface zone. The interface zone is used for the interface and the sealed zone for the private data. Reference protection ensures that only objects within the component can hold references to private data.

The partial isolation model, therefore, promises to deliver the level of isolation expected for components while allowing the sharing and method invocation that components support. Yet, the only prior work to implement this model for component isolation used costly run-time enforcement of the model. The next section describes Exported Types, which bridges this gap.

6.2 The Exported Types Design

Exported Types is a type system design that supports reference protection with the partial isolation model. Supporting reference protection in the type system allows the compiler to enforce the restrictions on reference propagation, avoiding run-time costs. The partial isolation model allows components to directly access the interface of other components, minimising the inter-component communication costs.

In a nutshell, the Exported Types design separates components by creating a type name space for each components. Components can export and import types. Exporting publishes the type so that other components can import it. Importing

gives the components access to the type and, therefore, to objects of the type. These exported types form the interfaces for inter-component communication.

The Exported Types design only covers the aspects of the type system required for overlaying the partial isolation model over the type hierarchy. It leaves other details of the type system to be specified by the designer. By only providing those features essential for reference protection, the Exported Types design can be used both as a basis for a type system of a new language and as an extension to an existing language.

By insisting on explicit import and export operations, the design follows the principle of fail-safe defaults [Saltzer and Schroeder, 1975]. That is, cross-component access to objects is only allowed if explicitly granted. Explicit operations also help in achieving backward compatibility when extending existing languages—they ensure that legacy code is not affected by the new semantics of the Exported Types design.

While the design can be used as an extension to existing languages, various features of the language's type system may conflict with the Exported Types design. The biggest limitation of the Exported Types design is that it is suitable only for type systems that use name-based type equivalence. It is not applicable for use with structure-based type equivalence. Language features that use structure-based type equivalence, e.g. Java arrays, conflict with the Exported Types design. Other language features, such as implicit operations on types and special semantics of some types, also cause problems. A common solution to many of these incompatibilities is to use wrapper classes. These wrapper classes, however, introduce a level of indirection which might affect performance.

Notwithstanding the conflicts with language features, the Exported Types design can be applied to existing languages. The third area of work in of this thesis is an implementation of Exported Types in Java. This implementation is the topic of the next section.

6.3 S-RVM

S-RVM is a Java virtual machine based on JikesRVM. The main motivation for implementing S-RVM is as a proof of implementation for the Exported Types design. It uses the Exported Types design to separate the virtual machine from the application in a meta-circular virtual machine. The main advantage of S-RVM over JikesRVM is that its design is more secure because it reduces the attack surface of the virtual machine. This increase in security is delivered at no cost to the steady-state performance of S-RVM.

S-RVM only addresses the large attack-surface area of JikesRVM. It does not, however, address many other security issues in JikesRVM. Consequently, S-RVM

is not secure. Nevertheless, by addressing one of the main security weaknesses of JikesRVM, S-RVM is a step forward towards a secure meta-circular virtual machine.

Using the Exported Types design in this specific scenario is a mixed blessing. On the one hand, the interface between the virtual machine and the application is complex and it is used frequently during a program execution. It is used, for example, for object allocation, synchronisation, cloning and reflection. This scenario, therefore, is useful in demonstrating that the costs of inter-component communication with the Exported Types design are negligible.

On the other hand, meta-circular machines are, admittedly, a rather specific context for using the Exported Types design. Applying the design in a richer scenario, such as for implementing a multi-tasking virtual machine, may be a more convincing example of its usefulness. Yet, by separating the application from the virtual machine and by implementing the type export and import mechanisms, S-RVM lays the foundation required for implementing the design in a more general context.

One incidental benefit of S-RVM is its marginally, albeit not statistically significant, improved steady-state performance. This improvement is the consequence of having two copies of the Java libraries which are optimised independently of each other. The library code for the virtual machine is optimised for the virtual machine workload whereas the application copy of the library is optimised for the application workload.

Duplicating library code also has some downsides. The memory footprint, including both the boot image and the heap, of S-RVM is about 10MB larger than that of JikesRVM. While 10MB is a small amount of memory in most modern architectures, it can still be a significant size in some environment, e.g. embedded machines. Furthermore, since the application's copy of the library is not preloaded in the S-RVM boot image, applications take longer to start and take longer to achieve the steady state than in JikesRVM.

6.4 Revisiting the Classification Framework

S-RVM is a proof-of-implementation of the Exported Types design. This thesis develops the Exported Types design to bridge a gap identified by the classification presented in Chapter 2. This section applies the classification to S-RVM and demonstrates that it, indeed, bridges the gap.

The first step in applying the classification is to identify the zones used in the system. Table 2.2 presents the zones used in the systems covered by Chapter 2. This table is presented here again as Table 6.1, amended with the information about S-RVM.

Through the use of unboxed wrapper classes, the root task in S-RVM can access

System	Zone	Zone Type
CoLoRs	Process heaps Shared region	Isolated Shared
Confined Types	Objects of confined types Objects in the package	Sealed Interface
J-Kernel	Domain Capabilities Domain objects	Interface Sealed
JNode	Root Isolate Application Isolates	Privileged Shared Isolated
JX	Domains	Isolated
KaffeOS	Kernel Heap Shared Heaps Process Heaps	Privileged Shared Shared Isolated
MVM	Isolates	Isolated
.Net	Application Domain	Isolated
OVM	User Domains Executive Domain	Isolated Privileged Isolated
Ownership Types	Owning objects Owned objects	Interface Sealed
Real-time Java	Regions	Shared
Rust	Processes Shared Values	Isolated Shared
Singularity	Process Exchange heap	Isolated Shared
S-RVM	Application objects Objects of exported root classes Other root task objects	Sealed Privileged Interface Privileged Sealed
XMem	Process heaps Shared region	Isolated Shared

Table 6.1: Zones in systems providing reference protection

every object in the system. Hence, zones within the root task are privileged. A zone is defined as a group of objects to which protection rules are applied uniformly. Based on this definition, S-RVM has four zones: objects of classes in the hierarchy of the application task, objects of root task classes that are not exported, objects of exported classes and objects of application-extended exported classes. It should be noted that by subtyping, objects of the last zone are also objects of exported classes. That is, the zones overlap and do not form a partition of the object space.

Objects of non-exported root classes may only be referenced by objects within the root task. Thus, these objects form a sealed zone, with objects of exported classes (and their subclasses) being the corresponding interface zone. Similarly, when ignoring the special rules for privileged zones, objects in the application task hierarchy form a sealed zone, with the objects of exported classes that the application extends being in the corresponding interface zone.

With the zone type identified, the reference protection and the reachability and propagation control mechanisms can be identified. Tables 6.2 and 6.3 show where S-RVM fits with respect to the system surveyed in Chapter 2.

System	**Run Time Model**	**Reachability Control**	**Propagation Control**
CoLoRs	Object Sharing	Multiple Types	Write Barriers
Confined Types	Partial Isolation	Shared	Sealed Types
J-Kernel	Partial Isolation	Multiple Types	Transfer Barriers
JNode	Complete Isolation	Indirection	Implicit
JX	Complete Isolation	Multiple Types	Implicit
KaffeOS	Object Sharing	Shared Multiple Types	Write Barriers
MVM	Complete Isolation	Indirection	Implicit
.Net	Complete Isolation	Indirection Separate Compilation	Implicit
OVM	Complete Isolation	Multiple Types	Implicit
Ownership Types	Partial Isolation	Shared	Sealed Types
Real-time Java	Object Sharing	Shared	Write Barriers
Rust	Object Sharing	Ownership	Shared Types
Singularity	Object Sharing	Multiple Types	Shared Types
S-RVM	Partial Isolation	Multiple Types	Sealed Types
XMem	Object Sharing	Separate Compilation	Write Barriers

Table 6.2: Classification of reference protection

Systems that use sealed zones and their corresponding interface zones offer the partial isolation reference-protection model. This is, therefore, the protection model offered by S-RVM. As S-RVM uses type information to decide whether an object is within a sealed zone, the propagation control mechanism it offers is sealed types.

S-RVM avoids sharing reachability roots. As the type hierarchies are separate, reachability roots in one task are not accessible from another. S-RVM only allows method members in the remote interface, never field members. This prevents direct access from code declared in one task to static fields declared by another, preventing shared access to these fields.

As it uses sealed types to control reference propagation, S-RVM provides partial isolation enforced at compile time (Table 6.3). Unlike the other systems that fits this description, S-RVM avoid shared reachability roots and is, therefore, suitable for component isolation. Hence, S-RVM bridges the gap identified in the survey presented in Chapter 2. Furthermore, as Chapter 5 demonstrates, the added security of reference protection comes at no performance cost. Thus, S-RVM fulfils the promise of achieving high performance with the added security of reference protection.

While S-RVM fills the gap, it only supports two components. Thus, S-RVM falls short of providing a generic component-based environment. Extending S-RVM to

	Run Time	Compile Time
Complete Isolation	JNode JX MVM .Net OVM	
Object Sharing	CoLoRs KaffeOS XMem Real-time Java	Rust Singularity
Partial Isolation	J-Kernel	Confined Types Ownership Types S-RVM

Table 6.3: Classification of reference protection

support multiple tasks would overcome this shortcoming. While this extension is beyond the scope of this thesis some initial directions towards it are presented in the next section.

6.5 A Multi-tasking Virtual Machine

While S-RVM is not a multi-tasking virtual machine, there are no arbitrary limitations in its implementation that prevent it from being extended to support multiple application tasks. The VM interface layer can support multiple tasks, as displayed in Figure 6.1. The trusted application running within the root task, which in S-RVM creates the application task object and initialises it, can be modified to create multiple tasks.

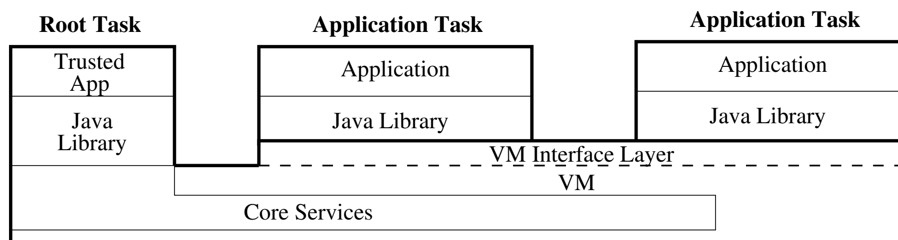

Figure 6.1: A multi-tasking virtual machine

Supporting the creation of multiple tasks is but one aspect of a multi-tasking virtual machine. Several areas in S-RVM need more work for the system to become a fully-fledged multi-tasking virtual machine.

Currently, only the root task exports classes. An application task (assuming multiple tasks are created) cannot export classes. To support application exporting classes, the design of the globally shared name space will have to change to accommodate for potential race conditions in allocating exported type names to

tasks.

A related issue is the lack of an interface for managing tasks. Currently, only the trusted application can create tasks and it does not have an easy mechanism for managing these tasks. For a system that runs only one task, this model is sufficient. However, in a multi-tasking virtual machine, tasks should be able to create, manage and destroy other tasks. This raises the issue of authorisation, e.g. deciding what permissions a task has for managing other tasks. It also raises the issue of the semantics of task termination. In particular, the design needs to take into account the possibility of one task terminating while another holds references to its objects and decide how this situation is handled.

Task termination also highlights another problem. S-RVM, like JikesRVM, does not currently support class unloading. Classes in S-RVM (and in JikesRVM) stay loaded for as long as the virtual machine executes. When there is only one application in the virtual machine, this issue mainly affect applications that dynamically create class loaders. In a multi-tasking system, where tasks are created dynamically, each terminated task will leave behind a large set of classes that will not be used. This memory leak limits the number of tasks that can be created over the running time of the virtual machine.

Another area for further research is the use of partial isolation on type systems that employ structure-based type equivalence. This is discussed in the next section.

6.6 Structure-based Type Equivalence

As Chapter 3 discusses, any type system that provides reference protection must be at least partially name-based. The Exported Types design relies on name-based type equivalence for its operation to the extent that it is incompatible with features of the underlying type system that use structure-based equivalence. As such, this thesis does not try to evaluate the extent to which a type system needs to be name-based to support reference protection.

To increase the understanding of the relationship between reference protection and type systems it may be useful to question whether component isolation can be implemented with type systems that predominantly use structure-based equivalence.

A possible direction for investigating this is to build upon tagging references as implemented in ownership types [Clarke, 2001, Clarke et al., 1998]. Ownership types use tags on references to indicate the zone of the objects they refer to. Using a similar technique, it may be possible to support structural equivalence within zones, limiting the use of name-based equivalence to the specification of the zones themselves.

6.7 Summary

This thesis makes several contributions to the area of reference protection. The first is the development of a novel framework for classifying reference-protection systems. The classification captures the salient features of reference protection and is, therefore, useful for comparing the properties of systems from across the domain.

The second and the most significant contribution of this thesis is the Exported Types design. Exported Types is the first type system design to support the partial isolation reference-protection model and is the first to provide component isolation at no performance costs. The design is complemented by an implementation which demonstrates that it is applicable to modern mainstream languages and that it supports full backwards compatibility.

The implementation of S-RVM is a contribution not only because it proves the feasibility of the Exported Types design. S-RVM explores the interface between the virtual machine and the application in meta-circular virtual machines. It increases the security of the system by clearly defining the boundaries between the two and by ensuring that the application is restricted to only use that boundary. The clear boundaries between the virtual machine and the application also makes S-RVM an excellent experimentation platform when code instrumentation is required. By separating the application from the virtual machine, S-RVM allows instrumentation of application code without the circular dependencies that result from instrumenting virtual machine code. Ideas embodied in this aspect of S-RVM have contributed towards work on library code cloning [Bond, 2013].

The last contribution of S-RVM is that it highlights the benefit of separate profiling for different workloads. Separate profiling of library code in S-RVM allows it to achieve a performance marginally better than that of the baseline JikesRVM, even though the S-RVM code is more complex than that of the baseline.

This thesis set out to explore the area of reference protection for component isolation. The specific question it addresses is whether the performance cost typically associated with component isolation can be eliminated. To answer this question, this thesis maps the research area, identifies a plausible hitherto unexplored system model, designs a system based on this model and implements the design. The implemented system demonstrates that, with the correct design, components in a system can be isolated with no performance loss. It is hoped that future work will build upon ideas developed in this thesis to provide effective and efficient software-based isolation mechanisms.

Appendix A

Exported Types Specifications

Chapter 3 presents an informal description of the Exported Types design. This appendix augments the description with a semi-formal specification of the type system. It starts with a definition of the relevant properties of type systems. It then describes the Exported Types design as a set of modification to an arbitrary type system. The last section presents the mapping of values to zones and argues that the mapping guarantees the security properties of the partial isolation reference-protection model.

A.1 Type Systems

Programs are sets of instructions for manipulating data values, specified by a programming language. The *type system* is a part of the programming language that superimposes a structure of sets on the value space of the program. Each such set corresponds to a *type*. The main purposes of a type system are to declare the structure of data values, creating a model of the objects these values represent, and to provide semantics to the values of the types by restricting the operations a program can apply on data values to those operations sanctioned by the values' types.

Type systems typically include two kinds of types. *Primitive types* are basic types that exist *a priori* and form the building blocks for creating other types. *Composite types* are types that are created using a *type algebra* by combining primitive and other composite types.

To allow the program to refer to types, the type system provides a *type-resolution* mechanism that matches textual type names with the types they refer to. To avoid name conflicts between different sections of the program, some type systems partition the program code into multiple *typing contexts*. Type names are unique within each typing context. However, the same type name may refer to different types in different typing contexts.

Types in a type system are ordered by the *subtype* relation, which is the type

systems' nomenclature for the subset relation. Subtyping allows programmers to benefit from inclusion polymorphism [Cardelli and Wegner, 1985] by re-using code for processing multiple similar or related values that share part of their semantics.

When a data value is created it is associated with an *implementation type*, that describes the structure of the value. Through subtyping, the value may be in the value set of multiple types. A value in the value set of a type is said to *have* that type. The Exported Types design assumes that the implementation type of a value, as well as the set of types the value has, do not change during the lifetime of the value. It further assumes that the implementation type of a value is the minimal type that the value has. That is, the implementation type of a value is a subtype of any other type the value has.

In addition to managing the types, the type system also manages a set of rules for accessing type members. The rules specify the members of types that are accessible at each code location. Typically, the specification associates with each code location a set of *scopes* which are determined based on the relationship between the code location and the types declarations. Access to members of the type is determined based on the scope of the accessing code and the permissions set for the member.

Thus, a type system is determined by four elements: the primitive types, the type algebra, the type-resolution process and the access-rules semantics. The next section describes how the Exported Types design extends a type system to support components.

A.2 Exported Types

In its core, the Exported Types design is a set of modifications to an underlying type system that extends the type system to support the partial isolation protection model. That is, given a type system for a language, Exported Types specifies how to modify its type algebra, type-resolution process and the access-rules semantics to support multiple components.

The main concept that Exported Types adds to the underlying type system is a *component*. Components are software packages that implement some functionality. The Exported Types design extends the concept to the type system.

The Exported Types design partitions the type hierarchy along component boundaries. Each component is associated with a part of the type hierarchy such that all the supertypes and all the subtypes of a type are in the same component as the type. The type algebra is restricted to maintain this property by not allowing the creation of types unless all their supertypes and subtypes are in the same component. These restrictions apply both for type extensions and to declaration of union and intersection types [Cardelli and Wegner, 1985].

For each type, the Exported Type design also tracks the type's declaring component. The declaring component of a type is the component that contains the code that declares the type. It should be noted that components can declare subtypes of types from other components. Hence, the declaring component of a type is not necessarily the same as the component the type is associated with.

Tracking the declaring component means that the type identity is dependent on its declaration. Consequently, irrespective of the underlying type system, Exported Types is not a structural type system [Albano et al., 1989].

The Exported Types design adds two operations to the type algebra of the underlying type system. These operations handle exporting types, i.e. making them available outside a component, and importing types, i.e. using types exported by other components.

Exporting and importing use a *globally shared name space* that maps names to exported types. The *export* operation associates a type with a name in the globally shared name space. The arguments for the operation are a reference to the type and the desired name. Depending on the implementation, the name may be implicit. For example, the S-RVM implementation of the Exported Types, uses the Java type name when exporting types.

The *import* operation associates a previously exported type, identified by the name used for exporting it, with a local name within a typing context in the importing component. For each typing context, Exported Types maintains a list of imported type names and the imported types they map to.

The imported-types list is used during the type type-resolution process. When resolving a type name within a typing context, the first step is to check if the name appears in the imported-types list. Only if a match is not found, the type-resolution process proceeds to use the type-resolution process of the underlying type system. To ensure the consistency of type references within a typing context, importing a type must take place before the first resolution of the type name within the typing context.

Primitive types are considered to be part of a special *system component*. They are implicitly exported by the system component and are implicitly imported in all typing contexts. The design assumes that values of primitive types are immutable. Note that this assumption does not imply that variables of primitive types are immutable.

The definition of scopes in the underlying type system is limited to the declaring component of a type. Hence, access to members of a type is only allowed to code within the component that declares the type. To allow access from other components, the Exported Types design adds a *remote* annotation. Type members annotated as remote members can be accessed by code outside the declaring com-

ponent. However, the access rules of the underlying type system still apply. For example, remote members with a `protected` scope can only be accessed by code in subtypes of the exported types.

Since non-exported types are not accessible outside a component, the remote annotation is only relevant to members of exported types. Further restrictions on remote members are that their types must be explicitly declared, i.e. the system cannot rely on type inference for determining the types of remote members, and that types used in creating the remote member signatures must all be exported. These restrictions are required to ensure that the types used on the interface are exported and that their names have the same meaning in both the exporting and the importing components.

A.3 Mapping to zones

This section demonstrates that the Exported Types design implements the partial isolation reference-protection model. Under partial isolation each component comprises two zones: a sealed zone, where private data is stored, and an interface zone used for communication with other components.

The mapping of values to components is based on the declaring components of the types the values have. If any of the types a value has is declared in a component, the value is part of that component.

A value is in the sealed zone of a component if the implementation type of the value is *internal* to the component. A type is internal to a component if neither the type nor any of its supertypes are exported. If the implementation type of a value or any of its supertypes are exported, the value is in the interface zone of a component.

To demonstrate that the design maintains partial protection it is sufficient to show that components cannot hold references to values in the sealed zones of other components. Generally speaking, components can obtain references to values either through instantiating the value or by receiving the reference from another component through the remote interface.

To instantiate a value, the instantiating component needs to specify the implementation type of the value. The type-resolution process in a component allows the component to name types it defines and types it imports. Thus, a component can instantiate values if it defines their implementation types. It may also be possible for a component to instantiate values whose implementation type is imported by the component. However, to do that, the implementation types of these values must be exported by the components that declare them. Hence, a component cannot instantiate values whose implementation type is internal to other components.

The only way a value can be transferred between components is through the remote interface of an exported type. Remotely accessible members have their types explicitly specified. Type safety ensures that values passed through the remote interface are assignment compatible with the members types. In other words, it ensures that the values have the types specified in the member signatures.

Types mentioned in member signatures of remote members must all be exported. As such, none of these types can be a supertype of an internal type. Hence, values in the sealed zone, i.e. whose implementation type is internal, are not assignment compatible with types used in the signatures of remote members. Consequently, references to values in the sealed zone of a component cannot be passed to other components.

Appendix B

S-RVM Implementation Details

This Appendix delves into the details of the implementation of S-RVM. It describes the mechanism presented in Chapter 4 and presents parts of the relevant source code.

The description assumes some familiarity with the JikesRVM source code. While an attempt to explain relevant JikesRVM concepts has been made, providing an introduction to the JikesRVM source is well beyond the scope of this work.

Areas covered include the creation and use of the upcall interface, task initialisation, handling `String` backing store and cross-task exception conversion.

B.1 The Upcall Interface

`MuLibInterface` is an abstract class that defines the main upcall interface to the application task. It is used for services that the application task provides to the root task. Before the `MuLibInterface` of the application task is created, the only access code running within the root task has to application task types is through reflection. The root task uses reflection to create the `MuLibInterface` object. This code is shown in Listing B.1.

Like many initialisation methods in JikesRVM, the code has two main cases. One for building the virtual machine itself (Lines 814–816) and the other for initialising tasks when the virtual machine is running (Lines 798–812). When the virtual machine is being built, the code executes on an existing Java virtual machine as part of the boot image writer program. (See Section 4.2.5.) The only task existing at this stage is the root task and the code uses Java reflection for creating the object.

When creating the application task, the code is a bit more complex. The Java reflection methods cannot be used because they are part of the Java library, which is task dependent. In particular, when the root task refers to the symbolic class name `org.jikesrvm.mu.MuImplementation.class` it always means the root task

```
794     private MuLibInterface createLibInterface() {
795       try {
796         MuLibInterface res;
797         if (VM.runningVM) {
798           TypeReference tr = getBootstrapType("Lorg/jikesrvm/mu/MuImplementation;");
799           RVMClass cls = tr.resolve().asClass();
800
801           cls.resolve();
802           cls.instantiate();
803           cls.initialize();
804
805           Atom getLibInterfaceName = Atom.findOrCreateAsciiAtom(("getLibInterface"));
806           Atom getLibInterfaceDescriptor =
                    Atom.findOrCreateAsciiAtom(("()Lorg/jikesrvm/mu/MuLibInterface;"));
807           RVMMethod getLibInterfaceMethod = cls.findDeclaredMethod(getLibInterfaceName,
                    getLibInterfaceDescriptor);
808
809
810           getLibInterfaceMethod.compile();
811
812           res = (MuLibInterface)Reflection.invoke(getLibInterfaceMethod, null, null, new Object[0], false);
813         } else {
814           Method method = org.jikesrvm.mu.MuImplementation.class.getDeclaredMethod("getLibInterface",
                    (Class[])null);
815           method.setAccessible(true);
816           res = (MuLibInterface) method.invoke(null, (Object[])null);
817         }
818         return res;
819       } catch (Exception e) {
820         throw new Error(e);
821
822       }
823     }
```

Listing B.1: `RVMTask`:Creating the upcall interface object

class. Instead, the code uses the JikesRVM reflection and its internal representation of the class.

Line 798 generates a type reference for the class `MuImplementation`. A type reference in JikesRVM is an unresolved symbolic reference to a type qualified by the classloader that created the reference. `MuImplementation` is the application task class that extends `MuLibInterface` and provides the implementation of the upcall interface.

Line 799 *resolves* the type reference. When resolving a type reference, the virtual machine finds the class loader that loads the class. (Due to class loader delegation this may be a different than the class loader which creates the reference.) The resolution process also loads the class to the virtual machine. The class is initialised in lines 801–803.

Lines 805–807 locate the static method `getLibInterface` in the loaded class. The method is compiled to machine code (line 810) and invoked (line 812).

Once a reference to the `MuLibInterface` object is obtained, the object can be used for upcalls. For example, Listing B.2 shows how the upcall interface is used to create the `Class` object corresponding to a loaded type.

The Java language specifies that `synchronized static` methods are synchronised using the object lock of the `Class` object of the class that defines a method.

```
686    private MuObject<Class> createClassForTypeUnchecked(RVMType type) {
687      if (VM.VerifyAssertions) VM._assert(libInterface != null);
688      MuObject<Class> rv = libInterface.createClassForType(type);
689      if (rv.isNull())
690        return rv;
691      RVMType rvType = Magic.getObjectType(rv);
692      if (rvType != getJavaLangClassType())
693        return null;
694      return rv;
695    }
```

Listing B.2: `RVMTask`:Creating class objects

To implement this requirement, the virtual machine needs a reference to the `Class` object, which it acquires using the code in Listing B.2. The upcall is invoked in Line 688. The code that implements the method is displayed in Listing B.3. The code calls the JikesRVM hook method that creates a `Class` object and wraps it as a `MuObject` reference.

```
103    public MuObject<Class> createClassForType(RVMType type) {
104      return MuObject.fromObject((Class)java.lang.JikesRVMSupport.createClass(type));
105    }
```

Listing B.3: `MuImplementation`:Creating class objects

Wrapping is required because the `Class` is an application object. and the reference protection prevents the root task from holding unwrapped references to the object. `MuObject` is an unboxed wrapper. `MuObject.fromObject()` is a magic method, which is intercepted by the compiler and replaced by a change of type in the parse tree. No machine code is generated for wrapping the reference and the bit value of the `MuObject` wrapper is the same as that of the reference to the `Class` object.

Another point worth attention is the tests in lines 689–693 of Listing B.2 which verify that the returned reference is, indeed, a reference to a `Class` object. This is part of enforcing the policy on the VM interface layer being a trust boundary. An untrusted application could return a wrapped reference to an object other than a `Class` object. It could, in theory, return a wrapped reference to an exported root task object, which would allow the root task to lock that object, bypassing the security of the VM interface layer. The test for object type prevents this risk.

S-RVM, like JikesRVM, creates the `Class` objects when classes are loaded to the virtual machine. When initialising the application task, this can cause a circular dependency. The next section discusses circular dependencies in the application task initialisation and presents S-RVM's solution.

B.2 Initialising `RVMTask`

Task initialisation deserves special attention. The main purpose of the initialisation phase is to get the task to a state that it can load classes. The main difficulty is circular dependency between classes.

The main causes of circular dependency is the creation of `String` and `Class` literals. As discussed above, when a class is loaded, a `Class` object representing the loaded class is created. To create the `Class` object, the `Class` class needs be loaded. Like all classes in Java, `Class` is a subclass of `Object`. So, before class `Class` is loaded, class `Object` needs to be loaded. However, class `Object` cannot be loaded without creating its `Class` object.

A similar problem occurs with `String` literals. To create `String` literals, class `String` must be loaded and the task must be able to intern strings. Classes involved in interning strings, as well as the class `String` itself, contain `String` literals. These `String` literals cannot be created before all the classes that use them are loaded and to load the classes the literals need be created.

To address these problems, S-RVM starts tasks in a *foetal* state. In this state, S-RVM loads classes, but only create the `Class` and `String` literals once it has enough context. When it is unable to create the string and class literals, S-RVM just records the classes it loads. Once enough classes are loaded for creating the literals, S-RVM uses the records it collected to create the literals it missed.

```
697    public MuObject<Class> createClassForType(RVMType type) {
698      if (fetalClasses != null) {
699        addFetalClass(type);
700      }
701      if (!canCreateClasses())
702        return null;
703      return createClassForTypeUnchecked(type);
704    }
705
706    private void addFetalClass(RVMType type) {
707      if (nFetalClasses == fetalClasses.length) {
708        if (VM.VerifyAssertions) VM._assert(nFetalClasses < 50);
709        RVMType[] newFetalClasses = new RVMType[nFetalClasses * 2];
710        System.arraycopy(fetalClasses, 0, newFetalClasses, 0, nFetalClasses);
711        fetalClasses = newFetalClasses;
712      }
713      fetalClasses[nFetalClasses++] = type;
714    }
```

Listing B.4: `RVMTask`:Creating foetal class objects

The method `createClassForTypeUnchecked` shown in Listing B.2 is `private` to the `RVMTask` class. Listing B.4 shows the public interface that `RVMTask` provides for creating the `Class` object. As the code demonstrates, if the task is in foetal state it keeps track of classes created (lines 698–700). The method, then proceeds to create the `Class` objects, but only if it can create classes. The method `addFetalClass` in line 706 adds newly created classes to the list of foetal classes.

```
218     if (VM.runningVM) {
219       getBootstrapType("Lorg/jikesrvm/mu/MuCharArray;").resolve();
220       getBootstrapType("Lorg/jikesrvm/mu/MuWriteableCharArray;").resolve();
221       getBootstrapType("Lorg/jikesrvm/mu/MuObject;").resolve();
222       getBootstrapType("Lorg/jikesrvm/mu/MuMagic;").resolve();
223
224       stringLiterals = new ImmutableEntryHashMapRVM<Atom, Integer>();
225
226       javaLangObjectType = javaLangObjectTypeRef.resolve().asClass();
227     } else {
228       javaLangObjectType = RVMType.JavaLangObjectType;
229     }
230
231     javaLangClassType = javaLangClassTypeRef.resolve().asClass();
232     javaLangThrowableType = javaLangThrowableTypeRef.resolve().asClass();
233     javaLangCloneableType = getBootstrapType("Ljava/lang/Cloneable;").resolve().asClass();
234     javaIoSerializableType = getBootstrapType("Ljava/io/Serializable;").resolve().asClass();
235
236
237     libInterface = createLibInterface();
238     if (VM.runningVM) {
239       patchFetalClasses();
240
241
242     }
243     flags |= TF_CAN_CREATE_CLASSES;
244
245     javaLangStringType = javaLangStringTypeRef.resolve().asClass();
246     if (VM.runningVM) {
247       internFetalStrings();
248     }
249
250     fetalClasses = null;
251
252     flags |= TF_CAN_INTERN_STRINGS;
```

Listing B.5: `RVMTask`: Initialisation

Listing B.5 shows the relevant code from the task initialisation code. The method creates the initial classes required to be able to load classes, including the magic unboxed wrapper types, `Object`, `Class`. It also creates and initialises the upcall interface object. In line 239 it calls the method `patchFetalClasses` to create the missed `Class` objects and sets the task flags to indicate it can create classes. Lines 245–252 handle the `String` literals in foetal classes.

Listing B.6 shows the `patchFetalClasses` method. As shown. the method iterates over the foetal classes. For each class it sets the `RVMType.classForType` field to a newly created `Class` object. `patchFetalClasses` uses the virtual machine reflection to bypass the `final` protection of `RVMType.classForType`. The loop and check at line 740 is required for handling classes loaded as part of creating the `Class` literals.

B.3 `String` Backing Store

Recall from Chapter 4 that the backing store for Java strings is implemented as `MuCharArray`, which is an unboxed wrapper for a root task array of characters.

```
731     private void patchFetalClasses() {
732       int i = 0;
733       int nFetalClasses;
734       do {
735         nFetalClasses = this.nFetalClasses;
736         for ( ; i < nFetalClasses; i++) {
737           RVMType type = fetalClasses[i];
738           Entrypoints.classForType.setObjectValueUnchecked(type, createClassForTypeUnchecked(type));
739         }
740       } while (nFetalClasses != this.nFetalClasses);
741     }
```

Listing B.6: `RVMTask`:Creating fetal Classes

Listing B.7 shows the code of `MuCharArray`.

```
24  @Export
25  @Unboxed
26  public final class MuCharArray implements MuGlobalSupertype {
27
28    char[] value;
29
30    public Object get() {
31      if (VM.VerifyAssertions) VM._assert(!VM.runningVM);
32      return value;
33    }
34
35    private MuCharArray(char[] v) { value = v;}
36
37    public static MuCharArray fromCharArray(char[] chars) {
38      return new MuCharArray(chars);
39    }
40
41    public char[] toCharArray() { return value; }
42
43    @Remote
44    public boolean isNull() { return value == null; }
45
46    @Remote
47    public boolean EQ(MuCharArray other) { return this == other; }
48
49    @Remote
50    public int length() { return value.length; }
51
52    @Remote
53    public char get(int index) { return value[index]; }
54  }
```

Listing B.7: `MuCharArray`

Several points are worth noting about `MuCharArray`. First, the class is exported. Hence, it can be used by both the root task and the application task. Second, it is an unboxed class. That means that S-RVM does not allocate an object for the class. Instead, the compiler intercepts the methods of the class, replacing them with its own implementation. The implementation of the methods in the class file is provided for use when building the virtual machine. During the build, the S-RVM code is executed by an external virtual machine that does not provide the magic semantics of unboxed wrappers. The implementation is semantically equivalent to the magic the S-RVM compiler implements.

The third point to note is that the he `fromCharArray` and `toCharArray` methods,

which convert a root task char array to and from a `MuCharArray` are not in the remote interface. These methods are only accessible by the root task.

Last, it should be noted that the backing store wrapped by `MuCharArray` can only be read. None of the remote operations supports changing the array.

Like `MuCharArray`, `MuWriteableCharArray` is an unboxed wrapper of a root task character array. The application task can, however, modify the contents of a `MuWriteableCharArray`. The class definition of `MuWriteableCharArray` is the same as that of `MuCharArray`. To modify the contents of a `MuWriteableCharArray`, the application task code uses the exported class `MuCharArrayUtils`. Some of the methods of `MuCharArrayUtils` are presented in Listing B.8.

```
22    @Remote
23    @Inline
24    public static MuWriteableCharArray create(int length) {
25      char[] value = new char[length];
26      return MuWriteableCharArray.fromCharArray(value);
27    }
28
29    @Remote
30    @Inline
31    public static void set(MuWriteableCharArray array, int index, char c) {
32      char[] value = array.toCharArray();
33      if (value[0] != 0)
34        throw new ArrayIndexOutOfBoundsException();
35      value[index] = c;
36    }
37
38    @Remote
39    @Inline
40    public static MuCharArray seal(MuWriteableCharArray src) {
41      char[] charArray = src.toCharArray();
42      charArray[0] = 1;
43      return MuCharArray.fromCharArray(charArray);
44    }
45
46    @Inline
47    static void safeCopy(char[] src, int src_off, char[] dst, int dst_off, int len) {
48      if (dst[0] != 0)
49        throw new ArrayIndexOutOfBoundsException();
50      RVMArray.arraycopy(src, src_off, dst, dst_off, len);
51    }
52
53    @Remote
54    @Inline
55    public static void copy(MuCharArray src, int src_off, MuWriteableCharArray dst, int dst_off, int len) {
56      safeCopy(src.toCharArray(), src_off, dst.toCharArray(), dst_off, len);
57    }
```

Listing B.8: `MuCharArrayUtils`

The `seal` method locks a `MuWriteableCharArray` object, preventing further changes to it and returns the backing store wrapped as a `MuCharArray`. For its implementation, `seal` needs to use a flag that indicates whether the backing store is writable. Typically, in Java, this requires creating an object that has a flag field and an array for backing store. This solution, however, results in a significant overhead, both in space, for allocating the object, and in time, for dereferencing the object to access the array. S-RVM reduces the overhead by using the first array element

(at index 0) as the flag. The `seal` method sets the value of the first element to 1, preventing further modifications of the array. All other operations, e.g. `set` at line 10 and `safeCopy` at line 15, check the flag before modifying the array.

```
1334    public String replace(char oldChar, char newChar)
1335    {
1336      if (oldChar == newChar)
1337        return this;
1338      int i = count;
1339      int x = offset − 1;
1340      while (−−i >= 0)
1341        if (value.get(++x) == oldChar)
1342          break;
1343      if (i < 0)
1344        return this;
1345      MuWriteableCharArray ca = MuCharArrayUtils.create(count + 1);
1346      MuCharArrayUtils.copy(value, offset, ca, 1, count);
1347      int offset = this.offset − 1;
1348      MuCharArrayUtils.set(ca, x − offset, newChar);
1349      while (−−i >= 0)
1350        if (value.get(++x) == oldChar)
1351          MuCharArrayUtils.set(ca, x − offset, newChar);
1352      return new String(MuCharArrayUtils.seal(ca), 1, count);
1353    }
```

Listing B.9: `String.replace()`

Listing B.9 demonstrates a typical use of `MuWriteableCharArray`. The `String.replace()` method returns a new `String` which is created by replacing every instance of the character `oldChar` with `newChar` in the `String` it applies to. Lines 1336–1344 check for the trivial cases that `newChar` is the same as `oldChar` and that `oldChar` does not appear in the string. Line 1345 allocates a backing store for the new string. Note that the allocated length is one more than the string length, to accommodate for the flag at index 0. Line 1346 copies the original string to the new backing store, followed by updating the values of the characters in Lines 1348–1351. Line 1352 seals the backing store and uses it to create a new `String` object.

B.4 Exception Conversion

As Section 4.8 discusses, exceptions generated at the root task are not application task objects and are not considered as exceptions by the application task and vice versa. For a task to catch exceptions generated by the another task, S-RVM converts the exceptions on crossing the boundary between the tasks. This section describes the mechanisms that catch and convert the exceptions.

To catch the exceptions the code of interruptible remote methods is extended with an exception handler. The code in Listing B.10 shows how S-RVM adds the exception handler. Lines 811–816 add an exception handler to the exception handlers table for the method. This exception handler covers all the code of the method and catches the `Throwable` type of the task the remote method is part of. That is, it is

equivalent to enclosing the code of the method with a `try{}catch(Throwable t)` block.

```
803    public void setRemote() {
804      if (remote)
805        return;
806      if (VM.VerifyAssertions) VM._assert(!getDeclaringClass().isResolved());
807      if (VM.VerifyAssertions) VM._assert(getDeclaringClass().isExported() ||
             getDeclaringClass().hasExportRuntimeEntrypointsAnnotation());
808      remote = true;
809      if (!isInterruptible())
810        return;
811      int exceptionConversionIndex = getDeclaringClass().getExceptionConversionIndex();
812      if (VM.VerifyAssertions) VM._assert(exceptionConversionIndex > 0);
813      int n = bytecodes.length;
814      if (exceptionHandlerMap == null)
815        exceptionHandlerMap = ExceptionHandlerMap.emptyExceptionHandlerMap();
816      exceptionHandlerMap.addHandler(0, n − 1, n,
             getDeclaringClass().getTask().getJavaLangThrowableTypeRef());
817      byte[] newBytecodes = new byte[n + 4];
818      System.arraycopy(bytecodes, 0, newBytecodes, 0, n);
819      newBytecodes[n++] = (byte)JBC_invokestatic;
820      newBytecodes[n++] = (byte)((exceptionConversionIndex >> 8) & 0xff);
821      newBytecodes[n++] = (byte)(exceptionConversionIndex & 0xff);
822      newBytecodes[n++] = (byte)JBC_athrow;
823      bytecodes = newBytecodes;
824      int calleeSize = summarySize;
825      calleeSize += CALL_COST + THROW_COST;
826      if (calleeSize > Character.MAX_VALUE) {
827        summarySize = Character.MAX_VALUE;
828      } else {
829        summarySize = (char) calleeSize;
830      }
831    }
```

Listing B.10: `NormalMethod`: Setting up a remote method

Lines 817–823 extend the method body and insert the bytecode that implements the `catch` block that handles the exception. The new bytecode inserted in lines 819–822, perform the equivalent of Java `throwMuStackTrace.convertException(t)`, where `t` is the caught exception. The code of `convertException` is presented in Listing B.11.

The exception handler invokes the first method displayed (lines 189–196). The method gets the task of the remote method and the task of the method that invoked it. If these are the same task, it returns the exception without conversion. To get the task, the code uses the method `getTaskFromStackFrame`, which, as its name suggests, scans the run-time stack to find the task. To reduce the overhead of stack tracing, the optimising compiler replaces the method call with the task reference if this is known during compilation. As the `convertThrowable` method is always inlined, the identity of the remote method is always available. Hence, stack trace is only required for finding the task that invoked the remote method and only when the remote method is not inlined.

If the tasks are different, the second method (lines 198–215) is invoked. It finds the first supertype of the thrown exception that is loaded by the base class loader of the thrown exception task. The assumption is that exception types generated by the

```
188     @Inline
189     public static MuObject<Throwable> convertThrowable(MuObject<Throwable> muThrowable) {
190       RVMTask thisTask = RVMClass.getTaskFromStackFrame(1);
191       if (VM.VerifyAssertions) VM._assert(thisTask == Magic.getObjectType(muThrowable).getTask());
192       RVMTask thatTask = RVMClass.getTaskFromStackFrame(2);
193       if (thisTask == thatTask)
194         return muThrowable;
195       return convertThrowable(muThrowable, thatTask);
196     }
197
198     public static MuObject<Throwable> convertThrowable(MuObject<Throwable> muThrowable,
            RVMTask thatTask) {
199       RVMType throwableType = Magic.getObjectType(muThrowable.toObject());
200       RVMTask throwableTask = throwableType.getTask();
201       RVMType taskThrowable = throwableTask.getJavaLangThrowableType();
202
203       if (throwableTask == thatTask)
204         return muThrowable;
205
206       MuStackTrace stackTrace = throwableTask.getThrowableStackTrace(muThrowable);
207
208       while (throwableType != taskThrowable) {
209         if (throwableType.getClassLoader().isBootstrapClassLoader())
210           return thatTask.createExceptionOfType(throwableType.getDescriptor(), stackTrace);
211         throwableType = throwableType.asClass().getSuperClass();
212       }
213       if (VM.VerifyAssertions) VM._assert(VM.NOT_REACHED);
214       return muThrowable;
215     }
```

Listing B.11: `MuStackTrace`: Converting exceptions

base class loader will be available in both tasks. It then uses the destination task to generate an equivalent exception type. The task, then, resolves the exception name within the context of its base class loader and uses an upcall to create the exception. (See Listing B.12.)

```
955     public MuObject<Throwable> createExceptionOfType(Atom name, MuStackTrace stackTrace) {
956       TypeReference tr = TypeReference.findOrCreate(getBootstrapClassLoader(), name);
957       RVMType type = tr.resolve();
958       return libInterface.createExceptionOfType(type, stackTrace);
959     }
```

Listing B.12: `RVMTask`: Creating an exception

Appendix C

Performance Data

The tables in this appendix present the performance results of the DaCapo benchmarks [Blackburn et al., 2006b, 2008] on an IBM x3500 server with two quad-core Xeon E5345 processors and 24GB of RAM; running Fedora release 16. Consistent with the DaCapo benchmark evaluation methodology [Blackburn et al., 2008], each benchmark is executed with varying heap pressures, ranging from twice the minimum heap size (Table 5.1) to ten times the minimum heap size.

Each benchmark is executed for 20 iterations to allow the virtual machine to adapt to the benchmark. Tables C.1–C.20 summarise the performance results at each iteration. Each table presents the performance data for each of the DaCapo benchmarks in each configuration for one of the iterations.

Three results are presented for each configuration: the performance of JikesRVM, the performance of S-RVM with the same heap size and the performance of S-RVM with the same heap pressure. (See Section 5.3.) The results are the time to complete the respective iteration, measured in milliseconds. Each result is the geometric mean of 12 runs, rounded to the nearest millisecond.

The *mean* column shows the geometric mean of the results for the DaCapo benchmarks. The numeric value of the mean has no real-world meaning. However, as the geometric mean is sensitive to the relative change in the data, this number can be used as a single figure that measures the relative performance of each configuration [Fleming and Wallace, 1986]. More specifically, as

$$GM\left(\frac{X_i}{Y_i}\right) = \frac{GM(X_i)}{GM(Y_i)}$$

the ratio of the geometric means of two data sets is the same as the geometric mean of the results in the first data set normalised to the second data set.

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	2,365	6,677	9,921	31,671	2,367	2,881	7,054	8,596	3,118	5,782	3,191	5,418
	S-RVM - Size	2,923	7,740	10,725	33,527	2,737	3,232	8,029	9,836	3,618	6,835	3,760	6,208
	S-RVM - Pressure	2,823	7,735	10,486	33,239	2,769	3,233	7,903	9,615	3,410	6,537	3,603	6,075
3X	JikesRVM	2,175	6,320	9,605	31,403	2,249	2,911	6,921	8,397	2,635	5,338	2,770	5,106
	S-RVM - Size	2,642	7,261	10,245	32,276	2,640	3,240	7,698	9,201	2,909	6,220	3,187	5,737
	S-RVM - Pressure	2,612	7,276	10,119	32,525	2,647	3,246	7,615	9,220	2,818	6,086	3,136	5,691
4X	JikesRVM	2,144	6,330	9,497	30,827	2,221	2,898	6,958	8,098	2,579	5,239	2,696	5,033
	S-RVM - Size	2,550	7,180	10,044	32,463	2,633	3,200	7,693	9,036	2,930	6,003	3,022	5,646
	S-RVM - Pressure	2,552	7,144	9,962	32,271	2,642	3,222	7,642	8,929	2,844	6,009	3,038	5,620
5X	JikesRVM	2,144	6,346	9,472	31,267	2,232	2,915	6,916	8,068	2,660	5,175	2,634	5,038
	S-RVM - Size	2,555	7,220	9,929	32,284	2,627	3,212	7,603	8,867	3,002	5,850	2,965	5,616
	S-RVM - Pressure	2,548	7,240	9,968	32,272	2,630	3,240	7,646	8,853	2,944	5,816	2,949	5,609
6X	JikesRVM	2,135	6,337	9,419	30,894	2,226	2,903	6,894	8,060	2,562	5,105	2,661	5,003
	S-RVM - Size	2,542	7,311	9,941	32,133	2,634	3,235	7,700	8,879	3,030	5,884	2,948	5,635
	S-RVM - Pressure	2,539	7,228	9,989	32,229	2,639	3,244	7,670	8,900	2,867	5,907	2,987	5,614
7X	JikesRVM	2,144	6,354	9,415	30,802	2,214	2,899	6,924	8,064	2,618	5,159	2,681	5,022
	S-RVM - Size	2,561	7,242	9,946	32,031	2,634	3,211	7,650	8,870	2,869	5,879	2,968	5,600
	S-RVM - Pressure	2,537	7,243	9,937	32,397	2,636	3,206	7,675	8,918	2,975	5,931	2,977	5,629
8X	JikesRVM	2,142	6,348	9,445	30,799	2,232	2,926	6,924	8,026	2,643	5,140	2,815	5,054
	S-RVM - Size	2,557	7,240	10,018	32,033	2,633	3,249	7,695	8,861	3,021	5,853	2,980	5,637
	S-RVM - Pressure	2,555	7,268	9,903	32,218	2,650	3,228	7,649	8,864	2,875	5,836	3,005	5,610
9X	JikesRVM	2,135	6,345	9,409	30,763	2,225	2,928	6,923	8,021	2,562	5,128	2,598	4,996
	S-RVM - Size	2,545	7,265	10,010	32,280	2,626	3,283	7,732	8,885	3,025	5,911	2,924	5,644
	S-RVM - Pressure	2,555	7,200	9,975	32,195	2,640	3,254	7,652	8,843	2,879	5,917	2,916	5,602
10X	JikesRVM	2,130	6,345	9,448	30,789	2,231	2,946	6,947	8,044	2,612	5,116	2,588	5,010
	S-RVM - Size	2,560	7,288	9,930	32,186	2,631	3,248	7,716	8,864	2,862	5,899	2,898	5,602
	S-RVM - Pressure	2,555	7,240	9,885	32,139	2,627	3,220	7,657	8,889	2,898	5,868	2,922	5,595

Table C.1: Mean execution times of the DaCapo benchmarks at the 1st iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,881	5,569	6,971	26,367	1,729	2,009	5,062	7,557	2,134	4,721	2,541	4,165
	S-RVM - Size	2,204	6,002	7,163	26,817	1,866	2,094	5,218	7,998	2,543	5,284	2,982	4,533
	S-RVM - Pressure	2,065	5,819	7,039	26,828	1,824	2,057	5,116	7,783	2,378	5,080	2,802	4,383
3X	JikesRVM	1,726	5,376	6,719	25,645	1,727	2,042	4,838	7,350	1,692	4,227	2,130	3,887
	S-RVM - Size	1,929	5,716	6,712	26,191	1,824	2,091	4,904	7,590	1,903	4,644	2,354	4,115
	S-RVM - Pressure	1,928	5,721	6,730	26,097	1,833	2,076	4,891	7,484	1,916	4,507	2,274	4,086
4X	JikesRVM	1,697	5,338	6,608	25,391	1,721	2,036	4,877	7,148	1,713	4,099	2,049	3,839
	S-RVM - Size	1,866	5,623	6,629	25,893	1,843	2,039	4,934	7,466	1,922	4,280	2,192	4,025
	S-RVM - Pressure	1,870	5,608	6,674	26,003	1,848	2,123	4,885	7,393	1,889	4,317	2,180	4,033
5X	JikesRVM	1,696	5,364	6,561	25,513	1,726	1,991	4,863	7,168	1,746	4,146	1,977	3,831
	S-RVM - Size	1,856	5,669	6,576	26,008	1,844	2,128	4,921	7,368	1,898	4,308	2,091	4,016
	S-RVM - Pressure	1,869	5,705	6,588	25,944	1,837	2,047	4,875	7,368	1,872	4,355	2,093	4,001
6X	JikesRVM	1,699	5,402	6,556	25,277	1,721	2,073	4,873	7,156	1,722	4,116	1,982	3,838
	S-RVM - Size	1,846	5,676	6,590	25,755	1,847	2,099	4,916	7,416	1,886	4,380	2,092	4,013
	S-RVM - Pressure	1,854	5,682	6,547	25,855	1,836	2,080	4,902	7,371	1,889	4,368	2,103	4,007
7X	JikesRVM	1,710	5,382	6,594	25,397	1,712	2,043	4,832	7,172	1,738	4,123	1,958	3,832
	S-RVM - Size	1,886	5,701	6,599	25,766	1,845	2,082	4,936	7,391	1,862	4,277	2,082	4,005
	S-RVM - Pressure	1,858	5,736	6,573	25,802	1,834	2,045	4,889	7,369	1,885	4,282	2,096	3,995
8X	JikesRVM	1,709	5,429	6,589	25,584	1,717	2,062	4,873	7,162	1,737	4,123	1,898	3,833
	S-RVM - Size	1,873	5,728	6,555	25,989	1,841	2,091	4,959	7,401	1,863	4,265	2,101	4,010
	S-RVM - Pressure	1,869	5,725	6,593	25,946	1,859	2,053	4,938	7,331	1,959	4,324	2,077	4,022
9X	JikesRVM	1,698	5,348	6,566	25,289	1,713	2,003	4,873	7,118	1,707	4,133	1,899	3,803
	S-RVM - Size	1,861	5,702	6,580	25,717	1,842	2,085	4,910	7,319	1,881	4,321	2,019	3,989
	S-RVM - Pressure	1,856	5,682	6,595	25,743	1,850	2,096	4,859	7,307	1,859	4,324	2,029	3,985
10X	JikesRVM	1,694	5,386	6,570	25,287	1,725	2,052	4,871	7,150	1,719	4,115	1,891	3,816
	S-RVM - Size	1,861	5,716	6,578	25,771	1,843	2,075	4,902	7,324	1,851	4,337	2,041	3,988
	S-RVM - Pressure	1,829	5,649	6,536	25,729	1,837	2,068	4,886	7,347	1,892	4,376	2,034	3,982

Table C.2: Mean execution times of the DaCapo benchmarks at the 2nd iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,827	5,416	6,888	25,688	1,684	1,941	4,676	7,484	1,867	4,588	2,432	3,998
	S-RVM - Size	2,112	5,890	6,901	26,207	1,772	1,998	4,902	7,901	2,443	4,921	2,737	4,335
	S-RVM - Pressure	2,001	5,809	6,779	26,146	1,729	2,026	4,758	7,674	2,260	4,715	2,586	4,205
3X	JikesRVM	1,693	5,324	6,638	25,083	1,674	1,986	4,517	7,289	1,615	4,061	2,013	3,767
	S-RVM - Size	1,844	5,551	6,570	25,519	1,743	2,050	4,527	7,491	1,767	4,335	2,142	3,926
	S-RVM - Pressure	1,822	5,557	6,544	25,378	1,742	1,977	4,534	7,408	1,797	4,200	2,118	3,893
4X	JikesRVM	1,641	5,217	6,550	24,836	1,671	1,969	4,497	7,108	1,596	3,997	1,961	3,710
	S-RVM - Size	1,792	5,523	6,496	25,278	1,749	1,964	4,581	7,295	1,827	4,004	2,004	3,848
	S-RVM - Pressure	1,765	5,533	6,432	25,394	1,754	2,010	4,536	7,265	1,769	4,048	2,041	3,844
5X	JikesRVM	1,650	5,309	6,531	24,981	1,685	1,977	4,525	7,119	1,597	4,054	1,892	3,718
	S-RVM - Size	1,772	5,564	6,411	25,353	1,752	2,017	4,562	7,225	1,780	4,076	1,932	3,832
	S-RVM - Pressure	1,796	5,597	6,409	25,328	1,745	1,973	4,545	7,247	1,741	4,031	1,939	3,819
6X	JikesRVM	1,629	5,270	6,503	24,664	1,887	1,985	4,575	7,117	1,647	3,998	1,896	3,756
	S-RVM - Size	1,759	5,553	6,440	25,141	1,745	2,018	4,557	7,263	1,759	4,129	1,947	3,831
	S-RVM - Pressure	1,780	5,565	6,380	25,195	1,741	2,007	4,540	7,272	1,777	4,062	1,926	3,824
7X	JikesRVM	1,653	5,261	6,483	24,760	1,656	1,994	4,491	7,122	1,656	4,040	1,868	3,712
	S-RVM - Size	1,791	5,572	6,436	25,136	1,749	2,023	4,560	7,260	1,772	4,051	1,955	3,837
	S-RVM - Pressure	1,768	5,535	6,396	25,257	1,740	2,003	4,540	7,217	1,746	4,047	1,933	3,811
8X	JikesRVM	1,657	5,366	6,480	24,892	1,679	1,989	4,514	7,124	1,622	4,034	1,811	3,709
	S-RVM - Size	1,786	5,616	6,381	25,242	1,748	1,995	4,571	7,244	1,779	4,068	1,858	3,817
	S-RVM - Pressure	1,782	5,573	6,419	25,240	1,754	1,936	4,549	7,229	1,810	4,027	1,885	3,812
9X	JikesRVM	1,650	5,262	6,469	24,787	1,656	1,961	4,551	7,085	1,669	4,015	1,817	3,699
	S-RVM - Size	1,801	5,588	6,383	25,095	1,747	1,999	4,573	7,213	1,714	4,124	1,864	3,809
	S-RVM - Pressure	1,764	5,592	6,423	25,127	1,755	2,031	4,530	7,194	1,804	4,036	1,888	3,822
10X	JikesRVM	1,656	5,283	6,487	24,723	1,676	1,974	4,530	7,127	1,611	4,003	1,827	3,697
	S-RVM - Size	1,787	5,540	6,397	25,152	1,755	1,968	4,534	7,208	1,925	4,062	1,863	3,832
	S-RVM - Pressure	1,757	5,553	6,376	25,090	1,745	1,979	4,526	7,208	1,779	4,045	1,888	3,801

Table C.3: Mean execution times of the DaCapo benchmarks at the 3rd iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,852	5,481	6,813	25,445	1,661	1,956	4,509	7,454	1,608	4,520	2,312	3,906
	S-RVM - Size	2,069	5,894	6,789	25,786	1,717	1,947	4,646	7,789	1,924	4,892	2,671	4,162
	S-RVM - Pressure	1,914	5,856	6,696	25,811	1,697	1,934	4,547	7,565	1,790	4,661	2,487	4,029
3X	JikesRVM	1,660	5,315	6,626	24,831	1,651	1,970	4,346	7,227	1,500	4,011	1,943	3,691
	S-RVM - Size	1,782	5,639	6,440	25,225	1,697	1,995	4,360	7,348	1,619	4,292	2,013	3,813
	S-RVM - Pressure	1,749	5,598	6,460	25,064	1,692	1,952	4,338	7,285	1,623	4,104	2,039	3,779
4X	JikesRVM	1,614	5,240	6,510	24,524	1,651	1,977	4,333	7,086	1,516	3,966	1,868	3,647
	S-RVM - Size	1,727	5,540	6,367	24,940	1,700	1,916	4,368	7,230	1,624	3,903	1,925	3,724
	S-RVM - Pressure	1,699	5,489	6,365	25,085	1,691	1,935	4,374	7,185	1,596	3,953	1,921	3,714
5X	JikesRVM	1,630	5,281	6,535	24,654	1,655	1,949	4,365	7,103	1,500	4,003	1,822	3,647
	S-RVM - Size	1,704	5,567	6,318	25,068	1,689	1,957	4,390	7,120	1,616	3,995	1,866	3,717
	S-RVM - Pressure	1,722	5,579	6,326	25,053	1,692	1,925	4,383	7,146	1,626	4,003	1,862	3,719
6X	JikesRVM	1,624	5,250	6,466	24,425	1,644	1,951	4,379	7,083	1,518	3,955	1,831	3,637
	S-RVM - Size	1,712	5,525	6,327	24,891	1,700	1,955	4,385	7,164	1,610	4,006	1,866	3,717
	S-RVM - Pressure	1,699	5,526	6,288	24,920	1,684	1,958	4,355	7,171	1,630	3,979	1,859	3,709
7X	JikesRVM	1,623	5,291	6,458	24,487	1,634	1,943	4,338	7,096	1,511	3,982	1,813	3,631
	S-RVM - Size	1,724	5,578	6,348	24,827	1,673	1,958	4,383	7,154	1,618	3,965	1,880	3,718
	S-RVM - Pressure	1,703	5,569	6,301	24,942	1,689	1,914	4,352	7,152	1,618	3,965	1,869	3,704
8X	JikesRVM	1,629	5,314	6,464	24,604	1,649	1,957	4,339	7,095	1,525	3,975	1,744	3,631
	S-RVM - Size	1,718	5,614	6,301	24,948	1,697	1,945	4,385	7,186	1,647	3,973	1,785	3,712
	S-RVM - Pressure	1,710	5,524	6,269	24,934	1,694	1,927	4,383	7,156	1,617	3,958	1,776	3,688
9X	JikesRVM	1,632	5,253	6,436	24,467	1,638	1,931	4,357	7,050	1,521	3,968	1,741	3,615
	S-RVM - Size	1,720	5,527	6,289	24,843	1,688	1,929	4,362	7,124	1,581	4,019	1,782	3,685
	S-RVM - Pressure	1,707	5,595	6,303	24,814	1,689	1,947	4,365	7,107	1,608	3,942	1,803	3,693
10X	JikesRVM	1,629	5,251	6,468	24,423	1,639	1,960	4,357	7,085	1,507	3,954	1,739	3,618
	S-RVM - Size	1,716	5,546	6,307	24,882	1,686	1,947	4,382	7,103	1,629	3,963	1,786	3,696
	S-RVM - Pressure	1,693	5,501	6,293	24,816	1,694	1,972	4,353	7,136	1,613	3,973	1,787	3,690

Table C.4: Mean execution times of the DaCapo benchmarks at the 4th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,761	5,627	6,792	25,302	1,655	1,940	4,396	7,495	1,552	4,568	2,300	3,873
	S-RVM - Size	2,021	6,116	6,765	25,681	1,690	1,895	4,518	7,782	1,899	4,820	2,650	4,124
	S-RVM - Pressure	1,902	6,075	6,656	25,641	1,667	1,932	4,443	7,587	1,613	4,633	2,428	3,972
3X	JikesRVM	1,629	5,338	6,606	24,574	1,619	1,973	4,254	7,230	1,483	4,041	1,900	3,660
	S-RVM - Size	1,745	5,695	6,432	25,246	1,651	1,959	4,283	7,348	1,489	4,084	1,980	3,736
	S-RVM - Pressure	1,725	5,612	6,432	24,936	1,654	1,874	4,244	7,261	1,485	4,078	1,992	3,702
4X	JikesRVM	1,619	5,236	6,497	24,379	1,621	1,971	4,231	7,063	1,490	3,963	1,821	3,615
	S-RVM - Size	1,692	5,592	6,330	24,809	1,657	1,943	4,267	7,208	1,489	3,906	1,872	3,665
	S-RVM - Pressure	1,664	5,605	6,337	24,911	1,657	1,924	4,270	7,143	1,507	3,925	1,888	3,664
5X	JikesRVM	1,601	5,258	6,507	24,467	1,628	1,946	4,240	7,096	1,479	4,003	1,791	3,609
	S-RVM - Size	1,676	5,554	6,291	24,951	1,658	1,935	4,275	7,113	1,491	3,981	1,816	3,651
	S-RVM - Pressure	1,681	5,604	6,274	24,933	1,654	1,888	4,263	7,125	1,503	3,966	1,824	3,647
6X	JikesRVM	1,598	5,278	6,462	24,261	1,615	1,952	4,273	7,095	1,502	3,941	1,790	3,606
	S-RVM - Size	1,676	5,535	6,292	24,735	1,667	1,918	4,283	7,154	1,496	3,993	1,812	3,650
	S-RVM - Pressure	1,690	5,596	6,249	24,787	1,656	1,925	4,250	7,144	1,502	3,946	1,823	3,650
7X	JikesRVM	1,602	5,316	6,435	24,323	1,615	1,964	4,234	7,107	1,492	3,950	1,771	3,603
	S-RVM - Size	1,686	5,592	6,309	24,673	1,642	1,935	4,281	7,134	1,505	3,937	1,823	3,651
	S-RVM - Pressure	1,665	5,609	6,268	24,838	1,656	1,907	4,233	7,139	1,488	3,974	1,821	3,642
8X	JikesRVM	1,594	5,351	6,443	24,467	1,622	1,902	4,243	7,079	1,499	3,962	1,732	3,591
	S-RVM - Size	1,679	5,615	6,263	24,765	1,649	1,943	4,284	7,158	1,504	3,936	1,741	3,639
	S-RVM - Pressure	1,675	5,578	6,245	24,705	1,658	1,899	4,283	7,139	1,489	3,928	1,745	3,624
9X	JikesRVM	1,603	5,205	6,413	24,320	1,619	1,948	4,275	7,069	1,481	3,942	1,717	3,581
	S-RVM - Size	1,672	5,575	6,243	24,652	1,646	1,891	4,270	7,091	1,507	3,983	1,720	3,619
	S-RVM - Pressure	1,677	5,607	6,267	24,624	1,660	1,931	4,250	7,120	1,494	3,935	1,736	3,629
10X	JikesRVM	1,609	5,260	6,452	24,286	1,628	1,966	4,267	7,096	1,472	3,932	1,737	3,594
	S-RVM - Size	1,682	5,613	6,267	24,727	1,650	1,887	4,284	7,115	1,505	3,987	1,729	3,630
	S-RVM - Pressure	1,672	5,491	6,261	24,640	1,666	1,920	4,253	7,128	1,496	3,950	1,746	3,624

Table C.5: Mean execution times of the DaCapo benchmarks at the 5th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,794	5,299	6,778	25,087	1,631	1,952	4,337	7,437	1,546	4,539	2,320	3,843
	S-RVM - Size	2,012	5,617	6,724	25,356	1,663	1,918	4,486	7,650	1,672	4,816	2,597	4,018
	S-RVM - Pressure	1,900	5,543	6,647	25,210	1,640	1,874	4,383	7,518	1,584	4,559	2,414	3,893
3X	JikesRVM	1,641	5,151	6,592	24,450	1,607	1,956	4,171	7,251	1,474	4,107	1,892	3,639
	S-RVM - Size	1,728	5,434	6,417	24,711	1,634	1,909	4,218	7,328	1,462	4,151	2,001	3,693
	S-RVM - Pressure	1,691	5,504	6,427	24,663	1,632	1,890	4,174	7,226	1,467	4,135	1,988	3,676
4X	JikesRVM	1,596	5,081	6,471	24,251	1,604	1,932	4,181	7,015	1,478	3,950	1,852	3,583
	S-RVM - Size	1,673	5,405	6,354	24,471	1,642	1,883	4,194	7,138	1,477	3,905	1,865	3,620
	S-RVM - Pressure	1,646	5,412	6,332	24,606	1,636	1,927	4,186	7,123	1,487	3,930	1,875	3,627
5X	JikesRVM	1,594	5,115	6,493	24,315	1,614	1,922	4,181	7,045	1,471	3,962	1,775	3,575
	S-RVM - Size	1,641	5,420	6,275	24,643	1,634	1,881	4,215	7,078	1,472	3,966	1,795	3,602
	S-RVM - Pressure	1,650	5,507	6,256	24,576	1,635	1,858	4,192	7,073	1,470	3,971	1,800	3,601
6X	JikesRVM	1,595	5,133	6,459	24,217	1,597	1,959	4,203	7,051	1,475	3,941	1,767	3,575
	S-RVM - Size	1,673	5,396	6,266	24,461	1,647	1,920	4,214	7,121	1,492	3,919	1,815	3,619
	S-RVM - Pressure	1,656	5,442	6,234	24,448	1,632	1,907	4,179	7,113	1,474	3,913	1,800	3,601
7X	JikesRVM	1,592	5,107	6,425	24,270	1,588	1,979	4,187	7,066	1,471	3,930	1,723	3,563
	S-RVM - Size	1,681	5,460	6,293	24,364	1,629	1,910	4,208	7,082	1,474	3,933	1,807	3,612
	S-RVM - Pressure	1,639	5,438	6,245	24,491	1,634	1,891	4,152	7,116	1,480	3,946	1,804	3,599
8X	JikesRVM	1,575	5,120	6,438	24,367	1,603	1,919	4,161	7,068	1,491	3,960	1,710	3,558
	S-RVM - Size	1,624	5,449	6,256	24,473	1,633	1,923	4,216	7,113	1,486	3,959	1,722	3,594
	S-RVM - Pressure	1,669	5,398	6,237	24,495	1,625	1,897	4,215	7,083	1,469	3,931	1,721	3,585
9X	JikesRVM	1,606	5,014	6,409	24,218	1,606	1,948	4,213	7,011	1,500	3,915	1,690	3,555
	S-RVM - Size	1,669	5,453	6,225	24,369	1,624	1,881	4,197	7,061	1,479	3,914	1,706	3,579
	S-RVM - Pressure	1,654	5,444	6,244	24,334	1,649	1,906	4,184	7,075	1,483	3,939	1,734	3,593
10X	JikesRVM	1,605	5,126	6,445	24,230	1,626	1,959	4,179	7,053	1,470	3,947	1,711	3,569
	S-RVM - Size	1,647	5,453	6,236	24,357	1,634	1,884	4,206	7,100	1,492	3,962	1,729	3,591
	S-RVM - Pressure	1,643	5,401	6,251	24,312	1,630	1,854	4,170	7,094	1,481	3,929	1,727	3,573

Table C.6: Mean execution times of the DaCapo benchmarks at the 6th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,770	5,204	6,754	24,925	1,634	1,896	4,275	7,394	1,544	4,437	2,291	3,800
	S-RVM - Size	1,971	5,693	6,706	25,241	1,636	1,871	4,397	7,627	1,674	4,700	2,570	3,977
	S-RVM - Pressure	1,924	5,466	6,631	25,051	1,604	1,902	4,320	7,472	1,565	4,536	2,360	3,866
3X	JikesRVM	1,604	5,120	6,586	24,296	1,594	1,929	4,112	7,194	1,458	3,922	1,895	3,595
	S-RVM - Size	1,698	5,334	6,398	24,531	1,609	1,903	4,157	7,290	1,469	4,072	1,941	3,650
	S-RVM - Pressure	1,687	5,387	6,423	24,482	1,604	1,880	4,127	7,186	1,485	3,971	1,943	3,635
4X	JikesRVM	1,577	5,051	6,470	24,071	1,597	1,923	4,115	7,021	1,475	3,916	1,828	3,559
	S-RVM - Size	1,641	5,360	6,285	24,342	1,612	1,877	4,148	7,081	1,480	3,841	1,838	3,583
	S-RVM - Pressure	1,635	5,369	6,337	24,455	1,609	1,884	4,127	7,049	1,497	3,856	1,838	3,589
5X	JikesRVM	1,579	5,073	6,476	24,211	1,590	1,921	4,133	7,054	1,453	3,907	1,778	3,550
	S-RVM - Size	1,628	5,337	6,275	24,447	1,615	1,882	4,150	7,041	1,478	3,892	1,774	3,572
	S-RVM - Pressure	1,631	5,450	6,270	24,747	1,611	1,819	4,139	7,022	1,476	3,905	1,763	3,568
6X	JikesRVM	1,590	5,027	6,443	24,050	1,593	1,928	4,157	7,047	1,470	3,884	1,768	3,549
	S-RVM - Size	1,647	5,347	6,266	24,265	1,620	1,920	4,139	7,064	1,486	3,859	1,779	3,581
	S-RVM - Pressure	1,636	5,396	6,243	24,266	1,613	1,907	4,125	7,062	1,475	3,890	1,789	3,578
7X	JikesRVM	1,582	5,064	6,427	24,131	1,582	1,894	4,126	7,053	1,476	3,891	1,732	3,535
	S-RVM - Size	1,641	5,398	6,292	24,185	1,605	1,906	4,164	7,017	1,483	3,867	1,800	3,581
	S-RVM - Pressure	1,628	5,390	6,261	24,338	1,611	1,905	4,108	7,068	1,488	3,887	1,734	3,568
8X	JikesRVM	1,579	5,095	6,440	24,245	1,586	1,914	4,140	7,060	1,464	3,902	1,699	3,536
	S-RVM - Size	1,631	5,404	6,257	24,370	1,601	1,866	4,157	7,051	1,485	3,902	1,699	3,559
	S-RVM - Pressure	1,648	5,378	6,249	24,330	1,602	1,860	4,159	7,050	1,468	3,880	1,692	3,552
9X	JikesRVM	1,583	4,976	6,399	24,021	1,580	1,933	4,157	7,009	1,465	3,901	1,698	3,526
	S-RVM - Size	1,660	5,372	6,229	24,136	1,604	1,884	4,152	6,999	1,489	3,863	1,688	3,554
	S-RVM - Pressure	1,641	5,364	6,252	24,207	1,611	1,908	4,145	7,026	1,468	3,852	1,699	3,555
10X	JikesRVM	1,581	5,065	6,445	24,073	1,592	1,942	4,137	7,045	1,460	3,901	1,701	3,537
	S-RVM - Size	1,614	5,336	6,243	24,212	1,618	1,902	4,141	7,067	1,473	3,861	1,702	3,552
	S-RVM - Pressure	1,639	5,323	6,239	24,125	1,614	1,881	4,134	7,051	1,485	3,885	1,691	3,551

Table C.7: Mean execution times of the DaCapo benchmarks at the 7th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,828	5,168	6,759	24,746	1,612	1,871	4,234	7,414	1,525	4,402	2,229	3,778
	S-RVM - Size	1,978	5,499	6,696	25,118	1,635	1,887	4,414	7,617	1,697	4,748	2,511	3,968
	S-RVM - Pressure	1,869	5,443	6,605	24,940	1,585	1,874	4,285	7,397	1,583	4,477	2,385	3,839
3X	JikesRVM	1,599	5,079	6,594	24,110	1,587	1,979	4,075	7,202	1,449	3,995	1,845	3,589
	S-RVM - Size	1,693	5,342	6,387	24,408	1,592	1,884	4,094	7,235	1,475	4,050	1,920	3,629
	S-RVM - Pressure	1,669	5,360	6,411	24,382	1,595	1,874	4,096	7,127	1,476	3,993	1,913	3,614
4X	JikesRVM	1,590	4,991	6,481	23,939	1,580	1,932	4,069	7,027	1,459	3,870	1,812	3,542
	S-RVM - Size	1,625	5,316	6,338	24,148	1,597	1,850	4,114	7,044	1,479	3,839	1,836	3,564
	S-RVM - Pressure	1,598	5,322	6,307	24,337	1,593	1,842	4,093	6,987	1,486	3,848	1,829	3,555
5X	JikesRVM	1,584	5,019	6,489	24,019	1,569	1,918	4,099	7,066	1,456	3,893	1,742	3,532
	S-RVM - Size	1,599	5,342	6,258	24,325	1,588	1,872	4,141	7,003	1,471	3,906	1,758	3,551
	S-RVM - Pressure	1,621	5,417	6,259	24,249	1,595	1,855	4,119	6,968	1,486	3,866	1,761	3,555
6X	JikesRVM	1,567	5,015	6,449	23,939	1,568	1,910	4,100	7,053	1,461	3,884	1,731	3,521
	S-RVM - Size	1,620	5,320	6,282	24,153	1,610	1,913	4,101	6,985	1,470	3,862	1,755	3,556
	S-RVM - Pressure	1,637	5,348	6,233	24,152	1,596	1,913	4,081	7,008	1,479	3,854	1,766	3,558
7X	JikesRVM	1,575	5,088	6,423	24,003	1,572	1,927	4,091	7,054	1,460	3,856	1,716	3,525
	S-RVM - Size	1,597	5,349	6,293	24,106	1,587	1,873	4,115	6,998	1,480	3,836	1,750	3,542
	S-RVM - Pressure	1,598	5,363	6,242	24,159	1,595	1,891	4,069	7,005	1,468	3,868	1,714	3,536
8X	JikesRVM	1,603	5,032	6,446	24,082	1,574	1,921	4,083	7,055	1,471	3,896	1,678	3,526
	S-RVM - Size	1,619	5,368	6,245	24,251	1,593	1,905	4,114	7,014	1,482	3,858	1,685	3,545
	S-RVM - Pressure	1,632	5,355	6,229	24,202	1,588	1,872	4,108	6,992	1,486	3,849	1,677	3,536
9X	JikesRVM	1,568	4,956	6,409	23,917	1,566	1,916	4,117	7,023	1,463	3,868	1,666	3,503
	S-RVM - Size	1,633	5,376	6,215	24,035	1,606	1,883	4,113	6,962	1,482	3,827	1,675	3,535
	S-RVM - Pressure	1,624	5,355	6,249	24,107	1,600	1,887	4,092	6,964	1,490	3,827	1,681	3,536
10X	JikesRVM	1,587	5,085	6,451	23,962	1,579	1,923	4,083	7,037	1,455	3,870	1,682	3,521
	S-RVM - Size	1,612	5,362	6,242	24,080	1,595	1,858	4,115	7,033	1,474	3,870	1,679	3,532
	S-RVM - Pressure	1,616	5,327	6,248	24,032	1,610	1,882	4,074	7,002	1,483	3,877	1,677	3,535

Table C.8: Mean execution times of the DaCapo benchmarks at the 8th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
	JikesRVM	1,788	5,145	6,767	24,630	1,615	1,885	4,196	7,444	1,530	4,397	2,234	3,771
2X	S-RVM - Size	1,999	5,470	6,712	24,943	1,607	1,891	4,369	7,553	1,677	4,655	2,528	3,947
	S-RVM - Pressure	1,890	5,424	6,592	24,779	1,596	1,904	4,254	7,422	1,577	4,528	2,356	3,843
	JikesRVM	1,647	5,059	6,575	24,041	1,574	1,927	4,033	7,199	1,463	3,972	1,864	3,585
3X	S-RVM - Size	1,694	5,316	6,386	24,334	1,577	1,863	4,070	7,220	1,481	3,965	1,918	3,611
	S-RVM - Pressure	1,626	5,368	6,393	24,250	1,576	1,906	4,063	7,102	1,468	3,910	1,909	3,592
	JikesRVM	1,559	5,000	6,481	23,825	1,570	1,950	4,027	7,019	1,472	3,882	1,783	3,530
4X	S-RVM - Size	1,627	5,345	6,278	24,078	1,587	1,873	4,065	7,012	1,467	3,825	1,808	3,550
	S-RVM - Pressure	1,584	5,333	6,299	24,148	1,579	1,862	4,060	6,965	1,484	3,835	1,808	3,541
	JikesRVM	1,600	4,993	6,488	23,955	1,562	1,917	4,052	7,065	1,461	3,869	1,718	3,522
5X	S-RVM - Size	1,611	5,340	6,258	24,196	1,576	1,924	4,099	6,980	1,480	3,862	1,754	3,551
	S-RVM - Pressure	1,600	5,381	6,260	24,146	1,573	1,862	4,097	6,955	1,468	3,891	1,756	3,539
	JikesRVM	1,561	5,024	6,442	23,806	1,569	1,895	4,066	7,053	1,461	3,884	1,739	3,515
6X	S-RVM - Size	1,581	5,308	6,252	24,060	1,595	1,896	4,082	6,984	1,478	3,837	1,754	3,537
	S-RVM - Pressure	1,596	5,342	6,224	24,053	1,584	1,873	4,083	6,988	1,476	3,837	1,740	3,531
	JikesRVM	1,594	5,040	6,414	23,926	1,564	1,905	4,050	7,062	1,460	3,871	1,666	3,508
7X	S-RVM - Size	1,625	5,374	6,276	24,014	1,580	1,929	4,091	6,986	1,483	3,825	1,752	3,553
	S-RVM - Pressure	1,603	5,335	6,232	24,100	1,592	1,865	4,069	6,993	1,485	3,828	1,754	3,536
	JikesRVM	1,593	5,041	6,429	24,051	1,561	1,903	4,055	7,037	1,468	3,890	1,642	3,507
8X	S-RVM - Size	1,617	5,359	6,228	24,155	1,577	1,874	4,091	6,973	1,476	3,865	1,680	3,528
	S-RVM - Pressure	1,619	5,323	6,215	24,143	1,581	1,893	4,082	6,979	1,482	3,830	1,664	3,524
	JikesRVM	1,561	4,976	6,404	23,817	1,557	1,888	4,051	7,019	1,469	3,884	1,646	3,488
9X	S-RVM - Size	1,626	5,367	6,212	23,945	1,595	1,844	4,086	6,937	1,481	3,843	1,667	3,520
	S-RVM - Pressure	1,613	5,378	6,248	24,029	1,586	1,867	4,085	6,959	1,471	3,805	1,659	3,517
	JikesRVM	1,563	5,057	6,438	23,856	1,564	1,926	4,071	7,037	1,453	3,876	1,659	3,504
10X	S-RVM - Size	1,607	5,309	6,240	24,012	1,583	1,849	4,073	7,003	1,485	3,834	1,665	3,515
	S-RVM - Pressure	1,623	5,298	6,232	23,979	1,600	1,875	4,051	6,998	1,473	3,846	1,667	3,521

Table C.9: Mean execution times of the DaCapo benchmarks at the 9th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,831	5,114	6,758	24,570	1,608	1,901	4,239	7,363	1,535	4,365	2,191	3,769
	S-RVM - Size	1,972	5,523	6,697	24,915	1,602	1,885	4,365	7,562	1,672	4,737	2,544	3,950
	S-RVM - Pressure	1,796	5,471	6,621	24,768	1,573	1,856	4,242	7,333	1,572	4,440	2,354	3,802
3X	JikesRVM	1,612	5,032	6,585	23,941	1,567	1,921	4,012	7,158	1,473	3,989	1,845	3,570
	S-RVM - Size	1,704	5,320	6,389	24,257	1,577	1,879	4,035	7,166	1,482	3,957	1,919	3,609
	S-RVM - Pressure	1,637	5,363	6,407	24,162	1,571	1,853	4,027	7,063	1,480	4,007	1,911	3,590
4X	JikesRVM	1,612	4,950	6,468	23,801	1,569	1,916	4,033	6,996	1,466	3,862	1,776	3,526
	S-RVM - Size	1,614	5,300	6,305	23,988	1,579	1,871	4,048	6,994	1,484	3,800	1,803	3,542
	S-RVM - Pressure	1,580	5,312	6,314	24,079	1,573	1,897	4,023	6,947	1,482	3,815	1,794	3,535
5X	JikesRVM	1,584	4,989	6,473	23,899	1,555	1,853	4,033	7,046	1,473	3,831	1,713	3,501
	S-RVM - Size	1,587	5,314	6,251	24,080	1,564	1,873	4,047	6,958	1,498	3,831	1,724	3,522
	S-RVM - Pressure	1,600	5,349	6,251	24,057	1,564	1,843	4,047	6,955	1,482	3,865	1,738	3,524
6X	JikesRVM	1,554	4,977	6,447	23,781	1,558	1,893	4,046	7,020	1,471	3,860	1,719	3,501
	S-RVM - Size	1,571	5,317	6,248	23,959	1,584	1,885	4,052	6,964	1,482	3,816	1,734	3,521
	S-RVM - Pressure	1,602	5,350	6,209	23,970	1,572	1,866	4,020	6,981	1,489	3,829	1,731	3,523
7X	JikesRVM	1,583	5,027	6,413	23,833	1,551	1,854	4,010	7,043	1,488	3,864	1,687	3,498
	S-RVM - Size	1,626	5,326	6,273	23,966	1,569	1,863	4,080	6,968	1,489	3,815	1,753	3,535
	S-RVM - Pressure	1,587	5,320	6,234	24,031	1,572	1,883	4,017	6,979	1,477	3,837	1,708	3,516
8X	JikesRVM	1,564	5,034	6,430	23,930	1,547	1,875	4,035	7,028	1,468	3,877	1,639	3,488
	S-RVM - Size	1,578	5,372	6,226	24,091	1,572	1,882	4,063	6,970	1,481	3,836	1,675	3,516
	S-RVM - Pressure	1,585	5,302	6,221	24,080	1,577	1,919	4,055	6,976	1,479	3,818	1,644	3,511
9X	JikesRVM	1,574	4,957	6,407	23,759	1,564	1,902	4,037	6,992	1,467	3,865	1,639	3,487
	S-RVM - Size	1,593	5,378	6,203	23,884	1,592	1,845	4,062	6,909	1,503	3,801	1,661	3,509
	S-RVM - Pressure	1,576	5,325	6,236	23,942	1,579	1,829	4,017	6,963	1,481	3,795	1,638	3,489
10X	JikesRVM	1,590	5,040	6,452	23,798	1,552	1,904	4,042	7,017	1,460	3,874	1,667	3,502
	S-RVM - Size	1,603	5,313	6,234	23,951	1,571	1,893	4,040	7,000	1,486	3,793	1,649	3,509
	S-RVM - Pressure	1,591	5,340	6,232	23,933	1,580	1,869	4,034	6,980	1,466	3,835	1,648	3,504

Table C.10: Mean execution times of the DaCapo benchmarks at the 10th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,737	5,098	6,785	24,504	1,615	1,842	4,127	7,369	1,534	4,422	2,210	3,739
	S-RVM - Size	1,970	5,487	6,665	24,843	1,595	1,886	4,406	7,506	1,786	4,635	2,506	3,954
	S-RVM - Pressure	1,804	5,387	6,606	24,621	1,562	1,831	4,215	7,335	1,575	4,442	2,342	3,786
3X	JikesRVM	1,609	4,991	6,574	23,883	1,562	1,883	3,982	7,162	1,460	3,999	1,828	3,550
	S-RVM - Size	1,710	5,314	6,374	24,176	1,562	1,832	4,033	7,112	1,471	4,048	1,934	3,602
	S-RVM - Pressure	1,656	5,301	6,402	24,103	1,565	1,864	3,979	7,025	1,462	3,931	1,904	3,572
4X	JikesRVM	1,566	4,978	6,480	23,718	1,559	1,885	3,983	6,996	1,468	3,867	1,777	3,508
	S-RVM - Size	1,584	5,303	6,264	23,906	1,571	1,832	4,005	6,960	1,481	3,809	1,794	3,518
	S-RVM - Pressure	1,627	5,281	6,316	24,037	1,566	1,877	4,004	6,932	1,484	3,789	1,805	3,536
5X	JikesRVM	1,518	4,998	6,489	23,847	1,552	1,846	3,987	7,022	1,461	3,838	1,701	3,477
	S-RVM - Size	1,577	5,307	6,243	24,032	1,559	1,868	4,030	6,930	1,491	3,834	1,736	3,515
	S-RVM - Pressure	1,572	5,345	6,240	23,972	1,557	1,841	3,980	6,936	1,486	3,861	1,726	3,506
6X	JikesRVM	1,582	5,005	6,454	23,690	1,557	1,894	4,026	7,024	1,472	3,879	1,729	3,510
	S-RVM - Size	1,586	5,303	6,258	23,931	1,573	1,883	4,009	6,961	1,479	3,812	1,728	3,516
	S-RVM - Pressure	1,573	5,349	6,210	23,958	1,574	1,835	4,009	6,961	1,486	3,818	1,728	3,508
7X	JikesRVM	1,556	5,000	6,417	23,832	1,558	1,869	3,989	7,036	1,473	3,874	1,680	3,489
	S-RVM - Size	1,588	5,312	6,268	23,913	1,573	1,860	4,031	6,945	1,490	3,815	1,724	3,516
	S-RVM - Pressure	1,567	5,313	6,247	24,004	1,567	1,843	3,957	6,954	1,477	3,833	1,682	3,493
8X	JikesRVM	1,565	5,001	6,436	23,880	1,564	1,860	4,014	7,015	1,466	3,855	1,623	3,479
	S-RVM - Size	1,561	5,306	6,230	24,037	1,560	1,800	4,043	6,951	1,500	3,856	1,665	3,492
	S-RVM - Pressure	1,581	5,297	6,218	23,960	1,575	1,876	4,000	6,952	1,486	3,804	1,641	3,495
9X	JikesRVM	1,552	4,912	6,406	23,718	1,557	1,864	4,019	6,994	1,455	3,848	1,626	3,463
	S-RVM - Size	1,586	5,309	6,215	23,842	1,580	1,862	4,019	6,896	1,493	3,841	1,631	3,495
	S-RVM - Pressure	1,575	5,319	6,238	23,926	1,577	1,817	4,023	6,934	1,495	3,805	1,643	3,489
10X	JikesRVM	1,567	5,036	6,450	23,804	1,563	1,850	4,011	7,004	1,471	3,865	1,639	3,484
	S-RVM - Size	1,557	5,319	6,226	23,882	1,572	1,869	4,003	6,982	1,485	3,821	1,658	3,495
	S-RVM - Pressure	1,575	5,265	6,233	23,847	1,573	1,828	4,013	6,983	1,486	3,843	1,670	3,494

Table C.11: Mean execution times of the DaCapo benchmarks at the 11th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,809	5,133	6,788	24,448	1,590	1,849	4,099	7,349	1,518	4,357	2,347	3,759
	S-RVM - Size	1,975	5,453	6,668	24,823	1,638	1,832	4,402	7,514	1,898	4,651	2,564	3,983
	S-RVM - Pressure	1,827	5,411	6,583	24,646	1,559	1,788	4,193	7,330	1,640	4,490	2,390	3,805
3X	JikesRVM	1,643	4,993	6,577	23,831	1,559	1,824	3,959	7,142	1,466	3,959	1,843	3,543
	S-RVM - Size	1,703	5,271	6,378	24,110	1,555	1,799	3,982	7,132	1,494	3,959	1,909	3,580
	S-RVM - Pressure	1,647	5,285	6,385	24,021	1,577	1,814	3,965	7,024	1,479	3,931	1,912	3,565
4X	JikesRVM	1,581	4,937	6,472	23,646	1,554	1,844	3,969	6,979	1,469	3,850	1,785	3,497
	S-RVM - Size	1,589	5,290	6,288	23,843	1,571	1,804	3,982	6,961	1,478	3,782	1,789	3,508
	S-RVM - Pressure	1,585	5,283	6,296	23,977	1,555	1,833	3,975	6,940	1,530	3,804	1,789	3,522
5X	JikesRVM	1,559	4,988	6,481	23,796	1,558	1,842	3,939	7,018	1,460	3,840	1,745	3,488
	S-RVM - Size	1,604	5,279	6,230	23,943	1,554	1,770	3,985	6,933	1,478	3,815	1,734	3,491
	S-RVM - Pressure	1,577	5,406	6,261	23,863	1,549	1,766	3,972	6,920	1,495	3,841	1,721	3,494
6X	JikesRVM	1,569	5,006	6,440	23,737	1,558	1,885	4,008	7,016	1,478	3,845	1,751	3,507
	S-RVM - Size	1,610	5,283	6,245	23,873	1,580	1,831	3,976	6,954	1,488	3,815	1,743	3,512
	S-RVM - Pressure	1,572	5,344	6,210	23,889	1,557	1,833	3,971	6,944	1,509	3,822	1,738	3,506
7X	JikesRVM	1,573	5,022	6,418	23,786	1,543	1,829	3,965	7,025	1,464	3,834	1,694	3,478
	S-RVM - Size	1,596	5,320	6,266	23,880	1,562	1,792	3,991	6,931	1,484	3,781	1,739	3,498
	S-RVM - Pressure	1,569	5,307	6,231	23,917	1,551	1,846	3,927	6,934	1,492	3,833	1,698	3,491
8X	JikesRVM	1,570	4,969	6,425	23,903	1,563	1,824	3,949	7,012	1,453	3,840	1,678	3,473
	S-RVM - Size	1,578	5,339	6,233	23,963	1,563	1,816	3,995	6,954	1,500	3,841	1,646	3,492
	S-RVM - Pressure	1,591	5,318	6,213	23,937	1,571	1,814	3,980	6,937	1,496	3,810	1,645	3,488
9X	JikesRVM	1,537	4,932	6,406	23,675	1,563	1,842	3,999	6,971	1,456	3,830	1,660	3,461
	S-RVM - Size	1,580	5,326	6,206	23,738	1,576	1,845	3,986	6,877	1,490	3,827	1,642	3,487
	S-RVM - Pressure	1,573	5,339	6,227	23,810	1,575	1,792	3,982	6,935	1,499	3,783	1,644	3,479
10X	JikesRVM	1,580	5,014	6,442	23,771	1,546	1,831	3,978	7,009	1,453	3,872	1,671	3,478
	S-RVM - Size	1,588	5,270	6,216	23,855	1,567	1,801	3,982	6,980	1,499	3,800	1,639	3,481
	S-RVM - Pressure	1,579	5,313	6,231	23,832	1,569	1,783	3,967	6,981	1,484	3,828	1,656	3,481

Table C.12: Mean execution times of the DaCapo benchmarks at the 12th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,771	5,102	6,759	24,398	1,611	1,851	4,114	7,362	1,524	4,413	2,333	3,758
	S-RVM - Size	1,947	5,555	6,687	24,723	1,578	1,782	4,406	7,477	2,037	4,640	2,602	3,989
	S-RVM - Pressure	1,827	5,450	6,598	24,560	1,561	1,777	4,132	7,327	1,829	4,420	2,430	3,838
3X	JikesRVM	1,624	5,011	6,579	23,807	1,559	1,806	3,947	7,168	1,462	3,938	1,880	3,541
	S-RVM - Size	1,708	5,297	6,368	24,130	1,561	1,755	3,967	7,093	1,508	3,998	1,973	3,589
	S-RVM - Pressure	1,648	5,342	6,388	24,034	1,554	1,794	3,958	7,021	1,512	3,941	1,935	3,572
4X	JikesRVM	1,560	4,970	6,471	23,647	1,553	1,832	3,938	6,979	1,461	3,831	1,828	3,495
	S-RVM - Size	1,574	5,272	6,316	23,870	1,565	1,790	3,938	6,939	1,500	3,815	1,846	3,515
	S-RVM - Pressure	1,590	5,282	6,296	23,987	1,550	1,772	3,933	6,938	1,511	3,783	1,824	3,508
5X	JikesRVM	1,561	4,959	6,470	23,778	1,554	1,827	3,940	7,016	1,457	3,832	1,780	3,488
	S-RVM - Size	1,598	5,233	6,228	23,965	1,553	1,785	3,941	6,931	1,477	3,818	1,758	3,490
	S-RVM - Pressure	1,583	5,343	6,242	23,881	1,551	1,774	3,917	6,911	1,516	3,833	1,763	3,499
6X	JikesRVM	1,533	4,965	6,443	23,652	1,543	1,837	3,976	7,008	1,467	3,834	1,746	3,478
	S-RVM - Size	1,588	5,303	6,241	23,884	1,565	1,799	3,945	6,951	1,513	3,805	1,760	3,506
	S-RVM - Pressure	1,569	5,299	6,206	23,878	1,548	1,818	3,937	6,955	1,529	3,816	1,782	3,507
7X	JikesRVM	1,558	5,000	6,408	23,706	1,549	1,824	3,952	7,031	1,471	3,804	1,760	3,483
	S-RVM - Size	1,586	5,331	6,263	23,838	1,554	1,761	3,945	6,922	1,497	3,779	1,764	3,492
	S-RVM - Pressure	1,568	5,268	6,213	23,892	1,559	1,790	3,874	6,926	1,549	3,810	1,790	3,501
8X	JikesRVM	1,538	4,960	6,430	23,900	1,544	1,816	3,958	6,999	1,481	3,848	1,683	3,468
	S-RVM - Size	1,583	5,319	6,220	23,899	1,553	1,787	3,940	6,951	1,511	3,839	1,705	3,492
	S-RVM - Pressure	1,577	5,291	6,212	23,912	1,567	1,805	3,937	6,942	1,514	3,807	1,689	3,489
9X	JikesRVM	1,520	4,951	6,403	23,658	1,569	1,857	3,966	6,976	1,459	3,830	1,695	3,467
	S-RVM - Size	1,590	5,307	6,199	23,683	1,570	1,799	3,954	6,868	1,509	3,818	1,704	3,489
	S-RVM - Pressure	1,593	5,301	6,221	23,762	1,562	1,781	3,956	6,926	1,498	3,780	1,695	3,482
10X	JikesRVM	1,538	5,011	6,438	23,687	1,541	1,821	3,945	7,007	1,465	3,857	1,729	3,474
	S-RVM - Size	1,568	5,265	6,218	23,768	1,558	1,804	3,938	6,951	1,519	3,821	1,679	3,483
	S-RVM - Pressure	1,576	5,242	6,204	23,765	1,565	1,737	3,935	6,966	1,480	3,824	1,680	3,464

Table C.13: Mean execution times of the DaCapo benchmarks at the 13th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,884	5,046	6,731	24,430	1,578	1,817	4,100	7,380	1,560	4,344	2,238	3,749
	S-RVM - Size	1,971	5,441	6,724	24,682	1,586	1,760	4,444	7,508	1,910	4,572	2,566	3,956
	S-RVM - Pressure	1,831	5,439	6,579	24,561	1,549	1,768	4,181	7,371	1,975	4,448	2,345	3,856
3X	JikesRVM	1,640	4,964	6,579	23,748	1,571	1,845	3,925	7,112	1,451	3,891	1,843	3,533
	S-RVM - Size	1,672	5,282	6,370	24,052	1,552	1,768	3,934	7,160	1,538	3,920	1,912	3,571
	S-RVM - Pressure	1,657	5,327	6,386	23,950	1,546	1,764	3,901	7,034	1,572	3,893	1,889	3,561
4X	JikesRVM	1,579	4,927	6,479	23,580	1,551	1,804	3,924	6,977	1,478	3,856	1,769	3,484
	S-RVM - Size	1,583	5,338	6,278	23,751	1,560	1,783	3,925	6,906	1,566	3,799	1,801	3,517
	S-RVM - Pressure	1,567	5,294	6,293	23,898	1,559	1,794	3,904	6,939	1,539	3,821	1,813	3,514
5X	JikesRVM	1,551	4,981	6,468	23,675	1,559	1,814	3,931	7,006	1,476	3,818	1,725	3,476
	S-RVM - Size	1,562	5,267	6,220	23,849	1,543	1,752	3,937	6,920	1,557	3,883	1,747	3,494
	S-RVM - Pressure	1,562	5,344	6,240	23,784	1,548	1,808	3,910	6,905	1,520	3,841	1,745	3,496
6X	JikesRVM	1,535	4,961	6,439	23,602	1,548	1,821	3,934	7,017	1,476	3,818	1,719	3,468
	S-RVM - Size	1,588	5,290	6,245	23,842	1,567	1,746	3,910	6,963	1,533	3,861	1,739	3,498
	S-RVM - Pressure	1,572	5,289	6,193	23,798	1,557	1,796	3,885	6,933	1,525	3,838	1,733	3,490
7X	JikesRVM	1,555	4,989	6,415	23,684	1,539	1,829	3,918	7,008	1,471	3,824	1,729	3,473
	S-RVM - Size	1,562	5,372	6,262	23,748	1,558	1,774	3,924	6,915	1,557	3,800	1,736	3,499
	S-RVM - Pressure	1,588	5,272	6,228	23,819	1,550	1,775	3,897	6,924	1,556	3,866	1,730	3,498
8X	JikesRVM	1,560	4,958	6,427	23,825	1,547	1,843	3,930	6,983	1,487	3,853	1,651	3,469
	S-RVM - Size	1,574	5,298	6,218	23,855	1,557	1,779	3,917	6,946	1,543	3,834	1,670	3,485
	S-RVM - Pressure	1,572	5,268	6,210	23,825	1,562	1,790	3,903	6,929	1,599	3,855	1,657	3,494
9X	JikesRVM	1,575	4,906	6,417	23,557	1,553	1,830	3,956	6,997	1,470	3,825	1,633	3,457
	S-RVM - Size	1,603	5,290	6,193	23,645	1,567	1,808	3,928	6,881	1,542	3,836	1,679	3,493
	S-RVM - Pressure	1,570	5,295	6,223	23,649	1,567	1,771	3,916	6,917	1,529	3,806	1,689	3,479
10X	JikesRVM	1,543	4,996	6,453	23,631	1,543	1,833	3,923	6,990	1,465	3,866	1,669	3,464
	S-RVM - Size	1,593	5,291	6,230	23,670	1,559	1,764	3,915	6,953	1,549	3,820	1,687	3,488
	S-RVM - Pressure	1,598	5,284	6,217	23,695	1,569	1,774	3,896	6,968	1,520	3,829	1,688	3,486

Table C.14: Mean execution times of the DaCapo benchmarks at the 14th iteration (ms)

Heap Size	**Configuration**	**antlr**	**bloat**	**chart**	**eclipse**	**fop**	**hsqldb**	**jython**	**luindex**	**lusearch**	**pmd**	**xalan**	**mean**
2X	JikesRVM	1,793	5,087	6,741	24,343	1,602	1,810	4,049	7,316	1,560	4,355	2,227	3,730
	S-RVM - Size	1,920	5,436	6,718	24,678	1,615	1,739	4,263	7,510	1,717	4,670	2,511	3,895
	S-RVM - Pressure	1,810	5,516	6,576	24,469	1,543	1,775	4,094	7,342	1,690	4,466	2,362	3,796
3X	JikesRVM	1,593	4,961	6,586	23,740	1,573	1,805	3,890	7,081	1,464	3,940	1,842	3,520
	S-RVM - Size	1,680	5,288	6,358	23,990	1,562	1,775	3,920	7,140	1,567	3,926	1,882	3,574
	S-RVM - Pressure	1,610	5,333	6,369	23,913	1,552	1,780	3,878	7,006	1,592	3,883	1,882	3,554
4X	JikesRVM	1,529	4,937	6,470	23,520	1,554	1,834	3,884	6,956	1,486	3,848	1,769	3,476
	S-RVM - Size	1,584	5,261	6,284	23,758	1,560	1,736	3,866	6,906	1,558	3,771	1,805	3,497
	S-RVM - Pressure	1,542	5,279	6,276	23,796	1,549	1,761	3,866	6,928	1,568	3,779	1,789	3,492
5X	JikesRVM	1,540	4,983	6,459	23,618	1,554	1,805	3,909	6,991	1,479	3,817	1,730	3,469
	S-RVM - Size	1,549	5,261	6,220	23,846	1,539	1,775	3,897	6,912	1,592	3,828	1,705	3,486
	S-RVM - Pressure	1,588	5,330	6,231	23,719	1,542	1,764	3,875	6,880	1,544	3,817	1,702	3,481
6X	JikesRVM	1,551	4,934	6,428	23,517	1,552	1,794	3,940	6,993	1,485	3,831	1,725	3,467
	S-RVM - Size	1,551	5,273	6,241	23,804	1,570	1,783	3,891	7,020	1,560	3,791	1,735	3,496
	S-RVM - Pressure	1,555	5,311	6,187	23,788	1,555	1,789	3,856	6,906	1,553	3,795	1,723	3,483
7X	JikesRVM	1,551	4,991	6,420	23,617	1,541	1,813	3,886	6,986	1,469	3,791	1,660	3,450
	S-RVM - Size	1,561	5,302	6,259	23,731	1,561	1,770	3,886	6,925	1,567	3,801	1,743	3,495
	S-RVM - Pressure	1,568	5,289	6,218	23,813	1,550	1,753	3,834	6,917	1,522	3,834	1,695	3,469
8X	JikesRVM	1,529	4,967	6,426	23,767	1,552	1,819	3,917	6,995	1,468	3,841	1,647	3,453
	S-RVM - Size	1,562	5,305	6,217	23,789	1,546	1,769	3,918	6,930	1,552	3,829	1,674	3,479
	S-RVM - Pressure	1,580	5,307	6,205	23,861	1,559	1,755	3,872	6,943	1,578	3,776	1,652	3,477
9X	JikesRVM	1,547	4,910	6,435	23,562	1,551	1,798	3,938	7,020	1,477	3,818	1,639	3,449
	S-RVM - Size	1,536	5,285	6,190	23,583	1,559	1,778	3,886	6,874	1,533	3,806	1,667	3,461
	S-RVM - Pressure	1,569	5,298	6,219	23,593	1,554	1,743	3,883	6,893	1,573	3,763	1,653	3,465
10X	JikesRVM	1,521	5,004	6,452	23,557	1,556	1,790	3,919	6,959	1,476	3,828	1,650	3,448
	S-RVM - Size	1,546	5,269	6,223	23,672	1,562	1,767	3,882	6,943	1,589	3,810	1,641	3,473
	S-RVM - Pressure	1,546	5,226	6,206	23,672	1,556	1,751	3,876	6,920	1,591	3,800	1,655	3,466

Table C.15: Mean execution times of the DaCapo benchmarks at the 15th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,842	5,064	6,751	24,353	1,610	1,796	4,051	7,326	1,613	4,365	2,218	3,749
	S-RVM - Size	1,899	5,659	6,728	24,698	1,617	1,761	4,161	7,497	1,667	4,666	2,557	3,898
	S-RVM - Pressure	1,785	5,503	6,605	24,526	1,541	1,748	4,069	7,291	1,543	4,396	2,377	3,748
3X	JikesRVM	1,608	4,983	6,577	23,690	1,565	1,800	3,883	7,073	1,514	3,875	1,845	3,525
	S-RVM - Size	1,645	5,313	6,380	23,957	1,557	1,738	3,859	7,096	1,514	3,944	1,878	3,545
	S-RVM - Pressure	1,639	5,290	6,368	23,899	1,551	1,758	3,830	7,004	1,496	3,832	1,865	3,521
4X	JikesRVM	1,523	4,906	6,460	23,490	1,544	1,839	3,871	6,965	1,489	3,849	1,766	3,470
	S-RVM - Size	1,580	5,281	6,267	23,733	1,553	1,762	3,860	6,900	1,531	3,761	1,777	3,486
	S-RVM - Pressure	1,570	5,314	6,276	23,732	1,551	1,779	3,903	6,928	1,497	3,765	1,792	3,490
5X	JikesRVM	1,515	4,958	6,464	23,638	1,543	1,791	3,891	6,991	1,556	3,820	1,713	3,470
	S-RVM - Size	1,554	5,281	6,234	23,758	1,538	1,737	3,857	6,914	1,511	3,801	1,722	3,462
	S-RVM - Pressure	1,576	5,347	6,242	23,711	1,538	1,750	3,882	6,887	1,513	3,833	1,708	3,474
6X	JikesRVM	1,543	4,969	6,441	23,539	1,552	1,777	3,922	6,989	1,508	3,817	1,700	3,464
	S-RVM - Size	1,554	5,321	6,235	23,770	1,563	1,760	3,851	7,006	1,534	3,787	1,714	3,480
	S-RVM - Pressure	1,553	5,310	6,204	23,717	1,547	1,750	3,848	6,909	1,537	3,785	1,715	3,467
7X	JikesRVM	1,540	4,958	6,418	23,664	1,543	1,830	3,878	6,965	1,514	3,831	1,677	3,464
	S-RVM - Size	1,558	5,372	6,252	23,743	1,555	1,732	3,871	6,906	1,508	3,768	1,725	3,470
	S-RVM - Pressure	1,573	5,294	6,207	23,770	1,548	1,746	3,816	6,913	1,521	3,812	1,683	3,461
8X	JikesRVM	1,524	4,970	6,450	23,681	1,535	1,823	3,877	6,999	1,519	3,849	1,647	3,458
	S-RVM - Size	1,554	5,311	6,210	23,741	1,542	1,738	3,869	6,930	1,532	3,815	1,641	3,455
	S-RVM - Pressure	1,558	5,270	6,216	23,793	1,559	1,749	3,863	6,925	1,503	3,790	1,616	3,447
9X	JikesRVM	1,563	4,938	6,426	23,492	1,558	1,820	3,903	7,016	1,478	3,800	1,643	3,454
	S-RVM - Size	1,559	5,322	6,191	23,567	1,553	1,752	3,864	6,866	1,542	3,796	1,632	3,454
	S-RVM - Pressure	1,546	5,272	6,191	23,573	1,553	1,735	3,853	6,909	1,529	3,769	1,645	3,444
10X	JikesRVM	1,527	4,972	6,456	23,526	1,555	1,805	3,900	6,964	1,524	3,823	1,644	3,457
	S-RVM - Size	1,558	5,274	6,226	23,635	1,549	1,715	3,845	6,954	1,514	3,788	1,624	3,441
	S-RVM - Pressure	1,568	5,241	6,210	23,570	1,543	1,753	3,857	6,916	1,548	3,797	1,638	3,454

Table C.16: Mean execution times of the DaCapo benchmarks at the 16th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,769	5,039	6,727	24,309	1,563	1,774	4,004	7,299	1,734	4,363	2,207	3,735
	S-RVM - Size	1,898	5,410	6,705	24,742	1,595	1,740	4,271	7,456	1,638	4,563	2,555	3,865
	S-RVM - Pressure	1,808	5,468	6,590	24,534	1,566	1,732	4,110	7,291	1,553	4,406	2,392	3,760
3X	JikesRVM	1,607	4,949	6,594	23,736	1,568	1,833	3,856	7,066	1,539	3,955	1,839	3,539
	S-RVM - Size	1,631	5,231	6,386	23,982	1,545	1,757	3,844	7,104	1,483	3,905	1,883	3,529
	S-RVM - Pressure	1,640	5,260	6,363	23,908	1,549	1,764	3,813	7,002	1,457	3,896	1,880	3,518
4X	JikesRVM	1,511	4,896	6,481	23,492	1,539	1,811	3,838	6,966	1,533	3,832	1,772	3,468
	S-RVM - Size	1,558	5,310	6,302	23,747	1,554	1,745	3,836	6,895	1,474	3,776	1,808	3,475
	S-RVM - Pressure	1,550	5,272	6,263	23,747	1,553	1,753	3,821	6,912	1,485	3,757	1,779	3,466
5X	JikesRVM	1,552	4,941	6,467	23,616	1,551	1,782	3,843	6,980	1,480	3,821	1,707	3,455
	S-RVM - Size	1,551	5,262	6,233	23,787	1,536	1,724	3,832	6,884	1,478	3,813	1,715	3,447
	S-RVM - Pressure	1,534	5,313	6,240	23,689	1,527	1,719	3,841	6,889	1,470	3,808	1,709	3,441
6X	JikesRVM	1,522	4,932	6,457	23,551	1,543	1,800	3,897	6,984	1,536	3,811	1,707	3,464
	S-RVM - Size	1,543	5,272	6,252	23,756	1,552	1,777	3,838	7,010	1,470	3,790	1,704	3,460
	S-RVM - Pressure	1,558	5,334	6,225	23,734	1,536	1,781	3,812	6,893	1,476	3,786	1,717	3,459
7X	JikesRVM	1,527	4,989	6,416	23,587	1,536	1,784	3,869	6,968	1,531	3,814	1,650	3,449
	S-RVM - Size	1,546	5,316	6,228	23,695	1,558	1,756	3,837	6,902	1,475	3,782	1,696	3,452
	S-RVM - Pressure	1,541	5,322	6,239	23,747	1,541	1,777	3,791	6,914	1,472	3,809	1,713	3,455
8X	JikesRVM	1,513	4,990	6,454	23,700	1,550	1,795	3,852	7,006	1,524	3,814	1,614	3,446
	S-RVM - Size	1,554	5,308	6,203	23,681	1,552	1,739	3,860	6,914	1,482	3,815	1,656	3,447
	S-RVM - Pressure	1,555	5,240	6,246	23,763	1,561	1,755	3,822	6,931	1,470	3,785	1,648	3,442
9X	JikesRVM	1,526	4,890	6,418	23,480	1,548	1,794	3,883	7,019	1,519	3,805	1,628	3,441
	S-RVM - Size	1,551	5,303	6,207	23,507	1,555	1,755	3,833	6,864	1,491	3,787	1,653	3,442
	S-RVM - Pressure	1,535	5,294	6,205	23,510	1,544	1,748	3,828	6,896	1,465	3,775	1,668	3,432
10X	JikesRVM	1,531	5,002	6,446	23,505	1,555	1,794	3,879	6,959	1,528	3,825	1,651	3,457
	S-RVM - Size	1,548	5,290	6,230	23,551	1,553	1,732	3,838	6,956	1,496	3,761	1,652	3,441
	S-RVM - Pressure	1,522	5,282	6,205	23,523	1,543	1,757	3,812	6,903	1,481	3,803	1,643	3,431

Table C.17: Mean execution times of the DaCapo benchmarks at the 17th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,695	5,024	6,756	24,305	1,550	1,810	3,980	7,294	1,535	4,374	2,246	3,688
	S-RVM - Size	1,924	5,511	6,710	24,684	1,575	1,702	4,153	7,457	1,661	4,580	2,538	3,858
	S-RVM - Pressure	1,773	5,399	6,590	24,561	1,535	1,730	3,977	7,278	1,555	4,414	2,373	3,730
3X	JikesRVM	1,579	4,965	6,577	23,668	1,544	1,793	3,826	7,060	1,464	3,916	1,801	3,492
	S-RVM - Size	1,620	5,255	6,399	23,942	1,542	1,757	3,854	7,129	1,466	3,874	1,897	3,526
	S-RVM - Pressure	1,585	5,340	6,347	23,902	1,541	1,737	3,805	6,970	1,455	3,967	1,885	3,509
4X	JikesRVM	1,495	4,875	6,471	23,457	1,535	1,792	3,836	6,970	1,472	3,827	1,751	3,442
	S-RVM - Size	1,554	5,254	6,308	23,687	1,555	1,753	3,828	6,890	1,469	3,777	1,779	3,466
	S-RVM - Pressure	1,518	5,266	6,254	23,684	1,546	1,758	3,785	6,924	1,472	3,764	1,785	3,454
5X	JikesRVM	1,510	4,915	6,466	23,578	1,555	1,808	3,820	6,980	1,489	3,817	1,714	3,451
	S-RVM - Size	1,562	5,288	6,237	23,737	1,533	1,702	3,814	6,883	1,473	3,797	1,737	3,446
	S-RVM - Pressure	1,535	5,348	6,241	23,657	1,532	1,723	3,809	6,869	1,468	3,806	1,723	3,443
6X	JikesRVM	1,526	4,945	6,455	23,500	1,535	1,820	3,871	6,980	1,521	3,809	1,731	3,465
	S-RVM - Size	1,534	5,279	6,230	23,702	1,548	1,761	3,808	6,993	1,467	3,770	1,717	3,450
	S-RVM - Pressure	1,566	5,291	6,236	23,686	1,529	1,765	3,804	6,876	1,480	3,786	1,722	3,454
7X	JikesRVM	1,501	4,971	6,414	23,543	1,532	1,785	3,837	6,964	1,472	3,778	1,666	3,426
	S-RVM - Size	1,537	5,317	6,225	23,632	1,548	1,730	3,827	6,893	1,469	3,762	1,716	3,443
	S-RVM - Pressure	1,543	5,294	6,232	23,748	1,542	1,739	3,797	6,915	1,460	3,817	1,740	3,451
8X	JikesRVM	1,481	4,940	6,435	23,605	1,531	1,798	3,853	6,988	1,479	3,826	1,638	3,426
	S-RVM - Size	1,541	5,304	6,198	23,708	1,543	1,741	3,848	6,893	1,464	3,820	1,649	3,437
	S-RVM - Pressure	1,542	5,243	6,233	23,724	1,549	1,748	3,816	6,918	1,478	3,786	1,645	3,436
9X	JikesRVM	1,512	4,897	6,411	23,379	1,541	1,806	3,845	7,012	1,512	3,791	1,655	3,437
	S-RVM - Size	1,546	5,267	6,216	23,422	1,549	1,745	3,855	6,864	1,492	3,776	1,653	3,437
	S-RVM - Pressure	1,535	5,276	6,224	23,478	1,541	1,712	3,825	6,898	1,477	3,766	1,638	3,421
10X	JikesRVM	1,538	4,974	6,453	23,441	1,545	1,787	3,854	6,952	1,472	3,797	1,634	3,434
	S-RVM - Size	1,542	5,255	6,221	23,516	1,552	1,751	3,802	6,950	1,465	3,788	1,667	3,436
	S-RVM - Pressure	1,524	5,261	6,217	23,441	1,547	1,751	3,825	6,897	1,468	3,754	1,634	3,421

Table C.18: Mean execution times of the DaCapo benchmarks at the 18th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,766	4,985	6,737	24,363	1,564	1,808	4,048	7,298	1,519	4,332	2,263	3,703
	S-RVM - Size	1,955	5,514	6,715	24,607	1,561	1,722	4,089	7,459	1,660	4,543	2,523	3,853
	S-RVM - Pressure	1,802	5,420	6,567	24,526	1,573	1,712	4,004	7,276	1,553	4,415	2,302	3,730
3X	JikesRVM	1,558	4,951	6,576	23,671	1,546	1,782	3,793	7,071	1,455	3,899	1,828	3,485
	S-RVM - Size	1,612	5,286	6,392	23,481	1,532	1,702	3,813	7,089	1,472	3,994	1,895	3,512
	S-RVM - Pressure	1,588	5,314	6,348	23,829	1,541	1,736	3,799	6,976	1,466	3,884	1,850	3,496
4X	JikesRVM	1,536	4,878	6,488	23,471	1,531	1,806	3,806	6,957	1,445	3,848	1,784	3,452
	S-RVM - Size	1,531	5,265	6,337	23,688	1,542	1,733	3,793	6,891	1,464	3,767	1,760	3,449
	S-RVM - Pressure	1,520	5,254	6,263	23,687	1,542	1,720	3,810	6,903	1,482	3,779	1,782	3,450
5X	JikesRVM	1,514	4,936	6,446	23,506	1,553	1,788	3,819	6,972	1,457	3,809	1,720	3,440
	S-RVM - Size	1,547	5,239	6,231	23,730	1,534	1,719	3,811	6,889	1,462	3,810	1,718	3,438
	S-RVM - Pressure	1,542	5,312	6,240	23,623	1,529	1,719	3,817	6,863	1,470	3,797	1,705	3,437
6X	JikesRVM	1,501	4,934	6,464	23,502	1,531	1,791	3,852	6,978	1,455	3,823	1,710	3,436
	S-RVM - Size	1,517	5,264	6,243	23,624	1,553	1,714	3,802	6,993	1,471	3,755	1,712	3,436
	S-RVM - Pressure	1,527	5,287	6,262	23,652	1,536	1,758	3,776	6,871	1,474	3,783	1,721	3,442
7X	JikesRVM	1,515	4,952	6,401	23,562	1,530	1,793	3,838	6,960	1,449	3,815	1,656	3,424
	S-RVM - Size	1,542	5,288	6,227	23,649	1,548	1,744	3,807	6,896	1,476	3,779	1,741	3,451
	S-RVM - Pressure	1,527	5,316	6,253	23,687	1,538	1,725	3,779	6,920	1,471	3,811	1,742	3,446
8X	JikesRVM	1,490	4,947	6,432	23,611	1,530	1,774	3,818	6,986	1,451	3,846	1,638	3,416
	S-RVM - Size	1,533	5,302	6,212	23,681	1,544	1,735	3,821	6,900	1,459	3,803	1,660	3,432
	S-RVM - Pressure	1,530	5,242	6,251	23,707	1,544	1,724	3,824	6,923	1,473	3,786	1,642	3,428
9X	JikesRVM	1,494	4,910	6,400	23,354	1,542	1,771	3,840	7,018	1,448	3,780	1,633	3,408
	S-RVM - Size	1,531	5,297	6,241	23,395	1,554	1,732	3,790	6,865	1,488	3,776	1,651	3,428
	S-RVM - Pressure	1,539	5,266	6,225	23,466	1,549	1,705	3,821	6,901	1,464	3,748	1,675	3,424
10X	JikesRVM	1,503	4,952	6,453	23,422	1,557	1,790	3,828	6,953	1,451	3,833	1,640	3,425
	S-RVM - Size	1,528	5,230	6,247	23,459	1,549	1,744	3,805	6,943	1,480	3,779	1,655	3,430
	S-RVM - Pressure	1,515	5,217	6,215	23,457	1,538	1,732	3,779	6,889	1,474	3,775	1,628	3,410

Table C.19: Mean execution times of the DaCapo benchmarks at the 19th iteration (ms)

Heap Size	Configuration	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	mean
2X	JikesRVM	1,740	5,027	6,742	24,262	1,518	1,743	4,002	7,276	1,483	4,298	2,212	3,655
	S-RVM - Size	1,918	5,393	6,687	24,577	1,526	1,679	4,050	7,405	1,606	4,562	2,485	3,799
	S-RVM - Pressure	1,750	5,379	6,541	24,407	1,518	1,669	3,946	7,238	1,511	4,357	2,279	3,671
3X	JikesRVM	1,556	4,912	6,546	23,618	1,520	1,743	3,776	7,045	1,427	3,860	1,802	3,451
	S-RVM - Size	1,575	5,211	6,339	23,847	1,500	1,680	3,773	7,057	1,429	3,870	1,835	3,458
	S-RVM - Pressure	1,534	5,260	6,306	23,773	1,502	1,686	3,734	6,949	1,436	3,857	1,813	3,440
4X	JikesRVM	1,487	4,853	6,437	23,426	1,513	1,780	3,775	6,935	1,429	3,824	1,741	3,412
	S-RVM - Size	1,542	5,195	6,262	23,519	1,505	1,669	3,748	6,862	1,448	3,738	1,738	3,407
	S-RVM - Pressure	1,495	5,211	6,222	23,578	1,509	1,705	3,730	6,868	1,454	3,719	1,722	3,400
5X	JikesRVM	1,486	4,889	6,417	23,454	1,507	1,775	3,792	6,950	1,423	3,789	1,671	3,397
	S-RVM - Size	1,501	5,187	6,193	23,643	1,492	1,677	3,757	6,850	1,444	3,763	1,645	3,379
	S-RVM - Pressure	1,490	5,240	6,203	23,531	1,499	1,679	3,759	6,818	1,439	3,767	1,651	3,380
6X	JikesRVM	1,496	4,888	6,445	23,442	1,517	1,761	3,824	6,950	1,435	3,775	1,665	3,402
	S-RVM - Size	1,485	5,214	6,187	23,579	1,511	1,679	3,737	6,946	1,454	3,742	1,663	3,387
	S-RVM - Pressure	1,504	5,232	6,219	23,604	1,509	1,701	3,717	6,848	1,447	3,749	1,681	3,394
7X	JikesRVM	1,474	4,899	6,376	23,507	1,503	1,745	3,781	6,929	1,436	3,759	1,617	3,376
	S-RVM - Size	1,522	5,280	6,184	23,133	1,521	1,677	3,745	6,852	1,439	3,718	1,682	3,389
	S-RVM - Pressure	1,510	5,258	6,204	23,623	1,516	1,661	3,721	6,887	1,440	3,761	1,604	3,378
8X	JikesRVM	1,482	4,883	6,401	23,538	1,509	1,735	3,770	6,958	1,433	3,794	1,614	3,379
	S-RVM - Size	1,516	5,207	6,178	23,574	1,503	1,708	3,751	6,869	1,457	3,766	1,580	3,381
	S-RVM - Pressure	1,506	5,225	6,206	23,640	1,513	1,681	3,744	6,870	1,446	3,726	1,583	3,374
9X	JikesRVM	1,500	4,856	6,370	23,277	1,525	1,769	3,788	6,999	1,431	3,760	1,588	3,381
	S-RVM - Size	1,481	5,206	6,175	23,300	1,536	1,715	3,734	6,827	1,440	3,753	1,570	3,368
	S-RVM - Pressure	1,508	5,244	6,192	23,409	1,505	1,684	3,757	6,862	1,448	3,729	1,587	3,372
10X	JikesRVM	1,497	4,924	6,424	23,390	1,522	1,750	3,789	6,921	1,420	3,793	1,598	3,384
	S-RVM - Size	1,491	5,250	6,195	23,302	1,509	1,684	3,764	6,889	1,456	3,745	1,590	3,374
	S-RVM - Pressure	1,482	5,184	6,168	23,322	1,515	1,671	3,738	6,848	1,447	3,748	1,575	3,358

Table C.20: Mean execution times of the DaCapo benchmarks at the 20th iteration (ms)

Bibliography

Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 1–10, San José, California, United States, October 2006. ACM.

Antonio Albano, Alan Dearle, Giorgio Ghelli, Chris Marlin, Ron Morrison, Renzo Orsini, and David Stemple. A framework for comparing type systems for database programming languages. In Richard Hull, Ron Morrison, and David Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages*, pages 170–178, Gleneden Beach, Oregon, United States, 1989. Morgan Kaufmann.

Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP04)*, pages 1–25, Oslo, Norway, June 2004. Springer-Verlag.

Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 311–330, Seattle, Washington, United States, November 2002.

Bowen Alpern, John J. Barton, Susan Flynn-Hummel, Ton Ngo, Janice C. Shepherd, Clement R. Attanasio, Anthoni Cocchi, Derek Lieber, Mark Mergen, and Stephen 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, November 1999.

Bowen Alpern, Clement Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn-Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd,

Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

American National Standards Institute, Inc. *The Programming Language Ada Reference Manual, ANSI/MIL-STD-1815A-1983*, volume 155 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

Apache Harmony. http://harmony.apache.org.

Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time Java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems*, 7(1), December 2007.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Architecture and policy for adaptive optimization in virtual machines. Research Report RC23429, IBM, November 2004.

Godmar Back. *Isolation, Resource Management and Sharing in the KaffeOS Java Runtime System*. PhD thesis, University of Utah, May 2002.

Godmar Back and Wilson C. Hsieh. Drawing the red line in Java. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116–121, 1999.

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

Godmar Back, Wilson C. Hsieh, and Jay 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 2000a. USENIX Association.

Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 197–210, San Diego, California, United States, June 2000b.

A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory. In *Proceedings of the Second ACM Symposium on Operating Systems Principles*, pages 30–42, Princeton University, New Jersey, United States, October 1969. ACM.

Brian N. Bershad, Thomas E. Anderson, Edward D. Lazoska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems (TOCS)*, 8(1):37–55, February 1990.

Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN—an extensible microkernel for application-specific operating sytem services. In *Sixth ACM SIGOPS European Workshop*, pages 68–71, Dagstuhl Castle, Germany, September 1994.

Brian N. Bershad, Stefan Savage, Przemysław Pardyak, David Becker, Marc Fiuczynski, and Emin Gün Sirer. Protection is a software issue. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 62–65, Orcas island, Washington, United States, May 1995a.

Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–283, Copper Mountain Resort, Colorado, United States, 1995b. ACM. ISBN 0-89791-715-4.

Walter Binder, Jarle G. Hulaas, and Alex Villazón. Portable resource control in Java: the J-SEAL2 approach. In *Proceedings of the 16th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 139–155, Tampa, Florida, United States, October 2001.

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 137–146, Edinburgh, United Kingdom, May 2004. ISBN 0-7695-2163-0.

Stephen M. Blackburn, Robin Garner, and Daniel Frampton. *MMTk: The Memory Management Toolkit*, September 2006a. URL http://cs.anu.edu.au/~Robin.Garner/mmtk-guide.pdf.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben 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, United States, October 2006b. ACM Press.

Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Framp-

ton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Communications of the ACM*, 51 (8):83–89, August 2008. ISSN 0001-0782.

Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000. ISBN 0-201-70323-8.

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the fourth Workshop on the Implementation, Compilation, Optimization of Object Oriented Languages and Programming Systems (ICOOOLPS'09)*, pages 18–25, Genova, Italy, 2009.

Michael Bond. Static cloning of library methods for application and VM contexts. `http://sourceforge.net/p/jikesrvm/research-archive/39/`, April 2013.

Gilad Bracha. Generics in the Java programming language. `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`, July 2004.

Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, Grenoble, France, November 2002.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

Shigeru Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP00)*, pages 313–336, Cannes, France, June 2000.

David Clarke. *Object Ownership & Containment*. PhD thesis, University of New South Wales, July 2001.

David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 33(10):48–64, 1998. ISSN 0362-1340.

Richard C. H. Connor, Alfred L. Brown, Quintin I. Cutts, Alan Dearle, Ron Morrison, and John Rosenberg. Type equivalence checking in persistent object systems. In *Implementing Persistent Object Bases, Principles and Practice, Proceedings of the Fourth International Workshop on Persistent Objects*, pages 154–167, Martha's Vineyard, Massachusetts, United States, September 1990. Morgan Kaufmann.

Grzegorz 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, October 2000.

Grzegorz Czajkowski and Laurent Daynès. Multitasking without compromise: a virtual machine evolution. In *Proceedings of the 16th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 125–138, Tampa, Florida, United States, October 2001. ACM.

Grzegorz Czajkowski, Laurent Daynés, and Nathaniel Nystrom. Code sharing among virtual machines. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP02)*, pages 155–177, Málaga, Spain, June 2002.

Grzegorz Czajkowski, Laurent Daynès, and Ben Titzer. A multi-user virtual machine. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 85–98, San Antonio, Texas, United States, June 2003.

Markus Dahm. Byte code engineering. In *JIT'99, Java-Informations-Tage*, pages 267–277, Düsseldorf, Germany, September 1999.

Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 190–200, Oakland, California, United States, May 1996.

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

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

Jason Evans. The HipHop virtual machine. `https://www.facebook.com/note.php?note_id=10150415177928920`, December 2011.

Fabian Fagerholm. Perl 6 and the Parrot virtual machine. `http://www.parrot.org/sites/www.parrot.org/files/Fagerholm-Parrot.pdf`, April 2005.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 2006 EuroSys Conference*, pages 177–190, Leuven, Belgium, April 2006. ACM.

Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.

Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, Washington, United States, October 1996. USENIX Association.

Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: High-level low-level programming. In *Proceedings of the 2009 International Conference on Virtual Execution Environments (VEE 2009)*, pages 81–90, Washington, DC, United States, 2009. ACM.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP93)*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, and Bertil 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, Lisbon, Portugal, June 2009. IEEE Computer Society.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 57–76, Montreal, Quebec, Canada, October 2007.

Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 367–384, Nashville, Tennessee, United States, October 2008.

GNU Classpath. `http://www.classpath.org/`.

Michael Golm, Jürgen Kleinöder, and Frank Bellosa. Beyond address spaces - flexibility, performance, protection, and resource management in the type-safe JX

operating system. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 3–8, Elmau/Oberbayern, Germany, May 2001.

Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX operating system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 45–58, Monterey, California, United States, June 2002.

Li Gong. Java security: Present and near future. *IEEE Micro*, 17(3):14–19, May/June 1997.

Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, pages 103–112, Monterey, California, United States, December 1997.

James Gosling. Java intermediate bytecodes. In *ACM SIGPLAN Workshop on Intermediate Representations*, pages 111–118, San Francisco, California, United States, January 1995.

James Gosling and Henry McGilton. The Java™ language environment. White paper, Sun Microsystems, 1996.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI '02: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002. ACM Press.

Chris Hawblitzel. *Adding Operating System Structure to Language-Based Protection.* PhD thesis, Cornell University, August 2000.

Chris Hawblitzel and Thorsten 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. USENIX Association.

Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, Louisiana, United States, June 1998.

Almut Herzog and Nahid Shahmehri. Performance of the Java security manager. *Computers & Security*, 24(3):192–207, May 2005.

Galen Hunt, James Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, July 2005.

International Standardization Organization. Programming languages — C. ISO/IEC 9899:1999, International Standardization Organization, December 1999.

Java Community Process. *JSR-175: A Metadata Facility for the Java™ Programming Language*, September 2004.

Java Community Process. *JSR 121: Application Isolation API Specification*, June 2006.

JikesRVM User Guide. `http://jikesrvm.org/User+Guide`.

JNode. `http://www.jnode.org/`.

Richard Jones and Chris Ryder. A study of Java object demographics. In *Proceedings of the Seventh International Symposium on Memory Management (ISMM'08)*, pages 121–130, Tucson, Arizona, United States, June 2008.

Kaffe. `https://github.com/kaffe/kaffe`.

Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the International Symposium on Memory Management*, pages 63–74, Seattle, Washington, United States, June 2013.

Butler W. Lampson and David D. Redell. Experience with processes and monitor in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

Sheng Liang. *The Java™ Native Interface Programmer's Guide and Specification.* Addison-Wesley, June 1999. ISBN 0-201-32577-2.

Sheng Liang and Gilad 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*, pages 36–44, Vancouver, British Columbia, Canada, 1998. ACM.

Yi Lin. Personal communication, 19 July 2012.

Yi Lin, Stephen M. Blackburn, and Daniel Frampton. Unpicking the knot: Teasing apart vm/application interdependencies. In *Proceedings of the Eighth International Conference on Virtual Execution Environments (VEE 2012)*, pages 181–190, London, United Kingdom, March 2012. ACM.

Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, April 1999. ISBN 0-201-43294-3.

Barbara Liskov. Data abstraction and hierarchy. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 17–34, Orlando, Florida, United States, 1987. ACM.

Pratyusa K. Manadhata and Jeannette M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, May 2011.

Naftaly H. Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP93)*, pages 189–209, Linz, Austria, July 1996. Springer-Verlag.

James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL-79-3, XEROX Palo Alto Research Center, April 1979.

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

Roger M. Needham and R. D.H. Walker. The Cambridge CAP computer and its protection system. *SIGOPS Operating Systems Review*, 11(5):1–10, November 1977.

Erich J. Neuhold and Harold W. Lawson. *The PL/I Machine: An Introduction to Programming*. Addison-Wesley, 1971.

Scott Oaks. *Java Security*. O'Reilly Media Inc., Sebastopol, California, United States, second edition, May 2001. ISBN 978-0-596-00157-5.

Oracle Corporation. OpenJDK. `http://openjdk.java.net/`.

Elliot I. Organick. *The Multics System: An Examination of Its Structure*. The MIT Press, 1972. ISBN 0-262-15012-3.

OSGi Service Platform Core Specification, Release 4, Version 4.3. The OSGi Alliance, April 2011.

David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2): 81–92, February 1980.

Rust Reference Manual. `http://doc.rust-lang.org/doc/0.6/rust.pdf`, April 2013.

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

Keith Allan Shillington and Gillian M. Ackland, editors. *UCSD Pascal System II.0 User's Manual*. March 1979.

Emin Gün Sirer, Stefan Savage, Przemysław Pardyak, Greg P. DeFouw, Mary Ann Alapat, and Brian N. Bershad. Writing an operating system with Modula-3. In *Workshop on Computer Support for System Software*, February 1996.

Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, February 1982.

Jesper Honig Spring, Filip Pizlo, Rachid Guerraoui, and Jan Vitek. Reflexes: Abstractions for highly responsive systems. In *Proceedings of the Third International Conference on Virtual Execution Environments (VEE 2007)*, pages 191–201, San Diego, California, United States, June 2007. ACM.

Sun Microsystems, Inc. JavaOS: A standalone Java environment. White Paper, May 1996.

Sun Microsystems, Inc. Java™ remote method invocation specification. `http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf`, 2004.

Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.

Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Princples of Programming Languages*, pages 188–201, Portland, Oregon, United States, January 1994.

Patrick Tullmann. The Alta operating system. Master's thesis, Department of Computer Science, The University of Utah, December 1999.

Patrick Tullmann and Jay Lepreau. Nested Java processes: OS structure for mobile code. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, September 1998.

Patrick Tullmann, Mike Hibler, and Jay Lepreau. Janos: A Java-oriented OS for active network nodes. *IEEE Journal on Selected Areas in Communications*, 19 (3):501–510, March 2001.

Jan Vitek and Boris Bokowski. Confined types in Java. *Software–Practice and Experience*, 31(6):507–532, May 2001.

Jim Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, September 1998.

Christian Wawersich, Meik Felser, Michael Golm, and Jürgen Kleinöder. The role of IPC in the component-based operating system JX. In *The Fifth ECOOP Workshop on Object-Orientation and Operating Systems*, pages 43–48, Málaga, Spain, June 2002.

Michal Wegiel and Chandra 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, Tucson, Arizona, United States, June 2008. ACM Press.

Michal Wegiel and Chandra 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*, pages 223–240, Reno/Tahoe, Nevada, United States, October 2010.

Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management*, pages 1–42, St. Malo, France, 1992. Springer-Verlag.

Yuval Yarom, Katrina Falkner, and David S. Munro. S-RVM: a secure design for a high-performance Java virtual machine. In *Proceedings of the Sixth International Workshop on Virtual Machines and Intermediate Languages (VMIL 2012)*, pages 13–22, Tucson, Arizona, United States, October 2012.

Frank Yellin. Low level security in Java. In *Proceedings of the Fourth International World Wide Web Conference*, pages 369–379, Boston, Massachusetts, United States, December 1995. O'Reilly.