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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Import and Export</td>
<td>72</td>
</tr>
<tr>
<td>4.5</td>
<td>The VM Interface Layer</td>
<td>73</td>
</tr>
<tr>
<td>4.6</td>
<td>Creating a Trust Boundary</td>
<td>76</td>
</tr>
<tr>
<td>4.7</td>
<td>Privileged Access for the virtual machine Task</td>
<td>77</td>
</tr>
<tr>
<td>4.8</td>
<td>Exceptions</td>
<td>79</td>
</tr>
<tr>
<td>4.9</td>
<td>Implementation Verification</td>
<td>80</td>
</tr>
<tr>
<td>4.10</td>
<td>Summary</td>
<td>81</td>
</tr>
<tr>
<td>5</td>
<td>Performance Evaluation</td>
<td>82</td>
</tr>
<tr>
<td>5.1</td>
<td>Memory Usage</td>
<td>83</td>
</tr>
<tr>
<td>5.2</td>
<td>Application Task Startup</td>
<td>85</td>
</tr>
<tr>
<td>5.3</td>
<td>Steady-State Execution Speed</td>
<td>86</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary</td>
<td>92</td>
</tr>
<tr>
<td>6</td>
<td>Summary and Conclusions</td>
<td>93</td>
</tr>
<tr>
<td>6.1</td>
<td>Classification Framework</td>
<td>93</td>
</tr>
<tr>
<td>6.2</td>
<td>The Exported Types Design</td>
<td>94</td>
</tr>
<tr>
<td>6.3</td>
<td>S-RVM</td>
<td>95</td>
</tr>
<tr>
<td>6.4</td>
<td>Revisiting the Classification Framework</td>
<td>96</td>
</tr>
<tr>
<td>6.5</td>
<td>A Multi-tasking Virtual Machine</td>
<td>99</td>
</tr>
<tr>
<td>6.6</td>
<td>Structure-based Type Equivalence</td>
<td>100</td>
</tr>
<tr>
<td>6.7</td>
<td>Summary</td>
<td>101</td>
</tr>
<tr>
<td>A</td>
<td>Exported Types Specifications</td>
<td>102</td>
</tr>
<tr>
<td>A.1</td>
<td>Type Systems</td>
<td>102</td>
</tr>
<tr>
<td>A.2</td>
<td>Exported Types</td>
<td>103</td>
</tr>
<tr>
<td>A.3</td>
<td>Mapping to zones</td>
<td>105</td>
</tr>
<tr>
<td>B</td>
<td>S-RVM Implementation Details</td>
<td>107</td>
</tr>
<tr>
<td>B.1</td>
<td>The Upcall Interface</td>
<td>107</td>
</tr>
<tr>
<td>B.2</td>
<td>Initialising RVMTask</td>
<td>110</td>
</tr>
<tr>
<td>B.3</td>
<td>String Backing Store</td>
<td>111</td>
</tr>
<tr>
<td>B.4</td>
<td>Exception Conversion</td>
<td>114</td>
</tr>
<tr>
<td>C</td>
<td>Performance Data</td>
<td>117</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>138</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.8</td>
<td>Mean execution times of the DaCapo benchmarks at the 8\textsuperscript{th} iteration (ms)</td>
<td>125</td>
</tr>
<tr>
<td>C.9</td>
<td>Mean execution times of the DaCapo benchmarks at the 9\textsuperscript{th} iteration (ms)</td>
<td>126</td>
</tr>
<tr>
<td>C.10</td>
<td>Mean execution times of the DaCapo benchmarks at the 10\textsuperscript{th} iteration (ms)</td>
<td>127</td>
</tr>
<tr>
<td>C.11</td>
<td>Mean execution times of the DaCapo benchmarks at the 11\textsuperscript{th} iteration (ms)</td>
<td>128</td>
</tr>
<tr>
<td>C.12</td>
<td>Mean execution times of the DaCapo benchmarks at the 12\textsuperscript{th} iteration (ms)</td>
<td>129</td>
</tr>
<tr>
<td>C.13</td>
<td>Mean execution times of the DaCapo benchmarks at the 13\textsuperscript{th} iteration (ms)</td>
<td>130</td>
</tr>
<tr>
<td>C.14</td>
<td>Mean execution times of the DaCapo benchmarks at the 14\textsuperscript{th} iteration (ms)</td>
<td>131</td>
</tr>
<tr>
<td>C.15</td>
<td>Mean execution times of the DaCapo benchmarks at the 15\textsuperscript{th} iteration (ms)</td>
<td>132</td>
</tr>
<tr>
<td>C.16</td>
<td>Mean execution times of the DaCapo benchmarks at the 16\textsuperscript{th} iteration (ms)</td>
<td>133</td>
</tr>
<tr>
<td>C.17</td>
<td>Mean execution times of the DaCapo benchmarks at the 17\textsuperscript{th} iteration (ms)</td>
<td>134</td>
</tr>
<tr>
<td>C.18</td>
<td>Mean execution times of the DaCapo benchmarks at the 18\textsuperscript{th} iteration (ms)</td>
<td>135</td>
</tr>
<tr>
<td>C.19</td>
<td>Mean execution times of the DaCapo benchmarks at the 19\textsuperscript{th} iteration (ms)</td>
<td>136</td>
</tr>
<tr>
<td>C.20</td>
<td>Mean execution times of the DaCapo benchmarks at the 20\textsuperscript{th} iteration (ms)</td>
<td>137</td>
</tr>
</tbody>
</table>
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.

¹Ethics of the Fathers 3:17. אָם עַדְּ קַמְחָא אֱלֹהִי מַדְּרָה. 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 [OSGi, 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 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].


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

---

2The 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 applica-
tion. 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 de-
scription 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](image)

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 zones, 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 zones, 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.

![Component System with Isolated Zones]

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...
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.

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 \( O_1 \) in Component \( B \) refers, via object \( O_2 \), to object \( O_3 \)—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 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 domains 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].
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 (HIPS) 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 (SIPS) 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.

![Image of exchange heap in Singularity](image)

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 heterogeneous 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 write-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 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. In 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. As the sealed types can be identified in compile time, reference protection in these systems 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.

\[\text{Note that for ownership types, tagged and untagged versions of the same type name are not assignment compatible, hence they are not the same type.}\]
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 \textit{a priori} accessible to a components. These include static values in classes, as well as other values that can be directly referenced, e.g. \texttt{Class} and \texttt{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 \textit{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 \texttt{Class} and \texttt{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.
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 downsides. 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>Run Time</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete Isolation</td>
<td>Implicit</td>
<td></td>
</tr>
<tr>
<td>Object Sharing</td>
<td>Write Barriers</td>
<td>Shared Types</td>
</tr>
<tr>
<td>Partial Isolation</td>
<td>Transfer Barriers</td>
<td>Sealed Types</td>
</tr>
</tbody>
</table>

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.

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

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.

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

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>
<thead>
<tr>
<th></th>
<th>Run Time</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete Isolation</td>
<td>JNode</td>
<td>JX</td>
</tr>
<tr>
<td></td>
<td>MVM</td>
<td>.Net</td>
</tr>
<tr>
<td></td>
<td>OVM</td>
<td></td>
</tr>
<tr>
<td>Object Sharing</td>
<td>CoLoRs</td>
<td>Rust</td>
</tr>
<tr>
<td></td>
<td>KaffeOS</td>
<td>Singularity</td>
</tr>
<tr>
<td></td>
<td>XMem</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Real-time Java</td>
<td></td>
</tr>
<tr>
<td>Partial Isolation</td>
<td>J-Kernel</td>
<td>Confined Types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ownership Types</td>
</tr>
</tbody>
</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 its 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 $O_1$ cannot hold a direct reference to Object $O_3$.

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
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
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.

![Component A](image)

![Component B](image)

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 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 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](image)

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-
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: \texttt{monitorenter}, which locks the lock, and \texttt{monitorexit}, which releases it. In the Java language, both these operations are implemented using the \texttt{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 \texttt{int} array, providing external components with read-only access to the array. The class \texttt{IntArrayWrapper} in the example has two remote methods: \texttt{getLength} and \texttt{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
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 allow 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 runtime 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.
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
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](image)

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
@Export
public class RVMClass extends RVMType {

private final RVMClass superClass;

private FieldLayoutContext fieldLayoutContext

@Remote
public RVMClass getSuperClass() {
    return superClass;
}

public FieldLayoutContext getFieldLayoutContext() {
    return fieldLayoutContext;
}
}

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.

@Import
public class RVMClass extends RVMType {
    public RVMClass getSuperClass() { return null; }
}

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.

![RVMTask class diagram](image)

**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
object provides the interface for the application task to invoke downcalls. The
ConcreteUpcalls, whose type extends the exported type AbstractUpcalls, imple-
ments 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 cre-
ate 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 con-
structor 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 \texttt{run()} method. To achieve this it invokes the \texttt{run()} method of \texttt{RVMThread} which makes an upcall to the \texttt{VMThread} hook object, which, in turn, invokes \texttt{Thread.run()}. The use of this approach introduces an extra object (of class \texttt{VMThread} in the example above) that acts as intermediary between the Java library object (\texttt{Thread}) and the root task representation of a thread (\texttt{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 \texttt{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 \texttt{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 MuWriteableCharArrays and from MuCharArrays 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 \texttt{MuObject} unboxed wrapper and the assignment compatibility of every type with the virtual machine’s \texttt{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 \texttt{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 \texttt{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 \texttt{MuObject} to the task’s \texttt{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 underperform 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.
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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Without Compiled Code</th>
<th>With Compiled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JikesRVM</td>
<td>S-RVM</td>
</tr>
<tr>
<td>antlr</td>
<td>22MB</td>
<td>30MB</td>
</tr>
<tr>
<td>bloat</td>
<td>40MB</td>
<td>45MB</td>
</tr>
<tr>
<td>chart</td>
<td>36MB</td>
<td>45MB</td>
</tr>
<tr>
<td>eclipse</td>
<td>60MB</td>
<td>69MB</td>
</tr>
<tr>
<td>fop</td>
<td>31MB</td>
<td>40MB</td>
</tr>
<tr>
<td>hasqldb</td>
<td>102MB</td>
<td>110MB</td>
</tr>
<tr>
<td>jython</td>
<td>35MB</td>
<td>44MB</td>
</tr>
<tr>
<td>luindex</td>
<td>24MB</td>
<td>31MB</td>
</tr>
<tr>
<td>lusearch</td>
<td>45MB</td>
<td>53MB</td>
</tr>
<tr>
<td>pmd</td>
<td>37MB</td>
<td>45MB</td>
</tr>
<tr>
<td>xalan</td>
<td>47MB</td>
<td>54MB</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

![Diagram 5.2: Class loading during task startup](image)

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 61 µs.

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 230 µs).

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 means that the optimising compiler needs to optimise more methods before S-RVM reaches its steady state.
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Heap Size</th>
<th>antlr</th>
<th>bloat chart</th>
<th>eclipse</th>
<th>fop</th>
<th>hsqldb</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>2X</td>
<td>123.6 (122.4–124.7)</td>
<td>115.9 (114.8–117.1)</td>
<td>108.1 (107.2–109.0)</td>
<td>105.9 (105.3–106.5)</td>
<td>115.6 (115.1–116.1)</td>
</tr>
<tr>
<td>3rd</td>
<td>2X</td>
<td>115.6 (114.3–117.0)</td>
<td>108.8 (107.4–110.1)</td>
<td>100.2 (99.3–101.1)</td>
<td>102.0 (101.7–102.4)</td>
<td>105.3 (104.4–106.1)</td>
</tr>
<tr>
<td>5th</td>
<td>2X</td>
<td>113.8 (112.5–115.1)</td>
<td>114.4 (113.8–115.1)</td>
<td>116.0 (107.7–125.0)</td>
<td>118.2 (116.4–120.1)</td>
<td>117.8 (117.3–118.4)</td>
</tr>
<tr>
<td>10th</td>
<td>2X</td>
<td>110.3 (107.2–113.4)</td>
<td>107.3 (106.0–108.5)</td>
<td>99.2 (98.6–99.8)</td>
<td>101.3 (100.8–101.8)</td>
<td>100.5 (100.0–101.0)</td>
</tr>
<tr>
<td>20th</td>
<td>2X</td>
<td>103.0 (101.9–104.1)</td>
<td>102.7 (101.9–103.6)</td>
<td>108.9 (107.3–110.5)</td>
<td>108.5 (107.1–110.0)</td>
<td>116.1 (114.2–118.0)</td>
</tr>
</tbody>
</table>

Table 5.2: Normalised execution times of the DaCapo benchmarks on S-RVM relative to JikesRVM with 90% confidence intervals (Same Heap Size scenario)
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Heap Size</th>
<th>antlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>fop</th>
<th>hsqldb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>119.4(118.2-120.5)</td>
<td>115.8(113.5-117.2)</td>
<td>105.7(104.5-106.9)</td>
<td>104.9(104.4-105.5)</td>
<td>117.0(116.3-117.7)</td>
<td>112.3(111.1-113.4)</td>
</tr>
<tr>
<td>1st</td>
<td>2X</td>
<td>120.1(119.1-121.1)</td>
<td>115.1(113.9-116.3)</td>
<td>105.4(104.7-106.1)</td>
<td>103.6(103.1-104.0)</td>
<td>117.7(116.9-118.6)</td>
<td>111.5(110.4-112.7)</td>
</tr>
<tr>
<td></td>
<td>3X</td>
<td>118.8(117.6-120.0)</td>
<td>114.1(113.2-115.0)</td>
<td>105.2(104.3-106.1)</td>
<td>103.2(102.9-103.5)</td>
<td>117.8(117.5-118.2)</td>
<td>112.0(111.0-112.3)</td>
</tr>
<tr>
<td></td>
<td>5X</td>
<td>106.3(104.6-107.7)</td>
<td>105.4(104.2-106.1)</td>
<td>98.3(97.5-99.1)</td>
<td>101.5(101.1-101.8)</td>
<td>104.4(103.5-105.3)</td>
<td>100.3(98.3-102.3)</td>
</tr>
<tr>
<td></td>
<td>10X</td>
<td>109.5(107.5-111.6)</td>
<td>107.3(105.3-109.3)</td>
<td>98.4(97.7-99.2)</td>
<td>101.8(101.4-102.2)</td>
<td>102.7(101.9-103.4)</td>
<td>104.4(102.6-106.2)</td>
</tr>
<tr>
<td>3rd</td>
<td>2X</td>
<td>107.6(105.9-109.8)</td>
<td>104.4(103.5-105.2)</td>
<td>98.6(97.9-99.3)</td>
<td>101.2(100.7-101.6)</td>
<td>104.1(103.2-105.0)</td>
<td>99.5(97.5-101.6)</td>
</tr>
<tr>
<td></td>
<td>3X</td>
<td>108.8(107.7-110.0)</td>
<td>105.4(103.4-106.6)</td>
<td>98.1(97.4-98.9)</td>
<td>101.4(101.1-101.6)</td>
<td>103.5(102.4-104.7)</td>
<td>99.8(98.0-101.7)</td>
</tr>
<tr>
<td></td>
<td>5X</td>
<td>100.1(98.7-101.5)</td>
<td>106.0(104.9-107.0)</td>
<td>96.6(95.9-97.3)</td>
<td>100.6(100.3-100.8)</td>
<td>101.9(101.3-102.5)</td>
<td>98.1(96.8-99.5)</td>
</tr>
<tr>
<td></td>
<td>10X</td>
<td>98.1(94.8-101.5)</td>
<td>107.0(104.9-109.0)</td>
<td>98.0(97.2-98.7)</td>
<td>100.8(100.5-101.1)</td>
<td>97.8(96.6-99.1)</td>
<td>97.6(95.7-99.6)</td>
</tr>
<tr>
<td>10th</td>
<td>2X</td>
<td>101.5(99.6-103.5)</td>
<td>106.6(103.3-107.9)</td>
<td>97.3(96.4-98.2)</td>
<td>100.9(100.6-101.2)</td>
<td>100.3(99.7-100.9)</td>
<td>96.4(94.5-98.4)</td>
</tr>
<tr>
<td></td>
<td>3X</td>
<td>101.0(99.6-102.5)</td>
<td>107.2(106.2-108.3)</td>
<td>96.6(95.8-97.4)</td>
<td>100.7(100.3-101.0)</td>
<td>100.6(99.8-101.4)</td>
<td>99.4(96.4-102.6)</td>
</tr>
<tr>
<td></td>
<td>5X</td>
<td>99.9(98.2-99.9)</td>
<td>105.3(104.1-106.5)</td>
<td>96.0(95.5-96.5)</td>
<td>99.7(99.5-99.9)</td>
<td>99.6(98.6-100.6)</td>
<td>95.5(93.7-97.3)</td>
</tr>
<tr>
<td></td>
<td>10X</td>
<td>100.6(98.1-103.2)</td>
<td>107.0(105.0-109.1)</td>
<td>97.0(96.6-97.4)</td>
<td>100.6(100.4-100.8)</td>
<td>100.0(98.4-101.6)</td>
<td>95.8(94.2-97.5)</td>
</tr>
<tr>
<td>20th</td>
<td>2X</td>
<td>98.7(97.2-101.0)</td>
<td>100.1(100.5-103.2)</td>
<td>96.3(96.1-96.6)</td>
<td>100.7(100.4-100.9)</td>
<td>98.8(98.2-99.5)</td>
<td>96.7(95.5-98.0)</td>
</tr>
<tr>
<td></td>
<td>3X</td>
<td>100.3(99.0-101.6)</td>
<td>107.2(105.9-108.4)</td>
<td>96.7(96.3-97.1)</td>
<td>100.3(100.1-100.5)</td>
<td>99.5(98.7-100.3)</td>
<td>94.6(92.9-96.2)</td>
</tr>
<tr>
<td></td>
<td>5X</td>
<td>99.9(98.2-99.9)</td>
<td>105.3(104.1-106.5)</td>
<td>96.0(95.5-96.5)</td>
<td>99.7(99.5-99.9)</td>
<td>99.6(98.6-100.6)</td>
<td>95.5(93.7-97.3)</td>
</tr>
</tbody>
</table>

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

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

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.

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

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 fulfills 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
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](image)

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
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 modifications 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 \textit{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...
private MuLibInterface createLibInterface() {
    try {
        MuLibInterface res;
        if (VM.runningVM) {
            TypeReference tr = getBootstrapType("Lorg/jikesrvm/mu/MuImplementation;".Cos);
            RVMClass cls = tr.resolve().asClass();
            cls.resolve();
            cls.instantiate();
            cls.initialize();
            Atom getLibInterfaceName = Atom.findOrCreateAsciiAtom("");
            Atom getLibInterfaceDescriptor = Atom.findOrCreateAsciiAtom("()Lorg/jikesrvm/mu/MuLibInterface;".Cos);
            RVMMethod getLibInterfaceMethod = cls.findDeclaredMethod(getLibInterfaceName, getLibInterfaceDescriptor);
            getLibInterfaceMethod.compile();
            res = (MuLibInterface) Reflection.invoke(getLibInterfaceMethod, null, null, new Object[0], false);
        } else {
            Method method = org.jikesrvm.mu.MuImplementation.class.getDeclaredMethod("getLibInterface", (Class[])null);
            method.setAccessible(true);
            res = (MuLibInterface) method.invoke(null, (Object[])null);
        }
    } catch (Exception e) {
        throw new Error(e);    
    }
    return res;
} catch (Exception e) {    
    throw new Error(e);} 

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.
private MuObject<Class> createClassForTypeUnchecked(RVMType type) {
    if (VM.VerifyAssertions) VM._assert(libInterface != null);
    MuObject<Class> rv = libInterface.createClassForType(type);
    if (rv.isNull())
        return rv;
    RVMType rvType = Magic.getObjectType(rv);
    if (rvType != getJavaLangClassType())
        return null;
    return rv;
}

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.

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.

```java
public MuObject<Class> createClassForType(RVMType type) {
    if (fetalClasses != null) {
        addFetalClass(type);
    }
    if (!canCreateClasses())
        return null;
    return createClassForTypeUnchecked(type);
}
```

```java
private void addFetalClass(RVMType type) {
    if (nFetalClasses == fetalClasses.length) {
        if (VM.Verify Assertions) VM.assert(nFetalClasses < 50);
        RVMType[] newFetalClasses = new RVMType[nFetalClasses * 2];
        System.arraycopy(fetalClasses, 0, newFetalClasses, 0, nFetalClasses);
        fetalClasses = newFetalClasses;
    }
    fetalClasses[nFetalClasses++] = type;
}
```

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.
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.
private void patchFetalClasses() {
    int i = 0;
    int nFetalClasses;
    do {
        nFetalClasses = this.nFetalClasses;
        for (; i < nFetalClasses; i++) {
            RVMType type = fetalClasses[i];
            Entrypoints.classForType.setObjectValueUnchecked(type, createClassForTypeUnchecked(type));
        }
    } while (nFetalClasses != this.nFetalClasses);
}

Listing B.6: RVMTask: Creating fetal Classes

Listing B.7 shows the code of MuCharArray.

@Export
@Unboxed
public final class MuCharArray implements MuGlobalSupertype {
    char[] value;
    public Object get() {
        if (VM.VerifyAssertions) VM.assert(!VM.runningVM);
        return value;
    }
    private MuCharArray(char[] v) { value = v; }
    public static MuCharArray fromCharArray(char[] chars) {
        return new MuCharArray(chars);
    }
    public char[] toCharArray() {
        return value;
    }
    @Remote
    public boolean isNull() { return value == null; }
    public boolean EQ(MuCharArray other) { return this == other; }
    @Remote
    public int length() { return value.length; }
    @Remote
    public char get(int index) { return value[index]; }
}

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 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.

```java
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.

```java
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.

```java
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() ||
808                     getDeclaringClass().hasExportRuntimeEntrypointsAnnotation());
809     remote = true;
810     if (!isInterruptible())
811         return;
812     int exceptionConversionIndex = getDeclaringClass().getExceptionConversionIndex();
813     if (VM.VerifyAssertions) VM._assert(exceptionConversionIndex > 0);
814     int n = bytecodes.length;
815     if (exceptionHandlerMap == null)
816         exceptionHandlerMap = ExceptionHandlerMap.emptyExceptionHandlerMap();
817     exceptionHandlerMap.addHandler(0, n - 1, n, getDeclaringClass().getTask().getJavaLangThrowableTypeRef());
818     byte[] newBytecodes = new byte[n + 4];
819     System.arraycopy(bytecodes, 0, newBytecodes, 0, n);
820     newBytecodes[n++] = (byte)JBC_invokestatic;
821     newBytecodes[n++] = (byte)((exceptionConversionIndex >> 8) & 0xff);
822     newBytecodes[n++] = (byte)(exceptionConversionIndex & 0xff);
823     newBytecodes[n++] = (byte)JBC_athrow;
824     bytecodes = newBytecodes;
825     calleeSize = summarySize + 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
public static MuObject<Throwable> convertThrowable(MuObject<Throwable> muThrowable) {
    RVMTask thisTask = RVMClass.getTaskFromStackFrame(1);
    if (VM.VerifyAssertions) VM._assert(thisTask == Magic.getObjectType(muThrowable).getTask());
    RVMTask thatTask = RVMClass.getTaskFromStackFrame(2);
    if (thisTask == thatTask)
        return muThrowable;
    return convertThrowable(muThrowable, thatTask);
}

public static MuObject<Throwable> convertThrowable(MuObject<Throwable> muThrowable,
    RVMTask thatTask) {
    RVMType throwableType = Magic.getObjectType(muThrowable.toObject());
    RVMTask throwableTask = throwableType.getTask();
    RVMType taskThrowable = throwableTask.getJavaLangThrowableType();
    if (throwableTask == thatTask)
        return muThrowable;
    MuStackTrace stackTrace = throwableTask.getThrowableStackTrace(muThrowable);
    while (throwableType != taskThrowable) {
        if (throwableType.getClassLoader().isBootstrapClassLoader())
            return thatTask.createExceptionOfType(throwableType.getDescriptor(), stackTrace);
        throwableType = throwableType.asClass().getSuperClass();
    }
    if (VM.VerifyAssertions) VM._assert(VM.NOT_REACHED);
    return muThrowable;
}

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.)

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

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JikesRVM</th>
<th>antlr bloat chart eclipse fop hsqldb jython luindex lusearch pmd xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap Size</td>
<td>Configuration with boot-strap phase for best class hierarchy and java</td>
<td>mean</td>
</tr>
<tr>
<td>10X</td>
<td>505.5</td>
<td>507.9</td>
</tr>
<tr>
<td>9X</td>
<td>507.9</td>
<td>508.2</td>
</tr>
<tr>
<td>8X</td>
<td>509.2</td>
<td>509.5</td>
</tr>
<tr>
<td>7X</td>
<td>510.3</td>
<td>510.6</td>
</tr>
<tr>
<td>6X</td>
<td>511.2</td>
<td>511.5</td>
</tr>
<tr>
<td>5X</td>
<td>512.1</td>
<td>512.4</td>
</tr>
<tr>
<td>4X</td>
<td>513.0</td>
<td>513.3</td>
</tr>
<tr>
<td>3X</td>
<td>513.9</td>
<td>514.2</td>
</tr>
<tr>
<td>2X</td>
<td>514.8</td>
<td>515.1</td>
</tr>
<tr>
<td>1X</td>
<td>515.7</td>
<td>516.0</td>
</tr>
<tr>
<td>Heap Size</td>
<td>Configuration</td>
<td>antlr</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------</td>
<td>-------</td>
</tr>
<tr>
<td>2X</td>
<td>JikesRVM</td>
<td>1,881</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>2,204</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>2,065</td>
</tr>
<tr>
<td>3X</td>
<td>JikesRVM</td>
<td>1,726</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,929</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,928</td>
</tr>
<tr>
<td>4X</td>
<td>JikesRVM</td>
<td>1,697</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,866</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,870</td>
</tr>
<tr>
<td>5X</td>
<td>JikesRVM</td>
<td>1,696</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,856</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,869</td>
</tr>
<tr>
<td>6X</td>
<td>JikesRVM</td>
<td>1,699</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,846</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,854</td>
</tr>
<tr>
<td>7X</td>
<td>JikesRVM</td>
<td>1,710</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,886</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,858</td>
</tr>
<tr>
<td>8X</td>
<td>JikesRVM</td>
<td>1,709</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,873</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,869</td>
</tr>
<tr>
<td>9X</td>
<td>JikesRVM</td>
<td>1,698</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,861</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,856</td>
</tr>
<tr>
<td>10X</td>
<td>JikesRVM</td>
<td>1,694</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,861</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,829</td>
</tr>
</tbody>
</table>

Table C.2: Mean execution times of the DaCapo benchmarks at the 2nd iteration (ms)
### Table C.3: Mean execution times of the DaCapo benchmarks at the 3rd

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Heap Size</th>
<th>Tiffen</th>
<th>Triangle</th>
<th>C#</th>
<th>Fop</th>
<th>Java</th>
<th>JikesRVM</th>
<th>S-RVM - Pressure</th>
<th>S-RVM - Size</th>
<th>JikesRVM</th>
<th>S-RVM - Pressure</th>
<th>S-RVM - Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This table shows the mean execution times of various DaCapo benchmarks across different configurations, with benchmark names and execution times listed for each configuration.
<table>
<thead>
<tr>
<th>Heap Size</th>
<th>Configuration</th>
<th>antlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>fop</th>
<th>hsqldb</th>
<th>jython</th>
<th>luindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>2X</td>
<td>JikesRVM</td>
<td>1,852</td>
<td>5,451</td>
<td>6,813</td>
<td>25,415</td>
<td>1,661</td>
<td>1,956</td>
<td>4,509</td>
<td>7,454</td>
<td>1,608</td>
<td>4,520</td>
<td>2,312</td>
<td>3,906</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>2,069</td>
<td>5,894</td>
<td>6,789</td>
<td>25,786</td>
<td>1,717</td>
<td>1,947</td>
<td>4,646</td>
<td>7,789</td>
<td>1,924</td>
<td>4,892</td>
<td>2,671</td>
<td>4,162</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,914</td>
<td>5,856</td>
<td>6,696</td>
<td>25,811</td>
<td>1,697</td>
<td>1,934</td>
<td>4,547</td>
<td>7,565</td>
<td>1,790</td>
<td>4,661</td>
<td>2,487</td>
<td>4,029</td>
</tr>
<tr>
<td>3X</td>
<td>JikesRVM</td>
<td>1,660</td>
<td>5,315</td>
<td>6,626</td>
<td>24,831</td>
<td>1,651</td>
<td>1,970</td>
<td>4,346</td>
<td>7,227</td>
<td>1,500</td>
<td>4,011</td>
<td>1,943</td>
<td>3,691</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,782</td>
<td>5,639</td>
<td>6,440</td>
<td>25,225</td>
<td>1,697</td>
<td>1,995</td>
<td>4,360</td>
<td>7,348</td>
<td>1,619</td>
<td>4,292</td>
<td>2,013</td>
<td>3,813</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,749</td>
<td>5,598</td>
<td>6,460</td>
<td>25,064</td>
<td>1,692</td>
<td>1,952</td>
<td>4,338</td>
<td>7,285</td>
<td>1,623</td>
<td>4,104</td>
<td>2,039</td>
<td>3,779</td>
</tr>
<tr>
<td>4X</td>
<td>JikesRVM</td>
<td>1,614</td>
<td>5,240</td>
<td>6,510</td>
<td>24,524</td>
<td>1,651</td>
<td>1,977</td>
<td>4,333</td>
<td>7,086</td>
<td>1,516</td>
<td>3,966</td>
<td>1,868</td>
<td>3,647</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,727</td>
<td>5,540</td>
<td>6,367</td>
<td>24,940</td>
<td>1,700</td>
<td>1,916</td>
<td>4,368</td>
<td>7,230</td>
<td>1,624</td>
<td>3,903</td>
<td>1,925</td>
<td>3,724</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,699</td>
<td>5,489</td>
<td>6,365</td>
<td>25,085</td>
<td>1,691</td>
<td>1,935</td>
<td>4,374</td>
<td>7,185</td>
<td>1,596</td>
<td>3,953</td>
<td>1,921</td>
<td>3,714</td>
</tr>
<tr>
<td>5X</td>
<td>JikesRVM</td>
<td>1,630</td>
<td>5,281</td>
<td>6,535</td>
<td>24,654</td>
<td>1,655</td>
<td>1,949</td>
<td>4,365</td>
<td>7,103</td>
<td>1,500</td>
<td>4,003</td>
<td>1,822</td>
<td>3,647</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,704</td>
<td>5,567</td>
<td>6,318</td>
<td>25,068</td>
<td>1,689</td>
<td>1,957</td>
<td>4,390</td>
<td>7,120</td>
<td>1,616</td>
<td>3,995</td>
<td>1,866</td>
<td>3,717</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,722</td>
<td>5,579</td>
<td>6,326</td>
<td>25,053</td>
<td>1,692</td>
<td>1,925</td>
<td>4,383</td>
<td>7,146</td>
<td>1,626</td>
<td>4,003</td>
<td>1,862</td>
<td>3,719</td>
</tr>
<tr>
<td>6X</td>
<td>JikesRVM</td>
<td>1,624</td>
<td>5,250</td>
<td>6,466</td>
<td>24,425</td>
<td>1,644</td>
<td>1,951</td>
<td>4,379</td>
<td>7,083</td>
<td>1,518</td>
<td>3,955</td>
<td>1,831</td>
<td>3,637</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,712</td>
<td>5,525</td>
<td>6,327</td>
<td>24,891</td>
<td>1,700</td>
<td>1,955</td>
<td>4,385</td>
<td>7,164</td>
<td>1,610</td>
<td>4,006</td>
<td>1,866</td>
<td>3,717</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,699</td>
<td>5,526</td>
<td>6,288</td>
<td>24,920</td>
<td>1,684</td>
<td>1,958</td>
<td>4,355</td>
<td>7,171</td>
<td>1,630</td>
<td>3,979</td>
<td>1,859</td>
<td>3,709</td>
</tr>
<tr>
<td>7X</td>
<td>JikesRVM</td>
<td>1,623</td>
<td>5,291</td>
<td>6,458</td>
<td>24,487</td>
<td>1,634</td>
<td>1,943</td>
<td>4,338</td>
<td>7,096</td>
<td>1,511</td>
<td>3,982</td>
<td>1,813</td>
<td>3,631</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,724</td>
<td>5,578</td>
<td>6,348</td>
<td>24,827</td>
<td>1,673</td>
<td>1,958</td>
<td>4,383</td>
<td>7,154</td>
<td>1,618</td>
<td>3,965</td>
<td>1,880</td>
<td>3,718</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,703</td>
<td>5,569</td>
<td>6,301</td>
<td>24,942</td>
<td>1,689</td>
<td>1,914</td>
<td>4,352</td>
<td>7,152</td>
<td>1,618</td>
<td>3,965</td>
<td>1,869</td>
<td>3,704</td>
</tr>
<tr>
<td>8X</td>
<td>JikesRVM</td>
<td>1,629</td>
<td>5,314</td>
<td>6,464</td>
<td>24,604</td>
<td>1,649</td>
<td>1,957</td>
<td>4,339</td>
<td>7,095</td>
<td>1,525</td>
<td>3,975</td>
<td>1,744</td>
<td>3,631</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,718</td>
<td>5,614</td>
<td>6,301</td>
<td>24,948</td>
<td>1,697</td>
<td>1,945</td>
<td>4,385</td>
<td>7,186</td>
<td>1,647</td>
<td>3,973</td>
<td>1,785</td>
<td>3,712</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,710</td>
<td>5,524</td>
<td>6,269</td>
<td>24,934</td>
<td>1,694</td>
<td>1,927</td>
<td>4,383</td>
<td>7,156</td>
<td>1,617</td>
<td>3,958</td>
<td>1,776</td>
<td>3,688</td>
</tr>
<tr>
<td>9X</td>
<td>JikesRVM</td>
<td>1,632</td>
<td>5,253</td>
<td>6,436</td>
<td>24,467</td>
<td>1,638</td>
<td>1,931</td>
<td>4,357</td>
<td>7,050</td>
<td>1,521</td>
<td>3,968</td>
<td>1,741</td>
<td>3,615</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,720</td>
<td>5,527</td>
<td>6,289</td>
<td>24,843</td>
<td>1,688</td>
<td>1,929</td>
<td>4,362</td>
<td>7,124</td>
<td>1,581</td>
<td>4,019</td>
<td>1,782</td>
<td>3,685</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,707</td>
<td>5,595</td>
<td>6,303</td>
<td>24,814</td>
<td>1,689</td>
<td>1,947</td>
<td>4,365</td>
<td>7,107</td>
<td>1,608</td>
<td>3,942</td>
<td>1,803</td>
<td>3,693</td>
</tr>
<tr>
<td>10X</td>
<td>JikesRVM</td>
<td>1,629</td>
<td>5,251</td>
<td>6,468</td>
<td>24,423</td>
<td>1,639</td>
<td>1,960</td>
<td>4,357</td>
<td>7,095</td>
<td>1,567</td>
<td>3,954</td>
<td>1,739</td>
<td>3,618</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,716</td>
<td>5,546</td>
<td>6,307</td>
<td>24,882</td>
<td>1,686</td>
<td>1,947</td>
<td>4,382</td>
<td>7,103</td>
<td>1,629</td>
<td>3,963</td>
<td>1,786</td>
<td>3,696</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,693</td>
<td>5,501</td>
<td>6,293</td>
<td>24,816</td>
<td>1,694</td>
<td>1,972</td>
<td>4,353</td>
<td>7,136</td>
<td>1,613</td>
<td>3,973</td>
<td>1,787</td>
<td>3,690</td>
</tr>
</tbody>
</table>

Table C.4: Mean execution times of the DaCapo benchmarks at the 4th iteration (ms)
Table C.5: Mean execution times of the DaCapo benchmarks at the 5% iteration (ms)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Heap Size</th>
<th>S-RVM - Size</th>
<th>JikesRVM - Size</th>
<th>S-RVM - Pressure</th>
<th>JikesRVM - Pressure</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>300 KB</td>
<td>624 KB</td>
<td>987 KB</td>
<td>1234 KB</td>
<td>1567 KB</td>
</tr>
<tr>
<td>X2</td>
<td>300 KB</td>
<td>624 KB</td>
<td>987 KB</td>
<td>1234 KB</td>
<td>1567 KB</td>
</tr>
<tr>
<td>X3</td>
<td>300 KB</td>
<td>624 KB</td>
<td>987 KB</td>
<td>1234 KB</td>
<td>1567 KB</td>
</tr>
<tr>
<td>X4</td>
<td>300 KB</td>
<td>624 KB</td>
<td>987 KB</td>
<td>1234 KB</td>
<td>1567 KB</td>
</tr>
<tr>
<td>X5</td>
<td>300 KB</td>
<td>624 KB</td>
<td>987 KB</td>
<td>1234 KB</td>
<td>1567 KB</td>
</tr>
<tr>
<td>X6</td>
<td>300 KB</td>
<td>624 KB</td>
<td>987 KB</td>
<td>1234 KB</td>
<td>1567 KB</td>
</tr>
<tr>
<td>X7</td>
<td>300 KB</td>
<td>624 KB</td>
<td>987 KB</td>
<td>1234 KB</td>
<td>1567 KB</td>
</tr>
<tr>
<td>X8</td>
<td>300 KB</td>
<td>624 KB</td>
<td>987 KB</td>
<td>1234 KB</td>
<td>1567 KB</td>
</tr>
<tr>
<td>X9</td>
<td>300 KB</td>
<td>624 KB</td>
<td>987 KB</td>
<td>1234 KB</td>
<td>1567 KB</td>
</tr>
<tr>
<td>Heap Size</td>
<td>JikesRVM - Size</td>
<td>S-RVM - Size</td>
<td>JikesRVM - Pressure</td>
<td>S-RVM - Pressure</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td>--------------</td>
<td>---------------------</td>
<td>------------------</td>
<td></td>
</tr>
<tr>
<td>1X</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>2X</td>
<td>2.000</td>
<td>0.000</td>
<td>2.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>3X</td>
<td>3.000</td>
<td>0.000</td>
<td>3.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>4X</td>
<td>4.000</td>
<td>0.000</td>
<td>4.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>5X</td>
<td>5.000</td>
<td>0.000</td>
<td>5.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>6X</td>
<td>6.000</td>
<td>0.000</td>
<td>6.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>7X</td>
<td>7.000</td>
<td>0.000</td>
<td>7.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>8X</td>
<td>8.000</td>
<td>0.000</td>
<td>8.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>9X</td>
<td>9.000</td>
<td>0.000</td>
<td>9.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>10X</td>
<td>10.000</td>
<td>0.000</td>
<td>10.000</td>
<td>0.000</td>
<td></td>
</tr>
</tbody>
</table>
Table C.7: Mean execution times of the DaCapo benchmarks at the 7th iteration (ms)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>S-RVM - Pressure</th>
<th>JikesRVM</th>
<th>S-RVM - Pressure</th>
<th>JikesRVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>mean</td>
<td></td>
<td>mean</td>
<td></td>
</tr>
<tr>
<td>Heap Size</td>
<td>contribution</td>
<td></td>
<td>contribution</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Size</td>
<td>Configuration</td>
<td>antlr</td>
<td>bloat</td>
<td>chart</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>5X</td>
<td>JikesRVM</td>
<td>1.584</td>
<td>5.019</td>
<td>6.489</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1.598</td>
<td>5.363</td>
<td>6.242</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1.619</td>
<td>5.368</td>
<td>6.245</td>
</tr>
</tbody>
</table>

Table C.8: Mean execution times of the DaCapo benchmarks at the 8th iteration (ms)
Table C.9: Mean execution times of the DaCapo benchmarks at the 9

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Heap Size</th>
<th>S-RVM - Pressure</th>
<th>S-RVM - Size</th>
<th>JikesRVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10X</td>
<td>5089</td>
<td>5511</td>
<td>5831</td>
<td>5361</td>
</tr>
<tr>
<td>9X</td>
<td>5089</td>
<td>5511</td>
<td>5831</td>
<td>5361</td>
</tr>
<tr>
<td>8X</td>
<td>5089</td>
<td>5511</td>
<td>5831</td>
<td>5361</td>
</tr>
<tr>
<td>7X</td>
<td>5089</td>
<td>5511</td>
<td>5831</td>
<td>5361</td>
</tr>
<tr>
<td>6X</td>
<td>5089</td>
<td>5511</td>
<td>5831</td>
<td>5361</td>
</tr>
<tr>
<td>5X</td>
<td>5089</td>
<td>5511</td>
<td>5831</td>
<td>5361</td>
</tr>
</tbody>
</table>

Configuration with boost default compiler flags enabled, but disabled for regression plan. Mean runtime (ms)
<table>
<thead>
<tr>
<th>Heap Size</th>
<th>Configuration</th>
<th>antlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>fop</th>
<th>hsqldb</th>
<th>jython</th>
<th>luaindex</th>
<th>lusarch</th>
<th>pmd</th>
<th>xalan</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>2X</td>
<td>JikesRVM</td>
<td>1,972</td>
<td>5,523</td>
<td>6,697</td>
<td>24,915</td>
<td>1,602</td>
<td>1,885</td>
<td>4,365</td>
<td>7,562</td>
<td>1,672</td>
<td>4,737</td>
<td>2,544</td>
<td>3,950</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,972</td>
<td>5,523</td>
<td>6,697</td>
<td>24,915</td>
<td>1,602</td>
<td>1,885</td>
<td>4,365</td>
<td>7,562</td>
<td>1,672</td>
<td>4,737</td>
<td>2,544</td>
<td>3,950</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,972</td>
<td>5,523</td>
<td>6,697</td>
<td>24,915</td>
<td>1,602</td>
<td>1,885</td>
<td>4,365</td>
<td>7,562</td>
<td>1,672</td>
<td>4,737</td>
<td>2,544</td>
<td>3,950</td>
</tr>
<tr>
<td>3X</td>
<td>JikesRVM</td>
<td>1,972</td>
<td>5,523</td>
<td>6,697</td>
<td>24,915</td>
<td>1,602</td>
<td>1,885</td>
<td>4,365</td>
<td>7,562</td>
<td>1,672</td>
<td>4,737</td>
<td>2,544</td>
<td>3,950</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,972</td>
<td>5,523</td>
<td>6,697</td>
<td>24,915</td>
<td>1,602</td>
<td>1,885</td>
<td>4,365</td>
<td>7,562</td>
<td>1,672</td>
<td>4,737</td>
<td>2,544</td>
<td>3,950</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,972</td>
<td>5,523</td>
<td>6,697</td>
<td>24,915</td>
<td>1,602</td>
<td>1,885</td>
<td>4,365</td>
<td>7,562</td>
<td>1,672</td>
<td>4,737</td>
<td>2,544</td>
<td>3,950</td>
</tr>
<tr>
<td>4X</td>
<td>JikesRVM</td>
<td>1,612</td>
<td>4,950</td>
<td>6,468</td>
<td>23,801</td>
<td>1,569</td>
<td>1,916</td>
<td>4,033</td>
<td>6,996</td>
<td>1,466</td>
<td>3,862</td>
<td>1,776</td>
<td>3,526</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,612</td>
<td>4,950</td>
<td>6,468</td>
<td>23,801</td>
<td>1,569</td>
<td>1,916</td>
<td>4,033</td>
<td>6,996</td>
<td>1,466</td>
<td>3,862</td>
<td>1,776</td>
<td>3,526</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,612</td>
<td>4,950</td>
<td>6,468</td>
<td>23,801</td>
<td>1,569</td>
<td>1,916</td>
<td>4,033</td>
<td>6,996</td>
<td>1,466</td>
<td>3,862</td>
<td>1,776</td>
<td>3,526</td>
</tr>
<tr>
<td>5X</td>
<td>JikesRVM</td>
<td>1,584</td>
<td>4,989</td>
<td>6,473</td>
<td>23,899</td>
<td>1,555</td>
<td>1,853</td>
<td>4,033</td>
<td>7,046</td>
<td>1,473</td>
<td>3,831</td>
<td>1,713</td>
<td>3,501</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,584</td>
<td>4,989</td>
<td>6,473</td>
<td>23,899</td>
<td>1,555</td>
<td>1,853</td>
<td>4,033</td>
<td>7,046</td>
<td>1,473</td>
<td>3,831</td>
<td>1,713</td>
<td>3,501</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,584</td>
<td>4,989</td>
<td>6,473</td>
<td>23,899</td>
<td>1,555</td>
<td>1,853</td>
<td>4,033</td>
<td>7,046</td>
<td>1,473</td>
<td>3,831</td>
<td>1,713</td>
<td>3,501</td>
</tr>
<tr>
<td>6X</td>
<td>JikesRVM</td>
<td>1,554</td>
<td>4,977</td>
<td>6,447</td>
<td>23,781</td>
<td>1,558</td>
<td>1,893</td>
<td>4,046</td>
<td>7,020</td>
<td>1,478</td>
<td>3,860</td>
<td>1,719</td>
<td>3,501</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,554</td>
<td>4,977</td>
<td>6,447</td>
<td>23,781</td>
<td>1,558</td>
<td>1,893</td>
<td>4,046</td>
<td>7,020</td>
<td>1,478</td>
<td>3,860</td>
<td>1,719</td>
<td>3,501</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,554</td>
<td>4,977</td>
<td>6,447</td>
<td>23,781</td>
<td>1,558</td>
<td>1,893</td>
<td>4,046</td>
<td>7,020</td>
<td>1,478</td>
<td>3,860</td>
<td>1,719</td>
<td>3,501</td>
</tr>
<tr>
<td>7X</td>
<td>JikesRVM</td>
<td>1,583</td>
<td>5,027</td>
<td>6,413</td>
<td>23,833</td>
<td>1,551</td>
<td>1,854</td>
<td>4,010</td>
<td>7,043</td>
<td>1,488</td>
<td>3,864</td>
<td>1,687</td>
<td>3,498</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,583</td>
<td>5,027</td>
<td>6,413</td>
<td>23,833</td>
<td>1,551</td>
<td>1,854</td>
<td>4,010</td>
<td>7,043</td>
<td>1,488</td>
<td>3,864</td>
<td>1,687</td>
<td>3,498</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,583</td>
<td>5,027</td>
<td>6,413</td>
<td>23,833</td>
<td>1,551</td>
<td>1,854</td>
<td>4,010</td>
<td>7,043</td>
<td>1,488</td>
<td>3,864</td>
<td>1,687</td>
<td>3,498</td>
</tr>
<tr>
<td>8X</td>
<td>JikesRVM</td>
<td>1,578</td>
<td>5,032</td>
<td>6,420</td>
<td>23,930</td>
<td>1,547</td>
<td>1,875</td>
<td>4,035</td>
<td>7,028</td>
<td>1,468</td>
<td>3,877</td>
<td>1,639</td>
<td>3,488</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,578</td>
<td>5,032</td>
<td>6,420</td>
<td>23,930</td>
<td>1,547</td>
<td>1,875</td>
<td>4,035</td>
<td>7,028</td>
<td>1,468</td>
<td>3,877</td>
<td>1,639</td>
<td>3,488</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,578</td>
<td>5,032</td>
<td>6,420</td>
<td>23,930</td>
<td>1,547</td>
<td>1,875</td>
<td>4,035</td>
<td>7,028</td>
<td>1,468</td>
<td>3,877</td>
<td>1,639</td>
<td>3,488</td>
</tr>
<tr>
<td>9X</td>
<td>JikesRVM</td>
<td>1,574</td>
<td>4,957</td>
<td>6,407</td>
<td>23,759</td>
<td>1,564</td>
<td>1,902</td>
<td>4,037</td>
<td>6,992</td>
<td>1,467</td>
<td>3,865</td>
<td>1,639</td>
<td>3,487</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,574</td>
<td>4,957</td>
<td>6,407</td>
<td>23,759</td>
<td>1,564</td>
<td>1,902</td>
<td>4,037</td>
<td>6,992</td>
<td>1,467</td>
<td>3,865</td>
<td>1,639</td>
<td>3,487</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,574</td>
<td>4,957</td>
<td>6,407</td>
<td>23,759</td>
<td>1,564</td>
<td>1,902</td>
<td>4,037</td>
<td>6,992</td>
<td>1,467</td>
<td>3,865</td>
<td>1,639</td>
<td>3,487</td>
</tr>
<tr>
<td>10X</td>
<td>JikesRVM</td>
<td>1,603</td>
<td>5,313</td>
<td>6,234</td>
<td>23,951</td>
<td>1,571</td>
<td>1,893</td>
<td>4,040</td>
<td>7,000</td>
<td>1,486</td>
<td>3,793</td>
<td>1,649</td>
<td>3,509</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,603</td>
<td>5,313</td>
<td>6,234</td>
<td>23,951</td>
<td>1,571</td>
<td>1,893</td>
<td>4,040</td>
<td>7,000</td>
<td>1,486</td>
<td>3,793</td>
<td>1,649</td>
<td>3,509</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,591</td>
<td>5,340</td>
<td>6,232</td>
<td>23,933</td>
<td>1,580</td>
<td>1,869</td>
<td>4,034</td>
<td>6,980</td>
<td>1,466</td>
<td>3,835</td>
<td>1,648</td>
<td>3,504</td>
</tr>
</tbody>
</table>

Table C.10: Mean execution times of the DaCapo benchmarks at the 10th iteration (ms)
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Heap Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr bloat</td>
<td>1.2GB</td>
</tr>
<tr>
<td>chart</td>
<td>1.6GB</td>
</tr>
<tr>
<td>eclipse</td>
<td>2.1GB</td>
</tr>
<tr>
<td>fop</td>
<td>2.5GB</td>
</tr>
<tr>
<td>hsqldb</td>
<td>3.0GB</td>
</tr>
<tr>
<td>jython</td>
<td>3.4GB</td>
</tr>
<tr>
<td>lusearch</td>
<td>3.8GB</td>
</tr>
<tr>
<td>pmd</td>
<td>4.2GB</td>
</tr>
<tr>
<td>xalan</td>
<td>4.6GB</td>
</tr>
<tr>
<td>JikesRVM</td>
<td>5.0GB</td>
</tr>
<tr>
<td>antlr bloat</td>
<td>5.4GB</td>
</tr>
<tr>
<td>chart</td>
<td>5.8GB</td>
</tr>
<tr>
<td>eclipse</td>
<td>6.2GB</td>
</tr>
<tr>
<td>fop</td>
<td>6.6GB</td>
</tr>
<tr>
<td>hsqldb</td>
<td>7.0GB</td>
</tr>
<tr>
<td>jython</td>
<td>7.4GB</td>
</tr>
<tr>
<td>lusearch</td>
<td>7.8GB</td>
</tr>
<tr>
<td>pmd</td>
<td>8.2GB</td>
</tr>
<tr>
<td>xalan</td>
<td>8.6GB</td>
</tr>
<tr>
<td>JikesRVM</td>
<td>9.0GB</td>
</tr>
</tbody>
</table>

Table C.11: Mean execution times of the DaCapo benchmarks at the 11th iteration (ms)
<table>
<thead>
<tr>
<th>Heap Size</th>
<th>Configuration</th>
<th>smtr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>fop</th>
<th>healdb</th>
<th>jython</th>
<th>lusearch</th>
<th>log4j</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>2X</td>
<td>JikesRVM</td>
<td>1,899</td>
<td>5,133</td>
<td>6,785</td>
<td>24,448</td>
<td>1,590</td>
<td>1,849</td>
<td>4,099</td>
<td>7,349</td>
<td>1,518</td>
<td>4,337</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,975</td>
<td>5,453</td>
<td>6,668</td>
<td>24,823</td>
<td>1,638</td>
<td>1,832</td>
<td>4,402</td>
<td>7,514</td>
<td>1,898</td>
<td>4,651</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,827</td>
<td>5,411</td>
<td>6,583</td>
<td>24,646</td>
<td>1,599</td>
<td>1,788</td>
<td>4,193</td>
<td>7,330</td>
<td>1,640</td>
<td>4,490</td>
</tr>
<tr>
<td>3X</td>
<td>JikesRVM</td>
<td>1,643</td>
<td>4,993</td>
<td>6,577</td>
<td>23,831</td>
<td>1,599</td>
<td>1,824</td>
<td>3,959</td>
<td>7,142</td>
<td>1,466</td>
<td>3,959</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,703</td>
<td>5,271</td>
<td>6,378</td>
<td>24,110</td>
<td>1,555</td>
<td>1,799</td>
<td>3,982</td>
<td>7,132</td>
<td>1,494</td>
<td>3,959</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,647</td>
<td>5,285</td>
<td>6,385</td>
<td>24,021</td>
<td>1,577</td>
<td>1,814</td>
<td>3,965</td>
<td>7,024</td>
<td>1,479</td>
<td>3,931</td>
</tr>
<tr>
<td>4X</td>
<td>JikesRVM</td>
<td>1,581</td>
<td>4,937</td>
<td>6,472</td>
<td>23,646</td>
<td>1,554</td>
<td>1,844</td>
<td>3,969</td>
<td>6,979</td>
<td>1,469</td>
<td>3,850</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,589</td>
<td>5,290</td>
<td>6,288</td>
<td>23,843</td>
<td>1,571</td>
<td>1,804</td>
<td>3,982</td>
<td>6,961</td>
<td>1,478</td>
<td>3,782</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,585</td>
<td>5,283</td>
<td>6,296</td>
<td>23,977</td>
<td>1,555</td>
<td>1,833</td>
<td>3,975</td>
<td>6,940</td>
<td>1,530</td>
<td>3,804</td>
</tr>
<tr>
<td>5X</td>
<td>JikesRVM</td>
<td>1,559</td>
<td>4,988</td>
<td>6,481</td>
<td>23,796</td>
<td>1,558</td>
<td>1,842</td>
<td>3,939</td>
<td>7,018</td>
<td>1,460</td>
<td>3,840</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,604</td>
<td>5,279</td>
<td>6,230</td>
<td>23,943</td>
<td>1,554</td>
<td>1,770</td>
<td>3,985</td>
<td>6,933</td>
<td>1,478</td>
<td>3,815</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,577</td>
<td>5,406</td>
<td>6,261</td>
<td>23,863</td>
<td>1,549</td>
<td>1,766</td>
<td>3,972</td>
<td>6,920</td>
<td>1,495</td>
<td>3,841</td>
</tr>
<tr>
<td>6X</td>
<td>JikesRVM</td>
<td>1,569</td>
<td>5,006</td>
<td>6,410</td>
<td>23,737</td>
<td>1,558</td>
<td>1,885</td>
<td>4,008</td>
<td>7,016</td>
<td>1,478</td>
<td>3,845</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,610</td>
<td>5,283</td>
<td>6,245</td>
<td>23,873</td>
<td>1,580</td>
<td>1,831</td>
<td>3,976</td>
<td>6,954</td>
<td>1,488</td>
<td>3,815</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,572</td>
<td>5,344</td>
<td>6,210</td>
<td>23,889</td>
<td>1,557</td>
<td>1,833</td>
<td>3,971</td>
<td>6,944</td>
<td>1,509</td>
<td>3,822</td>
</tr>
<tr>
<td>7X</td>
<td>JikesRVM</td>
<td>1,573</td>
<td>5,022</td>
<td>6,418</td>
<td>23,786</td>
<td>1,543</td>
<td>1,829</td>
<td>3,965</td>
<td>7,025</td>
<td>1,464</td>
<td>3,834</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,596</td>
<td>5,320</td>
<td>6,266</td>
<td>23,880</td>
<td>1,562</td>
<td>1,792</td>
<td>3,991</td>
<td>6,931</td>
<td>1,484</td>
<td>3,781</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,569</td>
<td>5,307</td>
<td>6,231</td>
<td>23,917</td>
<td>1,551</td>
<td>1,846</td>
<td>3,927</td>
<td>6,934</td>
<td>1,492</td>
<td>3,833</td>
</tr>
<tr>
<td>8X</td>
<td>JikesRVM</td>
<td>1,570</td>
<td>4,969</td>
<td>6,425</td>
<td>23,903</td>
<td>1,563</td>
<td>1,824</td>
<td>3,949</td>
<td>7,012</td>
<td>1,453</td>
<td>3,840</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,578</td>
<td>5,339</td>
<td>6,233</td>
<td>23,963</td>
<td>1,563</td>
<td>1,816</td>
<td>3,995</td>
<td>6,954</td>
<td>1,500</td>
<td>3,841</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,591</td>
<td>5,318</td>
<td>6,213</td>
<td>23,937</td>
<td>1,571</td>
<td>1,814</td>
<td>3,980</td>
<td>6,937</td>
<td>1,496</td>
<td>3,810</td>
</tr>
<tr>
<td>9X</td>
<td>JikesRVM</td>
<td>1,537</td>
<td>4,932</td>
<td>6,406</td>
<td>23,675</td>
<td>1,563</td>
<td>1,842</td>
<td>3,999</td>
<td>6,971</td>
<td>1,456</td>
<td>3,830</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,580</td>
<td>5,326</td>
<td>6,206</td>
<td>23,738</td>
<td>1,576</td>
<td>1,845</td>
<td>3,986</td>
<td>6,877</td>
<td>1,490</td>
<td>3,827</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,573</td>
<td>5,339</td>
<td>6,227</td>
<td>23,810</td>
<td>1,575</td>
<td>1,792</td>
<td>3,982</td>
<td>6,935</td>
<td>1,499</td>
<td>3,783</td>
</tr>
<tr>
<td>10X</td>
<td>JikesRVM</td>
<td>1,580</td>
<td>5,014</td>
<td>6,412</td>
<td>23,771</td>
<td>1,546</td>
<td>1,831</td>
<td>3,978</td>
<td>7,009</td>
<td>1,453</td>
<td>3,872</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,588</td>
<td>5,270</td>
<td>6,216</td>
<td>23,853</td>
<td>1,567</td>
<td>1,801</td>
<td>3,982</td>
<td>6,980</td>
<td>1,499</td>
<td>3,800</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,579</td>
<td>5,313</td>
<td>6,231</td>
<td>23,832</td>
<td>1,569</td>
<td>1,783</td>
<td>3,967</td>
<td>6,981</td>
<td>1,484</td>
<td>3,828</td>
</tr>
</tbody>
</table>

Table C.12: Mean execution times of the DaCapo benchmarks at the 12th iteration (ms)
Table C.13: Mean execution times of the DaCapo benchmarks at the 19 iteration (ms)

<table>
<thead>
<tr>
<th>Heap Size</th>
<th>S-RVM - Pressure</th>
<th>S-RVM - Size</th>
<th>JikesRVM</th>
<th>213</th>
<th>268</th>
<th>568</th>
<th>87X</th>
<th>10X</th>
<th>9X</th>
<th>8X</th>
<th>7X</th>
<th>6X</th>
<th>5X</th>
<th>4X</th>
<th>3X</th>
<th>2X</th>
<th>1X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>495</td>
<td>572</td>
<td>389</td>
<td>192</td>
<td>647</td>
<td>562</td>
<td>539</td>
<td>520</td>
<td>450</td>
<td>399</td>
<td>358</td>
<td>331</td>
<td>313</td>
<td>299</td>
<td>287</td>
<td>275</td>
<td></td>
</tr>
<tr>
<td>2M</td>
<td>506</td>
<td>589</td>
<td>392</td>
<td>194</td>
<td>654</td>
<td>568</td>
<td>541</td>
<td>520</td>
<td>455</td>
<td>406</td>
<td>371</td>
<td>342</td>
<td>324</td>
<td>309</td>
<td>295</td>
<td>283</td>
<td>270</td>
</tr>
<tr>
<td>4M</td>
<td>517</td>
<td>592</td>
<td>395</td>
<td>196</td>
<td>661</td>
<td>572</td>
<td>544</td>
<td>523</td>
<td>460</td>
<td>411</td>
<td>376</td>
<td>347</td>
<td>328</td>
<td>313</td>
<td>299</td>
<td>285</td>
<td>272</td>
</tr>
<tr>
<td>8M</td>
<td>528</td>
<td>595</td>
<td>398</td>
<td>198</td>
<td>668</td>
<td>575</td>
<td>546</td>
<td>525</td>
<td>465</td>
<td>416</td>
<td>381</td>
<td>352</td>
<td>333</td>
<td>318</td>
<td>304</td>
<td>290</td>
<td>277</td>
</tr>
<tr>
<td>16M</td>
<td>539</td>
<td>602</td>
<td>401</td>
<td>200</td>
<td>675</td>
<td>578</td>
<td>548</td>
<td>527</td>
<td>470</td>
<td>421</td>
<td>386</td>
<td>357</td>
<td>338</td>
<td>323</td>
<td>309</td>
<td>295</td>
<td>281</td>
</tr>
<tr>
<td>32M</td>
<td>550</td>
<td>605</td>
<td>404</td>
<td>202</td>
<td>682</td>
<td>581</td>
<td>550</td>
<td>529</td>
<td>475</td>
<td>426</td>
<td>391</td>
<td>362</td>
<td>343</td>
<td>328</td>
<td>314</td>
<td>299</td>
<td>285</td>
</tr>
<tr>
<td>64M</td>
<td>561</td>
<td>608</td>
<td>407</td>
<td>204</td>
<td>689</td>
<td>584</td>
<td>553</td>
<td>532</td>
<td>480</td>
<td>431</td>
<td>396</td>
<td>367</td>
<td>348</td>
<td>333</td>
<td>319</td>
<td>304</td>
<td>290</td>
</tr>
<tr>
<td>128M</td>
<td>572</td>
<td>611</td>
<td>410</td>
<td>206</td>
<td>696</td>
<td>587</td>
<td>556</td>
<td>535</td>
<td>485</td>
<td>436</td>
<td>401</td>
<td>372</td>
<td>353</td>
<td>338</td>
<td>324</td>
<td>309</td>
<td>295</td>
</tr>
<tr>
<td>256M</td>
<td>583</td>
<td>614</td>
<td>413</td>
<td>208</td>
<td>703</td>
<td>590</td>
<td>559</td>
<td>538</td>
<td>490</td>
<td>441</td>
<td>406</td>
<td>377</td>
<td>358</td>
<td>343</td>
<td>329</td>
<td>314</td>
<td>299</td>
</tr>
<tr>
<td>512M</td>
<td>594</td>
<td>617</td>
<td>416</td>
<td>210</td>
<td>710</td>
<td>593</td>
<td>562</td>
<td>541</td>
<td>495</td>
<td>446</td>
<td>411</td>
<td>382</td>
<td>363</td>
<td>348</td>
<td>334</td>
<td>319</td>
<td>304</td>
</tr>
<tr>
<td>1024M</td>
<td>605</td>
<td>620</td>
<td>419</td>
<td>212</td>
<td>717</td>
<td>596</td>
<td>565</td>
<td>544</td>
<td>500</td>
<td>451</td>
<td>416</td>
<td>387</td>
<td>368</td>
<td>353</td>
<td>338</td>
<td>324</td>
<td>309</td>
</tr>
</tbody>
</table>

Note: The table includes execution times for various heap sizes and benchmarks, with different numbers indicating the execution times in milliseconds.
<table>
<thead>
<tr>
<th>Heap Size</th>
<th>JikesRVM - Size</th>
<th>JikesRVM - Pressure</th>
<th>S-RVM - Size</th>
<th>S-RVM - Pressure</th>
</tr>
</thead>
<tbody>
<tr>
<td>2X</td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td></td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td>3X</td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td></td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td>4X</td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td></td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td>5X</td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td></td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td>6X</td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td></td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td>7X</td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td></td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td>8X</td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
<tr>
<td></td>
<td>1.971 6.721</td>
<td>1.760 0.841</td>
<td>1.313 3.941</td>
<td>1.313 3.941</td>
</tr>
</tbody>
</table>

Table C.14: Mean execution times of the DaCapo benchmarks at the 14th iteration (ms)
### Table C.15: Mean execution times of the DaCapo benchmarks at the 15\textsuperscript{th} iteration (ms)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JikesRVM</th>
<th>3</th>
<th>650</th>
<th>800</th>
<th>1</th>
<th>591</th>
<th>3</th>
<th>655</th>
<th>806</th>
<th>448</th>
<th>3</th>
<th>650</th>
<th>806</th>
<th>448</th>
<th>3</th>
<th>650</th>
<th>806</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>696</td>
<td>3</td>
<td>691</td>
<td>3</td>
<td>696</td>
<td>3</td>
<td>691</td>
<td>3</td>
<td>696</td>
<td>3</td>
<td>691</td>
<td>3</td>
<td>696</td>
<td>3</td>
<td>691</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>347</td>
<td>6</td>
<td>347</td>
<td>6</td>
<td>347</td>
<td>6</td>
<td>347</td>
<td>6</td>
<td>347</td>
<td>6</td>
<td>347</td>
<td>6</td>
<td>347</td>
<td>6</td>
<td>347</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>289</td>
<td>8</td>
<td>289</td>
<td>8</td>
<td>289</td>
<td>8</td>
<td>289</td>
<td>8</td>
<td>289</td>
<td>8</td>
<td>289</td>
<td>8</td>
<td>289</td>
<td>8</td>
<td>289</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>236</td>
<td>10</td>
<td>236</td>
<td>10</td>
<td>236</td>
<td>10</td>
<td>236</td>
<td>10</td>
<td>236</td>
<td>10</td>
<td>236</td>
<td>10</td>
<td>236</td>
<td>10</td>
<td>236</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>183</td>
<td>15</td>
<td>183</td>
<td>15</td>
<td>183</td>
<td>15</td>
<td>183</td>
<td>15</td>
<td>183</td>
<td>15</td>
<td>183</td>
<td>15</td>
<td>183</td>
<td>15</td>
<td>183</td>
<td>15</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>91</td>
<td>30</td>
<td>91</td>
<td>30</td>
<td>91</td>
<td>30</td>
<td>91</td>
<td>30</td>
<td>91</td>
<td>30</td>
<td>91</td>
<td>30</td>
<td>91</td>
<td>30</td>
<td>91</td>
<td>30</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
<td>42</td>
<td>60</td>
<td>42</td>
<td>60</td>
<td>42</td>
<td>60</td>
<td>42</td>
<td>60</td>
<td>42</td>
<td>60</td>
<td>42</td>
<td>60</td>
<td>42</td>
<td>60</td>
<td>42</td>
<td>60</td>
</tr>
<tr>
<td>120</td>
<td>120</td>
<td>23</td>
<td>120</td>
<td>23</td>
<td>120</td>
<td>23</td>
<td>120</td>
<td>23</td>
<td>120</td>
<td>23</td>
<td>120</td>
<td>23</td>
<td>120</td>
<td>23</td>
<td>120</td>
<td>23</td>
<td>120</td>
</tr>
<tr>
<td>240</td>
<td>240</td>
<td>16</td>
<td>240</td>
<td>16</td>
<td>240</td>
<td>16</td>
<td>240</td>
<td>16</td>
<td>240</td>
<td>16</td>
<td>240</td>
<td>16</td>
<td>240</td>
<td>16</td>
<td>240</td>
<td>16</td>
<td>240</td>
</tr>
</tbody>
</table>

**Heap Size Contribution:** The heap size contributes to the execution time due to the overhead involved in managing memory.
<table>
<thead>
<tr>
<th>Heap Size</th>
<th>Configuration</th>
<th>mntlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>fop</th>
<th>hsqldb</th>
<th>jython</th>
<th>landalone</th>
<th>lusarch</th>
<th>pmtd</th>
<th>xalan</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>2X</td>
<td>JikesRVM</td>
<td>1,842</td>
<td>5,064</td>
<td>6,751</td>
<td>24,353</td>
<td>1,610</td>
<td>1,796</td>
<td>4,051</td>
<td>7,326</td>
<td>1,613</td>
<td>4,365</td>
<td>2,218</td>
<td>3,748</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,899</td>
<td>5,659</td>
<td>6,728</td>
<td>24,698</td>
<td>1,617</td>
<td>1,761</td>
<td>4,161</td>
<td>7,497</td>
<td>1,667</td>
<td>4,666</td>
<td>2,557</td>
<td>3,898</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,785</td>
<td>5,503</td>
<td>6,605</td>
<td>24,526</td>
<td>1,541</td>
<td>1,748</td>
<td>4,069</td>
<td>7,291</td>
<td>1,543</td>
<td>4,396</td>
<td>2,377</td>
<td>3,748</td>
</tr>
<tr>
<td>3X</td>
<td>JikesRVM</td>
<td>1,608</td>
<td>4,983</td>
<td>6,577</td>
<td>23,690</td>
<td>1,565</td>
<td>1,800</td>
<td>3,883</td>
<td>7,073</td>
<td>1,514</td>
<td>3,875</td>
<td>1,845</td>
<td>3,525</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,645</td>
<td>5,313</td>
<td>6,380</td>
<td>23,957</td>
<td>1,557</td>
<td>1,738</td>
<td>3,859</td>
<td>7,096</td>
<td>1,514</td>
<td>3,944</td>
<td>1,878</td>
<td>3,545</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,639</td>
<td>5,290</td>
<td>6,368</td>
<td>23,899</td>
<td>1,551</td>
<td>1,758</td>
<td>3,839</td>
<td>7,004</td>
<td>1,496</td>
<td>3,832</td>
<td>1,865</td>
<td>3,521</td>
</tr>
<tr>
<td>4X</td>
<td>JikesRVM</td>
<td>1,523</td>
<td>4,906</td>
<td>6,460</td>
<td>23,490</td>
<td>1,544</td>
<td>1,839</td>
<td>3,871</td>
<td>6,965</td>
<td>1,489</td>
<td>3,849</td>
<td>1,766</td>
<td>3,470</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,580</td>
<td>5,281</td>
<td>6,267</td>
<td>23,733</td>
<td>1,553</td>
<td>1,762</td>
<td>3,860</td>
<td>6,900</td>
<td>1,531</td>
<td>3,761</td>
<td>1,777</td>
<td>3,486</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,570</td>
<td>5,314</td>
<td>6,276</td>
<td>23,732</td>
<td>1,551</td>
<td>1,779</td>
<td>3,903</td>
<td>6,928</td>
<td>1,497</td>
<td>3,765</td>
<td>1,792</td>
<td>3,490</td>
</tr>
<tr>
<td>5X</td>
<td>JikesRVM</td>
<td>1,515</td>
<td>4,958</td>
<td>6,464</td>
<td>23,638</td>
<td>1,543</td>
<td>1,791</td>
<td>3,891</td>
<td>6,991</td>
<td>1,556</td>
<td>3,820</td>
<td>1,713</td>
<td>3,470</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,554</td>
<td>5,281</td>
<td>6,234</td>
<td>23,758</td>
<td>1,538</td>
<td>1,737</td>
<td>3,857</td>
<td>6,914</td>
<td>1,511</td>
<td>3,801</td>
<td>1,722</td>
<td>3,462</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,576</td>
<td>5,347</td>
<td>6,242</td>
<td>23,711</td>
<td>1,538</td>
<td>1,750</td>
<td>3,882</td>
<td>6,887</td>
<td>1,513</td>
<td>3,833</td>
<td>1,708</td>
<td>3,474</td>
</tr>
<tr>
<td>6X</td>
<td>JikesRVM</td>
<td>1,543</td>
<td>4,969</td>
<td>6,441</td>
<td>23,539</td>
<td>1,552</td>
<td>1,777</td>
<td>3,922</td>
<td>6,989</td>
<td>1,508</td>
<td>3,817</td>
<td>1,700</td>
<td>3,464</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,554</td>
<td>5,321</td>
<td>6,235</td>
<td>23,770</td>
<td>1,563</td>
<td>1,760</td>
<td>3,851</td>
<td>7,006</td>
<td>1,534</td>
<td>3,787</td>
<td>1,714</td>
<td>3,480</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,553</td>
<td>5,310</td>
<td>6,204</td>
<td>23,717</td>
<td>1,547</td>
<td>1,750</td>
<td>3,848</td>
<td>6,909</td>
<td>1,537</td>
<td>3,785</td>
<td>1,715</td>
<td>3,467</td>
</tr>
<tr>
<td>7X</td>
<td>JikesRVM</td>
<td>1,540</td>
<td>4,958</td>
<td>6,418</td>
<td>23,664</td>
<td>1,543</td>
<td>1,830</td>
<td>3,878</td>
<td>6,965</td>
<td>1,514</td>
<td>3,831</td>
<td>1,677</td>
<td>3,464</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,558</td>
<td>5,372</td>
<td>6,252</td>
<td>23,743</td>
<td>1,555</td>
<td>1,732</td>
<td>3,871</td>
<td>6,906</td>
<td>1,508</td>
<td>3,768</td>
<td>1,725</td>
<td>3,470</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,573</td>
<td>5,294</td>
<td>6,207</td>
<td>23,770</td>
<td>1,548</td>
<td>1,746</td>
<td>3,816</td>
<td>6,913</td>
<td>1,521</td>
<td>3,812</td>
<td>1,683</td>
<td>3,461</td>
</tr>
<tr>
<td>8X</td>
<td>JikesRVM</td>
<td>1,524</td>
<td>4,970</td>
<td>6,450</td>
<td>23,681</td>
<td>1,535</td>
<td>1,823</td>
<td>3,877</td>
<td>6,999</td>
<td>1,519</td>
<td>3,849</td>
<td>1,647</td>
<td>3,458</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,554</td>
<td>5,311</td>
<td>6,210</td>
<td>23,741</td>
<td>1,542</td>
<td>1,738</td>
<td>3,869</td>
<td>6,930</td>
<td>1,532</td>
<td>3,815</td>
<td>1,641</td>
<td>3,455</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,558</td>
<td>5,270</td>
<td>6,216</td>
<td>23,703</td>
<td>1,559</td>
<td>1,749</td>
<td>3,863</td>
<td>6,925</td>
<td>1,503</td>
<td>3,790</td>
<td>1,616</td>
<td>3,447</td>
</tr>
<tr>
<td>9X</td>
<td>JikesRVM</td>
<td>1,563</td>
<td>4,938</td>
<td>6,426</td>
<td>23,492</td>
<td>1,558</td>
<td>1,820</td>
<td>3,903</td>
<td>7,016</td>
<td>1,478</td>
<td>3,800</td>
<td>1,643</td>
<td>3,454</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,559</td>
<td>5,322</td>
<td>6,191</td>
<td>23,567</td>
<td>1,553</td>
<td>1,752</td>
<td>3,864</td>
<td>6,866</td>
<td>1,542</td>
<td>3,796</td>
<td>1,632</td>
<td>3,454</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,546</td>
<td>5,272</td>
<td>6,191</td>
<td>23,573</td>
<td>1,553</td>
<td>1,735</td>
<td>3,853</td>
<td>6,909</td>
<td>1,529</td>
<td>3,769</td>
<td>1,645</td>
<td>3,444</td>
</tr>
<tr>
<td>10X</td>
<td>JikesRVM</td>
<td>1,527</td>
<td>4,972</td>
<td>6,456</td>
<td>23,526</td>
<td>1,555</td>
<td>1,805</td>
<td>3,909</td>
<td>6,964</td>
<td>1,524</td>
<td>3,823</td>
<td>1,644</td>
<td>3,477</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,558</td>
<td>5,274</td>
<td>6,226</td>
<td>23,635</td>
<td>1,549</td>
<td>1,715</td>
<td>3,845</td>
<td>6,954</td>
<td>1,514</td>
<td>3,788</td>
<td>1,624</td>
<td>3,444</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,568</td>
<td>5,241</td>
<td>6,210</td>
<td>23,570</td>
<td>1,543</td>
<td>1,753</td>
<td>3,857</td>
<td>6,916</td>
<td>1,548</td>
<td>3,797</td>
<td>1,638</td>
<td>3,454</td>
</tr>
</tbody>
</table>

Table C.16: Mean execution times of the DaCapo benchmarks at the 16th iteration (ms)
Table C.17: Mean execution times of the DaCapo benchmarks at the 17th iteration (ms)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>S-RVM - Size</th>
<th>S-RVM - Pressure</th>
<th>JikesRVM - Pressure</th>
<th>JikesRVM - Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap Size</td>
<td>1X</td>
<td>2X</td>
<td>3X</td>
<td>4X</td>
</tr>
<tr>
<td>128M</td>
<td>4142 16:21</td>
<td>4595 16:21</td>
<td>5244 16:21</td>
<td>3545 16:21</td>
</tr>
<tr>
<td>256M</td>
<td>4142 16:21</td>
<td>4595 16:21</td>
<td>5244 16:21</td>
<td>3545 16:21</td>
</tr>
<tr>
<td>512M</td>
<td>4142 16:21</td>
<td>4595 16:21</td>
<td>5244 16:21</td>
<td>3545 16:21</td>
</tr>
<tr>
<td>1GB</td>
<td>4142 16:21</td>
<td>4595 16:21</td>
<td>5244 16:21</td>
<td>3545 16:21</td>
</tr>
</tbody>
</table>
Table C.18: Mean execution times of the DaCapo benchmarks at the 18th iteration (ms)

<table>
<thead>
<tr>
<th>JikesRVM</th>
<th>S-RVM - Size</th>
<th>S-RVM - Pressure</th>
<th>JikesRVM</th>
<th>S-RVM - Size</th>
<th>S-RVM - Pressure</th>
<th>JikesRVM</th>
<th>S-RVM - Size</th>
<th>S-RVM - Pressure</th>
<th>JikesRVM</th>
<th>S-RVM - Size</th>
<th>S-RVM - Pressure</th>
<th>JikesRVM</th>
<th>S-RVM - Size</th>
<th>S-RVM - Pressure</th>
</tr>
</thead>
<tbody>
<tr>
<td>4X</td>
<td>9X</td>
<td>10X</td>
<td>4X</td>
<td>9X</td>
<td>10X</td>
<td>4X</td>
<td>9X</td>
<td>10X</td>
<td>4X</td>
<td>9X</td>
<td>10X</td>
<td>4X</td>
<td>9X</td>
<td>10X</td>
</tr>
</tbody>
</table>
Table C.19: Mean execution times of the DaCapo benchmarks at the 19

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>S-RVM - Size</th>
<th>S-RVM - Pressure</th>
<th>S-RVM - Speed</th>
<th>S-RVM - Pressure</th>
<th>S-RVM - Speed</th>
<th>S-RVM - Pressure</th>
<th>S-RVM - Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap Size</td>
<td>1 MB</td>
<td>2 MB</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>32 MB</td>
<td>64 MB</td>
</tr>
<tr>
<td>10X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Size</td>
<td>Configuration</td>
<td>antlr</td>
<td>bloat</td>
<td>chart</td>
<td>eclipse</td>
<td>FreeBSD</td>
<td>hsqldb</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>---------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>2X</td>
<td>JikesRVM</td>
<td>1,740</td>
<td>5,027</td>
<td>6,742</td>
<td>24.262</td>
<td>1.518</td>
<td>1.743</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,918</td>
<td>5,393</td>
<td>6,687</td>
<td>24.577</td>
<td>1.526</td>
<td>1.679</td>
</tr>
<tr>
<td>3X</td>
<td>JikesRVM</td>
<td>1,556</td>
<td>4,912</td>
<td>6,546</td>
<td>23.618</td>
<td>1.520</td>
<td>1.743</td>
</tr>
<tr>
<td>4X</td>
<td>JikesRVM</td>
<td>1,487</td>
<td>4,853</td>
<td>6,437</td>
<td>23.426</td>
<td>1.513</td>
<td>1.780</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,495</td>
<td>5,211</td>
<td>6,222</td>
<td>23.578</td>
<td>1.509</td>
<td>1.705</td>
</tr>
<tr>
<td>5X</td>
<td>JikesRVM</td>
<td>1,486</td>
<td>4,889</td>
<td>6,417</td>
<td>23.454</td>
<td>1.507</td>
<td>1.775</td>
</tr>
<tr>
<td>6X</td>
<td>JikesRVM</td>
<td>1,496</td>
<td>4,888</td>
<td>6,445</td>
<td>23.442</td>
<td>1.517</td>
<td>1.761</td>
</tr>
<tr>
<td>7X</td>
<td>JikesRVM</td>
<td>1,474</td>
<td>4,899</td>
<td>6,376</td>
<td>23.507</td>
<td>1.503</td>
<td>1.745</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,506</td>
<td>5,225</td>
<td>6,206</td>
<td>23.640</td>
<td>1.513</td>
<td>1.681</td>
</tr>
<tr>
<td>9X</td>
<td>JikesRVM</td>
<td>1,500</td>
<td>4,856</td>
<td>6,370</td>
<td>23.277</td>
<td>1.525</td>
<td>1.769</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Size</td>
<td>1,481</td>
<td>5,206</td>
<td>6,175</td>
<td>23.300</td>
<td>1.536</td>
<td>1.715</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,508</td>
<td>5,244</td>
<td>6,192</td>
<td>23.409</td>
<td>1.505</td>
<td>1.684</td>
</tr>
<tr>
<td>10X</td>
<td>JikesRVM</td>
<td>1,497</td>
<td>4,924</td>
<td>6,424</td>
<td>23.390</td>
<td>1.522</td>
<td>1.750</td>
</tr>
<tr>
<td></td>
<td>S-RVM - Pressure</td>
<td>1,482</td>
<td>5,184</td>
<td>6,168</td>
<td>23.322</td>
<td>1.515</td>
<td>1.671</td>
</tr>
</tbody>
</table>

Table C.20: Mean execution times of the DaCapo benchmarks at the 20th iteration (ms)


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 system services. In Sixth ACM SIGOPS European Workshop, pages 68–71, Dagstuhl Castle, Germany, September 1994.


Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampt-


Michael Golm, Jürgen Kleinöder, and Frank Bellosa. Beyond address spaces - flexibility, performance, protection, and resource management in the type-safe JX


144


Yi Lin. Personal communication, 19 July 2012.


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.


